Universidade Estadual de Londrina

LUIZ GUILHERME CASTILHO MARTINS

DESENVOLVIMENTO DE TÉCNICAS DE PARALELIZAÇÃO DE CÓDIGO

LONDRINA–PR

2013

LUIZ GUILHERME CASTILHO MARTINS

DESENVOLVIMENTO DE TÉCNICAS DE PARALELIZAÇÃO DE CÓDIGO

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Wesley Attrot

LONDRINA–PR

2013

Luiz Guilherme Castilho Martins
Desenvolvimento de Técnicas de Paralelização de Código/ Luiz Guilherme Castilho Martins. – Londrina–PR, 2013-
29 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Wesley Attrot

– Universidade Estadual de Londrina, 2013.

1. Palavra-chave1. 2. Palavra-chave2. I. Orientador. IIUniversidade xxx. IIIFaculdade de xxx. IVTítulo

CDU 02:141:005.7

ERRATA

Elemento opcional da

FERRIGNO, C. R. A. **Tratamento de neoplasias ósseas apendiculares com reimplantação de enxerto ósseo autólogo autoclavado associado ao plasma rico em plaquetas**: estudo crítico na cirurgia de preservação de membro em cães. 2011. 128 f. Tese (Livre-Docência) - Faculdade de Medicina Veterinária e Zootecnia, Universidade de São Paulo, São Paulo, 2011.

Folha	**Linha**	**Onde se lê**	**Leia-se**
1	10	auto-conclavo	autoconclavo

LUIZ GUILHERME CASTILHO MARTINS

DESENVOLVIMENTO DE TÉCNICAS DE PARALELIZAÇÃO DE CÓDIGO

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

Prof. Dr. Wesley Attrot
Universidade Estadual de Londrina
Orientador

Prof. Dr. Segundo Membro da Banca
Universidade Estadual de Londrina

Prof. Msc. Terceiro Membro da Banca
Universidade Estadual de Londrina

Londrina–PR, 24 de novembrode 2013

LONDRINA–PR

2013

A todos aqueles que
me apoiaram e contribuíram
para conclusão deste trabalho.

AGRADECIMENTOS

...

"O importante é ganhar. Tudo e sempre.
Essa história que o importante é competir não passa de demagogia.
(Ayrton Senna)

MARTINS, L. G. C.. **Desenvolvimento de Técnicas de Paralelização de Código**. 29 p. Trabalho de Conclusão de Curso (Graduação). Bacharelado em Ciência da Computação – Universidade Estadual de Londrina, 2013.

RESUMO

Paralelização de código permite ao programador a oportunidade de criar algoritmos para resolver problemas com maior eficiência. Programas são ditos paralelos quando existem duas ou mais ações executando simultaneamente em diferentes unidades de processamento. O objetivo deste trabalho é estudar técnicas de paralelização de código e tentar desenvolver uma nova otimização ou técnica de paralelização.

Palavras-chave: palavra chave1. palavra chave2.

MARTINS, L. G. C.. **Designing Techiniques for Code Paralellization**. 29 p. Final Project (Undergraduation). Bachelor of Science in Computer Science – State University of Londrina, 2013.

ABSTRACT

Code parallelization allows to the developer an oportunity to create algorithms to solve problems with more efficiency. A program is said to be parallel if it can have two or more action executing simultaneously in differents processing units. The goal of this work is studying parallelization techniques and try to develop a tecnique or optimization applied to code parallelization.

Keywords: word 1. key word2.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CPU	Central Processing Unit
DFG	Data Flow Graph
IW	Instruction Word
MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data
NASA	National Aeronautics and Space Administration
NOWs	Network of Workstations
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
VLIW	Very Long Instruction Word

LISTA DE SÍMBOLOS

Γ	Letra grega Gama
Λ	Lambda
ζ	Letra grega minúscula zeta
$\in$	Pertence

SUMÁRIO

INTRODUÇÃO

Diante da dificuldade de se melhorar processadores *single-core* devido ao alto consumo de energia e as altas temperaturas, a indústria de processadores adotou como solução à esses problemas, o desenvolvimento de processadores *multi-core*, além disso também considerou-se o poder de processamento quando se utilizam vários processadores *single-core* trabalhando simultaneamente, os quais oferecem grande desempenho, maior eficiência energética e menor custo. Porém as aplicações desenvolvidas para a arquitetura *single-core* não utilizam-se do poder computacional dos processadores *multi-core*. Para que as aplicações possam usufruir do alto desempenho que estes processadores oferecem, as aplicações devem ser divididas em múltiplas partes para que possam ser executadas em paralelo.

