UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA DE SÃO CARLOS
DEPARTAMENTO DE FÍSICA E INFORMÁTICA

Avaliação de Métodos de Paralelização Automática

EDSON PEDRO FERLIN

Dissertação apresentada ao Instituto de Física de São Carlos, da Universidade de São Paulo, para obtenção do título de Mestre em Ciências: Física Aplicada opção Física Computacional

Orientador: Prof. Dr. Carlos Antônio Ruggiero

SÃO CARLOS – SP
1997

Ferlin, Edson Pedro
Avaliação de Métodos de Paralelização Automática / Edson Pedro Ferlin. São Carlos, 1997.
122p.

Dissertação (Mestrado)—Instituto de Física de São Carlos, 1997.

Orientador: Prof. Dr. Carlos Antônio Ruggiero

1.Processamento Paralelo. 2.Programação Paralela. 3.Métodos de Paralelização.
I. Título.

UNIVERSIDADE DE SÃO PAULO
Instituto de Física de São Carlos

Av. Dr. Carlos Botelho, 1465
CEP 13560-250 - São Carlos - SP
Brasil

Fone (016) 272-6222
Fax (016) 272-2218

MEMBROS DA COMISSÃO JULGADORA DA DISSERTAÇÃO DE MESTRADO DE **EDSON PEDRO FERLIN** APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO CARLOS, DA UNIVERSIDADE DE SÃO PAULO, EM 24 DE MARÇO DE 1997.

COMISSÃO JULGADORA:

Prof. Dr. Carlos Antonio Ruggiero/IFSC-USP

Prof. Dr. Jan Frans Willem Slaets/IFSC-USP

Prof. Dr. Jairo Panetta/ITA-CTA

Agradeço a Deus, minha Mãe, Parentes e amigos pelo incentivo e motivação.

Meus agradecimentos especiais ao Prof. Dr. Gonzalo Travieso que orientou-me durante a primeira fase da dissertação, ao Prof. Dr. Carlos Antônio Ruggiero que me orientou até a conclusão da mesma, e também ao Prof. Dr. Jairo Panetta pelo auxilio nas correções.

Agradeço, ainda, a todos os professores e à seção de Pós-Graduação, na pessoa da Wladerez Aparecida Gounella Caiado, por toda a atenção a mim dispensada durante o Mestrado, e ao CNPQ pela concessão da bolsa de estudos.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CRC	-	Cyclic Redundancy Check
GCD	-	Greatest Commom Divisor
LINUX	-	Sistema Operacional para PC baseado no UNIX
MIMD	-	Multiple Instruction Stream, Multiple Data Stream
MPI	-	Message Passing Interface
MPICC	-	Compilador "C" para ambiente de passagem de mensagem (MPI)
MPIRUN	-	Interpretador para ambiente de passagem de mensagem MPI
MPMD	-	Multiple Program, Multiple Data
OCCAM	-	Linguagem de Programação para Transputers
PVM	-	Passing Virtual Machine
SISAL	-	Streams and Interactions in a Single Assignment Language
SPMD	-	Single Program, Multiple Data
SRP	-	Skewing - Reversal - Permutation

RESUMO

Este trabalho aborda alguns conceitos e definições de processamento paralelo, que são aplicados à paralelização automática, e também às análises e condições para as dependências dos dados, de modo a aplicarmos os métodos de paralelização: Hiperplano, Transformação Unimodular, Alocação de Dados Sem Comunicação e Particionamento & Rotulação.

Desta forma, transformamos um programa seqüencial em seu equivalente paralelo. Utilizando-os em um sistema de memória distribuída com comunicação através da passagem de mensagem MPI (Message-Passing Interface), e obtemos algumas métricas para efetuarmos as avaliações/comparações entre os métodos.

ABSTRACT

This work invoke some concepts and definitions about parallel processing, applicable in the automatic paralelization, and also the analysis and conditions for the data dependence, in order to apply the methods for paralelization: Hyperplane, Unimodular Transformation, Communication-Free Data Allocation and Partitioning & Labeling.

On this way, transform a sequential program into an equivalent parallel one. Applying these programs on the distributed-memory system with communication through message-passing MPI (Message-Passing Interface), and we obtain some measurements for the evaluations/comparison between those methods.

CAPÍTULO 1 - INTRODUÇÃO

Os sistemas de processamento paralelo, geralmente vistos como a opção mais viável para computação de alta performance, apresentam-se atualmente no que podemos denominar de uma crise. Apesar de haverem sido desenvolvidos diversas arquiteturas paralelas alternativas (por exemplo um apanhado em [Hockey *et al.* 1988; Ibbet *et al.* 1989]), e de muitas dessas arquiteturas apresentarem a possibilidade de bom crescimento da performance com o número de processadores e boas razões preço/desempenho, no entanto, o uso de processamento paralelo é ainda extremamente restrito. O problema básico que tem limitado o uso mais amplo de arquiteturas paralelas é relacionado à programação dessas máquinas.

Falando-se em termos gerais, a programação dessas máquinas é realizada em uma das três formas seguintes, conforme Perrot [1989]:

- **Paralelização explícita** onde o programador deve dizer, explicitamente, quais as tarefas que devem ser executadas em paralelo, e a forma como essas diversas tarefas devem cooperar entre si. Como exemplo, citamos occam [Pountain *et al.* 1988] e MPI [MPI Forum 1994].
- **Paralelização implícita** Neste caso, o programador desenvolve o seu algoritmo em alguma linguagem que represente uma descrição do que deve ser executado sem inclusão de seqüencialização, como por exemplo através de uma linguagem não-imperativa. Como exemplo podemos citar SISAL [Cann 1993].
- **Paralelização Automática** Aqui, o programador desenvolve um programa numa linguagem seqüencial tradicional, e o compilador é responsável por extrair o paralelismo. Por exemplo [Hilhorst *et al.* 1987; Wolf *et al.* 1991; D'Hollander 1992; Chen *et al.* 1994].

Infelizmente, cada uma dessas formas apresenta inconvenientes ainda não satisfatoriamente resolvidos. A programação explícita exige do programador afastar-se de considerações sobre a lógica do programa, para levar em conta fatores como a especificação da arquitetura paralela utilizada. Além disso, o desenvolvimento explícito de programas paralelos é lento, e as análises efetuadas podem ser invalidadas com as mudanças no algoritmo a ser implementado. A paralelização

implícita exige que o programador aprenda uma linguagem nova, e descreva o seu algoritmo nessa linguagem. Como muitas vezes a linguagem em questão tem pouco uso ou baixa eficiência fora da específica arquitetura paralela na qual é usada, esta é uma opção não muito atrativa. Ambas as formas anteriores têm ainda o inconveniente de que programas já existentes e extensivamente testados não podem ser transportados para a arquitetura paralela. A paralelização automática tenta resolver esse problema. No entanto, os métodos de paralelização automática apresentados até hoje têm tido efeito muito reduzido, e se caracterizado por uma falta de análises com base em arquiteturas reais. No entanto dada sua importância, novas pesquisas devem ser realizadas nessa direção.

O presente trabalho tem como objetivo o estudo dos métodos de paralelização automática propostos na literatura, tanto em termos de uma análise comparativa *a priori* dos diversos métodos, como de um estudo de sua adequação para sistemas paralelos com passagem de mensagem.

Foi realizado um extensivo levantamento bibliográfico na área de compiladores paralelizadores, com ênfase especial à literatura mais recente e aos métodos de caráter geral, isto é, que não estejam excessivamente vinculados a uma única arquitetura paralela.

Após este levantamento, foi realizado um estudo comparativo dos diversos métodos, principalmente tomando em consideração a sua adaptação a sistemas paralelos com passagem de mensagens.

Como uma fase final, foi realizado um levantamento da eficiência de alguns métodos através da sua implementação (manualmente) em um conjunto de algoritmos simples, em um sistema de passagem de mensagens.

No capítulo 2, são descritos alguns conceitos e definições de processamento paralelo, que são utilizadas no decorrer do trabalho; no capítulo 3, foi colocada a condição necessária (condição de *Bernstain*) para se verificar se os comandos podem ser executadas em paralelo; no capítulo 4, são descritos os tipos de dependência de dados; no capítulo 5, algumas técnicas para eliminação de algumas dependências; no capítulo 6, é descrito o primeiro dos métodos, o Método Hiperplano; no capítulo 7, o Método Transformações Unimodulares; no capítulo 8, o Método de Alocação de Dados Sem Comunicação, no capítulo 9, o Método de Particionamento & Rotulação; no capítulo 10, é descrita a interface de passagem de mensagem (MPI), sobre a qual foi realizada a implementação; no capítulo 11, é descrita toda a implementação dos algoritmos transformados (manualmente) por meio dos métodos em MPI; no capítulo 12, são mostrados os resultados de execução dos algoritmos em ambiente de passagem de mensagem; no capítulo 13, são tecidas

algumas considerações a respeito dos métodos e da implementação dos algoritmos transformados no ambiente de passagem de mensagens; no Anexo são apresentados os Laços Teste, os Laços Teste Paralelizados, e as Listagens dos Programa Paralelizados codificados em Linguagem "C" com o MPI; e por fim na Referência Bibliográfica, estão relacionadas todas as referências à bibliografia utilizadas e citadas no trabalho.

CAPÍTULO 2 - CONCEITOS E DEFINIÇÕES

Neste capítulo, mencionamos alguns conceitos e definições de processamento paralelo, utilizados no decorrer do trabalho, para facilitar o entendimento e compreensão do mesmo.

2.1. PROCESSAMENTO PARALELO

O que é processamento paralelo? Segundo a definição de Hwang *et al.* [1985]: é uma forma eficiente do processamento da informação com ênfase na exploração de eventos concorrentes no processo computacional.

A razão para o surgimento do Processamento Paralelo é a capacidade de aumentar o processamento com uma única máquina. Como o aumento da velocidade nas máquinas seqüenciais é limitado pela tecnologia, a solução empregada para aumentar o poder de processamento é a utilização de processadores em paralelo.

O Processamento Paralelo trata desde a aplicação; a partição de algoritmos paralelos, as linguagens paralelas; os compiladores paralelizadores, compiladores vetorizadores, compiladores otimizadores; os sistemas operacionais para gerenciar este paralelismo; as arquiteturas paralelas; e o mapeamento de algoritmos.

2.2. LINGUAGENS DE PROGRAMAÇÃO PARA PROCESSAMENTO PARALELO

As linguagens de programação hoje disponíveis em computadores paralelos podem ser grosseiramente classificadas em três grupos. O primeiro grupo é formado por novas linguagens projetadas especificamente para processamento paralelo. O segundo por linguagens convencionais ampliadas por primitivas para expressar o paralelismo; e o terceiro por linguagens convencionais sem extensões, onde o paralelismo é extraído automaticamente pelo compilador, de acordo com Mokarzel *et al.* [1988], Perrott [1990].

Atualmente, todos os computadores paralelos comercialmente disponíveis são programados ou por meio de linguagens convencionais com extensões ou por meio de novas linguagens paralelas. Este fato é inconveniente para os usuários, visto que o uso de tais linguagens exige alterações nos seus programas. Para evitar tal esforço, deve-se utilizar linguagens convencionais sem extensões, onde a tarefa de extrair paralelismo é efetuada automaticamente pelo compilador. A importância desta opção cresce quando se lembra que há trinta anos de programas desenvolvidos em FORTRAN, alguns deles com milhares de linhas de código fonte.

2.3. MICROTASKING, MACROTASKING E AUTOTASKING

O processamento paralelo a nível de subrotina é chamado Macrotasking, figura 2.1, onde as tarefas são dividas entre os processadores disponíveis na arquitetura.

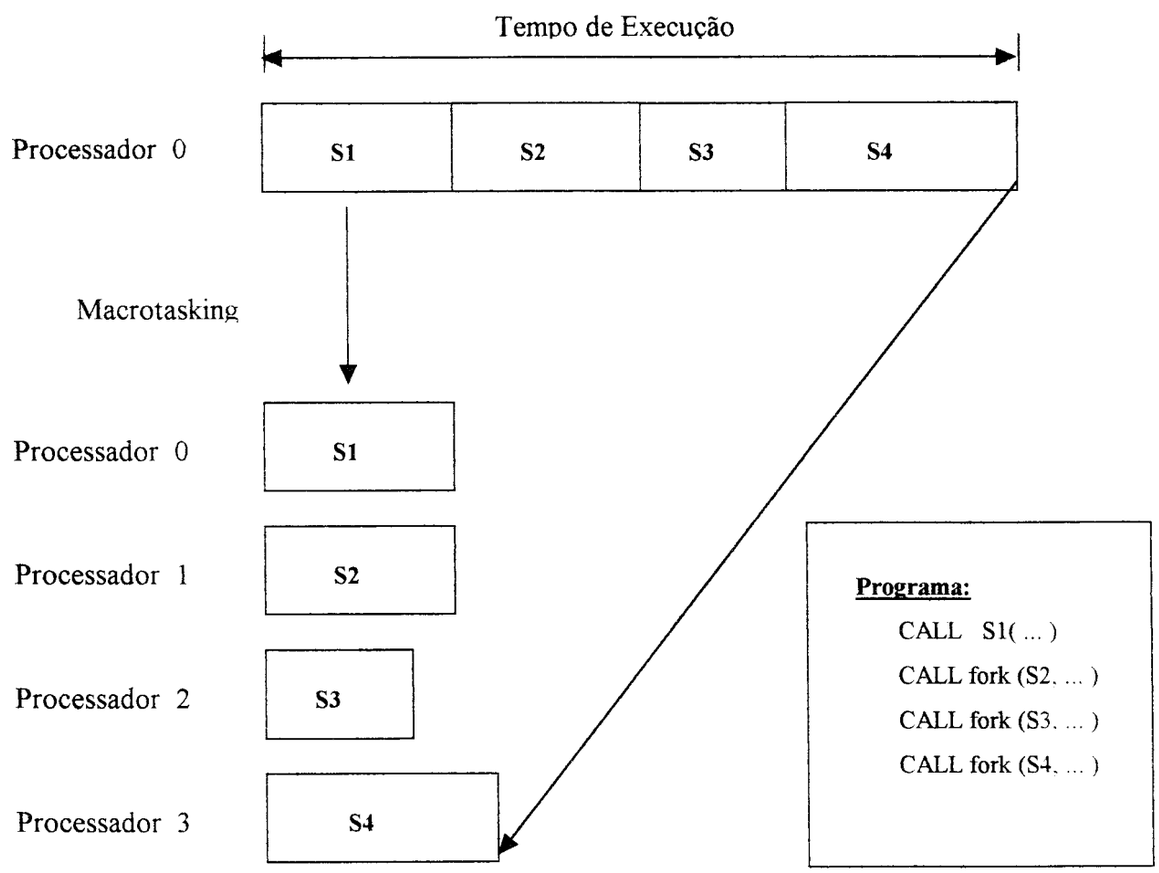


Figura 2.1. Processamento Paralelo a nível de Macrotasking

Microtasking é o processamento paralelo principalmente a nível de laços DO, mas são também diversas construções para sub-rotinas e para níveis de comandos, de acordo com Gentzsch [1990], NEC [1993], Tzen *et al.* [1993]. Com a microtasking não temos a necessidade do uso explicito da divisão do código dentro das tarefas, mas apenas uma indicação do lugar em que o código pode ser seguramente executado simultaneamente em múltiplas CPUs, figura 2.2.

