UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

PARALELIZAÇÃO AUTOMÁTICA DE ALGORITMOS MATRICIAIS

DOUTORAMENTO EM ENGENHARIA ELECTRÓNICA E COMPUTAÇÃO
NA ESPECIALIDADE DE SISTEMAS DE CONTROLO

HELDER ANICETO AMADEU DE SOUSA DANIEL

Faro
2003

HELDER ANICETO AMADEU DE SOUSA DANIEL

FACULDADE DE CIÊNCIAS E TECNOLOGIA

Orientador: Prof. Doutor ANTÓNIO EDUARDO DE BARROS RUANO

PARALELIZAÇÃO AUTOMÁTICA DE ALGORITMOS MATRICIAIS

À Lúria,
à Lis
e à Inês

Agradecimentos

Primeiramente quero manifestar os meus agradecimentos ao meu orientador, o Prof. Doutor António Eduardo de Barros Ruano, não só por todo o seu apoio na elaboração desta tese, mas também pela sua amizade e orientação na minha vida académica.

À Prof. Doutora Maria da Graça Cristo dos Santos Lopes Ruano, os meus agradecimentos por possibilitar-me e apoiar o meu primeiro contacto com arquitecturas de processamento paralelo.

Quero também prestar o meu reconhecimento à Fundação para a Ciência e Tecnologia, a qual através do programa Praxis XXI apoiou financeiramente a elaboração desta tese.

Não quero deixar passar esta oportunidade sem agradecer à minha esposa todo o apoio e compreensão ao longo dos anos de elaboração desta tese. Finalmente gostaria de agradecer ao meu pai por me ter despertado e estimulado o gosto pelo conhecimento em geral e especificamente pelas ciências tecnológicas.

Resumo

A introdução de arquitecturas de processamento paralelo permitiu que o tempo de processamento de um algoritmo possa ser reduzido dividindo o esforço computacional por mais do que um processador. Todavia para se tirar partido destas arquitecturas, devido à falta de ferramentas apropriadas, o projectista despende uma considerável quantidade de tempo na paralelização do algoritmo sequencial. Outro problema normalmente encontrado, no modelo de programação paralelo, relaciona-se com o facto de a paralelização destes algoritmos ser altamente dependente da arquitectura objecto. Assim, a portabilidade e adaptabilidade destas aplicações são tarefas consumidoras de tempo de desenvolvimento. Pelas razões apontadas, o tempo de implementação de um algoritmo paralelo é muito superior ao tempo de implementação sequencial do mesmo algoritmo.

Tais condições constituíram a motivação para o trabalho desenvolvido nesta tese, o qual consiste num sistema de paralelização automático de algoritmos matriciais. Este sistema é visto como um conjunto de níveis de abstracção que gradualmente se afastam do modelo de processamento paralelo e se aproximam do modelo sequencial. No nível mais elevado basta uma descrição do algoritmo, numa linguagem sequencial, e um diagrama de blocos da rede de processadores, para que o sistema, automaticamente, gere o código paralelo para a rede objecto.

Esta implementação, baseada em sucessivos níveis de abstracção, permite um elevado grau de portabilidade e flexibilidade do sistema, de modo que a introdução de novos processadores, com diferentes especificações de computação e comunicação, ou de operações matriciais não incluídas na biblioteca matricial que acompanha o sistema, seja uma tarefa facilitada.

Finalmente é estudada a paralelização automática de dois algoritmos, de modo a demonstrar o modelo de programação proposto bem como o desempenho dos algoritmos paralelos automaticamente gerados.

Palavras chave: Processamento paralelo, Geração de código, Sistemas de tempo real, Controlo adaptativo, Controlo predictivo, Redes neuronais

Abstract

The introduction of parallel processing architectures allows the reduction of computation time by mapping the algorithm over a network of cooperating processors. However, to take advantage of the processing power of these architectures, the programmer, due to the lack of appropriate tools, must spend a considerable amount of time in the parallelization of the algorithm. Another problem with the conventional parallel programming model is that the parallelization of these algorithms is strongly dependent on the hardware used, so portability and upgradability are also time consuming steps in the development process. This way, the development time of efficient parallel applications, with the conventional tools available, is many times superior to the development time of a sequential application.

As a response to the above constrains suggested, the programming environment which is proposed in this thesis, can automatic generate a parallel version of a matrix algorithm, from a sequential description. This environment can be decomposed in successive levels of abstraction, from the hardware, and the parallel processing model, rising to the top level, where a virtual sequential processing model is presented. At this level, from a sequential description of the algorithm, and a block diagram of the processor network, the parallel algorithm is automatically generated.

This abstraction level implementation was designed to allow an higher level of portability and flexibility, so that the extension of the proposed environment to handle new processors, with different specifications in terms of computation and communication, can be easily accomplished. The environment also allows an easy expansion of the basic linear algebra library included, to meet new algorithms demand.

To show the proposed environment, and the performance of the automatically generated parallel algorithms, two cases of implementation are studied.

Keywords: Parallel Processing, Code generation, Real-time systems, Adaptive control, Predictive control, Neural networks

Trabalhos realizados no âmbito da tese

No âmbito desta tese, e como resultado de investigação contínua e progressiva, foram apresentadas 7 comunicações e publicado um artigo de revista. Foi apresentada na conferência *2nd International Meeting on Vector and Parallel Processing (VECPAR 96)*, uma primeira implementação em paralelo, de um dos casos de estudo apresentados no capítulo 8: *Parallel Implementation of an Adaptive Generalized Predictive Control Algorithm*. Esta implementação, sobre uma rede de IMS T805 Transputers, provou a escalabilidade do algoritmo AGPC. No entanto, os tempos de cálculo das funções matriciais desenvolvidas, eram ainda bastante elevados. Além disso, a estratégia de gestão de memória, limitava o horizonte de predição a 50 instantes de amostragem futuros. Se bem que na maioria dos casos, um horizonte tão longínquo não implica significativamente uma maior precisão do sinal de controlo, tal limitação revela uma má gestão de memória. Ambas as limitações foram removidas e os resultados foram apresentados na conferência *European Control Conference (ECC 97)*, sob o título *Parallel Implementation of an Adaptive Generalized Predictive Control Algorithm*.

A possibilidade de generalizar a implementação do algoritmo a processadores mais recentes, e mais eficientes que os T805, nomeadamente os processadores digitais de sinal TMS320C40, foi apresentada na comunicação: *Implementation of an Adaptive Generalized Predictive Control Algorithm over an Heterogeneous Parallel Architecture*, apresentada na conferência *4th IFAC Workshop on Algorithms and Architectures for Real - Time Control (AARTC 97)*. Nesta comunicação é também avaliado o desempenho de redes de processamento heterogéneas constituídas por T805 e C40. Este conceito é estendido na comunicação *Adaptive Generalized Predictive Control Algorithm Implemented over an Heterogeneous Parallel Architecture*, apresentado na conferência: *14th IFAC International Workshop on Distributed Computer Control Systems*. Nesta fase foi ainda elaborada uma comunicação, apresentada na conferência *8th Annual International Conference on Signal Processing Applications and Technology (ICSPAT 97)*, sob o título: *Adaptive Generalized Predictive Control Algorithm Implemented over a DSP Network*, que avalia o desempenho do algoritmo AGPC paralelo, mapeado sobre uma rede homogénea constituída por C40s.

Da avaliação do desempenho das várias arquitecturas e topologias ensaiadas previamente, resultou uma comunicação publicada na edição especial *Microprocessors and Microsystems*, da revista *Journal of Automatic Control*, intitulada *Performance comparison of parallel architectures for real-time control*.

Finalmente, a investigação centrou-se na paralelização automática de algoritmos matriciais, de onde foi extraída a comunicação *Automatic parallelization of matricial algorithms*, apresentada na conferência *14th IFAC World Congress (IFAC 99)*, que constitui uma apresentação desta tese. Foi também apresentada uma comunicação intitulada *Automatic parallelization of an Adaptive Generalized Predictive Control Algorithm using MAPS 1.0 Environment*, na *6th IFAC/IFIP Workshop on Algorithms and Architectures for Real - Time Control (AARTC 2000)*, que ilustra um caso prático da utilização do ambiente de desenvolvimento de aplicações introduzido nesta tese, e apresentado no ponto 8.1

Índice

Nomenclatura

ADSP21060	Processador digital de sinal Analog Devices ADSP-21060 (Sharc)
ADT	Abstract Data Type
AGPC	Controlo Predictivo Generalizado Adaptativo (Adaptive Generalized Predictive Control)
AIDA	Ambiente Integrado de Desenvolvimento de Aplicações
ALU	Unidade lógico aritmética (Arithmetic Logic Unit)
ASCII	American Standard Code for Information Interchange
BNF	Backus Naur Form
b, *bit*	Digito binário
B, *byte*	Octeto
C40	Processador digital de sinal Texas Instruments TMS320C40
C4x	Família de processadores digitais de sinal desenvolvida pela Texas Instruments.
CPU	Unidade de processamento central (Central Processing Unit)
DMA	Acesso directo à memória (Direct Memory Access)

DRAM	RAM dinâmica (Dinamic Random Access Memory)
DSP	Processador Digital de Sinal (Digital Signal Processor)
EP	Processador (Elemento de Processamento)
FPU	Unidade de vírgula flutuante (Floating Point Unit)
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers, Inc.
MFlops	Milhões de operações de vírgula flutuante por segundo (Millions of Floating point operations per second)
MIPS	Milhões de instruções por segundo (Millions of Instructions Per Second)
LM	Método de Levenberg-Marquardt
MLP	Perceptrão multi-camada (Multilayer perceptron)
NP	Número de Processadores ou nós numa rede
npc	Número de Portos de Comunicação de um processador ou nó
P2	Processador Intel Pentium II

PC	Computador pessoal (Personal Computer)
PCI	Peripheral Component Interconnect
PCM	Protocolo de Controlo de Mensagem
Px	Processador *x*
RAM	Memória de acesso aleatório (Random Access Memory)
RLS	Método Recursivo dos Mínimos Quadrados (Recursive Least Squares)
RN	Rede Neuronal
ROM	Memória apenas de leitura (Read Only Memory)
s, seg	Segundos
Sharc	Família de processador digitais de sinal desenvolvida pela Analog Devices (Super Harvard Architecture Computer)
SPAM	Sistema de Paralelização Automática de Algoritmos Matriciais
SRAM	RAM estática (Static Random Access Memory)
T805	Processador IMS/Thompson T805 (Transputer)
Transputer	Família de processadores desenvolvida pela INMOS (Transistor Computer)

Notações utilizadas

escalar	Caracteres normais.
vector	Caracteres minúsculos a cheio.
Matriz	Caracter inicial maiúsculo a cheio.
$\mathbf{A}_{i,j}$ ou $\boldsymbol{A}_{\mathrm{i\,j}}$	Elemento da matriz **A**, correspondente à linha *i*, coluna *j*.
$\mathbf{A}_{i..k,j}$	Vector coluna que corresponde aos elementos da coluna *j* da matriz **A**, linhas *i* a *k*.
$\mathbf{A}_{\{c\},j}$	Vector coluna que corresponde aos elementos da coluna *j* da matriz **A**, cujas linhas são dadas pelo conjunto $\{c\}$.
$\mathbf{A}_{.,j}$ ou $\mathbf{A}_{*,j}$	Vector coluna que corresponde à coluna *j* da matriz **A.**
$\mathbf{A}\ (m \times n)$	Matriz **A** com *m* linhas e *n* colunas.
$x\ [k]$	Valor do escalar *x* na iteração *k*

1 Introdução

Nas mais variadas disciplinas científico-tecnológicas existe a necessidade ou conveniência de expressar determinados algoritmos no formato matricial. De facto, os modelos de sistemas físicos clássicos são normalmente representados dispondo as equações num formato matricial, o que traz algumas vantagens quanto ao formalismo das operações sobre esses modelos, simplificando assim a manipulação destes. Mais particularmente, na área de engenharia de controlo clássico, tais algoritmos podem ser utilizados para descrever técnicas de identificação de sistemas (Wellstead e Zarrop, 1991) ou estratégias de controlo adaptativo (Aström e Wittenmark, 1989).

Mas mesmo modelos não deterministas, como é o caso dos modelos de sistemas quânticos, podem ser representados segundo uma abordagem matricial, desenvolvida por Werner Heisenberg, Max Born e Pascual Jordan denominada mecânica de matrizes (Weinberg, 1992) e que John von Neumann provou ser matematicamente equivalente à mecânica ondulatória de Erwin Schrödinger (Weisstein, 1999), que é hoje o formalismo matemático mais utilizado para descrição de sistemas quânticos.

A aplicação deste tipo de algoritmos não se restringe apenas à descrição do mundo físico, podendo representar conceitos mais abstractos, como é o caso de algumas estratégias de optimização, sendo um caso comum em programação linear, o algoritmo simplex revisto (Heesterman, 1983).

Conceitos ainda mais abstractos, como a programação de tarefas cognitivas, sendo exemplo comum o reconhecimento de linguagem falada e escrita, podem ser implementadas recorrendo a estruturas denominadas redes neuronais artificiais, que de algum modo tentam tirar partido do que é conhecido sobre o funcionamento do cérebro. Para que estas estruturas possam resolver uma dada classe de problemas devem ser treinadas à priori. Ambas estas fases - treino e propagação - são mais convenientemente descritas recorrendo a algoritmos matriciais (Hassoun, 1995).

Por outro lado a representação de certos algoritmos no formato matricial simplifica a programação destes num computador digital. No entanto, e quando a dimensão das matrizes é grande, estes algoritmos são computacionalmente exigentes. Com o aparecimento das arquitecturas de computadores paralelos estes algoritmos tornaram-se sérios candidatos a versões paralelas. Com efeito, o tempo de execução destes algoritmos pode ser reduzido se o esforço computacional for dividido por mais do que um processador. Todavia, para se tirar partido destas arquitecturas, devido à falta de ferramentas apropriadas, o projectista despende uma considerável quantidade de tempo na paralelização do algoritmo sequencial. Outro problema normalmente encontrado relaciona-se com o facto desta paralelização ser altamente dependente do *hardware* utilizado. Assim, a portabilidade e adaptabilidade, bem como o processo de desenvolvimento de aplicações paralelas eficientes são tarefas que consomem imenso tempo, tornando o tempo de desenvolvimento destas aplicações bastante superior ao tempo de desenvolvimento da correspondente aplicação sequencial.

O conjunto de ferramentas que vai ser descrito nesta tese, também designado no texto como sistema de paralelização automático de algoritmos matriciais ou SPAM, permite que um algoritmo matricial paralelo seja automaticamente derivado a partir de uma descrição sequencial, tornando assim o tempo de desenvolvimento de aplicações paralelas e sequenciais virtualmente idêntico.

1.1 Paralelização de algoritmos matriciais

O particionamento de um algoritmo matricial pode ser conseguido dividindo o algoritmo em tarefas independentes que podem ser atribuídas a processadores diferentes - é este o caso da biblioteca de cálculo matricial paralelo Scalapack (Blackford et al, 1997), ou dividindo as matrizes operandos pelos processadores que executam o mesmo código. Este último método foi o adoptado nesta tese onde os operandos são divididos em conjuntos de linhas consecutivas. Desta forma o esforço computacional é razoavelmente bem equilibrado, com um custo mínimo em termos de código, de modo que são atingidos alto níveis de eficiência (Daniel e Ruano, 1999b). Este último indicador mede a percentagem do tempo total de execução em que os processadores estão a computar o algoritmo. O restante tempo é utilizado pelos processadores para comunicar dados entre si.

No entanto, embora este conceito de paralelização seja bastante simples, quando posto em prática usando ferramentas de programação convencionais, implica que tenham de ser desenvolvidos mecanismos de encaminhamento de mensagens através de uma rede de processadores, de modo que os dados possam ser comunicados entre os vários processadores

da rede. Como a eficiência do algoritmo paralelizado depende fortemente do tempo requerido por estes mecanismos de comunicação para enviar e receber mensagens, o programador é obrigado a dedicar grande parte do tempo atribuído ao projecto no desenvolvimento de mecanismos eficientes de comunicação.

Existem muitos estudos de encaminhamento óptimo de mensagens em topologias especificas tais como busca do caminho mais curto para rede incompletas WK-recursivas (Su et al, 1997), encaminhamento de mensagens em redes Bruijn (Hsu e Wei 1997) ou mais gerais tais como encaminhamento de mensagens, por deflexão, de muitos para muitos processadores em topologias tais como árvores e hipercubos (Borodin et al, 1997), encaminhamento de mensagens de todos para todos em arquitecturas de passagem de mensagem (Bruck et al, 1997), para citar apenas alguns. No entanto dificilmente estes algoritmos apresentam um desempenho óptimo para qualquer topologia de rede. Pacotes comerciais, como por exemplo o compilador Parallel C da 3L Ltd., implementam o conceito de canais de comunicação virtuais. Esta implementação é baseada num micro-núcleo alojado em cada processador, o qual gere a comunicação de mensagens, tornando a topologia de rede utilizada transparente para o programador, de modo que do ponto de vista deste, todos os processadores estão ligados directamente ou ponto a ponto. Existe no entanto uma perda de desempenho na velocidade de transferência de dados considerável quando esta facilidade é utilizada (3L Ltd., 1995), além de que esta não está implementada para todos os processadores suportados pelo compilador mas apenas para a família de processadores C4x. O SPAM incluí um mecanismo de encaminhamento de mensagens desenvolvido segundo uma estratégia que não privilegia nenhuma topologia em particular. Pelo contrário, implementa estratégias de comunicação de todos para todos os processadores e de um para um ou alguns, as quais tentam atingir um bom desempenho para qualquer rede, quer esta seja homogénea ou heterogénea, irregular ou regular, além de ser completamente transparente para o programador de aplicações.

Sobre este nível de abstracção assenta o programa automaticamente gerado pelo SPAM. Do ponto de vista do programador de aplicações, o código desta aplicação é descrito numa linguagem de programação sequencial, sendo depois traduzido por uma ferramenta, designada convenientemente por tradutor, para código fonte na linguagem C adequado ao modelo de programação paralelo adoptado. Este código utiliza uma biblioteca de cálculo matricial previamente desenvolvida, e optimizada para processamento paralelo, que poderá ser ampliada segundo as necessidades do programador.

Muitas vezes também é necessário investigar diferentes topologias de rede, de modo que seja possível optar pela mais adequada em termos de eficiência e velocidade de processamento.

Utilizando o SPAM, o programador de aplicações deve apenas indicar a topologia da rede alvo através de um diagrama de blocos sem ter de mudar uma linha de código.

Para que as diferentes topologias possam ser ensaiadas, um monitor permite um *interface* entre a consola e a rede de processadores. Este monitor permite carregar a aplicação para a rede, enviar dados para processamento, receber os dados tratados e, para que a eficiência da aplicação possa ser medida, conhecer os tempos de processamento em cada processador.

Está disponível no mercado um pacote (Jovian Systems Inc. 1997), o qual integra ferramentas para a geração de código paralelo, compatível com o compilador de C paralelo da 3L Ltd., que suportam redes homogéneas de Texas Instruments TMS320C40 DSPs e Analog Devices Sharc DSPs. Neste pacote o utilizador recorre a um diagrama de blocos para representar um sistema físico, a partir do qual o código paralelo é automaticamente gerado. Não é este o caso do SPAM, onde a fonte a partir do qual é gerado o código paralelo consiste numa linguagem de programação sequencial de alto nível.

O SPAM tal como o pacote acima descrito gera automaticamente código paralelo para o mesmo compilador. Correntemente suporta as seguintes plataformas paralelas homogéneas: Thompson Transputers, Texas Instruments TMS320C4x DSPs e Analog Devices Sharc ADSP2106xs, bem como uma plataforma heterogénea composta por Transputers e TMS320C4xs. No entanto como se tem verificado nos últimos anos uma rápida introdução de novos processadores, a construção do SPAM baseada em níveis de abstracção, permitirá que a introdução de novos processadores, com diferentes capacidades de computação e comunicação, seja uma tarefa facilitada.

1.2 Estrutura da tese

A tese está estruturada do seguinte modo:

Capítulo 2: Descrevem-se os modelos de processamento paralelo padrão e introduz-se o modelo utilizado. Discutem-se alguns métodos de paralelização de algoritmos, e extrapola-se o uso destes métodos para o caso de algoritmos matriciais. Descrevem-se os processadores utilizados em termos de diagrama de blocos. Finalmente os processadores são ensaiados em termos de desempenho.

Capítulo 3: O modelo de programação proposto é comparado com o modelo de programação paralelo convencional e o modelo de programação sequencial. É descrita a implementação da aplicação paralela gerada automaticamente, em termos de níveis cada vez mais abstractos, de modo que, do ponto de vista do nível superior, a arquitectura paralela física é invisível, sendo assumida uma arquitectura sequencial virtual.

Capítulo 4: O ambiente de comunicação, automaticamente criado pelo sistema de paralelização, é descrito. É introduzido o protocolo de comunicação utilizado e as comunicações são classificadas quanto à origem e ao destino. São discutidos mecanismos de comunicação em redes de processadores homogéneas e heterogéneas. São também introduzidos sub-programas de comunicação de matrizes através da rede paralela e com o computador anfitrião.

Capítulo 5: Neste capítulo são apresentadas as bibliotecas de suporte do sistema de desenvolvimento automático de algoritmos matriciais. Especial atenção é prestada às funções básicas de álgebra linear paralelas, incluídas nas biblioteca de cálculo matricial, bem como à estrutura das matrizes operandos, o seu modo de armazenamento e acesso a cada elemento. É introduzido o modelo de expansão desta biblioteca, por funções definidas pelo utilizador e formalizados os protótipos da mesma. São também abordadas funções de acesso aos recursos do computador anfitrião. São formalizadas as medidas de desempenho de algoritmos paralelos utilizadas, e avaliado o desempenho das funções de cálculo matricial básicas.

Capítulo 6: A sintaxe e a estrutura da linguagem sequencial suportada, SEQ 1.0, é introduzida. O tradutor é enquadrado dentro do SPAM 1.0, e o processo de tradução do código sequencial em código paralelo é pormenorizadamente descrito, indicando como são utilizadas as funções de comunicação e cálculo referenciadas nas bibliotecas anteriores.

Capítulo 7: O ambiente integrado de desenvolvimento de aplicações, o qual permite aceder a todas as ferramentas do sistema de desenvolvimento automático de aplicações matriciais paralelas, é abordado neste capítulo. São também descritas as ferramentas de configuração de redes de processadores e de gestão do projecto da aplicação. É apresentado um algoritmo de busca dos caminhos óptimos usando um grafo da rede alvo. Será ainda discutido um *interface* entre o SIMULINK e a aplicação paralela.

Capítulo 8: Neste capítulo são apresentados dois casos de estudo que ilustram a operacionalidade do ambiente de desenvolvimento de aplicações, bem como o desempenho das aplicações paralelas automaticamente geradas.

O primeiro consiste numa implementação de um algoritmo de controlo predictivo generalizado adaptativo. Este algoritmo é composto por dois estágios distintos: um estimador recursivo e o predictor propriamente dito. Visto que o primeiro opera normalmente sobre quantidades de dados muito inferiores às quantidades com que o segundo opera, o desempenho das aplicações geradas poderão ser comparados em ambos os casos.

O segundo caso consiste num algoritmo de treino *off-line* de uma rede neuronal, mais especificamente de um Perceptrão Multi-Camada, segundo um novo critério de aprendizagem desenvolvido por (Ruano, 1992).

Como ambos os casos requerem operações matriciais não implementadas nas bibliotecas de base, estes servem também para ilustrar como o utilizador poderá expandi-la.

2 Descrição do modelo de processamento

Neste capítulo são referenciados os paradigmas de programação correntemente aceites pela comunidade informática. São introduzidas arquitecturas e modelos de processamento aplicados ao paradigma de programação concorrente, e é apresentado o modelo de processamento paralelo adoptado. São discutidas técnicas de particionamento elementares sobre esse modelo. É feita uma breve descrição da arquitectura e do desempenho em condições ideais dos processadores utilizados. Além disso são estabelecidos alguns ensaios de desempenho aplicados a estes processadores de modo a avaliar e comparar o seu desempenho nas condições requeridas pelo modelo de implementação utilizado. Esta análise de desempenho em tempo real será utilizada para desenhar eficientes mecanismos de encaminhamento de mensagens entre processadores, descritos em pormenor no capítulo 4 e funções de cálculo matricial paralelas, descritas no capítulo 5.

2.1 Paradigmas de programação

Com o advento do computador digital emerge o conceito de linguagem de programação, um código simbólico usado para descrever uma sequência de acções a serem tomadas por este – o programa. Antes da década de 50 as linguagens caracterizavam-se por serem muito próximas da máquina e muito distantes da linguagem natural, sendo por isso classificadas como linguagens de baixo nível. São normalmente designadas por código máquina e consistem numa sequência de códigos numéricos - normalmente hexadecimais - que traduzem uma sequência de acções elementares a serem tomadas pelo processador alvo. Para tornar estas linguagens mais próximas do ser humano foi desenvolvido o conceito de linguagem *assembler*. Esta linguagem substitui os incompreensíveis códigos hexadecimais por menemónicas, ou abreviaturas de acções a operar no processador. No entanto estas linguagens

são fortemente dependentes da arquitectura dos processador alvo, tornando difícil a portabilidade do código entre diferentes plataformas.

As linguagens de programação de alto nível aproximam o código a uma linguagem natural, normalmente o inglês mas não exclusivamente, além de permitirem a portabilidade entre as plataformas que a suportam. A primeira linguagem de programação de alto nível é considerada como sendo FORTRAN, datando de 1954. Posteriormente foram-lhe adicionadas extensões surgindo o FORTRAN 77 e FORTRAN 90 (Nyhoff e Leestma, 1996). A partir da década de 50 deu-se uma proliferação de linguagens de programação, como é o caso da linguagens PASCAL (Cooper, 1983) e uma evolução desta: Modula-2 (Wirth, 1983) ou C (Kernighan e Ritchie, 1988), que tal como a linguagem FORTRAN implementam um algoritmo através de uma sequência de tarefas. Visto que esta sequência indica ao computador o procedimento a tomar para executar o algoritmo, estas linguagens são designadas por imperativas. Surge assim o paradigma de programação imperativo. Este paradigma reflecte a arquitectura dos computadores denominada por máquina de von Neumman (Burks et al, 1947; Aspray, 1991). Nesta arquitectura o processamento é sequencial visto que as tarefas que compõem um algoritmo são executadas sequencialmente. No entanto existem algoritmos que podem ser decompostos em tarefas independentes as quais podem ser executadas em simultâneo ou concorrentemente. Linguagens com este suporte, tais como o Occam 2 (Burns 1988), incluem-se no paradigma de programação concorrente. Por outro lado foram criadas extensões para suportar a programação concorrente em linguagens imperativas tais como PASCAL, C e FORTRAN. Com a evolução das ciências da computação outros paradigmas, que não são objecto de estudo desta tese, foram surgindo tais como os paradigmas orientados para objectos, funcional e lógico onde se incluem as linguagem C++, LISP e PROLOG respectivamente. (Bal e Grune, 1994).

2.2 Arquitecturas e modelos de processamento concorrente

Além do processamento concorrente ser um processo natural e universal - as células constituintes do cérebro, os neurónios, estão interligadas numa rede por onde se propagam sinais eléctricos; é vulgar a execução de tarefas em paralelo por seres humanos - uma vantagem óbvia da programação concorrente é a possibilidade de paralelização de um algoritmo, ou seja o mapeamento de tarefas concorrentes em mais do que um processador reduzindo assim o tempo de execução. Assim no caso de processadores alocados no mesmo computador designa-se este modelo de processamento por processamento paralelo e para o caso de processadores alocados em diferentes computadores, por processamento distribuído.

Mas para que um algoritmo possa ser mapeado por mais que um processador tem de existir forçosamente alguma transferência de informação entre processadores. Deste modo definem-se duas arquitecturas básicas em termos de acoplamento de processadores: passagem de mensagem e memória partilhada. No primeiro caso, ilustrado na Fig. 2-1, assume-se que os processadores têm memória local e trocam dados através de canais físicos dedicados.

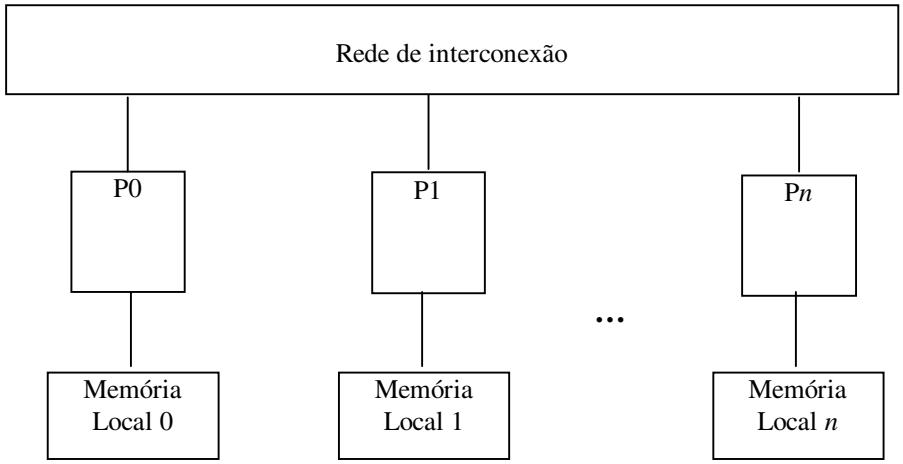


Fig. 2-1: Rede de processadores acoplados segundo a arquitectura de passagem de mensagem.

No segundo caso os processadores não possuem memória local. Estão conectados a um barramento comum, que por sua vez permite que qualquer processador possa aceder aos blocos de memória partilhada como se pode ver no diagrama da Fig. 2-2. Além disso existe a possibilidade de alternativas híbridas, onde processadores acoplados segundo a arquitectura de memória partilhada também possuem memória local. (Bertsekas e Tsitsiklis 1989).

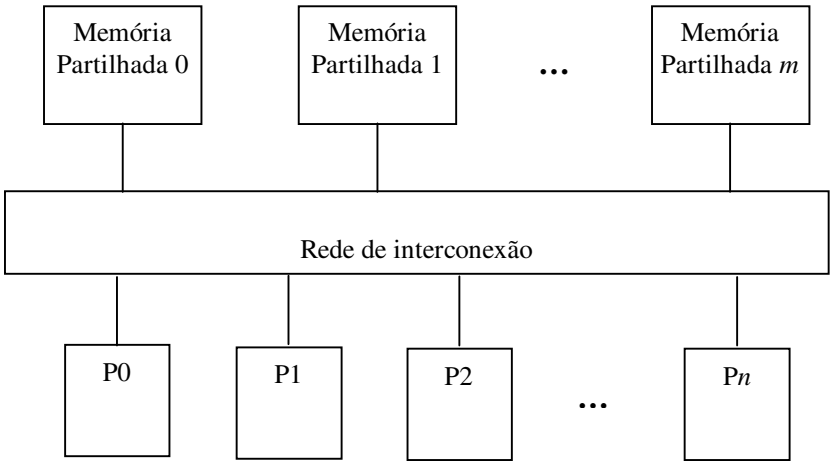


Fig. 2-2: Rede de processadores acoplados segundo a arquitectura de memória partilhada.

Até agora foi assumido que em cada processador existe apenas um processo ou tarefa. No entanto é comum serem atribuídas mais do que uma tarefa a um só processador. O modo

como estas tarefas comunicam entre si pode ser também expresso em termos das arquitecturas acima descritas. Deste modo todas as tarefas residentes num só processador podem aceder a um segmento de memória partilhada ou trocar mensagens entre si através de canais lógicos.

Em termos de programação concorrente o modelo vulgarmente aceite consiste na ideia de processos sequenciais comunicantes (Hoare 1985). Neste modelo um algoritmo é visto como uma colecção de processos concorrentes que comunicam através de canais. O processo ou tarefa é composto por um número qualquer de operações sequenciais. Embora este modelo reflicta a arquitectura de passagem de mensagem pode também ser implementado numa arquitectura de memória partilhada se os canais lógicos forem implementados como segmentos de memória partilhada.

Um aspecto importante nos modelos de programação concorrente é a sincronização de processos ou tarefas. Na arquitectura de passagem de mensagem a sincronização é feita pelas próprias primitivas de comunicação, no entanto na arquitectura de memória partilhada têm de existir mecanismos explícitos de sincronização que garantam a exclusão mútua. Ou seja, estes mecanismos devem impedir que mais do que um processo aceda a um mesmo segmento de memória, ou outro recurso, no mesmo intervalo de tempo. Neste cenário um dos processos terá de esperar que o outro liberte o recurso, dando-se assim uma colisão, mas evitando a corrupção de dados ao custo do desempenho. Tal sucede vulgarmente em algoritmos matriciais. Neste contexto, (Charny, 1996) desenvolveu um algoritmo que optimiza a paralelização do algoritmo matricial de modo a minimizar as colisões numa arquitectura de memória partilhada.

Quanto à arquitectura de passagem de mensagem, esta também sofre de uma perda de desempenho na medida em que a taxa de transferência de dados, através dos canais físicos, é normalmente inferior à velocidade de acesso a um banco de memória. Além desta condicionante, quando o número de canais físicos que cada processador dispõe é inferior ao número de processadores com que o primeiro deve comunicar, a transferência de mensagens entre dois processadores distantes terá de ser encaminhada através de outros processadores. Na Fig. 2-3 estão representados 2 caminhos possíveis para enviar mensagens entre os processadores P0 e P7 numa arquitectura de passagem de mensagem. Considerando que a rede paralela desta figura é homogénea, isto é todos os processadores são idênticos, tal como é idêntica a taxa de transferência de dados através dos canais físicos, o caminho representado a cheio é o mais curto, ao passo que o caminho a tracejado é mais longo. No entanto tal poderá não ser verdade considerando uma rede paralela heterogénea, onde os processadores podem ter características diferentes e consequentemente taxas de transferência de dados diferentes.

Além disso, nesta figura está representada uma topologia de rede paralela bastante regular, denominada matriz de processadores visto que estes estão dispostos numa configuração semelhante a uma matriz bidimensional. Assim, para o caso de uma rede heterogénea irregular o problema de encontrar o caminho mais curto entre dois processadores torna-se ainda mais complexo. Métodos para lidar com este problema serão referidos no capítulo 7. Pode-se assim generalizar que, independentemente da arquitectura, um algoritmo paralelo atinge um melhor desempenho quanto menor for o caminho entre processadores.

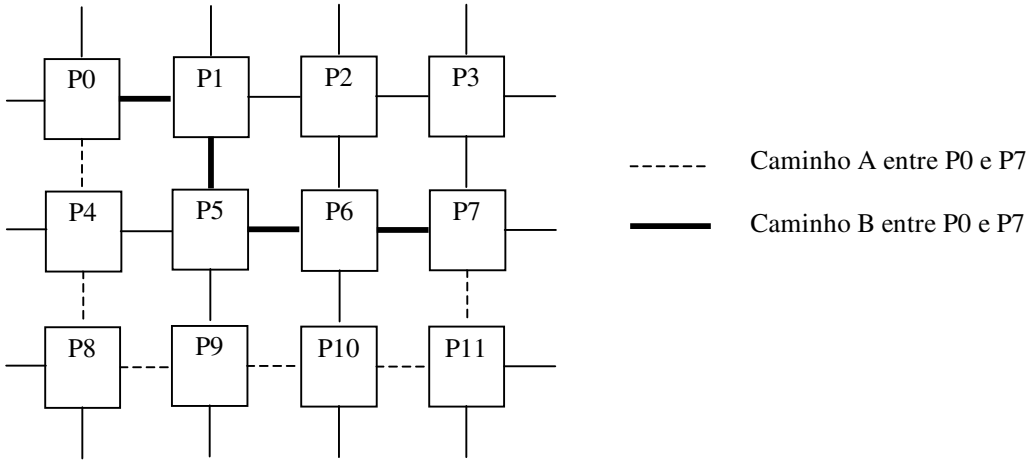


Fig. 2-3: Dois caminhos possíveis entre o processador 0 e o processador 7.

Para além de minimizar as comunicações, um bom particionamento do algoritmo deve dividir o esforço computacional por todos os processadores da rede paralela o mais homogeneamente possível. Deste modo maximiza-se a utilização dos processadores minimizando o tempo de execução do algoritmo. O equilíbrio de carga pode ser conseguido dividindo o algoritmo em tarefas independentes, com um grau de complexidade idêntico, as quais podem ser atribuídas a processadores diferentes. Quanto mais homogéneo for o grau de complexidade das tarefas, que se assume proporcional ao tempo de execução destas, mais eficaz é o equilíbrio da carga. O particionamento de uma simples equação matricial como a equação (2-1) ilustra esta técnica.

(2-1) $$\mathbf{R} = \mathbf{A} \cdot \mathbf{B} + \mathbf{C} \cdot \mathbf{D}$$

Na Fig. 2-4, está representada uma possível estratégia de atribuição de tarefas para esta equação matricial, em dois processadores. Assumindo que as matrizes **A** e **C** têm dimensões

idênticas bem como as matrizes **B** e **D**, pode-se dividir esta equação em 3 tarefas: os produtos **A** . **B** e **C** . **D** e a soma destes. Pode-se então atribuir estes produtos a processadores diferentes de modo que sejam computados concorrentemente. Finalmente será necessário transmitir o resultado de um dos produtos, por exemplo β = **C** . **D**, de um processador para outro e efectuar a soma dos produtos neste último processador. Deste modo a carga computacional está tanto mais desequilibrada quanto maior for a dimensão dos produtos α e β. Assim verifica-se que esta técnica é fortemente dependente do algoritmo. Existem outras estratégias de distribuição dos operandos possíveis. A biblioteca de álgebra linear Scalapack, por exemplo, divide as operações com matrizes em tarefas mais elementares que operam sobre sub-matrizes em vez de operar sobre todo o operando conseguindo assim um grão mais fino de particionamento e consequentemente um equilíbrio de tarefas mais eficaz (Blackford et al, 1997).

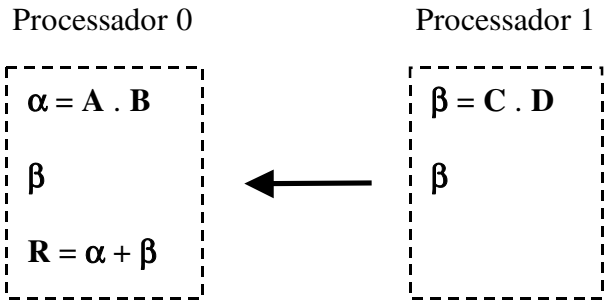


Fig. 2-4: Particionamento por tarefas da equação (2-1).

Esta distribuição de tarefas pode ser feita durante a compilação ou durante a execução. Um compilador para algoritmos matriciais, o qual representa as tarefas a ser escalonadas durante a compilação através de Grafos Acíclicos Directos, foi introduzido por (Prasanna e Musicus, 1996). No entanto o *speedup* ou aceleração, o qual pode ser visto como o número de vezes que o tempo de execução de determinado algoritmo é reduzido quando mapeado por mais do que um processador, atingido pelos algoritmos compilados com esta ferramenta poderia ser algo melhorado. Por outro lado existem algoritmos cujo escalonamento de tarefas só pode ser feito durante a execução. Por exemplo cláusulas *se-então-senão* podem necessitar de ser escalonadas de diferentes modos durante a execução. Nestes casos é necessário distribuir as tarefas durante a execução. Dois modelos de atribuição dinâmica de tarefas são propostos por (Chang e Oldham, 1995).

É ainda importante considerar o particionamento de ciclos. Podem ser usadas técnicas de *pipeline* (Chao et al, 1993), onde o código interior do ciclo é distribuído por vários

processadores, de modo que em cada iteração este é executado em paralelo. No entanto alguns ciclos encadeados não podem ser paralelizados desta forma devido a dependências. Isto quer dizer que quando uma operação depende de um valor obtido numa iteração precedente ambas as operações devem ser executadas sequencialmente. Uma nova classe de transformações de ciclos com o intuito de particionar tais ciclos é tratada por (Rim e Rajiv, 1996). Estas transformações são aplicadas à paralelização de multiplicações matriciais e vectoriais. O particionamento de tarefas ainda se torna mais complexo quando consideramos ambientes de processamento paralelo heterogéneos. Nestes ambientes o particionamento deve ser efectuado de modo que os recursos dos processadores sejam usados eficientemente. Assim se numa rede paralela existirem um ou mais processadores vectoriais, processadores estes optimizados para o cálculo vectorial e matricial, as tarefas de cálculo vectorial devem ser a estes atribuídas. Uma contribuição que tenta resolver o problema do equilíbrio da carga de tarefas em tais ambientes é apresentado em (Tan e Antonio, 1997).

Mas o equilíbrio da carga também pode ser feito ao nível dos dados. Algoritmos que operam sobre grandes quantidades de dados, tais como operações matriciais de grandes dimensões, são bons candidatos a esta técnica de particionamento.

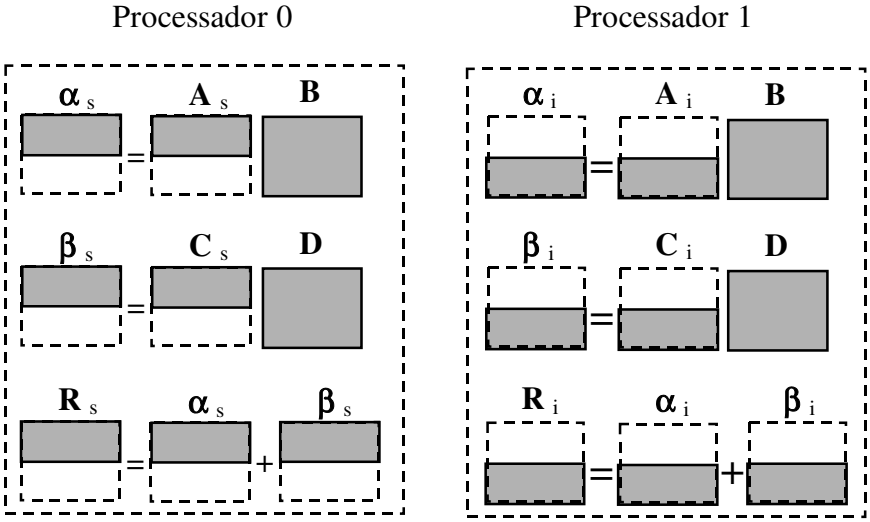


Fig. 2-5: Particionamento de dados da equação (2-1).

Na Fig. 2-5 está esquematizado o particionamento dos operandos usados na equação (2-1) por dois processadores. É assumido que ambos os processadores executam o mesmo código mas operam sobre conjuntos de dados diferentes. Assim o conjunto de operações executado consiste na decomposição da equação de acordo com a precedência dos operadores:

$$\alpha = A \cdot B$$

$\beta = C \cdot D$

$R = \alpha + \beta$

Um processador opera sobre a metade superior dos operandos e o outro sobre a metade inferior, reduzindo assim o tempo de cálculo para metade, desprezando o tempo de comunicação. Ao contrário do particionamento de tarefas este é independente da topologia da rede paralela alvo. Facilmente se pode generalizar esta técnica de particionamento a uma rede com mais de dois processadores, dividindo as matrizes operandos em tantos conjuntos de linhas consecutivas como o número de processadores da rede. Um equilíbrio de carga óptimo, considerando uma rede homogénea, é atingido se os conjuntos de linhas consecutivas forem compostos pelo mesmo número de linhas em cada processador. No caso de uma rede heterogénea, o equilíbrio de carga óptimo pode ser atingido, armazenando um número de linhas proporcional às características da família do processador, em cada processador.
São ainda possíveis outras estratégias de particionamento dos elementos das matrizes mas a técnica descrita acima é das mais simples em termos de implementação.

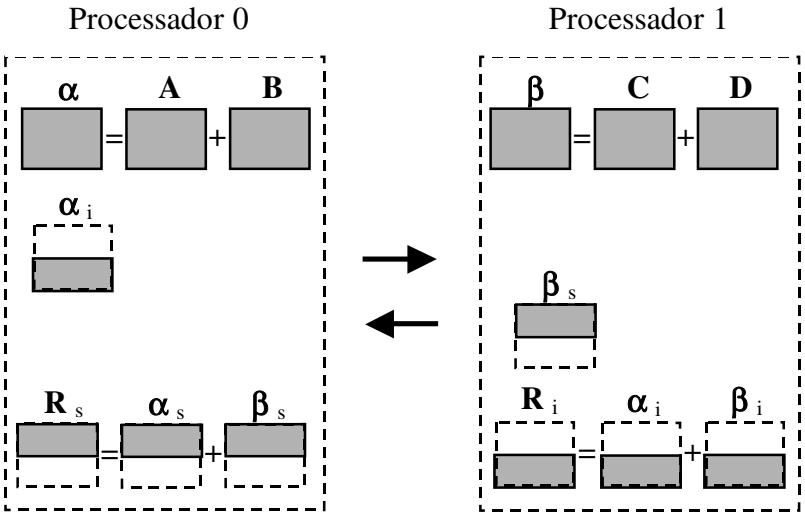


Fig. 2-6: Particionamento híbrido da equação (2-1).

Uma estratégia de particionamento híbrida também poderá ser implementada. Voltando ao exemplo de particionamento da equação (2-1), segundo esta estratégia os produtos matriciais são paralelizados recorrendo ao particionamento de tarefas. Seguidamente os dados são redistribuídos de modo que a soma possa ser paralelizada recorrendo ao particionamento de dados como ilustra a Fig. 2-6.

Esta estratégia é mais eficiente que o particionamento de tarefas pois ambos os processadores são rentabilizados ao máximo. No entanto este método de particionamento, tal como o primeiro apresentado, sofre de dependências algorítmicas.

O modelo de processamento paralelo adoptado usa uma arquitectura de passagem de mensagem. Neste modelo, o particionamento é feito ao nível dos dados, por ser mais simples de implementar e evitar dependências algorítmicas, como já foi atrás referido. Além disso, esta técnica de particionamento apresenta altos níveis de eficiência para algoritmos matriciais massivos (Piedra, 1991; Daniel e Ruano, 1999b). A implementação deste modelo é descrita no capítulo 3. Esta implementação suporta redes homogéneas compostas por Thompson Transputers (ou simplesmente Transputers), Texas Instruments TMS320C40 DSPs (ou C40s) e Analog Devices Sharc DSPs (ou Sharcs para abreviar). São também suportadas redes heterogéneas constituídas por C40s e Transputers. O desempenho destes processadores será descrito e avaliado nos seguintes pontos. No entanto, como se tem verificado nos últimos anos uma rápida introdução de novos processadores, este modelo permite a introdução de novos processadores correntemente não suportados.

2.3 Características técnicas dos processadores utilizados

As características apresentadas neste ponto baseiam-se em dados recolhidos em manuais técnicos. Na sua maior parte traduzem desempenhos em condições óptimas, as quais nem sempre podem ser estabelecidas. Como, no âmbito desta tese, os processadores são utilizados como elementos de construção de uma rede de processamento paralelo, com o intuito de processar algoritmos de cálculo intensivo, a seguinte descrição centrar-se-á principalmente na capacidade de processamento e comunicação de dados destes processadores.

2.3.1 Transputer

Os Transputers surgiram em meados da década de 80 e foram considerados por alguns como um computador num único circuito integrado. De qualquer forma, o facto é que um destes dispositivos inclui um processador ou CPU de 32 *bits* que suporta dois níveis de prioridade para processos, até 4Kbytes de memória RAM local de acesso rápido e até quatro vias de comunicação série, denominadas *links*, que permitem a conexão ponto a ponto entre Transputers, ou entre Transputers e outras famílias de processadores, desde que se disponha de um adaptador apropriado. Contêm também um interface de memória programável, que permite o acesso a um espaço de endereços até 4GBytes. Passado mais de uma década, verifica-se que os microprocessadores actuais contêm num único circuito integrado um

conjunto de facilidades semelhante ou superior ao Transputer, como se verá a seguir. O Transputer utilizado neste trabalho foi o T805 com velocidade de relógio de 25 MHz. O seu diagrama de blocos simplificado está na Fig. 2-7.

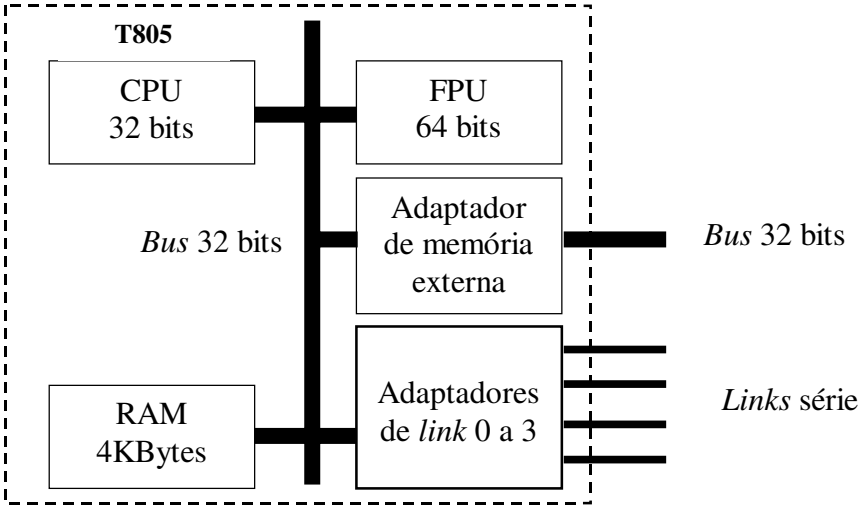


Fig. 2-7: Diagrama de blocos simplificado de um Transputer T805.

Para além das características já apontadas, este processador contém uma unidade de vírgula flutuante, também designada por FPU (do inglês *floating point unit*) de 64 bits. Esta unidade permite operações aritméticas de precisão simples e dupla, executando-as concorrentemente com a CPU, alcançando um desempenho de pico de 2,75 milhões de operações de vírgula flutuante por segundo ou MFlops. Os 4 *links* permitem uma taxa de transferência de informação máxima de 20Mbits por segundo ou 2,35 Mbytes por segundo bidireccionalmente e concorrentemente. A quantidade de informação mínima que pode ser transmitida é o *byte*. (Inmos Ltd., 1990a; 1990b)

Quando uma rede de Transputers está fisicamente alocada num computador anfitrião, também denominado *host*, como por exemplo um PC, um dos processadores, denominado raiz ou *root*, é usado para estabelecer uma via de comunicação com o anfitrião através de um dos *links*. É o caso do material utilizado neste trabalho, no qual toda a comunicação entre qualquer dos processadores da rede e o anfitrião deve ser direccionada através dessa via (Inmos Ltd., 1990c; Transtech Paralell Systems Ltd., 1991a; 1991b).

2.3.2 TMS320C40

A família de processadores TMS320C4x, desenvolvida pela Texas Instruments, é composta por processadores digitais de sinal com aritmética de virgula flutuante. No diagrama de blocos representado na Fig. 2-8 pode-se observar a arquitectura simplificada de um TMS320C40. A unidade central de processamento incluí uma unidade lógico aritmética, ALU, e um

multiplicador que suportam aritmética inteira de 32 bits e aritmética de vírgula flutuante de 32 ou 40 bits. Instruções paralelas permitem que ambos os dispositivos funcionem concorrentemente de modo que uma operação de multiplicação e uma operação aritmético-lógica sejam executadas no mesmo ciclo. Para tal duas unidades aritméticas auxiliares geram dois endereços num só ciclo. O CPU dispõe também 12 registos de precisão estendida de 40 bits e 8 registos auxiliares de 32 bits que podem ser acedidos pelo multiplicador e pela ALU.

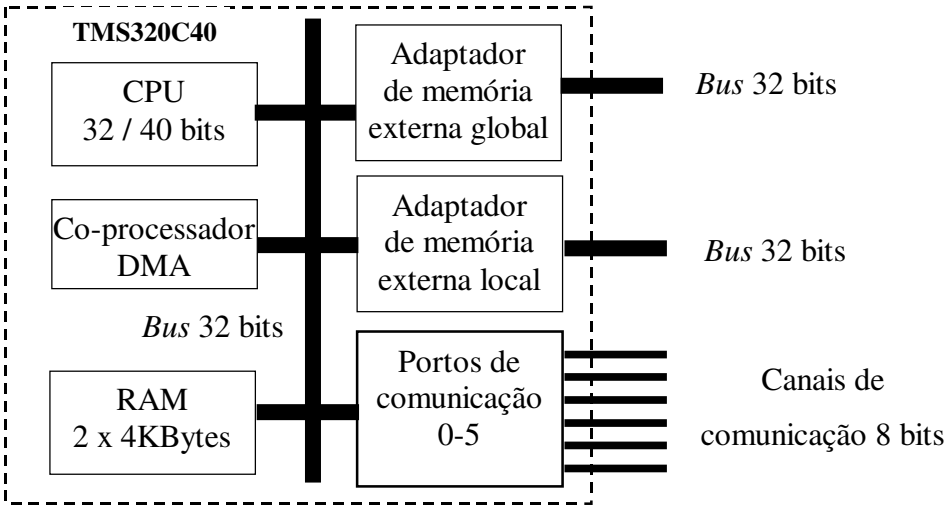


Fig. 2-8: Diagrama de blocos simplificado de um TMS320C40.

Dois blocos de 4 Kbytes de memória RAM, de acesso rápido, estão alocados no circuito integrado. A arquitectura de duplo barramento permite dois acessos à memória de dados em cada ciclo. Além destes blocos de memória interna o C40 dispões de dois adaptadores de memória externa idênticos denominados de global e local, atingindo-se um espaço de endereçamento total de 16 Gbytes. Dispõe também de um co-processador DMA, que permite o acesso a qualquer endereço de memória sem interferir com a operação do CPU.

Para comunicar com outros processadores, da mesma família, dispõe de seis portos de comunicação paralelos bidireccionais *half duplex*. Para obter comunicação bidireccional *full duplex* usa-se um par de portos. Cada porto tem uma taxa de transferência de informação máxima de 20 Mbytes por segundo, sobre 8 linhas de dados e 4 linhas de controlo. Os portos de comunicação operam concorrentemente. A quantidade atómica de informação que pode ser transmitida é a palavra de 32 bits (Texas Instruments, 1996).

Para este trabalho foram utilizados módulos de processamento HET40SDX, os quais incluem um TMS320C40 com velocidade de relógio de 50 MHz, o que significa que cada ciclo demora 40 ns, 1 banco de memória externa local, com 1 MB SRAM e dois bancos de memória externa global, com 2 MB de RAM cada (Warnes, 1994). Assim, como é possível

efectuar duas operações de vírgula flutuante num só ciclo, o pico de desempenho do CPU atinge os 50 MFlops. Módulos adaptadores entre os portos de comunicação paralelos dos C40s e os links série dos Transputers (Williams, 1995), foram utilizados para construir redes heterogéneas baseadas nestes processadores. A carta mãe utilizada é inserida num suporte de expansão de um PC. A comunicação entre este e a rede de C40s é estabelecida através de um dos portos de comunicação do processador raiz (Warnes, 1993).

2.3.3 ADSP21060

Este processador, desenvolvido pela Analog Devices, também é um processador digital de sinal, no entanto no âmbito deste trabalho, será utilizado apenas para cálculo em redes homogéneas. Na Fig. 2-9 está representada a arquitectura simplificada deste processador.

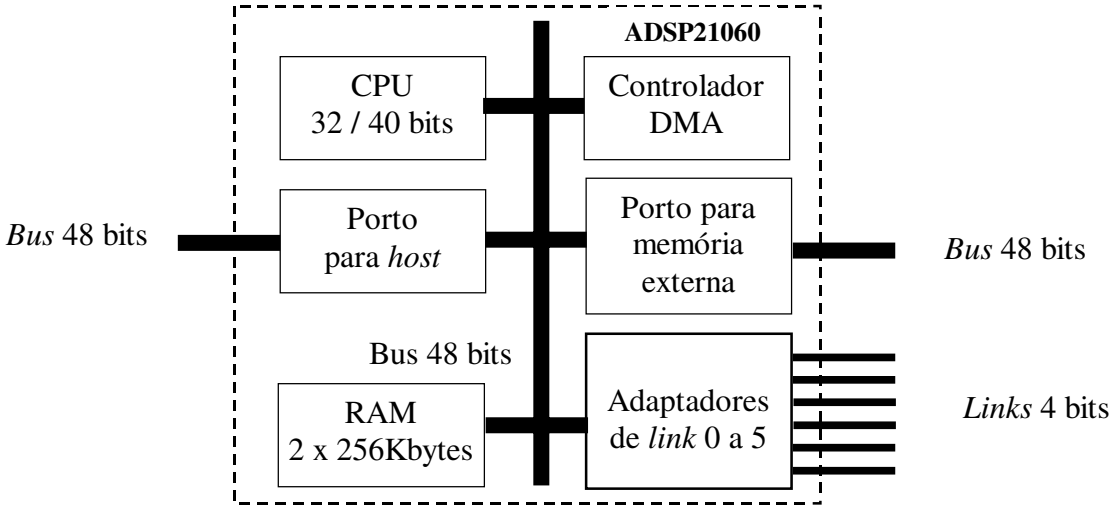


Fig. 2-9: Arquitectura simplificada de um ADSP21060.

A unidade central de processamento contêm uma unidade aritmético-lógica e um multiplicador que podem operar em paralelo sobre dados inteiros ou de vírgula fixa de 32 bits e sobre números de vírgula flutuante de 32 ou 40 bits. Esta família de processadores executa todas as instruções em apenas um ciclo. Além das instruções tradicionais de um DSP - multiplicação, adição e multiplicação e adição combinadas – contém instruções que retornam o mínimo e o máximo de dois operandos, instruções que permitem operar em simultâneo com a ALU e o multiplicador ou fornecer dois operandos a uma destas unidades computacionais num só ciclo. Para que tais operações simultâneas sejam possíveis, o CPU dispõe de dois geradores de endereços, os quais permitem calcular simultaneamente dois endereços para ler ou escrever dois operandos. Além disso uma mesma instrução, a qual é codificada numa palavra de 48 bits, suporta também operações aritméticas em paralelo com transferências de

dados. A unidade central de processamento contém 16 registos primários e 16 secundários. Todos os registo têm comprimento de 40 bits e são de uso geral. Os registos secundários são usados para partilha de dados entre diferentes contextos. A necessidade de mudança de contexto surge quando dois ou mais processos partilham o mesmo processador. Assim, o tempo de execução deve ser partilhado por estes processos, de modo que, periodicamente é necessário que a um processo seja atribuído o processador, para que seja executado, e que seja previamente retirado a outro processo. O uso de registos secundários permite que o acesso a dados partilhados por vários processos seja mais rápido do que se verificaria se estes estivessem guardados em endereços de memória.

Este processador contém também dois bancos de 256 Kbytes de memória SRAM interna. Estes bancos podem ser configurados para armazenar dados ou código. Cada banco de memória tem um duplo porto, o que permite, em apenas um ciclo, acessos independentes a partir do CPU, do processador de entrada saída e do controlador de DMA. Os acessos à memória são mais eficientes se num dos blocos for guardado código e dados, e noutro bloco apenas dados, usando cada um destes bancos um barramento independente para transferência de dados. Desta forma, num único ciclo, é possível executar uma operação com duas transferências de dados, desde que a palavra de instrução esteja na memória de *cache* interna. Também é possível executar uma instrução semelhante num único ciclo, se um dos operandos estiver guardado na memória externa. Esta memória externa, organizada em apenas um banco, tem um espaço de endereçamento máximo de 32 Gbytes.

Um processador de entrada-saída está também integrado no mesmo invólucro. Este processador incluí dois portos série assíncronos, para conexão com periféricos, que têm uma taxa de transferência máxima de 40 Mbits. Contêm também um controlador DMA e seis portos para comunicação ponto a ponto com processadores da mesma família. Estes portos, também denominados *links*, são composto de 4 linhas de comunicação. Podem executar duas transferências de dados em cada ciclo, transmitindo assim um byte por ciclo; a quantidade de informação mínima que pode ser transmitida é no entanto uma palavra de 32 bits. Tal como no caso do Transputer estes portos operam concorrentemente e bidireccionalmente.

Um porto de comunicação, dedicado à comunicação com computador anfitrião, liberta um *link* do processador raiz, para comunicação com outros Sharcs. Tal não sucede no caso dos Transputers e dos C40s, em que uma das vias de comunicação do processador raiz é usada para comunicação com o anfitrião (Analog Devices, 1997).

Os módulos de processamento utilizados operam a uma velocidade de relógio de 40 MHz, de modo que o pico de performance do CPU é de 120 MFlops. Os *links* têm uma taxa de

comunicação de 40 Mbytes por segundo. Estes módulos dispõem de 1 banco de memória externa com 3 Mbytes SRAM (Spectrum Signal Processing Inc.,1998).

2.4 Avaliação de desempenho dos processadores utilizados.

Se bem que a linguagem de programação natural do Transputer seja o Occam2, visto que foram desenvolvidos simultaneamente com o intuito desta reflectir a arquitectura do Transputer, existem implementações eficientes de outras linguagens, tais como PASCAL e C. Por questões de portabilidade e futura expansão, foi adoptada para este trabalho a linguagem C. Mais especificamente versões de ANSI C com extensões para processamento paralelo desenvolvidas pela 3L Ltd. para Transputer (3L Ltd., 1991), C4x (3L Ltd., 1995) e Sharc (3L Ltd., 1998), as três famílias de processadores utilizados. Deste modo, as medidas de desempenho a seguir abordadas, serão obtidas com estas linguagens, não podendo ser consideradas absolutas, mas sim referentes aos compiladores utilizados. No entanto estes compiladores incorporam facilidades de optimização de código, quer automáticas quer controladas pelo programador, as quais permitem aproximar o desempenho das aplicações compiladas das aplicações desenvolvidas em código nativo.

2.4.1 Medidas de desempenho do CPU

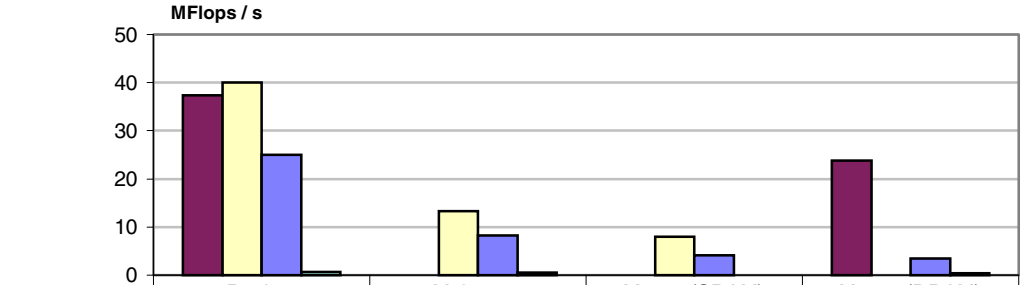
A medida de desempenho de cálculo escalar do CPU, indica o tempo necessário para computar um determinado número de operações de virgula flutuante. Esta medida não deve ser vista como uma medida computacional pura, mas sim como o tempo necessário para aceder a dois operandos, armazenados em dois determinados blocos de memória, realizar uma operação de vírgula flutuante sobre estes, e finalmente armazenar o resultado desta operação. Assim, na Fig. 2-10, está indicado o número de operações de vírgula flutuante realizadas num segundo, MFlops, nas condições acima descritas.

Como se pode observar, os processadores utilizados neste trabalho, são comparados com um processador vulgarmente usado na gama alta dos computadores pessoais, aproximadamente à data da introdução do ADSP21060, o Pentium II - 266 MHz, de modo a enquadrar esta avaliação de desempenho também no âmbito destas arquitecturas mais comuns. É conveniente no entanto referir que a arquitectura do Pentium II é cerca de 12 anos mais recente que a arquitectura T80X e sete que a C4X.

Os resultados desta medida de desempenho são apresentados em quatro grupos, correspondendo a cada um deles uma estratégia de armazenamento de operandos diferente. Assim, no primeiro grupo, ambos os operandos são armazenados em registos. No segundo,

ambos os operandos são armazenados num banco de memória interna. Este caso não foi ensaiado no P2, pois esta arquitectura não possuí bancos de memória interna endereçáveis. No terceiro caso, ambos os operandos encontram-se armazenados na memória externa SRAM. Dos processadores testados, apenas o ADSP21060 e o C40 dispõem de bancos de memória deste tipo. O último grupo avalia o desempenho, quando os operandos estão também armazenados em bancos de memória externa, mas neste caso DRAM. O ADSP21060 é o único processador testado que não possuí bancos de memória externa deste tipo. As medidas de desempenho destes dois últimos grupos, são fortemente dependentes das características da memória externa, e apenas são precisas para as versões dos módulos de processamento ensaiados.

Nestes quatro casos foi considerado que o código está armazenado na memória externa. No caso do C40 é irrelevante em que bloco de memória está armazenado o código, o que sugere que o processamento deste algoritmo, é efectuado de modo que a instrução seguinte encontra-se sempre disponível na memória de *cache* interna.



	Registo	M. interna	M. ext. (SRAM)	M. ext. (DRAM)
P2	37,40			23,81
ADSP21060	40,00	13,33	8,00	
C40	25,00	8,26	4,15	3,51
T805	0,61	0,54		0,44

Fig. 2-10: Desempenho escalar das várias FPUs.

Foi verificado, em qualquer dos processadores ensaiados, que a relação entre o número de operações de virgula flutuante e o tempo necessário para as computar é linear. Como está patente na figura, em modos gerais o ADSP21060 é o processador mais rápido, seguido pelo C40 e finalmente pelo T805. Comparando estes processadores com o P2, em termos de computação absoluta, isto é, considerando que os operandos estão armazenados em registos do núcleo do processador, verifica-se que este segue de perto o ADSP21060. Armazenando no entanto os operandos na memória externa, qualquer que seja o tipo, o P2 é o processador mais rápido, seguido de longe pelo ADSP21060. Por outro lado, como seria de esperar, verifica-se que o desempenho é o melhor quando os operandos estão armazenados nos

registos. Excluindo os registos, o melhor desempenho é atingido armazenando os operandos na memória interna. Por outro lado, armazenando os operandos na memória externa, obtém-se o pior desempenho. Se estiverem disponíveis bancos de memória externa de diversos tipos, o que só sucede no C40, verifica-se que é preferível armazenar dados na memória SRAM. Outras medidas a tomar em conta, de um modo geral, para obter uma estratégia de armazenamento de dados que optimize o tempo de acesso a estes, incluem o armazenamento de dados no *heap* ou no *stack*, evitar declarar os ponteiros para esses como voláteis e armazená-los em registos.

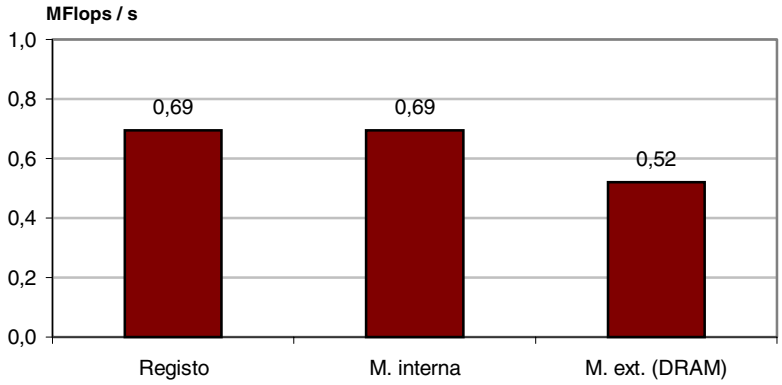


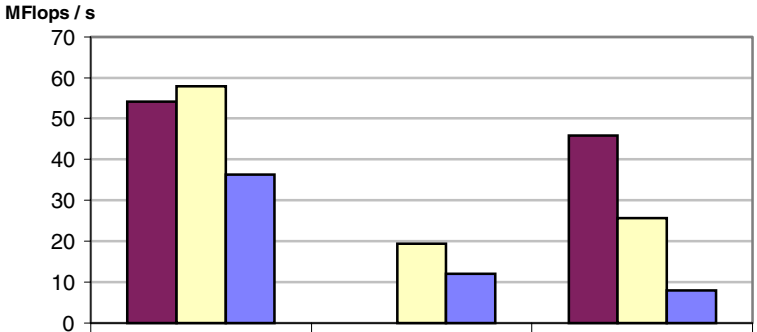
Fig. 2-11: Desempenho da FPU do T805 armazenando o código na memória interna.

O desempenho pode ser melhorado, no caso do T805, se existir espaço na memória interna para armazenar também o código. Tal, no entanto nem sempre é possível devido à reduzida dimensão deste banco de memória. A Fig. 2-11 ilustra o desempenho neste caso. Repare-se que, se o código estiver armazenado na memória interna, o tempo de processamento é idêntico quer os operandos residam em registos ou na memória interna. Além disso, sempre que possível, é preferível armazenar código na memória interna em detrimento dos operandos, pois esta estratégia permite atingir uma velocidade de processamento de 0,52 MFlops, como se pode observar nesta figura, enquanto com a estratégia inversa apenas se obtém 0,44 MFlops, como está patente na figura anterior.

No caso do processador ADSP21060, é ainda possível usar estratégias de alocação de memória que tirem partido dos múltiplos barramentos e blocos de memória deste processador, permitindo reduzir o tempo de acesso à memória externa. Para tal é necessário que o segmento de código da aplicação resida na memória interna. Deste modo, o tempo de acesso a operandos na memória externa é igual ao tempo requerido para aceder à memória interna. Pode-se assim generalizar, para este processador, que a estratégia de optimização do tempo de acesso à memória, qualquer que seja o tipo ensaiado, implica armazenamento de código e

dados em bancos de memória diferentes. Por outro lado, se os operandos estiverem armazenados em registos, é indiferente em que tipo de memória o código é armazenado.

Verificou-se ainda que, se uma rede de Sharcs usar o barramento PCI para aceder aos recursos do computador anfitrião, e o código for armazenado na memória externa, independentemente do tipo de memória onde são armazenados os dados, o desempenho do processador raiz deteriora-se. Nestas condições, o desempenho não é constante, mas sim variável. Esta variação tem como limite superior o desempenho não deteriorado, ou seja o desempenho de qualquer processador da rede que não o raiz, e como limite inferior metade do anterior. Também se verificou, que existe uma maior tendência para que o desempenho deteriorado se aproxime do limite inferior. Este comportamento sugere que o *micro-kernel*, alocado automaticamente pelo compilador usado e já referido neste ponto, lança processos que continuamente interrogam o barramento PCI por dados, reduzindo assim o desempenho.



	REG	MEM1	MEM2
P2 / T805	54,20		45,79
ADSP21060 / T805	57,97	19,32	25,63
C40 / T805	36,23	11,97	7,98

Fig. 2-12: Comparação do desempenho dos processadores ensaiados.

A figura acima compara o desempenho dos vários processadores ensaiados, quando os operandos estão armazenados nos registos, a mais eficiente estratégia de armazenamento de operandos na memória interna, e a melhor estratégia de armazenamento de operandos matriciais de grandes dimensões, referidas na figura como REG, MEM1 e MEM2 respectivamente.

Neste último caso, foi considerado que os operandos são armazenados na memória externa, para o T805 e na memória local para o C40, visto que os bancos de memória interna destes processadores são demasiado reduzidos para desenvolver uma estratégia realística de armazenamento operadores de grandes dimensões. No caso do Sharc, ambas as estratégias de

armazenamento são idênticas, visto que o tempo de acesso à memória interna é idêntico ao tempo de acesso à memória externa SRAM.

Visto que o T805 é o processador absolutamente mais lento, este é usado como ponto de referência para comparar todos os outros processadores. Pode-se assim observar, que todos os processadores são muito mais rápidos que o T805 em qualquer dos casos, mas principalmente em computação pura, onde os operandos estão armazenados em registos. Assim, em termos absolutos ADSP21060 é aproximadamente 58 vezes mais rápido que o T805, 1,6 vezes que o C40 e 1,07 que o P2. No caso de armazenamento de operandos grandes em memória, o P2 é o processador mais rápido, batendo o T805 por um factor de 45,79, o C40 por 5,74 e mesmo o ADSP21060 por 1,79.

2.4.2 Medidas de desempenho dos portos de comunicação

Estas medidas de desempenho consistem na avaliação da velocidade de transferência de dados entre dois processadores da mesma família, e entre processadores de famílias diferentes, nomeadamente entre Transputers e C40s, através da medição dos tempos de comunicação de vectores de dados. O comprimento destes vectores foi variado, de modo a se comprovar a linearidade desta taxa de transferência em relação ao comprimento dos dados.

Serão avaliadas transferências de dados unidireccionais e bidireccionais *full-duplex*, além de transferências concorrentes através de portos diferentes. Será também avaliada a velocidade de transferência de dados entre zonas de memória diferentes.

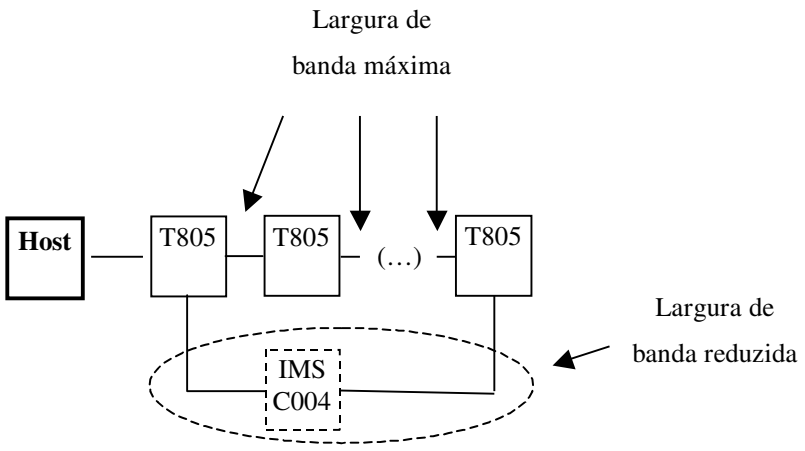


Fig. 2-13: Largura de banda dos portos de comunicação numa rede homogénea T805

Antes de apresentar as medidas de desempenho é conveniente tecer algumas considerações sobre a configuração de cada uma das topologias ensaiadas. No caso de redes homogéneas constituídas por T805s, baseadas em cartas mãe IMSB008 (Inmos Ltd, 1990c) e TMB16

(Transtech Paralell Systems Ltd., 1991b), usam um dispositivo electrónico chamado comutador de ligação - IMSC004 - para conectar dois portos de comunicação, em processadores diferentes. Pode-se assim recorrer a um aplicativo para configurar a rede instantaneamente e evitando-se configurações através de condutores. No entanto esta facilidade implica uma redução da largura de banda para valores próximos de 75 % do máximo (Inmos Ltd, 1990c), excepto para as conexões por defeito, estabelecidas através da própria carta mãe, sem recorrer ao referido comutador, e por isso inalteráveis como mostra a Fig. 2-13.

Além disso, se se pretender conectar duas cartas mãe, usando dois comutadores de ligação, a largura de banda será reduzida para aproximadamente 50 % do máximo. Assim, visto que as conexões por defeito configuram os processadores numa fila, sempre que se pretender ligar um T805, a pelo menos outros 2 T805, pelo menos uma das ligações terá de ser feita recorrendo ao comutador de ligação. Como a comunicação pelas várias ligações é efectuada concorrentemente, o tempo total de comunicação será sempre proporcional à taxa de comunicação da ligação mais lenta.

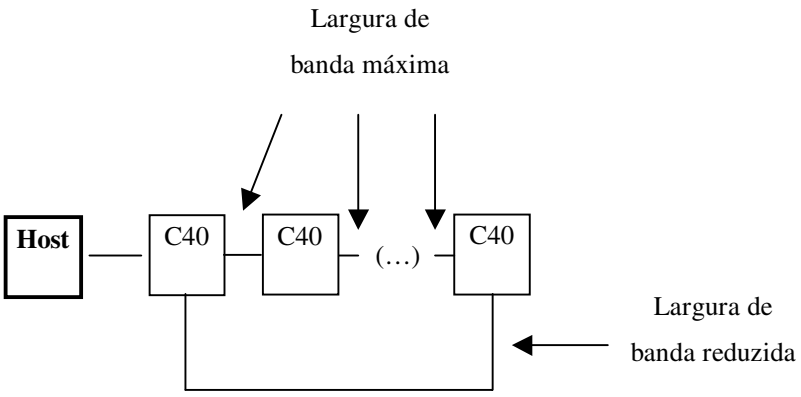


Fig. 2-14: Largura de banda dos portos de comunicação numa rede homogénea C40

Para redes homogéneas constituídas por C40s, foi observado que a largura de banda também é reduzida quando as ligações não são as por defeito, estabelecidas através da carta mãe. Neste caso, se for necessário recorrer a um cabo de ligação, como mostra a Fig. 2-14, a largura de banda será reduzida.

Além disso, visto que o C40 tem 6 portos de comunicação bidireccionais *half duplex*, para obter o máximo de largura de banda, em caso de comunicação bidireccional concorrente, é necessário ligar os processadores usando um par de comportas com sentidos de comunicação opostos, o que pode ser executado, no ambiente de desenvolvimento da 3L, recorrendo ao aplicativo *maplink*. Usando apenas uma comporta a largura de banda descerá para 50% do

máximo, indicado em 2.3.2, isto é 10 Mbytes por segundo, visto que numa primeira fase a informação será transmitida num sentido e só depois no outro.

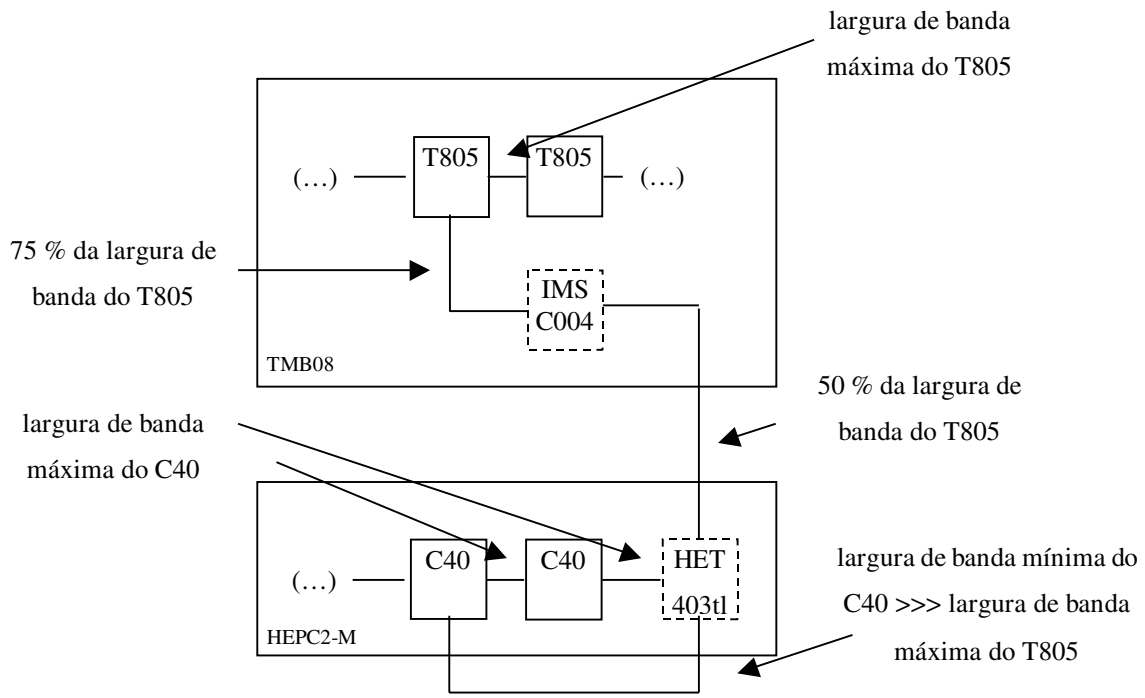


Fig. 2-15: Largura de banda dos portos de comunicação numa rede heterogénea T805 - C40

No caso das redes heterogéneas ensaiadas foi necessário usar um adaptador de ligação HET403tl, que recorre a adaptadores IMSC011, para conectar cada *link* de um T805 a duas comportas de um C40, em sentidos opostos. Como mostra a Fig. 2-15, está-se numa situação semelhante ao da ligação de duas cartas mãe através de dois IMSC004, já referida atrás, sendo assim a largura de banda reduzida para pouco menos de metade da largura de banda máxima do Transputer, isto aproximadamente 0,825 Mbytes por segundo. É conveniente notar que a largura de banda desta rede heterogénea é apenas dependente da largura de banda do T805, pois esta é muito inferior à taxa de transferência mínima das comportas do C40.

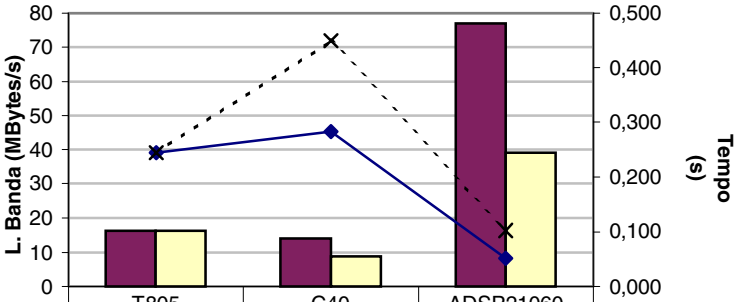
Neste tipo de redes heterogéneas, é ainda necessário considerar mais um factor que reduz a largura de banda. Quando se comunicam mensagens compostas por números de vírgula flutuante, visto o T805 usar uma representação interna deste tipo de dados, segundo o padrão IEEE, e o C40 usar uma representação própria da Texas Instruments Inc., é necessário operar uma conversão entre ambos os padrões. Isto implica que o C40 irá ter uma carga computacional adicional antes de enviar e depois de receber cada mensagem.

Quanto ao último caso de redes estudado, redes homogéneas constituídas por ADSP21060s, a largura de banda mantém-se a mesma independentemente do tipo físico da ligação. No entanto, para obter o máximo de desempenho, é necessário indicar ao processador que deve

transferir dados ao dobro da velocidade do relógio para cada porto, isto é dois *nibles* em cada ciclo, visto que o número de linhas de dados, bidireccionais *full duplex*, para cada *link* é de apenas 4. Para tal é necessário seleccionar os bits 12 a 17 do registo LCOM (*Link Common Control Register*), um dos registos de entrada saída do ADSP21060. Como este conjunto de registos está mapeado na memória interna do processador, é simples, a partir de uma linguagem de alto nível aceder-lhes, podendo-se evitar assim o uso de um segmento de código de máquina. Assim, basta colocar no endereço de memória $00C7_{(16)}$ o valor $0003F000_{(16)}$, de modo que a velocidade de transferência de dados, para todos os *links* do processador considerado seja máxima. Todos os processadores da rede, devem ser inicializados deste modo.

2.4.2.1 Transferências de dados num só processador

Neste ponto serão ensaiadas as taxas de comunicação entre bancos de memória diferentes num só processador. Para cada tipo de processador será indicado como obter o melhor desempenho e em que situação é que o desempenho é o pior, de modo a se estabelecer limites superior e inferior para a largura de banda.



	T805	C40	ADSP21060
L. Banda Máxima	16,360	14,134	76,923
L. Banda Mínima	16,360	8,908	39,060
Tempo mínimo	0,244	0,283	0,052
Tempo máximo	0,244	0,449	0,102

Fig. 2-16: Largura de banda unidireccional num só processador

Assim, abordar-se-á primeiro o caso das comunicações unidireccionais, isto é o envio de um vector de dados entre um processo produtor e um consumidor, alocados num único processador. Cada um destes processos usa um segmento para o código, um segmento para o *stack* ou pilha e um segmento para dados dinâmicos ou *heap*, onde se encontram armazenados os vectores a transferir. O resultado desta análise é mostrado na Fig. 2-16. Para o caso do T805, visto que a memória interna, embora de acesso mais rápido, é insignificante para

armazenar objectos de grandes dimensões, não foi considerada o armazenamento dos vectores nesta. Assim o desempenho máximo e o mínimo são idênticos e são obtidos alocando *heap* e código na memória externa e o segmento da pilha na memória interna.

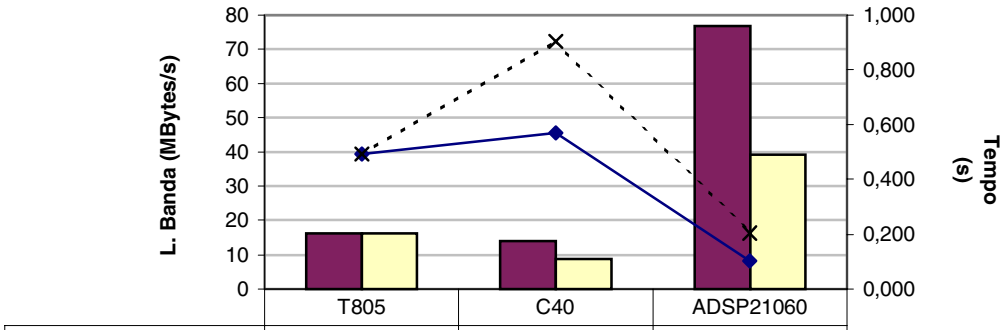
Para o caso do C40 o melhor desempenho é obtido alocando os segmentos de dados dinâmicos de ambos os processos no banco de memória externa Local ou um neste banco e o outro no banco de memória externa Global. O pior desempenho é obtido alocando ambos os segmentos na memória Global. Do mesmo modo que no Transputer, como os bancos de memória interna, de acesso mais rápido, são muito reduzidos, a pilha é alocada nestes para ambos os casos de desempenho. Também para ambos estes casos o código é armazenado na memória Local.

O ADSP21060, como já foi referido, tem o mesmo tempo de acesso ao banco de memória externa (SRAM), que a ambos os bancos de memória externa. Além disso, neste caso tanto a pilha como o *heap* estão alocados no mesmo segmento de dados, iniciando-se cada um num extremo e crescendo em sentidos opostos. Deste modo, a melhor estratégia de alocação não passa por armazenar dados num banco com acesso mais rápido, mas sim por distribuir os segmentos de dados e os segmentos de código por três bancos de memória diferentes, ou alocando ambos os segmentos de dados num banco e o código noutro. O pior desempenho obtém-se alocando no mesmo banco de memória todos os segmentos de código e dados. Uma situação intermédia, não indicada na figura seguinte, ocorre quando ambos os segmentos de dados estão alocados em bancos diferentes, mas os dois segmentos de código estão alocado num desses bancos, atingindo-se uma largura de banda de aproximadamente 52 Mbytes por segundo. Analisando a figura acima, e comparando o desempenho dos três tipos de processadores, o que apresenta um melhor desempenho é o ADSP21060, como era de esperar. O que surpreende é que o T805 consegue atingir um desempenho superior ao C40, provando que aquele processador, embora desactualizado, foi desenhado para ter um bom desempenho na transferência de dados. Por outro lado, era de esperar que o C40 atingisse uma largura de banda igual ou superior à taxa de transferência de dados máxima de cada porto de comunicação do próprio processador, isto é 20 Mbytes no caso do C40, visto tal suceder com o ADSP21060 e mais dramaticamente com o T805.

Os tempos indicados nas figuras são referentes a enviar 100 vezes consecutivas um vector com 10000 *floats* (32 bits), isto é 4 Mbytes.

A comunicação unidireccional é o caso mais simples de transferência de dados ensaiado. No entanto em processamento paralelo simétrico é comum que a comunicação seja bidireccional, pois admitindo que dois processos executam o mesmo conjunto de operações sobre conjuntos

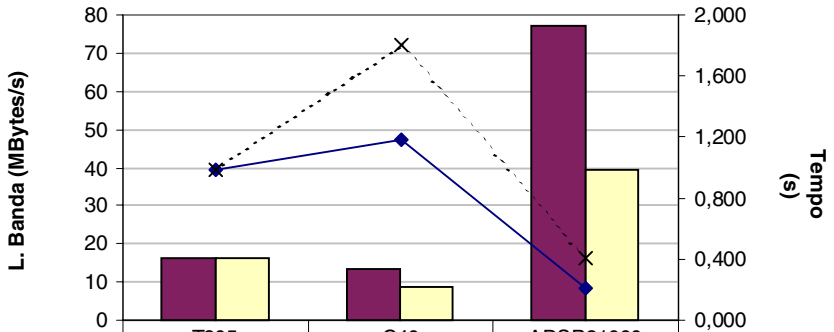
de dados diferentes, é razoável admitir que em algum ponto do algoritmo cada processo necessitará de aceder ao conjunto de dados dos outros, e disponibilizar o seu próprio conjunto de dados.



	T805	C40	ADSP21060
L. Banda Máxima	16,288	14,052	76,766
L. Banda Mínima	16,288	8,858	39,200
Tempo mínimo	0,491	0,569	0,104
Tempo máximo	0,491	0,903	0,204

Fig. 2-17: Largura de banda bidireccional num só processador

Enviando a mesma quantidade de dados duas vezes entre dois processos, ou seja 2 x 4 Mbytes, mas em sentidos opostos, o tempo de comunicação aumenta por um factor muito próximo de 2, mas a largura de banda é mantida, como mostra a figura acima.



	T805	C40	ADSP21060
L. Banda Máxima	16,260	13,500	76,880
L. Banda Mínima	16,260	8,864	39,196
Tempo mínimo	0,984	1,185	0,208
Tempo máximo	0,984	1,805	0,408

Fig. 2-18: Largura de banda bidireccional dupla num só processador

Poder-se-á ainda ter casos em que uma comunicação bidireccional é mantida entre um processo e vários outros. Se o comprimento dos vectores a comunicar for igual, o tempo de comunicação aumenta por um factor igual ao número de comunicações bidireccionais a estabelecer, embora se mantenha a largura de banda. Na Fig. 2-18 está representada uma

situação em que um processo mantêm duas comunicações bidireccionais com outros dois processos. Nesta situação, o tempo de comunicação aumenta por um factor de 4, em relação à comunicação unidireccional, e 2 em relação à situação de comunicação bidireccional simples apresentada na Fig. 2-17.

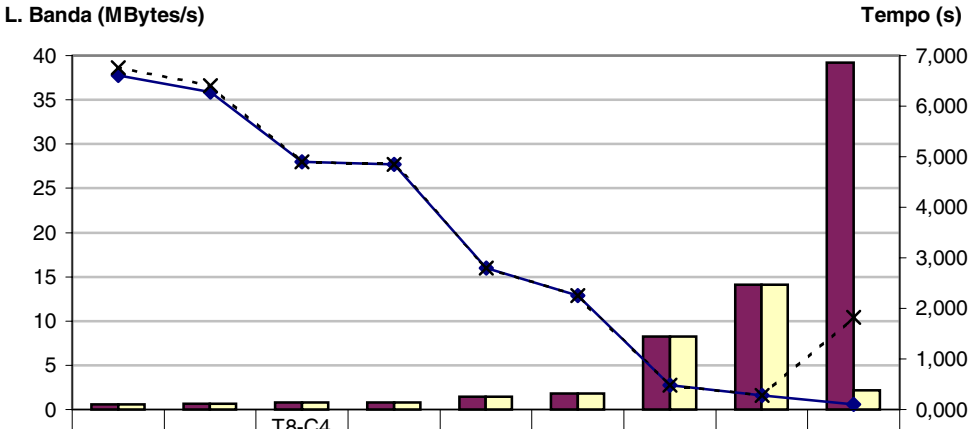
2.4.2.2 Transferências de dados entre vários processadores

Do mesmo modo que no ponto anterior, para cada uma das seguintes medidas de desempenho será indicado um limite inferior e superior, e como obtê-los. Serão, no entanto, neste ponto ensaiadas as taxas de comunicação entre bancos de memória diferentes e entre processadores diferentes.

Mais uma vez, o caso mais simples a considerar será a comunicação unidireccional, entre dois processadores, ou seja, o envio de um vector de dados de um processo alocado num processador para outro processo alocado noutro processador. Para o caso de redes homogéneas T805 e C40, os desempenhos máximos e mínimos são alcançados recorrendo às estratégias de alocação já referidas no ponto 2.4.2.1, o que implica que para o T805 o máximo e o mínimo são idênticos. Assim para o caso de redes heterogéneas T805 - C40 os desempenhos máximos e mínimos só dependerão da estratégia de alocação no C40. Quanto às redes homogéneas ADSP21060, o desempenho mínimo só é atingido, se pelo menos o segmento de dados de um processador for alocado na memória externa.

No entanto no caso de redes T805 o desempenho será reduzido sempre que seja necessário recorrer ao comutador de ligação. Deste modo são considerados dois casos para esta rede, indicados nas figuras como T805 (fila) e T805 (C004), respectivamente para o caso de comunicação através da carta mãe ou recorrendo ao comutador de ligação IMS C004. Do mesmo modo, como também já atrás foi referido, a taxa de comunicação em redes C40 será reduzida quando se necessitar de ligar dois processadores através de um cabo de ligação, situação esta referida como C40 (cabo) nas figuras. Do mesmo modo a taxa de transferência de ligações que não recorrem ao cabo será identificada como C40 (fila). Assim no caso das redes heterogéneas, estes factores também irão afectar a largura de banda, como é indicado na Fig. 2-15. Visto que no caso da carta TMB08, será necessário recorrer ao comutador de ligação, a única diferença será observável na carta HEPC2-M, entre o adaptador de ligação HET 403tl e os C40s, ou seja, via cabo de ligação ou através das ligações por defeito da carta HEPC2-M, respectivamente identificados como T8-C4 (II) (cabo) e T8-C4 (II) (fila). Neste tipo de redes, há ainda a considerar a redução de largura de banda, introduzida pela conversão entre as representações internas dos números de vírgula flutuante de ambos os tipos de

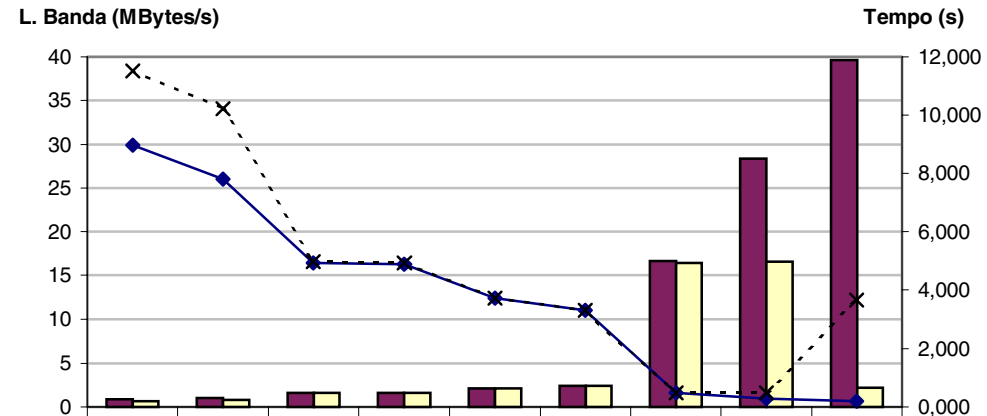
processador, identificados como T8-C4 (I) (cabo) e T8-C4 (I) (fila), as quais são efectuadas no C40.



	T8-C4 (I) (cabo)	T8-C4 (I) (fila)	T8-C4 (II) (cabo)	T8-C4 (II) (fila)	T805 (C004)	T805 (fila)	C40 (cabo)	C40 (fila)	ADSP 21060
L. Banda Máxima	0,605	0,637	0,817	0,825	1,428	1,775	8,281	14,134	39,216
L. Banda Mínima	0,592	0,625	0,817	0,825	1,428	1,775	8,281	14,134	2,189
Tempo mínimo	6,612	6,279	4,896	4,848	2,801	2,254	0,483	0,283	0,102
Tempo máximo	6,757	6,400	4,896	4,848	2,801	2,254	0,483	0,283	1,827

Fig. 2-19: Largura de banda unidireccional entre dois processadores

Observando a Fig. 2-19, verifica-se que sempre que a comunicação não pode ser feita recorrendo às ligações por defeito das respectivas cartas mãe, a largura de banda é reduzida. A largura de banda mínima observada refere-se a redes heterogéneas T8-C4, incluindo a conversão de dados. Pouco aumenta a largura de banda para estas mesmas redes, quando não se considera a conversão de dados, a qual é aproximadamente metade da largura de banda de redes T805, como indicam as especificações técnicas (Inmos Ltd., 1990c). Apenas se verifica um aumento consideravel da largura de banda para redes C40, embora 14,134 MBytes/s, seja abaixo da taxa esperada, isto é 20 MBytes/s. Curiosamente este valor é idêntico ao valor máximo medido para transferências de informação num só C40, apresentado no ponto 2.4.2.1. Finalmente o ADSP21060 apresenta a maior largura de banda para o caso de melhor desempenho, no entanto para o caso de pior desempenho esta é reduzida para um valor pouco maior do que numa rede T805, 2,189 MBytes/s. Tal não é o caso das restantes redes em que a diferença entre o máximo e o mínimo desempenho não existe ou não é significativa. Convém referir que para o caso de REDES T8-C4 (II) a diferença também não é significativa, quer a ligação entre o adaptador HET 403tl e o C40, seja feita através da carta HEPC2-M ou através de um cabo de ligação.



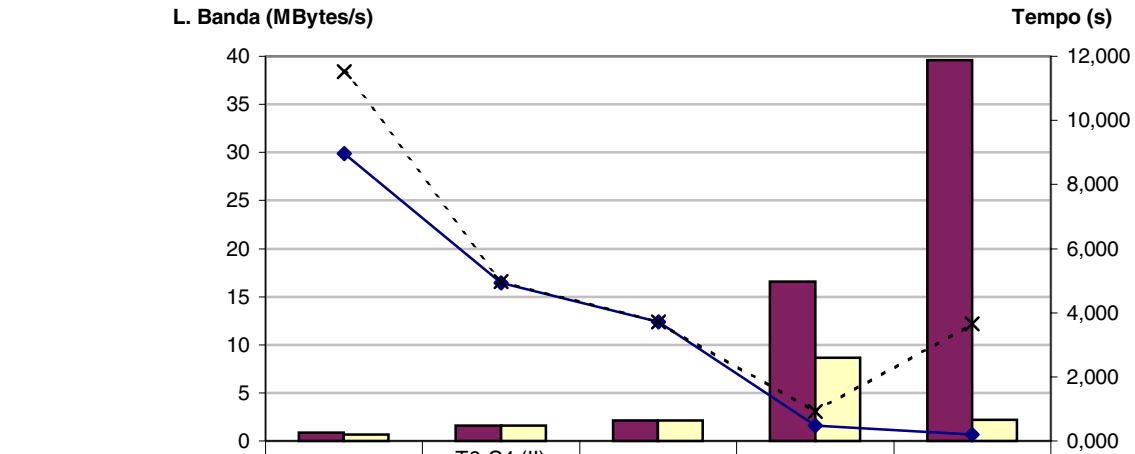
	T8-C4 (I) (cabo)	T8-C4 (I) (fila)	T8-C4 (II) (cabo)	T8-C4 (II) (fila)	T805 (C004)	T805 (fila)	C40 (cabo)	C40 (fila)	ADSP 21060
L. Banda Máxima	0,892	1,026	1,622	1,638	2,158	2,420	16,606	28,338	39,600
L. Banda Mínima	0,694	0,782	1,612	1,628	2,158	2,420	16,402	16,580	2,188
Tempo mínimo	8,969	7,797	4,932	4,884	3,707	3,306	0,482	0,282	0,202
Tempo máximo	11,527	10,230	4,963	4,914	3,707	3,306	0,488	0,483	3,656

Fig. 2-20: Largura de banda bidireccional *full-duplex* entre dois processadores

Numa investigação independente (Tokhi et al, 1995) foram observadas taxas de transferência de dados superiores, para redes C40 e T8-C4 (II), respectivamente 17,7 MBytes/s e 1 MByte/s. Este último caso poderá ser explicado, se em vez de uma TMB08 for usada uma carta mãe para os Transputers tipo TMB04. Nesta arquitectura não se pode recorrer a um comutador de ligação, visto que esta última carta não o inclui, o que, embora impeça que a alteração da configuração das ligações possa ser feita a "quente", tornando por isso esse processo mais moroso e incómodo, permite uma taxa de transferência de dados superior. Quanto ao outro caso, redes C40, não foi possível reunir dados para explicar a diferença de 25% na largura de banda do C40. Poderá depender do *hardware* utilizado em (Tokhi et al, 1995), mas não foi possível aceder a essa informação.

Comparando a Fig. 2-19 com a figura Fig. 2-20, pode-se observar que, exceptuando redes homogéneas ADSP21060, a largura de banda aumenta para o caso da comunicação bidireccional *full-duplex* em relação à comunicação unidireccional. Tal é esperado em todas as topologias que incluam Transputers, pois esta é uma característica dos portos de comunicação destes dispositivos, os quais só assim atingem a máxima taxa de transferência de 20 Mbits/s. No caso de redes homogéneas C40 (fila), a taxa de transferência dobra, sendo no entanto usados dois portos de comunicação em cada processador, logo a taxa de comunicação por cada porto mantém-se aproximadamente nos 14 Mbytes/s. No caso da rede ADSP21060, o tempo de comunicação dobra, visto que mais uma vez, os tempos indicados nas figuras são

referentes a enviar 4 Mbytes em cada sentido, o que implica que o dobro dos dados são comunicados por cada porto. Assim, a largura de banda deste dispositivo mantém-se muito próxima de 40 Mbytes/s por porto de comunicação.



	T8-C4 (I) (cabo)	T8-C4 (II) (cabo)	T805 (C004)	C40 (cabo)	ADSP 21060
L. Banda Máxima	0,892	1,622	2,156	16,586	39,600
L. Banda Mínima	0,694	1,612	2,156	8,648	2,188
Tempo mínimo	8,969	4,932	3,711	0,482	0,202
Tempo máximo	11,527	4,963	3,711	0,925	3,656

Fig. 2-21: Largura de banda bidireccional *full-duplex* por ligação entre três processadores

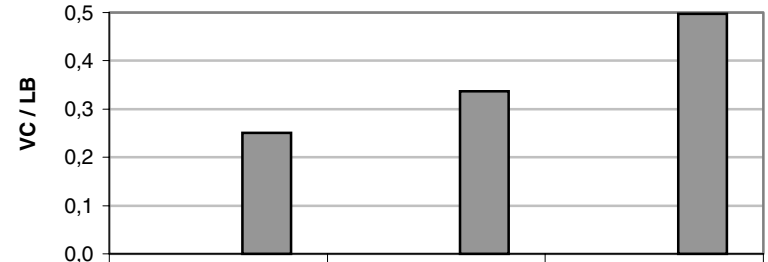
Pode-se ainda verificar que para redes T8-C4 (I) e C40 (fila), a diferença entre o desempenho máximo e mínimo já é considerável. No caso de redes T8-C4 (I) está-se a considerar agora duas conversões de números de virgula flutuante em cada C40, visto que a comunicação é bidireccional. No caso anterior apenas se considerava a conversão da representação IEEE para a representação interna do C40, que é mais rápida que a conversão inversa. Como estas operações de conversão não podem ser efectuadas concorrentemente, o seu tempo total será superior ao dobro do caso de transferência de dados unidireccional.

A figura acima resume as observações das velocidades de transferência de dados em redes com três nós, em que cada nó mantém duas comunicações bidireccionais *full-duplex* com outros dois. Neste caso é considerado que para as arquitecturas baseadas em T805s e C40s, pelo menos uma das ligações terá taxa de transferência reduzida, o que implica que, como ambas as ligações em cada nó são concorrentes o tempo de comunicação será o pior.

Esta figura, se bem que respeitante a medições independentes, representa o subconjunto da figura anterior, referente aos piores casos em termos da ligação física entre processadores. Convém ainda indicar que as taxas apresentadas referem-se apenas a uma ligação entre

processadores, isto é dois portos em sentidos opostos no caso do C40, e um porto de comunicação nos restantes casos. Como na realidade se estão a considerar duas ligações por processador, para medir a largura de banda total, os valores indicados na figura terão se ser duplicados. Isto indica que o ADSP21060 mantém uma largura de banda total muito próxima de 240 MBytes/s, distribuída por seis portos de comunicação, ou seja o pico de desempenho. No caso do T805 e do C40, a largura de banda sustentada será 8,624 MBytes/s, distribuídos por quatro portos de comunicação, e 49,758 MBytes/s, distribuídos por três pares de comportas bidireccionais *full-duplex,* respectivamente. Muito próximo do pico de desempenho de 9,6 MBytes/s mantém-se o T805, o que não sucede com o C40, cujo valor de pico anunciado é de 120 Mbytes/s.

2.4.2.3 Relação entre a largura de banda e a velocidade de processamento



	T805	ADSP21060	C40
VC	0,540	13,330	8,260
LB	2,156	39,600	16,586
VC / LB	0,250	0,337	0,498

Fig. 2-22: Comparação do desempenho dos vários processadores usados num ambiente de processamento paralelo

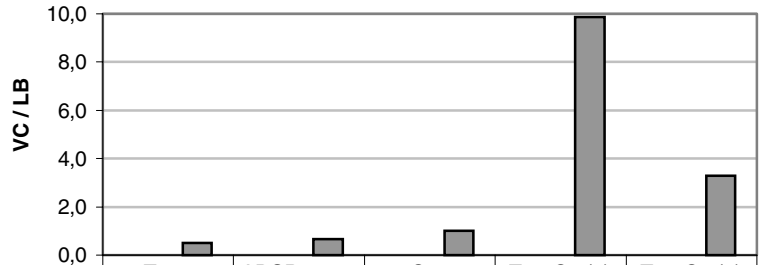
Uma medida de desempenho que permite comparar o desempenho relativo de processadores de diferentes famílias, quando integrados em redes paralelas, é dada por:

$$\frac{VC}{LB} \quad (2\text{-}2)$$

Onde a velocidade de cálculo, VC, é o valor obtido no ponto 2.4.1, expresso em MFlops e a largura de banda, em MBytes/s, LB, refere-se aos valores obtidos no ponto anterior. Deste modo, para um dado tipo de processador, quanto menor for a relação (2-1), menor será o tempo perdido em comunicações de dados entre processadores, relativamente ao

processamento desses mesmos dados, isto é, maior será o desempenho ou eficiência de uma rede constituída por aquele tipo de processador.

Para os processadores usados neste trabalho, a Fig. 2-22 mostra a relação entre a velocidade de cálculo e as taxas de transferências de dados internas. Já considerando a velocidade de cálculo na memória interna para dois processadores, que é o dobro da apresentada na Fig. 2-10, e a largura de banda bidireccional por porto de comunicação indicada na Fig. 2-21, tem-se o espectro observado na Fig. 2-23. Como se pode observar, para redes homogéneas, a maior eficiência será atingida por redes compostas de T805s e a pior por redes compostas de C40s, estando a eficiência de redes ADSP21060 próxima do desempenho da primeira.



	T805	ADSP21060	C40	T8 - C4 (1)	T8 - C4 (3)
VC	1,080	26,660	16,520	8,8	8,8
LB	2,156	39,600	16,586	0,892	2,676
VC / LB	0,501	0,673	0,996	9,865	3,288

Fig. 2-23: Comparação do desempenho de redes de dois processadores.

Se se considerar uma rede heterogénea a relação (2-2) será generalizada para:

$$(2\text{-}3) \quad \frac{\sum_i \mathrm{VC}_i}{\sum_j \mathrm{LB}_j}$$

Onde o numerador representa a soma da velocidade de cálculo dos *i* processadores presentes na rede e o denominador a soma da largura de banda das *j* ligações existentes entre esses processadores. Convém referir que esta relação indica qual das topologias comparadas terá a maior eficiência, mas não qual o valor dessa eficiência, visto que esta depende do algoritmo.

O caso mais simples de rede é composto por dois processadores interligados por uma ligação. A figura acima mostra o desempenho destas topologias, e compara-as com uma rede constituída por um T805 e um C40, ligados por três portos, de modo a triplicar a largura de banda, identificada como T8 – C4 (3). Como se pode observar, os desempenhos relativos mantêm-se para o caso das 3 redes homogéneas. Por outro lado, a reduzida largura de banda

da rede heterogénea mais simples, T8 - C4 (1), onde a comunicação entre os processadores é efectuada apenas através de 1 porto, faz com que a sua eficiência seja muito reduzida quando comparada com as redes homogéneas. Mesmo maximizando a largura de banda, para este tipo de redes, T8 - C4 (3), a relação (2-3), continua a ser muito elevada. Estes resultados indicam que não se poderá esperar um bom desempenho deste tipo de rede heterogénea, para algoritmos com uma forte componente de transferência de dados.

2.4.2.4 Modelo de comunicação e custos adicionais à transferência de dados

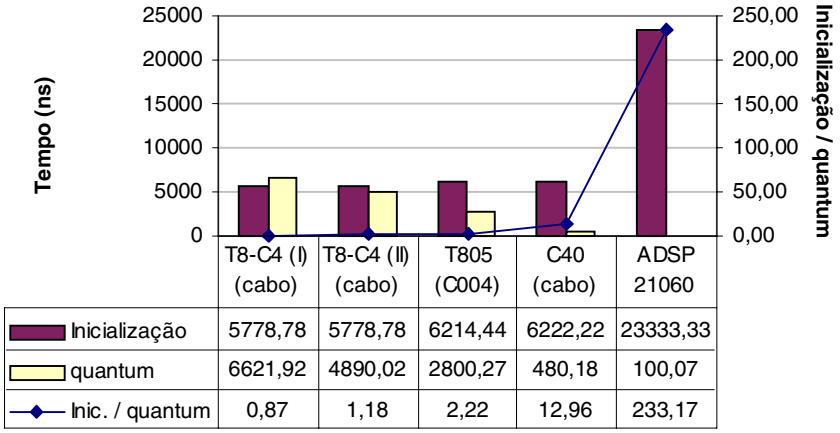
Nos pontos anteriores foram consideradas transferências de vectores de dados muito grandes, logo foram desprezado os tempos de inicialização dos canais de comunicação; no entanto ao longo desta tese, o modelo linear (Fraigniaud e Lazard, 1994) será usado para estimar a complexidade das comunicações. Neste modelo, o tempo (T) para enviar uma mensagem com n *bytes* de comprimento, de um processador para outro, é dado pela seguinte expressão:

$$T = \beta + n\,\tau \tag{2-4}$$

onde β é o tempo de inicialização do canal de comunicação. Esta parcela depende não só do *hardware*, como por exemplo o armazenamento temporário dos dados que chegam ou saiem de um porto num *buffer*, mas também do *software* usado para implementar a comunicação, o que pode incluir o incremento de um contador e algumas comparação para codificar ou descodificar o cabeçalho de uma mensagem; τ é o *quantum* de comunicação, isto é o tempo para comunicar um *byte* com outro processador e n o comprimento do vector de dados a transferir em *bytes* (neste ponto será conveniente referir que no âmbito do SPAM a partícula de informação é uma palavra com o comprimento de 32 bits, quer sejam números de vírgula flutuante ou inteiros). Utilizando o modelo linear pretende-se ter uma aproximação razoável da comunicação numa rede de processadores, de modo a ensaiar no papel algumas estratégias de comunicação com vista ao desenvolvimento de eficientes mecanismos de comunicação, assunto que será abordado no capítulo 4.

Poder-se-ia ter utilizado outros modelos mais detalhados, como o LogP (Culler et al., 1993), que supõe que existe ainda uma latência na comunicação, parcela esta que deverá ser somada a β. Já no modelo BSP, *Bulk Synchronous Parallelism* (Valiant, 1990), o tempo de comunicação é obtido não a partir do *quantum* de comunicação mas sim a partir de uma expressão semelhante a (2-3), tomando em conta a relação entre a largura de banda e a velocidade de processamento da rede. O modelo postal, *postal model* (Bar-Noy e Kipnis,

1994; Bar-Noy et al., 1998) no original, toma em consideração um atraso no envio de uma mensagem, o qual resulta de conflitos no envio de mais que uma mensagem pelo mesmo porto. Como estas colisões são dependentes do algoritmo e da configuração da rede não é possível estimar com exactidão o tempo de atraso para o caso geral. Assim, ensaiar estratégias de comunicação neste modelos é muito mais complexo que no modelo linear, e como as estratégias a ensaiar são bastante diferentes não sendo assim necessário tanta precisão, será utilizado o modelo linear, sendo todos os atrasos na comunicação representados também por β.



	T8-C4 (I) (cabo)	T8-C4 (II) (cabo)	T805 (C004)	C40 (cabo)	ADSP 21060
Inicialização	5778,78	5778,78	6214,44	6222,22	23333,33
quantum	6621,92	4890,02	2800,27	480,18	100,07
Inic. / quantum	0,87	1,18	2,22	12,96	233,17

Fig. 2-24: Comparação dos *quanta* de comunicação externos unidireccionais

Na figura anterior estão representadas ambas as parcelas para as arquitecturas utilizadas neste trabalho, no caso de ligações *full-duplex* considerados na Fig. 2-21. Pode-se observar que para o caso do ADSP21060 e do C40 a relação entre o tempo de inicialização e o *quantum* é bastante significativa, aproximadamente 13 e 233 respectivamente, o que implica uma redução drástica de desempenho para transferências de pequenos vectores de dados.

2.4.3 Processamento e transferência de dados concorrentes

Tendo sido ensaiados, separadamente, o desempenho em termos de velocidade de cálculo e transferência de informação, neste ponto será analisado como os três tipos de processadores ensaiados se comportam numa situação em que, concorrentemente, transferem dados com outros processadores e desenvolvem cálculo de vírgula flutuante.

Para cada processador foram medidas quatro situações em que o tempo envolvido no cálculo de vírgula flutuante é idêntico ao tempo despendido nas transferências de dados. Foram somados estes tempos, os quais são identificados nas figuras com a etiqueta *soma*. Os tempos que foram medidos para as quatro situações são apresentados pela curva identificada nas

figuras como *real*. Finalmente a relação entre os tempos somados e os tempos medidos é apresentado pela curva *soma/real*.

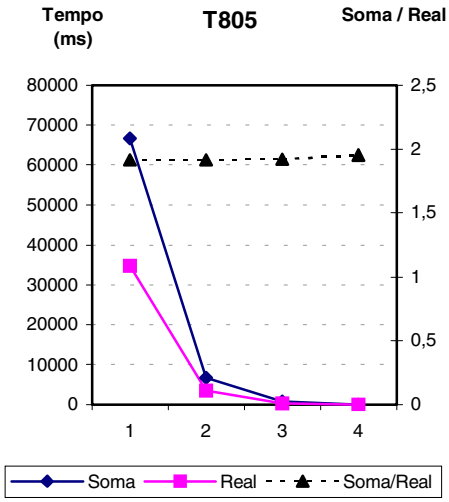


Fig. 2-25: Desempenho do T805 em caso de cálculo e comunicação concorrentes

Pela figura acima pode-se observar que o T805 praticamente consegue transmitir dados sem que haja perda de desempenho no cálculo de vírgula flutuante, pois os tempos medidos concorrentemente são aproximadamente metade da soma dos tempos separados.

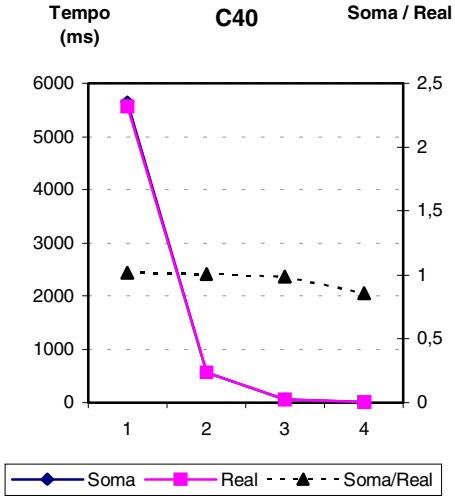


Fig. 2-26: Desempenho do C40 em caso de cálculo e comunicação concorrentes

Tal não sucede no entanto para o caso do C40, figura anterior, nem do ADSP21060, figura seguinte, onde o tempo medido concorrentemente é aproximadamente igual à soma dos tempos, indicando que estes processadores não calculam e comunicam concorrentemente. Mas como já foi visto no ponto 2.3, ambos estes processadores dispõe de dispositivos de

acesso directo à memória, designados nos diagramas por DMA, que permitem transferências de dados, quer entre os bancos de memória, quer com outros processadores, sem que haja degradação do desempenho da unidade de vírgula flutuante. É se assim forçado a concluir que as primitivas de comunicação implementadas pelo compilador utilizado, para ambos os casos, não aproveitam esta facilidade.

No entanto a documentação do compilador 3L para ADSP21060 (3L, 1998) refere que a primitiva de comunicação utiliza implicitamente o co-processador de DMA, sempre que os dados se encontram na memória interna, como foi o caso deste ensaio. Realmente verificou-se uma degradação do desempenho sempre que os dados a transferir para outro processador se encontram armazenados na memória externa. Mas neste ensaio, visto que os dados estão armazenados na memória interna, e segundo (Analog Devices, 1997) as transferências de dados deveriam ocorrer simultaneamente sem custo no desempenho do processador, o que sugere que a estratégia de DMA utilizada pela 3L poderá ser melhorada.

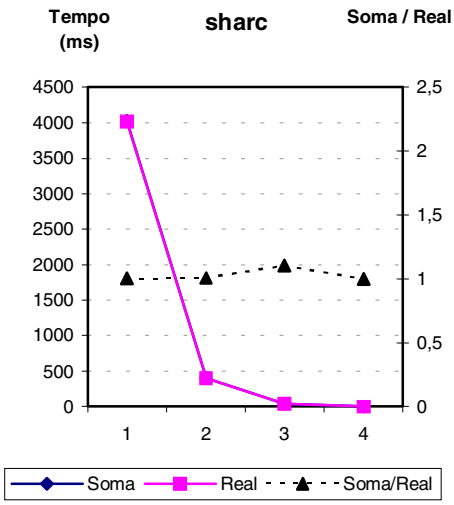


Fig. 2-27: Desempenho do Sharc em caso de cálculo e comunicação concorrentes

O modelo BSP pode ser usado para prever o custo temporal de um algoritmo mapeado sobre uma rede paralela, onde a computação local ocorre simultaneamente com transferencias de dados entre nós, como sucede no T805. De acordo com este modelo o custo em cada nó, será dado pelo maior de entre os custos de comunicação e computação. O custo total do algoritmo paralelo é assim dado pelo máximo dos custos de cada nó mais um custo referente à sincronização dos nós, sempre que necessário, após cada ronda de comunicações. Tal será necessário se um passo computacional for dependente de dados que devem ser distribuídos numa ronda de comunicação prévia, de modo a garantir que a computação não se inicia antes

de terminada a distribuição. Esta sincronização será necessária em qualquer rede, mesmo que seja homogénea e por melhor equilibrada que seja a distribuição de carga.

O modelo linear pode ser generalizado para tomar em conta também a computação. No caso de *hardware* que permite que esta seja simultânea com a comunicação o custo local será dado pelo maior dos custos entre a comunicação e a computação, ao passo que se tal não for possível será dado pela soma de ambos os custos. Do mesmo modo que para o modelo BSP, o custo total será o máximo dos custos totais. Quanto à sincronização, esta será feita por *software*, como prevê o modelo LogP. Se esta sincronização for feita recorrendo ao envio de uma mensagem atómica de cada nó para todos os outros, os quais devem aguardar pela recepção de todas essa mensagens, o custo da sincronização é um custo de comunicação já estando embebido no modelo.

2.5 Resumo

Neste capítulo foram apresentados os paradigmas de programação mais comuns de modo a integrar o SPAM num contexto que estabelece uma ponte entre dois paradigmas: o paradigma imperativo e o paradigma concorrente. Foram abordadas arquitecturas básicas usadas em processamento paralelo bem como técnicas de particionamento de algoritmos de modo a introduzir o modelo de processamento paralelo adoptado. Finalmente foram apresentadas as características de cada processador utilizado neste trabalho e avaliado o seu desempenho, quer em termos de velocidade de cálculo de vírgula flutuante como em termos de taxas de transferência de dados. Foi também comparado o desempenho de cada um destes processadores quando integrados em ambientes de processamento paralelo, e foi feita uma comparação do desempenho de redes constituídas pelos processadores ensaiados. Estes ensaios servirão como referência para a implementação das operações de cálculo matricial e no desenho dos mecanismos de encaminhamento de mensagens descritos nos capítulos seguintes.

3 Modelo de programação e sua implementação

Neste capítulo é descrito o modelo de programação proposto, bem como a sua implementação em diferentes níveis de abstracção. Este modelo é comparado com o modelo de programação sequencial e o modelo de programação paralela, de modo a especificar como o SPAM estabelece uma ponte, transparente para o programador de aplicações, entre estes dois modelos.

3.1 Descrição do modelo de programação

Antes de descrever o modelo de programação adoptado pelo SPAM, é conveniente identificar em que etapas difere a implementação de um algoritmo paralelo, da implementação de um algoritmo sequencial.

3.1.1 Implementação de um algoritmo paralelo

Segundo (Sarkar, 1989) existem três etapas na transformação de um algoritmo sequencial num algoritmo paralelo:

- Identificação do paralelismo no algoritmo.
- Particionamento do algoritmo em tarefas realizáveis concorrentemente.
- Atribuição dessas tarefas aos processadores disponíveis.

Convencionalmente, a identificação do paralelismo é função do programador. Depende pois da sua capacidade de visualização, e da filosofia a seguir para paralelizar o algoritmo. Na etapa seguinte o algoritmo deverá ser particionado, de acordo com essa filosofia, de modo que as tarefas resultantes possam ser executadas concorrentemente. Finalmente, as partições deverão ser atribuídas aos processadores disponíveis.

3.1.2 Geração automática de algoritmos matriciais paralelos

O modelo de programação, descrito pelo fluxograma da Fig. 3-1, representa o processo de geração automático de uma aplicação paralela. A etapa de identificação do paralelismo não faz parte deste modelo, porque é assumido nesta tese, para o caso de algoritmos matriciais, que o volume de dados a processar é elevado, sendo assim o particionamento efectuado ao nível destes dados, segundo a filosofia apresentada no ponto seguinte. Como se pode observar no fluxograma, o programador de aplicações apenas necessita de introduzir o algoritmo matricial no editor, na forma de código sequencial, e descrever através de um diagrama de blocos a configuração da rede, no configurador visual. Seguidamente todo o processo de geração da aplicação paralela é automático.