A paralelização de código permite ao programador resolver problemas com maior eficiência, porém projetar e codificar um programa paralelo continua sendo uma tarefa difícil, uma vez que além de dividi-lo em pequenas tarefas, deve-se considerar a concorrência entre elas, as dependências de dados, entre outros fatores que dificultam o trabalho de paralelização

1 TAXONOMIA DE FLYNN

Taxonomia de Flynn, proposta em 1966 [1] por Michael Flynn e expandida em 1972 [2], é uma das formas de classificar o paralelismo disponível no processador.

Taxonomia de Flynn utiliza o conceito de sequência de objetos ou ações, que são chamados de *stream*. Flynn introduziu dois tipos de *stream*, o *stream* de instrução e também o *stream* de dados.

O *stream* de instrução consiste em uma sequência de instruções. Uma instrução ou *instruction word (IW)* é uma cadeia de 0's e 1's que representa a menor operação visível ao programador e que será executada pelo processador. Uma instrução pode conter uma ou mais operações, devido a esta peculiaridade, alguns autores utilizam *instruction* para instruções que contenham apenas uma operação e *instruction word* para instruções que contenham mais de uma operação.

Existem no entanto quatro combinações de *streams* que descrevem as arquiteturas de computadores mais comuns [3]:

1. **SISD:** *Single Instruction, Single Data*
2. **SIMD:** *Single Instruction, Multiple Data*
3. **MISD:** *Multiple Instruction, Single Data*
4. **MIMD:** *Multiple Instruction, Multiple Data*

Cada combinação de *streams* caracteriza uma classe de arquitetura e cada uma destas classes possui seus tipos de paralelismo específicos.

1.1 SISD: SINGLE INSTRUCTION, SINGLE DATA

A classe de arquiteturas de processadores *SISD*, inclui a maior parte dos processadores ainda em uso em 2013, os processadores *single-core*, embora os programadores não percebam o paralelismo inerente destes, muita concorrência pode estar presente.

Em 1966 Flynn cita o *pipeline* como uma forma de se obter concorrência nos processadores *SISD*, embora Flynn considere a decodificação das inúmeras *instructions* como sendo um *bottleneck*, devido a tecnologia da época. Em 2013, grande parte dos dos processadores utilizam-se de *pipeline* assim como também se aproveitam de alguma forma de múltiplas *instructions*.

A concorrência em processadores *SISD* são explorados em tempo de execução, realizando mais de uma operação por ciclo de *clock* da máquina.

A quantidade e o tipo de paralelismo possível em processadores *SISD* é determinada por quatro fatores principais:

1. O número de operações que podem ser executadas concorrentemente.
2. A forma como as operações serão arranjadas para execução, podendo ser estaticamente, dinamicamente ou até mesmo utilizando ambas.
3. A ordem em que as operações são colocadas e retiradas em relação a ordem original do programa.
4. A maneira como o processador irá tratar cada exceção, podendo ser; preciso, impreciso ou ambos.