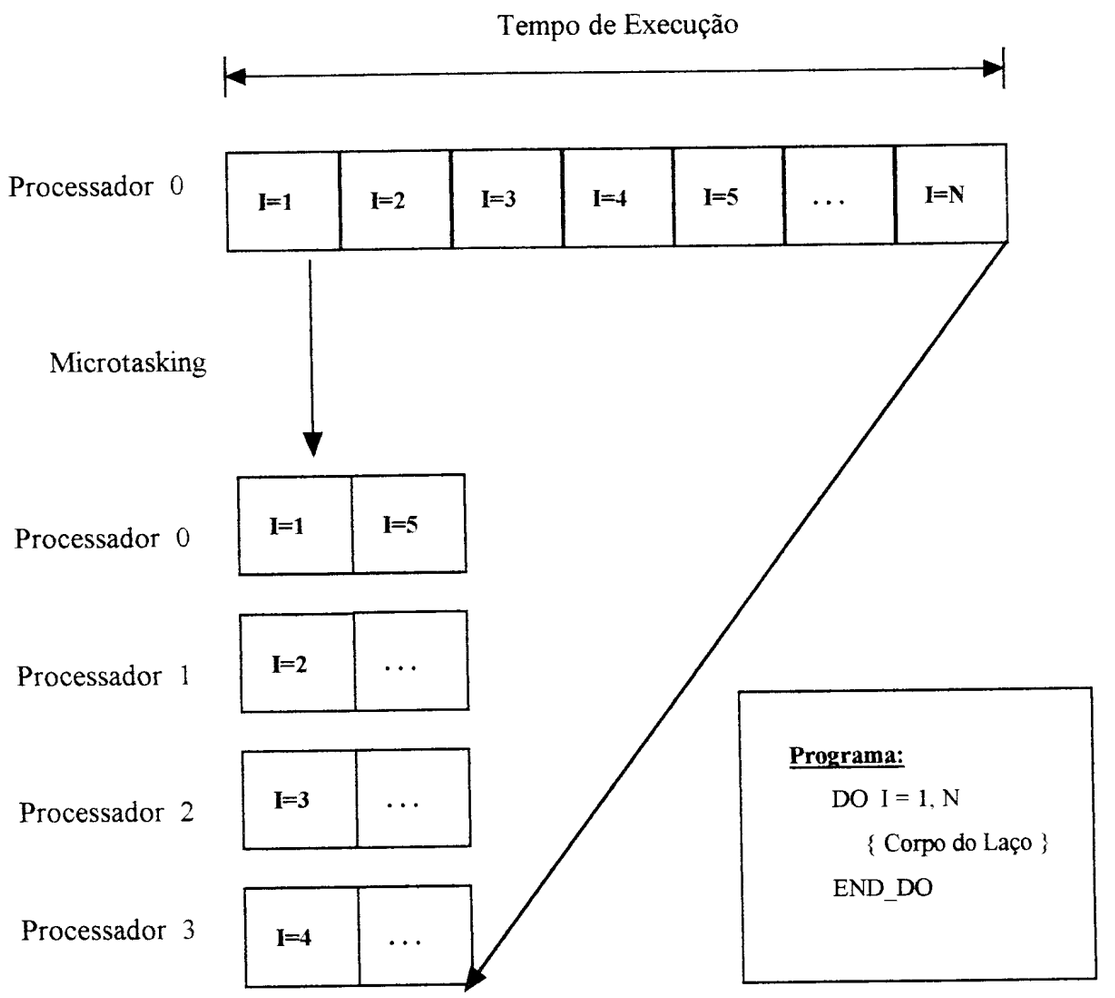


Figura 2.2. Processamento Paralelo a nível de Microtasking

Para a paralelização automática ou Autotasking, figura 2.3, temos uma Macrotasking seguida de uma Microtasking, de acordo com Gentzsch [1990],

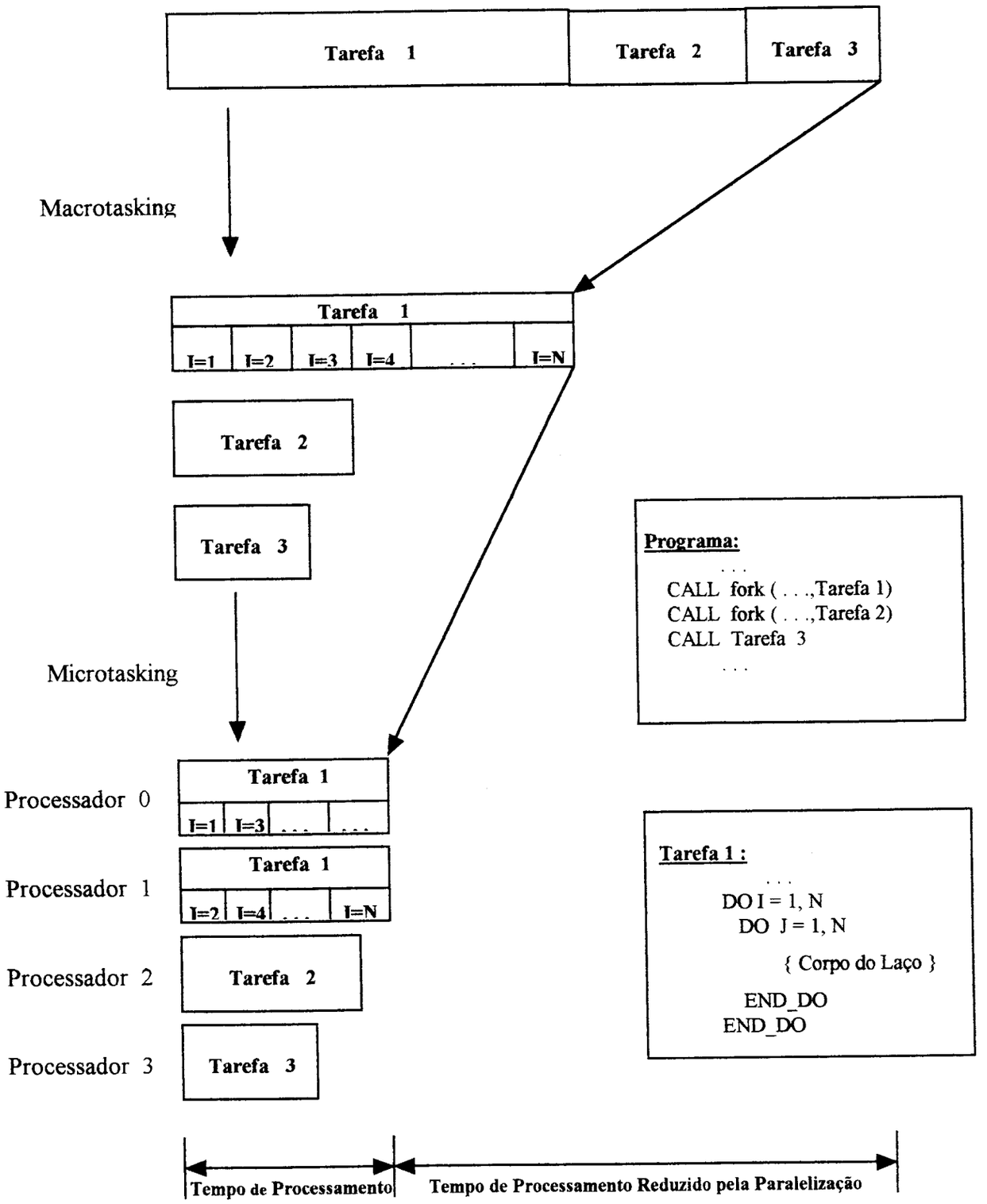


Figura 2.3. Processamento Paralelo a nível de Autotasking

2.4. FASES DE UM DETECTOR AUTOMÁTICO

Um compilador-detector de paralelismo possui todas as fases usuais de um compilador (análise sintática, análise semântica, análise léxica, geração do código e otimização do código) e mais duas fases exclusivas, uma de detecção de paralelismo e outra de alocação de recursos, segundo Navaux [1988].

2.4.1. DETECÇÃO DE PARALELISMO

A fase de detecção de paralelismo produz sucessivas versões do programa fonte, aplicando transformações que mantém a semântica original. Essas transformações buscam obter a independência, quanto à ordem de execução, dos comandos que compõe o programa. Quanto maior a independência entre comandos, maior a simultaneidade na execução e maior o grau de paralelismo que pode ser atingido.

Para a divisão de um programa num conjunto de processos que possam ser executados simultaneamente é importante a identificação dos objetos que são partilhados pelos processos. Esta divisão pode ser feita de forma explícita pelo próprio programador, ou de forma implícita quando o compilador detectar o paralelismo.

Uma das formas de estabelecer o paralelismo explicito é através de comandos *FORK* e *JOIN*. Existem também outras formas como *COBEGIN*, *COEND* e *PARFOR*.

No caso de paralelismo implícito, torna-se necessário reconhecer os processos passíveis de paralelismo pela análise do programa fonte, o que é bem mais complexo. Uma forma de estabelecer isto, é pela análise da dependência de dados.

2.4.2. ALOCAÇÃO DE RECURSOS

Uma vez restruturado, o programa estará particionado em conjuntos de tarefas, de modo que todas as tarefas de um mesmo conjunto possam ser executadas em paralelo. Já os conjuntos de

tarefas devem ser executados em uma certa ordem, devido às dependências entre os componentes desses conjuntos.

Como diversas técnicas de restruturação não consideram o número de processadores efetivamente disponíveis, é necessário alocar tarefas a processadores em uma fase posterior à restruturação.

O balanceamento da carga entre processadores é um item pesquisado com vistas a otimizar o paralelismo. Este balanceamento pode ser dinâmico, quando os processos são alocados aos processadores em tempo de execução, ou então estáticos quando esta alocação for estabelecida por ocasião da compilação, de acordo com Navaux [1988].

Há alternativas de escalonar processadores estaticamente (durante a compilação) ou dinamicamente (durante a execução), ou ainda de forma mista.

O escalonamento dinâmico de processos paralelos depende das condições de execução; soluções ótimas são impossíveis. Existem várias técnicas heurísticas tais como *first-in-first-out* (primeiro a chegar primeiro a ser atendido), *round robin* (atendimento segundo um escalonamento circular), *shortest-process-first* (processo menor primeiro a ser atendido), *least-memory-first* (memória menor primeiro atendido), e outros, conforme Hwang *et al.* [1985].

O escalonamento estático é determinado por ocasião da compilação, porém normalmente os resultados de utilização dos processadores são pobres.

2.5. NÍVEIS DE PARALELISMO - GRANULARIDADE

O paralelismo no processamento é encontrado em diversos níveis diferentes. Existem diversas classificações com um maior ou menor número de níveis, dependendo do autor; Hwang [1984] sugere cinco níveis: nível 5 - de processos independentes (*jobs*) e programas; nível 4 - sub-processos e pontes de programas; nível 3 - rotinas, sub-rotinas e co-rotinas; nível 2 - iterações (laços) e por último, nível 1 - de instruções. O nível mais elevado é o algorítmico e o inferior é o de instruções que é implementado em hardware. Quanto mais baixo o nível mais fina é a granularidade do processamento. O processamento paralelo explora um destes níveis ou até mesmo combinação destes, dependendo da máquina e do software de paralelização (compilador).

A execução de grandes porções de software em paralelo com pouca ou nenhuma comunicação entre elas é chamado paralelismo de granularidade grossa. O extremo oposto, tal como

encontrado no processamento vetorial que é a nível de laços, é chamado paralelismo de granularidade fina, segundo Tanenbaum [1992].

2.6. COMPILADORES PARALELIZADORES

As duas formas principais de paralelismo que se empregam atualmente na criação de compiladores são: a vetorização e o multiprocessamento, de acordo com Padua *et al.* [1986].

O atual estágio de desenvolvimento dos compiladores vetorizadores já é significativo. O funcionamento é baseado na detecção de laços DO, adaptando-os de forma a possibilitar que a máquina em questão (em geral pipeline ou matricial) execute partes do programa concorrentemente.

