UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

PAULO SÉRGIO MORANDI JÚNIOR

Estudo da Paralelização de um Programa de Dinâmica de Fluidos

Prof. Dr. Nicolas Maillard
Orientador

Porto Alegre, dezembro de 2006

“The only thing necessary for the triumph of evil
is for good men to do nothing.”
— EDMUND BURKE

AGRADECIMENTOS

Primeiramente agradeço aos meus pais, Evanir Rogrigues e Paulo Sérgio Morandi pela educação e a constante busca da excelência. Dois guerreiros nos quais procuro sempre espelhar-me. Vocês foram a base que me sustentou nessa longa caminhada, seja financeiramente, seja emocionalmente, seja afetivamente. Sem vocês nada teria eu para acrescentar nesse mundo.

Agradeço também as pessoas que me acolheram aqui no Rio Grande do Sul. Pessoal da NDI, Mairo, Priscilla (Pri), Kassick (Paican), Carolina (Carol), Tiagos, Thiagos e Rodrigos, Julio (não sei o que é mas eu compro), enfim todos aqueles que contribuiram com suas amizades nas horas de alegria e de tristeza. Valeu pessoal pelas noites de discussão etilicamente regada de bom humor e diversão. Vocês foram imprescindíveis para minha formação pessoal. Devo fazer um agradecimento especial também a um amigo, e vizinho, pelas *quebradas de galho*, pelas corridas frenéticas ao som de Linkin Park, pelos conselhos, por ter uma prima tão linda, por ter escutado meus problemas, pela parceria nas festas e nos velhos almoços nos velhos shopping centers, não é mesmo sr. Rodrigo Esser, vulgo Gôgui? Obrigado por toda ajuda prestada em todo esses anos. Queria agradecer também a Dalila, irmã do Gôgui e minha *prinhada* “querilda” e como diria ela, "você que tem contrato comilgo...", você é “faxa”, você sabe que eu te considero para (ok, sem palavras de baixo calão por aqui) vida inteira (melhorou). Valeu pela pareceria nos tragos e que “as nossas sejam sempre nossas...”, ops, acho que não é bem assim, mas eu prefiro essa versão.

Também quero fazer um agredecimento muito especial a mim, ao Mairo e a Pri, os sobreviventes da barra 01/2, é nóis! Mairo, valeu pela parceria nos trabalhos e desculpe à vezes que essa parceria ficou a desejar, fazer o que, o que vale é a intenção, ou não. Pri, sempre um prazer te ajudar nas mudanças e obrigado pelos conselhos e palavras de apoio. Devo agradecer também aos conselhos e horas de trago com Marcelo Claro Zembrzuski, foi inspirador e solucionador de problemas muitas vezes, as vezes com a parceria do sr. Juliano Sumino, quando a senhora sua esposa June, permitia esse encontro de discussões avançadas sobre os problemas que sempre cercam a cabeça de um homem: mulheres. E àqueles que andaram na traseira da Eleonora (Volkswagen Saveiro 1.8, ano 2000/2001), obrigado por me proporcionar deliciosos momentos de sarcasmo e alegria interna quando eu utilizava os freios de maneira brusca ou fazia uma curva um pouco acima da velocidade máxima permitida fazendo com que vocês se debatessem prazerosamente de um lado para o outro.

Diante dessas amizades que conquistei nesse estado, posso dizer que me sinto meio cidadão Rondoniense, meio cidadão Paranaense e meio cidadão Rio Grandense. Foram apenas quatro anos e meio até o presente momento, mas foi como se o tempo tivesse se dilatado e eu tenha vivido os melhores anos da minha vida. Mas acredito que anos

melhores continuarão a vir, afinal estar ao lado de quem ama proporciona momentos de imensa felicidade sempre.

Devo agradecer também ao grande Era o Que Tinha Futebol Clube, pela honra de ter defendido as cores grená no disputado Campeonato do Dacomp e as minhas sinceras desculpas aos times que se sentiram prejudicado devido minha atuação, digamos, pouco proveitosa.

Não posso deixar de citar o Grupo de Processamento Paralelo e Distribuído (GPPD, do qual fiz parte durante três anos como bolsista de IC[1]) e o Grupo de Matemática Computacional e Processamento de Alto Desempenho, em especial o professor Doutor Tiarajú Asmuz Divério, pelo incentivo a pesquisa na área de processamento paralelo que foi um motivador da escolha do tema deste Trabalho de Conclusão. Bem como ao também doutor Carlos Hölbig, orientador da bolsa, e ao Bernardo Frederes Krämer Alcalde, colega e bolsista IC também. Agradeço também aos alunos (de graduação, mestrado e doutorado) pelas festas proporcionadas nos mais diferentes congressos da área. Não poderia deixar de citar o professor Doutor Nicolas Maillard, orientador deste trabalho, também membro do grupo GPPD, que foi rigoroso quando teve que ser, sempre prestativo, participativo, incentivador e bastante paciente com o orientado, eu, no caso.

Como não poderia ser diferente, agradeço também a Universidade Federal do Rio Grande do Sul pela excelente qualidade de ensino que ao longo dos anos vêm se aprimorando ainda mais. Nós, alunos, nos sentímo honrados e privilegiados de fazer parte de uma Universidade pública e de qualidade como a UFRGS.

E uma homenagem final, mas não menos importante, à minha musa inspiradora, namorada, companheira, amiga e paciente, diga-se de passagem, muito paciente, pois teve toda a paciência do mundo para me aguentar nessa caminha final no curso de Ciências da Computação, Michelle de Souza Lima, pessoa que eu amo muito e tenho um carinho muito especial e sempre vou ter. Você é assim, um sonho para mim, já diria aquela música.

[1] Iniciação Científica

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ENS	Equações de Navier-Stokes
CFD	Computional Fluid Dynamics
EFFS	European Flood Forecasting System
EFAS	European Flood Alert System
PCAM	Partitioning, Communication, Agglomeration and Mapping

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O presente trabalho de diplomação apresenta um estudo a respeito da programação paralela, aplicando esse conhecimento na implementação de uma versão paralela de um programa seqüencial que faz a previsão da inundação de terrenos próximos do principal rio de Nottingham. Esse programa desenvolvido pela Universidade de Nottingham utiliza os conceitos da Dinâmica de Fluidos nos cálculos da inundação. Além desse estudo, fará parte também deste trabalho a reestruturação e a otimização desse programa seqüencial proposto. Será verificado, também, o desempenho do programa seqüencial otimizado e da versão paralela.

Palavras-chave: Dinâmica de Fluidos, Programação paralela, mpi.

Study and Implementation of a Parallel Dynamic Fluid Program

ABSTRACT

This bachelor's thesis presents a study in the parallel programming area which is mapped directly to make a parellel version of a sequential program initially developed at Nottingham's University that makes use of Computational Dynamic Fluid methods to compute the flood propagation of an water inflow at Nottingham's main river. Beyond this study, this thesis also make an study and implementation of optimizations into the sequential program. An restruturation of the program was also required to make this optimizations trully fast. The performance of the sequential and the parallel version of this program will also be covered by this thesis.

Keywords: CFD, parallel programming, mpi.

1 INTRODUÇÃO

Existem diversas áreas que necessitam do alto desempenho fornecido pelos sistemas multi-processados, como por exemplo a Dinâmica de Fluidos Computacional. Neste ramo, a Universidade de Nottingham desenvolveu um algoritmo para calcular a dinâmica da inundação do principal rio da cidade. Ele calcula a região alagada após um aumento no fluxo de águas do rio principal. Esse algoritmo seqüencial foi implementado e pode apresentar uma potencial demanda por grande poder computacional. Atualmente, a entrada fornecida para testes é restrita a poucos passos, logo não demanda um poder computacional excepcional para calcular a área alaguada, porém, a medida que se tomam mais passos na simulação, o tempo de execução tende a crescer muito e isso pode exigir mais poder computacional para tentar diminuir esse tempo. Caso a área de cobertura fosse ampliada, haveria problemas de armazenamento nas estruturas de dados internas do do programa também, logo uma solução paralela seria de grande utilidade, uma vez que seria possível espalhar os dados entre computadores, mas isso será visto com mais detalhes no decorrer da dissertação. Seguindo essa linha de raciociocínio, a Universidade de Nottingham apresentou uma sugestão (**?**) de paralelizar algoritmo seqüencial de maneira a reduzir o tempo de resposta, possibilitando assim uma maior quantidade de dados e de tempo de simulação a ser feita. Em parceria com a UFRGS, a Universidade de Nottingham disponibilizou a implementação desse modelo seqüencial, conhecido como Trent, para uma análise mais detalhada da paralelização desse algoritmo utilizando o suporte oferecido pelo MPI.

Tem-se como objetivo deste trabalho de diplomação, fazer um estudo da computação paralela visando a construção de um programa paralelo a partir do Trent seqüencial produzido na Universidade de Nottingham que faz-se do uso da dinâmica de fluidos computacional nos cálculos da inundação de um rio. A construção desse tipo de programa (programa paralalelo a partir de um seqüencial) está longe de ser trivial, logo como um objetivo secundário, este trabalho pretende abordar também algumas das dificuldades que podem ser encontradas ao se trabalhar com a programação paralela, em especial com MPI que exige um bom conhecimento técnico. Esse estudo e implementação irá se basear em técnicas aprendidas durante o curso de Ciência da Computação, como técnicas de otimização vista na disciplina de Compiladores, bem como sincronização entre processos visto em Sistemas Operacionais e na disciplina de Programação Distribuída e Paralela.

O primeiro passo tomado neste trabalho foi a otimização do programa Trent, que originalmente foi construído por pessoas que não detiam conhecimento técnico da área da computação e não conheciam as boas práticas da programação estruturada e nem as nuances da linguagem de programação C/C++. Será mostrado que uma simples alteração em um laço pode garantir um bom ganho de desempenho e que tornar o código mais legível e compacto pode mostrar caminhos antes não percebidos de otimizações como a de transformar duas funções aparentemente diferentes em uma só.

Após essa otimização do programa seqüencial, partiu-se para o estudo da programação paralela e da parte ténica do MPI. O resultado desse estudo foi um protótipo de validação de conceitos e uma primeira versão paralela do programa implementado pela Universidade de Nottingham. Esse estudo deu-se em duas partes, na primeira delas foi feito um estudo técnico de algumas técnicas da distribuição de dados que se farão presentes no protótipo bem como na implementação do Trent paralelo. Na segunda parte do estudo foi implementado essas ténicas em um protótipo para serem validadas e colocadas no Trent paralelo. Apenas o caso mais simples dessa distribuição foi implementada no programa paralelo. Foram feitas medições para verificar o desempenho da versão paralela implementada e apresentações de melhorias nos pontos problemáticos dessa versão de modo a tornar o programa de fato mais efeciente que o seqüencial.

Nos próximos capítulos o leitor será contextualizado aos problemas de inundações na Europa, que motivou o desenvolvimento do programa seqüencial, e também à Dinâmica de Fluídos Computacional que é utilizada nos cálculos do presente programa.

2 CONTEXTO CIENTÍFICO

A resolução numérica de problemas matemáticos é um ramo altamente explorado na comunidade científica. Os algoritmos desenvolvidos para resolver esses problemas demandam, em geral, grande quantidade de processamento. Essa demanda por processamento é fonte inspiradora de motivações à descobertas de como se obter alto desempenho. Com o surgimento de sistemas multi-processados, vislumbrou-se a possibilidade de utilizar-se desse poder computacional paralelizando as aplicações. O desenvolvimento de especificações e, por conseqüência, de bibliotecas que auxiliam na obtenção de desempenho foi inevitável, surgindo assim o MPI, uma especificação de uma biblioteca de troca de mensagens.

A Universadade de Nottingham, como a maioria das universidades e centros de pesquisas europeus, desenvolveu um programa baseado em um modelo de previsação de inundação, a ser introduzido mais tarde, chamado de Trent, que é responsável pelo cálculo da previsão de inundação do rio principal da cidade de Nottingham. A implementação desse modelo usa as técnicas da Dinâmica de Fluidos computacional em seus cálculos. Para conhecer mais sobre a Dinâmica de Fluidos Computacional ou CFD (do inglês, *Computational Fluid Dynamic*), será mostrado a teoria e as principais equações envolvidas na Dinâmica de Fluidos. Será apenas uma introdução rápida a esses conceitos, apenas para situar o leitor do presente trabalho em qual contexto está inserido o Trent e quais sãos as fórmulas matemáticas envolvidas.

2.1 Inundações na Europa

Recentemente as inundações na Europa vem chamando a atenção dos cientistas. Nos últimos anos a freqüência e as áreas atingidas vêm crescendo (**?**), bem como as tragédias proporcionadas por essas inundações. Segundo Milly (**?**), a freqüência de grandes inundações no século 20 aumentou em bacias maiores que 200000 km^2. Cresce também as evidências de que a probabilidade de extrema precipitação e de grandes inundações (*hence flooding*) são causadas pelo aquecimento global. Depois da desastrosa inundação das bacias dos rios Elbe e Danube, em Agosto de 2002, a Comissão Européia anunciou o desenvolvimento de um sistema de alerta de inundações que será capaz de prever inundações de médio porte com três a dez dias de antecedência. Desde então, um modelo de previsão de inundações da Europa, conhecido como *European Flood Forecasting System*, de agora em diante tratado apenas por *EFFS* (**?**), vem sendo construído e testado no Centro de Pesquisa Conjunta da Comissão Européia (*European Commission Joint Research Center*) (**?**).

2.1.1 Sistema de Previsão de Inundações

O objetivo principal do EFFS é desenvolver um protótipo de um sistema de previsão de inundações com três a dez dias de antecedência. Esse sistema pode ser usado como um pré-aviso em regiões que já possuem um sistema de previsão bem como servir de de apoio para regiões que não possuem tal sistema de previsão (como os países do leste Europeu). O protótipo do EFFS consiste nos seguintes componentes:

1. Modelos globais de previsão do tempo
2. Modelos otimizados de precipitação global de menor escala utilizando modelos numéricos locais de previsão do tempo
3. *Catchment Hydrology Model* divido em dois modelos:
    - Balanceamento de Água no Solo (*Soil Water Balance*, medido todos os dias)
    - Simulação de Inundação (medido de hora em hora)
4. Um modelo de inundação de alta-resolução

Esses componentes estão integrados dentro de uma ferramenta de modelagem genérica, o que permite que diferentes modelos possam ser usados em cada um dos componentes. Essa ferramenta está ligada a uma central de dados aberta e independente de arquitetura, permitindo assim o encapsulamento de vários códigos de simulação pré-existentes. Com *wrappers* é possível converter os dados de entrada, como tempo e parâmetros da base principal, para o formato particular de um modelo em particular e gravar os resultados re-convertidos na base de novo. Dentre os objetivos do EFFS, também estava o desenvolvimento de um sistema mais preciso de previsão das inundações. Atualmente, os métodos baseados somente nas descargas e nas previsões de precipitações não é suficiente para atingir períodos longos de previsão. Diante disso desenvolveu-se um modelo de balanceamento de água (*water balance*) espacialmente distribuído conhecido como LISFLOOD, formando uma rede mais detalhada de modelos em diferentes regiões da Europa. Vale salientar que o LISFLOOD não veio substituir os modelos existentes, visto que outros modelos podem integrar o EFFS. A idéia é que o LISFLOOD seja usado em regiões que não tenham um modelo próprio de previsão.

2.1.2 O Modelo LISFLOOD

O modelo do LISFLOOD é um modelo de simulação de inundação distribuído. Essa simulação é executada todos os anos para prevenção de inundações e durante uma inundação para simular as conseqüências para os dias seguintes. Com LISFLOOD pode-se saber a influência da solo e atingir maiores áreas de previsão. O LISFLOOD leva em consideração a influência da topografia, quantidade e intensidade de precipitação, o tipo do solo, o tipo de uso e o índice de umidade do solo. Os principais processos simulados pelo LISFLOOD são a chuva, a neve, o derretimento da neve, congelamento do solo, interceptação, infiltração, transpiração, evaporação, redistribuição vertical de água do solo (incluindo ascenção capilar) entre outros.

Como entrada para o LISFLOOD, tem-se medidas reais tiradas de diversos departamentos, como a base de dados Metereológica MARS. Com esses parâmetros o LISFLOOD calcula diariamente um modelo de balanceamento de água (*water balance*). As saídas

desse modelo servem de entrada para o cálculo do modelo de inundação que é executado num intervalo de 15 minutos.

O diagrama de fluxo do LISFLOOD pode ser observado na figura 2.1.

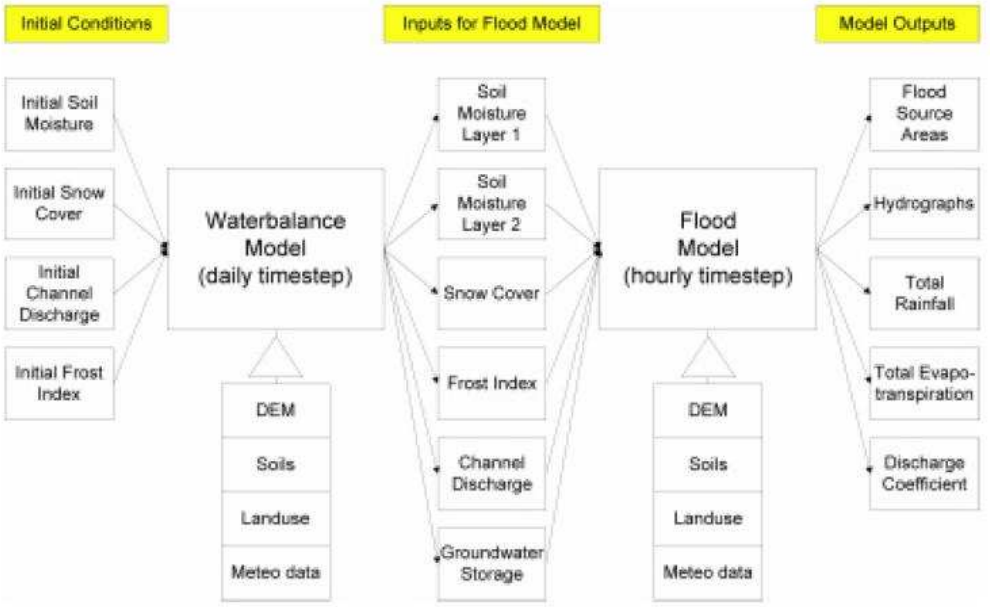

Figura 2.1: Modelo de fluxo do LISFLOOD

2.2 Introdução a dinâmica de fluidos

A Dinâmica de Fluidos é uma subdisciplina da Mecânica de Fluidos que estuda o movimento dos Fluidos, como gases e líquidos. A solução de um problema da Dinâmica de Fluidos envolve tipicamente o cálculo de propriedades dos Fluidos como velocidade, pressão, densidade e temperatura, em função do espaço e do tempo.