1.1.1 Processadores Escalares

Processadores escalares são processadores simples, que executam no máximo uma instrução e no máximo uma operação por ciclo de *clock* de máquina. As instruções do *stream* de instruções são executadas sequencialmente, assim uma nova instrução não será executada até que a execução da instrução em execução seja finalizada e seu resultado devidamente armazenado. A semântica de instrução determina que uma sequência de ações devem ocorrer para que se obtenha o resultado esperado, sendo: buscar a instrução, decodifica-la, acessar o dado ou registrador, execução da operação e armazenar o resultado. Podendo ocorrer *overlap* entre as ações mas o resultado deve aparecer na ordem especificada. Esse comportamento sequencial descreve o modelo de execução sequencial. No modelo de execução sequencial, a execução é dita *instruction-precise* se encontrar as seguintes condições:

- Todas as instruções ou operações que precederam a instrução atual ou operação atual já foram executadas e seus resultados armazenados.
- Todas as instruções ou operações na fila de execução não foram executadas ou seus resultados ainda não foram armazenados.
- A instrução ou operação em execução no momento está em um dos estados de execução, tendo ou não seu resultado já armazenado.

A maioria dos processadores escalares implementam diretamente o modelo de execução sequencial.

1.1.2 Processadores Superescalares

Enquanto processadores escalares estão limitados a executar uma única instrução por ciclo de *clock* de máquina os processadores superescalares decoficam várias instruções por ciclo de *clock* de máquina, utilizando várias unidades funcionais e alocação dinâmica para executar várias instruções por ciclo de *clock* de máquina. Processadores superescalares tem um comportamento similar ao *pipeline*.

A capacidade de executar múltiplas instruções implica em verificar se existe dependências entre as instruções, essa verificação tem que ser feita em nível de *hardware*. Processadores superescalares mais avançados geralmente incluem *hardwares* que preservam a ordem e precisamente lida com as exceções, assim simplificando o modelo de programação.

Devido a complexidade da lógica para alocação dinâmica das instruções, processadores superescalares de alto desempenho em geral estão limitados a executarem de quatro a oito instruções por ciclo de *clock* de máquina.

1.1.3 Processadores VLIW

Processadores VLIW (*Very Long Instruction Word*) assim como os processadores superescalares decodificam inúmeras instruções por ciclo de *clock* de máquina e utilizam várias unidades funcionais.

Ao contrário dos processadores superescalares que utilizam *hardware* para realizar alocação dinâmica das instruções, os processadores VLIW executam as instruções através de alocação estática, esta alocação depende de uma análise do compilador. Assim os processadores VLIW são menos complexos e apresentam desempenho potencialmente maior.

Processadores VLIW apresentam grande desempenho em aplicações que podem ser efetivamente alocadas de forma estática, embora nem todas as aplicações tenham esta característica, assim, a ordem de execução estática determinada pelo compilador não procede. Duas classes de execução podem ocorrer e afetar o comportamento da execução estática:

1. Atrasos de resultados das operações, devido a diferença da latência ocorrida com a latência considerada durante a alocação pelo compilador.
2. Exceções ou interrupções que colocam a ordem de execução em um estado não antecipado pelo compilador.

O processador consegue lidar com atrasos, embora isso tenha um impacto significante no desempenho. A causa mais comum de atrasos na execução devem ao dado não estar mais na memória cache, esse fator é tratado considerando-se o pior caso de latência

possível e até evitando o uso da memória cache. Na falta de paralelismo para cobrir as lacunas da latênica, resulta na alocação de instruções com um número menor de operações que o processador consegue executar, assim diminuindo o desempenho.

1.2 SIMD: SINGLE INSTRUCTION, MULTIPLE DATA

A classe de processadores *SIMD* incluem dois tipos de processadores, vetoriais e matriciais. Processadores *SIMD* são projetados para utilizarem determinadas estruturas de dados, como vetores e matrizes. Em nível de código de máquina, programar para processadores *SIMD* é bastante similar a processadores *SISD*, a diferença é realizar operações nas estruturas de dados agregadas. Como no processamento de algoritmos científicos há um grande uso de vetores e matrizes, processadores *SIMD* tem obtido grande desempenho na área.

Processadores vetoriais e matriciais apresentam diferenças tanto na implementação quanto na organização dos dados.

Processadores matriciais consistem em elementos de processos interconectados, cada um tendo seu próprio espaço de memória. Processadores vetoriais consistem em um único processo que referencia a um espaço de memória global.

1.3 MISD: MULTIPLE INSTRUCTION, SINGLE DATA

A classe de processadores *MISD* abstratamente é um *pipeline* de múltiplas unidades funcionais operando independentemente sob um único *stream* de dados. Em nível de micro-arquitetura é exatamente o que os processadores vetoriais fazem.