A outra forma de paralelismo, bem menos pesquisada, mas que agora está recebendo uma maior ênfase é o multiprocessamento. Neste caso, o compilador busca a detecção de trechos de programas que possam ser executados paralelamente. Basicamente estes compiladores detectam as dependências de dados de forma a separar os trechos de programas independentes para que possam ser executados paralelamente, de acordo com Navaux [1988].

CAPÍTULO 3 - CONDIÇÃO DE *BERNSTAIN*

Neste capítulo, descrevemos a condição de *Bernstein*, sob a qual verificamos se os comandos podem ou não serem executados em paralelo.

3.1. DEFINIÇÃO

DEFINIÇÃO: Dois comandos S e T podem ser executadas em paralelo (ou concorrentemente) se existir um resultado igual, quando executamos primeiro S e depois T, ou primeiro T e depois S, ou T e S simultaneamente, segundo Hilhorst *et al.* [1987].

Denotamos S como o comando A = f(x1, x2) e dizemos que x1 e x2 são variáveis de entrada de S e que A é a variável de saída de S. Isto é denotado por x1,x2 $\in$ in(S) e A $\in$ out(S).

3.2. TEOREMA

TEOREMA 3.1:(Condição de *Bernstein*). Dois comandos S e T podem ser executados em paralelo, se e somente se

- in(S) $\cap$ out(T) = $\varnothing$
- in(T) $\cap$ out(S) = $\varnothing$
- out(S) $\cap$ out(T) = $\varnothing$

Similarmente, os N comandos Ti, para i=1,...,N podem ser executados em paralelo, se e somente se

- in(Ti) $\cap$ out(Tj) = $\varnothing$, para todo i$\neq$j
- out(Ti) $\cap$ out(Tj) = $\varnothing$, para todo i$\neq$j

Para expressar a execução concorrente para os comandos dentro do laço, usamos a notação

DO **CONC** FOR ALL I$\in${i: 1$\leq$i$\leq$N}

{ **Corpo do Laço** }

END_DO

Isso significa que atribuímos um processador para cada valor do índice I de 1 até N. Os processadores atribuídos para i $\in$ {1,...,N} conjuntos I=i e executam os comandos para o corpo do laço. Todos os processadores executam concorrentemente, independentemente um dos outros, de acordo com Hilhorst *et al.* [1987].

A seguir expressamos para a execução simultânea para os comandos de um corpo de laço

```
DO SIM FOR ALL I∈{i: 1≤i≤N}
        { Corpo do Laço }
END_DO
```

Aqui, o primeiro comando do corpo do laço é executado simultaneamente em todos os processadores, onde cada um destes valores do elemento do vetor é alterado (como um resultado da computação), apenas depois que todas as possibilidades, tiverem usados os elementos de computação para o corpo do laço. Depois o segundo comando, e assim sucessivamente.

Obviamente, se a execução concorrente para um laço é possível, então a execução simultânea também é possível. Entretanto, a execução simultânea pode ser possível, enquanto que a execução concorrente não é possível.

Por exemplo, consideremos o laço

```
DO I = 1, N
        A(I) = A(I+1)*5
END_DO
```

pode ser re-escrito da seguinte forma

```
DO SIM FOR ALL I∈{i: 1≤i≤N}
        A(I) = A(I+1)*5
END_DO
```

Entretanto, o laço

```
DO CONC FOR ALL I∈{i: 1≤i≤N}
        A(I) = A(I+1)*5
END_DO
```

não faz sentido, pois se denotamos S e T as atribuições

S: A(3) = A(4) * 5

T: A(4) = A(5) * 5

então o resultado dependerá se S é executado depois de T, ou se T é executado depois de S.

CAPÍTULO 4 - DEPENDÊNCIA DOS DADOS

A dependência de dados é que irá determinar a forma na qual podemos proceder a partição do programa e do conjunto dos dados, isto é, o espaço de iteração.

As dependências são obtidas mediante a verificação da condição de *Bernstain*, que foi descrita no capítulo anterior.

4.1. INTRODUÇÃO

As dependências podem ser de três tipos:

- *Flow*-dependência
- *Anti*-dependência
- *Output*-dependência

A *Flow*-dependência ocorre quando uma variável é atribuída ou definida em um comando e usada em um comando executado subseqüentemente, segundo Lu *et al.* [1992], Wolfe [1992], Lee [1995].

A *Anti*-dependência ocorre quando uma variável é usada em um comando e re-atribuída em um comando executado subseqüentemente.

A *Output*-dependência ocorre quando uma variável é atribuída em um comando e re-atribuída em um comando executado subseqüentemente.

Anti-dependência e *output*-dependência são geradas a partir de variáveis reutilizadas ou re-atribuídas e são algumas vezes chamadas de falsas dependências.

A *flow*-dependência é também chamada dependência verdadeira, desde que seja inerente na computação e não possa ser eliminada pela renomeação das variáveis.

As relações de dependência podem ser encontradas pela comparação dos conjuntos IN e OUT de cada nó em um Grafo de Dependências.

O conjunto IN de um comando S, IN(S), é o conjunto das variáveis (ou, mais precisamente, o conjunto das posições de memória, usualmente referenciado pelos nomes das variáveis) que devem ser lidas (ou buscadas) por este comando.

O conjunto OUT de um comando S, OUT(S), é um conjunto de posições de memória que podem ser modificadas (escritas ou armazenadas) pelo comando.

Note que estes comandos incluem todas as posições de memória que podem ser buscadas ou modificadas; os conjuntos podem ser grandes.

Assumindo dois comando S_1 e S_2, mostramos que a interseção destes conjuntos testam as dependências de dados, conforme a condição de *Bernstain*:

- $OUT(S_1) \cap IN(S_2) \neq \emptyset$ $\quad S_1\ \delta^f\ S_2$ $\quad$ *Flow*-dependência
- $IN(S_1) \cap OUT(S_2) \neq \emptyset$ $\quad S_1\ \delta^a\ S_2$ $\quad$ *Anti*-dependência
- $OUT(S_1) \cap OUT(S_2) \neq \emptyset$ $\quad S_1\ \delta^o\ S_2$ $\quad$ *Output*-dependência

Que informação é obtida quando a interseção de dois conjuntos IN não é vazio: $IN(S_1) \cap IN(S_2) \neq \emptyset$? Isto significa que S_1 lê a mesma posição de memória antes que S_2 leia esta posição (ou mais precisamente, podem ler a mesma posição). Esta relação é chamada *Input*-dependência, e é escrita $S_1\ \delta^i\ S_2$; usualmente isto é tratado como uma relação não dirigida, de acordo com Wolfe [1996].

4.2. DEFINIÇÕES

DEFINIÇÃO: Dois comandos S e T, onde S é executado antes de T na ordem seqüencial para a execução (S < T). Então

- T é **dado dependente** de S, o qual denotaremos por $S \rightarrow T$, se a variável de saída para S é a variável de entrada para T, ou seja, existe uma **dependência direta** de S para T (S **dd** T), de acordo com Mokarzel *et al.* [1988] e Saltz *et al.* [1991];

O laço abaixo tem a relação ***flow-dependente*** do comando S_1 para si mesmo, desde que o valor atribuído para A(i + 1) seja usado na próxima iteração do laço, e denotamos por $S_1\ \delta^f\ S_1$, segundo Wolfe *et al.* [1992] e Wolfe [1992].

```
        FOR i = 1 to (N-1) do
S1:             A( i+1 ) = A( i  ) + B( i )
        END_FOR
```

- T é **antidependente** de S, a qual denotaremos por $S \dashrightarrow T$, se a variável de saída para T é a variável de entrada para S, ou seja, existe uma antidependência de S para T (S **da** T), de acordo com Mokarzel *et al.* [1988], Saltz *et al.*[1991], Lu *et al.*[1992], Lee [1995].

No laço abaixo, existe uma relação ***anti-dependente*** de S_1 para S_2, apenas B(i, j+1) é usado em S_1 e subseqüentemente re-atribuido em S_2 na próxima iteração do laço j, e é denotado por $S_1\ \delta^a\ S_1$, segundo Wolfe *et al.* [1992] e Wolfe [1992].

```
        FOR i = 1 to N do
            FOR j = 1 to M-1 do
S1:             A( i , j ) = B( i , j+1 ) + 1
S2:             B( i , j) = C( i ) - 1
            END_FOR
        END_FOR
```

- T é **saída dependente** de S, a qual denotaremos por $S \xrightarrow{o} T$, se T e S tem variáveis de saída iguais; ou seja, existe uma dependência de saída de S para T (S **ds** T), de acordo com Hilhorst *et al.* [1987], Mokarzel *et al.* [1988], Saltz *et al.* [1991], Lu *et al.* [1992], e Lee [1995].

Um exemplo disto é mostrado abaixo, onde existe uma relação ***output dependente*** de S_2 para S_1, apenas a variável B(i+1) é atribuída em S_2 e pode ser re-atribuida na próxima iteração para o laço por S_1, e é denotado por $S_2\ \delta^o\ S_1$, segundo Wolfe *et al.* [1992] e Wolfe [1992].

```
        FOR i = 1 to N-1 do
S1:         if (A( i ) > 0) B( i ) = C( i ) / A( i )
S2:         B( i + 1) = C( i ) / 2
        END_FOR
```

Existe uma **dependência condicional** de **b** para **c** (b **dc** c) se **b** for o cabeçalho de um comando condicional e **c** for um comando no escopo de **b**. Finalmente, se **d** é o cabeçalho de um comando repetitivo e **c** é um comando em seu escopo, então existe uma **dependência de repetição** de **d** para **c** (d **dr** c), de acordo com Mokarzel *et al.* [1988].

As dependências no escopo de comandos repetitivos são munidas de uma direção e uma distância.

No laço seguinte

```
        DO  i = 1, n
S1:         a(i) = b(i) + 5
S2:         c(i) = a(i) * 2
        END_DO
```

a dependência (S_1 **dd** S_2) é interna à iteração i. Diz-se que sua direção é do tipo "=" e sua distância é zero.

Já no laço seguinte

```
        DO  i = 1, n
S1:             a(i) = b(i) + 5
S2:             c(i) = a(i-3) * 2
        END_DO
```

a dependência (S_1 **dd** S_2) vai da iteração i para a iteração (i-3). Diz-se então que sua direção é do tipo "<" e sua distância é 3. Também existem direções do tipo ">" e distâncias negativas, como veremos a seguir.

Em aninhamentos de comandos repetitivos, é possível que existam mais do que uma direção e uma distância para cada dependência, devido aos múltiplos laços. Ordenando-se as direções e as distâncias de acordo com a sequência de comandos repetitivos, constroem-se vetores chamados de vetor direção e vetor distância.

Por exemplo, no aninhamento seguinte

```
        DO  i = 1, n
          DO  j = 1, n
S1:             a(i,j) = b(i,j) + c(i,j)
S2:             d(i,j) = a(i-2,j+1) + c(i,j)
          END_DO
        END_DO
```

a dependência (S_1 **dd** S_2) tem o vetor direção (<,>) e o vetor distância (2,-1).

4.3. Distância e Vetor Distância

Para aplicar uma ampla variedade de transformações, as relações de dependência de dados são anotadas com informações, mostrando como elas são afetadas pelos laços. Muitas das relações de dependência tem uma distância constante em cada dimensão do espaço de iteração, conforme Wolfe [1992].

Quando este for o caso, o vetor distância pode ser construído, onde cada elemento é uma constante inteira, representando as distâncias das dependências no laço correspondente. Por exemplo, no laço seguinte, há uma relação de dependência de dados no espaço de iteração como

mostrado; onde cada iteração (i,j) depende dos valores computados na iteração (i,j-1). As distâncias para esta relação de dependência são zero no laço i e um no laço j, de denotada por $S_1\delta_{(0,1)}S_1$, de acordo com Wolfe [1992].

```
        FOR i = 1 to N do
            FOR j = 2 to M do
S1:             A( i , j ) = A( i , j-1 ) + B( i , j )
            END_FOR
        END_FOR
```

Cada vetor distância terá **n** entradas, onde **n** é o nível de aninhamento. Como as distâncias das dependências são usualmente pequenas, palavras curtas ou bytes sinalizados podem ser utilizados, para armazená-las, de acordo com Wolfe [1992].

Para algumas transformações, a magnitude da distância não é necessária, desde que a direção seja conhecida; freqüentemente a distância não é constante no laço, muito embora possam sempre ser positivas (ou sempre negativas). Como exemplo, no laço:

```
        FOR i = 1 to N do
            FOR j = 1 to N do
S1:             X( i+1 , 2*j ) = X( i , j ) + B( i )
            END_FOR
        END_FOR
```

a referência X(i+1,2*j) é usada em algumas iterações dos laços i e j pela referência X(i,j). Algumas das relações de dependência para este laço são mostradas na tabela 4.1 abaixo:

Elemento	de		para		Distância da Dependência
	i	j	i	j	
X(2 , 2)	1	1	2	2	(1 , 1)
X(3 , 4)	2	2	3	4	(1 , 2)

Tabela 4.1. Relações de Dependência do Laço anterior

As distâncias no laço j são positivas, mas não são constantes. Uma maneira de representar isto é, armazenar um vetor de sinais das distâncias das dependências, chamado de vetor direção, segundo Zhiyan *et al.* [1990], Wolfe *et al.* [1992], e Wolfe [1992].

Cada elemento do vetor distância será um dos símbolos { +,0,- }; por razões históricas são usualmente escritos { <,=,> }.

CAPÍTULO 5 - ELIMINAÇÃO DE DEPENDÊNCIAS

Neste capítulo são descritas algumas técnicas, utilizadas para eliminar as dependências mais comuns.

5.1. RENOMEAÇÃO

Algumas vezes, o uso de posições de memória iguais para diferentes propósitos, impõem restrições desnecessárias no arranjo de programas paralelos.