2.2.1 Equações da dinâmica de Fluidos

Os axiomas fundamentais da dinâmica de Fluidos são as leis de conservação[1], especificamente, conservação de massa, conservação da quantidade de momento[2] e conservação de energia.

Pode-se ainda classificar um Fluido em Newtoniano ou Não-Newtoniano. Um Fluido Newtoniano é um Fluido no qual a força de atrito é linearmente proporcional ao gradiente da velocidade na direção perpendicular do plano do atrito. A constante de proporcionalidade é conhecida como viscosidade. Por definição a viscosidade em Fluido Newtoniano é dependente da temperatura, da pressão e da composição química do Fluido, caso ele não seja uma substância pura. Num Fluido Não-Newtoniano a viscosidade muda com a aplicação da força de atrito, logo, como resultado, tem-se uma viscosidade não bem definida.

As equações de quantidade de momento para Fluidos Newtonianos são as equações de Navier-Stokes, que são equações diferenciais não lineares.

[1] leis de conservação: uma propriedade particular de um sistema físico isolado não muda enquanto o sistema evolui

[2] Primeira Lei de Newton

2.2.1.1 Equações de Navier-Stokes

As Equações de Navier-Stokes, de agora em diante abreviadas por ENS, descrevem o movimento dos Fluidos (gases ou líquidos). Essas equações estabelecem que mudanças na quantidade de momento das partículas de um Fluido é o produto das mudanças na pressão e nas forças de viscosidade dissipativas que atuam dentro do líquido. Essas forças de viscosidade descrevem o quanto viscoso é um Fluido.

As ENS são o conjunto de equações mais úteis porque descrevem a física e um grande número de fenômenos de interesses acadêmicos e econômicos. Elas são úteis para modelar o tempo, as correntes oceânicas, fluxo de água em canos, movimentos das estrelas nas galáxias, etc.

Como dito anteriormente, as ENS são equações diferenciais não lineares, e, diferentemente das equações algébricas, as equações diferenciais não se preocupam com a relação entre as variáveis de interesse (velocidade, por exemplo) e sim nas mudanças das taxas ou fluxos dessas quantidades. Em termos matemáticos essas taxas correspondem as suas derivadas. Logo, para um Fluido ideal com viscosidade zero, as ENS indicam que a aceleração (taxa de mudança ou derivada da velocidade) é proporcional a derivada da pressão interna.

Em termos práticos, apenas os casos mais simples podem ser resolvidos utilizando equações simples do cálculo matemático e a solução exata nesse caso é conhecida. Em casos mais complexos necessita-se o uso da computação para chegar em aproximações da solução exata. A ciência que estuda o uso da computação na dinâmica de Fluidos é a *Computational Fluid Dynamics* (CFD, em português Dinâmica de Fluidos Computacional)

2.2.1.2 Equações Básicas

O movimentos dos Fluidos é governado pelos princípios da mecânica e termodinâmica clássica (conservação de massa, quantidade de momento e energia). Pode-se destacar as seguintes equações de conservação:

$$\frac{d}{dt}\int_V \rho + \int_\Sigma (\ \rho\ \mathbf{u}\cdot\mathbf{v}\ )d\Sigma = 0 \tag{2.1}$$

$$\frac{d}{dt}\int_V \rho\ \mathbf{u}\ dV + \int_\Sigma [\mathbf{n}\cdot\mathbf{u}\ \rho\ \mathbf{u} - \mathbf{n}\sigma]\ d\Sigma = \int_V f_e\ dV \tag{2.2}$$

$$\frac{d}{dt}\int_V \rho\ E\ dV + \int_\Sigma \mathbf{n}\cdot[\rho\ E\ \mathbf{u} - \sigma\mathbf{u} + \mathbf{q}]\ d\Sigma = \int_V (f_e\cdot\mathbf{u})\ dV \tag{2.3}$$

Onde t é tempo, ρ é densidade, **u** é a velocidade da particula material do Fluido, σ é a tensão de stress, **q** é o fluxo de aquecimento, f_e é força externa por unidade de volume, **n** é a normal da superfície Σ incluindo o volume V do Fluido, e é a energia interna específica e E é o total específico de energia dado por:

$$E = e + \frac{1}{2}\,\mathbf{u}\cdot\mathbf{u} \tag{2.4}$$

As soluções de 2.1, 2.2 e 2.3 não são necessariamente funções contínuas e essa é a razão pela qual essas equações são escritas na forma de integrais. Entretanto, se o fluxo de densidade, velocidade e energia são suficientemente suaves, então essas equações podem ser equivalentemente transformadas em um conjunto de equações diferenciais parciais através do uso do teorema da divergência:

$$\partial_t(\rho) + \nabla \cdot (\rho\, \mathbf{u}) = 0 \tag{2.5}$$

$$\partial_t(\rho\, \mathbf{u}) + \nabla \cdot (\rho\, \mathbf{u}\, \mathbf{u} - \sigma) = f_e \tag{2.6}$$

$$\partial_t(\rho\, E) + \nabla \cdot (\rho\, E\, \mathbf{u} - \rho\, \mathbf{u} + \mathbf{q}) = f_e \cdot \mathbf{u} \tag{2.7}$$

2.2.2 Dinâmica de Fluidos Computacional

O interesse nas técnicas numéricas de como computar de maneira fácil e eficiente os métodos da dinâmica de Fluidos vem aumentando conforme cresce o poder de processamento dos computadores (**?**). As soluções da mecânica de Fluidos tem tornado-se tão importante que cerca de um terço dos pequisadores tem interesse nesse estudo. Esse ramo da ciência é conhecido como Dinâmica de Fluidos Computacional (do inglês, CFD).

Como dito anteriormente, respostas exatas dessas equações existem somente em casos simplificados. Logo na maioria dos casos essas simplificações são baseadas em aproximações e análise dimensional. Entradas empíricas são quase sempre necessárias. Para obter uma solução aproximada das equações da dinâmica de Fluidos, é necessário métodos de discretização que aproximam as equações diferenciais em um sistema de equações algébricas. Essas aproximações são aplicadas em pequenos domínios no espaço e/ou tempo, resultando assim em uma localização discreta de espaço e tempo. A precisão desses resultados é dependente da qualidade da discretização. No caso das equações de Navier-Stokes, algumas operações básicas são necessárias para a sua resolução, como por exemplo o cálculo da integral, de projeções e de derivadas. Aplicando o cálculo dessas equações básicas nos cálculos da dinâmica de Fluidos obtém-se a resposta parcial desejada.

Quando as equações tem uma solução com precisão conhecida (por exemplo as equações de Navier-Stokes para Fluidos Newtonianos que não se comprimem), soluções com qualquer precisão podem ser atigindas, a princípio. Porém, para muitos fenômenos (turbulência, combustão, fluxos de multifase, etc.) as equações exatas não estão nem disponíveis e nem possuem uma solução numérica atingível. Isso implica em uma necessária introdução a modelos de resolução. Mesmo que as equações fossem resolvidas exatamente, os valores podem não corresponder a realidade. Esses erros vindos da discretização podem diminuir utilizando-se de interpolações ou aproximações mais precisas ou então aplicar essa discretização em regiões de tamanho menor. Essa aplicação pode aumentar o tempo e o custo de obter a solução, logo uma relação custo/benefício é necessária caso essa solução seja tomada.

Tendo calculado as soluções das equações, um bom método de visualização é bem vindo. Porém é preciso ter cuidado com a qualidade dessas soluções, um resultado visual ótimo e colorido pode esconder erros nos resultados dos cálculos. é preciso analisar com cuidado e com crítica os resultados apresentados antes de tê-los como certo. Em (**?**) tem-se um exemplo de como a qualidade do resultado pode influênciar no resultado final dos cálculos.

Apesar dessa necessária preocupação com a qualidade do resultado, o presente trabalho de conclusão não entrou nesse mérito, uma vez que foge do escopo pretendido. Em trabalhos futuros, com certeza uma olhada mais crítica a respeito desses resultados é necessária.

3 IMPLEMENTAÇÃO DO TRENT

Nas seções que seguem discute-se a forma de como foi implementadado originalmente o Trent e são sugeridas algumas alterações no programa de modo a torná-lo mais eficiente. Como será visto no capítulo 4, um programa parelelo desenvolvido a partir do melhor algoritmo seqüencial é o ideal. Porém isso não implica necessariamente no melhor desempenho, devido a características intrínsecas do programa ou da arquitetura. Por isso a análise das estruturas dos dados e da reestruturação do programa é essencial para o desenvolvimento de um programa paralelo eficiente. A paralelização do programa pode servir também para aumentar o tamanho da entrada, no caso específico do Trent, pode-se aumentar a área de varredura do programa, ou seja, calcular os efeitos da inundação em áreas maiores, ou então calcular mais passos em um tempo menor.

Nesse capítulo será utilizada algumas técnicas de otimização vistas durante a disciplina de Compiladores, como a de retirar constantes do laço. Será utlizado também um pouco da Engenharia de Software, para que o código produzido seja mais legível para trabalhos futuros, como a não utilização de variáveis globais e a modularização do programa.

3.1 Análise do Programa Seqüencial

Além de tornar mais eficiente o programa, é necessário também deixá-lo mais legível para que trabalhos futuros sejam possíveis. Um programa mal estruturado pode acarretar em dias perdidos procurando onde começa uma função e onde termina outra. O Trent original, que será o alvo da análise, está mal estruturado, funções declaradas dentro de funções, mal identado (começo e fim de funções alinhados com começo e fim de trechos de código), excesso de variáveis não utilzadas (pode confudir o programador), muitas variáveis globais (péssimo para legibilidade de código), etc. Logo uma reestruturação é totalmente necessária.

3.1.1 Dimensão do problema

Antes da apresentação do programa, é importante discutir um pouco sobre a representação do problema. Para ilustrar mais essa área, tem-se as figuras 3.1 e 3.2 que representam, respectivamente, a entrada e a saída do programa. A área em verde é a entrada para o programa, que, após os cálculos, irá gerar como saída os pontos em azul da figura 3.2. O ponto preto na entrada é a fonte do alagamento, é onde começa a acumular a água.

A área coberta pelo programa é discretizada, isto é, não eh uma área contínua. É posto uma malha sobre o terreno e são escolhidos pontos que por sua vez são representados por uma matriz mas armazenada no programa na forma de um vetor. Para os testes deste

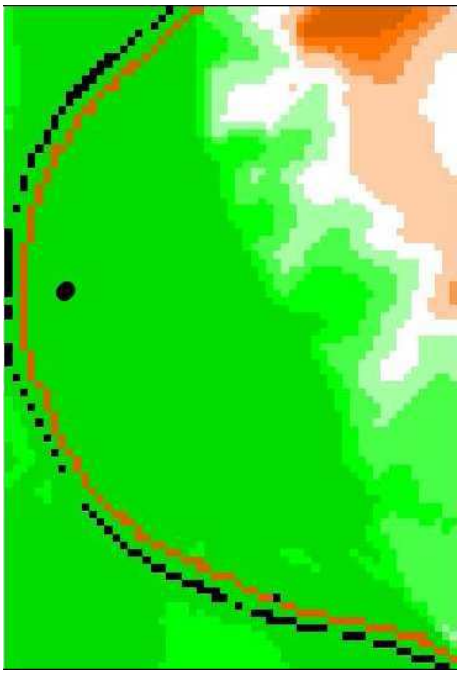

Figura 3.1: Área inicial passada como entrada para o Trent

trabalho foi disponibilizado uma entrada de dimensão 57x89, o que corresponde a um vetor de 5133 elementos. Essa matriz é obtida através de medições reais no terreno que se pretende analisar. No capítulo 5 será discutido mais sobre esta forma de representar uma matriz utilizando-se um vetor.

3.1.2 Manipulação de Arquivos

O programa Trent manipula uma série de arquivos de entrada e gera outros muitos de saída. A leitura e escrita desses arquivos estão espalhadas no código junto com a lógica do programa. O ideal é isolar essas leituras e escritas em procedimentos específicos. Para tanto, foi criado um procedimento para cada arquivo manipulado, dentre eles:

- Arquivos de entrada: parameters.dat, inflow.dat, dem.dat, spills.dat;
- Arquivos de saída: case.dat, vários arquivos impressos durante a execução de um passo do algoritmo.

São desses arquivos de saída do Trent que obtem-se a figura 3.2. Esses arquivos são repassados a um programa que gera o alagamento (cor azul). Esse programa não foi disponibilizado para este trabalho, mas o que se pretende é que a saída do programa paralelo mantenha-se a mesma do seqüencial, logo o resultado esperado como alagamento seria o mesmo.

3.1.3 Métodos principais

Para retirar declarações e implementações de procedimentos de dentro de outros procedimentos precisa-se fazer uma análise do fluxo de execução do Trent. Assim pode-se encontrar os procedimentos principais e descobrir quais outros procedimentos são chamados e em que ordem. Uma análise inicial do código produziu a árvore de execução presente na figura 3.3. A árvore de execução do programa mostra como é a estrutura dos

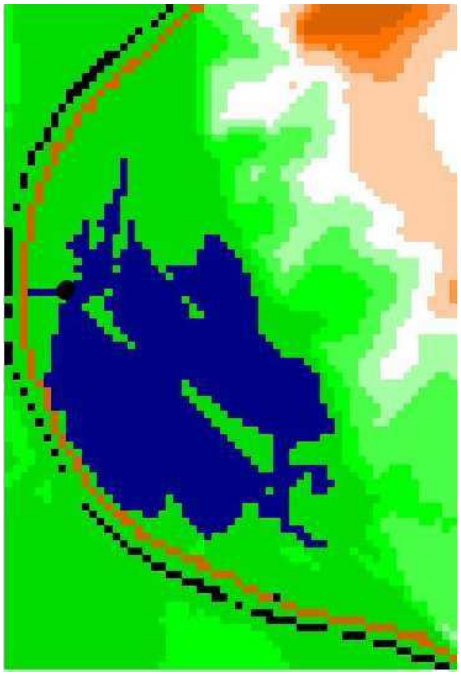

Figura 3.2: Resultado da execução do Trent, em azul a área alagada

métodos e como esses métodos são chamados. As setas indicam o método filho chamado pelo método pai.

Como pode-se observar na árvore da figura 3.3, dentro do programa principal existe um procedimento chamado *programa*. Esse procedimento é o responsável pela inicialização das matrizes utilizadas em todo o programa. Após a inicialização, o procedimendo *step_cb* é chamado em um laço de controle que itera um número pré-estabelecido de vezes. Esse procedimento é responsável pela coordenação principal do cálculo da dinâmica de fluidos. Esse procedimento por sua vez chama a função de cálculo numérico da dinâmica de fluidos, o *flow2d*. Os dois últimos procedimentos são especialmente interessantes pois executam o mesmo código e que poderiam ser unidas em um único método chamado *flow_manning*.

E assim está organizado o Trent, com funções embutidas em outras funções, com várias variáveis não utilizadas, métodos repetidos, etc. Será proposto agora uma série de alterações de modo a tornar melhor não só a legibilidade quanto o desempenho do programa.

3.2 Reestruturação do Código

3.2.1 Isolando a saída do programa

Antes de começar a reestruturação do código, é necessário isolar as saídas do programa ainda inalterado. Tendo como base essas saídas, pode-se verificar, após concluída a reestruturação, ou mesmo durante o desenvolvimento da mesma, se o programa manteve seu correto funcionamento. Os arquivos gerados pelo programa inalterado foram os seguintes:

- case.dat: a cada passo, armazena o tempo, o influxo, o volume de água sobre o terreno alagado e a profundidade da água no ponto do influxo.

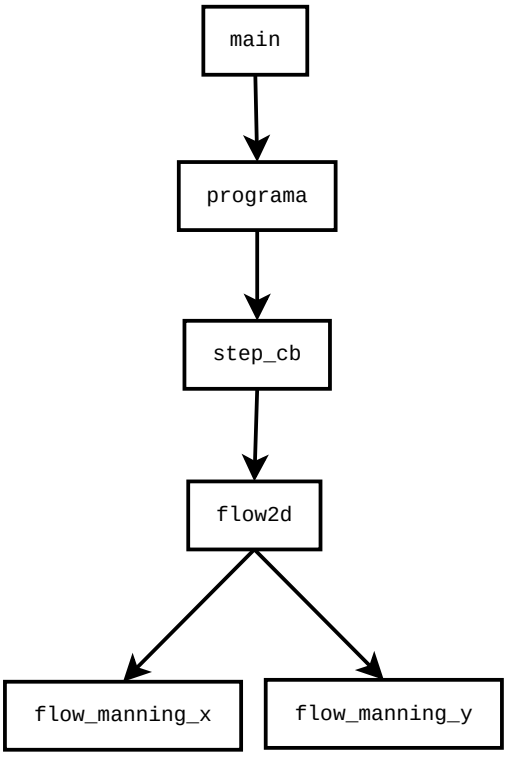

Figura 3.3: Árvore de execução inicial do programa *Trent*

- lisflood_1s1fp_v1-output.txt: armazena o tempo total gasto e tempo entre cada passo.
- state2_*.dat: Arquivos salvos a cada passo da iteração do programa, guardam informações sobre o passo-a-passo da inundação dos terrenos.