Segundo [4] exceto no caso de um cientista da computação interessado em estranhas formas de computação a classe *MISD* é uma forma restritiva e impraticável de paralelismo.

1.4 MIMD: MULTIPLE INSTRUCTION, MULTIPLE DATA

Na classe de processadores *MIMD* estão os multiprocessadores com alguma forma de interconexão entre os processadores. Do ponto de vista do programador, cada processo é executado independentemente e de forma cooperativa para solucionar um mesmo problema, embora alguma forma de sincronização entre os processos é necessária para que as informações e dados sejam trocados entre os processadores.

Não existe limitações em que todos os processadores sejam idênticos, embora a maioria das configurações *MIMD* sejam homogêneas, com processadores idênticos. Configurações heterogênias de processadores são geralmente utilizados para aplicações com propósitos específicos.

Da perspectiva de *hardware* existem dois tipos de *MIMD*, sendo os processadores *multi-cores* e processadores *multi-threaded*.

1.4.1 Processadores Multi-Threaded

Em *multi-threaded MIMD*, um processador base é estendido para incluir múltiplos conjuntos de registradores para dados e para o programa. Com essa configuração, diferentes *threads* ou programas ocupam cada conjunto de registrador. Assim que recursos se tornam disponíveis as *threads* continuam sua execução.

Uma vez que cada *thread* é independente, também o é no uso dos recursos disponíveis, assim múltiplas *threads*, fazem melhor uso dos recursos e em consequência aumenta-se o números de instruções executadas por ciclo de *clock* de máquina.

Threads ditas críticas, podem ter prioridade na execução para garantir que sejam executadas em menor tempo, enquanto *threads* não críticas se utilizam de recursos ociosos.

1.4.2 Processadores Multi-Core

Os processadores *multi-core* e também os múltiplos *multi-core*, necessitam comunicar os resultados de suas execuções através de uma rede de intercomunicação e de controle de tarefas. Assim sua implementação é significativamente mais complexa que processadores *multi-threaded*.

A rede de intercomunicação de troca dados entre os processadores realiza a sincronização das execuções independentes.

Quanto a comunicação realizada entre os processadores através de memória compartilhada surgem dois principais problemas, manter a consistência da memória e também a coerência de cache. A solução para ambos os problemas se dão em técnicas de *software* e *hardware*.

2 MEMÓRIA

A mais simples maneira de se melhorar o desempenho de um sistema é replicar os computadores e criar uma forma destes trocarem dados. Desta forma consegue-se aumentar o desempenho sem que seja necessário alterar a *CPU*.

Com o aumento contínuo da necessidade de desempenho em aplicações cada vez mais custosas a maioria dos sistemas paralelos utilizam-se de uma entre duas tecnologias, memória distribuída ou memória compartilhada.

2.1 MEMÓRIA DISTRIBUÍDA

Memória distribuída ou *distributed memory* ou *shared-nothing* é a mais simples abordagem do ponto de vista de *hardware*. A premissa desta abordagem é utilizar vários computadores interligados através de uma rede.

O modelo padrão de programação consiste de processos separados para cada computador que se comunicam através da troca de mensagem ou *message passing*, o que normalmente é feito através de bibliotecas desenvolvidas com esse propósito. Sendo este o modelo mais clássico de computação paralela. A forma moderna de sistemas com memória distribuída iniciou a partir do trabalho de Seitz em 1985 [**??**].

Devido ao baixo custo de processadores voltados ao mercado consumidor e da fácil montagem, alguns grupos exploraram tais fatores e começaram a construir *cluster* de computadores pessoais. Tais *clusters* já chamados de *NOWs*, *Network of Workstations*. Combinando todos estes fatores com o rápido avanço de desempenho de computadores pessoais e o avanço do *open-source* junto com versões de sistemas operacionais UNIX, ajudaram a difundir sistemas com tais caracteristicas. Hoje estes sistemas são comumente conhecidos como *Beowulfs* ou *Beowulf Cluster* devido ao projeto de Thomas Sterling e Donald Becker realizado na NASA.