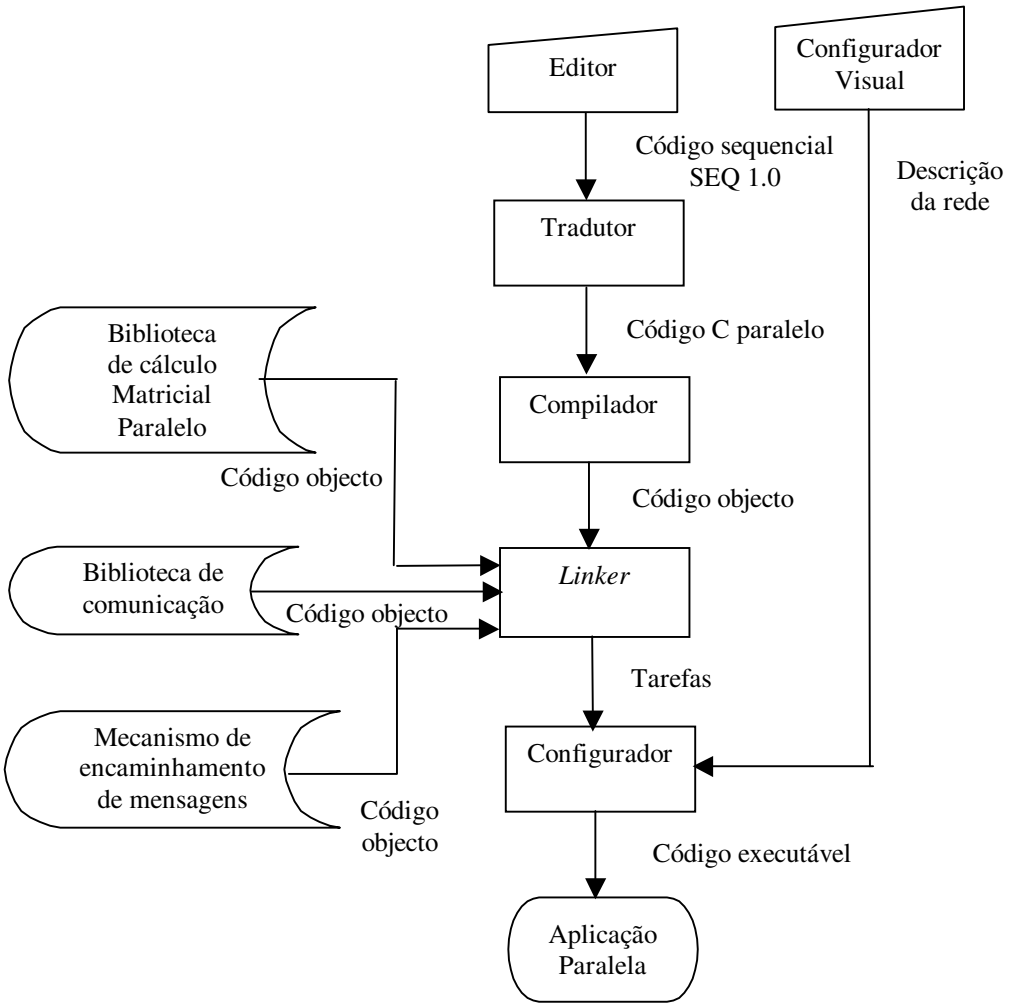


Fig. 3-1: Modelo de programação.

O processo inicia-se no tradutor, onde o código sequencial SEQ 1.0 é particionado e convertido em código fonte C paralelo. Este código é enviado para um compilador adequado aos processadores alvos e ligado com uma biblioteca de cálculo matricial paralelo, com uma biblioteca de comunicação e com o mecanismo de encaminhamento de mensagens, obtendo-

se assim processos ou tarefas. Finalmente cada tarefa é configurada, em termos de ambiente de comunicação, de acordo com a descrição da rede paralela alvo, sendo assim gerada a aplicação paralela.

Este modelo de programação facilita a investigação do desempenho do algoritmo sobre diversas topologias de rede, pois basta especificar no configurador visual uma qualquer topologia e ordenar ao sistema que invoque o configurador, sendo a aplicação reconfigurada sem haver necessidade de alterar uma única linha de código. Além disso a sua estrutura modelar permite que sejam adicionadas novas funções de cálculo matricial, ou que as bibliotecas de cálculo matricial e de comunicações sejam completamente substituídas, bem como possa ser utilizada uma diferente estratégia de encaminhamento de mensagens.

É conveniente salientar que o compilador, o *linker*, e o configurador são ferramentas existentes no mercado para os processadores alvo e são portanto externas ao sistema de paralelização automático.

3.2 Particionamento dos operandos matriciais

Dependendo da operação matricial a executar, como será referido no capítulo 5, os operandos devem estar completamente armazenados em todos os processadores da rede paralela ou distribuídos por estes. Na Fig. 3-2 está esquematizado a estratégia de particionamento dos operandos por *n* processadores.

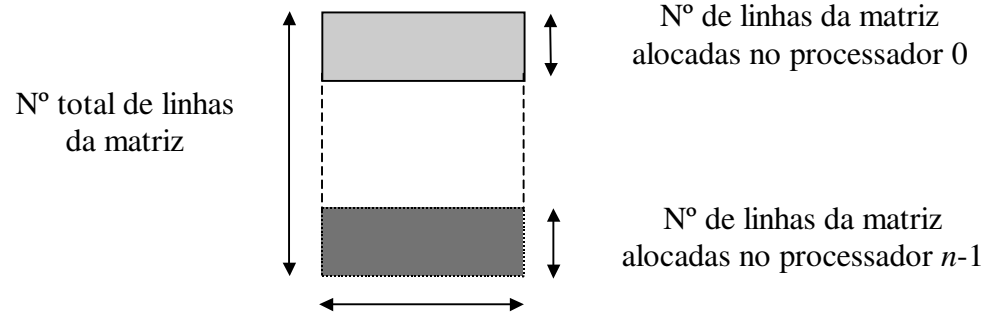


Fig. 3-2: Distribuição de uma matriz por *n* processadores

Como se pode observar, a distribuição é feita alocando conjuntos de linhas consecutivas da matriz em cada processador. O volume de dados armazenado em cada processador, não tem de ser necessariamente idêntico, mas sim dependente das características de processamento de cada processador.

3.3 Estrutura das aplicações geradas automaticamente pelo SPAM

No diagrama da Fig. 3-3 está representada a estrutura da aplicação em termos de níveis de abstracção. O nível inferior consiste na rede de elementos de processamento, a qual pode ser construída com qualquer tipo de processadores suportados. Esta rede tem pelo menos um processador, o processador raiz, que está fisicamente conectado ao computador anfitrião. No caso de uma rede com apenas um processador, o modelo de processamento degenera em processamento sequencial.

Directamente em contacto com a rede de processadores tem-se o micro-núcleo fornecido pelos compiladores da 3L usados. Este disponibiliza funções de gestão de processos - criação, eliminação e escalonamento – de gestão de memória e de comunicação.

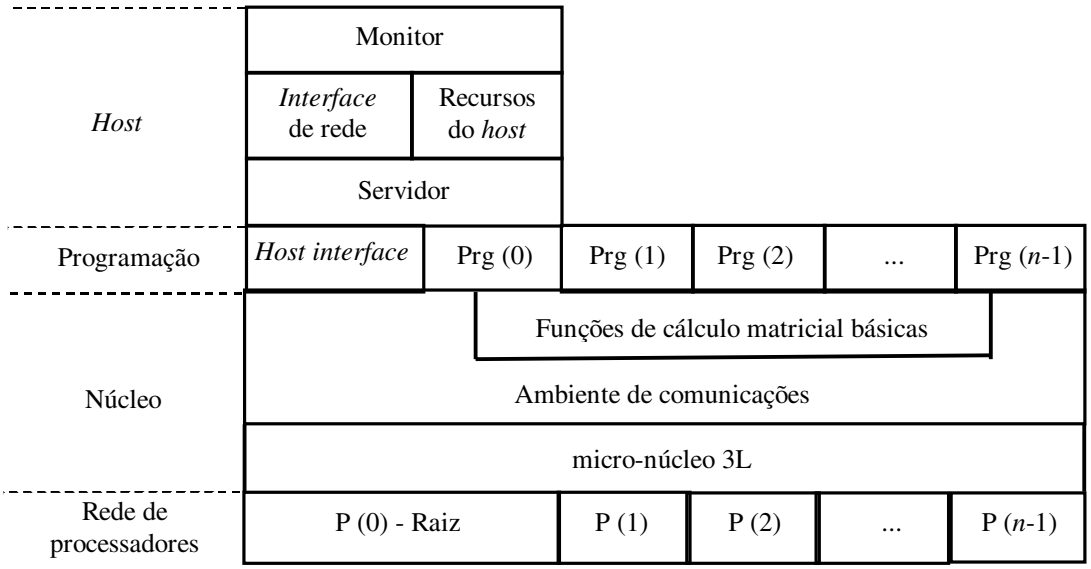


Fig. 3-3: Estrutura em níveis de abstracção de uma aplicação gerada pelo SPAM.

Sobre o micro-núcleo aloja-se o ambiente de comunicações e as funções de cálculo matricial básicas. No entanto estes dois últimos níveis podem ultrapassar o micro-núcleo e aceder directamente ao *hardware* - embora tal não seja comum - sendo apenas utilizado para tirar partido de facilidades especificas de cada tipo de processador. Todos estes 3 níveis formam o núcleo da aplicação. Uma instância deste está alocada em cada processador da rede. O núcleo é assim responsável pelo *interface* entre o nível de programa e uma qualquer rede de processadores. Deste modo, para transferir dados entre processadores, ao nível de programação, são usadas duas primitivas, send e receive, que empacotam os dados e alguma informação adicional, necessária para que seja estabelecida a comunicação, utilizando o protocolo reconhecido pelo núcleo. Este é o responsável pelo controlo do processo de comunicação, deixando o nível de programação livre para continuar a processar.

O nível seguinte é o nível de programação. Este nível é composto pelos processos de cálculo de alto nível, Prg (0) ... Prg (*n*-1), gerados automaticamente pelo tradutor ou desenvolvidos pelo programador. Estes processos são idênticos em termos de código, operando sobre diferentes conjuntos de dados. Além destes processos este nível é ainda composto pelo *Host interface*. Este processo, situado no nível de programação, pode ser visto como uma extensão do núcleo, no processador raiz, de modo que este e consequentemente toda a rede, possam aceder aos recursos do computador *Host*, tornando-os disponíveis para a aplicação. Estes dois níveis serão aprofundados em pontos posteriores.

Para terminar a descrição da estrutura da aplicação, falta referir o nível superior, o qual reside no computador *Host* e por isso tem esse nome. Este nível pode ser visto como um conjunto de três subníveis. O subnível inferior consiste num programa servidor, incluído no pacote do compilador que gera código para o processador raiz. É a este servidor que o *Host interface* irá efectuar pedidos de acesso aos recursos do *Host*. No subnível seguinte, temos no que respeita a *hardware* os recursos do *Host* e no que respeita a *software* o *interface* de rede. Este pode ser visto como um canal directo entre o subnível superior e o processador raiz, de modo que um programa monitor possa comunicar com a rede de processadores, lançando aplicações sobre a rede, estabelecendo uma via de comunicação com essas aplicações e medindo os tempos de processamento de cada processador, bem como a eficiência total da aplicação. Esta estrutura é suficientemente flexível para que o monitor possa ser substituído ou coexistir com um *interface* com o exterior, de modo que a aplicação possa controlar um dispositivo externo.

Embora a aplicação gerada seja direccionada para uma arquitectura de passagem de mensagens, a estrutura da aplicação em níveis de abstracção permite que esta possa ser gerada para qualquer arquitectura paralela, constituída por quaisquer tipos de processadores, memória partilhada incluída, desde que o ambiente de comunicações seja redesenhado de acordo com essa arquitectura e suporte uma arquitectura de passagem de mensagem virtual ao nível de programação.

3.3.1 Núcleo

Na Fig. 3-4 está representada esquematicamente a hierarquia do núcleo. Este pode ser dividido em três secções: o ambiente de comunicação, as funções de cálculo matricial básicas e o *interface* com o nível de programação, o qual permite, que a esse nível, as rotinas implementadas no núcleo possam ser acedidas.

O ambiente de comunicações, como já foi referido, liberta o nível de programação do encaminhamento de mensagens, deixando-o livre para computar. A comunicação entre este

nível e o núcleo, no seu modelo mais simples, recorre a duas primitivas: enviar e receber, também referidas como send e receive, e num modelo de mais alto nível a sub-programas de comunicação. Estes, constituídos essencialmente por chamadas ordenadas às primitivas de comunicação, permitem implementar conceitos de difusão de mensagens mais abstractos, tais como distribuir uma matriz, ou uma secção de uma matriz, pela rede de processadores. Estes conceitos abstractos, são suportados pelos mecanismos de encaminhamento de mensagens, os quais são responsáveis por enviar os dados pelos canais de comunicação físicos adequados, adaptando protocolos de comunicação quando necessário, recorrendo a duas funções de comunicação básicas: putmessage e getmessage. Estas funções, que foram desenvolvidas para compatibilizar as funções de comunicação fornecidas pelo micro-núcleo 3L, para todos os tipos de processadores suportados, serão descritas no capítulo seguinte:

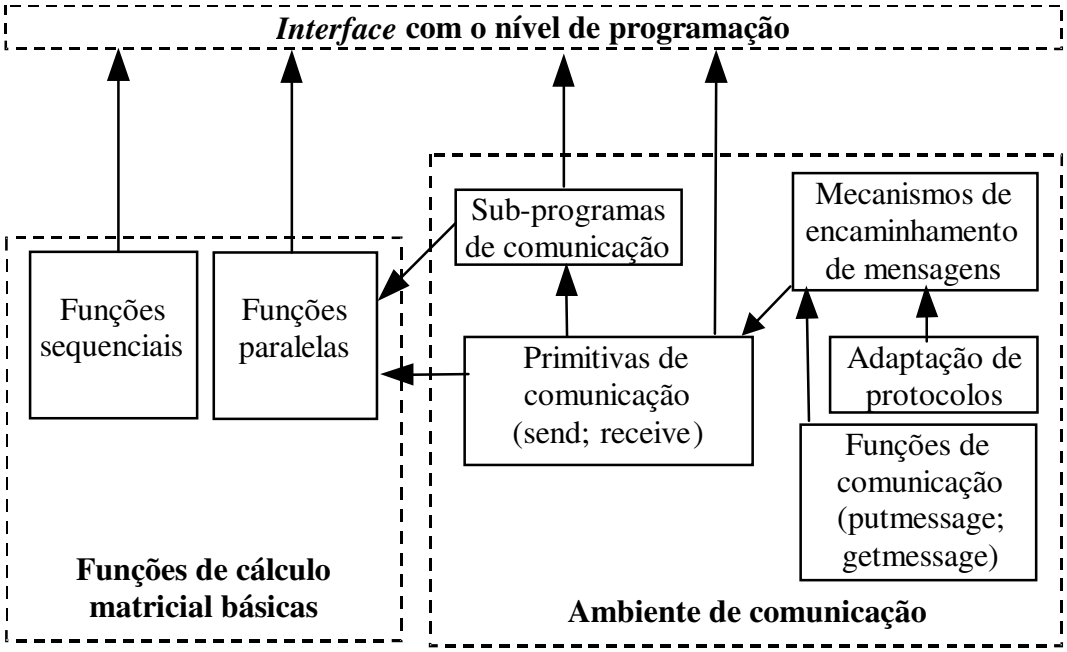


Fig. 3-4: Hierarquia do núcleo.

As funções de cálculo matricial, por sua vez dividem-se em dois grupos: funções de cálculo matricial sequenciais e funções de cálculo matricial paralelas. As primeiras são rotinas que tanto podem ser usadas em processamento sequencial como em processamento paralelo. A adição de matrizes constitui um exemplo destas funções. As funções paralelas, por outro lado, foram desenvolvidas segundo o paradigma de programação concorrente, e apenas podem ser aplicadas neste contexto. Estas últimas envolvem transferências de dados entre os vários processadores, recorrendo às primitivas de comunicação ou aos sub-programas de comunicação, implementados na secção de comunicações. Como exemplo deste tipo de

funções têm-se a inversão matricial. Ambos os grupos de funções são disponibilizados ao nível de programação, através do *interface* próprio acima referido.

Os componentes do núcleo podem também ser classificados de acordo com o âmbito em que operam. Componentes locais operam com dados locais a cada nó da rede, ao passo que os globais operam com dados distribuídos pela rede. Na figura seguinte, no diagrama da esquerda, pode-se observar que os únicos componentes globais consistem nas funções paralelas.

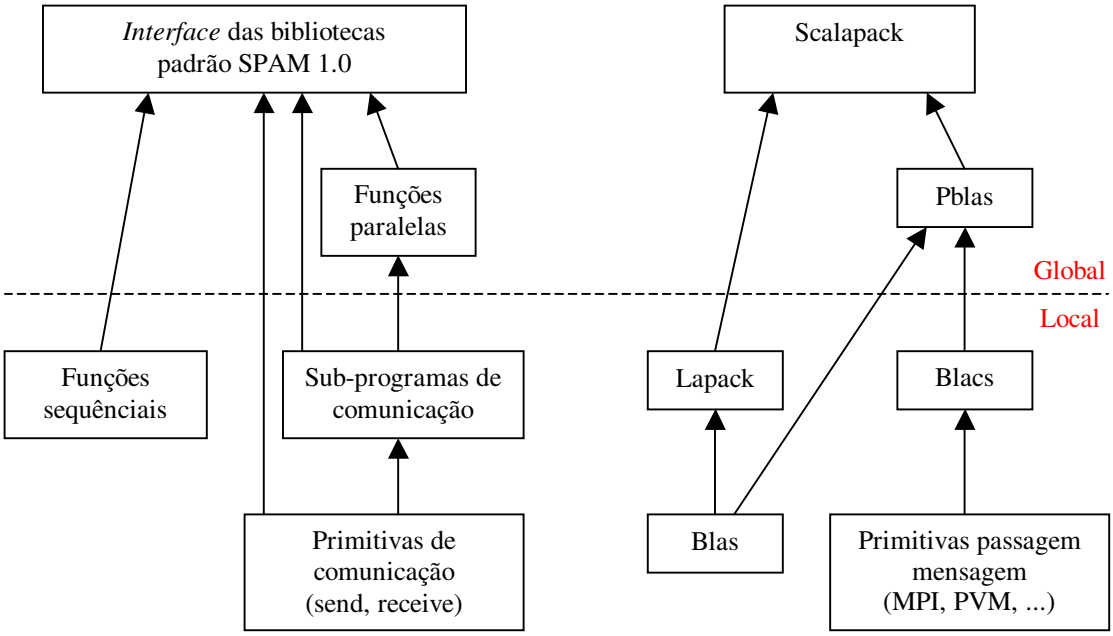


Fig. 3-5: Comparação entre a hierarquia do núcleo de uma aplicação SPAM 1.0 e a hierarquia do Scalapack.

Neste diagrama, por uma questão de simplicidade, não foram incluídos os componentes hierarquicamente abaixo das primitivas de comunicação - mecanismos de encaminhamento de mensagens, adaptação de protocolos e as funções de comunicação - no entanto é assumido que estes estão presentes, implementando assim estas primitivas.

Ao descrever o núcleo de uma aplicação SPAM neste formato, é também possível efectuar uma comparação com a hierarquia de *software* da biblioteca de álgebra linear Scalapack (Blackford et al, 1997), a qual está representada no diagrama da direita.

Em ambos os diagramas o ambiente de comunicações representa um componente basilar na hierarquia. Se no caso do SPAM as primitivas de comunicação, send e receive, são fornecidas por um ambiente de comunicações próprio, já no caso do Scalapack estas são fornecidas por ambientes de comunicação independentes como por exemplo o PVM (Geist et al.,1994) e o

MPI (MPI Forum, 1994; 1997). Em termos de SPAM, imediatamente acima das primitivas estão os sub-programas de comunicação, os quais constituem as rotinas de comunicação mais particulares, e que está ao nível do módulo Blacs, *Basic Linear Algebra Communication Subprograms* no original, do Scalapack. Embora no caso da distribuição de dados através da rede parte dos sub-programas disponibilizados por ambos os módulos sejam semelhantes, como por exemplo difusão de matrizes ou submatrizes, a estratégia de particionamento é diferente. De facto o Scalapack segue uma estratégia de particionamento de tarefas em oposição ao particionamento de dados usado pelo SPAM, o que implica que o módulo Blacs incluí rotinas apropriadas para esta realidade e que não são necessárias no âmbito do SPAM.

Em termos de módulos locais falta ainda referir o módulo de funções sequenciais, e que providencia um subconjunto de operações de álgebra linear quando comparado com as operações fornecidas pelo Lapack e pelo Blas.

Em termos globais, as funções paralelas operam sobre dados distribuídos do mesmo modo que o Pblas. Pode-se observar que este último módulo depende do Blas - *Basic Linear Algebra Subprograms*, consistindo aliás numa versão paralela do mesmo. De facto, como o Scalapack é uma implementação para arquitecturas de passagem de mensagem da biblioteca Lapack - *Linear Algebra Package* - pretendeu-se que o código de ambas fosse o mais próximo possível. Já no caso do SPAM, como a implementação foi feita de raiz, as funções paralelas não dependem das funções sequenciais. Estas últimas, embora locais, também podem ser usadas sobre matrizes distribuídas, desde que a operação de álgebra linear não envolva transferência de elementos entre as várias partições da matriz, como é o caso da multiplicação e adição de matrizes, e desde que sejam executadas sincronizadamente em todos os nós da rede. Para garantir este sincronismo, em alguns casos será necessário de recorrer a sinais entre os nós, difundidos pelo ambiente de comunicações. Por outro lado, operações que envolvam transferência de elementos entre as várias partições de uma matriz, estão disponíveis no módulo de funções paralelas.

Em termos de *interface* da biblioteca Scalapack, este é constituído pelo Scalapack propriamente dito, e que consiste num conjunto de funções baseadas no Lapack e no Blas. No caso do SPAM o *interface*, que é usado para comunicar com o nível de programação é constituído pelas operações de álgebra linear - módulos funções sequenciais e paralelas, mas também pelos sub-programas e primitivas de comunicação, de modo que, em caso de necessidade, estes possam ser utilizados para transmitir dados mesmo no nível de programação.

3.3.2 Nível de programação

O código C paralelo automaticamente gerado pelo tradutor, também designado por programa, é idêntico em cada processador, excepto em dois parâmetros, dependentes dos processadores utilizados e especificados no configurador visual, os quais indicam algumas diferenças no tratamento dos dados. O primeiro, um número inteiro único, identifica o processador ao ambiente de comunicação. O segundo parâmetro indica uma percentagem do total de linhas dos operandos matriciais, que será armazenada na memória local de cada processador, de modo que para diferentes tipos de processadores, a quantidade de dados a processar seja equilibrada. Quanto a este equilíbrio convém referir que, numa rede homogénea, o caso ideal em termos de eficiência, significa armazenar o mesmo número de linhas, das matrizes operandos, em todos os processadores; tal só é atingido se o número de linhas das matrizes operandos for um múltiplo do número dos processadores disponíveis. Isto significa que se tal não ocorrer, um ou mais processadores computarão mais uma linha que os restantes. Por outro lado, no caso de redes heterogéneas, embora o SPAM faça uma estimativa do particionamento de dados, de modo a equilibrar o esforço computacional, é conveniente permitir suficiente flexibilidade ao programador para que este possa fazer uma sintonia fina deste equilíbrio, alterando a percentagem de linhas armazenadas em cada processador automaticamente estimadas.

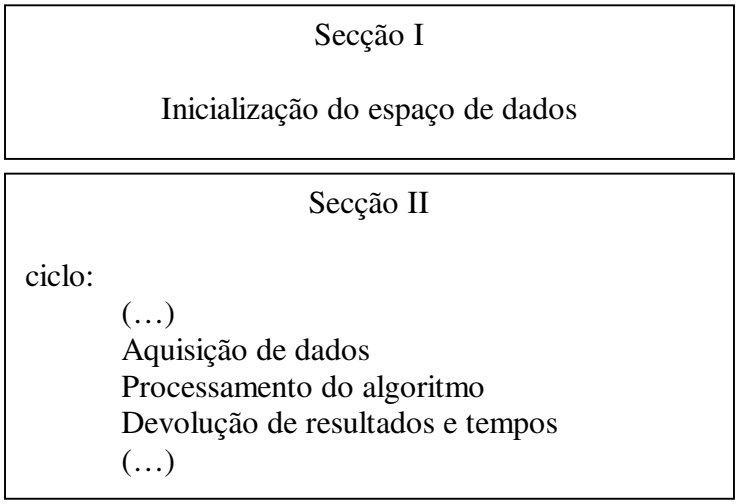


Fig. 3-6: Secções de código de um programa gerado pelo SPAM.

É também conveniente apontar que o programa, visto assentar num nível de abstracção inferior – o núcleo – como se pode observar no diagrama da Fig. 3-3, é independente do tipo e do número de processadores utilizados para cálculo. Apenas o número de linhas armazenado em cada processador varia.

No diagrama da Fig. 3-6 estão representadas as duas secções em pode ser dividido o código C paralelo fonte. A primeira secção, que normalmente é executada apenas uma vez, consiste na inicialização do espaço de dados. Não confundir com inicialização da aplicação, a qual será discutida no próximo capítulo e não necessita de ser explicitamente programada. Nesta secção deve ser reservada memória para os operandos, de acordo com o valor dos parâmetros acima referidos. A segunda secção consiste na parte de cálculo propriamente dita e poderá ser executada uma só vez ou tantas vezes quanto pretendido, desde que a dimensão dos operandos não varie. É assim possível criar um algoritmo, que em cada instante de amostragem trate os dados que lhe são enviados pelo *Host* e devolva resultados, como um estimador RLS (Wellstead e Zarrop, 1991), que será usado no primeiro caso de teste. O código desta secção, consiste basicamente na tradução das equações matriciais para chamadas a funções da biblioteca de cálculo matricial paralelo e de chamadas aos sub-programas de comunicação sempre que necessário. Nesta secção pode ainda ser introduzido código para medir os tempos de processamento de cada iteração, de modo que o desempenho do algoritmo possa ser avaliado.

Esta divisão do código em duas secções, embora não obrigatória, permite encapsular a aplicação numa S-function, tornando assim possível criar blocos para diagramas de simulação em SIMULINK (The Math Works Inc., 1997a; 1997b), os quais são processados numa rede paralela. A geração de uma aplicação deste tipo será abordada no capítulo 7.

3.3.3 Mapa de alocação de processos

Descrita a estrutura hierárquica da aplicação, apresenta-se no diagrama de blocos da Fig. 3-7 o mapa de alocação de processos simplificado e os fluxos de dados entre eles. Pode-se observar que a transferência de dados entre processadores é tratada por um mecanismo de entrada, E, e um mecanismo de saída, S. Além disso, em caso de necessidade de adaptação de protocolos entre processadores de diferentes tipos, podem ser incluídos nos canais de entrada ou de saída processos AP. Pode-se ainda observar um canal directo entre o mecanismo E e o mecanismo S, com esse mesmo sentido, usado para encaminhar mensagens não destinadas a este processador para o canal de saída correspondente. Esta estrutura simplificada, representa os mecanismos de encaminhamento de mensagens, que fazem parte do ambiente de comunicação e que serão discutidos com maior pormenor no capítulo seguinte. Entre os mecanismos E e S encontra-se o programa, gerado automaticamente pelo SPAM, que é servido por estes em termos de entrada e saída de mensagens.

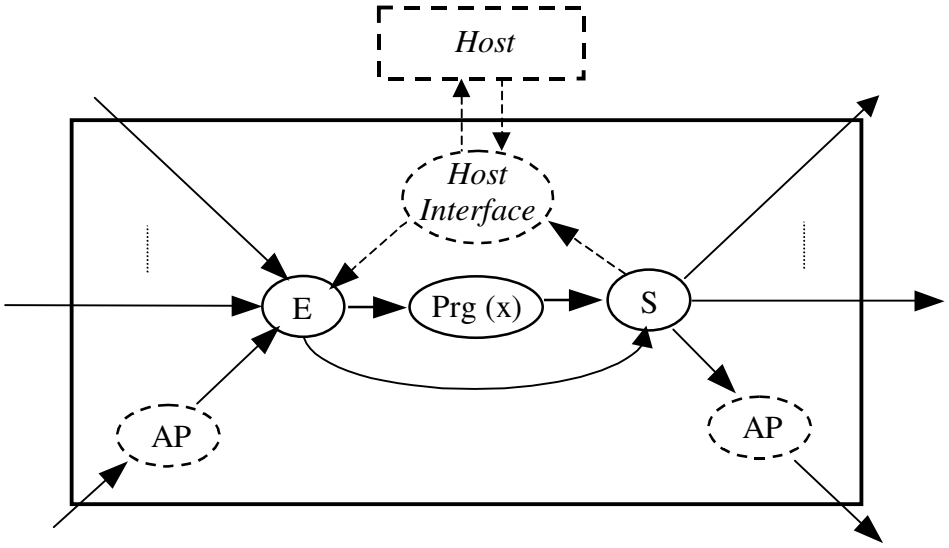


Fig. 3-7: Processos alocados num elemento de processamento.

Para generalizar este mapa também ao processador raiz, é necessário considerar um processo adicional, o *Host Interface*, conectado entre os mecanismos de entrada e saída de mensagens e o computador anfitrião, mais precisamente o processo servidor que se encontra alocado neste computador e que forma o subnível inferior, do nível do *Host*, como já foi atrás descrito.

3.4 Resumo

O modelo de programação proposto foi apresentado num fluxograma, de modo a indicar as etapas de desenvolvimento automático de código paralelo, bem como as ferramentas utilizadas. Este processo de geração automática de código, liberta o programador da identificação do paralelismo, e do particionamento do algoritmo pois este é feito automaticamente ao nível dos operandos matriciais, como já indicado no ponto 3.2. O programador deve apenas descrever o algoritmo no código sequencial SEQ 1.0, que será descrito em pormenor no capítulo 6, e descrever a rede alvo num diagrama de blocos, que será lido por uma ferramenta de configuração, de modo que a atribuição de tarefas aos processadores disponíveis seja também automaticamente efectuada pelo sistema. Esta ferramenta permite também que o algoritmo paralelo seja ensaiado sobre redes diferentes, alterando apenas o diagrama de blocos para representar as redes em questão. São estas as vantagens do modelo de programação proposto quando comparado com o modelo de programação paralela convencional, e que se traduzem por uma redução no tempo de desenvolvimento de uma aplicação paralela, aproximando-o do tempo de desenvolvimento de uma aplicação sequencial.

Foram também descritos os níveis de abstracção que compõem uma aplicação gerada pelo SPAM, e que serão analisados em pormenor nos capítulos seguintes. A estratégia de alocação de processos foi também introduzida. Esta será descrita em pormenor nos capítulos 4 e 5.

4 Ambiente de Comunicação

O ambiente de comunicação, disponibilizado pelo núcleo da aplicação gerada pelo SPAM, implementa uma camada de abstracção entre o *hardware* e o nível de programação. Assim, neste último nível a comunicação será feita recorrendo a sub-programas de transferências de dados, completamente independentes da realidade física da rede de processadores onde é executado o código. Os mecanismos de baixo nível usados para estabelecer a comunicação entre processadores têm um papel fundamental no desempenho de uma aplicação paralela, pois quanto menor fôr o tempo gasto em transferências de dados sobre a rede paralela, mais tempo de processador fica disponível para a computação do algoritmo, e assim maior será a eficiência do algoritmo paralelo.

Neste capítulo é analisado em pormenor o desenvolvimento destes mecanismos, recorrendo a várias estratégias com vista a rentabilizar ao máximo o *hardware*, mas sempre com o intuito de manter a compatibilidade e portabilidade do algoritmo gerado pelo SPAM. São assim ensaiados analiticamente vários modelos, de forma a obter o mais eficiente para a gama de processadores utilizados, sem no entanto recorrer a optimizações específicas para cada processador. O modelo adoptado é descrito pormenorizadamente e é introduzido o protocolo de comunicação utilizado. É descrita exaustivamente a construção dos mecanismos de encaminhamento de mensagens adoptados, de acordo com os casos de comunicação possíveis, e são abordadas algumas questões práticas sobre a sua implementação.

Baseado neste modelo são desenvolvidas funções de comunicação, com o intuito de compatibilizar os vários tipos de processadores ao nível dos mecanismos de comunicação, e assim disponibilizar duas primitivas de comunicação, send e receive, para os níveis seguintes. A um nível de abstracção ainda superior são desenvolvidos sub-programas de comunicação, que permitem a distribuição de dados pela rede paralela, e comunicação com o PC anfitrião, no nível de programação.

Finalmente o desempenho deste mecanismos será estudado para alguns casos reais de distribuição de dados.

4.1 Modelos de mecanismos de encaminhamento de mensagens

O tempo de transferência de dados pode ser representado, como já foi discutido no ponto 2.4.2.4, pelo modelo linear, equação (2-4), onde β representa os atrasos na comunicação também referidos nesse ponto. No caso dos mecanismos de encaminhamento de mensagens abordados neste capítulo, a parcela β também indica um atraso referente ao empacotamento e desempacotamento dos vectores de dados ou processamento do cabeçalho da mensagem, operações que serão descritas a seguir. No entanto, para a analise de modelos de mecanismos de comunicação que se irá efectuar a seguir, foi considerado que β tende para zero, o que sucede na realidade quando o comprimento das mensagens é considerável.
O modelo mais simples, pode ser retirado do mapa de alocação de processos apresentado no capítulo anterior, Fig. 3-7:

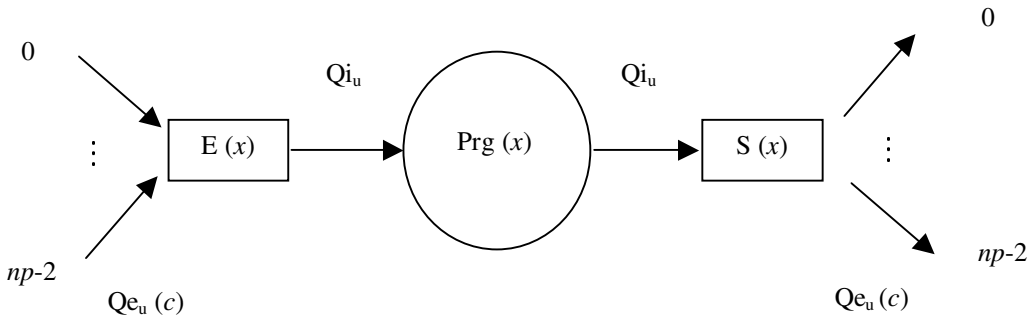


Fig. 4-1: Modelo de mecanismo de encaminhamento de mensagens Mec1, alocado no processador x

Neste modelo, designado por Mec1, é assumido que o número de processadores na rede é dado por np, e que todos os processadores estão ligados directamente. Assim em cada processador x, existem dois processos que gerem as transferências de dados com o exterior, os mecanismos de encaminhamento de mensagens propriamente ditos, designados por E (x) e S (x). Cada um destes processos opera sobre np-1 portos de comunicação unidireccionais. Existe ainda em cada processador, um processo de cálculo, Prg (x), onde $0 \leq x < np$. Qe$_u$ (c), representa o *quantum* de transferência de dados unidireccional sobre o porto de comunicação c, Qi$_u$ o *quantum* de transferência de dados unidireccional entre dois processos alocados no mesmo processador e $0 \leq c < (np\text{-}1)$.

Neste modelo, os processos de entrada e saída, dispõem de *c buffers*, um para cada porto de entrada e apenas um *buffer* para todos os portos de saída. As mensagens são tratadas numa base em que a primeira a chegar é a primeira a ser tratada. Deste modo, outras mensagens poderão ter de aguardar tratamento nos outros portos de comunicação.

Esta estratégia de encaminhamento, conhecida por *store and forward routing,* não é exclusiva. Poder-se-ia derivar um modelo baseado no conceito de *hot potato routing* ou *deflection routing*, onde as mensagens que deveriam aguardar tratamento são encaminhadas por um caminho alternativo, de modo que vão viajando pela rede até atingirem o seu destino final, nunca sendo armazenadas em *buffers* intermédios (Naor et al., 1998). Mas em termos de SPAM, como será descrito a seguir, o sistema supõe que o caminho entre dois nós é o mais curto, logo único, de modo que não existem caminhos alternativos para o mesmo destino, inviabilizando o encaminhamento por deflexão.

Para ensaiar os modelos, considere-se a seguinte experiência conceptual: cada processador envia uma mensagem, com o mesmo comprimento de dados, para todos os outros processadores, de modo que, por cada porto de comunicação de entrada e de saída, irá circular o mesmo volume de dados. Desta forma, exprimindo os tempos de comunicações exterior como bidireccionais, pode-se simplificar o modelo:

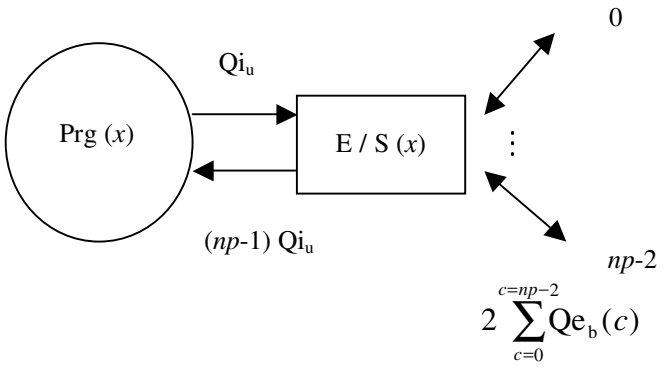


Fig. 4-2: Modelo simplificado do mecanismo de encaminhamento de mensagens Mec1, alocado no processador *x*

Na figura acima, em cada ramo está indicado não só o *quantum* de tempo de comunicação, mas também o número de vezes que esse ramo será utilizado em cada processador, em função do número de processadores na rede. O número de comunicações unidireccionais internas é dado por [(*np*-1) + 1] = *np*, pois é necessário comunicar entre os *buffers* de entrada e o processo de cálculo as (*np*-1) mensagens recebidas dos outros processadores da rede, ao passo que quando uma mensagem é enviada, só é necessário transferir entre o processo de cálculo e o *buffer* de saída uma mensagem. Como cada processador efectua estas transferências internas

concorrentemente com os outros processadores da rede, o tempo de comunicação interno da rede é dado pelo maior dos tempos de comunicação interno dos processadores que formam a rede, como é descrito pela primeira parcela da equação (4-1).

Quanto ao tempo de transferência externo da rede, é dado pela soma dos tempos de transferência de cada porto de comunicação. Visto que neste modelo os portos de comunicação são bidireccionais, ao passo que no modelo anterior são unidireccionais, a quantidade de informação que os atravessa é o dobro. Como o *quantum* unidireccional indica o tempo para comunicar uma palavra num só sentido, se se pretender considerar a comunicação de uma palavra em cada sentido o tempo requerido será o dobro deste *quantum*. Assim, a segunda parcela da equação (4-1) é multiplicada por 2, pois é assumido que em simultâneo é transferida a mesma quantidade de informação em ambos os sentidos. Então, o tempo total para efectuar o ensaio, será dado por:

$$(4\text{-}1) \quad T = \left[np \,.\, \max\left(\mathrm{Qi}_u(0),\ldots,\mathrm{Qi}_u(np-1)\right) + 2 \sum_{c=0}^{c=np-2} \mathrm{Qe}_b(c) \right].n$$

onde n é o comprimento da mensagem enviada por cada processador em palavras. Se a rede for homogénea, o *quantum* de comunicação interno será igual em todos os processadores e o *quantum* de comunicação externo em cada porto de comunicação é igual e designado por Qe_b. Neste caso, o tempo total será:

$$(4\text{-}2) \quad T = \left[np.\mathrm{Qi}_u + 2.(np-1).\mathrm{Qe}_b \right].n$$

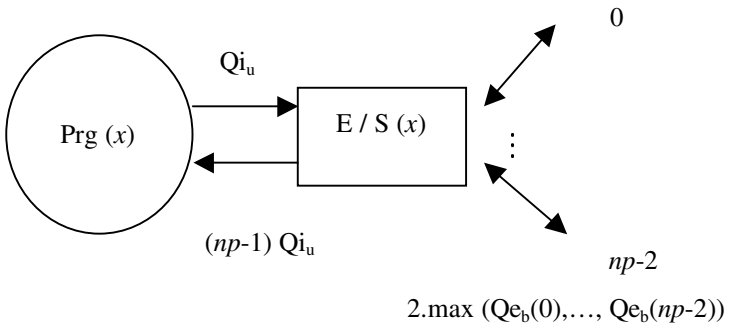


Fig. 4-3: Modelo simplificado do mecanismo de encaminhamento de mensagens Mec2, alocado no processador *x*

Considerando, que o *hardware* suporta portos de comunicação que operam concorrentemente, como é o caso dos processadores utilizados neste trabalho, já ensaiados no capítulo 2, obtém-se o modelo representado na figura anterior, que será designado por Mec2.
Segundo este modelo, o tempo total para efectuar este ensaio será:

$$(4\text{-}3) \quad T = \left[np \,.\, \max\left(\mathrm{Qi}_u(0),...,\mathrm{Qi}_u(np-1)\right) + 2.\max\left(\mathrm{Qe}_b(0),...,\mathrm{Qe}_b(np-2)\right) \right] n$$

ou se se tratar de uma rede homogénea:

$$(4\text{-}4) \quad T = \left[np.\mathrm{Qi}_u + 2.\mathrm{Qe}_b \right] n$$

O modelo Mec2, apresenta um ganho de desempenho considerável, em relação ao anterior, pois tornando a comunicação externa concorrente, (*np*-1) transferências de dados são efectuadas no tempo de apenas uma, no caso homogéneo, ou no tempo da mais morosa, no caso heterogéneo. É ainda conveniente apontar que este ganho de desempenho, só é válido para topologias que impliquem, e onde seja possível, a utilização de mais do que um porto de comunicação bidireccional por processador e para $np > 2$. Assim, uma fila de processadores, não tira partido das optimizações efectuadas neste modelo.
Analisando o *hardware* utilizado mais uma optimização é ainda sugerida, a qual reduz o tempo de comunicação: eliminar os *buffers* existentes nos processos de entrada e saída E/S (*x*), permitindo que as mensagens vindas dos portos de comunicação sejam comunicadas directamente para o segmento de memória do processo de cálculo Prg (*x*).

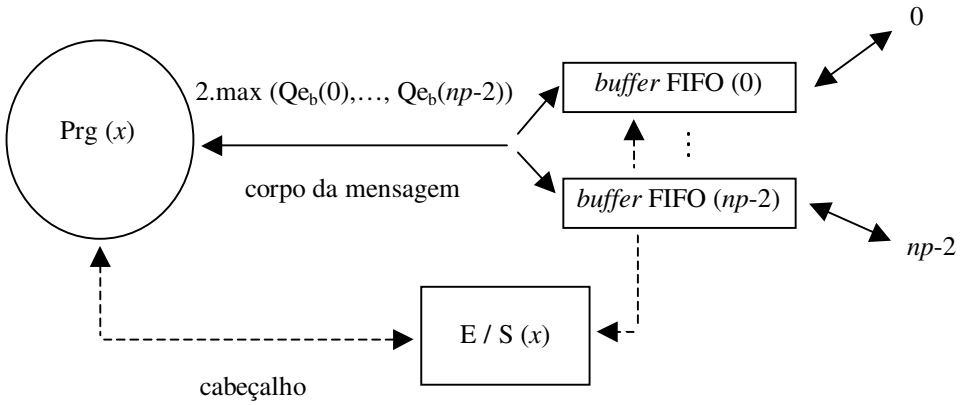


Fig. 4-4: Modelo simplificado do mecanismo de encaminhamento de mensagens Mec3, alocado no processador *x*

Na realidade, os processadores em estudo usam para cada porto de comunicação, pequenos *buffers* FIFO, com um comprimento de algumas palavras. Isto implica que enviar ou receber dados por esse portos, significa transferir dados ordenadamente entre estes *buffers* e um determinado segmento de memória, revelando-se a existência de *buffers* adicionais nos mecanismos de entrada e saída redundante, servindo apenas para atrasar a transferência de mensagens entre processadores. Assim, de acordo com este último modelo, lendo o cabeçalho que identifica uma mensagem, os mecanismos de encaminhamento de mensagens decidem para onde a devem dirigir, e se for o caso, transferem directamente o corpo da mensagem dos *buffers* FIFO, para o segmento de memória do processo de cálculo, sem que estes dados sejam temporariamente armazenados nos processos E/S, como é o caso dos dois outros modelos anteriores. Eliminando assim o tempo de comunicação interno, obtém-se o modelo Mec3, apresentado na Fig. 4-4.

Como foi referido atrás, nesta análise é considerado que o atraso nas comunicações, introduzido pelos mecanismos de entrada e saída, quando descodificando o cabeçalho é desprezado, de modo que o tempo de comunicação total, para o ensaio já referido, é dado por:

(4-5) $$T = 2.\max\left(\mathrm{Qe}_b(0), \ldots, \mathrm{Qe}_b(np-2)\right).n$$

no caso de uma rede heterogénea e no caso de uma rede homogénea por:

(4-6) $$T = 2.\mathrm{Qe}_b.n$$

o que significa uma redução do tempo de comunicação, em relação ao modelo Mec2, de np.max (Qi_u (0),…, Qi_u (np-2)) e de np.Qi_u, respectivamente para os casos heterogéneo e homogéneo.

Pode-se assim comparar analiticamente os 3 modelos apresentados para redes homogéneas constituídas pelos processadores ensaiados no capitulo 2. Considerando as melhores estratégias de alocação, indicadas nesse mesmo capítulo, mas para o caso em que o canal físico de comunicação é o mais lento, ou seja usando o comutador de ligação no caso de redes T8 e um cabo de ligação no caso de redes C40, o seguinte quadro resume a largura de banda máxima interna unidireccional e externa bidireccional já apresentadas na Fig. 2-16 e na Fig. 2-21, respectivamente:

	T8	C40	ADSP21060
LB interna unidireccional (MBytes / s)	16,360	14,134	76,923
LB externa bidireccional (MBytes / s)	2,156	16,586	39,6

tabela 4-1: Larguras de Banda interna e externa

Para obter os *quanta* de comunicação correspondentes, isto é o tempo de transferência de uma palavra de 4 *bytes* por um dado canal, pode-se usar:

$$Quantum(\mu s) = 4.\frac{1}{\text{Largura de Banda}(\text{MBytes/s})} \quad (4\text{-}7)$$

obtendo-se a tabela:

	T8	C40	ADSP21060
Qi_u (µs)	2,444	0,283	0,052
Qe_b (µs)	1,855	0,241	0,101

tabela 4-2: *Quanta* de comunicação interna e externa

Considere-se o caso de uma rede com 3 processadores, onde cada um deles armazena um terço das linhas de uma matriz quadrada.

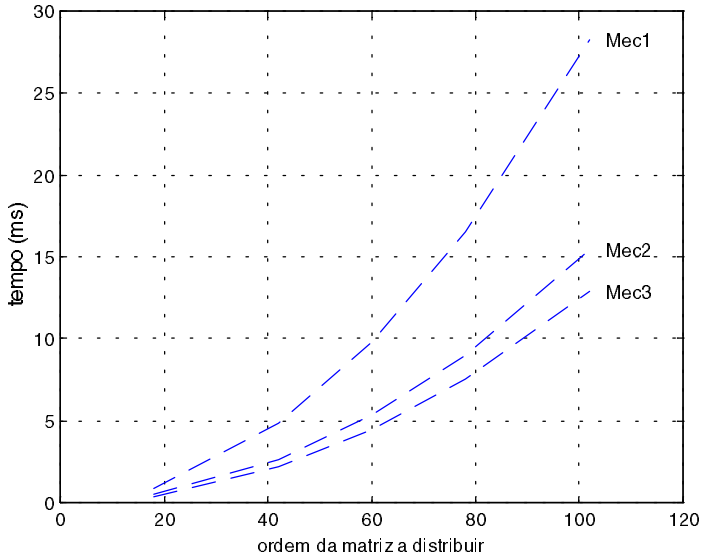


Fig. 4-5: Comparação dos 3 modelos de mecanismos de comunicação para uma rede T8

Redistribuir essa matriz, isto é, cada processador difundir o seu terço de linhas da matriz para os outros 2, de modo que em cada processador exista uma representação completa dessa matriz, representa um cenário comum quando se executa um algoritmo matricial paralelo gerado pelo SPAM, e está de acordo com a experiência conceptual usada para derivar os 3 modelos de mecanismos. Aplicando os *quanta* indicados na tabela 4-2 às equações que descrevem os referidos modelos, para o cenário acima descrito, obtêm-se os resultados apresentados na figura anterior e nas duas seguintes.

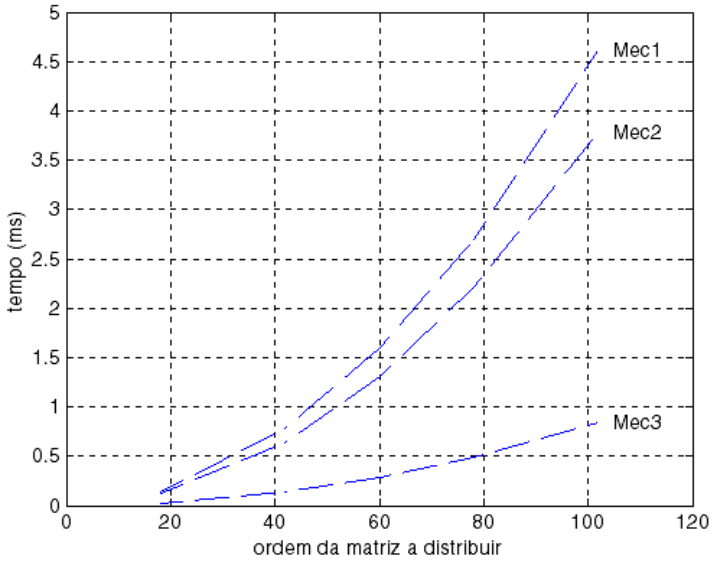


Fig. 4-6: Comparação dos 3 modelos de mecanismos de comunicação para uma rede C40

Os tempos apresentados referem-se a topologias com 3 processadores, pois se forem consideradas topologias com 2 processadores os modelos Mec1 e Mec2 são idênticos. Por outro lado, se se considerar topologias com mais de 3 processadores, numa rede C40 não é possível ligar directamente todos os processadores, existindo atraso nas comunicações pelo redireccionamento de mensagens, adulterando a medida de desempenho que se pretende apresentar.

Comparando as três figuras, observa-se que o tempo de comunicação é reduzido de um factor inferior a 2, entre o modelo Mec1 e o modelo Mec2, para qualquer dos três tipos de rede. No caso de uma rede com 3 processadores, visto que segundo o modelo Mec2 as comunicações em ambos os portos são concorrentes, este factor deve ser 2. De facto os tempos são divididos por esse factor em termos de comunicação externa, mas têm ainda de ser consideradas as comunicações internas existentes nestes dois modelos modelos.

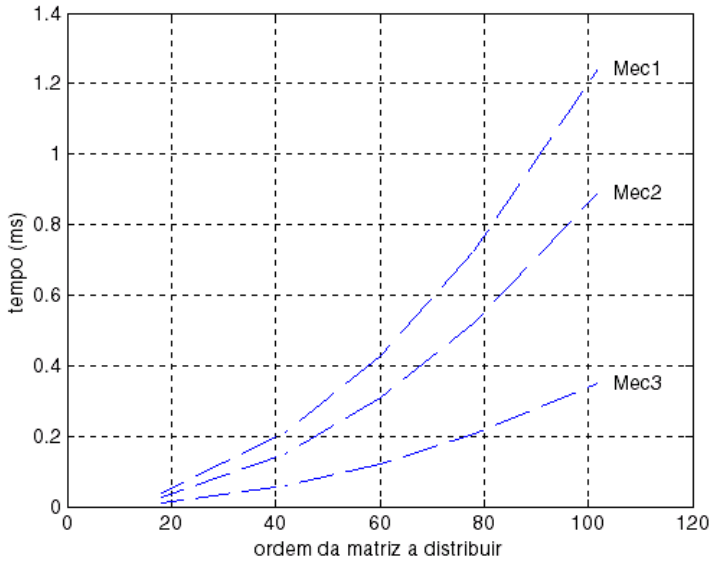


Fig. 4-7: Comparação dos 3 modelos de mecanismos de comunicação para uma rede Sharc

De acordo com as figuras anteriores, os factores de redução de tempos de comunicação entre o modelo Mec1, e os outros modelos é:

Ganho	T8	C40	ADSP21060
$\frac{\text{tempo comunicação Mec1}}{\text{tempo comunicação Mec2}}$	1,84	1,22	1,39
$\frac{\text{tempo comunicação Mec1}}{\text{tempo comunicação Mec3}}$	2,20	5,52	3,54

tabela 4-3: Factor de redução do tempo de comunicação entre o modelo Mec1 e os outros modelos

Usando a seguinte relação para comparar os tempos de comunicação externo e interno:

(4-8) $$\frac{Qeb}{Qib}$$

e considerando comunicações internas bidireccionais, o que implica que os valores apresentados na 1ª linha da tabela 4-2 são duplicados, obtém-se:

	T8	C40	ADSP21060
Qi_b (μs)	0,489	0,566	0,104
Qe_b (μs)	1,855	0,241	0,101
$\frac{Qeb}{Qib}$	3,795	0,426	0,973

tabela 4-4: Relação entre os *quanta* de comunicação interna e externa

Para uma rede T8, cujo factor de redução entre os modelos Mec1 e Mec2 é o que mais se aproxima do ideal, isto é 2, a relação (4-8) é a maior, ao passo que para uma rede C40, onde esse factor toma o valor 1,36 o mais afastado de 2, esta relação é a menor. Tal comportamento mostra como as comunicações internas mais lentas podem ter uma influência negativa apreciável no desempenho dos modelos. No entanto, o modelo Mec3 vem eliminar a necessidade de comunicações internas, de modo que as arquitecturas que mais beneficiam com isto são as mais atrasadas pelas comunicações internas, isto é, as que tem a relação anterior menor. É o caso da rede C40, com um factor de redução de tempo entre modelo Mec3 e o modelo Mec1 de 5,52.

4.2 Primitivas de comunicação

Qualquer que seja o mecanismo de encaminhamento utilizado, são definidas duas primitivas de comunicação, as quais permitem receber e enviar mensagens, encapsulando dados e informação sobre o destino e a origem no protocolo PCM, ou protocolo de controlo de mensagens. Estas primitivas permitem aceder aos mecanismos de encaminhamento, de modo que a transferência de mensagens entre os vários processadores da rede, incluindo o processador raiz e o anfitrião, seja feita a um nível de abstracção independente da topologia e da natureza física da rede. As funções tem o seguinte formato:

```
send (descritor, dados)
receive (descritor, dados)
```

É pois adicionado ao início dos dados a transferir um descritor ou cabeçalho, com comprimento de 4 *bytes*. Podemos assim definir as mensagens transmitidas segundo o protocolo PCM em duas áreas distintas:

Mensagem	Comprimento
descritor	4 *bytes*
dados	1 a 16384 elementos de 4 *bytes* ou 4 a 65535 *bytes*

tabela 4-5: Protocolo PCM

O descritor consiste num inteiro com comprimento de 4 *bytes*, onde os conjuntos de *bits* indicados na tabela seguinte descrevem a mensagem. O primeiro campo indica o comprimento da mensagem em elementos de 4 *bytes*, visto que para certas famílias de processadores o elemento atómico de comunicação tem este comprimento. Além disso 4 *bytes* é o comprimento dos tipos de dados inteiros (int) ou virgula flutuante (float), em ANSI C.
Como o comprimento máximo de uma mensagem é dado pelos 14 *bits* do campo MSG_LEN, cada pacote pode transportar apenas 16K elementos, de modo que se o comprimento do vector de dados a transmitir for superior, terá de ser dividido em vários pacotes. Assim, se o pacote em causa for o último pacote que transporta a mensagem, o campo MSG_END toma o valor 1, senão toma o valor 0.

Nome do campo	*bits*	Descrição
MSG_LEN	0 - 13	Comprimento da mensagem em elementos de 4 *bytes*. Máximo 16 K elementos.
MSG_END	14	0 - Pacote não contém todo o vector de dados 1 - Último pacote da mensagem
MSG_SRC	15 - 21	Processador origem (0 a 127)
MSG_DST	22 - 28	Processador destino (0 a 127)
MSG_ERR	29	0 - Mensagem de dados 1 - Mensagem de erro
MSG_FLT	30	0 - dados inteiros (int) 1 - dados representados em vírgula flutuante (float)
MSG_RSV	31	Reservado para futuras implementações

tabela 4-6: Descritor de mensagem segundo o protocolo PCM

Os dois campos seguintes indicam o processador ou nó origem e destino respectivamente.

Constante	Valor	Descrição
SRC_MASTER	126	origem: processo de controlo
SRC_ANY	127	origem: qualquer instância Prg (*x*)
DST_MASTER	126	destino: processo de controlo
DST_ALL	127	destino: todas as instâncias Prg (*x*) excepto a origem

tabela 4-7: Constantes de encaminhamento de mensagens segundo o protocolo PCM

Cada instância da aplicação, os processos Prg (*x*), podem ter índices entre 0 e 125, o que implica que se pode distribuir uma aplicação por uma rede com o máximo de 126 processadores. Por outro lado, o *Host interface* ou o processo de controlo, que reside sempre no nó 0, é identificado por 126, ao passo que uma mensagem cujo destino são todas as instâncias Prg (*x*) excepto a origem, é identificada por 127. São assim definidas, quanto à origem e destino da mensagem as constantes apresentadas na tabela 4-7.

Os dois *bits* seguintes definem o tipo da mensagem. O campo MSG_ERR toma o valor 1 se se estiver na presença de uma mensagem de erro. Este tipo de mensagens é muito útil para indicar erros na altura da execução. Existem dois tipos de mensagens de erro. Se o comprimento da mensagem for 1, consiste apenas no código de erro que será interpretado pelo processo de controlo. Se o comprimento for maior que 1, a mensagem é composta pelo código de erro, um inteiro, e por uma cadeia de caracteres fornecida pelo programador. O tratamento deste tipo de mensagens é feito no processo de controlo e será descrito mais à frente.

Se a mensagem não fôr de erro, está-se na presença de uma mensagem de dados. O campo MSG_FLT toma o valor 1 se os dados forem números de vírgula flutuante e 0 se forem números inteiros. Tal distinção é necessária, pois alguns processadores representam internamente os números de virgula flutuante em formatos que não seguem o padrão IEEE 754. Assim estes dados devem ser convertidos para a representação interna do processador destino sempre que esta seja diferente do processador origem. Como no caso do *hardware* correntemente suportado, apenas o C40 tem uma representação interna diferente do padrão, este processador será o responsável pela conversão. Para o caso de redes mais heterogéneas, será conveniente considerar uma padrão comum para o transporte de qualquer tipo de dados na rede, como é o caso do XDR (Sun Microsystems, Inc., 1987), que suporta também números inteiros e complexos.

Finalmente, o último *bit* do cabeçalho não é correntemente utilizado, estando reservado para futuras implementações.

4.3 Classificação das comunicações quanto à origem e ao destino

O ambiente de comunicações suporta vários casos de comunicação, classificados quanto ao destino e origem da mensagem. Todos estes casos são baseados nos dois casos simples apresentados a seguir. Assim a estratégia de encaminhamento varia de acordo com estes casos básicos.

Na exposição seguinte, é considerado que a rede é composta por *np* nós, isto é o número de instâncias Prg (*x*) da aplicação. Em cada nó residem também os processos que formam o ambiente de comunicações. No primeiro nó ou raiz está ainda alocado o processo de controlo. Na análise é admitido que cada nó é alocado num processador, mas também é válida para o caso de simulação de uma rede, onde todos os nós são alocados no processador raiz.

4.3.1 Um para um

Este é o caso mais simples. A mensagem tem origem num dado nó e percorre a rede até chegar ao destino. Como o destino é único, não é deixada cópia da mensagem em qualquer outro nó que possa existir no caminho. Assim no descritor ou cabeçalho da mensagem os campos que indicam o destino e a origem tomam o valor dos índices dos nós respectivos. Em termos do padrão MPI (Snir, et al., 1995) este tipo de comunicação é referido como ponto a ponto.

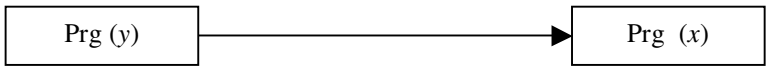


Fig. 4-8: Comunicação entre as instâncias *x* e *y* da aplicação alocadas em dois nós diferentes

A partir deste caso podem ser derivados os seguintes:

4.3.1.1 Controlo para um

O processo de controlo, também referenciado como *Host interface*, envia uma mensagem para um nó da rede. Como este processo está sempre situado no nó 0 ou raiz, e este também aloca a instância da aplicação Prg (0), para diferenciar este processo da instância da aplicação alocada no mesmo nó, o primeiro é identificado, de acordo com o protocolo PCM já apresentado, por SRC_MASTER.

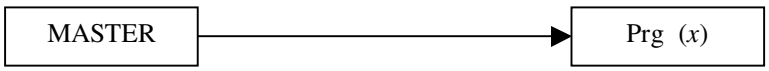


Fig. 4-9: Comunicação entre o processo de controlo e uma dada instância da aplicação

4.3.1.2 Controlo para todos (distribuição)

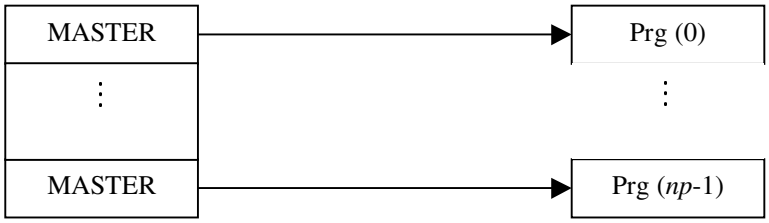


Fig. 4-10: Distribuição de uma matriz proveniente do processo de controlo para todos os nós

Este caso é aplicável quando se pretende distribuir um operando matricial, proveniente do processo de controlo, e possivelmente do anfitrião, por todos os nós da rede. Neste caso o operando é particionado em tantos conjuntos de linhas consecutivas como o número de nós, e é usado o tipo de comunicação apresentado em 4.3.1.1, "processo de controlo para um", simultaneamente entre cada uma das partições e o nó correspondente, como mostra o diagrama anterior.

4.3.1.3 Um para controlo

Um qualquer nó da rede envia uma mensagem para o processo de controlo. Este é o caso inverso de 4.3.1.1. Neste caso, o destino é indicado pela constante DST_MASTER.

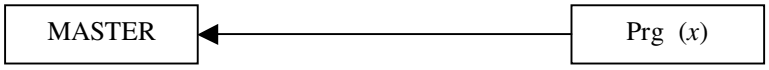


Fig. 4-11: Comunicação entre uma instância da aplicação alojada num dado nó *x* e o processo de controlo

4.3.1.4 Todos para controlo (colecção)

Este caso é utilizado quando se pretende agrupar no processo de controlo, um operando matricial distribuído por todos os nós da rede, e possivelmente enviá-lo para o anfitrião. Este é o caso inverso de 4.3.1.2. É assim usado o tipo de comunicação "um para controlo" simultaneamente entre cada instância da aplicação Prg (*x*) e o processo de controlo.

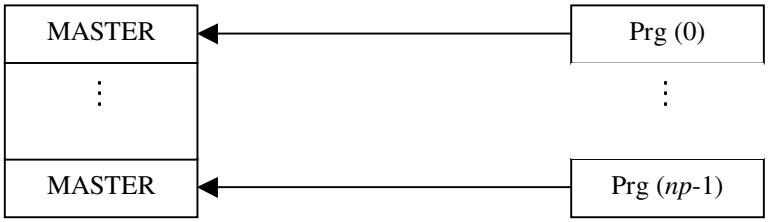


Fig. 4-12: Colecção de uma matriz distribuída por todos os nó, no processo de controlo

4.3.1.5 Todos para todos I (indexação)

Todos os nós enviam parte de um operando neles armazenado para um outro determinado nó. Isto é, cada nó efectua *np*-1 operações "um para um". Este caso é útil para implementar a transposição de uma matriz distribuída como um problema de comunicações, como será abordado no capítulo seguinte.

4.3.2 Um para todos (difusão)

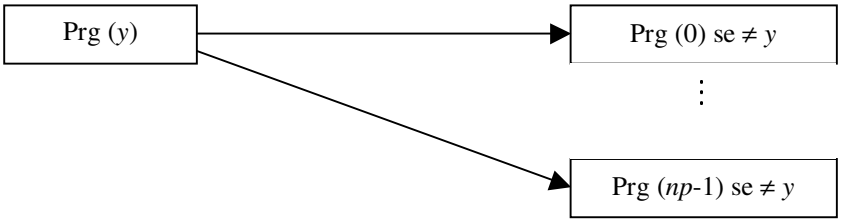


Fig. 4-13: Difusão de uma mensagem de um nó para todos os outros nós da rede

Neste caso, uma mensagem originada num dado nó *y* é enviada para todos os outros nós. Como o destino é toda a rede excepto o processo que a envia, e de acordo com o protocolo já referido, o destino é DST_ALL. O padrão MPI já referido, designa este tipo de comunicação como colectiva.

Os seguintes casos são baseados no anterior:

4.3.2.1 Controlo para todos (difusão)

Em tudo semelhante ao anterior, no entanto a origem é o processo de controlo e o destino todos os nós da rede. Este caso é útil quando é necessário atribuir valores iguais a uma mesma variável definida em todos os nós, possivelmente um escalar como é o caso de uma medida física, adquirida de um sistema externo.

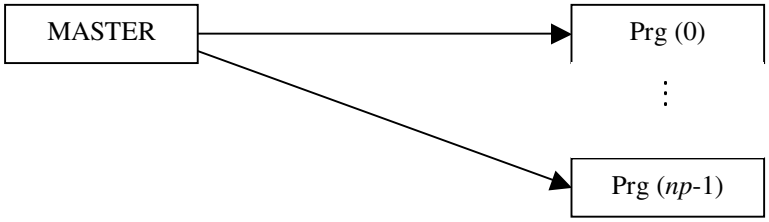


Fig. 4-14: Difusão de uma mensagem do processo de controlo para todos os nós da rede

4.3.2.2 Todos para todos II (difusão)

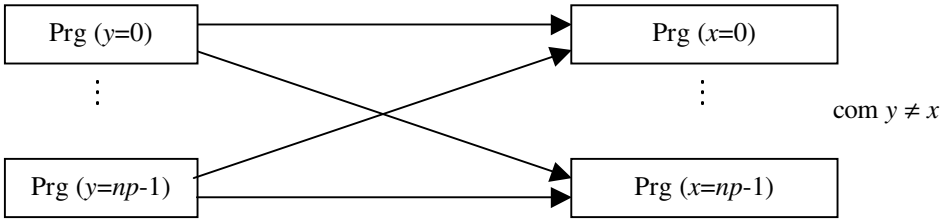


Fig. 4-15: Difusão de uma mensagem de todos os nós para todos os outros

Cada nó da rede efectua a operação "um para todos". Esta operação é requerida para enviar a parte de um operando armazenada em cada nó para todos os outros nós, de modo a obter-se um operando completo em cada nó.

4.4 Tabelas de encaminhamento de mensagens

Após indicar os tipos de transferência de mensagens suportados, é conveniente descrever o método utilizado para indicar aos mecanismos de encaminhamento de mensagens o percurso destas. Para tal pode-se partir do seguinte axioma:

Axioma 4-1: Numa difusão de uma mensagem entre um nó e todos os outros nós, o ambiente de comunicações, residente em cada nó da rede, tem apenas de saber por que porto ou portos de comunicação envia a mensagem que chegou por um outro qualquer porto.

Nos modelos de comunicações apresentados, todos os nós da rede podem comunicar com todos, mesmo que não existam ligações directas entre nós. Além disso uma mensagem pode ter como destino todos os nós da rede ou apenas um. Isto implica que qualquer peça de

informação que chegue a um determinado nó pode ter como destino esse mesmo nó, ou esse nó pode ser apenas um ponto de passagem para o destino da mensagem. Assim, são utilizadas em cada nó, duas tabelas de encaminhamento diferentes.

De acordo com o axioma anterior, quando em presença de uma difusão, isto é de uma mensagem que deve ser enviada a todos os nós, em cada nó deverá ser usada uma tabela tal que, dependente do porto de entrada, envia a mensagem para um conjunto de portos de saída tais que a mensagem atinja todos os nós. Esta tabela deverá consistir numa matriz de valores lógicos ou binários, com tantas linhas e colunas como os portos de comunicação existentes num dado nó *x*, *npc* (*x*). Deverá ter ainda mais uma última linha e uma última coluna para indicar o porto de comunicação interno entre o ambiente de comunicação e a aplicação alocados num nó.

$$(4\text{-}9)\ \mathbf{TED}(x) = \begin{bmatrix} 0 & \cdots & \mathbf{TED}(x)_{0,npc\,(x)} \\ \vdots & \ddots & \vdots \\ \mathbf{TED}(x)_{npc\,(x),0} & \cdots & 0 \end{bmatrix}$$

Nesta matriz, denominada tabela de encaminhamento de difusão, cada linha está associada a um porto de entrada. Assim, uma mensagem que chega pelo porto de entrada *k*, é reenviada pelos portos de saída cujo número é dado pelos índices de coluna, ao longo dessa mesma linha *k*, que se referem a elementos cujo valor lógico é verdadeiro. Como neste modelo não é necessário reenviar mensagens pelo mesmo porto por onde foram recebidas, a diagonal principal é preenchida por elementos com valor lógico falso, inibindo tal caminho.

Nesta tabela não existe, no entanto, informação para indicar quando uma mensagem já chegou a todos os nós, e portanto esta poderá não ser descartada da rede de comunicação, e no pior dos casos pode circular indefinidamente numa malha da rede. Para evitar esta situação, é necessário recorrer a outra tabela complementar, que indique ao ambiente de comunicações de um determinado nó *x*, que uma mensagem proveniente de um qualquer nó *y*, pode ou não ser difundida para os portos de comunicação de saída do nó *x*, dados pela **TED**. Esta tabela de fim de difusão consiste num vector coluna para cada nó, com tantas linhas quantos os nós existentes na rede, *np*, e onde cada elemento também toma um valor lógico ou binário. Assim, no caso do se estar em presença de um valor verdadeiro, a difusão de uma mensagem proveniente do nó correspondente ao índice dessa linha, deve terminar neste nó.

$$(4\text{-}10)\quad \mathbf{TFD}(x) = \begin{bmatrix} \mathbf{TFD}(x)_{0,0} \\ \vdots \\ \mathbf{TFD}(x)_{np\text{-}1,0} \end{bmatrix}$$

Como uma mensagem chegada ao ambiente de comunicação proveniente do próprio nó *x*, deve sempre ser enviada para os portos de saída, o elemento da linha *x* deve ser nulo.

Se bem que esta técnica satisfaça as condições necessárias à difusão de mensagem de todos os nós para todos, resulta um pouco complexa a consulta de duas tabelas. Como é preferível tornar o ambiente de comunicações o mais simples possível, de modo que o atraso introduzido por este seja mínimo, será preferível consultar apenas uma tabela. Além disso poupa-se ainda um pouco da memória requerida pelo ambiente de comunicações, o que poderá revelar-se útil em determinados casos.

Tal é possível, se a tabela de consulta se abstrair um pouco das características físicas dos nós. Rescrevendo o Axioma 4-1 da seguinte forma:

Axioma 4-2: Numa difusão de uma mensagem entre um nó e todos os outros nós, o ambiente de comunicações, residente em cada nó da rede, tem apenas de saber por que porto ou portos de comunicação envia a mensagem que chegou proveniente de um determinado nó.

Assim, se os índices das linhas da tabela de encaminhamento de difusão representarem nós de origem da mensagem, em vez de portos de entrada, a nova tabela única contem toda a informação necessária para a difusão de mensagens na rede.

$$(4\text{-}11)\quad \mathbf{TEG}(x) = \begin{bmatrix} 0 & \cdots & \mathbf{TEG}(x)_{0,npc\,(x)\text{-}1} \\ \vdots & \ddots & \vdots \\ \mathbf{TEG}(x)_{np\,\text{-}1,0} & \cdots & 0 \end{bmatrix}$$

Nesta nova tabela, que deve ter tantas linhas como o número de nós, pode-se remover ainda a última coluna, necessária na tabela representada em (4-9) para representar o caminho para as mensagens que se destinam ao próprio nó, adicionando-se uma simples regra à estratégia de encaminhamento:

regra 4-1: Numa difusão de uma mensagem entre um nó e todos os outros nós, todas as mensagens chegadas a um determinado nó, tem como destino esse nó, desde que não provenham desse mesmo nó.

Obtém-se então a tabela de encaminhamento geral, representada em (4-11), com tantas colunas como o número de portos de saída. Nesta tabela, um elemento binário **TEG** $(x)_{l,c}$ não nulo, indica que se a mensagem proveniente do nó l deve ser encaminhada para o porto de saída c.

Além das difusões, existem ainda outros tipos de comunicação, também já atrás indicados, que não podem ser implementados recorrendo à tabela anterior. Estes tipos tem por base o envio de uma mensagem, não de um nó para toda a rede, mas sim entre dois nós definidos. Seja o seguinte axioma:

Axioma 4-3: Quando uma mensagem tem como destino um nó definido, o ambiente de comunicações, residente em cada nó da rede, tem apenas de saber por que porto de comunicação envia a mensagem.

Assim, quando em presença de mensagens com destino definido, como uma determinada instancia da aplicação Prg (x), ou nó, ou o processo de controlo, denominado na Fig. 3-3 como *host interface*, é usada uma outra tabela, denominada de encaminhamento particular. Esta, também alocada em cada nó, consiste num vector com tantos elementos inteiros como o número de nós existentes na rede.

$$(4\text{-}12)\quad \mathbf{TEP}(x) = \begin{bmatrix} \mathbf{TEP}(x)_{0,0} \\ \vdots \\ \mathbf{TEP}(x)_{np\text{-}1,0} \end{bmatrix}$$

Neste vector, cada elemento indica o número do porto de comunicação que deve ser utilizado para enviar uma mensagem para o nó dado pelo índice da linha. Assim o elemento da linha 0 indica o porto de comunicação requerido nesse nó, para enviar uma mensagem para o processo de controlo, visto este estar alocado no processador ou nó raiz cujo índice é 0. Nesta tabela, um elemento negativo indica que não existe um porto de saída para o destino

correspondente. É o caso de uma mensagem que tem como destino o nó em causa. Assim, o elemento **TEP** $(x)_{x,0}$, respeitante à tabela alocada no nó *x*, toma o valor -1.

Nestes modelos de encaminhamento, dependendo da topologia da rede, é possível que num dado nó, o porto de saída seja idêntico para mais que um destino. Por outro lado não implica que o nó destino seja o seguinte. Desta forma, o mecanismo de encaminhamento residente em cada nó, dispõe apenas de informação sobre qual o ou os portos de comunicação pelos quais deve ser enviada uma mensagem destinada a um ou mais nós. A informação sobre todos os caminhos necessários para enviar uma mensagem entre quaisquer dois nós, só está disponível ao nível de toda a rede, isto é agrupando todas as tabelas de encaminhamento de mensagens alocadas em cada nó. Deve assim existir uma aplicação que veja a rede no seu todo e que gere as tabelas de encaminhamento para cada nó. Tal tarefa poderá ser efectuada durante o processo de geração automática de código paralelo. De facto é assim que são geradas as tabelas já referidas, por uma ferramenta que será apresentada no capítulo 7.

4.5 Descrição pormenorizada dos mecanismos de comunicação

Como já foi demonstrado anteriormente, o modelo que minimiza os custos no encaminhamento de mensagens é Mec3. O ambiente de comunicações adoptado, foi pois baseado neste modelo.

Antes de iniciar a descrição do ambiente de comunicação, convém referir que o protocolo PCM, já introduzido neste mesmo capítulo não é respeitado no interior dos mecanismos de encaminhamento. Assim, se bem que nos modelos Mec1 e Mec2 as primitivas send e receive pressupõem uma comunicação do vector de dados entre a instância da aplicação Prg (*x*) e a instância dos mecanismos de comunicação E / S (*x*), no caso de Mec3 tal não é necessário. Realmente, o aumento de desempenho deste último em relação ao anterior deve-se precisamente a este facto.

Deste modo, para evitar transferências de dados desnecessárias, e consequentemente reduzir o tempo total despendido em comunicações, no modelo Mec3 o protocolo PCM só é utilizado nas mensagens entre as instâncias dos mecanismos de comunicação E / S (*x*), isto é entre nós diferentes, ao passo que a informação transferida internamente é composta pelo descritor e por um ponteiro para o vector de dados, em vez do próprio vector.

Quanto às primitivas send e receive, estas são o meio básico de interagir com os mecanismos de comunicação. Qualquer comunicação interna aos mecanismos de encaminhamento, usa as funções de comunicação já referidas anteriormente e relacionadas em termos de hierarquia na Fig. 3-4:

putmessage (*c1, p1, l1*)

getmessage (*c2, p2, l2*)

onde a primeira envia o vector apontado por *p1*, com o comprimento *l1*, pelo canal *c1* e a segunda recebe um vector de dados pelo canal *c2* e coloca-o na área de memória com o comprimento *l2* e apontada por *p2*. Estas duas funções assumem que os comprimentos dos vectores, *l1* e *l2*, são expressos em palavras de 4 *bytes*, que é o comprimento da mínima partícula de informação que pode ser comunicada num ADSP2106x ou num C4x. No entanto no T805 esta partícula tem o comprimento de apenas um *byte*, de modo que as duas funções acima, são também responsáveis por padronizar o comprimento da partícula de comunicação para 4 *bytes* independentemente do *hardware* utilizado.

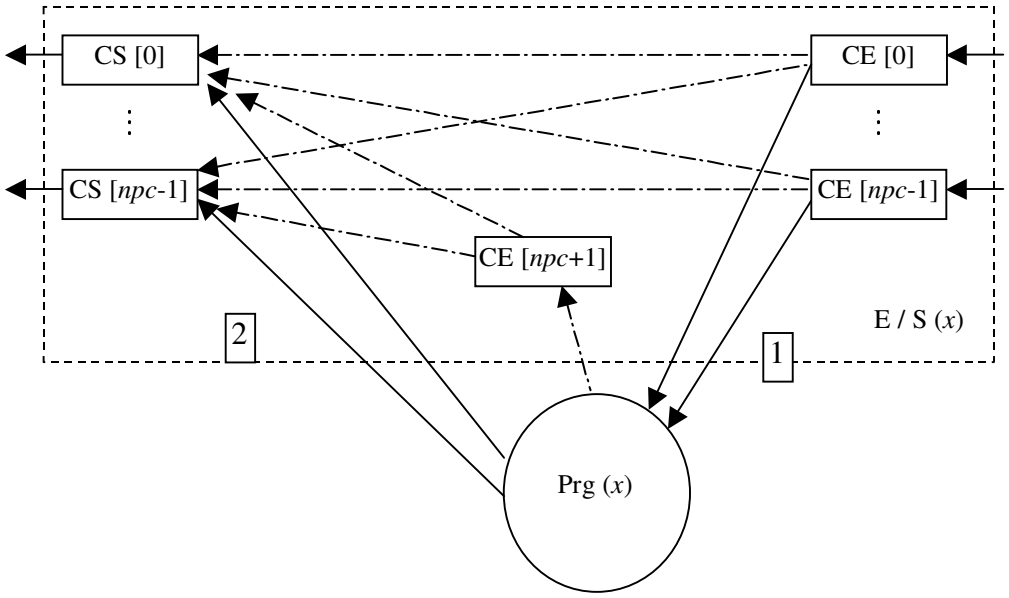


Fig. 4-16: Diagrama simplificado dos fluxos de dados no interior do ambiente de comunicação

A descrição do ambiente de comunicação será feita a partir do modelo Mec3, o qual está ilustrado na Fig. 4-4. Será pois descrito o bloco E / S (*x*), o mecanismo de comunicação alocado em qualquer nó, e a interacção deste com a instância da aplicação alocada no mesmo nó, o processo Prg (*x*).

O mecanismo E / S (*x*) é construído a partir de processos de controlo, os quais interactuam com os portos de entrada e de saída, isto é os *buffers* FIFO. Os processos que controlam a entrada aguardam que um descritor de mensagem seja fornecido pelo *buffer* FIFO ou pela

instância da aplicação Prg (*x*). De acordo com este e as tabelas de encaminhamento iniciam a transferência de informação entre os *buffers* FIFO e o segmento de memória atribuído a Prg (*x*). Se o destino de uma mensagem não for apenas o nó corrente, o processo que controla a entrada do porto por onde a mensagem foi recebida, indica aos processos que controlam a saída dos portos que dão acesso aos nós destino, que devem enviar uma cópia da mensagem recebida por um porto de entrada. Assim, assumindo que os *buffers* FIFO fazem parte dos processos de controlo, e referenciando estes processos por CS e CE no caso da saída e da entrada respectivamente, o diagrama simplificado acima ilustra esta estratégia.

Toda a mensagem com destino ao nó corrente e a outro nó é transferida primeiro entre o *buffer* FIFO do controlo de entrada e um segmento de memória de Prg (*x*). Depois, para poupar recursos de memória e tempo em transferências de informação internas desnecessárias, o controlo de entrada indica ao controlo ou aos controlos de saída correspondentes que devem ler a informação contida no próprio segmento de memória de Prg (*x*), e enviá-la pelo porto de comunicação que lhes está atribuído. No diagrama, o traçado contínuo indica o caminho que os dados seguem entre os portos de entrada ou saída e o segmento de memória de Prg (*x*), ao passo que o traçado intermitente representa os sinais de indicação entre os processos de controlo.

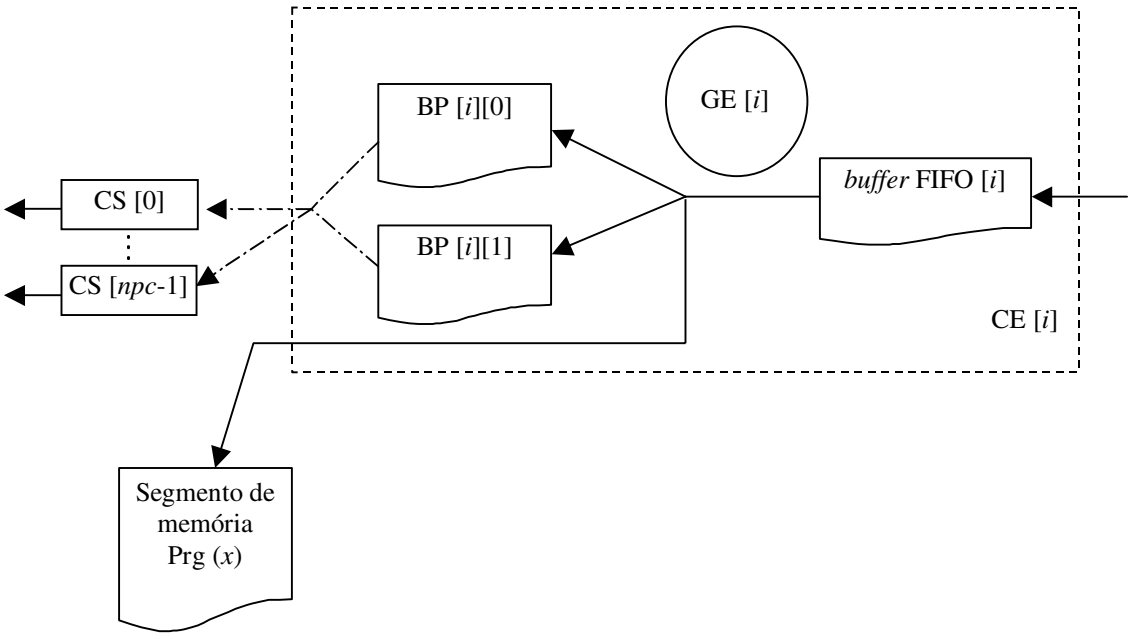


Fig. 4-17: Diagrama simplificado dos fluxos de dados no interior de um processo de controlo de entrada

Neste diagrama, está também representado o processo de envio de uma mensagem com origem num dado nó, isto é em Prg (*x*). Neste caso, o processo de entrada CE [*npc*+1] indica

aos processos de saída correspondentes o endereço de memória de Prg (*x*) onde devem ir buscar a informação a enviar para a rede.

Falta ainda analisar o caso em que uma mensagem não tem como destino o nó corrente, isto é se este for apenas um ponto de passagem no caminho entre o nó origem e o destino. Nestes casos não existe nenhum segmento de memória de Prg (*x*) disponível para guardar o vector passante. Isto obriga que seja alocada alguma memória para guardar temporariamente esse vector. Esta memória pode ser alocada nos processos de controlo como indica a figura anterior. Assim, uma mensagem passante pode ser temporariamente guardada no próprio processo de controlo de entrada, após o que este indica aos processos de controlo de saída correspondentes que devem enviar o vector guardado no *buffer* de passagem BP [*i*][0] ou BP [*i*][1], pelo porto que lhe está afecto.

Neste ponto convém referir que, em todos os processos de controlo de entrada, com excepção de CE [*npc*+1], são utilizados *buffers* duplos, de modo que em simultâneo, desde que o *hardware* o permita, um vector possa ser transferido entre o porto de entrada, ou seja o *buffer* FIFO [*i*] e o *buffer* de passagem 0, e outro entre o *buffer* de passagem 1 e o porto ou os portos de saída. Deste modo, explorando ao máximo o paralelismo existente no próprio *hardware*, pode-se minimizar o tempo perdido na passagem de uma mensagem através de um nó que não é destino.

É no entanto dispendioso em termos de recursos de processador, a existência de *buffers* com um comprimento igual ao comprimento máximo de uma mensagem segundo o protocolo PCM, que é de 64 KBytes. É pois necessário recorrer a *buffers* mais pequenos, que por defeito têm o comprimento de 1 *KByte*, obrigando uma mensagem de maior dimensão a ser segmentada em pacotes com este comprimento. Esta medida vai no entanto implicar uma perda de eficiência, tanto maior quanto menor for a dimensão do pacote, pois o tempo necessário para inicializar um canal, quer externo ou interno vai ser multiplicado pelo número de segmentos em que a mensagem será divida. Além disso, para sincronizar toda esta sequência de operações e decisões recorre-se a um sub-processo guarda ou GE [*i*], em cada entrada e GS [*j*] em cada saída, o que vai introduzir mais um pequeno atraso na comunicação quando comparado com o modelo óptimo Mec3 dado pelas equações (4-5) e (4-6).

É assim necessário proceder a uma implementação de uma rede comutada de pacotes, como é o caso das redes TCP/IP (Comer, 1988), embora os mecanismos de comunicação sejam desenvolvidos para um protocolo PCM mais simples, cuja interpretação introduz um atraso menor. Se bem que esta simplicidade do protocolo PCM, possa aparentemente traduzir-se nalguma perda da fiabilidade na entrega de pacotes, a qual é reforçada nas redes TCP/IP pelo

reenvio de pacotes perdidos, os mecanismos de comunicação são desenhados de modo que a perda de pacotes seja impossível, excepto em caso de mau funcionamento do *hardware*. Aliás este é também o caso da máquina virtual paralela ou PVM (Geist et al., 1994) e também do *interface* para passagem de mensagem ou MPI (Snir, et al., 1995) quando operando em modo padrão. No entanto o MPI, em modo síncrono, indica a recepção de um pacote devolvendo um sinal à origem.

Se uma falha de *hardware* suceder, como a versão corrente do SPAM não é tolerante a falhas, a aplicação tomará um comportamento imprevisível, mas muito provavelmente ficará bloqueada, sendo necessário reiniciá-la.

4.5.1 Tipos de pacotes

Como já foi descrito em 4.2, o protocolo PCM suporta pacotes com um máximo de 64 KBytes, de modo que o maior número de elementos que pode ser enviado num vector de inteiros ou números de vírgula flutuante é 16384. Assim, sempre que seja necessário no nível de programação enviar ou receber vectores com comprimento superior, e visto que as primitivas de comunicação send e receive estão perfeitamente de acordo com as especificações do protocolo já referido, terá de se recorrer não a apenas uma primitiva para efectuar a comunicação, mas sim a uma sequência destas. Esta limitação justifica-se para se obter a vantagem de encapsular todo o descritor dentro de apenas um inteiro de 32 *bits*, a partícula de comunicação básica dos mecanismos de encaminhamento. Este tipo de pacote é designado por pacote PCM.

Mas para além da limitação referida no parágrafo anterior, e como já tinha sido abordado no ponto anterior, uma mensagem pode não ter como destino um dado nó, sendo este apenas um ponto de passagem. Neste caso, para poupar memória, a mensagem é segmentada em pacotes menores com um número de elementos de 32 *bits* dado pela constante PACKET_LEN que por defeito tem o valor 256 isto é 1 KByte, e designados por pacotes passantes.

4.5.2 Sincronização do trafego de pacotes dentro do ambiente de comunicação

O fluxo de mensagens entre os portos de entrada e os portos de saída, podem ser vistos como uma requisição de um serviço efectuada por um cliente, o controlo de entrada, a um servidor, o controlo de saída. O serviço em causa será o envio de uma mensagem, que o cliente detém, pelo servidor.

Formulando então o algoritmo segundo uma arquitectura cliente-servidor (Berson, 1996), e generalizando para o caso em que um cliente pode pedir o mesmo serviço a vários servidores,

de modo que a mesma mensagem possa ser enviada para destinos diferentes, pode-se usar uma estratégia de fila de espera em cada servidor, tal que as requisições de serviço colocadas nessas filas pelos clientes sejam atendidas por ordem de chegada.

Consegue-se assim a sincronização do trafego de pacotes, e desde que as filas de espera estejam preenchidas, garante-se um aproveitamento máximo da largura de banda disponível, pois todos os portos de comunicação operam em simultâneo.

Aliás esta mesma estratégia é seguida pelo PVM, onde a operação de encaminhamento em cada nó pode ser vista como a sequência de 4 etapas principais (Geist et al, 1994):

i) Lê cabeçalho ou descritor de um pacote (pode provir da rede ou de tarefas locais)

ii) Se pacote provém da rede recebe-o num *buffer* com o comprimento máximo que o pacote pode ter. Se provém de tarefas locais aloca um *buffer* para o pacote com o comprimento deste.

iii) Replica o descritor e adiciona-o às filas de espera correspondentes, efectuando assim os pedidos de envios.

iv) Após o último pedido de envio ter sido atendido liberta o *buffer*.

Como se pode observar, *buffers* podem ser alocados e libertos quando da operação de encaminhamento de pacotes. Esta dinâmica, que também é seguida pelo MPI (MPI Forum, 1994), introduz um atraso no encaminhamento.

No SPAM, pelo contrário, toda a alocação de *buffers* é feita apenas uma vez na inicialização da aplicação, com um comprimento estático que será igual ao comprimento do maior pacote passante, que como já foi visto é dado por PACKET_LEN. No entanto, alterações a este comprimento só serão visíveis, após a recompilando da aplicação.

Estratégia semelhante segue o ambiente de encaminhamento de mensagens da 3L, *virtual channel router* (3L, 1995).

Neste ponto, é conveniente apontar que implementações de PVM e MPI podem tomar partido de serviços disponibilizados pelo sistema operativo, como é o caso de implementações para Unix, o que as simplifica. Aliás esta é uma directiva clara na referência do padrão MPI (MPI Forum, 1997): MPI não deve tomar controlo sobre aspectos da responsabilidade do sistema operativo, mas sim servir de *interface* limpo entre uma aplicação e o *software* de sistema.

Deste modo a comunicação entre processos poderá ser implementado recorrendo a serviços providos pelo sistema operativo, como é o caso de *sockets* BSD (Stevens, 1998) e para o caso

de processos executados no mesmo computador de mecanismos de comunicação através de memória partilhada (Stevens, 1999), também disponibilizados por sistemas operativos Unix.

No caso do SPAM, como este foi desenvolvido sobre um micro-núcleo que não providência serviços de tão alto nível, a sincronização de todos os processos que compõem os mecanismo de comunicação são da responsabilidade do SPAM, mais precisamente do próprio ambiente de comunicação.

Voltando a analisar o modelo cliente-servidor atrás apresentado, concluí-se que existem algumas particularidades na sua implementação que seria preferível evitar. Estas podem ser reveladas recorrendo à analise da operação do mecanismo de encaminhamento, a partir do momento em que uma mensagem chega a um controlo de entrada.

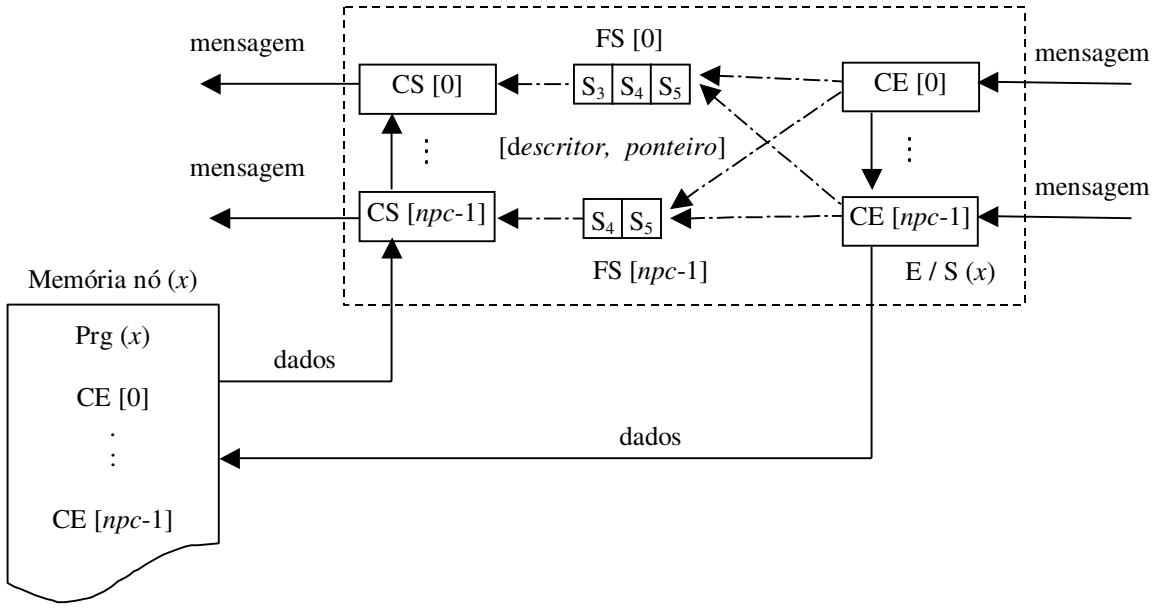


Fig. 4-18: Diagrama de fluxo de mensagens no interior do ambiente de comunicação, segundo uma abordagem cliente-servidor.

Neste ponto é conveniente referir que a notação utilizada nos diagramas está de acordo com a seguinte com a tabela:

notação	**Descrição**
⟶	Informação enviada por um canal
⟶	acesso directo à memória
{ ... }	{identificação do canal}
dado [dado , ... , dado]	Descrição da informação transferida

tabela 4-8: Notação utilizada nos diagramas de transferência de dados

Na estratégia representada na figura acima podem-se distinguir três fases de operação: recepção dos dados, pedido de serviço, e envio de dados ou atendimento do pedido de serviço. As duas primeiras fases são executadas pelo processo guarda de entrada. Como uma mensagem é composta por um descritor mais um vector de dados, GE [*i*] deve ler primeiro apenas o descritor e de acordo com este armazenar o vector de dados seguinte num endereço apropriado, que pode ser no segmento de memória de Prg (*x*) ou no próprio segmento de memória de CE [*i*]. Neste ponto termina a recepção da mensagem.

Seguidamente o pedido do serviço S_r é adicionado às filas de espera correspondentes aos portos de saída desejados, os quais são determinadas por GE [*i*], analisando o descritor e recorrendo às tabelas de encaminhamento **TEG** (*x*) e **TEP** (*x*). Este pedido de serviço consiste no descritor mais um ponteiro para a posição de memória onde na primeira fase foi armazenado o vector de dados.

Finalmente na terceira e última fase, CS [*j*] lê os pedidos de serviço da sua fila FS [*j*], e efectua o envio das mensagens correspondentes, isto é o descritor mais o vector de dados apontado pelo ponteiro incluído no pedido de serviço.

Aprofundando um pouco mais a implementação da segunda fase, verifica-se que a fila de pedidos pode ser criada recorrendo a alocação dinâmica de memória, de modo que sempre que um pedido é adicionado à fila de espera é alocada memória para este, e sempre que um serviço é atendido é libertada a memória ocupada pelo pedido correspondente. A fila pode pois ser implementada recorrendo a uma lista encadeada (Horowitz et al, 1993).

O problema com esta implementação é o tempo perdido na alocação e libertação de memória sempre que um pedido é efectuado. Na corrente implementação do SPAM, estas operações são executadas pelo *micro-kernel* 3L, já referenciado anteriormente, e introduzem um atraso cujo valor é variável.

Este atraso pode ser eliminado se a fila tiver uma dimensão definida à partida, e a colocação e atendimento de pedidos de serviços seja feita recorrendo a uma técnica de lista circular (Horowitz et al, 1993). Deste modo não existe necessidade de alocar espaço para novos pedidos, pois a partir de uma dada dimensão os pedidos mais recentes vão ser rescritos sobre os antigos. Mas como o número de pedidos que se podem colocar na fila circular é limitado, se por algum motivo o atendimento estiver atrasado em relação à colocação destes, chega-se a uma situação em que a fila está cheia. Neste caso, o guarda de entrada fica bloqueado à espera que a fila tenha uma posição livre para a colocação do pedido. Isto pode ser um inconveniente, se o processo guarda bloqueado ainda necessitar de colocar pedidos de serviços noutras filas, o que é muito provável que suceda numa difusão de mensagens, casos

já referidos em 4.3.2. Deste modo, um porto de comunicação mais lento, pode condicionar o atendimento de pedidos de tal forma que portos mais rápidos fiquem sem serviço durante algum tempo, não se aproveitando eficientemente a largura de banda.

Deve no entanto ser apontado que no caso de um difusão de todos para todos, referenciado em 4.3.2.2, este atraso estará sempre presente independentemente da estratégia utilizada, se para se efectuar a operação seguinte em Prg (*x*), for necessário distribuir os dados de cada nó por todos os outros. Nesta situação a continuação do processamento em Prg (*x*) terá de aguardar que todas as mensagens, vindas de cada nó, atinjam todos os nós. Como todas as comunicações são concorrentes, o tempo da distribuição será aproximadamente o tempo necessário para enviar as mensagens pelo maior caminho, isto é o mais lento.

Outro comportamento não desejável de uma fila implementada segundo um lista encadeada, e que também surge quando a taxa de transferência de dados não é idêntica em cada porto, tem a ver com o seu crescimento. Basta o caso mais simples de fluxo de mensagens no interior dos mecanismos de encaminhamento para revelar tal comportamento, desde que o porto de entrada tenha uma taxa de transferência superior ao de saída. Se a mensagem for constituída por vários pacotes, a fila de espera irá crescer tanto mais rapidamente quanto maior a diferença entre as taxas de transferência de ambos os portos, podendo ocupar toda a memória disponível. Este efeito pode ser eliminado recorrendo a uma fila circular finita, já acima referida. Mas neste caso, mais uma vez se a fila estiver cheia, o processo controlo de entrada deve aguardar que esta esteja disponível.

É assim desejável redesenhar esta estratégia de modo que os processos controlo de entrada nunca fiquem bloqueados, e que assim os processos controlo de saída sejam continuamente alimentados com pacotes de dados. Para atingir esse objectivo a arquitectura cliente servidor acima descrita será abandonada, no sentido em que o cliente envia o pedido para o servidor. Pelo contrário esta segunda estratégia implementa um servidor que continuamente verifica se algum cliente tem alguma mensagem para enviar.

Fazendo uma analogia da estratégia anterior com um serviço de correio, o cliente dos correios, isto é o processo de controlo de entrada, é responsável por introduzir na mala postal correcta, isto é o processo controlo de saída, uma mensagem com um certo endereço, e que fica aguardando a vez de ser despachada numa fila de espera. Se a caixa postal está cheia, o cliente tem que aguardar nessa caixa que o posto de correios arranje espaço para alojar mais uma mensagem, mesmo que tenha mensagens para outras caixas.

Por outro lado a nova estratégia poderá ser representada por um conjunto de carteiros, um para cada mala postal, que quando solicitados percorrem as caixas de correio dos clientes à

procura de mensagens, e que as colocam nas malas postais correspondentes aos endereços, libertando o cliente dessa tarefa.

É assim necessário introduzir o conceito de caixa de saída ou CDS [*i*], estrutura de dados que é associada a cada controlo de entrada CE [*i*]. É nesta caixa, que após ser recebido um pacote, o cliente coloca o descritor respectivo e um ponteiro para os dados desse pacote. São também seleccionadas, de um conjunto de 32 bandeiras, as que correspondem aos índices dos processos controlo de saída que devem encaminhar o pacote em causa. Ou utilizando ainda a analogia anterior, o carteiro após ter recolhido uma cópia da mensagem numa dada caixa de saída, baixa a bandeira corresponde à mala postal onde colocou a mensagem, de modo que se voltar a passar por ela antes que o cliente substitua a mensagem por uma mais recente não tenha de voltar a verificá-la. Só após a mensagem corrente ter sido enviada é que o cliente pode colocar uma nova mensagem. Mais especificamente, a estrutura de cada CDS [*i*] é:

bandeiras	: 32 *bits* (1 bit para cada porto de saída)
descritor	: 32 *bits* (segundo o protocolo PCM)
ponteiro	: 32 *bits*

É importante salientar que nesta implementação, visto que apenas existem 32 bandeiras, e é necessário associar uma a cada porto de saída, o número máximo de portos de saída suportados em cada nó é 32, número este suficiente para os processadores suportados pelo SPAM 1.0 e para a generalidade dos processadores comerciais até à data. No entanto o sistema poderá no futuro ser actualizado para suportar um maior número de portos de saída, simplesmente adicionando tantos conjuntos de 32 bandeiras quanto as necessárias.

A Fig. 4-19 representa esquematicamente todo o tratamento de mensagens num nó, segundo esta última estratégia. Todo o funcionamento dos mecanismos de comunicação podem ser retirados desta figura. Como já foi descrito anteriormente, é assumido que os processos que em cada porto de comunicação controlam a entrada e a saída, CE [*i*] e CS [*j*] respectivamente, são constituídos por um sub-processo guarda de entrada ou de saída, e *buffers* FIFO por onde são recebidos os dados ou onde são colocados os dados a enviar. Além desta estrutura, como ilustra a Fig. 4-17, os processos CE [*i*] dispõem ainda de dois *buffers* de passagem, para armazenamento temporário de pacotes passantes.

No entanto o processo de controlo CE [*npc*+1], o qual gere o fluxo de dados proveniente da instância Prg (*x*), não necessita de qualquer tipo de *buffer* e é apenas composto por um processo guarda, de modo que referências a CE [*npc*+1] e GE [*npc*+1] são equivalentes.

Além destes processos, no caso do nó 0, existem dois processos extras para implementar a comunicação entre este nó e o *host interface*, CE [*npc*] e CS [*npc*]. Assim os processos guarda, responsáveis pela lógica de encaminhamento de mensagens, são de três tipos e não apenas dois. Os tipos GS [*j*] e GE [*i*] onde $i \neq npc+1$ e finalmente GE [*npc*+1]. Estes algoritmos serão introduzidos analisando os trajectos possíveis dos pacotes no ambiente de comunicações.

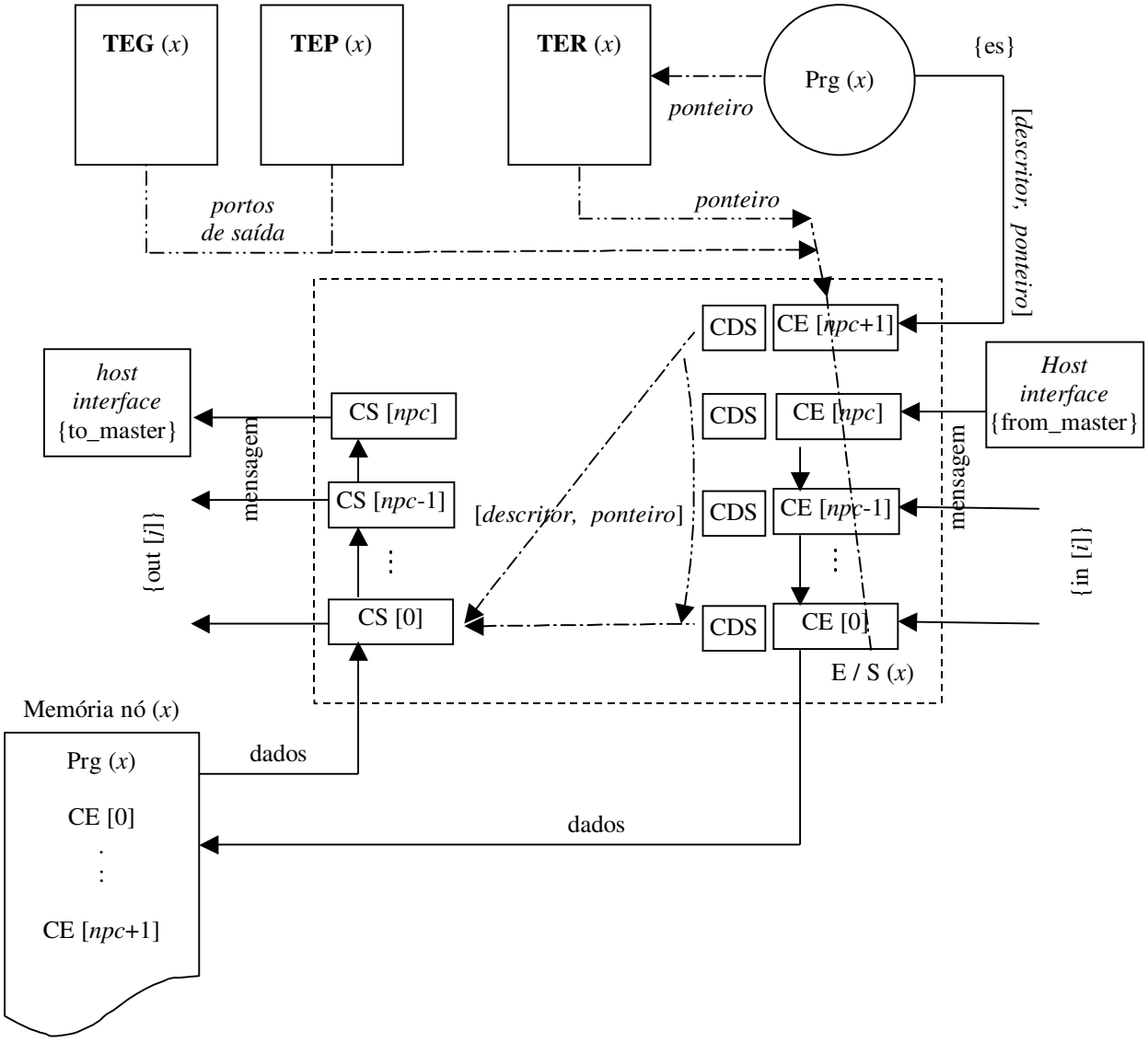


Fig. 4-19: Diagrama completo de fluxo de mensagens num dado nó

Qualquer trajecto inicia-se na primitiva send e termina na primitiva receive. Estas primitivas operam de modo diferente, dependendo da origem ou destino da mensagem. Seguidamente será descrito o caso em que a origem e destino da mensagem são as instâncias da aplicação Prg (*x*). As diferenças para o caso em que a origem e o destino são o *host interface* serão abordado mais à frente.

Antes de pedir ao ambiente de comunicações para enviar uma mensagem, a primitiva send deve criar o descritor, de acordo com os campos já descritos na tabela 4-6. Depois deve enviar este seguido de um ponteiro (de 32 *bits*) para o vector de dados, através do canal es para o processo CE [*npc*+1], o qual pode ser visto como a porta de saída de mensagens provenientes de Prg (*x*). O algoritmo para send no contexto de Prg (*x*), expresso à custa das funções de comunicação putmessage e getmessage é apresentado a seguir:

algoritmo 4-1:

```
send (descritor, dados, espera)
início
se descritor.MSG_SRC ≠ descritor.MSG_DST
        putmessage (es, descritor, 1)
        ponteiro = endereço de dados
        putmessage (es, ponteiro, 1)
        se espera = verdadeiro wait (SFE)
fim
```

algoritmo 4-2:

```
send_nowait (descritor, dados, espera)
início
        send (descritor, dados, falso)
fim
```

Para simplificar o algoritmo foi assumido que um descritor adequado à mensagem a enviar é fornecido à primitiva send. No entanto tal não sucede. De facto esta primitiva é responsável por construir o descritor a partir do tipo de mensagem sua origem e destino, segundo o protocolo PCM. Durante a construção do descritor é verificado se o comprimento da mensagem é inválido, o que fará com que a mensagem não seja enviada. Por outro lado o envio de uma mensagem para o próprio nó de origem não é permitido, ao contrário de outros ambientes, tal como o MPI. Se tal suceder a mensagem será também descartada pela primitiva, nunca chegando aos mecanismos de comunicação. O mesmo sucederá se a mensagem for enviada para um destino que não existe na rede.

Assumindo que destinos negativos não existem e que o número de nós numa dada rede é dado pela constante N_WRK, os destinos possíveis são apenas: $0 \leq x <$ N_WRK e $x =$ DST_MASTER. Este último destino, o *host interface*, não está previstos nas tabelas de encaminhamento de mensagens, pois estas apenas se referem às instancias da aplicação. Mas como o *host interface* tem uma ligação ao nó 0 isto é a E / S (0), as mensagens que lhe são destinadas podem ser enviadas pelos mesmos portos usados para aceder ao nó zero, e só em

E/S (0) será feita a separação das mensagens destinadas a Prg (0) e ao *host_interface*. Esta operação pode ser traduzida num regra que os mecanismos de comunicação respeitem:

regra 4-2: Mensagens cujo destino é DST_MASTER são enviadas para o porto de comunicação dado por TEP $(x)_{0,x}$, se $0 < x <$ W_NRK. Se $x = 0$ são enviadas pelo canal de ligação ao *host_interface*.

Após a construção e validação do descritor, este é enviado para os mecanismos o descritor, bem como um ponteiro para os dados a enviar. A variável *espera* é um inteiro de 32 *bits* que toma o valor numérico 0 para representar o valor lógico falso e qualquer outro valor numérico para o valor lógico verdadeiro. Se *espera* for verdadeiro, Prg (*x*) deve aguardar que a mensagem seja despachada para todos os destinos. Caso contrario, a instância da aplicação pode continuar o processamento enquanto a mensagem é enviada. Assim, relativamente ao fluxo de processamento da instância da aplicação Prg (*x*), donde é chamada, esta primitiva tem dois modos de operação: um bloqueante e outro não bloqueante, como também é o caso nos ambientes PVM e MPI.

No caso não bloqueante, embora o desempenho aumente, nada impede Prg (*x*) de modificar o vector de dados enquanto este está a ser enviado, o que põe em risco a integridade da informação contida na mensagem. Assim, só quando existir garantia que a mensagem é enviada antes que Prg (*x*) modifique o vector de dados, é que pode ser utilizado a versão não bloqueante de send, send_nowait. De qualquer modo, mesmo no caso não bloqueante, pode-se sempre fazer com que Prg (*x*) aguarde que a mensagem seja enviada, indicando no código explicitamente para aguardar no semáforo SFE, em qualquer ponto após a chamada a send_nowait.

Finalmente send, e consequentemente Prg (*x*), aguardam no semáforo SFE que a mensagem seja enviada por todos os portos de saída. Por outro lado, send_nowait espera apenas que o descritor e o ponteiro sejam enviados para CE [*npc*+1] através do canal es. Assim a sincronização entre os vários processos é feita recorrendo a dois semáforos para Prg (*x*), nomeadamente o semáforo de fim de emissão, SFE e o semáforo de fim de recepção SFR, e a um semáforo para cada processo de controlo. Estes últimos semáforos são agrupados em dois vectores, SCE e SCS, respectivamente referentes aos processos de controlo de entrada e de saída. A sincronização de processos nestes semáforos é efectuada recorrendo a duas primitivas:

signal (*semáforo*, *n*)

wait (*semáforo*, *n*)

A primeira assinala, isto é incrementa o contador associado ao *semáforo* em *n* unidades, ou, se esta variável for omitida, em apenas uma unidade. A segunda obriga um processo a esperar no *semáforo*, que o contador seja decrementado *n* vezes, ou se este for omitido, apenas uma vez.

Quanto ao processo GE [*npc*+1], após receber o descritor e o ponteiro, opera de dois modos distintos de acordo com o destino da mensagem. Se a mensagem tiver como destino todos os nós, é usada a função tegx (*idx*), para levantar as bandeiras (cada *bit* toma o valor 1) de acordo com os elementos não nulos de **TEG** $(x)_{idx,*}$ e coloca o descritor, o ponteiro para o vector de dados e as bandeiras na caixa de saída. Seguidamente assinala os semáforos SCS correspondentes aos portos de saída indicados em **TEG** $(x)_{x,*}$ e aguarda no semáforo SCE [*npc*+1] que os processos controlo de saída seleccionados terminem de enviar a mensagem.

Mas se o destino da mensagem for um nó específico é necessário partir a mensagem em pacotes com o comprimento PACKET_LEN, isto é o comprimento dos *buffers* de passagem. Para cada um desses pacotes será necessário levantar a bandeira correspondente ao porto de saída dada a origem *idx*, isto é o *bit* dado pelo valor da linha *idx* de **TEP** (*x*) usando a função tepx (*idx*); reformular o descritor e actualizar a CDS[*npc*+1]. Após assinalar o semáforo SCS [**TEP** $(x)_{x,0}$], correspondente ao porto de saída indicado em **TEP** $(x)_{x,0}$, aguarda no semáforo SCE [*npc*+1] que o processo controlo de saída seleccionado termine de enviar o pacote. Assim a mensagem é partida em pacotes que são enviados sequencialmente para o nó destino. Finalmente, assinala o semáforo SCE [*npc*+1].

Qualquer que seja o caso o semáforo SFE é sempre incrementado.

algoritmo 4-3:

```
GE [i]                                          "com i = npc+1 = MAX_LINKS+1"
início
ciclo infinito
        getmessage (es, descritor, 1)
        getmessage (es, ponteiro, 1)
        len = descritor.MSG_LEN
        ti2ieee (descritor, ponteiro, len)
        Se descritor.MSG_DST = DST_MASTER  idx = 0                  "regra 4-2"
        Senão idx = descritor.MSG_DST                  "neste caso MSG_SRC=x"
        se descritor.MSG_DST = DST_ALL
                bandeiras = tegx (descritor.MSG_SRC)
                coloca na CDS [npc+1] descritor, ponteiro, bandeiras
```

```
            signal (SCS [b])      "onde b são os índices das bandeiras levantadas"
            wait (SCE [npc+1], n)                  "onde n é o número de bandeiras
                                                                levantadas"
        senão
            descritor.MSG_LEN = PACKET_LEN
            descritor.MSG_END = 0
            desde p = 0 até len-1 por PACKET_LEN
                bandeiras = tepx (idx)
                se len - p ≤ PACKET_LEN      "cria descritor para último pacote"
                    descritor.MSG_LEN = len – p + 1
                    descritor.MSG_END = 1
            coloca na CDS [npc+1] descritor, ponteiro, bandeiras
            ponteiro = ponteiro + PACKET_LEN.
            signal (SCS [b])              "onde b é o índice da bandeira levantada"
            wait (SCE [npc+1])
        ieee2ti (descritor, ponteiro, len)
        signal (SFE)
fim
```

As macros ieee2ti (*d*, *v*, *l*) e ti2ieee (*d*, *v*, *l*) verificam se o vector *v*, descrito por *d*, é composto por *l* números de vírgula flutuante, e se for o caso convertem-no entre a norma IEEE 754 e a representação interna do C4x e vice-versa. Estas macro só produzem trabalho útil nos mecanismos de comunicação alocados num C4x, quando integrado numa rede heterogénea com Transputers. Estas conversões são discutidas mais à frente neste capítulo.

Na implementação será necessário traduzir portos físicos na faixa [0, MAX_LINKS+1] em portos lógicos na faixa [0, *npc*+1], o que pode ser efectuado pela tabela **PT**. No entanto para simplificar a implementação, estruturas com índice *npc*, que constituem dispositivos de comunicação com o *host interface*, e *npc*+1, entrada de dados no ambiente de comunicações provenientes de Prg (*x*), são referenciados com índices MAX_LINKS e MAX_LINKS+1 respectivamente. Sempre que necessário, para clareza da implementação, esta informação foi incluída nos algoritmos em forma de comentário.

Os processos CS são os responsáveis por despachar as mensagens pelos portos de saída. O processo guarda de saída GS [*j*], consiste num ciclo infinito que espera no semáforo associado SCS [*j*] que algum GE [*i*] o assinale, indicando assim que existe uma mensagem para ser enviada pelo porto de saída *j*. GS [*j*] inicia então uma busca ordenada por todas as caixas CDS [*i*], até encontrar uma cuja bandeira *j* esteja levantada. Quando a encontra lê o descritor e o ponteiro armazenados nela, e baixa a bandeira *j*. Seguidamente envia a mensagem para a rede pelo porto de saída associado e assinala o semáforo SCE [*c*] indicando que foi terminada a difusão a partir deste nó.

Assumindo que a função bandeira (*b*, *i*) retorna verdade se a bandeira índice *b* da CDS [*i*] está levantada e a função baixa (*b*, *i*) baixa a bandeira índice *b* da CDS [*i*], isto é coloca no *bit* corresponde o valor 0, o seguinte algoritmo descreve GS [*j*]

algoritmo 4-4:

```
GS [j]
início
se (x=0)  nc = npc+2
senão nc = npc+1
c = 0
ciclo infinito
        wait (SCS [j])
        enquanto bandeira (j, PT[c]) = falso
                c = c + 1
                se (c > nc )  c = 0
        lê descritor e ponteiro de CDS (PT[c])
        baixa (j, PT[c])
        putmessage (out [j], descritor, 1)
        putmessage (out [j], ponteiro, descritor.MSG_LEN)
        signal (SCE [PT[c]])
fim
```

A ordem com que é feita a busca de caixas de saída com mensagens garante que não existe privilégio a atender portos de entrada mais rápidos, como é o caso do porto interno donde provém as mensagens de Prg (*x*), pois após um dado CS [*j*] ter atendido um CE [*i*], quando for assinalado uma nova mensagem para o mesmo CS [*j*] a busca inicia-se no CE seguinte isto é CE [*i*+1]. Esta ordem também garante que no caso de mais de um CE assinalar o mesmo CS [*j*], os portos de entrada são atendidos independentemente da ordem com que assinalaram o semáforo SCS [*j*], de modo que não existe nenhum CE com atendimento prioritário.

No entanto na inicialização da aplicação, o início da busca é feito a partir do CE (*npc*) e de CE *(npc+1)* no caso do nó 0, de modo que a primeira vez que um semáforo SCS é assinalado é verificado se existe alguma mensagem nas fontes, isto é em Prg (*x*) e no *host interface*. Embora não faça nenhuma diferença em termos de lógica do algoritmo, é mais provável que o porto de entrada que conterá a primeira mensagem seja um destes.

É conveniente também apontar que após a inicialização da aplicação todas as bandeiras em todas as caixas são baixadas.

Para introduzir o algoritmo do guarda de entrada será analisada a recepção de uma mensagem. Mais uma vez tudo se inicia em Prg (*x*) usando a primitiva receive. Neste contexto esta primitiva tem o seguinte algoritmo.

algoritmo 4-5:

receive (*descritor*, dados)
início
TER (*x*) *descritor*.MSG_SRC, 0 = endereço de dados
TCM [*descritor*.MSG_SRC] = 0
signal (SIR[*descritor*.MSG_SRC])
fim

Para receber uma mensagem é necessário uma chamada a receive. Assim esta primitiva não pode ser bloqueante, para que possam ser iniciadas várias recepções de mensagens, com origens diferentes, concorrentemente. Mas para garantir que o acesso aos dados recebidos só é efectuado depois da conclusão da recepção, após *v* pedidos receive, de *v* fontes diferentes, Prg (*x*) deve esperar no semáforo SFR, *v* vezes, usando uma instrução wait (SFR, *v*).

Cada chamada a receive deve também especificar o endereço onde deve ser guardada a mensagem. Para permitir recepções concorrentes, em cada nó *x* existe uma tabela de endereços de recepção, **TER** (*x*), que consiste num vector coluna com tantas entradas como o número de nós da rede mais o nó raiz, isto é MAX_WRK+1. No índice de linha correspondente ao número do nó origem, receive coloca um ponteiro para o endereço onde a mensagem proveniente dessa origem deve ser guardada. Para retornar o comprimento da mensagem recebida é usado o vector **TCM** (*x*), com a mesma dimensão do anterior, e onde após a recepção é colocado o comprimento da mensagem recebida em palavras de 32 *bits*.

Finalmente para indicar ao ambiente de comunicações que pode iniciar a recepção da mensagem, assinala no vector de semáforos de início de recepção, SIR, o semáforo cujo índice corresponde ao nó origem.

$$\text{(4-13)}\quad \mathbf{TER}(x) = \begin{bmatrix} \mathbf{TER}(x)_{0,0} \\ \vdots \\ \mathbf{TER}(x)_{\text{SRC\_MASTER},0} \end{bmatrix}$$

A recepção da mensagem é então tratada pelos guardas de entrada de cada porto. Neste algoritmo é assumido que a função switch_buffer ( ) aguarda a disponibilidade do *buffer* de passagem 0 ou 1, recebe nele uma mensagem e retorna o endereço do *buffer* onde a guardou.

algoritmo 4-6:

```
GE [i]                                                    "com i ≠ npc+1 ≠ MAX_LINKS+1"
início
ciclo infinito
        getmessage (in [i], descritor, 1)
        Se descritor.MSG_DST = DST_MASTER  idx = 0                  "regra 4-2"
        Senão idx = descritor.MSG_DST
        se descritor.MSG_DST = DST_ALL ou x                         "regra 4-1"
                wait (SIR [descritor.MSG_SRC])
                ponteiro = TER (x) descritor.MSG_SRC, 0
                TCM (x) descritor.MSG_SRC, 0=TCM (x) descritor.MSG_SRC, 0 +descritor.MSG_LEN
                bandeiras = tegx (descritor.MSG_SRC)
                getmessage (in [i], ponteiro, descritor.MSG_LEN)
        senão
                ponteiro = switch_buffer ( )
                bandeiras = tepx (idx)

        se descritor.MSG_DST = x                                                  "poço"
                TER (x) descritor.MSG_SRC, 0=TER (x) descritor.MSG_SRC, 0 + descritor.MSG_LEN
                se descritor.MSG_END signal (SFR)
                senão signal (SIR [descritor.MSG_SRC])
        senão
                se bandeiras não nulo                                "reencaminhamento"
                coloca na CDS [i] descritor, ponteiro, bandeiras
                signal (SCS [b])                "b são os índices de bandeira levantados"
                        se descritor.MSG_DST = DST_ALL
                                wait (SCE [npc+1], nb)                  "nb é o número de
                                                                       bandeiras levantadas"
        se descritor.MSG_DST = DST_ALL ou x
                ieee2ti (descritor, ponteiro, descritor.MSG_LEN)
        signal (SFR)
fim
```

No caso de envio de mensagens a partir do próprio nó existem sempre destinos possíveis, pois tal já é validado pela primitiva send. No entanto, quando da recepção de mensagens que provém doutro nó, poderá não existir destino possível sendo o nó corrente o destino final da mensagem ou pura e simplesmente um poço ou extremidade do grafo de difusão de mensagens dado por **TEG**(0 .. N_WRK-1). Assim, ao contrário do algoritmo 4-3, este último deve verificar se existe algum destino para cada mensagem que chega, antes de despoletar o envio destas. No caso de uma difusão, esta validação é equivalente a verificar se *bandeiras* tem um valor nulo.