A renomeação atribui diferentes nomes para usos diferentes de variáveis iguais. Como uma conseqüência, as relações de dependências iguais e antidependencias podem ser removidas, segundo Hilhorst *et al.* [1987].

Por exemplo:

S: A = B + C

T: D = A + E

U: A = A + D

é transformado em

S': A1 = B + C

T': D = A1 + E

U': A = A1 + D

5.2. SUBSTITUIÇÃO A FRENTE

Uma eliminação da dependência de dados, por Substituição a Frente, substitui o lado direito de um comando de atribuição dentro do lado direito de outros comandos de atribuição, de acordo com Hilhorst *et al.* [1987] e Blume *et al.* [1992].

Por exemplo:

S: A = B + C

T: D = A + E

U: A = A + D

é transformado em

```
T':     D = B + C + E
U':     A = B + C + B + C + E
```

5.3. EXPANSÃO ESCALAR

Uma variável usada dentro de um laço DO, é alterada para um elemento de um vetor dimensional superior, segundo Hilhorst *et al.* [1987], Blume *et al.* [1992] e Eigenmann *et al.* [1992].

Por exemplo:

```
DO I = 1, N
S:      X = C( I )
T:      D( I ) = X + 1
END_DO
```

é transformado em

```
DO I = 1, N
S':     X( I ) = C( I )
T':     D( I ) = X( I ) + 1
END_DO
```

5.4. DISTRIBUIÇÃO DO LAÇO

A técnica de distribuição do laço consiste em decompor laços DO em outros mais simples; com isso será possível paralelizar de imediato alguns deles.

Segundo Hilhorst *et al.* [1987], o algoritmo para um determinado laço é:

1. Analisar as dependências;
2. Estabelecer uma partição Π dos comandos $\{S_1, S_2, \ldots\}$; dois comandos S_i e S_j são semelhantes dentro de um subconjunto, chamado Π-bloco, se as relações de dependência constituem um ciclo, isto é, se ali existir $S_{K_1}, \ldots, S_{K_n}, S_{l_1}, \ldots, S_{l_n}$, tal que $S_i \rightarrow S_{k_1} \rightarrow \ldots \rightarrow S_{k_n} \rightarrow S_j \rightarrow S_{l_1} \rightarrow \ldots \rightarrow S_{l_n} \rightarrow S_i$;

3. Estabelecer a ordem parcial em Π; os blocos serão seqüencializados, tal que a origem de uma seta, seja executada antes que o final da seta;
4. Trocar o laço original por laços para $\Pi 1, \Pi 2, \ldots$, de acordo com a ordem definida previamente.

Por exemplo:

```
        DO  I = 1, N
S1:             A1(I-1) = A2(I-1) + 1
S2:             Y( I ) = A1(I-1) ** 2
S3:             A2( I ) = X(I+1)
S4:             X( I ) = W(I-1) + 1
S5:             W(I+1) = X(I-1) + 1
        END_DO
```

é transformado em

```
        DO  I = 1, N
S3:             A2( I ) = X(I+1)
        END_DO
        DO  I = 1, N
S1:             A1(I-1) = A2(I-1) + 1
        END_DO
        DO  I = 1, N
S2:             Y( I ) = A1(I-1) ** 2
        END_DO
        DO  I = 1, N
S4:             X( I ) = W(I-1) + 1
S5:             W(I+1) = X(I-1) + 1
        END_DO
```

Os comandos dentro do laço 1 podem ser executados paralelamente para todos os valores do índice do laço; da mesma forma, os laços 2 e 3 podem ser paralelizados, de acordo com Hilhorst *et al.* [1987]

CAPÍTULO 6 - MÉTODO HIPERPLANO

Neste capítulo é descrito o primeiro dos métodos estudados, o Método Hiperplano. É demonstrado o seu uso para processamento paralelo e concorrente por meio de um exemplo.

6.1. INTRODUÇÃO

Este método permite a paralelização parcial de aninhamentos com múltiplos laços, segundo Hilhorst *et al.* [1987]. Resulta um aninhamento onde o laço externo é seqüencial e os demais paralelos. Com isso, o aninhamento ficará reduzido para a seguinte forma, de acordo com Darte *et al.* [1994]

```
FOR time = time_min to time_max do
    FOR ρ ∈ E(time) do in parallel
        P(ρ)
    END_FOR
END_FOR
```

O laço externo avança passo a passo no tempo. Em cada passo, E(*time*) é o conjunto dos pontos do espaço de índices que podem ser computados simultaneamente. O laço interno representa um aninhamento que pode ser executado em paralelo ou concorrentemente.

Considerando o exemplo

```
DO I = 1, L
    DO J = 1, M
        DO K = 1, N
            A(J,K) = (A(J+1,K) + A(J,K+1) + A(J-1,K) + A(J,K-1)) / 4
        END_DO
    END_DO
END_DO
```

Após a aplicação do método, o aninhamento (para execução concorrente) ficará

```
DO I' = 4, (2 * L + M + N)
  DO CONC FOR ALL (J',K') ∈ {(j,k): 1 ≤ j ≤ L, 1 ≤ k ≤ N, 1 ≤ I' - 2 * j - k ≤ M }

      A(I'- 2 * J' - K', K') = (A(I' - 2 * J' - K'+ 1), K') + A(I' - 2 * J' - K', K' + 1) +
                               + A(I' - 2 * J' - K' - 1, K') + A(I' - 2 * J' - K', K' - 1)) / 4
  END_DO
END_DO
```

6.2. NOTAÇÃO

Primeiramente, definimos os conceitos de referência geradora e referência usuária. Uma referência é dita *geradora* quando ocorre do lado esquerdo de uma atribuição, isto é, quando gera um novo valor para uma variável. Uma referência é dita *usuária* quando ocorre do lado direito de uma atribuição, ou seja, quando utiliza, sem modificar, o valor de uma variável.

Adotaremos a notação $\mathbf{r}_i$, onde i >= 1, para representar as referências a uma variável.

Por exemplo, na atribuição

$$A(J,K) = (A(J+1,K) + A(J,K+1) + A(J-1,K) + A(J,K-1)) / 4 \qquad (6.1)$$

$\mathbf{r}_1$ $\mathbf{r}_2$ $\mathbf{r}_3$ $\mathbf{r}_4$ $\mathbf{r}_5$

$\mathbf{r}_1$ é uma referência *geradora* e $\mathbf{r}_2,\ldots,\mathbf{r}_5$ são referências *usuárias*.

Consideraremos aninhamentos na forma

```
DO  I^1 = l^1, u^1
. . .
    DO  I^N = l^N, u^N
        { CORPO do ANINHAMENTO }          (6.2)
    END_DO
. . .
END_DO
```

onde os laços são enumerados em ordem crescente, do mais externo para o mais interno, e essa enumeração sobreescreve o índice do laço, que é representado por I. Os limites do laço, representados por l e u, são constantes inteiras.

Segundo Hilhorst *et al* [1987], o corpo do aninhamento tem as seguintes restrições:

(i) Não contém comandos de E/S (*Entrada/Saída*);

(ii) Não contém transferências de controle para comando(s) fora do aninhamento;

(iii) Não contém invocações de sub-rotinas ou funções que possam modificar o valor das variáveis;

(iv) Cada referência de uma variável V no corpo do aninhamento, ou é da forma $V(I^1+m^1, I^2+m^2, \ldots, I^N+m^N)$, onde $m^i, i=1,\ldots,N$ são constantes inteiras, ou é da forma $V(I^2+m^2, \ldots, I^N+m^N)$. No segundo caso, I^1 é um índice dito *ausente*.

Na primeira forma, temos uma referência onde todos os índices estão presentes. Com isso não temos o índice *ausente*, de modo que devemos adotar um índice adicional que seja *ausente*, com os limites superior e inferior iguais a "1".

Denominamos *vetor distância* de duas referências $r_1 = V(u, I^2+m^2, \ldots, I^N+m^N)$ *e* $r_2 = V(u, I^2+\tilde{m}^2, \ldots, I^N+\tilde{m}^N)$, e representamos por $<r_1, r_2>$ à tupla

$$<r_1, r_2> := (*, m^2-\tilde{m}^2, \ldots, m^N-\tilde{m}^N)$$

O asterisco (*) no vetor distância representa o índice *ausente*.

O vetor distância serve para representar a distância, em iterações no aninhamento, entre duas referências à mesma variável.

O vetor distância será calculado para os pares de referências à uma mesma variável, em que pelo menos uma das referências é geradora. Por exemplo, para a atribuição (6.1), temos os seguintes vetores distância:

Pares de Referências	Vetores distância
$<r_1,r_1>$	(*,0,0)
$<r_1,r_2>$	(*,-1,0)
$<r_2,r_1>$	(*,1,0)
$<r_1,r_3>$	(*,0,-1)
$<r_3,r_1>$	(*,0,1)
$<r_1,r_4>$	(*,1,0)
$<r_4,r_1>$	(*,-1,0)
$<r_1,r_5>$	(*,0,1)
$<r_5,r_1>$	(*,0,-1)

Tabela 6.1. Pares de referências e os vetores distância

Para facilitar a representação utilizamos o vetor de índices, onde usamos a notação $\vec{I} = \left(I^1, \ldots, I^N\right)$.

6.3. O MÉTODO

O princípio básico é transformar os índices do aninhamento para uma nova forma, a qual permite que as iterações do aninhamento sejam executadas simultaneamente. E, para garantir a mesma semântica do laço original, basta que a transformação preserve as relações de dependência entre as referências.

Primeiramente, é preciso fazer um mapeamento um-para-um dos índices antigos para os índices novos, $J{:}Z^N \rightarrow Z^N$, da forma $J\left[\left(I^1, \ldots, I^N\right)\right] = \left(\sum_{j=1}^{N} a_j^1 I^j, \ldots, \sum_{j=1}^{N} a_j^N I^j\right) = \left(J^1, \ldots, J^N\right)$, onde $a_j^i \in N$ para todo ($1 \leq i, j \leq N$), o qual transforma o aninhamento (6.2) no aninhamento (6.3).

Este mapeamento produzirá o índice J^1 seqüencial, que é uma combinação linear dos antigos índices. É o responsável pela sequência de execuções, pois determina o deslocamento da linha imaginária que corta o espaço de iteração, ou seja, pela frente de onda que faz a varredura deste mesmo espaço, determinando quais conjuntos de iterações que podem ser executados simultaneamente. E denotamos por $\Pi{:}Z^N \rightarrow Z$ este mapeamento $\Pi\left[\left(I^1, \ldots, I^N\right)\right] = J^1$, que é a renomeação dos antigos índices (I^1 a I^N) para J^1 a J^N.

Os laços que tem os índices J^2 a J^N, são paralelos, determinam as iterações que podem ser executadas em paralelo, dentro do conjunto finito definido pelo índice J^1.

```
DO  J^1 = λ^1, μ^1
  DO  CONC FOR ALL (J^2, ..., J^N) ∈ S_J1
        { CORPO do ANINHAMENTO }                    (6.3)
  END_DO
END_DO
```

S_{J^1} é um subconjunto de Z^{N-1}, o qual depende de J^1.

O corpo do aninhamento (6.3) é sucessivamente executado nos hiperplanos $J^1 = \Pi\left(\vec{I}\right)$.

A execução do corpo do aninhamento para $\vec{i}_1$ antecede o para $\vec{i}_2$ na nova ordem de execução se $\Pi(\vec{i}_1) < \Pi(\vec{i}_2)$.

A proposta é construir um aninhamento no padrão do aninhamento (6.3). Primeiro buscamos as condições sobre Π, as quais garantem que o aninhamento (6.3) gere um algoritmo paralelo equivalente ao aninhamento (6.2). Desta forma, para que a ordem de execução seja preservada no aninhamento (6.3), basta impor que as relações de dependências do aninhamento (6.2) sejam obedecidas.

Sendo r_1 e r_2 referências de V, tal que pelo menos uma delas é uma *geradora* e sejam dadas por

$$r_1 : V(I^2 + m^2, \ldots, I^N + m^N)$$

$$r_2 : V(I^2 + \tilde{m}^2, \ldots, I^N + \tilde{m}^N)$$

para a referência r_1 temos o vetor de índices $\vec{i}_1 \in S := \{(*, I^2 - m^2, \ldots, I^N - m^N)\}$ e para r_2 o vetor $\vec{i}_2 \in \tilde{S} := \{(*, I^2 - \tilde{m}^2, \ldots, I^N - \tilde{m}^N)\}$, onde * denomina valores inteiros, então existe uma relação de dependência entre o vetor $\vec{i}_1 \in S$ e o vetor $\vec{i}_2 \in \tilde{S}$, se $\vec{i}_1 < \vec{i}_2$.

Como Π é uma transformação linear, a imposição $\Pi(\vec{i}_1) < \Pi(\vec{i}_2)$ é equivalente a $\Pi(\vec{i}_2 - \vec{i}_1) > 0$, ou seja $\Pi(*, m^2 - \tilde{m}^2, \ldots, m^N - \tilde{m}^N) > 0$.

LEMA 6.1. Seja Π um mapeamento linear. Se, para todas as variáveis V e todos os pares de referências r_1 e r_2, onde pelo menos uma das referências é *geradora* e para todo $X \in < r_1, r_2 >$ com $X > 0$, o mapeamento Π satisfaz $\Pi(X) > 0$, então o algoritmo do aninhamento (6.3) é equivalente ao do aninhamento (6.2) [Hilhorst *et al* 1987].