2.2 MEMÓRIA COMPARTILHADA

Memória compartilhada ou *shared memory* é uma abordagem mais complexa, tornando a memória visível a todos os processadores, permitindo que todos possam carregar e gravar do mesmo endereço de memória.

Entre as dificuldades desta abordagem, os dois que chamam mais atenção são coerência e consistência. Sendo a consistência o mais problemático para o programador.

3 PROGRAMAÇÃO PARALELA

A programação paralela é uma camada abstrata sobre o *hardware*, e em geral os modelos de programação paralela não são específicos para a arquitetura de *hardware*.

Existem vários modelos de programação paralela em uso em 2013.

a) Memória compartilhada ou *shared memory* (sem *threads*).

b) *Threads*.

c) Memória distribuída ou *distributed memory* (Troca de mensagens ou *message passing*).

d) *Data parallel*.

e) Híbrido.

f) SPMD (*Single Program, Multiple Data*).

g) MPMD (*Multiple Program, Multiple Data*).

4 DATA FLOW GRAPH

Data Flow Graph (DFG) ou grafo de fluxo de dados, é um modelo para programas que expressa a possibilidade de execução concorrente de partes do programa. Nos DFGs os nós representam operações (funções) e predicados a serem aplicados a objetos de dados e as arestas representam a ligação entre o nó que produz o dado e o nó que irá consumir aquele dado. Na literatura os nós também são chamados de atores. Assim, aspectos de controle e de dados de um programa podem ser representados em um único modelo integrado.

Embora muitas versões de DFGs tenham sido estudadas na literatura, elas possuem algumas características em comum:

a) DFG é um grafo orientado onde uma aresta é um caminho que um dado percorre do nó produtor para o nó consumidor.

b) Dinamicamente, o nó de um DFG aceita um ou mais dado como entrada, realizando computações e devolvendo os dados do retorno para suas saídas.

c) Uma ação de um nó é ativada com a presença dos dados de entrada.

Os estudos em DFGs tem sido focado principalmente em três modelos bem definidos: DFGs estáticos, DFGs dinâmicos e DFGs síncronos.

5 DEPÊNDENCIA

Quando um programador escreve um programa em uma linguagem sequêncial, o resultado esperado será obtido pela execução da primeira linha, depois a segunda e assim em diante, considerando exceções de controles de fluxos como *loops* e ramificações. Uma vez que o programador especificou a ordem que ele espera que as computações sejam realizadas. Obter paralelismo de um programa respeitando a estas especificações não é possível, uma vez que obter paralelismo implica em alterar a ordem das operações realizadas.

Paralelizar um programa sequêncial significa encontrar uma ordem de execução diferente da especificada e que irá sempre computar o mesmo resultado. A programação sequêncial introduz restrições que podem ser críticas para o resultado esperado do programa, assim para transformar um programa em paralelo é importante encontrar as restrições menos críticas e realizar transformações para que o programa continue retornando o resultado correto para qualquer entrada.

Uma dependência é uma relação entre duas declarações no programa. Um par de declarações $< S_1, S_2 >$ está em uma relação se S_2 é executada depois de S_1 em um programa sequêncial, e deve ser executada após S_1 em qualquer reordenação válida do programa se a ordem de acesso a memória será preservada.

```
S1   PI = 3.14159
S2   R = 5
S3   AREA = PI * R * R
```

Os resultados obtidos por estas declarações são definidas por aqueles obtidos quando a ordem da execução realizada seja $< S_1, S_2, S_3 >$. No entanto nada neste trecho de código torna obrigatório a execução de S_2 depois de S_1, desta forma, os resultados obtidos pela ordem de execução $< S_2, S_1, S_3 >$ serão os mesmos para a variável $AREA$, seja qual for o valor da entrada. Por outro lado, o momento de execução de S_3 é mais crítico, se S_3 for executada antes de S_1 ou de S_2, os resultados obtidos por esta execução diferenciariam dos resultados obtidos das computações realizadas na ordem original. Em termos de dependência pode-se observar que os pares $< S_1, S_3 >$ e $< S_2, S_3 >$ estão em uma relação de dependência, embora o par $< S_1, S_2 >$ não. Dependências desse tipo são ditas dependência de dados.