A implementação da **regra 4-1** é garantida por este último algoritmo e pela primitiva de comunicação send. Ao passo que a primitiva garante que não podem ser enviadas mensagens para o próprio nó de origem, este algoritmo, quando de uma difusão, ou seja se

descritor.MSG_DST = DST_ALL, encarrega-se de deixar uma cópia da mensagem no nó corrente, antes de a difundir para os outros destinos dados por **TEG** (x).

Por outro lado, a implementação do **Axioma 4-2** e do **Axioma 4-3**, é efectuada nos guardas de entrada, algoritmo 4-3 e algoritmo 4-6, por consulta das tabelas de encaminhamento de mensagens **TEG** (*x*) e **TEP** (*x*) respectivamente. Estes algoritmos também respeitam a regra 4-2.

Quanto à conversão entre representações diferentes de números de vírgula flutuante, se tal for necessário, a conversão de uma mensagem que chegou apenas é efectuada sobre a cópia armazenada no nó corrente e após cópias dessa mensagens terem sido enviadas para todos os destinos possíveis.

É ainda significativo que este algoritmo assume que uma mensagem com destino: todos os nós, e proveniente do processo de controlo é constituída por um pacote com comprimento máximo PACKET_LEN. Tal simplifica consideravelmente o algoritmo e tem razão de ser, pois como o processo de controlo é visto como um ponto de passagem entre o anfitrião e a rede, tem *buffers* de memória para guardar apenas PACKET_LEN dados, não sendo por isso possível apontar um endereço com comprimento superior a PACKET_LEN no *host_interface.*

Para completar a descrição da operação dos guardas de entrada falta abordar a operação do duplo *buffer*. Como já foi referido anteriormente, esta estratégia permite aumentar o desempenho da transferência de dados entre dois nós isolados, permitindo que num nó intermédio, os pacotes que constituem a mensagem sejam simultaneamente recebidos e enviados com o desfasamento de um pacote.

Em termos de SPAM os dados são distribuídos por todos os nós da rede, de modo que as operações de comunicação necessárias para executar um dado algoritmo, normalmente implicam que todos os nós troquem informação entre si, caso da difusão descrita no ponto 4.3.2.2. Assim um nó terá de enviar dados locais por um dado porto enquanto recebe dados remotos por outro porto qualquer, sendo assim toda a largura de banda aproveitada. Além disso nos casos de difusão, os dados são armazenados directamente na memória da instância da aplicação alocada em cada nó e a partir daí podem ser enviados por outro porto de comunicação. Neste caso, não existe necessidade de recorrer à estratégia de duplo *buffer*, pois a memória de Prg (*x*) pode ser vista como um *buffer* múltiplo, já que existirá sempre memória reservada para os dados que lhe são destinados. O aumento de desempenho será sim visível nas transferências entre nós individuais, já abordadas no ponto 4.3.1, quando uma mensagem tem de atravessar vários nós para chegar ao seu destino, e terá de ser temporariamente armazenada nos nós intermédios. Já que a mensagem não é destinada a esse nós intermédios,

não existe memória reservada nas instâncias da aplicação correspondentes, tendo os pacotes de ser armazenados em qualquer lugar no ambiente de comunicações.
Utilizando apenas um *buffer* em cada nó, o tempo necessário para enviar uma mensagem com *n* pacotes, desprezando o *descritor*, através de um caminho com *h* nós entre a origem e o destino, é dado por:

(4-14) $T_{bs} = Q_{cp} (h\text{-}1 + 2(n\text{-}1))$

Onde Q_{cp} é assumido constante e representa o tempo para enviar um pacote com um dado comprimento entre quaisquer dois nós contíguos, como é o caso de uma rede homogénea.
De facto a memória para receber o pacote seguinte só ficará disponível após o pacote corrente ter sido enviado para um qualquer destino, sendo isso denotado pela parcela $2(n\text{-}1)$.
No entanto se se usar uma estratégia de duplo *buffer*, o tempo para enviar uma mensagem entre quaisquer 2 nós, é dado por:

(4-15) $T_{bd} = Q_{cp} (h\text{-}1 + n\text{-}1)$

Dado que a recepção e o envio de um pacote é efectuado simultaneamente em cada buffer, a parcela dependente de *n* é agora apenas $n\text{-}1$. Deste modo, quanto maior for o número de pacotes a que compõem a mensagem maior será o aumento de desempenho do duplo *buffer*.
Este tempo de comunicação no entanto poderá ser aumentado se existirem colisões de pacotes, o que pode suceder se existir comunicação entre mais que um par de nós em simultâneo, onde os caminhos de comunicação entre dois nós contém nós comuns. Assim no caso de uma rede irregular o tempo degradar-se-á em função do número de colisões, o qual depende fortemente da topologia da rede.
No caso de redes heterogéneas a estratégia de duplo *buffer* também reduz o tempo de latência na transmissão de pacotes, mas não por um factor de dois como no caso heterogéneo, pois isso dependerá dos custos de comunicação ao longo da rede. O que se poderá afirmar é que o tempo de passagem de um pacote em cada nó será reduzido para o máximo entre o tempo de recepção e o de envio, em vez de ser a soma de ambos.
Para tornar a utilização do duplo *buffer* transparente para o guarda de entrada, é utilizada a função switch_buffer ( ), como já foi atrás indicado. Nesta função, cujo algoritmo é apresentado a seguir, a variável BP representa as áreas de armazenamento temporárias denominadas por *buffers* de passagem existentes em cada nó. Esta toma as dimensões

MAX_LINKS × PACKET_LEN × 2, pois por cada porto de entrada existe um duplo *buffer* com o comprimento de PACKET_LEN palavras. Assume-se também que as variáveis *buffers_livres* e *buffer_corrente* são dois vectores com MAX_LINKS elementos, os quais tomam o valor 1 na incialização da aplicação. Estes indicam para cada porto, se ambos os *buffers*, do duplo *buffer*, estão cheios e qual dos dois está pronto para receber dados, respectivamente.

algoritmo 4-7:

```
switch_buffer (i, l)
início
se buffer_corrente [i] = 0 buffer_corrente [i] = 1
senão buffer_corrente [i] = 0

getmessage (in [i], BP [i][ buffer_corrente [i] ], l )

se buffers_livres [i] = 0 wait (SCE [i])
senão buffers_livres [i] = buffers_livres [i] - 1

devolve endereço de BP [i][ buffer_corrente [i] ]
fim
```

A espera no semáforo SCE [*i*], garante que um *buffer* só fica pronto para receber dados após o guarda de saída ter despachado um eventual pacote armazenado nesse *buffer*.

4.5.2.1 Conversão de dados

Como já foi indicado acima, a conversão de dados para redes heterogéneas T805-C40 é efectuada pelas macros ieee2ti e ti2ieee que operam sobre vectores de números de vírgula flutuante. Como a versão actual do SPAM suporta apenas diferenças na representação de dados entre a representação de números de vírgula flutuante de 32 *bits*, segundo a norma IEEE 754, a qual é a adoptada pela maior parte das famílias de processadores comerciais, e o formato interno de vírgula flutuante do C4x, ou formato C4f para abreviar, as macros acima mencionadas são relativamente simples de implementar e de rápida execução. No entanto, se se pretender generalizar o ambiente de comunicações para suportar qualquer tipo de nó, será conveniente usar uma norma de transporte de dados comum, mais geral, que suporte conversões entre outros tipos de dados, como a XDR. De facto, esta norma é usada para transportar dados entre nós no âmbito do PVM. No caso do ambiente MPI, esta norma já é opcional, podendo ser utilizadas funções de conversão especificas para o tipo de nós utilizado, como é o caso do SPAM.

No caso do SPAM a conversão de dados é efectuada entre uma instância Prg (*x*) alocada num C4x e o ambiente de comunicações E / S (*x*) pelos guardas de entrada. Se bem que não seja obrigatório, existe toda a conveniência em efectuar as conversões num C4x, pois este processador dispõe de instruções em código nativo dedicadas para tal. Deste modo as conversões serão efectuadas o mais eficientemente possível.

Assim os dados de vírgula flutuante são transportados no formato IEEE 754 e só em instâncias da aplicação alocadas em C4x é que tomam o formato C4f, como mostra a seguinte figura.

No entanto esta estratégia de conversão implica que os números de vírgula flutuante transferidos entre dois C40 tem de ser convertidos entre os formatos C4f e IEEE 754 na origem e efectuar a conversão contrária no destino. Esta limitação é necessária, pois no caso de difusão de mensagens numa rede heterogénea, estas serão enviadas para processadores de famílias diferentes, de modo que é obrigatório que a mensagem seja transportada num formato que poderá ser descodificado por qualquer processador, tendo sido adoptado é o formato IEEE 754.

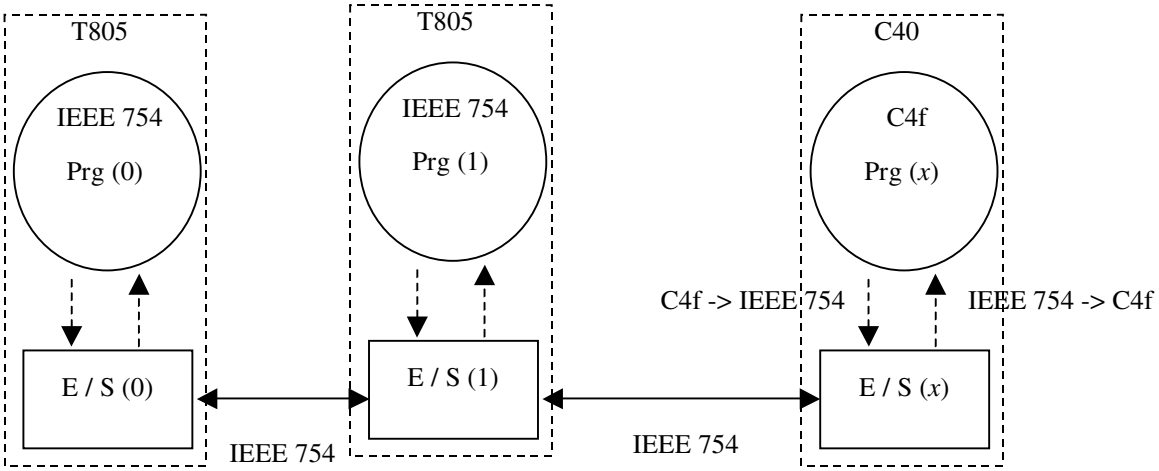


Fig. 4-20: Formato dos números de vírgula flutuante ao longo da rede

A implementação desta estratégia deve ser feita de modo a poupar os recursos do processador, bem como simplificar o código dos mecanismos de encaminhamento, de modo a evitar atrasos desnecessários nas comunicações.

Quanto à conversão para o formato C4f, é efectuada após o vector de dados ter sido recebido e colocado na área de memória que lhe está destinada, elemento a elemento, no próprio lugar, sem necessidade de recorrer a variáveis auxiliares.

Para simplificar a conversão contrária, isto é de um vector que será enviado de um C4x para a rede, pode-se recorrer a um algoritmo semelhante, ou seja efectuar a conversão do vector de dados para o formato IEEE 754, no lugar, e só depois enviar este para a rede. No entanto, muito provavelmente o vector armazenado no C4x será usado no futuro. Isto vai implicar que os elementos desse vector terão de ser transformados novamente na representação C4f. Tem-se assim uma conversão extra, que irá demorar o seu tempo. Seguidamente encontra-se representada a sequência de operações e o tempo requerido por este algoritmo:

Operação	**Tempo de execução (μs)** por número de vírgula flutuante
conversão de C4f para IEEE 754	1,994
Comunicação externa uni-direccional T8-C40 (Qe_u)	4,896
conversão de IEEE 754 para C4f	1,731
Tempo total por número de vírgula flutuante	**8,621**

algoritmo 4-8: Envio de um vector de números flutuantes para uma rede heterogénea a partir de um C40

Como indicado, o tempo perdido na reconversão do vector para o formato C4f é de 1,731 μs por cada número de vírgula flutuante, o que representa um acréscimo de 25% no custo da comunicação.

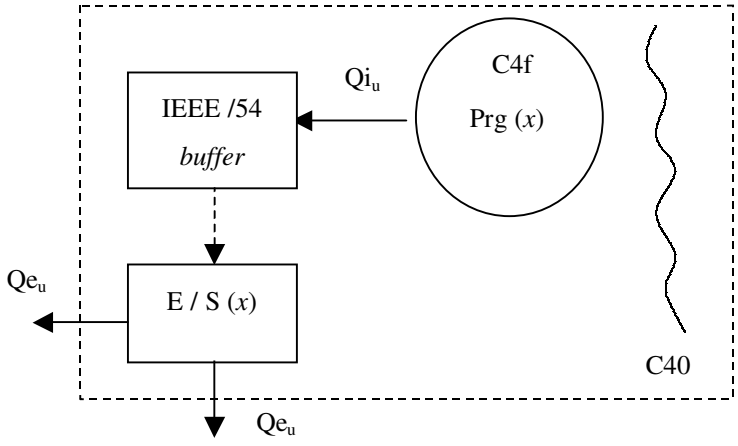


Fig. 4-21: Conversão do entre formatos de vírgula flutuante usando um *buffer*

Deste modo, para se obter uma maior eficiência na comunicação deve-se usar uma estratégia um pouco mais complexa. Como indicado na figura anterior, é possível eliminar a

necessidade de reconversão, desde que uma cópia do vector a converter seja colocada num segmento de memória auxiliar, sobre a qual é operada a conversão.

O custo da reconversão pode ser assim trocado pelo custo da comunicação interna uni-direccional, que é bastante inferior, reduzindo-se o acréscimo ao custo da comunicação para apenas 4%. No entanto é necessário gastar memória para criar mais um *buffer* adicional, o qual deve ter o comprimento dos pacotes.

Operação	**Tempo de execução (μs)** por número de vírgula flutuante
Comunicação interna uni-direccional T8-C40 (Qi_u)	0,283
conversão de C4f para IEEE 754	1,994
Comunicação externa uni-direccional T8-C40 (Qe_u)	4,896
Tempo total por número de vírgula flutuante	**7,173**

algoritmo 4-9: Envio de um vector de números flutuantes para uma rede heterogénea a partir de um C40

Se bem que esta estratégia permita reduzir o tempo de conversão, requer mais memória interna do processador, a qual em alguns processadores não é abundante. Além disso a implementação desta estratégia requer algoritmos mais complexos e longos do que os apresentados anteriormente. Se por um lado algoritmos mais complexos requerem mais tempo para processamento, algoritmos mais longos requerem mais memória. De facto, se se considerar difusão de mensagens de um para todos, o tamanho máximo de um pacote PCM é bastante grande: 64 KBytes, para aumentar a eficiência. Usar um *buffer* com esta dimensão é impraticável. Uma forma de resolver a situação seria converter os pacotes PCM para séries de pacotes passantes com o comprimento PACKET_LEN, no entanto isto iria aumentar ainda mais a complexidade dos mecanismos de comunicação.

Deste modo, para não aumentar a complexidade dos mecanismos de comunicação, optou-se por manter a técnica indicada no algoritmo 4-8, a qual os algoritmos dos guardas de entrada já descritos seguem.

Finalmente é conveniente referir que o protocolo PCM suporta apenas dados inteiros ou vírgula flutuante. O campo MSG_FLT do descritor, isto é o *bit* 30 toma o valor 1 se os dados de uma mensagem forem números de vírgula flutuante e 0 se inteiros, de modo que as macros de conversão apenas operem sobre números de virgula flutuante, visto que a representação dos inteiros é comum a todos os processadores utilizados.

4.5.2.2 Gestão de erros em tempo de execução

No modelo de processos sequenciais comunicantes (Hoare, 1985), adoptado pelo SPAM e aplicado sobre uma arquitectura de passagem de mensagem, se um processo por algum motivo anormal ficar bloqueado, normalmente todo o sistema bloqueia, ou pelo menos a aplicação retornará resultados imprevisíveis.

Para que um processo fique bloqueado, basta que fique indefinidamente à espera que uma mensagem chegue através de um dado canal, que por algum motivo anormal se perdeu. Outra razão para bloqueio de um processo reside na possibilidade da sequência normal de processamento ser interrompida por algum erro em tempo de execução, que poderá surgir da não disponibilidade de algum recurso numa dada altura. Por outro lado, pelo menos durante o tempo de desenvolvimento da aplicação, será conveniente indicar ao programador se alguma operação foi efectuada de modo ilegal, não existindo assim garantias quanto à fidelidade do seu resultado, como por exemplo operações sobre matrizes cujas dimensões não estejam de acordo com a formulação de álgebra linear.

Para lidar com estas situações foi desenvolvido um sistema de tratamento de erros em tempo de execução, o qual pode ser habilitado pelo programador durante a compilação, e que permite que um processo envie para o *host interface* uma mensagem de erro indicando em que processo ou nó sucedeu e porquê, sendo esta impressa na consola do computador anfitrião antes que a aplicação seja terminada.

Esta mensagem especial é transportada pelo ambiente de comunicações até ao *host interface*, onde é tratada por um processo adequado. Para garantir a maior fiabilidade na entrega da mensagem de erro, esta tem o tamanho máximo de um pacote passante, isto é PACKET_LEN, pois estes são os pacotes usados para comunicar entre a rede e o processo de controlo, como será introduzido no ponto 4.5.2.3. De qualquer modo, mesmo que só o cabeçalho da mensagem de erro atinja o processo de gestão de erros, é sempre indicado que sucedeu um erro e em que nó, antes da aplicação bloquear. O formato de uma mensagem de erro é o seguinte:

Item	comprimento	descrição
Descritor	32 bit	segundo o protocolo PCM
Código de erro	32 bit	Inteiro
Mensagem do utilizador	1 a PACKET_LEN-1 caracteres (de 32 bits)	Texto

tabela 4-9: Formato de uma mensagem de erro

Uma mensagem de erro pode ser constituída apenas pelo descritor e pelo código de erro, de modo que a mensagem do utilizador é opcional. Os códigos de erro estão divididos em várias classes, como erros do ambiente de comunicação, erros da biblioteca de cálculo, etc. A sua descrição completa pode ser consultada no apêndice B.

Esta estratégia apresenta custos, tanto num acréscimo do código, como na perda de desempenho tanto de cálculo como de comunicação, devido às validações adicionais que devem ser efectuadas. Assim, no ambiente de desenvolvimento de aplicações, descrito no capítulo 7, a gestão de erros em tempo de execução pode ser desabilitada. É no entanto aconselhável que pelo menos durante o tempo de desenvolvimento da aplicação não seja desabilitada. Quando habilitada, a macro RTE está definida.

Definindo a função erro (*rtec* [, rtem]) que envia para o processo de controlo uma mensagem de erro cujo código é *rtec*, contendo uma mensagem do utilizador opcional rtem, a primitiva send poderá ser rescrita para lidar com erros nos dados fornecidos para compor o descritor:

algoritmo 4-10:

```
send (descritor, dados)
início
#SE DEFINIDO RTE
se descritor.MSG_LEN ≠ [0, 16383] erro (MSG_DIM)
senão se descritor.MSG_DST = x erro (MSG_CIRCULAR)
senão se (descritor.MSG_DST ≠ [0, N_WRK-1] e ≠ DST_MASTER) erro (MSG_ADR)
#FIMSE
        putmessage (es, descritor, 1)
        ponteiro = endereço de dados
        putmessage (es, ponteiro, 1)
        wait (SFE)
fim
```

As directivas #SE <condição> - #SENAO - #FIMSE, são utilizadas para indicar a dependência da geração de código da avaliação verdadeira da <condição>.

Para simplificar o código da função erro, já que esta consiste no envio de uma mensagem para o processo de controlo, esta pode ser definida recorrendo à primitiva send:

algoritmo 4-11:

```
erro (rtec, [rtem])
início
dados = [rtec, rtem]
descritor.MSG_LEN = PACKET_LEN
descritor.MSG_DST = DST_MASTER
```

```
descritor.MSG_ERR = 1
send (descritor, dados)
fim
```

Para tornar o código o mais simples possível, os mecanismos de encaminhamento de mensagens não são reentrantes, o que significa que não é possível chamar a primitiva send recursivamente. É assim necessário modificar o código desta primitiva de modo que não chame a função de erro, mas que possa de qualquer modo enviar uma mensagem de erro. Uma solução possível é modificar a mensagem a enviar para uma mensagem de erro, modificando o descritor para enviar uma mensagem de erro com uma palavra para o processo de controlo. Consistindo a palavra a enviar no código de erro ocorrido:

algoritmo 4-12:

```
send (descritor, dados, espera)
início
#SE DEFINIDO RTE
se descritor.MSG_LEN ≠ [0, 16383] dados = MSG_DIM
senão se descritor.MSG_DST = x dados = MSG_CIRCULAR
senão se (descritor.MSG_DST  ≠ [0, N_WRK-1] e ≠ DST_MASTER e ≠ DST_ALL)
        dados = MSG_ADR
descritor.MSG_LEN = 1
descritor.MSG_DST = DST_MASTER
descritor.MSG_ERR = 1
#SENÃO
se descritor.MSG_LEN ≠ [0, 16383]  OU
    descritor.MSG_DST = x OU
   (descritor.MSG_DST  ≠ [0, N_WRK-1] e ≠ DST_MASTER e ≠ DST_ALL)  retorna 0
#FIMSE
senão
        putmessage (es, descritor, 1)
        ponteiro = endereço de dados
        putmessage (es, ponteiro, 1)
        se espera = verdadeiro wait (SFE)
        retorna 1
fim
```

Por outro lado, como num algoritmo paralelo a comunicação é tão crítica que falhas podem fazer com que uma aplicação se comporte imprevisivelmente, mesmo que o programador opte por não utilizar o sistema de tratamento de erros, esta função retorna 1 ou o valor lógico verdade apenas se não foi detectada alguma anomalia.

4.5.2.3 *Comunicação com o processo de controlo*

A comunicação entre o *host interface* e o primeiro nó da rede, ou o ponto de entrada nesta, está esquematizada na figura seguinte. Como se pode observar o nó 0 possuí dois portos de comunicação extras, um para enviar dados e outro para receber dados do *host interface*. No entanto por parte do *host interface* a recepção de dados não é assim tão simples. Como uma mensagem proveniente da rede pode ser uma mensagem para enviar para o anfitrião ou uma mensagem de erro, para distinguir ambos os tipos existe um guarda de entrada que se encarrega de dirigir o descritor da mensagem para o Gestor de erros em tempo de execução ou para o *interface* como o anfitrião ou *Host*. Como só o descritor é dirigido qualquer dos processos lê directamente os dados provenientes da rede sem necessidade de recorrer a um *Buffer* intermédio.

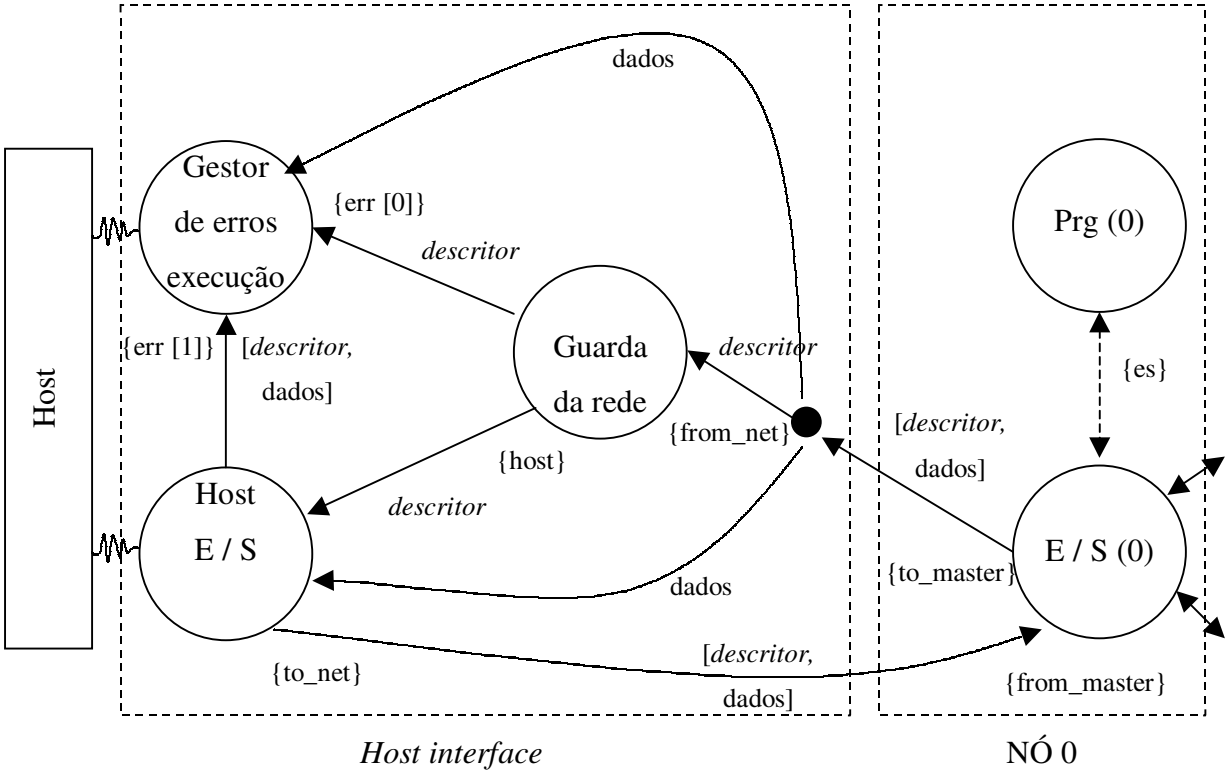


Fig. 4-22: Diagrama blocos dos processos que compõem o *host interface* e o nó 0

O processo Host E/S consiste num servidor de acesso aos recursos e/ou periféricos do computador anfitrião, o qual reconhece pedidos enviados pelas instâncias aplicação Prg (*x*), no protocolo PESA, encapsulado em pacotes PCM e que será descrito no capítulo seguinte.
O acesso ao sistema de gestão de erros de execução a partir de Host E/S é feito por um canal privilegiado, de modo que a função erro (*rtec* [, *rtem*]) é implementada de modo ligeiramente diferente no *host interface*. Por isso mesmo também serão necessárias algumas alterações na

primitivas send e receive. Na realidade, como será visto mais à frente, estas primitivas terão de ser praticamente rescritas, pois neste processo memória para armazenar dados seria redundante, servindo este apenas de *interface* com o anfitrião. No capítulo seguinte será introduzido um conjunto de funções especificas para lidar com estes aspectos de comunicação com o anfitrião. Todas elas recorrem a um *buffer* com o comprimento de PACKET_LEN, de modo que a comunicação com o anfitrião é efectuada também por pacotes passantes.

É conveniente clarificar que a comunicação entre o nó 0 e o *host interface* é feito através de canais de modo que ambos possam ser mapeados em processadores diferentes, o que poderá ser útil em redes heterogéneas. Assim a comunicação torna-se um pouco mais lenta pois é necessário enviar vectores de dados, em vez de passar apenas um ponteiro para dados numa zona de memória partilhada, o que só seria possível mantendo o conjunto sempre no mesmo processador.

Assim serão necessário um guarda de entrada e outro de saída extras neste nó. Além disto, os guardas de entrada algoritmo 4-3 e algoritmo 4-6, devem ser modificados quanto á implementação da regra 4-2, pois neste caso o destino não será o nó 0, mas sim o porto de saída *npc*, apenas existente neste nó.

Quanto às primitivas send e receive, as suas versões para o *host interface* serão sempre bloqueantes, pois aguardam que o vector de dados seja enviado por um canal.

algoritmo 4-13:

```
send (descritor, dados)
início
#SE DEFINIDO RTE
se descritor.MSG_LEN ≠ [0, 16383] dados = MSG_DIM
senão se descritor.MSG_DST = x dados = MSG_CIRCULAR
senão se (descritor.MSG_DST  ≠ [0, N_WRK-1] e ≠ DST_ALL)  dados = MSG_ADR
descritor.MSG_LEN = 1
descritor.MSG_ERR = 1
putmessage (err[1], descritor)
putmessage (err[1], dados)
retorna 0
senão
#SENÃO
se descritor.MSG_LEN ≠ [0, 16383]  OU
    descritor.MSG_DST = x OU
   (descritor.MSG_DST  ≠ [0, N_WRK-1] e ≠ DST_ALL)  retorna 0
#FIMSE
comprimento = descritor.MSG_LEN
Enquanto comprimento > 0
        putmessage (to_net, descritor, 1)
        putmessage (to_net, dados, PACKET_LEN)
```

```
        se (descritor.MSG_DST=DSTALL) comprimento=0
        senão comprimento = comprimento – PACKET_LEN
retorna 1
fim
```

Repare-se que no caso de erro, em vez de alterar o descritor para uma mensagem de erro e usar o próprio corpo da primitiva para enviar a mensagem de erro, esta é enviada directamente pelo canal de acesso ao gestor de erros err [1], e a função é terminada.

Se não existir nenhum erro e o destino for um nó específico a mensagem necessita ser partida em pacotes com o comprimento dos *buffers* de passagem, isto é PACKET_LEN. Estes pacotes são enviados sequencialmente para o nó 0, pelo canal to_net. Se por outro lado o destino for toda a rede não será necessário partir a mensagem em pacotes, pois esta pode ser armazenada em cada nó na sua totalidade.

Quanto à primitiva receive, esta faz precisamente o contrário, isto é recebe sequencialmente pacotes passantes através do canal from_net. Neste caso todas as mensagens terão de ser partidas em pacotes, de modo que atinjam o processo de controlo encaminhadas pelos *buffers* de passagem.

algoritmo 4-14:

```
receive (descritor, dados)
início
Enquanto descritor.MSG_END ≠ 1
        getmessage (from_net, descritor, 1)
        getmessage (from_net, dados, PACKET_LEN)
        se descritor.MSG_SRC ≠  SRC_MASTER
                #SE DEFINIDO RTE
                descritor.MSG_LEN = 1
                descritor.MSG_ERR = 1
                putmessage (err[1], descritor)
                putmessage (err[1], MSG_ADR)
                #FIMSE
                retorna 0
retorna 1
fim
```

A única validação que é feita nesta primitiva é verificar se cada pacote tem o destino correcto.

4.5.2.4 Inicialização da aplicação

Assim que a aplicação é lançada na rede, automaticamente é necessário configurar alguns parâmetros físicos bem como o ambiente de comunicações para a rede alvo. Esta fase, deno-

minada inicialização da aplicação segue os seguintes passos:

a) Inicializar o *hardware*

   i) Seleccionar a taxa de transferência de dados dos portos de comunicação do ADSP2106x para o máximo, que como já foi indicado no capítulo 2, é de 40 MBytes/s.

   ii) Habilitar o *cache* de instruções do C40, de modo que faltas no fornecimento ou *fetching* de instruções ao *pipeline* seja reduzido.

b) Inicializar o ambiente de comunicação

Os mecanismos de encaminhamento de mensagens são inicializados segundo os dados guardados no ficheiro de inicialização da aplicação (*.mif), descrito no capítulo 7. É garantido que os mecanismos correspondentes ao *host interface* são inicializados antes dos correspondentes às instâncias da aplicação alocados em cada nó.

c) Inicializar a percentagem de linhas de cada operando armazenada em cada nó.

Além dos caminhos óptimos previamente calculados, o ficheiro referido na alínea anterior, armazena também o *quantum* de processamento de cada nó, num vector com tantos elementos como o número de nós da rede, identificado por **PQ**. No capítulo seguinte, será apresentado um método de distribuição de dados, ponderando a capacidade de processamento de cada nó.

Assim a macro init_proc, incluída no cabeçalho de compatibilização de *hardware*, compat.h, deve ser invocada primeiramente, de modo a executar o passo a).

Seguidamente, para que todos os processadores possam efectuar os passos b) e c), as tabelas **PQ**, **TEP** e **TEG** devem ser extraídas do ficheiro de inicialização da aplicação pelo *host interface* e distribuídas por todos os processadores da rede. No entanto o ambiente de comunicações ainda não foi inicializado, não podendo ser utilizado para distribuir estes dados pela rede.

Para ultrapassar este problema, recorre-se a uma estratégia de encaminhamento simples, que assume que todos os processadores estão interligados numa fila. Tal geralmente é caso das ligações por defeito que existem nas cartas que alojam os processadores. Se para uma determinada carta isto não se verificar, é da responsabilidade do utilizador garantir que tal fila de processadores existe, configurando o *hardware*. Também é da sua responsabilidade descrever este caminho no modelo da rede, como será abordado no capítulo 7.

Segundo esta estratégia as tabelas **PQ, TEP**(0 ... *np*-1) **e TEG** (0 ... *np*-1) são enviadas sequencialmente ao longo dessa fila, de modo a garantir que cada nó receba uma cópia de **PQ**, e cada nó *x* receba **TEP** (*x*) e **TEG** (*x*).

Para identificar, em cada nó, quais os portos de ligação ao nó anterior e posterior, pode-se recorrer às tabelas de encaminhamento particular, **TEP** (*x*). Visto que em cada nó, esta tabela corresponde a um vector onde cada elemento indica qual o porto usado para ligar ao nó dado pelo índice desse elemento, se se dispuserem os vectores **TEP** (0 ... *np*-1) como colunas de uma matriz, obtém-se:

(4-16)

$$\mathbf{TEP}(0 \ldots np-1) = \begin{array}{c} \begin{array}{cccccc} \mathbf{TEP}(0) & \mathbf{TEP}(1) & \mathbf{TEP}(2) & \cdots & \mathbf{TEP}(np-2) & \mathbf{TEP}(np-1) \end{array} \\ \begin{bmatrix} -1 & PS_{0,1} & PS_{0,2} & \cdots & PS_{0,np-2} & PS_{0,np-1} \\ PS_{1,0} & -1 & PS_{1,2} & \cdots & PS_{1,np-2} & PS_{1,np-1} \\ PS_{2,0} & PS_{2,1} & -1 & \cdots & PS_{2,np-2} & PS_{2,np-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ PS_{np-2,0} & PS_{np-2,1} & PS_{np-2,2} & \cdots & -1 & PS_{np-2,np-1} \\ PS_{np-1,0} & PS_{np-1,1} & PS_{np-1,2} & \cdots & PS_{np-1,np-2} & -1 \end{bmatrix} \end{array}$$

B

A

Onde a diagonal principal é preenchida com um índice de porto de saída igual a -1, o que indica que não existe saída para o próprio nó, como já tinha sido apontado anteriormente. Qualquer outro elemento $PS_{y,x}$ da matriz acima, indica o índice do porto de saída do nó *x* para o nó *y*. Assim traçando a linha A, obtêm-se uma sequência de portos de saída ao longo de uma fila:

[$PS_{1,0}$ $PS_{2,1}$ ... $PS_{np-1,np-2}$ nulo]

enquanto traçando a linha B obtêm-se os portos de saída no sentido inverso nessa mesma fila.

[nulo $PS_{0,1}$ $PS_{1,2}$ ... $PS_{np-2,np-1}$]

Assim pode-se atribuir a cada nó *x* um par (PSf; PSi), onde PSf representa o índice do porto de saída para o nó seguinte, até se atingir o fim da fila, isto é no sentido nó 0 para o nó *np*-1, e PSi o índice do porto de saída no sentido inverso:

[($PS_{1,0}$; nulo) ($PS_{2,1}$; $PS_{0,1}$) ... (nulo; $PS_{np-2,np-1}$)]

Deve ainda ser referido que no nó 0 não existe porto de saída para o nó anterior, tal como no último nó, nó *np*-1, não existe porto de saída para o nó seguinte na fila, pois PSi e PSf tomam um valor nulo. Tal sucede quando as linhas A e B ultrapassam o limite da matriz anterior. Nulo foi convencionado tomar o valor -1.

Como o configurador, descrito no ponto 7.1, gera, entre outros elementos, as tabelas de encaminhamento particular, **TEP** e o ficheiro de configuração, (*.cf), a informação sobre o porto de ligação ao nó anterior e posterior na fila pode ser inserida neste ficheiro, de modo que cada nó recupera esta informação durante o processo de carregamento do executável da aplicação na rede.

4.6 Sub-programas de comunicação

A um nível de programação mais elevado podem-se definir rotinas para executar operações de comunicação comuns. Estas rotinas consistem em chamadas organizadas às primitivas send e receive. Podemos agrupar estas rotinas em duas classes:

i) Comunicação entre instâncias da aplicação Prg (*x*)
ii) Comunicação com o anfitrião.

A última classe já foi abordada neste capítulo e será descrita no capítulo seguinte. Quanto à primeira, em termos de SPAM, visto que os operandos estão distribuídos pelos nós, poderá em algum caso ser necessário guardar um operando completo em todos os nós. Algumas operações matriciais podem tirar partido disso em termos de redução de tempo de execução à custa da memória, como é o caso da multiplicação, que será descrita no capítulo seguinte. O sub-programa seguinte efectua essa operação de comunicação, pois em cada nó, após ser enviada parte do operando nele armazenado, são guardadas em *buffer*, além da sua parte as partes recebidas dos outros nós.

collect (*buffer*)

Outra situação comum será a redistribuição equilibrada de parte do operando pela rede. Isso será necessário quando se pretender operar sobre uma sub-matriz .

OP2 = redist (**OP1**, **OP2**, *li, ci, lf, cf*)

Este sub-programa redistribui a sub-matriz **S** de **OP1,** onde o elemento superior esquerdo é dado por **OP1** $_{li, ci}$ e o elemento inferior direito por **OP1** $_{lf, cf}$. A matriz resultante é referenciada por **OP2.** Essencialmente os segmentos *ci* a *cf*, das linhas *li* a *lf* de **OP1,** são enviadas por ordem crescente para os nós 0 a *np*-1, mantendo a distribuição de carga óptima.

Este sub-programa constituí um bom exemplo de programação sobre o ambiente de comunicações descrito neste capítulo. A operação de redistribuição envolve chamadas ordenadas às primitivas send e receive, bem como transferências internas de dados. Esta operação pode ser dividida em 3 etapas principais:

algoritmo 4-15:

#1 determinação das partições
#2 envio das linhas locais para o nó destino
#3 recepção das linhas remotas

A primeira etapa consiste na obtenção de dois vectores de inteiros, **wop1** e **wop2**, com tantas linhas como as linhas a redistribuir, isto é $nld = cf - ci + 1$. Assim os elementos de **wop1** correspondem às linhas *ci* a *cf* de **OP1**, isto é às linhas de **S**, e os elementos de **wop2** corresponde às linhas 1 a *nld* de **OP2.** Cada elemento destes vectores, indica qual o nó em que a linha dada pelo índice *l* está ou será armazenada, para **S** e **OP2** respectivamente.

É assim possível usar estes vectores, na segunda e terceira etapas, para indicar qual o destino para enviar uma linha *l* de **S**, **wd**[*l*], e qual a fonte donde se deve receber a linha *l* de **OP2**, **ws**[*l*].

Pode também suceder que uma dada linha *l* de **S** esteja armazenada no mesmo nó que a correspondente linha *l* da matriz destino **OP2**. Tal sucede se:

ws [*l*] = **wd** [*l*]

Como as primitivas send e receive não suportam transferências de dados entre o mesmo nó, estas linhas devem ser copiadas de **S** para **OP2**.

Finalmente, para garantir que as operações efectuadas nos nós são sincronizadas é usado o sub-programa:

sinc( ),

que na realidade consiste numa operação collect com o comprimento da partícula de comunicação atómica. É assim possível fazer com que todos os nós esperem que o mais atrasado termine uma dada operação nesse ponto do código. Esta sincronização é equivalente à sincronização de barreira de MPI.
Entre o processo de controlo e as instâncias da aplicação também é utilizada uma operação de comunicação de uma partícula atómica de modo que todas as instâncias da aplicação aguardem por um sinal do processo de controlo. Pode também ser utilizada para distribuir um escalar *w* por toda a rede, sendo por isso definida como:

dist (*w*)

Poder-se-ia neste ponto também abordar a transposição de uma matriz distribuída, pois esta é essencialmente uma operação de comunicação. No entanto, para manter alguma coerência, visto que é uma operação matricial, esta será abordada no capítulo seguinte, juntamente com outras operações de álgebra linear.

4.7 Considerações finais sobre o ambiente de comunicação

A primitiva receive foi descrita neste capítulo, como orientada para receber uma mensagem de uma dada fonte, isto é, o utilizador deve indicar qual a fonte donde se espera a mensagem.
Mas é comum serem esperadas várias mensagens em simultâneo. De facto o sub-programa collect faz exactamente isso, lançando uma primitiva receive não bloqueante, para todos os *np*-1 processadores restantes na rede. Como a finalização de todas as operações de comunicação pedidas com receive é feita assinalando o semáforo SFR, esperando *np*-1 vezes nesse mesmo semáforo, após as chamadas a receive, têm-se a garantia que todas essas operações de comunicação foram concluídas e efectuaram-se concorrentemente.
Por outro lado poder-se-á dar o caso em que se pretende receber uma mensagem apenas, mas de qualquer fonte. Para tal deve ser efectuado um pedido de recepção de mensagem onde receive receberá como fonte esperada SRC_ANY. Neste caso a operação desta primitiva não é tão simples como o já descrito no algoritmo 4-5. Como não se sabe qual a origem da mensagem têm de ser actualizadas as tabelas para todas as fontes possíveis, isto é **TCM** [0..*np*-1] deverá ser inicializada a zero e **TER** [0..*np*-1] deverá conter o endereço de um *buffer* onde será armazenada a mensagem. Da mesma forma, todos os semáforos do vector

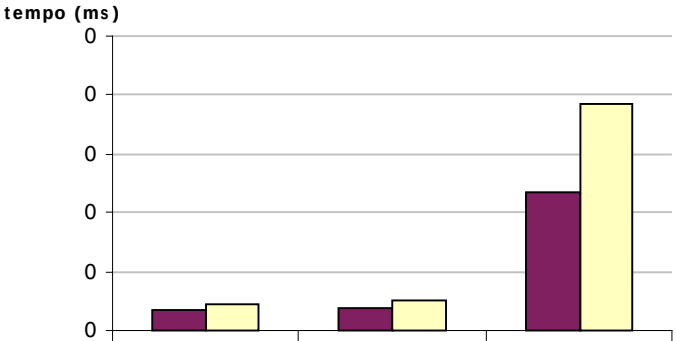
SIR devem ser assinalados. Deste modo indica-se aos guardas de entrada que uma mensagem proveniente de qualquer nó deve ser colocada no mesmo *buffer*.

Neste caso receive deve ser bloqueante de modo que no fim da recepção todos os semáforos que foram assinalados, excepto o semáforo correspondente à fonte da mensagem, sejam restaurados à sua condição prévia. Assim, neste caso, receive aguarda o fim da recepção no semáforo SFR, não se devendo programar um pedido de espera explicito.

4.8 Performance dos MECs

4.8.1 Duplo buffer

Para comparar o desempenho dos mecanismos de comunicação Mec3 com duplo *buffer* dos mesmos sem essa particularidade, foi efectuada uma experiência, que consiste em medir o tempo gasto no envio de um vector com 16000 elementos ao longo de 3 nós e recebê-lo na origem. Como já foi indicado as diferenças de desempenho serão tanto mais visíveis consoante maior fôr o número de pacotes em que a mensagem é partida. Neste caso o tamanho do pacote por defeito é de 256 elementos, implicando que a mensagem é partida em 67 pacotes passantes. Além disso, quanto maior fôr o número de nós a percorrer maior será também o aumento de desempenho. A figura acima mostra que esta estratégia reduz o tempo de comunicações, qualquer que seja o *hardware* utilizado.



	Sharc	C4	T8
buffer duplo	0,0170	0,0180	0,1169
buffer simples	0,0220	0,0260	0,1918

Fig. 4-23: Desempenho escalar das várias FPUs.

Como sempre que uma mensagem tem um destino específico terá de ser partida em pacotes, como já foi abordado, o uso de um *buffer* duplo terá impacto nas mensagens entre o processo

de controlo e a rede, e também na operação de transposição de uma matriz, onde cada mensagem tem sempre um destino específico, como será abordado no capítulo seguinte.

Comparando o desempenho do Sharc com o C40, verifica-se que este último não fica tão distante do primeiro como seria de esperar pela largura de banda anunciada, e também pelas medidas apresentadas no capítulo 2. Um explicação para esta perda de desempenho reside no tempo de inicialização do canal de comunicação ser muito grande no Sharc comparativamente aos outros processadores suportados, como mostra a Fig. 2-24. De acordo com (2-4), como a mensagem é partida em 67 pacotes, a inicialização deve ser efectuada 67 vezes em vez de uma apenas, fazendo com que o tempo de inicialização introduza um atraso mais significativo no caso dos Sharcs.

4.8.2 Ensaio dos mecanismos de comunicação

Pode-se agora comparar o desempenho dos mecanismos de comunicação implementados segundo os modelo teóricos apresentados no ponto 4.1.

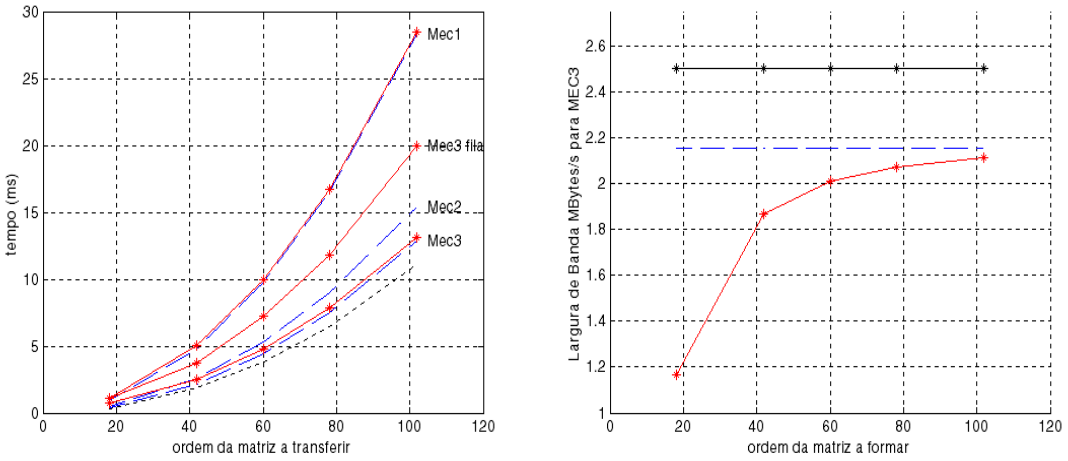


Fig. 4-24: Comparação dos modelos teóricos Mec1-3 com os mecanismos de comunicação implementados sobre uma rede T8

As topologias de rede utilizadas são constituídas por 3 processadores com ligações directas entre eles. Nas figuras à esquerda têm-se o tempo que demora a formar uma matriz completa em todos os nós e à direita têm-se a evolução da largura de banda mas apenas para os mecanismos MEC3. A azul tracejado encontram-se os valores teóricos já apresentados no ponto 4.1, ao passo que as linhas sólidas vermelhas indicam os resultadados medidos na rede alvo. Adicionalmente, também a vermelho sólido e indicado por Mec3 - fila encontram-se as medições sobre uma rede com 3 processadores em fila, para o caso dos mecanismos Mec3.

Assim pode-se comparar a melhor rede em termos de comunicação, onde todos os processadores tem ligações directas entre si, com a pior, isto é uma fila de processadores. Ainda a negro têm-se os valores de pico anunciados pelo fabricante.

Convém ainda referir que os mecanismos Mec2 não foram implementados, visto que Mec3 apresenta desempenho superior, sendo os primeiros uma evolução de Mec1. A resposta teórica dos modelos Mec2 está incluída na figura apenas como referência.

Para o caso da figura acima, pode-se observar que os mecanismos implementados seguem muito de perto o modelo teórico, o que indica que o atraso introduzido pelo ambiente de comunicação é reduzido no caso de T805. Para o caso de uma fila, o atraso deve-se em parte à partição da mensagem em pacotes pelos mecanismos Mec3, mas na sua maior parte ao caminho a percorrer que é maior. De qualquer modo, mesmo quando a rede é uma fila, Mec3 apresenta um desempenho superior a Mec1 sobre uma rede com ligações directas entre nós.

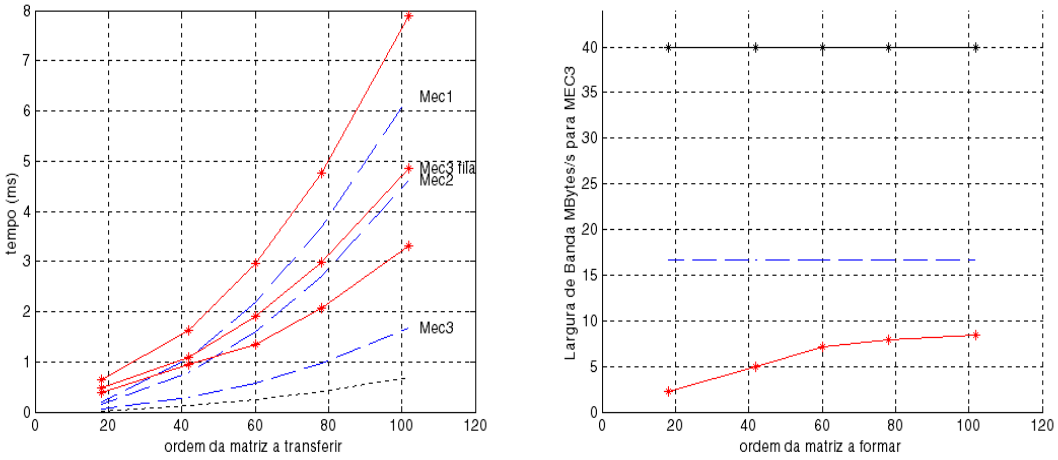


Fig. 4-25: Comparação dos modelos teóricos Mec1-3 com os mecanismos de comunicação impelmentados sobre uma rede C4

Para o caso de redes C40, apresentadas na figura acima, verifica-se que quando o volume de dados aumenta, o desempenho dos portos de comunicação dos C40s aproxima-se um pouco dos valores medidos no capítulo 2, isto é 2 x 8.293 *MBytes* / *s* bidireccionais, mas ainda muito longe dos 2 x 20 *MBytes* / *s* de largura de banda anunciada.

Neste caso pode-se observar que para Mec1 e Mec3 o atraso introduzido pelo ambiente de comunicações é grande, mas este é menos significativo quanto maior for o volume de dados a comunicar.

Para este processadores também se volta a verificar que Mec3, mesmo sobre uma fila de processadores, introduz menor atraso que Mec1 sobre uma rede com ligações ponto a ponto. De facto está muito próximo dos valores teóricos para Mec2.

No caso dos ADSP21060, Fig. 4-26, continua a verificar-se que Mec3 é muito mais eficiente que Mec1, mesmo no caso de uma fila, como mostra a curva Mec3-fila. Neste caso o atraso introduzido por ambos os mecanismo é grande para pequenos volumes de dados a comunicar aproximando-se bastante do modelo teórico quando esse volume aumenta, apresentando Mec3 um desempenho mais próximo do teórico.

No entanto repare-se que neste caso, a largura de banda do modelo teórico, 2 x 19.8 *MBytes*/*s*, é muito próxima da anunciada 2 x 20 *MBytes* / *s*. Mesmo mais próxima que no caso de redes T8, como já foi mostrado no capítulo 2.

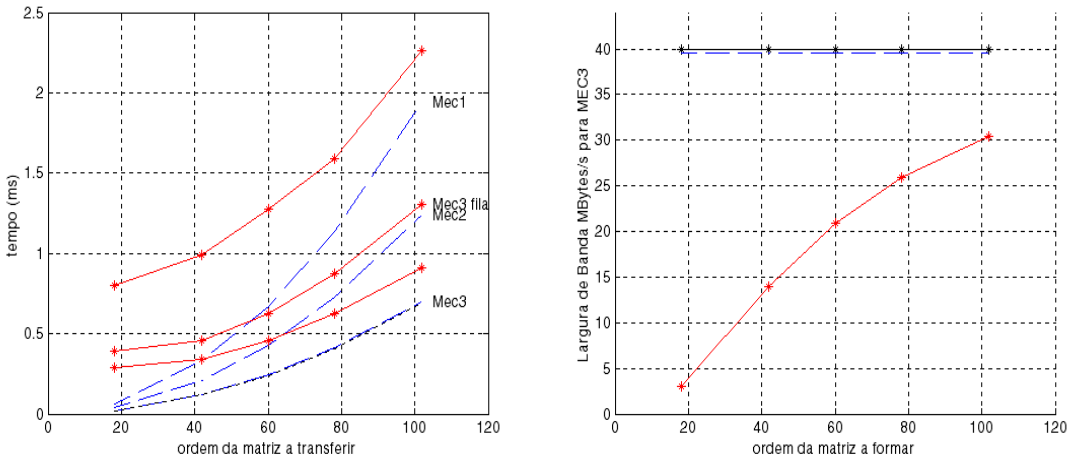


Fig. 4-26: Comparação dos modelos teóricos Mec1-3 com os mecanismos de comunicação implementados sobre uma rede ADSP21060

De qualquer forma, o ensaio desta arquitectura não segue tão de perto o modelo teórico como as baseadas em T805. Como já foi abordado no ponto 2.4.3, as comunicações nesta arquitectura, se bem que recorram a estratégias de DMA tratadas pelo micro-núcleo 3L, provavelmente poderão ser ainda optimizadas.

Além disso, como o tempo de inicialização do canal de comunicação é muito elevado nesta arquitectura, para volumes de dados menores aumenta a distância entre o modelo teórico e o desempenho dos mecanismos de comunicação, quaisquer que eles sejam.

Como o Transputer permite cálculo e comunicação concorrente sem grande perda de desempenho, arquitecturas baseadas neste processador seguem muito de perto o modelo teórico, pois os mecanismos de comunicação não introduzem um atraso significativo.

A medida que permite estabelecer uma relação entre o poder de cálculo e a largura de banda, permitindo assim comparar arquitecturas homogéneas em termos de eficiência, dada pela expressão (2-2), indica para as medidas de desempenho apresentadas no capítulo 2, que a arquitectura mais eficiente é constituída por T8s, seguida de perto pelas baseadas em Sharcs e finalmente as baseadas em C40s.

Aplicando esta medida ao Mec3, para os diferentes volumes de dados que têm vindo a ser utilizados neste ponto, mantendo os valores da velocidade de cálculo para três processadores constante, isto é o triplo do apresentado na Fig. 2-10, obtêm-se os resultados apresentados na figura seguinte:

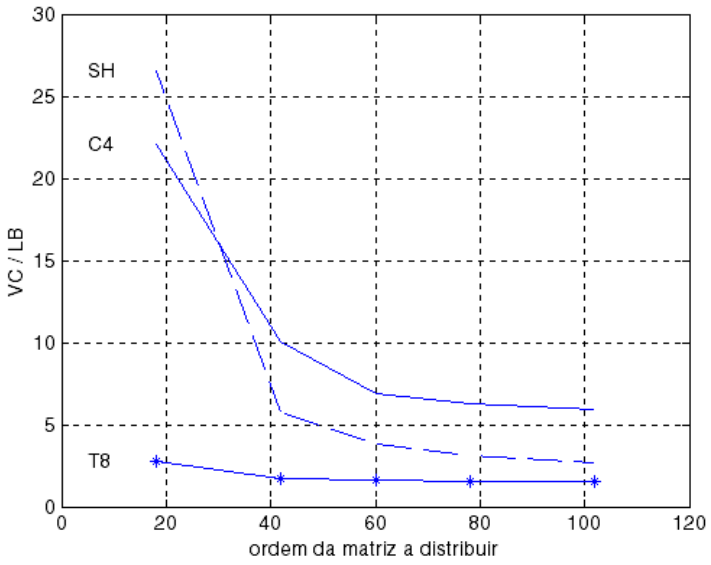


Fig. 4-27: Relação velocidade de cálculo / largura de banda para Mec3

Como se pode ver a arquitectura mais eficiente continua a ser sempre a baseada em T8. Mas para pequenos volumes de dados redes de C40s são mais eficientes que de Sharcs. De facto apenas para ordem de matrizes próximas de 30 é que as redes de Sharcs passam a ser mais eficientes que as de C40s. Como a largura de banda por porto do Sharc, medida no capítulo 2, é mais do dobro do que a do C40, o mecanismo de comunicação introduz um atraso menor nesta última topologia.

Convém ainda mencionar que embora tenham sido utilizados volumes de dados correspondentes a matrizes de ordem 20 a 100, e nos ensaios de rotinas de cálculo apresentadas nos capítulos seguintes sejam utilizadas as mesmas ordens, esta relação não implica que para qualquer algoritmo, desde que a ordem seja superior a 30, as arquitecturas Sharc são mais eficientes que as C40. De facto, em alguns dos algoritmos cuja implementação

é discutida nos capítulos seguintes, o volume de dados a comunicar tem de ser dividido em mensagens com um comprimento reduzido, consistindo apenas numa linha da matriz, de modo que arquitecturas C40 serão mais eficientes que Sharc. Pode-se no entanto afirmar que arquitecturas T8 serão sempre mais eficientes.

Quanto a redes heterogéneas irregulares, os tempos dependem fortemente da topologia da rede. Pode-se no entanto considerar que o tempo de comunicação tem como limite inferior o tempo necessário para atravessar o maior troço da rede. Por outro lado, se se simplificar a rede numa fila eliminado bifurcações, tomando em conta que o custo do novo arco é a soma dos arcos que constituíam a bifurcação, pode-se encontrar um limite superior para o tempo de comunicação.

Finalmente falta referir que visto um dos objectivos desta implementação do SPAM ser a compatibilidade ao nível da linguagem C, com extensões para processamento paralelo da 3L, não foram desenvolvidas primitivas de comunicação que utilizem o DMA para o caso do C40, nem reescritas as fornecidas pelo micro-núcleo 3L para o caso do ADSP21060.

4.9 Resumo

Neste capítulo foi apresentado o desenvolvimento de um eficiente ambiente de comunicação, que será a base do SPAM. De facto, muita da funcionalidade deste transparente ambiente de passagem de mensagem irá simplificar a implementação dos níveis seguintes, principalmente as funções de cálculo paralelas descritas no capítulo seguinte.
No entanto este ambiente não está restrito ao uso do SPAM. É aliás intuito da organização do SPAM, que o sistema possa ser visto como camadas consecutivas onde a abstracção do *hardware* é tanto maior quanto mais alta for a camada, mas garantindo que cada camada traga alguma funcionalidade que possa ser utilizada num ambiente de programação paralelo.
Assim, os mecanismos Mec3, ao disponibilizarem as primitivas de comunicação send e receive, e alguns sub-programas baseados nestas, extendem o ambiente de programação Parallel C da 3 L. Se bem que estas extensões apenas tenham significado sobre compiladores compatíveis com o já referenciado, nada impede que com algumas adaptações se possa extender o ambiente de programação de qualquer compilador de C paralelo. Apenas no caso da arquitectura ser de memória partilhada serão necessárias profundas modificações nos mecanismos de comunicação, visto estes terem sido desenhados para uma arquitectura de passagem de mensagem.

Além disso o sistema de tratamento de erros em tempo de execução adiciona robustez ao ambiente de comunicação, à custa de uma muito pequena perda de desempenho. Esta robustez poderá ser útil durante o desenvolvimento de aplicações.

O ambiente de comunicação proposto permite desenvolver aplicações segundo um modelo mestre-escravos, onde deve ser desenvolvido código diferente para o mestre e para os escravos. No mestre ou no processo de controlo deverá ser desenvolvido código C paralelo para efectuar a comunicação entre a rede e o anfitrião, e consequentemente ordenar as tarefas dos escravos. Nos nós da rede ou nos escravos deve ser desenvolvido o código das tarefas a efectuar. É conveniente notar que o código deverá ser igual para todos os escravos, que operam sobre conjuntos de dados distribuídos diferentes. Pode no entanto ser especificada uma tarefa para ser efectuada apenas por um único nó, usando a variável que guarda a identificação do nó: WRK.

Em termos de SPAM, como será abordado no capítulo seguinte, será desenvolvido um servidor que irá ser executado no processo de controlo e que servirá a rede em termos de acesso aos recursos do anfitrião, simplificando a geração automática de código.

5 Bibliotecas de suporte

As bibliotecas descritas neste capítulo dividem-se em funções de cálculo, entrada saída com o anfitrião e outras funções auxiliares necessárias à manutenção de matrizes distribuídas por uma rede de processadores.

As funções de cálculo são divididas em dois grupos. No primeiro encontram-se as funções de álgebra linear. O segundo grupo engloba outras funções, as quais podem operar sobre polinómios ou tabelas, distribuídas ou locais.

As funções de entrada saída podem ser vistas como uma extensão do ambiente de comunicações descrito no capítulo anterior, para além das primitivas e sub-programas de comunicação já descritos, de modo que todas as instâncias de uma aplicação SPAM possam aceder transparentemente aos recursos do anfitrião. Tal é efectuado através de um servidor que corre no processador raiz e que passa a constituir o processo de controlo. Deste modo, deixa de ser necessário gerar código para o processo de controlo, simplificando o tradutor de SEQ 1.0 para Paralell C, descrito no capítulo seguinte, que apenas necessita de gerar código para as instâncias da aplicação.

5.1 Operandos suportados e sua distribuição

As bibliotecas seguintes suportam dois tipos de operandos reais: matrizes e escalares. Suportam também escalares inteiros, os quais podem ser usados como valores lógicos, considerando que 0 representa o valor lógico falso, enquanto verdade pode ser representado por qualquer outro valor.

Como a estratégia de paralelização usada nestas bibliotecas consiste em executar operações idênticas sobre dados distribuídos, e como não é possível distribuir um escalar pela rede, não é possível paralelizar operações sobre escalares. Deste modo cada nó da rede guarda uma cópia de cada escalar, e as mesmas operações sobre escalares deverão ser executadas em

todos os nós. Assim apenas se poderá referir a operações paralelas, desde que pelo menos um dos operandos seja uma matriz, pois estas são distribuídas pela rede em conjuntos de linhas sucessivas. Se sobre cada uma destas secções forem operadas concorrentemente as funções básicas de álgebra linear, obtém-se um modelo de cálculo matricial paralelo em $\mathbf{R}^2$.

5.1.1 Distribuição da carga computacional

A figura seguinte representa esquematicamente a distribuição de uma matriz por *np* nós de uma rede.

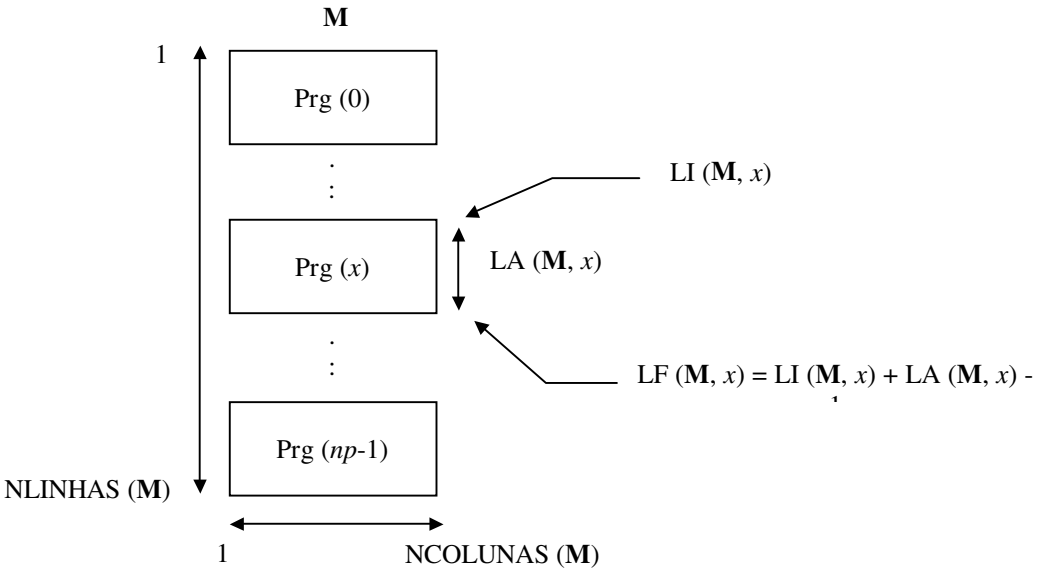


Fig. 5-1: Distribuição de uma matriz **M** por *np* processadores

Onde **M** é uma matriz cujas linhas e colunas têm índices 1 a NLINHAS (**M**) e 1 a NCOLUNAS (**M**) respectivamente. Como a matriz é distribuída pela rede, por conjuntos consecutivos de linhas, num nó *x* a primeira linha da matriz **M** armazenada é dada por LI (**M**, *x*) e a última por LF (**M**, *x*), retornando a função LA (**M**, *x*) o número de linhas armazenadas nesse nó. Esta funções são pois utilizadas para indicar ao sistema como deve ser distribuída a carga computacional, de modo a distribui-la o mais uniformemente possível pela rede.

Considerando que uma rede é homogénea, admite-se que todos os processadores tem o mesmo poder de cálculo, logo a carga deve ser distribuída igualmente. No entanto, como o elemento atómico desta distribuição é uma linha da matriz a distribuir, poderá dar-se o caso em que o número de linhas de uma matriz não poderá ser igualmente distribuída por todos os processadores. De qualquer forma, o pior cenário será que alguns nós, poderão armazenar mais uma linha que outros. O número de nós que ficará com carga superior é dado pelo resto da divisão inteira do número total de linhas pelo número de nós:

(5-1) $\mathrm{LS}(\mathbf{M}) = \mathrm{NLINHAS}(\mathbf{M})\ \%\ np$

onde o operador % indica o resto da divisão inteira. No caso homogéneo a linha inicial é dada por:

(5-2) $$\mathrm{LI}(\mathbf{M}, x) = \left[(\mathrm{int}) \frac{\mathrm{NLINHAS}(\mathbf{M})}{np} \right] x + \mathrm{LK}(\mathbf{M}, x) + 1$$

onde o operador (int) transforma o número real seguinte num inteiro, pela simples remoção da parte fraccionária. A função auxiliar LK (**M**, *x*) define-se como:

(5-3) $$\mathrm{LK}(\mathbf{M}, x) = \begin{cases} x & \text{se } x < \mathrm{LS}(\mathbf{M}, x) \\ \mathrm{LS}(\mathbf{M}, x) & \text{se } x \geq \mathrm{LS}(\mathbf{M}, x) \end{cases}$$

Desta forma o número de linhas armazenado num nó *x* é dado por:

(5-4) $$\mathrm{LA}(\mathbf{M}, x) = \begin{cases} \mathrm{NLINHAS}(\mathbf{M})\ \%\ np + 1 & \text{se } x < \mathrm{LS}(\mathbf{M}, x) \\ \mathrm{NLINHAS}(\mathbf{M})\ \%\ np & \text{se } x \geq \mathrm{LS}(\mathbf{M}, x) \end{cases}$$

Finalmente pode-se obter a linha final num dado nó com:

(5-5) $\mathrm{LF}(\mathbf{M}, x) = \mathrm{LI}(\mathbf{M}, x) + \mathrm{LA}(\mathbf{M}, x) - 1$

Estas equações foram desenhadas para poderem ser calculadas independentemente em qualquer nó, assim cada nó *x* deve possuir toda a informação necessária para tal, isto é o número de linhas e colunas da matriz **M** e, o número de nós na rede.

No caso de uma rede heterogénea esta propriedade é mantida, garantindo que em cada nó existe informação sobre a velocidade de cálculo relativa de todos os nós da rede. Essa informação é guardada no vector **PQ**, que tem tantos elementos como o número de nós na rede. Um elemento *x* deste vector guarda o *quantum* de processamento do nó *x*. No entanto também poderá guardar um factor que indique, relativamente aos outros nós, qual o desempenho de um dado nó, visto que todos os valores desta tabela serão sempre convertidos para uma percentagem, relativa ao desempenho do nó, na inicialização da aplicação, usando:

$$\text{(5-6)} \quad \mathbf{SQ}[x] = \frac{\mathbf{PQ}[x]}{\sum_{k=0}^{np-1} \mathbf{PQ}[k]} 100\,\%$$

Assim as percentagens em **SQ** podem ser vistas como o desempenho relativo de todos os nós e são utilizadas para distribuir a carga computacional por cada nó. Os conjuntos de linhas das matrizes são distribuídas de acordo com estas percentagens, mas mais uma vez o elemento atómico da distribuição, a linha, não poderá ser partida e distribuída. Assim, alguns processadores terão um pouco mais da carga computacional ideal dada por **PQ**. A carga computacional que não pode ser idealmente distribuída pode ser vista como um número de linhas, menor que o número de nós na rede, e obtém-se com:

$$\text{(5-7)} \quad \mathrm{LS}(\mathbf{M}) = \mathrm{NLINHAS}(\mathbf{M}) - \sum_{k=0}^{np-1} (\mathrm{int})\mathbf{PQ}[k]\mathrm{NLINHAS}(\mathbf{M})$$

A estratégia de distribuição utilizada garante que os processadores com maior desempenho serão os primeiros a ser penalizados, com mais uma linha no máximo, de modo a distribuir a carga o mais homogeneamente possível. É pois utilizado um vector auxiliar **PQpri** com tantos elementos como o número de processadores, onde cada elemento varia entre 0 e *np*-1. Este vector indica a prioridade de atribuição de cada linha da carga computacional restante, a que não pôde ser distribuída idealmente segundo **SQ** e que é dada pela equação (5-7), a cada processador por ordem de desempenho, onde 0 indica a máxima prioridade.
Deste modo, no caso heterogéneo, a linha inicial em cada nó é dada por:

$$\text{(5-8)} \quad \mathrm{LI}(\mathbf{M}, x) = \left[\sum_{k=0}^{x} \mathrm{LA}(\mathbf{M}, k) \right] + 1$$

e o número de linhas armazenado num nó x é dado por:

$$\text{(5-9)} \quad \mathrm{LA}(\mathbf{M}, x) = (\mathrm{int}) \begin{cases} (\mathbf{PQ}[x]\mathrm{NLINHAS}(\mathbf{M}) + 1 & \text{se } \mathbf{PQpri}[x] < \mathrm{LS}(\mathbf{M}, x) \\ \mathbf{PQ}[x]\mathrm{NLINHAS}(\mathbf{M}) & \text{se } \mathbf{PQpri}[x] \geq \mathrm{LS}(\mathbf{M}, x) \end{cases}$$

Como se pode observar, por simples comparação das funções para o caso homogéneo e heterogéneo, estas são mais complexas e consequentemente mais lentas, para o último caso.

Se para o caso homogéneo estas funções operam em tempo constante, já no caso heterogéneo envolvem um número de operações elementares directamente proporcional ao número de nós da rede. Assim embora as funções desenvolvidas para este último caso sejam genéricas, isto é válidas para ambos os casos, apenas devem ser utilizadas em redes heterogéneas, para evitar atrasos desnecessários na computação, sobre redes homogéneas.

Qualquer destas estratégias garante que as linhas são armazenadas em cada nó, por ordem crescente do seu índice, isto é as instâncias da aplicação, Prg (x), com x menor, guardam as linhas com índice menor.

5.1.2 Representação física dos operandos matriciais

As matrizes são armazenadas como vectores de números reais, mais precisamente **float** ou seja uma representação de vírgula flutuante com 4 *bytes*. Outra representação mais precisa, como por exemplo 8 *bytes* ou **double**, será em alguns casos preferível, e é discutida no apêndice A. As bibliotecas descritas neste capítulo foram no entanto ensaiadas para números de vírgula flutuante de 32 *bits*, pois alguns processadores de 32 *bits*, como é o caso do C40, representam um **double**, apenas em 32 *bits*, não existindo assim diferenças para **float**.

Cabeçalho	**M** + 0 NLINHAS (**M**)	**M** + 1 NCOLUNAS (**M**)	**M** + 2 LI (**M**, x)
	End. Físico + 3 LF (**M**, x)	End. Físico + 4 TRANSPOSTO (**M**)	Reservado para futuras implementações
Espaço de Dados	**M** + MDATA **M** $_{LI(x),1}$	(...)	**M** + MDATA + NCOLUNAS (**M**) - 1 **M** $_{LI(x),NCOLUNAS(\mathbf{M})-1}$
	(...)	**M** + MDATA + (l - 1) * NCOLUNAS(**M**) + c - 1	(...)
	M + MDATA + (LA(x) -1) * NCOLUNAS(**M**) **M** $_{LF(x),1}$	(...)	**M** + MDATA + LA(x) * NCOLUNAS(**M**) - 1 **M** $_{LF(x),NCOLUNAS(\mathbf{M})-1}$

Fig. 5-2: Representação física de uma matriz **M** num nó x

A figura acima mostra a organização de uma matriz na memória. Os primeiros elementos guardam a informação sobre o formato da matriz, já discutida no ponto anterior. Este cabeçalho tem um comprimento dado por MDATA. Seguidamente ao cabeçalho são

ordenados os elementos da matriz, linha a linha de modo que um dado elemento $\mathbf{M}_{l,c,}$ encontra-se na posição linear, isto é tomando **M** como um vector, dada por:

(5-10) MDATA + (l - 1) * NCOLUNAS(**M**) + c - 1

Para facilitar o acesso à informação contida no cabeçalho foram definidas algumas macros. Assim para definir um ponteiro para um vector de **float,** os seja uma matriz **M**, pode-se usar:

matrix **M**;

Para definir o tipo de dados de vírgula flutuante, deve-se usar a macro REAL:

REAL r;

Para ler o número de linhas e colunas de uma matriz **M**, a linha inicial e a final no nó corrente, a informação contida respectivamente nos quatro primeiros elementos do vector, armazenados em cada nó, foram definidas as macros:

NLINHAS (**M**)
NCOLUNAS (**M**)
LINHA_INICIO (**M**)
LINHA_FIM (**M**)

O número total de elementos de uma matriz é dado por:

MTXELEM (**M**)

e o número de elementos armazenados no nó corrente dado por:

SMTXELEM (**M**)

É conveniente ter em conta que, se se modificar o cabeçalho, após a alocação de memória para a matrix, o número de linhas e de colunas não deve ultrapassar o alocado. Se tal suceder o espaço indicado no cabeçalho será maior que o espaço alocado e os dados irão corromper

outra área da memória, pois correntemente não existe nenhum mecanismo que detecte tal em tempo de execução. No entanto se for tomado o devido cuidado, a modificação do cabeçalho pode ser usada para alterar a dimensão de um *buffer* geral, para receber o resultado de uma dada operação. Neste caso poder-se-á armazenar dados temporários sem perder tempo inicializando e alocando espaço para uma matriz sempre que necessário. Basta alocar um *buffer* no início do programa, com uma dimensão que satisfaça todos os casos possíveis, se tal puder ser previsto ou estimado, e alterar apenas o cabeçalho deste *buffer* sempre que necessário.

5.1.2.1 Declaração e atribuição

Para alocar uma matriz e inicializá-la são usados o conjunto de funções seguintes, cuja definição se encontra em "maux.c". É conveniente indicar que antes de trabalhar com qualquer função matricial é necessário declarar um ponteiro e inicializá-lo como nulo. Isso pode ser feito recorrendo à macro já referida e igualando a NULL:

matrix **M =** NULL;

ou simplesmente usando a macro:

matrixn (**M**)

Após a declaração do ponteiro deve ser usada a função:

matrix setm (matrix ***P**, INTEGER *l*, INTEGER *c*, char *name, int *x*, int *np*)

que armazena espaço em cada nó *x* para uma matriz distribuída segundo o esquema da Fig. 5-1 por *np* nós , e retorna um ponteiro para ela. A nova matriz terá $l > 0$ linhas e $c > 0$ colunas. Se a matriz apontada por ***P** já tiver sido definida, liberta primeiro o espaço que lhe tinha sido atribuído anteriormente. O valor de cada elemento não é inicializado por esta função. Para isso deve-se recorrer a uma das funções descritas a seguir.
Se existir espaço no *heap*, setm retorna um ponteiro para o início da matriz, se não retorna NULL e a *string* *name é enviada para o processo de gestão de erros em tempo de execução recorrendo a uma função erro (<código do erro>, *name). Deste modo, se em *name estiver o

nome da matriz é possível determinar com mais precisão, em que ponto do código sucedeu o erro em tempo de execução.

Os valores das linhas e colunas, *l* e *c* são inteiros de 32 *bits*, definidos com a macro INTEGER. Esta macro facilita a alteração do número de *bits* dos inteiros em cada implementação das bibliotecas. Se bem que o número de *bits* dos inteiros definidos com esta macro tenha influência nas dimensões máximas que se pode atribuir a cada operando matricial, em alguns processadores o tempo de computação é directamente proporcional ao número de *bits* do inteiro, podendo-se encontrar uma solução de compromisso diferente para cada caso.

Para libertar o espaço alocado no *heap* com a função anterior deve ser usada a função:

void clear_matrix (matrix ***M**)

a qual também coloca no ponteiro ***M** o valor NULL.

Como setm não inicializa os elementos de uma matriz, é disponibilizado um conjunto de funções para esse efeito. Assim o utilizador pode inicializar os elementos como lhe convier, não sendo perdido tempo em inicializações padrão, como é o caso de colocar o valor 0.0 em cada elemento.

A função seguinte permite colocar em todos os elementos da matriz apontada por **M**, o valor *value*.

matrix matrix_fill (matrix **M**, REAL *value*)

Já a função:

matrix matrix_ident (matrix **M**)

terá uma aplicação maior na álgebra linear, pois inicializa os elemento de uma matriz apontada por **M** como uma matriz identidade. Se a matriz não for quadrada será preenchida a diagonal principal com o valor 1.0 e todos os restantes elemento com o valor 0.0.

A função load_matrix permite colocar numa matriz distribuída apontada por **M** os elementos descritos na *string* *data.

void load_matrix (matrix **M**, char *data)

Estes elementos devem ser separados por espaços. Se o número de elementos for maior que MTXELEM(**M**) os restantes serão ignorados. Se for inferior a matriz será preenchida por linhas com os elementos em *data. Os restantes elementos não serão alterados. Se se pretender que estes tomem o valor nulo, será necessário uma chamada prévia à função matrix_fill para colocar em toda a matriz o valor 0.0.
Para ler um elemento da linha *l* e coluna *c*, de uma matriz apontada por **M** é usada a seguinte função:

void matrix_get (matrix **M**, INTEGER *l*, INTEGER *c*, REAL* *n*)

como já foi referido anteriormente, um escalar tem uma cópia em cada nó da rede. Então esta função é responsável por extrair o elemento no nó onde está armazenado e enviá-lo para todos os outros, deixando as cópias no escalar apontado por *n*. Já a função inversa permite colocar um valor real numa dada posição da matrix:

void matrix_put (matrix **M**, INTEGER *l*, INTEGER *c*, REAL *n*)

Finalmente para copiar todos os elementos de uma matriz apontada por **ORG** para a matriz apontada por **DST** deve ser utilizada a seguinte função.

matrix mcopy(matrix **ORG**, matrix **DST**)

As dimensões da matriz destino **DST** são definidas pela própria função e iguais às da matrix origem **ORG**.

5.1.3 Distribuição de um vector

No caso de um vector pode surgir uma perda de eficiência se a distribuição da carga for efectuada apenas como já foi indicado. De facto a distribuição de um vector coluna não implica uma perda de eficiência se distribuído por conjuntos de linhas consecutivas, como é o caso. Mas um vector linha, mantendo a mesma estratégia de distribuição resulta que todo o vector será armazenado em apenas um processador, implicando uma perda de eficiência no cálculo. Para evitar esta perda de eficiência qualquer vector é sempre armazenado como um vector coluna. Se for um vector linha, no cabeçalho da matriz tal é indicado colocando um

valor não nulo no endereço dado por **M** + 4, como mostra a Fig. 5-2. Assim no início da função setm (**M**, ...), já descrita devem ter lugar as seguinte validações:

se $l = 1$ e $c > 1$

$l = c$

$c = 1$

TRANSPOSTO (**M**) = 1

senão TRANSPOSTO (**M**) = 0

De modo a transformar numa alocação de um vector coluna e indicar a transposição usando a macro TRANSPOSTO. Esta macro poderá ser utilizada também para consulta desse estado do operando.

Será, no entanto necessário tomar em conta, em todas as funções de cálculo matricial que possam utilizar vectores, as quais serão descritas nos pontos seguintes, que o operando pode ter sido armazenado transposto.

5.2 Entrada saída com o anfitrião

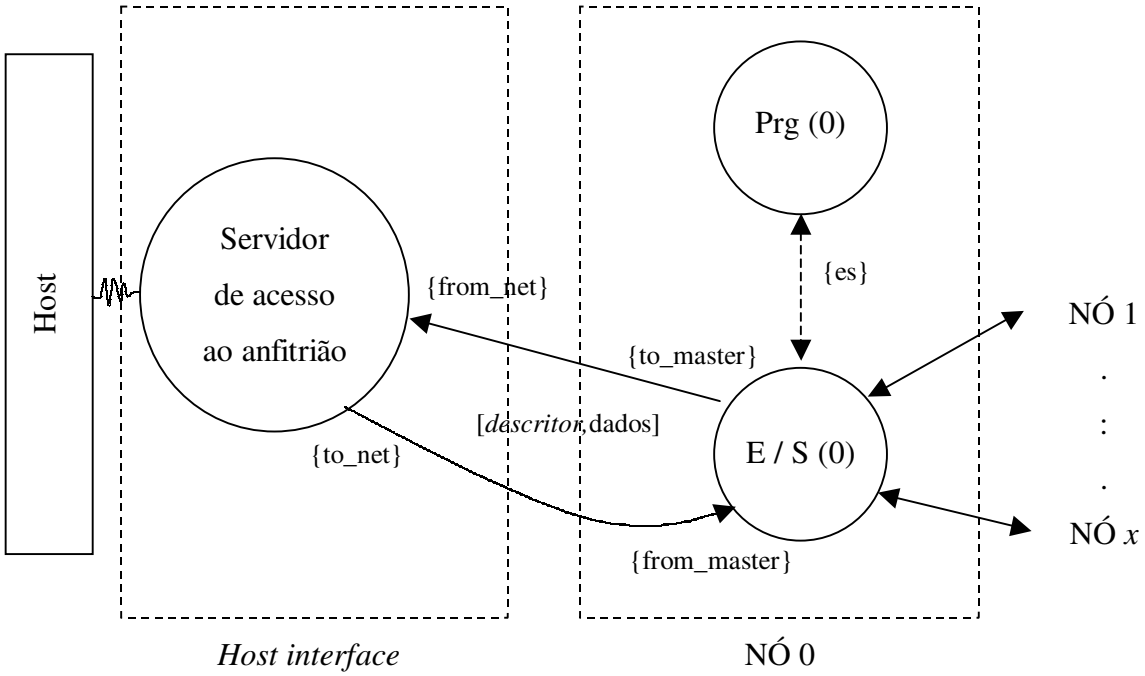


Fig. 5-3: Diagrama blocos que representa o acesso ao anfitrião

No capítulo anterior foi apresentado um modelo de programação onde deve ser desenvolvido código tanto para o processo de controlo com para as instâncias da aplicação. No entanto, em

termos de SPAM, o processo de controlo pode ser visto apenas como um meio de aceder aos recursos do anfitrião. É assim possível desenvolver um processo que possa ser acedido por todas as instâncias da aplicação e que forneça serviços de entrada e saída entre o computador anfitrião e a rede paralela. Assim é também simplificado o desenvolvimento de uma aplicação paralela, visto apenas ter de ser escrito o mesmo código para todos os nós da rede, isto é apenas para as instâncias da aplicação.

O *host interface* é também simplificado, visto que ao contrário do indicado anteriormente, agora apenas consiste no servidor de acesso ao anfitrião. De facto, a partir de agora podem ser vistos como sinónimos.

De qualquer modo o único nó em contacto directo com o *host interface* continua a ser o nó 0. Pedidos de outro nós ao servidor terão de ser encaminhadas por este nó, no protocolo PESA. Este protocolo é montado sobre o protocolo PCM que consiste num *descritor* e um vector de dados. Isto quer dizer que os pedidos de serviços serão codificados no vector de dados no protocolo PESA cujo descritor se apresenta na seguinte tabela.

Nome do campo	*bits*	Descrição
RQST_ID	0 - 7	Código do pedido de serviço
RQST_RSVD	8	Reservado para futuras implementações
RQST_CON	9	Formato dos dados a trocar com o anfitrião
RQST_DATA	10 - 15	Tipo de dados
RQST_NUM	16-31	Campo para valores inteiros de uso geral

tabela 5-1: Descritor de pedido segundo o protocolo PESA

Assim no primeiro elemento de dados será colocado um descritor PESA, com 32 *bits*. Os 2 elementos seguintes podem conter alguma informação significativa, dependendo do tipo de pedido.

Os restantes elementos do vector dados serão inteiros ou reais, dependendo do tipo de pedido. Quando se envia um pedido para o servidor, no campo RQST_ID deve ser colocado o pedido de serviço. Os possíveis pedidos estão indicados na tabela seguinte.

O formato dos dados a trocar com o anfitrião, campo RQST_CON, toma um valor verdadeiro se os dados forem formatados baseados na representação em texto dos valores e introduzindo um espaço após cada um deles e uma mudança de linha após o fim de cada linha de uma matriz. Trata-se efectivamente de uma formatação em ASCII, destinada principalmente a entrada e saída de dados num terminal, por um utilizador humano.

Código do pedido		Descrição
END_SERVER	0	Terminar servidor e consequentemente a aplicação
GET_PARAMETER	1	Ler parâmetros da linha de comandos
OPEN	2	Abrir ficheiro
CLOSE	3	Fechar ficheiro
WRITE	4	Escrever para ficheiro
READ	5	Ler de ficheiro
RESET	6	Reinicializar ficheiro
PAUSE_SERVER	7	Parar servidor temporariamente
GET_TIME	8	Tempo de execução do servidor em segundos
SYNC	9	Sincronizar todas as instâncias com o servidor

tabela 5-2: Pedidos possíveis ao servidor de acesso ao anfitrião

No caso contrário, os dados são enviados num vector de inteiros ou reais IEEE 754, precedido de um inteiro com 4 *bytes* que indica o número de elementos que se lhe seguem. Esta é a forma principal de troca de dados com o exterior, e deve ser utilizada sempre que se pretender comunicar dados com uma aplicação em execução no anfitrião, como seria o caso de um programa monitor, o qual é discutido no ponto 7.2.

Nome do campo	Descrição
TYPE_INT	Escalar inteiro
TYPE_FLT	Escalar real
TYPE_MTX	Matriz
TYPE_ASC	Matriz (Elementos tratados como caracteres)
TYPE_STR	Cadeia de caracteres

tabela 5-3: Tipos de dados segundo o protocolo PESA

É conveniente lembrar que a representação interna do número de vírgula flutuante, é convertida sempre que necessário entre a representação IEEE 754, que circula na rede, entre os nós, e a representação interna do processador onde está alocada a instância da aplicação.
O campo RQST_DATA indica qual o tipo de dados que será comunicado e pode tomar os valores apresentados na tabela anterior.

É conveniente também indicar que a cadeia de caracteres, TYPE_STR, é um tipo de dados artificial, isto é, não é suportado pelas bibliotecas de cálculo. Esta cadeias só são utilizadas para auxiliar a entrada e saída de dados com o anfitrião, sendo apenas armazenadas no nó que está em contacto com o *host interface*, o nó 0.

RQST_ID	**RQST_ CON**	**RQST_ DATA**	**RQST_ NUM**	**DADOS ENVIADOS**	**DADOS DEVOLVIDOS**
END_SERVER					
PAUSE_SERVER					
SYNC					
OPEN				Nome do ficheiro ou dispositivo a abrir: Cadeia com um máximo de 256 caracteres montada num vector de inteiros.	inteiro que indica qual o canal aberto
CLOSE			HCHAN		
RESET			HCHAN		
GET_TIME			HCHAN		
GET_PARAMETER		TYPE_INT TYPE_FLT TYPE_STR	-1 a *argc* - 1		inteiro real *string*
WRITE	0 ou 1	TYPE_INT TYPE_FLT TYPE_MTX TYPE_ASC TYPE_STR	HCHAN	escalar inteiro escalar real linhas, colunas, matriz linhas, colunas, matriz *string*	
READ	0 ou 1	TYPE_INT TYPE_FLT TYPE_MTX TYPE_ASC TYPE_STR	HCHAN		escalar inteiro escalar real matriz matriz string

tabela 5-4: Preenchimento do descritor PESA

Assim se o pretendido for processar cadeias de caracteres em paralelo, esta representação não é a indicada. Mas, cadeias de caracteres podem ser vistas também como vectores, onde cada elemento contém o código ASCII de um caracter. Como um vector é sempre distribuído por todos os nós da rede, já será possível operar sobre cadeias de caracteres em paralelo. Para

indicar que os elementos de uma matriz devem ser tratados como caracteres, o tipo de dados utilizados deverá ser TYPE_ASC.

Finalmente, o campo RQST_NUM, é um campo que guarda um valor inteiro de 2 *bytes*, cujo uso depende do pedido de serviço.

Podemos assim classificar os pedidos de acordo com o descritor PESA, na tabela 5-4. Assim os pedidos END_SERVER e PAUSE_SERVER, que respectivamente terminam a aplicação ou fazem com que todas as instâncias da aplicação aguardem que o utilizador pressione uma tecla, apenas necessitam de enviar no descritor o código do pedido.

Também o pedido SYNC que obriga que todas as instâncias da aplicação esperem que uma tecla seja pressionada apenas precisa de indicar o RQST_ID. Este pedido não deve ser confundido com a sincronização entre as instâncias Prg (*x*) descritas no capítulo anterior. O que faz é pedir ao servidor que envie um sinal para todas as instâncias que ficarão a aguardá-lo. Na realidade este pedido não despoleta qualquer mecanismo no servidor, pois após qualquer pedido o servidor sincroniza sempre as instâncias segundo este método.

O algoritmo seguinte mostra como o servidor trata os pedidos. Primeiro é recebido um pacote PCM, para o *buffer* TMP.

Se for uma mensagem de erro é chamada uma rotina para tratá-lo e é encerrada a aplicação. Em essência esta rotina faz o mesmo que o processo de tratamento de erros apresentado no capítulo anterior, e que faz parte da versão do processo de controlo descrita na Fig. 4-22, já substituída neste capítulo.

Se o pacote PCM não contiver uma mensagem de erro o descritor do pedido é extraído de TMP[1] e o pedido é tratado por uma rotina apropriada. Após o seu tratamento é enviada uma palavra para todas as instâncias da aplicação de modo a sincronizá-las.

algoritmo 5-1:

```
CICLO INFINITO
        descritor.MSG_SRC = 0
        receive (descritor, TMP);

        se EXT_MSG_ERR(descritor) processa erro e termina aplicação
        senão
                RQST_ID = TMP[1] AND 255
        caso RQST_ID
                SYNC:

                END_SERVER:      termina aplicação
                (...)
                GET_PARAMETER:  “processa pedido de parâmetros”
```

descritor.MSG_RES = RES_ALL
send (*descritor*, palavra)

Já o serviço GET_PARAMETER requer além do código do pedido o índice do parâmetro da linha de comandos a devolver. Este é passado no descritor no campo RQST_NUM. O parâmetro cujo índice é zero corresponde ao nome com que a própria aplicação é referenciada pelo sistema operativo ou *filename*. Índices superiores correspondem aos parâmetros imediatamente a seguir ao *filename*, e que devem ser separados por espaços. Se se pretender saber qual o número de parâmetros na linha de comandos, *argc*, RQST_NUM deve ter o valor -1, sendo retornado para todos os nós o inteiro *argc*.

O serviço OPEN requer além do código de pedido, uma cadeia de caracteres com o nome do ficheiro ou dispositivo a abrir. Após uma abertura bem sucedida é retornado para todos os nós o índice do canal aberto, para futuros acessos.

O servidor permite abrir um número limitado de canais, por defeito, 32. Este limite justifica-se para reduzir a memória ocupada no processo servidor pelos descritores de canais. O descritor 0 e o 1 estão reservados e correspondem aos canais de entrada e saída a partir de um terminal ou consola, respectivamente. Todos os outros canais após abertos permitem quer entrada quer saída de dados.

Para fechar um canal, excepto o 0 ou 1, é usado o serviço CLOSE. Este serviço bem como os restantes necessitam que o canal a operar seja indicado no descritor. Este valor é passado para o servidor através do campo RQST_NUM.

RESET faz com que o ponteiro para o próximo elemento a ser lido ou escrito num ficheiro seja o primeiro.

GET_TIME escreve para o canal especificado em RQST_NUM o tempo passado desde que o servidor foi iniciado até ao pedido ser efectuado, em segundos.

READ e WRITE permitem escrever ou ler, de um ficheiro especificado por RQST_NUM, um escalar, uma matriz ou uma cadeia de caracteres. No caso de escrita os valores são passados no vector de dados imediatamente a seguir ao descritor PESA. No caso de uma leitura são devolvidos pelo servidor numa mensagem.

Visto não existir necessidade de todos os nós enviarem um pedido ao servidor foi convencionado que o pedido é apenas enviado pelo nó mais próximo, o nó 0. Dependendo do serviço pedido os dados necessários para completar o serviço serão transmitidos entre o servidor e os nós que os armazenam. Assim no caso de escrita de um escalar, visto que existe uma cópia desse mesmo escalar em todos os nós, foi convencionado que será enviado apenas

a partir do nó 0 para o servidor. Pelo mesmo motivo, no caso de leitura de um escalar, esse será enviado para todos os nós da rede. Tal é também o caso da abertura de um ficheiro no anfitrião onde a identificação do canal aberto, que é um escalar inteiro, deve ser devolvida para todos os nós.

No caso de comunicação de matrizes com o anfitrião, como este dados estão distribuídos por toda a rede, todos os nós devem enviar ou receber parte da matriz. Foi assim convencionado que a ordem de transmissão se iniciae no nó 0 e segue ascendentemente até ao último nó da rede. Tal é devido á forma como está distribuída uma matriz na rede, onde linhas com índices menores encontram-se armazenadas em nós cujo índice é menor como mostra a Fig. 5-1. É assim possível desmontar um operando, que está armazenado completamente num ficheiro, nas suas partes e atribuí-las aos nós correspondentes. É também possível fazer o inverso, isto é montar num dado ficheiro um operando completo a partir das partes armazenadas em cada nó, mantendo a sequência das linhas intacta.

Os ficheiro são tratados sequencialmente pelas instruções de entrada saída. Assim sempre que uma é concluída, o ponteiro é avançado de acordo com o número de *bytes* lidos ou escritos por essa instrução.

Para facilitar os pedidos que cada nó possa efectuar ao servidor de acesso ao anfitrião foram desenvolvidas as seguintes rotinas:

void END()

Pede o serviço END_SERVER que termina a aplicação

void PAUSE()

Pede o serviço PAUSE_SERVER o qual bloqueia todas as instâncias Prg (*x*) até que o utilizador pressione uma tecla.

void MSYNC()

Pede o serviço SYNC que força a sincronização das instancias Prg (*x*), aguardando um sinal do servidor.

int ARGS (int *a*, char* *s*)

int ARGF (int *a*, float * *f*)

int ARGI (int *a*, int * *i*)

int ARGC (int * *i*)

Pedem o serviço GET_PARAMETER, que retorna para o *buffer* dado por *s*, *f* ou *i*, o parâmetro com índice *a*, no formato de um inteiro, real ou cadeia de caracteres respectivamente. ARGC retorna no endereço dado por *i* o número de parâmetros na linha de comandos.

int HOPEN (char* *f*)
Abre um canal de entrada saída, para comunicação com um ficheiro ou dispositivo do anfitrião, especificado pela *string f*. Retorna o índice do canal aberto.

void HCLOSE (int *c*)
Fecha o canal especificado por *c*.

int HRESET (int *c*)
Se o canal estiver aberto, coloca o ponteiro para o próximo *byte* a ser lido ou escrito no ficheiro, no início deste.

int PRINT(int *c*, char **s*)
Efectua um pedido WRITE para escrever no ficheiro correspondente ao canal *c*, a cadeia de caracteres apontada por *s*.

int WRITEI (int *c*, int* *n*)
int WRITEF (int *c*, float* *n*)
Efectuam um pedido WRITE para escrever no ficheiro correspondente ao canal *c*, o inteiro ou real apontado por *n*, respectivamente. Para o servidor, é apenas enviada a cópia armazenada no nó 0.

int READI (int *c*, int* *n*)
int READF (int *c*, float* *n*)
Efectuam um pedido READ para ler do ficheiro correspondente ao canal *c*, o inteiro ou real apontado por *n*, respectivamente. O servidor envia o escalar para todos os nós da rede.

int WRITEM (int *c*, matriz *m*)
int WRITEMA (int *c*, matriz *m*)

int putvector (int *c*, matriz *m*)

Efectuam um pedido WRITE para escrever no ficheiro correspondente ao canal *c*, a matriz distribuída cuja parte armazenada no nó correspondente é apontada por *m*. WRITEM formata a saída em ASCII. Assim, juntamente com o pedido, o nó 0 envia informação sobre as linhas e colunas da matriz, de modo que o servidor formate correctamente a saída. Se se tratar de um vector transposto as linhas e colunas são trocadas de modo que a saída não seja transposta.
As macros MDigits e MPrecison são utilizadas para formatar a precisão de cada número real. Respectivamente referem-se ao número de dígitos na parte inteira e ao número de dígitos na parte fraccionária. Por defeito ambas têm o valor 4.
WRITEMA escreve os elementos da matriz, linha a linha, sem nenhuma formatação, tratando-os como códigos ASCII. Já putvector escreve o vector de reais precedido por um inteiro que indica o número de elementos nesse vector, como já descrito.
Para garantir que o servidor recebe as partes armazenadas em cada instância com a ordem correcta, estas aguardam um sinal do servidor, que é enviado com a ordem 0 até *np*-1. Quando uma instância recebe o sinal, envia para o servidor as linhas da matriz nela armazenadas.

int READM (int *c*, matriz *m*)
int READMA (int *c*, matriz *m*)
int getvector (int *c*, matriz *m*)

Efectuam um pedido READ para ler do ficheiro correspondente ao canal *c*, a matriz distribuída cuja parte armazenada no nó correspondente é apontada por *m*. READM lê reais no formato ASCII, ao passo que getvector lê um inteiro que indica o número de elementos nesse vector, e só depois o vector de reais no formato IEEE 754 de 32 *bits*. READMA lê caracter a caracter, converte-os no código ASCII correspondente e escreve esse código numa matriz, preenchendo-a por linhas. Constituí um modo de ler *strings* de um ficheiro e montá-las numa matriz distribuída.
Para garantir que cada instância recebe a parte da matriz correspondente, estas aguardam um sinal do servidor, que é efectuado com a ordem 0 até *np*-1. Quando uma instância recebe o sinal envia para o servidor o número de elementos que está à espera de receber, após o que o servidor lhos envia.

int WRITEMDIM (int *c*, matriz *m*)

Efectua um pedido WRITE de modo que o cabeçalho das partes de uma matriz, armazenadas em todos os nós, são enviadas para o canal *c*, com a ordem 0 até *np*-1. Este cabeçalho corresponde às primeiras cinco palavras do operando matriz, como já definido na Fig. 5-2. Deste modo é possível registar as dimensões das partes bem como da matriz completa. Se se tratar de um operando transposto o número de linhas estará trocado com o número de colunas, no entanto o campo que indica tal terá um valor não nulo.

int TIME(*c*)

Escreve no ficheiro correspondente ao canal *c* o tempo passado desde que o servidor foi iniciado até ao pedido do serviço GET_TIME, em segundos.

int TIMEW(*c*)
int TIMEWF(*c*, *f*)

Escreve no ficheiro correspondente ao canal *c* um lapso de tempo passado em todas as instâncias da aplicação, em segundos. Na realidade os tempos estão armazenados num vector coluna chamado TIMES, com tantos elementos como nós. Em cada nó é armazenado o elemento correspondente. Assim o que TIMEW (*c*) faz é chamar WRITEM (*c*, TIMES) de modo que esse vector seja enviado para o ficheiro referente a *c*. Já TIMEWF (*c*, *f*) retorna o número de operações de vírgula flutuante por segundo, de um total de *f*.
Os valores neste vector podem ser vistos como leituras de cronómetro, existindo um diferente para cada nó. Podem ser inicializados, isto é tomar o valor 0, com a função:

void START_CLOCK( )

e medido o tempo que passou desde a última inicialização com:

void READ_CLOCK( )

TIMEW e TIMEWF retornam valores mais precisos de tempo que TIME, pois esta última mede também o tempo necessário para que o pedido de medição de tempo seja efectuado no

servidor. As primeiras devem pois ser considerada como o principal elemento para avaliar o desempenho de um algoritmo.

Finalmente falta definir que, se existir um erro de entrada saída as funções retornam 0. Caso não tenha sucedido nenhuma anomalia retornam 1. Se a detecção de erros em tempo de execução estiver habilitada, é enviada uma mensagem correspondente para o servidor, e a aplicação é terminada.

Funções que não retornam um valor inteiro não podem falhar. É o caso de HCLOSE, que se o canal não estiver aberto ou não existir pura e simplesmente não faz nada.

5.3 Bibliotecas de cálculo

Se bem que as bibliotecas descritas neste capítulo sejam fortes candidatos a programação orientada por objectos, para garantir a compatibilidade com compiladores ANSI C, não foi utilizada esta facilidade. Assim recorreu-se a tipos de dados abstractos (Horowitz et al, 1993), ou ADT, do inglês *Abtract Data Types*, os quais permitem que programas clientes acedam aos dados da matriz, através de funções definidas na sua *interface*. Deste modo a representação dos dados e funções que implementam as operações estão completamente separadas do cliente por uma *interface* opaca, visto que os clientes não podem determinar a implementação através dela. Assim, embora em ANSI C não sejam disponibilizados mecanismos explícitos que permitem separar a *interface* da implementação, de modo a facilitar a implementação de objectos, como é o caso das classes em C++ (Drake, 1998), recorrendo a tipos de dados abstractos, e assim separando a implementação do *interface*, pode-se garantir a segurança do acesso aos dados, por um programa cliente.

Para manter alguma simplicidade e possibilitar expansão das funções de cálculo matricial, estas são distribuídas por três bibliotecas: bmcalc, que íncluí as funções mais comuns da algebra linear, como a adição, a multiplicação, a inversão e a transposição; mcalc, onde estão agrupadas funções que operam sobre *arrays*, que não correspondem necessáriamente ao objecto matemático matriz, tais como rotação de vectores, multiplicação e divisão polinomial; e mcalcext. Esta última pode ser utilizada para adição de funções às bibliotecas pelo utilizador. O ficheiro fonte já contêm todas definições para que o ambiente de comunicações seja utilizado. O utilizador deve apenas acrescentar as funções por si definidas ao ficheiro “mcalcext.c”, e o protótipo correspondente a “mcalcext.h”. No entanto deve-se tomar atenção que a lista de argumentos de entrada e saída deve obedecer a regras especificas, de modo que o SPAM, mais precisamente o tradutor, reconheça as funções como válidas, e que serão descritas mais adiante.

Todas as funções a seguir descritas validam os seus argumentos, excepto se a detecção de erros em tempo de execução for desabilitada quando da compilação. É assim possível reduzir o tempo de execução, à custa de alguma segurança. De qualquer modo, após a fase de desenvolvimento de uma aplicação, se o código cliente validar apenas os argumentos críticos, antes de os passar para as funções, e fôr desabilitada a opção de detecção de erros atrás mencionada, deverá ser possível obter um melhor desempenho.

A indicação de quais as validações que deverão ser feitas pelo programa cliente, caso a detecção de erros em tempo de execução esteja desabilitada, estão indicadas em cada função como pré-condições, segundo a técnica de projecto por contracto (Meyer, 1997). Estas indicam qual o estado esperado do programa, pelo menos das variáveis relevantes, antes de se executar uma dada função. Segundo esta técnica definem-se também pós-condições, que são declarações de como ficará o estado do programa após a execução da função em causa, isto é as condições que a função garante satisfazer após ser executada.

Estas pré e pós-condições são assim vistas como a definição de um contrato entre a função e o código que a utiliza, ou cliente, de modo que o estado final referido pelas pós-condições é garantido se forem cumpridas as pré-condições, antes de chamar a função. Se a função não é capaz satisfazer as suas pós-condições, diz-se que houve uma quebra de contracto. Se as pré-condições forem satisfeitas isto indica um erro na função, se não forem satisfeitas indica um erro no cliente. Segundo esta técnica é costume, antes do protótipo de uma função, indicar a lista de pré e pós condições, bem como excepções que possam suceder. Estas são eventos assíncronos que ocorrem em tempo de execução, mesmo quando as pré-condições são satisfeitas, e por razões que são estranhas à função em causa impedem o cumprimento das pós-condições. A razão mais comum para que ocorram excepções, no caso destas bibliotecas, é a insuficiência de memória disponível para efectuar uma operação.

As condições são definidas como asserções, isto é afirmações que devem ser sempre verdadeiras. Estas sempre que possível são expressas formalmente através de expressões lógicas. Além disso, sempre que necessário indicar o estado de uma variável, antes da função ser chamada, é usado a notação "old <identificador>".

Para evitar repetições desnecessárias na documentação das funções, nas pré-condições não é indicado que os ponteiros para matrizes devem ser não nulos, nem que o número de linha e colunas deve ser maior que zero, no entanto tal deve ser sempre assumido. Isto significa que as matrizes devem ser alocadas antes de chamar as funções. Por outro lado, se uma matriz for alocada pela própria função tal é expressamente indicado como comentário. Neste caso se o ponteiro para essa matriz apontar uma outra qualquer matriz o seu conteúdo é destruído.

Deve ser tomado em conta que as pré-condições podem ser validadas pela própria função, libertando dessa tarefa o programa cliente, se a detecção de erros em tempo de execução for habilitada. No entanto as excepções nunca poderão ser validadas pelo programa cliente, visto que dependem da execução da própria função. Isto implica que se for desabilitada a detecção de erros, a ocorrência de excepções nunca é verificada.

5.3.1 Tipos de funções

Os cabeçalhos das funções da biblioteca de cálculo matricial obedecem a um padrão, dependente dos operandos necessários para cada operação, que também é suportado pelo tradutor, descrito no capítulo seguinte. As funções são assim classificadas em 4 tipos principais:

O tipo I divide-se em dois sub-tipos:

TIPO I A

resultado = nome_da_função (**operando_1**, **resultado**)

A operação requer apenas um operando matricial. O resultado pode ser guardado em **resultado**, ou no próprio operando se **resultado** e **operando1** apontarem a mesma matriz.

TIPO I B

nome_da_função (**operando_1**)

A operação requer também só um operando matricial. A função opera sobre o argumento de entrada destruindo o seu estado antes da chamada. Se se pretender operar sobre uma cópia de uma variável matricial, de modo a preservar o estado inicial desta, deve ser utilizada a função mcopy, descrita acima, que pertence ao tipo I A.

O tipo II também se divide em dois sub-tipos:

TIPO II A

resultado = nome_da_função (**operando_1**, **operando_2**, **resultado**)

A operação requer dois operandos matriciais. O resultado deve ser guardado em **resultado.** Para algumas operações, tais como a soma, pode-se guardar também o resultado num dos operandos.

TIPO II B

resultado = nome_da_função (**operando_1**, *operando_2*, **resultado**)

A operação requer dois operandos. Um matricial e um escalar. O resultado pode ser guardado em **resultado**, ou no próprio operando.

TIPO III

resultado = nome_da_função (**operando_1**)

Este tipo é usado para operações que operam sobre uma matriz e retornam um escalar. Se se pretender um que o resultado seja inteiro, ou um valor lógico, isto é um inteiro que toma o valor 0 para falso e outro qualquer para verdade, o resultado que é um escalar do tipo REAL, deve ser convertido coerciva e explicitamente, para inteiro após a chamada à função ou à saída desta.

TIPO IV

resultado = nome_da_função (**operando_1, operando_2**)

Este tipo pode ser usado para operações que operam ou comparam duas matrizes e retornam um valor escalar ou lógico. O resultado pode ser convertido para inteiro como já foi indicado para o tipo III.

Deve ser apontado que todos os operandos devem existir e estar dimensionados, ou pelo menos devem apontar para NULL, antes de ser chamada qualquer função. A necessidade de passar um ponteiro para a matriz **resultado**, na lista de argumentos de entrada, pode implicar que o conteúdo seja destruído, sendo retornado o novo conteúdo na lista de argumentos de saída.

Como se pode observar as funções não foram desenhadas para operar sobre sub-matrizes, o que implica que se tal for necessário, deverá ser obtida uma sub-matriz, distribuída

equilibradamente antes da chamada à função. Tal pode ser obtido com a operação redist, já descrita no capítulo anterior:

sub_matriz = redist (**matriz**, **sub_matriz**, *linha_inicial*, *coluna_inicial*, *linha_final*, *coluna_final*)

Da mesma forma, se se pretender operar sobre operandos transpostos, estes devem ser obtidos à priori, com a função própria para esse efeito descrita a seguir, excepto no caso da multiplicação, que como será visto foram definidas duas operações, que devem ser usadas dependo de se pretender operar com a transposta do segundo operando.

Poder-se-ía ainda ter definido o tipo I C, que consiste numa operação unária sobre um escalar:

resultado = nome_da_função (*operando_1***,** *resultado*)

bem como o tipo II C, onde a operação é aplicada a dois escalares, resultando um escalar:

resultado = nome_da_função (*operando_1***,** *operando_2***,** *resultado*)

No entanto estas operações são efectuadas recorrendo aos operadores definidos na própria linguagem C, de modo que não é necessário desenvolver funções especificas.

5.3.2 Biblioteca de cálculo Matricial elementar (bmcalc)

Nesta biblioteca estão definidas as funções básicas da álgebra linear, a adição, multiplicação e inversão, bem como a transposição. Esta última é abordada já a seguir.

5.3.2.1 Transposição de uma matriz

Objectivo: Retorna **C**, a matriz transposta de **A**.
POS: **A** = = $\mathbf{C}^T$
NLINHAS (**A**) = = NCOLUNAS (**C)** &&
NCOLUNAS (**A**) = = NLINHAS (**C**)
Excepção: Se não houver memória suficiente para alocar *buffers* temporários, gera um erro em tempo de execução.
Tipo: I A

matrix mtrans (matrix **A**, matrix* **C**)

Esta operação é na realidade um problema de comunicações, visto que nenhuma operação é efectuada sobre os elementos da matriz **A** para se obter **C**, mas estes são redistribuídos, segundo um critério onde os elementos da linha *i* de **A**, passam a pertencer à coluna *i* de **C**, mantendo-se as suas posições relativas.

Como a matriz a transpor se encontrar distribuída, a operação transposição corresponde à operação de comunicação já definida no ponto 4.3.1.5, como indexação.

a)
$$\mathbf{A} = \begin{array}{c} \begin{matrix} wt=0 & wt=1 & \cdots & wt=np-1 \end{matrix} \\ \begin{bmatrix} \mathbf{X}_0 & \mathbf{M}_0 & \cdots & \mathbf{M}_1 \\ \mathbf{M}_2 & \mathbf{X}_1 & \cdots & \mathbf{M}_3 \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{M}_4 & \mathbf{M}_5 & \cdots & \mathbf{X}_{np-1} \end{bmatrix} \end{array} \begin{matrix} w=0 \\ w=1 \\ \\ w=np-1 \end{matrix}$$

b)
$$\mathbf{C} = \mathbf{A}^T = \begin{array}{c} \begin{matrix} w=0 & w=1 & \cdots & w=np-1 \end{matrix} \\ \begin{bmatrix} \mathbf{X}_0^T & \mathbf{M}_2^T & \cdots & \mathbf{M}_4^T \\ \mathbf{M}_0^T & \mathbf{X}_1^T & \cdots & \mathbf{M}_5^T \\ & & \ddots & \\ \mathbf{M}_1^T & \mathbf{M}_3^T & \cdots & \mathbf{X}_{np-1}^T \end{bmatrix} \end{array} \begin{matrix} wt=0 \\ wt=1 \\ \\ wt=np-1 \end{matrix}$$

Fig. 5-4: Transposição de uma matriz distribuída

Na figura acima pode-se observar a partição de uma matriz e da sua transposta, de acordo com os *np* nós existentes na rede. Esta é assim particionada em sub-matrizes, cujas linhas estão armazenadas num mesmo nó, indicado por *w*, e cujas colunas correspondem às linhas a armazenar num dado nó *wt*, da matriz transposta.

Pode-se observar que entre a) e b) as sub-matrizes da diagonal principal, $\mathbf{X}_0$ a $\mathbf{X}_{np-1}$, não precisam de ser trocados entre nós diferentes. Assim a transposição destas é local a cada processador e é efectuada trocando a linha e a coluna de cada elemento:

(5-11) $x0_{i,j} = x1_{j,i,}$ com $x0 \ni \mathbf{X}_{0...np-1}$ e $x1 \ni \mathbf{X}_{0...np-1}{}^{T}$

Pode-se pensar em usar uma técnica análoga para as sub-matrizes $\mathbf{M}_0$ a $\mathbf{M}_5$, que devem ser transferidas entre nós. Assim deve-se transferir elemento a elemento de cada uma destas para o nó e posição correspondente à sua transpostas, isto é $\mathbf{M}_0{}^{T}$ a $\mathbf{M}_5{}^{T}$.

No entanto, mensagens com apenas um elemento fazem com que a largura de banda do ambiente de comunicações seja reduzida a 50%, visto que o cabeçalho da mensagem tem o mesmo comprimento que os dados a transferir. É pois preferível enviar mensagens com o maior comprimento possível.

Para maximizar o comprimento destas mensagens pode-se enviar de uma só vez toda uma linha de cada sub-matriz, visto que os elementos ocupam posições de memória contíguas. No entanto, na matriz transposta os elementos não serão posicionados em posições de memória consecutivas, mas sim afastados por tantos elementos como o comprimento das linhas desta matriz. Deste modo é necessário receber a mensagem num *buffer*, e depois armazenar os elementos recebidos na matriz transposta nas posições correspondentes.

Além do atraso inerente à cópia dos elementos do *buffer* para a transposta, também se deve considerar que se só um *buffer* for utilizado, não é possível receber mensagens de nós diferentes concorrentemente. De facto tal só é possível se existir memória reservada com comprimento suficiente para receber as diferentes mensagens, como é o caso da própria matriz transposta. Mas como já se viu que não se pode receber as mensagens directamente na própria transposta, devem ser alocados *buffers* suficientes para receber as mensagens concorrentemente, isto é um por cada porto de comunicação, e com o comprimento máximo de elementos que pode ter a linha de uma qualquer sub-matriz.

No entanto, para arquitecturas heterogéneas, onde em cada nó o número de elementos por linha pode variar bastante, esta estratégia pode implicar uma perda de memória significativa. Outra estratégia consiste na alocação de um *buffer* para cada nó, onde o número de linhas da sub-matriz é conhecido e sempre constante. Mas se o número de nós for grande, também existirá uma perda de memória significativa. Apesar disto, esta última estratégia de alocação de *buffers* é mais simples e por isso mais rápida de executar que a anterior, sendo por isso a utilizada.

Pode-se assim dividir o algoritmo de transposição em 4 etapas principais.

algoritmo 5-2:

#1 Identificar as sub-matrizes segundo o esquema da Fig. 5-1
#2 transferir as sub-matrizes entre nós e *buffers* temporários
#3 copiar as linhas dos *buffers* temporários para a coluna transposta correspondente
#4 transpor as sub-matrizes locais segundo (5-11)

A etapa #1, vai introduzir um atraso que é impossível de eliminar e que dependerá da implementação das equações (5-7) a (5-9), que como já foi visto no ponto 5.1.1, são usadas para a distribuição equilibrada da carga computacional.

Já a etapa #3 poderia ser eliminada se existisse uma função de recepção que pudesse receber uma mensagem e armazenar os seus elementos em posição de memória com afastamento constante. O problema é que o micro-núcleo 3L, base de uma aplicação, não fornece tal função, embora os co-processadres de DMA normalmente o possibilitem. Mas como já foi referido, em termos de compatibilidade optou-se por usar o mesmo compilador para todas as famílias de processadores, de modo que esta limitação só poderia ser removida sacrificando a compatibilidade. De facto, visto os portos de comunicação terem um pequeno *buffer* FIFO associado, onde as mensagens são temporariamente armazenadas, em segmentos, antes de serem copiadas para a memória principal, é possível rescrever a função de recepção do micro-núcleo, usando o co-processador de DMA se presente, de modo que transfira os dados destes *buffers*, elemento a elemento, e os coloque em posições de memória com afastamento constante mas que podem ser não contíguas.

Este algoritmo não é no entanto muito eficiente. Tal pode ser observado na Fig. 5-29, página 200, onde estão comparados os tempos para algumas operações em três nós e para matrizes de ordem 100 x 100. Pode-se pois observar que o tempo da operação transposição descrita acima, e que será referida a partir de agora como versão 1, ou mtrans v1, é muito superior ao da variante mais rápida da operação collect, collect2 descrita mais à frente. Isto é, a obtenção de uma matriz completa em todos os nós é muito mais rápida que a obtenção de transpostas parciais. Tal acontece por os dados parciais em cada nó, numa operação collect, serem enviados numa só operação de comunicação, ao passo que na transposição são enviados vários conjuntos de elementos. Ora como já foi indicado os mecanismos de comunicação são comparativamente mais rápidos para mensagens maiores, justificando assim as diferenças entre estas operações. Esta diferença será ainda mais visível em arquitecturas ADSP21060, pois como já foi indicado a inicialização de um canal de comunicação é muito pesada.

Pode-se no entanto abordar a transposição de outra forma, tornando o algoritmo mais simples e reduzindo o número de operações de comunicação

algoritmo 5-3:

#1 Formar em cada nó o operando completo
#2 Obter a transposta parcial a partir do operando completo

Assim o primeiro passo consiste numa operação collect2, sendo a obtenção da matriz parcial transposta referente a um dado nó, obtida por cópia dos elementos do operando completo nas colunas correspondentes.

Como se mostrará mais à frente a diferença de tempo entre as duas versões não é muito significativa para Transputers. É conveniente recordar que ao contrário dos outros processadores utilizados, estes comunicam e processam em simultâneo, sem necessidade recorrer a estratégias de DMA, e o tempo de inicialização do canal não é significativo; no entanto para as restantes arquitecturas suportadas a diferença já é bastante significativa. Foi pois escolhida a implementação da versão 2 de mtrans para todas as famílias de processadores suportadas.

No entanto a partir de uma dada ordem, bastante grande, mesmo sem recorrer a estratégias de DMA, a versão 1 deverá ser mais rápida, pois o número de elementos da matriz comunicado por esta versão é menor. Tal sucederá quando os vários segmentos de elementos comunicados, na versão 1, forem tão grandes que o tempo de inicialização do canal e restante atraso introduzido pelo ambiente de comunicações, principalmente na formação do pacote de comunicações, seja desprezável.

Após a descrição do caso de transposição geral, falta ainda referir o caso da transposição de um vector, que é igual em ambas as versões. Como estes são sempre armazenado da mesma forma, independentemente de serem coluna ou linha, a sua transposição é muito mais simples e eficiente, bastando copiar concorrentemente os elementos de **A**, existentes em cada nó para os elementos correspondentes de **C**, e alterar o atributo que indica a transposição.

algoritmo 5-4:

C = **A**
TRANSPOSTO (**C**) = ! (TRANSPOSTO (**A**))

No caso de transposição de vectores, não existe um pré-condição que obrigue o atributo TRANSPOSTO de **C** a tomar o valor o inverso do mesmo atributo de **A**. De facto tal

validação nem é efectuada pelo sistema de detecção de erros em tempo real. Assim ambos o operando e o resultado podem ser vectores linha ou coluna, com a mesma dimensão, pois a inversão deste atributo é sempre efectuada pela função mtrans. Podem inclusivamente ser o mesmo vector; neste caso, no entanto a cópia dos elementos de **A** para **C**, isto é para si próprios, é redundante sendo preferível alterar apenas o atributo transposto, que é precisamente o que a macro TRANSPOR descrita a seguir faz.

Todas as funções desta biblioteca podem também ser acedidas através de macros, cuja utilidade será desempenhar alguma manipulação antes e ou depois da chamada à função correspondente. No caso de MTRANS não existe necessidade de proceder a nenhuma manipulação, assim esta macro apenas transfere os argumentos para a função mtrans.

MTRANS (**A, C**) = = mtrans (**A**, &**C**)

É ainda definida uma macro que apenas inverte o atributo TRANSPOSTO, de uma matriz ou vector, não efectuando uma transposição real dos elementos, pois não chama a função mtrans:

TRANSPOR (**M**) = = TRANSPOSTO (**M**) = ! (TRANSPOSTO (**M**))

Não deve ser vista como uma função da biblioteca de cálculo algébrico, mas apenas como um artificio que poderá ser útil para alterar esse atributo do ADT matrix, o que só seria possível com mtrans fazendo com que esta actuasse sobre o mesmo vector operando e resultado, perdendo-se bastante em eficiência, pois seria necessário copiar todos os elementos para si próprios.