TEOREMA 6.1. (Teorema do Hiperplano Concorrente)

Assumimos que nenhum dos índices $I^2, \ldots, I^N$ são *ausentes*. Então pode-se re-escrever o aninhamento (6.2) para a forma do aninhamento (6.3). Além disso, o mapeamento J é usado para re-escrever os novos índices do aninhamento de (6.2), de acordo com Hilhorst *et al.* [1987].

O Lema 6.1 permite obtermos as condições para a construção do mapeamento Π com o qual faremos o mapeamento J de um-para-um como segue.

$$J^1 = \Pi\left[\left(I^1,\ldots,I^N\right)\right] = a_1 I^1 + \ldots + a_{k-1} I^{k-1} + I^k$$

$$J^2 = I^1$$

$$\vdots$$

$$J^k = I^{k-1}$$

$$J^{k+1} = I^{k+1}$$

$$\vdots$$

$$J^N = I^N$$

Observe que isto é uma das possibilidades para se definir $J^2,\ldots,J^N$.

6.4. APLICAÇÃO DO MÉTODO

Aplicaremos o método para o aninhamento exemplo, onde a referência r_i, $i = 1,\ldots,5$ é definida como em (6.1). Primeiramente calculamos os vetores distância, para em seguida obter as condições necessárias para a determinação dos coeficientes e aplicar no mapeamento Π.

Para o cálculo dos vetores distância, determinamos que uma das distâncias será sempre (*), o qual denota o índice *ausente*. Esta distância assumirá os valores 0 e 1, desta forma, gerando várias combinações que poderão se repetir e então, serão desconsideradas nas desigualdades.

Os vetores distância cujos elementos forem todos negativos, também serão desconsiderados.

Do conjunto dos vetores distância positivos restantes, são obtidas as desigualdades, que irão compor o sistema para a obtenção dos coeficientes $a_1 \ldots a_n$.

Pares de Referências	Elementos Positivos	Condição
$<r_1,r_1>$ = (*,0,0)	(1,0,0)	$a_1 > 0$
$<r_1,r_2>$ = (*,-1,0)	(1,-1,0)	$a_1 - a_2 > 0$
$<r_2,r_1>$ = (*,1,0)	(1,1,0)	$a_1 + a_2 > 0$
	(0,1,0)	$a_2 > 0$
$<r_1,r_3>$ = (*,0,-1)	(1,0,-1)	$a_1 - a_3 > 0$
$<r_3,r_1>$ = (*,0,1)	(1,0,1)	$a_1 + a_3 > 0$
	(0,0,1)	$a_3 > 0$
$<r_1,r_4>$ = (*,1,0)	= <r2,r1>	
$<r_4,r_1>$ = (*,-1,0)	= <r1,r2>	
$<r_1,r_5>$ = (*,0,1)	= <r3,r1>	
$<r_5,r_1>$ = (*,0,-1)	= <r1,r3>	

Tabela 6.2. Mapeamento para o aninhamento 6.1

O asterisco (*) na tabela 6.2 representa o índice *ausente*, e assume valores 0 e 1.

As condições devem ser satisfeitas ao mesmo tempo. Em virtude disto, as condições $a_1 + a_2 > 0$ e $a_1 + a_3 > 0$ podem ser desconsideradas porque elas são combinações lineares das condições, $a_1 > 0$, $a_2 > 0$ e $a_3 > 0$, desta forma, redundantes.

Deste modo, temos como resultado um sistema composto por 5 desigualdades e 3 coeficientes:

$$a_1 > 0$$
$$a_2 > 0$$
$$a_3 > 0$$
$$a_1 - a_2 > 0 \therefore a_1 > a_2$$
$$a_1 - a_3 > 0 \therefore a_1 > a_3$$

Arbitrando os seguintes valores para os coeficientes:

$$a_1 = 2$$

$$a_2 = 1$$

$$a_3 = 1$$

os quais geram uma solução particular para este sistema, mas também poderiam ter sido arbitrados outros valores , desde que satisfizessem as mesmas condições.

De posse dos coeficientes, basta aplicá-los no mapeamento $\prod$ para obter os novos índices do aninhamento e a respectiva mudança dos índices na atribuição.

Olhamos para o mapeamento Π da forma

$$\Pi\left[\left(I^1, I^2, I^3\right)\right] = a_1 I^1 + a_2 I^2 + a_3 I^3, a_1, a_2, a_3 \in \mathrm{N}.$$

Com isso, obtemos

I' = Π(I,J,K) = 2I + J + K

J' = I

K' = J

(uma possibilidade também, é de escolher K' = K, desde que o coeficiente de J seja igual a 1)

O que resulta em:

```
DO  I' = 4, (2 * L + M + N)
  DO  CONC  FOR  ALL (J',K') ∈ {(j,k): 1 ≤ j ≤ L, 1 ≤ k ≤ N, 1 ≤ I' - 2 * j - k ≤ M }

      A(I'- 2 * J' - K', K') = (A(I' - 2 * J' - K'+ 1), K') + A(I' - 2 * J' - K', K' + 1) +
                                + A(I' - 2 * J' - K' - 1, K') + A(I' - 2 * J' - K', K' - 1)) / 4
  END_DO
END_DO
```

6.5. EXECUÇÃO SIMULTÂNEA

Mostramos agora um melhoramento do método Hiperplano, para o caso de execução simultânea. A idéia é transformar o aninhamento (6.2) para a forma do aninhamento (6.4).

```
DO  K^1 = v^1, n^1
  DO SIM FOR ALL (K^2,...,K^N) ∈ S_K^1
      { CORPO do ANINHAMENTO }                    (6.4)
  END_DO
END_DO
```

S_{K^1} é um subconjunto de Z^{N-1}, o qual depende de K^1.

LEMA 6.2: Seja Π um mapeamento linear. Se, para todas as variáveis V e todos os pares de referências r_1 e r_2, onde pelo menos uma das referências é *geradora* e para todo $X \in < r_1, r_2 >$ com $X > 0$, segundo Hilhorst *et al.* [1987] teremos:

(i) $\Pi(X) \geq 0$

(ii) Se $\Pi(X) = 0$, então $r_1 \rightarrow r_2$,

então o aninhamento (6.4) é equivalente ao aninhamento (6.2).

Na construção do mapeamento Π, temos que

$\vartheta_j^+ = \{ X \in \vartheta_j : X \in < r_r, r_2 >$ para alguns r_1, r_2 cada qual $r_1 \rightarrow r_2 \}$

e os Lemas 6.1 e 6.2 informam as condições para a escolha dos a_j como números inteiros não-negativos, tal que

$$a_j x^j + \ldots + a_N x^N \geq 0 \quad \text{se} \quad \left(0,\ldots,0,x^j,\ldots,x^N\right) \in \vartheta_j$$

e

$$a_j x^j + \ldots + a_N x^N > 0 \quad \text{se} \quad \left(0,\ldots,0,x^j,\ldots,x^N\right) \in \vartheta_j^+ .$$

Se paralelizarmos o aninhamento (6.1) no caso da execução simultânea, obtemos o resultado de forma como no caso da execução concorrente. Se trocarmos (6.1) por

$$\underset{\mathbf{r_1}}{A(J,K)} = (\underset{\mathbf{r_2}}{A(J+1,K)} + \underset{\mathbf{r_3}}{A(J,K+1)} + \underset{\mathbf{r_4}}{A(J-1,K)} + \underset{\mathbf{r_5}}{A(J,K+1)} + \underset{\mathbf{r_6}}{A(J+1,K-1)})/5.$$

no aninhamento, então temos que

$$\langle r_6, r_1 \rangle = (1,-1) > 0$$

e

$$r_6 \rightarrow r_1.$$

Para a execução concorrente, impusemos que $a_1 - a_2 > 0$ e assim $a_1 = 2$, $a_2 = 1$, enquanto que, para a execução simultânea, é suficiente termos $a_1 - a_2 \geq 0$, e com isso $a_1 = a_2 = 1$, como mostrado na figura 6.1.

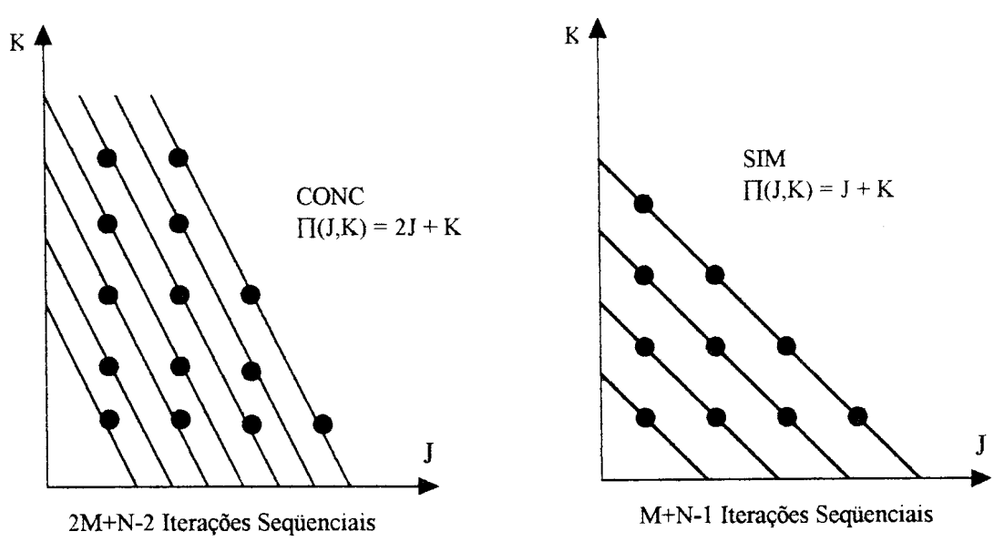


Figura 6.1. Execução Concorrente e Simultânea

CAPÍTULO 7 - MÉTODO TRANSFORMAÇÕES UNIMODULARES

Neste capítulo, descrevemos o Método Transformações Unimodulares, utilizado na paralelização de laços aninhados.

7.1. INTRODUÇÃO

Este método, utiliza as transformações unimodulares (*Permutation*, *Reversal*, e *Skewing*) para poder realizar a paralelização de laços, e permite tanto o paralelismo de granularidade grossa quanto fina.

7.2. NOTAÇÃO

7.2.1. REPRESENTAÇÃO DE VETOR DEPENDÊNCIA

Um vetor dependência, em um laço aninhado de profundidade **n**, é denotado por um vetor $\vec{d}$ = (d_1,d_2, ... ,d_n), e é lexicograficamente positivo, escrito $\vec{d} \succ \vec{0}$, se $\exists i$: ($d_i > 0$ e $\forall j < i$: $d_j \geq 0$).($\vec{0}$ denota um vetor zero, isto é, um vetor com todos os componentes iguais a 0.)

7.2.2. TRANSFORMAÇÕES UNIMODULARES

As transformações como tais *Permutation*, *Reversal*, e *Skewing* são úteis na paralelização, e podem ser modeladas como uma matriz elementar de transformação.

Segundo Wolf *et al.* [1991], existem três transformações elementares:

- ***Permutation***: Uma *permutation* σ em um laço aninhado transforma a iteração $(p_1,\ldots,p_n)$ para $(p_{\sigma_1},\ldots,p_{\sigma_2})$. Esta transformação pode ser expressa na forma matricial como I_σ, ou seja, a matriz identidade I de n x n com linhas permutadas por σ.
- ***Reversal***: O *Reversal* do i-ésimo laço é representado pela matriz identidade, mas com o i-ésimo elemento da diagonal igual a (-1) e os outros na diagonal iguais a (1). Por exemplo, a

matriz representando o *reversal* do laço mais externo dos laços aninhados de profundidade 2 é $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$.

- ***Skewing***: O *Skewing* do laço I_j por um fator inteiro f com o laço I_i, mapeia a iteração

$$(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_{j-1}, p_j, p_{j+1}, \ldots, p_n)$$

para

$$(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_{j-1}, p_j, fp_i, p_{j+1}, \ldots, p_n).$$

A matriz transformação T a qual produz o *Skewing* é a matriz identidade, mas com o elemento $t_{j,i}$ igual a f e os outros elementos iguais a zero.

Uma matriz unimodular tem três importantes propriedades. Primeira, é quadrada, significando que mapeamos um espaço de iteração n-dimensional para outro espaço de iteração n-dimensional. Segunda, tem todos os componentes como inteiros, e apenas mapeia vetores inteiros para vetores inteiros. Terceira, o valor absoluto do determinante é um.

Uma transformação composta pode ser sintetizada na forma de uma seqüência de transformações primitivas, e o efeito destas transformações é representado pelo produto das várias matrizes de transformação uma para cada transformação primitiva, de acordo com Wolf *et al.* [1991].

7.2.3. Validade da Transformação Unimodular

Dizemos que é válido aplicarmos a transformação para um laço aninhado, se o código transformado puder ser executado seqüencialmente, ou na ordem lexicográfica do espaço de iteração.

Definição 7.1: Uma transformação do laço é válida se os vetores dependência transformados forem todos lexicograficamente positivos.

Teorema 7.1: Sendo D o conjunto de distâncias vetoriais de um laço aninhado, então uma transformação unimodular T é válida, se e somente se $\forall \vec{d} \in D: T\vec{d} \succ \vec{0}$.

Considerando o seguinte exemplo:

FOR I_1 = 1 to N do

FOR I_2 = 1 to N do

$$a[I_1, I_2] = f(a[I_1, I_2], a[I_1 + 1, I_2 - 1]);$$

END_FOR

END_FOR

Este código tem a dependência (1,-1). A transformação *permutation*, é representada por

$$T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

é inválida, desde que T(1,-1) = (-1,1) seja lexicograficamente negativo. Entretanto, compondo a *permutation* com a *reversal*, representada pela transformação

$$T = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

é válida, desde que T'(1,-1) = (1,1) seja lexicograficamente positivo. Similarmente, o Teorema 7.1 também ajuda deduzir o conjunto de transformações compostas que são válidas. Por exemplo, se T é uma matriz triangular inferior com diagonais unitárias (matriz *skewing*), então um laço aninhado válido aparecerá somente depois da transformação.