Dependência em linhas de código sequêncial como visto anteriormente, é um conceito simples de entender. O problema é que examinar somente linhas de códigos sequênciais não garante eficiência em termos de paralelismo. Para se obter tal eficiência deve-se

considerar os trechos de código que são mais executados, ou seja, devemos expandir o conceito de dependência para *loops* e vetores. O trecho de código a seguir ilustra a complexidade introduzida ao expandirmos o conceito de dependência.

```
        for (int I = 1; I < N; I++){
S1          A[I]   = B[I] + 1;
S2          B[I+1] = A[I] - 5;
        }
```

Este *loop* mostra a dependência entre $< S_1, S_2 >$, uma vez que o resultado computado de A é imediatamente utilizando por S_2 em todas as iterações, e também a dependêndia entre $< S_2, S_1 >$ exceto na primeira iteração, uma vez que o resultado obtido por S_2 será utilizado na iteração anterior. Detectar estas dependências é difícil, considerando que cada iteração acessa diferentes elementos do vetor.

Loops e vetores são apenas parte do problema que envolve a análise de dependência, deve-se considerar também as estruturas condicionais, como as declarações IF.

Deve-se então entender um outro tipo de dependência, dado o trecho de código a seguir:

```
S1     if(d != 0)
S2           a = a / d;
```

A declaração $S2$ não pode ser executada antes de $S1$, uma vez que essa transformação pode ocasionar em uma divisão por zero, o que não ocorreria no programa original. Essa dependência é chamada de dependência de controle.

5.1 DEPENDÊNCIA DE DADOS

Em dependência de dados deve-se garantir que um dado seja produzido e consumido na ordem correta, assim, cuidando para que não se intercale o *load* e o *store* em um mesmo local da memória, desta forma, o próximo *load* pode obter um valor errado. Da mesma forma, dois *stores* devem ocorrer na ordem correta para que no próximo *load* seja obtido o valor correto. Assim, dependência de dados pode ser definida como:

```
Definição 1: Existe dependência de dados da declaração $S_1$ para a declaração
$S_2$ (declaração $S_2$ depende da declaração $S_1$) se e somente se:
        1 -> Ambas as declarações acessem o mesmo local de memória e ao menos
        umas delas realizará \textit{store} na memória, e
        2 -> Existe um caminho de execução viável de $S_1$ para $S_2$.
```

Neste capítulo serão apresentadas várias propriedades onde as dependências podem ser classificadas.

5.1.1 Classificação de *Load-Store*

Em termos da ordem de *load-store*, as dependências podem ocorrer de três formas em um programa:

a) *True dependence.* Onde uma declaração realiza *store* em um local da memória em que será realizado um *load* por uma segunda declaração.

```
S1    x = ...
S2    ... = x
```

A dependência garante que S_2 irá ler exatamente o que foi computado por S_1. Esse tipo de dependência é também conhecida por dependência de fluxo e é denotada por $S_1 \delta S_2$ (lê-se, S_2 depende de S_1).

b) *Antidependence.* Uma primeira declaração realiza *load* de um local onde uma segunda declaração irá escrever.

```
S1    ... = x
S2    x = ...
```

Esta dependência previne a troca na ordem de execução entre S_1 e S_2, tal qual poderia resultar em S_1 utilizando-se do valor computado por S_2. Em essência essa dependência existe para prevenir uma transformação que introduziria uma nova dependência do tipo *true dependence* que de fato não existe no programa original. *Antidependence* é denotado $S_1 \delta^- S_2$ ou $S_1 \delta^{-1} S_2$.

c) *Output dependence.* Ocorre quando duas declaração realizam *store* em um mesmo local.

```
S1    x = ...
S2    x = ...
```

Essa dependência previne que ocorra uma troca entre as declarações e faça com que uma declaração que irá realizar *load* do valor computado não leia o valor errado.

```
S1    x = 1
S2    ...
S3    x = 2
S4    y = 2 * x
```

Output dependence é denotado $S_1\delta^0 S_2$.