3.2.2 Automatização da verificação do resultado

Uma forma de verificar se a saída do programa novo continua coerente com o programa não-reestruturado é utilizar o programa *diff*. Com esse comando é possível verificar a diferença entre dois arquivos ou até mesmo de dois diretórios inteiros, verificando arquivo por arquivo, linha à linha de cada arquivo. Com esse programa facilmente pode-se saber se alguma alteração influenciou no resultado do programa.

3.2.3 Modularização do programa

As boas práticas de programação indicam que um programa modularizado (composto por vários procedimentos independentes) é que mostra melhores resultados a respeito de tempo gasto para uma manutenção. Da forma como estava estruturado, dificilmente o programador conseguiria achar corretamente um problema na implementação do programa. A idéia aplicada no Trent foi tirar as declarações de procedimentos feitas uma dentro das outras para um esquema mais organizado. Essa modularização ocorreu da seguinte forma:

1. Isolando entradas e saídas: todas as leituras e escritas dos arquivos foram colocadas de forma isolada recebendo como parâmetros as estruturas necessárias para o resto do código. Funções criadas:
    - read_parameters: leitura do parameters.dat

- read_inflow: leitura do inflow.dat
- read_dem: leitura do dem.dat

2. Construindo procedimentos separados para cálculo: Os procedimentos de cálculos estavam declarados uns dentro dos outros. O procedimento *step_cb* que controla os passos do algoritmo tinha um procedimento chamado *flow2d* responsável pelo cálculo da dinâmica de Fluidos que por sua vez tinha mais duas funções declaradas dentro do corpo do procedimento, *flow_manning_x* e *flow_manning_y*. A idéia é separar todos esses procedimentos.

3. Construir um arquivo de definições: Prática muito comum na programação em C/C++, trata-se de isolar do arquivo principal todas as declarações de constantes, macros e tipos novos criados.

3.2.4 Alterações nas estruturas de dados

Inicialmente, a estrutura de dados do Trent não foi bem planejada. Pela estrutura apresentada, os dados foram colocados conforme a necessidade que o programa demandava. Segue abaixo a estrutura de um nodo básico (uma célula) do Trent original:

```
typedef struct
{
   double h,hu,hv,zb,st,sb,sl,sr,cn,qm;
   int ni,nj,t,b,l,r,qwt,qwb,qwl,qwr,wet,source,ocup;
   double f1t,f1b,f1l,f1r,f2t,f2b,f2l,f2r,f3t,f3b,f3l,f3r;
   double h1t,h1b,h1l,h1r,h2t,h2b,h2l,h2r,h3t,h3b,h3l,h3r;
   double sf;
}basicnode ;

basicnode cell[MAXCELLS];
```

Um primeiro problema, que diz respeito mais a legibilidade do código, é que essas são variáveis globais. A definição da estrutura poderia estar em um arquivo de definições separado. O vetor declarado como global (`cell`) também afeta a legibilidade pelo motivo que qualquer trecho do código pode alterar essa variável, ficando difícil achar pontos de erro. Fora esse detalhe, existiam muitas variáveis declaradas na estrutura que não era utilizadas, como `cn` e `sr`. Outro ponto que chamou a atenção foram os inteiros presentes na estrutura. Fora os inteiros não utilizados, os outros inteiros, em especial `t`, `b`, `r` e `l`, eram utilizados como índice dentro do próprio vetor, como no exemplo abaixo:

```
cell[i].sb=-(cell[cell[i].b].zb-cell[i].zb)/dl;
```

No caso acima, o compilador deve calcular duplamente um endereço para acessar a memória na posição correta. Primeiro o compilador deve calcular o endereço onde está armazenado `cell[i]`. Achado esse endereço, o compilador deve buscar o valor da variável `b`. Após esse cálculo, mais uma vez o compilador deve calcular o endereço correto da variavel `zb` apontada por `cell` no índice `cell[i].b`. Ou seja, uma complicação um pouco desnecessária de cálculos de índices. E esse tipo de cálculo também vale para os outros índices citados. A interpretação natural desse índice levou a seguinte simplificação:

```
typedef struct _basicNode
{
   double h;
   double zb;
   double qm;

   struct _basicNode * pos[4];
   short qw[4];

   short ocup;

   double f1[4];

   int index;

} basicNode;
```

Ou seja, a interpretação natural referida é que os índices são indicadores dentro do próprio vetor, logo parece razoável criar um vetor que refere-se a própria estrutura (`struct _basicNode * pos[4]`). Uma análise mais detalhada do código trouxe um significado mais qualificado às variáveis. Descobriu-se que, na verdade, `b` refere-se a *bottom*, do inglês, "parte de baixo", `t` refere-se a *top*, do inglês, "parte de cima", `l` refere-se a *left*, do inglês, "esquerda", `r` refere-se a *right*, do inglês, "direita". Então pode-se reescrever o trecho de acesso `cell[cell[i].b]` da seguinte forma:

```
cell[i].sb=-(cell[i]->pos[BOTTOM]->zb-cell[i].zb)/dl;
```

Inicialmente também estava definido um matriz de inteiros que serviam de índices para o vetor global. A matriz (`link`) poderia ser reestruturada de uma maneira bem parecida com a do `basicNode`. Analisando a forma com que é utilizada a matriz tem-se os seguintes trechos de códigos como exemplo:

```
cell[link[i][j]].qwb=1;
cell[link[i][j]].qwr=1;
cell[cell[link[i][j]].t].qwb=1;
cell[cell[link[i][j]].l].qwr=1;
cell[link[i][j+1].h+=0.5*spillac[i][j]/(dl*dl);
```

Percebe-se através desse exemplo que `link[i][j]` seria um ponteiro para um nodo simples de `cell[i]` e no caso de `cell[link[i][j]].t`, `link` estaria apontando para a posição `TOP` de `cell[i]`, ou seja, essa linha poderia ser substituida por

```
link[l].node->pos[TOP]->qw[BOTTON]=1;
```

A nova estrutura do `link` deve ter então, um ponteiro para um `basicNode` atual e um para o próximo nodo do link o que resulta na seguinte estrutura:

```
typedef struct _link
{
   basicNode * node;
   basicNode * nextNode;

} connection;
```

Tabela 3.1: Análise de desempenho: Trent inicial x Re-estruturado inicial (amostra de 15 execuções)

	Trent original (seg)	**Trent Re-estruturado (seg)**
Média	30,721	16,5073
Desvio Padrão	1,11891	1,09563

Mapeando o trecho que gerou toda essa discussão para essa nova estrutura resultaria em algo do tipo

```
link[l].node->qw[BOTTON]=1;
link[l].node->qw[RIGHT]=1;
link[l].node->pos[TOP]->qw[BOTTON]=1;
link[l].node->pos[LEFT]->qw[RIGHT]=1;
link[l].nextNode->h+=0.5 * (*spillac) / (DLxDL);
```

O simples fato de reestruturar o Trent e melhorar a estrutura de dados reduziu quase que pela metade o tempo de execução do programa. A tabela 3.1 mostra esses tempos iniciais obtidos executando o Trent quinze vezes em um processador Intel Pentium III de 1.1 GHz e 1 Gb de memória RAM (considerando um conjunto de 15 execuções em seqüencia). Observe que o ganho de desempenho em relação ao programa original foi muito grande, o que mostra que esses tipos de otimizações são muito importantes.

Com essa nova versão em mãos partiu-se para as otimizações mais sutis que podem ser feitas. Existem algumas otimizações que o próprio compilador executa, porém, além de serem dependentes do compilador, o mesmo as vezes não consegue otimizar o código devido a problemas na interpretação dos dados. Por exemplo, pode ser difícil para o compilador identificar que uma variável é constante naquele trecho de código. Uma ajuda do programador é sempre bem vinda em situações como essa, bastando por exemplo, tirar uma variável de dentro de um laço como será visto mais adiante. Então, para ajudar o compilador a identificar otimizações foi trocado dentro do laço principal do programa (*flow2d*) as referências de `cell[i]` por um ponteiro para esse endereço. Isso porque esse valor era usado várias vezes e talvez o compilador tivesse dificuldade em perceber. O mesmo acontecendo para `link[l]`. O trecho abaixo ilustra o que potenciamente o compilador não conseguiria identificar:

```
link[l].node->qw[BOTTON]=1;
link[l].node->qw[RIGHT]=1;
link[l].node->pos[TOP]->qw[BOTTON]=1;
link[l].node->pos[LEFT]->qw[RIGHT]=1;
link[l].nextNode->h+=0.5 * (*spillac) / (DLxDL);
```

O elemento `link[l]` foi substituído por `linkl` que era um ponteiro para a posição `l`. Após feito esse tipo de alteração em todo o código, o tempo médio de execução continuou praticamente o mesmo, 16,6040 segundos. Isso mostra que o compilador foi sagaz o suficiente para descobrir que precisava calcular apenas uma vez o endereço de `link[l]` e que depois era só utiliza-ló nos outro casos.

3.2.5 Otimizações nas operações matemáticas

Muitas vezes uma simples alteração no código dentro de um laço faz com que o programa salte no desempenho. Certas operações matemáticas, em especial a divisão e a multiplicação, são custosas computacionalmente. Elas podem ser reduzidas adiantando-as da seguinte maneira

```
#define DL           27.
#define DLxDL        (DL * DL)
#define POWCONST     (5./3.)
```

Isso reduz o tempo gasto nos cálculos envolvendo multiplicações e divisões. Outras alterações sutis, diz-se respeito as operações matemáticas. O trecho de código mostrado no exemplo anterior `link[l].nextNode->h+= 0.5 * (*spillac) / (DLxDL)` encontra-se dentro de um laço que potencialmente será executado várias vezes. Chama a atenção o seguinte cálculo: `0.5 * (*spillac) / (DLxDL)`. Esse cálculo na realidade é uma constante, visto que `spillac` é constante durante a chamada do *flow2d*. Se retirado do laço essa constante, ela será calculada apenas uma vez, o que consequentemente aumenta o desempenho. Mas isso também é algo que a maioria dos compiladores deve detectar. Tirando essa constante do laço e realizando mais algumas pequenas alterações nesse mesmo sentido não reduziu muito o tempo de execução, mostrando que esse laço era na verdade pouco executado. O tempo médio ficou em 16,4862 segundos.

Obviamente esse ganho de desempenho obtido pela redução das operações matemáticas só será significativo se forem muitas operações realizadas (especialmente dentro de laços). Caso esse tipo de alteração não seja possível no código, pelo menos a redução do número de divisões (e multiplicações) já seria suficiente para aumentar o desempenho do programa. Se uma divisão pode ser retirada de um laço cujo números de passos seja suficientemente grande, então será possível obter um ganho de desempenho considerável. Os compiladores atuais conseguem achar esse tipo de otimização, porém, se o cálculo não estiver bem escrito, o compilador não conseguirá achar o ponto onde as divisões e as multiplicações são operações constantes e que podem ser retiradas do laço.

No caso do programa seqüencial analisado neste trabalho, o compilador não conseguiu achar o trecho de código na qual a divisão foi constante. Segue abaixo o trecho de código analisado no programa seqüencial.

```
for (i=1;i<=NCELLS;i++)
{
     if (cell[i].h > DSHORE)
     {
         inct = cfl*DL/(2.*pow(GRAVITY*cell[i].h,0.5));
         if (inct < dtnew)
            dtnew = inct;
     }
}
```

Observando o cálculo acima, percebe-se que o cálculo de `inct` possui trechos constantes durante todo o laço, pois no resto do código as variáveis `cell[i].h` e `cfl` não sofrem alterações e `GRAVITY` e `DL` são constantes definidas no início do programa. O laço é executado, segundo os dados de entrada utilizados e analisando a saída do *gprof* a ser comentado na seção de análise do desempenho do Trent, 18742804 vezes. Logo,

Tabela 3.2: Tabela de tempos de execução. Tempo antes das modificações e tempo de execução após as modificações (amostra de 15 execuções)

	Tempo antes (seg)	**Tempo depois (seg)**
Média	16,4862	14,3262
Desvio Padrão	1,09570	1,09579

retirando esse cálculo constante pode-se adquirir um ganho razoável no desempenho. Reescrevendo a equação de uma maneira mais legível tem-se a equação 3.1.

$$inct = \frac{cfl \times DL}{2 \times (GRAVITY \times cell[i].h)^{\frac{1}{2}}} \tag{3.1}$$

Dadas as informações das partes constantes pode-se reescrever a equação 3.1 da maneira descrita nas equações 3.2 e 3.3

$$mult\_factor = \frac{DL \times cfl}{2 \times \sqrt{GRAVITY}} = \frac{1}{2} \times \frac{DL \times cfl}{\sqrt{GRAVITY}} = \frac{DL \times cfl \times 0,5}{\sqrt{GRAVITY}} \tag{3.2}$$

$$inct = \frac{mult\_factor}{\sqrt{cell[i].h}} \tag{3.3}$$

Aplicando a modificação no código, tem-se:

```
mult_factor = DL * cfl * 0.5 / sqrt(GRAVITY);

for (i=1;i<=NCELLS;i++)
{
     if (celli->h > DSHORE)
     {
        temp1 = sqrt(cell[i].h);
        inct = mult_factor / temp1;
        if (inct < dtnew)
            dtnew = inct;
     }
}
```

Esse tipo de otimização não foi detectada pelo compilador pois o mesmo não consegue definir se as variáveis `cell[i].h` e `cfl` mudam durante o laço. Na tabela 3.2 tem-se o tempo obtido com essas alterações (média obtida com 15 execuções).

3.3 Análise do Desempenho do Trent

O próximo passo em direção a paralelização é analisar o desempenho do programa seqüencial de uma maneira um pouco mais detalhada do que simplemente olhando para os resultados de medição de tempos. Segundo a *lei de Amdahl* (**?**), deve-se tornar mais rápidos os caminhos mais comuns. Tornar mais rápidos os caminhos comuns de um programa influencia muito mais positivamente na performance do que a otimização de

um outro evento que ocorra raramente. Para descobrir esses caminhos comuns, faz-se o uso de ferramentas de análise como o *GNU profiler*, de agora em diante tratado apenas de *gprof* que servirá para descobrir esses caminhos. Os caminho mais comuns são os procedimentos mais utilizados e nas próximas seções será mostrado a análise do resultado do *gprof*.

3.3.1 Geração de estatísticas do programa

Para analisar a performance de um programa seqüencial, prepara-se o programa de uma maneira especial de forma que, quando o mesmo seja executado, seja gerada uma estatística que descreva, entre outras coisas, o tempo decorrido, quantidade de vezes que os procedimentos foram acessados e quais são os mais utilizados. Esse tipo de preparação é chamado de *Code Profiling* e pode ser obtido utilizando um programa chamado *profiler*. No caso deste trabalho será utilizado o profiler *gprof*.

A geração das estatísticas do programa a ser analisado ocorre em três etapas: preparação do código, geração da saída em tempo de execução e geração das estatíticas. A preparação do código ocorre em tempo de compilação. Ao compilar o código deve-se passar as opções de *Code Profile* (no caso do compilador *gcc*, *-pg*) e só então executar o programa para que as informações do programa sejam colocadas em um arquivo de saída (segunda etapa). O próximo passo é executar o programa *gprof* no programa a ser analisado para gerar a outra saída contedo os dados interpretados e as estatísticas do programa.

3.3.2 Desempenho do programa alterado

A tabela 3.3 mostra as estatísticas geradas pelo *gprof* do programa seqüencial que foi totalmente reformulado.

Tabela 3.3: Estatísticas do programa seqüencial obtidas pelo *gprof*

% Tempo	Segundos	Número de chamadas	Nome do método
$81,14$	$19,13$	12398	flow2d
$18,54$	$4,37$	18742804	flow_mannning
$0,21$	$0,05$	12398	step_cb
$0,17$	$0,04$	73	saveData

No caso, *% Tempo* significa a porcentagem do total de tempo de execução do programa que o método utilizou, *Segundos* é o tempo gasto pelo método, *Número de chamadas* é a quantidade de vezes que o método foi chamado.

Pode-se perceber através dessa saída que o método mais custoso, por conseqüência o que será alvo da paralelização, foi o *flow2d*. Uma estatística interessante é que nem sempre o método mais chamado é o que demora mais para ser executado. No caso analisado, o que acontece é que o método *flow_manning* é chamado de dentro do *flow2d*. A árvore de execução do programa também é descrita pelo *gprof* e segue na figura 3.4.

Nesse capítulo foram apresentados várias ténicas de otimização e reestruturação de código, bem como apresentado um pouco sobre *code profiling*, um poderoso aliado na análise de desempenho de um programa seqüncial. Com essa parte de otimização concluída, segue-se no trabalho discutindo-se um pouco sobre a programação paralela e seus desafios.