5.3.2.2 Operações entre matrizes e escalares

Esta operações envolvem um operando matricial e um escalar. São as funções mais simples da biblioteca, e consequentemente as mais eficientes, visto que apenas os endereços dos elementos de um operando matricial devem ser descodificados. Em cada nó, cada elemento nele armazenado é operado com o operando escalar. Correspondem pois ao tipo II B. As operações que podem ser efectuadas são a adição e a multiplicação. Deste modo são definidas 2 funções. Se se pretender efectuar as operações inversas, basta que o escalar fornecido seja o simétrico ou o inverso do operando respectivamente. A primeira função consiste na soma de uma matriz com um escalar:

Objectivo: Retorna a matriz **C** a soma da matriz **A** com o escalar *b*
POS: **C** - *b* = = **A**
NLINHAS (**A**) = = NLINHAS (**C)** &&
NCOLUNAS (**A**) = = NCOLUNAS (**C**) &&
TRANSPOSTO (**A**) = = TRANSPOSTO (**C**)
Tipo: II B

matrix numadd (matrix **A**, REAL *b*, matrix* **C**)

Alternativamente as seguintes macros podem ser utilizadas para efectuar a adição e a subtracção. Para a primeira operação têm-se:

NUMADD(**A**, *b*, **C**) = = numadd(**A**, *b*, &**C**)

E para subtrair o escalar *b* a todos os elementos da matriz **A**:

NUMSUB(**A**, *b*, **C**) = = numadd(**A**, - *b*, &**C**)

no entanto, no caso em que ao operando esquerdo escalar é subtraída uma matriz, será preferível evitar desperdiçar memória criando uma matriz temporária com elementos simétricos. Para evitar isso foi definida uma função que trata este caso específico:

Objectivo: Retorna a matriz **C,** a subtracção da matriz **A** ao escalar *b*
POS: **C** + *b* == **A**
NLINHAS (**A**) = = NLINHAS (**C)** &&
NCOLUNAS (**A**) = = NCOLUNAS (**C**) &&
TRANSPOSTO (**A**) = = TRANSPOSTO (**C**)
Tipo: II B

matrix nummsub (matrix **A**, REAL *b*, matrix* **C**)

sendo também definida a macro:

NUMMSUB(**A**, *b*, **C**) = = nummsub(**A**, *b*, &**C**)

Para multiplicar uma matriz por um escalar deve ser usada a função:

Objectivo: Retorna na matriz **C** o produto da matriz **A** com o escalar *b*
POS: **C** / *b* = = **A**
NLINHAS (**A**) = = NLINHAS (**C)** &&
NCOLUNAS (**A**) = = NCOLUNAS (**C**) &&
TRANSPOSTO (**A**) = = TRANSPOSTO (**C**)

Tipo: II B

matrix nummult (matrix **A**, REAL *b*, matrix* **C**)

Alternativamente as seguintes macros podem ser utilizadas para efectuar a multiplicação e a divisão. Para a multiplicação:

NUMMULT(**A**, *b*, **C**) = = nummult (**A**, *b*, &**C**)

E para dividir a matriz **A** pelo escalar *b*:

NUMDIV(**A**, *b*, **C**) = = nummult (**A**, 1/(REAL) *b*, &**C**)

São também definidas duas funções que operam apenas sobre os elementos da diagonal principal de uma matriz. Se se pretender que os restantes elementos da matriz operando existam na matriz resultado, ambos os ponteiros **A** e **C** devem indicar a mesma matriz. Para somar aos elementos da diagonal principal de **A** o escalar *b* deve ser usada a função:

Objectivo: Retorna na diagonal principal da matriz quadrada **C** a soma algébrica da diagonal principal da matriz **A** com o escalar *b*. Os restantes elementos de **C** não são alterados.
PRE: NLINHAS (**A**) = = NCOLUNAS (**C**)
POS: diagonal principal de **C** - *b* = = diagonal principal de **A**
NLINHAS (**A**) = = NLINHAS (**C**) &&
NCOLUNAS (**A**) = = NCOLUNAS (**C**) &&
TRANSPOSTO (**A**) = = TRANSPOSTO (**C**)
Tipo: II B

matrix diagadd (matrix **A**, REAL *b*, matrix* **C**)

Alternativamente, as seguintes macros podem ser utilizadas para efectuar a adição e a subtracção de um escalar à diagonal principal, respectivamente:

DIAGADD(**A**, *b*, **C**) = = diagadd (**A**, *b*, &**C**)
DIAGSUB(**A**, *b*, **C**) = = diagadd (**A**, - *b*, &**C**)

E para multiplicar a diagonal principal por um escalar:

Objectivo: Retorna na diagonal principal da matriz quadrada **C** o produto da diagonal principal da matriz **A** com o escalar *b*. Os restantes elementos de **C** não são alterados.
PRE: NLINHAS (**A**) = = NCOLUNAS (**C**)

POS: diagonal principal de **C** / *b* = = diagonal principal de **A**
NLINHAS (**A**) = = NLINHAS (**C)** &&
NCOLUNAS (**A**) = = NCOLUNAS (**C**) &&
TRANSPOSTO (**A**) = = TRANSPOSTO (**C**)
Tipo: II B

matrix diagmult (matrix **A**, REAL *b*, matrix* **C**)

Alternativamente, as seguintes macros podem ser utilizadas para efectuar a multiplicação e divisão da diagonal principal de **A** pelo escalar *b*, respectivamente:

DIAGMULT(**A**, *b*, **C**) = = diagmult(**A**, *b*, &**C**)
DIAGDIV(**A**, *b*, **C**) = = diagmult(**A**, 1/(REAL) *b*, &**C**)

5.3.2.3 Adição de vectores ou matrizes

Estas operações são realizadas pela mesma função. Cada elemento de um operando é somado com o elemento da posição correspondente do outro operando. Não são tão eficientes como as anteriores pois é necessário descodificar o endereço dos elementos de dois operandos matriciais. Os vectores ou matrizes operandos devem ter a mesma dimensão da matriz resultado.

Objectivo: Retorna na matriz **C** a soma da matriz **A** com a matriz **B**
PRE: NLINHAS (**A**) = = NLINHAS (**B**) &&
NCOLUNAS (**A**) = = NCOLUNAS (**B**) &&
TRANSPOSTO (**A**) = = TRANSPOSTO (**B**)
POS: NLINHAS (**A**) = = NLINHAS (**C)** &&
NCOLUNAS (**A**) = = NCOLUNAS (**C**) &&
TRANSPOSTO (**A**) = = TRANSPOSTO (**C**)
madd: **C** - **B** = = **A**
msub: **C** + **B** = = **A**
Tipo: II A

matrix madd (matrix **A**, matrix **B**, matrix* **C**)
matrix msub (matrix **A**, matrix **B**, matrix* **C**)

Para efectuar a operação inversa, é utilizada uma função idêntica, que só difere na operação a efectuar sobre cada par de elementos, que passa a ser agora uma subtracção. Esta redundância de código justifica-se para optimizar estas funções em termos de velocidade. Para evitar a redundância de código poderia utilizar-se para a subtracção as seguintes instruções:

NUMMULT (**B**, **B, TMP**);

C = MADD (**A, TMP, C**);

No entanto seria requerida memória para a matriz temporária **TMP ,** que armazena a simétrica de **B**, e uma operação matricial extra, para obter essa simétrica. Outra solução seria ter uma função apenas, que mediante um parâmetro de entrada efectuasse a operação adição ou subtracção em cada elemento. Mas isso implicaria que para cada par de elementos a operar fosse verificado o valor desse argumento, mediante uma condição o que iria atrasar o processamento.

Embora a distribuição de um vector seja diferente de uma matriz, como já foi visto, estas operações consistem em somar ou subtrair todos elementos de **A** armazenados em cada nó, com os elementos correspondentes de **B**, que estão armazenados no mesmo nó. Visto que não são permitidas operações entre vectores e matrizes, o vector ou matriz resultado **C** têm os elementos distribuídos segundo a mesma estratégia que ambos os operandos. Assim a operação é efectuada elemento a elemento e o resultado é guardado na posição correspondente de **C**, sendo irrelevante a estratégia de distribuição de operandos utilizada.
Alternativamente, as seguintes macros podem ser utilizadas para efectuar a soma ou subtracção de duas matrizes, respectivamente:

MADD(**A**, **B**, **C**) = = madd (**A**, **B**, &**C**)
MSUB(**A**, **B**, **C**) = = msub (**A**, **B**, &**C**)

A sua utilidade consiste apenas numa homogeneização das macro da biblioteca, visto que na realidade apenas passam os operandos para a função correspondente.

5.3.2.4 Multiplicação de vectores e matrizes

Neste caso será necessário implementar funções especificas para efectuar estas operações, dada que a estratégia de distribuição de matrizes é diferente dos vectores, como já foi visto.
Tem-se assim 3 casos principais: a multiplicação de dois vectores, a multiplicação de duas matrizes e a multiplicação de um vector e de uma matriz.

5.3.2.4.1 Multiplicação de vectores

Este caso ainda se pode subdividir em mais dois sub-casos. Quando o primeiro operando é um vector linha e o segundo um vector coluna, onde o resultado é uma matriz com um só

elemento ou um escalar, e o caso inverso em que o resultado é uma matriz quadrada. O sub-programa seguinte identifica o caso e procede de acordo com este.

Objectivo: Retorna em **C** a multiplicação do vector **a** pelo vector **b**
PRE: (NLINHAS (**a**) = = 1 || NCOLUNAS (**a**) **= =** 1) &&
(NLINHAS (**b**) = = 1 || NCOLUNAS (**b**) **= =** 1) &&
(NLINHAS(**a**) > 1 && NLINHAS(**b**) >1) && (
TRANSPOSTO (**a**) ! = TRANSPOSTO (**b**)) &&
SMTXELEM(**a**) = = SMTXELEM(**b**)) &&
POS: **C** = = **a** * **b**
TRANSPOSTO (**a**) && (NLINHAS (**C**) = = NCOLUNAS (**a**) &&
NCOLUNAS (**C**) = = NCOLUNAS (**b**))
TRANSPOSTO (**b**) && (NLINHAS (**C**) = = NLINHAS (**a**) &&
NCOLUNAS (**C**) = = NLINHAS (**b**))
TRANSPOSTO (**a**) && TRANSPOSTO (**b**) (
NLINHAS (**C**) = = NCOLUNAS (**a**) &&
NCOLUNAS (**C**) = = NLINHAS (**b**))
NLINHAS (**C**) = = NLINHAS (**a**) && NCOLUNAS (**C**) = = NCOLUNAS(**b**)
Excepção: Se não houver memória suficiente para alocar *buffer* temporário, gera um erro em tempo de execução.
Tipo: II A

matrix vmult (matrix **a**, matrix **b**, matrix* **C**)

As duas primeiras pré-condições destinam-se a garantir que os ponteiros **a**, **b** e **C** apontam para vectores, ou no pior caso para vectores com apenas um elemento, onde a multiplicação de vectores degenera numa multiplicação de escalares. As seguintes garantem que um vector é coluna e outro linha, e que têm ambos o mesmo número de elementos. Garantem também que ambos os vectores pode ter apenas um elemento não sendo necessário transposição. Já as pós-condições indicam qual a dimensão da matriz resultado **C,** dadas as dimensões de ambos os operandos, e tomando em consideração que um vector é sempre armazenado como coluna, sendo indicado se é linha pelo atributo TRANSPOSTO.
Como já foi referido existem dois casos a identificar e tratar. Como já foi garantido que ambos os operandos são vectores, e estão transpostos entre si, basta verificar-se a condição:

NLINHAS(**a**) = = 1 || (NCOLUNAS (**a**) = = 1 && TRANSPOSTO (**a**))

para que este sub-programa efectue o primeiro caso. Ou ainda simplesmente:

TRANSPOSTO (**a**)

visto que vectores armazenados como colunas são linhas se transpostos. Ainda para garantir que é efectuada a operação produto interno mesmo quando número de elemento é apenas 1, onde esta degenera na multiplicação de escalares, esta última condição transforma-se em:

TRANSPOSTO (**a**) || NLINHAS (**a**) = =1

Assim o produto interno de dois vectores **a** e **b**, com comprimento *m*, e que resulta num escalar ou numa matriz com apenas uma linha e uma coluna:

(5-12) $c = \mathbf{a} \,.\, \mathbf{b}$

(5-13) $$c = \begin{bmatrix} a_1 & a_2 & a_3 & \cdots & a_m \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_m \end{bmatrix}$$

deve ser efectuado distribuindo ambos os vectores em *np* nós da seguinte forma:

(5-14) $$c = \begin{bmatrix} \mathbf{a}^0 & \vdots & \mathbf{a}^1 & \vdots & \cdots & \vdots & \mathbf{a}^{np-1} \end{bmatrix} \begin{bmatrix} \mathbf{b}^0 \\ \cdots \\ \mathbf{b}^1 \\ \cdots \\ \vdots \\ \cdots \\ \mathbf{b}^{np-1} \end{bmatrix}$$

onde os vectores $\mathbf{a}^x$ e $\mathbf{b}^x$, estão armazenados no nó *x*, quer estes sejam linha ou coluna, como já foi referido no ponto 5.1.3.

Como mostra esquematicamente a figura acima, basta então em cada nó calcular o produto interno dos vectores nele armazenados, e enviar o resultado de cada nó, escalar c^x, para o nó 0, onde é sempre armazenada a primeira linha de **C**. No nó 0 devem ser recebidos os resultados parciais de cada nó, somados e atribuído o seu resultado *c*, ao elemento da linha 1 e coluna 1 de **C.**

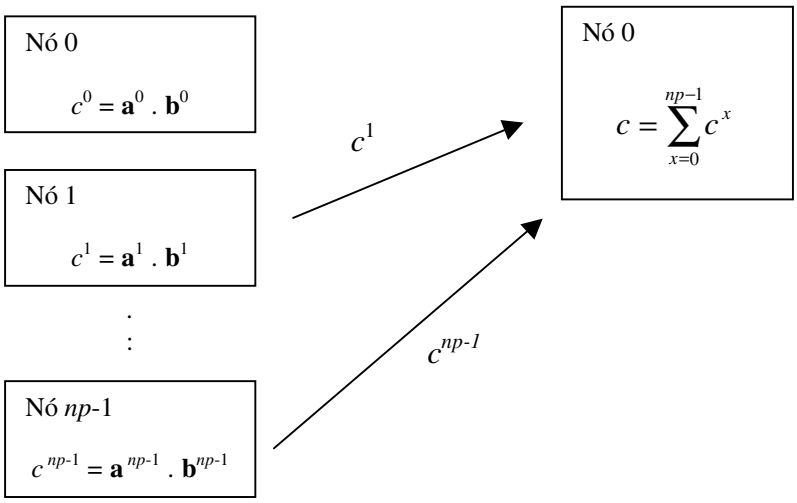


Fig. 5-5: Computação em paralelo do produto interno de 2 vectores

E daqui pode-se obter o seguinte algoritmo, que se executa em todas as instâncias da aplicação Prg (*x*), onde os nós têm o mesmo índice *x* da instância.

algoritmo 5-5:

```
c^x = a^x . b^x
se x ≠ 0  send (c^x, 0)
senão                                   "Se nó 0"
        c = c^0
        desde y = 1 até  np-1
                descritor.MSG_SRC = SRC_ANY
                receive (descritor, tmp)
                c = c + tmp
```

No entanto, embora a etiqueta de fonte SRC_ANY seja muito útil para indicar que se pretende receber um determinado número de mensagens, *y* neste caso, independentemente do nó fonte e da ordem, tem a desvantagem de no caso de se receber várias mensagens em simultâneo, provenientes de nós diferentes como é o caso, estas serem todas aceites e guardadas num único endereço, *tmp* no algoritmo acima, o que implica que a última recebida será a única preservada. Uma solução possível seria alterar o comportamento da primitiva receive, para que sempre que recebesse uma mensagem terminasse. Mas tal implicaria alterações ao ambiente de comunicações tornando-o mais complexo e que provavelmente implicaria também um atraso maior introduzido por este. Outro meio de resolver este problema consiste em alterar a técnica usada para receber os produtos internos parciais, dando as ordens de recepção em simultâneo para endereços de memória diferentes:

algoritmo 5-6:

$c^x = \mathbf{a}^x \,.\, \mathbf{b}^x$
se $x \neq 0$ send (c^x, 0)
senão "Se nó 0"
    $c = c^0$
    desde y = 1 até np-1
        descritor.MSG_SRC = y
        receive (*descritor*, *tmp* [y-1])
    wait (SFE, np-1)
    desde y = 1 até np-1 $c = c + tmp[y\text{-}1]$

Assim, à custa de um *buffer* para recepção maior, a recepção é mais rápida, pois não é necessário receber mensagens uma a uma como era o objectivo do algoritmo 5-5.

Repare-se que para indicar as partições dos vectores armazenadas num dado nó x, estas são indicadas por um índice x: $\mathbf{a}^x$, $\mathbf{b}^x$. No entanto, ao longo desta tese, quando não existir um índice explicito para cada nó é usada uma plica a seguir à variável, para indicar que se está a referir a uma partição: **a**', **b**'.

O algoritmo pode também ser descrito por um diagrama de paralelização, ver apêndice C, onde são mais evidentes as relações entre as comunicações e o cálculo. Este tipo de diagramas é composto por 3 áreas. Na área da direita é indicado o segmento de código que é executado em todas as instâncias da aplicação, Prg (x). Na área da esquerda são indicados os segmentos de código que só são executados numa dada instância x. Na área central está indicada a direcção, tipo e quantidade de dados a comunicar.

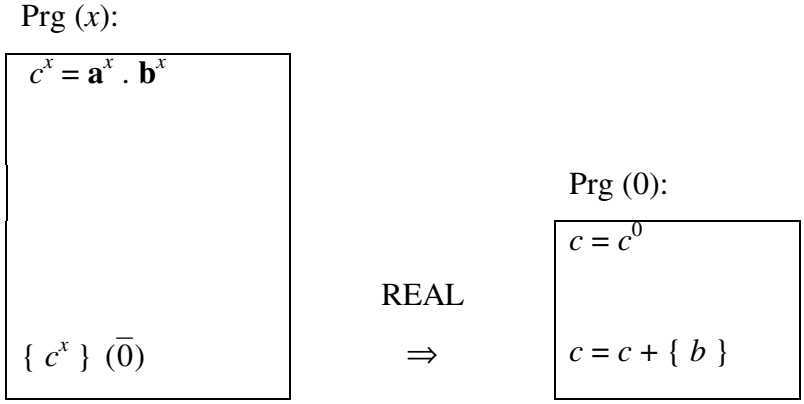


tabela 5-5: Diagrama de paralelização do produto interno

Repare-se que Prg (0) calcula primeiro o código indicado no 1º bloco da coluna da esquerda, antes de executar o código indicado na coluna da direita. Desta forma o fluxo de execução é dado ao longo das linhas do diagrama. É também conveniente apontar que as variáveis a

comunicar estão entre { }, de modo a poder escrever expressões que aguardam dados antes de serem calculadas, como é o caso da última linha de código de Prg (0):

$$c = c + \{ b \}$$

onde após receber c^x de cada nó, e guardar o seu valor em b, soma este a c, antes de receber o próximo c^x.

Falta então abordar o cálculo dos produtos internos locais $c^x = \mathbf{a}^x \,.\, \mathbf{b}^x$, os quais não são efectuados em paralelo. O produto interno de dois vectores com m elementos, um linha e outro coluna, define-se como:

$$(5\text{-}15) \quad c = \sum_{k=1}^{m} \mathbf{a}_k \,.\, \mathbf{b}^{\mathrm{T}}{}_k$$

para a partição armazenada no nó x:

$$(5\text{-}16) \quad c^x = \sum_{k=0}^{m^x-1} \mathbf{a}^x{}_k \,.\, \left(\mathbf{b}^{\mathrm{T}}\right)^x{}_k$$

onde m_x representa o número de elementos na partição x, que pode ser traduzido para um algoritmo em português estruturado e próximo das linguagens de programação imperativas como:

algoritmo 5-7:

$c^x = 0$
desde $k = 0$ até $m^x - 1$
    $c^x = c^x + \mathbf{a}^x[k] \,.\, \mathbf{b}^x[k]$

No entanto, se calcular o número de operações de vírgula flutuante necessárias para calcular o produto interno de dois vectores a partir da expressão (5-15) ou para este caso concreto a partir de (5-16), verifica-se que são necessárias m^x multiplicações e $m^x - 1$ adições, isto é um total de $2\,m^x - 1$ operações. Mas a partir do anterior algoritmo são necessárias $2\,m^x$ operações. Esta primeira aproximação não é pois a mais eficiente. A mais eficiente passa pela inicialização da variável escalar que guardará o resultado final.

Como é o caso da linguagem C, não existe garantia que uma dada linguagem inicializa uma variável com o valor 0, quando da sua declaração. Assim é necessário fazê-lo explicitamente antes de calcular o produto interno, como é efectuado pelo algoritmo anterior. Mas se se inicializar a variável com o resultado da primeira multiplicação evita-se uma iteração do ciclo seguinte e consequentemente uma adição.

algoritmo 5-8:

$c^x = \mathbf{a}^x[0] \,.\, \mathbf{b}^x[0]$
desde $k = 1$ até $m^x - 1$
    $c^x = c^x + \mathbf{a}^x[k] \,.\, \mathbf{b}^x[k]$

Esta optimização, se bem que não seja significativa para o cálculo do produto interno, visto só reduzir em uma unidade o número de operações face ao algoritmo 5-7, poderá revelar-se bastante significativa na multiplicação de matrizes, onde são efectuadas tantas operações idênticas ao produto interno como o número de linhas do primeiro operando multiplicado pelo número de colunas do segundo, e que será tratado no ponto 5.3.2.4.3.
Deve ser dito, para clarificar, que os índices de linha e coluna do objecto matemático matriz assumem-se maiores que 0. No entanto grande parte das linguagens de programação imperativas permitem definir índices 0. Daí que nos algoritmos em português estruturado seja usada esta última convenção, ao passo que nas expressões matemáticas seja usada a primeira.

Finalmente, para terminar este caso, deve ser indicado que para converter a matriz resultado **C**, com apenas um elemento, o qual está armazenado no nó 0, para um escalar REAL *n,* que têm uma instância com o mesmo valor em todos os nós, pode ser usado:

matrix_get (**C**, 1, 1, &*n*)

Quanto ao segundo caso, que sucede quando o primeiro operando é um vector coluna e o segundo linha, isto é se:

NLINHAS(**a**) > 1

corresponde ao algoritmo geral de multiplicação de matrizes descrito nos pontos seguintes. Como os vectores são sempre distribuídos como coluna, segundo a equação (5-14), tal deve ser tomado em consideração, justificando uma função diferente da que trata da multiplicação

de matrizes. De acordo com esta equação o que se pretende é calcular uma matriz **C** com tantas linhas como **b** e colunas como **a**:

C = b . a

isto é:

(5-17) $\mathbf{C}_{l,c} = \mathbf{b}_l \,.\, \mathbf{a}_c \quad \text{com } l = 0,1,2,...,m-1 \quad \text{e} \quad c = 0,1,2,...,m-1$

e para a partição armazenada no nó *x*:

$\mathbf{C}^x = \mathbf{b}^x \,.\, \mathbf{a}$

ou seja:

(5-18) $\mathbf{C}^x{}_{l,c} = \mathbf{b}^x{}_l \,.\, \mathbf{a}_c \quad \text{com } l = 0,1,2,...,m^x - 1 \quad \text{e} \quad c = 0,1,2..,m-1$

onde *m* representa o número de linhas de **b** e **C** e m^x representa o número dessas linhas armazenadas na partição *x*. É no entanto necessário que em cada nó exista a representação completa do vector **a**, o que obriga que este seja formado em todos os nós com uma operação collect, já descrita no capítulo anterior:

Prg (*x*):

$\cup \{\mathbf{a}\}$
$\mathbf{C}^x = \mathbf{b}^x \,.\, \mathbf{a}$

tabela 5-6: Diagrama de paralelização para o segundo caso de multiplicação de vectores

Pode-se agora escrever o algoritmo que deve ser executado em cada nó *x*:

algoritmo 5-9:

collect ($\mathbf{a}^x$, **a**)
desde $l = 0$ até $m^x - 1$
    desde $c = 0$ até $m - 1$

$$\mathbf{C}^x[l][c] = \mathbf{b}^x[l] * \mathbf{a}[c]$$

e que garante que o resultado **C**, encontra-se também optimamente distribuído.

No entanto a operação collect apresenta uma desvantagem, que se traduz em perda de eficiência. Em cada nó, para formar o vector completo **a**, a operação collect além de receber as partições dos outros nós deve copiar a partição armazenada nesse nó, $\mathbf{a}^x$, para a área correspondente de **a**. É assim definida a operação de comunicação collect2, idêntica a collect, excepto que não efectua a cópia da partição armazenada no próprio nó. Mas para isso o algoritmo anterior deve ser modificado de modo que aceda ao vector **a** para manipular elementos provenientes de outros nós e a $\mathbf{a}^x$ para os elementos locais:

$$\mathbf{a} = \begin{bmatrix} 0 \\ \vdots \\ li-1 \\ \vdots \\ lf+1 \\ \vdots \\ m-1 \end{bmatrix} \quad \text{e} \quad \mathbf{a}^x = \begin{bmatrix} li \\ \vdots \\ lf \end{bmatrix}$$

onde:

li = LINHA_INICIO(a, x)-1

e

lf = LINHA_FIM(a, x)-1

de modo a traduzir os índices das matrizes baseados em 1 para baseados em 0.

algoritmo 5-10:

collect2 ($\mathbf{a}^x$, **a**)
desde $l = 0$ até $m^x - 1$
    desde $c = 0$ até $li - 1$ $\mathbf{C}^x[l][c] = \mathbf{b}^x[l] * \mathbf{a}[c]$
    desde $c = li$ até lf $\mathbf{C}^x[l][c] = \mathbf{b}^x[l] * \mathbf{a}^x[c - li]$
    desde $c = lf+1$ até $m - 1$ $\mathbf{C}^x[l][c] = \mathbf{b}^x[l] * \mathbf{a}[c]$

Esta estratégia é também usada no produto de matrizes, de modo a optimizar não só o tempo de formação do operando completo num nó, mas também a evitar desperdício de memória.

Como um vector é sempre armazenado como sendo coluna, indicando o atributo TRANSPOSTO se é ou não linha, não existe necessidade de transpor vectores antes de chamar esta função para proceder à multiplicação de dois vectores linha ou coluna. No entanto, se o sistema de detecção de erros em tempo de execução estiver habilitado, será gerada uma mensagem de erro e o programa terminado. É assim necessário "enganar" este sistema antes de proceder à operação. Tal pode ser feito invertendo o atributo TRANSPOSTO de um vector antes de proceder à multiplicação e voltando a restaurá-lo após esta. Este artificio pode ser efectuado sobre o primeiro operando, o segundo, ambos ou nenhum, que é o que as macros seguintes efectuam:

```
VMULTt (a, b, C)    = =   TRANSPOR (a)
                          vmult (a, b, &C)
                          TRANSPOR (a)

VMULT_t (a, b, C)   = =   TRANSPOR (b)
                          vmult (a, b, &C)
                          TRANSPOR (b)

VMULTtt (a, b, C)   = =   TRANSPOR (a)
                          TRANSPOR (b)
                          vmult (a, b, &C)
                          TRANSPOR (a)
                          TRANSPOR (b)

VMULT (a, b, C)     = =   vmult (a, b, C)
```

Como se pode observar a inversão ou negação do atributo TRANSPOS, que na realidade é um valor lógico, é efectuada recorrendo a macro TRANSPOR já definida em 5.3.2.1.

5.3.2.4.2 Multiplicação de matrizes

Sejam as matrizes **A** ($m \times p$), **B** ($p \times n$) e **C** ($p \times n$) com as dimensões dadas por m, n e p. O produto:

(5-19) $\mathbf{C} = \mathbf{A} \,.\, \mathbf{B}$

define-se como:

(5-20) $\mathbf{C}_{l,c} = \mathbf{A}_{l,k} \,.\, \mathbf{B}_{k,c}$ com $l = 1,2,...,m \quad c = 1,2,...,n$ e $k = 1,2,...p$

e para a partição armazenada no nó x:

(5-21) $\mathbf{C}^x = \mathbf{A}^x \mathbf{.\ B}$

ou, se m^x representar o número de linhas de **A** e **C** armazenadas no nó x:

(5-22) $\mathbf{C}^x{}_{l,c} = \mathbf{A}^x{}_{l,k} \mathbf{.B}_{k,c}$ com $l = 1,2,...,m^x \quad c = 1,2,...,n$ e $k = 1,2,...p$

visto que as matrizes são distribuídas por linhas. É no entanto necessário que em cada nó x o operando esquerdo **B** esteja completamente armazenado.

(5-23) $$\begin{bmatrix} \mathbf{C}^0 \\ \cdots \\ \mathbf{C}^1 \\ \cdots \\ \vdots \\ \cdots \\ \mathbf{C}^{np-1} \end{bmatrix} = \begin{bmatrix} \mathbf{A}^0 \\ \cdots \\ \mathbf{A}^1 \\ \cdots \\ \vdots \\ \cdots \\ \mathbf{A}^{np-1} \end{bmatrix} .\mathbf{B} \quad \text{com}$$

$$\mathbf{C}^x = \begin{bmatrix} \mathbf{C}_{\text{LINHA\_INICIO}(x),*} \\ \mathbf{C}_{\text{LINHA\_INICIO}(x)+1,*} \\ \vdots \\ \mathbf{C}_{\text{LINHA\_FIM}(x),*} \end{bmatrix} \quad \text{e} \quad \mathbf{A}^x = \begin{bmatrix} \mathbf{A}_{\text{LINHA\_INICIO}(x),*} \\ \mathbf{A}_{\text{LINHA\_INICIO}(x)+1,*} \\ \vdots \\ \mathbf{A}_{\text{LINHA\_FIM}(x),*} \end{bmatrix}$$

Assim, um diagrama de paralelização possível será:

Prg (x):

$\cup \{\mathbf{B}\}$

$\mathbf{C}^x = \mathbf{A}^x \mathbf{.\ B}$

5-7: Primeiro diagrama de paralelização para o produto matricial

Visto que é necessário obter o operando **B** completo em todos os nós, é possível simplificar a operação:

(5-24) $\mathbf{C = A \,.\, B^T}$

isto é o produto matricial com prévia transposição do operando direito. Assim em vez de se efectuar uma operação de transposição sobre **B** antes do produto matricial, basta que os índices deste operando sejam trocados na expressão (5-22):

(5-25) $\mathbf{C}^x{}_{l,c} = \mathbf{A}^x{}_{l,k} \,.\, \mathbf{B}_{c,k}$ com $l = 1,2,...,m^x$ $c = 1,2,...,n$ e $k = 1,2,...p$

São pois definidas duas funções: mmult (**A**, **B,** **C**) e mmult_t (**A**, **B,** **C**), as quais correspondem às operações, $\mathbf{C = A \,.\, B}$ e $\mathbf{C = A \,.\, B^T}$.

Objectivo: mmult: Retorna em **C** o produto da matriz **A** pela matriz **B**
mmult_t: Retorna em **C** o produto da matriz **A** pela transposta da matriz **B**

PRE: (! ISVECTOR (**A**) && ! ISVECTOR (**B**)) &&
NCOLUNAS(**A**) == NCOLUNAS(**B**)

POS:

mmult:

C = = **A** * **B**
NLINHAS(**C**) = = NLINHAS(**A**) &&
NCOLUNAS(**C**) = = NCOLUNAS(**B**)

mmult_t:

C = = **A** * $\mathbf{B^T}$
NLINHAS(**C**) = = NLINHAS(**A**) &&
NCOLUNAS(**C**) = = NLINHAS(**B**)

Excepção: Se não houver memória suficiente para alocar *buffer* temporário, gera um erro em tempo de execução.

Tipo: II A

matrix mmult (matrix **A**, matrix **B**, matrix* **C**)
matrix mmult_t (matrix **A**, matrix **B**, matrix* **C**)

Como estas funções não foram desenhadas para operar com vectores, o primeiro grupo de pré-condições encarregam-se de indicar isso, através da macro ISVECTOR, que retorna 1 se o operando for um vector linha ou coluna. O segundo grupo indica que o número de colunas do operando esquerdo, **A**, é igual ao número de linhas do operando direito, **B**, condição necessária para que seja definido o produto de matrizes. As pós-condições indicam que as dimensões da matriz resultado devem ser NLINHAS(**A**) × NCOLUNAS(**B**). No caso de mmult_t, como se assume que **B** será operado transposto, isso é tomado em consideração.

No entanto, formar uma matriz completa é um gasto de memória que deve ser evitado. Assim uma versão alternativa dos algoritmos anteriores, denominada versão 2, será apresentada.
Como o produto matricial, que não é mais que uma operação produto interno para obter cada elemento da matriz resultado, pode-se criar um *buffer* que receba apenas um vector completo de cada vez, para o cálculo de cada elemento. Assim o produto matricial definido em termos de produto interno será:

(5-26) $\mathbf{C}^x{}_{l,c} = \mathbf{A}^x{}_{l,*} \,.\, \mathbf{B}_{*,c}$ com $c = 1,2,...,n \qquad l = 1,2,...,m^x$

onde . indica a operação produto interno, já definida no ponto 5.3.2.4.1.
Pode-se assim percorrer a matriz resultado por colunas, recebendo em cada nó, num *buffer* temporário, o vector coluna correspondente de **B**, e operá-lo com todas as linhas armazenadas nesse nó. A opção de percorrer a matriz por colunas em vez de linhas, evita que o mesmo vector coluna completo $\mathbf{B}_{*,\,c}$ seja formado em cada nó *x*, tantas vezes como o número de linhas de $\mathbf{A}^x$, mas sim apenas uma vez. Então o diagrama de paralelização, considerando os índices das matrizes baseados em zero, será:

Prg (*x*):

desde $c = 0$ até n-1
- { $\mathbf{B}^x{}_{*,c}$ } ⇒
- { **tmp** } ⇐
- desde $l = 0$ até $m^x - 1$
  - $\mathbf{C}^x{}_{l,c} = \mathbf{A}^x{}_{l,*}$. **tmp**

tabela 5-8: Segundo diagrama de paralelização para o produto matricial

Como já foi atrás discutido, o vector coluna a enviar terá de ser copiado para um b*uffer*, visto que a primitiva send não permite enviar mensagens com elementos não consecutivos. Este é o caso de uma coluna de uma matriz, onde os elementos ao longo dessa são armazenados na memória separados por tantos elementos como o número de colunas. Assim primeiro devem ser copiados os elementos locais para um *buffer*. Para poupar mais cópias internas, estes podem ser directamente copiados para a sua posição no vector completo **tmp**, que guardará a coluna completa de **B.** Finalmente, estes devem ser enviados e devem ser recebidos os elementos remotos, nesse mesmo vector **tmp** usando a operação collect2. Assim, seja:

lib = LINHA_INICIO(**B**, *x*)-1

lfb = LINHA_FIM(**B**, *x*)-1

e

$\mathbf{tmp}^x = \mathbf{tmp}_{\,lib\,..\,lfb}$

de modo a que os índices sejam baseado em zero, o produto matricial pode ser definido como:

algoritmo 5-11:

```
desde c = 0 até n-1
        desde i = lib até lfb  tmp [i] = B [k-lib][c]
        collect2 (tmp^x, tmp)

        desde l = 0 até m^x – 1
                C^x [l][c] = A^x [l][0] . tmp [0]
                desde k = 1 até p - 1
                        C^x [l][c] = A^x [l][k] . tmp [k]
```

É pois necessário transferir uma matriz inteira para cada nó, se se considerar as cópias internas para **tmp**. Já no caso de mmult_t estas cópias internas não serão necessárias, sendo apenas transferida para cada nó a matriz remota, isto é o conjunto de matrizes parciais, não armazenadas no próprio nó.

Um algoritmo possível para mmult_t será calcular primeiro as parciais locais:

$\mathbf{C}^x_{\,*,\,lib\,..\,lfb} = \mathbf{A}^x \,.\, (\mathbf{B}^x)^{\mathrm{T}}$

e depois receber as remotas em cada processador para calcular o restante de $\mathbf{C}^x$:

$\mathbf{C}^x_{\,*,\;1\,..\,lib\,-\,1\;;\;lfb\,+\,1\,..\,ncb}$

onde:

ncb = NCOLUNAS (**B,** *x*) - 1

nlb = NLINHAS (**B,** *x*) – 1

Então um diagrama de paralelização possível será:

Prg (x):

$\mathbf{C}^x{}_{*,\ \text{LINHA\_INICIO}(\mathbf{B},x)\ ..\ \text{LINHA\_FIM}(\mathbf{B},x)} = \mathbf{A}^x \,.\, (\mathbf{B}^x)^T$	
desde c = 0 até nlb-1	
se (c >= lib e c <= lfb)	
{ $\mathbf{B}_{c,*}$ }	⇒
senão	
{ **tmp** }	⇐
desde l = 0 até $m^x - 1$	
$\mathbf{C}^x_{l,c} = \mathbf{A}^x_{l,*}$. **tmp**	

tabela 5-9: Primeiro diagrama de paralelização para mmult_t

A principal diferença para o diagrama de mmult apresentado na tabela 5-8, esquecendo a ordem com que as operações locais e remotas são efectuadas, pois na realidade o número de operações é o mesmo para mmult e mmult_t, consiste na forma como são comunicados os vectores temporários. No caso de mmult, pode-se considerar que **tmp** resulta de uma operação collect do vector $\mathbf{B}_{*,c}$. Isto é cada processador envia a sua parte $\mathbf{B}^x_{*,c}$, ao mesmo tempo que recebe as partes remotas. Deste modo numa rede homogénea, o tempo desta operação de comunicação é aproximado do tempo de comunicação de um vector com esse comprimento, dividido pelo número de nós da rede, já que se assume que as partes armazenadas em cada nó têm um comprimento idêntico. Já numa rede heterogénea, a premissa anterior não é verdadeira, e o tempo da operação será fortemente dependente do comprimento da maior partição e da largura de banda do canal por onde será enviada.

Utilizando o caso homogéneo para comparar ambas as operações, verifica-se que no caso de mmult_t, **tmp** é formado em cada nó recebendo o vector completo $\mathbf{B}_{c,*}$, excepto no nó que o tem alocado. Deste modo, se em ambos os casos de multiplicação **tmp** tiver o mesmo comprimento, e o tempo de comunicação desse vector seja T, em mmult a operação de comunicação será da ordem de $\frac{T}{np}$ ao passo que em mmult_t apenas T.

Além deste atraso, para receber um dada linha, é necessário indicar à primitiva receive qual o nó onde essa linha está armazenada, isto é o nó fonte:

descritor.MSG_SRC = **wrk_src** [*c*]
receive (*descritor*, **tmp**)

De modo que é necessário criar o vector **wrk_src**, com tantas linhas como **B,** o qual indica para cada linha qual o nó fonte, o que irá ainda mais degradar o desempenho de mmul_t.
De qualquer forma não se está a explorar o paralelismo em termos de recepção de mensagens. Mas se em vez de se alocar apenas o vector temporário **tmp**, for alocado um para cada nó **tmp**[*np*-1], será possível, se a topologia da rede o permitir, receber mensagens de todos os outros nós, no mesmo lapso de tempo, fazendo com que o tempo da operação de comunicação se aproxime de $\frac{T}{np}$.

Assim no diagrama de paralelização seguinte, assume-se que todos os nós enviam a sua matriz parcial linha a linha, em cada iteração em *c*. Em cada uma destas iterações cada nó recebe uma linha de todos os outro em simultâneo, e vai armazená-las no *buffer* correspondente ao nó fonte, **tmp** [$\bar{x}$], que pode ser qualquer um no espectro [0, *np*-1] com excepção do próprio nó *x*. Antes de terminar a iteração, cada um desses *buffers* irá ser operado com todas as linhas de **A** armazenadas no nó *x*.

Prg (*x*):

$\mathbf{C}^x_{*,\ \text{LINHA\_INICIO}(\mathbf{B},x)\ ..\ \text{LINHA\_FIM}(\mathbf{B},x)} = \mathbf{A}^x \,.\, (\mathbf{B}^x)^T$	
desde $c = 0$ até max (**nlb**)	
enquanto $c < nlb$ “#1”	
{ $\mathbf{B}_{c,*}$ }	⇒
enquanto $c <$ **nlb** [$\bar{x}$] “#2”	
{ **tmp** [$\bar{x}$] }	⇐
enquanto $c <$ **nlb** [$\bar{x}$] “#3”	
desde $l = 0$ até $m^x - 1$	
$\mathbf{C}^x_{l,c} = \mathbf{A}^x_{l,*}$. **tmp** [$\bar{x}$]	

tabela 5-10: Segundo diagrama de paralelização para mmult_t

Seja o vector **nlb**, com *x* elementos e que contém o número de linhas de **B** armazenadas em cada nó *x*. O número das iterações em *c* é dada pelo máximo de linhas de **B** armazenadas num

só nó. Assim é possível que um nó tenha menos linhas que outro, de forma que numa iteração esse nó não tem nenhuma linha para enviar. Da mesma forma nos outros nós não se deve receber nenhuma linha proveniente desse nó, nem operá-la com $\mathbf{A}^x$. Esse casos são validados pelas condições com etiquetas #1, #2 e #3.

A obtenção do vector **nlb**, implica que em cada nó devem ser identificadas todas partições de **B**. Também deve também ser criado um vector que indique qual a linha inicial da partição alocada em cada nó, de modo que o resultado de cada produto interno $\mathbf{A}^x_{l,*}$. **tmp** [$\bar{x}$], seja colocado no elemento correspondente de $\mathbf{C}^x_{,}$ embora o diagrama acima, não o indique explicitamente.

Tudo isto vai criar um atraso em relação a mmult. Desta forma mmult_t tem um desempenho ligeiramente inferior, em termos de tempo de execução, do que transpor **B** com mtrans e depois aplicar mmult:

MTRANS (**B**, **TMP**) **C =** MMULT_T (**A**, **B**, **C**)
C = MMULT (**A**, **B**, **C**)

como se pode verificar consultando as medidas de desempenho no ponto 5.4. No entanto, para operadores de grandes dimensões, o uso de mmult permite evitar alocar memória para a matriz temporária **TMP**, com a mesma dimensão de **B**, sendo o atraso desprezável.

Ambas as versões 1 e 2 destas operações serão comparadas mais à frente neste capítulo. Pode-se, no entanto, prever, para o caso da operação **C = A** . **B** que a versão 1, para tipos de processadores onde a implementação do ambiente de comunicações não segue exactamente o modelo teórico Mec-3, como é o caso do ADSP2106x, será mais eficiente que a versão 2. Isto porque na versão 1, cada nó envia a sua parte local da matriz B numa só vez, e não coluna a coluna, de modo que o atraso na formação e envio de pacotes é reduzido apenas a um por cada nó.

Por outro lado a descodificação de um endereço de um elemento do vector, neste caso o temporário **tmp**, requer menos operações elementares que no caso de um elemento de uma matriz. Assim na versão 1, a descodificação dos endereços de **B,** ao longo de cada coluna desta matriz, introduz um certo atraso. No caso de um processador onde a implementação do ambiente de comunicações segue muito de perto o modelo Mec-3, como é o caso do T8, a melhoria em termos de comunicações não é tão significativa, de modo que dificilmente se poderá obter um melhor desempenho do que com a versão 2.

Mesmo no caso do ADSP2106x, à medida que a ordem das matrizes aumenta, o que implica que um vector coluna passa a ter um comprimento cada vez maior, chegar-se-á a um ponto

onde a melhoria de desempenho nas comunicações introduzida pela versão 1 deixa de ser significativa comparada com a versão 2. A partir deste ponto o atraso no endereçamento dos elementos da matriz **B** dominará, traduzindo-se num desempenho inferior da versão 1 quando comparada com a versão 2.

Já no caso da operação **C = A** . $\mathbf{B}^T$, o acesso a elementos da matriz **B** é mais simples, pois ao assumir-se que B é transposto, tanto os elementos de A como os de B são acedidos ao longo das linhas. Deste modo, como para aceder ao próximo elemento não é necessário adicionar um *offset*, como é o caso do acesso a elementos em colunas, não existe um atraso tão pronunciado na descodificação dos endereço da versão 1 em relação à versão 2.

Convém salientar que o aumento de desempenho da versão 1 para a versão 2, nos casos em que se verifica, faz-se sempre à custa de uma maior utilização de memória.

Em termos de selecção da versão 1, em detrimento da versão 2 destas operações, é conseguida definindo na biblioteca as macros:

#define MMULT1
#define MMULTt1

Finalmente são definidas as macros:

MMULT (**A**, **B**, **C**) = = mmult (**A, B, &C**)
MMULT_t (**A**, **B**, **C**) = = mmult_t (**A, B, &C**)

5.3.2.4.3 Multiplicação entre matrizes e vectores

Incluí mais dois sub-casos. Quando o primeiro operando é um vector e o segundo uma matriz e o caso inverso. Para lidar com estes dois sub-casos são definidas duas funções: vmmult e mvmult respectivamente.

Quanto ao primeiro caso:

Objectivo: Retorna em **C** a multiplicação do vector linha **A** pela matriz **B**
PRE: (TRANSPOSTO (**a**) = = 0 && (
(NLINHAS (**a**) = = 1) &&
(NCOLUNAS (**a**) = = NLINHAS (**B**))) | |

TRANSPOSTO (**a**) = = 1 && (
(NCOLUNAS (**a**) **= =** 1) &&

(NLINHAS (**a**) = = NLINHAS (**B**))) &&

POS: **C** = = **a** * **B**
NCOLUNAS (**C**) = = 1 && NLINHAS (**C**) = = NCOLUNAS (**B**)

Excepção: Se não houver memória suficiente para alocar *buffer* temporário, gera um erro em tempo de execução.
Tipo: II A

matrix vmmult (matrix **a**, matrix **B**, matrix* **C**)

As pré-condições indicam que o operando esquerdo tem de ser um vector e o direito uma matriz. Ambos os grupos indicam as dimensões dos operandos no caso do operando direito ser ou não transposto.

Esta função é idêntica ao caso de multiplicação de um vector linha por um vector coluna, excepto que em vez do operando direito ser um vector coluna é uma matriz. Como esta matriz pode ser vista como uma composição de vectores colunas, a sua operação consiste em operar o vector linha com os NCOLUNAS(**B**) vectores colunas de **B**, de modo a obter cada elemento do vector linha resultado.

algoritmo 5-12:

desde *c* = 1 até NCOLUNAS (**B**)
    C $_{1, c}$ = vmult (**a**, **B** $_{*, c}$)

Quanto ao segundo caso:

Objectivo: Retorna em **C** a multiplicação da matriz **A** pelo vector coluna **b**
PRE: (TRANSPOSTO (**b**) = = 0 && (
(NCOLUNAS (**b**) **= =** 1) &&
(NCOLUNAS (**A**) = = NLINHAS (**b**))) | |

TRANSPOSTO (**b**) = = 1 && (
(NLINHAS (**b**) = = 1) &&
(NCOLUNAS (**A**) = = NCOLUNAS (**b**))) &&

POS: **C** = = **A** * **b**
NCOLUNAS (**C**) = = 1 && NLINHAS (**C**) = = NLINHAS (**A**)
Excepção: Se não houver memória suficiente para alocar *buffer* temporário, gera um erro em tempo de execução.
Tipo: II A

matrix mvmult (matrix **A**, matrix **b**, matrix* **C**)

As pré-condições indicam que o operando esquerdo é uma matriz e o direito um vector. Os dois grupos indicam as dimensões dos operandos e resultado no caso do operando esquerdo ser ou não transposto.

Prg (x):

$\cup \{\mathbf{b}\}$
$\mathbf{C}^{x}{}_{l,1} = \mathbf{A}^{x}{}_{l,*} . \mathbf{b}$

tabela 5-11: Diagrama de paralelização para mvmult

A recepção do operando esquerdo é efectuada de modo semelhante ao caso de multiplicação de um vector coluna com um linha. Cada elemento do vector coluna resultado é obtido fazendo o produto interno de cada linha da matriz operando direito com o operando esquerdo. Como o operando esquerdo já foi distribuído para todos os nós, estes produtos internos são operações locais, efectuadas no processador que contém as respectivas linhas do operando direito.

Como já foi visto para o caso da multiplicação de vectores, visto que um vector é sempre armazenado como sendo coluna, e o atributo TRANSPOSTO é que indica se é ou não linha, são definidas também duas macros para implementar os casos em que o vector é transposto.

VMMULT (**A**, **b**, **C**)	= =	vmmult (**A, b, C**)
VMMULTt (**A**, **b**, **C**)	= =	TRANSPOR (**A**) vmmult (**A, b, C**) TRANSPOR (**A**)
MVMULT (**A**, **b**, **C**)	= =	mvmult (**A, b, C**)
MVMULT_t (**A**, **b**, **C**)	= =	TRANSPOR (**b**) mvmult (**A, b, C**) TRANSPOR (**b**)

Em termos de desempenho vmmult terá um comportamento inferior a mvmult, pois é baseada na paralelização do produto interno, a qual requer mais comunicações.

5.3.2.5 Inversão de uma matriz

A inversão de uma matriz quadrada **A**, é efectuada pelo método de Gauss-Jordan. Nesta operação a matriz resultado terá as mesmas dimensões do operador e tal é indicado também

pelas pós-condições. Se não for possível alocar *buffers* temporários, a função aborta e é gerada uma mensagem de erro em tempo de execução.

Objectivo:	Retorna em **C** a inversa da matriz
PRE:	NCOLUNAS(**A**) = = NLINHAS(**A**)
POS:	$\mathbf{C}^{-1}$ = = **A** NCOLUNAS(**A**) = = NCOLUNAS(**C**) && NLINHAS(**A**) = = NLINHAS(**C**) &&
Excepção:	Se não houver memória suficiente para alocar *buffer* temporário, ou se a matriz **A** não for invertível, gera um erro em tempo de execução correspondente.
Tipo:	I A

matrix gjinv (matrix **A**, matrix* **C**)

O método de Gauss-Jordan, consiste na modificação de uma matriz de modo a tornar nulos todos os elementos situados fora da diagonal principal, e estes unitários. É um método iterativo com tantos passos como a ordem da matriz. Para inverter uma matriz segundo este método, a matriz identidade com a mesma ordem é justaposta à matriz a inverter:

$$[\ \mathbf{A} \mid \mathbf{I}\ ]$$

Se todas as operações efectuadas sobre **A**, de modo a diagonizá-la, forem também efectuadas sobre a **I**, após concluída a última iteração obtém-se em **A** a matriz identidade e em **I** a inversa $\mathbf{A}^{-1}$. Assim, seja **A** uma matriz quadrada de ordem n com determinante não nulo, em cada passo k as seguinte operações são efectuadas sobre **T** = [**A** | **I**]:

$$\mathbf{T}_{k,c}^{(k+1)} = \frac{\mathbf{T}_{k,c}^{(k)}}{\mathbf{T}_{k,k}^{(k)}} \qquad (5\text{-}27)$$

$$\mathbf{T}_{l,c}^{(k+1)} = \mathbf{T}_{l,c}^{(k)} - \mathbf{T}_{l,k}^{(k)}\,\mathbf{T}_{k,c}^{(k+1)} \qquad (5\text{-}28)$$

onde:

$k = 1, 2, \ldots n$

$l = 1, \ldots, k\text{-}1, k\text{+}1, \ldots, n$

$c = k\text{+}1, \ldots, n\text{+}1$

e

$\mathbf{A}_{k,k}^{(k)}$ = Elemento *pivot* para passo k

$\mathbf{T}_{k,*}^{(k+1)}$ = Linha *pivot* para passo k

Um caso especial ocorre quando um elemento *pivot* é nulo. Como não se pode obter a linha *pivot* dividindo a linha k por um *pivot* nulo deve ser efectuada uma manipulação adicional. Como é costume nos métodos de eliminação, pode-se trocar a linha em causa com uma outra em busca de um *pivot* não nulo. Se não for encontrado uma linha que satisfaça esta condição então o determinante da matriz é nulo e esta não admite inversa. Também é possível trocar colunas com o mesmo objectivo.

O algoritmo paralelo utilizado consiste numa modificação do apresentado em (Daniel e Baltazar, 1994) para a resolução de sistemas de equações lineares pelo mesmo método. Em termos de operandos distribuídos tem-se:

$\mathbf{T}' = [\ \mathbf{A}' \mid \mathbf{I}'\ ]$

e no final da iteração n:

$\mathbf{T}' = [\ \mathbf{I}' \mid (\mathbf{A}^{-1})'\ ]$

Segundo o algoritmo anterior, em cada passo k a coluna de **A** com o mesmo índice fica calculada. Deste modo, para um dado passo k apenas serão operados os elementos $\mathbf{A}_{*,\,k\,..\,n}$, o que mostra que a distribuição por linhas é a mais adequada para este algoritmo, pois todos os processadores operam sobre o mesmo número de colunas parciais, em cada iteração.

Para tirar partido desta distribuição, ao tratar o caso de *pivot* nulo, optou-se por trocar colunas até encontrar uma com *pivot* não nulo. Deste modo evita-se comunicar linhas entre nós diferentes, sendo a troca das colunas parciais efectuada concorrentemente em todos os nós.

Assim o algoritmo paralelo para cada passo k, obedece a uma determinada ordem de acontecimentos:

#1 Se necessário busca *pivot* não nulo e troca as colunas parciais de acordo.

#2 O nó que detêm a linha *pivot* resolve a equação (5-27) sobre essa linha.

#3 A linha *pivot* é enviada para todos os outros nós.

#4 Em todos os nós é aplicada a equação a todas as linhas menos a *pivot*.

Finalmente, após a conclusão do passo *n*, a ordem com que as colunas parciais foram trocadas é invertida de modo que as colunas de $\mathbf{A}^{-1}$ sigam a mesma ordem de **A**. Sendo:

lia = LINHA_INICIO (**A,** *x*)-1
lfa = LINHA_FIM (**A,** *x*)-1
lla = LLINHAS (**A,** *x*)

O diagrama de paralelização, para matrizes com índices baseados em zero, será:

Prg (*x*):

desde $k = 0$ até $n - 1$
    se ($k >= lia$ e $k <= lfa$) "se nó *x* contém linha *k (pivot)*"
        se não encontra *pivot* não nulo
            Matriz não é invertível
            fim
        $\mathbf{T}_{k,k..2n-1} = \frac{\mathbf{T}_{k,k..2n-1}}{\mathbf{A}_{k,k}}$
        { $\mathbf{T}_{k,*}$ } ⇒
    senão
        { **tmp** } ⇐

    desde $l = 0$ até $lla - 1$
        se ($l + lia \neq k$) "se não é linha *pivot*"
            desde $c = k$ até $2n - 1$
            $\mathbf{T}_{l,c}$ ' = $\mathbf{T}_{l,c}$ '– $\mathbf{T}_{l,k}$ '. $\mathbf{tmp}_{c}$

Reordena as colunas de $\mathbf{A}^{-1}$

tabela 5-12: Diagrama de paralelização para gjinv

Repare-se que para indicar um matriz parcial é utilizada a plica, para evitar confusão com o passo a que se refere a matriz. De resto no diagrama não é indicado o passo a que se refere uma dada matriz, como foi indicado nas expressões (5-27) e (5-28). É assumido que a matriz **T**, resultante de cada passo, vai ser operada no passo seguinte. Por outro lado, não há

necessidade de indicar a matriz parcial **T** para a obtenção da linha *pivot*, pois em cada passo *k* esta só existe num dado nó, e este cálculo é efectuado nesse nó.

Finalmente é utilizada uma pilha onde são colocadas todas as trocas de colunas efectuadas durante os *n* passos, na busca de um *pivot* não nulo. Para reordenar a matriz inversa essa pilha é consultada de modo que as trocas nela registadas são efectuadas com ordem inversa. Cada elemento da pilha é constituído por um par ordenado de escalares (*a*, *b*) onde *a* e *b* indicam quais as colunas trocadas num dado passo.

Como usual é definida a macro:

GJINV (**A**, **C**) = = gjinv (**A,** &**C**)

5.3.2.6 Integração das operações sobre tipos de dados diferentes

As funções apresentadas nos pontos seguintes fazem a integração das operações de adição, multiplicação e inversão sobre os tipos de dados escalares ou matriciais. Esta funções são também capazes de transpor operandos antes de efectuar a operação.

5.3.2.6.1 Integração das operações de multiplicação

Com esta função é possível calcular qualquer caso de multiplicação de matrizes, mesmo com transposição de operandos. Se não existir uma função que lide com o operando transposto, é criada uma matriz temporária com este transposto e chamada a função de multiplicação adequada. Esta função também pode ser usada para efectuar operações entre matrizes e escalares.

Objectivo: Agrupa todas as funções de multiplicação matricial com opção de transpor operandos. Retorna a multiplicação de **A** por **B**.
PRE: Depende da função chamada.
POS: **C** = = **A** * **B** | | **C** = = $\mathbf{A}^T$ * **B** | | **C** = = **A** * $\mathbf{B}^T$ | | **C** = = $\mathbf{A}^T$ * $\mathbf{B}^T$
O espaço ocupado por **C** é libertado.
Excepções: Se não houver memória suficiente para alocar *buffers* temporários gera um erro em tempo de execução.
Tipo: Não definido

matrix multw (matrix **A**, matrix **B**, matrix **C,** INTEGER *ea,* INTEGER *eb*,
INTEGER *at,* INTEGER *bt*)

Se forem verdadeiros, os parâmetros *ea* e *eb* indicam respectivamente que o primeiro ou o segundo operando são escalares. Já os parâmetros *at* e *bt* indicam respectivamente que o

primeiro ou o segundo operando devem ser transpostos, antes de executar a multiplicação. Convém referir que para efectuar divisões que envolvam escalares, é necessário inverter o operando escalar.

Quanto ao tipo de operandos, os casos possíveis estão agrupados na seguinte tabela, onde o sinal ? indica que o valor lógico da variável não interessa.

A (*ea*)	**B** (*eb*)	*at*	*bt*	Operações
escalar	escalar	?	?	**C**[0] = **A**[0] * **B**[0]
escalar	não escalar	?	0	**C** = NUMMULT (**B**, **A**[0], **C**)
		?	1	**T** = MTRANS (**B, T**) **C** = NUMMULT (**T**, **A**[0], **C**)
não escalar	escalar	0	?	**C** = NUMMULT (**A**, **B**[0], **C**)
		1	?	**T** = MTRANS (**A**, **T**) **C** = NUMMULT (**T**, **B**[0], **C**)
não escalar	não escalar	?	?	Consultar tabela 5-14

tabela 5-13: Operação de multw para diferentes tipos de operandos

E quando ambos os operandos não são escalares, último caso da tabela anterior:

A	**B**	*at*	*bt*	Operações
vector	vector	0	0	**C** = VMULT (**A**, **B**, **C**)
		0	1	**C** = VMULT_t (**A**, **B**, **C**)
		1	0	**C** = VMULTt (**A**, **B**, **C**)
		1	1	**C** = VMULTtt (**A**, **B**, **C**)
vector	matriz	0	0	***C** = VMMULT (**A**, **B**, **C**)*
		0	1	**T** = MTRANS (**B, T**) **C** = VMMULT (**A**, **T**, **C**)
		1	0	**C** = VMMULTt (**A**, **B**, **C**)
		1	1	**T** = MTRANS (**B, T**) **C** = VMMULTt (**A**, **T**, **C**)

tabela 5-14: Operação de multw para matrizes e / ou vectores

A	**B**	*at*	*bt*	Operações
matriz	vector	0	0	**C =** MVMULT (**A**, **B**, **C**)
		0	1	**C =** MVMULT_t (**A**, **B**, **C**)
		1	0	**T** = MTRANS (**A, T**) **C =** MVMULT (**T**, **B**, **C**)
		1	1	**T** = MTRANS (**A, T**) **C =** MVMULT_t (**T**, **B**, **C**)
matriz	matriz	0	0	**C =** MMULT (**A**, **B**, **C**)
		0	1	**C =** MMULT_t (**A**, **B**, **C**)
		1	0	**T** = MTRANS (**A, T**) **C =** MMULT (**T**, **B**, **C**)
		1	1	**T** = MTRANS (**A, T**) **C =** MMULT_t (**T**, **B**, **C**)

tabela 5-14 (cont.): Operação de multw para matrizes e / ou vectores

Como se pode observar em 16 casos possíveis 12 são resolvidos pelas funções já implementadas. Apenas em 6 será necessário recorrer à função de transposição MTRANS. Além dos casos já referidos, se um dos operadores for uma matriz com apenas um elemento, este é distribuído para todos os nós e tratado como um escalar, sendo utilizada a função NUMMULT. Por outro lado, se ambos os operandos forem matrizes com apenas um elemento, será utilizada a função MMULT. Deste modo garante-se que a matriz resultado terá as dimensões correctas em qualquer dos casos.

Para verificar se um operando matricial **M** é um vector, pode ser usada a macro:

INTEGER ISVECTOR (**M**)

que retorna verdade, isto é 1, se o número de linhas for igual a 1 e o de colunas maior que 1, ou no caso inverso. Para identificar se uma matriz contém apenas um elemento e pode ser convertida num escalar com matrix_get, como já foi abordado no ponto 5.3.2.4.1, deve ser usada:

INTEGER ISSCALAR (**M**)

Foi também definida uma macro que reconhece um operando matricial com mais de uma linha e uma coluna:

INTEGER ISMATRIX (**M**)

Por questões de coerência da biblioteca também foi criada a macro:

MULTW (**A**, **B**, **C**) = = multw (**A, B, C**)

Convém ainda referir que o espaço ocupado por **C** é libertado. É assim possível atribuir dados a um identificador e libertar a memória reservada a este num só passo:

C = MULTW (**A, B, C**)

isto é o equivalente a:

CLEAR_MATRIX(**C**) "após a conclusão **C =** NULL"
C = MULTW (**A, B, C**)

No entanto se o identificador a atribuir o resultado for diferente do 2º argumento:

C0 = MULTW (**A, B, C1**)

O espaço ocupado por **C1** é libertado, mas nada lhe é atribuído ao passo que o espaço alocado a **C0** antes da atribuição é perdido. Por isso este caso nunca deve suceder.
Esta característica aplica-se também aos integradores apresentados a seguir, ADDW e INVW.

5.3.2.6.2 Integração das operações de adição

Do mesmo modo que a anterior, esta função integra todas as operações de adição ou subtracção, entre os diferentes tipos de dados. Como não são previstas funções que somam vectores com matrizes estes casos não são considerados.

Objectivo: Agrupa todas as funções de adição com opção de transpor operandos. Retorna a soma ou subtração de **A** por **B**.
PRE: Depende da função chamada.

POS: **C** = = **A** ± **B** | | **C** = = $\mathbf{A}^T$ ± **B** | | **C** = = **A** ± $\mathbf{B}^T$ | | **C** = = $\mathbf{A}^T$ ± $\mathbf{B}^T$
O espaço ocupado por **C** é libertado.
Excepções: Se não houver memória suficiente para alocar *buffers* temporários gera um erro em tempo de execução.
Tipo: Não definido

matrix addw (matrix **A**, matrix **B**, matrix **C,** INTEGER *opadd*, INTEGER *ea,* INTEGER *eb*, INTEGER *at,* INTEGER *bt*)

Se forem verdadeiros, os parâmetros *ea* e *eb* indicam respectivamente que o primeiro ou o segundo operando são escalares. Já os parâmetros *at* e *bt* indicam respectivamente que o primeiro ou o segundo operando devem ser transpostos, antes de executar a operação. A operação a efectuar é indicada por *opadd*. Se este for verdadeiro indica que a operação é uma adição, senão é uma subtracção. No caso de escalares pode ser efectuada uma subtracção simplesmente fornecendo o simétrico do operando escalar.
Quanto ao tipo de operandos, os casos possíveis estão agrupados na seguinte tabela:

A (*ea*)	**B** (*eb*)	*at*	*bt*	Operações
escalar	escalar	?	?	**C**[0] = **A**[0] + **B**[0]
escalar	não escalar	?	0	**C** = NUMADD (**B**, **A**[0], **C**)
		?	1	**T** = MTRANS (**B, T**) **C** = NUMADD (**T**, **A**[0], **C**)
não escalar	escalar	0	?	**C** = NUMADD (**A**, **B**[0], **C**)
		1	?	**T** = MTRANS (**A**, **T**) **C** = NUMADD (**T**, **B**[0], **C**)
não escalar	não escalar	0	0	**C** = MADD (**A**, **B**, **C**)
		0	1	**T** = MTRANS (**B, T**) **C** = MADD (**A**, **T**, **C**)
		1	0	**T** = MTRANS (**A, T**) **C** = MADD (**T**, **B**, **C**)
		1	1	**T1** = MTRANS (**A, T1**) **T2** = MTRANS (**B, T2**) **C** = MADD (**T1**, **T2**, **C**)

tabela 5-15: Operação de addw para diferentes tipos de operandos

Se *opadd* não for verdadeiro a operação escalar + deve ser substituída por -, a mista NUMADD por NUMSUB e a matricial MADD por MSUB.

Finalmente é definida a macro:

ADDW (**A**, **B**, **C**) = = addw (**A, B, C**)

5.3.2.6.3 Integração das operações de inversão

Na realidade esta operação não necessita de integração, sendo que a única utilidade desta função é transpor o operando a inverter antes da operação, se tal for requerido.

Objectivo: Dá à função de inversão a opção de transpor o operando antes de inverte-lo.
PRE: Idênticas a gjinv
POS: $\mathbf{C}^{-1}$ = = **A** | | $(\mathbf{C}^{-1})^{T}$ = = **A**
O espaço ocupado por **C** é libertado.
Excepções: Se não houver memória suficiente para alocar *buffers* temporários gera um erro em tempo de execução.
Tipo: Não definido

matrix invw (matrix **A**, matrix **C,** INTEGER *ea,* INTEGER *at*)

Se for verdadeiro o parâmetro *ea* indica que o operando é escalar, e é retornado o inverso deste. Caso contrário, será efectuada a operação gjinv, com transposição de **A**, se *at* for verdadeiro.

A (*ea*)	*at*	Operações
escalar	?	$\mathbf{C}[0] = \frac{1}{\mathbf{A}[0]}$
não escalar	0	**C =** GJINV (**A**, **C**)
	1	**T** = MTRANS (**A**, **T**) **C =** GJINV (**T**, **C**)

tabela 5-16: Operação de invw para diferentes tipos de operandos

É também definida a macro

INVW (**A**, **C**) = = invw (**A, C**)

5.3.3 Biblioteca de cálculo geral (mcalc)

Nesta biblioteca residem funções que operam sobre vectores ou tabelas mas que não são necessariamente utilizadas na álgebra linear. Podem não estar enquadradas nos tipos acima descritos. Algumas destas funções são usadas para implementar comandos da linguagem SEQ 1.0, descrita no capítulo seguinte.

5.3.3.1 rotação de vectores

Objectivo: shfr e shfl rodam os elementos do vector **v**, da posição *i* à *f*, uma posição para a direita ou esquerda respectivamente, e na posição vaga inserem o escalar *n*.
PRE: (*i* > *f*) && (ISVECTOR (**v**) = = 1)
POS: shfr: Desloca elementos de $\mathbf{v}_{i..f}$ uma posição para a direita && $\mathbf{v}_i$= = *n*
shfl: Desloca elementos de $\mathbf{v}_{i..f}$ uma posição para a esquerda && $\mathbf{v}_f$==*n*
Tipo: Não definido

matrix shfr (matrix **v**, INTEGER *i*, INTEGER *f*, REAL *n*)
matrix shfl (matrix **v**, INTEGER *i*, INTEGER *f*, REAL *n*)

Na realidade estas funções também são funções de comunicação, que transferem um elemento entre cada nó. Seja o vector **v** com *ve* elementos:

$$\mathbf{v} = [\mathbf{v}_1 .. \mathbf{v}_{ve}]$$

particionado por *np* nós:

$$\lfloor 1 \cdots \mathbf{v}_{p_0} \rfloor \lfloor \mathbf{v}_{p_0+1} \cdots \mathbf{v}_i \cdots \mathbf{v}_{p_1} \rfloor \lfloor \mathbf{v}_{p_1+1} \cdots \mathbf{v}_{p_2} \rfloor \lfloor \mathbf{v}_{p_2+1} \cdots \mathbf{v}_f \cdots \mathbf{v}_{p_3} \rfloor \cdots \lfloor \mathbf{v}_{p_{np-2}+1} \cdots \mathbf{v}_{p_{np-1}} \rfloor$$

com $i \ni [p_0+1, p_1]$ e $f \ni [p_2+1, p_3]$

onde o elemento final da partição *x* é dado por p_x.
No caso de shfr, os elementos nas partições entre as que contém o elemento *i* e o *f*, são deslocados uma posição para a direita e o último elemento dessa partição é transferido para a partição seguinte. O elemento da posição *i* passa a ter o valor *n*. O elemento da posição *f* é eliminado.

$$\lfloor 1 \cdots \mathbf{v}_{p_0} \rfloor \lfloor \mathbf{v}_{p_0+1} \cdots n, \mathbf{v}_i \cdots \mathbf{v}_{p_{1-1}} \rfloor \lfloor \mathbf{v}_{p_1} \cdots \mathbf{v}_{p_{2-1}} \rfloor \lfloor \mathbf{v}_{p_2} \cdots \mathbf{v}_{f-1} \cdots \mathbf{v}_{p_3} \rfloor \cdots \lfloor \mathbf{v}_{p_{np-2}+1} \cdots \mathbf{v}_{p_{np-1}} \rfloor$$

com $i \in [p_0+1, p_1]$ e $f \in [p_2+1, p_3]$; $\mathbf{v}_i = n$; $\mathbf{v}_f$ é eliminado

No caso de shfl tudo se passa ao contrário. O elemento *i* é o que será eliminado, a posição *f* passa a ter o valor *n*, e os elementos entre *i* e *f*, são deslocados para a esquerda.
Se se tratar de um vector coluna em vez de um vector linha shfr e shfl devem ser interpretados com deslocação para baixo e para cima respectivamente.
Mais uma vez são definidas duas macros:

SHFR(**v**,*i*,*f*,*n*) = = shfr(**v**,*i*,*f*,*n*)
SHFL(**v**,*i*,*f*,*n*) = = shfl(**v**,*i*,*f*,*n*)

cuja utilidade consiste apenas numa homogeneização das macro da biblioteca, visto que na realidade apenas passam os operandos para a função correspondente.

5.3.3.2 Comparação de matrizes

As seguintes funções implementam alguns testes sobre operandos matriciais.

5.3.3.2.1 iseqmtx

Objectivo: Compara se duas matrizes são absolutamente idênticas.
PRE:
POS: retorna 0 sse **A** != **B**
Excepções: Se não houver memória suficiente para alocar *buffer* temporário gera um erro em tempo de execução.
Tipo: IV

REAL iseqmtx (matrix **A**, matrix **B**)

Compara as matrizes **A** e **B** em termos de dimensão e depois elemento a elemento. Se **A** e **B** apontarem para a mesma matriz retorna 1. Se se pretender determinar se ambos os operadores são na realidade o mesmo, deve-se usar **A = = B**.
A comparação elemento a elemento é efectuada localmente. Se um dos elementos, com a mesma posição em ambas as matrizes, for diferente o resultado local é 1, senão 0. O resultado lógico local é então enviado para todos os nós, de modo que a comparação global possa ser efectuada em todos.

O vector **r**, com tantos elementos como o número de nós, guarda em cada nó, os resultados locais de todos os nós. O resultado local corresponde portanto a **r** [*x*]. Nas outras posições são recebidos os resultados remotos.

Prg (*x*):

se existem elementos diferentes em **A** e **B** **r** [*x*] = 1	
senão **r** [*x*] = 0	
{ **r** [*x*] }	⇒
{ **r** [$\bar{x}$] }	⇐
retorna ! $\sum_{k=0}^{k<n\_wrk} \mathbf{r}_k$	

tabela 5-17: Diagrama de paralelização da comparação de matrizes

Somando todos os elementos deste vector obtêm-se 0 se não existe nenhum elemento diferente em ambos os operandos. Um valor maior que zero indica que pelo menos um elemento diferente foi encontrado.
O retorno da função é precisamente a negação desta soma.

A seguinte macro converte o resultado para um inteiro:

ISEQMTX(**A**, **B**) = = (INTEGER) iseqmtx (**A**, **B**)

5.3.3.2.2 issqrmtx

Objectivo: Verifica se uma matriz é quadrada
PRE:
POS: retorna 0 se NLINHAS (**A**) **=** = NCOLUNAS (**A**)
Tipo: III

REAL issqrmtx (matrix **A**)

A seguinte macro converte o resultado para um inteiro:

ISSQRMTX(**A**) = = (INTEGER) mtxcmp (**A**)

Nesta biblioteca poderiam ainda ser incluídas outras operações, tais como operações sobre polinómios armazenados em vectores distribuídos. Mas como o primeiro caso de teste, o

algoritmo de controlo AGPC paralelo requer algumas destas funções, estas serão nesse capítulo tratadas, de modo a ilustrar também como adicionar funções definidas pelo utilizador, às já existentes nas bibliotecas de cálculo.

5.3.4 Adição de funções definidas pelo utilizador (mcalcext)

Como já foi visto as funções dividem-se em 4 tipos principais, onde dois destes são ainda subdivididos em 2 sub-tipos. Assim tem-se 6 sub-tipos:

Identificador de tipo	Tipo
0	Tipo I A
1	Tipo I B
2	Tipo II A
3	Tipo II B
4	Tipo III
5	Tipo IV

tabela 5-18: Identificação interna dos tipos de funções

Para que o utilizador destas bibliotecas possa adicionar funções por ele definidas, de modo a expandir estas bibliotecas, as funções devem obedecer a um dos tipos acima definidos. O seu protótipo deve ser adicionado ao ficheiro "mcalcext.h" e a definição a "mcalcext.c". Tal também pode ser feito através do ambiente de programação AIDA, descrito no capítulo 7.

No desenvolvimento de funções pelo utilizador, podem ser usadas todas as funções de atribuição e cópia de matrizes, bem como as macros para acesso aos atributos das matrizes, e as funções de introdução e extracção de elementos de uma matriz: matrix_put e matrix_get, em suma as facilidades descritas no ponto 5.1. O mesmo é válido para as funções de definição e dimensionamento de matrizes, desde que não operem sobre os atributos dos argumentos de entrada ou saída, que com já foi visto devem ser definidos pelo código cliente, antes da chamada à função. Pode também ser usado o sistema de detecção de erros em tempo de execução. Nos casos de teste apresentados, será mostrado em pormenor como adicionar novas funções à biblioteca.

5.4 Desempenho das funções de cálculo básicas

Neste ponto é indicado o desempenho das principais operações de álgebra linear, quando executadas nas famílias de processadores utilizados. Através destas medidas será possível ter uma aproximação do desempenho de um algoritmo matricial.

A estratégia de alocação de memória utilizada é a que permite um maior desempenho, para cada família de processadores, considerando que operadores desta dimensão só podem estar armazenados em bancos de memória com capacidade suficiente, e que já foi descrita no capítulo 2. São utilizadas topologias compostas por três processadores idênticos, o que significa que no caso óptimo o desempenho de operações de vírgula flutuante deverá ser o triplo do observado num só processador.

Operando A	**Operando B**	**A * B**	**A + B**	**A^{-1}**
Escalar	-	-	-	1
Escalar	Escalar	1	1	-
Escalar	Matriz ($m \times n$)	$m \,.\, n$	$m \,.\, n$	-
Matriz ($m \times n$)	Matriz ($m \times n$)	-	$m \,.\, n$	-
Matriz ($m \times n$)	Matriz ($n \times p$)	$m \,.\, p \,.\, (2\,n - 1)$	-	-
Matriz ($m \times m$)	-	-	-	$m\,(m+1)\,[1+2(m-1)]$

tabela 5-19 - Operações de virgula flutuante requeridas para efectuar as operações de multiplicação e adição matriciais

No entanto estas operações são formadas essencialmente por ciclos, e requerem a descodificação dos endereços dos elementos da matriz, o que irá implicar um certo atraso. Além disso, algumas operações requerem transferência de dados entre nós, provocando ainda mais degradação no desempenho, quando comparado com cálculo de vírgula flutuante puro. Na tabela acima está indicado o número *flops* necessário para efectuar uma operação dada as dimensões dos operandos. Estas estão indicadas entre parênteses a seguir ao tipo de operando. Este cálculo é baseado nos algoritmos descritos nos pontos 5.3.2.3, 5.3.2.4.2 e 5.3.2.5 respectivamente para a adição, multiplicação e inversão de matrizes. Nos primeiros dois casos as expressões são facilmente obtidas, a partir destes algoritmos. No caso da inversão, visto que existe uma grande variedade de métodos de a obter, é conveniente indicar que a expressão seguinte se refere apenas ao método de Gauss-Jordan:

(5-29) $$2\left[\sum_{k=1}^{m} k + \sum_{k=1}^{m} 2k(m-1)\right]$$

onde o primeiro somatório indica o número de operações de virgula flutuante para a linha *pivot*, uma por cada elemento da linha, sobre tantas colunas como o passo *k*. O número de passos necessários para completar a inversão é dado por *m*, isto é, a ordem da matriz quadrada. O segundo somatório indica o número de operações para as *m*-1 restantes linhas da matriz, que são duas por cada elemento. O número de colunas a operar em cada passo varia sempre com este, isto é de *m* até 1.

Ambas estas parcelas são multiplicadas por 2, pois os cálculos aplicam-se à matriz a transpor e à matriz identidade da mesma ordem *m*. Simplificando (5-29) chega-se à expressão da tabela anterior:

$$2\sum_{k=1}^{m} k\left[1+2(m-1)\right] \;=\; 2\frac{m(m+1)}{2}\left[1+2(m-1)\right]$$

Outro factor de atraso a ter em conta, consiste na etapa de identificação de partições, que algumas funções executam antes de efectuar a operação propriamente dita, como é o caso de mtrans e gjinv e algumas operações de multiplicação. Este atraso será pouco significativo para matrizes com ordem elevada, mas tanto maior quanto maior o número de nós na rede.

Por outro lado, os ensaios realizados, permitiram verificar que o comportamento é linear, isto é o número de operações de vírgula flutuante, mais precisamente milhões de operações de vírgula flutuante por segundo ou MFlops/s, não é afectado pelo número de repetições da mesma operação. Assim as medidas apresentadas a seguir representam o tempo tomado por apenas uma repetição da operação em causa, se bem que em alguns casos, para obter uma melhor aproximação, visto que a resolução do relógio de tempo real de alguns processadores é de apenas 1 ms, tenham sido feitas várias iterações da mesma operação.

No ponto seguinte, antes de se apresentar o desempenho destes algoritmos, são formalizadas as medidas de desempenho de algoritmos paralelos mais comuns.

5.4.1 Medidas de desempenho de algoritmos paralelos

Normalmente são usadas duas medidas para analisar o desempenho de um algoritmo paralelo, designadas por aceleração ou *speedup* e eficiência (Codenotti e Leoncini, 1992).

5.4.1.1 Aceleração

Esta medida de desempenho indica um factor pelo qual o tempo de execução de um algoritmo é reduzido quando mapeado sobre uma rede paralela, em comparação com a execução sobre um só processador. O valor ideal de aceleração é igual ao número de processadores sobre os quais está mapeado o algoritmo. Assim, considerando um algoritmo paralelo mapeado sobre uma máquina com *p* processadores, onde Ts é o tempo de execução num só processador e Tp (*p*) indica o tempo de execução do algoritmo em *p* processadores, a aceleração é expressa por:

$$\text{(5-30)} \quad \text{Aceleração} = \frac{\text{Ts}}{\text{Tp}\,(p)}$$

Desta forma pode-se medir o desempenho de uma algoritmo paralelo. No entanto pode-se querer obter o desempenho absoluto, para uma mesma máquina entre o melhor algoritmo sequencial conhecido e um dado algoritmo paralelo. Neste caso Ts é o tempo de execução do melhor algoritmo sequencial conhecido e Tp (*p*) o tempo de execução do algoritmo paralelo em *p* processadores.

No caso dos algoritmos matriciais paralelos automaticamente gerados pelo SPAM, Ts, refere-se ao tempo de execução do algoritmo paralelo num só processador. De acordo com a estratégia de particionamento utilizada, o algoritmo é idêntico independentemente do número de nós de processamento utilizados, simplesmente o volume de dados sobre o qual cada nó opera é tanto menor quanto maior for o número de nós sobre o qual o algoritmo é mapeado.
Neste ponto convém lembrar que o SPAM cria um determinado número de instancias da aplicação, sendo cada uma destas atribuída a um nó de uma rede virtual. É assim possível que estes nós possam ser alocados cada um num processador físico ou toda a rede seja alocada no mesmo processador. Para efeitos de medidas de desempenho é, no entanto, assumido que cada nó está alocado num processador independente.

5.4.1.2 Eficiência

O valor de eficiência indica a percentagem de tempo em que um algoritmo paralelo está a produzir cálculo numérico. A percentagem complementar indica uma perda de desempenho em tarefas necessária à execução de um algoritmo paralelo, tais como comunicação de dados entre processadores. Além disso, esta perda de eficiência será ainda mais acentuada, quanto menor for o segmento do algoritmo sequencial paralelizado. Assim, o valor de eficiência que

indica um máximo desempenho é 100%, independentemente do número de processadores existentes na rede paralela. O valor desta medida, para *p* processadores, pode ser expresso como:

$$\text{(5-31)}\quad \text{Eficiência}\ (p) = \frac{\text{Ts}}{\text{Tp}\,(p)\,.\,p} \times 100\,\%$$

que pode ser relacionado com o aceleração por:

$$\text{(5-32)}\quad \text{Eficiência}\ (p) = \frac{\text{Aceleração}}{p} \times 100\,\%$$

5.4.1.3 Algumas considerações sobre as medidas de desempenho

A expressão anterior, indica como obter os valores de eficiência no caso de redes homogéneas. No entanto, se a rede for constituída por processadores de diferentes tipos, está-se em presença de uma rede heterogénea, onde diferentes processadores tem diferentes desempenhos, e desta forma a expressão (5-31) não se pode aplicar. Neste caso, assumindo que as partições dos operandos matriciais são feitas sobre conjuntos de linhas consecutivas, como é o caso do esquema de particionamento usado nesta tese, o valor da eficiência poderá ser obtido de acordo com o seguinte procedimento:

Seja uma rede constituída por *p* processadores, onde a função $T(K(x), R_o(x))$ retorna o tempo de execução requerido para o processador tipo $K(x)$ resolver o algoritmo sobre $R_o(x)$ linhas. Considerando um equilíbrio de carga óptimo e que o tempo de comunicação é nulo, o tempo de computação paralelo óptimo é dado por:

$$\text{(5-33)}\quad T_o = \max\left[T\left(K(1), R_o(1)\right), \cdots, T\left(K(p), R_o(p)\right)\right]$$

É conveniente tomar em conta que um equilíbrio de carga óptimo não implica que o tempo de processamento seja igual em todos os processadores. Este tempo é medido sobre conjuntos de linhas e é comum que o tempo de processamento de uma linha num processador não seja um múltiplo exacto do tempo de processamento dessa mesma linha num processador doutro tipo. De qualquer modo, é assumido que para grandes volumes de dados, os valores óptimos de

$T(K(x), R_o(x))$ são muito próximos, minimizando a perda de eficiência devida a estas variações, para valores próximos de zero. Nestas condições, a eficiência da rede heterogénea pode ser obtida através da relação entre o tempo óptimo T_o, e o tempo medido, T_m:

$$(5\text{-}34)\quad \text{Eficiência } (p) = \frac{T_o}{T_m} 100\ \%$$

Aplicando o resultado de (5-34) em (5-32), obtém-se o valor de aceleração.

Esta medida é no entanto dependente da forma como as matrizes são particionadas pelos nós, isto é por linhas.

Tal como no caso de redes homogéneas, uma medida de eficiência absoluta e independente da forma como a matriz é particionada, para redes heterogéneas, é desejável. Esta pode ser obtida a partir do desempenho de um dado algoritmo em cada tipo de processador que compõe a rede. Após se obter esse desempenho sequencial, pode-se obter um factor de desempenho relativo ao processador mais rápido e que será sempre superior a 0 e igual ou inferior a 1:

$$(5\text{-}35)\quad 0 < \text{Fd}\,(x) = \frac{\text{Tempo de computação no processador mais rápido}}{\text{Tempo de computação no processador } x} \leq 1$$

O tempo de computação óptimo do algoritmo paralelo é dado pelo tempo de computação do algoritmo no processador mais rápido dividido pela soma dos factores de desempenho relativo de todos os processadores que compõem a rede:

$$(5\text{-}36)\quad T_o = \frac{\text{Tempo de computação no processador mais rápido}}{\sum_{x=0}^{p-1} \text{Fd}(x)}$$

A eficiência pode ser agora obtida comparando o tempo medido, T_m, com o tempo T_o, segundo a expressão (5-34).

Os resultados desta medida são muito semelhantes aos da anterior, mas enquanto a anterior assume que o particionamento mínimo é a linha enquanto esta assume que é o elemento da

matriz, de modo que no caso de particionamento por linhas, a eficiência medida pela primeira é ligeiramente superior. Esta diferença no entanto é apenas de algumas decimas percentuais.

Por outro lado, os valores óptimos destas medidas de desempenho, isto é 100% de eficiência e aceleração superior ao número de processadores, podem ser ultrapassados. Esta situação, designada por processamento super-escalar, tem a ver com a libertação de recursos dos processadores quando o algoritmo é distribuído pela rede. Um caso comum pode ser usado para ilustrar tal situação:

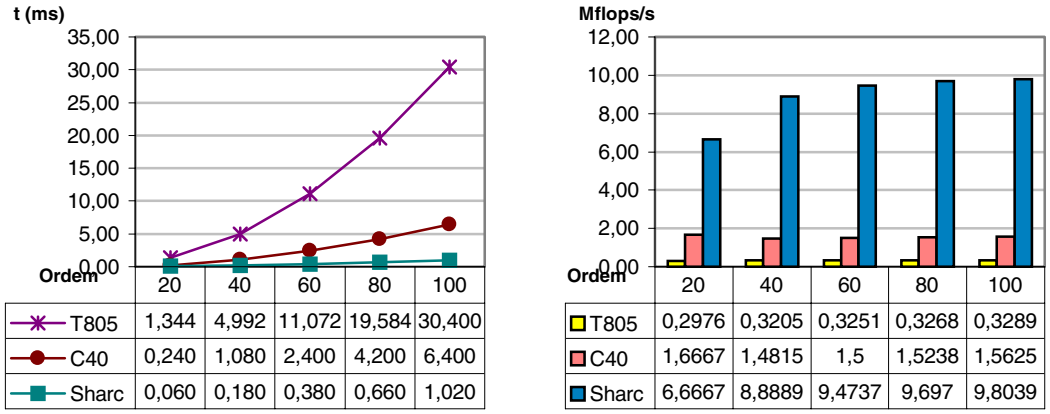
Seja uma rede homogénea composta por *p* processadores que contêm dois bancos de memória A e B, de tipos diferentes, com tempos de acesso T_a e T_b e dimensões M_a e M_b. Seja $T_a < T_b$ e $M_a < M_b$, tal que para o caso de um só processador, as necessidades de memória requeridas por um determinado algoritmo implicam armazenamento de dados no banco M_b. Considerando agora o caso da mesma rede, mas para $p > 1$ e admitindo a possibilidade de distribuir os dados apenas pelos bancos A de cada processador, o tempo de acesso à memória será reduzido em relação ao caso de $p = 1$. Será pois possível, se o tempo de comunicação não for significativo, ter uma desigualdade entre o tempo sequencial e o tempo paralelo invertida em relação ao caso sub-escalar, isto é:

(5-37) $Ts > Tp(p) * p$

o que permite atingir valores de eficiência superiores a 100 % e aceleração superior a *p*.

5.4.2 Adição e subtracção

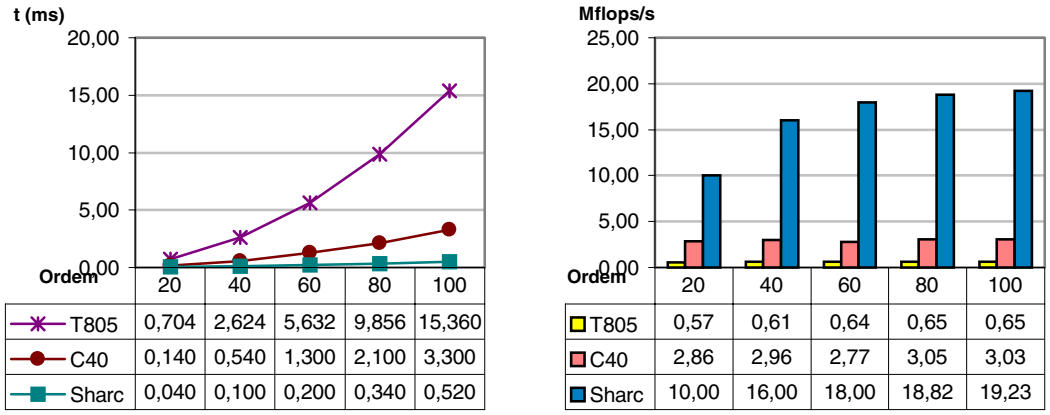
A primeira medida de desempenho apresentada consiste na adição ou subtracção de duas matrizes com as mesmas dimensões. Para todos ensaios são consideradas matrizes quadradas com ordens 20, 40, 60, 80 e 100. As figuras seguintes mostram, à esquerda, o desempenho em termos de tempo de execução da operação. À direita é apresentada uma medida que indica a relação entre o tempo de execução e o número de operações de vírgula flutuante indicados na tabela anterior. Devido aos atrasos já acima mencionados, para redes de 3 nós homogéneos, o desempenho deverá ser inferior a três vezes o desempenho de um só processador da mesma família, indicado no ponto 2.4, e que se pode resumir por 0,52 x 3 = 1,56 MFlops/s para a família T8xx, 4,15 x 3 = 12,45 para C4x, e 13.33 x 3 = 39.99 para ADSP2106x.



Ordem	20	40	60	80	100
T805	1,344	4,992	11,072	19,584	30,400
C40	0,240	1,080	2,400	4,200	6,400
Sharc	0,060	0,180	0,380	0,660	1,020

Ordem	20	40	60	80	100
T805	0,2976	0,3205	0,3251	0,3268	0,3289
C40	1,6667	1,4815	1,5	1,5238	1,5625
Sharc	6,6667	8,8889	9,4737	9,697	9,8039

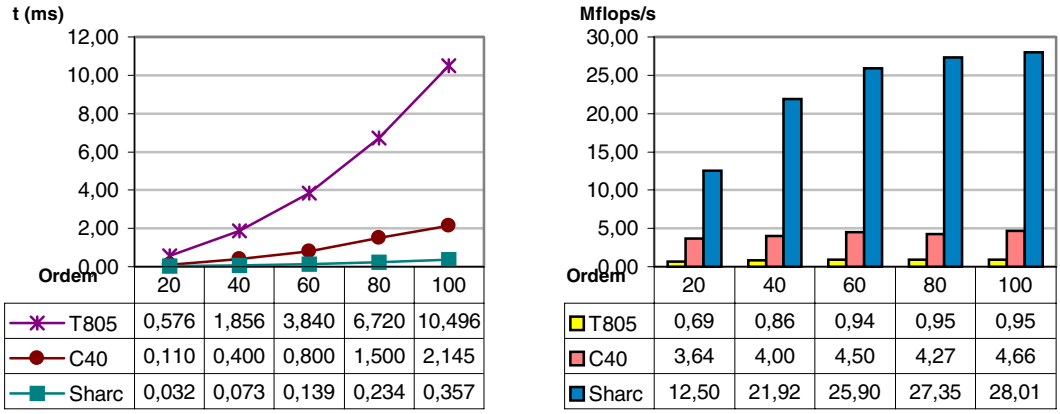
Fig. 5-6: Desempenho da operação adição ou subtracção em 1 nó homogéneo.



Ordem	20	40	60	80	100
T805	0,704	2,624	5,632	9,856	15,360
C40	0,140	0,540	1,300	2,100	3,300
Sharc	0,040	0,100	0,200	0,340	0,520

Ordem	20	40	60	80	100
T805	0,57	0,61	0,64	0,65	0,65
C40	2,86	2,96	2,77	3,05	3,03
Sharc	10,00	16,00	18,00	18,82	19,23

Fig. 5-7: Desempenho da operação adição ou subtracção em 2 nós homogéneos.



Ordem	20	40	60	80	100
T805	0,576	1,856	3,840	6,720	10,496
C40	0,110	0,400	0,800	1,500	2,145
Sharc	0,032	0,073	0,139	0,234	0,357

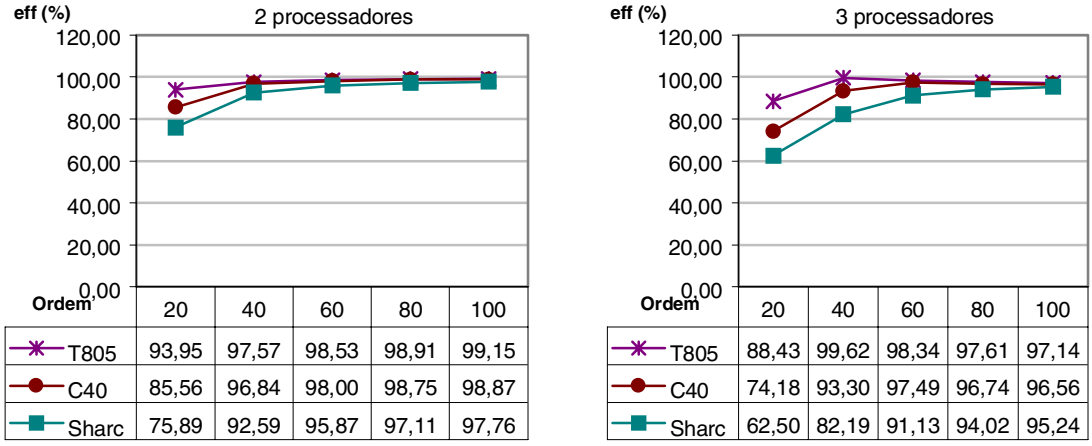
Ordem	20	40	60	80	100
T805	0,69	0,86	0,94	0,95	0,95
C40	3,64	4,00	4,50	4,27	4,66
Sharc	12,50	21,92	25,90	27,35	28,01

Fig. 5-8: Desempenho da operação adição ou subtracção em 3 nós homogéneos.

Como se pode consultar na figura acima, à esquerda, o número de operações de virgula flutuante para a rede T8[3], com três T805 ligados ponto a ponto ronda 950 000 por segundo,

bem abaixo dos 1,56 óptimos. No caso da rede C4[3] aproxima-se de 5 MFlops/s, e para a rede SH[3], com 3 processadores ADSP21060, ronda os 27 MFlops/s. Em ambos estes dois últimos casos também francamente abaixo dos valores óptimos já indicados.

Quanto aos tempos de execução, como seria de esperar, a arquitectura C40 é mais rápida que a T8 e mais lenta que ADSP21060. Mais precisamente, C40 é em média aproximadamente 4,8 vezes mais rápida que T8, ao passo que ADSP21060 é, também em média, aproximadamente 25,2 vezes mais rápida que T8. Estes valores referem-se a topologias com 3 processadores, no entanto neste caso as diferenças para 1 e 2 processadores não são significativas.



Ordem	20	40	60	80	100
T805	93,95	97,57	98,53	98,91	99,15
C40	85,56	96,84	98,00	98,75	98,87
Sharc	75,89	92,59	95,87	97,11	97,76

Ordem	20	40	60	80	100
T805	88,43	99,62	98,34	97,61	97,14
C40	74,18	93,30	97,49	96,74	96,56
Sharc	62,50	82,19	91,13	94,02	95,24

Fig. 5-9: Eficiência da operação adição

Em (Daniel e Ruano, 1999), foi investigado o desempenho de um só processador das famílias T8xx e C4x, referidos daqui em diante como T8[1] e C4[1], em termos de adição e multiplicação de matrizes quadradas. Nesse trabalho, a relação entre C4[1] e T8[1] medida é superior 8,3. No caso presente o tempo de processamento é menor. Isto deve-se a um método de descodificação de endereços dos elementos das matrizes, com um desempenho superior. Verifica-se no entanto que o principal beneficiado com a nova estratégia de endereçamento foi o T8, pois permitiu reduzir a diferença para o C40.

Quanto ao desempenho do algoritmo quando mapeado por mais que um processador é muito próximo dos 100%, pois as operações são efectuadas em cada processador, sobre os dados nele alocados, sem necessidade de transferência de informação entre eles. No entanto para matrizes com ordens inferiores, a eficiência não será tão boa. Isto deve-se à alocação de memória para a matriz resultado. Esta parte do código não pode ser paralelizada, o que

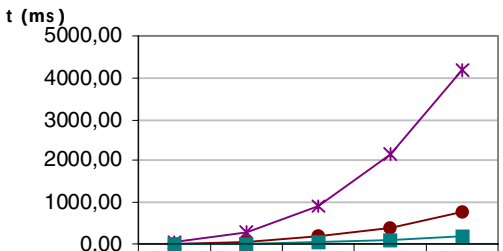
significa que independentemente do número de processadores, todos vão ter um tempo de atraso semelhante e indivisível na alocação da partição da matriz resultado local.
À medida que o tempo necessário para alocar memória deixa de ser significativo a eficiência aproxima-se de 100%. Pode-se observar no entanto que a família menos afectada por este atraso é a T8xx, seguida da C4x e finalmente dos ADSP21060.

Como nota final deve ser ainda indicado que os valores apresentados se referem à operação de adição. No entanto para todos os processadores utilizados a operação de subtracção tem o mesmo desempenho que a de adição, excepto no caso dos T8xx, onde a subtracção é cerca de 5% mais lenta que a adição.

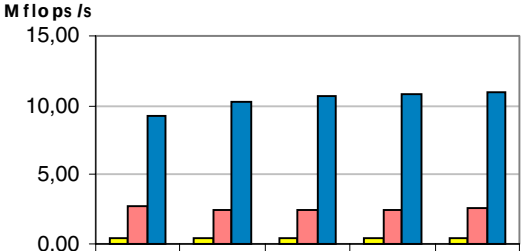
5.4.3 Multiplicação

5.4.3.1 C = A . B

O desempenho da operação de multiplicação matricial, é apresentada nas figuras seguintes. Para esta operação foram desenvolvidas duas versões. Os resultados apresentados a seguir referem-se à versão1, a que implica o alocamento completo do operando direito em cada nó. Posteriormente será apresentada a relação entre as duas versões.



Ordem	20	40	60	80	100
T805	36,3	275,3	915,4	2153,9	4188,5
C40	5,6	50,9	171,1	403,7	785,0
Sharc	1,7	12,4	40,4	94,2	182,1



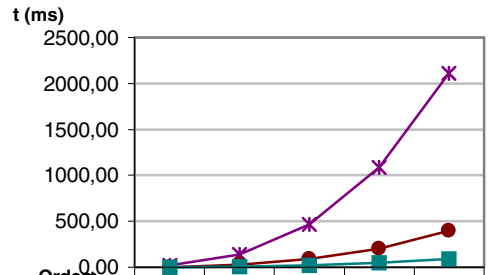
Ordem	20	40	60	80	100
T805	0,430	0,459	0,468	0,472	0,475
C40	2,771	2,481	2,504	2,521	2,535
Sharc	9,176	10,218	10,609	10,807	10,927

Fig. 5-10: Desempenho da operação multiplicação em 1 nó homogéneo.

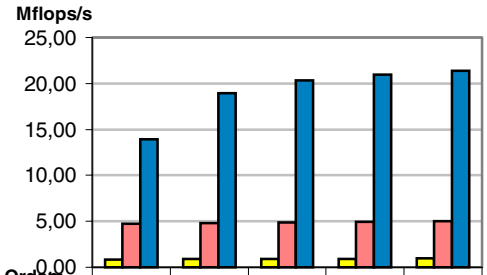
Nesta operação os tempos apresentados a seguir são também inferiores aos medidos em (Daniel e Ruano, 1999), mais uma vez devido à optimização do método de descodificação de endereços. Quanto à relação entre arquitecturas C4[1] e T8[1] é em média 5,6. Um pouco inferior à apresentada em (Daniel e Ruano, 1999), que é aproximadamente 10. mais uma vez,

a família mais beneficiada com a descodificação de endereços foi a T8xx. Já no caso da relação entre SH[1] e T8[1], verifica-se que esta última apresenta um desempenho médio 22.4 vezes inferior.

À medida que o número de processadores na rede aumenta a distância das outras famílias para os Transputers é reduzida. Para o caso de 2 processadores a relação T8[2] / C4 [2] é de 5.4 e a relação T8[2] / SH [2] é 21.1.

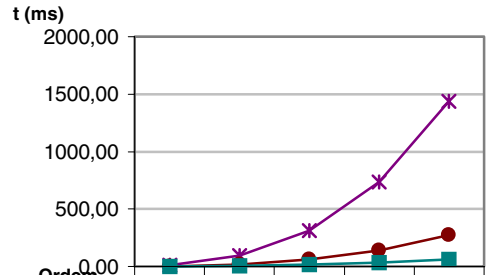


Ordem	20	40	60	80	100
T805	19,3	141,1	464,5	1088,8	2112,3
C40	3,3	26,3	87,3	204,4	397,0
Sharc	1,1	6,7	21,1	48,5	93,1

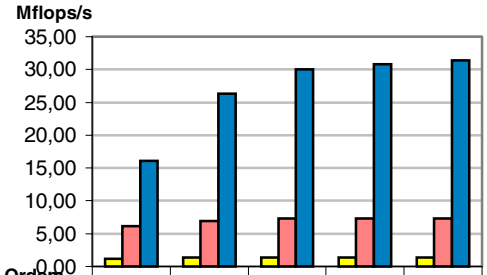


Ordem	20	40	60	80	100
T805	0,807	0,896	0,922	0,935	0,942
C40	4,742	4,806	4,907	4,978	5,013
Sharc	13,929	18,922	20,342	20,990	21,366

Fig. 5-11: Desempenho da operação multiplicação em 2 nós homogéneos.



Ordem	20	40	60	80	100
T805	13,7	92,1	310,3	735,6	1437,3
C40	2,5	18,2	58,5	139,1	271,0
Sharc	1,0	4,8	14,3	33,0	63,5



Ordem	20	40	60	80	100
T805	1,140	1,372	1,381	1,383	1,385
C40	6,166	6,945	7,323	7,316	7,343
Sharc	16,082	26,279	30,042	30,855	31,348

Fig. 5-12: Desempenho da operação multiplicação em 3 nós homogéneos.

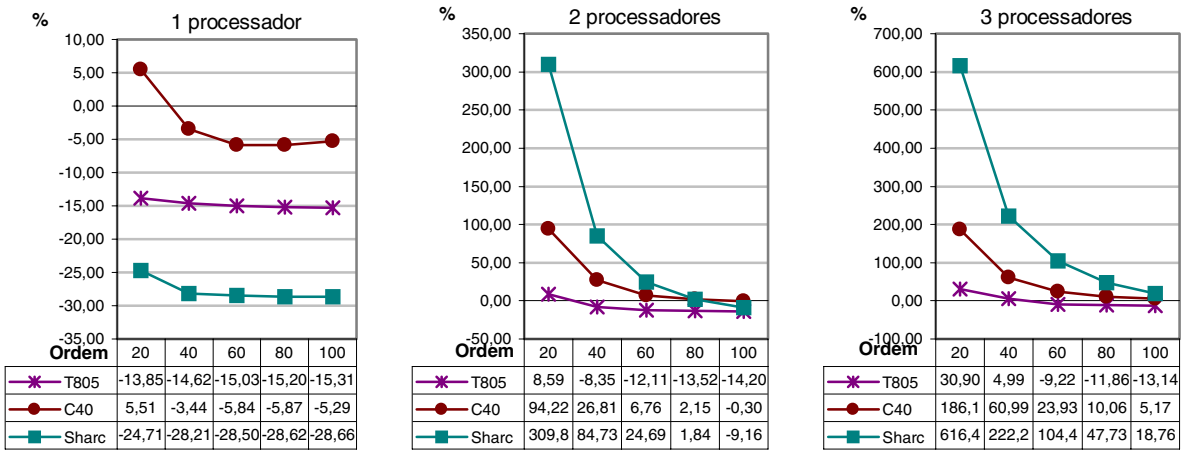
Já para 3 processadores as relações passam a ser:

T8[3] / C4 [3] = 5.3

T8[3] / SH [3] = 19.9

Na realidade esta aproximação também existe para a operação adição, no entanto a diferença entre as relações para 1, 2 e 3 processadores é muito menos significativa.

Seguidamente é apresentada a relação de desempenho entre a versão1 e a versão 2, como uma percentagem de perda de desempenho para a versão 2. Assim quanto maior for essa percentagem mais lenta é a versão 2 em relação à versão 1. No entanto percentagens negativas indicam que a versão 2 é mais rápida que a versão 1.



Ordem	20	40	60	80	100
T805	-13,85	-14,62	-15,03	-15,20	-15,31
C40	5,51	-3,44	-5,84	-5,87	-5,29
Sharc	-24,71	-28,21	-28,50	-28,62	-28,66

Ordem	20	40	60	80	100
T805	8,59	-8,35	-12,11	-13,52	-14,20
C40	94,22	26,81	6,76	2,15	-0,30
Sharc	309,8	84,73	24,69	1,84	-9,16

Ordem	20	40	60	80	100
T805	30,90	4,99	-9,22	-11,86	-13,14
C40	186,1	60,99	23,93	10,06	5,17
Sharc	616,4	222,2	104,4	47,73	18,76

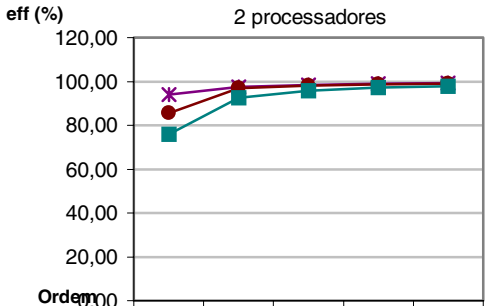
Fig. 5-13: Comparação entre as versões 1 e 2 da multiplicação matricial

No caso de um processador, como não há comunicação e a descodificação de endereços de elementos de uma matriz, necessária pela versão 1, é mais complexa que a descodificação de elementos de vectores colunas, usada na versão 2, as percentagens são negativas indicando que a versão 2 é mais rápida.

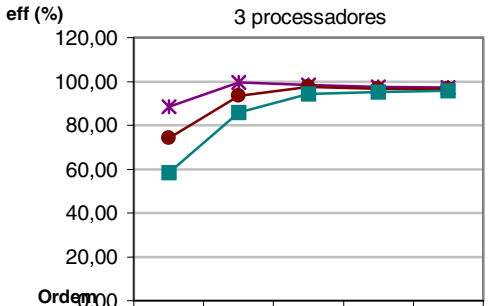
Para dois e três processadores verifica-se que para ordens menores a versão 1 é muito mais rápida que a versão 2, mas à medida que a ordem aumenta, o que implica que um vector coluna já tem um comprimento significativo e o ambiente de comunicações não introduz um atraso relevante, passa a dominar o atraso introduzido pela descodificação de endereços de uma matriz em relação a um vector, de modo que a versão 2 passa a ser mais rápida.

Por outro lado, à medida que o número de processadores na rede aumenta, e como um vector coluna é dividido pelo número de processadores na rede, ficando assim cada cópia local menor, aumenta a ordem para a qual a versão 2 passa a ser mais rápida que a versão 1.

Isto sugere que para 1 nó deve ser utilizada a versão 2, para dois ou três nós depende do processador. Para o caso do T8xx continua a ser mais rápida a versão 2. Mas para mais de 3 processadores a versão 1 tem tendência para ter um melhor desempenho.



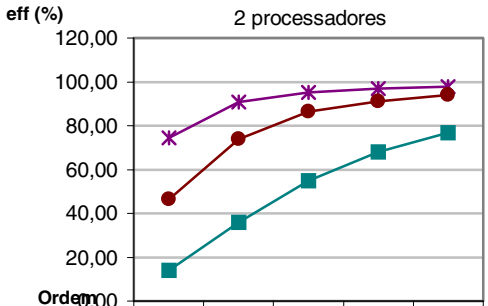
	20	40	60	80	100
T805	93,95	97,57	98,53	98,91	99,15
C40	85,56	96,84	98,00	98,75	98,87
Sharc	75,89	92,59	95,87	97,11	97,76



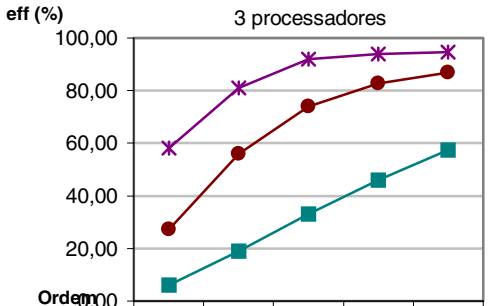
	20	40	60	80	100
T805	88,43	99,62	98,34	97,61	97,14
C40	74,18	93,30	97,49	96,74	96,56
Sharc	58,42	85,72	94,39	95,17	95,63

Fig. 5-14: Eficiência da operação multiplicação versão 1

Quanto à eficiência, observável nas figuras imediatamente acima e abaixo, verifica-se que a versão 2, permita poupar memória mas de um modo geral a eficiência é baixa. Para a versão 1 a eficiência já é muito superior. Têm-se no entanto que tomar em conta que o tempo de execução num só processador, e que serve de referência para esta medida, é um pouco inflacionado pela descodificação de endereços, como já foi apontado. Assim, para se ter uma ideia mais correcta do desempenho da versão 1 pode-se usar como referência o melhor algoritmo sequencial conhecido, isto é para medir tempos de execução num só processador utilizar a versão 2, e utilizar a versão 1 para qualquer outra topologia multi-processador.



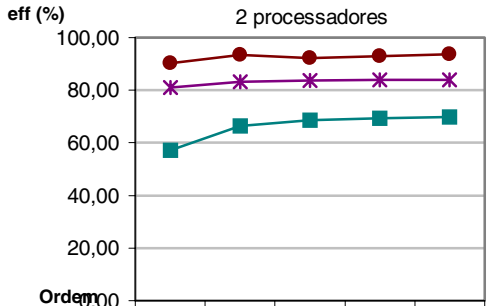
	20	40	60	80	100
T805	74,54	90,89	95,26	96,99	97,87
C40	46,48	73,75	86,43	91,00	93,92
Sharc	13,94	35,98	54,97	68,07	76,78



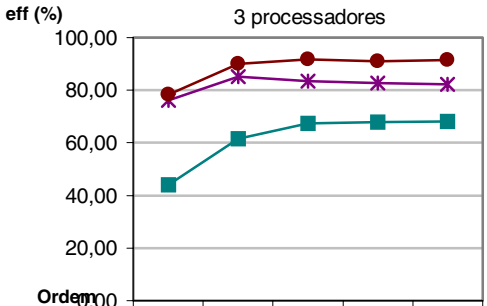
	20	40	60	80	100
T805	58,20	81,01	92,05	93,91	94,71
C40	27,35	55,96	74,07	82,73	86,96
Sharc	6,14	19,10	33,01	45,98	57,44

Fig. 5-15: Eficiência da operação multiplicação versão 2

Para esse caso a eficiência é a seguinte:



	20	40	60	80	100
T805	80,94	83,30	83,72	83,87	83,97
C40	90,27	93,52	92,27	92,95	93,64
Sharc	57,14	66,47	68,54	69,32	69,74



	20	40	60	80	100
T805	76,19	85,06	83,56	82,77	82,27
C40	78,26	90,09	91,79	91,06	91,45
Sharc	43,99	61,54	67,48	67,93	68,22

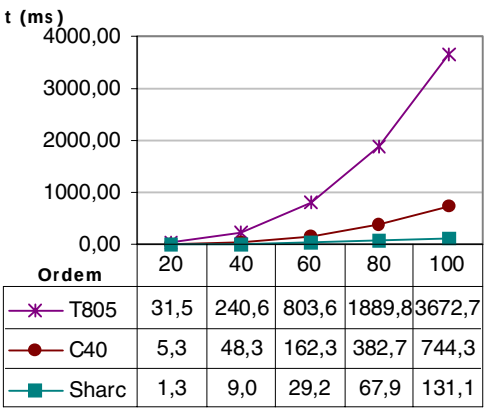
Fig. 5-16: Eficiência da operação multiplicação. Versão 2 para 1 processador e versão 1 para topologias multi-processador

Como se pode ver a melhoria não é tão dramática como a apresentada na Fig. 5-14, mas já é substancial, principalmente para matrizes de ordem inferior, e para as famílias C4x e ADSP2106x.

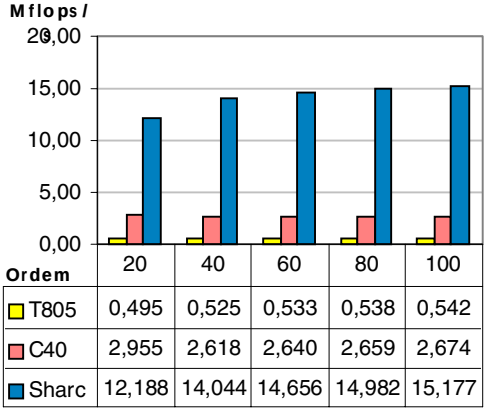
5.4.3.2 $C = A \cdot B^T$

Já no caso da função de multiplicação com transposição prévia do operando direito, o ponto onde a versão 1 deixa de ser mais rápida que a 2 encontra-se para ordens muito superiores. Esta operação foi desenhada para evitar a transposição de um operando antes de multiplicá-lo por outro, evitando assim uma operação de transposição. Nas figuras seguinte estão apresentados os tempos para a versão 1. Como se pode observar é mais rápida que a própria operação de multiplicação versão 1. Como já foi indicado isto deve-se ao acesso aos elementos do operando direito ser simplificado, se se assumir que este deve ser acedido como estando transposto.

De notar que nesta operação se retira melhor partido do processador do que na medida de desempenho escalar apresentada no ponto 2.4, para as famílias T8xx e ADSP2106x. Assim, para três processadores a partir de matrizes com ordem 60 o desempenho é superior a 0,52 x 3 = 1,56 MFlops/s e 13.33 x 3 = 39.99 respectivamente. Isto não revela nenhuma inconsistência, pois o capacidade teórica anunciada destas famílias é superior. Apenas indica que esta operação tira melhor partido do processador que a operação utilizada para a medida de desempenho escalar.

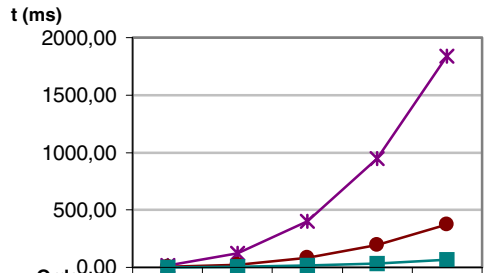


Ordem	20	40	60	80	100
T805	31,5	240,6	803,6	1889,8	3672,7
C40	5,3	48,3	162,3	382,7	744,3
Sharc	1,3	9,0	29,2	67,9	131,1

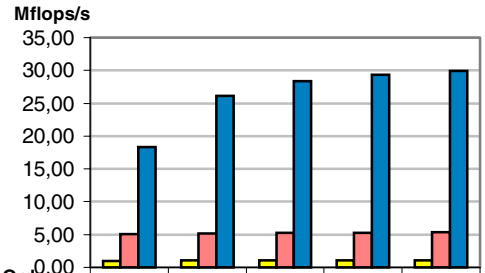


Ordem	20	40	60	80	100
T805	0,495	0,525	0,533	0,538	0,542
C40	2,955	2,618	2,640	2,659	2,674
Sharc	12,188	14,044	14,656	14,982	15,177

Fig. 5-17: Desempenho da operação multiplicação com transposição operando direito em 1 nó.

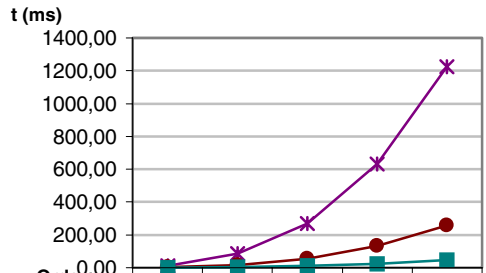


Ordem	20	40	60	80	100
T805	16,6	120,6	402,7	945,0	1836,5
C40	3,1	24,5	81,9	192,8	374,2
Sharc	0,9	4,8	15,1	34,7	66,6

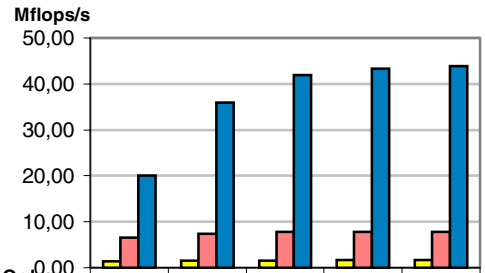


Ordem	20	40	60	80	100
T805	0,938	1,048	1,064	1,077	1,084
C40	5,115	5,157	5,231	5,278	5,318
Sharc	18,353	26,170	28,352	29,343	29,889

Fig. 5-18: Desempenho da operação multiplicação com transposição operando direito em 2 nós homogéneos.



Ordem	20	40	60	80	100
T805	11,7	84,1	269,0	630,0	1224,3
C40	2,4	17,1	55,2	131,0	256,0
Sharc	0,8	3,5	10,2	23,5	45,4



Ordem	20	40	60	80	100
T805	1,332	1,503	1,593	1,615	1,625
C40	6,555	7,392	7,761	7,768	7,773
Sharc	20,000	35,909	41,959	43,247	43,852

Fig. 5-19: Desempenho da operação multiplicação com transposição operando direito em 3 nós homogéneos.

No caso desta operação, a versão 1 é mais rápida que a 2, ou praticamente idêntica para ordens de matrizes até 100, como é o caso de num só T8.

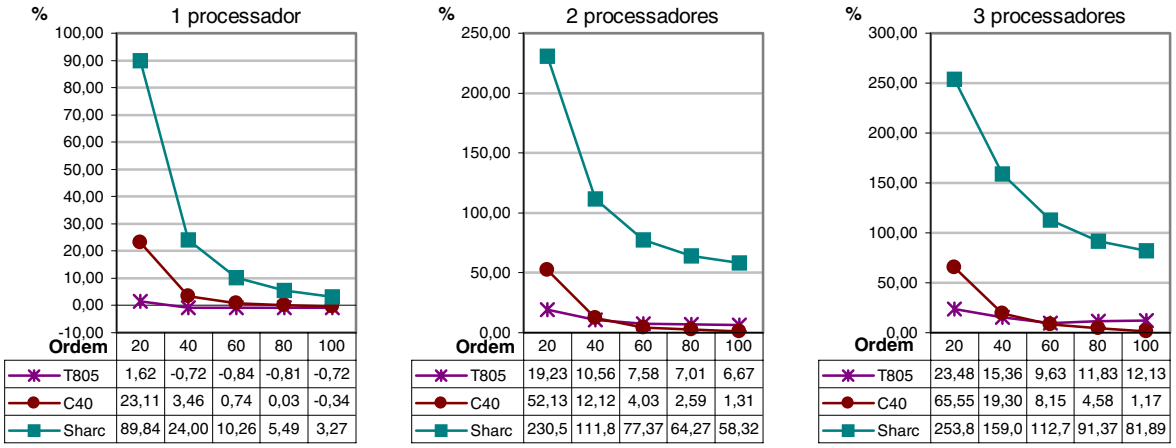


Fig. 5-20: Comparação entre as versões 1 e 2 da multiplicação matricial com transposição do operando direito

Assim para qualquer rede, desde que a ordem dos operandos fique dentro da gama acima apresentada e não haja memória disponível, deve sempre ser utilizada a versão 1.

Quanto à eficiência, a versão 1 está próxima dos 100% para matrizes com ordens grandes. No caso da versão 2, embora para um processador os tempos sejam muito idênticos aos da versão 1, perde muito em eficiência para esta. Isto verifica-se mais drásticamente para as redes ADSP21060. Esta maior perda de tempo em comunicações justifica também que seja mais lenta.

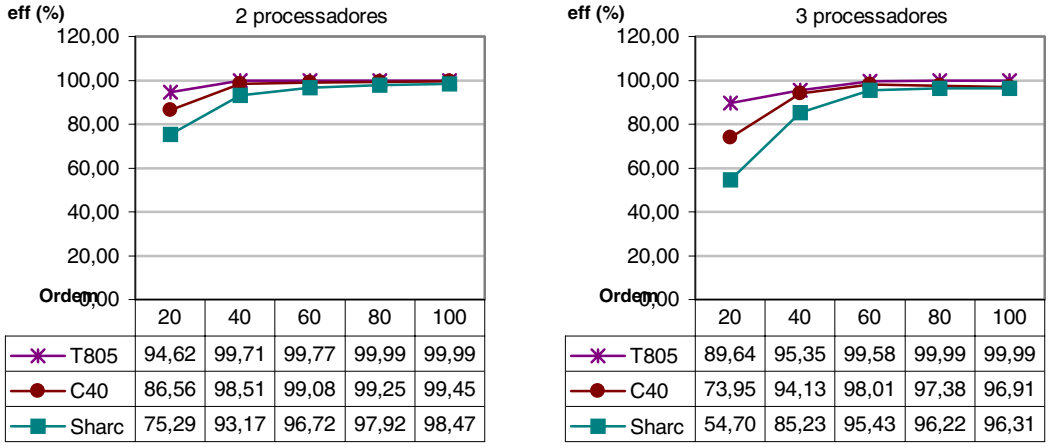
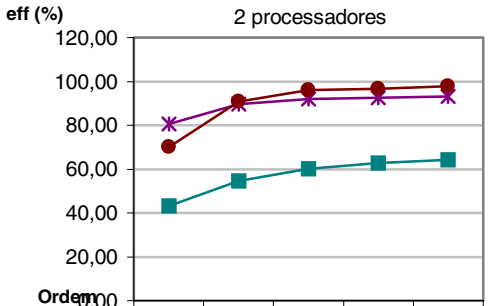
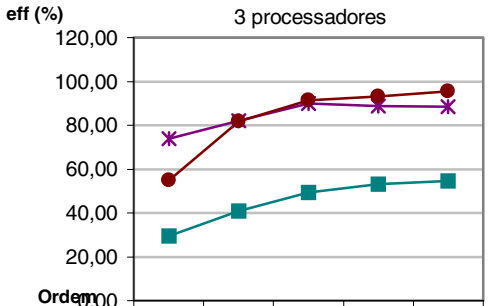


Fig. 5-21: Eficiência da operação multiplicação com transposição operando direito versão 1



	20	40	60	80	100
T805	80,65	89,53	91,96	92,69	93,07
C40	70,04	90,90	95,95	96,76	97,84
Sharc	43,24	54,55	60,13	62,88	64,23

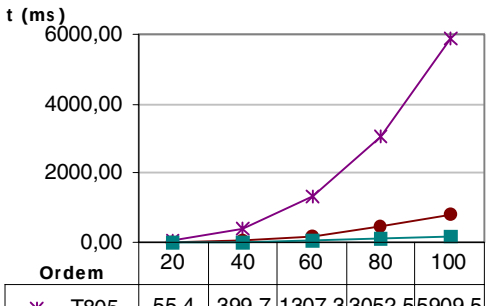


	20	40	60	80	100
T805	73,77	82,06	90,06	88,69	88,53
C40	54,99	81,63	91,29	93,14	95,47
Sharc	29,35	40,79	49,46	53,04	54,68

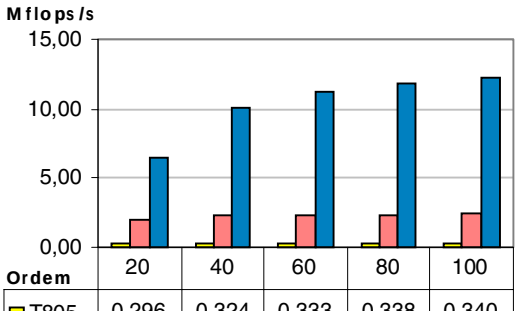
Fig. 5-22: Eficiência da operação multiplicação com transposição operando direito versão 2

5.4.4 Inversão

No caso da avaliação do desempenho da inversão matricial, pelo método de Gauss-Jordan, são utilizadas matrizes cujo elemento *pivot* nunca será nulo, de modo a evitar perda de tempo em busca de um *pivot*. Assim se durante a operação forem encontrados *pivots* nulos, o tempo será sempre superior ao indicado na figura.