5.1.2 Dependência em Loops

Extendendo o conceito de dependência para *loops* requer de alguma forma parametrizar as declarações pelas iterações do *loop* que são executadas. Para um simples *loop*:

```
        for(int i = 1; i < N; i++)
S1          a[i+1] = a[i] + b[i]
```

A declaração S_1 em qualquer iteração do *loop* depende dela mesmo da iteração anterior. Embora uma simples alteração no *index* do vetor pode fazer com que a declaração tenha dependência de duas iterações anteriores.

```
        for(int i = 1; i < N; i++)
S1          a[i+2] = a[i] + b[i]
```

Para melhor definir dependências em *loops*, pode se fazer necessário o uso de alguma forma de parametrização das declarações, para que através desta representação seja possível identificar em qual iteração as declarações ocorrem. Para isso faz-se necessário o uso de números inteiros auxiliares (ou um vetor de números inteiros) os quais irão representar o número da iteração de cada *loop* onde as declaração estão aninhadas. Para um *loop* simples

```
for(int i = 1; i < N; i++)
    ...
```

o número da iteração é igual ao número do *loop index*, neste caso o i. O *index* inicia em 1 na primeira iteração, assumindo o valor 2 na segunda iteração e assim por diante.

Considerando um *loop* parametrizado

```
for(int i = L; i < U; i = i + S)
    ...
```

o número da iteração será 1 quando i for igual a L, será 2 quando i for igual a $L + S$ e assim em diante. Formalizando-se o conceito de parametrização;

> Definição 2: Para um loop arbitrário em que o index I do loop é incrementado de L até U em incrementos de S, o número da iteração (normalizado) i de uma iteração qualquer terá o valor de (I - L + 1)/ S, onde I é o valor do index daquela iteração.

No caso de *loops* aninhados, o nível de aninhamento de um determinado *loop* será igual ao número de *loops* que estão iterando sobre ele somado de um. Em aninhamento de *loops* os mesmo serão enumerados do mais externo para o mais interno, começando em 1. Formalizando-se este conceito tem-se;

> Definição 3: Para um aninhamento de n loops, o vetor de iteração i de uma determinada iteração no loop mais interno é um vetor de inteiros que contém o número da iteração de cada loop na ordem do aninhamento. O vetor de iterações é dado pela seguinte equação
>
> $i = {i_1, i_2, ..., i_n}$
>
> onde i_k, 1 <= k <= n, representa o número da iteração do k-ésimo loop aninhado.

Considerando-se um aninhamento de dois *loops* e o vetor de iteração $it\_array[2]$, onde $it\_array[0]$ contém o número da iteração do *loop* mais externo e $it\_array[1]$ do loop mais interno. O conjunto de todas as possibilidades do vetor de iteração é chamado de espaço de iteração (*iteration space*). O espaço de iteração para este aninhamento seria $\{(1,1),(1,2),(2,1),(2,2)\}$.

Devido a importância da ordem de execução para tratamento de dependência, vetores de iteração precisam ser precisos quanto a ordem dos seus elementos internos em relação a ordem de aninhamento dos *loops*.

REFERÊNCIAS

1 FLYNN, M. Very high-speed computing systems. *Proceedings of the IEEE*, v. 54, n. 12, p. 1901–1909, 1966. ISSN 0018-9219.

2 FLYNN, M. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21, n. 9, p. 948–960, 1972. ISSN 0018-9340.

3 FLYNN, M. J.; RUDD, K. W. Parallel architectures. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 28, n. 1, p. 67–70, mar. 1996. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/234313.234345>.

4 OPENSHAW, S.; TURTON, I. *High Performance Computing and the Art of Parallel Programming: An Introduction for Geographers, Social Scientists and Engineers*. Taylor & Francis, 1999. ISBN 9780415156929. Disponível em: <http://books.google.com.br-/books?id=Q-ovcxwhwLwC>.