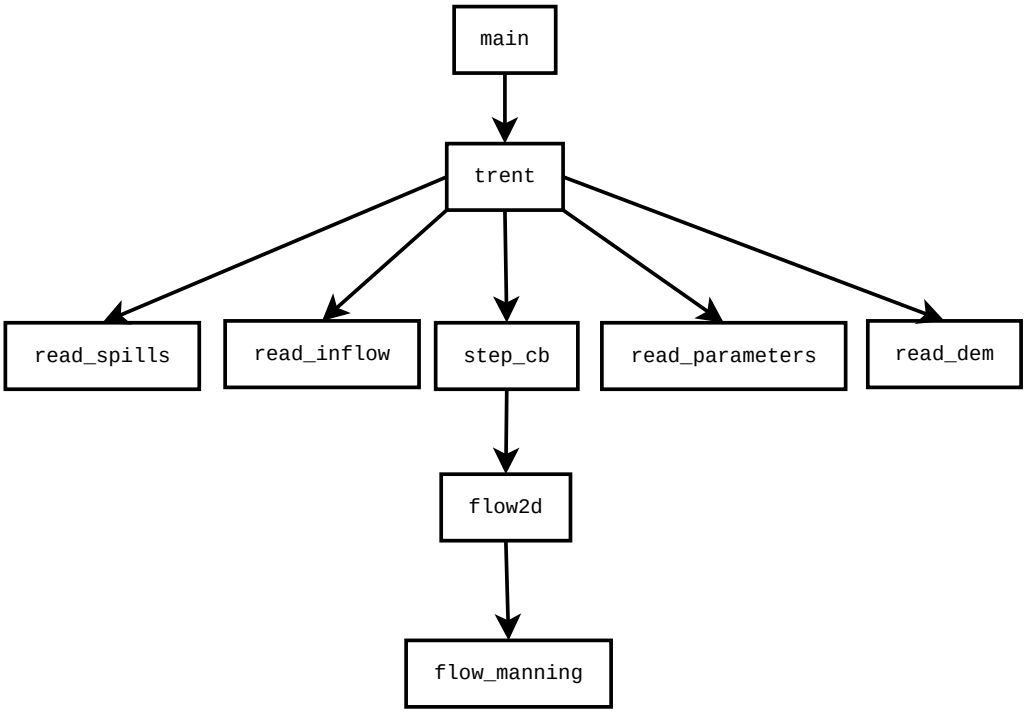

Figura 3.4: Árvore de execução do *Trent* remodelado

4 ESTUDO DA COMPUTAÇÃO PARALELA

Terminada a análise e otimização do programa seqüencial, o próximo passo do trabalho é estudar a paralelização do programa. Esse estudo envolve conhecer um pouco mais sobre a programação paralela, bem como o ambiente de execução desse programa programa paralelo. O ambiente e o programa paralelo estão intimamente ligados, pois questões como compartilhamento de memória, comunicação e distribuição dos dados são dependentes do ambiente computacional escolhido. Os mais comuns são Sistema Multiprocessado com Memória Compartilhada, Multicomputador com Troca de Mensagens e Memória Compartilhada Distribuída (**?**). Nas próximas seções será discutido mais sobre esses ambientes bem como alguns desafios da programação paralela.

4.1 Computadores Paralelos

Atualmente existe uma crescente demanda pelo aumento da velocidade de processamento para que problemas mais complexos sejam resolvidos de maneira mais rápida. A velocidade de processamento de um computador não acompanha essa demanda crescente, logo vê-se necessário uma maneira alternativa de obter esse alto poder de processamento. Uma forma de obter esse processamento é o uso de vários processadores operando em conjunto em um mesmo problema.

A idéia do computador paralelo não é nova (**?**), desde 1958 têm-se falado sobre um computador (ou sistema computacional) capaz de executar um número arbitrário de subprogramas simultaneamente. Esse computador, ou sistema computacional, poderia ser um computador especialmente feito contendo múltiplos processadores, ou vários computadores ligados em uma rede local operando como se fosse um único sistema computacional. A idéia é que n computadores (ou processadores) pudessem aumentar n vezes a computação de um simples computador. Isso seria o ideal, mas na prática isso está longe de ser atingido. Muitas vezes o problema que está sendo resolvido não pode ser divido perfeitamente em partes independentes, logo interações entre as partes são necessárias. Essas interações demandam tempo, o que influencia no tempo total da resolução do problema. Entretanto, uma diminuição substancial no tempo da resolução pode ser atingida, dependendo do problema e da quantidade de paralelismo do problema.

Fica claro que a computação paralela depende de um ambiente de computação paralelo, que pode ser descrito tanto como um simples computador com vários processadores ou vários computadores interligados. Nas próximas subseções será discutida as possíveis organizações de um computador paralelo.

4.1.1 Sistema Multiprocessado com Memória Compartilhada

Um computador convencional consiste em um processador executando um programa armazenado em uma memória principal (figura 4.1). Pode-se extender esse conceito para n processadores interconectados a n módulos de memória de modo que qualquer processador possa acessar qualquer módulo. Essa configuração pode ser vista na figura 4.2 e é chamada de Memória Compartilhada (do inglês, *Shared Memory*).

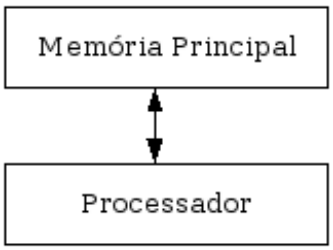

Figura 4.1: Computador convencional com um simples processador e memória

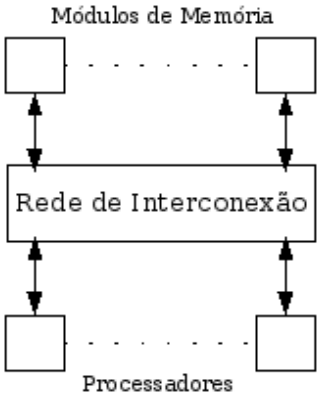

Figura 4.2: Modelo tradicional de memória compartilhada

Programar em um ambiente de memória compartilhada envolve ter um código executável armazenado na memória para que cada processador possa executá-lo. Existem várias maneiras de de desenvolver esse código. Pode ser feito uma nova linguagem de programação com construções paralelas e maneiras de se declarar variáveis compartilhadas. Assim o compilador seria o responsável por gerar o código executável final. Pode-se ampliar o poder das linguagens existentes de maneira a aplicar esses conceitos, como por exemplo a biblioteca *pthreads*, que extende a linguagem C/C++ para o conceito da memória compartilhada.

4.1.2 Multicomputador com Troca de Mensagens

O computador com memória compartilhada e multiprocessadores é um sistema de hardware especial projetado. Uma outra forma de se obter um sistema multiprocessado é interconectar através de uma rede local vários computadores simples (um processador e uma memória local). A rede local serve para que os processadores mandem informações uns para os outros através de mensagens. Esse tipo de sistema é conhecido como Multiprocessado com Troca de Mensagens (do inglês, Message-Passing Multiprocessor)

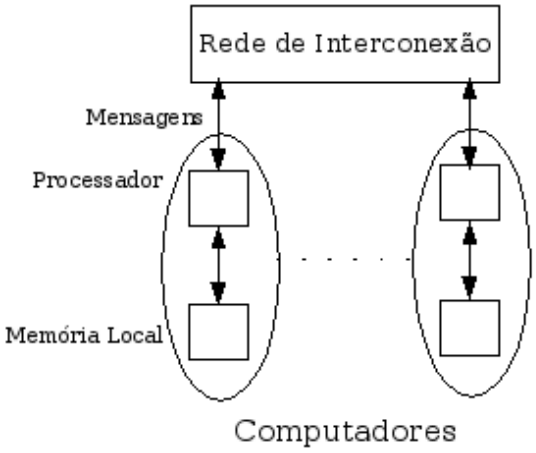

Figura 4.3: Modelo de Multicomputador com Troca de Mensagens

e pode ser visto na figura 4.3. Nesse caso os processadores da rede não tem acesso a memória local dos outros computadores, por isso é necessário as trocas de mensagens.

Programar em um ambiente como esse, também envolve dividir o problema em várias partes que são pretendidas para ser executado em paralelo. A programação pode usar uma linguagem paralela ou extender uma linguagem seqüencial. O mais comum é ter uma biblioteca de troca de mensagens que pode ser colocada em um programa seqüencial. É comum o termo processo para identificar subpartes independentes e logo paralelizáveis. O problema é divido entre os processos que são executados em computadores individuais.

4.1.3 Memória Compartilhada Distribuída

O sistema computacional com memória compartilhada distribuída seria uma união entre o modelo de troca de mensagens e o de memória compartilhada. Nesse caso, mesmo sendo fisicamente distribuídas as memórias, cada processo utilizaria um único endereço de memória global de uma maneira automatizada, que ficaria transparente para o processador de onde ele estaria buscando os dados. Essa idéia de memória é conhecida como Memória Virtual Compartilhada (do inglês, *Shared Virtual Memory*). Obviamente um acesso a memória global acrescentaria um grande atraso caso o dado não estivesse na memória do processador em questão. Além disso, seria necessário implementar um sistema de mapeamento entre as memórias que fazem parte da memória global. A única maneira de se obter memória compartilhada em um sistema cujas memórias estão fisicamentes separadas é implementando uma espécie de sistema somente-cache (*cache-only*), sem uma memória principal efetiva. E, quando for necessário, copiar os dados entre as caches. Esse tipo de sistema é pouco comum, devido ao difícil gerenciamento dessa memória global entre os computadores. Esse modelo também perde em termos de custos quando comparado ao de troca de mensagens por exemplo. A figura 4.4 ilustra esse tipo de sitema.

4.2 Considerações na Programação Paralela

Um dos grandes desafios da programação paralela é dividir o programa seqüencial em partes a serem executadas em paralelo. Para essa tarefa é necessário o conhecimento da arquitetura na qual o programa paralelo irá ser exectudado.

Dentre as estratégias de decomposição do programa existem duas que merecem des-

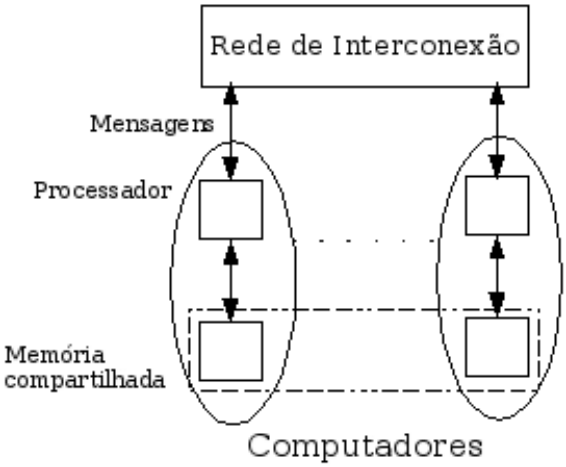

Figura 4.4: Modelo de Memória Compartilhada Distribuída

taque: por tarefas ou por dados. A divisão por dados é mais comum em aplicações científicas. Essa divisão consiste em distribuir os dados entre os processadores, muitas vezes envolvendo compartilhamento de dados nas extremidades.

Decidido qual a estratégia de decomposição do programa e o conhecimento da arquitetura, o próximo passo para a paralelização é escolher um modelo de programação: modelo de memória compartilha ou o modelo de troca de messangens.

Como visto anteriomente, o modelo de programação está intimamente ligado ao hardware. Ambos modelos possuem prós e contras, cabe ao programador escolher qual modelo se adapta melhor a sua realidade. No modelo de memória compartilhada (*Shared Memory* em inglês), como o próprio nome diz, há compartilhamento de memória entre os processadores. A forma de comunicação entre os processadores é através da memória, em um espaço de endereçamento único entre os processos. Para evitar que dados fiquem inconsistentes na memória algum tipo de controle entre os processos deve ser utilizado. Esse tipo de controle entre os processos é conhecido como sincronização de processos e pode ser feito através de recursos oferecidos pela própria linguagem como semáforos, mutexes, barreiras, etc. Do ponto de vista do programador esse tipo de modelo é atrativo uma vez que facilmente pode-se compartilhar os dados entre os processos (lembre-se que os dados nesse modelo de programação estão armazenados em uma memória de uso comum entre todos os processadores). Entretanto, é difícil implementar um hardware para que sejam atingidos valores baixos de tempo de acesso a memória. Na prática esses tipos de sistema aplicam níveis diferentes de hierarquia de memória. Em geral, hardware com esse tipo de configuração são caros e difíceis de encontrar na prática.

No caso do modelo de troca de mensagens, as memórias estão distribuídas, assim, a forma de comunicação dos processos é através da troca de mensagem. Assim como no modelo de memória compartilhada, mecanismos de sincronização são necessários para evitar estados inválidos. Nesse caso também o programador deverá utilizar os recursos da própria linguagem de programação para enviar os dados de um processo para outro. Esse modelo de programação é mais comum de ser utilizado por ser diretamente aplicável a computadores ligados em rede, sem nenhum design especial no hardware (procesador e memória, no caso) e é o escolhido para este trabalho de diplomação. Uma vantagem desse modelo é a escalabilidade dos computadores. Se um computador está ultrapassado (em termos de memória e processador), a sua substituição é imediata e sem acarretar danos

para o sistema de multicomputadores. Caso novas máquinas sejam adquiridas, sua integração na rede existente é trivial, diferentemente do modelo de memória compartilhada, que uma mudança simples acarretaria em uma mudança total do hardware.

Em ambos os casos é necessário um cuidado maior do programador na sincronização dos processos, pois facilmente pode-se criar situações de inconsitência ou situações inválidas permanentemente como o dead-lock. Por exemplo, devido a um descuido do programador, o processo A fica esperando receber uma mensagem do processo B e o processo B entra também numa espera de mensagem do processo A. Nunca mais os processos sairão desse processo de recebimento, trancando para sempre a execução do programa (salvo em casos que essa espera é por tempo limitado, conhecido como tempo de timeout). Na programação, para atingir a alta performance, o programador deve olhar principalmente a escalabilidade e balanceamento de carga de um programa paralelo.

Outro fator que pesa a favor do modelo de trocas de mensagens é o fato da crescente disseminação dos *clusters* de computadores. Segundo (**?**), *clusters* de computadores nada mais é do que um aglomerado de computadores pessoais (workstations) ligados em uma rede local especializada. Hoje em dia é muito mais vantajoso se ter um cluster do que se ter um supercomputador. Além de caros, rapidamente um supercomputador é superado por outro modelo mais novo que custa um valor na casa dos milhões de dólares. Enquanto que, em um cluster, a maioria dos computadores são computadores pessoais baratos (as vezes existe um servidor que é uma máquina um pouco mais avançada e cara que os demais). E, caso um computador se torne obsoleto, sua substituição é bem inferior, em termos de custos, à substituição de um supercomputador. Além do mais, facilmente se chega ao mesmo poder de processamento total de um supercomputador usando clusters de computadores. A idéia com que foi concebida o cluster já mostra que seu objetivo é um alto desempenho por baixo custo (**?**). Pesquisadores da NASA precisavam de um computador que fosse poderoso o suficiente para resolver seus problemas mas não queriam gastar muito com isso. Então surgiu a idéia de ligar alguns computadores pessoais velhos que existiam no centro espacial em rede e configurá-los de maneira a trabalhar como um único sistema. Essa idéia deu certo e hoje os clusters são uma realidade na maioria dos centros de pesquisa universitários e comerciais. A Universidade Federal do Rio Grande do Sul (de agora em diante UFRGS) não fica atrás dos grandes centros de pesquisa no mundo, possui dois clusters dedicados a pesquisa do processamento de alto desempenho. Esse ambiente será o utlizado para executar testes e simulações deste trabalho de diplomação.

4.3 Especifações de um modelo de Troca de Mensagens

Muitas variações do modelo de troca de mensagens foram implementadas e os programas só poderiam ser portáveis de um sistema para outro com muitas dificuldades. Vários vendedores de sistemas de computadores paralelos distribuiram diferentes bibliotecas de troca de mensagens proprietárias. Os fatores que contribuíram para essa situação foram a falta de um padrão definido e a competição do mercado, mesmo que isso tornasse as aplicações não portáveis.

Em Abril de 1992, o Centro de Pesquisa em Computação Paralela patrocinou um workshop de um dia sobre Padrões para Troca de Mensagens em um Ambiente de Memória Distribuída. Depois, em Novembro, na conferência de Supercomputação, um comitê formalizou um padrão para troca de mensagens. Esse padrão é conhecido como MPI (*Message-Passing Interface*) e tem como objetivo:

- Definir um padrão portável de troca de mensagens;
- Operar de maneira aberta, permitindo que qualquer um junte-se a comunidade para discutir, seja pessoalmente, seja via lista de email de discussões.

Dentre os participantes desse fórum, conhecido como MPI Forum (**?**), destacamos Convex, Cray, IBM, Intel, Meiko, nCUBE, NEC e Thinking Machines. O padrão MPI foi finalizado em Maio 1994 e tem as seguintes características principais:

- MPI é uma biblioteca que especifica nomes, seqüencias de chamadas e resultados de subrotinas;
- MPI é uma especificação, não uma implementação em particular;
- MPI implementa o modelo de troca de mensagens, a computação continua uma coleção de processos que se comunicam via mensagens.

Várias implementações desse padrão surgiram desde então, mas duas merecem destaque: *mpich* e *mpi-lan*. Nas próximas subseções discute-se os comandos básicos de sincronização entre os processos utilizando o padrão especificado pelo MPI bem como um pouco sobre o funcionamento dos programas feitos em MPI.

4.3.1 Comandos básicos

A sincronização entre processos ocorre quando um processo necessita de alguma informação contida na memória local de um outro processo. Nesse momento o processo que contém a informação deve *enviar* a informação para o processo que precisa dessa informação, processo esse que deve executar um *recebimento*. Essas são as primitivas básicas de comunicação e que estão presentes também na especificação:

- *MPI_Send*: primitiva de comunicação utilizada para enviar dados de um processo para outro de maneira síncrona (o processo fica bloqueado até que o envio seja completado);
- *MPI_Isend*: o mesmo que o MPI_Send, porém de maneira assíncrona, ou seja, o processo segue o processamento mesmo que o envio não tenha sido completado. Cabe ao programador evitar que o buffer utilizado no envio seja alterado antes que seja completado.
- *MPI_Recv*: primitiva de comunicação utilizada para receber dados de um processo, também de maneira síncrona.
- *MPI_Irecv*: assim como o envio, o recebimento também pode ser feito de maneira assíncrona.

Outras primitivas de comunicação não menos importantes são as seguintes:

- *MPI_Bcast*: primitiva de comunicação utilizada para enviar dados para todos os processos de um grupo de comunicação. Todos os processos devem fazer *MPI_Bcast*, e um dos parâmetros dessa função é quem é o processo “root”, ou seja, de qual fonte a informação a ser difundida deve vir. Aquele processo que tiver a identificação igual ao “root” é o processo fonte, todos os outros serão receptores;

- *MPI_Allgather*: primitiva de comunicação usada para coletar e distribuir uma informação por todos os processos de um grupo de comunicação;
- *MPI_Barrier*: Sincroniza todos os processos. Quando um processo executa esse comando, o mesmo é bloqueado até que todos os outros processos atinjam esse ponto;
- *MPI_Comm_size*: Retorna a quantidade de processos envolvidos na execução do programa;
- *MPI_Comm_rank*: Retorna a identificação do processo corrente.