	20	40	60	80	100
T805	55,4	399,7	1307,3	3052,5	5909,5
C40	8,4	55,9	185,2	433,6	827,1
Sharc	2,5	12,9	38,7	87,0	164,9

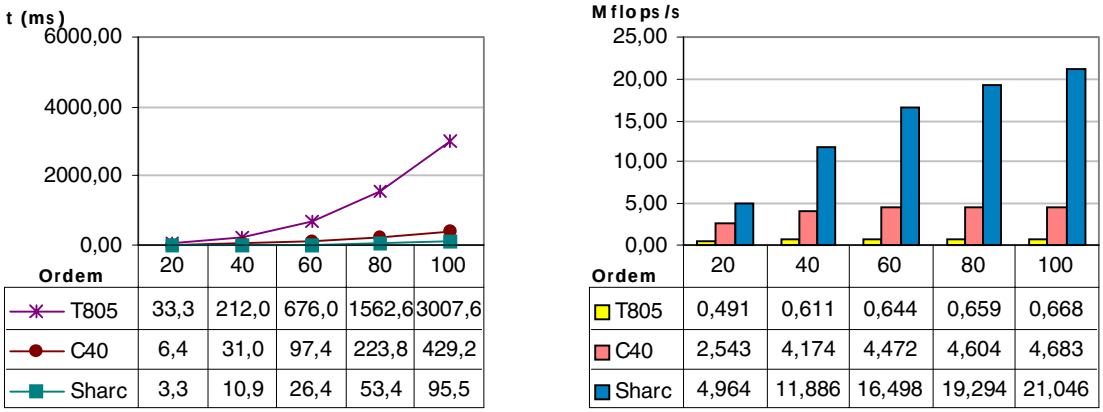


	20	40	60	80	100
T805	0,296	0,324	0,333	0,338	0,340
C40	1,950	2,316	2,352	2,376	2,430
Sharc	6,552	10,043	11,254	11,843	12,189

Fig. 5-23: Desempenho da operação inversão em 1 nó

Esta operação é executada sobre matrizes quadradas de ordem n, logo tem uma complexidade algorítmica na ordem de $O(n^3)$ (Aho et al, 1983), tal como a multiplicação matricial para a mesma matriz. Também como na multiplicação, uma matriz inteira é também comunicada para todos os processadores linha a linha, no caso da inversão a linha *pivot*. Mas a inversão opera sobre 2 matrizes de ordem n, a matriz identidade onde ficará a inversa, e a matriz a

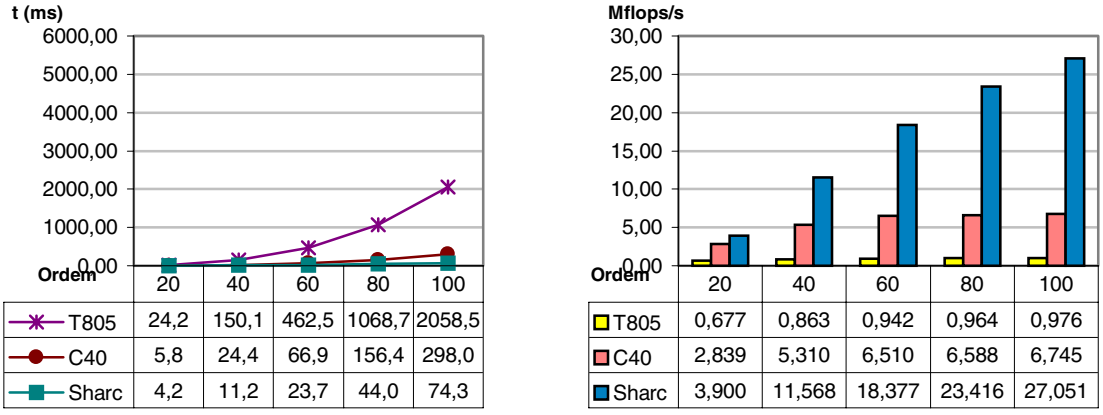
inverter, embora em cada iteração, existindo tantas como a ordem da matriz a inverter, opere sobre menos uma coluna dessa matriz. Daí que o tempo de execução deve ser superior ao de mmult e inferior ao dobro deste, pois a complexidade algorítmica será superior a O(n^3) e inferior a O(2 n^3).



Ordem	20	40	60	80	100
T805	33,3	212,0	676,0	1562,6	3007,6
C40	6,4	31,0	97,4	223,8	429,2
Sharc	3,3	10,9	26,4	53,4	95,5

Ordem	20	40	60	80	100
T805	0,491	0,611	0,644	0,659	0,668
C40	2,543	4,174	4,472	4,604	4,683
Sharc	4,964	11,886	16,498	19,294	21,046

Fig. 5-24: Desempenho da operação inversão em 2 nós homogéneos



Ordem	20	40	60	80	100
T805	24,2	150,1	462,5	1068,7	2058,5
C40	5,8	24,4	66,9	156,4	298,0
Sharc	4,2	11,2	23,7	44,0	74,3

Ordem	20	40	60	80	100
T805	0,677	0,863	0,942	0,964	0,976
C40	2,839	5,310	6,510	6,588	6,745
Sharc	3,900	11,568	18,377	23,416	27,051

Fig. 5-25: Desempenho da operação inversão em 3 nós homogéneos

Nesta operação verifica-se que T8[3] é em média 6,2 vezes mais lento que C4[3] e 16,9 vezes mais lento que SH[3]. No entanto como sucede na multiplicação à medida que a ordem das matrizes aumenta mais significativa é a diferença entre Transputers e Sharcs, embora entre Transputers e C40s permaneça quase constante. Por outro lado à medida que o número de nós aumenta, diminuí a relação para o Transputer. Para o caso de 2 processadores têm-se:

T8[2] / C4 [2] = 6.6

T8[2] / SH [2] = 21.5

e para um:

T8[1] / C4 [1] = 7

T8[1] / SH [1] = 29

Isto deve-se ao facto da implementação do ambiente de comunicações em T8 ser próxima do ideal, pois o Transputer é capaz de comunicar e processar concorrentemente e implicitamente, o que não sucede nas outras arquitecturas suportadas, pelas razões já abordadas no capítulo anterior. Por essa razões a arquitectura menos eficiente é a ADSP21060, como mostra a figura seguinte:

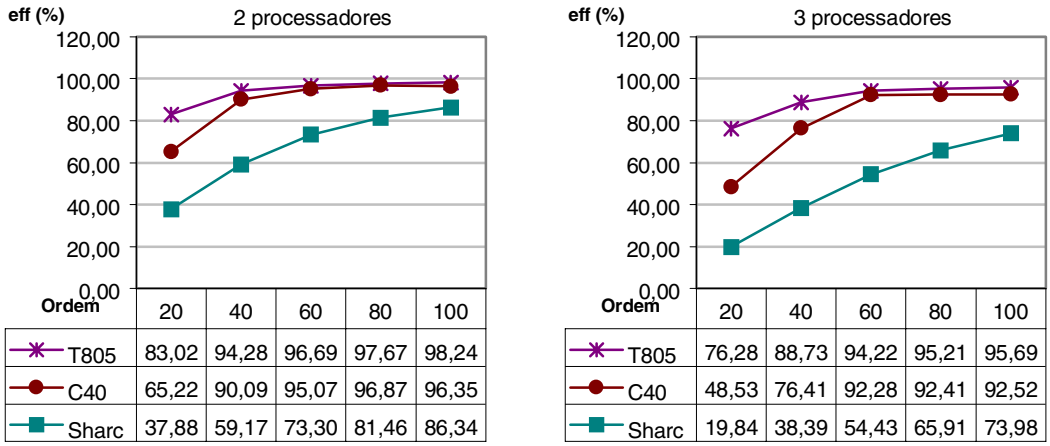
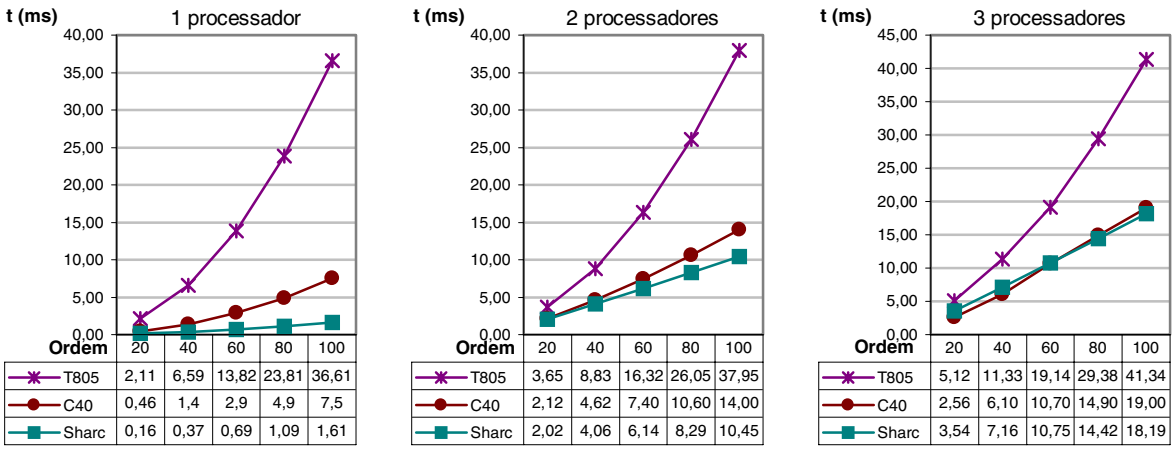


Fig. 5-26: Eficiência da operação inversão segundo o método de Gauss-Jordan

5.4.5 Transposição

A última medida de desempenho apresentada consiste na transposição de matrizes. Este é um algoritmo essencialmente baseado em transferências de dados entre processadores. Por isso mesmo não faz sentido apresentar uma medida de *flops*.

No entanto é necessário algum processamento para receber e posicionar os blocos de dados recebidos, e transpor os elemento locais. Deste modo a diferença entre T8[3] e as outras arquitecturas não é tão significativa como anteriormente, visto que o T805 comunica e processa concorrentemente sem recorrer a mecanismos de DMA. Assim o desempenho desta operação irá depender do paralelismo interno do próprio processador, quando não explicitamente programado.



Ordem	20	40	60	80	100
T805	2,11	6,59	13,82	23,81	36,61
C40	0,46	1,4	2,9	4,9	7,5
Sharc	0,16	0,37	0,69	1,09	1,61

Ordem	20	40	60	80	100
T805	3,65	8,83	16,32	26,05	37,95
C40	2,12	4,62	7,40	10,60	14,00
Sharc	2,02	4,06	6,14	8,29	10,45

Ordem	20	40	60	80	100
T805	5,12	11,33	19,14	29,38	41,34
C40	2,56	6,10	10,70	14,90	19,00
Sharc	3,54	7,16	10,75	14,42	18,19

Fig. 5-27: Desempenho da operação transposição em 1 a 3 nós homogéneos (v1)

A relação entre as várias arquitecturas está patente na seguinte tabela, que pode também ser vista como um indicador do desempenho dos processadores quanto a transferências de dados internas e externas em simultâneo:

	C4 / T8	**SH / T8**
p=1	4,760	19,122
p=2	2,201	2,683
p=3	1,959	1,824

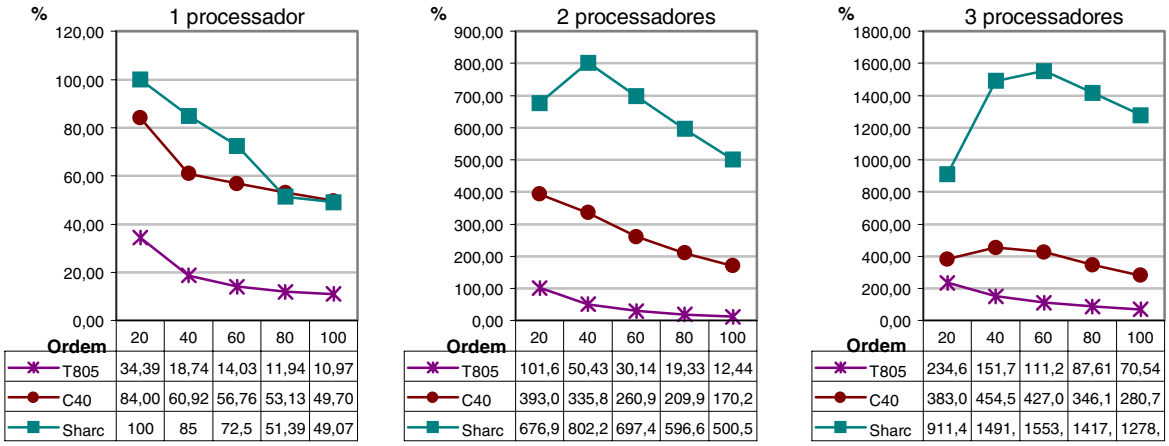
tabela 5-20: Desempenho comparativo da operação transposição (v1)

Cada valor constituí a média para matrizes de ordem 20 a 100.

No caso de um só nó, p=1, a diferença refere-se apenas a transferências internas e de algum modo é próximo das relações indicadas nas outras operações. No entanto para p=2 e p=3, quando são efectuadas transferências externas, ou entre nós, verifica-se mais uma vez que a implementação do ambiente de comunicações em Transputers é mais eficiente, pois a relação é muito inferior à esperada se se considerar a largura de banda bidireccional anunciada por porto, isto é 2,35 MBytes para T8, 20MBytes para C4x e 40MbBytes para ADSP2106x, como já tinha sido indicado no ponto 2.3.

A comparação da versão 2 com a versão 1 pode-se consultar na Fig. 5-28, onde os valores percentuais indicam quanto é que a versão 1 é mais lenta que a 2. Como se pode observar a partir de uma dada ordem a versão dois aproxima-se da versão 1. Como já foi indicado, num dado ponto em que os segmentos que são comunicados na versão 1 forem suficientemente

grandes para que o empacotamento de mensagens pelo ambiente não seja significativo, esta versão deve ser mais rápida que a 2, pois o número de elementos a transferir entre nós é menor. Os gráficos acima mostram esta tendência.



Ordem	20	40	60	80	100
T805	34,39	18,74	14,03	11,94	10,97
C40	84,00	60,92	56,76	53,13	49,70
Sharc	100	85	72,5	51,39	49,07

Ordem	20	40	60	80	100
T805	101,6	50,43	30,14	19,33	12,44
C40	393,0	335,8	260,9	209,9	170,2
Sharc	676,9	802,2	697,4	596,6	500,5

Ordem	20	40	60	80	100
T805	234,6	151,7	111,2	87,61	70,54
C40	383,0	454,5	427,0	346,1	280,7
Sharc	911,4	1491,	1553,	1417,	1278,

Fig. 5-28: Comparação entre as versões 1 e 2 da transposição

Em termos de comparação entre processadores diferentes, têm-se para a versão 2, onde cada valor constituí a média para matrizes de ordem 20 a 100:

	T8 / C4	**SH / T8**
p=1	6,488	27,553
p=2	5,753	14,807
p=3	4,197	12,632

tabela 5-21: Desempenho comparativo da operação transposição (v2)

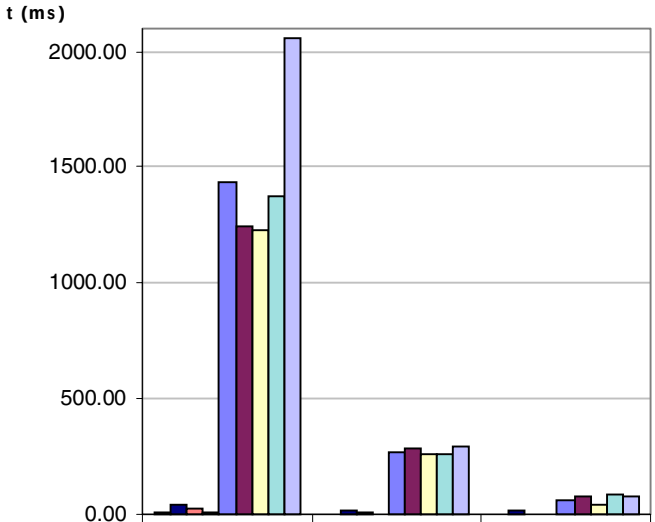
Repare-se que a relação para o T8 aumenta em todos os casos. Isto mostra mais uma vez que os Transputers são os melhores processadores em termos de comunicação.

5.4.6 Comparação das várias operações

Na figura seguinte apresenta-se a comparação do desempenho das operações acima ensaiadas, para matrizes de ordem 100, para topologias com 3 nós conectados ponto a ponto.

Sejam α e β, duas funções dependentes de um dado algoritmo, poder-se-ia dividir as operações em duas classes quanto ao tempo de processamento. Na classe superior ter-se-ía as operações de multiplicação (mmult...) e inversão (gjinv), cuja complexidade algorítmica, como já foi visto é na ordem de $O(\alpha\, n^3)$. Na classe inferior as operações de adição (madd),

com uma complexidade algorítmica de $O(\beta\ n^2)$ e as operações que apenas implicam transferências de dados e não cálculo de vírgula flutuante, mtrans e collect2, a qual permite formar uma matriz completa em todos os nós a partir das partes distribuídas.



	T805	C40	ADSP21060
collect2	12.83	3.21	0.91
mtrans (v1)	41.34	19.00	18.19
mtrans (v2)	24.24	4.99	1.32
madd	10.496	2.145	0.365
mmult (v1)	1437.25	271	63.48
mmult (v2)	1248.44	285	75.39
mmult_t (v1)	1224.3	256	45.38
mmult_t (v2)	1372.86	259	82.54
gjinv	2058.50	298.00	74.30

Fig. 5-29: Comparação das grandezas das operações para matrizes de ordem 100

Assim as diferenças de desempenho entre arquitecturas de famílias diferentes, tem a ver com o algoritmo em si e também com as características de cada processador, não podendo se atribuir um factor constante para todos as operações matriciais aqui apresentadas. Pode-se no entanto inferir que o desempenho de um dado algoritmo paralelo, segue o desempenho das operações mais pesadas, o que significa normalmente a multiplicação e a inversão.

5.5 Resumo

Neste capítulo são apresentadas as bibliotecas de cálculo que operam sobre operandos distribuídos, quer estes sejam vectores, matrizes ou outro tipo de dados. Estes operandos são distribuídos de forma a equilibrar a carga computacional em todos os nós da rede. Por outro lado as funções são optimizadas não só em termos de velocidade de execução e minimização

das transferências de dados entre nós, mas também de modo que dados transferidos em cada passo requeiram um espaço de armazenamento temporário o menor possível, como é o caso das versões 2 das operações de multiplicação de matrizes

As funções são separadas em sequenciais e paralelas, segundo o diagrama da Fig. 3-4 e Fig. 3-5. As primeiras, são as que não necessitam de transferir dados entre nós, tais como a adição matricial ou operações entre escalares e matrizes. Estas são as mais eficientes, pois se a carga computacional estiver equilibrada em toda a rede o tempo de execução será reduzido proporcionalmente ao número de processadores na rede. Já nas funções paralelas esta relação não é linear, mas sim dependente do tempo requerido pelas transferências de dados entre nós, durante a execução do algoritmo

Estas funções foram também classificadas pelo número e tipo de operandos de modo que estas bibliotecas possam ser extendidas pelo utilizador, segundo regras que o tradutor apresentado no próximo capítulo reconheça. A sua principal utilidade no entanto é apenas possibilitar a redefinição dos operadores matriciais, pelo utilizador.

Para que se possa ter uma ideia aproximada do tempo de execução de um algoritmo, o desempenho das operações básicas de álgebra linear foi apresentado.

Neste capítulo também foi apresentada uma extensão ao ambiente de comunicações, que permite facilitar a comunicação entre as instâncias da aplicação e os recursos do anfitrião. Baseado num servidor que corre no processador raiz ao qual as instâncias pedem serviços de entrada e saída, facilita também o desenvolvimento de uma aplicação, pois não é necessário desenvolver código para fazer o *interface* entre a rede e o anfitrião como no caso do modelo de programação paralela apresentado no capítulo anterior. Assim é também simplificado o desenvolvimento do tradutor, pois apenas tem de gerar código para as instâncias da aplicação.

Concluí-se assim a introdução dos principais elementos que estão na base do SPAM, o ambiente de comunicações com as extensões para entrada-saída com o anfitrião e as bibliotecas de cálculo distribuído. De acordo com estes elementos, o tradutor introduzido no próximo capítulo gera automaticamente código C paralelo a partir de código fonte SEQ 1.0.

6 Tradutor

A operação do tradutor consiste numa transformação do código sequencial em código C paralelo. Neste capítulo a operação do tradutor é descrita e enquadrada dentro do SPAM 1.0. É apresentada a definição da linguagem SEQ 1.0, bem como a correspondência entre as instruções desta linguagem e funções das bibliotecas introduzidas nos capítulos anteriores. A operação do tradutor é explicada e enquadrada dentro do SPAM 1.0. A implementação do tradutor é também pormenorizadamente descrita.

6.1 Linguagem SEQ 1.0

A linguagem SEQ 1.0 é apresentada a seguir em notação EBNF (Sethi, 1996), ou BNF extendida. A notação *Backus Normal Form* foi introduzida por (Backus, 1960) como uma gramática livre de contexto. Após os reconhecidos contributos de (Naur, 1963) passou a ser um acrónimo de *Backus Naur Form*. Depois de ter sido utilizada para descrever a linguagem algorítmica Algol, tem vindo a ser utilizada na descrição de muitas outras linguagens de programação. A versão da linguagem SEQ apresentada a seguir, é uma evolução da linguagem sequencial introduzida em (Daniel e Ruano, 1999).

6.1.1 Formalização em EBNF

programa ::= {declaração | declaraçãofunção}
program
{declaração}
{comando}

declaraçãofunção ::= [tipo] identificador parâmetros '{'
{declaração}
{comando} '}'

declaração ::= tipo identificador {‘,’ identificador} ‘;’

tipo ::= **integer** | **real** | **string** | **matrix**

parâmetros ::= ‘(‘ [lista_parâmetros] ‘)’

lista_parâmetros ::= tipo identificador | tipo identificador ‘,’ lista_parâmetros

Instrução ::= ‘{‘ {comando} ‘}’

comando ::= nulo ‘;’ |
chamadafunção ‘;’ |
atribuição ‘;’ |
if expEscalar **then** Instrução [**else** Instrução] |
for identificador = expEscalar (**to** | **downto**) expEscalar [**step** expEscalar] Instrução |
while expEscalar Instrução |
exit [Expressão] ‘;’
dim identificador [expEscalar [, expEscalar]] ‘;’
clear identificador ‘;’

atribuição ::= identificador [expEscalar [, expEscalar]] ‘=’ expressão |
variável | lista | “” cadeia “”

chamadafunção ::= identificador argumentos

argumentos ::= ‘(‘ [lista_argumentos] ‘)’

lista_argumentos ::= (expEscalar | identificador | chamadafunção) |
(expEscalar | identificador | chamadafunção ‘,’ lista_argumentos)

expressão ::= ‘(‘expressão’)’ | operando op_binário expressão |
op_prefixo expressão | expressão op_sufixo |
chamadafunção | variável | escalar

expEscalar ::= ‘(‘expEscalar’)’ | operandoEscalar op_binário expEscalar |
op_prefixo expEscalar | chamadafunção | identificador | escalar

operando ::= escalar | Variável

operandoEscalar ::= escalar | Identificador

variável ::= identificador [área]
constante ::= escalar | ““cadeia”” | lista
área ::= ‘[‘ faixa [, faixa] ‘]’
faixa ::= expEscalar ‘:’ expEscalar | expEscalar | ‘:’
lista ::= ‘#’ escalar { ‘,’ escalar}
operadores ::= op_prefixo | op_sufixo | op_binário
op_binário ::= op_relacional | op_aritmético | op_lógico
op_relacional ::= ‘= =’ | ‘!=’ | ‘<’ | ‘<=’ | ‘>=’ | ‘>’
op_lógico ::= ‘&&’ | ‘||’
op_aritmético ::= ‘ +’ | ‘-‘ | ‘*’ | ‘/’ | ‘%’
op_prefixo ::= ‘+’ | ‘-‘ | ‘!’
op_sufixo ::= ‘’’

escalar ::= real | inteiro
matricial ::= matriz | cadeia
matriz ::= {vector}
vector ::= {real}
real ::= [inteiro] . natural
inteiro ::= [(- | +)] natural
cadeia ::= {ASCII}
identificador ::= literal | “_” { alfanumérico | “_”}
alfanumérico ::= {literal | digito}
literal ::= ‘a’ .. ‘z’ | ‘A’ .. ‘Z’
digito ::= ‘0’ .. ‘9’
natural ::= {digito}
ASCII ::= 0 .. 255
nulo ::= ‘’

6.1.2 Tipos de dados e operadores

Os tipos de dados suportados dividem-se primeiramente em escalares, os quais têm uma instância em cada nó e em matriciais, vectoriais incluídos, que são distribuídos por todos os nós da rede.

escalares: inteiros (**integer**)
reais (**real**)

matriciais: matriz (**matrix**)
cadeia de caracteres (**string**)

Não existe um tipo próprio para guardar caracteres ou valores lógicos, de modo que estes são guardados num inteiro. No caso dos caracteres é guardado o código ASCII. Já no caso dos valores lógicos **falso** e **verdade**, estes são representados respectivamente por zero e por um qualquer outro valor diferente de zero. As cadeias de caracteres, tipo **string**, consistem em vectores distribuídos onde é guardado o código ASCII de cada caracter. Se bem que se possam processar caracteres como inteiros, visto que estes não são distribuídos, o tipo **string** é o mais indicado para tirar partido do paralelismo do sistema.

	Operadores	Tipo suportados	Classe	Comentários
7	‘	**integer**, **real** e **matrix**	unário sufixo	
6	!	**integer**, **real**	unário prefixo	
5	+ -	**integer**, **real** e **matrix**	unário prefixo	
4	* / + - == !=	**integer**, **real** e **matrix**	binário	impossível: escalar ‘/’ **matrix** ***matrix*** *(‘= =’ | ‘!=’) escalar*
3	< <= > >= && ||	**integer**, **real**	binário	
2	%	**integer**	binário	resto da divisão inteira
1	[ ]	**matrix, string**	unário sufixo	**matrix** ‘[‘faixa’,’ faixa’]’ **string** ‘[‘faixa’]’ onde faixa > 0
0	( )	*todos*		agrupamento

tabela 6-1: Operadores e tipos suportados

Na tabela anterior estão indicados os tipos suportados para cada operador. Nesta tabela também está classificado o operador quanto ao número de operandos. É conveniente indicar que não está prevista a divisão de um escalar por uma matriz nem a comparação de uma

matriz com um escalar. No caso do operador '[]', faixa deve ser uma expressão que retorne um inteiro positivo.

A conversão entre os dois tipos escalares é tratada pelo compilador de C e é coerciva, isto é, se ambos os tipos estiverem envolvidos numa operação, um inteiro é sempre transformado num número de vírgula flutuante, antes da operação se efectuar. Isto implica que o resultado será também um número de vírgula flutuante. Por outro lado, não está definida nenhuma forma para converter escalares para matrizes. Se se pretender converter um escalar para uma matriz com uma linha e uma coluna, este escalar deve ser atribuído ao elemento da posição 1,1 da matriz.

O tipo **void** não foi referido, pois é assumido que a ausência de um indicador de tipo implica que este é **void.** É o caso do tipo de funções que não retornam valores ou funções sem parâmetros.

Os parênteses curvos são usados para agrupamento de expressões e por isso são considerados operadores com maior prioridade. Esta é indicada na tabela seguinte:

Prioridade	*Operadores*	
maior	( ) [ ] ‘	150
	! + -	140
	* / %	130
⇓	+ -	120
	< <= > >=	110
	= = !=	100
	&&	90
menor	‖	80

tabela 6-2: Prioridade de operadores

6.1.3 Tradução para C paralelo

Nesta secção é apresentada a correspondência entre o código sequencial SEQ 1.0 e C paralelo. A secção é sub-dividida em três partes. Na primeira são definidas as traduções para as construções já introduzidas no ponto 6.1.1, sendo no entanto a tradução de expressões e atribuições tratada separadamente. É também reservada uma parte para directivas do tradutor, as quais não são parte da linguagem SEQ 1.0.

6.1.3.1 Construções

Antes de iniciar a apresentação da tradução das construções, deve-se referir que os delimitadores de blocos de código '{' e '}' são idênticos em ambas as linguagens. O mesmo sucede para os operadores com excepção do operador de transposição ''' e da igualdade ou desigualdade entre matrizes, que é efectuada recorrendo à função ISEQMTX. Assim, com excepção destes casos não existe necessidade de traduzir estes lexemas.

A tradução das construções é introduzida numa perspectiva *top-down*. Assim um programa é composto por declarações globais, antes da palavra reservada **program** e depois pelo código principal. Em C paralelo a organização de um programa é basicamente a mesma, excepto que o código principal é considerado como o corpo de uma função de nome main. Este lexema traduz-se:

program ⇒ **void** main (**int** argc**, char** *argv[], **char** *envp[],
chan *in_ports[], **int** ins, **chan** *out_ports[], **int** outs)

As declarações de variáveis traduzem-se da seguinte forma:

integer | **real** identificador ';' ⇒ **INTEGER** | **REAL** identificador ';'
string | **matrix** identificador ';' ⇒ **matrix** identificador '=' **NULL** ';'

e o dimensionamento de *strings* e matrizes:

dim identificador '[' expEscalar ']' ';' ⇒
SETM '(' identificador, expEscalar, 1, ""'identificador'"" ')';

dim identificador '[' $expEscalar_0$ ',' $expEscalar_1$ ']' ';' ⇒
SETM '('identificador, $expEscalar_0$, $expEscalar_1$, ""'identificador'""')';

SEQ 1.0 apresenta a possibilidade de redimensionar uma matriz ou uma *string* em tempo de execução. Esta característica advém do facto de matrizes serem objectos alocados no *heap*. Assim o comando **dim** pode ser utilizado em qualquer ponto do código, sobre a mesma variável, tantas vezes quantas as necessárias. Deve ser tomado em conta que esta operação poderá destruir o conteúdo anterior da variável. Por outro lado, é necessário declarar as

variáveis antes do seu uso. Tal é validado no analisador semântico, o que faz com que SEQ 1.0 não seja uma linguagem livre de contexto.
A eliminação destas variáveis têm a seguinte tradução:

clear identificador ';' ⇒ **free '**('identificador ')' ';'

O comando **clear** faz com que o apontador para matriz em C fique a apontar para NULL.
Quanto às declarações de funções:

[tipo] identificador '(' [lista_parâmetros] ')' ⇒
tipoC identificador '(' lista_parâmetros ')'

se em SEQ 1.0 o tipo estiver ausente ou for nulo em C paralelo corresponde a **void**. Assim os tipos traduzem-se:

tipo ::= **integer** | **real** | **string** | **matrix** | nulo ⇒
tipoC ::= **INTEGER** | **REAL** | **matrix** | **matrix** | **void**

E a lista de parâmetros:

tipo identificador | tipo identificador ',' lista_parâmetros ⇒
tipoC identificador | tipoC identificador ',' lista_parâmetros

O tipo **matrix** em C refere-se a um ponteiro para **REAL**, isto é um **float** por defeito ou opcionalmente um **double**. Por isso mesmo, em SEQ não é necessário especificar a dimensão de um argumento matricial, quando se descreve a lista de parâmetros de uma função. É no entanto aplicada a verificação de compatibilidade entre os tipos dos parâmetros e dos argumentos.
Por outro lado, nem todos os tipos de parâmetros podem ser utilizados para retornar valores, apenas matrix e **string**. Valores escalares, inteiros ou reais, só podem ser devolvidos pela função após a chamada a esta, o que implica que apenas um valor pode ser devolvido.
Quanto ao corpo da função, tanto em C como em SEQ, este é constituído por um bloco de código delimitado por chavetas onde no início devem ser colocadas as declarações e só depois um conjunto de comandos. Estes comandos são terminados com ';', em ambas as linguagens e

é também possível introduzir comandos nulos, isto é conjuntos consecutivos do *token* ';'. As expressões e as chamadas a funções são também consideradas comandos. As primeiras serão tratadas no ponto seguinte, quanto às chamadas de funções estas seguem a mesma sequência em ambas as linguagens:

identificador '(' [lista_argumentos] ')'
lista_argumentos ::= (expEscalar | identificador | chamadafunção) |
(expEscalar | identificador | chamadafunção ',' lista_argumentos)

Os comandos que faltam referir têm a ver com o fluxo de execução do programa. A execução dependente da avaliação de uma dada expressão, tem a seguinte tradução:

if expEscalar **then** Instrução [**else** Instrução] ⇒
if '('expEscalar ')' Instrução [**else** Instrução]

A linguagem suporta duas sintaxes para ciclo. A primeira executa uma instrução ou um bloco de comandos, um dado número de iterações. Este número é especificado pelo início, o fim e o valor do incremento da variável de ciclo. Os limites e o incremento podem ser expressões desde que retornem um valor inteiro, ou que este valor possa ser convertido para inteiro:

for identificador '=' expEscalar $_0$ (**to** | **downto**) expEscalar $_1$ [**step** expEscalar $_2$] Instrução ⇒
for '(' identificador '=' expEscalar $_0$';' identificador ('<=' | '>=')
expEscalar $_1$';' identificador '+=' expEscalar $_2$ ')' Instrução
ou na ausência de **step**:
expEscalar $_1$';' identificador ('++' | '—') expEscalar $_2$ ')' Instrução

A segunda forma de ciclo, executa um conjunto de comandos enquanto uma determinada expressão for verdadeira, isto é, a sua avaliação deve resultar num valor não nulo.

while expEscalar Instrução ⇒ **while** '('expEscalar ')' Instrução

Para interromper um ciclo, de modo que o fluxo de execução se reinicie após o ciclo, deve-se utilizar o comando:

exit ';' ⇒ **break** ';'

Este comando no entanto tem um significado diferente, dependendo de onde se encontra. Se for encontrado numa função, faz com que o contador de programa seja carregado com o endereço da instrução seguinte à chamada a essa função. Na prática a função é interrompida, podendo ser devolvido um valor ao segmento de código donde foi chamada:

exit [Expressão] ';' ⇒ **return** [Expressão] ';'

Se for encontrado no programa principal, isto é a seguir à palavra reservada **program**, que quando traduzido para C será colocado no corpo da função main, envia uma mensagem ao servidor de entrada-saída, pedindo o serviço END_SERVER, que faz com que este servidor termine e dessa forma termine também a aplicação.

6.1.3.2 Expressões e atribuições

As expressões aritméticas e lógicas, referidas também como escalares, são idênticas tanto em SEQ 1.0 como em C. No caso de operações com matrizes deve ser efectuada uma tradução de modo que sejam chamadas as funções correspondentes aos operadores.
Uma operação é matricial se um dos operandos for uma matriz. É assim necessário, que quando da análise semântica, discutida nos pontos seguintes, a cada operando seja atribuído uma classificação quanto ao tipo, ou pelo menos quanto à sua condição ou não de escalar.
Assim, em expressões matriciais o operando '+' será tratado pela função ADDW, o '*' por MULTW e a inversão por INVW, já introduzidas no capítulo anterior e que permitem que se especifique que as operações operam sobre escalares ou matrizes, e que estas podem ser transpostas. Só no caso de se pretender atribuir a um identificador o resultado da transposição doutro é que será necessário chamar a função de transposição MTRANS, explicitamente.
No caso dos operadores '= =' e '!=', respectivamente igualdade e desigualdade, quando ambos os operadores forem matrizes, será utilizada a função ISEQMTX.
Já as atribuições também podem ser de vários tipos, por isso a sua tradução é complexa e melhor ilustrada por um diagrama de sintaxe, o qual pode ser consultado no apêndice G.

6.1.3.3 Directivas

O tradutor suporta algumas directivas, que não fazem parte da linguagem SEQ 1.0. A primeira permite incluir um ficheiro noutro:

inclusão ::= ***include*** *cadeia*

É assim possível expandir um ficheiro noutro. Esta directiva é tratada em primeiro lugar, de modo que a análise do código SEQ 1.0 seja feita num ficheiro temporário expandido. A segunda permite definir macros, e é traduzida por:

macro ::= ***define*** $cadeia_0$ $cadeia_1$ ⇒ ***#define*** $cadeia_0$ $cadeia_1$

Sendo o pré-processador de C o responsável pelas substituições. Finalmente a directiva:

C ::= **C** { "código fonte c" } ⇒ "código fonte c"

permite escrever código fonte C. Todo o código entre as duas chavetas não será traduzido e será igual em todos os nós. Como o código C não é analisado, qualquer erro será apenas reportado pelo compilador de C utilizado e não pelo tradutor. No entanto, para eliminar ambiguidades quanto aos símbolos '{' e '}', também usados em C como delimitadores de blocos de comandos, é efectuada uma contagem destes símbolos, tomando em atenção que caracteres entre "" "" são parte de uma cadeia de caracteres. Assim, se no código C um destes símbolos não tiver uma chaveta contrária, irá induzir o tradutor em erro, pois o fim da directiva é encontrado quando o número de chavetas abertas for igual ao número de chavetas fechadas.

Esta directiva permite que se aceda em baixo nível, e sem restrições, ao código gerado pelo SPAM, e por isso mesmo é desencorajada a sua utilização. Pode no entanto ser útil para ultrapassar limitações da linguagem SEQ 1.0, visto que permite acesso de baixo nível ao SPAM, e através da linguagem C, usando a directiva asm permite também programação em *assembler*.

Quanto à passagem de variáveis entre SEQ 1.0 e C, esta ser feita utilizando o mesmo identificador, visto que estes representam o mesmo objecto tanto no contexto do código sequencial, como no contexto do código C paralelo gerado pelo SPAM.

Definem-se ainda os comentários, que também não fazem parte da linguagem e podem ser colocados em qualquer zona como qualquer cadeia de caracteres delimitada por /* e */, isto é:

comentário ::= '/' cadeia '*/' ⇒ '/*' cadeia '*/'*

6.1.4 Funções de base

As funções de base da linguagem SEQ 1.0, são na realidade chamadas às funções já definidas em C nos capítulos anteriores. Por isso todas as pré e pós condições anteriormente indicadas para cada função permanecem válidas. Para chamar funções C a partir de código SEQ 1.0 é usada a directiva **C**. Para as poder utilizar é necessário incluir o ficheiro onde estão definidas com:

include "SEQFBASE.INC"

As funções estão divididas em três grupos, onde os principais são constituídos por funções de entrada saída e funções de cálculo.

6.1.4.1 Entrada saída

Este grupo de funções permite implementar operações sobre ficheiros. Tal como em C,a consola é considerada composta por dois ficheiros, um de saída para o monitor e outro de entrada de dados provenientes do teclado. Estes dois ficheiros são abertos automaticamente após a inicialização da aplicação e têm como identificação 1 e 0, respectivamente. Também um descritor para um ficheiro de saída de erros padrão encontra-se permanentemente aberto e tem como identificação o inteiro 2. O utilizador pode pedir a abertura de outros ficheiros, cada um de entrada saída usando a função:

```
integer open (string s) {
C {     char s0[80];

        formstr(s, s0);
        return HOPEN(s0); } }
```

e onde a função auxiliar formstr se define como:

```
C {
char* formstr(matrix src, char *dst) {
int c, l;
matrix m;
```

```
        SETM (m, MTXELEM(src), 1, “SEQ 1.0 form temp local C string");
        COLLECT (src, m);
        for (i = 0; i < MTXELEM(m); i ++) dst[i] = (char) m [i + MDATA];
        free  (m);
        return dst; }
}
```

O parâmetro de entrada de open é uma **string**, isto é um ponteiro para uma cadeia distribuída. Como a função HOPEN actua apenas no nó 0, **s** é reconstruída nesse nó antes de ser pedida a abertura do canal usando formstr. A reconstrução da cadeia de caracteres que guarda o nome do ficheiro é feita em todos os nó pela função COLLECT. Uma optimização em termos de velocidade seria faze-la apenas no nó 0, no entanto o atraso não consiste na reconstrução dessa cadeia mas sim no pedido de abertura do canal ao anfitrião, de modo que para que o código seja mais compacto e legível, optou-se por esta abordagem.

Se a operação fôr bem sucedida, isto é se o canal for aberto normalmente, é retornada a identificação desse canal, um inteiro entre 3 e 32. Se algum erro suceder o resultado será menor que zero. Se o ficheiro não existir é criado um novo ficheiro.

Um ficheiro aberto, mesmo com índice inferior a 3, podem ser fechados com:

```
close (integer c) { C { HCLOSE(c);  } }
```

Um ficheiro é lido ou escrito sequencialmente. A única forma de alterar a posição do ponteiro obriga a reinicializar o descritor de ficheiro, fazendo com que o ponteiro volte a apontar o início do ficheiro:

```
reset (integer c) { C { HRESET(c);  } }
```

Sobre um qualquer ficheiro aberto podem ser efectuadas as seguintes operações de entrada saída:

```
writefi (integer c, integer n)      { C { WRITEI(c, &n); } }
writefr (integer c, real n)         { C { WRITEF(c, &n); } }
writefm (integer c, matrix m)       { C { WRITEM(c, m); } }
writefs (integer c, string s)       { C { WRITEMA(c, s); } }
integer readfi (integer c)          { C {   int t;
                                            READI(c, &t);
                                            return t; } }
real readfr (integer c)             { C {   int t;
                                            READF(c, &t);
                                            return t; } }
```

```
readfm (integer c, matrix m)        { C { READM(c, m); } }
```

As funções writef? e read? permitem escrever ou ler um determinado tipo de dados para um descritor de ficheiro especificado por *c*. Os caracteres ? indicam qual o tipo de dados que deve ser tratado: i, para **integer,** r para **real,** m para **matrix** e s no caso de **string**.

Já a função:

```
writefmd (integer c, matrix m) { C { WRITEMDIM(c, m); } }
```

permite imprimir as dimensões de uma matriz.
Se se pretender trabalhar com os ficheiros de entrada e saída padrão, podem-se utilizar as variantes:

```
writei (integer n)          { C { WRITEI(1, &n); } }
writer (real n)             { C { WRITEF(1, &n); } }
writem (matrix m)           { C { WRITEM(1, m); } }
writes (string s)           { C { WRITEMA(1, s); } }
integer readi ( )           { C {   int t;
                                    READI(0, &t);
                                    return t; } }
real readr ( )              { C {   int t;
                                    READF(0, &t);
                                    return t; } }
readm (matrix m)            { C { READM(0, m); } }
```

Também podem ser usadas macros pré-definidas para formatar a saída para a consola, mudando de linha ou adicionando espaços respectivamente;

```
define nl NL
define sp SP
```

Existe ainda a possibilidade de comunicar matrizes com um dispositivo qualquer, através de um ficheiro, no formato de vectores onde os elementos são arranjados ao longo das linhas da matriz. Estes vectores são precedidos por um inteiro de 4 *bytes* que indica o número de palavras, de 4 *bytes* também, que compõem o vector:

```
putv (integer c, matrix m) { C { PUTV(c, m); } }
getv (integer c, matrix m) { C { PUTV(c, m); } }
```

Embora não seja uma função de entrada saída, a função seguinte permite suspender temporariamente o servidor de acesso aos recursos do anfitrião.

```
pause ( ) { C { PAUSE( ); } }
```

6.1.4.2 Funções de cálculo

Se bem que o operador '/' permita multiplicar o operando esquerdo pelo inverso do direito, pode ser necessário obter apenas o inverso de uma matriz. A função seguinte faz exactamente isso. Deve no entanto ser tomado em conta que se o identificador ao qual for atribuído o resultado já guardar uma matriz, o espaço por esta ocupado deverá ser disponibilizado antes de chamar a função, usando o comando **clear.** Se tal não for efectuado, esse bloco de memória ficará indisponível durante o tempo de vida da aplicação.

```
matrix inv(matrix a) { C { return INVW(a, NULL, 0, 0); } }
```

As funções seguintes permitem adicionar e multiplicar a diagonal principal de uma matriz com um escalar real:

```
matrix dadd(matrix a, real n, matrix r) { C { return DIAGADD(a, n, r); } }
matrix dmult(matrix a, real n, matrix r) { C { return DIAGMULT(a, n, r); } }
```

Estão definidas também duas funções que implementam a rotação de elementos em vectores. Também podem ser utilizados em cadeias de caracteres, visto que estas são implementadas sobre vectores.

```
shl (matriz v, integer início, integer fim, real n) {
        C { SHFL(v,  início, fim, n); } }

shr (matriz v, integer início, integer fim, real n) {
        C { SHFR(v,  início, fim, n); } }
```

início e *fim* delimitam a zona do vector onde é efectuada a rotação para a esquerda ou para a direita. O elemento que é rodado para fora dessa área é eliminado, enquanto o espaço que fica livre no extremo é preenchido com *n*. É preciso tomar em conta que embora **v** tenha que ser um vector para que a operação seja efectuada correctamente, esta validação só é feita em tempo de execução. Aliás todas as validações referentes à dimensão dos operandos são apenas efectuadas em tempo real.

As seguintes operações são definidas apenas sobre escalares, podendo também ser aplicadas a elementos individuais de matrizes.

A conversão de dados de inteiro para real é implícita; no entanto para se obter a parte inteira ou a fraccionária de um real, com sinal, devem ser usadas as funções:

```
integer rint (real n) {
        C { return  (int) floor (n); } }

integer rfrac (real n) {
        C { double f;
              return  (int) modf (n, &f); } }
```

O valor absoluto de um escalar e o seu sinal são retornados por:

```
real mod   (real r)    { C { return fabs(r); } }

real sgn   (real r) {
       if r<0 then { exit -1; }
       else        { exit  1; }  }
```

Exponenciais naturais e logaritmos tanto naturais como decimais bem como potências de qualquer base são definidas como:

```
real expn  (real r)             { C { return exp(r); } }
real logn  (real r)             { C { return log(r); } }
real logd  (real r)             { C { return log10(r); } }
real power (real b, real e)  { C { return pow(b, e); } }
```

Finalmente as funções trignométricas básicas:

```
real cosen (real r)             { C { return cos(r); } }
real sen   (real r)             { C { return sin(r); } }
```

Como se pode observar grande parte destas funções são definidas à custa da versão em C. Isto indica que um programa SEQ pode sempre recorrer a qualquer função definida em C sobre escalares reais e inteiros, simplesmente usando a directiva C { }, e utilizando em C o mesmo identificador usado para as variáveis SEQ.

6.1.4.3 Funções Diversas

As funções descritas neste ponto têm usos diversos. As três primeiras retornam retorna o comprimento de uma cadeia de caracteres distribuída, e o número de linhas ou colunas de uma matriz respectivamente:

```
integer length (string s)     { C { return MTXELEM(s); } }
integer rows (matrix m)       { C { return NLINHAS(m); } }
integer cols (matrix m)       { C { return NCOLUNAS(m); } }
```

O acesso aos argumentos da linha de comandos, no formato de inteiro, real ou string:

```
integer argi (integer c)      { C {  int t;
                                     ARGI(c, &t);
                                     return t; } }
real argr (integer c)         { C {  float  t;
                                     ARGF (c, &t);
                                     return t; } }
args (integer c, string s)    { C {  char buf [256]; int i;
                                     ARGS (c, buf);
                                     for(i=0; i!='\0'; i++) MATRIX_PUT(s, i, 1, buf [i]); } }
```

onde *c* indica qual o argumento que se pretende aceder. O argumento índice 0 refere-se ao nome da aplicação. O número de argumentos pode ser determinado usando:

```
integer argcnt ( )    { C {  int t;
                             ARGC (&t);
                             return t; } }
```

A atribuição de valores a matrizes pode ser feita usando:

```
matrix fill (matrix m, real n)      { C {return matrix_fill(m, n); } }
matrix ident (matrix m)             { C { return matrix_ident(m); } }
matrix random (matrix m)            { C { return matrix_rand(m); } },
```

onde a função fill atribuí a todos os elementos de uma matriz o escalar *n*, se **m** for quadrada, ident transforma-a na matriz identidade, e random atribuí valores aleatórios reais entre 0 e 1 aos elementos de **m**. Para que o gerador de números pseudo-aleatórios tenha um comportamento diferente em cada execução, deve ser usado:

```
randomize ( )         { C { RANDOMIZE(time(NULL)); } }
```

Finalmente, pode ser terminado o programa e enviada uma mensagem de erro para a consola, dada por s, com:

```
error (string s) {
C {     char s0[80];

        #ifdef RTE
                formstr(s, s0);
                ERRORMSG(USER_DEF, s0);
        #endif
} }
```

6.1.4.4 Medição do tempo de execução

O tempo do sistema pode ser obtido com:

```
integer times ( ) { C { return timer_now( ); } }
```

ou

```
integer times ( ) { C { return time(NULL); } },
```

dependendo da implementação. O desempenho em cada processador pode ser medido e guardado num vector distribuído usando:

```
reset_clock( )      { C {START_CLOCK( ); } }
watch_clock( )      { C {READ_CLOCK( ); } }
timem()             { C { TIMEM; } }
```

O valor temporal entre a chamada a reset_clock e watch_clock, em cada processador, é guardado no vector de tempos usando a última função. Estes tempos podem ser impressos para um ficheiro *c* usando:

```
timev (integer c)     { C { TIMEW(c); } }
```

Também se pode imprimir a relação MFlops / s para um dado número de *flops*:

```
timef (integer c, integer f)   { C { TIMEWF(c, f); } }
```

Estas funções lêem o relógio de tempo real do processador, de modo que se for utilizado um micro-núcleo de tempo real como base do SPAM, como é o caso da implementação aqui apresentada onde é usado um micro-núcleo 3L para todos os processadores, podem ser usadas para verificar o cumprimento de metas temporais, ou *deadlines*, e permitir que o fluxo do programa seja desviado para efectuar o tratamento necessário.

6.2 O tradutor

Tipicamente a compilação segue o fluxo da figura seguinte (Aho et al, 1986). Antes do módulo compilador pode ainda existir um pré-processador, que prepara o código fonte, expandindo macros, e processando directivas. A etapa final do compilador pode gerar código *assembler* que será injectado num assemblador de modo a ser produzido o código objecto, como é o caso dos compiladores de C. No entanto, em linguagens como o Pascal, o compilador poderá produzir como código objecto p-code em vez de código de máquina.
Se se comparar a figura seguinte com o modelo de programação introduzido no capítulo 3, Fig. 3-1, verifica-se serem muito semelhantes. No caso do modelo de programação paralelo, falta ainda um módulo a seguir ao *linker*, responsável por alocar e assignar as tarefas aos processadores mediante instruções do programador. No caso do modelo de programação adoptado pelo SPAM 1.0, existe também um tradutor entre o editor e o compilador, responsável por traduzir a linguagem sequencial SEQ 1.0 em código C paralelo.

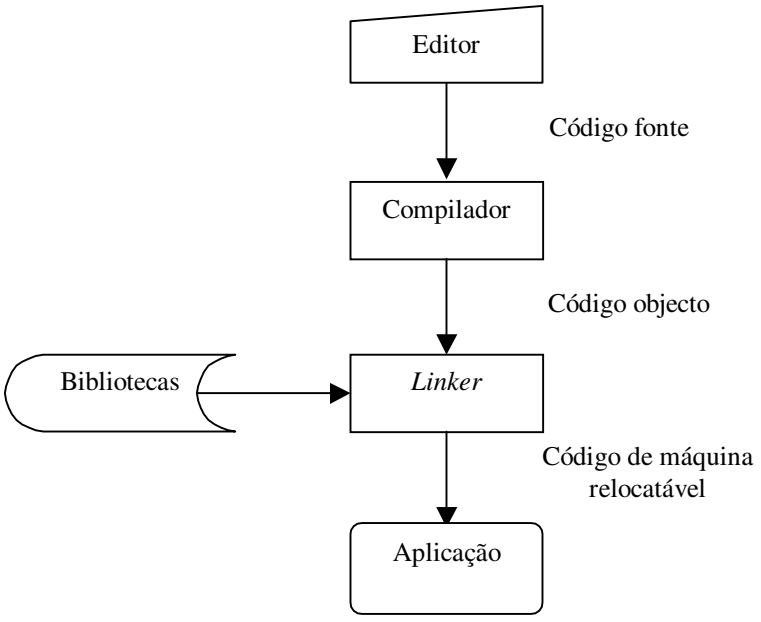


Fig. 6-1: Geração de código executável

A acção do tradutor é semelhante à vanguarda de um compilador. De facto, um compilador pode ser dividido em duas etapas principais: vanguarda e retaguarda. A vanguarda é responsável pela análise do código fonte e gera uma representação intermédia que é injectada na retaguarda. Esta por sua vez é responsável por gerar o código objecto e é diferente para cada tipo de processador alvo.

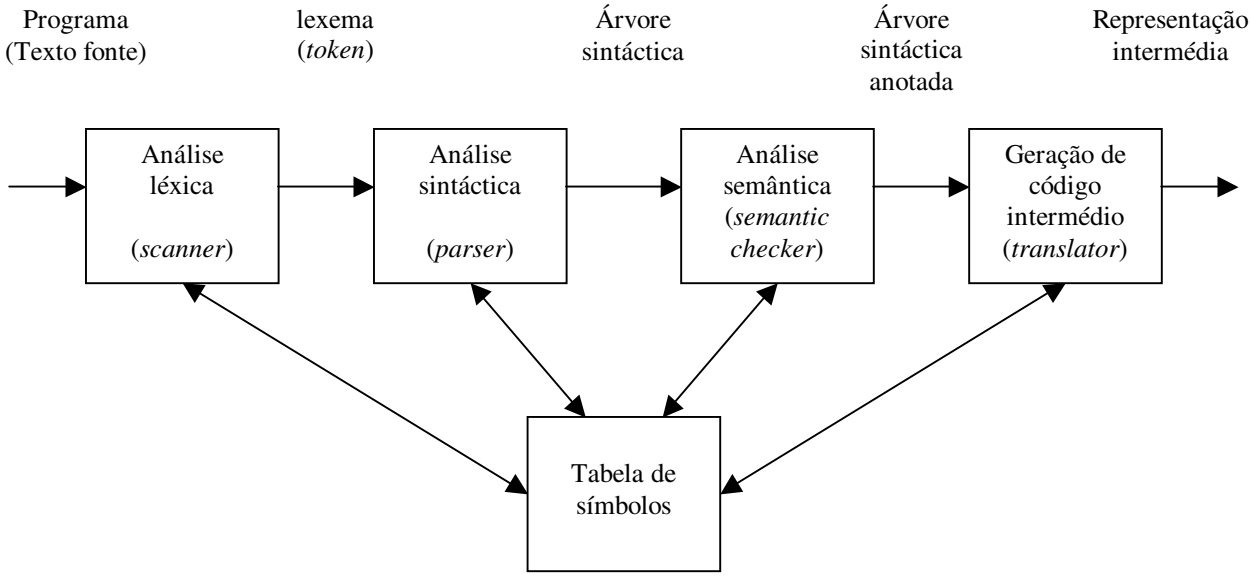


Fig. 6-2: Vanguarda do compilador

A grande vantagem desta divisão, e da representação intermédia, é precisamente facilitar a adaptação da geração de código a várias famílias de processadores. Na figura acima está representado o diagrama da vanguarda de um compilador e na figura abaixo do tradutor desenvolvido. Neste último caso, a representação intermédia consiste em código C paralelo.
Neste tradutor, a implementação da análise sintáctica e léxica é efectuada num só módulo, como aliás é prática corrente. O analisador léxico funciona como um servidor do analisador sintáctico e semântico, o qual pede lexemas ao primeiro á medida que necessita destes para efectuar a tradução. O objectivo de cada uma das etapas de análise do tradutor é a seguinte:

a) análise léxica
   - Separa o texto de entrada em *tokens* ou lexemas
   - Detecta textos de entrada com *tokens* ou caracteres ilegais:
   - se for achado um *token* que não existe na gramática, é considerado um identificador

- todos os caracteres ASCII são apenas válidos em cadeias de caracteres delimitadas com o lexema “
- localiza *tokens* na linha do texto fonte
- elimina separadores e espaços em branco

b) análise sintáctica
- Detecta erros de sintaxe, isto é sequências de *tokens* não suportadas pela gramática de SEQ 1.0, seguindo os diagramas de sintaxe que serão apresentados a seguir.

c) análise semântica
- verifica se todos os identificadores foram declarados
- verifica se não existem declarações múltiplas ou incompatíveis
- determina os tipos de expressões e variáveis e a sua compatibilidade com os operadores
- verifica se o tipo dos argumentos de uma chamada a função correspondem ao tipo dos parâmetros da declaração de função.
- evita a utilização de palavras reservadas como identificadores

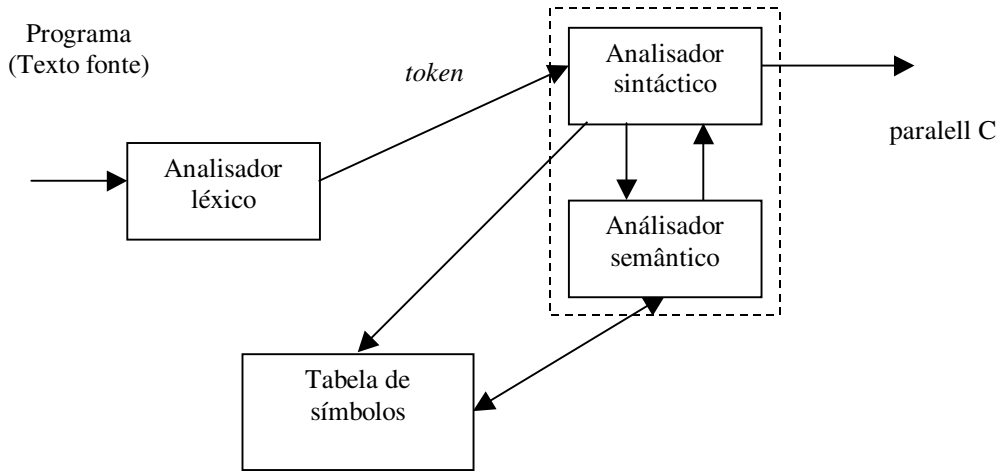


Fig. 6-3: Tradutor

A tabela de símbolos normalmente é actualizada por todos os módulos. Na análise léxica são criadas as entradas, enquanto na análise sintáctica os atributos são preenchidos, e na análise semântica esta é consultada de modo a validar operações sobre estes símbolos. No entanto o tradutor implementado cria as entradas e preenche-as na análise sintáctica. Como esta é efectuada pelo mesmo módulo que a análise semântica a tabela de símbolos só é acedida por

este. No tradutor, a tabela de símbolos consiste principalmente em validações do tipo de dados em expressões, atribuições e passagem de argumentos para funções.
A estrutura de cada entrada na tabela de símbolos é a seguinte:

```
struct symbol {
        char nome [16];
        Tipo_dados dtipo;
        Tipo_id tipo;
        int dim;
        Tipo_dados *param; }

enum (nulo, integer, real, string, matrix) Tipo_dados;
enum (var, fnc) Tipo_id;
```

Onde o nome é o lexema, isto é o identificador, dtipo corresponde ao tipo de dados da variável ou ao tipo de função, isto é o tipo de dados que é retornado por esta. Só uma função pode ter tipo nulo. Também neste caso, param aponta uma lista de tipos de dados de cada parâmetro. No caso de uma variável ou de uma função sem parâmetros, o ponteiro *param é nulo.
O atributo tipo indica se o identificador se refere a uma função ou variável e o atributo dim é não nulo se a variável já foi dimensionada com um comando **dim**. Se a memória alocada a uma variável fôr libertada com um comando **clear** este atributo volta a ser nulo. No caso de uma função é sempre não nulo.
Esta tabela de símbolos guarda a informação suficiente para que as expressões matriciais, possam ser decompostas nas funções que implementam as operações. No entanto, num compilador outro atributos, tais como o endereço relativo, global, local, modificável, poderiam ser necessários.

6.2.1 Enquadramento do tradutor no SPAM 1.0

Visto que o processo de controlo é composto por um servidor que estabelece a comunicação com o anfitrião, mediante pedidos dos nós, este código não é modificável, constituindo uma tarefa que será simplesmente alocada no processador raiz. Deste modo o tradutor apenas gera o código para os nós, isto é as instancias da aplicação Prg (*x*). A figura seguinte mostra o tradutor em termos de entradas e saídas.
Este é composto basicamente por dois módulos. TransRouter, que cria o ambiente de comunicações a partir de um ficheiro de base, idêntico para qualquer topologia de rede. Durante a inicialização da aplicação de acordo com a informação existente nos ficheiros de

inicialização e de configuração da rede, gerados pelo módulo configurador, este ambiente é configurado adequadamente à rede alvo.

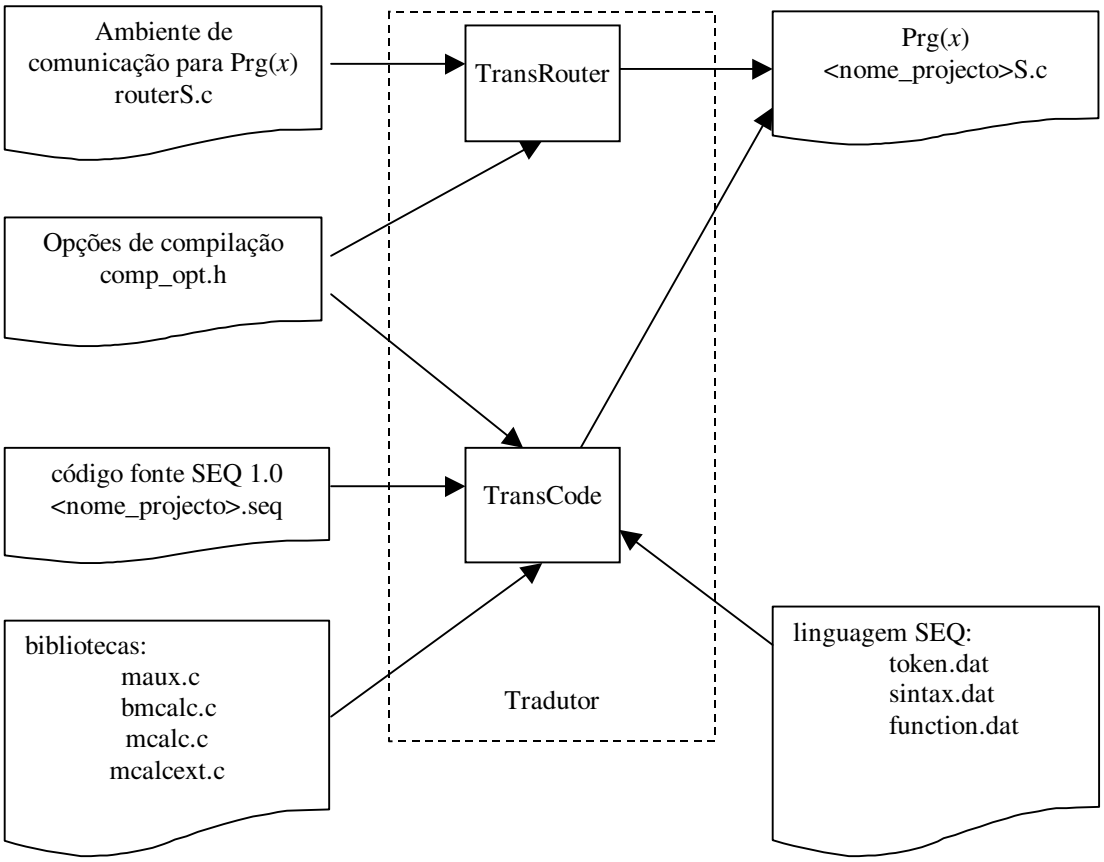


Fig. 6-4: O tradutor quanto a entradas e saídas

O outro módulo, TransCode, é o responsável pela tradução de código escrito na linguagem SEQ para C paralelo, e gerar o código das instâncias da aplicação, e cuja decomposição já foi indicada na Fig. 6-3.

Ambos os módulos recebem como entrada o ficheiro de opções do tradutor, onde é definido se o código gerado imprime informação adicional para depuração, ou se o sistema de tratamento de erros em tempo real deve ser ou não incluído e activado. Neste ficheiro também se define o comprimento do pacote de informação que circula na rede.

O tradutor é parcialmente configurável para que o SPAM seja aberto a futuras expansões e alterações. Assim, para que o utilizador possa adicionar novas funções às bibliotecas padrão do SPAM, o tradutor deve ser suficientemente flexível para reconhecer novos pares de funções na linguagem SEQ 1.0 e a sua correspondência para C paralelo. Estes pares são assim adicionados a um ficheiro, "function.dat" onde são mantidas todas as funções da linguagem

SEQ 1.0 e a sua correspondência em C paralelo. É assim possível alterar comandos padrão e adicionar novos. Também na sua maior parte a gramática fonte e a sua tradução podem ser alteradas, visto que o tradutor é essencialmente um mecanismo que segue os diagramas de sintaxe da linguagem SEQ 1.0. Estes diagramas são grafos orientados que são expressos em matrizes de adjacência, e são guardados no ficheiro "sintax.dat". Os *token* e outros símbolos são guardados no ficheiro "token.dat"

Se se pretender encapsular o código num bloco *SIMULINK* é necessário dividir o código em duas secções, tal como está referenciado na Fig. 3-6. Assim, em cada instante, o SIMULINK envia dois possíveis sinais para a aplicação paralela, que consistem numa palavra, do modo descrito no capítulo seguinte. No lado da aplicação, este sinal deve ser descodificado em cada instante, de modo que se entre numa secção de inicialização e dimensionamento de dados ou que se execute o processamento requerido pelo algoritmo. Isto é da responsabilidade do programador, que deve usar funções putv para receber os sinais. Este encapsulamento será tratado no próximo capítulo.

6.2.2 Implementação do tradutor

A notação EBNF em que a linguagem SEQ 1.0 foi definida pode ser convertida em diagramas de sintaxe, que na realidade são grafos orientados. Estes podem ser expressos por uma matriz de adjacência de modo que um mecanismo automático possa percorrê-los, validando uma secção de código fonte, e operando transformações nos lexemas. Este é o princípio do tradutor implementado. A sua implementação foi efectuada com o intuito de ser o mais simples possível, no entanto adaptável. Por esta mesma razão, e também para manter o SPAM o mais independente possível, não se recorreu a geradores de analisadores léxicos, como por exemplo o flex (Paxson1990).

6.2.2.1 O analisador léxico

Um modo de implementar a análise léxica consiste no uso de um autómato, isto é um grafo orientado onde cada nó corresponde a um estado e cujos arcos indicam qual o *token* que valida determinada transição entre dois estados. Uma forma mais simples de abordar a implementação, consiste em formar cadeias de caracteres, por concatenação dos caracteres provenientes do ficheiro com o código fonte, enquanto a cadeia fizer parte da linguagem. Desta forma garante-se que o maior lexema pertencente ao vocabulário é o reconhecido, o que

será necessário para distinção de lexemas onde apenas os primeiros caracteres são idênticos, como é o caso dos operadores formados com dois caracteres.
Os lexemas da linguagem SEQ 1.0 são classificados nos seguintes tipos:

tipo_token ::= identificador | inteiro | real | cadeia | opBinário | opPrefixo | opSufixo | parenteses | pontuação | palavra_reservada

parenteses ::= parentesesRecto | parentesesCurvo
parentesesRecto ::= '[' | ']'
parentesesCurvo ::= '(' | ')'

pontuação ::= ',' | ';' | ':' | '{' | '}' | '/*' | '*/' | ' " ' | '#'

palavra_reservada ::= **if | then | else | for | to | downto | step | while | exit | program | integer | real | matrix | string | dim | clear | '='**

A atribuição, a qual é efectuada usando o símbolo '=', embora não seja um operador, poderá ser vista como tal, mas sendo o de menor prioridade, de modo que a expressão à sua direita seja avaliada antes do resultado ser atribuído ao identificador à sua esquerda. Por outro lado pode também ser visto como uma palavra reservada. Qualquer das abordagens é perfeitamente válida, e apenas depende de opções de implementação dos analisadores. Para a corrente implementação optou-se por tratá-lo como uma palavra reservada.
O analisador léxico empacota os *tokens* num registo, juntamente com os seus atributos antes de os enviar como resposta ao pedido do analisador sintáctico e semântico. Esta estrutura tem a forma:

```
struct token {
        Tipo_token ttipo;
        int linha;
        union {
                char *nome;
                INTEGER valor_inteiro;
                REAL valor_real; } lexema; }
```

Os atributos consistem no tipo do *token*, na linha em que foi encontrado no texto fonte, e no lexema ou se for uma constante inteira ou real o seu valor.

O analisador léxico também é responsável por copiar os caracteres brancos para o ficheiro com destino, na mesma posição onde foram encontrados no código fonte SEQ 1.0, de modo a manter a formatação na tradução para C paralelo. Os caracteres brancos são:

```
brancos ::=     TAB | LF | NL | SP
TAB     ::=     9
LF      ::=     10
CR      ::=     13
SP      ::=     32
```

Também os comentários são copiados para o ficheiro destino exactamente na mesma posição que ocupavam no código fonte. A directiva C têm também um tratamento semelhante. Tudo o que se encontrar entre os limitadores de bloco '{' e '}' é copiado para o ficheiro destino por este módulo.

O tratamento da directiva include não é efectuada pelo analisador léxico, mas sim numa etapa prévia, onde todos os ficheiros são expandidos num temporário, que será a fonte deste analisador. A busca do ficheiro é efectuada primeiro relativamente à pasta onde estão guardados os ficheiros de projecto, seguidamente à pasta que contém as bibliotecas, e finalmente à raiz do sistema de ficheiros.

A configuração deste analisador é efectuada através de um ficheiro de texto onde na primeira linha devem estar os códigos dos caracteres brancos separados por espaços, na segunda os caracteres de pontuação, na terceira os operadores e na quarta as palavras reservadas. Todos os lexemas devem estar separados por espaços. Os operadores devem ainda ter a seguir ao símbolo os índices de prioridade da operação. Este índices devem ser dois, o primeiro para operador unário e o segundo para operador binário. Um valor não nulo de um destes índices indica a aridade do operador.

6.2.2.2 O analisador sintáctico e semântico

Este analisador valida a sintaxe, isto é uma sequências de lexemas válida em SEQ 1.0, percorrendo diagramas de sintaxe derivados da formulação em EBNF já apresentada. O conjunto destes diagramas pode ser consultado no apêndice G. Neste ponto serão apresentados apenas os necessários para ilustrar o processo de análise sintáctica e tradução associada. Os diagramas são organizados numa perspectiva de cima para baixo, de modo que a análise inicia-se em:

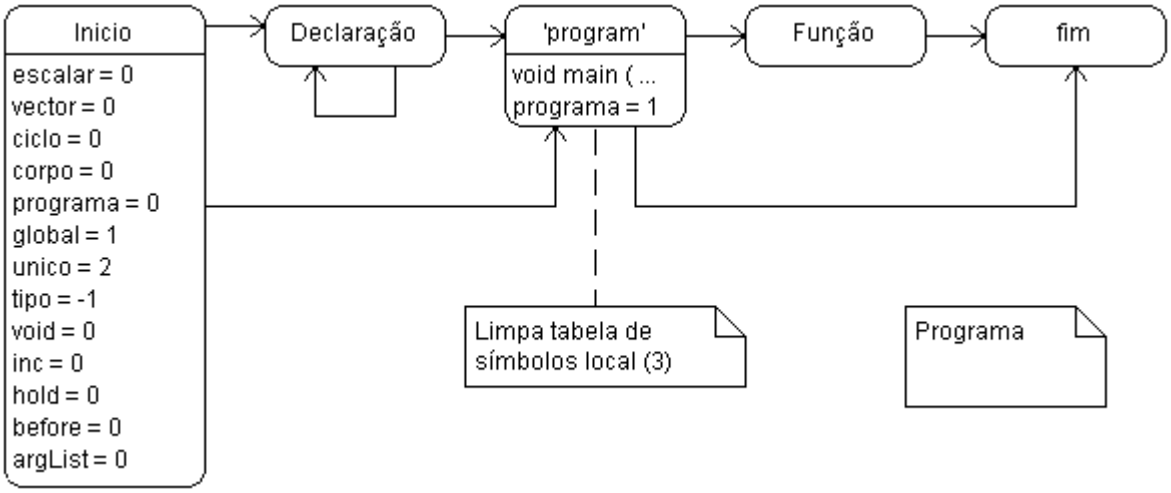


Fig. 6-5: Diagrama de entrada na linguagem SEQ 1.0

Cada nó destes diagramas tem no topo a sua identificação, que pode ser dada por um lexema, pelo tipo deste ou pelo nome de outro diagrama para o qual se deve navegar. O tradutor usa este campo para validar a sintaxe.

Nas linhas seguintes tem-se as regras de tradução, e finalmente nas últimas linhas as actuações nas variáveis de estado. Estas variáveis de estado são principalmente usadas para comutar arcos virtuais. Este tipo de arco distingue-se por ter associada a indicação da variável de estado a que responde, sendo esta precedida por um ponto de exclamação se obedecerem a lógica inversa.

As variáveis inicializadas no nó Início do diagrama Programa, acima apresentado podem ser vistas como as condições iniciais de uma máquina de estados. A partir deste diagrama acima pode-se navegar para outros dois, Função e Declaração:

As regras de tradução são muito simples. Consistem numa cadeia de caracteres que será escrita no ficheiro traduzido. O símbolo especial @ indica que se deve utilizar um dado *token* anterior. O *token* a recuperar é indicado por um digito que segue @. Quanto maior for este digito mais atrasado será o *token* a recuperar, sendo 0 o *token* actual.

Para este efeito é utilizada uma estrutura de dados onde são armazenados os últimos *tokens*, e que se comporta como uma pilha endereçável. Cada vez que a navegação entra num novo diagrama, isto é num nó Início, é criada uma nova pilha apenas com o *token* mais recente, de modo que esta estrutura é local a cada diagrama sendo sempre eliminada num nó Fim. Esta pilha têm um tamanho limitado apenas pela memória disponível, mas pode e deve ser limpa sempre que não seja necessário utilizar *tokens* anteriores na tradução, usando @@. É ainda utilizado @! para substituir a sequência @0@@, que é bastante utilizada.

Não deve ser confundido o número de *tokens* armazenados na pilha, com os *tokens* necessários de analisar à frente para resolver ambiguidades na linguagem SEQ 1.0, e que é apenas de um. Isto é a gramática pode ser classificada como LL (1). O número de *tokens* armazenados na pilha é sim o necessário para que se possa efectuar a tradução, sempre que a sintaxe de duas linguagens implicar que a ordem pela qual são traduzidos os *tokens* não é idêntica em ambas.

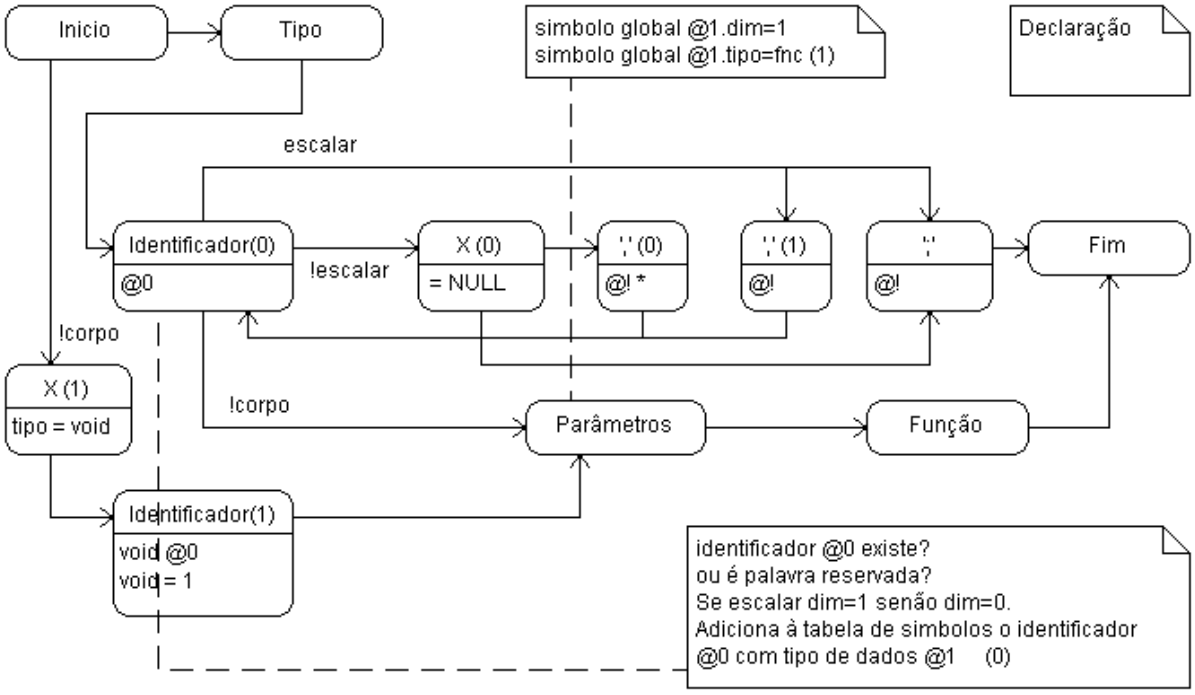


Fig. 6-6: Declaração em SEQ 1.0

Sempre que existe uma chamada a outro diagrama como é o caso de Parâmetros no diagrama anterior, o motor chama-se a si próprio recursivamente de modo a preservar o estado, sendo possível voltar ao ponto seguinte à chamada. Assim, traduzir, alterar valores das variáveis de estado ou chamar funções de análise numa referência para um outro diagrama, é efectuado antes de se comutar de diagrama. Se for necessária alguma operação após a comutação, pode-se sempre adicionar um nó virtual depois da referência ao diagrama.

Este tipo de nós não corresponde a um *token* lido do código fonte, mas indica alguma acção de tradução baseada nos *tokens* anteriores. Nestes, @0 referencia o *token* correspondente ao próximo nó não virtual e já existente na pilha, pois o próximo *token* é carregado imediatamente após o anterior ter sido tratado. Os nós Início e Fim, que obrigatoriamente indicam o início e o fim de um diagrama são nós virtuais. Nós virtuais definidos pelo utilizador, são convencionalmente designados por X.

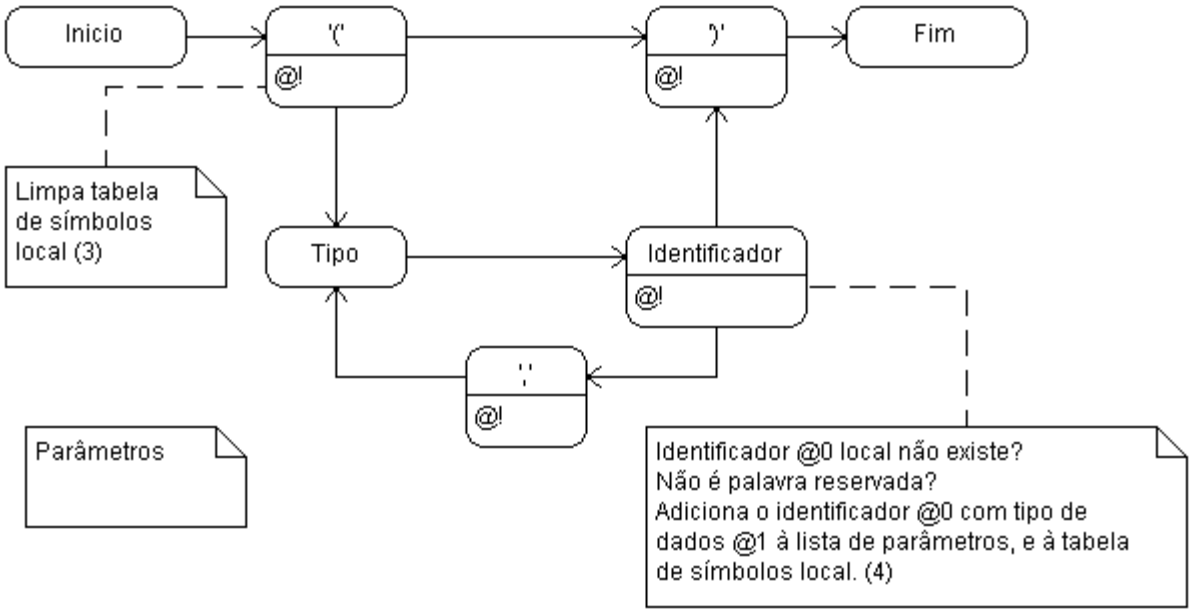


Fig. 6-7: Parâmetros de uma função

Aos nós podem ainda estar associados um comentário e que indica principalmente quais as validações, ou acções necessária para futuras validações semânticas a tomar nesse nó. As validações são apresentadas na forma de uma pergunta que, se a resposta for falsa, implica a geração de um erro de análise. Já as acções são afirmações que só serão executadas se todas as perguntas prévias obtiverem uma resposta verdadeira. Estes comentários são implementados como funções, que devem ser redefinidas para cada conjunto de diagramas de sintaxe. A análise indicada nos comentários é efectuada antes do motor comutar os arcos de acordo com as variáveis de estado actuadas no nó, de modo que é possível alterar variáveis de estado e desse modo também diagramas de sintaxe.

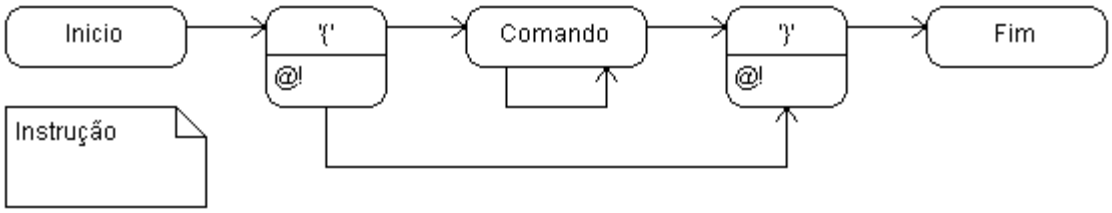


Fig. 6-8: Instrução SEQ 1.0

Informação sobre os identificadores é mantida em duas tabelas de símbolos, global e local, de modo que seja possível verificar a sua existência no contexto local ou global e evitar também

a sua duplicação. Deste modo, antes do lexema **program**, único em cada programa, e fora das declarações de funções são adicionados símbolos à tabela global. Em qualquer outro caso os símbolos são adicionadas à tabela local a cada função, sendo o corpo do programa, isto é, o código que segue o lexema **program,** visto também como uma função.

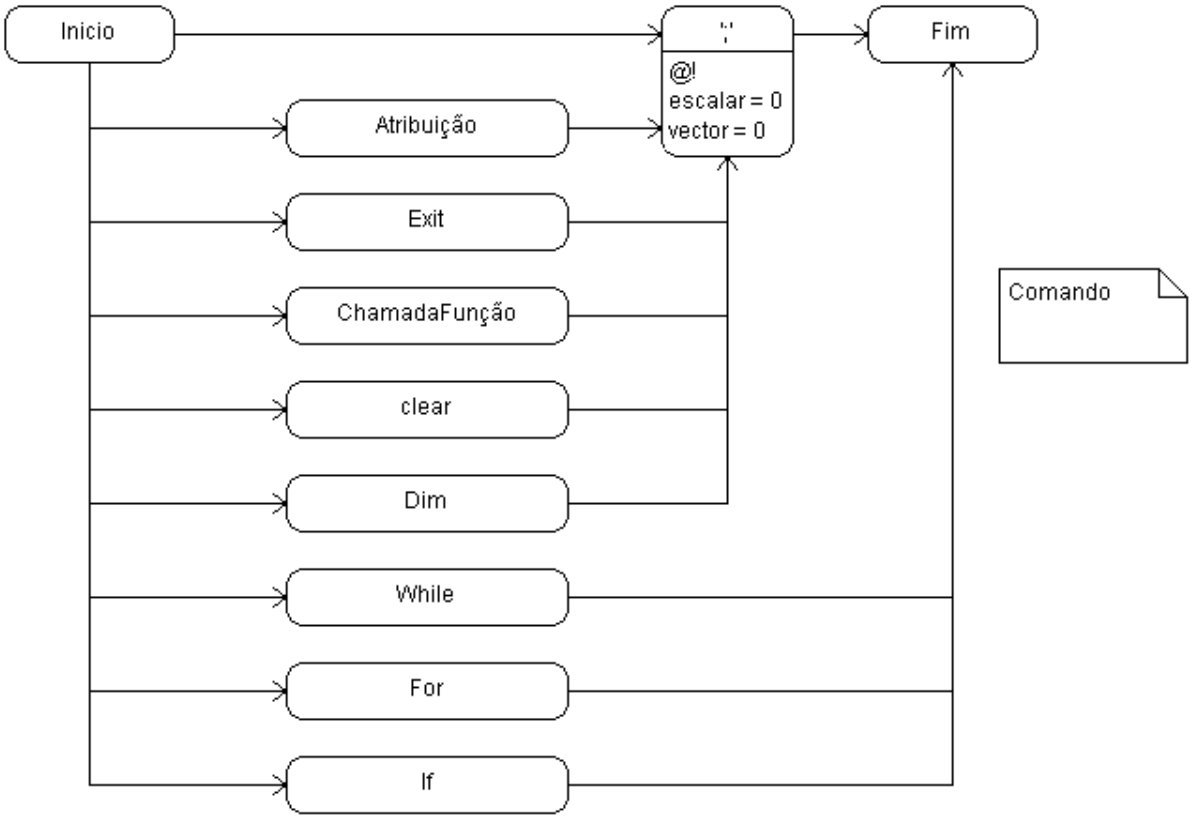


Fig. 6-9: Comandos SEQ 1.0

O contexto das variáveis locais termina quando termina uma declaração de função, de modo que a tabela local só existe enquanto é analisada esta declaração, tendo cada função a sua própria tabela de símbolos.

Podem existir símbolos locais com o mesmo nome de globais, tendo os primeiros predominância sobre os globais, pois a busca de símbolos é efectuada primeiro na tabela de símbolos local.

O diagrama da Fig. 6-9, faz a chamada dos sub-diagramas que processam os comandos. Repare-se que expressões matriciais apenas podem ser utilizadas nas atribuições e como argumento para o comando Exit. Em qualquer outro caso terão de ser utilizadas apenas expressões escalares. Deste modo existe um processamento diferente para ambos estes tipos de expressões, onde as escalares consistem numa tradução idêntica ao passo que as matriciais

requerem um módulo próprio, o analisador de expressões, que deverá ser iniciado apenas quando necessário no nó ';' do diagrama Fig. 6-9.

O diagrama anterior é relativo à análise de expressões. O símbolo @#X é usado para adicionar a uma fila infixa o X último *token* recebido. Os *tokens* desta fila só serão processados pelo analisador de expressões, que será discutido no ponto seguinte.

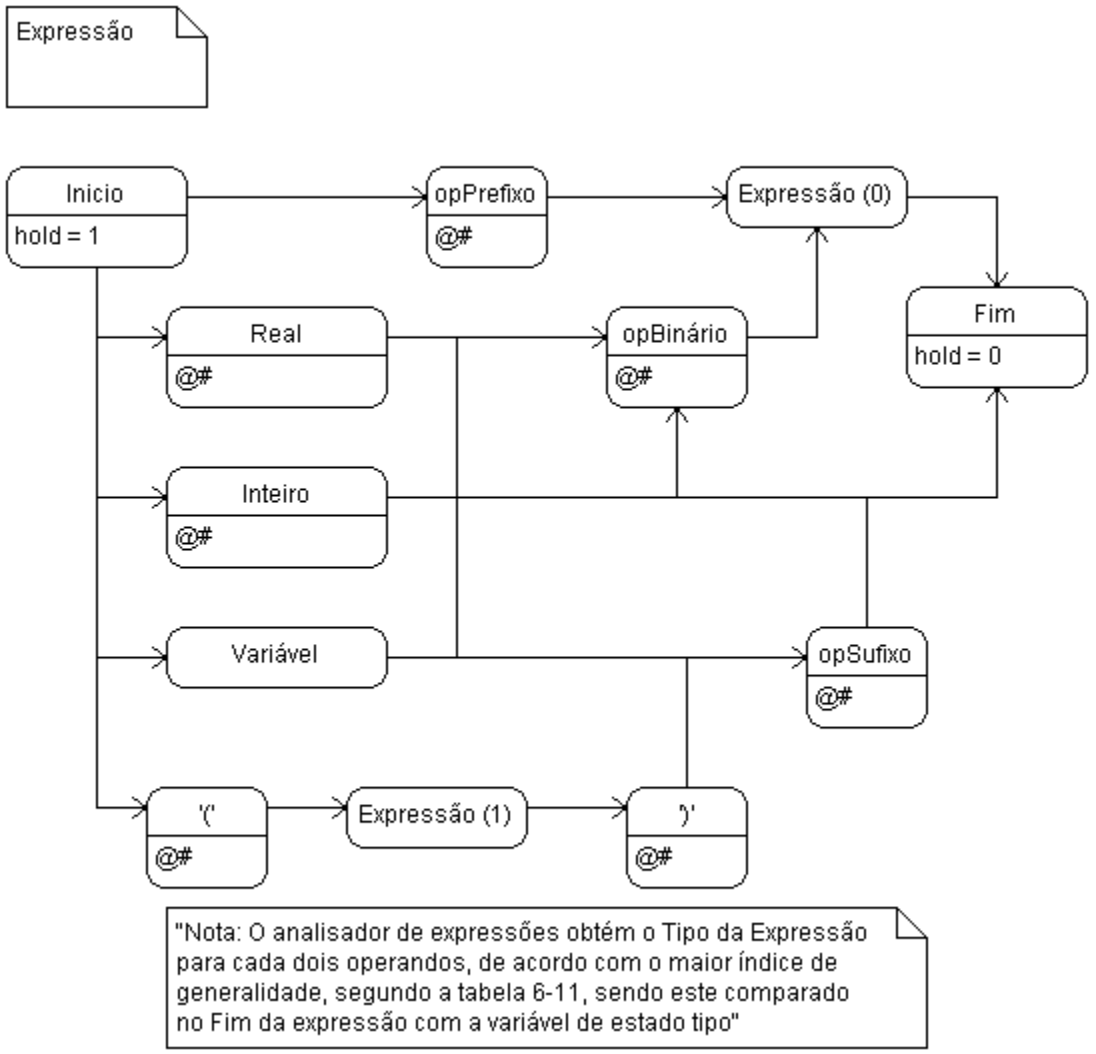


Fig. 6-10: Análise de expressões

Para navegar nos diagramas de sintaxe e efectuar a tradução é utilizado o seguinte motor de tradução, onde a identificação do diagrama é dada por **diag**, a posição na matriz de adjacência por *linha*, *coluna*, o lexema corrente por *token*.

Algoritmo 6-1:

sintaxe (**diag**, *token*)

```
início
coluna = 0, linha = 0
STACK = nova pilha endereçável
processaNó(diag, coluna, STACK)        "processa nó virtual Início"

ciclo
        coluna = buscaNó(diag, linha, 0, token) "Busca lexema, estado virtual ou
                                                 referência a diagrama"
        Se não encontra caminho gera erro de sintaxe
        Senão       Se (coluna é diagrama)          "chama-se recursivamente"
                processaNó(diag, coluna, STACK)
                STACK PUSH sintaxe (Diagrama dado por [linha, coluna], token)
        Senão
                processaNó(diag, coluna, STACK)

                Se (coluna = ordem de [diag])       "Encontrou nó virtual Fim"
                        retorna token               "Sai do diagrama corrente"
                Se (coluna não é nó Virtual)
                        token = gettoken()          "Pede próximo lexema ao scanner"
                STACK PUSH token
        linha = coluna
fim
```

Onde a função gettoken( ) pede ao analisador léxico o próximo lexema e a função processaNó(**diag**, *coluna*, STACK) efectua a tradução em cada nó *coluna* do diagrama **diag**, efectuando com a seguinte ordem as operações:

- Processa análise indicada nos comentários
- Processa variáveis de estado e arcos virtuais
- Processa regras de tradução

6.2.2.2.1 Tradutor de expressões

Este analisador surge da limitação da linguagem alvo, o C paralelo, de implementar operadores. Se a linguagem alvo permitir a sobrecarga de operadores com prioridade, como por exemplo o C++, as bibliotecas de cálculo matricial poderão ser redesenhadas de modo a tirar partido desta capacidade, sendo a tradução de expressões matriciais directa, de modo que será efectuada apenas por um diagrama de sintaxe, juntamente com as expressões escalares.

Como em C paralelo as operações são implementadas à custa de funções, que não obedecem a nenhuma prioridade consoante a operação a efectuar, será necessário decompor uma expressão num conjunto de chamadas a funções, que é precisamente o que este módulo faz.

O diagrama seguinte mostra o fluxo de dados durante o processo de tradução. O gerador de expressões, o conversor entre notação infixa e posfixa e o interpretador de diagramas de

sintaxe são sub-módulos do analisador sintáctico e semântico. A tradução de expressões matriciais processa-se do seguinte modo: O analisador léxico lê um token da Pilha de tokens, valida-o e coloca-o numa pilha. O analisador sintáctio e semântico lê o *token* da pilha endereçavel e efectua a tradução de acordo com os diagramas de sintaxe, e as funções de análise em comentário.

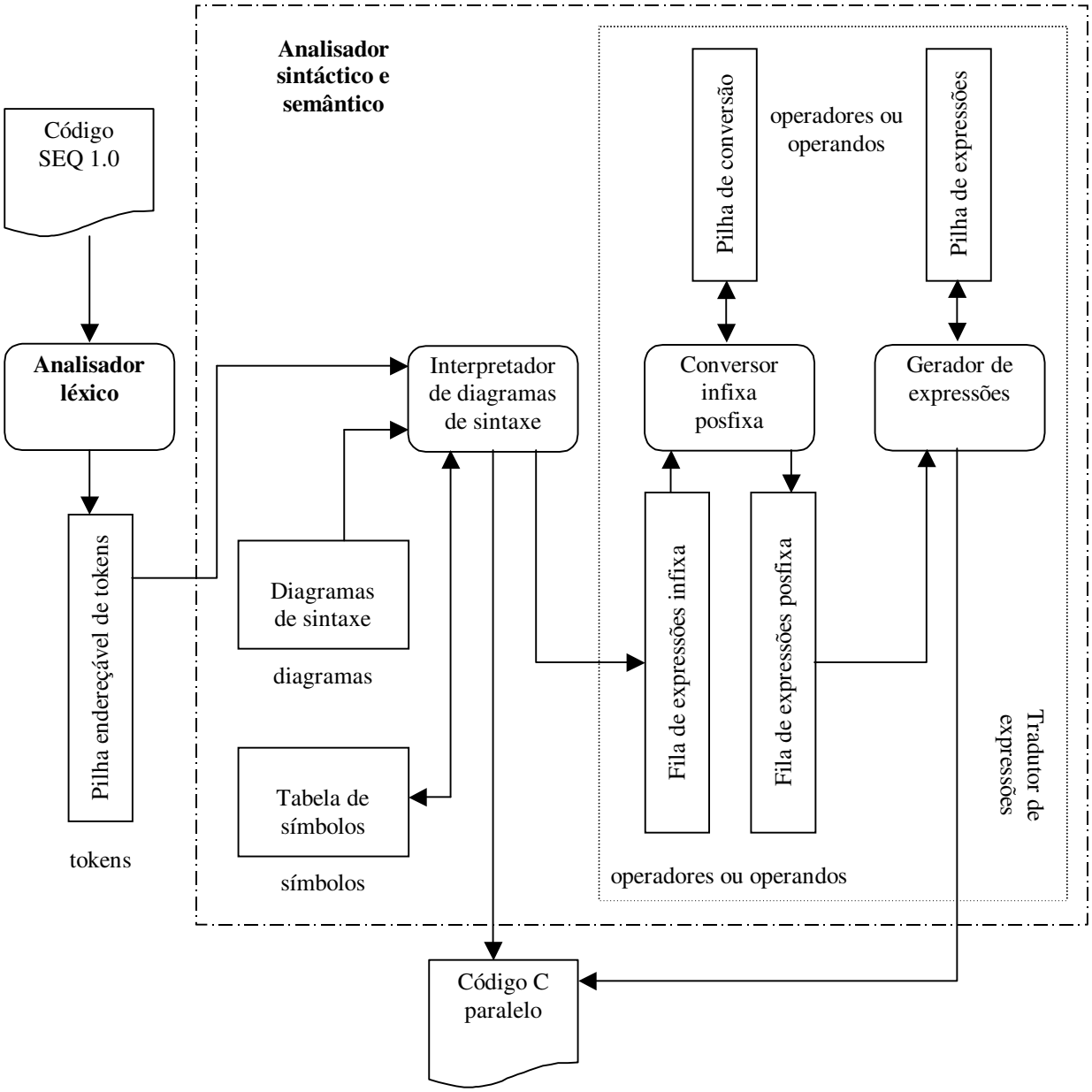


Fig. 6-11: Fluxo de dados no interior do tradutor

Mas, se este *token* for um operador ou operando e se estiver a processar uma expressão matricial, coloca-o na fila infixa, validando-a segundo o diagrama de sintaxe atrás apresentado. Se um operando se referir ao resultado de uma função, ou uma sub-área de uma matriz, é criado código para chamar a função, sendo o resultado atribuído a um identificador

auxiliar. Este código é escrito para o ficheiro de saída, antes da expressão traduzida. Na fila infixa, no lugar deste operando é colocado o identificador auxiliar, de modo que a expressão é reconstruída apenas com operadores, operandos escalares constantes e identificadores.

Seguidamente os operandos e operadores da fila infixa são reordenados em notação posfixa, de modo que a precedência de operadores da tabela 6-2 seja respeitada. Para tal é seguido um simples conjunto de regras, de modo que a expressão posfixa fique com a ordem de execução das operações sem recurso a parênteses. Este conjunto de regras pode ser expresso no formato da função recursiva:

algoritmo 6-2:

in2post ( )
início
- Cria pilha local vazia.
- Enquanto fila infixa não vazia:
  - Se encontra um operando este é copiado para a fila posfixa.
  - Senão, se encontra um operador este é colocado numa pilha de modo que quando chegar outro operador a sua precedência possa ser comparada com a do anterior:
    - Se a precedência do operador corrente for menor que a do operador no topo da pilha, este é transferido para fila posfixa e o operador corrente é colocado na pilha.
    - Se a precedência do operador corrente for maior ou igual é colocado no topo da pilha
  - Senão, se encontra um parênteses curvo aberto chama recursivamente in2post()
  - Senão, se encontra um parênteses fechado saí do ciclo

- Quando se chega ao fim da fila infixa, todos os operadores que se encontram na pilha são transferidos para a fila posfixa.

fim

No caso de operadores + e – que podem ser binários ou unários, estes são tratados como unários sempre que se sigam a um parênteses aberto, a um operador unário prefixo ou a um operador binário.

Os operandos e operadores são mantidos na estrutura:

```
typedef  struct {
        char id [IDLEN+1];
        tipoDados dtipo;
        int trans;
} op;
```

onde id representa o lexema, quer seja um operador ou um operando. O tipo de dados de um operando é indicado por dtipo e se for necessário transpor este operando, trans terá um valor não nulo. Por outro lado, se dtipo for nulo, a estrutura guarda um operador.
Finalmente o gerador de expressões pode transformar a sequência de operações na fila posfixa em chamadas às funções de cálculo correspondentes. Assim, sempre que uma operando é analisado, o seu tipo de dados, que pode ser inteiro, real ou matricial, é armazenado. Quando uma operação é efectuada, é verificado se, dado o operador, os tipos de ambos os operandos são suportados, de acordo com a tabela 6-1. O resultado da operação terá um tipo que será igual aos operadores se estes forem do mesmo tipo. No caso de operações entre operadores com tipos diferentes, o resultado será do tipo com maior índice de generalidade.

Tipo de dados	Índice de generalidade
matrix ou string	100
real	40
integer	20

tabela 6-3: Índice de generalidade para os tipos de dados numéricos

Este módulo usa uma pilha auxiliar de modo que os operandos são lidos da fila posfixa e colocados no topo daquela até ser encontrado um operador. Se o operador for a transposição, no atributo trans do operando anterior na pilha será colocado um valor não nulo, e o operador transposição removido da pilha.
Seguidamente é dimensionada uma variável temporária de acordo com a operação a efectuar. A esta é atribuído o resultado de uma função de cálculo correspondente, das já definidas no capítulo anterior, nos pontos 5.3.2.6 e 5.3.3.2, e de acordo com o especificado no ficheiro "function.dat". No fim de cada passo o identificador temporário é colocada no topo da pilha, de modo que possa ser usado para cálculo futuro.
Este processo prossegue até que se chegue ao fim da fila posfixa e está expresso em termos de pseudo-código no apêndice G.

Ainda de acordo com a tabela anterior, o índice de generalidade dos tipos de dados **string** e **matrix** é idêntico. De facto a representação interna de ambos os tipos de dados é igual. É assim possível atribuir a um identificador do tipo **matrix** uma **string**. O inverso já não é possível pois uma **string** têm de ser armazenada num vector e nada garante que a matriz a atribuir seja um vector.

6.2.3 Adicionar funções definidas pelo utilizador

Existem dois métodos de acrescentar funções ou sub-programas em C paralelo à biblioteca de base. Como já foi indicado e será novamente abordado no capítulo seguinte, um método consiste em defini-los e acrescentá-los ao ficheiro que suporta extensões à biblioteca de base C, ”mcalcext.h”, actualizando a tabela de funções “function.dat” com o nome da função e o tipo segundo a tabela 5-18. A principal utilidade deste primeiro método consiste em possibilitar a alteração da funcionalidade dos operadores matriciais SEQ, atribuindo-lhes outras funções.

O segundo método, já descrito neste capítulo, consiste em usar a directiva C { ... }para embeber código C paralelo, num programa SEQ. É assim possivel adicionar sub-programas de índole geral, em C paralelo.

O primeiro método, ao possibilitar a extensão das bibliotecas em C, permite que a compilação do código fonte seja mais rápida. Além disso consiste na única forma de alterar a funcionalidade dos operadores matriciais. Está no entanto limitado a funções cuja lista de parâmetros obedecem a tipos definidos e é mais complexo para o utilizador.

6.3 Resumo

Neste capítulo foi descrito o tradutor que suporta a linguagem SEQ 1.0. O facto de ser configurável, através da representação de diagramas de sintaxe, permite que futuras versões da linguagem sejam facilmente implementadas e ensaiadas. Sendo a linguagem SEQ 1.0 experimental e principalmente um *front-end* para as bibliotecas de cálculo matricial, e assim para todo o SPAM, é desejável um método de ensaiar evoluções desta linguagem facilmente.

O tradutor de expressões matriciais representa um módulo não configurável mas que poderá ser eliminado se a linguagem alvo for C ++ paralelo, e as bibliotecas redesenhadas de acordo. Neste caso apenas a análise indicada nos comentários dos nós terá de ser programada em C, recorrendo aos recursos indicados nos apêndices F, sendo todo o resto do tradutor configurado pelos diagramas de sintaxe.

7 Descrição do ambiente integrado de desenvolvimento de aplicações

O ambiente integrado de desenvolvimento de aplicações, referenciado a partir deste ponto como AIDA 1.0 ou simplesmente AIDA, consiste numa aplicação gráfica de alto nível, a partir da qual todas as ferramentas necessárias para criar, alterar, compilar e executar aplicações, bem como gerir bibliotecas, no contexto do SPAM, podem ser lançadas e executadas no anfitrião. Se bem que estas ferramentas, baseadas em conceitos descritos nos capítulos anteriores e neste mesmo capítulo, possam ser executadas independentemente, este ambiente facilita ao programador de aplicações o acesso a elas, bem como a todas as operações necessárias à manutenção de um projecto sob o ponto de vista do SPAM.

Neste capítulo são introduzidos o configurador e o *interface* com SIMULINK, ferramentas ainda não descritas. O configurador é o responsável por gerar não só o ficheiro de configuração a injectar no compilador ou nos compiladores alvo, mas também o ficheiro de gestão do projecto e o ficheiro de inicialização da aplicação. É assim apresentado um algoritmo, que busca o caminho óptimo no grafo da rede alvo, de modo a minimizar o tempo gasto em comunicações, e a partir destes gerar as tabelas de encaminhamento de mensagens já descritas no capítulo 4, e incluídas neste último ficheiro.

É também descrito a integração destas ferramentas, bem como do tradutor e das bibliotecas, descritos nos dois capítulos precedentes, de modo a obter um sistema de geração automática de código.

7.1 Configurador

Como já foi introduzido no capítulo 3, para que seja gerada automaticamente uma aplicação paralela, além do código sequencial traduzido para código C paralelo pelo tradutor descrito no capítulo anterior, a rede de processadores onde a aplicação será executada deve ser descrita. Para tal é necessário criar um ficheiro de texto, segundo a sintaxe apresentada em (3L Ltd.,

1991; 1995; 1998), onde também deve ser especificado quais as tarefas em que a aplicação é dividida, os recursos de memória que estas requerem, a afectação destas a processadores, e as vias de comunicação entre tarefas. Para simplificar a tarefa ao programador de aplicações, e minimizar erros nesta descrição, foi desenvolvida uma ferramenta de configuração para integrar no SPAM, que permite criar esse ficheiro de texto a partir de um diagrama de blocos, onde os nós representam os processadores e os arcos as vias de comunicação bidireccionais entre estes. Toda a outra informação relevante, acima mencionada, pode ser introduzida nas caixas de diálogos de cada nó.

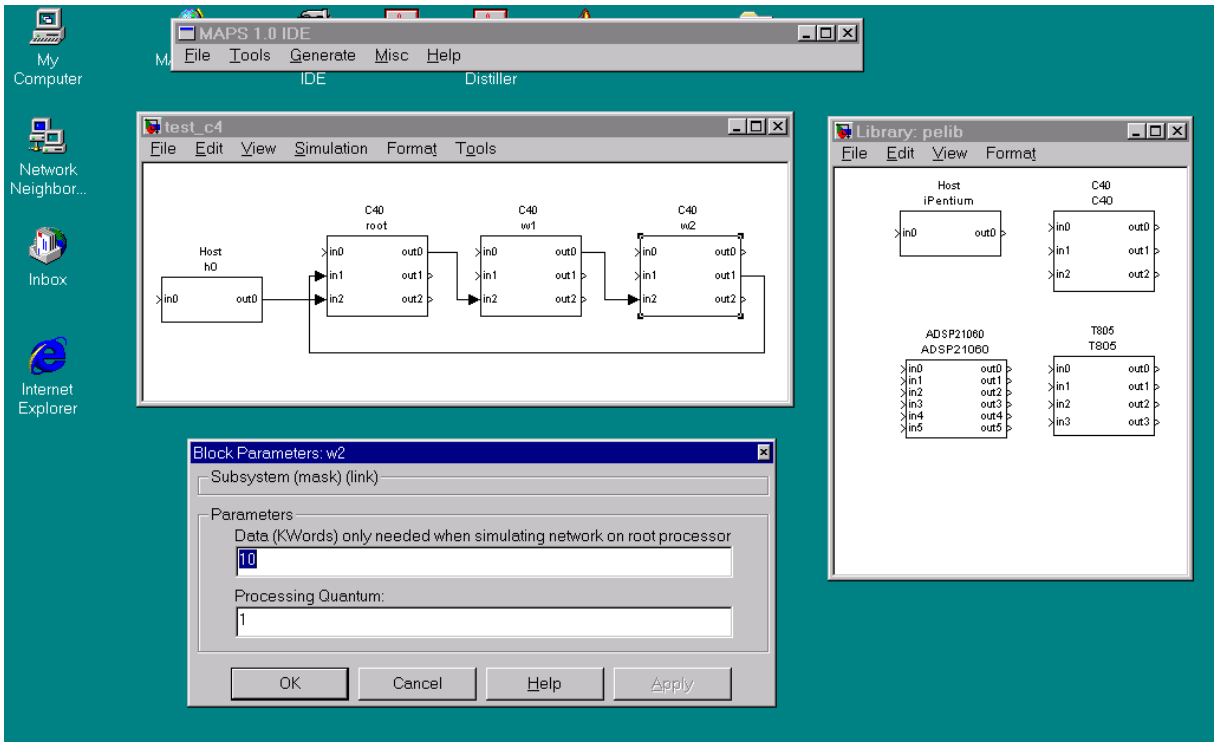


Fig. 7-1: Descrição da rede alvo em SIMULINK

Para descrever a rede alvo num diagrama de blocos, é utilizada uma aplicação externa ao SPAM, o ambiente SIMULINK da Mathworks. Como mostra a Fig. 7-1, a partir de uma biblioteca predefinida, designada por "pelib.mdl", onde são incluídos os blocos que representam os processadores suportados, o utilizador pode construir o modelo da rede alvo, indicando as características de cada processador na janela correspondente. Este ambiente gera como resultado um ficheiro de texto, que será traduzido pelo configurador para a sintaxe esperada pelos compiladores alvo utilizados. O diagrama da Fig. 7-2 descreve as várias etapas do configurador e o resultado de cada uma delas. O processo inicia-se injectando no configurador o ficheiro que descreve um modelo em SIMULINK, designado genericamente

por "*.mdl", e o ficheiro "comqt.dat", onde estão expressos os *quanta* de comunicação entre os processadores suportados. Este último ficheiro tem o seguinte formato:

np	Nº de processadores suportados	(escalar inteiro)
CQ	*Quanta* de comunicação	(matriz real triangular inferior *np* x *np*)

Para os processadores suportados, de acordo com as medições efectuadas no ponto 2.4.2, esse ficheiro será:

```
3
0.4638
1.1211 0.0603
-1     -1     0.0253
```

Isto é, tomando como índices para linhas e colunas da matriz **CQ**, a série {T805, C40, ADSP210060}, que define os processadores suportados, em $\mathbf{CQ}_{0,0}$, $\mathbf{CQ}_{1,1}$ e $\mathbf{CQ}_{2,2}$ estão definidos os *quanta* de comunicação entre processadores idênticos:

T805 - T805
C40 - C40
ADSP210060 - ADSP210060

enquanto em $\mathbf{CQ}_{1,0}$ está definido o *quantum* de comunicação entre um T805 e um C40. Nas restantes posições o valor -1 indica que as ligações entre os pares de processadores T805 - ADSP210060 e C40 - ADSP210060 não são suportados.

7.1.1 Nível de abstracção entre a ferramenta de desenho da rede e o configurador

Como se pode observar no diagrama, a primeira etapa, executada pelo módulo "modelint", consiste na tradução para um formato intermédio das características dos processadores e das ligações entre eles. O ficheiro resultante "pdef" guarda uma entrada para cada processador da rede, onde são discriminadas as suas características.

As primeiras duas características, que se aplicam também ao processador anfitrião, devem ser introduzidas nas linhas imediatamente acima do bloco que representa o processador, e são respectivamente:

i) Tipo de processador. Além dos processadores correntemente suportados, também é considerado o processador anfitrião, um INTEL 80386 compatível.

ii) Nome do processador.

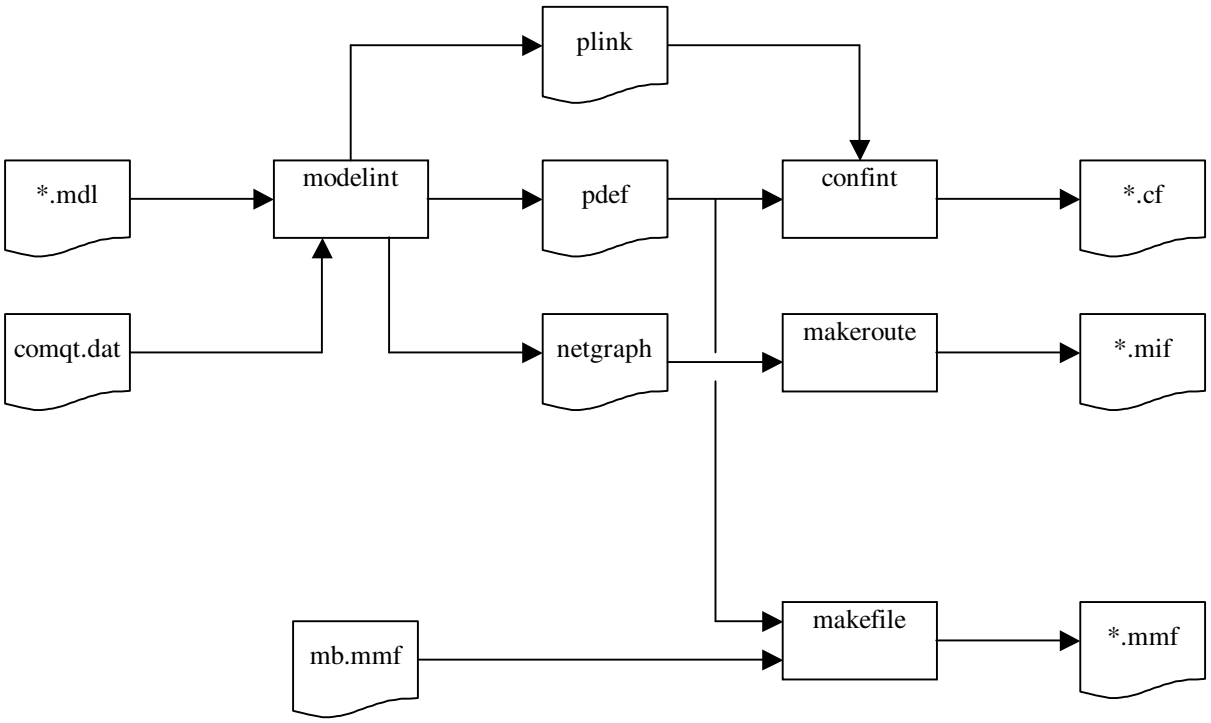


Fig. 7-2: Etapas envolvidas na acção do configurador

As restantes duas características, não aplicáveis para o processador anfitrião, devem ser indicadas na janela de entrada de dados de cada processador:

iii) Quantidade de palavras reservadas para o segmento de dados. No caso de Transputers uma palavra equivale a um *byte*, enquanto para os restantes processadores suportados, uma palavras é composta por 4 *bytes*. Este parâmetro é apenas requerido quando se pretende simular uma rede num só processador. Caso contrário toda a memória disponível, não usada para armazenar o código, é assignada ao segmento de dados.

iv) *Quantum* de processamento. Normalmente refere-se ao tempo necessário para calcular uma operação de vírgula flutuante. No entanto, como é usada uma relação entre o *quantum* de cada processador e o *quantum* do processador mais lento, para automaticamente equilibrar a distribuição de dados, este valor pode ser igual a 1 para o processador mais lento, e para os restantes processadores

indicar um factor pelo qual estes são mais rápidos. Este valor é usado pelos sub-programas de comunicações, chamados a partir da aplicação, que distribuem os dados pelos nós da rede, equilibrando o esforço computacional de acordo com as características de processamento de cada nó.

Quanto ao ficheiro "plink", este guarda todas as ligações bidireccionais entre processadores. Para cada ligação, existe uma entrada neste ficheiro que incluí dois pares:

i) Processador de origem e porto de comunicação do processador de origem.
ii) Processador de destino e porto de comunicação do processador de destino.

Assim, estes dois ficheiros implementam um nível de abstracção, onde toda a informação sobre a rede alvo é conhecida, independentemente da ferramenta usada para criar o diagrama de blocos. Desta forma o configurador não fica dependente do SIMULINK, podendo este ser substituído por outra aplicação, apenas tendo de ser alterado o módulo "modelint". Nesta etapa é ainda gerado o ficheiro "netgraph". Este ficheiro guarda todos os caminhos da rede na forma de um grafo, que será usado pelo módulo "makeroute" para encontrar os caminhos óptimos, em termos de largura de banda dos arcos, entre todos os nós, como será adiante descrito.

Ainda neste módulo, são efectuadas algumas validações sobre os ficheiros "pdef" e "plink" que garantem a coerência do modelo. São assim validados os processadores e tipos de redes homogéneas e heterogéneas suportadas. É garantido que o nome de um processador é único e que existem pelo menos o processador anfitrião e o processador no raiz modelo. É garantido que só existe uma ligação entre dois processadores. É também garantido, que se o processador raiz for um T805 ou um C40, estes devem estar ligados ao anfitrião; por outro lado é impedida uma ligação entre qualquer outro processador e o anfitrião. É ainda invalidado um modelo que contenha pelo menos um nó sem qualquer ligação a outro, estando portanto isolado sem qualquer meio de comunicação com a rede.

Além das validações anteriormente descritas, são feitas ainda algumas correcções. No caso de simulação de uma rede no processador raiz, o comprimento do segmento de dados não pode ser nulo. Assim o configurador atribuí-lhe um valor predefinido, avisando o utilizador. O *quantum* de processamento também não pode ser nulo. Se tal suceder, o valor por defeito é seleccionado e o utilizador é disso informado. Nesta última situação deve-se tomar especial cuidado, pois como já visto acima, isto pode não ser o pretendido.

7.1.2 Geração automática do ficheiro de configuração da rede

Continuando a analisar o diagrama encontra-se um módulo identificado por "confint". Este módulo é o responsável por gerar o ficheiro de configuração a injectar no compilador alvo. Vai pois usar os ficheiros intermédios "pdef" e "plink", criando um ficheiro de texto, designado genericamente por "*.cf" , e que consiste numa sequência ordenada de comandos onde é descrito:

i) Tipos de processadores utilizados
ii) ligações físicas entre estes
iii) Tarefas que compõem a aplicação e sua configuração.
iv) Atribuição das tarefas a processadores. Como já foi visto anteriormente, só no processador raiz são alocadas duas tarefas. Uma para permitir que a aplicação comunique com os recursos do processador anfitrião e outra, que será a única alocada nos restantes processadores, e que consiste na aplicação distribuída.
v) vias de comunicação bidireccionais entre as tarefas que compõem a aplicação.
vi) Portos de comunicação requeridos para estabelecer uma fila de processadores.

Na etapa iii) é empregue uma estratégia de alocação de segmentos de dados e código em bancos de memória tais, que minimizem os tempos de acesso a estes. Esta estratégia permite, como já indicado no capítulo 2, para partido das facilidades dos processadores utilizados.

7.1.3 Geração das tabelas de encaminhamento de mensagens

O módulo "makeroute" gera as tabelas de encaminhamento de mensagens, referidas no capítulo 4. Para minimizar o tempo perdido em comunicações devem ser encontrados os caminhos óptimos entre todos os nós. Para tal, a cada arco é associado um custo, o *quantum* de comunicação bidireccional entre os processadores que este liga. Assim, entre quaisquer dois nós, deve ser obtido o caminho cuja soma dos *quanta* associados aos arcos seja mínima. Isto é um problema clássico de busca de caminho óptimo, e um algoritmo simples para o resolver é o algoritmo de Ford, cuja formulação pode ser obtida em (Kaufmann, 1976). Se bem que possam existir algoritmos que apresentem a solução em menos tempo, como esta análise não é feita em tempo de execução mas sim de compilação, este factor não é muito relevante.

Segundo o algoritmo de Ford, para encontrar o caminho óptimo entre quaisquer dois nós de um grafo orientado, inicia-se atribuindo o valor $\lambda = 0$ ao nó de origem e $\lambda = \infty$ a todos os outros nós. Seguidamente aplicam-se o seguinte conjunto de regras para cada par de nós:

Seja λ_i o valor no nó X_i.

Seja $\nu(X_i, X_j)$ o custo associado ao arco entre o nó X_i e o nó X_j.

Procura-se um arco (X_i, X_j) tal que $\lambda_j - \lambda_i > \nu(X_i, X_j)$.

Calcula-se δ_j usando a seguinte expressão:

$$\delta_j = \lambda_i + \nu(X_i, X_j) \quad (7\text{-}1)$$

sendo este valor δ_j menor que o valor de λ_j.

Substitui-se então λ_j por δ_j.

Deve-se prosseguir desta forma até que nenhum arco permita diminuir os λ, o que significa que se obtiveram todos os valores mínimos dos custos dos caminhos com origem num dado nó e fim nos outros nós. Finalmente os nós incluídos nos caminhos mais curtos são obtidos viajando do nó destino para o nó origem, percorrendo os nós adjacentes com menor valor de λ.

Seguindo o mesmo procedimento tantas vezes como o número de nós, e alternando em cada uma das vezes o nó de origem, de modo que todos os nós do grafo sejam a origem dos caminhos, obtêm-se os caminhos mínimos entre quaisquer dois nós da rede.

Deste algoritmo pode ainda retirar-se um corolário: um caminho óptimo não pode passar mais que uma vez pelo mesmo nó.