7.3. PARALELIZAÇÃO DE LAÇOS COM VETOR DISTÂNCIA

Agora estudamos a aplicação da teoria de transformação de laço para a paralelização, utilizando o vetor distância.

O algoritmo consiste de dois passos: o primeiro transforma o laço original para a forma canônica, isto é, um laço permutável completamente; e então transforma os laços permutáveis completamente para explorar o paralelismo de granularidade grossa ou fina de acordo com a arquitetura destino.

7.3.1. FORMA CANÔNICA: ANINHAMENTOS PERMUTÁVEIS COMPLETAMENTE

Os laços com vetores distância tem uma propriedade especial, eles podem sempre ser transformados em um laço aninhado permutável completamente utilizando *skewing*, segundo Wolf *et al.* [1991]. Por exemplo, se um laço aninhado duplo tem dependências {(0,1),(1,-2),(1,-1)}, então o

skewing do laço interno por um fator de dois com os laços externos produz {(0,1), (1,0), (1,1)}. O teorema seguinte explana como um laço aninhado válido pode ser transformado em um permutável completamente.

Teorema 7.2: Sendo $L=\{I_1, \ldots, I_n\}$ um laço aninhado com vetores distância lexicograficamente positivos $\vec{d} \in D$. Os laços no aninhamento podem ser tornados permutáveis completamente pelo *skewing*.

Se $\forall \vec{d} \in D$ e $d_{i+1} \geq 0$, então o laço I_{i+1} é permutável com o laço I_i. Por outro lado, como todas as dependências são lexicograficamente positivas, então $d_{i+1} < 0$, e isto implica que um dos $d_1, \ldots, d_i$, dizemos d_j, é positivo.

Por isso o *skewing* do laço I_{i+1} com o I_j por um fator

$$f \geq \max_{\{\vec{d} \mid \vec{d} \in D \wedge d_j \neq 0\}} \left\lceil -d_{i+1} / d_j \right\rceil$$

faz a transformação do (i+1)-ésimo componente do vetor dependência não-negativo.

7.3.2. PARALELIZAÇÃO

As iterações de um laço podem ser executadas em paralelo, se e somente se não existirem dependências transportadas por esse laço. Cada laço é chamado de laço DO_ALL. Segundo Wolf *et al.* [1991], para maximizar o grau de paralelismo, basta transformar o laço aninhado de modo a maximizar o número de laços DO_ALL.

Teorema 7.3: Sendo $(I_1,\ldots,I_n)$ um laço aninhado com dependências lexicográficas positivas $\vec{d} \in D$. O laço I_i é paralelizável, se e somente se $\forall \vec{d} \in D$, $(d_1, \ldots, d_{i-1}) \succ \vec{0}$ ou $d_i = 0$.

Uma vez que os laços são transformados em permutáveis completamente, os passos para gerar um paralelismo DO_ALL são simples. Primeiro, mostramos que os laços na forma canônica podem ser transformados para dar um máximo grau de granularidade. E então mostramos como produzir ambas as granularidades de paralelismo, fina e grossa.

1) Granularidade fina de paralelismo: Um aninhamento de **n** laços permutáveis completamente pode ser transformado em um código contendo no máximo (**n**-1) graus de paralelismo. Quando as dependências não forem transportadas por estes **n** laços, o grau de paralelismo é **n**. Por outro lado, os (**n**-1) laços paralelos podem ser obtidos pelo *skewing* do laço mais interno, no laço permutável completamente, por cada um dos outros laços e movendo-se o laço interno para a posição mais externa, de acordo com Wolf *et al.* [1991].

Esta transformação, a qual chamamos de transformação *wavefront*, é representada pela seguinte matriz

$$\begin{bmatrix} 1 & 1 & \dots & 1 & 1 \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix}.$$

A transformação *wavefront* coloca o máximo de laços DO_ALL nos laços mais internos, maximizando o paralelismo de granularidade fina.

2) Granularidade grossa de paralelismo: A transformação *wavefront* produz o máximo grau de paralelismo, mas torna o laço mais externo seqüencial. Por exemplo, considerando o seguinte laço aninhado:

```
FOR I1 = 1 to N do
    FOR I2 = 1 to N do
        a[I1,I2] = f(a[I1-1,I2-1]);
    END_FOR
END_FOR
```

Este laço aninhado tem a dependência (1, 1), onde o laço mais externo é seqüencial e o laço mais interno é um DO_ALL. A transformação *wavefront* $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ não muda isto. Por outro lado, a transformação unimodular $\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$ mudará a dependência para (0, 1), tornando o laço externo um DO_ALL e o laço interno seqüencial.

Por escolher a matriz transformação T com primeira linha $\vec{t}_1$, tal que $\vec{t}_1.\vec{d} = 0$ para todos os vetores dependência $\vec{d}$, a transformação produz um DO_ALL mais externo. Em geral, se a

profundidade do laço aninhado é **n**, e a dimensionalidade do espaço gerado pelas dependências é $\mathbf{n_d}$, então é possível fazer as primeiras ($\mathbf{n}$ - $\mathbf{n_d}$) linhas, de uma matriz transformação T, pequenos sub-espaços ortogonais a S do espaço gerado pelas dependências. Isto produzirá o máximo número de laços DO_ALL mais externos dentro do aninhamento, segundo Wolf *et al.* [1991].

Uma outra maneira para fazer laços externos DO_ALL é simplesmente identificar laços I_i, tal que todos os d_i sejam zero.

7.4. PARALELIZAÇÃO DE LAÇOS COM DISTÂNCIAS E DIREÇÕES

Na presença de vetores direção, nem todos os laços podem ser tornados permutáveis completamente, e com isso reduz-se o grau de paralelismo.

O problema de paralelização é, desta forma, o de buscar o máximo número de laços paralelizáveis. O algoritmo de paralelização consiste de dois passos: o primeiro transforma os laços para a forma canônica, e o segundo o transforma para o paralelismo de granularidade fina ou grossa, de acordo com Wolf *et al.* [1991].

Teorema 7.4: Sendo D o conjunto de vetores dependência de uma computação. A transformação unimodular T é válida se $\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$.

7.4.1. EXEMPLO COM VETOR DIREÇÃO

Ilustramos o procedimento com o seguinte exemplo:

```
FOR I1 = 1 to N do
    FOR I2 = 1 to N do
        FOR I3 = 1 to N do
            (a[I1,I3], b[I1,I2,I3] = f(a[I1,I3], a[I1+1,I3-1], b[I1,I2,I3], b[I1,I2,I3+1]));
        END_FOR
    END_FOR
END_FOR
```

O corpo do laço acima é representado por um espaço de N x N x N iterações. As referências $a[I_1,I_3]$ e $a[I_1+1,I_3-1]$ geram a dependência (1,'±',-1). As referências $a[I_1,I_3]$ e $a[I_1,I_3]$ não geram a dependência (0,'±',0), a qual não é lexicograficamente positiva, mas geram a dependência (0,'+',0).

As referências $b[I_1,I_2,I_3]$ e $b[I_1,I_2,I_3]$ geram a dependência (0,0,0), a qual ignoramos, desde que esta não seja uma borda de iteração cruzada. E finalmente, as referências $b[I_1,I_2,I_3]$ e $b[I_1,I_2,I_3+1]$ geram a dependência (0,0,1). Os vetores dependência para este aninhamento são

$$D = \{(0,'+',0), (1,'\pm',-1), (0,0,1)\}.$$

Nenhum dos três laços do programa podem ser paralelizados como estão, contudo existe um grau de paralelismo, que pode ser explorado em qualquer um dos níveis de granularidade grossa ou fina, conforme Wolf *et al.* [1991].

Pela permutação dos laços I_2 e I_3, e o *skewing* do novo laço central com o laço externo por um fator de 1, gera o seguinte código na forma canônica:

```
FOR I'1 = 1 to N do
    FOR I'2 = I'1 + 1 to I'1 + N do
        FOR I'3 = 1 to N do
            (a[I'1,I'2 - I'1], b[I'1, I'3, I'2 - I'1] =
                f(a[I'1, I'2 - I'1], a[I'1+1, I'2 - I'1-1],
                b[I'1, I'3, I'2 - I'1], b[I'1, I'3, I'2 - I'1-1]));
        END_FOR
    END_FOR
END_FOR
```

A matriz transformação T e as dependências transformadas D' são

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

e

$$D' = \{(0,0,'+'), (1,0,'\pm'), (0,1,0)\}.$$

A transformação é válida, desde que as dependências resultantes sejam todas lexicograficamente positivas. O resultado é que os dois laços mais externos, serão permutáveis completamente, desde que o intercambiamento destes laços deixe as dependências lexicograficamente positivas. O último laço é permutável completamente por si mesmo, de acordo com Wolf *et al.* [1991].

O segundo laço permutável completamente será transformado, para prover um grau de paralelismo, pela aplicação da *wavefront*. A matriz transformação T para esta fase da transformação, e as dependências transformadas D" são

$$T' = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

e

$$D'' = \{(0,0,'+'), (1,1,'\pm'), (1,0,0)\}$$

e o código transformado é

```
FOR I"1 = 3 to 3N do
    DO_ALL I"2 = max(1,⌈(I"1 - N)/2⌉) to min(N,⌊(I"1 - 1)/2⌋) do
        FOR I"3 = 1 to N do
            (a[I"2,I"1 - 2I"2], b[I"2, I"3, I"1 - 2I"2] =
                f(a[I"2, I"1 - 2I"2], a[I"2+1, I"1 - 2I"2-1],
                  b[I"2, I"3, I"1 - 2I"2], b[I"2, I"3, I"1 - 2I"2-1]));
        END_FOR
    END_DO_ALL
END_FOR
```

Aplicando a transformação *wavefront* para todos os laços aninhados permutáveis completamente, produzimos um laço aninhado com o máximo grau de paralelismo.

Para implementar esta técnica, não há necessidade da geração do código para o passo intermediário. A transformação parte diretamente do código origem para o código final, onde a matriz transformação será:

$$T'' = T'T = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

7.4.2. FORMA CANÔNICA: ANINHAMENTOS PERMUTÁVEIS COMPLETAMENTE

Tentamos criar o máximo número possível de laços aninhados permutáveis completamente, apartir dos laços mais externos. Primeiro aplicamos as transformações unimodulares, para mover os laços sobre o laço aninhado permutável completamente mais externo, se as dependências assim o permitirem. Uma vez que os laços não podem mais serem postos no laço aninhado permutável

completamente mais externo, os laços restantes serão recursivamente postos em um aninhamento permutável completamente.

Para laços cujas dependências são vetores distância, a seção 7.3 mostra que seu paralelismo é facilmente acessível, uma vez que eles são realizados em um aninhamento permutável completamente. Similarmente, uma vez que um laço aninhado com vetores dependência é colocado em um aninhamento permutável completamente, o paralelismo é imediatamente extraído.

Segundo Wolf *et al.* [1991], esta forma canônica tem três importantes propriedades, que simplificam a computação, e que tornam o paralelismo facilmente explorável:

1) O grande aninhamento permutável completamente mais externo, é o único incluso nos laços, e é um super-conjunto de todos os possíveis laços aninhados permutáveis mais externos. Necessitamos considerar, apenas as combinações de laços que constituem o grande aninhamento permutável completamente mais externo.
2) Se uma transformação válida existe para um laço aninhado original, então também existirá uma transformação válida, que gerará o código com o grande laço aninhado permutável completamente possível mais externo, colocado mais externamente.
3) Um algoritmo que coloca o grande laço aninhado permutável completamente mais externo, e então recursivamente chama a si mesmo nos laços restantes para colocar laços adicionais, gerando o máximo grau de paralelismo possível, através das transformações unimodulares. O máximo grau de paralelismo no laço aninhado, é simplesmente a soma dos máximos graus de paralelismo de todos os aninhamentos permutáveis completamente na forma canônica.

7.4.3. Paralelismo de granularidade Fina e Grossa

Para obter a máxima granularidade fina de paralelismo, realizamos um *wavefront* em todos os aninhamentos permutáveis completamente, e permutamos, assim todos os laços DO_ALL que são mais internos, os quais são sempre válidos.

Seja S o sub-espaço ortogonal das dependências. Se o aninhamento é permutável completamente $d_k^{min} \geq 0$, então podemos assumir sem perda de generalidade, que $d_k^{max} = d_k^{min}$ ou $d_k^{max} = \infty$ pela enumeração de todas as faixas finitas dos componentes. Se alguns componentes de $\vec{d}$ estão sem limite, isto é $d_k^{max} = \infty$, então nossa exigência em $\vec{s} \in S$ que $\vec{d} = 0$, e isto implica que $s_k = 0$. Isto coloca as mesmas restrições em S como se necessita em um vetor distância

(0,...,0,1,0,...,0), com um simples 1 na k-ésima entrada. Assim, podemos calcular S pela busca do espaço nulo da matriz com linhas consistindo de (d_1^{min}, ..., d_n^{min}) e dos apropriados vetores (0,...,0,1,0,...,0).

Estas linhas são um super-conjunto de dependências dentro do laço aninhado permutável completamente, mas elas são todas lexicograficamente positivas e todos os seus elementos são não-negativos. Assim, podemos usar estas linhas como os vetores distância para o aninhamento e aplicamos a técnica da seção 7.3.2-2 para obter a transformação válida T para utilizarmos, fazendo a primeira linha $|S|$ da matriz distância transformação S, e assim sucessivamente, de acordo com Wolf *et al.* [1991].