Maiores detalhes sobre essas funções, como parâmetros ou algum outro comportamento que não o explicado brevemente aqui pode ser consultado em (**?**) ou (**?**).

4.3.2 Funcionamento básico

O funcionamento básico de um programa MPI consiste em que todos os processos executam o mesmo código. Cabe ao programador diferenciar através do *rank* (identificação do processo) do processo quem é responsável por qual parte do código. Por exemplo, em um modelo de mestre-escravos, onde o rank 0 é o mestre e todos os outros são escravos, podería seguir com o seguinte pseudo-código:

```
SE rank é 0
        ENTÃO
                Inicializa dados
                Distribui dados (MPI_Send)
                Executa cálculos locais (caso necessário)
                Recebe resultados (MPI_Recv)
        SENÃO
                Recebe dados (MPI_Recv)
                Executa cálculos locais
                Envia resultados ao mestre (MPI_Send)
```

Analisando o trecho de código acima, pode-se perceber o desafio do programador em encontrar os pontos paralelizáveis e a dificuldade na sincronização entre os processos. Um exemplo dessa utilização mestre-escravo pode ser visto em (**?**).

4.4 Distribuíndo os dados

Depois da análise de como implementar um programa MPI, parte-se para a análise de como distribuir os dados entre os processos. Várias técnicas de distribuição existem atualmente e aqui serão abordadas as principais delas.

4.4.1 Considerações iniciais da distribuição dos dados

Como visto anteriormente, na programação paralela pode-se dividir o trabalho a ser executado dividindo-se os dados e distribuíndo os mesmos entre os processos (em geral, um processo está associado a um processador, mas existem casos de mais de um processo estar associado ao mesmo processador, nesse caso o paralelismo não seria o ideal). Esse tipo de distribuição é largamente estudado e existem diversas referências sobre esse

Tabela 4.1: Decomposição por Blocos. 3 Blocos, 4 elementos para o processo 0 e 1, 2 elementos para o processo 2.

m	0	1	2	3	4	5	6	7	8	9
p	0	0	0	0	1	1	1	1	2	2
i	0	1	2	3	0	1	2	3	0	1
bloco	0	0	0	0	1	1	1	1	2	2

Tabela 4.2: Conteúdo dos vetores locais a cada processo. No caso esses vetores são da decomposição por blocos.

i	0	1	2	3
vec_bloc0	0	1	2	3
vec_bloc1	4	5	6	7
vec_bloc2	8	9	-	-

assunto e foge do escopo do trabalho tratar de todas. Entretanto, três tipos em especial merecem destaque: a decomposição por *bloco*, a decomposição *cíclica* e a decomposição *bloco-cíclica*.

A decomposição por blocos, comumente usada quando a carga computacional é distribuída homogeneamente em uma estrutura de dados regular como um grade Cartesiana, associa entradas contíguas do vetor global a processadores em blocos. A tabela 4.1 mostra um exemplo de distribuição em blocos de um vetor de 10 elementos, onde m é o índice no vetor global, p é a identificação do processo, i é o indice dentro do bloco de cada processador e $bloco$ é o índice do bloco.

A decomposição cíclica é comumente utilizada para aprimorar o balanceamento de carga quando a carga computacional é não homogeneamente distribuída em uma estrutura de dados regular. Como na decomposição por blocos, cada processo fica responsável por um bloco de dados. E cada bloco contém exatamente um elemento do vetor global. Logo, ciclicamente alternando os processos, pode-se dividir os dados entre os processos de maneira a se obter um melhor balanceamento de carga. Um exemplo dessa distribuição pode ser vista na tabela 4.3 (essa tabela também segue as definições dadas no exemplo de decomposição por blocos).

Considerando que $vec\_bloc\mathbf{x}$ é um vetor local a cada processo e x é o índice de cada bloco, o conteúdo desses vetores são os índices do vetor global, as tabelas 4.2 e 4.4 mostram exemplos desses vetores. Observe nessas tabelas a diferença na quantidade de índices que cada processo obteve. A tabela 4.4 possui um melhor balanceamento de carga.

A decomposição cíclica parece em um primeiro momento mais adequada para a maioria dos casos, visto que no caso da decomposição por blocos acontece de processos ficarem extremamente atarefados enquantos outros processos rapidamente se tornam inativos. Porém existe um fator importantíssimo que deve ser levado em consideração: o peso da comunicação entre os processos. Primeiro considere que o envio de um elemento seja o triplo do tempo para se calcular uma soma e que exista um cálculo suficientemente complexo que exija que se saiba informações dos elementos vizinhos, como por exemplo, o elemento de índice 3 precisa do resultado da soma dos elementos de índice 1, 2 e 4. De

Tabela 4.3: Decomposição Cíclica. Mapeando um índice m na tupla $(p, i, bloco)$.

m	0	1	2	3	4	5	6	7	8	9
p	0	1	2	0	1	2	0	1	2	0
i	0	0	0	1	1	1	2	2	2	3
bloco	0	1	2	0	1	2	0	1	2	0

Tabela 4.4: Conteúdo dos vetores locais a cada processo. No caso esses vetores são da decomposição cíclica.

i	0	1	2	3
vec_bloc0	0	3	6	9
vec_bloc1	1	4	7	-
vec_bloc2	2	5	8	-

acordo com a decomposição cíclica (tabela 4.4), tem-se que será necessário o envio de 3 elementos, onde o processo 2 enviaria o elemento de índice 2 e o processo 1 enviaria o elemento de índice 1 e 4, para o processo 0 efetuar os cálculos. Observa-se que no caso da decomposição por blocos (tabela 4.2) essa comunicação reduziría-se ao envio de um elemento por parte do processo 1, o que tornaria bem mais eficiente o cálculo.

Logo, a união entre esses dois métodos vê-se necessária para obtenção de uma distribuição mais eficiente, essa distribuição de dados é a já conhecida *bloco-cíclica* e um exemplo dessa distribuição pode ser vista nas tabelas 4.5 e 4.6. Para o exemplo dado, além de uma distribuição mais balanceada da carga, o número de comunicações também foi reduzido. A idéia básica da decomposição bloco-cíclica é aumentar o cálculo local de uma maneira a manter o balanceamento de carga e diminuir a probabilidade de comunicação entre processos.

O próximo passo é estudar as fórmulas envolvidas nos cálculos dos índices, para que se possa compreender os controles que devem ser feitos no protótipo em questão. Esse controle consiste em definir qual o processo será responsável por qual parte do vetor global nos seus cálculos locais e será mostrado na próxima seção.

4.5 Fórmulas de Mapeamento e de Controle

Nos cálculos matemáticos será mencionado “vetor global” e “cálculo a ser feito” por um processo. Está se considerando como um vetor global um vetor de dados qualquer de entrada que sofrerá um processamento que é o “cálculo a ser feito”. Esse cálculo nada mais é do que alguma modificação e/ou cálculo feito em cima do vetor global. Esse cálculo pode ser, por exemplo, um cálculo simples como a média dos elementos do vetor global, ou até um cálculo mais complexo como a área alagada por um rio. A montagem do resultado final não será considerada nesse capítulo.

Primeiramente algumas definições de constantes utilizadas nos cálculos matemáticos que seguirão nesta seção. Considere que $BLOC\_SIZE$ é a quantidade de índices do vetor global que um processo deve utilizar nos cálculos, ou seja, o tamanho do bloco (assim tratado de agora em diante) e que $SIZE$ é a quantidade de processos que irão

Tabela 4.5: Decomposição Bloco-Cíclica. Mapeando um índice m do vetor global em uma tupla $(p, i, bloco)$.

m	0	1	2	3	4	5	6	7	8	9
p	0	0	0	1	1	1	2	2	2	0
i	0	1	2	0	1	2	0	1	2	0
bloco	0	0	0	1	1	1	2	2	2	3

Tabela 4.6: Conteúdo dos vetores locais a cada processo. No caso esses vetores são da decomposição bloco-cíclica. Observe que o tamanho dos blocos é 3 e processo 0 ficou com 2 blocos.

i	0	1	2	3
vec_bloc0	0	1	2	9
vec_bloc1	3	4	5	-
vec_bloc2	6	7	8	-

realizar a tarefa. Considere também que N é o tamanho (dimensão) do vetor global. Considere também, como na seção anterior, que m é o índice no vetor global, p é a identificação do processo, i é o índice dentro do bloco de cada processador e $bloco$ é o índice do bloco.

Tem-se que o índice $bloco$ é dado por:

$$bloco = \lfloor \frac{m}{BLOC\_SIZE} \rfloor \tag{4.1}$$

onde $BLOC\_SIZE$ é o tamanho dos blocos. O tamanho dos blocos pode ser um parâmetro configurável. Os valores desse parâmetro podem se enquadrar em alguns casos especiais que serão tratatos mais adiante. O processo p responsável pelo bloco de índice $bloco$ pode ser obtido da seguinte maneira

$$p = bloco \ mod \ SIZE \tag{4.2}$$

onde mod é o resto da divisao inteira de $bloco$ por $SIZE$.

Agora será mostrado uma generalização da obtenção dos índices m (do vetor global) que o processo p será responsável. Com isso tem-se $SIZE$ vetores locais contendo $BLOC\_SIZE$ índices para o vetor global. Essa generalização mostrará que, na verdade, trata-se da decomposição bloco-cíclica e que as distribuições por bloco e a cíclica são um caso especial da bloco-cíclica.

Considerando que no vetor local tem-se um conjunto de índices globais, pode-se definir esse conjunto como sendo E_PROC. E_PROC nada mais é do que uma união de k conjuntos de índices. Mas antes de se definir esses conjuntos, há a necessidade de mostrar o cálculo de um índice isolado que será tratado por um processador p. Logo, o j-ésimo índice do vetor global, dado o processador p e uma variável k pertencente ao conjunto dos números Naturais, tem-se que

$$i_{j(k,p)} = BLOC\_SIZE \cdot k \cdot SIZE + BLOC\_SIZE \cdot p + j \tag{4.3}$$

Variando o índice k, tem-se os subconjuntos $Conj_{k,p}$ de índices do vetor global definidos como se segue

$$Conj_{k,p} = \{i_{0(k,p)}, i_{1(k,p)}, \ldots, i_{(BLOC\_SIZE-1)(k,r)}\} \tag{4.4}$$

Observe que cada subconjunto possui no máximo $BLOC\_SIZE$ elementos. No fim, a união de todos os subconjuntos será o conjunto de índices que o processo irá trabalhar. Existe a seguinte restrição aos elementos do subconjunto

$$\{\forall x \in Conj_{k,p} \mid 0 \leq x \leq N-1\} \tag{4.5}$$

ou seja, os índices do subconjunto obviamente não devem ser maior que a dimensão do vetor global. Definindo mais formalmente o conjuto E_PROC do processo p tem-se

$$E\_PROC_p = \{Conj_{0,r} \cup Conj_{1,r} \cup \ldots \cup Conj_{k-1,r} \mid k \in \mathbb{N}\} \tag{4.6}$$

ou seja, tem-se k subconjuntos de índices do vetor global a ser processado.

Com isso é possível saber exatamente quais os índices que o processador p irá receber para fazer os cálculos, bem como saber qual bloco $bloco$ o processador p será responsável dado um índice m do vetor global. Outras constantes importantes também devem ser definidas. A quantidade total de blocos, de agora em diante tratado como N_BLOCS, que o vetor global foi dividido pode ser dado por

$$N\_BLOCS = \begin{cases} \frac{N}{BLOC\_SIZE} & \text{se } N\%BLOC\_SIZE = 0 \\ \\ \frac{N}{BLOC\_SIZE} + 1 & \text{se } N\%BLOC\_SIZE \neq 0 \end{cases} \tag{4.7}$$

A quantidade de blocos $BLOC\_PER\_PROC$ a serem processados pelo processo p:

$$BLOC\_PER\_PROC_p = \begin{cases} \frac{N\_BLOCS}{SIZE} + 1 & \text{se } p < (N\%BLOC\_SIZE) \\ \\ \frac{N\_BLOCS}{SIZE} & \text{se } p \geq (N\%BLOC\_SIZE) \end{cases} \tag{4.8}$$

4.5.1 Casos especiais

No caso de $BLOC\_SIZE = \lfloor \frac{N}{SIZE} \rfloor$, ou seja, caso o resto da divisão seja zero (isto é, a divisão é inteira), se trata de uma distribuição por blocos. Logo, substituindo essa informação de $BLOC\_SIZE$ em 4.3, tem-se:

$$i_{j(k,p)} = \frac{N}{SIZE} \cdot k \cdot SIZE + \frac{N}{SIZE} \cdot p + j = N \cdot k + \frac{N}{SIZE} \cdot p + j \tag{4.9}$$

Como $i_{j(k,p)}$ deve ser um valor entre 0 e $N-1$, o único k que satisfaz essa restrição é o $k = 0$, o que simplifica a fórmula para

$$i_{j(0,p)} = \frac{N \cdot p}{SIZE} + j = BLOC\_SIZE \cdot p + j \tag{4.10}$$

ou seja, há exatamente um único subconjunto de índices (a saber $Conj_{0,p}$) que conterá todos os $BLOC\_SIZE - 1$ índices do vetor global a serem utilizados pelo processo p.

Outro caso especial é quando o $BLOC\_SIZE = 1$. Aqui tem-se a distribuição cíclica. Novamente, substituindo-se em 4.3, tem-se:

$$i_{j(k,p)} = k \cdot SIZE + p + j \tag{4.11}$$

Com isso demonstrou-se toda teoria envolvida nos controles dos laços e de sincronizações entre os processos. De posse dessas informações, será implementado um protótipo de um programa paralelo que irá validar essas fórmulas discutidas bem como preparará o ambiente que será inserido a versão paralela do Trent.

5 PARALELIZAÇÃO DO TRENT

Após as considerações sobre o desenvolvimento de um programa paralelo, o próximo passo é desenvolver um protótipo considerando apenas os controles sobre os processos. Esse controle é muito importante no programa paralelo. Como visto anteriormente, esses controles é que definem a lógica de paralelização do programa. Num primeiro momento serão apresentados os controles de laços, depois será mostrado uma técnica de programação muito útil que pode evitar que *deadlocks* aconteçam, bem como erros de lógica na programação.

Depois essas alterações serão mapeadas no programa Trent e então tem-se o programa paralelo para ser executado no cluster. Apesar do protótipo abranger todos os casos de distribuição de dados, o Trent paralelo foi implementado utilizando o caso mais simples da distribuição que é o por blocos. Essa primeira versão paralela mostrará todas as características e comportamento dos dados no ambiente paralelo, ficando claro a forma de propagação das informações e, o mais importante, quais, como e quando as informações precisam ser efetivamente trocadas entre os processos.

5.1 Estruturas de Controle e Considerações do Protótipo

Como dito no capítulo anterior, como todos os processos executam o mesmo código, cabe ao programador decidir qual trecho de código deve ser executado por qual processo. No caso do Trent, o que se quer é que cada processo seja responsável por um pedaço do vetor principal. O que esse protótipo prentende simular é justamente os controles necessários para que cada processador cuide de um pedaço do vetor. Aqui cabe um parênteses importante sobre o vetor global. Esse vetor representa, na verdade, no Trent, uma matriz da estrutura de dados apresentada no capítulo anterior. A tabela 5.1 mostra como funciona esse mapeamento de uma matriz em um vetor no Trent. Em negrito os índices da matriz e dentro os índices do vetor. Exemplo: considere uma matriz A conforme a tabela em questão. Então o elemento $A(2,3)$ é representado no vetor global v pelo elemento $v(13)$. Tem-se também que o vizinho TOP é o elemento $v(8)$, o vizinho $BOTTOM$ é o elemento $v(18)$, o vizinho $LEFT$ é o elemento $v(12)$ e por fim o vizinho $RIGHT$ é o elemento $v(14)$.

Para uma tentativa de minimizar as comunicações, decidiu-se dividir a matriz em colunas para cada processo, ou seja, cada processo é responsável por uma certa quantidade de colunas e, caso seja necessário, haverá comunicação nas fronteiras da coluna. Assim os elementos TOP e $BOTTOM$ estarão sempre no mesmo processo, mas os elementos $RIGHT$ e $LEFT$ podem estar em outros processos, nesse momento será necessário uma sincronização entre processos. Por questões de simplicidade no protótipo o vetor global não representa uma matriz, apenas um vetor normal e cada processo é responsável por

Tabela 5.1: Exemplo de representação de uma matriz 5x4 em um vetor de 20 posições.

	0	**1**	**2**	**3**	**4**
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

uma quantidade de elementos desse vetor.

É importante citar também que o trabalho em questão fará uso de primitivas de comunicação não bloqueantes, ou seja, o processo vai executar envios e recebimentos de maneira assíncrona e continuará o processo normalmente. Outra importante consideração é que o processo não pode encerrar enquanto todos seus envios e recebimentos não forem completados e isso deve ser considerado no protótipo também.

O algoritmo pensado para o protótipo foi o seguinte:

```
1. Inicializacao do vetor global
2. Executa todos os recebimentos assincronos
3. Para cada bloco do vetor
     3.1 Executa calculo local
     3.2 Se possui um vizinho a direita do processo local
          3.2.1 Envia resultado local de maneira assincrona
                para o vizinho da direta
     3.3 Se possui um vizinho a esquerda do processo local
          3.3.1 Envia resultado local de maneira assincrona
                para o vizinho da esquerda
4. Espera que todos os envios e recebimentos
   sejam concluidos
```

Cada parte do algoritmo será detalhada e analisada nas próximas seções[1].