Particularizando o algoritmo de Ford para uma rede com ligações bidireccionais, como é o caso das redes utilizadas no contexto do SPAM, está-se na presença de um grafo não orientado, onde os arcos não têm indicação de sentido. Assim, embora na Fig. 7-1, se verifique que nas conexões entre blocos existe a indicação de apenas um sentido, ambos os sentidos devem ser considerados, de modo que a matriz de adjacência ao grafo da rede é simétrica em relação à diagonal principal. Por outro lado, como neste modelo de comunicação não são necessárias comunicações com origem e destino no mesmo nó, a diagonal principal da matriz de adjacência é nula, o que é indicado pelo valor -1 no custo ou *quantum* de

comunicação correspondente. Além disso, como um nó isolado não faz sentido em termos de rede, a matriz de adjacência não tem linhas nem coluna nulas.

Seja esta matriz representada por **M**, seja *np* o número de nós na rede e *Ci* um índice para um vector de caminhos possíveis, o algoritmo a seguir apresentado, consiste numa implementação baseada no algoritmo de Ford, para este caso específico.

algoritmo 7-1:

```
caminhos (M, n, Co, custo)
início
desde c = 0 até np-1
        Se M (n, c) ≠ -1 e o nó c não pertence ao caminho Co
                Se Co < 0
                        criar novo caminho Ci nulo
                        adicionar ao caminho Ci o nó n com o custo M (n, c)
                Senão
                        criar novo caminho Ci igual ao caminho anterior Co
                adicionar ao caminho Ci o nó c com o custo dado por: custo + M (n, c)
                caminhos (m, c, Ci, custo + M (n,c))
                Ci = Ci + 1
fim
```

Como resultado da função recursiva acima descrita, todos os nós incluídos nos caminhos possíveis entre o nó de origem e os restantes, bem como o custo desses caminhos, encontram-se armazenados num vector. Finalmente para cada destino, onde o destino é sempre diferente da origem, deve-se escolher o caminho com o menor custo.

Assim, para calcular os caminhos óptimos entre todos os nós da rede, visto que a função anterior só determina os caminhos óptimos entre um nó e os restantes, esta deve ser executada tantas vezes quantas os nós existentes nesta rede, colocando de cada vez um dos nós na origem do caminho:

algoritmo 7-2:

```
desde i = 0 até i < np-1
        Ci = 0
        caminhos ( M, i, -1, 0)
```

No entanto o algoritmo 7-1 requer que todos os caminhos possíveis sejam armazenados na memória, para que finalmente se obtenham os óptimos. Pode-se modificar este algoritmo, de tal modo que o custo de cada caminho é computado em cada instância da função recursiva, e sempre que um caminho mais curto entre dois nós é encontrado, é registado substituindo o

anterior caminho óptimo. Assim é eliminada a etapa final de procura do caminho óptimo, através de todos os caminhos possíveis, requerida pelo algoritmo anterior. Desta forma também não existe necessidade de armazenar todos os caminhos possíveis, mas apenas os óptimos, o que resulta numa poupança drástica de memória. Pode-se assim derivar um novo algoritmo, mais eficiente em termos de tempo de execução e requisitos de memória. Mas antes disso, é ainda necessário incluir na matriz de adjacência informação sobre os portos de comunicação que constituem as saída e entradas em cada processador. Deste modo, a matriz de adjacência **M2**, quadrada com tantas linhas e colunas como o número de nós, é dada por:

$$\text{(7-2)} \quad \mathbf{M2} = \begin{bmatrix} \mathbf{M2}_{0,0} & \cdots & \mathbf{M2}_{0,np-1} \\ \vdots & \ddots & \vdots \\ \mathbf{M2}_{np-1,0} & \cdots & \mathbf{M2}_{np-1,np-1} \end{bmatrix}$$

onde cada elemento é constituído por duas peças de informação. A primeira, identificada como *cq*, representa um custo real ou o *quantum* de comunicação, associado ao arco entre o nó de origem dado pela linha *l* e o nó de destino dado pela coluna *c*. A segunda é um inteiro que indica, para cada arco, o porto de saída no nó X_l para o nó de X_c, e é referido como *lo* :

$$\text{(7-3)} \quad \mathbf{M2}_{l,c} = \{ cq,\ lo \}$$

É conveniente indicar, que como a matriz **M2** é simétrica, o porto de entrada num nó X_c, correspondente a um arco com origem em X_l, pode ser obtido através dos portos de saída, trocando o índices de linha com o de coluna e lendo o valor do porto de saída para esse elemento:

$$\text{(7-4)} \quad li = \mathbf{M2}_{c,l}\ \{lo\}$$

Pode-se agora expressar o algoritmo modificado.

algoritmo 7-3:

caminhos(**M2**, *n*, *custo*, *Ca*)
início
criar caminho local *Ct*
desde *c* = 0 até *np*-1
    Se **M2** (*n*, *c*) ≠ -1 e o nó *c* não pertence ao caminho anterior *Ca*

```
        Se o caminho anterior Ca for nulo
            limpar caminho Ct
            adicionar ao caminho local Ct o nó n com o custo -1
        Senão
            Ct = Ca
        adicionar ao caminho local Ct o nó c com o custo dado por: custo +
                                                                   M (n, c)
        se Ct tem menor custo que caminho óptimo correspondente substituí-o
                                                                   por Ct
        caminhos (M2, c, custo + M (nó,c), Ct)
eliminar caminho local Ct
fim
```

E para calcular todos os caminhos óptimos a partir de todos os nós, a função acima deve ser chamada, uma vez para cada nó:

algoritmo 7-4:

desde $i = 0$ até $i < np\text{-}1$

caminhos (**M2**, *i*, 0, NULL)

Como está expresso na função recursiva acima, sempre que um caminho entre dois nós tem um custo inferior ao caminho correspondente previamente registado, este é substituído pelo caminho local *Ct*, que é local à função e tem um tempo de vida limitado, evitando-se assim gastos de memória desnecessários. Assim os requisitos de memória fixos consistem apenas no registo dos caminhos óptimos. Este registo é feito numa estrutura de dados que consiste numa matriz **CO** (*x*), para cada nó de origem *x*, com dimensão *np* x *np*. Os elementos de cada linha indicam os nós de um caminho desde a origem *x* até ao nó destino, que é dado pelo índice de linha.

$$\text{(7-5)} \quad \mathbf{CO}(x) = \begin{bmatrix} \mathbf{CO}(x)_{0,0} & \cdots & \mathbf{CO}(x)_{0,np\text{-}1} \\ \vdots & \ddots & \vdots \\ \mathbf{CO}(x)_{x,0} = -1 & \cdots & \mathbf{CO}(x)_{x,np\text{-}1} = -1 \\ \vdots & & \vdots \\ \mathbf{CO}(x)_{np\text{-}1,0} & \cdots & \mathbf{CO}(x)_{np\text{-}1,np\text{-}1} \end{bmatrix}$$

Cada elemento desta matriz é também composto por duas peças de informação: o nó pertencente ao caminho, *nc*, e o porto de saída deste nó para o nó seguinte no mesmo caminho, *lo*. O nó seguinte é indicado pelo campo *nc* do elemento seguinte na mesma linha.

(7-6) $\mathbf{CO}(x)_{l,c} = \{nc, lo\}$

Nesta representação o último nó de um caminho não tem saída, e portanto o porto de saída deste é igual a -1. Por outro lado o nó inicial de cada caminho é dado por *x*. Finalmente, se um caminho tiver menos nós que *np*, os elementos seguintes na linha tem nos campos *nc* e *lo* o valor -1.

Durante a execução do algoritmo 7-3, é ainda necessário guardar o valor dos custos de cada candidato a caminho óptimo, num vector **CCO** (*x*), com tantas linhas como o número de processadores, onde cada elemento é um número real.

(7-7) $$\mathbf{CCO}(x) = \begin{bmatrix} \mathbf{CCO}(x)_{0,0} \\ \vdots \\ \mathbf{CCO}(x)_{x,0} = -1 \\ \vdots \\ \mathbf{CCO}(x)_{np,0} \end{bmatrix}$$

Deve ainda ser apontado que, como um caminho óptimo não pode passar mais do que uma vez pelo mesmo nó, o número de nós num caminho nunca pode ultrapassar o número de nós na rede, *np*. Por outro lado, como o destino não pode ser o mesmo que a origem, as linhas *x* de **CO** (*x*) e **CCO** (*x*) são nulas, tendo todos os elementos dessas linhas o valor -1 nos campos *nc* e *lo*.

Após serem determinadas as matrizes **CO** (0 ... *np*-1), podem ser obtidas as tabelas de encaminhamento de mensagens. A tabela de encaminhamento particular de um determinado nó *x* é facilmente obtida de **CO** (*x*), pois consiste dos campos *lo* da primeira coluna dessa matriz:

(7-8) $\mathbf{TEP}\ (x)_{\ 0\ \ldots\ np\text{-}1,\ 0} = \mathbf{CO}\ (x)_{\ 0\ \ldots\ np\text{-}1,\ 0}\ \{lo\}$

A geração da tabela de encaminhamento geral é um pouco mais complexa. Assim para todos os elementos das matrizes **CO** (0 ... *np*-1), cujo campo *lo* for diferente de -1, o elemento da matriz **TEG** (índice dado pelo campo *nc* do elemento de **CO**), linha dada pelo índice *x* da matriz **CO** e coluna dada pelo campo *lo* já referido, toma o valor 1. Este conjunto de regras está expresso no algoritmo seguinte:

algoritmo 7-5:

desde $p = 0$ até $np-1$
    desde $l = 0$ até $np-1$
        desde $c = 0$ até $np-1$
            Se $\mathbf{CO}\ (p)_{l,c}\ \{lo\} \neq -1$
                $\mathbf{TEG}\ (\mathbf{CO}\ (p)_{l,c}\ \{nc\})_{p,\mathbf{CO}(p)_{l,c}\ \{lo\}} = 1$

Finalmente pode ser escrito o ficheiro de inicialização da aplicação, com a seguinte organização:

np
***Quantum* de processamento do nó** (0)
(...)
***Quantum* de processamento do nó** (*np*-1)
TEP (0)
TEG (0)
(...)
TEP (*np*-1)
TEG (*np*-1)

7.1.4 Gestão automática de projecto e actualização dos processadores suportados

Finalmente falta descrever o módulo "makefile". Este módulo vai gerar um ficheiro de gestão de projecto, "*.mmf", compatível com ferramentas desta natureza, como o Microsoft NMAKE. Assim toda a gestão de dependências entre módulos é efectuada automaticamente, deixando o programador de aplicações liberto desta tarefa.

Como ficheiro de entrada, têm-se mais uma vez o ficheiro que guarda as definições dos processadores da rede. Deste modo podem ser geradas regras de gestão, para compilar e ligar o código paralelo gerado pelo tradutor, pelos compiladores adequados. Estas regras são organizadas de acordo com o modelo de programação apresentado na Fig. 3-1, pela seguinte ordem:

i) Compilação de todos os módulos necessários para os tipos de processadores presentes no modelo da rede.

ii) Compilação da tarefa de controlo e das instâncias da aplicação, referenciadas na Fig. 3-3 como Prg(*n*), para os processadores adequados. Todas estas tarefas incluem o ambiente de comunicação e o código da aplicação em si.

iii) Gestão de dependências. Ligação do código objecto gerado no passo anterior com os módulos necessários e adequados ao tipo de processador.

iv) Criação da aplicação executável para a rede alvo, montando as tarefas geradas no passo anterior, de acordo com o ficheiro de configuração "*.cf"

Os compiladores adequados aos processadores devem estar definidos num outro ficheiro de entrada deste módulo, designado por "mb.mmf". Neste ficheiro de texto, o utilizador deve indicar os caminhos para os compiladores adequados, mais os parâmetros de linha de comandos que achar necessário para optimizar a geração de código. Para os processadores suportados estas definições já estão incluídas, de modo que modificações neste ficheiro só serão necessárias se se pretender adicionar um novo tipo de processador. Neste caso, deve ser adicionado também um novo módulo à biblioteca "pelib.mdl". Tal é efectuado facilmente copiando um bloco já existente nessa biblioteca, modificando-lhe o tipo e o nome, respectivamente nas duas linhas de texto no topo do bloco, e finalmente na janela de entrada de dados do bloco, devem ser especificados a quantidade de palavras reservadas para o segmento de dados e o *quantum* de processamento. Deve-se ainda indicar no ficheiro "comqt.dat", o *quantum* de comunicação entre estes novos processadores, bem como os *quanta* de comunicação entre os processadores já definidos e esta nova definição de processador. Finalmente deve ser incrementado o primeiro elemento deste ficheiro, para indicar que o número de processadores suportados aumentou.

No entanto convém apontar que processadores com alterações drásticas à arquitectura prevista pelo SPAM 1.0, podem ser impossíveis de adicionar com foi indicado acima. Não é pois possível garantir que qualquer processador poderá ser adicionado ao sistema sem revisão do código de alguns módulos do SPAM.

7.2 Gerador de *interface* com SIMULINK

Uma aplicação gerada pelo SPAM, pode comunicar com o anfitrião recorrendo a duas instruções especiais, incluídas na biblioteca de entrada-saída "mio", já descrita no ponto 5.2:

putvector (*c*, *n*, data)

getvector (*c*, *n*, data)

A primeira função permite enviar um vector com *n* inteiros ou números de vírgula flutuante, apontado pelo ponteiro data, para um canal de *c* de comunicação com o computador anfitrião, que poderá ser um ficheiro. A segunda permite fazer o inverso, isto é receber um vector com *n* elementos do computador anfitrião e guardá-lo na posição de memória indicada por data. Estas funções não são mais que mascaras de pedidos ao servidor de entrada-saída, HOST E/S alocado no processador raiz, a partir das instâncias Prg (*x*).

Para estabelecer esta comunicação, deve existir um controlador de dispositivo no computador anfitrião que permita comunicar com o processador raiz vectores de dados, precedidos por um cabeçalho que indique o tamanho destes vectores. Além disso, este controlador deve ser desenhado para um dispositivo de caracter, e não de bloco. Assim o controlador de dispositivo deve operar em modo binário, de modo que caracteres de controle sejam ignorados (Hogan, 1988).

Tal controlador está disponível para o T805. Assim, nesta versão do SPAM, foi desenvolvido um *interface*, apenas para redes que possuem um T805 como processador raiz. No entanto, desde que se desenvolva um controlador de dispositivo com as características básicas acima descritas, para qualquer tipo de processador, a exposição seguinte também será aplicável.

Após estabelecida a comunicação com o computador anfitrião, é necessário criar ainda uma via de comunicação com o espaço de trabalho, ou *workspace* no original, do MATLAB, de modo que este possa comunicar com a aplicação paralela. Esta via ou *gateway*, é implementada recorrendo a um ficheiro MEX, isto é uma biblioteca ligada dinamicamente, ou *dynamic linked library*, compatível com o sistema operativo que administra o computador anfitrião de modo que o MATLAB possa interagir com o controlador de dispositivo (The Math Works Inc., 1997c), podendo ser usado para monitorizar a aplicação. A versão corrente do SPAM 1.0 é executada apenas sobre o sistema operativo WINDOWS.

Esta DLL, denominada "T8int.dll", disponibiliza para o MATLAB as operações de abertura e fecho do dispositivo, e escrita e leitura neste. Assim, ao nível do MATLAB, o acesso a este dispositivo é transparente e pode ser visto como o acesso a um ficheiro. São pois adicionadas as seguintes instruções ao MATLAB:

```
[errno, desc] = opent8
errno = closet8 (desc)
errno = writet8 (desc , vector, n)
[errno, vector] = readt8 (desc, n);
```

A primeira abre um canal de comunicação com o processador raiz e retorna o descritor de ficheiro desc, ao passo que a segunda fecha o canal especificado pelo mesmo descritor. A terceira permite escrever um vector de dados com *n* elementos para o dispositivo cujo descritor é desc. Finalmente a quarta permite ler um vector com *n* elementos a partir do dispositivo indicado por desc. Se ocorrer um erro em qualquer destas funções, o código de erro é retornado na variável errno.

Devem ainda ser feitas algumas conversões de acordo com o controlador de dispositivo. No caso do T805, o cabeçalho enviado antes do vector é um inteiro de 16 *bits*, que indica o comprimento do vector em *bytes*, e que tem por limite 32768 (Transtech, 1991c). Por outro lado, a representação interna de um número de vírgula flutuante no MATLAB ocupa 8 *bytes*. Deste modo as funções de leitura e escrita devem converter esta representação para 4 *bytes*, que é a suportada pelas aplicações geradas pelo SPAM 1.0. Deve também ser calculado o cabeçalho de 16 *bits* de acordo com o número de elementos no vector, dado por *n*, multiplicado por 4. Assim o limite de elementos que podem ser escritos ou lidos de cada vez é dado por 32768 / 4 = 8192 elementos.

Posto isto, para se criar um *interface* entre a aplicação paralela e o MATLAB, é necessário ler o código sequencial, dado por um ficheiro "*.seq", procurar instruções de comunicação putv e getv, que irão gerar na aplicação paralela as instruções putvector e getvector já apresentadas, e criar um ficheiro "*.m", ou um *script* em MATLAB, que tenha uma sequência de funções readt8 e writet8 de acordo com o código SEQ.

Já o interface com o SIMULINK tem de obedecer a um conjunto de regras mais restrito. Para tal é necessário usar uma função especial do MATLAB denominada S-function, função esta que irá descrever a acção de um dado bloco. A estrutura desta função é semelhante a uma possível estrutura de uma aplicação gerada pelo SPAM, já apresentada na Fig. 3-6, visto que tem uma secção que só é executada uma vez, na inicialização da simulação, e outra que é executada em cada iteração da simulação (The Math Works Inc., 1997b).

Assim, na secção iterativa da S-function, gerada automaticamente pelo módulo "matint", devem existir apenas duas instruções. Primeiro uma instrução de escrita para a aplicação paralela, writet8, e depois uma instrução de leitura, readt8. Quanto à aplicação paralela, esta deve ter no início da secção iterativa uma instrução de aquisição de dados, getvector. Deve então processá-los e devolver os resultados com uma instrução putvector. Isto implica que no código sequencial deve existir apenas uma instrução getv no início e putv no fim, o que é da responsabilidade do programador de aplicações.

Deste modo, um bloco em SIMULINK pode ser usado apenas como uma porta para aceder a uma aplicação paralela, a qual implementa a função de transferência atribuída a esse bloco.

7.3 Integração das ferramentas

Para facilitar o desenvolvimento de uma aplicação paralela, todas as ferramentas anteriormente descritas podem ser chamadas a partir de uma ferramenta de topo. Esta consiste num menu, o qual permite que o programador de aplicações interactue com o AIDA e através deste com o SPAM.

Este *interface* deve assim permitir que se possam abrir e criar ou modificar projectos, que se possa invocar um editor de texto, para que o código sequencial possa ser introduzido, e que se possa aceder às ferramentas descritas nos dois pontos anteriores.

Deve também permitir que se possa interactuar com o gestor de projectos, de modo a efectuar a manutenção deste, o que incluí gerar a aplicação paralela invocando o tradutor e o configurador, ou apenas configurar uma aplicação previamente gerada, para uma rede alvo diferente, sem que seja necessário gerar e recompilar todos os módulos. Todo este processo é monitorizado em janelas, as quais reportam o estado da operação correspondente, como ilustra a Fig. 7-3. Nestas janelas é indicado passo a passo as acções tomadas pelo configurador e pelo tradutor, bem como qualquer anomalia ou erro encontrado. Na janela correspondente ao tradutor podem ser observados os erro de sintaxe respeitantes à linguagem sequencial SEQ 1.0, introduzida no capítulo anterior.

Pode finalmente ser observado o processo de geração da aplicação através da execução do arquivo "*.mmf" pelo NMAKE ou ferramenta compatível. Nesta última fase, embora pouco provável, podem ainda surgir algumas mensagens de erro, as quais tem a ver com o processos de geração automática de código e não com erros humanos.

Além das opções de gestão de projecto acima referidas, existem ainda opções para limpar qualquer traço do processo de geração automática, eliminando qualquer ficheiro temporário requerido por esta operação, bem como o executável da própria aplicação, se tal for requerido.

Após a aplicação gerada, esta pode ser executada também a partir do AIDA. O ambiente reconhece qual o processador raiz da rede alvo e selecciona o método de carregamento e execução apropriado para esta rede. Também é possível passar argumentos de entrada para a aplicação. Tal deve ser feito recorrendo a um campo apropriado numa janela de configuração como pode ser consultado no apêndice A. Este apêndice deve ser consultado se se pretender compreender melhor o desempenho e configuração do AIDA, bem como pormenores técnicos sobre o SPAM 1.0.

A partir do AIDA, é ainda possível controlar como é gerada a aplicação, em termos de rede alvo. Normalmente esta é gerada para uma rede descrita num modelo pelo utilizador. No entanto, o utilizador poderá pretender simular no processador raiz uma rede para a qual não tem suporte de *hardware*. Tal é possível seleccionando uma opção para tal efeito e apenas pedindo ao AIDA para invocar o configurador, de modo que seja reconfigurada a aplicação.

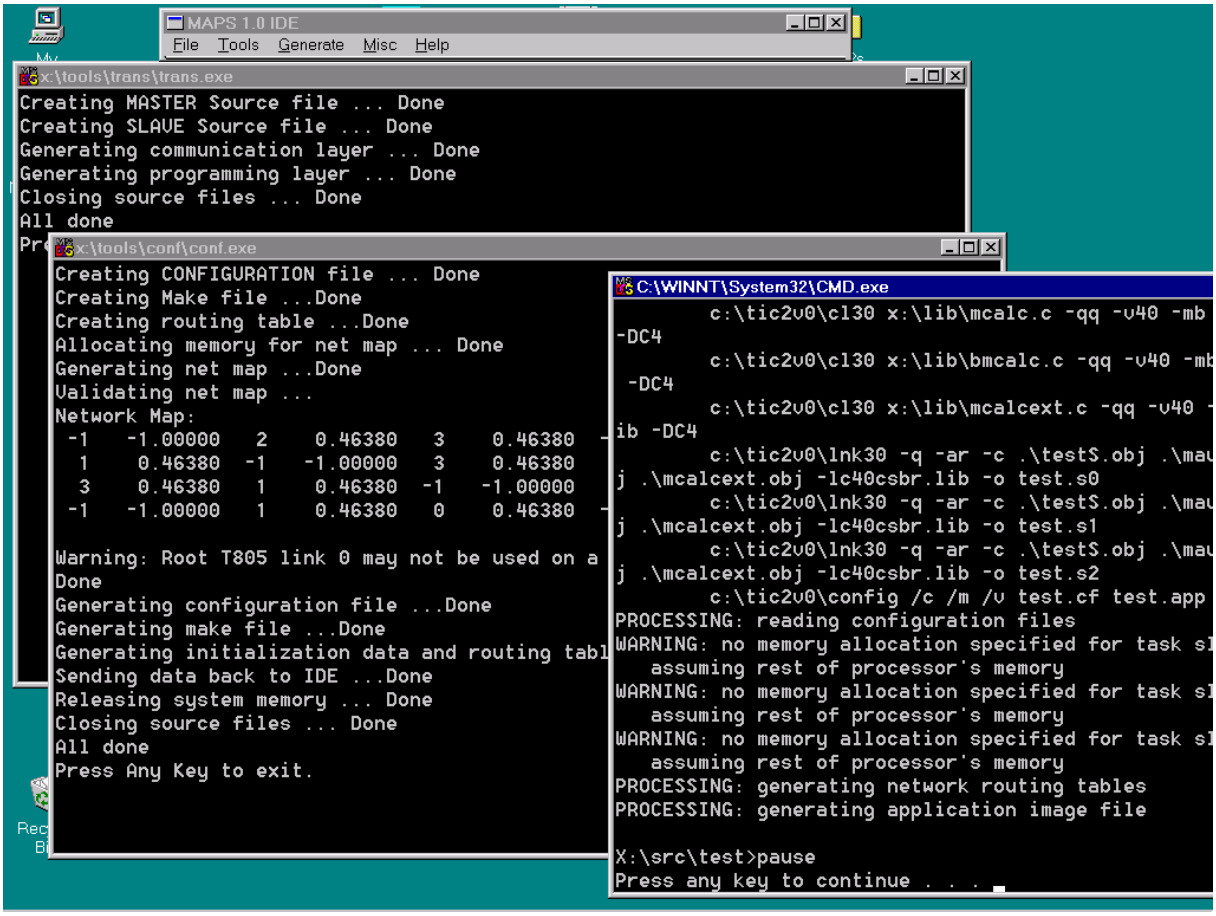


Fig. 7-3: Resultado do processo de geração automático de uma aplicação pelo SPAM, para o modelo de rede da Fig. 7-1.

Por outro lado, é comum que o processador raiz de uma rede homogénea, tenha algumas características diferentes dos restantes, como por exemplo bancos de memória local com maior capacidade. Se o utilizador pretender ensaiar o comportamento da aplicação numa rede verdadeiramente homogénea, deve existir um meio de executar a aplicação sem que esta recorra ao processador raiz. Assim a custo de ignorar um dos processadores, limitando o mapeamento da aplicação a menos um processador, tal também é possível seleccionando uma opção e apenas dando ordem para que seja reconfigurada a aplicação. Assim, o ficheiro de configuração "*.cf", gerado pelo configurador, inibe a colocação da primeira instância da

aplicação - Prg(0) - no processador raiz, deslocando-a para o processador seguinte na rede e alocando no primeiro processador apenas o processo de controlo ou *host interface*, para assegurar a comunicação com o computador anfitrião.

7.4 Adição de funções definidas pelo utilizador à biblioteca de cálculo

As bibliotecas de cálculo que acompanham o SPAM, descritas no capítulo 5, abrangem apenas a álgebra linear básica. No entanto um utilizador pode adicionar uma função de cálculo paralela, definida por si, respeitando as regras já introduzidas nos dois capítulos anteriores.

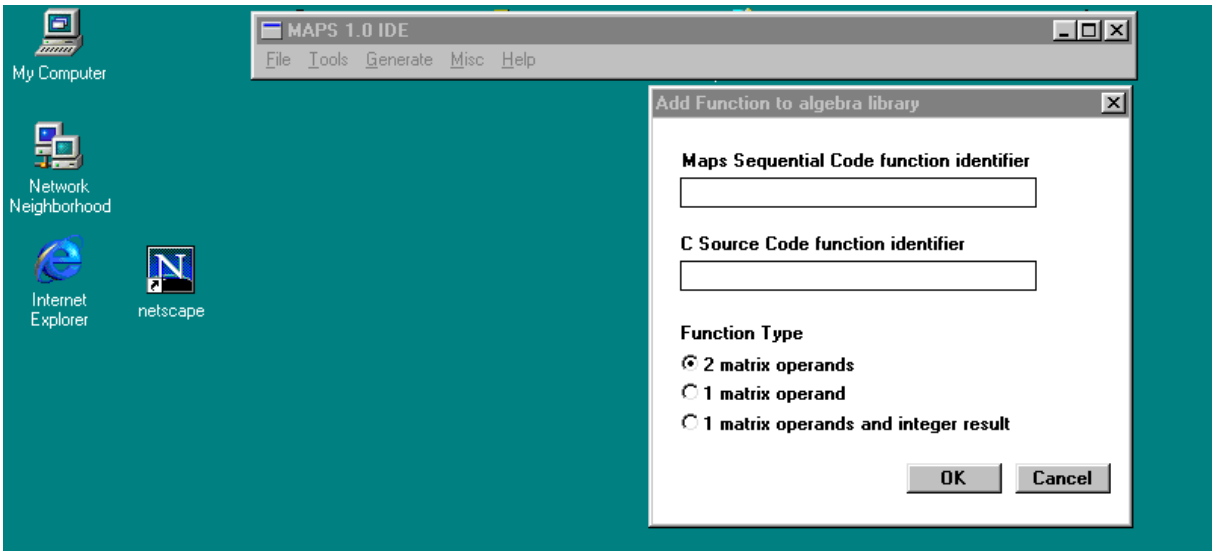


Fig. 7-4: Adição de funções definidas pelo utilizador à lista de funções reconhecidas pelo SPAM

Para que essa função passe a fazer parte integrante do SPAM, e que o tradutor a reconheça pode ser utilizada a caixa de diálogo acima exibida. Nesta caixa, o utilizador deve indicar qual o nome pelo qual a função é reconhecida pelo tradutor, no âmbito da linguagem sequencial SEQ 1.0. Deve também ser indicado o nome com o qual a função paralela foi identificada na biblioteca de cálculo paralelo. Finalmente deve ser indicada qual a classe da função em termos de operandos de entrada e saída:

dois operandos matriciais de entrada e um operando matricial de saída
um operando matricial de entrada e um operando matricial de saída
um operando matricial de entrada e um operando inteiro de saída

7.5 Resumo

Neste capítulo foram descritas as restantes ferramentas que compõem o SPAM 1.0: o configurador visual e o gerador de *interface* com SIMULINK. Foram descritos pormenorizadamente os vários módulos que compõem o configurador visual, de modo a demonstrar a importância desta ferramenta na gestão automática de um projecto, e até que ponto simplifica a tarefa do programador de aplicações. Foram identificados os passos que são executados automaticamente na configuração da aplicação para a rede alvo, como a busca de caminhos óptimos para uma qualquer rede alvo, e o controlo que o utilizador tem sobre a geração automática de código, em termos de simulação da rede alvo.

Foram também indicados pormenores a ter em atenção, para se escrever uma aplicação SEQ que irá comunicar com o ambiente SIMULINK, e em que casos este *interface* pode ser gerado automaticamente.

Foi feita uma breve descrição do procedimento para que uma função definida pelo utilizador, e adicionada à biblioteca de cálculo, possa ser reconhecida pelo SPAM.

Finalmente foi dada uma visão geral da integração de todas as ferramentas no AIDA, a ferramenta de topo que permite facilmente aceder a todas as facilidades do SPAM, e uma breve descrição da sua operação. No capítulo seguinte serão apresentados dois casos de estudo que permitiram descrever mais aprofundadamente as facilidades integradas neste ambiente de desenvolvimento integrado, bem como medir o desempenho das aplicações paralelas automaticamente geradas.

8 Casos de Estudo

Neste capítulo serão apresentados dois casos de estudo que ilustram a operacionalidade do ambiente de desenvolvimento de aplicações, bem como o desempenho das aplicações paralelas automaticamente geradas, utilizando as medidas de desempenho de algoritmos paralelos já formalizadas no ponto **5.4.1**.

O primeiro caso de estudo, consiste numa implementação de um algoritmo de controlo predictivo generalizado adaptativo. Este algoritmo é composto por dois estágios distintos: um estimador recursivo e o predictor propriamente dito. Visto que o primeiro estágio opera normalmente sobre volumes de dados muito inferiores aos volumes operados no segundo estágio, o desempenho deste algoritmo dependerá fortemente deste último.

No segundo caso de estudo, será abordada a implementação de um algoritmo de treino de redes neuronais, mais precisamente de um perceptrão multi-camada. Ambos os algoritmos requerem operações matriciais não implementadas na biblioteca de base, de modo que servem também para ilustrar como o utilizador poderá expandir o SPAM, de modo que este se adapte às necessidades.

8.1 AGPC

O algoritmo de controlo predictivo generalizado, *Generalizad Predictive Control* – GPC – no original, foi introduzido na década de 80 por (Clarke et al, 1987a; 1987b). Subsequentemente tem provado ser superior a outros algoritmos de controlo auto-regulados, tais como aqueles baseados em posicionamento de pólos ou variância mínima generalizada ou *Generalized Minimum Variance* - GMV (Clarke e Gawthrop, 1975). Se bem que o algoritmo GPC possa ser visto como um sucessor do GMV ou este um caso especial do GPC, este último conserva a sua robustez mesmo quando o tempo de atraso ou a ordem do modelo do sistema não são conhecidos (Aström e Wittenmark, 1989). Mas para que tal robustez seja atingida, existe um

custo substancial em termos de esforço computacional, sugerindo a sua implementação num ambiente de processamento paralelo.

Para além do algoritmo básico, (Clarke et al, 1987b) reformularam-no para se enquadrar numa estratégia de controlo adaptativo, de seguimento de modelo de referência.

Embora o algoritmo básico seja baseado num modelo de processo descrito por função de transferência, (Ordys e Clarke, 1993) propuseram uma formulação do algoritmo GPC em espaço de estados, estendida por (Ordys e Pike, 1998).

Tal como o GPC, o MUSMAR ou *Multistep Multivariable Adaptive Regulator* (Menga e Mosca, 1980) é outro algoritmo baseado também na perdição da saída do processo ao longo de um tempo futuro alargado e selecção de um sinal de controlo que tenta reduzir o erro futuro entre a referência e a saída do processo, mas formulado em espaço de estados. A principal diferença consiste no método de minimização da função de custo, no sentido em que no GPC os valores do sinal de controlo futuro não são restritos e no MUSMAR dependem de uma realimentação de estado. Tal como o GPC, este algoritmo também é robusto face à variação ou incerteza no valor do atraso do processo (Greco et al., 1984).

Existem no entanto formulações do GPC em que a minimização da lei de controlo está sujeita a restrições físicas, como limites de funcionamento de actuadores, tanto em termos de gama de operação como de velocidade de resposta. Também podem ser impostas restrições às variáveis do processo, por exemplo por razões de segurança ou de gestão de produção. A lei de controlo é assim tipicamente sujeita a restrições na amplitude e na taxa de variação do sinal de controlo e na amplitude do sinal de saída y. Um tratamento destas extensões pode ser consultado em (Maciejowski, 2001), e uma aplicação ao controlo de estufas de produção agrícola em (Cunha, 2002).

Outra diferença em relação ao algoritmo GPC reside na formulação original do MUSMAR ser multivariável, o que permite controlar processos MIMO, ao passo que a formulação original do GPC, baseada em função de transferência apenas permite o controlo de sistemas SISO.

Os algoritmos de controlo predictivo abordados até agora e implementados são algoritmos discretos no tempo. Existe no entanto uma formulação do MUSMAR em tempo contínuo o CTMUSMAR que pode ser consultado em (Costa, 1996).

Uma formulação do GPC em tempo contínuo, o CGPC foi proposta por (Demircioglu e Gawthorp, 1991). Segundo os autores "a formulação em tempo contínuo permite expor fundamentos do problema de controlo obscurecidos pela amostragem como é o caso particular de sistemas não lineares". A semelhança dos outros algoritmos de controlo já referidos,

também foi proposta versão do CGPC formulada em espaço de estados (Gawthorp e Siller-Alcada 1996).

Como o algoritmo GPC adaptativo – AGPC - necessita de conhecer a função de transferência do sistema, um primeiro estágio no algoritmo terá de identificar a função de transferência do sistema a controlar.

Existe uma grande variedade de estimadores de parâmetros de processos disponíveis, sendo possivelmente dos mais utilizados o estimador de mínimos quadrados recursivo ou *Recursive Least Squares* – RLS (Plackett, 1950). As formulações mais básicas deste algoritmo apresentam um desvio na estimação dos parâmetros se existir ruído correlacionado com os sinais medidos. Para resolver este problema pode ser utilizado o método das variáveis instrumentais (Söderström e Stoica, 1989), o qual introduz variáveis auxiliares correlacionadas com as variáveis de regressão e não correlacionadas com o ruído. Outra solução para este problema consiste no método de Koopmans-Levin (Fernando e Nicholson, 1985).

Por outro lado os métodos de mínimos quadrados não foram desenhados para serem numericamente robustos, de facto foram derivados numa base analítica, não tomando em conta a precisão numérica limitada dos computadores digitais. Um dos problemas consiste na inversão directa de matrizes; se fôr possível estimar os parâmetros sem se inverter explicitamente matrizes, aumenta-se a robustez e a precisão dos cálculos (Bierman, 1977). Métodos comuns para evitar a inversão directa de matrizes são a decomposição em valores singulares ou *Singular Value Decomposition* - SVD e a decomposição QR (Björck, 1996).

Uma alternativa simples e de computação rápida aos estimadores já descritos, consiste em aplicar métodos de gradiente ao estimador. A convergência dos parâmetros é no entanto lenta e a adpatação da estimação a alterações nos parâmetros do sistema é pobre (Wellstead e M Zarrop (1991).

Os algoritmos evolutivos, são outra classe de algoritmos que mais recentemente começou a ser aplicada na identificação de sistemas (Kristinn e Guy, 1992; Iba e Kurita, 1993; De Moura Oliveira e Jones 1998). Algoritmos genéticos têm vindo a ser utilizados na determinação de funções de transferência de sistemas discretos LTI onde a ordem não é conhecida à partida. Embora normalmente possibilitem encontrar boas soluções são no entanto computacionalmente muito pesados e de convergência lenta**.**

Como o objectivo dos casos de estudo é demonstrar a implementação sobre o SPAM e analisar o desempenho dessa, foi implementado o algoritmo básico (Clarke et al, 1987a), que é relativamente simples, não sendo muito longa a exposição da sua implementação, mas computacionalmente intensivo o suficiente para permitir avaliar o desempenho paralelo desta.
Para identificar os parâmetros da função de transferência é utilizado um estimador RLS, semelhante ao utilizado no algoritmo básico.
Assim, o primeiro de dois estágios do algoritmo AGPC consiste na estimação da função de transferência do sistema a controlar, pelo estimador RLS. Os parâmetros estimados são então injectados no segundo estágio, o predictor GPC, que calcula a saída prevista do sistema e o sinal de controlo, minimizando uma função de custo apropriada, equação (8-20). O diagrama de blocos deste esquema de controlo é apresentado na Fig. 8-1.

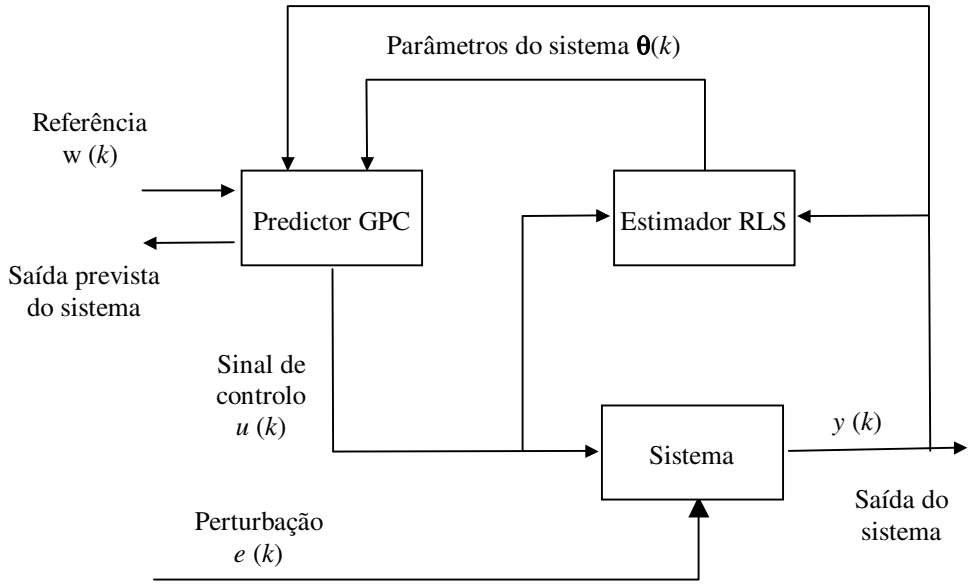


Fig. 8-1 Diagrama de blocos de um sistema controlado por um algoritmo AGPC

Ambos estes dois estágios são implementados recorrendo a álgebra matricial. Normalmente o primeiro opera sobre pequenos conjuntos de dados e o segundo sobre conjuntos de dados de proporções já consideráveis.
O modelo discreto do sistema é dado por:

(8-1) $A(q^{-1})\, y(k) = B(q^{-1})\, u(k-d_o) + C(q^{-1})\, e(k)$

onde: $y(k)$ representa o sinal de saída no instante k
$u(k-d_0)$ representa o sinal de entrada no instante $k - d_0$

$e\ (k)$	representa o sinal de erro no instante k. O erro é dado por uma sequência aleatória não correlacionada.
d_0	representa o atraso máximo entre a entrada e a saída do sistema, e é dado por nb - na.

A (q^{-1}), B (q^{-1}) e C (q^{-1}) são polinómios definidos em q^{-1} (operador atraso) tais que:

$$A(q^{-1}) = 1 + a_1 q^{-1} + ... + a_{na} q^{-na}$$
$$B(q^{-1}) = b_0 + b_1 q^{-1} + ... + b_{nb} q^{-nb}$$
$$C(q^{-1}) = 1 + c_1 q^{-1} + ... + c_{nc} q^{-nc}$$

Assumindo que o grau do polinómio A é igual ao grau do polinómio C, isto é, n_a=n_c, utiliza-se n para se referir ao grau de ambos os polinómios. Assumindo também que o grau de B é menor em uma unidade que o grau de A, isto é $d_0 = 1$, o modelo pode ser então rescrito:

$$(8\text{-}2) \quad A(q^{-1})\, y(k) = q^{-d} B(q^{-1})\, u(k\text{-}1) + C(q^{-1})\, e(k)$$

onde d é dado por $d = d_0 - 1$, e é assumido ≥ 0, o que implica que se $d > 0$, os primeiro d termos de B são nulos. Esta é a definição do modelo CARMA (*Controled auto-regressive moving average*) ou ARMAX.

No entanto, em algumas aplicações industriais a perturbação é não estacionária. Na prática existem dois principais tipos de perturbações: Degraus aleatórios em tempos aleatórios - por exemplo alterações na qualidade do material - e movimento Browniano - encontrado em sistemas dependentes de um equilíbrio de energia. Nestes casos a descrição do ruído é melhor representada por um ruído branco integrado. Isto leva-nos ao modelo CARIMA (*Controlled auto-regressive integrated moving average*) ou ARIMAX:

$$(8\text{-}3) \quad A(q^{-1})\, y(k) = q^{-d} B(q^{-1}) u(k-1) + \frac{C(q^{-1})}{\Delta} e(k)$$

onde Δ é o operador diferencial: $1 - q^{-1}$.

Finalmente, para definir completamente o modelo de sistema assumido, falta indicar que, se o polinómio C for de ordem igual ou superior a 1 está-se em presença de ruído colorido, se for de ordem 0 está-se na presença de ruído branco.

Nos pontos seguintes serão analisados os algoritmos usados para estes dois estágios.

8.1.1 Estimador

Um estimador, como o próprio nome indica deve estimar o valor dos parâmetros da função de transferência do sistema, isto é, os polinómios A, B e C, em tempo real. Como se pode ver pela Fig. 8-1, um estimador tem como entradas o sinal que é fornecido ao sistema e o sinal de saída deste, o qual pode ser influenciado pelo ruído presente no sistema.
Sejam os vectores:

$$\text{(8-4)} \quad \boldsymbol{\theta}^{\mathrm{T}} = \left[-a_1,...,-a_{n_a}, b_0,...,b_{n_b}, c_1,...,c_{n_c}\right]$$

$$\text{(8-5)} \quad \mathbf{x}^{\mathrm{T}}(k) = \left[y(k-1),...,y(k-n_a),u(k\text{-}d\text{-}1),...,u(k-d-n_b\text{-}1),e(k-1),...,e(k-n_c)\right]$$

onde $\boldsymbol{\theta}$ é o vector dos parâmetros desconhecidos e $\mathbf{x}$ é chamado vector de regressão. Os seguintes algoritmos recursivos, usam o vector regressão para armazenar o estado passado do sistema e retornam a estimativa de parâmetros no vector $\boldsymbol{\theta}$, em cada intervalo de amostragem. O primeiro algoritmo implementado (Wellstead e Zarrop, 1991), baseia-se no lema de inversão matricial e é dado por:

algoritmo 8-1: RLS MIL *(Recursive Least Squares, Matrix Inversion Lemma)*
Para o passo *k*:

#1 Formar a matriz de covariância $\mathbf{P}(k)$:

$$\text{(8-6)} \quad \mathbf{P}(k) = \mathbf{P}(k-1)\left[\mathbf{I}_m - \frac{\mathbf{x}(k)\mathbf{x}^{\mathrm{T}}(k)\mathbf{P}(k-1)}{1+\mathbf{x}^{\mathrm{T}}(k)\mathbf{P}(k-1)\mathbf{x}(k)}\right]$$

#2 Actualizar o vector de parâmetros desconhecidos $\boldsymbol{\theta}(k)$:

$$\text{(8-7)} \quad e(k) = y(k) - \mathbf{x}^{\mathrm{T}}(k)\boldsymbol{\theta}(k-1)$$

$$\text{(8-8)} \quad \boldsymbol{\theta}(k) = \boldsymbol{\theta}(k-1) + \mathbf{P}(k)\mathbf{x}(k)e(k)$$

#3 Actualizar o vector de regressão $\mathbf{x}(k+1)$ com os dados presentes: $u(k\text{-}d)$ e $y(k)$

Onde $\mathbf{x}(k)$ e $\boldsymbol{\theta}(k\text{-}1)$ são inicializados com zeros e $\mathbf{P}(k\text{-}1)$ a matriz de covariância é inicializada como o produto de uma matriz identidade por um escalar. Considerando C = 0 ou $e(k) = 0$ ou muito pequeno, a convergência do algoritmo 8-1 é muito rápida e constante, como se pode ver pela Fig. 8-2, onde está representada a identificação do sistema dado por:

$$\text{(8-9)} \quad (1+0{,}9q^{-1})\,y(k) = (0{,}5+0{,}8q^{-1})\,\mathrm{u}(k-1)$$

onde $u(k)$ é um onda quadrada de período igual a 20 intervalos de amostragem.

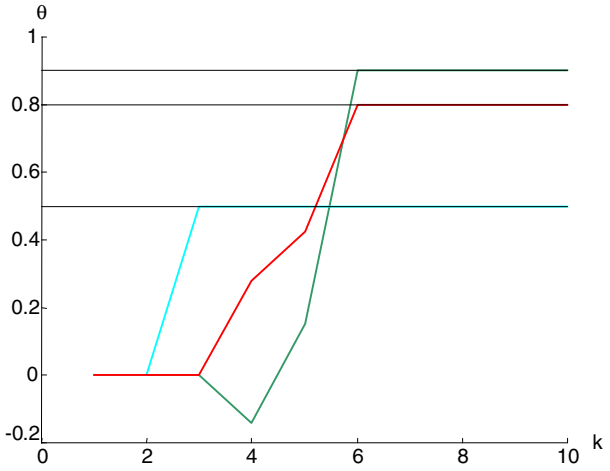


Fig. 8-2: Desempenho do algoritmo 8-1, para o sistema (8-9), com e(k) = 0

No caso de C = 1 este algoritmo pode ser aplicado embora a convergência seja mais lenta. Considerando outra vez o sistema dado por (8-9), mas com C = 1 e sendo $e(k)$ ruído branco de média zero e variância 0,1, o desempenho do algoritmo é mostrado na seguinte figura.

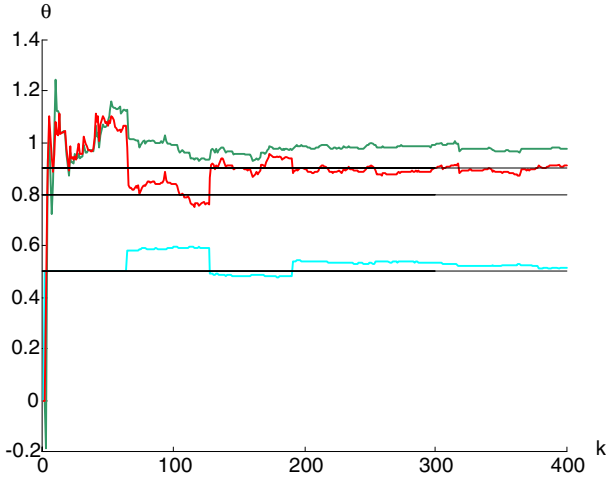


Fig. 8-3: Desempenho do algoritmo 8-1, para o sistema (8-9), na presença de ruído branco

Como se pode observar a precisão da medida dos parâmetros é consideravelmente menor. Se, além de ruído, os parâmetros do sistema variarem no tempo este algoritmo vai ter um desempenho ainda inferior, em termos de convergência. Seja o sistema:

(8-10) $(1-q^{-1}+0{,}25q^{-2})\,y(k) = (b_0)\,u(k-1)$

onde $u(k)$ é um onda quadrada de período igual a 20 intervalos de amostragem; e(k) é uma sequência de ruído branco de média zero e variância 0,1; $b_0 = 1$ para valores de $k < 200$ e b_0=

2 para valores de k > 200. Na seguinte figura pode-se analisar o comportamento do algoritmo 8-1, nesta situação.

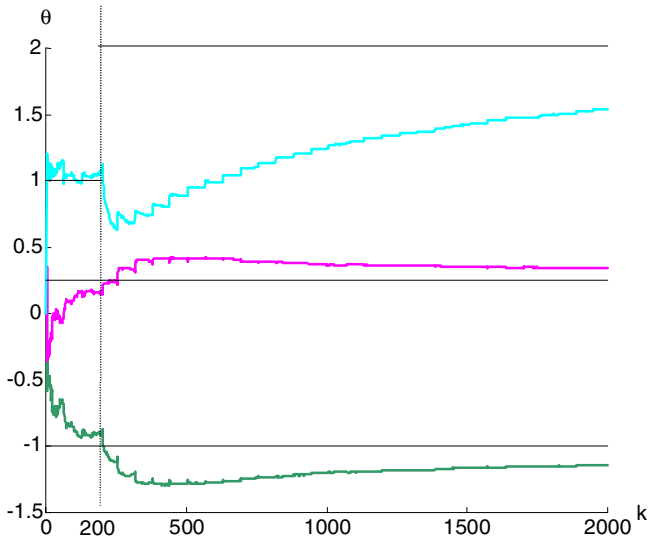


Fig. 8-4: Desempenho do algoritmo 8-1 em presença de ruído branco, quando os parâmetros variam no tempo - sistema dado por(8-10)

Como se pode observar, no instante k = 200, quando b_0 muda de valor a convergência para o novo valor de b_0 é muito lenta. Uma modificação no algoritmo 8-1, que permite reduzir o tempo de convergência (Wellstead e Zarrop, 1991), implica a introdução de uma nova variável chamada factor de esquecimento (λ), a qual afecta o peso das entradas passadas, de modo que a influência destas na estimação seja cada vez menor quanto mais atrasadas forem estas.

algoritmo 8-2: RLS MIL com Factor de Esquecimento (versão 1)

Modificar o algoritmo 8-1 de modo que no passo #1 a equação (8-6) seja substituída por:

$$\textbf{(8-11)}\ \mathbf{P}(k) = \lambda^{-1}\mathbf{P}(k-1)\left[\mathbf{I}_m - \frac{\mathbf{x}(k)\mathbf{x}^{\mathrm{T}}(k)\mathbf{P}(k-1)}{\lambda + \mathbf{x}^{\mathrm{T}}(k)\mathbf{P}(k-1)\mathbf{x}(k)}\right]$$

Onde λ é um escalar que toma valores menores ou iguais à unidade, embora tipicamente varie entre 0,99 e 0,95. Se λ = 1 o desempenho deste algoritmo é idêntico ao algoritmo anterior, como se pode ver pela Fig. 8-5 a) e b). À medida que λ vai decrescendo a velocidade de convergência após a mudança dos parâmetros aumenta no entanto a precisão com que os parâmetros são estimados diminui – casos c) e d) da mesma figura, onde o desempenho deste algoritmo modificado é comparado com o algoritmo 8-1, para o caso do sistema descrito acima por (8-10).

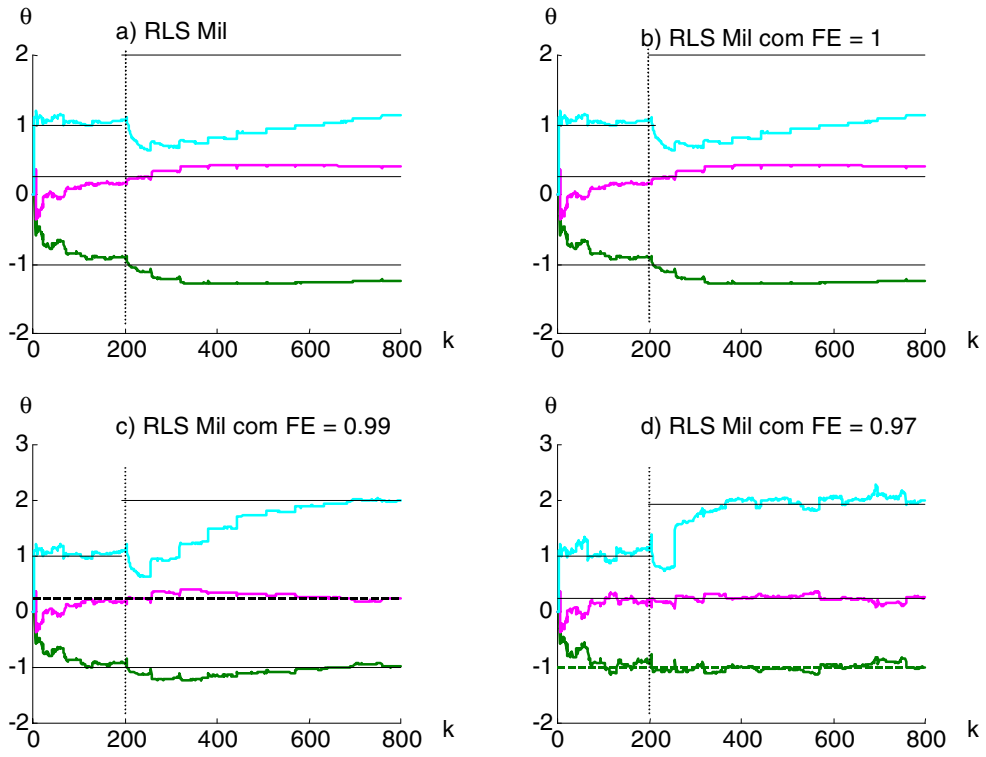


Fig. 8-5: Comparação do algoritmo 8-1 com o algoritmo 8-2, considerando diversos factores de esquecimento (FE) - sistema dado por(8-10)

Chega-se assim a um algoritmo que poderá ser usado para o estágio de estimação do AGPC, quer se esteja em presença de ruído branco ou colorido. No entanto ambos os algoritmos podem ainda ser melhorados em termos de esforço computacional (Wellstead e Zarrop, 1991), sem que a diferença entre a estimação dos valores dos parâmetros varie significativamente.
Aliás foi observada uma diferença na ordem de 10^{-10}, a qual tende rapidamente para zero com o tempo, mesmo quando existe variação de parâmetros no tempo, entre os algoritmos anteriores e o seguinte.

algoritmo 8-3: RLS MIL com Factor de Esquecimento (versão 2)

Modificar o algoritmo 8-2 de modo que o passo #1 fica:

#1 Formar a matriz de covariância $\mathbf{P}(k)$:

(8-12) $$\mathbf{K}(k) = \frac{\mathbf{P}(k-1)\mathbf{x}(k)}{\lambda + \mathbf{x}^{\mathrm{T}}(k)\mathbf{P}(k-1)\mathbf{x}(k)}$$

(8-13) $$\mathbf{P}(k) = \frac{\left[\mathbf{I}_m - \mathbf{K}(k)\mathbf{x}^{\mathrm{T}}(k)\right]\mathbf{P}(k-1)}{\lambda}$$

e no passo #2 a equação (8-8) é substituída por:

(8-14) $\boldsymbol{\theta}(k) = \boldsymbol{\theta}(k-1) + \mathbf{K}(k)e(k)$

Quanto a diferenças em termos de complexidade computacional dos vários algoritmos, a figura abaixo é suficientemente ilustrativa. Como se pode ver o algoritmo 8-2 é o mais complexo. É mais complexo que o primeiro algoritmo apresentado, visto ter de efectuar cálculos adicionais referentes à computação do factor de esquecimento. No entanto o algoritmo 8-3, que também computa o factor de esquecimento, e tem um desempenho idêntico ao algoritmo 8-2 em termos de estimação de parâmetros é substancialmente menos complexo em termos computacionais.

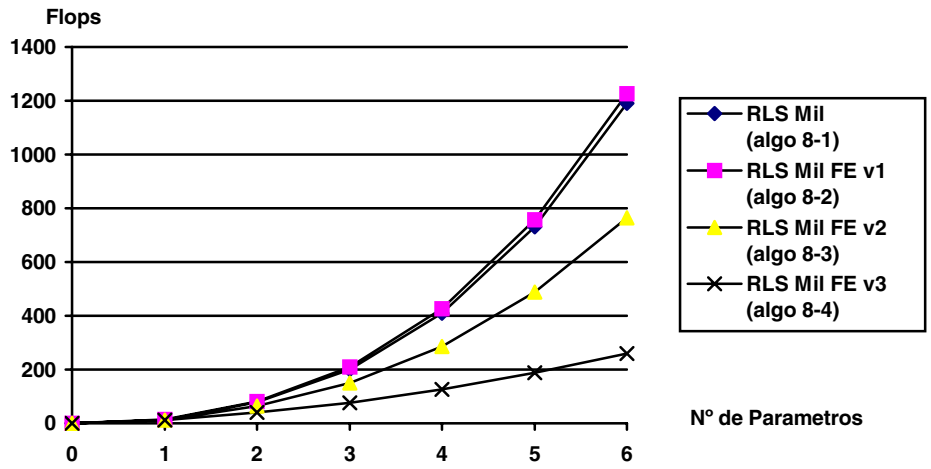


Fig. 8-6: Comparação dos vários algoritmos em termos de complexidade computacional.

Uma optimização em termos de complexidade computacional pode ainda ser operada no algoritmo 8-3, de modo a reduzir a complexidade computacional para os valores indicados na figura acima como RLS Mil FE v3.
Na equação (8-12), $\mathbf{P}(k-1)\,\mathbf{x}(k)$ é calculado no numerador e no denominador. Separando esta equação nas duas seguintes, evita-se calcular duas vezes o mesmo valor:

(8-15) $\alpha(k) = \mathbf{P}(k-1)\mathbf{x}(k)$

(8-16) $\mathbf{K}(k) = \dfrac{\alpha(k)}{\lambda + \mathbf{x}^{\mathrm{T}}(k)\alpha(k)}$

Assim, sendo m o número de parâmetros a estimar, esta transformação permite poupar um esforço computacional de $m.(2.m - 1)$ operações de vírgula flutuante ou *flops*, visto que $\mathbf{P}(k\text{-}1)$ é uma matriz quadrada e $\mathbf{x}(k)$ é um vector ambos, com m linhas.

Quanto à equação (8-13), esta poderá ser optimizada, desenvolvendo o numerador:

$$(8\text{-}17)\quad \mathbf{P}(k)=\frac{\mathbf{P}(k-1)-\mathbf{K}(k)\mathbf{x}^{\mathrm{T}}(k)\mathbf{P}(k-1)}{\lambda}$$

Como, de acordo com (Wellstead e Zarrop, 1991), $\mathbf{P}(k\text{-}1)$ é uma matriz simétrica, então a seguinte identidade é válida:

$$(8\text{-}18)\quad \mathbf{x}^{\mathrm{T}}(k)\mathbf{P}(k-1)=(\mathbf{P}(k-1)\mathbf{x}(k))^{\mathrm{T}}$$

e o membro direito de (8-18) é o vector resultante de (8-15) transposto. Aplicando (8-18) em (8-17) obtém-se:

$$(8\text{-}19)\quad \mathbf{P}(k)=\frac{\mathbf{P}(k-1)-\mathbf{K}(k)\alpha(k)^{\mathrm{T}}}{\lambda}$$

Novamente o esforço computacional é reduzido. De (8-17) e (8-19) observa-se que uma multiplicação da matriz $\mathbf{P}(k\text{-}1)$ pela matriz identidade é evitada, poupando um esforço computacional de $m^2.(2.m - 1)$ *flops*, e que a multiplicação de um vector por uma matriz também é evitada, recorrendo à transposta de α, que não necessita de ser computada novamente, reduzindo ainda o esforço computacional em $m.(2.m - 1)$ *flops*. No total de optimizações sequenciais verifica-se que o esforço computacional é reduzido em $\{2.[m.(2.m - 1)] + m^2.(2.m - 1)\} = (m^2 + 2.m).(2.m - 1)$ *flops*.
Assim o algoritmo do estimador que será usado no AGPC será:

algoritmo 8-4: RLS MIL com Factor de Esquecimento (versão 3)

Em cada instante de amostragem *k*:

#1 Formar a matriz de covariância $\mathbf{P}(k)$. Equações (8-15); (8-16) e (8-19).

#2 Actualizar o vector de parâmetros estimados $\theta(k)$. Equações (8-7) e (8-14).

#3 Actualizar o vector de regressão $\mathbf{x}$ $(k+1)$ com os dados presentes: $u\,(k\text{-}d)$ e $y\,(k)$

8.1.2 Predictor

O algoritmo AGPC implementado neste ponto, é o básico introduzido em (Clarke et al, 1987a), no entanto isto não implica que esteja tudo dito quanto a esta estratégia de controlo.

Pelo contrário em (Clarke et al, 1987b), algumas extensões ao algoritmo básico são apresentadas. De qualquer modo, ao longo do tempo, vários autores propuseram modificações e refinamentos adaptando este algoritmo a diversos casos de controlo, como já foi referido no inicio deste capítulo. No âmbito desta tese será apenas abordado o algoritmo básico, pois este, sendo composto de operações matriciais massivas, constituí um bom indicativo do desempenho de um algoritmo matricial paralelizado.

O algoritmo de Controlo Predictivo Generalizado básico, consiste na aplicação de uma sequência de controlo que minimiza a seguinte fu*-nção de custo:

$$(8\text{-}20)\quad J(N_1,N_2,N_u) = \mathrm{E}\left[\sum_{j=N_1}^{N_2}\delta(j)\left[\hat{y}(k+j\mid k)-w(k+j)\right]^2+\sum_{j=1}^{N_u}\lambda(j)\left[\Delta u(k+j-1)\right]^2\right]$$

onde: E [...] representa a esperança matemática

$\hat{y}(k+j\mid k)$ representa um predictor óptimo, que prevê a saída do sistema para j passos futuros, usando dados até ao instante k

N_1 representa o Mínimo horizonte de custo

N_2 representa o Máximo horizonte de custo

N_u representa o Horizonte de controlo

$\lambda(j)$ e $\delta(j)$ representam séries de compensação. Para o algoritmo básico, $\lambda(j)$ é considerado constante e $\delta(j) = 1$. Mas tal nem sempre é o óptimo. Para um problema GPC com restrições, o aumento de $\lambda(j)$ reduz o efeito de controlo. Em (Lee et al, 1997) é proposto um algoritmo, que em cada instante de amostragem calcula o $\lambda(j)$ óptimo, tal que todas as saídas do GPC satisfaçam as restrições magnitude do controlo e da variação deste.

$w(k+j)$ representa a futura trajectória de referência

Minimizando $J(N_1, N_2, N_u)$ obtém-se a sequência de controlo $[u(k), u(k+1), ..., u(k+N-1)]$, onde $N = N_1 - N_2 + 1$, de modo que a saída futura do sistema siga $w(k+j)$.
Para estimar a saída j instantes no futuro é necessário resolver a seguinte equação Diofantina:

$$(8\text{-}21)\quad C(q^{-1}) = E_j(q^{-1})\,\tilde{A}(q^{-1}) + q^{-j}F_j(q^{-1})$$

onde: $\tilde{A}(q^{-1}) = \Delta A(q^{-1})$ com: $\Delta = 1 - q^{-1}$

ordem $E_j = j - 1$

ordem $F_j = n_a$

E_j (q^{-1}) e F_j (q^{-1}) podem ser obtidos dividindo C (q^{-1}) por $\tilde{A}$ (q^{-1}) até que o resto possa ser factorizado como q^{-j} F_j (q^{-1}), sendo E_j (q^{-1}) o quociente (Clarke et al, 1987a; Camacho e Bordons, 1994). Seja G dado por:

(8-22) $G_j = E_j \,.\, B, \quad j = 1,\ldots, N.$

Considerando a seguinte série de *j* futuras predições óptimas:

$$\begin{aligned}
\hat{y}(k+d+1 \mid k) &= G_{d+1}(q^{-1})\Delta u(k) + F_{d+1}(q^{-1})y(k) \\
\hat{y}(k+d+2 \mid k) &= G_{d+2}(q^{-1})\Delta u(k+1) + F_{d+2}(q^{-1})y(k) \\
&\vdots \\
\hat{y}(k+d+N \mid k) &= G_{d+N}(q^{-1})\Delta u(k+N-1) + F_{d+N}(q^{-1})y(k)
\end{aligned}$$

que podem ser escritas na forma matricial por:

(8-23) $\hat{\mathbf{y}} = \mathbf{G}\Delta\mathbf{u} + \mathbf{f}$

onde:

$$\Delta\mathbf{u}^{\mathrm{T}} = \left[\Delta u(k), \Delta u(k+1), \cdots, \Delta u(k+N-1)\right]$$
$$\hat{\mathbf{y}}^{\mathrm{T}} = \left[\hat{y}(k+d+1 \mid k), \hat{y}(k+d+2 \mid k), \cdots, \hat{y}(k+d+N \mid k)\right]$$

(8-24) $$\mathbf{G} = \begin{bmatrix} g_0 & 0 & \cdots & 0 \\ g_1 & g_0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ g_{N-1} & g_{N-2} & \cdots & g_0 \end{bmatrix}, \; g_n = n-\text{ésimo coeficiente de } G_{d+N}, n = 0,\ldots,N-1$$

(8-25) $$\mathbf{f} = \begin{bmatrix} (G_{d+1}(q^{-1}) - g_0)\,\Delta u(k\text{-}1) + F_{d+1}(q^{-1})\,y(k) \\ (G_{d+2}(q^{-1}) - g_0 - g_1 q^{-1})\,\Delta u(k\text{-}1) + F_{d+2}(q^{-1})\,y(k) \\ \vdots \\ (G_{d+N}(q^{-1}) - g_0 - g_1 q^{-1} - \cdots - g_{N-1} q^{-(N-1)})\,\Delta u(k\text{-}1) + F_{d+N}(q^{-1})\,y(k) \end{bmatrix}$$

ou de outro modo, seja **g** o vector dado por:

$$(8\text{-}26)\quad \mathbf{g} = \begin{bmatrix} (\mathrm{G}_{d+1}(q^{-1}) - g_0) \\ (\mathrm{G}_{d+2}(q^{-1}) - g_0 - g_1 q^{-1}) \\ \vdots \\ (\mathrm{G}_{d+N}(q^{-1}) - g_0 - g_1 q^{-1} - \cdots - g_{N-1} q^{-(N-1)}) \end{bmatrix}$$

isto é, os coeficientes de menor potência, não nulos, de cada polinómio G_j, e **F** uma matriz com *j* linhas, onde cada linha *j* representa os coeficientes de $F_j(q^{-1})$, **f** pode ser rescrita como:

$$(8\text{-}27)\quad \mathbf{f} = \mathbf{F1}.\mathbf{yu} = [\mathbf{g} \mid \mathbf{F}] \begin{bmatrix} \Delta u(k-1) \\ y(k) \\ \vdots \\ y(k-N-1) \end{bmatrix}$$

e

$$(8\text{-}28)\quad N = N_2 - N_1 + 1 = N_u.$$

É conveniente notar que N_1 deve ser igual a d+1, pois no caso de ser inferior os termos adicionados à função de custo (8-20) dependem apenas dos sinais de controlo passados, enquanto que se for superior os primeiros pontos na sequência de referência, aqueles previstos com mais precisão, não são tomados em conta. Por outro lado, se o atraso temporal não for conhecido deve ter o valor 1, isto é d_0=1 $\Rightarrow$ d=0. N_2 deve ser escolhido de forma que (N_2 . t_s) seja da mesma magnitude que o tempo de crescimento do sistema, onde t_s é o tempo de amostragem do controlador. Em (Clarke et al, 1987a) é apontado que uma vasta classe de sistemas pode ser estabilizado por esta técnica se N_1 = 1 e N_2 = 10. No entanto para sistemas complexos deve ser pelo menos igual ao número de pólos instáveis. Se o atraso do sistema for superior a 1, isto é se $d > 0$ as primeira d linhas de **G** serão nulas, mas se este for conhecido e N_1 for igual a d+1, as linhas nulas não terão influência no cálculo. No entanto uma solução estável do algoritmo é possível mesmo se as linhas iniciais de **G**, forem nulas (Clarke et al, 1987a). Outros autores propõem horizontes muito grandes, como é o caso do modelo de controlo predictivo não linear proposto por (Chen e Allgöwer, 1997).

A solução mínima de (8-20), **Δu,** pode ser obtida com:

$$(8\text{-}29)\quad \Delta\mathbf{u} = (\mathbf{G}^{\mathrm{T}}\ \mathbf{G} + \lambda\ \mathbf{I})^{-1}\ \mathbf{G}^{\mathrm{T}}\ (\mathbf{w} - \mathbf{f})$$

onde o vector:

$$\mathbf{w} = \left[\mathrm{w}(k+d+1), \mathrm{w}(k+d+2), \cdots, \mathrm{w}(k+d+N)\right]$$

é definido como uma sequência de referência, a qual deve ser seguida pela resposta do sistema. Em alguns casos, como por exemplo em robótica, futuras variações nesta série, w(k+j), serão conhecidas.

Como no algoritmo IDCOM, (Richalet et al., 1978) pode ser requerida uma aproximação suave da saída à referência, obtida pelo seguinte modelo de primeira ordem, onde para uma transição lenta entre a saída e a referência, α deve ser aproximadamente 1.

(8-30) $\mathrm{w}(k) = \mathrm{y}(k)$

$\mathrm{w}(k+j) = \alpha\, \mathrm{w}(k+j\text{-}1) + (1\text{-}\alpha)\, \mathrm{w}(k+j), \qquad j = 1, 2, 3, ..., N$

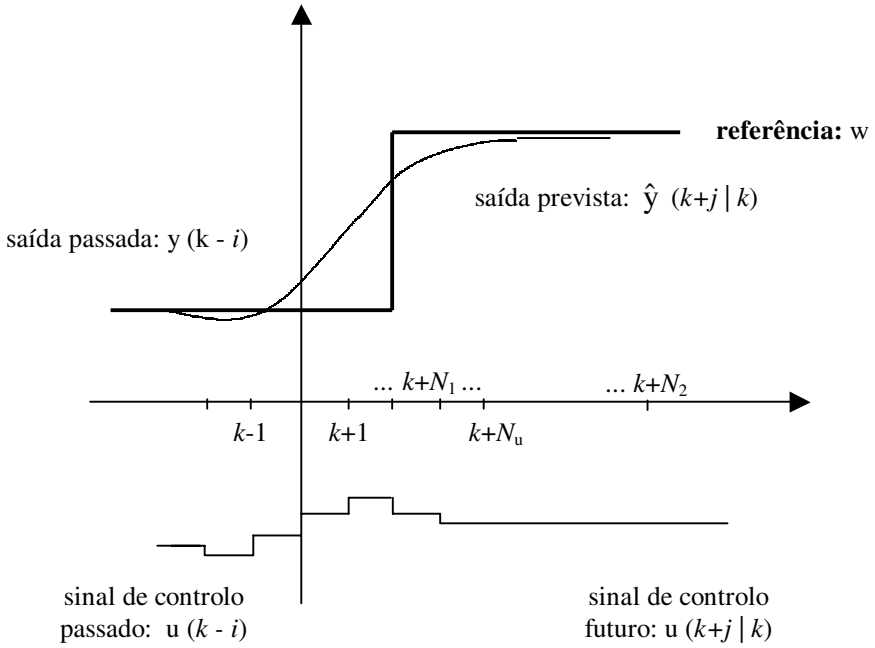


Fig. 8-7: Estratégia de controlo GPC.

Finalmente o sinal de controlo é dado por:

(8-31) $u(k) = u(k\text{-}1) + \Delta\mathbf{u}(1)$

Para ilustrar esta estratégia de controlo, seja o sistema dado pela função de transferência:

$$(8\text{-}32)\quad H(s) = \frac{1}{25s^2 + 10s}$$

que tem um pólo na origem. A resposta ao degrau unitário deste sistema, característica de um sistema instável, está representada na figura seguinte:

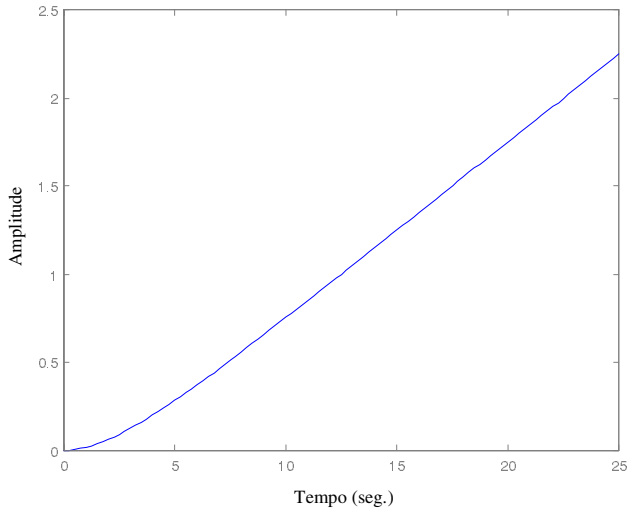


Fig. 8-8: Resposta do sistema (8-32) a um degrau unitário.

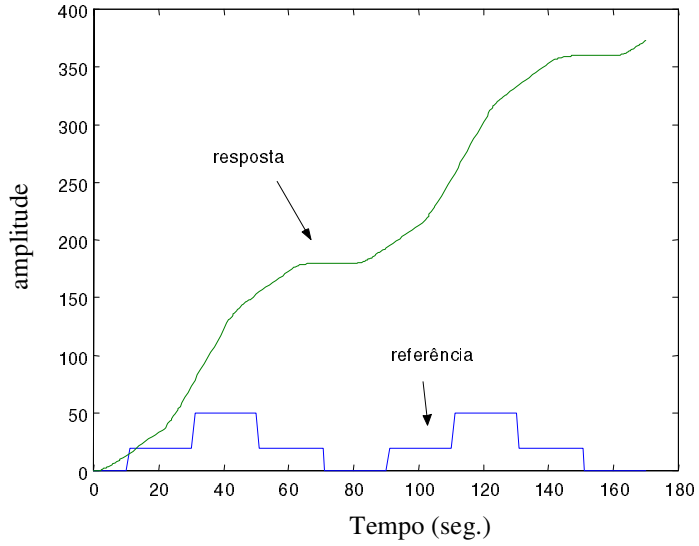


Fig. 8-9: Resposta do sistema (8-32) em malha aberta.

Na Fig. 8-9 é dada a resposta deste sistema, em malha aberta, a um determinado sinal de referência, e no topo da Fig. 8-10 é dada a resposta do mesmo sistema, ao mesmo sinal de referência, ao longo de 240 intervalos de amostragem, de um segundo cada, mas controlado pelo algoritmo AGPC. Na metade inferior da mesma figura é indicado o sinal de controlo $u(k)$, calculado pelo algoritmo e aplicado ao sistema. Para os primeiros 10 intervalos de amostragem, isto é o comprimento do horizonte de controlo, o sinal de controlo foi considerado constante e igual a 10.

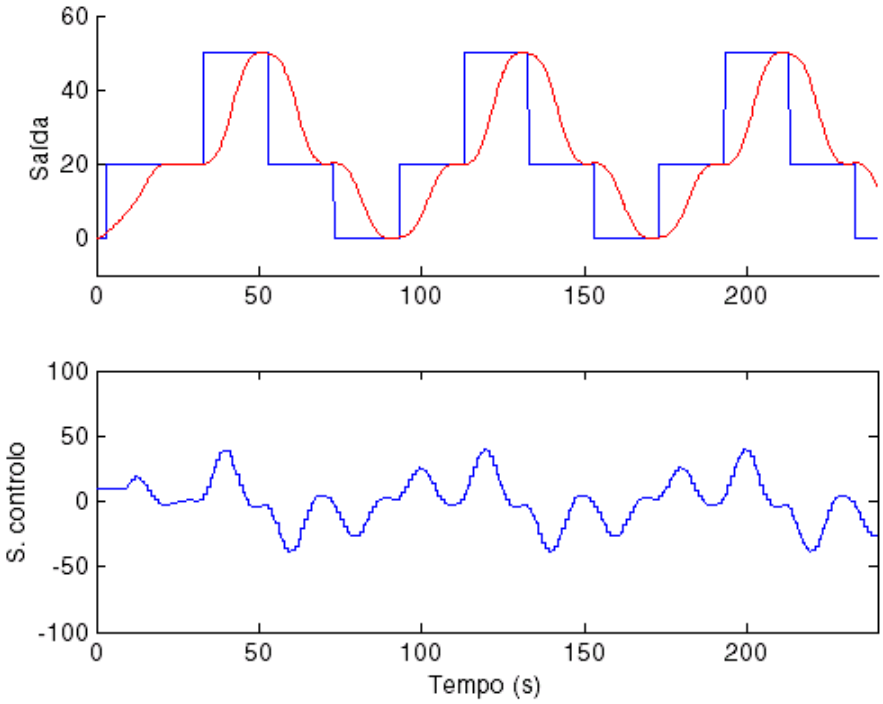


Fig. 8-10: Resposta do sistema (8-32) com controlo AGPC.

Para a estimação da função de transferência, foi utilizado o estimador RLS descrito pelo algoritmo 8-4, configurado para identificar um numerador de 1ª ordem e um denominador de 2ª ordem, com factor de esquecimento 0.9. O horizonte de controlo seleccionado foi Nu=1, e o mínimo e máximo horizontes de custo N_1=1 e N_2=10 respectivamente. A sequência λ foi considerada constante e igual a 0.8 e $\alpha = 0.7$.

Esta estratégia de controlo pode ser aplicada mesmo quando a função de transferência do sistema varia. No exemplo seguinte a função de transferência do sistema varia no intervalo k=170, onde os intervalos de amostragem têm período unitário. A função de transferência expressa tanto em transformada de Laplace como de Z, é dada pela tabela seguinte:

k	Modelo (s)	Modelo (z)
0-169	$H(s) = \frac{0.5s + 1}{s + 0.2}$	$H(z) = \frac{0.5z + 0.4063}{z - 0.8187}$
170-319	$H(s) = \frac{0.5}{s + 0.25}$	$H(z) = \frac{0.4424}{z - 0.7788}$

tabela 8-1: Modelo do sistema ao longo do tempo

Neste caso o estimador RLS foi configurado para estimar um numerador e um denominador de 1ª ordem ao longo de todo o ensaio, com factor de esquecimento 0.95. No predictor, os horizontes seleccionados foram $N_1 = 1$ e $N_2 = Nu = 3$, a sequência λ foi também considerada constante e igual a 0.8 e a constante $\alpha = 0.6$. O sinal de controlo aplicado nos primeiros 3 intervalos de amostragem foi unitário.

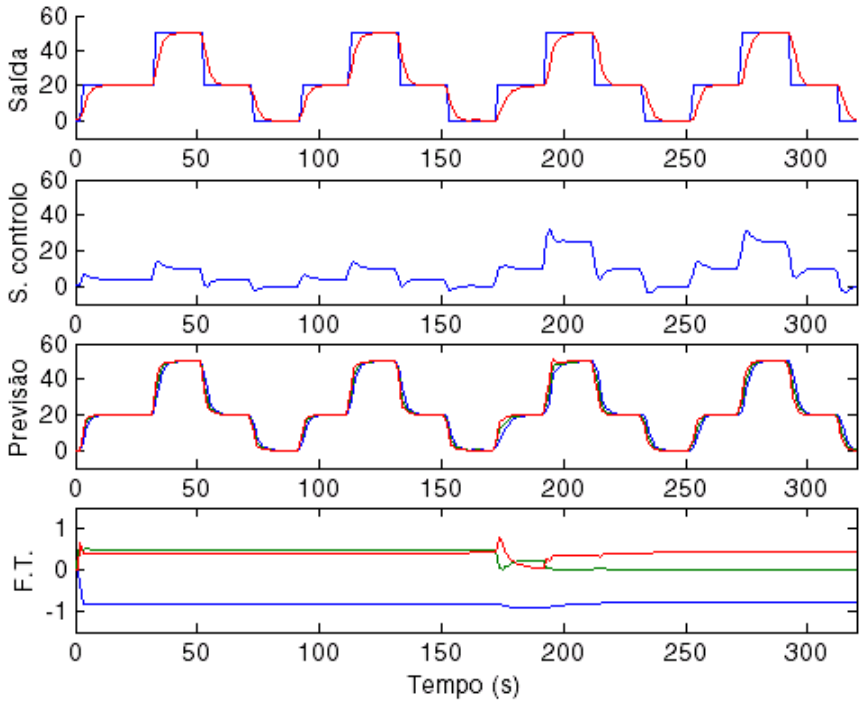


Fig. 8-11: Resposta do sistema (8-32) com controlo AGPC.

Nestas condições obtém-se a resposta apresentada na Fig. 8-11. Nesta figura, de cima para baixo são apresentadas as amplitudes da resposta do sistema à referência, do sinal de controlo,

das previsões da saída 3 intervalos de amostragem no futuro, e da estimação dos parâmetros do sistema.

Em (Clarke et al, 1987a) é apresentada uma comparação entre o algoritmo AGPC e outras estratégias de controlo, nomeadamente PID não adaptativo, GMV e colocação de pólos adaptativa. Nesse exemplo, a função de transferência do sistema que se pretende controlar varia ao longo do tempo, de uma forma drástica e pouco comum, mas que serve para demonstrar a robustez do algoritmo AGPC face a outras estratégias de controlo, quando o sistema varia em atraso, ordem e valores dos parâmetros da função de transferência.

8.1.3 Implementação do algoritmo AGPC em SEQ

A implementação de um algoritmo matricial como este em SEQ, garante que a paralelização deste é efectuada ao nível dos dados automaticamente, de modo que praticamente bastará traduzir o algoritmo matemático acima para a linguagem de programação SEQ. No entanto, para a implementação do predictor será necessário desenvolver algumas rotinas que não existem na biblioteca. Estas devem serão desenvolvidas em C mantendo a filosofia do particionamento de dados por conjuntos de linhas consecutivas, e sempre que possível utilizando funções da biblioteca padrão descritas no capítulo 5.
Assim o núcleo computacional deste algoritmo poderá ser dividido em 3 partes.

Algoritmo 8-5:

```
while 1 {
        #1 Receber entrada, saída e referência actuais
        #2 Estimar ok, vector de parâmetros da função de transferência do processo
                                                                        (RLS)
        #3 Calcular uk, o sinal de controlo (GPC)
        }
```

A primeira instrução é executada sempre que se inicia um novo período de amostragem, de modo a actualizar o valor da entrada e da saída do processo, bem como da referência.
Seguidamente deve ser estimada a função de transferência do processo (RLS) e calculado o sinal de controlo. Seguidamente apresenta-se o algoritmo completo, traduzido para SEQ, onde ordA e ordB são variáveis escalares, definidas pelo utilizador e que indicam a ordem do numerador e denominador da função de transferência do sistema a estimar e controlar. É assumido também que ordA é maior que ordB.

Algoritmo 8-6:

```
k=1;
while 1 {

/*Recebe do GPC a entrada do sistema, (o sinal de controlo), uk, e do exterior a
saída e setpoint, yk e wk */

/* RLS */
ak = Pk * xk;                    /*eq. (8-15)*/
kk = ak / (ff + xk' * ak);       /*eq. (8-16)*/
Pk = (Pk - kk * ak') / ff;       /*eq. (8-19)*/

bk = xk' * ok;                   /*eq. (8-7)*/
ek = yk - bk[1,1];
ok = ok + kk * ek;               /*eq. (8-14)*/

shr(xk, 1, ordA, yk);            /*eq. (8-5)*/
shr(xk, 1+ordA, 1+ordA+ordB, uk);

/* AGPC */
A = diff1(ok, ordA);             /*Ã em eq. (8-21)*/
B = ok[1+ordA:1+ordA+ordB, 1];

shr(W, 1, N, wk);
YU[1,1]=duk_1;
shr(YU, 2, 1+N, duk_1);

polydiv (A, E, F, N);            /*E e F em eq. (8-21)*/
polymult (E, B, G1);             /*eq. (8-22)*/
agpctrii (F, G1, F1, G);         /*eq. (8-24)*/

if (k<N) then { uk=uinit; }
else {
        f = F1*YU;               /*eq. (8-27)*/

        T0=G'*G;                         /*eq. (8-29)*/
        T0=dadd(T0, lambda, T0);
        T0 = inv (T0);
        U=T0*G'*(W-f);
        Ya= G*U+f;                       /*eq. (8-23)*/

        duk_1=U[1,1];
        uk = uk_1+du_1;                  /*eq. (8-31)*/
    }

uk_1 = uk;
k = k + 1;
}
```

O estágio RLS consiste apenas numa tradução das equações indicadas em comentários para SEQ, de modo a implementar o algoritmo 8-4: RLS MIL com Factor de Esquecimento (versão 3). No fim deste estágio são preparados os dados para a próxima iteração.

Já o estágio GPC inicia-se preparando os dados para a iteração corrente, usando também o vector de parâmetros do sistema: **ok**. Esta preparação de dados é efectuada de acordo com a formulação matemática já apresentada. Para isso foi necessário desenvolver algumas funções que são definidas a seguir.

8.1.3.1 Funções de suporte para AGPC

Para obter as matrizes F e E, a equação (8-21) deve ser resolvida. Desta forma é necessário obter a partir dos parâmetros do denominador da função de transferência do sistema o polinómio Ã. As duas funções seguintes são usadas para resolver esta equação:

Objectivo: calcula Ã para utilizar na equação (8-21) como: A . (1 - q^{-1})
PRE: rows(ok)>=ordA && cols(ok)==1 && rows(Ã)==ordA+2 && cols(Ã)==1
POS: retorna A . (1-q^{-1}) && old(Ã) é destruído
Excepção: Se não houver memória suficiente para dimensionar o vector b gera um erro em tempo de execução.

```
matrix diff1(matrix ok, integer ordA, matrix Ã) {
integer i;
real buf;
matrix b;

dim b[ordA+2, 1];

Ã=-ok;
shr(Ã, 1, 2+ordA, 1);
Ã[ordA+2,1]=0;

b=Ã;
shr(b, 1, 2+ordA, 0);


Ã = Ã-b;
clear b;
exit Ã;
}
```

Repare-se que é recebido o vector **ok**, que contém os parâmetros tanto do numerador como do denominador da função de transferência. Assim A é extraído deste.

Como a operação Ã = A . (1 - q^{-1}), na realidade significa deslocar o vector A uma posição para a direita e subtrair-se a si próprio não deslocado, a primeira parte desta função consiste na obtenção destes dois vectores, Ã e b, utilizando as funções de deslocamento, o que implica transferências de dados locais e remotas, resultando nalguma perda de eficiência.

Finalmente a operação de subtracção, Ã - b, é executada em paralelo.

Convém referir que é necessário retornar Ã com exit Ã, pois como este vector é também usado para guardar o resultado da operação de subtracção, e as operações matriciais criam sempre uma nova matriz para o resultado, mesmo que esta já exista, o argumento de entrada deixa de apontar para a zona de memória correspondente.

Esta funcionalidade foi efectuada para evitar que a matriz resultado tivesse que ser dimensionada pelo utilizador de acordo com a operação e a dimensão dos operandos. Usando esta técnica cada operação sabe qual a dimensão da matriz resultado, como uma função das dimensões dos operandos. É no entanto necessário não assumir que se pode devolver o resultado de uma operação matricial pelo parâmetros de entrada de uma função.

Após a obtenção de Ã já se pode resolver a equação (8-21):

Objectivo: constrói F e E tal que:
$1 = Ej(q^{-1})\ \tilde{A}(q^{-1}) + q^{-j}\ Fj(q^{-1}),$ com j = 1 .. N
PRE: cols(Ã) == 1 && rows(E) == N && cols(E) == N &&
rows(F) == N && cols(F) == lins (Ã) - 1 && N > 0
POS:
Excepção:

```
polydiv (matrix A, matrix E, matrix F, integer N) {
real r,q;
integer l,k,c,oA;

fill (E, 0);
E[1,1]=1;

oA=rows(A)-1;

for k=1 to N {
        if (k==1) then {
                q=1;
                for c=1 to oA {
                        r=-A[c+1, 1];
                        F[1, c]=r; }
                }
        else {
                q=F[k-1,1];
```

```
                for c=2 to oA+1 {
                        if (c>oA) then { r=0; }
                        else          { r=F[k-1, c]; }
                        r=r-A[c, 1]*q;
                        F[k, c-1]=r; }
            }
        for l = k to N {
                E[l,k]=q; }
        }
}
```

O que esta função faz é resolver uma equação diofantina na forma:

(8-33) $1 = \mathrm{E}_{j,*}(x)\,\tilde{\mathrm{A}}(x) + x^{j}\mathrm{F}_{j,*}(x) \quad \text{com } j = 1 \,..\, \mathrm{N}$

onde Ã, $\mathrm{E}_{j,*}$ e $\mathrm{F}_{j,*}$ são polinómios em *x*, cuja ordem de $\mathrm{E}_{j,*}$ é N, e de $\mathrm{F}_{j,*}$ é N - 1. As matrizes E (*x*) e F (*x*) são obtidas dividindo 1 por Ã (*x*) até que o resto possa ser factorizado como x^{j} $\mathrm{F}_{j,*}$ (*x*) e onde $\mathrm{E}_{j,*}$ (*x*) é o quociente.

O polinómio Ã é um vector coluna, ao passo que $\mathrm{E}_{j,*}$ e $\mathrm{F}_{j,*}$, representam a linha *j* das matrizes E e F respectivamente. De qualquer modo, sejam linha ou coluna os vectores representam polinómios com a ordem dada pelo número de elementos menos um, e onde os coeficientes das potências maiores são dados pelos primeiros elementos do vector.

Neste caso como não é utilizada nenhuma operação da biblioteca de cálculo matricial, mas simplesmente manipulação de elementos entre vectores e matrizes, já é seguro retornar nos parâmetros de entrada E e F.

Esta função foi desenhada exactamente como uma função sequencial, de modo a deixar que o paralelismo possível seja disponibilizado pelo ambiente de suporte da linguagem SEQ.

Não é no entanto a melhor técnica. Se bem que o SPAM paralelize o algoritmo, o que sucede principalmente dentro do ciclo principal é o acesso a elementos remotos um a um. Isto representa o menor desempenho do ambiente de comunicação, pois como cada pacote só contém um elemento, o comprimento dos dados é igual ao comprimento do cabeçalhos, fazendo com que a largura de banda caía automaticamente para 50%. Além disso tem-se uma grande densidade de tráfego ao enviar concorrentemente um grande número de pacotes, com apenas um elemento entre todos os processadores da rede.

Esta estratégia foi ensaiada numa rede de Transputers, onde o ambiente de comunicações tem um desempenho quase ideal e numa rede de Sharc onde o desempenho deste já não é tão

próximo do modelo teórico. Utilizou-e um sistema de 2ª ordem onde A é um vector com 4 elementos, obtendo-se:

	2 processadores		**3 processadores**	
	T8	**ADSP2106x**	**T8**	**ADSP2106x**
polydiv v1	271 ms	101 ms	265 ms	123 ms
polydiv v2	58 ms	4 ms	38 ms	2 ms

tabela 8-2: Comparação das duas versões de polydiv em termos de tempo em mili-segundos para um sistema de 2ª ordem

Onde polydiv v1 refere-se à versão 1 de polydiv, isto é ao algoritmo acima apresentado enquanto a versão 2 representa um algoritmo sequencial onde todos os processadores calculam toda a matriz F e E, descartando depois as linhas que não interessam. No entanto como as comunicações são todas eliminadas com excepção da obtenção de uma cópia local do vector A em todos os nós logo no início, o tempo de execução caí drasticamente.
Como seria de esperar, devido ao melhor desempenho do ambiente de comunicação em Transputers, esta melhoria beneficia mais o Sharc que o Transputer. De facto para o Sharc a versão 1 ocupa um tempo significativo quando comparado com todo o algoritmo AGPC, ao passo que no Transputer é praticamente negligenciável.
Para optimizar ao máximo a versão 2, o seu código foi implementado em C.

Podia-se no entanto desenvolver uma versão paralela. Uma abordagem possível consiste em paralelizar a própria operação de divisão como efectuada no papel, isto é distribuir F por colunas, em vez da abordagem utilizada no SPAM por linhas. Assim, omitindo os coeficientes das potências para simplificar, tem-se:

$$F = \begin{array}{r} 1 \\ -\ \underline{1 \quad q^{-1} \quad q^{-2}} \\ 0 \quad q^{-1} \quad q^{-2} \\ -\ \underline{q^{-1} \quad q^{-2} \quad q^{-3}} \\ 0 \quad q^{-2} \quad q^{-3} \\ -\ \underline{q^{-2} \quad q^{-3} \quad q^{-4}} \\ 0 \quad q^{-3} \quad q^{-4} \\ (...) \end{array} \qquad E = \begin{array}{r} \tilde{A}^T \\ |\underline{1 \quad + q^{-1} \quad + q^{-2} \quad ....} \\ 1 \quad 0 \quad 0 \\ 1 \quad q^{-1} \quad 0 \\ 1 \quad q^{-1} \quad q^{-2} \\ (...) \end{array}$$

Se Ã estiver distribuído por linhas, do mesmo modo que F por colunas, cada operação de subtracção pode ser feita concorrentemente.

Repare-se que para cada linha *j* os elementos da linha seguinte de F teriam de ser deslocados para a esquerda, para manter a mesma distribuição. Já a matriz E, para cada linha consiste na cópia da anterior acrescentando mais um elemento não nulo. Este esquema tem no entanto o inconveniente de existir a necessidade frequente de comunicação de elementos entre nós.

No entanto, como neste caso de teste se utilizam sistemas de 2ª ordem e não é normal utilizar sistemas de ordem muito superior, os vectores envolvidos são relativamente pequenos de modo que não se justificava um esforço de paralelização desta rotina, pois a versão sequencial tem um desempenho muito bom e o seu tempo de cálculo é pouco importante se se comparar com todo o algoritmo AGPC. Se no entanto forem utilizados sistemas de ordem muito superior, onde o vector A será grande, já se justifica um esquema de paralelização como o atrás referido.

Antes de executar o núcleo do algoritmo AGPC, têm-se ainda de obter G1 na equação (8-22). Para tal é utilizada a função:

Objectivo: retorna G1 tal que:
$G1_{k,*} = E_{k,*} \times B$ (onde x representa a multiplicação polinomial)
PRE: rows(E) == cols(E) == N && cols(B) == 1 &&
rows(G1) == rows(E) && cols(G1) == (rows(B)-1)+cols(E)
POS:
Excepção: Se não houver memória suficiente para alocar o vector BL, a cópia local de B, gera um erro em tempo de execução.

```
polymult (matrix E, matrix B, matrix G1) {

C {   matrix BL=NULL;
      REAL n;
      INTEGER loop, l, lin=(int) NLINHAS(B);
      INTEGER ordE=(int) NCOLUNAS(E)-1,
                        ordB=(int) NLINHAS(B)-1,
                        ordG1=(int) NCOLUNAS(G1)-1;

      INTEGER l1, l2, ll=(int) LLINHAS(E);
      matrix r=G1+MDATA;
      matrix e=E+MDATA;
      matrix b;

      BL=SET_MATRIX(BL, lin, 1, "polymult: B buffer");
      b=BL+MDATA;
```

```
      COLLECT (B, BL);

       matrix_fill(G1,0);
       for (loop=0; loop<ll; loop++) {
                    for (l1=ordE; l1>=0; l1--)
                          for (l2=ordB; l2>=0; l2--)
                   r[l1+l2+loop*(ordG1+1)]+=e[l1+loop*(ordE+1)]*b[l2];}
      free(BL);
  }
}
```

Repare-se que esta função efectua a multiplicação do polinómio B por todas as linhas de E. No entanto como B é um vector coluna, deve ser primeiro criada uma cópia local desse, BL em cada nó, o que implica alguma perda de eficiência. Depois, pode ser efectuado em paralelo a multiplicação da cópia local por cada linha alocada num nó.

A criação de uma cópia local de B não pode ser efectuada em SEQ, pois esta linguagem abstrai-se do paralelismo. Assim a função foi programada em C.

Finalmente a matriz triangular inferior G (8-24) e a matriz F1 em (8-27), são obtidas com:

```
Objectivo:      retorna F1 = [g | F]
                        onde g é coeficiente x0 da linha x0 de G1
                        e G segundo a eq. (8-24)
PRE:            rows(F) == N && cols(F) == cols(A) &&
                rows(G1) == N && cols(G1) == (rows(B)-1)+N &&
rows(F1) == N && cols(F1) == cols(F)+1 &&
         rows(G) == N && cols(G) == N &&
         rows(YU) == N+1 && cols(YU) == 1 && N > 0
POS:
Excepção:

agpctrii (matrix F, matrix G1, matrix F1, matrix G) {
integer l, c, c1, k0, k1, k2, k3;
integer x;

C {     k0=k1=k2=k3=0;
       for(l=LINHA_INICIO(G1); l<=LINHA_FIM(G1); l++) {
              for(c=0; c<l; c++)
                     G[MDATA+k0+c]=G1[MDATA+k1+(l-c-1)];
              for(c1=c; c1<NCOLUNAS(G); c1++)
                     G[MDATA+k0+c1]=0;

              F1[MDATA+k2]=G1[MDATA+k1+c];

              for(c=0; c<NCOLUNAS(F); c++)
```

```
            F1[MDATA+k2+c+1]=F[MDATA+k3+c];
          k0+=NCOLUNAS(G);
          k1+=NCOLUNAS(G1);
          k2+=NCOLUNAS(F1);
          k3+=NCOLUNAS(F);
          }
  }
}
```

Esta função retorna a triangularização de uma matriz G1 na matriz G segundo o seguinte critério:

$$(8\text{-}34)\quad G=\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{22} & a_{21} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{N\,M-1} & a_{N\,M-2} & \cdots & a_{N\,1} \end{bmatrix} \qquad G1=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NM} \end{bmatrix}$$

e cria a matriz F1 como:

$$(8\text{-}35)\quad F1 = [g \mid F]$$

onde g é um vector coluna, que em cada linha contém o último elemento não nulo da linha correspondente da matriz G1. Consiste essencialmente em manipulações de elementos de vectores, todos eles locais. Foi programada em C para optimizar o acesso a estes elementos.
Se bem que todas as operações de transferência de dados sejam locais, a carga computacional não está optimamente distribuída, pois a obtenção da matriz triangular inferior G1, implica maior número de transferências de dados para os nós que armazenam as linhas de maior índice, como aliás pode ser visto em (8-34),

Antes de se passar à analise de desempenho do algoritmo AGPC convém fazer algumas considerações quanto à programação do núcleo deste algoritmo, mais precisamente a equação (8-29). O código SEQ para esta operação já foi apresentado e é o seguinte:

```
/*eq. (8-29)*/
T0=G'*G;
T0=dadd(T0, lambda, T0);
T0 = inv (T0);
U=T0*G'*(W-f);
```

Na última linha existe uma série de operações que podia ser decomposta da seguinte forma:

```
T1 = T0 * G';
U = T1 * (W-f);
```

Esta decomposição permite tirar partido da operação de multiplicação com prévia transposição do operando direito, que é mais rápida que a própria operação de multiplicação, como foi discutido e analisado nos pontos 5.3.2.4.2 e 5.4.3.2. No entanto pode-se efectuar outra decomposição:

```
T1 = G' * (W-f);
U = T0 * T1;
```

Neste caso como W e f são vectores o resultado da sua subtracção é um vector, de modo que a multiplicação de G' por um vector resulta também que T1 será um vector. Agora a multiplicação de T0 por T1 consiste na multiplicação de uma matriz por um vector que é muito mais rápida que a de duas matrizes quadradas, como era o caso na estratégia anterior para obter T1 = T0 * G'. Evitando-se assim a multiplicação de duas matrizes quadradas o tempo de execução é bastante reduzido.

Neste caso o decompositor de equações segue precisamente a última estratégia, porque os parênteses em torno de (W-f), fazem com que esta operação seja primeiro executada, sendo depois multiplicado o resultado desta com G'. No entanto não deve ser assumido que o decompositor automaticamente efectua esta optimização. De facto é da responsabilidade do programador indicar explicitamente recorrendo a parênteses como deve ser efectuada a decomposição em caso de uma ambiguidade deste tipo, sob pena do decompositor não utilizar uma sequência que leve a menos operações elementares. Assim seria possível efectuar a operação:

```
U=T0*G'*(W-f);
```

como:

```
U=(T0*G')*(W-f);
```

ou

```
U=T0*(G'*(W-f));
```

sendo equivalentes, mas a última mais rápida. Por isso em caso de dúvida deve-se recorrer ao uso de parênteses.

8.1.4 Ambiente de ensaio

Antes de apresentar as medidas de desempenho do algoritmo AGPC paralelo vai-se descrever o ambiente utilizado para o ensaio tanto deste caso de teste como do seguinte.

Em termos de ambiente de programação é utilizado o SPAM 1.0 onde a linguagem de programação utilizada é a SEQ 1.0, ambos já descritos nesta tese.

Quanto ao *hardware,* são utilizados os processadores descritos no ponto 2.3, T8, C4 e ADSP21060 ou SH para abreviar em topologias homogéneas. São também ensaiadas topologias heterogéneas compostas de T8 e C4.

As topologias homogéneas consistem em redes de 1 a 3 processadores conectados ponto a ponto:

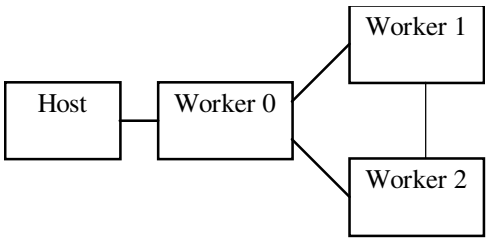


Fig. 8-12: Topologia homogénea

São designadas pela abreviatura do tipo de processador já indicada acima seguida do número de nós dentro de parênteses rectos.

Quanto às topologias heterogéneas são consideradas duas. Uma primeira que consiste num T8 como processador raíz ao qual se adiciona um ou dois C4 conectados ponto a ponto:

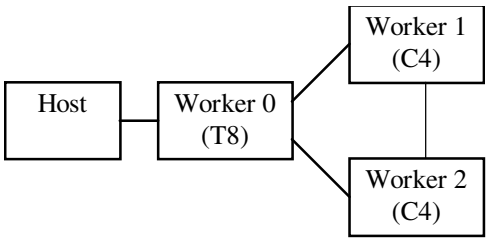


Fig. 8-13: Topologia Heterogénea hetA

Esta é designadas por hetA seguido do número de nós entre parêntese rectos, onde hetA[1] significa apenas um T8, hetA[2] um T8 e um C4 e hetA[3] a topologia da figura acima.

A outra composta apenas por Transputeres, mas com velocidades de relógio diferentes, designada por hetB:

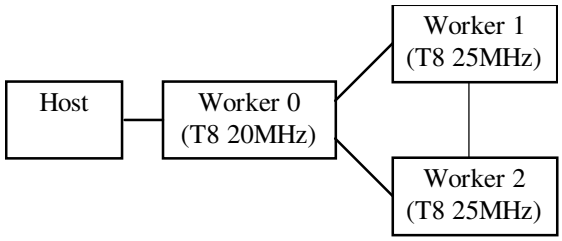


Fig. 8-14: Topologia Heterogénea hetB

Onde hetB[1] representa apenas um T8 - 20 MHz e hetB[3] a topologia da figura acima, onde todos os Transputeres, com excepção do raiz operam a uma velocidade de relógio de 25 MHz.

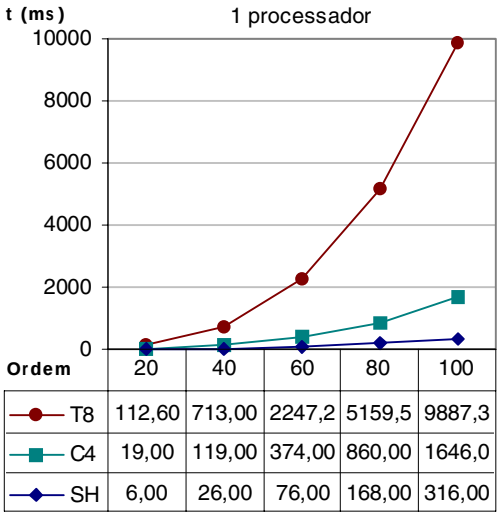
8.1.5 Análise de desempenho do algoritmo AGPC implementado em SEQ 1.0 (SPAM 1.0)

As medidas de desempenho apresentadas neste capítulo, são o fruto do desenvolvimento do algoritmo AGPC paralelo, que foi inicialmente desenhado apenas para arquitecturas T805 homogéneas (Daniel e Ruano, 1996). Em (Ruano e Daniel, 1997) a gestão de memória foi melhorada, o que permitiu aumentar o horizonte de perdição para mais de 50 passos futuros. O desempenho das funções de cálculo foi também melhorado. O algoritmo foi extendido a arquitecturas homogéneas C40 em (Daniel et al, 1997) e (Daniel e Ruano, 1997b). Em (Ruano e Daniel, 1997) e (Daniel e Ruano, 1997a) este algoritmo foi também implementado sobre arquitecturas heterogéneas baseadas em T805s e C40s. Nestas o T805 é usado apenas como elemento de comunicação num nó onde o elemento de cálculo é o C40. Como esta estratégia não conseguiu maior desempenho que arquitecturas C40 homogéneas, principalmente devido à muito estreita largura de banda entre o C40 e o T805, o algoritmo foi desenvolvido para usar o T805 como elemento de cálculo a par do C40, sendo a carga computacional equilibrada entre ambos (Daniel e Ruano, 1999b). No entanto, a mesma limitação de largura de banda já referida, que é ainda inferior à largura de banda entre T805s, a mais baixa nas arquitecturas homogéneas utilizadas, impede resultados eficientes para essa arquitectura heterogénea. Neste último trabalho, o algoritmo foi também implementado sobre arquitecturas homogéneas ADSP21060. Em (Daniel e Ruano, 1999a, 2000) são apresentadas as primeiras versões de um sistema de paralelização automático de algoritmos matriciais. O algoritmo AGPC paralelo é implementado sobre estas.

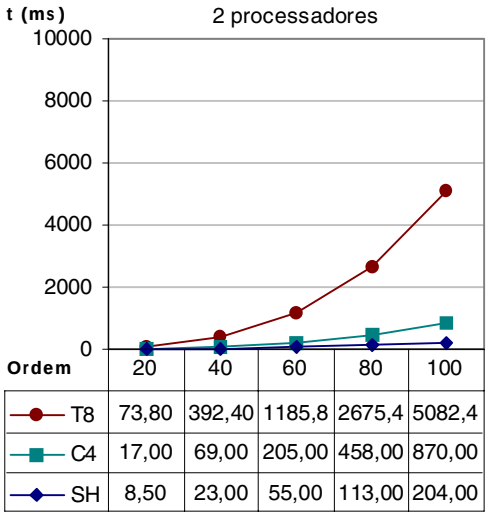
Na versão actual do SPAM, apresentada nesta tese, o ambiente de comunicações foi ainda redesenhado e optimizado. Foi também optimizado o acesso aos elementos individuais das matriz locais, o que se vai traduzir num aumento de desempenho nas funções de cálculo, como já foi descrito nos capítulo anteriores.

Os ensaios seguintes são efectuados para um sistema de 2ª ordem sem ruído colorido presente, e o horizonte de controlo varia de 20 a 100, isto é N_1=1 e N_2={20 ,40, 60, 80, 100}. Como num sistema deste tipo o número de parâmetros a identificar são quatro, assumindo um numerador de primeira ordem, isto é d = 0, as matrizes no estágio RLS são de 4ª ordem. Já no caso do estágio GPC são de ordem $N = N_1 - N_2 + 1$.

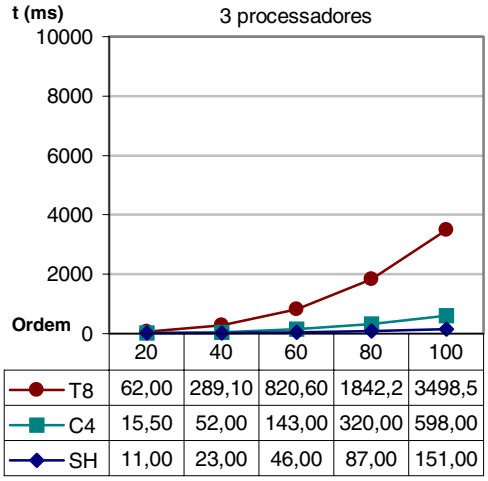
Deste modo o tempo de estimação de parâmetros pelo estágio RLS não é significativo para as medidas de tempos e de eficiência.



Ordem	20	40	60	80	100
T8	112,60	713,00	2247,2	5159,5	9887,3
C4	19,00	119,00	374,00	860,00	1646,0
SH	6,00	26,00	76,00	168,00	316,00



Ordem	20	40	60	80	100
T8	73,80	392,40	1185,8	2675,4	5082,4
C4	17,00	69,00	205,00	458,00	870,00
SH	8,50	23,00	55,00	113,00	204,00



Ordem	20	40	60	80	100
T8	62,00	289,10	820,60	1842,2	3498,5
C4	15,50	52,00	143,00	320,00	598,00
SH	11,00	23,00	46,00	87,00	151,00

Fig. 8-15: Tempos de execução do algoritmo AGPC para redes homogéneas

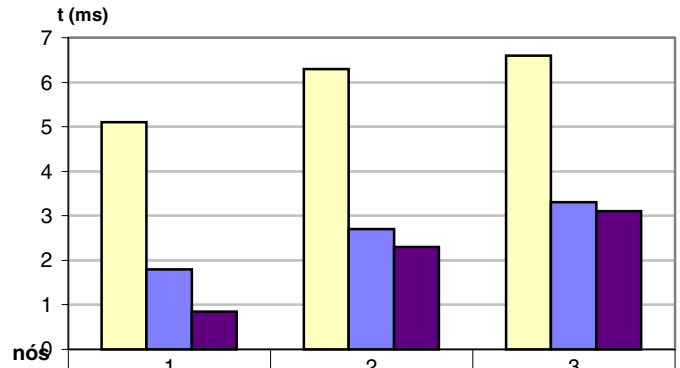
Para as redes T8 é utilizada a versão 2 da multiplicação matricial pois com excepção de horizontes de predição iguais ou inferiores a 20 e a 40, para respectivamente T8[2] e T8[3], este algoritmo tem o melhor desempenho. No entanto utilizando esta versão para esse casos obtém-se tempos de execução um pouco inferiores.

Para redes C4 e SH, para ordem de matrizes 20 a 100, a versão 2 da multiplicação matricial é mais eficiente para um nó ao passo que a versão 1 é mais eficiente para redes com mais nós. Assim a versão 2 é utilizada somente para topologias com um nó, pois é o melhor algoritmo sequencial conhecido (pelo menos no âmbito deste trabalho). No entanto, para redes C4[2] e T8[2] para ordens iguais ou superiores a 100, a versão 2 volta a ser mais rápida, como já foi mostrado na avaliação do desempenho das operações matriciais. Deste modo podia-se obter um menor tempo de execução, e consequentemente uma melhor eficiência para estes casos utilizando a versão 2.

Quanto à multiplicação de matrizes com transposição prévia do 2º operando é habilitada a versão 1, pois é a mais rápida para esta ordem de matrizes, no entanto na sequência de operações do algoritmo AGPC, este caso não ocorre. O desempenha de ambas as versões destas duas operações já foi discutido no ponto 5.4.3.

Finalmente na inversão matricial, pelo método de Gauss-Jordan, as medidas foram efectuadas de modo que não fosse encontrado um elemento *pivot* nulo durante o processo de inversão. Se tal for encontrado a coluna onde se encontra o *pivot* nulo é trocada com uma das seguintes até que se encontre um *pivot* não nulo, provocando um atraso. Assim os tempos medidos são limites mínimos. Os máximos podem variar dependendo do número de trocas de colunas, e são dependentes da própria matriz a inverter.

As figuras acima e as duas seguintes referem-se às redes homogéneas. As redes heterogéneas serão tratadas em separado. Na figura anterior são apresentados os tempos de execução do algoritmo AGPC, ao passo que na seguinte são apresentados os tempos de execução para o estágio de estimação, algoritmo RLS apenas para sistemas de 2ª ordem como já foi mencionado.



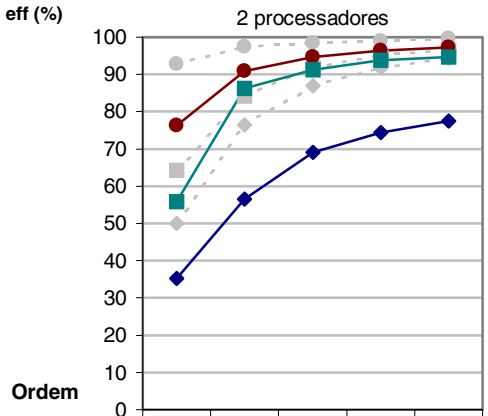
nós	1	2	3
T8	5,10	6,30	6,60
C4	1,80	2,70	3,30
SH	0,85	2,30	3,10

Fig. 8-16: Tempos de execução para o estágio RLS para redes homogéneas

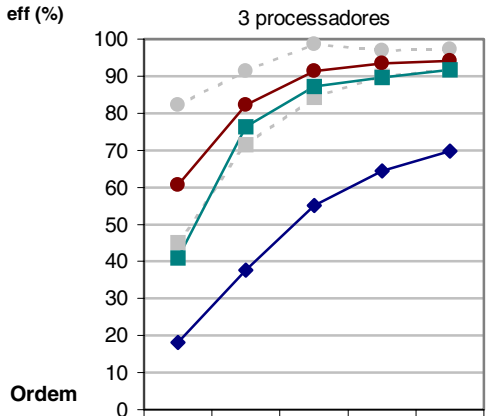
Como se pode observar não existe vantagem em utilizar mais que um processador para operações com matrizes de ordens tão pequenas. Além disso como os tempos de execução são de uma magnitude muito inferior aos do estágio GPC, a eficiência do algoritmo AGPC completo segue a deste último estágio.

A figura seguinte mostra a eficiência para redes com 2 e 3 processadores. Esta eficiência é comparada com as medidas em (Daniel e Ruano, 2000), para uma versão experimental ainda do SPAM 1.0. As redes utilizadas nesse trabalho são idênticas e são referenciadas por T8o, C4o, Sho e hetAo, para as distinguir. Não foram no entanto apresentadas medidas para Sho[3] nesse trabalho.

Esta comparação é interessante pois na versão experimental do SPAM 1.0 a decomposição de equações matriciais em operações elementares não era feita automaticamente pelo decompositor de operações. Assim, na versão experimental do SPAM, sempre que necessário e recorrendo à linguagem C, acedendo pois a um nível de programação inferior à linguagem SEQ, as operações devem ser decompostas em operações elementares e atribuídas a um operando temporário pelo programador.



	20	40	60	80	100
T8o	92,72	97,41	98,50	98,96	99,61
C4o	64,29	84,27	91,80	95,15	96,64
Sho	50,00	76,52	86,99	91,73	94,30
T8	76,29	90,85	94,75	96,42	97,27
C4	55,88	86,23	91,22	93,89	94,60
SH	35,29	56,52	69,09	74,34	77,45



	20	40	60	80	100
T8o	82,19	91,39	98,55	96,84	97,19
C4o	45,00	71,43	84,23	89,92	91,76
Sho					
T8	60,54	82,21	91,28	93,36	94,21
C4	40,86	76,28	87,18	89,58	91,75
SH	18,18	37,68	55,07	64,37	69,76

Fig. 8-17: Eficiência do algoritmo AGPC para redes homogéneas

Já no caso da versão final do SPAM 1.0, apresentada nesta tese, a decomposição é efectuada automaticamente pelo decompositor, facilitando a programação mas perdendo-se em termos

de eficiência. No entanto deve ser indicado que o tempo de computação do algoritmo AGPC apresentado nesta tese é bastante inferior, principalmente devido à optimização do endereçamento dos elementos das matrizes, como já foi indicado no capítulo 5.

Em (Daniel e Ruano, 2000), devido a não serem alocadas ou dealocadas matrizes durante o processamento, pois todas as matrizes necessárias são pré-alocadas, mesmo as temporárias, não se perde tempo na gestão de memória sendo o algoritmo mais eficiente. Isto implica que tal como já foi visto para o caso da operação de adição no ponto 5.4.2, uma maior eficiência em relação à versão final do SPAM 1.0.

Ao ser desenvolvido o decompositor automático de operações para a versão final do SPAM 1.0, foi seguida uma abordagem que facilita a programação, de modo que as rotinas que implementam as operações matriciais passaram a criar uma matriz resultado, deixando o programador de ter de se preocupar com o dimensionamento correcto da matriz resultado. Assim é impossível na versão actual pré-alocar matrizes. Deste modo, na decomposição, cada operação cria uma matriz para o resultado intermédio, sendo esta memória disponibilizada à medida que os resultados intermédios já não são necessários. É assim necessário proceder a esta gestão de memória em cada processador, e como o tempo perdido nesta gestão não é divisível pelo número de processadores, pois a dimensão das matrizes não afecta o desempenho das operações de gestão de memória, leva a uma perda de eficiência. Além disso esta gestão de memória pode levar a uma fragmentação do *heap*, pois operandos com dimensões diferentes são alocados e libertados durante a decomposição, de modo que o tempo de busca de um bloco de memória livre com o comprimento pretendido pode aumentar bastante, o que aumenta a ainda mais a perda de eficiência. Como está patente na analise de desempenho seguinte, as arquitecturas que mais são afectadas são as compostas por ADSP21060.

Por outro lado, em (Daniel e Ruano, 2000) são pré-alocadas matrizes globais temporárias de uso geral, com uma dimensão suficiente para o operando maior que nela será armazenado durante toda a execução do algoritmo AGPC. Para guardar nessas áreas temporárias operandos com dimensões diferentes, o número de linhas e colunas é alterado rapidamente, apenas pela modificação do cabeçalho onde se encontra essa informação, como já foi descrito no ponto 5.1. Deste modo poupa-se na memória utilizada para vários operando temporários, além de se poupar no tempo de gestão de memória.

Na versão final do SPAM 1.0, a decomposição de equações em operações binárias ou unárias, e a necessária alocação de operandos temporários vai fazer com que seja impossível manter

na memória interna do ADS21060 todos os operandos matriciais necessários à decomposição das operações do algoritmo GPC, para matrizes com ordem 100, num só processador. Para redes com 2 e 3 processadores tal não se verifica pois os operandos são distribuídos, sendo possível alocá-los na memória interna. Para o caso de um só processador, pode-se resolver esta limitação alocando toda a aplicação nos 3 MBytes de memória externa, mas como já foi visto no ponto 2.4 o desempenho vai-se degradar, neste caso para 540 ms. Neste cenário é possível atingir-se uma aceleração super-escalar, sendo a eficiência para 2 e 3 processadores respectivamente 132,35 e 119,20.

No entanto, como o executável tem um comprimento inferior a 512 KBytes, é possível alocar o código na memória interna e os dados na memória externa. Assim maximiza-se o desempenho, pois o código e os dados estão alocados em bancos de memória separados, tal como no caso em que são alocados nos dois bancos de memória interna. Também como já foi indicado no capítulo 2, a alocação na memória externa só vai ter uma influência negativa no desempenho se forem comunicados dados entre processadores, o que não é o caso para uma rede com um só nó. Chega-se assim às medidas apresentadas nas figuras acima.

Outra razão para a eficiência medida em (Daniel e Ruano, 2000) ser superior, deve-se ao facto do melhor algoritmo sequencial conhecido à data ser na realidade o mesmo algoritmo que é executado tanto num só processador como numa rede. Tal já não sucede na versão corrente. Como já foi referido, foram desenvolvidas duas versões diferentes de algumas operações matriciais. Quando comparada com a outra, uma versão apresenta um desempenho superior num só nó superior e inferior quando mapeada numa rede. Verifica-se assim que o melhor algoritmo paralelo perde um pouco de eficiência relativamente ao melhor algoritmo sequencial.

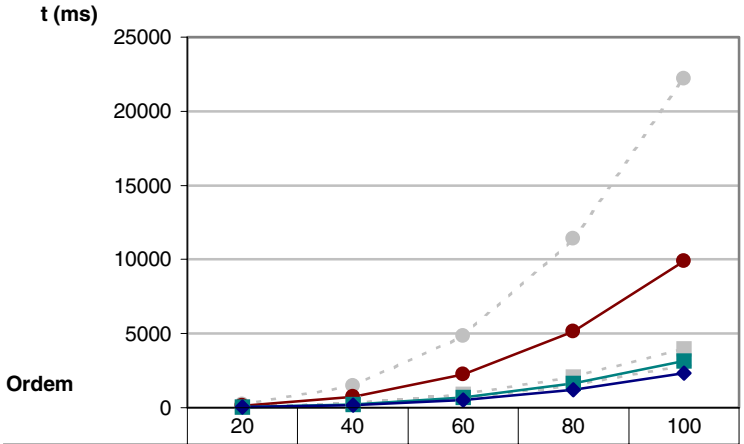
De um modo geral a eficiência das redes T8 é superior à das restantes redes, pois também como já foi visto no ponto 2.4.3, esses processadores podem processar e comunicar em simultâneo sem recurso a estratégias de DMA, o que já não é verdade para outros tipos de processadores utilizados. Assim, como todo o código do SPAM foi desenvolvido com o intuito de ser o mais compatível possível, independentemente do processador alvo, não se recorreu à programação explicita de transferências de dados recorrendo a DMA. No entanto uma boa utilização do co-processador de DMA deverá permitir melhorar a eficiência para redes com nós C4 e ADSP21060. Tal implica no entanto a programação de funções específicas ao nível do assembler, de modo a tirar partido do suporte DMA do processador

respectivo. Para tal, teriam de ser reescritas de acordo, as funções de comunicação utilizadas pelo ambiente de comunicações, e já descritas no ponto 4.5:

putmessage (*c1, p1, l1*)
getmessage (*c2, p2, l2*)

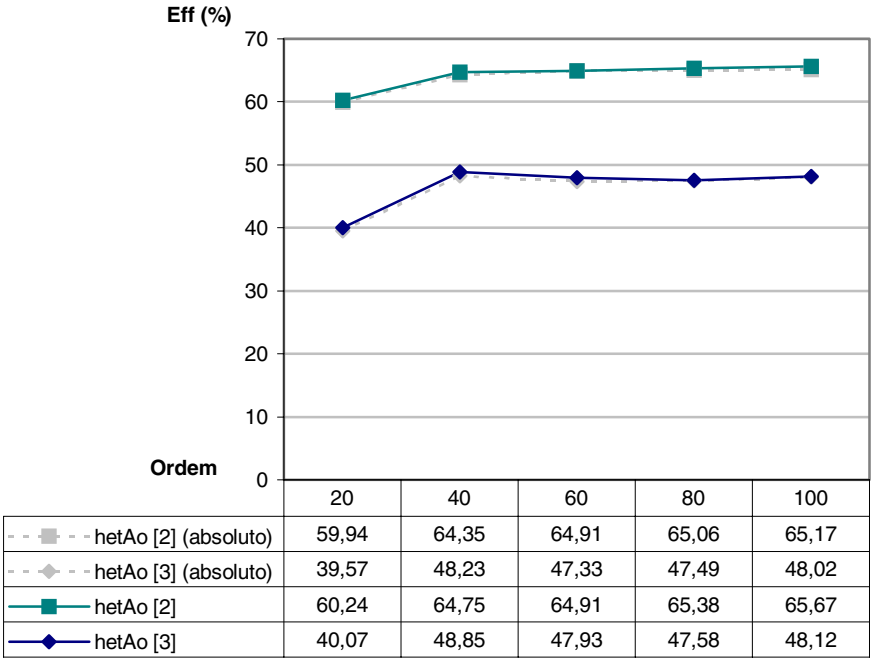
Falta ainda referir o desempenho das arquitecturas homogéneas. No trabalho (Daniel e Ruano, 2000) já referenciado, a arquitectura heterogéneas hetA, composta por um T8 25 Mhz raiz e dos C4s, foi também ensaiada. A Fig. 8-18 mostra a comparação entre o desempenho dessa arquitectura para essa versão do MAPS (hetAo) e para a presente (hetA). Os tempos reduzem-se para a versão actual do SPAM como no caso homogéneo, pelas mesmas razões.