7.5. ACHANDO OS LAÇOS PERMUTÁVEIS COMPLETAMENTE

Agora discutiremos o algoritmo para achar o grande laço aninhado permutável completamente mais externo, que é o passo chave na paralelização de um laço aninhado. O problema referenciado aqui como *cone time*, é o de não achar uma matriz transformação linear para um conjunto finito de vetores distância, tal que o primeiro componente de todos os vetores distância transformado seja positivo.

O algoritmo consiste em se usar uma técnica simples em todos os laços quanto for possível, e aplicar a técnica de *cone time* apenas para os laços restantes.

Segundo Wolf *et al.* [1991], classificamos os laços em três categorias:

1) Os laços *serializing* são os cujos componentes dependência são $+\infty$ e $-\infty$; estes laços não podem ser incluídos no aninhamento permutável completamente mais externo, e devem ser ignorados neste aninhamento.
2) Os laços que podem ser incluídos no aninhamento, através transformação SRP, que é uma eficiente transformação, que combina *skewing*, *reversal*, e *permutation*.
3) Os laços restantes que talvez possam ser incluídos no aninhamento, através uma transformação geral, usando a técnica de *cone time*, que será descrita na seção 7.5.3.

7.5.1. LAÇOS *SERIALIZING*

Como será mostrado abaixo, um laço com os componentes infinito positivo e negativo, não pode ser incluído no aninhamento permutável completamente mais externo. Consequentemente, este

laço irá em um aninhamento permutável completamente mais a frente, e não pode ser executado em paralelo com outros laços no aninhamento permutável completamente mais externo. Chamamos cada um destes laços de um laço *serializing*.

Teorema 7.5: Se o laço I_k tem dependências, tal que $\exists \vec{d} \in D: d_k^{min} = -\infty$ e $\exists \vec{d} \in D: d_k^{max} = \infty$, então o aninhamento permutável completamente mais externo, consiste apenas de uma combinação dos laços não incluindo I_k.

Em nosso exemplo na seção 7.4, o laço central é *serializing*, desta forma ele não pode estar no laço aninhado permutável completamente mais externo. Uma vez que os outros dois laços foram colocados mais externos, apenas a dependência (0,0,'+') apartir D' não foi tornada lexicograficamente positiva pelos primeiros dois laços. Para o próximo aninhamento permutável completamente, não mais teremos os laços *serializing*, pois serão removidos apartir desta consideração.

Quando um laço for removido, as dependências resultantes podem não ser necessariamente lexicograficamente positivas.

Por exemplo, supondo que um laço aninhado tenha as dependências

$$\{(1,'\pm',0,0), (0,1,2,-1), (0,1,-1,1)\}.$$

O segundo laço é *serializing*, com isso apenas necessitamos considerar a colocação no aninhamento mais externo dos laços: primeiro, terceiro e quarto. Se examinarmos imparcialmente as dependências para estes laços {(1,0,0), (0,2,-1), (0,-1,1)}, vemos que algumas delas tornam-se lexicograficamente negativas.

7.5.2. TRANSFORMAÇÃO SRP

Agora que removemos os laços *serializing*, aplicamos a transformação SRP. A SRP é uma extensão da transformação *skewing*, que usamos na seção 7.3, para gerar um aninhamento permutável completamente com distâncias lexicográficas positivas. O laço na seção 7.4 é um exemplo no qual foi aplicado esta técnica.

Consideramos agora a eficácia do *skewing* quando aplicamos em laços com vetores dependência. Primeiro, pode não ser uma transformação válida que torne os dois laços com

dependências permutáveis completamente. Por exemplo, nenhum fator de *skewing* pode tornar os componentes do vetor (1,'-') todos não-negativos. Além disso, o conjunto de vetores dependência sob esta consideração podem não ser sempre lexicograficamente positivos.

Por exemplo, supondo um subconjunto de dois laços em um aninhamento, que tem os seguintes vetores dependência {(1,-1), (-1,1)}. Estas dependências são anti-paralelas, pois nenhuma linha de uma matriz T pode ter um produto não-negativo em ambos os vetores, a não ser que a linha por si mesma seja um vetor zero, a qual gerará a matriz singular. No entanto, existem muitos casos para os quais uma simples transformação *reversal* e/ou *skewing* pode extrair o paralelismo, de acordo com Wolf *et al.* [1991].

Teorema 7.6: Sendo L = $\{I_1,\ldots,I_n\}$ um laço aninhado com dependências lexicograficamente positivas $\vec{d} \in D$, e $D^i = \{\vec{d} \in D \mid (d_1,\ldots,d_{i-1}) \not\succ \vec{0}\}$. O laço I_j pode ser transformado sobre um aninhado permutável completamente com laço I_i, onde i < j, através de *reversal* e/ou *skewing*, se

$$\forall \vec{d} \in D^i : \left(d_j^{\min} \neq -\infty \wedge \left(d_j^{\min} < 0 \rightarrow d_i^{\min} > 0\right)\right)$$

ou

$$\forall \vec{d} \in D^i : \left(d_j^{\max} \neq \infty \wedge \left(d_j^{\max} > 0 \rightarrow d_i^{\min} > 0\right)\right).$$

Segundo Wolf *et al.* [1991], a transformação SRP tem algumas propriedades importantes. Primeira, embora a transformação final seja constituída como uma série de combinações de *skewing*, *reversal*, e *permutation*, a transformação final pode ser expressa como uma transformação *permutation*, seguida por uma *reversal*, e por uma *skewing*. Isto é , podemos escrever a transformação T como T = SRP, onde P é uma matriz *permutation*, R é uma matriz *reversal*, e S é uma matriz *skewing* .

Segunda, o SRP converte um laço aninhado com apenas vetores distância lexicograficamente positivos em um laço aninhado permutável completamente simples. Assim, o SRP acha o máximo grau de paralelismo para laços nestes casos especiais. Se existe uma transformação SRP que gere vetores dependência lexicograficamente positivos, e nenhum dos vetores dependência transformados tem um componente negativo ilimitado $\left(\forall \vec{d} : d^{\min} \neq -\infty\right)$, então o algoritmo colocará todos os laços em um aninhamento permutável completamente.

7.5.3. TRANSFORMAÇÕES BI-DIMENSIONAIS GERAIS

Em geral, a transformação SRP sozinha pode não obter todos os níveis de paralelismo possíveis. Considerando novamente o exemplo de um laço com as dependências

$$D = \{(1,'\pm',0,0), (0,1,2,-1), (0,1,-1,1)\}.$$

O primeiro laço pode ser posto no aninhamento permutável completamente mais externo. O segundo laço é *serializing*, e não pode ser posto no laço aninhado permutável completamente mais externo. A questão é o terceiro e o quarto laço podem ser postos no aninhamento permutável completamente mais externo. Isto é, fazer com que uma transformação exista, e que torne todos os componentes de dependências {(1,0,0), (0,2,-1), (0,-1,1)} não-negativos depois da transformação. O primeiro laço pode ir no aninhamento permutável completamente; o problema é obter um T, tal que T(2,-1) e T(-1,1) sejam permutáveis completamente.

Resolvemos este problema da transformação, no caso especial, quando existirem exatamente dois laços no aninhamento, utilizando um algoritmo constituído de dois passos.

O primeiro é para achar uma direção $\vec{t}$ no espaço de iteração, tal que $\forall \vec{d} \in D: \vec{t}.\vec{d} > 0$. O segundo é para obter uma matriz unimodular T que contenha $\vec{t}$ na primeira linha, sendo que $\vec{t}$ é a direção do laço mais externo depois da transformação.

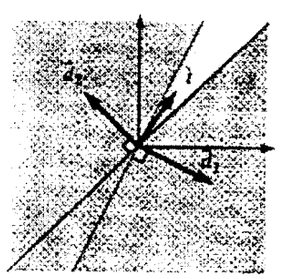


Figura 7.1. Um exemplo de um *cone-time* bi-dimensional

Para achar a direção $\vec{t}$, observamos que cada dependência limitará o espaço válido de $\vec{t}$ em um *cone* no espaço de iteração. Ilustramos esta técnica usando o exemplo de sub-vetores (2,-1) e (-1,1) na figura 7.1, onde cada dependência limita o $\vec{t}$ válido para um semi-plano que tem um produto não-negativo com $\vec{t}$. As interseções *times* válidas formam um *cone*, algumas das quais é um laço externo válido. Se o *cone* é vazio, então não há transformação possível que realize o objetivo, segundo Wolf *et al.* [1991].

Por outro lado, as bordas externas do *cone* podem ser somadas para produzir uma direção $\vec{t}$. Os componentes do vetor $\vec{t}$ são escolhidos, tal que seus **gcd** (*greatest common divisor*) sejam um. Em nosso exemplo, $(t_1,t_2) = (2,3)$. A técnica pode ser facilmente estendida para um número maior de distâncias e/ou vetores direção.

A matriz transformação T transformará as dependências de uma maneira desejável se

$$T = \begin{bmatrix} t_1 & t_2 \\ x & y \end{bmatrix}$$

e sendo T unimodular. Para obter isto, escolhemos x e y inteiros, tal que o determinante de T é a unidade (1), a qual ocorre quando $t_1y - t_2x = 1$. O par x e y formado por t_1 e t_2 é facilmente estabelecido usando o algoritmo *Extended Euclidean*.

Para nosso exemplo, o algoritmo *Euclid* produz x = 1 e y = 2, e pegamos

$$T = \begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix} \text{ e } T^{-1} = \begin{bmatrix} 2 & -3 \\ -1 & 2 \end{bmatrix}.$$

A iteração (I_1,I_2) é transformada pela T na iteração (I'_1,I'_2). Pela construção da transformação T, o componente I'_1 para todas as dependências é positivo, pois todas as dependências são lexicograficamente positivas. Assim, podemos fazer o *skew* do laço I'_2 com o laço I'_1 pela aplicação de SRP, para tornar o aninhamento permutável completamente.

7.6. IMPLEMENTANDO AS TRANSFORMAÇÕES

7.6.1. UTILIZANDO AS TRANSFORMAÇÕES UNIMODULARES

Supondo o seguinte laço aninhado

```
FOR I1 = ...
    ⋮
    FOR In = ...
        S(I1, ..., In);
    END_FOR
    ⋮
END_FOR
```

para o qual aplicamos a transformação unimodular T. A transformação de S requer apenas que I_j seja substituído pela combinação linear apropriada de I' 's, onde os I' 's são os índices para o laço

aninhado transformado, ou seja, $\begin{bmatrix} I_1 \\ \vdots \\ I_n \end{bmatrix} = T^{-1} \begin{bmatrix} I'_1 \\ \vdots \\ I'_n \end{bmatrix}$, e esta substituição é toda a necessária para o corpo do laço.

Agora discutiremos a transformação dos limites do laço.

1) Faixa dos limites do laço: A técnica de determinação dos novos limites do laço depois da transformação unimodular, requer que os limites do laço sejam da forma

$$\text{FOR } I_i = \max(L^1_i, L^2_i, \ldots) \text{ to } \min(U^1_i, U^2_i, \ldots) \text{ do}$$

onde

$$L^j_i = \lceil (l^j_{i,0} + l^j_{i,1} I_1 + \ldots + l^j_{i,i-1} I_{i-1}) / l^j_{i,i} \rceil$$

e

$$U^j_i = \lfloor (u^j_{i,0} + u^j_{i,1} I_1 + \ldots + u^j_{i,i-1} I_{i-1}) / u^j_{i,i} \rfloor$$

e todos $l^j_{i,k}$ e $u^j_{i,k}$ são constantes conhecidas, exceto para $l^j_{i,0}$ e $u^j_{i,0}$, os quais serão ainda variáveis no laço aninhado. (Se um limite superior ocorre quando necessitamos de um inferior é uma simples questão de ajustar $l^j_{i,0}$ e $u^j_{i,0}$ e substituir o limite superior com o inferior, e da mesma forma, se um limite inferior ocorrer onde necessitamos de um superior)

2) Transformando os limites do laço: Determinamos os novos limites do laço aninhado depois da transformação unimodular utilizando a matriz unimodular T.

PASSO 1: Extraindo as desigualdades. O primeiro passo é a extração de todas as desigualdades apartir do laço aninhado. Usando a notação apartir Seção 7.6.1-1, os limites podem ser expressos como uma série de desigualdades da forma $I_i \geq L^j_i$ e $I_i \leq U^j_i$.

PASSO 2: Obter o máximo e o mínimo absoluto para cada laço. Este passo usa as desigualdades para obter os valores máximos e mínimos possíveis para cada laço. Isto pode ser facilmente realizado apartir do laço mais externo para o mais interno, pela substituição dos máximos e mínimos externos do laço corrente sobre a expressão dos limites do aninhamento. Isto é, como o limite inferior de I_i é $\max_j (L^j_i)$, o menor valor possível para I_i é o máximo dos pequenos valores possíveis de L^j_i.

Usamos I_i^{min} e $L_i^{j,min}$ para denotar o valor possível para I_i e L^j_i, e $I_k^{min_{i,j}}$ para denotar os outros I_k^{min} ou I_k^{min}, o que resulta em

$$I_i^{min} = \max_j(L_i^{j,min})$$

onde

$$L_i^{j,min} = \left\lceil \left(l_{i,0}^j + \sum_{k=1}^{i-1} l_{i,k}^j I_k^{min_{i,j}} \right) / l_{i,i}^j \right\rceil$$

e

$$I_k^{min_{i,j}} = \begin{cases} I_k^{min}, & sign(l_{i,k}^j) = sign(l_{i,i}^j) \\ I_k^{max}, & outros \end{cases}$$

Também existem expressões similares para $U_i^{j,min}$, $L_i^{j,max}$, e $U_i^{j,max}$.