5.1.1 Análise do algoritmo

Na parte de inicialização do vetor global é importante que somente a parte que o processo é responsável seja inicializada, uma vez que as outras partes não entram no cálculo, seria um desperdício de tempo de processamento inicializar todo o vetor (no caso do protótipo esse valor seria desprezível visto que não existe nenhuma complexidade nessa inicialização). Antes de mostrar o trecho de código responsável pela inicialização, algumas definições se fazem necessárias. Para saber se um processo é responsável por um índice `i` qualquer, dado que existem `proc_size` processos e `BLOC_SIZE` blocos é definido na seguinte macro:

```
#define proc(i,proc_size) (((i)/BLOC_SIZE)%(proc_size))
```

Essa macro é a junção das fórmulas 4.1 e 4.2 mostradas no capítulo 4. Após essa observação segue abaixo o trecho de código que inicializa os dados. Considere que `rank` é

[1]Nos trechos de códigos a seguir foram retirado os comentários e algumas variáveis que não eram importantes e nem influenciam na lógica do programa. Eram apenas variáveis para depuração.

a identificação do processo em questão e que `size` é a quantidade de processos envolvido no cálculo.

```
for(i=0;i<N;i++)
{
    if( rank == (proc(i,size)) )
    {
        test[i].index = i;
        test[i].value = i;
    }
}
```

No caso, é alocado (reservado espaço na memória) um vetor de `N` posições para `test`. Poderia ser alocado apenas a quantidade de blocos que cada processador ficaria responsável. Mais uma vez, por questões de simplificação, utilizou-se o vetor com `N` posições.

O próximo passo é a execução dos recebimentos assíncronos. Essa é uma das grandes vantagens de se usar comunicações assíncronas, uma vez que pode-se seguir o processamento normalmente. A sincronização acontece aos pares, quando um processo executa um recebimento, seja ele síncrono ou não, deve existir um envio por outro processo, senão o processo que ficou em recebimento pode ser bloqueado para sempre caracterizando um *deadlock*. Logo, se fosse possível prever antecipadamente a quantidade de envios que devem ser feitos, a quantidade de recebimentos seria automaticamente descoberta. Essa previsão é interessante caso haja a necessidade de se receber várias informações de vários vizinhos diferentes e essa informação fosse útil somente no final, sem influenciar no cálculo local. O protótipo em questão usará essa abordagem, o objetivo final é ter um vetor preenchido com todos os cálculos locais de cada processador (observe que esse vetor teria o tamanho da quantidade de blocos que um processo terá que manipular - ver fórmula 4.8). Ou seja, não há a necessidade dos dados dos vizinhos para os cálculos locais.

Seria fácil supor, então, que cada processo irá fazer pelo menos *dois* recebimentos/envios conforme a figura 5.1, uma vez que cada processo tem um vizinho a esquerda e a direita. Observe que na figura existe uma peculiaridade com relação aos processos dos extremos, ou seja, eles não possuem vizinhos, logo o processo 0 em questão deve fazer um recebimento a menos pois não tem vizinho a esquerda, idem para o processo 2 que não possue vizinho a direita. Mas essa tarefa não é tão simples quanto parece, pois observando as outras figuras (5.2 e 5.3) tem-se a clara idéia da complexidade envolvida na sincronização de processos. A figura 5.3 mostra uma particularidade de quando o processo inicial, que no caso do protótipo é o primeiro a começar receber os blocos a serem processados, além de não possuir vizinhos a esquerda, também é o processo mais a direita, então nesse caso específico seriam dois recebimentos a menos. O trecho de código responsável pelo cálculo dos recebimentos é o seguinte:

```
if( rank < (N_BLOCS % size) )
{
   number_of_irecvs = 2 * ((N_BLOCS/size) + 1);
}
else
{
   number_of_irecvs = 2 * ((N_BLOCS/size));
}
```

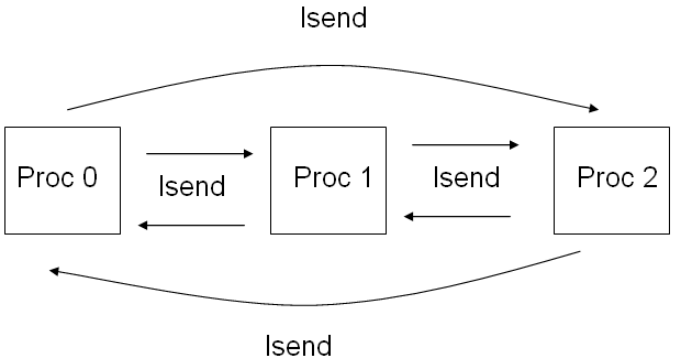

Figura 5.1: Troca de mensagens entre os vizinhos

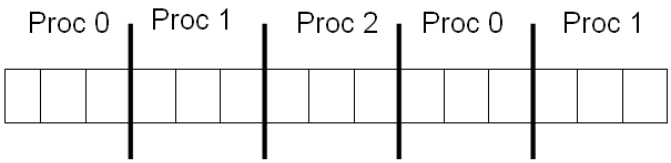

Figura 5.2: Exemplo de vizinhaça cíclica

No começo do código (`if`), tem-se o cálculo para saber se aquele processo está processando um bloco a mais. Esse bloco a mais só acontece caso o número de blocos (`N_BLOCS`) não seja divisível pela quantidade de processos (`size`). Obviamente, caso um processo tenha um bloco a mais então vão existir mais dois recebimentos, um para o vizinho da direita e um para o vizinho da esquerda. Se for divisível, para todos os processos envolvidos (processo 0 inclusive) esse `if` vai ser sempre falso. Mas, como dito anteriormente, pode acontecer de não existirem vizinhos na esquerda e/ou na direita, nesse caso seria um recebimento a menos. Os casos em que esse tipo de comportamento pode acontecer é quando o processo 0 é o responsável pelo primeiro bloco e quando o último processo responsÃ¡vel pelo Ãºltimo bloco. Segue abaixo o trecho de código que contempla essa lógica:

```
if( (rank == 0) || (rank == proc(N-1,size)) )
{
   if( (rank == 0) && (rank == proc(N-1,size)) )
   {
      number_of_irecvs--;
   }
   number_of_irecvs--;
}
```

Primeiro verifica se é o processo 0, caso isso seja verdade já deve fazer um recebimento a menos. Depois está verificando se o rank atual é o responsável pelo último bloco. A segunda verificação (`if` mais interno) é feita para o caso particular do processo 0 ser também o responsável pelo último bloco.

Tendo em mãos a quantidade de recebimentos a ser feita pode-se partir para a execução dos recebimentos assíncronos. Nesse ponto a técnica mencionada no começo dessa seção pode se fazer presente. Na programação paralela é difícil em um momento inicial contemplar todos os possíveis casos de *deadlock* que o seu programa poderia enfrentar.

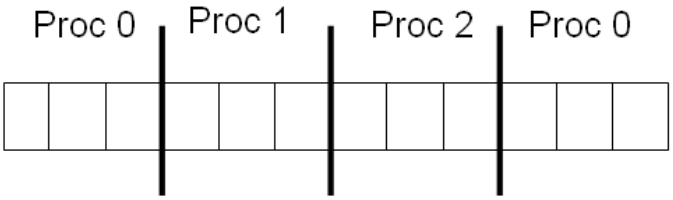

Figura 5.3: Exemplo de vizinhaça cíclica com zero sendo o útimo processo

Então uma técnica é utilizar uma escrita em arquivo no lugar do recebimento em si. Algo, como por exemplo, "PROCESSO 0 envia 'xyz' para PROCESSO 1". Então, claramente deve constar no arquivo impresso pelo processo 1 algo como "PROCESSO 1 espera por dados do PROCESSO 0", assim evita-se que o programa pare em pontos inesperados e evita, também que o programador tenha que percorrer o código copiando e colando escritas em arquivos com os famosos "Passei por aqui", "Passei por aqui 2", até descobrir onde foi que errou. Observe que essa técnica também irá fazer, em parte, o papel das escritas em arquivo menciondas, porém somente erros não ligados a comunicação entre os processos serão detectados. Obviamente essa técnica está longe de ser perfeita e exige uma análise de vários arquivos de saída, porém ela demonstrou-se bastante eficiente quando da troca da impressão para arquivo por um recebimento ou envio efetivo.

Cabe ressaltar também que essa técnica não está descrita em nenhum livro técnico-científico de programação paralela, é apenas uma técnica simples adotada pelo autor do presente trabalho que mostrou-se bastante eficiente no desenvolvimento do trabalho. E uma dica, que serve tanto para quem utiliza técnicas de "passou por aqui", quanto para quem utiliza a técnica do autor deste trabalho, é após os `fprintf` (escrita em arquivo) ou mesmo um `printf` (escrita na saída padrão) utilizar o procedimento `fflush` do C/C++. Esse procedimento força a escrita das palavras pretendidas nos métodos `fprintf` e `printf`. As vezes as escritas ficam *bufferizadas* (guardadas dentro de um buffer) e não são descarregadas na saída pretendida no momento que são executadas e isso pode prejudicar na depuração de um programa paralelo. Segue, então, o trecho de código responsável por disparar os recebimentos assíncronos (passo 2 do algoritmo). Logo abaixo do recebimento, tem-se um exemplo da utilização da técnica mencionada.

```
for(i=0;i<number_of_irecvs;i++)
{
   MPI_Irecv(&recvs[i], 1, recvs[i].udtMPI, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &mpi_req[i]);
   fprintf(out,"PROC#%i says: Waiting for any source "
                    +"to give me some data...\n",rank);
   fflush(out);
}
```

O próximo passo no algoritmo (passo 3) é o cálculo local junto com o envio do resultado para os vizinhos. A estrutura de dados utilizada no envio das informações foi a seguinte:

```
typedef struct __IsendStruct
{
   int sender_index;
```

```
   double sender_media;
   int tag;

   MPI_Datatype udtMPI;
   MPI_Aint addrs[3];
   int block_lens[3];
   MPI_Datatype oldTypes[3];

} IsendStruct;
```

Na estrutura, `sender_index` representa a identificação do processo que está enviando a informação, `sender_media` é o resultado do cálculo local e `tag` a identificação da mensagem que pode ser `TAG_RIGHT` e `TAG_LEFT` caso seja uma mensagem do vizinho da direita ou da esquerda, respectivamente. Na verdade, não há a necessidade das variáveis `sender_index` e `tag` da estrutura, poderia ser enviado diretamente o valor `sender_media`. Essas informações extras podem ser obtidas através do `MPI_Status` que está associado ao recebimento, como visto anteriormente. Mas foi utilizado essa abordagem para testar o envio de uma estrutura de dados, coisa que não pode ser feita de forma direta como

```
ISendStruct send;

send.sender_index = 3;
send.sender_media = 2.34443;
send.tag = TAG_LEFT;

MPI_Isend(&send,sizeof(IsendStruct),MPI_BYTE,2,TAG_LEFT,
                    MPI_COMM_WORLD,&mpi_req);
```

Esse tipo de construção não funcionaria pois uma estrutura de dados não é armazenada de forma contígua na memória. Assim a primitiva de comunicação `MPI_Isend` não consegue acessar os dados da estrutura para enviar de maneira correta. Logo, caso o Trent paralelo precisasse do envio de uma estrutra de dados mais complexa, seria um mapeamento quase direto do protótipo. Segue abaixo um trecho do protótipo que mostra como funciona o envio dessa estrutura:

```
for(i=0;i<N;i+=BLOC_SIZE)
{
  if( rank == proc(i,size) )
  {
     //..calculo local...

     if( i != 0 )
     {
        vizinho = (rank - 1 + size) % size;

        //..inicializacoes necessarias...
```

```
        MPI_Isend(p_bloc,1,p_bloc->udtMPI,vizinho,TAG_LEFT,
                    MPI_COMM_WORLD,&mpi_req[i_mpi_req++]);
        total_isend++;
     }
     if( (i + BLOC_SIZE) <= (N-1) )
     {
        vizinho = (rank + 1 + size) % size;

        //..inicializacoes necessarias..

        MPI_Isend(p_bloc,1,p_bloc->udtMPI,vizinho,TAG_RIGHT,
                     MPI_COMM_WORLD,&mpi_req[i_mpi_req++]);
        total_isend++;
     }
  }
}
```

5.2 Implementando a Versão paralela

Nessa seção será descrita a forma gradual que foi executada na construção da versão paralela do Trent, ilustrando as dificuldades e problemas encontrados durante esse desenvolvimento. Será discutido também o desempenho obtido por essa implementação, bem como os pontos de falha no programa, sugerindo-se melhorias para uma versão futura.

5.2.1 Considerações Iniciais

Antes de prosseguir com as considerações sobre a versão paralela é necessário uma pequena retomada da estrutura de dados principal do programa, a `basicNode`:

```
typedef struct _basicNode
{
   double h;
   double zb;
   double qm;

   struct _basicNode * pos[4];
   short qw[4];

   short ocup;

   double f1[4];

   int index;

} basicNode;
```

O vetor global do Trent é um vetor de $MAXCELLS$ elementos, onde $MAXCELLS$ é uma constante de valor 1.000.000 definida no código. Esse vetor, a saber

`basicNode cell[NCELLS]`, foi dividido de maneira inteira entre os processos, caracterizando, portanto, uma distribuição por blocos. Cada processo fica responsável por um pedaço do vetor e, caso essa divisão tenha resto, algum processo ficará com carga a menos de trabalho. Dada essa informação de distribuição e a estrutura acima, conclui-se que, tudo que se tratar de `cell[i]`, está sendo processada uma informação local ao processo, caso, obviamente, o elemento i pertença ao processo atual. Durante as otimizações do Trent foi definido um ponteiro para `cell[i]`, `celli` que será utilizado nos exemplos de códigos. Como pode ser observado na estrutura acima, `celli->pos` vai ser um ponteiro para um elemento do próprio vetor global. Logo, `celli->pos[RIGHT]` seria um ponteiro para o elemento $i + 1$. Com isso tem-se que `celli->f1[RIGHT]` por exemplo, será local ao processo mas `celli->pos[RIGHT]->f1[LEFT]` pode ser potencialmente pertencente a outro processo. Isso ficará mais claro a medida que exemplos forem dados mais adiante.

5.2.2 Inicialização dos Dados

Primeiramente nessa versão paralela, será tratado a inicialização dos dados. Como no protótipo aqui é importante que somente os dados referentes ao processo em questão é que devem ser inicializados, que é exatamente o papel do `if(rank == proc(k-1,size))` apresentado no trecho de código abaixo:

```
for(i=1;i<=YNODES;i++)
{
   for(j=1;j<=XNODES;j++)
   {
      k++;
      zb = 0.0;
      if( (j >= 1) && (j != XNODES) )
      {
         fscanf(file,"%lf ",&zb);
      }
      if( j == XNODES )
      {
         fscanf(file,"%lf \n",&zb);
      }

      if( rank == proc(k-1,size) )
      {
         cell[k].zb = zb;
      }

      (..incializacao dos vizinhos..)
}
```

Isso vai garantir que o processo tenha apenas informações que lhe dizem respeito. Se, eventualmente, o processo precisar de uma informação que pertence a um vizinho uma sincronização será requerida. Relembrando um pouco da estrutura do Trent seqüencial, a cada passo é executada uma iteração sobre *todos* os elementos do vetor global. Analisando essa iteração, observa-se que qualquer informação que o elemento i necessitar do elemento $i + 1$ a mesma *deve* ser *desatualizada*, uma vez que essa iteração ainda

não passou por esse elemento. Porém o sentido inverso não uma é identidade, ou seja, o elemento $i+1$, caso queira alguma informação do elemento i, tem que obter a informação mais atual. Em termos de programação seqüencial isso não é problema, uma vez que a iteração chegar no elemento $i+1$, terá, necessariamente, passado pelo elemento i. Esse mesmo comportamento é alterado um pouco na programação paralela. Definindo que i é o elemento mais a direita pertencente ao processo 0 e que $i+1$ é o elemento mais a esquerda do processo 1, como ilustra a figura 5.4, tem-se que, caso o processo 0 esteja processando o elemento i e necessite de alguma informação do elemento $i+1$, o mesmo deve pedir ao processo 1 por essas informações desatualizadas. Logo, aqui vê-se um claro local para utilização da sincronização assíncrona de processos. O processo 1, antes de iniciar qualquer cálculo, poderia imediatamente enviar ao processo 0 essas informações. No caso médio, espera-se que o processo 0 não fique esperando pela resposta do processo 1. Tomando como hipóse que o processo 0 começa seu processamento no elemento $i-4$ (figura 5.4) e que o processo 1 começa no elemento $i+1$, se o processo 1 enviar os dados no momento que estiver processando o elemento $i+1$, a probabilidade do processo 0 já obter os dados quando chegar no elemento i é bem alta.