No entanto a medida de eficiência usada em (Daniel e Ruano, 2000) é dada pelo método dependente da distribuição das matrizes por linhas, que como já foi discutido no capítulo 5 não é absoluto. Na Fig. 8-19 estão apresentadas as eficiências para redes com 2 e 3 nós, segundo os dois métodos apresentados nesse capítulo. Pode-se observar que ambas são praticamente idênticas, com diferenças na ordem das decimas percentuais.



Ordem	20	40	60	80	100
hetAo [1]	201,20	1471,70	4841,70	11397,00	22203,00
hetAo [2]	50,10	284,10	892,70	2058,00	3959,00
hetAo [3]	40,30	201,30	650,10	1497,20	2852,80
hetA [1]	112,60	713,00	2247,20	5159,50	9887,30
hetA [2]	37,26	225,39	710,36	1640,30	3148,07
hetA [3]	32,62	156,06	520,59	1209,38	2317,40

Fig. 8-18: Tempos de execução do algoritmo AGPC para redes hetA e hetAo



Ordem	20	40	60	80	100
hetAo [2] (absoluto)	59,94	64,35	64,91	65,06	65,17
hetAo [3] (absoluto)	39,57	48,23	47,33	47,49	48,02
hetAo [2]	60,24	64,75	64,91	65,38	65,67
hetAo [3]	40,07	48,85	47,93	47,58	48,12

Fig. 8-19: Eficiência do algoritmo AGPC para redes hetAo, segundo os dois métodos apresentados no capítulo 5.

A medida absoluta fornece valores de eficiência um pouco inferiores, pois não toma em consideração que a distribuição das matrizes é feita por linhas, mas sim por elementos individuais, como já foi discutido no ponto 5.4.1.3.

Para se obter a eficiência pelo método absoluto, dado pelas equações **(5-34)** e **(5-36)**, para as medições apresentadas em (Daniel e Ruano, 2000), são utilizados os tempos de execução para um nó T8 - 25MHz e um C4 medidos nesse trabalho e que são:

ordem:	20	40	60	80	100
T8 (ms):	201,2	1471,7	4841,7	11397,0	22097,0
C4 (ms):	34,0	207,0	656,0	1516,0	2921,0
C4/T8 = Fd (T8, hetAo)	0,1699	0,1406	0,1355	0,1331	0,1322

tabela 8-3: Comparação entre o desempenho do algoritmo AGCP, rede hetAo num T8 - 25MHz e num C4

Para a relação entre o C4 e o T8 - 25 MHz vai ser utilizada a medida para matrizes de ordem 100: 0,1322 pois, normalmente quanto maior é o volume de dados a processar, maior deverá ser a precisão das medidas. Por outro lado, a base de comparação assume-se igual a um, isto é: Fd(C4) = 1. Então o tempo de execução óptimo deste algoritmo é dado por:

(8-36) $T_o(\text{hetAo}[2]) = \frac{\text{Tempo C4}}{1+0{,}1322}$, pois é composto por um C4 e um T8 - 25MHz e

(8-37) $T_o(\text{hetAo}[3]) = \frac{\text{Tempo C4}}{2+0{,}1322}$, pois é composto por dois C4 e um T8 - 25MHz.

e fazendo a relação entre o óptimo e o efectivamente medido, equação **(5-34**), obtém-se a eficiência apresentada na figura seguinte.

A partir deste ponto as medidas de eficiência apresentadas para arquitectura heterogéneas serão sempre obtidas pelo método absoluto. Quanto à distribuição das matrizes, em (Daniel e Ruano, 2000) o equilíbrio da carga computacional óptimo foi encontrado por teste e erro, ao passo que a corrente versão do SPAM, apresentada nesta tese, é efectuado automaticamente como já descrito no ponto 5.1, usando a relação dada pela velocidade de relógio para hetB:

(8-38) $\text{Fd}(\text{T8 - 20 MHz}, \text{hetB}, [20,100]) = \frac{20\text{MHZ}}{25\text{MHZ}} = 0.8 \approx \frac{\text{Tempo de execução T8 - 25 MHZ}}{\text{Tempo de execução T8 - 20 MHZ}}$

Verifica-se que para ordens de matrizes 20 a 100, a relação entre os tempos de execução, que é inversa da relação entre as velocidades de relógio, é muito próxima de 0.8.

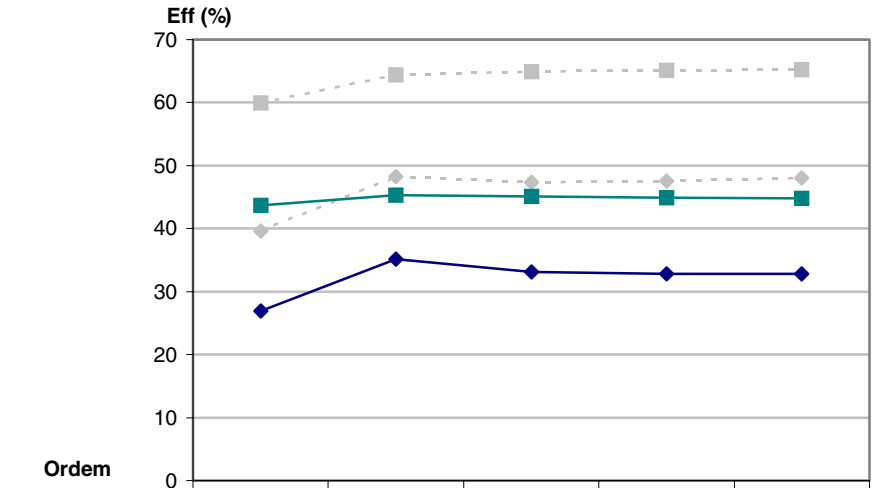
Para hetA foi escolhida a relação para matrizes de ordem 100, que à semelhança de hetA e ao contrário de hetAo, mantém-se muito próxima para ordens de matrizes de 20 a 100:

(8-39) Fd(T8 – 25 MHz, hetA, 100) = 0.1664

Para as medidas de eficiência, foi usada como termo de comparação entre nós diferentes a relação para matrizes de ordem 100, pois deve ser mais precisa, visto que o volume de dados a processar é superior. Os nós mais rápidos têm um valor comparativo de 1, como já foi utilizado para hetAo, em (8-36) e (8-37).

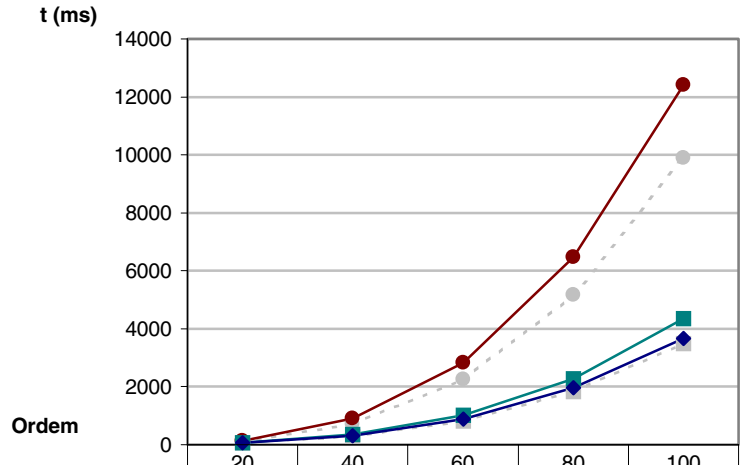
Na Fig. 8-20 têm-se a comparação das eficiências para redes hetA e hetAo. Pode-se observar que a eficiência é um pouco superior para hetAo, pelas razões já indicadas. De um modo geral esta arquitectura é pouco eficientes por culpa da largura de banda entre um T8 e um C4, que é muito inferior mesmo à largura de banda entre dois T8s. Este já não é o caso da outra rede heterogénea, hetB, onde a largura de banda não varia, variando apenas a velocidade de

cálculo, que é assumida proporcional ao relógio. Nesta rede o T8 raiz opera a uma frequência de relógio de 20 MHz, ao passo que os restantes a 25 MHz.



Ordem	20	40	60	80	100
hetAo [2]	59,97	64,39	64,94	65,10	65,20
hetAo [3]	39,58	48,24	47,34	47,50	48,04
hetA [2]	43,72	45,26	45,14	44,95	44,82
hetA [3]	26,88	35,20	33,16	32,82	32,78

Fig. 8-20: Eficiência do algoritmo AGPC para redes hetA e hetAo



Ordem	20	40	60	80	100
T8 [1]	112,60	713,00	2247,20	5159,50	9887,30
T8 [3]	62,00	289,10	820,60	1842,20	3498,50
hetB [1]	141,20	895,40	2823,20	6481,00	12417,90
hetBn [3]	73,80	356,90	1007,70	2283,00	4352,40
hetB [3]	67,00	307,20	887,20	1968,10	3655,10

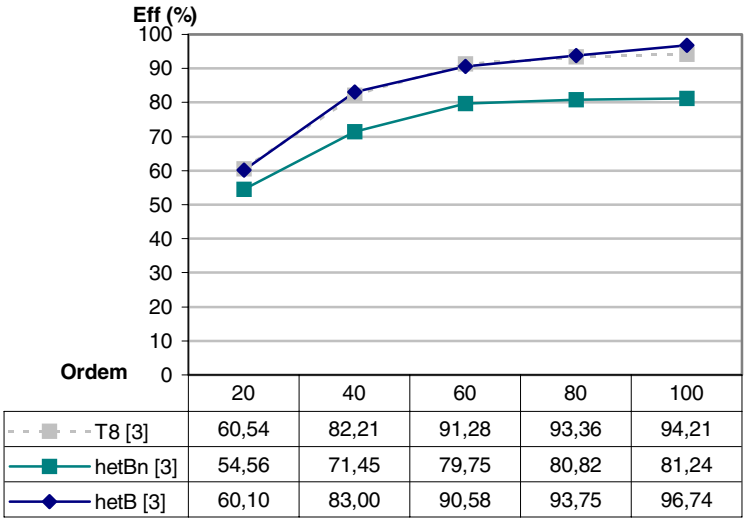
Fig. 8-21: Tempos de execução do algoritmo AGPC para redes hetB com distribuição de linhas igual e optimizada por nó.

A figura anterior mostra os tempos de cálculo quando a carga é distribuída optimamente e quando é distribuída igualmente por cada nó. Para obter estes tempos, referenciados como hetBn, foi desligado o distribuidor de carga óptima do SPAM. Além disso é comparada com a rede homogénea T8. Repare-se que hetB[1] é uma rede com apenas um T8 - 20MHz e T8[1] com um T8 - 25MHz.

Em hetBn[3] a carga computacional é distribuída igualmente pelos 3 nós, ao passo que em hetB[3] é distribuída optimamente pelo mesmo número de nós. Como se pode observar o tempo de execução é reduzido se a carga for distribuída optimamente e automaticamente pelo SPAM.

Quanto à eficiência, se a carga for distribuída optimamente aproxima-se dos valores da rede homogénea T8, como era de esperar, pois a largura de banda entre os Transputers mantém-se, qualquer que seja a frequência do relógio.

A partir deste ponto, em todas as medidas apresentadas para redes heterogéneas, assume-se que a distribuição de carga foi efectuada automaticamente pelo SPAM, se bem que esta possa ser explicitamente definida pelo programador no ambiente AIDA, como indicado no capítulo 7.



Ordem	20	40	60	80	100
T8 [3]	60,54	82,21	91,28	93,36	94,21
hetBn [3]	54,56	71,45	79,75	80,82	81,24
hetB [3]	60,10	83,00	90,58	93,75	96,74

Fig. 8-22: Eficiência do algoritmo AGPC para redes hetB com distribuição de linhas igual e optimizada por nó.

8.2 Treino de redes neuronias

Neste ponto será analisada a implementação em SEQ, bem como o correspondente desempenho paralelo, de um algoritmo de treino de uma rede neuronal artificial, mais

concretamente o treino de um Perceptrão de Multi-Camada. Este tipo de rede foi escolhido como caso de teste do SPAM, devido ao seu vasto campo de aplicação, onde se incluí: Previsão, Controlo, Reconhecimento de padrões, Optimização, etc.. Antes de se descrever o algoritmo de treino serão indicados alguns conceitos e terminologias usadas.

8.2.1 Redes neuronais artificiais

8.2.1.1 Neurónio biológico

As redes neuronais artificiais, têm como inspiração biológica os grupos de neurónios, ou células nervosas, que compõem o cérebro (Strange, 1989). Assim, um neurónio biológico é composto pelo corpo da célula com uma série de prolongamentos curtos – os dentritos – e um prolongamento longo – o axónio ou fibra nervosa – que liga o corpo da célula ao terminal nervoso. A sua representação esquemática pode ser observada na Fig. 8-23. O terminal nervoso de um neurónio pode formar junções com os dentritos ou do corpo de outra célula, ou grupo de células nervosas. Estas junções são designadas sinapses. Por outro lado, os dentritos de um só neurónio, podem formar sinapses com vários outros neurónios. Pode-se assim estabelecer uma rede densamente conectada, onde cada nó pode ter as suas entradas e a sua saída conectadas a muitos nós.

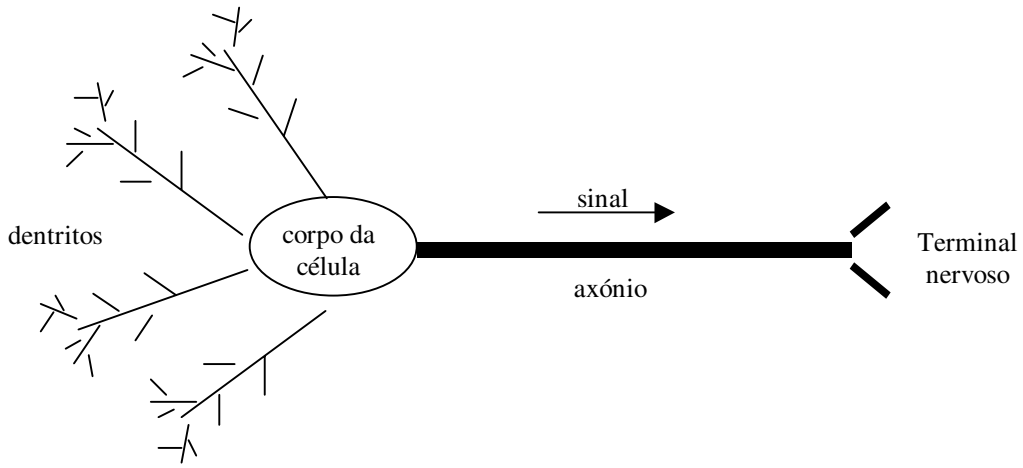


Fig. 8-23 Neurónio biológico

Os neurónios podem gerar sinais eléctricos, que são propagados pelo axónio no sentido do terminal nervoso. O sinal tem dois estados, de forma que a informação é codificada na frequência do sinal. A influencia de cada sinapse, para o neurónio receptor, depende de vários factores, e pode ser considerado como um peso associado ao sinal de entrada. Este peso varia

ao longo do tempo; é assumido que este processo esta relacionado com a capacidade de aprendizagem do cérebro.

8.2.1.2 Modelo do neurónio

Tomando em consideração o neurónio biológico, assumindo que o efeito de cada sinapse é independente de todas as outras sinapses, bem como da actividade do neurónio, e que este é visto como um integrador dos sinais présinapticos, um modelo do neurónio pode ser definido (Kohonen, 1988) do seguinte modo:

$$(8\text{-}40)\quad \frac{d\mathbf{out}_i}{dt} = \sum_{j=1}^{k} \mathbf{X}_{i,j}(\mathbf{out}_j) - g(\mathbf{out}_i) = \sum_{j\in\{J\}} \mathbf{X}_{i,j}(\mathbf{out}_j) - g(\mathbf{out}_i)$$

onde **out** é um vector linha que contém a frequência de oscilação de cada neurónio. $\mathbf{X}_{i,j}(\ )$ representa a influência da entrada $\mathbf{in}_j$, onde j pertence a um conjunto {J} de entradas no neurónio i, e g ( ) é uma função de custo, não linear para o neurónio i, de modo a tomar em conta efeitos de saturação.

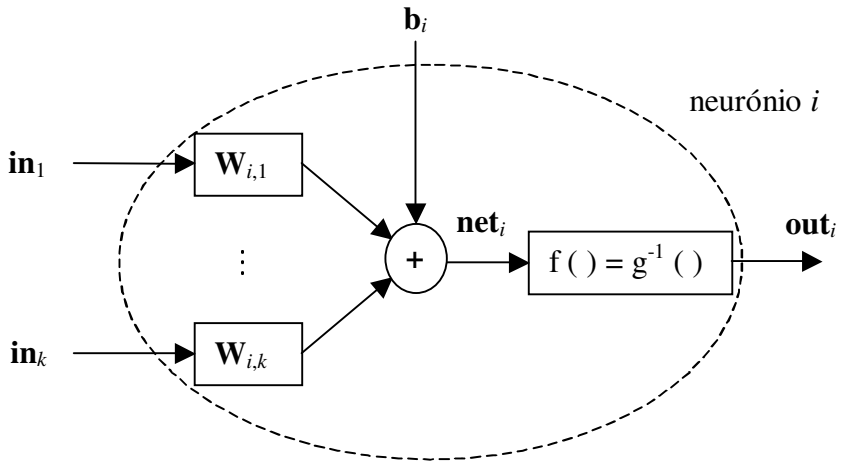


Fig. 8-24 Neurónio artificial

Se se considerar apenas a resposta em estado estacionário da equação (8-40), como é o caso da maioria dos modelos de redes neuronais, e que pode ser obtido igualando esta equação a 0 e assumindo que existe inversa de g ( ), $\mathbf{out}_i$ pode ser dado por:

$$(8\text{-}41)\quad \mathbf{out}_i = g^{-1}\left(\sum_{j\in\{J\}} \mathbf{X}_{i,j}(\mathbf{out}_j) + \mathbf{b}_i\right)$$

onde $\mathbf{b}_i$ é o *bias* para o neurónio *i*, que pode ser considerado como uma entrada constante e igual a 1. Considerando:

(8-42) $\mathbf{X}_{i,j}(\mathbf{out}_j) = \mathbf{out}_j \mathbf{W}_{i,j}$

(8-43) $\mathbf{net}_i = \sum_{j \in \{J\}} \mathbf{out}_j \mathbf{W}_{i,j} + \mathbf{b}_i = \mathbf{out}_{\{J\}} \mathbf{W}_{i,\{J\}} + \mathbf{b}_i$,

onde **W** se refere a uma matriz de pesos, e considerando a função $g^{-1}(\,) = f(\,)$, normalmente chamada de saída ou função de activação, o modelo simplificado do neurónio é dado por:

(8-44) $\mathbf{out}_i = f(\mathbf{net}_i)$

net é normalmente denominada de entrada total do neurónio. Tipicamente f é uma função tal que satura para valores altos e baixos de entrada, e que tem uma resposta aproximadamente linear entre estes dois limites.
Pode-se incorporar o vector a matriz de *bias* na matriz de pesos, usando:

(8-45) $\begin{bmatrix} \mathbf{out}'_{\{j\}} = \lfloor \mathbf{out}_{\{j\}} & 1 \rfloor \\ \mathbf{W}' = \begin{bmatrix} \mathbf{W} \\ \mathbf{b}^T \end{bmatrix} & \end{bmatrix}$

e assim:

(8-46) $\mathbf{net}_i = \mathbf{out}'_{\{J\}} \mathbf{W}'_{i,\{J\}}$

Dependendo também do modelo de neurónio assumido, o valor admissível da saída deste pode ser um valor real ilimitado, ou um valor real nos intervalos [-1, 1] ou [0, 1], ou ainda um valor discreto tal como {-1, 1} ou {0, 1}.

8.2.1.3 Mecanismo de aprendizagem

Um mecanismo de aprendizagem pode ser visto como o processo em que a rede é treinada para que a sua resposta **O**, a uma matriz de entrada **I**, seja o mais próximo possível de uma matriz de treino **T**.

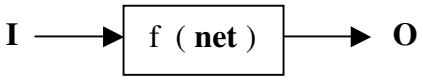


Fig. 8-25 Rede neuronal vista como um dispositivo de entrada saída

Ou seja, encontrar valores para os pesos (**W**) e *bias* (**b**) presentes na rede, de modo que o erro dado por:

(8-47) $\mathbf{E} = \mathbf{T} - \mathbf{O}$

Seja minimizado, segundo um determinado critério. O critério mais usado é a soma dos quadrados dos erros:

(8-48) $\Omega = \sum_{j \geq 0} \left(\mathbf{E}^{T} \mathbf{E} \right)_{j,j}$

A dimensão das matrizes **E**, **T** e **O** é $(m \times n_{out})$ e da matriz **I** é $(m \times n_{in})$, onde m é o número de padrões no conjunto de treino, n_{in} é o número de entradas na rede e n_{out} o número de saídas.

8.2.1.4 Perceptrão multi-camada

Este tipo de rede neuronal, também designado por MLP, é constituído por neurónios agrupados em camadas. Estas são classificadas como de entrada, de saída ou escondidas, dependendo do modo como comunicam dados com o meio externo. Os neurónios em camadas adjacentes estão completamente conectados ao passo que normalmente não são usadas sinapses entre camadas não adjacentes. As conexões propagam sinais unidireccionalmente, no sentido da entrada para a saída do perceptrão, como ilustra a Fig. 8-26. A topologia deste tipo de redes neuronais é indicada especificando o número de neurónios em cada camada, começando na camada de entrada.

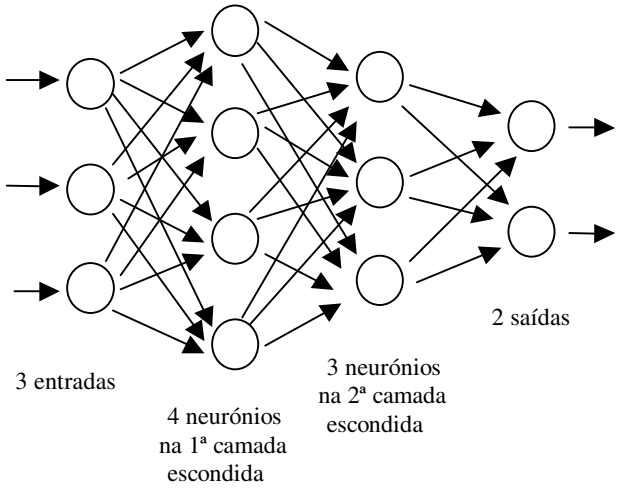


Fig. 8-26 MLP (3, 4, 3, 2)

A operação de um perceptrão deste tipo inicia-se com a chegada de dados à camada de entrada, a qual actua como uma área de armazenamento temporário dos dados que chegam à rede. Para cada neurónio na camada seguinte é calculada a sua entrada total de acordo com a equação (8-43), sendo esta aplicada a uma função de saída (8-44), obtendo-se o vector linha de saída para a 1ª camada escondida e que será a entrada da seguinte camada. Esta operação é repetida até se obter o vector de saída correspondente à última camada, isto é a camada de saída. Esta operação é normalmente designada por *recall*. A função de saída mais vulgarmente utilizada é dada por:

$$\text{(8-49)} \quad \mathbf{out}_i = \frac{1}{1+e^{-\mathbf{net}_i}}$$

Para aplicações de aproximação de funções, como é o caso da maioria das aplicações de controlo a função de activação do neurónio de saída é linear

Consoante a actualização dos pesos se faça amostra a amostra, ou mediante a propagação de um conjunto de amostras, o treino é considerado *on-line*, ou *off-line* respectivamente. Neste último caso, quantidades tais como entradas, saídas e erros, são matrizes onde a linha *l* indica o padrão *l*, num conjunto de *m* padrões. A aprendizagem segundo este esquema de apresentação de múltiplos padrões à rede é chamada de aprendizagem *off-line*, *epoch, batch* ou ainda apenas treino, ao passo que aprendizagem, isto é actualização de pesos sempre que cada padrão é apresentado à rede, é chamada de aprendizagem *on-line,* instantânea, por padrões ou apenas adaptação.

Quando se dispõe de um conjunto de padrões de treino, previamente obtidos, podem ser utilizados ambos os métodos. No entanto, em termos de desempenho, os métodos *off-line* superam amplamente os métodos *on-line*.

Deve ainda ser realçado que existe uma classe de algoritmos *on-line* que aplica, amostra a amostra, algoritmos *off-line* a uma janela deslizante, composta pelos valores correntes e por um conjunto de valores previamente aplicados à rede. Esta classe de algoritmos atinge um desempenho próximo dos métodos *off-line*. Exemplos da sua aplicação podem ser consultados em (Asirvadam, 2002; Ferreira, et al. 2001).

Também em termos do esquema de particionamento de matrizes utilizado no SPAM, i.e. conjuntos de linhas consecutivas, a aprendizagem *off-line* permite desenvolver mais facilmente algoritmos com um desempenho paralelo superior, pois como já foi indicado quanto mais linhas existirem nos operandos maior é a eficiência. Assim, no caso do algoritmo de treino cuja implementação é apresentada nesta tese, a eficiência é superior se os m padrões forem apresentados de uma só vez, quando comparado com a apresentação de um padrão l de cada vez, m vezes. A notação utilizada é a seguinte:

q	Número de camadas do MLP (sendo 1 a camada de entrada e q a de saída).
$\mathbf{k}$	Vector com q elementos, (o elemento z indica o número de neurónios na camada z).
$\mathbf{Net}^{(z)}$	Matriz ($m \times \mathbf{k}_z$) de entrada dos neurónios da camada z (não definida para $z = 1$)
$\mathbf{O}^{(z)}$	Matriz ($m \times \mathbf{k}_z$) de saída dos neurónios da camada z (a matriz de entrada da rede **in** é igual $\mathbf{O}^{(1)}$)
$\mathbf{T}$	Matriz ($m \times \mathbf{k}_q$) de saída desejada ou de Treino.
$\mathbf{E}$	Matriz ($m \times \mathbf{k}_q$) de erro ($\mathbf{E} = \mathbf{T} - \mathbf{O}^{(q)}$).
$\mathbf{F}^{(z)}(\ )$	Função de saída para os neurónios da camada z. Retorna uma matriz embora aplicada elemento a elemento, isto é: $\mathbf{O}^{(z)}_{i,j} = \mathbf{F}^{(z)}_{i,j}(\mathbf{Net}^{(z)}) = \mathbf{F}^{(z)}(\mathbf{Net}^{(z)}_{i,j})$
$\mathbf{W}^{(z)}$	Matriz ($\mathbf{k}_z$+1) × $\mathbf{k}_{z+1}$ de pesos entre as camadas z e z+1. O valor de bias associado a cada neurónio da camada z+1 pode ser reflectido como um peso normal, o qual conecta um neurónio adicional (o neurónio $\mathbf{k}_z$+1) na camada z, e que pode ter uma saída fixa e igual a 1. $\mathbf{W}_{i,j}$ é o peso que conecta o neurónio i na camada z, com o neurónio j na camada z+1. $\mathbf{Net}^{(z+1)} = \mathbf{O}^{(z)}\mathbf{W}^{(z)}$
$\mathbf{J}^{(z)}$	Matriz Jacobiana ($\mathbf{k}_z$+1) × $\mathbf{k}_{z+1}$. Derivadas das saídas em relação aos pesos.
$\mathbf{g}^{(z)}$	Vector gradiente ($m \times \mathbf{k}_q$)

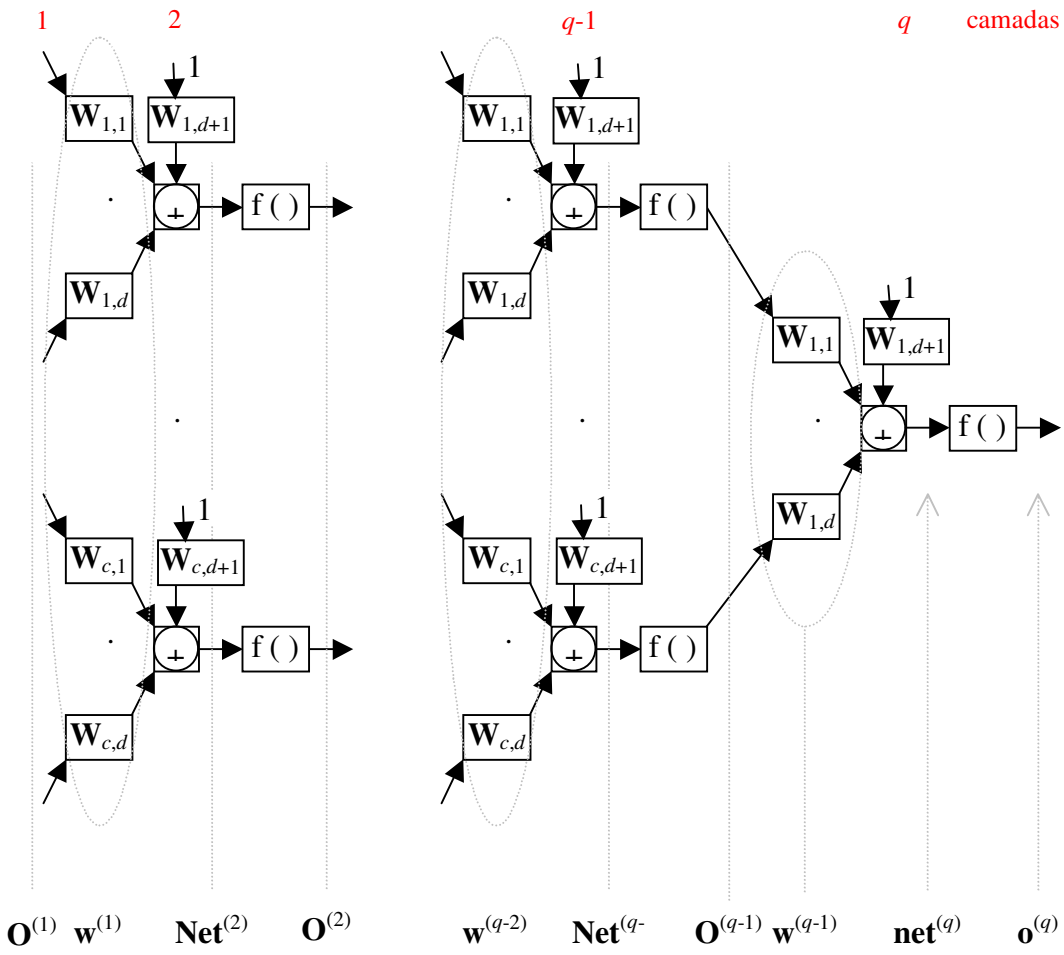


Fig. 8-27 Matrizes associadas a um MLP

Pode ser conveniente representar vectorialmente a matriz de pesos $\mathbf{W}^{(z)}$, isto é como um arranjo das colunas num vector coluna:

$$(8\text{-}50)\quad \mathbf{w}^{(z)} = \begin{bmatrix} \mathbf{W}^{(z)}.,1 \\ \vdots \\ \mathbf{W}^{(z)}.,\mathbf{k}_{z+1} \end{bmatrix}$$

e o vector de pesos completo da rede:

$$(8\text{-}51)\quad \mathbf{w} = \begin{bmatrix} \mathbf{w}^{(q\text{-}1)} \\ \vdots \\ \mathbf{w}^{(1)} \end{bmatrix}$$

Para terminar a introdução ao MLP, falta referir que usualmente, os pesos das sinapses são associados com valores reais sem limites.

8.2.2 Algoritmo de treino

O algoritmo de treino que será implementado, foi desenvolvido por (Ruano, 1992). O desempenho da implementação em SEQ, será comparado com trabalhos prévios na área (Ruano, 1992; Ruano et al, 1992), numa secção posterior.

A topologia do perceptrão utilizado é **k =** (n_{in}, x_1, x_2, n_{out} = 1), isto é, a camada de saída tem apenas um neurónio, como mostra a Fig. 8-28

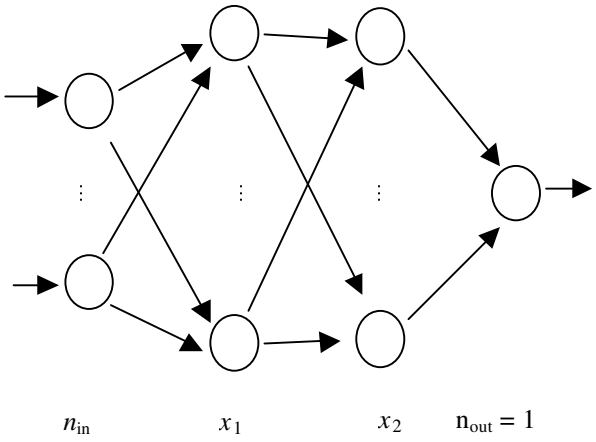


Fig. 8-28 MLP (n_{in}, x_2, x_1, 1)

.

Desta forma, as matrizes, **E**, **T**, **O**, respectivamente erro, treino e saída da rede, cuja dimensão é ($m \times n_{out}$), degeneram em vectores coluna com m elementos, visto que o número de saídas do perceptrão é n_{out} = 1. Assume-se que o neurónio de saída tem uma função de activação linear. A equação de erro será então:

(8-52) $\mathbf{e} = \mathbf{t} - \mathbf{o}$

O critério de minimização normalmente utilizado é:

(8-53) $$\Omega = \frac{1}{2}(\mathbf{e}^T\mathbf{e}) = \frac{\|\mathbf{e}\|_2^2}{2}$$

A regra de aprendizagem mais normalmente utilizada é o algoritmo de retro-propagação de erros (*error back-propagation*), introduzido por (Rumelhart, et. al, 1986). A actualização do vector de pesos, para a iteração η, é dada por:

(8-54) $\mathbf{w}\,[\eta+1] = \mathbf{w}\,[\eta] - \alpha\,\mathbf{g}\,[\eta]$

onde α é denominada a taxa de aprendizagem.
O vector gradiente, para um vector de peso com *p* elementos, é dado por:

$$(8\text{-}55)\quad \mathbf{g} = \begin{bmatrix} \dfrac{\partial \Omega}{\partial w_1} \\ \vdots \\ \dfrac{\partial \Omega}{\partial w_p} \end{bmatrix} = -\mathbf{J}^{\mathrm{T}}\mathbf{e}$$

Nesta última equação, **J** refere-se à matriz Jacobiana, dada por:

$$(8\text{-}56)\quad \mathbf{J} = \begin{bmatrix} \dfrac{\partial o_i}{\partial w_j} \end{bmatrix}$$

Este algoritmo sofre de deficiências várias, tais como não garantia de convergência e uma muito baixa taxa de convergência.
Este critério será substituído por um novo, introduzido por (Ruano et al, 1991), que acelera o processo de aprendizagem, no sentido de reduzir o número de iterações necessárias, para um mesmo algoritmo de treino. Assim rescrevendo (8-51) como:

$$(8\text{-}57)\quad \mathbf{w} = \begin{bmatrix} \mathbf{w}^{(q\text{-}1)} \\ \begin{bmatrix} \mathbf{w}^{(q\text{-}2)} \\ \vdots \\ \mathbf{w}^{(1)} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix}$$

onde o vector **u** contém os pesos (lineares) e *bias* que são associados ao neurónio de saída e **v** todos os outros pesos (não lineares) e *bias* associados às restantes camadas do MLP. Convém

relembrar que o vector **b** ou *bias* pode ser considerado constante e igual a **1**. Visto que a camada de saída é linear, pode-se expressar o vector de saída como:

(8-58) $$\mathbf{o} = \mathbf{o}^{(q)} = \left[\mathbf{O}^{(q-1)} \quad \mathbf{b}\right]\mathbf{u} = \left[\mathbf{O}^{(q-1)} \quad \mathbf{1}\right]\mathbf{u} = \mathbf{A}\mathbf{u}$$

Onde **A** é uma matriz de dimensão ($m \times \mathbf{k}_q$+1) que contém a saída da última camada escondida e uma coluna de uns para tomar em consideração o *bias* desta última camada. Usando, agora (8-58) e substituindo em (8-53), obtém-se:

(8-59) $$\Omega = = \frac{\left\|\mathbf{t} - \mathbf{A}\mathbf{u}\right\|_2^2}{2}$$

Para minimizar (8-59), para qualquer valor dos parâmetros não lineares **v**, em ordem a **u**, pode-se usar:

(8-60) $$\hat{\mathbf{u}} = \mathbf{A}^{+}\mathbf{t}$$

onde $\mathbf{A}^{+}$ indica a pseudo inversa da matriz **A**. Aplicando (8-60) em (8-59) obtém-se um novo critério de aprendizagem que depende apenas dos pesos não lineares **v**:

(8-61) $$\Psi = \frac{\left\|\mathbf{t} - \mathbf{A}\hat{\mathbf{u}}\right\|_2^2}{2}$$

Para minimizar (8-61), será utilizado o método Levenberg-Marquardt (Gill et al, 1981) ou LM, visto que foi demonstrado em (Ruano, 1992) que o algoritmo de retro propagação de erros é lento a convergir e que é difícil seleccionar valores apropriados para o parâmetro de aprendizagem. Para usar o método LM, será necessário calcular uma matriz Jacobiana adequada a (8-61).

(8-62) $$\mathbf{J}\big|_{\hat{\mathbf{u}}=\mathbf{A}^{+}\mathbf{t}} = \left[\mathbf{A} \quad \boldsymbol{J}\right]$$

Em (Ruano et al, 1991) é demonstrado que essa matriz é dada por ***J*** e é definida por (8-62)**.**

Assim o algoritmo de treino proposto em (Ruano et al., 1992), é o seguinte:

algoritmo 8-7:

Desde η = 1 até número de iterações pretendidas

#1 Calcular: $\mathbf{A} = [\mathbf{O}^{(q-1)}\ 1\ ]$	(algoritmo 8-8)
#2 $\hat{\mathbf{u}} = \mathbf{A}^{+}\mathbf{t}$; $\mathbf{u} = \mathbf{W}^{(q-1)} = \hat{\mathbf{u}}$	(algoritmo 8-12)
#3 Calcular: $\mathbf{o} = \mathbf{o}^{(q)}$	(algoritmo 8-9)
#4 $\mathbf{e} = \mathbf{t} - \mathbf{o}$	
#5 Calcular Jacobiano **J** e gradiente **g**	(algoritmo 8-10)
#6 Calcular $\Delta\mathbf{w} = \mathbf{w}\ [\eta + 1]\ \mathbf{w}\ [\eta]$	(algoritmo 8-11)

Onde as matrizes de pesos, **W** podem ser inicializadas com zero ou com valores aleatórios. Um tratamento mais aprofundado da inicialização dos parâmetros, e dos critérios de convergência de aprendizagem, pode ser consultada em (Ruano, 2002). Além disso, as amostras e os conjuntos de treino devem ser escalados, de modo que os seus valores se situem entre [-1, 1].

Se bem que em termos de ensaios de desempenho do algoritmo de treino, seja explicitamente especificado um dado número de iterações, e medido o tempo necessário para as completar, não se trata de uma terminação correcta do processo de treino. Um critério de convergência que pode ser utilizado consiste na verificação em simultâneo das seguintes condições (Ruano, 2002), em cada iteração η:

(8-63) $\psi\ [\eta\text{-}1] - \psi\ [\eta] < \theta\ [\eta]$

(8-64) $\| \mathbf{w}[\eta\text{-}1] - \mathbf{w}[\eta] \|_2 < \sqrt{\tau_f}\ (1 + \|\mathbf{w}[\eta]\|_2)$

(8-65) $\| \mathbf{g}[\eta] \|_2 \leq \sqrt[3]{\tau_f}\ (1 + \psi\ [\eta]^2)$

onde τ_f representa um número de dígitos correctos na função objectivo, obtendo-se assim uma medida de precisão absoluta:

$$\theta\ [\eta] = \tau_f * (1 + \psi\ [\eta])$$

Este critério de convergência não foi utilizado, pois neste trabalho o objectivo é determinar o desempenho do ambiente SPAM 1.0 na paralelização automática do algoritmo.

O algoritmo para a operação de *recall* até à penúltima camada é:

algoritmo 8-8:

desde $z = 1$ até q-2

#1 $\mathbf{Net}_{l,.}^{(z+1)} = \left[\mathbf{O}_{l,.}^{(z)} \quad 1\right]\mathbf{W}^{(z)}$

#2 $\mathbf{O}_{l,.}^{(z+1)} = \mathbf{f}^{(z+1)}\left(\mathbf{Net}_{l,.}^{(z+1)}\right)$

Já a operação de *recall* para última camada é dada por:

algoritmo 8-9:

#1 $\mathbf{Net}^{(q)}{}_{l,.} = \mathbf{A} \,.\, \mathbf{u}$;

#2 $\mathbf{o}_l = \mathbf{O}_{l,.}{}^{(q)} = \mathbf{f}^{(q)}(\mathbf{Net}^{(q)}{}_{l,.})$ "ou como $\mathbf{f}^{(q)}$ é a função identidade basta: $\mathbf{o}_l = \mathbf{Net}^{(q)}{}_{l,.}$"

No passo #2, para o caso de um MLP com um só neurónio na camada de saída e considerando uma função de activação linear, o valor de saída $\mathbf{o}_l$ é igual a $\mathbf{Net}^{(q)}{}_{l,.}$

Cálculo da matriz Jacobiana e do vector gradiente:

algoritmo 8-10:

desde $z = q$ até 2

#1 se $z = q$

#2 $\mathbf{d} = \mathrm{f}'\left(\mathbf{Net}^{(q)}{}_{l,.}\right) = 1$ "pois é a derivada de f (x) = x"

#3 senão

#4 $\mathbf{d} = \mathbf{d}.\left(\mathbf{W}^{(z)}{}_{1..\mathbf{k}(z),.}\right)^{\mathrm{T}} \otimes \mathrm{f}'\left(\mathbf{Net}^{(z)}{}_{l,.}\right)$

#5 desde j=1 até $\mathbf{k}$ (z)

#6 $\mathbf{J}^{(z-1)}{}_{.,j} = \mathbf{d}_j \left[\mathbf{O}^{(z-1)}{}_{l,.} \quad 1\right]$

O passo #2, para o caso de um MLP com um só neurónio na camada de saída e considerando uma função de activação linear, tem como derivada 1.

No passo #4 a o símbolo $\otimes$ indica a multiplicação dos elementos de uma tabela pelos elementos correspondentes de outra tabela, distingindo-se assim do produto vectorial.

O método de Levenberg-Marquardt de acordo com (Fletcher, 1987), está implementado da seguinte forma:

algoritmo 8-11:

#1 $$r = \frac{\mathbf{e}_{old}^{\mathrm{T}}\mathbf{e}_{old} - \mathbf{e}^{\mathrm{T}}\mathbf{e}}{\mathbf{e}_{old}^{\mathrm{T}}\mathbf{e}_{old} - \mathbf{e}_p^{\mathrm{T}}\mathbf{e}_p}$$

#2 se $r < 0.25$ $\upsilon = \upsilon * 4$
senão se $r > 0.75$ $\upsilon = \upsilon / 2$

#3 se $r > 0$ $\mathbf{e}_{old} = \mathbf{e}$;

#4 $p = (\mathbf{J}^T \mathbf{J} + \upsilon \mathbf{I}_s)^{-1} \mathbf{g} = (\mathbf{J}^T \mathbf{J} + \upsilon \mathbf{I}_s)^{-1} \mathbf{J}^T \mathbf{e} = \begin{bmatrix} \mathbf{J} \\ \mathbf{I}_r \sqrt{\upsilon} \end{bmatrix}^+ \begin{bmatrix} \mathbf{e}_{old} \\ \mathbf{0} \end{bmatrix}$

#5 $\hat{\mathbf{v}} = \mathbf{v} - p$

#6 $\mathbf{e}_p = \mathbf{e}_{old} - \mathbf{J}\, p$

onde ν é inicializado com 1, $\mathbf{I}_s$ é a matriz identidade $m \times m$, $\mathbf{I}_r$ é a matriz identidade $n \times n$, e $\mathbf{0}$ representa o vector zero $n \times 1$.

A resolução de sistemas de equações lineares indeterminados, dada pelo método dos mínimos quadrados, passo #4 do algoritmo acima, é efectuada utilizando uma decomposição QR, por razões de robustez numérica. Assim sendo, para o sistema na forma:

$$(8\text{-}66) \quad p = \begin{bmatrix} \mathbf{J} \\ \mathbf{I}_r \sqrt{\upsilon} \end{bmatrix}^+ \begin{bmatrix} \mathbf{e}_{old} \\ \mathbf{0} \end{bmatrix} = \mathbf{C}^+ \mathbf{b}$$

onde se assume que $\mathbf{C}$ é $(n + m) \times n$ e $m \geq n$. Pode-se decompor $\mathbf{C}$ num produto de duas matrizes $\mathbf{Q}$ e $\mathbf{R}$, tal que:

$$(8\text{-}67) \quad \mathbf{C} = \mathbf{QR} = \begin{bmatrix} \mathbf{Q}_1 & \mathbf{Q}_2 \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{bmatrix}$$

onde $\mathbf{Q}_1$ têm tamanho $(n + m) \times n$, $\mathbf{Q}_2$ $(n + m) \times n$, $\mathbf{R}_1$ é uma matriz triangular superior $n \times n$ e $\mathbf{0}$ é $m \times m$.

Deste modo o sistema é resolvido em duas fases. Primeiro a fase de triangularização, onde se obtém $\mathbf{R}_1$ e também:

$$(8\text{-}68) \quad \mathbf{y} = \mathbf{Q}_1^T \mathbf{b}$$

E depois a fase de solução onde se obtém $\mathbf{x}$:

(8-69) $\mathbf{x} = \mathbf{R}_1^{-1}\mathbf{y}$

Este algoritmo pode ser implementado na forma de uma função:

algoritmo 8-12:

x = lms (**C**, **b**)
    [$\mathbf{R_1}$, **y**] = qr (**C**, **b**)
    x = bsubst ($\mathbf{R_1}$, **y**)

Onde as duas fases estão separadas em funções diferentes, sendo a solução dada por retro substituição das variáveis ou *backsubstitution*.

A fase de triangularização, função qr, pode ser implementada de diversas formas. Aqui utilizou-se o algoritmo de *householder* (Stewart, 1973):

algoritmo 8-13:

[**R**, **y**] = qr (**R**, **y**)
desde i = 1 até mínimo de {m, n}
    $\mathbf{x} = \mathbf{R}_{i..m,\, i}$
    [**h**, s] = hhv (**x**)
    $\mathbf{R}_{i,\, i} = -s$
    se $i < n$
        $\mathbf{r} = \mathbf{h}^T \mathbf{R}_{i..m,\, i+1..n}$
        $\mathbf{R}_{i..m,\, i+1..n} = \mathbf{R}_{i..m,\, i+1..n} - \mathbf{h}\,\mathbf{r}$
    $l = \mathbf{h}^T \mathbf{y}_{i..m,1}$
    $\mathbf{y}_{i..m,1} = \mathbf{y}_{i..m,1} - \mathbf{h}\,l$

onde a hhv (**x**), calcula o vector de *householder* e define-se como:

algoritmo 8-14:

[**h**, s] = hhv (**x**)
    $l = ||\mathbf{x}||_2$
    s = sinal ($\mathbf{x}_1$) . l
    $r = l + |\,\mathbf{x}_1\,|$
    $\mathbf{x}_1$ = sinal($\mathbf{x}_1$) . r
    $\mathbf{h} = \dfrac{\mathbf{x}}{\sqrt{l.r}}$

onde a função sinal retorna -1 ou 1, dependendo do argumento de entrada ser negativo, ou não.

Para demonstrar a convergência deste algoritmo, pode-se treinar uma rede para determinar a concentração de um dado composto químico dado o seu pH. Aliás este exemplo é também apresentado em (Ruano, 1992), sendo comparados vários algoritmos de treino.

O pH, que é uma medida da actividade dos iões de hidrogénio numa solução, pode ser relacionado com a concentração x de um dado químico. O controlo do pH não é trival, pois a relação anterior não é linear, causando grandes variações na dinâmica do processo. Uma estratégia usada em alguns métodos de controlo de pH, para tornear os problemas causados pela não linearidade, consiste em utilizar a concentração como saída em vez do pH, linerizando o controlo (Aström e Wittenmark, 1989).

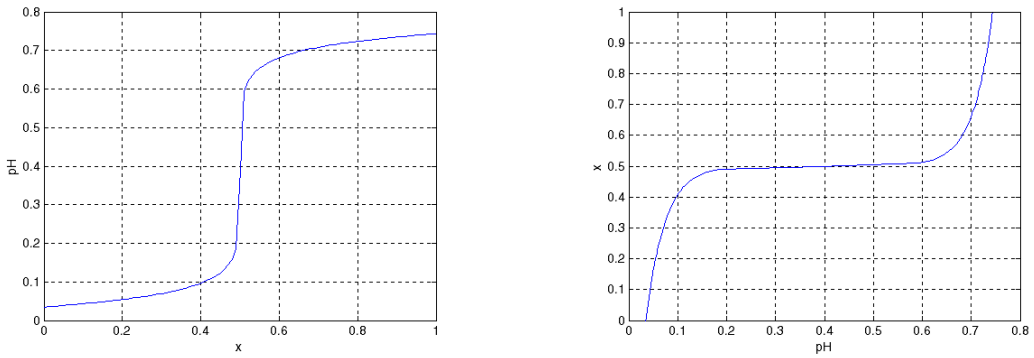


Fig. 8-29: Relação entre a concentração x e o pH, e sua inversa

Na figura acima têm-se um exemplo da relação entre o pH e a concentração, segundo:

$$\text{(8-70)}\quad \text{pH} = -\frac{\log\left(\sqrt{\frac{y^2}{4}+10^{-14}}-\frac{y}{2}\right)+6}{26}$$

$$\text{(8-71)}\quad y = 2\,.\,10^{-3}\,x - 10^{-3}$$

Para aproximar a inversa é treinado um MLP com a topologia (1, 4, 4, 1). O conjunto de treino consiste em 101 valores de concentração x, entre 0 e 1, igualmente espaçados por 0.01.

O neurónio da camada de saída tem uma função de activação linear. Para as restantes camadas do MLP foi utilizada a função sigmoidal. Esta função e a sua derivada são definidas como:

$$\text{(8-72)}\quad \mathrm{f}^{(z)}(x) = \frac{1}{1+\mathrm{e}^{-x}}, \quad z < q$$

(8-73) $$f^{(z)}{}'(x) = \frac{e^{-x}}{\left(1+e^{-x}\right)^2} = f^{(z)}(x)\left(1-f^{(z)}(x)\right), \quad z < q$$

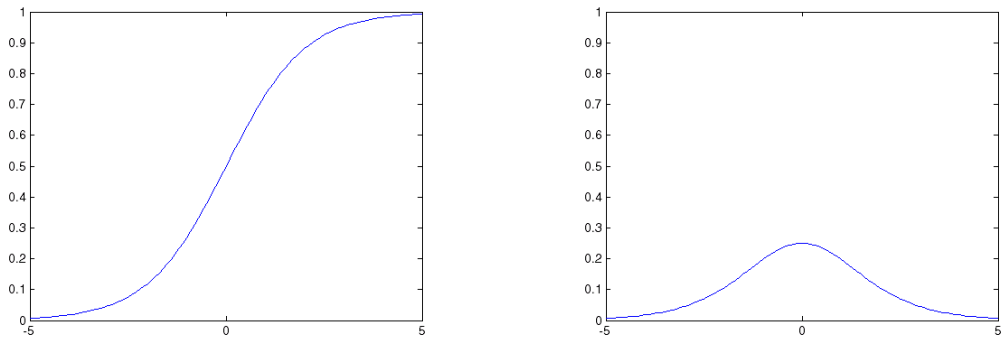


Fig. 8-30 Função de activação para todas as camadas excepto a
(função sigmoidal e sua derivada)

Repare-se que a função sigmoidal é uma função que satura para valores altos e baixos de entrada.

O conjunto de entrada consiste nos valores de pH correspondentes aos do conjunto de treino, calculados usando (8-70). Os valores iniciais dos pesos foram obtidos por uma estratégia que pode ser encontrada em (Ruano, 2002).

A figura abaixo representa a evolução da norma do erro, para 100 iterações do algoritmo de treino:

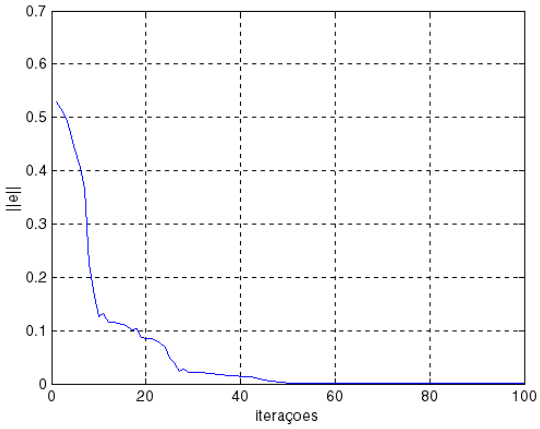


Fig. 8-31: Convergência da norma do erro em 100 iterações do algoritmo de treino

8.2.3 Implementação do algoritmo de treino de MLPs em SEQ

A implementação deste algoritmo em SEQ 1.0, não é totalmente suportada pela linguagem e biblioteca base, o que implica que terão de ser desenvolvidas algumas funções paralelas em C.
Um dos principais problemas advém das funções de cálculo da biblioteca base não operarem sobre áreas de matrizes. Tal é possível obtendo uma nova matriz com os elementos dessa área, optimamente distribuída, usando a função redist (**op1**, **OP2**, *li, ci, lf, cf*) fornecida pelo ambiente de comunicações, operar sobre esta nova matriz e colocar o resultado na área correspondente da matriz original. No entanto não existe nenhuma função definida para colocar uma matriz numa área de outra. Além disso desenhando apenas algumas novas funções que operam sobre uma área da própria matriz e deixam nessa o resultado, poupa-se tempo de cálculo considerável.

Também como se verá mais adiante é conveniente usar expressão $\mathbf{W}^{(q)}$ na forma vectorial dada por (8-50), para a implementação do algoritmo 8-11, embora não o seja para a operação de *recall*, como será tratado já a seguir.
Como já foi visto, o novo critério de aprendizagem utilizado, introduzido por (Ruano, 1992), vai actualizar um conjunto de pesos não lineares, entre as primeiras q-1 camadas, visto que a função de activação é a função não linear sigmoidal (8-72). Os vectores de *bias* correspondentes, $\mathbf{b}_v$ são juntos ao fim do vector de pesos lineares **v:**

$$\text{(8-74)} \quad \mathbf{w}_v = \begin{bmatrix} \mathbf{w}^{(q-2)} \\ \vdots \\ \mathbf{w}^{(1)} \\ \mathbf{b}^{(q-2)} \\ \vdots \\ \mathbf{b}^{(1)} \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ \mathbf{b}_v \end{bmatrix}$$

Aliás esta é a abordagem utilizada em (Ruano, 1992) e (Ruano et al, 1992), e como o desempenho do algoritmo em SEQ 1.0 será comparado com um algoritmo paralelo optmizado e escrito em Occam, para arquitecturas T8, pelos autores atrás referidos, foi escolhida uma abordagem o mais semelhante possível. No entanto o particionamento das matrizes é efectuado por linhas e não por colunas, pois é a estratégia que o SPAM fornece. Se bem que programando em C paralelo embebido no código SEQ seja possível outros tipos de

particionamento, tal requer um grande esforço de desenvolvimento, pois todas as rotinas da biblioteca base estão orientadas para um particionamento dos operandos matriciais por linhas.

O corpo principal do algoritmo de treino de um mlp com a topologia (kp, kh1, kh2, 1) em SEQ é o seguinte:

```
ini = 1;
for iter =1 to iterations {

        mlpdist(W0, w1, w2, bv1, bv2, kp, kh1, kh2);

        /*recall até q-2*/
        Net2 = I * w1' + ones * bv1';        /*O1 = Conjunto de entrada (vector I)*/
        O2=sigmoid(Net2, O2);
        Der2=dsigmoid(O2, Der2);

        Net3 = O2 * w2' + ones * bv2';
        O3=sigmoid(Net3, O3);
        Der3=dsigmoid(O3, Der3);

        /*Novo critério*/
        Av=addlc(O3, 1, Av);                        /*Av=[O3 1]*/
        u=solve (Av, T, 0, u, 0);
        E=Av*u-T;

        /*Matriz Jacobiana*/
        for i=1 to kh2 {
                x=u[i,1];
                colmult (Der3, i, x); }

        idxv=1;
        idxb=kp*kh1+kh1*kh2+1;
        for i=1 to kh1 {
                tmp=extcol (w2, tmp, i);  /*tmp = w2[:, i];*/
                d=Der3*tmp;
                ac_mult(d, 1, Der2, i, Jv, idxb);
                for j=1 to kp {
                        ac_mult(Jv, idxb, I, j, Jv, idxv);
                        idxv=idxv+1; }

                idxb=idxb+1; }

        for i=1 to kh2 {
                for j=1 to kh1 {
                        ac_mult(Der3, i, O2, j, Jv, idxv);
                        idxv=idxv+1; }

                inscol(Der3, i, Jv, idxb);
                idxb=idxb+1; }
```

```
/* LM Update */
W0=LMupdate(W0, E, Jv, ini);
ini=0;
}
```

Este algoritmo é baseado num versão sequencial optimizada para um MLP com duas camadas escondidas (Ruano, 2002), no entanto w1 e w2 são retornados transpostos, pois existem algumas vantagens em termos de paralelização do algoritmo, quando o particionamento é efectuado por linhas.

Inicialmente o vector W0, que guarda os pesos e *bias* no formato dado por (8-74), e que é vantajoso para a actualização de pesos efectuada pela função LMupdate é separado nos pesos e *bias* para cada uma das camadas escondidas w1, w2, bv1 e bv2, de acordo com a topologia dada por kp, kh1 e kh2, pela função:

PRE: kp > 0 && kh1 > 0 && kh2 > 0
POS:
Excepção: Se não houver memória suficiente para dimensionar o vector W0 local temporário gera um erro em tempo de execução.

matrix mlpdist(matrix W0, matrix w1, matrix w2, matrix bv1, matrix bv2, integer kp, integer kh1, integer kh2)

De notar que w1 e w2 são retornados transpostos, de modo que na Obtenção de Net2 e Net3 possam ser multiplicados, usando a versão da multiplicação que assume que o 2º operando é transposto. Como já foi mostrado no capítulo 5, esta operação é mais rápida que a multiplicação matricial sem transposição do 2º operando.
Além disso na obtenção da matriz Jacobiana, podem-se operar colunas em vez de linhas das matrizes w1 e w2, o que é mais eficiente, pois as colunas já estão distribuídas pela rede, podendo as operações serem efectuadas simultaneamente em todos os nós.

Seguidamente é efectuada operação de *recall*. Para efectuar esta operação é mais vantajoso que os pesos e *bias* estejam separados em matrizes independentes. Se bem que seja possível a manipulação da posição de elementos de matrizes em SEQ 1.0 em alguns casos terá de ser necessário deslocar elemento a elemento e a descodificação de endereços torna-se um factor de atraso apreciável. Por este motivo, para optimizar a manipulação de elementos não

convencional no algoritmo de treino, foram desenhadas funções optimizadas em parallel C. Estas são guardadas na biblioteca EXTMATH.INC.

Para adicionar uma coluna com o mesmo valor escalar ao fim de uma matriz, isto é:

newm = [oldm 1]

é utilizada a função:

```
PRE:
POS:   rows(new) == rows(old) && cols(new) == cols(old)+1

matrix addlc(matrix oldm, integer n, matrix newm)
```

Esta função vai ser necessária para preparar a matriz do sistema a resolver, **Av**, segundo o novo critério.
Para extrair o vector coluna col de uma matriz a e guardá-lo no vector c é utilizada:

```
PRE:   col >= 1 && col <= cols(a)
POS:
EXC:  Se não houver memória suficiente para dimensionar o vector coluna termina o
       programa e gera uma mensagem de erro

matrix extcol (matrix a, matrix c, integer col)
```

Para efectuar o contrário, isto é inserir o vector dado pela coluna col1 da matriz a na matriz c coluna col2 é utilizado:

```
PRE:  (col1 >= 1 && col1 <= cols(a)) &&  (col2 >= 1 && col2 <= cols(c)) &&
       rows(a)==rows(c)
POS:

inscol (matrix a, integer col1, matrix c, integer col2)
```

A função de activação não linear e a sua derivada, sigmoid e dsigmoid, respectivamente, foram implementadas em C. A sua implementação em SEQ seria eficiente a partir do momento em que se definissem funções para as operações sobre tabelas (ou *arrays),* isto é que operem cada elemento de uma matriz apenas com o elemento correspondente de outra matriz.

Seguidamente obtém-se a matriz Jacobiana. Para isso é outra vez necessário multiplicar elementos de matrizes um a um. Assumindo que os operandos e o resultado são colunas de matrizes, a função seguinte multiplica o elemento da coluna ca da matriz a pelo elemento correspondente da coluna cb da matriz b e guarda o resultado na coluna cres da matriz res.

```
PRE:  (c0 >= 1 && c0 <= cols(a)) &&
      (c1 >= 1 && c1 <= cols(b)) &&
      (c2 >= 1 && c2 <= cols(res))
POS:

ac_mult (matrix a, integer ca, matrix b, integer cb, matrix res, integer cres)
```

É também necessário multiplicar um vector coluna, dado pelo índice col, de uma matriz a por um escalar n e voltar a guardar o resultado na mesma posição. Essa operação é efectuada por:

```
PRE:   col >= 1 && col <= cols(a)
POS:

colmult (matrix a, integer col, real n)
```

Ambas estas funções são também implementadas em parallel C, pelas razões já apontadas. Finalmente é chamada a função LMupdate que actualiza o vector de pesos e *bias* W0, para a iteração seguinte, a partir do vector de erro E e da matriz Jacobiana Jv, segundo o algoritmo 8-11:

```
PRE:          cols(W0)==1  &&  cols(E)==1  &&
              rows(W0)==(k(1)+1)*k(2)+(k(2)+1)*k(3) && rows(E)==m
POS:
Excepção:     Se não houver memória suficiente para dimensionar o vector W0 local
              temporário  gera um erro em tempo de execução.

matrix LMupdate(matrix W0, matrix E, matrix Jv, integer ini);
```

O passo #4 deste algoritmo requer outra vez que seja resolvido um sistema de equações através de uma pseudo inversa, o que implica uma chamada a:

```
p = solve (Jv, E, ν, p, 1);
```

A implementação da resolução de sistemas indeterminados pela função solve, requer um tratamento e análise mais extenso, que será apresentado no ponto seguinte.

A indicação da convergência do treino é dado pela norma do erro. Esta também é necessária no cálculo do vector de *householder*, e o seu quadrado no algoritmo LM que actualiza o vector de pesos não lineares. O calculo da norma-2 e do seu quadrado é dado pelas funções norm2 e norm22 respectivamente.

PRE: rows(v) == 1 || cols(v) == 1
POS:

real norm2 (matrix v)
real norm22 (matrix v)

Ambas são implementadas à custa do produto de vectores coluna:

(8-75) $\|\mathbf{v}\|_2 = \sqrt{\mathbf{v}^T\mathbf{v}}$

(8-76) $\|\mathbf{v}\|_2^2 = \mathbf{v}^T\mathbf{v}$

É também definida a função vec_prod_li, que calcula o produto vectorial de duas colunas de duas matrizes e retorna um escalar. Esta será utilizada no cálculo do vector de *householder*:

PRE: c1 >= 1 && c1 <= cols(m1) &&
c2 >= 1 && c2 <= cols(m2) &&
li >= 1 && li <= rows(m1)
POS:
Excepção: Se não houver memória suficiente para dimensionar o vector temporário para receber os resultados parciais, termina o programa e gera uma mensagem de erro

real vec_prod_li (matrix m1, integer c1, matrix m2, integer c2, integer li)

Esta implementação é bastante simples e semelhante à já apresentada no capítulo 5, sendo a única diferença que opera sobre colunas de uma matriz em vez de vectores e retorna um número real em vez de uma matriz 1×1. O parâmetro inteiro li permite indicar a partir de que linha dos vectores coluna se inicia o cálculo do produto vectorial. será mostrada a sua necessidade mais à frente. As funções de cálculo de normas, atrás referenciadas, assumem no entanto que li é igual a 1, e que m1 = m2 e c1 = c2.

8.2.3.1 Resolução de sistemas indeterminados

A resolução de sistemas indeterminados pelo método dos mínimos quadrados é efectuada utilizando uma decomposição QR, pela função solve:

PRE: rows(**A**) > cols(**A**) && rows(**b**) == rows(**A**) && cols(**b**) == 1 && rows(**x**) == rows(**A**) && cols(**x**) == 1
POS:
Excepção: Se não houver memória suficiente para dimensionar as matrizes redistribuídas entrelaçadas ou o vector temporário que indica o início das partições, termina o programa e gera uma mensagem de erro

matrix solve (matrix **A**, matrix **b**, real *v*, matrix **x**, integer *mode*)

Esta função opera em dois modos dependendo do valor do argumento mode, Se *mode* nulo retorna a solução de:

$$\mathbf{x} = \mathbf{A}^{+}\mathbf{b}$$

e se *mode* diferente de zero:

$$\mathbf{x} = \begin{bmatrix} \mathbf{A} \\ \mathbf{I}\sqrt{v} \end{bmatrix} \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix}$$

Como já tinha sido indicado o algoritmo tem de ser dividido em duas fases, uma de obtenção de uma matriz $\mathbf{R}_1$ e um vector **y**, efectuada pela função qr e uma outra de solução, função bsubst, como mostra o seguinte segmento de código SEQ:

Algoritmo 8-15:

interlace(**A**, **b**, $\mathbf{R}_1$, **y**, *v*, *mode*, **MAP**);
qr($\mathbf{R}_1$, **y**, **MAP**);
bsubst($\mathbf{R}_1$, **y**, **MAP**);
extsol(**y**, **x**, **MAP**);

No entanto como $\mathbf{R}_1$ é uma matriz triangular superior, a distribuição pela rede por conjuntos de linhas consecutivas não permite obter o máximo de eficiência, visto que alguns processadores terão que operar mais elementos que outros. Uma forma de resolver isto é entrelaçar a matriz, no sentido em que a distribuição das linhas deixa de ser efectuada por conjuntos de linhas consecutivas. Esta fase é cumprida pela função interlace. No entanto, no

final os elementos do vector solução **x** têm de ser colocados na ordem original, pela função extsol.

Como à medida que o índice de linha aumenta diminuí o número de elementos em cada linha, pode ser adotada uma estratégia de entrelaçamento das linhas, aliás também usada por (Ruano, 1992). Segundo esta estratégia a linha 1 da matriz é alocada no nó 0, a linha 2 no nó 1 e assim sucessivamente até à última linha, numa forma circular, em que do último nó se regressa ao nó 0. Em termos de distribuição de elementos por nós, a diferença entre o nó que tem mais elementos e o que tem menos é dada por:

$$(8\text{-}77)\quad \omega_0 = \left\lceil m - \frac{m}{p} \right\rceil$$

onde p e m indicam respectivamente o número de linhas da matriz e o número de nós na rede, e o operador $\lceil\ \rceil$ indica que é efectuado um arredondamento para o inteiro imediatamente superior. Esta diferença ω_0 aumenta com o número de nós, no entanto o diferencial será cada vez menor.

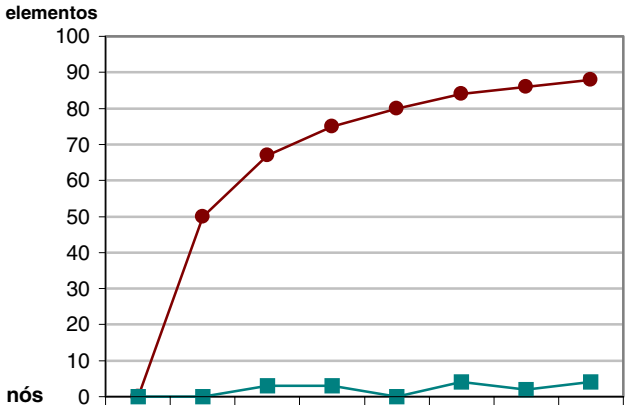
Outra estratégia de entrelaçamento, mais equilibrada, consiste em distribuir as linhas de um modo reflectido, isto é, quando se chega ao último nó, em vez de se alocar a linha seguinte no nó 0, aloca-se no último e começa-se a distribuir-se as linhas seguintes para trás até se atingir o nó 0. A partir daqui volta-se a distribuir as linhas ao longo dos nós com índices crescentes até se atingir o nó com índice mais alto, o último, onde se volta a distribuir as linhas pelo nó com índice decrescentes, e assim sucessivamente. Iniciando a distribuição na linha com mais elementos, a diferença entre o nó que tem mais elementos e o que tem menos é dada por:

$$(8\text{-}78)\quad 0 \le \omega_1 < p$$

e que depende fortemente do número de nós da rede. No entanto, o valor de ω_1 depende do número de nós que têm mais uma linha alocada.

Para entrelaçar uma matriz com 100 linhas e 100 colunas, por redes até 8 nós têm-se o cenário apresentado na Fig. 8-32. Repare-se que o diferencial vai decrescendo para ω_0. Para matrizes desta ordem ω_0 tende para uma assimptota paralela ao eixo horizontal, não muito superior ao valor para 8 nós. De facto essa assimptota tem ordenada igual ao número de colunas da matriz, neste caso 100. A diferença ω_1 só se aproxima de ω_0 para uma rede com um número

de nós igual à ordenada dessa assimptota. Deste modo, para uma situação normal, onde *m* é muito maior que *p*, a segunda distribuição, cuja diferença é dada por ω_1 é mais equilibrada, mas nunca é superior ao comprimento de uma linha. Assim o mais que se pode ganhar com esta segunda distribuição é uma linha, o que não é muito significativo em termos de SPAM, pois como a distribuição é feita por linhas, se o número de linhas da matriz não for um múltiplo exacto do número de processadores, pelo menos um nó terá mais uma linha atribuída que os outros.



nós	1	2	3	4	5	6	7	8
w0	0	50	67	75	80	84	86	88
w1	0	0	3	3	0	4	2	4

Fig. 8-32: Diferença máxima de elementos alocados

De qualquer forma a segunda distribuição permite aumentar um pouco a eficiência, de modo que foi escolhida para a implementação.

O peso relativo desta fase e a reorganização final do vector resultado, funções interlace e extsol, correspondem apenas a menos de 2% do tempo total de solução de um sistema 100 x 100, pelo Algoritmo 8-15, em dois T8 a 25 MHz. Na realidade o tempo de interlace é da ordem da operação transposição de matrizes já apresentado.

Se bem que estas percentagens possam variar com as dimensões das matrizes, as famílias de processadores e a topologia da rede, para uma distribuição de carga equilibrada, a complexidade relativa de cada fase deve ser próxima das medidas na rede acima mencionada. Nesta rede a etapa de *backsubstitution* é inferior a 1,5%, donde se concluí claramente que a etapa mais pesada e a qual merece maior atenção na paralelização é a obtenção das matrizes $\mathbf{R}_1$ e **y,** função qr, que tem um peso de cerca de 97.5%.

A função interlace, cujo protótipo é:

```
PRE:        rows(A) > cols(A) && rows(b) == rows(A) && cols(b) == 1 &&
```

cols(A1) == cols(A) && cols(b1) == cols(b) &&
(mode==0) && (
rows(A1) == rows(A) && rows(b1) == rows(b))
(mode!=0) && (
rows(A1) == rows(A)+cols(A) && rows(b1) == rows(A)+cols(A))

POS:

Excepção: Se não houver memória suficiente para dimensionar as matrizes redistribuídas ou o vector temporário que indica o início das partições, termina o programa e gera uma mensagem de erro

interlace(matrix A, matrix b, matrix A1, matrix b1, real v, integer mode, matrix MAP)

deixa em A1 e b1 as matrizes entrelaçadas. Repare-se que esta função retorna também um mapa das linhas alocadas em cada nó. No entanto, como o número de linhas pode ser inferior ao número de colunas, apenas entrelaça um número de linhas igual ao número de colunas pois as restantes linhas serão sempre calculadas ao longo de todas as iterações, como mostra a seguinte representação esquemática das matrizes do sistema e sua relação com $\mathbf{R}_1$ e $\mathbf{y}$:

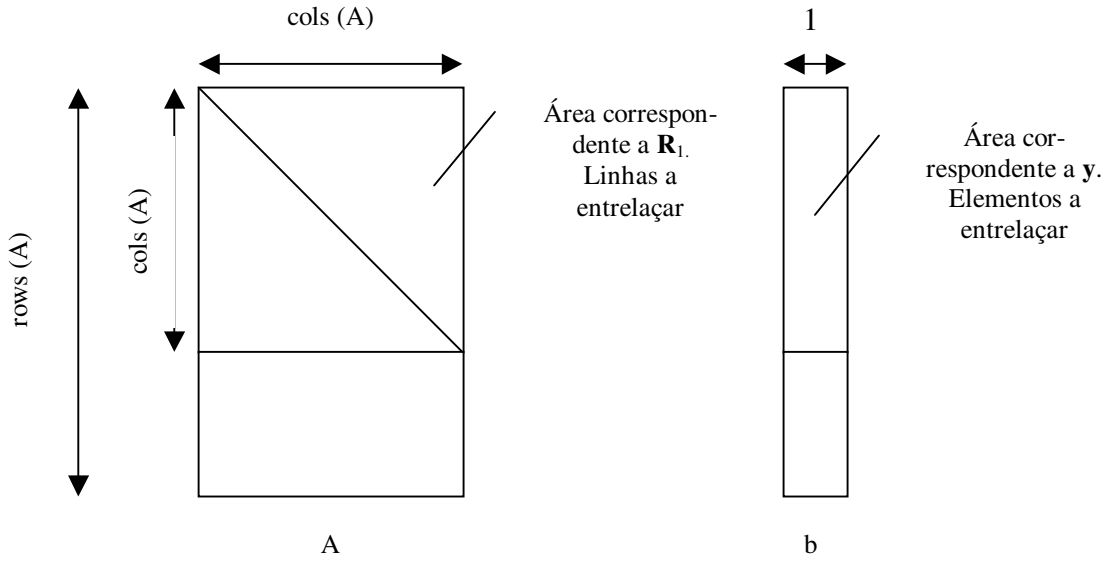


Fig. 8-33: Matrizes do sistema e sua relação com $\mathbf{R}_1$ e $\mathbf{y}$.

Assim o mapa de distribuição, existente em cada nó, tem a forma apresentada na tabela seguinte, onde cada linha contém tantos elementos como as linhas de $\mathbf{R}_1$ alocadas no nó correspondente, mais o elemento terminador que é igual a zero. Repare-se que desde que a estratégia de entrelaçamento implique que a sucessão de índices de linhas num dado nó seja monótona crescente ou decrescente, esta pode ser implementada simplesmente alterando MAP de acordo. Todas as funções e segmentos de código que operam sobre colunas com início numa dada linha, como vec_prod_li, funcionarão perfeitamente.

nó				
0	1	p+1	...	0
1	2	:	...	0
:		:		
p-1	p	:	...	0

tabela 8-4: Representação esquemática da tabela de distribuição de linhas

Quanto à etapa bsubst, esta opera também apenas sobre as linhas entrelaçadas dadas por MAP. Segue uma abordagem convencional onde o cálculo é efectuado desde a última coluna até à primeira e onde em cada uma é primeiro obtido o elemento do vector solução **x** correspondente ao índice da coluna. Este é depois enviado para todos os processadores, de modo a actualizar os restantes elementos da coluna.

```
PRE:   rows(R) == cols(R) && rows(x) == rows(R) && cols(x) == 1 &&
       "R é triangular superior direita".
POS:   guarda solução em x

bsubst (matrix R, matrix x, matrix MAP)
```

Finalmente a etapa de triangularização, função qr, consiste na implementação do algoritmo 8-13 e do cálculo do vector de *householder,* algoritmo 8-14:

PRE: rows(**y**) == rows($\mathbf{R}_1$) && cols(**y**) == 1
POS:

qr (matrix $\mathbf{R}_1$, matrix **y**, matrix **MAP**)

A sua implementação é efectuada em SEQ com código C embebido para implementar eficientemente operações sobre áreas de matrizes, pois em cada iteração do algoritmo qr é calculada uma linha e uma coluna de $\mathbf{R}_1$, da esquerda para a direita e de cima para baixo. Será assim necessário operar sobre uma área da matriz cada vez menor. Em termos de cálculo de cada coluna será então necessário indicar em que linha se inicia o processamento, daí o argumento li da função vec_prod_li.
São também reescritas versões de algumas funções já definidas anteriormente neste capítulo, que operam sobre uma coluna, col, de uma matriz, a, mas a partir de uma dada linha li:

```
matrix extcol_li (matrix a, matrix c, integer col, integer li)
colmult_li (matrix ma, integer col, real r, integer li)
```

Em cada iteração, é pois necessário calcular primeiro a linha inicial em cada nó, usando:

Algoritmo 8-16:

```
C {     for(;;) {
                    if (MAP[WRK][li] == 0) break;
                    else if (MAP[WRK][li] > i) break;
                    else if (MAP[WRK][li] == i) { lia=LINHA_INICIO(R)+li; break; }
                    li++; }
    }
```

onde a linha inicial de cada nó li e a linha inicial referente à matriz global lia são inicialmente 0 e -1 respectivamente. WRK representa o índice do nó corrente.

Seguidamente deve-se calcular o vector de *householder*. Este é calculado pela função hhv, totalmente programada em SEQ, mas que usa as funções vec_prod_li, para cálculo da norma e colmult_li, escritas em C paralelo:

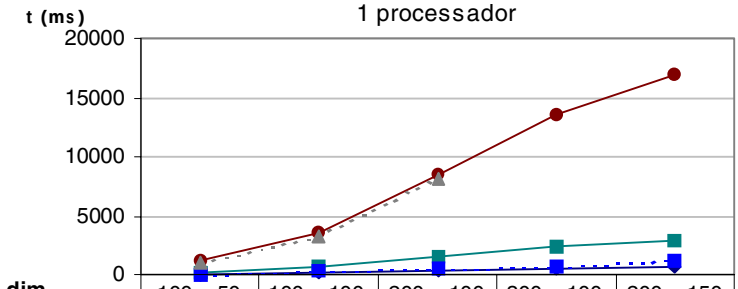
```
PRE:    cols(x) == 1 & li >= 0 && lia > 0
POS:

hhv (matrix h, integer li, integer lia) {
real q, s, m, tmp;

        q = vec_prod_li (h, 1, h, 1, li);
        q = power(q, 0.5);

        tmp=h[lia, 1];
        s=q*sgn(tmp);
        m=q+mod(tmp);
        h[lia, 1]=sgn(tmp)*m;

        colmult_li (h, 1, 1/power(q*m, 0.5), li);
        exit s;
}
```

Finalmente, para terminar a iteração, a implementação da actualização de $\mathbf{R}_1$ e **y,** consiste num segmento de código totalmente programado em C, pois as operações matriciais multiplicação e subtracção da biblioteca base não permitem operar sobre área de matrizes, como já foi discutido.

8.2.3.1.1 Análise do desempenho da função solve

A resolução de sistemas indeterminados, pelo algoritmo atrás apresentado, é a etapa que tem maior peso no algoritmo de treino do perceptrão, por isso justifica-se uma análise de desempenho em separado. As figuras seguintes apresentam os tempos para as redes homogéneas e a eficiência para 2 e 3 nós. A análise do desempenho em redes heterogéneas será analisado apenas para o algoritmo de treino completo. Será no entanto comparado o desempenho com o algoritmo desenvolvido por (Ruano, 1992), optimizado para redes T8, já mencionado e que será designado por T8r.

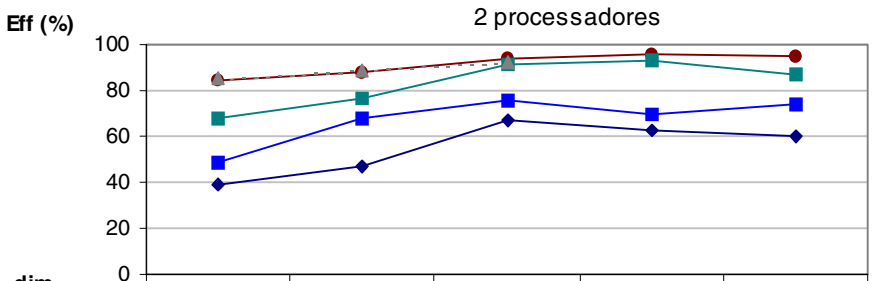
É apresentado o desempenho para sistemas 100×50, 100×100, 200×100, 300×100 e 200×150. No entanto para T8r, os resultados para as duas últimas dimensões de sistemas não estão disponíveis. Como se pode ver a implementação sobre T8 segue de perto T8r. Também como seria de esperar a implementação sobre C4 é mais rápida e sobre ADSP21060 ainda mais. No caso de um processador ADSP21060 foi necessário alocar o código na memória interna e os dados na memória externa, para sistemas com dimensão 200×100, 300×100 e 200×150, mas como não existem comunicações o desempenho é idêntico independentemente da zona de memória onde os dados estão armazenados. Para 2 nós deste tipo, foi ainda necessário usar a mesma estratégia para sistemas 300×100 e 200×150. Nestes dois casos existirá uma perda de desempenho devido a transferências de dados alocados na memória externa, entre nós. O tempo de execução num só nó é:



dim	100 x 50	100 x 100	200 x 100	300 x 100	200 x 150
T8	1153,60	3499,30	8491,80	13486,00	16928,80
C4	202,00	617,00	1488,00	2300,00	2826,00
SH	48,00	139,00	326,00	512,00	640,00
SH1	73,00	325,00	511,00	698,00	1256,00
T8r	1036,00	3267,00	8073,00		

Fig. 8-34: Tempos de resolução de sistemas indeterminados pela função solve

Quanto à eficiência para dois e três nós tem-se:

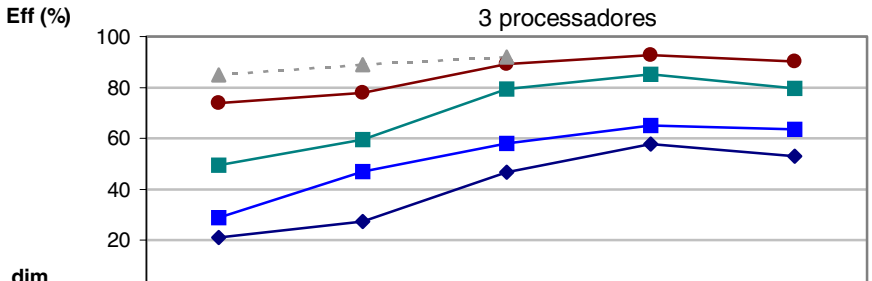


dim	100 x 50	100 x 100	200 x 100	300 x 100	200 x 150
T8	84,77	87,61	94,08	95,97	94,83
C4	67,79	76,36	90,95	93,27	87,22
SH	38,71	46,64	67,08	62,90	59,70
SH1	48,67	67,43	76,04	69,66	74,32
T8r	85,00	89,00	92,00		

Fig. 8-35: Eficiência da função solve para 2 nós

Pode-se verificar que T8 segue de perto T8r. C4 também atinge uma boa eficiência. No entanto as arquitecturas baseadas em ADSP21060, não se conseguem aproximar em termos de eficiência, o que deixa alguma margem de optimização do algoritmo para esta arquitectura.

Para todas as arquitecturas foi ensaiada a resolução de sistemas quando *mode* é nulo, mas para as baseadas em ADSP21060, foi também ensaiado o caso em que *mode* não é nulo, referido nos gráficos por SH1. Repare-se que neste caso, já referido acima, as matrizes do sistema vão ter um acréscimo de tantas linhas como o número de colunas do sistema, o que permite ensaiar o algoritmo para um maior volume de dados a processar. Como se pode observar, tanto para 2 como para 3 nós, verifica-se uma pequena melhoria no desempenho.



dim	100 x 50	100 x 100	200 x 100	300 x 100	200 x 150
T8	73,76	78,00	89,22	92,69	90,31
C4	49,51	59,44	79,36	85,19	79,56
SH	21,05	27,42	46,84	57,85	52,94
SH1	28,97	46,90	57,94	65,17	63,63
T8r	85,00	89,00	92,00		

Fig. 8-36: Eficiência da função solve para 3 nós

Em termos de comparação de eficiência com (Ruano, 1992), deve ser indicado que os valores nos dois gráficos acima referem-se a 5 nós, o que quer dizer que o algoritmo desenvolvido neste trabalho pode ainda ser optimizado mesmo para T8s, pois como se pode ver para 3 nós a eficiência de T8 afasta-se de T8r e essa tendência deve manter-se á medida que os número de nós se aproxima de 5. No entanto (Ruano, 1992) utiliza um processador extra que funciona como *master* e efectua algumas operações de cálculo, mas nos cálculos de eficiência esse processador não é considerado. Se tal fosse a eficiência seria menor, pois na realidade são utilizados 6 processadores.

A implementação do algoritmo apresentada nesta tese não necessita desse processador extra. Se bem que o SPAM utilize um processo extra, para comunicação com o computador anfitrião, este processo não é necessário para as operações de cálculo e é alocado no mesmo processador que detém a primeira instância da aplicação, Prg (0), de modo que apenas requer algum espaço de memória extra nesse processador (o processador raiz), mas não têm influência nas operações de cálculo.

8.2.4 Análise de desempenho do algoritmo de treino do MLP em SEQ 1.0 (SPAM 1.0)

O ambiente de ensaio utilizado é igual ao descrito para o caso de teste anterior, no ponto 8.1.4., no entanto neste algoritmo não é usada a multiplicação matricial entre duas matrizes de modo que não é necessário ensaiar as duas versões, como o foi para o algoritmo AGPC. Deve no entanto ser referido, que para redes com três ADSP21060, nos casos em que foi necessário alocar dados na memória externa do ADSP21060, as medições apresentadas são pouco credíveis. Isto deve-se ao ADSP21060, da placa-mãe utilizada, ESTORIL, apresentar uma variação de tempos de execução enorme entre duas execuções do mesmo algoritmo, nas mesmas condições, sempre que parte da aplicação tem de ser alocada na memória externa. De facto chegaram a ser observadas variações onde o tempo de execução se aproxima do dobro do esperado. Como não foi possível eliminar este fenómeno os tempos apresentados nestas condições são meramente indicativos.

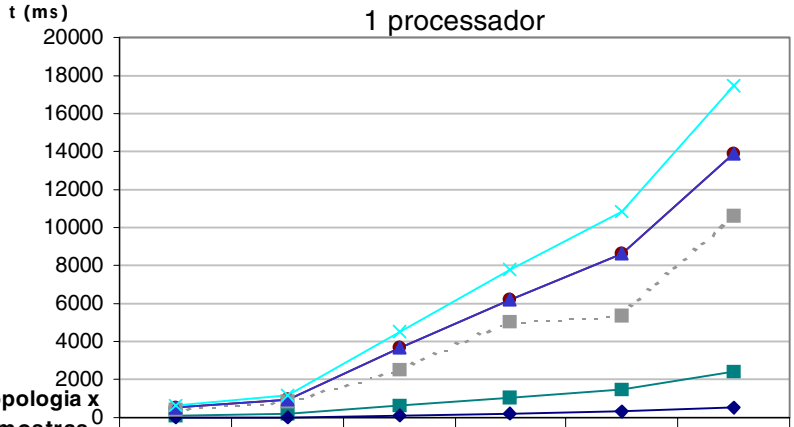
As topologias do MLP utilizadas para ensaio foram:

k = [3 3 3 1]	=	24 colunas
k = [3 6 6 1]	=	66 colunas
k = [5 7 7 1]	=	98 colunas

essencialmente para comparação com o desempenho do algoritmo desenvolvido por (Ruano, 1992). Para cada uma destas topologias o número de colunas na matriz jacobiana, isto é a matriz do sistema indeterminado a resolver pela função solve, aumenta. Para verificar a influência do aumento do número de linhas no sistema foram testadas estas topologias com conjuntos de amostras com 100 padrões e 200.

A distribuição de carga óptima, para hetA, é efectuada segundo a relação de desempenho entre um C4 e um T8-25MHz para o caso de treino mais pesado, 5,7,7,1 × 200, isto é 0,1710. Esta relação é no entanto muito próxima para os outros casos menos pesados ensaiados. Além disso também é semelhante, se bem que um pouco superior, à usada no caso do algoritmo AGPC: 0,1664. Para distribuir a carga na rede hetA, é usada a relação entre a velocidade de relógio T8-20MHZ / T8-25MHz = 0,8, tal como também foi efectuado para o algoritmo AGPC. Volta a verificar-se que para ordens de matrizes de 20 a 100 a relação é muito próxima de 0,8. De facto varia entre 0,8016 e 0,7978.

Para as medidas de eficiência, tal como para o algoritmo AGPC, é utilizada a relação para o caso mais pesado em termos de processamento, isto é 5,7,7,1 × 200, que para hetA é 0,1710 e hetB 0,7978.



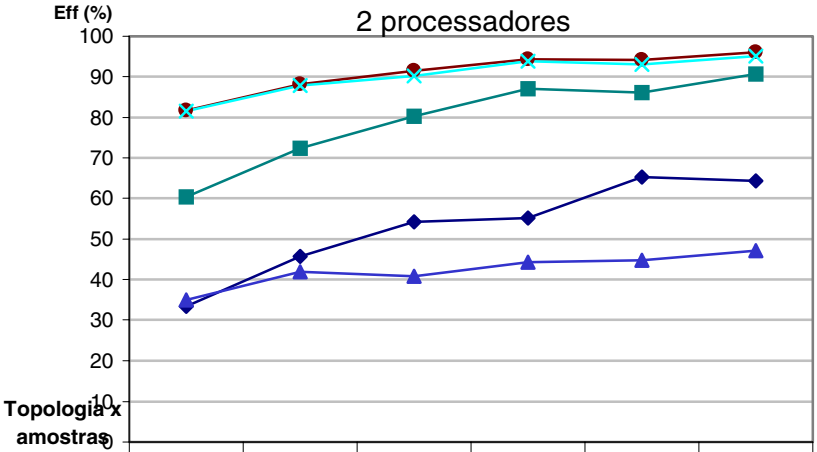
Topologia x amostras	3331 x 100	3331 x 200	3661 x 100	3661 x 200	5771 x 100	5771 x 200
T8	535,3	970,3	3653,4	6177,8	8683,5	13946,9
C4	93,1	163,3	632,1	1055,2	1485,6	2386,1
SH	24,0	42,0	142,0	235,0	326,0	518,0
T8r	413,0	818,0	2558,0	5055,0	5370,0	10660,0
hetA	535,3	970,3	3653,4	6177,8	8683,5	13946,9
hetB	667,8	1210,5	4574,7	7739,2	10884,4	17481,6

Fig. 8-37: Tempos de execução do algoritmo de treino do mlp para um nó

A figura acima mostra os tempos para as arquitecturas homogéneas e heterogéneas bem como os tempos de (Ruano, 1992), para um processador. Estes tempos constituem a média das 10 primeiras iterações

Como já foi referido hetB[1] refere-se apenas a um T8 - 20 MHz, por isso é um pouco mais lenta que T8. Já hetA[1] é constituída por um T8 - 25 MHz de modo que é idêntica a T8[1]. Quanto à comparação entre T8 e T8r, verifica-se que para dimensões grandes tende a afastar-se, o que sugere que, à semelhança da triangularização QR, também a operação de *recall* e o cálculo da jacobiana podem ser optimizados. C4 e SH são muito mais rápidas.

As medidas de eficiência para T8r, apresentadas em (Ruano 1992), foram efectuadas em redes com três T8 – 25 MHz, excepto para os casos $5,7,71 \times 100$ e $5,7,7,1 \times 200$, onde o algoritmo foi mapeado sobre 7 destes processadores. No entanto estes valores são colocados também no gráfico para 3 nós como medida de comparação, mas como aliás era de esperar a eficiência é um pouco menor.

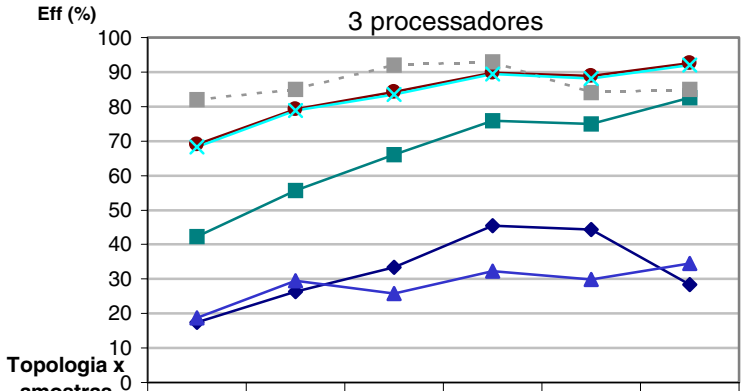


	3331 x 100	3331 x 200	3661 x 100	3661 x 200	5771 x 100	5771 x 200
T8	81,70	88,13	91,55	94,38	94,14	96,00
C4	60,38	72,38	80,28	87,11	86,12	90,75
SH	33,33	45,65	54,20	55,16	65,20	64,27
hetA	34,86	41,80	40,73	44,19	44,65	47,05
hetB	81,49	87,79	90,21	93,84	93,03	95,03

Fig. 8-38: Eficiência do algoritmo de treino do MLP para 2 nós

A eficiência de T8 e hetA é muito semelhante. No caso de 3 nós é próxima de T8r, com excepção dos dois últimos casos onde é superior, mas T8r está mapeado sobre 7 processadores, o que implica alguma perda de eficiência para apenas 3 processadores. C4 têm um eficiência razoável para grandes volumes de dados. SH [2] e SH [3] têm uma eficiência muito baixa, pois segue a eficiência da fase mais pesada, a triangularização QR. A eficiência é

ainda menor para 2 nós, casos 3,6,6,1 × 200 e 5,7,7,1 × 200, e para três nós, caso 5,7,71 × 200, pois os dados tiveram de ser alocados na memória externa. Neste último caso, como é usada a memória externa do Sharc raíz, pelas razões já referidas neste ponto, os tempos medidos variam muito, só em poucos casos se aproximando do esperado, fazendo com que o valor de eficiência apresentado, a média das 10 primeiras iterações, seja bastante abaixo do esperado.



Topologia x amostras	3331 x 100	3331 x 200	3661 x 100	3661 x 200	5771 x 100	5771 x 200
T8	68,95	79,14	84,26	89,72	88,91	92,56
C4	42,28	55,66	65,97	75,94	74,94	82,54
SH	17,39	26,42	33,33	45,54	44,35	28,45
T8r	82,00	85,00	92,00	93,00	84,00	85,00
hetA	18,67	29,47	25,84	32,28	29,78	34,57
hetB	68,23	78,77	83,58	89,35	88,13	91,97

Fig. 8-39: Eficiência do algoritmo de treino do MLP para 3 nós

Mais uma vez verifica-se que hetB é a topologia menos eficiente para grandes volumes de dados, devido à comunicação entre o T8 e C4 ter uma largura de banda muito estreita.

8.3 Desempenho do gerador de código C paralelo

Se bem que não seja um objectivo primário deste trabalho, pode-se também apresentar o desempenho do tradutor. A geração de código C paralelo fonte, divide-se em duas etapas:

1) Geração do ficheiro de configuração da rede e do *makefile*.
2) Tradução do código SEQ 1.0 para código C paralelo.

Como já foi abordado, um módulo especifico do SPAM trata cada uma delas. O tempo de execução da primeira etapa, para as redes utilizadas nos dois casos de teste ronda os 200 ms. Convém no entanto lembrar que estas redes são bastante simples. Redes mais complexas implicam que o tempo cresça proporcionalmente ao número de nós e ligações entre eles. No entanto os tempos indicados incluem todos os acessos necessários a ficheiros, bem como a impressão de algumas mensagens de progresso na consola, e a criação de um *makefile*, de modo que a proporção não pode ser feita em relação ao tempo total indicado, isto em relação a 200 ms, mas sim a um valor francamente inferior.

O tempo de execução da etapa de tradução, para os dois casos de teste é próximo, mas inferior a um segundo. No caso desta etapa, não se pode contar apenas a tradução dos algoritmos AGPC e o algoritmo de treino de um MLP, pois todas as bibliotecas, sejam de base ou definidas pelo utilizador também são traduzidas para C paralelo, sempre que é gerada uma aplicação. Da mesma forma que para a primeira etapa, o tempo indicado incluí todos os acesso necessários a ficheiros e a impressão de mensagens de progresso na consola.

Estes tempos foram medidos numa versão do SPAM 1.0 para Windows 98, executada sobre uma plataforma Intel Celeron 633 MHz, com 128 MBytes de RAM e bus de acesso 67 MHz, e um disco rígido com tempo médio de acesso de 13 ms.

Os tempos indicados destas duas etapas são aproximados e medidos garantindo que a única aplicação em *foreground*, no Windows 98 é o SPAM. No entanto é impossível garantir que os tempos permanecem constantes entre várias execuções, devido por exemplo a tarefas *background* que o sistema possa lançar, o que gera alguma imprevisibilidade, se bem que na maior parte dos ensaios estas duas etapas são efectuadas em menos de dois segundos.

8.4 Resumo

Na maior parte dos casos, o ambiente gera automaticamente código eficiente para algoritmos baseados na álgebra linear básica, no entanto o desempenho de alguns processadores pode ser melhorado reprogramando as funções de comunicação básicas putmessage e getmessage para tirar partido dos mecanismos de DMA. Tal não é necessário no Transputer pois este processa e comunica em simultâneo transparentemente para o programador.

Além disso, estes processadores perdem também tempo significativo se a gestão de memória dinâmica fôr efectuda durante a evolução do algoritmo. Como já foi apontado o decompositor de expressões cria automaticamente matrizes temporárias para guardar resultados intermédios de operações binárias e unárias. Se bem que esta estratégia liberte o programador de ter de dimensionar as matrizes resultado de acordo com os operandos e operadores, perde-se algum

tempo na gestão de memória. Isto implica uma perda de eficiência, que é visível quando comparado com a implementação do algoritmo AGPC desenvolvida nesta tese, com a implementação do algoritmo AGPC, apresentado em (Daniel e Ruano, 2000), Fig. 8-17, onde todas as matrizes são pré-alocadas.

Se bem que todas as funções de cálculo básicas foram desenvolvidas no sentido de criarem a matriz resultado, estas podem ser facilmente modificadas, para não o fazer, eliminando simplesmente a função de alocação da matriz. Neste caso é no entanto conveniente que esteja habilitada a detecção de erros em tempo de execução, de modo a validar as dimensões dos operadores, e poder proceder-se de acordo.

Segundo esta aproximação é possível reescrever o decompositor de expressões de modo que use sempre as mesmas matrizes temporárias, pré-alocadas, para guardar os resultados intermédios. Estas matrizes devem ser de uso geral, e por isso ter uma dimensão que permita guardar qualquer operando durante o tempo de execução, sendo a indicação da dimensão actual efectuada apenas alterando o cabeçalho. Como a dimensão destes operandos pode depender das entradas de um processo durante a sua execução, não é possível dimensioná-los em tempo de compilação, sendo necessário que o utilizador da aplicação o indique, por exemplo num ficheiro de configuração. Se em alguma situação, durante a execução fôr necessário aumentar a dimensão destas matrizes temporárias, pode-se sempre escrever as funções de cálculo para executarem uma função que o faça automaticamente; tal no entanto isto levar a alguma imprevisibilidade.

De qualquer forma, para a decomposição de expressões em operações binárias não é necessário mais do que três operandos temporários, de modo que o espaço de memória ocupado por estes não é assim tão proibitivo, permitindo que se possa usar uma margem de segurança na escolha da dimensão inicial das matrizes, embora esteja-se sempre dependente da dimensão do problema e da memória disponível.

Deste modo consegue-se a mesma transparência na linguagem SEQ, com algum aumento de eficiência.

Foi também observado, que em alguns casos poderá ser necessário definir funções especificas que devem ser programadas em parallel C embebido em SEQ de modo a optimizar o seu desempenho, se bem que estas possam ser programadas em SEQ puro. Tal é recomendado para rotinas que necessitam de manipular elementos de matrizes.

Isto no entanto sugere que futuras versões da linguagem SEQ, devam ser acompanhadas de um vasto e eficiente conjunto de bibliotecas, como é o caso da linguagem C/C++, de modo que diferentes tipos de aplicações possam ser programadas em SEQ puro.

9 Conclusões e trabalho futuro

9.1 Conclusões

Neste trabalho foi desenvolvido e ensaiado um sistema automático de paralelização de algoritmos matriciais, SPAM 1.0, que a partir de uma linguagem imperativa sequencial proposta, SEQ 1.0, estabelece uma ponte para o paradigma concorrente transparente para o programador.

Para ser gerada uma aplicação paralela, além se descrever o algoritmo em SEQ é apenas necessário descrever a rede paralela alvo num diagrama de blocos, onde devem ser expressos como estão conectados os nós e o poder de cálculo relativo de cada um.

Para tornar mais amigável a gestão de projectos, foi implementado um ambiente de desenvolvimento integrado de aplicações, AIDA. As ferramentas de geração de aplicações estão também disponíveis na linha de comandos.

Para ser gerada a aplicação, o código fonte SEQ é traduzido para código fonte C paralelo. Este código é injectado em compiladores da 3L indicados para os processadores utilizados, e ligado com o ambiente de comunicações e as bibliotecas base da linguagem SEQ ou outras definidas pelo utilizador.

Assim uma aplicação gerada pelo SPAM, pode ser vista num conjunto de camadas de abstracção, onde o *interface* com o *hardware* é efectuado por um micro-núcleo fornecido com os pacotes de ferramentas 3L. O ambiente de comunicações consiste na camada seguinte, as bibliotecas de cálculo matricial são implementadas sobre este ambiente, e no topo têm-se o código sequencial SEQ, que implementa comandos para usufruir das facilidades das duas camadas abaixo, completamente abstraído do *hardware* onde a aplicação executa.

Deste modo muito do desempenho do sistema reside na eficiência dos mecanismos de comunicação descritos no capítulo 4, e das funções de cálculo matriciais, descritas no capítulo 5. Assim antes da implementação destas duas camadas, cada tipo de processador utilizado foi ensaiado em separado e em conjunto, de modo a obter dados comparativos do seu desempenho em termos de operações de vírgula flutuante e taxas de transferência de dados entre diferentes nós de uma rede.

O ambiente de comunicações foi desenhado para arquitecturas de passagem de mensagem, assumindo que cada nó é capaz de comunicar por todos os portos em simultâneo e bidireccionalmente. A gestão de trafego é efectuada recorrendo a um algoritmo optimizado em termos de selecção do porto de saída, quer para uma mensagem com origem no próprio nó, quer proveniente de outro nó, de modo a reduzir a latência no encaminhamento de pacotes.

O ambiente disponibiliza duas primitivas de comunicação básicas, send e receive, onde apenas é necessário indicar o destino ou origem, e um vector de dados a enviar. Se bem que este ambiente disponibilize funções de comunicação mais especializadas, nomeadamente para distribuição de matrizes, estas são desenvolvidas à custa destas primitivas.

Para simplificar a comunicação entre nós da rede afastados do anfitrião com este, foi desenvolvido um servidor, alocado no nó raiz, que permite o acesso de qualquer nó aos recursos do anfitrião, apenas pela submissão de um pedido de serviço.

Esta facilidade é também usada para a implementação de um sistema de detecção de erros, onde no caso de um ocorrer pode ser enviada uma mensagem para a consola e terminada a aplicação.

Todas estas facilidades são utilizadas na implementação das funções de cálculo matricial paralelas. Estas operam sobre áreas de matrizes compostas por conjuntos de linhas consecutivas, distribuídas pela rede de um modo equilibrado, dependente do poder de cálculo de cada nó. Se bem que outros particionamentos sejam possíveis, verificou-se que para grandes volumes de dados, esta estratégia permite atingir uma boa distribuição de carga computacional, e simplifica o desenvolvimento de funções paralelas.

As funções de cálculo matricial básicas foram também ensaiadas em várias redes. Estes dados podem ajudar o programador a ter uma ideia do desempenho de uma dada aplicação, desde que seja desenvolvida à custa destas funções. Podem também servir para a implementação de um simulador.

Para decompor expressões complexas o tradutor usa um decompositor de expressões que as decompõe em operações binárias ou unárias gerindo automaticamente a memória dinâmica para guardar os resultados intermédios. Esta abordagem no entanto implica alguma perda de eficiência na gestão de memória dinâmica. Uma alternativa discutida no capítulo 8, mas que no entanto não foi implementada, consiste em usar matrizes temporárias pré-alocadas, para guardar os resultados intermédios.
Com excepção do decompositor de expressões, o tradutor é totalmente configurável, através do ficheiros de configuração. Esta facilidade foi criada para permitir um fácil desenvolvimento e teste da linguagem SEQ, mas também permite que outra tradução seja configurada.

As aplicações geradas, e executadas sobre o hardware suportado podem ser integradas num ambiente de tempo-real, pois entre o *hardware* e o SPAM existe um micro-núcleo de tempo real, desenvolvido pela 3L. Como foi descrito no capítulo 5 e 6, a verificação do cumprimento de metas temporais pode ser efectuado recorrendo a funções C e SEQ. Também a aquisição de dados com o exterior pode ser efectuada em SEQ através das funções putv e getv ou mesmo em parallel C. Estas funções utilizam um serviço do servidor de entrada saída residente no processador raíz que envia ou recebe vectores de dados do computador anfitrião. Se no computador anfitrião for executado um processo servidor que ineractue com o processador raíz e com os portos de comunicação do anfitrião ou outro *hardware* de aquisição de dados específico ou actuadores é possível desenvolver aplicações para controlar dispositivos externos.

Os algoritmos paralelos gerados foram ensaiados sobre redes homogéneas compostas por Transputers, TMS320C40, e ADSP21060 e redes heterogéneas compostas de Transputers e TMS320C40 de modo a mostrar o desempenho para uma gama de situações variadas.
Confirmaram-se as expectativas de que redes T8 são as mais eficientes, pois são capazes de comunicar e efectuar cálculo de vírgula flutuante concorrentemente. Se bem que os outros tipos de processadores utilizados também o permitam, requerem a programação do co-processador de DMA. Se bem que o micro-núcleo 3L para a família ADSP21060, utilize estratégias de DMA (3L, 1998), como foi mostrado no capítulo 2, as comunicações implicam um custo em termos de tempo de cálculo para esta família. No entanto em (Analog Devices, 1997) é referido que usando o co-processador de DMA, comunicações podem ser efectuadas sem penalização da unidade de virgula flutuante, sugerindo que a estratégia de DMA utilizada

pelo implementada no micro-núcleo da 3L talvez possa ser melhorada. De qualquer forma neste trabalho, para manter o código C paralelo gerado pelo SPAM o mais compatível possível, as transferências de dados são efectuadas utilizando as primitivas de comunicação básicas fornecidas pelo micro-núcleo 3L, não sendo programado explicitamente os co-processadores de DMA.

Quanto a arquitecturas heterogéneas, verificou-se que desde que a largura de banda entre nós de tipos diferentes não se reduza descomunalmente, como é o caso entre um T8 e um C40, a distribuição automática de carga, baseada no poder de cálculo de cada nó, permite atingir uma boa eficiência.

9.2 Trabalho futuro

Algumas extensões e optimizações quer ao nível da geração automática de código, do ambiente de comunicação ou da biblioteca de cálculo matricial poderão aumentar o desempenho das aplicações automaticamente geradas pelo SPAM. Seguidamente apresentam-se algumas sugestões:

- O sistema de paralelização automática gera código fonte C, mais concretamente 3L parallel C, de modo que as aplicações por este geradas, sejam portáveis para redes paralelas, compostas por qualquer processador suportado por este compilador. Além disso, com pequenas modificações ao nível da biblioteca de comunicações, as aplicações podem ser portadas para redes compostas por qualquer processador, desde que se disponha de um compilador ANSI C que os suporte. Por outro lado, para atingir tal portabilidade, a eficiência da implementação para processadores específicos não é optimizada. No entanto, como a aplicação está estruturada em níveis de abstracção, as seguintes optimizações ao nível do núcleo, permitiriam aumentar a eficiência de cada implementação, mantendo a portabilidade, ao custo de código do núcleo, diferente para cada processador:

  - Rescrever, para cada processador suportado, as rotinas computacionais do núcleo em código máquina, de modo a explorar eficientemente todas as facilidades de cada arquitectura, optimizando assim o tempo de cálculo.
  - Implementar estratégias de transferências de dados usando o co-processador de DMA nas famílias ADSP2106x e TMS320C4x, que permitam reduzir a perda de velocidade de cálculo enquanto se reencaminham mensagens.

– Em determinadas topologias, a largura de banda pode ser optimizada, desde que os processadores constituintes disponham de portos de comunicação extras. Deste modo, uma via de comunicação entre processadores pode ser implementada sobre mais do que um porto de comunicação, aumentando assim a largura de banda. Para se obter este cenário, fortemente dependente da topologia da rede paralela e das características dos processadores constituintes, será necessário redesenhar os mecanismos de encaminhamento de mensagens de modo que possam enviar uma mensagem para o mesmo destino por múltiplas comportas. No caso da família ADSP21060, que dispõe de 6 portos, a largura de banda anunciada poderia atingir os 6 x 40 MBytes.

- Ao nível da biblioteca de cálculo matricial existe ainda espaço para algumas optimizações:

  - Adaptação para suportar álgebra de complexos
  - Desenvolvimento de uma classe de funções de nível mais elevado para resolver problemas específicos, seguindo de alguma forma a filosofia do Lapack (Anderson et al, 1995) e Scalapack (Blackford et al, 1997).

  Outro caminho, consiste na substituição da biblioteca por outra mais completa, como é o caso da biblioteca Scalapack, e adaptar todo o SPAM em torno desta nova biblioteca.

- Substituindo os mecanismos de encaminhamento de mensagens, por outros apropriados, poderá estender-se o SPAM, de modo que também sejam geradas aplicações para o modelo de processamento distribuído, baseado nos padrões PVM (Geist et al., 1994) ou MPI (MPI Forum, 1994; 1997). Será assim possível dispor do poder de geração automática, de algoritmos matriciais paralelos, sem dispor de uma plataforma paralela, mas apenas de uma rede de computadores.

- Também substituindo os mecanismos de comunicação, criando uma arquitectura de passagem de mensagem virtual a um nível inferior, poder-se-á adaptar o SPAM para gerar código para uma arquitectura de memória partilhada.

- Em alguns casos, dependendo da arquitectura e propósito do sistema anfitrião, o processador desse computador apenas é utilizado para operações de entrada-saída, não efectuando nenhum ou pouco esforço computacional na maior parte do tempo. Visto que cada vez mais estes processadores são mais poderosos em termos de cálculo de vírgula flutuante, será conveniente no futuro alterar o sistema de distribuição de dados, e também o ambiente de comunicação, de modo que o anfitrião possa também colaborar no cálculo matricial.

- No caso de grandes redes heterogéneas irregulares, o caminho a tomar para enviar uma mensagem no menor espaço de tempo pode não ser evidente, e não automaticamente determinado pelos algoritmos tradicionais utilizados. É o caso de redes com nós onde o encaminhamento das mensagens tem um custo na capacidade de processamento, de modo que o tempo de comunicação não depende apenas do caminho mais curto, isto é do mais rápido, mas também da perda em termos de processamento útil dos nós que reencaminham os dados. Não é o caso dos Transputers como já foi visto, mas outros processadores, mesmo recorrendo a algumas estratégias de comunicação baseadas em DMA, podem sofrer uma perda, visível em termos de capacidade de cálculo.
  Uma solução consiste no desenvolvimento de um algoritmo, possivelmente genético, que optimize estes percursos.

- Correntemente a linguagem SEQ 1.0 constituí o *front-end* para o SPAM. Seria talvez conveniente adaptar o SPAM para suportar outros *front-ends* mais convencionais, como é o caso do C, Fortran ou Pascal. Para tal o processo de tradução deveria ser dividido em dois níveis. O mais próximo do utilizador traduziria o código sequencial de alto nível, para uma linguagem simples orientada para o paralelismo ao nível dos dados. O nível seguinte seria encarregado de traduzir esta última para a linguagem C paralela referente ao compilador para o *hardware* alvo.

- Actualizações em termos de operadores à versão 1.0 da linguagem SEQ :

  - Como foi visto para o segundo caso de teste, a linguagem SEQ não dispõe de operadores que suportem operações elemento a elemento entre matrizes, sendo necessário desenvolver algumas funções para o fazer. Estes operadores devem tratar

matrizes como tabelas, de modo que o operadores na posição correspondente, de duas matrizes, são operados e o resultado será colocado na posição correspondente de uma terceira matriz resultado.

Linguagens como o Matlab (The Math Works Inc., 1997a), usam um ponto antes do operador para indicar este tipo de operações.

- O operador expoente para um escalar não está implementado na versão corrente da linguagem. Este pode ser implementado como uma série de somas da base. A sua eficiente implementação em paralelo é trivial se cada nó processar um número de multiplicações da base por si própria, dado por expoente / número de nós. Estes resultados parciais devem ser enviados para cada nó de modo que em cada um sejam somados, obtendo-se o resultado final em todos os nós.

- Quando a plataforma alvo não está disponível ou quando existem várias plataformas candidatas a suportar uma determinada aplicação, é conveniente ter uma ideia aproximada do desempenho da aplicação sobre cada uma destas plataformas. Partindo das seguintes premissas:

  – O tempo de computação de cada processo é proporcional ao número de operações de vírgula flutuante que este executa;
  – O tempo de comunicação entre processadores é composto por um tempo de inicialização do canal de comunicação e pelo tempo necessário á transmissão de um vector de elementos atómicos sobre esse canal;

  as quais são constantes para cada tipo de processador. Foram desenvolvidos simuladores para plataformas especificas, tais como Transputers (Ruano, 1996) e TMS320C40 / Transputer (Sustelo e Ruano, 1998), os quais permitem obter o desempenho aproximado de uma aplicação sobre essa plataforma. Poder-se-ía, no entanto, seria generalizar este conceito de modo a desenvolver uma ferramenta, a incluir no SPAM, que efectue um simulação razoável de qualquer plataforma alvo, incluindo plataformas distribuídas.

  Ao longo desta tese, foi colhida informação não apenas relativa aos *quanta* de processamento e comunicação mas também relativas ao desempenho de operações matriciais básicas, as quais podem também ser usadas no desenho de um simulador.

  Por outro lado pode também ser necessário tomar em conta que comunicação e cálculo concorrente traduzem uma degradação do desempenho.

- Nos casos de estudo foi apresentado apenas um algoritmo de treino *off-line* de redes neuronais, mas algoritmos *on-line* de janela deslizante devem apresentar uma eficiência semelhante, como foi abordado no capítulo 8. Um bom candidato para avaliar o desempenho do SPAM, para esta classe de consiste num algoritmo desenvolvido para controlo de estufas (Ferreira et al. 2001).

  Também para além do treino de um MLP, algoritmos de treino de outros tipos de rede neuronal podem ser programados, se bem que possivelmente seja necessário adicionar algumas funções de cálculo às existentes nas bibliotecas que acompanham a presente versão do SPAM. De facto a implementação de algoritmos de treino de outros tipos de redes pode mesmo ser usada para sugerir quais as funções a incluir numa actualização das bibliotecas do SPAM. Para explorar por exemplo a adaptação de B-Splines (Brown e Harris 1994) pode ser implementado o algoritmo de sintonia do controlador PID assistido por uma destas redes, proposto em (Ruano e Azevedo, 1999), (Lima e Ruano, 2000) e desenvolvido em (Ruano et al., 2002).

  Para além das redes neuronais, podem também ser implementados outros tipos de algoritmos, com o intuito de recolher informação para o desenvolvimento das bibliotecas do SPAM. Uma classe de algoritmos que têm vindo a ser utilizados, são os algoritmos evolutivos. A implementação do algoritmo neuro-genético de sintonia do controlador PID apresentada em (Ruano e Azevedo, 1999) e (Lima e Ruano, 2000) permitiria também avaliar as bibliotecas para aplicações de controlo.

  Para avaliar o desenvolvimento de aplicações de controlo de tempo real, poderá ser implementado no SPAM o sistema de aquisição de dados em tempo real proposto por (Ferreira et al., 2000).

  Apenas algumas classe de algoritmos foram sugeridas. Muitas outras áreas poderão ser exploradas, não relacionadas com a engenharia de controlo, para recolher informação para o desenvolvimento das bibliotecas de base do SPAM.

Bibliografia

3L Ltd. (1991). *Parallel C User Guide*. 3L Ltd.

3L Ltd. (1995). *Parallel C User Guide - Texas Instruments TMS320C40.* 3L Ltd.

3L Ltd. (1998). *Diamond User Guide – Analog Devices ADSP-2106x.* 3L Ltd.

Aho, A., j. Hopcroft e J. Ullman (1983). *Data Structures and Algorithms.* Addison – Wesley

Aho, A., R. Sethi e J. Ullman (1986). Compilers, principles, techniques and tools. Addison – Wesley

Analog Devices (1997). *ADSP-2106x SHARC™ User's Manual*. Analog Devices.

Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. Mckenney, S. Ostrouchov e D. Sorensen (1995). *Lapack User's Guide*, SIAM.

Asirvadam, V. S. (2002). *On-line learning and construction for neural networks*. Tese de Doutoramento, Universidade de Belfast

Aspray, W. (1991). *John von Neumann and the Origins of Modern Computing (History of Computing)*. MIT Press

Aström, K. J. e B. Wittenmark (1989). *Adaptive Control*. Addison – Wesley

Backus, J. W. (1960). The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. *International Conference on Information Processing*. Unesco

Bal, H. e D. Grune (1994). *Programming Language Essentials*. Addison – Wesley

Bar-Noy, A. e Shlomo Kipnis (1994). Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems, *Mathematical Systems Theory, vol. 27, No. 5, 431-452*

Bar-Noy, A., S. Guha, J. Naor, e B. Schieber (1998). Multicasting in Heterogeneous Networks, *STOC 98*.

Berson, A. (1996). *Client/Server Architecture*. McGraw Hill.

Bertsekas, P. e J. Tsitsiklis (1989). *Parallel and Distributed Computation*. Prentice-Hall

Bierman, G. J. (1977). *Factorization methods for discrete sequential estimation.* Academic Press

Björck, Å (1996). *Numerical methods for least squres problems.* SIAM

Blackford, L. S., J. Choi, A. Cleary, E. D'Azevedo e J. Demmel (1997). *Scalapack User's Guide*. Society for Industrial & Applied Mathematics

Borodin, A., Y. Rabani, e B. Schieber (1997) Deterministic Many-to-Many Hot Potato Routing. IEEE transactions on Parallel and Distributed System, Vol. 8, No. 6, 587-595

Brown, M., C. Harris (1994). Neurofuzzy Adaptive Modelling and Control, Prentice Hall

Bruck, j., C-T. Ho, S. Kipnis, E. Upfal e D. Weathersby (1997). Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 11, 1143-1156

Burks, A., H. Goldstine e J. von Neumann (1947). Preliminary discussion of the logical design of an electronic computing instrumemt. In *John von Neumman: Collected Works*, Vol. V, Macmillan

Burns, A. (1988). *Programming in Occam 2*. Addison - Wesley

Camacho, E. F. e J. C. Bordons (1994). *Model Predictive Control in the Process Industry.* Springer Verlag

Cunha, J. B. (2002). *Estudo e Desenvolvimento de Solução de Instrumentação e Controlo aplicado a Estufas de Produção Agrícola*, Tese de Doutoramento, Universidade de Trás-os-Montes e Alto Douro

Chang, H. e W. Oldham (1995). Dynamic Task Allocation Models for Large Distributed Computing Systems, *IEEE transactions on Parallel and Distributed System*, Vol. 6, No. 12, 1301-1315

Chao, L., A. LaPaugh e E. Sha (1993) Rotation Scheduling: A Loop Pipelining Algorithm, *Proc ACM / IEEE Design Automation Conference*

Charny, B.(1996). Matrix Partitioning on a Virtual Shared Memory Parallel Machine, *IEEE transactions on Parallel and Distributed System*, Vol. 7, No. 4, 343-355

Chen, H. e F. Allgöwer (1997). A QUASI-INFINITE HORIZON NONLINEAR MODEL PREDICTIVE CONTROL SCHEME WITH GUARANTEED STABILITY. *Proc European Control Conference (ECC 97)*

Clarke, D. W., C. Mohtadi and P. S. Tuffs (1987a) Generalized Predictive Control - Part I. The Basic Algorithm. *Automatica,* 23, 137 - 148

Clarke, D. W., C. Mohtadi e P. S. Tuffs (1987b). Generalized Predictive Control - Part II. Extensions and Interpretations. *Automatica*, 23, 149 - 160

Clarke, D. W. e P. J. Gawthrop. (1975). Self-tunning controller. *Proc. IEE*, 122, 929-934

Codenotti, B. e M. Leoncini (1992). *Introduction to Parallel Processing*, International Computer Science Series, Addison-Wesley

Comer, D. (1988). *Internetworking with TCP/IP: Principles, Protocols, and Architecture.* Prentice-Hall International

Cooper, D. (1983). *Standard Pascal: User Reference Manual.* W.W. Norton & Company

Costa B. M. (1996). *Controlo predictivo de horizonte estendido em tempo contínuo: Os algoritmos adaptativos CTMUSMAR e δMUSMAR*, Tese de Doutoramento, Instituto Superior Técnico

Culler, D., R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian e T. von Eicken (1993). LogP: Towards a Realistic Model of Parallel Computation, proc. 4th Sigplan Symp. *Principles and Practicies Parallel Programming.* ACM.