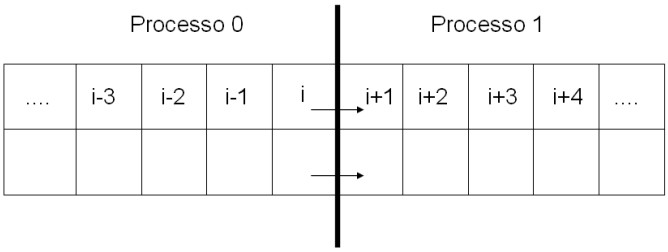

Figura 5.4: Exemplo de interação entre os elementos da fronteira

5.2.3 Indentificação e Implementação dos Sincronismos

Isso colocado, observando-se a implementação do Trent, tem-se que os elementos `zb` e `h`, presentes na estrutura de dados `basicNode`, do elemento do vetor global $i+1$, são requeridos pelo elemento i durante os cálculos, como sugere o código abaixo

```
if ( celli->qw[RIGHT] != 0)
{
   temp = celli->pos[RIGHT];

   celli_f1[RIGHT] =
       flow_manning(celli->h, temp->h,
                    celli->zb, temp->zb, dt);

   temp->f1[LEFT] = - celli_f1[RIGHT];
}
```

Ou seja, caso `celli` seja um elemento de fronteira (como o elemento i da figura 5.4), será necessário uma sincronização dos elementos `zb` e `h` vindos do vizinho da direita (`celli->pos[RIGHT]`). O trecho de código abaixo mostra como foi feito o cálculo para saber se deve ser feito um envio (nota-se que somente os elementos da fronteira é que precisarão trocar informações).

```
if( ((i-1)%XNODES) % BLOC_SIZE == 0 )
{
   if( index_left != 0 )
   {
      if( rank != proc(index_left-1,size) )
      {
         send_buffer[0] = celli->h;
         send_buffer[1] = celli->zb;

         MPI_Isend(&send_buffer[0],2,MPI_DOUBLE,
                   left_neighbor,index_left,MPI_COMM_WORLD,
                   &mpi_send_req[i_send_req++]);
      }
   }
}
```

O primeiro `if`, controla se é um elemento de borda ($i + 1$, no caso da figura 5.4), o segundo se realmente possui um elemento a esquerda (o elemento $i = 0$ não possui elementos a esquerda) e o terceiro verifica se o índice do elemento da esquerda em questão não pertence ao processo atual. Se pertence, não é necessário nenhum envio. O recebimento atua de maneira recíproca, porém testando se é um elemento mais a direita (i, no caso da figura 5.4) e o código segue abaixo.

```
if( ( ((i-1)%XNODES) % BLOC_SIZE) == (BLOC_SIZE-1) )
{
   if( index_right != 0 )
   {
      if( rank != proc(index_right-1,size) )
      {
          MPI_Irecv(&recv_buffer[0],2,MPI_DOUBLE,
                    right_neighbor,i,MPI_COMM_WORLD,
                    &mpi_recv_req);
          irecvs++;
      }
   }
}
```

Numa primeira tentativa de execução do Trent, somente com essas alterações, rodando 5 processos com 12 blocos para cada um, gerou a situação de espera indefinida. Algum processo havia ficado esperando indefinidamente por uma resposta vinda de um vizinho, que não respondeu porque, ou ficou esperando uma resposta também (*deadlock*) ou já havia terminado o seu processamento. Fazendo uso de análises do arquivo de informações gerado utilizando-se da técnica mencionada na seção do protótipo foi detectado que todos os processos terminaram o processamento e somente o processo 0 não. Aqui observa-se que, claramente, o critério de parada está divergindo entre os processos, logo essa informação também deverá ser sincronizada. O critério de parada é calculado a cada novo elemento do `celli`. Ou seja, qualquer processo pode atualizar esse valor. Vale salientar que as iterações sobre os outros elementos seguem, o critério de parada é o produto final dessa iteração, controlando apenas se deve continuar com os passos. Logo,

raciocinando sempre em cima do comportamento do programa seqüencial, chegou-se a seguinte questão: Supondo que eventualmente esse valor fosse atualizado no elemento $i+5$ e depois no elemento $i+20$, qual valor será utilizado? Obviamente o valor de $i+20$, uma vez que o produto final desse laço de iterações é o critério de parada. Para ficar mais claro, de forma reduzida, o critério de parada (`dtnew`), no programa seqüencial é calculado da seguinte forma:

```
double flow2d( ..parametros da funcao.... )
{
   (...calculos locais....)

   mult_factor = DL * cfl * 0.5 / sqrt(GRAVITY);
   for (i=1;i<=NCELLS;i++)
   {
      celli = &cell[i];
      celli_f1 = celli->f1;

      if (  celli->qw[BOTTOM] + celli->qw[RIGHT] != 0 ||
         (fabs(celli_f1[LEFT]) + fabs( celli_f1[TOP])) > 0.)
      {

         (.....varios calculos locais.....)

         if (celli->h > DSHORE)
         {
            temp1 = sqrt(celli->h);
            inct = mult_factor / temp1;
            if (inct < dtnew)
               dtnew = inct;
          }
      }
   }
   return dtnew;
}
```

Esse valor retornado é usado em um outro cálculo mais complexo que será omitido aqui, mas que usa apenas dados locais de outros vetores e que todos os processos possuem uma cópia idêntica. Voltando a resolução do problema, basta sincronizar entre todos os processos (*broadcast*) o `dtnew` atualizado pelo maior índice. Para solucionar esse problema foi implementado o seguinte, quando um processo termina seus cálculos locais ele espera por todos os outros terminarem também, ou seja, ele encontra uma barreira que o impede de prosseguir, daí o nome da função do MPI `MPI_Barrier`. Quando todos atingirem essa barreira, todos enviam para todos o índice que foi guardado durante a atualização[2] do `dtnew`. Então, de posse de todos os índices os processos irão buscar o índice de maior valor, de posse dessa informação fica fácil saber quem deve ser o processo que irá distribuir o valor de `dtnew` atualizado (também chamado de elemento *root* do *broadcast*). Por partes, primeiro o trecho de código responsável por espalhar as informações dos índices:

[2]Caso não seja atualizado o *dtnew*, -1 é enviado

```
MPI_Barrier(MPI_COMM_WORLD);

indexOfUpdatedDtnew = (int *)malloc(size * sizeof(int));

MPI_Allgather(&dtNewStruct.index,1,MPI_INT,
              indexOfUpdatedDtnew,1,MPI_INT,
              MPI_COMM_WORLD);
```

Aqui o vetor `indexOfUpdateDtnew`, que tem a dimensão da quantidade de processos envolvidos (`size`), será preenchido com todos os índices guardados por todos os processadores (`dtNewStruct.index`). Percorrendo o vetor facilmente encontra-se o maior índice, encontrado o maior índice, utiliza-se a fórmula herdada do protótipo para calcular qual o processador responsável por um determinado índice do vetor global. Esse processo (`bcast_root` no próximo trecho de código) será o processo que espalhará essa informação entre os demais através de um `MPI_Bcast` como abaixo:

```
bcast_root = proc(tempIndex-1,size);

MPI_Bcast(&dtStruct.dtnew,1,MPI_DOUBLE,bcast_root,
          MPI_COMM_WORLD);
```

Ou seja, se `bcast_root` for o identificador do processo atual, o processo atual é o “root” e todos os outros serão os receptores. Como esperado, todos os processos coordernaram-se e o programa executou normalmente após essas alterações. Porém o resultado final do processamento, analisando os diversos arquivos de saída do Trent, não foi correto. Mais uma busca no código por potenciais problemas de sincronização levou a mais uma pecularied ade do programa. Notou-se no código do Trent a utilização do elemento `celli->pos[LEFT]` em alguns trechos, a saber

```
if (celli->h > DTOL)
{
   celli->qw[BOTTOM] = 1;
   celli->qw[RIGHT] = 1;
   celli->pos[TOP]->qw[BOTTOM] = 1;
   celli->pos[LEFT]->qw[RIGHT] = 1;
   vol2d+=celli->h*DLxDL;
}
```

Observe que os seguinte elementos são locais:

```
celli->qw[BOTTOM]
celli->qw[RIGHT]
celli->pos[TOP]->qw[BOTTOM]
```

Como a divisão é por colunas todos os elementos

```
celli->pos[TOP]
celli->pos[BOTTOM]
```

são locais ao processo. Seguindo a definição anterior feita para a figura 5.4, e agora supondo que `celli` seja um ponteiro para o elemento $i+1$ da figura e que `pos[LEFT]` de `celli` seja um ponteiro para o elemento i em questão. Ou seja, o que se prentende é atualizar uma informacão do vizinho a esquerda do elemento $i+1$, que no caso não pertence ao processo 1. Raciocinando-se em termos do comportamento da versão seqüencial, o que significa um elemento $i+1$ atualizar uma informação do elemento i? Significa que essa informação será útil efetivamente somente no próximo passo. Então, cada processo que encontra-se nessa mesma situação, guardará um vetor coluna de valores `qw[RIGHT]` do elemento `celli->pos[LEFT]` que será enviado no final de cada passo de execução. Exemplo de código para armazenar essa informação:

```
if( ( ((i-1)%XNODES) % BLOC_SIZE) == 0 )
{
   qwBuffer[(i-(rank*BLOC_SIZE))/XNODES] = 1;
}
else
{
   celli->pos[LEFT]->qw[RIGHT] = 1;
}
```

O cálculo do índice do `qwBuffer` é feito em função da posição relativa na coluna da fronteira. A figura 5.5 ilustra essa correspondência. Logo, antes de começar os cálculos, utilizando-se da comunicação assíncrona, dispara-se um recebimento desse vetor coluna:

```
vec_qw = (int *)malloc(YNODES*sizeof(int));

if( rank != (size-1) )
{
   MPI_Irecv(&vec_qw[0],YNODES,MPI_INT,
             right_neighbor,TAG_QW,
             MPI_COMM_WORLD,&req_qw_rec);
}
```

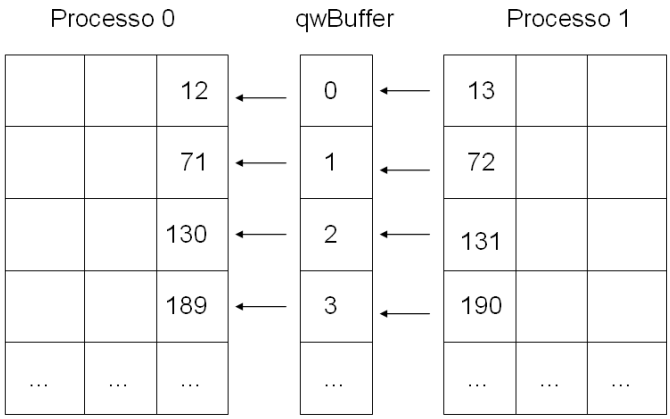

Figura 5.5: Exemplo de correspondencia entre as linhas da borda e o vetor coluna *qwBuffer*

Naturalmente, o tamanho do vetor (`vec_qw`) tem a dimensão representada pela quantidade de linhas (`YNODES`) que a matriz possui. Sempre relembrando que essa matriz está mapeada em um vetor. Terminado o processamento do passo N, antes de seguir para o passo $N + 1$

```
if( rank != 0 )
{
   MPI_Isend(&qwBuffer[0],YNODES,MPI_INT,
             left_neighbor,TAG_QW,
             MPI_COMM_WORLD,&req_qw_send);
}

if( rank != (size-1) )
{
   MPI_Wait(&req_qw_rec,&wait_qw);
}

updateQW(cell);
```

Lembrando também que esta é uma distribuição por blocos e não cíclica, o primeiro `if` existe porque o processo 0 não possue vizinhos a esquerda e o segundo existe para evitar que o processo de id `size-1`, ou seja, o último processo a processar o último bloco, não fique esperando o resultado vir do vizinho da direita, pois o mesmo não possui um.

Feito isso o funcionamento do Trent paralelo, para poucos passos, estava em conformidade, apesar de ter um desempenho um pouco baixo. O desempenho baixo será explicado com maiores detalhes na próxima seção. Porém, quando o número de passos era aumentado, esse desempenho piorava ainda mais, mas numa escala bem superior se comparado com o aumento do número de laços. E com o agravante que para pouco passos o resultado estava correto, mas para mais passos não. Uma análise mais profunda nos arquivos de saída (no estilo, "Passei por aqui"), mostrou que somente o processo 0 estava efetivamente trabalhando. Relembrando o laço da função `flow2d` citada a momentos atrás, logo depois das primeiras linhas do laço existe um teste que se extende até o fim do laço, ou seja, quem não entrar nesse primeiro teste só terá feito envios e recebimentos assíncronos e não terá participado de forma efetiva em nenhum cálculo. Esse ponto foi um ponto crítico no trabalho e será discutido também mais adiante. Como dito anteriormente, tudo que buscado de `celli->pos[RIGHT]` poderia ser com o valor desatualizado, porém, mesmo o leitor mais atento teria reparado que nos primeiros trechos de código mostrados após a inicialização dos dados, existia um pequeno, mas crucial, detalhe. Observando-se novamente o trecho, nota-se que na última linha tem-se algo do tipo `temp->f1[LEFT] = - celli_f1[RIGHT];`, onde `temp` é um ponteiro para `celli->pos[RIGHT]`. Voltando novamente a figura 5.4, e tomando as mesmas definições inicialmente feitas, o que se está fazendo é que o processo 0, quando chega no elemento i, e o mesmo eventualmente entre nos testes referidos, ele irá atualizar valores de $i+1$. Até o momento isso não necessariamente seria um problema, visto que, caso esse valor não fosse utilizado no momento pelo processo 1 tudo estaria acontecendo de forma coerente. Porém, retomando o teste feito no início do laço, tem-se o seguinte cálculo: `(fabs(celli_f1[LEFT]) + fabs( celli_f1[TOP])) > 0.)`. Se bem observado nesse trecho de código, `celli_f1[LEFT]`, é na verdade um ponteiro para

`celli->f1[LEFT]` e aqui está o problema. Retomando novamente o raciocínio seqüencial, fazer essa conta `temp->f1[LEFT] = - celli_f1[RIGHT];`, significa atualizar o elemento $i + 1$ e, quando for a próxima iteração do laço, o elemento `f1[LEFT]` de $i + 1$ estará atualizado.

Mapeando isso para o programa paralelo, significa que o processamento não pode continuar até que se tenha o valor de `f1[LEFT]` atualizado. O que foi feito, então, foi disparar um recebimento assíncrono, porém com uma espera logo em seguida. O trecho de código foi inserido juntamente com o trecho responsável por enviar os valores de `zb` e `h` para o vizinho da esquerda, uma vez que os testes a serem feitos antes do recebimento são os mesmos. Segue abaixo o trecho:

```
MPI_Irecv(&f1Left_Irecv[f1_index_Irecv],1,MPI_DOUBLE,
          left_neighbor,i+TAG_F1_LEFT,MPI_COMM_WORLD,
          &wait_for_f1);

MPI_Wait(&wait_for_f1,&wait_status_for_f1);
```

No caso foi utilizado um *buffer* de recebimento, porém não haveria essa necessidade uma vez que é somente um elemento que vai chegar por vez que o processo entrar nesse recebimento. O *buffer* aqui utilizado não apareceu apenas por aparecer. Essa é uma prática comum para envios e recebimentos assíncronos. Por ser assíncrono, esse envio/recebimento pode ocorrer a qualquer momento, sem que o programador saiba (caso ele não faça um *wait*) e o dado a ser enviado não pode ser modificado antes da transmissão, podendo gerar desde erros do tipo "falha de segmentação" (*segmentation fault*) até inconsistências de dados chegando na outra ponta. Mas no caso acima realmente não faria diferença porque a informação recebida é consumida logo em seguida e tem um *wait* que fará a espera do dado. Em termos de desempenho alterar para uma única variável não acarretaria em grandes mudanças, mas em termos de economia de armazenamento na memória faria muita diferença. Após esses problemas, o Trent paralelo apresentou um funcionamento coerente como era esperado. Terminada a implementação, chega-se ao momento de executar em cluster de computadores para saber o desempenho real da aplicação. Em geral, o programa paralelo é desenvolvido em uma máquina menos poderosa que um cluster, monoprocessada muitas vezes (workstations), apenas para testes.

5.2.4 Análise dos Resultados

As próximas seções discurão os efeitos da paralelização no tempo de execução do trent. Também será feita uma análise dos principais problemas encontrados nessa paralelização e será mostrado sugestões para otimização do programa.

5.2.4.1 Desempenho do programa paralelo

Aqui será apresentado os dados referentes a execução do programa Trent paralelo no cluster de computadores da UFRGS. Antes da apresentação dos resultados, faz-se necessário uma rápida introdução a análise de desempenho de um programa paralelo. A análise do desempenho de um programa seqüencial é trivial, basta observar o tempo que o mesmo levou para realizar uma determinada tarefa. Porém na programação paralela isso não acontece exatamente dessa maneira, existem outros fatores para se observar. Por exemplo, considerando que um programa paralelo que resolvia um problema A de complexidade x demorou 30 segundos para resolver. Considerando um outro programa paralelo que resolvia um problema B de complexidade x também, e demorou 40 segundos.

Qual dos dois foi mais eficiente na resolução? Se conderar que, um programa seqüencial bem feito, demora 37 segundos para resolver o mesmo problema A e que um outro programa seqüencial que implementa o melhor algoritmo para resolver o problema B demora 70 segundos o programa paralelo que resolve o problema B foi bem mais eficiente. Diz-se que o programa paralelo teve um melhor *SpeedUp* em relação ao programa seqüencial. A forma de cálculo do SpeedUp é

$$S_p(n) = T^*(n)/T_p(n) \tag{5.1}$$

onde, $T^*(n)$ é o tempo do programa sequencial e $T_p(n)$ é o tempo de execução em p processos. A figura 5.10 mostra o *speedup* obtido no Trent paralelo. No eixo x tem-se o *speedup* e no eixo y a quantidade de processos. Em geral, espera-se que $S_p(n) \leq p$, porém o que se procura em um programa paralelo é $S_p \approx p$ e o que se observa na realidade com o Trent é que houve um *Speeddown*. A tabela 5.2 mostra a média dos tempos de execução do Trent paralelo em diversas configurações de número de processos. Retomando resultados apresentados anteriormente, para critério de comparação, o Trent seqüencial original executa em um tempo médio de 30,4927 segundos, enquanto que o Trent seqüencial otimizado executa em um tempo médio de 14,3262. Observa-se que o desempenho obtido por apenas 1 processo foi dentro do esperado devido as cópias de buffers desnecessárias que acabam ocorrendo e aos controles extras adicionados na versão paralela. A explicação para números tão baixos de desempenho da versão paralela foi devido às sincronizações do Trent.

Como foi apresentado nesse capítulo, existiam diversos pontos de sincronização no programa. Esses pontos de sincronização tornaram-se um gargalo na execução do programa acarretando em grande perda de desempenho. Em um primeiro momento pensou-se que a distruibuição por colunas fosse um problema, porém analisando a forma com que a informação segue seu fluxo dentro do programa, percebeu-se que invariavelmente, ou seja, não importando que tipo de distribuição fosse escolhida, por linhas ou por colunas, devido as caracteristicas da entrada, o programa age como observado nas figura 5.6, 5.7 e 5.8. Na figura 5.6 ocorreu uma ativação qualquer na célula i. Essa ativação desencadeia um conjunto de reações nas células vizinhas (fluxo do alagamento) como mostra a figura 5.7. Exatamente nesse ponto, as células da fronteira estão prestes a ativar suas vizinhas. Enquanto isso, os vizinhos estão parados, uma vez que os mesmos precisam da informação do `f1[LEFT]` vinda do vizinho a esquerda. Isso desencadeia uma serie de *waits*, o que implica, em um começo quase seqüencial. Só então, após essa sincronização do `f1[LEFT]`, ocorre a ativação da figura 5.8, e o próximo processo começa a trabalhar de forma mais efetiva (não só repassando dados e fazendo testes). Esse é fluxo de informação supondo uma distribuição por colunas, como a do trabalho de conclusão em questão. Observa-se que esse mesmo fato irá ocorrer caso seja feita uma distribuição por linhas como ilustra a figura 5.9. Alí tem-se o que seria o passo $n + 1$ (*step* $n + 1$) de uma distribuição em colunas.

5.2.4.2 Otimização do programa paralelo

Esse tipo de comportamento gera um desbalanceamento de carga, o que indica que uma opção para melhorar o desempenho desse algoritmo seja utilizar o esquema de particionamento *bloco-cíclico*. Outro ponto de otimização está na forma de atualizar os dados da fronteira. Como ilustrou esse capítulo e as figuras vistas aqui, as informações das fronteiras precisam ser trocadas. Essas informações são essenciais para o correto funcionamento do algoritmo. As sincronizações presentes acabaram por transformar a execução

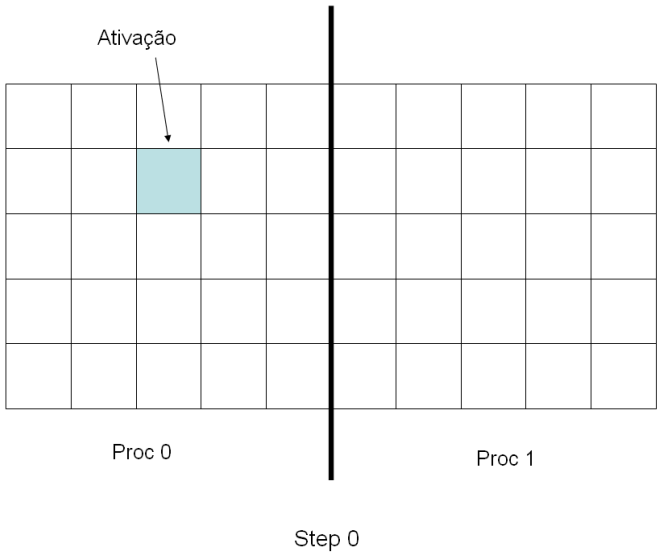

Figura 5.6: Fluxo dos dados no Trent - Step 0

Tabela 5.2: Tempo de execução do Trent

Número de processos	Tempo (seg)	Desvio padrão
1	35,5678	1,0220
3	155,4458	1,0074
5	144,5243	1,0021
8	157,7864	1,0042
10	166,8455	1,0344

em um esquema de pipeline. As figuras 5.11, 5.12, 5.13 e 5.14 ilustram essa característica. Apesar de parecer uma execução seqüencial devido às várias dependências encontradas, existem atividades ocorrendo em paralelo. Percebe-se também, como no exemplo dado, que assim que o processo termina seu trabalho, o mesmo fica esperando pela conclusão dos demais para que todas as fronteiras sejam sincronizadas e que todos juntos comecem o próximo passo. Essa espera poderia ser melhorada, trabalhando com um processo de ativação de uma maneira diferente como ilustrado nas figuras 5.15 e 5.16. Assim o processo 0 não precisaria ficar parado esperando pelo término dos outros processos e poderia prosseguir para os próximos passos. Isso fará com que o processo 0 entre no passo mais adiantado, adiantando também os resultados das fronteiras a serem enviados. Assim, quando o processo 1 entrar no mesmo passo do processo 0 e tiver que esperar pelo `f1[LEFT]`, o mesmo já terá sido enviado, o que tornaria bem mais eficiente o programa paralelo. Porém deve-se levar em conta a variável que controla o critério de parada do algoritmo.

Outra idéia para melhorar o desempenho é utilizar uma malha bi-dimensional, ou seja, dividir o trabalho para os processadores em sub-matrizes menores, o que reduziria o número de comunicações, mas aqui de novo poderia acontecer um desbalancemento na carga computacional. Novamente o bloco cíclico faz-se interessante, pois todos os processadores poderiam compartilhar o tempo que não fazem nada também. Ou seja, primeiro o processo 0 trabalha e o processo 1 “descansa”, depois o processo 1 trabalha e

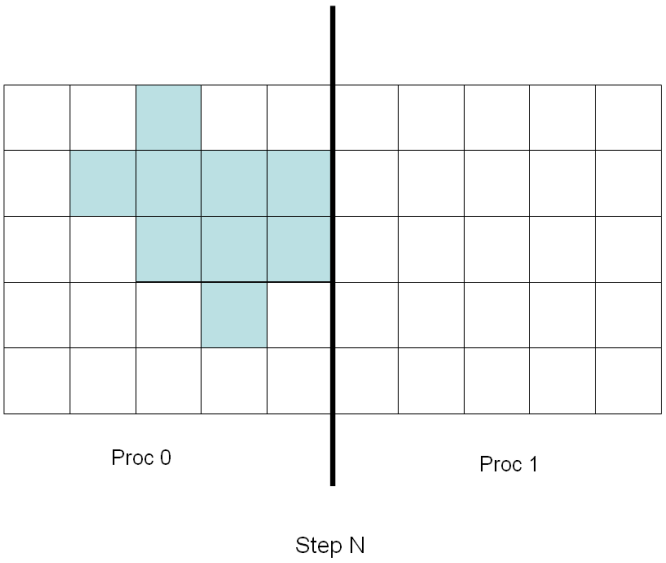

Figura 5.7: Fluxo dos dados no Trent - Step N

é a vez do processo 0 “descansa”.

Com isso conclui-se a implementação do Trent paralelo mostrando todos os problemas encontrados e os resultados obtidos dessa primeira versão paralela. Problemas como identificação de pontos de sincronização e desbalanceamento de carga foram discutidos. Bem como a implicação dessas sincronizações no tempo de execução do programa. Uma conclusão para fechar o trabalho e propostas de trabalhos futuros virá no próximo capítulo.

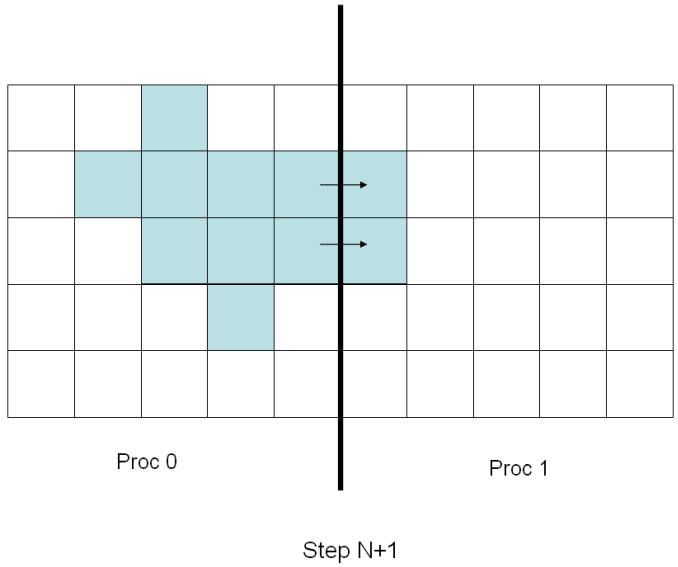

Figura 5.8: Fluxo dos dados no Trent - Step N+1

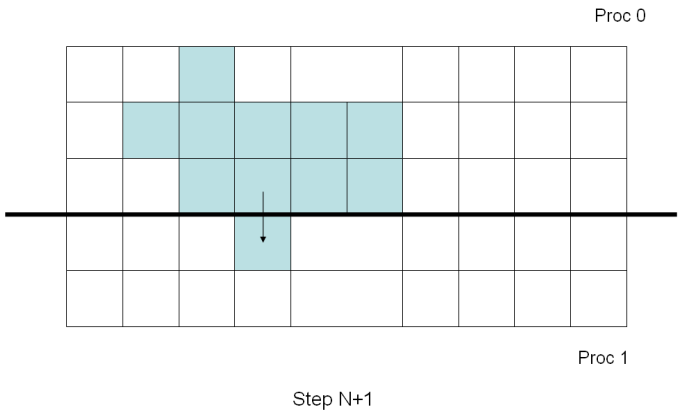

Figura 5.9: Fluxo dos dados no Trent - Exemplo caso distribuição por linhas

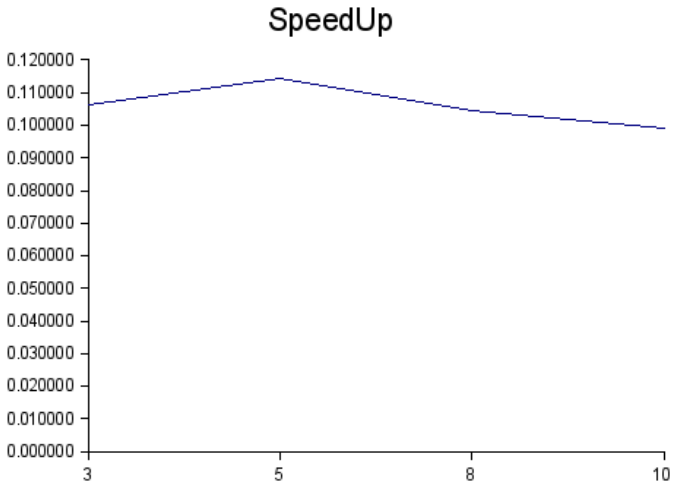

Figura 5.10: Gráfico do Speedup do Trent paralelo

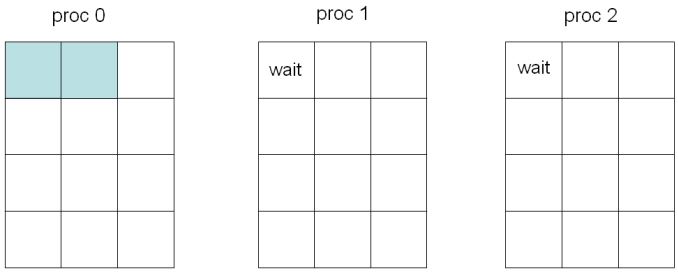

Figura 5.11: Fluxo de informações na forma de pipeline - passo 1

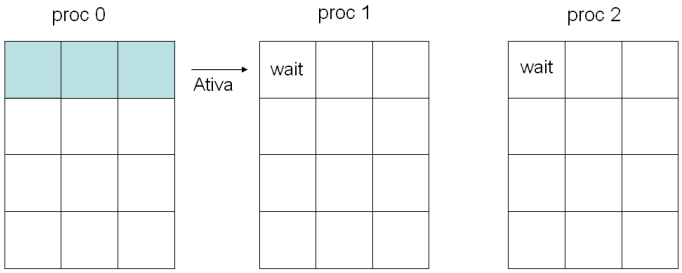

Figura 5.12: Fluxo de informações na forma de pipeline - passo 2

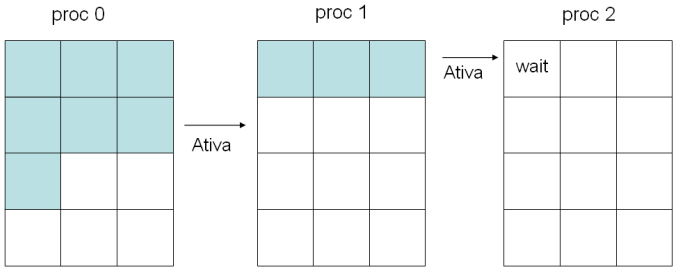

Figura 5.13: Fluxo de informações na forma de pipeline - passo 3

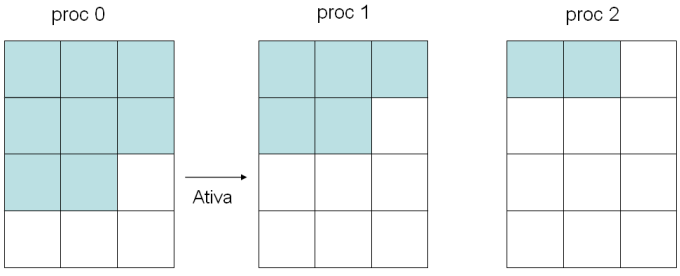

Figura 5.14: Fluxo de informações na forma de pipeline - passo 4

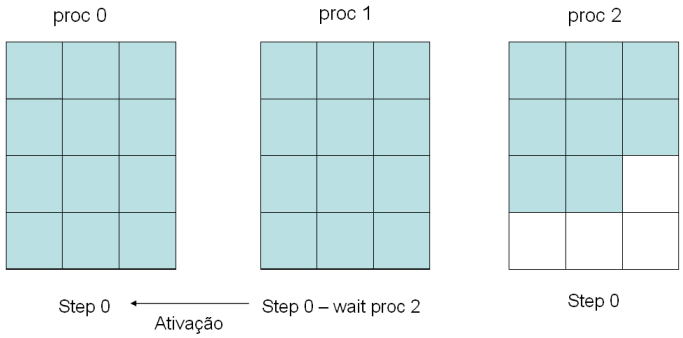

Figura 5.15: Fluxo de informações na forma de pipeline melhorado - passo 1

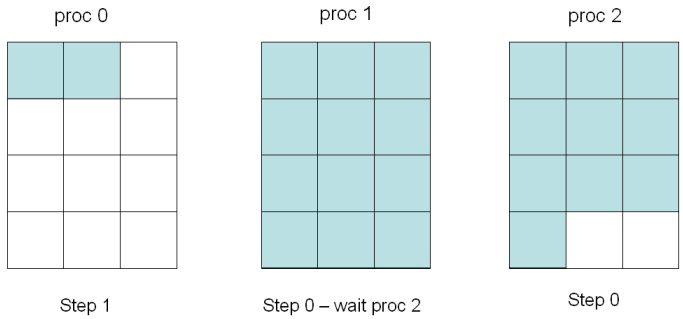

Figura 5.16: Fluxo de informações na forma de pipeline melhorado - passo 2

6 CONCLUSÃO

Este trabalho mostrou que a construção de um programa paralalelo a partir de um seqüencial está longe de ser trivial e é uma área muito desafiadora. Num primeiro momento importantes técnicas de otimização foram mostradas o que acarretou em uma excelente redução no tempo de execução em relação ao Trent original. Mostrou também que o uso de ferramentas de analise de execução de programas (como o *gprof*) são importantes para uma análise de pontos em que essa otimização deve atacar. Observou-se também, que um código legível pode ajudar a identificar padrões no código de maneira a identificar mais rápido trechos que possam ser otimizados.

Foi observado também que nem sempre é possível obter o melhor desempenho no desenvolvimento de um programa paralelo. Demonstrou também que o desempenho depende também da forma com que o programa tem o fluxo de dados na sua execução, e que o programador tem o desafio de achar os pontos de sincronismo e ao fazê-lo, procurar a melhor forma de lidar com o balanceamento de carga entre os processos. Esse estudo da programação paralela apresentou também algumas importantes técnicas de distribuição de dados e de, uma certa forma, depuração de um programa paralelo. Esse estudo da programação paralela mostrou que, mesmo o MPI sendo um padrão ele exige muito conhecimento técnico, o que demanda um certo tempo para um completo domínio do mesmo.

Deve ser salientado também que a troca de mensagens pode não ter sido a melhor estratégia para se criar o programa paralelo. Para uma afirmação mais concreta é necessário mais testes comparativos com essa versão e outras que serão sugeridas como trabalhos futuros. Com este trabalho tem-se a noção necessária das dependências entre os dados, além do conhecimento do fluxo dos dados. Apesar de mais raras, soluções para computadores vetoriais poderiam ser uma alternativa de melhor desempenho nesse caso, bem como a programação com *threads* (memória compartilhada).

Inicialmente havia sido proposto fazer um estudo comparativo entre a versão MPI aqui apresentada com uma versão em PVM que estava sendo desenvolvida na Inglaterra. Porém houve pouca interação entre os grupos envolvidos. Uma outra idéia inicial era mostrar os resultados na forma gráfica utilizando um programa especialmente desenvolvido para essa funcionalidade na Universidade de Nottingham.

Durante o desenvolvimento deste trabalho foi utilizado todo conhecimento adquirido ao longo das disciplinas cursadas no curso de Ciência da Computação. Na parte de otimização, por exemplo, foi utilizada a experiência adiquirida durante a disciplina de Compiladores. A reestruturação de código utilizou as técnicas de Engenharia de Software. A disciplina de Programação Distribuída e Paralela também foi utilizada ao longo do capítulo 5. A experiência anterior do presente autor com trabalhos de Iniciação Científica envolvendo MPI e uma biblioteca matemática de alta exatidão, como bem explicado em

(**?**) e (**?**).

Como trabalhos futuros pretende-se atualizar o Trent paralelo para que possa suportar também a idéia da decomposição por bloco cíclico. Diante dessa implementação fazer um estudo comparativo em termos de tempo de execução entre as diversas formas de distribuição de dados. Aplicar a melhoria sugerida no capítulo 5, de uma forma que os processos não fiquem mais parados entre um passo do algoritmo e outro. Implementar uma versão bloco-cíclica utilizando uma distribuição por linhas e comparar com a implementação com divisão por colunas.

Outra sugestão de trabalho futuro, é a implementação de uma distribuição de submatrizes, ao invés de vetores linhas ou colunas e fazer uma comparação entre todas essas ténicas em busca daquela que mais seja adequada ao caso do Trent.