Daniel, H. e A. Ruano (1996). Parallel Implementation of an Adaptive Generalized Predictive Control Algorithm. *Proc 2nd International Meeting on Vector and Parallel Processing (VECPAR 96)*

Daniel, H. e A. Ruano (1997a). Adaptive Generalized Predictive Control Algorithm Implemented over an Heterogeneous Parallel Architecture. *Proc 14th IFAC International Workshop on Distributed Computer Control Systems (DCCS 97)*, 185-190

Daniel, H. e A. Ruano (1997b). Adaptive Generalized Predictive Control Algorithm Implemented over a DSP Network. *Proc 8th Annual International Conference on Signal Processing Applications and Technology (ICSPAT 97)*, Vol. I, 35-39

Daniel, H. e A. Ruano (1999a). Automatic parallelization of matricial algorithms. *Proc 14th IFAC World Congress (IFAC 99)*. Vol. Q, 453-458

Daniel, H. e A. Ruano (1999b). Performance comparison of parallel architectures for real-time control. *Microprocessors and Microsystems*, 23 (1999), 325-336

Daniel, H. e A. Ruano (2000). Automatic parallelization of an Adaptive Generalized Predictive Control Algorithm using MAPS 1.0 Environment, *Proc. 6th IFAC/IFIP Workshop on Algorithms and Architectures for Real - Time Control (AARTC 2000)*

Daniel, H., A. Ruano e P. Fleming (1997). Implementation of an Adaptive Generalized Predictive Control Algorithm over an Heterogeneous Parallel Architecture. *Proc 4th IFAC Workshop on Algorithms and Architectures for Real - Time Control (AARTC 97)*, 287 – 292.

Daniel, H. e S. Baltazar (1994). *Paralelização do método de Gauss-Jordan de resolução de sistemas de equações lineares*. Projecto de Licenciatura, U.AL.

De Moura Oliveira, P. B. e Jones, A. H. (1998). Cooperative co-evolutionary multivariable system identification using structured genetic algorithms. *Proc of AMST 98*, 149-158

Demircioglu, H e P. J. Gawthorp (1991). Continuos time-generalised predictive control. *Automatica*, 27(1), 55-74

Dourado, A. (1998). Learning in Intelligent Control, *Cosy Annual Meeting, Ohrid, Macedonia*.

Drake, C. (1998). *Object-Oriented Programming with C++ and Smalltalk*, Prentice Hall

Fernando, K. V. e H. Nicholson (1985) Identification of linear systems with input and output noise: the Koopmans-Levin method. *Proc. IEE*, 132, 30-36

Ferreira, P. M., E. A. Faria e A. E. Ruano (2000). Design and implementation of a real-time data acquisition system for the identification of dynamic temperature models in a hydroponic greenhouse, *Acta Horticulturae*, ISHS, No. 519, 191-199

Ferreira, P. M., E. A. Faria e A. E. Ruano (2001) Neural Network Models in Greenhouse Environmental Control, *Neurocomputing*, Special Issue on Engineering Applications of Neural Networks, Vol. 43, No. 1-4, 2001, 51-75.

Fletcher, R. (1987). *Practical Methods of Optimizatio*n. John Wiley & Sons

Fraigniaud, P. e E. Lazard (1994). Methods and Problems of Communication in Usual Networks. *Discrete Applied Math.,* Vol. 53, 79-133

Gawthorp, P. J e I. Siller-Alcada (1996). *Multivariable continuos-time generalised predictive control: A state space approach*. Technical report, Center for Systems and Control, Glasgow University

Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek. e V. Sunderam (1994) *PVM: Parallel Virtual Machine*, MIT press

Gill, P., W. Murray e M. Wright (1981). *Pratical Optimization*, Academic Press

Greco, C., G. Menga, E. Mosca e G. Zappa (1984). Performance improvements of self-tunning controllers by multistep horizons: The MUSMAR. *Automatica*, Vol. 20, 334-341

Hassoun, M. (1995). *Fundamentals of ARTIFICIAL NEURAL NETWORKS*. The MIT press

Heesterman, A. (1983). *Matrices and Simplex Algorithms: A Textbook in Mathematical Programming and Its Associated Mathematical Topics*. D Reidel Pub Co.

Hoare, C. (1985) *Communicating sequential processes*. Prentice-Hall

Hogan, T. (1988), *The programmer's PC sourcebook*. Microsoft Press

Horowitz, E., S. Sahni e S. Anderson-Freed (1993). *Fundamentals of Data Structures in C*, Computer Science Press.

Hsu, D. F., e D. S. L. Wei (1997). Efficient Routing and Sorting Schemes for de Bruijn Networks. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 11, 1157-1170

Iba, H e T. Kurita, (1993). System Identification using structured genetic algorithms. *Proc of 5th International Conf. on Genetic Algorithms,* 279-286

Inmos Ltd. (1990a). *Transputer - Transputer Architecture and Overview.* CSA - Computer System Architects, Inmos Ltd.

Inmos Ltd. (1990b). *Transputer - Transputer Technical Specifications*. CSA - Computer System Architects, Inmos Ltd.

Inmos Ltd. (1990c). *IMS B008 User guide and reference manual*. Inmos Ltd.

Jovian Systems Inc. (1997). *Pegasus Design Environment*. Jovian Systems Inc.

Lima, J. M. e A. E. Ruano. (2000). Neuro-genetic PID autotuning: Time invariant case. *IMACS Journal of Mathematics and Computers in Simulation*, Vol. 51, 287-300

Kaufmann, A. (1976). *Des points et des flèches: la théorie des graphes*. Dunod

Kernighan, B. W. e D. M. Ritchie (1988), *The C programming language*. Prentice-Hall

Kohonen, T. (1988). Self-organization and associative memory, 2ª Ed. Springer-Verlag

Kristinn, K. e A. D. Guy (1992). System Identification and control using genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics,* Vol.22, No. 5

Lee, S. J., S. H. Park, C. H. Kim, C. S. Ham e J. H. Lee (1997). Generalized Predictive Control with Input Constraints. *Proc European Control Conference (ECC 97)*

Maciejowski, J.M. (2001). *PREDICTIVE CONTROL with Constraints.* Pearson Education, Prentice Hall.

Menga, G e E. Mosca (1980). MUSMAR: multivariable adaptive regulators based on multistep cost functionals. *Advances in Control*, D. G. Lainiotis and N. S. Tzannes (Eds)D. Reidel Pu. Co.

Meyer, B. (1997). Object-Oriented Software Construction, Prentice Hall

MPI Forum (1994). *MPI: A message-passing interface standard*, University of Tennessee

MPI Forum (1997). *MPI-2: Extensions to the Message-Passing Interface*, University of Tennessee

Naor, J., A. Orda e R. Rom (1998). Scheduled Hot-Potato Routing, *Jornal of Graph Algorithms and Applications*, Vol. 2, No. 4, 1-20

Naur, P. (1963). Revised report on the algorithmic language Algol 60. *Comm. ACM* 6:1, 1-17

Nyhoff, L. e S. Leestma (1996). *Fortran 77 for Engineers and Scientists: With an Introduction to Fortran 90.* Prentice Hall

Ordys, A. W. e D. W. Clarke. (1993) A state-space description for GPC controllers, *Int. J. Systems Science*, Vol. 23, No. 2, 1993.

Ordys A. W e A. W. Pike. (1998) State-space generalized predictive control incorporating direct through terms, *Proc. 37th IEEE Control and Decision Conference*1998.

Paxson, Vern (1990). *Flex version 2.5, A fast scanner generator*. University of California

Piedra, R. (1991), *A parallel Approach for Solving Matrix Multiplication on the TMS320C4x DSP*. Texas Instruments, Inc.

Plackett, R. L. (1950) Some theorems in least squares, *Biometrika,* 37, 149-157

Prasanna, G. e B. Musicus (1996). Generalized Multiprocessor Scheduling and Applications to Matrix Computations, *IEEE transactions on Parallel and Distributed System*,Vol. 7, No. 6, 650-664

Richalet, J., J. L. Testud and J. Papon (1978), Model Predictive heuristic control: applications

to industrial processes, *Automatica*, 14, 413-428

Rim, M.e J. Rajiv (1996) Valid Transformations: A New Class of Loop Transformations for High-Level Synthesis and Pipelined Scheduling Applications, *IEEE transactions on Parallel and Distributed System*, Vol. 7, No. 4, 399-410

Ruano, A. (1992). *Applications of neural networks to control systems*. PhD Thesis, Scholl of Electronic Engineering Science, University of Wales, Bangor.

Ruano, A. (1996). A Matlab Toolbox for Simulating Transputer Applications, *Proc 2nd International Meeting on Vector and Parallel Processing (VECPAR 96)*

Ruano, A. (2002). *Apontamentos da cadeira de Redes Neuronais*. UAlg [http://fourier.fct.ualg.pt/web-ct/courses/Redes_Neuronais]

Ruano, A. E. e A. B. Azevedo. (1999) B-Splines neural networks assisted PID autotuning. *International Journal of Adaptive Control & Signal Processing*, Vol. 13, No. 4, 291-307.

Ruano, A. E., C. Cabrita, J. V. Oliveira, L. T. Kóczy (2002). Supervised Training Algorithms for B-Spline Neural Networks and Neuro-Fuzzy Systems, International Journal of Systems Science, 33, 8, 689-711

Ruano, A., D. Jones e P. Fleming (1991). A new formultion of the learning problem for a neural network controller. *30th IEEE CDC*, Vol. I, 865-866

Ruano, A., D. Jones and P. Fleming (1992). Parallel implementation of a learning algorithm for multilayer perceptrons using transputers. *PARALLEL COMPUTING AND TRANSPUTER APPLICATIONS*, 1395-1404

Ruano, A. e H. Daniel (1997). Parallel Implementation of an Adaptive Generalized Predictive Control Algorithm. Proc European Control Conference (ECC 97)

Rumelhart, D., McClelland, J. e The PDP Research Group (1986). Parallel distributed processing, Vols. 1 e 2. MIT Press

Sarkar, V. (1989). *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London; The MIT Press

Sethi, R. (1996). *Programming lamguages: concepts & constructs*, 2nd edition. Addison – Wesley

Snir, M., S. Otto, S. Huss-Lederman, D. Walker e J. Dongarra (1995). *MPI: The Complete Reference*, MIT Press

Söderström, T. e P. Stoica, (1989). *System Identification.* Prentice Hall

Spectrum Signal Processing Inc. (1998). *Processing SharcPAC Users Manual*. Spectrum Signal Processing Inc.

Stewart, G. (1973). *Introduction to matrix computations.* Academic press

Stevens W. R. (1998). *UNIX Network Programming, Volume 1: Networking APIs - Sockets and XTI*, Prentice-Hall

Stevens W. R. (1999). *UNIX Network Programming, Volume 2: Interprocess Communications*, Prentice-Hall

Strange, P. (1989) Basic brain mechanisms: a biologist's view of neural networks', IEEE Colloquium on Current issues in neural networks research, Digest Nº1989/83, 1-5

Su, M., G. Chen e D. Duh (1997). A Shortest-Path Routing Algorithm for Incomplete WK-Recursive Networks. *IEEE transactions on Parallel and Distributed System*, Vol. 8, No. 4, 367-379

Sun Microsystems, Inc., 1987. *XDR: External Data Representation Standard*. RFC 1014, Sun Microsystems, Inc.

Sustelo, F. e A. E. Ruano (1998). A Matlab Toolbox for simulating Transputer and Digital Signal Processing Applications, *Proc 5th IFAC Workshop on Algorithms and Architectures for Real - Time Control (AARTC 98)*, 167-172

Tan, M. e J. Antonio (1997). Minimizing the Application Execution Time Through Scheduling of Subtask and Communication Traffic in a Heterogeneous Conputing System, *IEEE transactions on Parallel and Distributed System*, Vol. 8, No. 8, 857-871

Texas Instruments (1996). *TMS320C4x User's Guide.* Texas Instruments

The Math Works Inc. (1997a). *Using Matlab*, The Math Works Inc.

The Math Works Inc. (1997b). *Using Simulink*, The Math Works Inc.

The Math Works Inc. (1997c). *Matlab External Interface Guide*, The Math Works Inc.

Tokhi, M. O., M. A. Hossain, M. J. Baxter e P. J. Fleming (1995). Heterogeneous and homogeneous parallel architectures for real-time active vibration control. *IEEE Proc Control Theory Appl.*, Vol. 142, No. 6, 625-632

Transtech Paralell Systems Ltd. (1991a). *Transtech TMB04 – PC Transputer board.* Transtech Paralell Systems Ltd.

Transtech Paralell Systems Ltd. (1991b). *TMB16 Hardware – 16-bit PC TRAM board.* Transtech Paralell Systems Ltd.

Transtech Paralell Systems Ltd. (1991c). *TMB16 Software.* Transtech Paralell Systems Ltd.

Valiant, L. G. (1990). A Bridging Model for Parallel Computation, Comm. *ACM*, vol. 33, No. 8, 103-111

Warnes, P. (1993). *Hunt Engineering HEPC2-M TIM Motherboard for AT bus User Manual*. Hunt Engineering

Warnes, P. (1994). *Hunt Engineering HET40SDX TIM SRAM/DRAM Processing Module User Manual*. Hunt Engineering

Weinberg, S. (1992). *Sonhos de uma teoria final*. Gradiva

Weisstein, E. (1999), *The CRC Concise Encyclopedia of Mathematics*. CRC Press LLC

Wellstead, P. e M. Zarrop (1991) *Self-Tunnig systems*. John Wiley & sons

Williams, R. (1995). *Hunt Engineering HET403tl Transputer Link Interface TIM-40 Module User Manual*. Hunt Engineering

Wirth, Niklaus. (1983). *Programmung in Modula-2*. 2nd edition. Springer-Verlag

Apêndice A – Pormenores técnicos sobre SPAM 1.0

A.1 Configuração do SPAM 1.0 e AIDA

A corrente versão do AIDA foi desenhada para a família de sistemas operativos WINDOWS de 32 bits, produzida pela Microsoft Corporation, e que incluí WINDOWS 9x e WINDOWS NT 4.0. Assim, para integrar todas as ferramentas de desenvolvimento neste ambiente, será necessário configurá-lo, dependendo do sistema operativo e das aplicações instaladas no computador anfitrião. Na janela apresentada na figura seguinte, acedida a partir do menu Misc item paths, devem ser indicadas as localizações das ferramentas externas requeridas pelo AIDA: o configurador visual, um editor de texto ASCII da preferência do utilizador e um gestor de projectos compatível com o NMAKE da Microsoft Corporation, tal como ou como o MAKE da Borland International, ferramentas externas requeridas pelo AIDA. Nesta janela, deve ainda ser especificado o caminho para a raiz do SPAM 1.0.

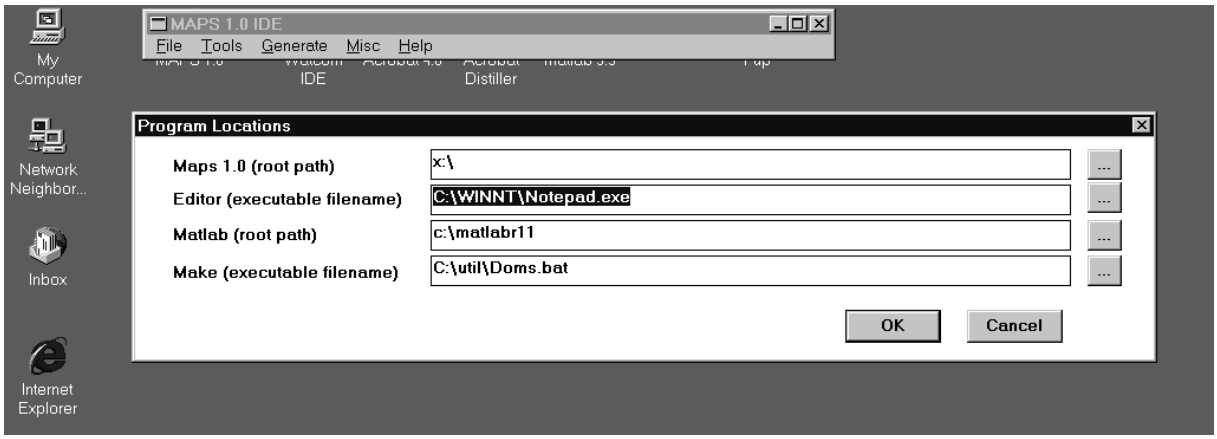


Fig. A-1: Localização das ferramentas externas requeridas pelo SPAM 1.0

Na figura acima pode-se ver especificada a raiz do MATLAB, visto que o SIMULINK é utilizado para descrever visualmente a rede de processadores alvo; o editor de texto padrão de qualquer versão do sistema operativo WINDOWS, o NOTEPAD; e um utilitário de gestão de projectos, compatível como o NMAKE.

No ficheiro maps10.ini, situado no mesmo directório que o executável, é guardada esta informação, na forma de texto ASCII na seguinte ordem:

- Editor de texto (EDIT_TOOL = Notepad.exe)
- Configurador visual (VC_TOOL = Matlab)
- Caminho para raiz SPAM (MAPS_PATH)
- Ferramenta de gestão de projectos (MAKE_TOOL = nmake)

A segunda coluna indica a variável que referencia cada linha e o seu valor por defeito.

A.2 Definições de cada projecto

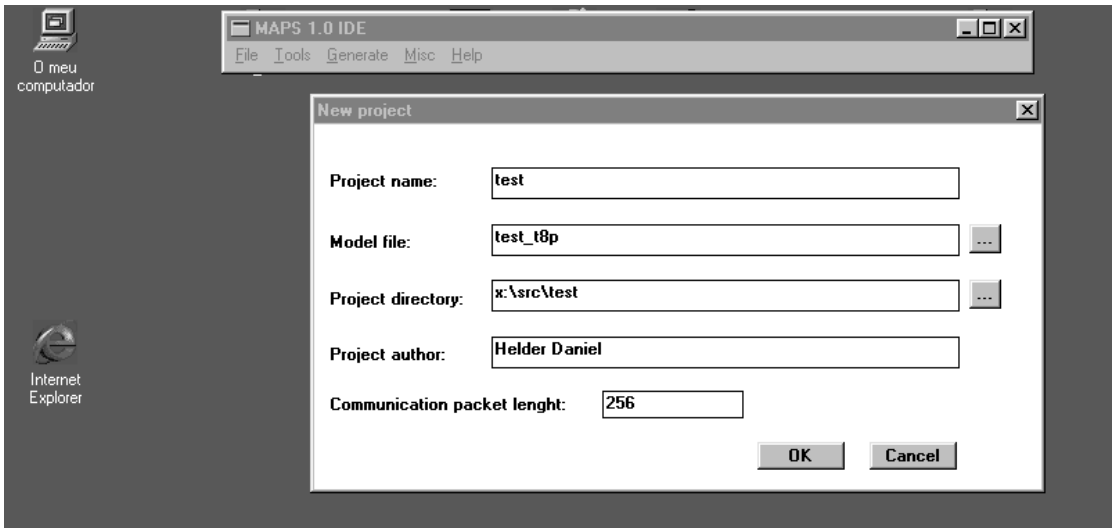


Fig. A-2: Definições de projecto

Além da configuração de base do ambiente, um projecto tem algumas definições que devem ser indicadas quando da sua criação, ou modificas posteriormente a partir do menu File, item New/Change, como indicado na figura acima. São essas o nome, que em certos sistemas operativos MS-DOS ou MS-WINDOWS pode estar limitado a 7 caracteres e a localização do ficheiro com o modelo da topologia alvo. Este parâmetro deve ser mudado sempre que se pretender gerar uma aplicação para uma diferente topologia. É também necessário indicar a directoria do projecto e o nome do autor, o qual será incluído nos ficheiros fonte gerados pelo

SPAM 1.0. Finalmente é necessário indicar o comprimento dos pacotes de dados que circularam na rede em palavras de 32 bits. Este valor será usado também para alocar *buffers* na inicialização da aplicação. O valor por defeito é 256.

No ficheiro <nome do projecto>.mpj é guardada a informação indicada acima, na forma de texto ASCII na seguinte ordem:

- Nome do projecto (máximo 7 caracteres em sistemas operativos baseados no MS-DOS e MS - Windows) (PROJECT_NAME)
- Modelo de rede em simulink (PROJECT_CONFIG [*.mdl])
- Caminho para projecto (PROJECT_PATH)
- Autor do projecto (PROJECT_AUTHOR)
- Comprimento do pacote de comunicação (PACKET_LEN)

A segunda coluna indica a variável que referencia cada linha.

A.3 Outras definições

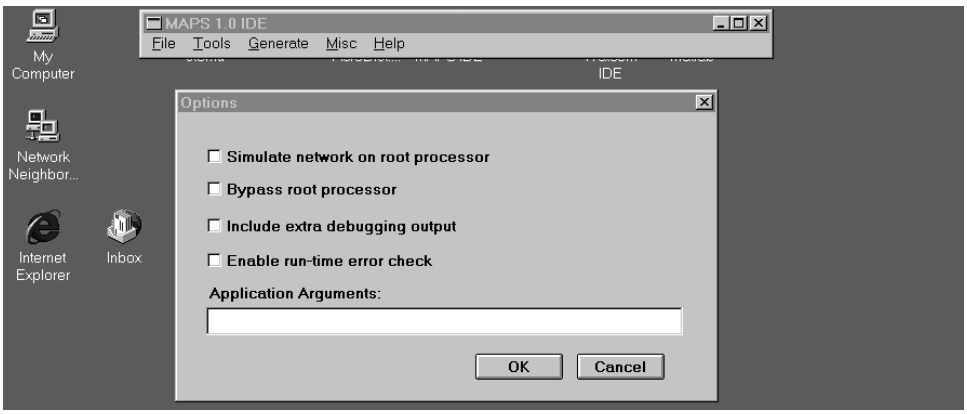


Fig. A-3: Opções de geração de código e argumentos da linha de comandos

O item Options, do menu Misc, permite aceder ainda a uma janela de opções, onde se incluem a possibilidade de indicar ao configurador para gerar uma aplicação que simula uma rede num só processador ou não usar o processador raiz para cálculo. É ainda possível gerar código que envia para a consola informação relevante para depuração, bem como habilitar a gestão de erros em tempo de execução. Estas duas últimas opções vão no entanto implicar uma perda de desempenho. Quanto às duas primeiras opções, mutuamente exclusivas, podem revelar-se muito úteis quando do desenvolvimento de uma aplicação. A primeira, permite o desenvolvimento e ensaio de aplicações para uma qualquer topologia, quando se dispõe de

apenas um processador. Por outro lado, a segunda opção poderá ser útil no caso de uma rede homogénea, onde o processador raiz, embora da mesma família que os restantes, difere em pequenas características tais como quantidade de memória interna, ou número de ligações disponíveis. Assim, se se pretender avaliar o desempenho de uma rede absolutamente homogénea, o processador raiz não deve ser utilizado.

Além das opções acima referidas, existe ainda a possibilidade de indicar quais os argumentos de linha de comando que a aplicação irá receber. Esta facilidade será também útil no caso de se estar na presença de uma carta mãe para transputers, dotada de um comutador de ligação IMSC004, como é o caso da TMB08 e da TMB16. Neste caso, a necessidade de configurar esse comutador usando um ficheiro adequado para tal "*.wir", implica que como primeiro argumento deve ser indicado o caminho para o esse ficheiro de configuração, e só após os argumentos requeridos pela aplicação.

A.4 Localização dos compiladores

A localização das ferramentas de compilação, adequadas para cada arquitectura alvo, e sua configuração, é discriminada num ficheiro que constitui a base de qualquer ficheiro de gestão de projecto, "mb.mmf". A seguir é apresentado este ficheiro por defeito:

```
C40path    = c:\tic2v0
T8path     = c:\tc2v2
ADSPpath   = c:\adc1v0

C4LINK     = $(C40path)\lnk30 -q -ar -c
C4CONFIG   = $(C40path)\config
C4CC       = $(C40path)\cl30

C4LINKOPT  = -lc40cfbr.lib
C4LINKrOPT = -lc40csbr.lib
C4CONFOPT  = /c /m /v
C4CCOPT    = -qq -v40 -mb -mi -mr -x2 -p? -o3 -I$(LIB) -DC4

T8INC      = $(T8path)\CC
T8LINK     = $(T8path)\linkt
T8CONFIG   = $(T8path)\config
```

```
T8CC      = $(T8path)\t8cc

T8LINKOPT  = $(T8path)\libct8 $(T8path)\crtlt8   $(T8path)\taskharn.t8
T8LINKrOPT = $(T8path)\libct8 $(T8path)\sacrtlt8 $(T8path)\taskharn.t8
T8CONFOPT  =
T8CCOPT    = /I$(LIB) /I$(T8INC) /DT8

SHLINK     = $(ADSPpath)\slink
SHCONFIG   = $(ADSPpath)\config
SHCC       = c:\adi_dsp\bin\g21k

SHLINKOPT  = -L$(ADSPpath) -lrtl
SHLINKrOPT = -L$(ADSPpath) -lsartl
SHCONFOPT  = /v
SHCCOPT    = -O3 -c -I$(ADSPpath) -I$(LIB) -D_SHARC -D_3L -DSHARC
```

Qualquer alteração às localizações e configurações destas ferramentas, deve estar patente neste ficheiro.

A.5 Definições da aplicação

Quando uma aplicação gerada pelo SPAM 1.0 é inicializada necessita de saber qual o *Quantum* de processamento de cada nó da rede e qual a tabela de encaminhamento a utilizar pelo ambiente de comunicação. Essa informação encontra-se no ficheiro <nome do projecto>.mif, na seguinte ordem:

- *Quantum* de processamento (1 float por cada processador na rede)
- Tabelas de encaminhamento de mensagens, **TEG** e **TEP**, descritas no capítulo 4.

A.6 Breve descrição da operação das ferramentas

A.6.1 Configurador

Esta ferramenta gera o ficheiro de configuração da rede paralela, na sintaxe esperada pelo compilador alvo, e um ficheiro de gestão de projecto no formato esperado pela ferramenta

NMAKE, a partir de um modelo esquemático dessa rede, no formato do SIMULINK 3.0 e 4.0 da Mathworks.

- Nome do ficheiro executável:

conf.exe

- Sintaxe na linha de comandos:

conf <nome do projecto>.mpj <caminho para bibliotecas>
<normal, simular rede ou *bypass root* = 0 | 1 | 2> <informação depuração = 0 | 1>

- Ficheiros de entrada:

<nome projecto>.mdl {Modelo da rede (topologia)}
mb.mmf {Localizações dos compiladores e opções destes}

- Ficheiros de saída:

<nome projecto>.cf {Ficheiro de configuração da rede (descrição da topologia)}
<nome projecto>.mmf {ficheiro de gestão de projecto para nmake ou compatíveis}
<nome do projecto>.mif {Definições da aplicação (*Quantum* de processamento)}
<nome do projecto>.mif {Definições da aplicação (tabelas de encaminhamento)}

A.6.2 Tradutor

Gera o código C paralelo, de acordo com o compilador e os processadores alvo, para a tarefa Master e para as tarefas Slave, a partir do código sequencial SPAM 1.0.

- Nome do ficheiro executável:

trans.exe

- Sintaxe na linha de comandos:

trans <nome projecto>.mpj <caminho bibliotecas> <informação depuração = 0 | 1>
<tratamento erros tempo execução = 0 | 1>

- Ficheiros de entrada:

<nome projecto>.seq {Código fonte sequencial}

routersM.c {Ambiente de comunicação para tarefa Master (código C paralelo)}

routersS.c {Ambiente de comunicação para tarefas(s) Slave (código C paralelo)}

- Ficheiros de saída:

<nome projecto>M.c {Código C paralelo para tarefa Master}

<nome projecto>S.c {Código C paralelo para tarefa(s) Slave}

A.6.3 Interface com SIMULINK

Gera um ficheiro MATLAB que permite que a aplicação possa comunicar dados com o ambiente do MATLAB da Mathworks, através de bibliotecas dinâmicas desenvolvidas para esse efeito, as quais servem de portão entre este ambiente e um gestor de dispositivo adequado ao processador raiz.

- Nome do ficheiro executável:

matint.exe

- Sintaxe na linha de comandos:

matint <nome projecto>.mpj <caminho bibliotecas>

- Ficheiros de entrada:

<nome projecto>.seq {Código fonte sequencial}

- Ficheiros de saída:

<nome projecto>.m {Código matlab}

- Ficheiros utilizados em tempo de execução entrada:

T8int.dll {Biblioteca de acesso ao processador raiz (T805)}

A.7 Localização das ferramentas e outros componentes do SPAM 1.0

<raiz SPAM> \tools \conf \conf.exe {configurador}

\ide \maps10.exe {AIDA}

		\maps10.ini	{Definições do AIDA}
	\matint	\matint.exe	{gera interface SIMULINK}
	\trans	\trans.exe	{tradutor}
	\delexe.bat		{elimina temporários}
	\deltrash.bat		{elimina temporários}
	\runc40.bat		{executa aplicação em C40}
	\runcsharc.bat		{executa aplicação em ADSP2106x}
	\runt8c40.bat		{executa aplicação em T8 ou T8-C40}
\lib	\bmcalc.c		{Bib. Calc. Mat. funções básicas}
	\c40krn		{par de comportas por link nos w orkers}
	\com_varm.h		{variáveis básicas para master}
	\com_vars.h		{variáveis básicas para slave}
	\comp_opt.h		{Opções de compilação}
	\comonext		{extensões DOS comuns}
	\compat.h		{compatibiliza hardware e 3L C}
	\debug.h		{funções de depuração}
	\edit.h		{funções para tratamento de texto}
	\errordef.h		{Códigos de erro}
	\errormsg.h		{Gestão de erros durante a execução}
	\functions.dat		{Definições de funções para tradutor }
	\harddef.h		{Definições dependentes do hardware}
	\hio.h		{ protocolo HIO }
	\matmacro.h		{macros para Bib. Calc. Matricial}
	\maux.c		{Bib. Calc. Mat. funções auxiliares}
	\mb.mmf		{Caminhos para ferramentas de compilação}
	\mbd.mmf		{Caminhos para ferramentas de compilação - versão com opções de depuração…}
	\mcalc.c		{Bib. Calc. Matricial}
	\mcalcext.c		{Bib. Calc. Mat. extensões de utilizador}
	\mcomm.c		{Bib. de funções de comunicação}
	\mios.c		{Bib. Calc. Mat. Servidor de E / S}
	\mio.c		{Bib. Calc. Mat. funções de E / S}

\parext.h	{extensões a par.h de 3L C}
\Pelib.m	{Blocos Matlab para processadores}
\rengM.h	{Motor de encaminhamento (Master)}
\rengS.h	{Motor de encaminhamento (Slave)}
\rinitM.h	{difusão tabelas de encaminhamento}
\rinitS.h	{recepção tabelas de encaminhamento}
\rootc40.krn	{par de comportas por link no root}
\routersM.c	{Ambiente de comunicação (Master)}
\routersS.c	{Ambiente de comunicação (Slave)}
\seqbase.inc	{ Funções básicas SEQ 1.0}
\sintax.dat	{Diagramas de sintaxe para tradutor }
\T8int.dll	{Portão de acesso a uma rede (raiz T8)}
\timefun.h	{Medição de tempo de processamento}
token.dat	*{Lexemas para tradutor }*
\transext.h	{*interface* entre tradutor e bibliotecas}

A.8 Localização dos componentes de um projecto no âmbito do SPAM 1.0

<raiz projecto>	\<nome do projecto>.mdl	{Modelo da rede (SIMULINK)}
	\<nome do projecto>.mpj	{Definições do projecto}
	\<nome do projecto>.seq	{Código sequencial SPAM}
	\<nome do projecto>M.c	{Tarefa Master}
	\<nome do projecto>S.c	{Tarefa Slave}
	\maps_udf.h	{Funções e variáveis globais C, definidas pelo o utilizador, para tarefa Slave}
	\<nome do projecto>.m	{interface SIMULINK }
	\<nome do projecto>.app	{aplicação executável}
	\<nome do projecto>.mif	{Definições da aplicação}

A.9 Implementação das primitivas de comunicação

As primitivas de comunicação são implementadas de modo diferente dependendo de serem chamadas no processo de controlo ou num nó, embora a lista de argumentos de entrada seja idêntica. Nesta implementação é passada, em forma de argumentos, informação para formar o descritor, no caso de um envio, ou para retirar deste informação no caso de uma recepção.

Assim a primitiva send tem a seguinte sintaxe:

send (int *destino*, int *comprimento*, int *Float*, void **buffer*, int *espera*) (instância)
send (int *destino*, int *comprimento*, int *Float*, void **buffer*, int espera) (processo controlo)

onde *destino* representa o nó destino na gama [0, N_WRK-1] ou DST_MASTER ou DST_ALL; *comprimento* indica o número de elementos a enviar (Float / int); *Float* toma o valor verdadeiro ($\neq 0$) se os dados forem números de vírgula flutuante e falso (= 0) caso sejam inteiros; *buffer* é um ponteiro para os dados a enviar e finalmente *espera* será verdade para que a primitiva seja bloqueante. Este argumento no entanto não tem o mesmo significado no processo de controlo, pois send aguarda sempre que a mensagem seja enviada para o nó 0. Assim, neste caso, se espera for falso obriga que o bit MSG_END do cabeçalho do último pacote não seja seleccionado, fazendo com que não haja indicação de fim de mensagem, podendo ser usado para concatenar duas mensagens. Isto será útil para efectuar operações de entrada saída com o anfitrião, para mensagens com comprimento superior ao *buffer* do processo de controlo, de modo que as instâncias recebam os vários pacotes pertencentes à mesma mensagem com apenas uma chamada a receive. Se tal não fosse possível, seriam necessário tantas chamadas a receive, quantas vezes a mensagem fosse maior que PACKET_LEN.

receive (int **comprimento*, int **fonte*, void **buffer*) (processo controlo)
receive (int **emissor*, int *fonte*, void **buffer*) (instância)

onde *comprimento* indica o número de elementos recebidos, reais ou inteiros, no caso do processo de controlo, mas que não é utilizado nos nós; *fonte* indica a origem da mensagem, no caso do processo de controlo, ao passo que no caso de uma instância da aplicação, indica que se deve aguardar por uma mensagem daquela *fonte*; Nestas, *emissor* indica qual a fonte da mensagem, e será sempre igual à pedida, isto é *fonte*, excepto se se esperar por uma mensagem de qualquer nó, o que é efectuado atribuindo a *fonte* o valor SRC_ANY. Finalmente *buffer* é um ponteiro para um endereço de memória onde serão armazenados os dados após recepção.

Para se determinar o comprimento de uma mensagem recebida no caso das instâncias, deve ser lida a tabela **TCM** [*fonte*] após a recepção de todas as *n* mensagens, seguindo a forma:

```
sema_wait_n (&SFR, n);
comprimento0 = TCM [fonte];
(...)
comprimento4 = TCM [fonte];
(...)
```

Se a fonte for uma qualquer, isto é SRC_ANY, não é necessário aguardar explicitamente no semáforo SFR, pois neste caso receive é bloqueante.
Se por outro lado só existir uma mensagem a receber pode-se usar a macro:

RECEND (*comprimento*, *fonte*)

que deixa em *comprimento* o correspondente à mensagem recebida da *fonte*, após aguardar no semáforo SFE.

Legenda	**Significado**
MAX_LINKS	{O número de portos bidireccionais máximos num só nó ou processador. Para o *hardware* suportado toma o valor dos 6 portos bidireccionais do ADSP2106x }
NET_PORT	0
MASTER_PORT	MAX_LINKS
from_net	in [NET_PORT]
to_net	out [NET_PORT]
from_master	in [MASTER_PORT]
to_master	out [MASTER_PORT]
MAX_WRK	Máximo número de nós numa rede (correntemente 126)
WRK	{x como em Prg(x)}
N_WRK	{Número de nós na rede: 0 < x < N_WRK}

tabela A-1: Descrição das legendas dos diagramas de processos dos mecanismos de comunicação

A tabela anterior descreve as legendas usadas nos mecanismos de comunicação, bem como algumas macro usadas na implementação.

Falta ainda referir, que no caso do processo de controlo, existe uma variável global, identificada por HEAD, a qual guarda o cabeçalho da última mensagem recebida.

A.10 identificadores reservados

Os seguintes identificadores são usados pelas bibliotecas e pelo ambiente de comunicação, de modo que não podem ser redefinidos tanto em SEQ como em C paralelo:

MAX_WRK	FROM_NET	SFE	MASTER_TAS K
N_WRK	TO_NET	CDS	SLAVE_TASK
WRK	TO_HOST	cds	MAPS_APP
MAX_LINKS	ES	HEAD	IO_REQ
MST_FREE_IN	ERR	TER	IO_LEN
MASTER_PORT	HOST	TEP	IO_SRC
PNEXT	MASTER_READY	TEG	TMP
PPREV	MASTER_INIT	PQ	
in	SIR	PT	
out	DFR	BP	

A.11 Adaptação do SPAM para aritmética de vírgula flutuante de 8 *bytes*

Para adaptar o sistema para trabalhar com números de vírgula flutuante de maior precisão, como é o exemplo do **double** com 64 bits, as primitivas de comunicação têm de ser alteradas. Neste caso especifico devem ser reescritas de modo que, para comunicar cada número de vírgula flutuante, sejam enviadas duas palavras de 4 *bytes.*
Ao nível da biblioteca de cálculo as macros matrix e REAL, são redefinidas como:

```
#define matrix double
#define REAL double
```

Apêndice B – Erros em tempo de execução

B.1 Tabela de códigos de erro em tempo de execução

A tabela seguinte indica para cada código de erro de execução a sua descrição. A ocorrência de qualquer destes erros implica que a aplicação será encerrada. A descrição pormenorizada do tratamento deste tipos de erros é abordada no capítulo 4.

Código de erro	Identificação	Descrição do erro
Erros gerais:		
1000	OUT_OF_MEM	Não existe espaço disponível no *Heap* para alocar o objecto
Erros de comunicação:		
2000	MSG_DIM	Comprimento da mensagem fora da faixa [0, 16383] palavras (32 *bits*)
2001	MSG_DIM_S	idem no ENVIO (send) da mensagem
2002	MSG_DIM_R	idem na RECEPÇÃO (rec) da mensagem
2010	MSG_CIRCULAR	Destino da mensagem igual à origem
2011	MSG_CIRCULAR_S	idem no ENVIO (send) da mensagem
2012	MSG_CIRCULAR_R	idem na RECEPÇÃO (rec) da mensagem
2020	MSG_ADR	Destino da mensagem não existe
2021	MSG_ADR_S	idem no ENVIO (send) da mensagem

tabela B-1: Descrição dos erros em tempo de execução

Código de erro	Identificação	Descrição do erro
2022	MSG_ADR_R	idem na RECEPÇÃO (rec) da mensagem
2030	MSG_NULL	Ponteiro para vector de dados é nulo.
2031	MSG_NULL_S	idem no ENVIO (send) da mensagem
2032	MSG_NULL_R	idem na RECEPÇÃO (rec) da mensagem
Erros de acesso aos operandos:		
3000	DIM_TO_LOW	Dimensões da matriz inferior a 1
3001	DIM_DIFF	Dimensões das matrizes devem ser iguais
3002	DIM	Posição não existe na matriz
3003	VECT_ONLY	Só opera com vectores
3004	LIM	Primeira posição do vector não pode ser maior que a última
3005	TRANS_DIM	Dimensões da transposta resultado devem ser o oposto do operando
3006	SUB_MAT	Linha final não pode ser inferior à inicial
3007	DIM_DST	Matriz destino têm dimensões erradas
3008	DIM_SQR	Matriz deve ser quadrada
3009	DIM_MULT	Dimensão das matrizes para multiplicação deve ser m x p x n
3010	NO_VECT	Não opera com vectores
3011	MATX_ONLY	Só opera com matrizes
3012	ELEM_DIFF	Ambos os vectores devem ter o mesmo número de elementos
3013	POLY_DST	Número de elementos do vector destino deve ser a soma do número de elementos dos dois operandos menos 1
3014	QR_DST	Matrizes resto ou quociente têm dimensões erradas

tabela B-1 (cont.): Descrição dos erros em tempo de execução

Código de erro	Identificação	Descrição do erro
Erros de inicialização:		
4000	RE_FLT	Não é possível inicializar os mecanismos de comunicação
4001	CB_FLT	Não é possível inicializar os buffers gerais de cálculo
4002	VAR_ALLOC	Por alguma razão não foi possível alocar a matrix
Erros de entrada-saída:		
5000	CHAN_CLOSED	Tentativa de E/S em canal não aberto
5001	ARG_TYPE	Argumento pode ser apenas *string*, int ou float.
5002	NON_ARG	Argumento # não existe
5003	NO_ARG	A aplicação requer pelo menos um elemento – o seu nome, para reparar bug no compilador para T8
5004	FILE_NOT_FOUND	Ficheiro não encontrado
5005	READ_ERR	Erro na leitura
5006	WRITE_ERROR	Erro na escrita
5010	HOST_IO_UKN	Servidor de acesso ao anfitrião não reconhece o código de pedido.
Outros erros:		
6000	GJ_NULL	GJ encontrou um pivot nulo
6001	OPR_UKN	Operação matemática não conhecida
Erro não previsto:		
outro	DFLT_ERR	Erro desconhecido

tabela B-1 (cont.): Descrição dos erros em tempo de execução

Apêndice C – Sintaxe dos diagramas de paralelização

Na sua forma mais geral estes diagramas são compostos por cinco áreas. A primeira área representa o canal de comunicação entre o anfitrião e a raiz. Seguidamente tem-se a área de processamento da raiz, a qual comunica com a área de processamento dos nós de cálculo através de um canal de comunicação. A última área representa o canal de comunicação entre os nós de cálculo, como mostra o exemplo seguinte:

	ÁREA 1	ÁREA 2	ÁREA 3	ÁREA 4	ÁREA 5
	CANAL	Raiz	CANAL	Nó de cálculo	CANAL
0	4 REAL	**v1**			
	⇒				
1		**v2** = 2 * **v1**			
2			REAL		
		*{**v2** 1,1}*	⇒	*{*r1*}*	
3			REAL, 5 INTEGER		
		*{*r0, i0*}*	⇒	*{*r2, i1*}*	

tabela C-1: Exemplo de um diagrama de paralelização geral

Nos canais de comunicação, áreas 1, 3 e 5, deve existir um caracter ⇒ ou ⇐ ou ⇔, o qual indique em que sentido a comunicação é feita, pois os canais assumem-se bidireccionais. Sobre a seta, sempre que necessário é indicado o tipo de dados a transferir e a quantidade destes, no formato:

n <Tipo de dados>

Assim, na linha 0 da tabela anterior, está indicada a comunicação de um vector de quatro REAL, isto é números de vírgula flutuante de 4 *bytes*, entre o anfitrião e a raiz. Note-se que é assumido que a raiz representa um nó independente que pode estar alocado num processador próprio ou no processador que contém o nó de cálculo 0.
Já na linha 2 está indicada a comunicação de um elemento REAL do vector **v2**, para todos os nós de cálculo, onde será referenciado por *r1*. Deve ser tomado em conta que embora não seja indicado explicitamente, qualquer que seja o comprimento da mensagem, deve ser adicionado a esta o comprimento do cabeçalho do pacote de comunicação, que no caso do SPAM 1.0 é de 32 *bits*.
Se for pretendido indicar a comunicação de 2 ou mais variáveis de tamanhos diferentes a seguinte sintaxe deve ser utilizada:

n_0 <Tipo de dados>, n_1 <Tipo de dados>, ...

$\Rightarrow$

onde os tipos de dados sobre a seta devem seguir a mesma ordem das variáveis nas área de processamento, como mostra a linha 3 do diagrama anterior.
A área 1 representa um só canal bidireccional de comunicação entre a raiz e o anfitrião. Um exemplo de comunicação de uma variável neste canal pode ser observada na linha 0 do exemplo acima. No entanto as área 3 e 5 podem representar mais de um canal de comunicação, logo se existirem *n* processadores existirão *n* canais. Estes são canais virtuais, assumindo uma ligação ponto a ponto entre a raiz e cada nó de cálculo e entre um qualquer nó de cálculo e qualquer outro.

Quanto ao código alocado nos nós de cálculo, é assumido que este é idêntico para todos eles, embora certos segmentos possam ser executados apenas em determinados nós, mediante a verificação da variável *x* que os identifica.

A referência a operandos distribuídos é efectuada considerando que se LLINHAS (**M**, *x*) indica o número de linhas a armazenadas no processador *x*, para a matriz **M**, LINHA_INICIO (**M**, *x*) o índice de posição da primeira linha da matriz **M** alocada no processador *x*, então **M**' representa o conjunto de linhas consecutivas de **M** com início na linha dada por:

LINHA_INICIO (**M**, *x*)

e linha final dada por:

LINHA_INICIO (**M**, *x*) + LLINHAS (**M**, *x*),

isto é a partição de **M** armazenada no processador *x* que pode ser representada pelo índice em sobrescrito ou apenas seguida de uma plica. Pode-se assim definir a distribuição de uma variável completa da raiz para todos os nós de cálculo:

<Tipo de dados>

M ⇒ **M'**

e no caso inverso, isto é a construção de uma variável completa, na raiz, a partir de parciais provenientes dos nós de calculo:

<Tipo de dados>

M ⇐ $\mathbf{M}^x$

Como já foi indicado, tipos cheios em maiúscula referem-se a matrizes. Se forem cheios em minúsculas, vectores. Os escalares por tipos em itálico. Além disso, quando a variável a comunicar se encontra numa expressão, pode-se observar qual é a variável que será transferida pois encontra-se entre { }.

Em termos de SPAM no entanto, não existe possibilidade de especificar código para a raiz, visto que esta consiste num servidor que permite que qualquer nó de cálculo, ou Prg (*x*) comunique com o anfitrião. Assim o diagrama geral é reduzido para um com apenas 3 áreas. A área de processamento de todas as instância da aplicação Prg (*x*), uma área de comunicação que corresponde à rede de interconexão dos nós, e uma terceira área que poderá referir-se a um conjunto qualquer de instâncias ou ao anfitrião, visto que a raiz em termos de SPAM é transparente. Para indicar qual o nó ou nós a que se refere o código da coluna da direita deve ser usada uma etiqueta antes do código correspondente na área 3:

Prg (**s**)

onde **s** é um escalar que indica apenas um nó ou então um ou mais conjuntos de nós consecutivos $\mathbf{s}_i$, separados por uma vírgula:

(C-1) $\mathbf{s} = [\ \mathbf{s}_0, \mathbf{s}_1\ , ..., \mathbf{s}_n\ ]$

e onde cada conjunto de nós consecutivos tem um nó inicial *i* e um final *f*, separados por .. :

i .. f

Se **s** for omitido é assumido que se está a referir a todos os nós; se for substituído por *x*, assume-se que se está a referir a um qualquer nó *x*. Se pelo contrário se pretender indicar todos os nós excepto os contidos em **s** deve ser utilizado $\bar{\mathbf{s}}$.
Quanto ao anfitrião deverá ser indicado como tal. Veja-se o seguinte diagrama exemplo:

ÁREA 1	ÁREA 2	ÁREA 3
	REDE	
Prg (*x*):		Anfitrião:
v2' = 2 * **v1**'		
	m REAL	
{**M**'} = **v2**' * **v1**'^{T}	⇒	{**M**}
		Prg (0):
	REAL	
{$\mathbf{M}_{1,x}$}	⇒	*r0* = *r0* + {*c*}

tabela C-2: Exemplo de um diagrama de paralelização SPAM

Repare-se que na última instrução de comunicação do diagrama é enviado um elemento de cada nó para o nó 0, que é recebido no escalar *c* e que após a recepção de cada um é somado com *r0*. Assim, quando usado num operando, o operador { } indica que existe prioridade na transferência de dados antes de se efectuar a operação. Pelo contrário, se usado no resultado indica que este deve ser obtido pela expressão do membro direito da equação, antes de ser enviado para a rede, que é o caso da primeira instrução de comunicação do mesmo diagrama. Também por isto, para evitar confusão, a variável que guardará o resultado deve estar isolada no membro esquerdo das expressões.

Esta última comunicação é do tipo todos para um, semelhante à já abordada no ponto 4.3.1.4, simplesmente o receptor não é a raiz mas sim uma instância da aplicação, a única que não emite. Pode ser referida também como muitos para um:

Prg (*x*):		Prg (*l*):
	n <tipo de dados >	
{ **M** } (**s**)	⇒	{ **M** }

onde o conjunto de muitos processadores é dado por **s**, segundo o formato já descrito para (C-1). Como já visto, se (**s**) for omitido, é assumido que se está a referir á totalidade dos nós na rede, isto é todos para um. A constante inteira *l* identifica um só nó, e *x* é uma variável inteira que pode tomar qualquer valor entre 0 e *np*-1, o número de nós na rede. Este notação também pode ser usada para indicar comunicação um para um, ponto 4.3.1, se **s** for um inteiro que indica apenas um nó.
O tipo de comunicação um para todos, ponto 4.3.2, é descrito como:

Prg (*x*):		Prg (*l*):
	n <tipo de dados >	
{ **M** } (**s**)	⇐	{ **M** }

Falta ainda referir o caso todos para todos, ponto 4.3.2.2. Aqui teremos que substituir *l* por *x*, para indicar que a coluna da direita também se refere a qualquer nó da rede:

Prg (*x*):		Prg (*x*):
	n <tipo de dados >	
{ **M** } (**s**)	⇒	{ **M** }

ou então por **s** se se referir a muitos:

Prg (*x*):		Prg (**s**):
	n <tipo de dados >	
{ **M** } (**s**)	⇒	{ **M** }

Alternativamente, pode ser usada só uma coluna para representar a comunicação de todos para todos:

Prg (*x*):

n <tipo de dados >

{ **M** } ⇒

{ **M** } ⇐

Da mesma forma a recepção de variáveis parciais de todos os nós de cálculo excepto o corrente, e consequente união destas para formar a variável completa em todos os nós, isto é o sub-programa de comunicação collect, referido no capítulo 4, é descrito numa só coluna como:

Prg (*x*):

n <tipo de dados >

{ **M**' } ⇒

{ **M** } ⇐

ou simplesmente por:

Prg (*x*):

∪ { **M** }

onde as fontes são **M**'. Deve ser tomado em atenção que embora **M**' seja parte de **M**, em termos de simplificação de tradução do diagrama para um algoritmo ou código de alto nível, deve ser considerado que se referem a variáveis diferentes.

Enviar uma variável completa ou parcial para todos os nós, excepto o que a calculou, que é uma operação de um para todos, poderá ser também abreviada:

Prg (*x*):

∩ { **M** }

e neste caso os destinos são **M**'.

Apêndice D – Resumo das principais funções e sua relação com o tradutor

D.1 Sub-programas de comunicação

D.1.1 Primitivas *(implicitos)*

SEND(d,l,f,p,e)

REC(l,s,p)

D.1.2 Operações comuns entre nós *(implicitos)*

COLLECT(s,g) ou COLLECTx(s,g,l,c,e)

COLLECT2(s,g) ou COLLECT2x(s,g,l,c,e)

SSYNC( )

REDIST(s,d,li,ci,lf,cf) *(operador [área])*

D.1.3 Entrada-saída com o anfitrião *(funções SEQ 1.0 base)*

D.1.3.1 Gerais

END() *(exit)*

PAUSE()

MSYNC() *(implicito)*

D.1.3.2 Manuseamento de ficheiros

HOPEN(f)

HCLOSE(c)

HRESET(c)

PRINT(c,s)

NL

SP

WRITEI(a,s)

WRITEF(a,s)

READI(a,s)

READF(a,s)

WRITEMDIM(a,s)

WRITEMA(a,s)

WRITEM(a,s)

READM(a,s)

PUTV(a,s)

GETV(a,s)

D.1.3.3 Medidas de desempenho

TIME(c)

START_CLOCK()

READ_CLOCK()

TIMEW(c)

TIMEWF(c,f)

D.1.3.4 Acesso aos argumentos da linha de comandos

ARGS(a,s)

ARGF(a,f)

ARGI(a,i)

ARGC(i)

D.2 Definição e manutenção de matrizes

D.2.1 Dimensionamento

SETM(p,l,c,name) ***(dimensionamento e re)***

SET_MATRIX(p,l,c,name) ***(implicitos)***

CLEAR_MATRIX(m) ***(funções SEQ 1.0 base)***

D.2.2 Atribuição e extracção de elementos

matrix_zeros(m) ***(não usados)***

matrix_ones(m) ***(não usados)***

void matrix_fill (matrix m, float value) ***(identificador = expressão)***

void matrix_rand (matrix m) ***(funções SEQ 1.0 base)***

void load_matrix (matrix m, char *data) ***(identificador = lista)***

void matrix_ident (matrix m) ***(funções SEQ 1.0 base)***

MATRIX_GET(m,l,c,n) ***(retorna identificador[l, c])***

MATRIX_PUT(m,l,c,n) ***(identificador [l,c] = expressão)***

D.2.3 Outras

RANDOMIZE(s) ***(funções SEQ 1.0 base)***

matrix mcopy(matrix a, matrix r) ***(identificador = identificador)***

D.3 Álgebra linear

D.3.1 Principais ***(operadores)***

MTRANS(o1,r)

ADDW(o1,o2,r,a,e1,e2,t1,t2)

MULTW(o1,o2,r,e1,e2,t1,t2)

INVW(o1,r,e1,t1)

DIAGADD(o1,o2,r) ***(funções SEQ 1.0 base)***

D.3.2 Auxiliares ***(operadores)***

SHFR(m,i,f,n)

SHFL(m,i,f,n)

MTXCMP (o1,o2)

ISSQRMTX (o1)

Apêndice E – Erros léxicos, sintácticos e semânticos

E.1 Tabela de códigos de erro

A tabela seguinte indica para cada código de erro em tempo de tradução a sua descrição. A ocorrência de um determinado número destes erros implica que o processo de tradução é abortado. Por defeito o número de erros para abortar é 1, mas pode ser alterado pelo utilizador.

Código de erro	Identificação	Descrição do erro
Erros léxicos:		
1000	CHR_ILL	Caracter ilegal
1001	UKN_TOK	Lexema desconhecido
1002	EOF_UXP	Fim de ficheiro não esperado
1003	INVID	identificador não começou com literal ou '_'
Erros sintácticos:		
2000	OPENB	Experado '('
2001	CLOSEB	Experado ')'

tabela E-1: Descrição dos erros léxicos e sintácticos

Código de erro	Identificação	Descrição do erro
Erros semânticos:		
3000		Reservado
3001	DUP_ID	Identificador já definido
3002	WRD_ID	Identificador é igual a palavra reservada
3003	UKN_ID	Identificador desconhecido
3004	WRG_TYP	Tipo de dados diferente do esperado
3005	SCL_EXP	Valor escalar esperado
3006	MTX_EXP	Operando matricial esperado
3007	WRG_LST	Lista de parâmetros não é idêntica à de argumentos
3008	FNC_EXP	Função esperada
3009	NOT_DIM	Variável não dimensionada
3010	FNC_TYP	Tipo de retorno da função não corresponde ao tipo retornado por exit
3011	ID_TYP	Tipo de identificador não esperado
3012	NUL_TYP	Tipo de identificar não pode ser nulo

tabela E-2: Descrição dos erros semânticos

Os restantes são erros gerados automaticamente pelo tradutor na forma:

esperado $token_1$ ... $token_n$

encontrado $token_x$

Apêndice F – Ficheiros de configuração do tradutor

F.1 Token.dat

Este ficheiro define em quatro secções distintas os componentes da linguagem e outros símbolos necessários para a operação do analisador léxico, isto é os separadores ou brancos, identificados pelo seu código ASCII, os símbolos de pontuação, os operadores seguidos da precedência e do tipo (0 - unário prefixo e 1 - posfixo ou 2 - binário) e as palavras reservadas. Tanto a lista destas como a dos operadores e pontuação é terminada pelo símbolo ETT.

```
9 10 13 32

, ; : { } /* */ " ETT

' 15 1
! 14 0
+ 14 0
- 14 0
* 13 2
/ 13 2
% 13 2
+ 12 2
- 12 2
< 11 2
<= 11 2
```

> 11 2

>= 11 2

== 10 2

!= 10 2

&& 9 2

|| 8 2

ETT

if then else for to step while exit program integer real matrix string dim clear = ETT

F.2 Sintax.dat

O ficheiro que permite configurar a sintaxe suportada é dividido em 3 áreas:

i) Variáveis de estado e valores iniciais

ii) Definição dos diagramas de sintaxe e dos nós não nulos

iii) Arcos actuados pelas variáveis de estado

A área i) têm a estrutura:

n

$nome_0$ x_0

...

$nome_{n-1}$ x_{n-1}

Onde n indica o número de variáveis de estado, e é seguido pelo seu valor inicial.

A área ii) têm a estrutura:

m

$nome_0$ ordem

Início

...

Fim

...

$nome_{m-1}$ ordem
Início
...
Fim

nó_fonte nó_destino
(...)
END

Onde m indica o número de diagramas de sintaxe. Cada diagrama é seguido por uma linha com o seu nome e seguida pela ordem da matriz de adjacência, que é quadrada. Seguidamente são definidos tantos nós como a ordem, onde o primeiro será Início e o último Fim. Os nós são definidos da seguinte forma:

TipoToken nome diagid vnome
trans
k var_nome_0 $value_0$ (...) var_nome_{k-1} $value_{k-1}$
diagrama
fnc

Onde a primeira, a segunda, a terceira ou a quarta palavra dão o nome ao nó e por isso apenas uma deverá ser não nula. Para indicar que estes campos estão vazios, isto é uma cadeia de caracteres nula, deve ser usado o símbolo “.
Trans, isto é toda a segunda linha consiste numa *string* com as regras de tradução. Seguidamente k indica o número de actuações sobre as variáveis de estado. Estas consistem em pares, onde o primeiro elemento indica o nome da variável de estado a actuar, e o segundo a actuação. Esta pode ser uma atribuição de modo que o segundo elemento deve ser um inteiro positivo, ou um incremento ou decremento, onde o inteiro será precedido por um sinal + ou -, respectivamente. A penúltima palavra indica um salto para outro diagrama, e que deve tomar o nome de diagramas existentes, que serão traduzidos para valores entre 0 e m-1. Na última palavra é indicada uma possível validação semântica, por um índice para um vector de ponteiros para funções tipo void ( ). No caso de não ser necessária qualquer validação devem ter um valor negativo.

Finalmente são definidos os nós não nulos em termos do diagrama corrente, por um par nó_fonte nó_destino, isto é a linha e coluna da matrix de adjacência, numa série que deverá terminar no símbolo END.
Finalmente a área iii) define os arcos que serão actuados por uma dada variável de estado até ao fim do ficheiro:

nome x
$diagrama_0$ $nó\_fonte_0$ $nó\_destino_0$ $normal_0$
(...)
$diagrama_{x-1}$ $nó\_fonte_{x-1}$ $nó\_destino_{x-1}$ $normal_{x-1}$

Deve ser iniciada com o nome da variável que actua os arcos virtuais, seguido do número de arcos a actuar e de x conjuntos de quatro coordenadas, as quais indicam os arcos a actuar no diagrama linha e coluna e o estado normal do arco se aberto (0) ou fechado (1).

A especificação de nomes nós permite destinguir nós com nomes iguais no diagrama corrente, como é exemplo mais que uma chamada a um outro mesmo diagrama, terminando o nome do nó com um número.

F.3 Sem.h / c

Neste ficheiro deve ser implementada a análise indicada em comentários nos diagramas de sintaxe. Essa análise deve ser efectuada por funções do tipo void (astack*, parser*) apontadas por um vector de ponteiros onde os índices correspondem ao campo fnc da definição dos nós.
Nestas funções, o valor das variáveis de estado pode ser acedido pelo seu nome:

```
int* svalue (char *)
```

Os tokens da pilha endereçável pela sua posição na pilha local:

```
tokenrec* token (int, astack *)
```

e a geração de erros de sintaxe ou semântica pelo código de erro referente a um *token*:

```
serror (int , tokenrec*);
```

Quanto às tabelas de símbolos, estas podem ser acedidas através das referências, TSlocal e TSglobal respectivamente. Podem ser manipuladas ou consultadas através das funções:

```
void TSxxxx.clr ()

void TSxxxx.put(symbol)

int idExist (tokenrec*, int)

int varExist (tokenrec*)

int fncExist (tokenrec*)
```

As duas primeiras permitem respectivamente, eliminar todos os símbolos da tabela local ou global e colocar um símbolo numa delas. A terceira verifica se um dado *token* existe numa das tabelas de símbolos ou apenas na tabela local. As duas última verificam se um dado *token* é um símbolo respeitante a uma variável ou função.
A função seguinte permite verificar se um dado *token* é idêntico a uma palavra reservada:

```
int wordExist (tokenrec*)
```

Para determinar o código do tipo de dados a que corresponde um lexema pode ser usado:

```
tipoDados varTipo(tokenrec* )
```

O nome actual do identificador que está a ser declarado pode ser guardado em:

```
char  fncID [IDLEN]
```

de modo que pode ser usada a função:

```
symbol*          fncSymb (parser*)
```

para retornar um ponteiro para o símbolo global, correspondente ao identificador corrente. Esta função pode ser muito útil para aceder a uma lista de parâmetros de uma função. Para indexar esta lista pode ser usada a variável:

```
int  argIdx
```

F.4 Function.Dat

Consiste numa tabela com três colunas. A primeira tem o nome do operador ou função em SEQ 1.0 a segunda o nome da função correspondente em C e a terceira o tipo da função como apresentado no capítulo 7.

Apêndice G – Diagramas de sintaxe SEQ 1.0 e nota técnicas

G.1 Diagramas de sintaxe

Os diagramas de sintaxe da linguagem SEQ 1.0 apresentados a seguir, estão organizados numa perspectiva de cima para baixo de modo que o ponto de entrada é:

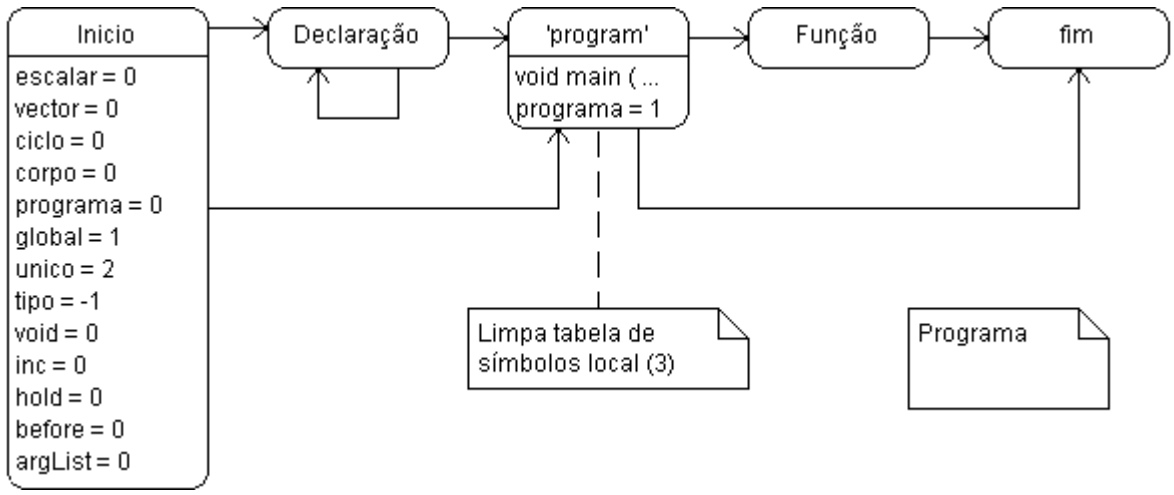


Fig. G-1: Diagrama de sintaxe Programa

Se bem que nas regras de tradução do nó 'program', esteja indicado a tradução para o cabeçalho da função main do C paralelo, como o ambiente de comunicações constituí também a base da instância Prg(*x*), definindo no ficheiro "routerss.c" uma área de declarações globais e também o início da função main( ), que inicializa este ambiente, a directiva **program** na realidade traduz-se apenas por chamar adicionar código ao fim de main() para chamar rotina de alocação de matrizes globais no *heap*. Esta rotina, bem como as declarações globais,

devem ser traduzidas para um ficheiro que será incluído antes da função main(), com o nome genérico “maps_udf.h”

É conveniente também indicar que a tradução de um programa SEQ para parallel C termina sempre com uma chamada à função END, de modo que o servidor de entrada saída com o anfitrião seja encerrado, fazendo desta forma com que a aplicação termine.

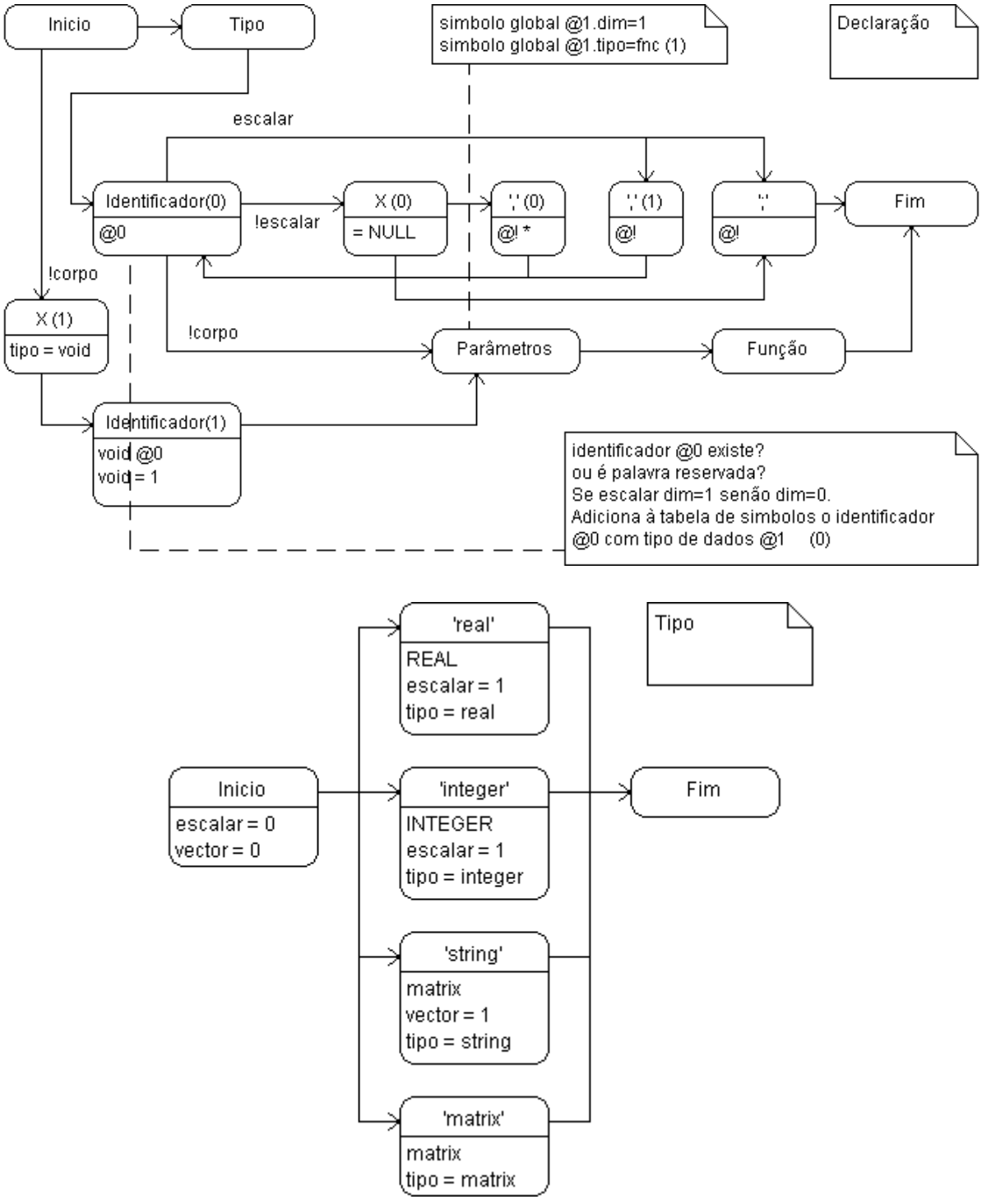


Fig. G-2: Diagramas de sintaxe Declaração e Tipo

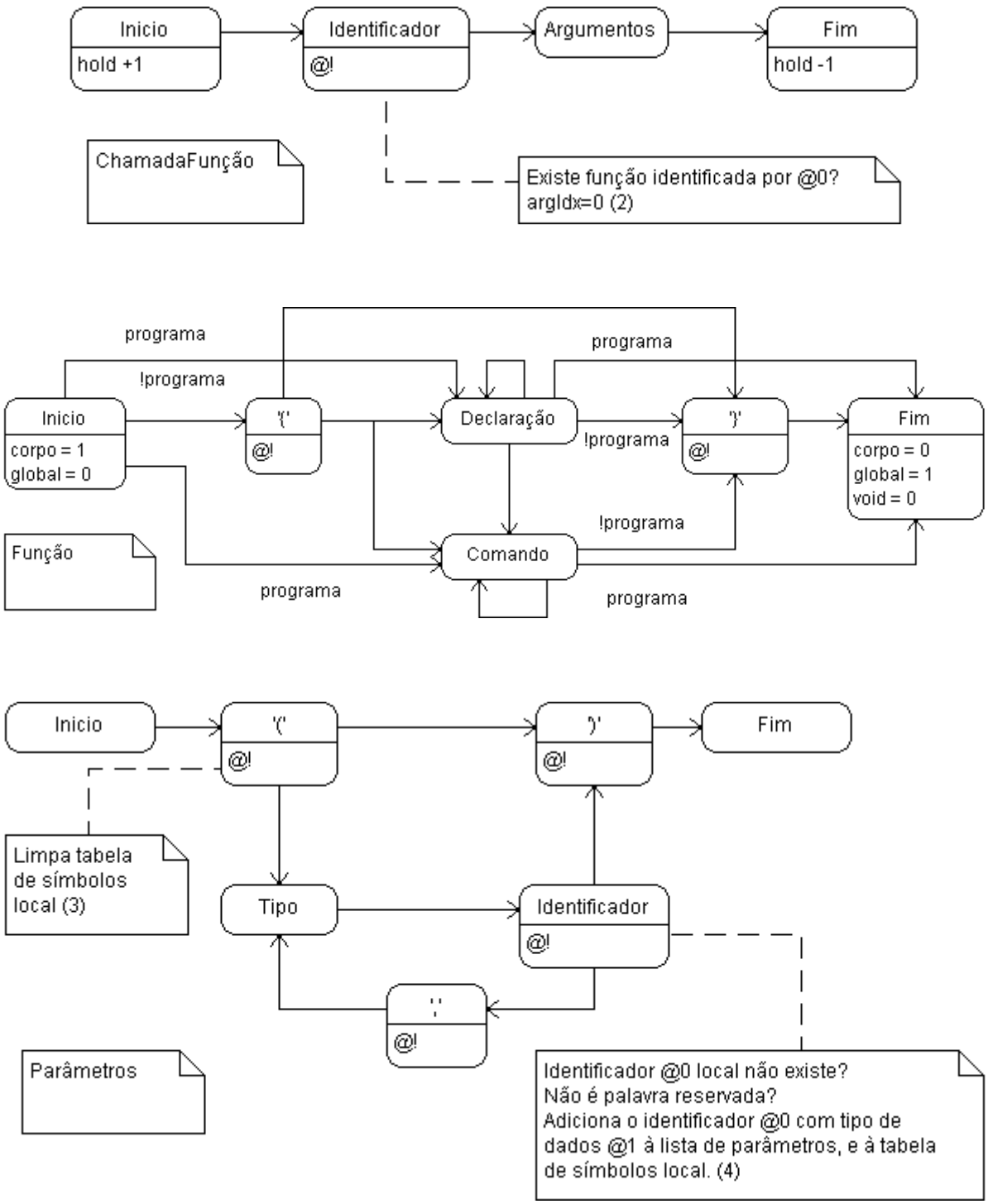


Fig. G-3: Diagramas de sintaxe ChamadaFunção, Função e Parâmetros

Não é validada a dimensão dos operandos matriciais, pois estes são traduzidos para C paralelo como ponteiros. A detecção de operações inválidas entre matrizes, dadas as suas dimensões, é efectuada pelo sistema de detecção de erros em tempo de execução.

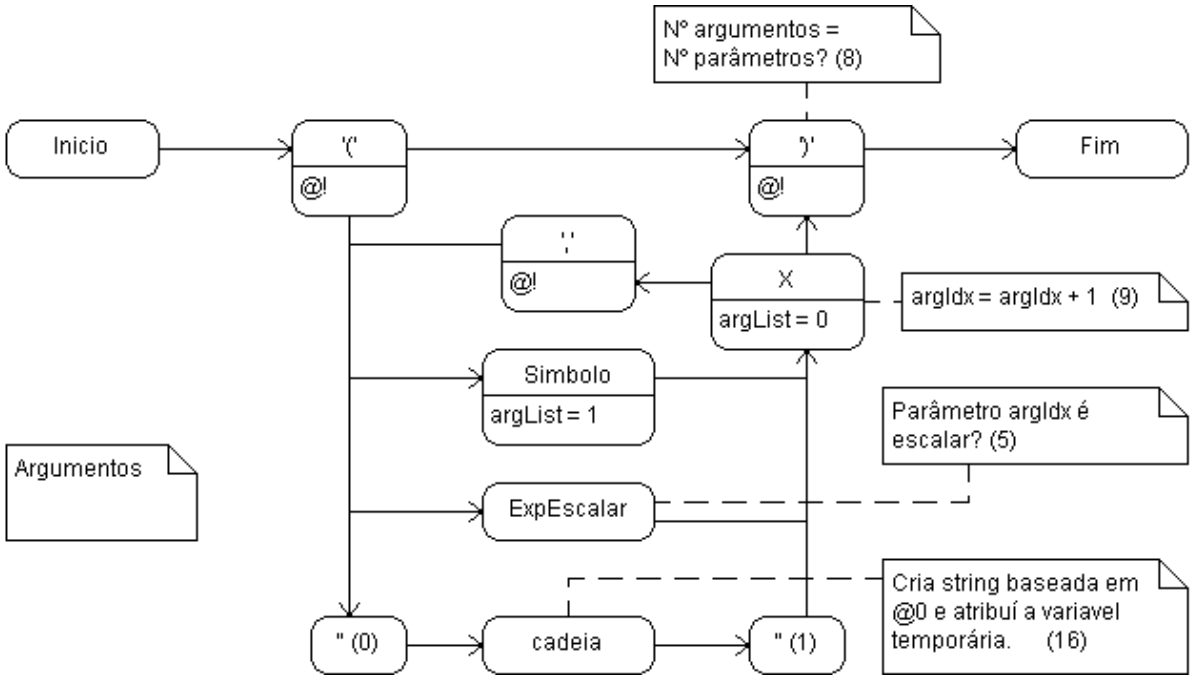


Fig. G-4: Diagrama de sintaxe Argumentos

Os argumentos escalares são passados por valor. Já no caso das matrizes e *strings*, é passado um ponteiro. Deste modo os argumentos escalares não podem ser alterados por uma função, e todas as alterações feitas por uma função num não escalar reflectem-se no argumento.

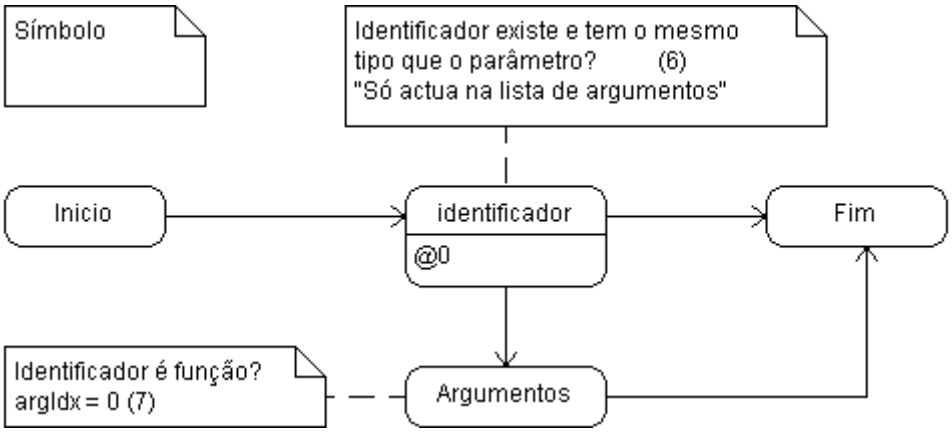


Fig. G-5: Diagrama de sintaxe Símbolo

A passagem de constantes tipo string como argumentos é tratada pela função de análise 16. Se a linguagem alvo fôr polimórfica, como o C++ , esta função não é necessária para criar um

objecto string a partir de uma constante, pois será feita pelo constructor adequado, quer o argumento seja char* ou float*, isto é matrix.

No diagrama seguinte, deve ser processada a tradução de expressões, sempre que a fila infixa não esteja vazia. Esta pode ser preenchida pelas atribuições e pelo comando Exit, que também pode retornar uma expressão.

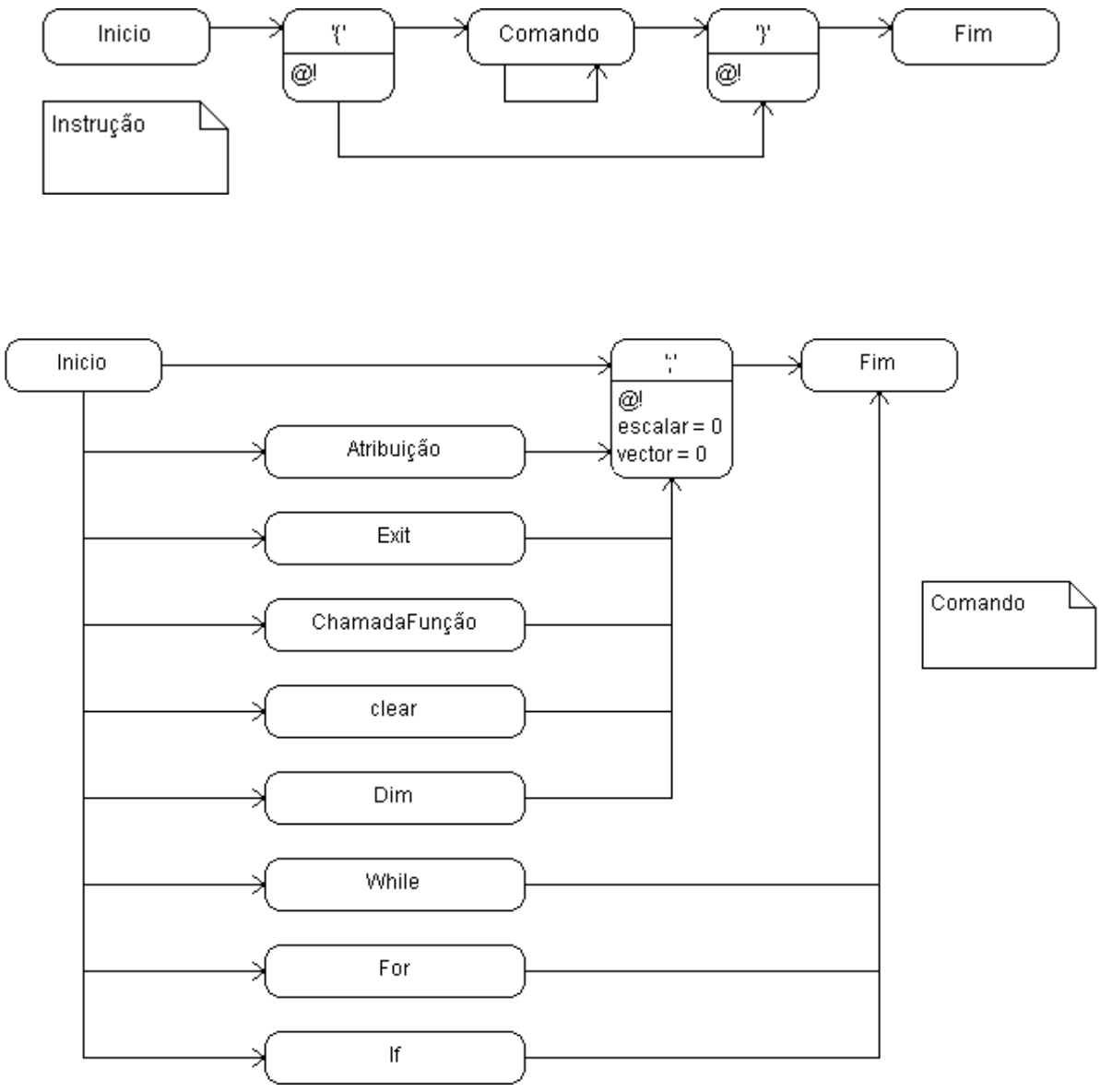


Fig. G-6: Diagramas de sintaxes Instrução e comando

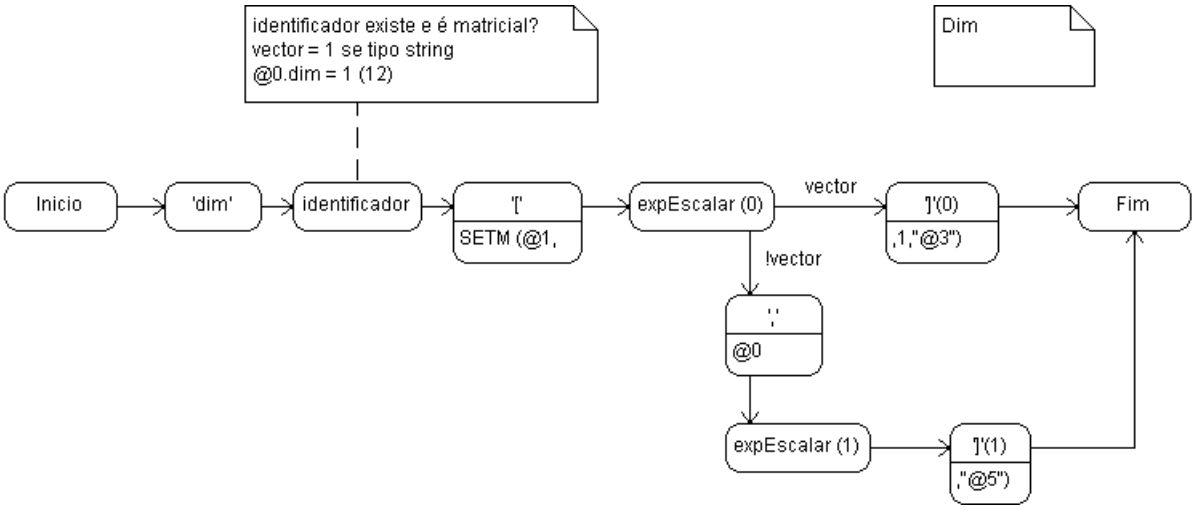


Fig. G-7: Diagrama de sintaxe Dim

Neste caso, tem-se um arco virtual dependente da variável de estado denominada escalar. No caso desta ser verdadeira não existe nenhum caminho possível de modo que o motor de análise deve emitir uma mensagem de erro. Esta mensagem deve ser também emitida sempre que num dado ponto de um diagrama não seja possível navegar para um nó que represente o *token* seguinte.

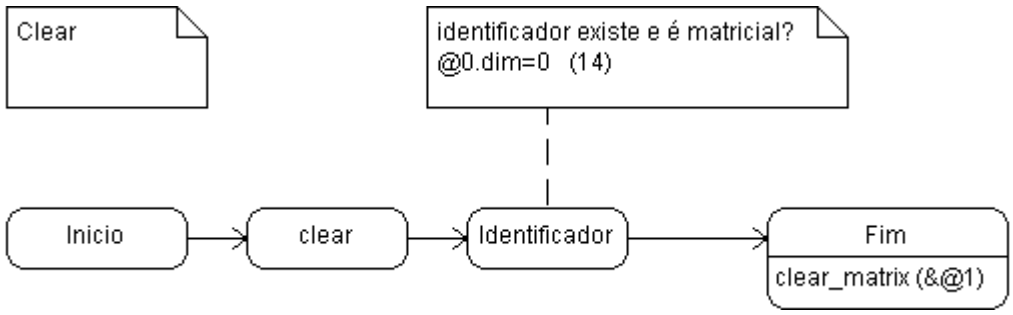


Fig. G-8: Diagrama de sintaxe Clear

A variável de estado hold, quando não nula, indica que a tradução é efectuada para um *buffer* na memória em vez do ficheiro alvo, de modo que se possa adicionar cadeias de caracteres antes do corrente ponto de tradução. De facto esta funcionalidade pode ser seleccionada automaticamente se a variável before não for nula. É assim possível traduzir a decomposição de uma expressão, substituindo nesta as operações por identificadores, os quais guardam o resultado da operação que lhes é atribuída previamente.

Os dados guardados neste *buffer* são escritos para o ficheiro alvo quando a variável de estado hold voltar a ser nula.

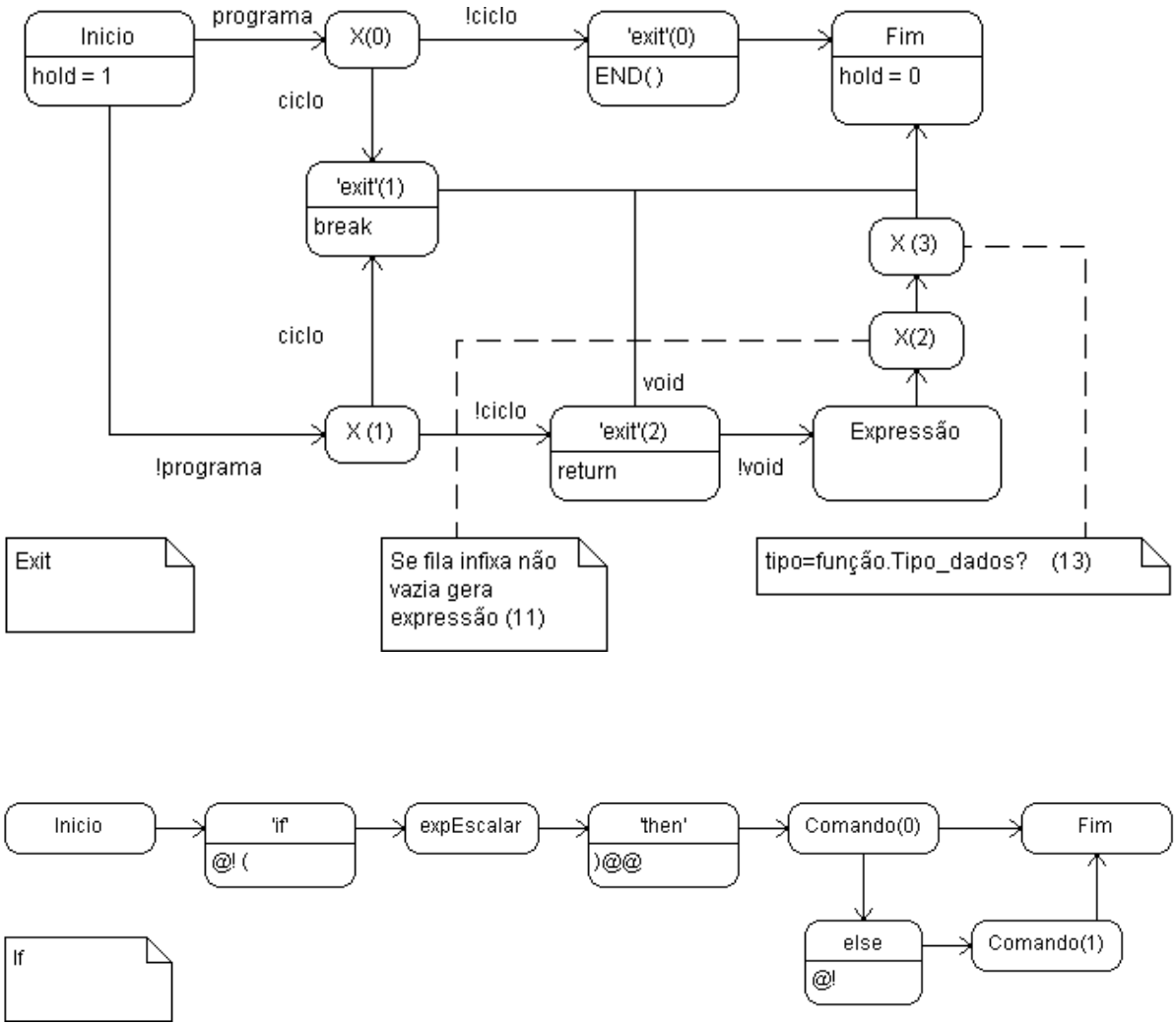


Fig. G-9: Diagramas de sintaxe Exit e If

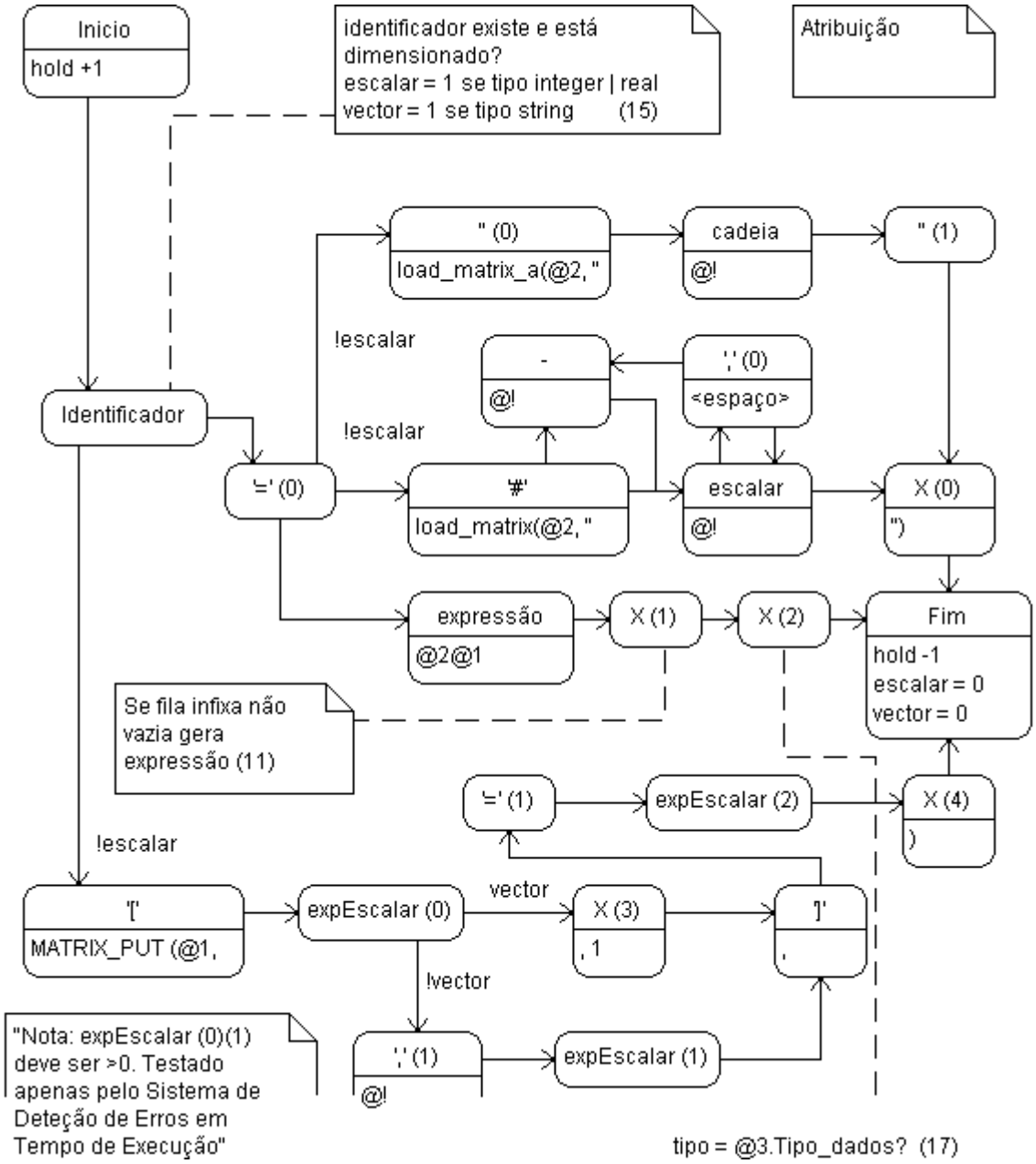


Fig. G-10: Diagrama de sintaxe Atribuição

A função load_matrix_a é uma modificação de load_matrix, apresentada no capítulo 5, que atribuí a cada elemento do vector passado como primeiro argumento o código ASCII de cada caracter da cadeia passada como segundo argumento.

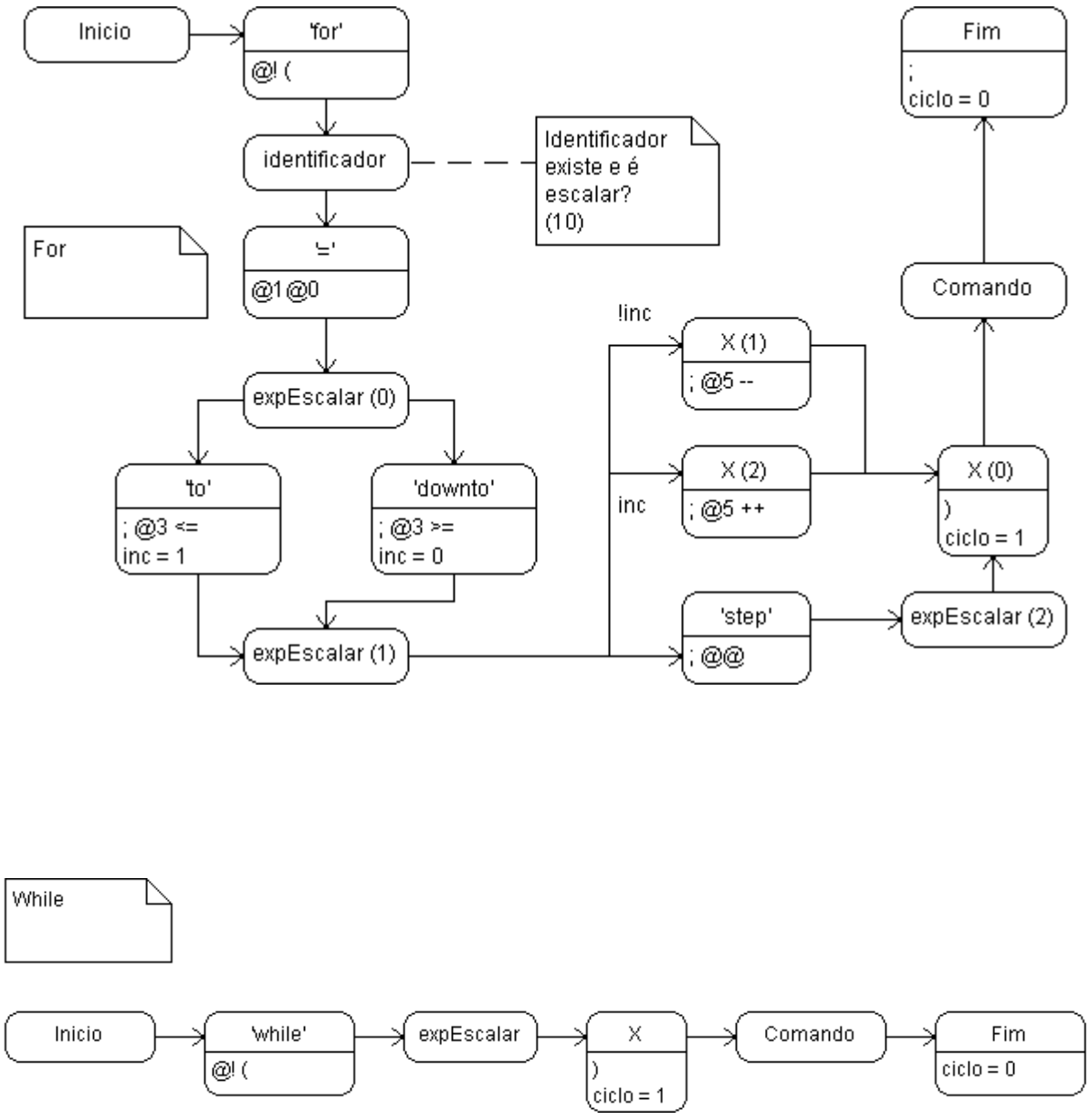


Fig. G-11: Diagramas de sintaxe For e While

Funções auxiliares e variáveis auxiliares são codificados em C paralelo e estão definidos numa extensão das bibliotecas padrão, especialmente desenvolvida para facilitar o *interface* com o tradutor: "transext.h"

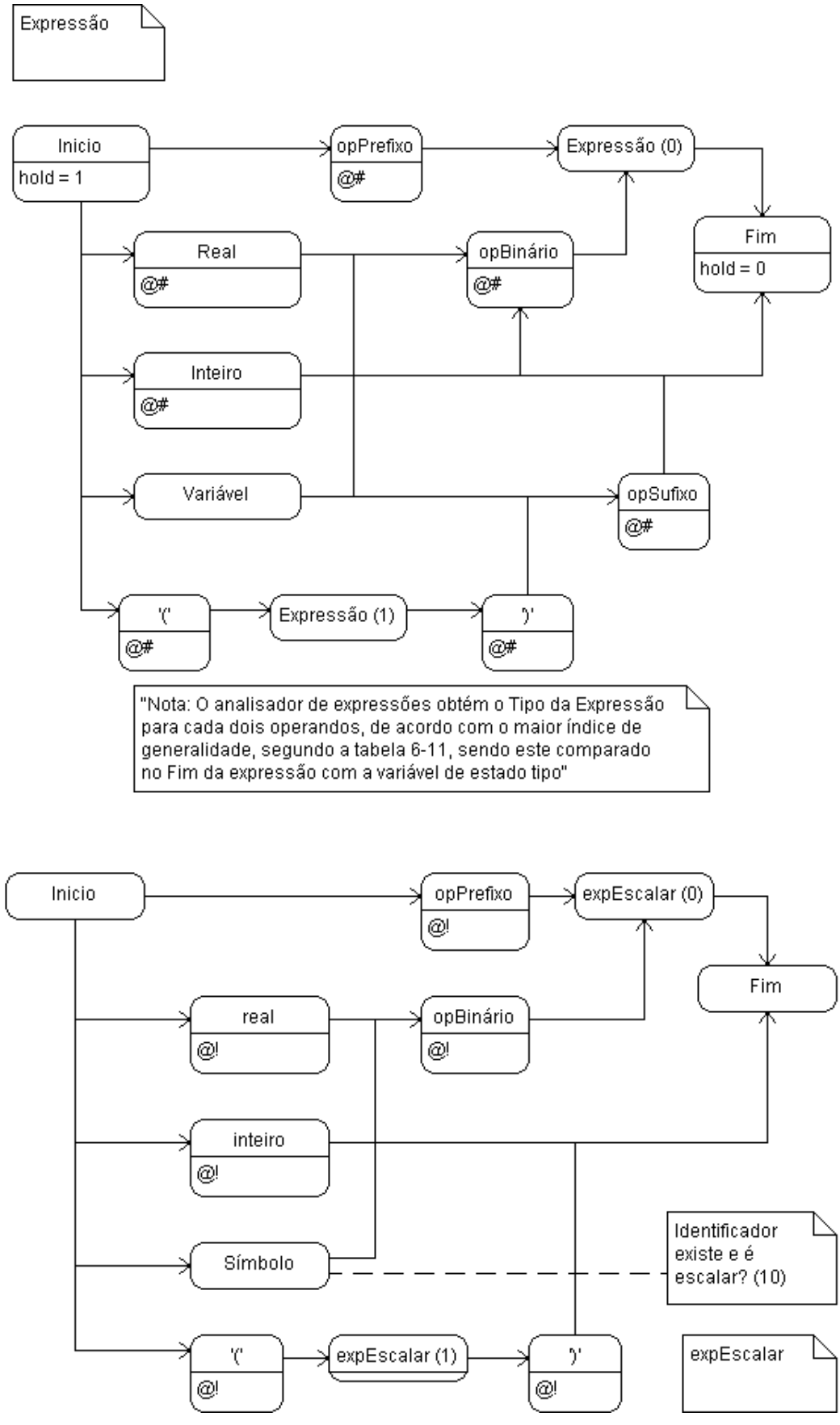


Fig. G-12: Diagramas de sintaxe Expressão e expEscalar

O acesso a áreas ou elementos de matrizes, é processado por funções, de modo que é necessário que estas sejam efectuadas antes da avaliação da expressão. O seu resultado é atribuído a um identificador, que irá ser colocado na posição correspondente à operação na expressão, usando a pilha de expressões.

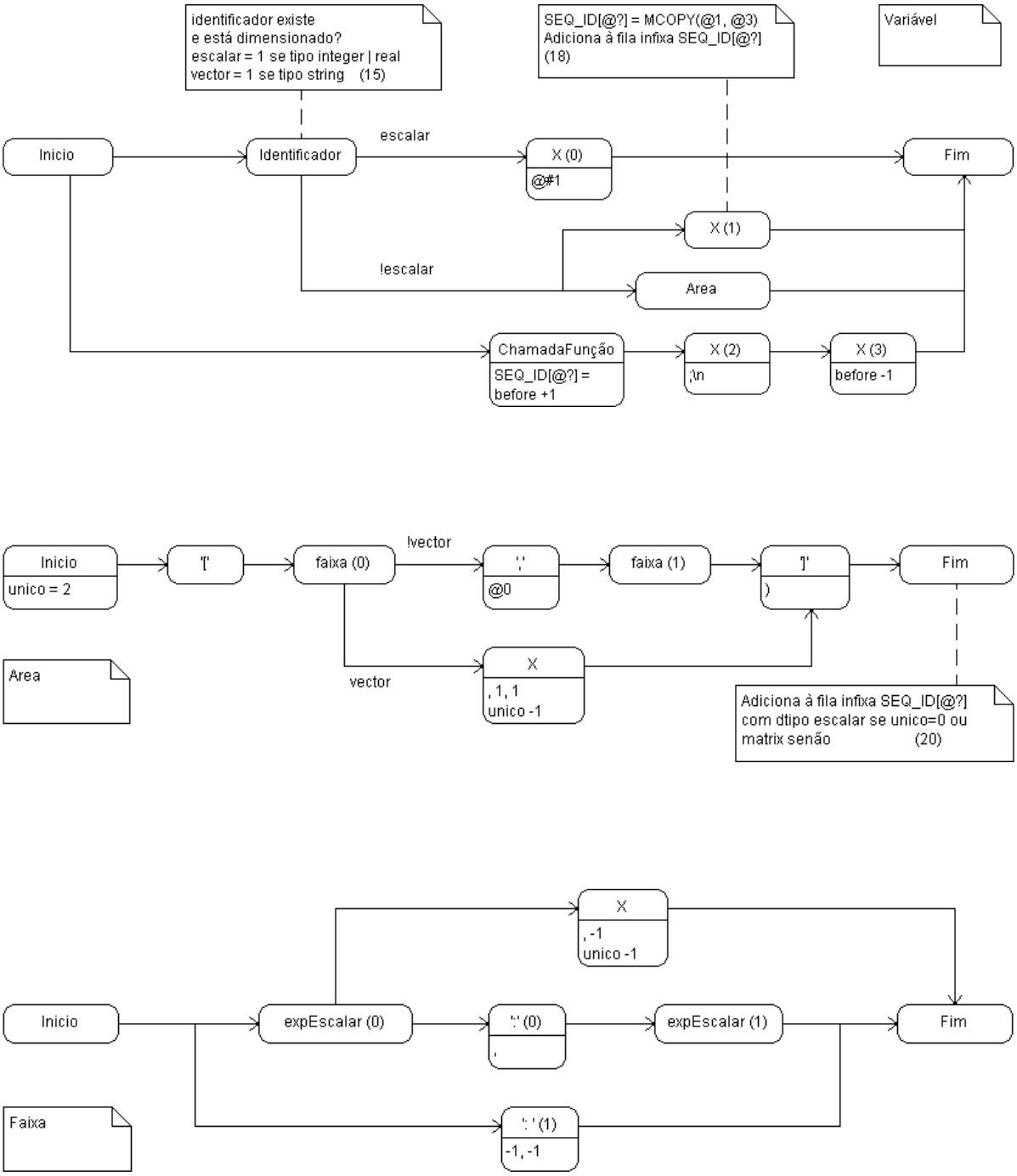


Fig. G-13: Diagramas de sintaxe Variável, Área e Faixa

O símbolo @? indica o índice da variável auxiliar corrente, referida na secção seguinte. A actualização deste índice é efectuado automaticamente, mas também podia ser implementado como uma variável de estado, que deveria ser actualizada explicitamente.

G.2 Funções e variáveis auxiliares "transext.h"

As variáveis auxiliares e seus tipos, necessárias para suporte de acesso a variáveis ou partes destas em código C paralelo, já atrás referenciadas definem-se como:

```
/*    Expression auxiliar variables  */
#define MAX_AUX_VAR 100
int SEQ_ID[MAX_AUX_VAR];

/*  Scalar allocation */
REAL* SETS(REAL *n) {
        free (n);
        return n=(REAL*) malloc(sizeof(REAL));
};

/*   Funções de acesso a partes de matrizes */
matrix area (matrix src, matrix dst, INTEGER li, INTEGER lf, INTEGER ci,
                INTEGER cf,) {
int s0=0, s1=0;         /* only if [1,2], [2:2, 3:3] is one by one matrix */

        if (li>0 && lf<=0)      { lf=li; s0=1; }
        else if (lf>0 && li<=0) { li=lf; s0=1; }
        else if (li <= 0 || lf <= 0) {li=1; lf=NLINHAS(src); }    /* li<=0 || lf<=0 => todas as
                                                                                        linhas */
        else if (li==lf) s0=1;

        if (ci>0 && cf<=0)       { cf=ci; s1=1; }
        else if (cf>0 && ci<=0) { ci=cf; s1=1; }
        else if (ci <= 0 || cf <= 0) {ci=1; cf=NCOLUNAS(src); }   /* ci<=0 || cf<=0 =>
                                                                todas as colunas*/
        else if (ci==cf) s1=1;

        if (s0 && s1) {                                 /* um só elemento */
                dst = SETS(dst);
                matrix_get (src, li, ci, dst, wrk, n_wrk); }
        else {                                          /* Sub-Matriz */
                setm(&dst, lf-li+1, cf-ci+1, "SEQ_ID_xxx", wrk, n_wrk);
                redist(src, dst, li, ci, lf, cf, wrk, n_wrk); }
        return dst;
}
```

G.3 Estruturas para análise sintáctica e semântica

```
struct sintrec{
        tipoToken tipo;                 //Tipo de lexema
        char nome[TOKENLEN+3];          //Lexema
        char diagid[IDLEN+3];           //Atalho para diagrama
        char virtid[IDLEN+3];           //Nome do nó virtual
        char trans[TOKENLEN+1];         //Regras de tradução
        int nact;                       // nº acts
        svaract* state;                 //Actuação na variável de estado
        int  fnc;                       //índice no vector de funções. < 0 se
                                        //não usado
   }
```

Apenas um dos campos Tipotoken, diagid, lexema, e virtual devem ter uma valor não nulo e funcionam como a identificação do nó. O último identifica um nó virtual.

O campo trans guarda as regras de tradução, que consistem na cadeia de caracteres que será escrita no ficheiro traduzido, e que usa símbolos @<digito> de acesso às pilhas já descritas.

O campo state guarda a actuação nas variáveis segundo pares de números onde o primeiro representa o índice da variável a actuar (var_idx) e o segundo o valor a nela colocar (valor). A operação a efectuar sobre a variável será uma atribuição (atr), excepto se o valor fôr precedido de um sinal + ou –, onde a operação a efectuar consistirá num incremento (inc) ou decremento (dec), respectivamente.

```
enum {dec, inc, atr} svarop;

struct svaract {
        int var_idx;
        int valor;
        svarop operacao;
}
```

O campo diagid é utilizado para aceder ao nó Início de outro diagrama e o campo fnc guarda o índice de uma função de verificação semântica, indicada nos diagramas anteriores como comentários aos nós. Este último campo toma um valor negativo se não utilizado.

Após se navegar para outro diagrama, chega-se sempre ao nó Início, que deverá ser a primeira linha e coluna da matriz de adjacência, ao passo que a última é o nó Fim.

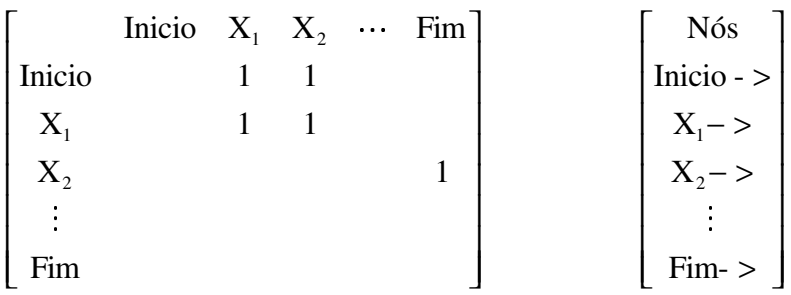


Fig. G-14: Matrizes de adjacência e vector de operações de um diagrama de sintaxe

Um diagrama de sintaxe é representado por uma matriz de adjacência como mostra a figura acima e também por um vector onde estão guardadas as operações a efectuar em cada nó. No caso de um nó não ser encontrado - por exemplo o caminho de X_1 para Fim não é possível - será gerada uma mensagem de erro automaticamente, indicando que os token esperados a seguir a X_1 seriam os elementos não nulos ao longo dessa linha, isto é X_1 ou X_2.

Por outro lado, se o *token* corrente não estiver previsto na linha corrente, mas se se puder navegar para Fim, a responsabilidade de verificar se o novo *token* respeita a sintaxe será do diagrama anterior. Só será gerado um erro se não existir nenhum diagrama anterior, isto é se se estiver no diagrama índice 0.

Como já foi referido, os diagramas possuem arcos virtuais, os quais dependem de uma dada variável de estado ser verdadeira ou falsa. Estas variáveis são guardadas num vector. Cada variável guarda informação sobre o seu valor e sobre os arcos que dela dependem.

```
struct varEstado {
        char nome [IDLEN];
        int valor;
        int narcos;
        arcVirtual* arco;
}

struct arcVirtual {
        int diagrama;
        int linha;
        int coluna;
        int normal;             //normalmente aberto: 0  fechado: 1
}
```

G.4 Decomposição de expressões matriciais

Esta secção descreve o algoritmo que decompõe uma expressão em notação posfixa numa sequência de chamadas às funções ou operadores que implementam as operações suportadas em SEQ 1.0.

Operador em SEQ	Tipo de dados	Função ou Operador em C	Comentários
Unárias:			
‘	**escalar a** **matriz a**	trans (a)	não altera o operando
!	**escalar a** **matriz a**	! a impossível	
+ ou -	**escalar a** **matriz a**	- MULTW (a, -1)	+ unário é simplesmente removido
Binárias:			
*	**escalar a, b** **escalar, matriz**	a * b MULTW	
/	**escalar a, b** **escalar a / matriz b** **matriz a / escalar b** **matriz a / matriz b**	a / b impossível MULTW (a, 1 / b) MULTW (a, INVW b)	
+ ou -	**escalar a, b** **escalar, matriz**	a (+ ou -) b ADDW (a, b, (+ ou -))	
== ou !=	**escalar a, b** **escalar, matriz** **matriz a, b**	a (== ou !=) b impossível iseqmtx (a, b) ou ! iseqmtx (a, b)	
< <= > >= && ‖	**escalar a, b** **escalar, matriz** **matriz a, b**	a < <= > >= && ‖ b impossível impossível	
%	**escalar**	%	resto da divisão inteira só opera com escalares, tratando-os como inteiros

tabela G-1: Correspondência entre operações suportadas e as funções que as implementam

Como já foi referido, se a linguagem alvo, isto é a linguagem para a qual se pretende traduzir o código SEQ 1.0, fôr C++, toda esta avaliação de expressões será simplificada recorrendo à sobrecarga de operadores ou funções característica do C++; no entanto as operações válidas e o tipos de dados delas resultantes terão sempre de ser verificados de acordo com a tabela

acima. Nesta tabela estão também indicadas as funções da biblioteca de cálculo matricial ou os operadores C utilizados para implementar em C as operações SEQ 1.0. Para além desta última informação, esta tabela é semelhante à tabela 6-1, tomando em considerando que escalar se refere aos tipos de dados integer e real, e matriz aos tipos matrix e string.

E o algoritmo que decompõe um expressão em notação posfixa:

algoritmo G-1:

```
x = lint(0,2)                                   "Inteiro: 0 1 2 0 1 2 ..."

Enquanto !POSFIXA.empty()

l = POSFIXA.len()
OPER = POSFIXA.get()

Se OPER é operando
        PILHA.push(OPER)
        tipo_exp = OPER.dtipo

Senão                                                   "OPER  é operador"
        Se OPER = Sufixo (transposição ')

                PILHA.pop (op0)
                Se l = 1
                        Se op0.dtipo = escalar      fim: op0.id;
                        Senão                       fim: trans (op0.id);
                Senão
                        op0.transposto = 1
                        PILHA.push(op0)
                tipo_exp = op0.dtipo

        Senão
                Se OPER = Prefixo (- !)         "Se OPER= Prefixo (+) Não fax nada"

                        PILHA.pop (op0);
                        Se pode efectuar a operação OPER com o operador op0
                                                                (tabela G-1)
                                "Se último token l=1 tem comportamento diferente"
                                Se op0.dtipo = escalar
                                        op0.id = OPER op0.id
                                        PILHA.push(op0);
                                        Volta ao início do ciclo
                                Senão
                                        início: chama operação adequada (tabela G-1)
                                tipo_exp = op0.dtipo
```

```
                Senão
                        Erro operação OPER inválida para operadores tipo
                        op0.tipo

        Se OPER = Binário

                PILHA.pop (op0)
                PILHA.pop (op1)
                Se pode efectuar a operação OPER com os op. op0 e op1
                                                        (tabela G-1)
                        "Se último token l = 1 tem comportamento diferente"
                        Se op0.dtipo e op1.dtipo = escalar
                                op0.id = op0.id OPER op1.id
                                Se op0.dtipo ≠ op1.dtipo op0.dtipo = real
                                PILHA.push(op0);
                                Volta ao início do ciclo
                        Senão
                                início: chama operação adequada (tabela G-1)
                        Actualiza tipo_exp de acordo com o operador com maior
                        índice de generalidade segundo a tabela 6-3.

                Senão
                        Erro operação OPER inválida para operadores tipo
                        op0.tipo
```

O tipo de dados lint consiste num número inteiro positivo com um limite superior especificado pelo segundo argumento. Sempre que o valor corrente ultrapassar este limite tomará o valor zero. O primeiro argumento indica qual o valor que deve ser dado inicialmente à instância. Este é inicializado com o valor 0 e limite superior 2, e é um índice para o vector de variáveis temporárias SEQ_ID.

Como na decomposição de uma expressão, usando uma pilha, num dado passo serão necessárias no máximo 3 variáveis temporárias para guardar resultados intermédios, os primeiros 3 elementos de SEQ_ID são reservadas para uso pelo decompositor de expressões acima transcrito. Esta variável é incrementada sempre que uma operação requer um lugar de armazenamento temporário.

Quanto às etiquetas fim: e início:, indicam se o texto a traduzir indicado após estas a **cheio** deve ser colocado antes ou depois do texto corrente.

Glossário

Algoritmo – Procedimento estruturado com o intuito de resolução de um problema.

Aceleração - Medida de desempenho de um algoritmo paralelo. Indica o número de vezes que o tempo de execução de um algoritmo é reduzido, quando mapeado sobre uma rede paralela. O valor ideal de aceleração é igual ao número de processadores sobre os quais está mapeado o algoritmo.

Barramento - Conjunto de condutores eléctricos que servem para transportar informação entre diferentes dispositivos, tais como memória e processadores. O comprimento da palavra transmitida por um barramento, em cada ciclo deste, depende do número de condutores desse barramento. Assim um barramento referido como 32-bit, por exemplo, transporta palavras de 32 bits. Os barramentos podem ter finalidades diferentes tais como transporte de dados ou endereçamento de células. Podem também ter funções de controlo, como por exemplo indicar o sentido de uma transferência de dados.

benchmark – Termo utilizado para referir uma medida de desempenho de um determinado segmento de código num determinado equipamento.

Bit - Digito binário. Unidade mínima de representação de informação, a qual pode tomar 2 estados.

Buffer – Segmento de memória para armazenamento temporário.

Bias - viés; inclinação, desequilíbrio, peso que provoca desequilíbrio; influência;

Byte – ver **octeto.**

Cache – Sistema de memória de acesso rápido, mas com um espaço de endereçamento reduzido, normalmente usada para guardar as instruções ou dados acedidos mais recentemente a um sistema de memória de acesso mais lento. Como os processos frequentemente usam um subconjunto de instruções ou de dados repetidamente, a memória *cache* é um meio de baixo custo de aumentar o desempenho do acesso à memória, segundo um método estatístico, evitando que se tenha de optar por uma solução mais dispendiosa, em que a totalidade do sistema de memória seja de acesso rápido.

Computador anfitrião - ou simplesmente anfitrião. Computador onde está alocada uma rede de processadores paralelos.

Eficiência – Medida de desempenho, a qual reflecte a percentagem de tempo em que um algoritmo paralelo está a computar o algoritmo. A percentagem complementar indica uma

perda de desempenho em tarefas necessária à execução de um algoritmo paralelo, tais como comunicação de dados entre processadores. Assim, o valor de eficiência que indica o máximo de desempenho é 100%, independentemente do número de processadores existentes na rede paralela.

Escalabilidade – Característica de determinados tipos de algoritmos, que implica uma redução do tempo de execução destes, mapeando-os por uma rede de processamento com um número crescente de processadores.

Hardware – designação que engloba todos os elementos da aparelhagem e acessórios que fazem parte de um computador

Heap – Espaço de memória utilizado para alocação de memória – dinâmica – durante a execução.

Host – ver **computador anfitrião**.

Kernel – ver **núcleo**.

Largura de Banda - Volume de dados transferidos numa unidade de tempo, normalmente o segundo.

Lexema – Menor constituinte com significado de uma linguagem.

Linker – Ferramenta de programação utilizada para agregar vários módulos de código objecto numa aplicação.

Links – ver **portos de comunicação**.

Máquina de von Neumann – ver **processamento sequencial**.

Mecanismos de encaminhamento de mensagens – Dispositivos de *software*, alocados em cada nó da rede paralela, que permitem encaminhar dados através dessa rede.

Memória partilhada – Arquitectura paralela em que os processadores não possuem memória local, mas partilham segmentos de memória. A comunicação entre processadores é feita através desta memória partilhada.

Núcleo – ou ***kernel***. Conjunto de processos de baixo nível, que asseguram e simplificam o acesso de processos de nível superior, normalmente chamados aplicativos, ao *hardware*.

Octeto – ou **Byte**. Unidade de informação composta por 8 bits, podendo tomar portanto 256 estados.

***Parser* –** analisador sintáctico.

Particionamento – (..) de um algoritmo. Divisão deste em tarefas concorrentes ao nível dos dados ou das instruções.

Passagem de mensagem – Arquitectura paralela onde os processadores possuem apenas memória local. A comunicação entre processadores é feita pela comunicação de mensagens através de canais físicos dedicados.

Pilha – Zona de memória principalmente dedicada ao armazenamento de variáveis automáticas e contador de programa.

Porto de comunicação – Dispositivo que permite ligações ponto a ponto entre processadores com o intuito de transferir informação entre estes. Também são chamados ***links*** no caso de Transputers ou Sharcs.

Processamento distribuído – ver **processamento paralelo**.

Processamento paralelo – Modelo de processamento caracterizado por uma arquitectura baseada em várias unidades de processamento operando em paralelo, ou concorrentemente, alocadas no mesmo computador anfitrião. Se todas as unidades de processamento forem iguais está-se em presença de uma rede de **processamento paralelo homogénea**, caso contrário a rede é referida como de **processamento paralelo heterogéneo**. Quando estas unidades de processamento estão distribuídas por mais do que um computador, o modelo é designado por **processamento distribuído**. Este modelo foi desenvolvido com o intuito de reduzir o tempo de execução de algoritmos complexos, pois teoricamente qualquer algoritmo pode ser particionado de modo que possa ser mapeado por vários processadores concorrentes.

Processamento paralelo heterogéneo – ver **processamento paralelo**.

Processamento paralelo homogéneo – ver **processamento paralelo**.

Processamento sequencial – Modelo de processamento caracterizado por uma arquitectura baseada numa única unidade de processamento. Também designado por **máquina de von Neumann**.

Root – ou processador raiz. Processador usado para estabelecer uma via de comunicação com o anfitrião. Toda a comunicação entre qualquer dos processadores da rede paralela e o anfitrião deve ser direccionada através desta via.

Router – ver **mecanismos de encaminhamento de mensagens**.

Scanner – analisador léxico.

Setpoint - ver **sinal de referência**.

Semantic checker – analisador semântico.

Sinal de referência - ou série de referência. Série cujo sinal de saída de um sistema, deve de algum modo em cada instante acompanhar.

SPAM - Sistema de Paralelização Automática de Algoritmos Matriciais

Software – conjunto dos meios não materiais (em oposição a *hardware*) que servem para o tratamento automático da informação e permitem a interacção entre o homem e o computador; conjunto de programas que possibilita o funcionamento do computador no tratamento do problema que lhe é posto.

Speedup – ver **aceleração.**

***Stack* –** ver **pilha**.

Taxa de transferência de dados - ver **largura de banda**.

Token – ver **lexema.**

Topologia – (…) de uma rede de processamento paralelo. Modo como os processadores desta rede estão ligados entre si. São exemplos de topologias regulares comuns árvores e anéis.

Word - ou palavra. Assumida como unidade de informação composta por 4 bytes ou 32 bits a qual pode tomar 4.294.967.296 estados.