PASSO 3: <u>Transformação dos índices</u>. Agora usamos $(I_1,..., I_n) = T^{-1}(I_1',..., I_n')$ para substituir os índices I_j 's nas desigualdades pelos I_j' 's. O máximo e o mínimo para um I_j' é facilmente calculado. Por exemplo, se $I_1' = I_1 - 2I_2$, então $I_1'^{max} = I_1^{max} - 2I_2^{min}$ e $I_1'^{min} = I_1^{min} - 2I_2^{max}$. Em geral, usamos $(I_1',..., I_n') = T(I_1,..., I_n)$ para expressar os I_j' 's em termos de I_j 's.

Se $I_j' = \Sigma a_j I_j$, então

$$I_k'^{min} = \sum_{k=1}^{n} a_j \begin{cases} I_j^{min}, & a_j > 0 \\ I_j^{max}, & outros \end{cases}$$

e igualmente para $I_k'^{max}$.

PASSO 4: <u>Cálculo dos novos limites do laço</u>: Os limites inferior e superior para o laço I_1' são $I_1'^{min}$ e $I_1'^{max}$. Por outro lado, para determinar os novos limites do laço I_i', primeiro re-escrevemos cada desigualdade apartir do **PASSO 3** contendo I_1', produzindo uma série de desigualdades da forma $I_i' \leq f(...)$ e $I_i' \geq f(...)$. Cada desigualdade, da forma $I_i' \leq f(...)$, contribui para o limite superior. Se existe mais de uma expressão, então o mínimo das expressões é o limite superior. Da mesma forma, cada desigualdade da forma $I_i' \geq f(...)$, contribui para o limite inferior.

Uma desigualdade da forma $I_i' \geq f(I_j')$ contribui para o limite inferior do laço índice I_i'. Se $j < i$, isto significa que o laço I_j' é o laço externo I_i', então a expressão não precisa ser alterada, desde que o limite do laço de I_i' possa ser uma função do laço externo I_j'. Se o laço I_j' é aninhado contendo I_i', então o limite para o laço I_i' pode não ser uma função do laço I_j'. Neste caso, trocamos I_i' pelos mínimos ou máximos de qualquer função (f) minimizada.

Um procedimento similar é aplicado para os limites superiores do laço.

CAPÍTULO 8 - MÉTODO DE ALOCAÇÃO DE DADOS SEM COMUNICAÇÃO

Neste capítulo é descrito o terceiro método estudado, o qual permite a paralelização de laços com ou sem a duplicação dos dados na memória local dos processadores.

8.1. INTRODUÇÃO

Este método permite a paralelização de laços com ou sem comunicação de dados entre os processadores que compõem o sistema.

8.2. NOTAÇÃO

Um laço aninhado é considerado com a seguinte forma:

```
FOR  I1 = 1 to u1 do
        FOR I2 = 1 to u2 do
                ⋮
                FOR In = 1 to un do
                        { Corpo do Laço }                ( L )
                END_FOR
                ⋮
        END_FOR
END_FOR
```

onde n é o número de laços e u_j são as expressões lineares dos limites dos índices $I_1, I_2, \ldots, I_{j-1}$, para $1<j\leq n$.

Os símbolos Z^n e R^n representam o conjuntos de n-tuplas de inteiros e o conjunto de n-tuplas de números reais, respectivamente. O espaço de iteração de um laço aninhado é um sub-conjunto de Z^n e é definido como $I^n = \{(I_1, I_2, \ldots, I_n) \mid 1 \leq I_j \leq u_j, \text{ para } 1 \leq j \leq n\}$. O vetor $\bar{i} = (i_1, i_2, \ldots, i_n)$ em I^n é representado como uma iteração de um laço aninhado. Na ordem lexicográfica das iterações a

iteração $\bar{i} = (i_1,i_2,\ldots,i_n)$ é executada antes que a iteração $\bar{i}' = (i'_1,i'_2,\ldots,i'_n)$ se $i_1=i'_1$, $i_2=i'_2$,..., i'_{j-1}, e $i_j < i'_j$, para $1 \leq j \leq n$.

Segundo Chen *et al.* [1994], a função linear h: $Z^n \rightarrow Z^d$ é definida como uma função-referência $h(I_1,\ldots,I_n) = (a_{1,1}I_1 + \ldots + a_{1,n}I_n , \ldots , a_{d,1}I_1 + \ldots + a_{d,n}I_n)$ e é representada pela matriz:

$$H = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{d,1} & \cdots & a_{d,n} \end{bmatrix}_{dxn}$$

onde $a_{i,j} \in Z$, para $1 \leq j \leq n$. No corpo do laço, um elemento vetor d-dimensional $A[h(i_1,i_2,\ldots,i_n)+ \bar{c}]$ poderá ser referenciado pela função-referência **h** em uma iteração $(i_1,i_2,\ldots,i_n)$ em I^n, onde $\bar{c}$ é conhecida como a constante vetor *offset* em Z^d. O espaço de dados do vetor A é um sub-conjunto de Z^d e é definido sobre o conjunto dos índices subscritos dos vetores.

Para o vetor A, todas as **s** referências $A[H_p\bar{i} + \bar{c}_p]$, para $1 \leq p \leq s$, são chamadas **referências geradas uniformente**, se $H_1=H_2=\ldots=H_s$, onde H_p é a função transformação linear apartir de Z^n para Z^d, onde $\bar{i} \in I^n$, e $\bar{c}_p$ é uma constante vetor *offset* em Z^d.

Os diferentes vetores podem ter diferentes funções-referência .

Exemplo 8.1: Considerando o seguinte laço aninhado L1,

```
FOR i = 1 to 4 do
    FOR j = 1 to 4 do
        S1 : A[2i, j] = C[i, j] * 7;
        S2 : B[j, i+1] = A[2i -2, j-1] + C[i-1, j-1];          (L1)
    END_FOR
END_FOR
```

Neste exemplo, o espaço de iteração é $I^2 = \{(i,j) \mid 1 \leq i,j \leq 4\}$. No laço L1, com três vetores A, B, e C, temos as seguintes funções-referência:

$$H_A = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}, H_B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \text{ e } H_C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Existe uma dependência de fluxo entre a referência A[2i, j] na atribuição S_1 e A[2i - 2,j-1] na atribuição S_2, com as diferenças de vetores *offset* (0,0) e (-2,-1), respectivamente. Para vetor o C, há apenas leitura pelo laço L1, e existe uma dependência de entrada entre a referência C[i, j] na atribuição S_1 e C[i-1, j-1] na atribuição S_2, com as diferenças de vetor *offset* (0,0) e (-1,-1),

respectivamente. A referência B[j, i+1] é gerada apenas na atribuição S_2, e o vetor *offset* é (0,1). O laço L1 tem assim referências geradas uniformemente nos vetores A, B, e C.

Definição 8.1: Se existir, em um laço aninhado L com referências geradas uniformemente, duas referências $A[H\bar{i} + \bar{c}_1]$ e $A[H\bar{i} + \bar{c}_2]$ para o vetor A, então o vetor $\bar{r} = \bar{c}_1 - \bar{c}_2$ é chamado de vetor-referência de dados do vetor A.

O vetor-referência de dados *($\bar{r}$)* representa o vetor diferença entre dois elementos $A[H\bar{i} + \bar{c}_1]$ e $A[H\bar{i} + \bar{c}_2]$, os quais são referenciados pela iteração $\bar{i}$. Observe que algumas das dependências de dados no laço L existem entre duas referências distintas, $A[H\bar{i} + \bar{c}_1]$ e $A[H\bar{i} + \bar{c}_2]$; ou seja, duas iterações $\bar{i}_1$ e $\bar{i}_2$ podem referenciar os mesmos elementos, se e somente se $H\bar{i}_1 + \bar{c}_1 = H\bar{i}_2 + \bar{c}_2$; isto é, $H(\bar{i}_2 - \bar{i}_1) = \bar{r}$, de acordo com Chen *et al.* [1994].

O exemplo 8.1 é usado para ilustrar a idéia da estratégia de alocação dos dados sem comunicação. Os vetores A, B, e C do laço L1 tem as referências A[2i, j], A[2i-2, j-1], B[j, i+1], C[i, j], e C[i-1, j-1], respectivamente. Os vetores-referência de A e C são $\bar{r}_1$ = (2,1) e $\bar{r}_2$ = (1,1), respectivamente. Não existe vetor-referência de B, pois há apenas a existência de uma referência ao vetor B.

Na iteração (1,1), a referência A[2,1] é gerada por S_1, e A[0,0] é usada em S_2. Então, na iteração (2,2), a referência A[4,2] é gerada por S_1, e A[2,1] é usada em S_2, e assim por diante.

Sobram duas iterações, $\bar{i}_1$=(1,1) e $\bar{i}_2$ = (2,2), que satisfazem a condição $H_A(\bar{i}_2 - \bar{i}_1) = \bar{r}_1$ e podem acessar o mesmo elemento A[2,1]. O espaço de dados do vetor A é entretanto particionado ao longo do vetor-referência $\bar{r}_1$ sobre os blocos de dados B_j^A, para $1 \leq j \leq 7$, fechando os pontos com linhas, como mostrado na figura 8.1(a). Estas usam e geram elementos agrupados nos mesmos blocos de dados que estão para serem alocados para os mesmos processadores. Similarmente, o vetor C é também particionado ao longo do vetor-referência $\bar{r}_2$ sobre seus blocos de dados correspondentes B_j^C, para $1 \leq j \leq 7$, como mostrado na figura 8.1(c). Observa-se que o espaço de iteração é particionado ao longo da direção (1,1), como mostrado na figura 8.2, e não existe comunicação inter-bloco para os vetores A e C. Entretanto, o vetor B será particionado ao longo da direção (1,1) sobre os blocos de dados correspondentes B_j^B, para $1 \leq j \leq 7$, como mostrado na figura 8.1(b), tal que os blocos de iteração particionados B_j, para $1 \leq j \leq 7$, possam ser executados em paralelo sem comunicação inter-bloco.

Na figura 8.1, observamos que não existem transferências de dados entre os processadores, desde que os blocos de dados correspondente B_j^A, B_j^B, e B_j^C sejam atribuidos para os processadores PE_j, para $1 \leq j \leq 7$.

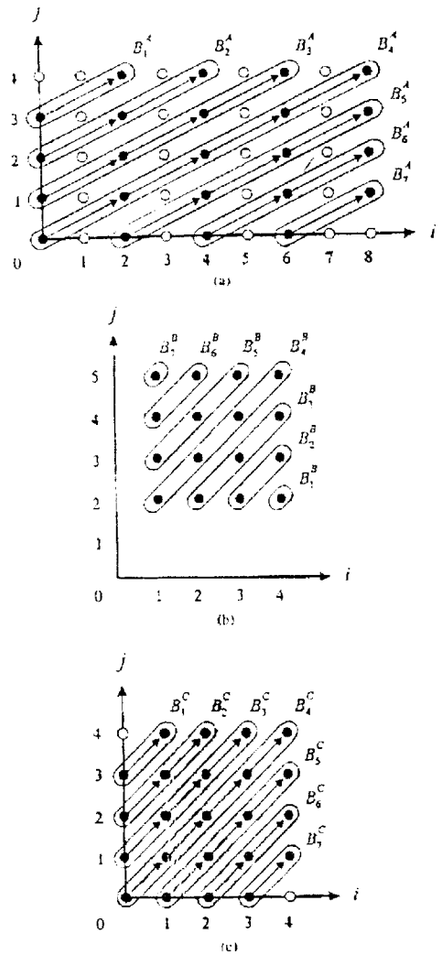


Figura 8.1. Vetores particionados A,B, e C do laço L1 sobre seus correspondentes blocos de dados. (a) vetor A[0:8,0:4], (b) vetor B[1:4,2:5], (c) vetor C[0:4,0:4].

8.3. PARTIÇÃO DE VETORES SEM COMUNICAÇÃO

Nesta seção, descrevemos o esquema de partição de vetores sem comunicação, que analisa o uso dos dados de cada vetor e deriva as condições suficientes para a determinação do padrão de partição dos elementos no laço.

8.3.1. SEM DADOS DUPLICADOS

Agora discutimos a partição do vetor sem comunicação e sem que tenhamos dados duplicados; isto é, existirá somente uma cópia de cada elemento durante a execução do programa. Não existindo transferências de dados durante a execução paralela dos programas, obtemos uma melhor eficiência em multicomputadores de memória distribuída. Contudo, não ter comunicação inter-processador é impossível se uma dependência de fluxo existir entre os programas particionados, de acordo com Chen *et al.* [1994].

Dado um laço aninhado L, o problema é de como particionar as referências sobre o laço L, tal que não apenas o *overhead* de comunicação não seja necessário, mas também o grau de paralelismo possa ser obtido tão grande quanto possível. Primeiro analisamos as relações entre todas as referências do laço L, e então o espaço de iteração é particionado sobre os blocos de iteração, tal que não existam comunicações inter-blocos. Para cada bloco de iteração particionado, os dados referenciados por estas iterações, devem ser agrupados sobre seus blocos de dados correspondentes para cada vetor.

Por outro lado, um bloco de iteração particionado e o seu correspondente bloco de dados particionado será alocado para o mesmo processador. O método proposto pode tornar o tamanho dos blocos de iteração particionados tão pequeno quanto possível, de modo a gerar um alto grau de paralelismo.

A partir da definição de um espaço-vetor, um espaço-vetor dimensional V sobre R pode ser gerado usando exatamente **n** vetores linearmente independentes. Sendo X um conjunto de **p** vetores linearmente independentes, onde $\mathbf{p} \leq \mathbf{n}$. Estes **p** vetores formam uma base de um sub-espaço-dimensional, denotado por *span*(X), de V sobre **R**. A dimensão de um espaço-vetor V é denotado por *dim*(V), segundo Chen *et al.* [1994].

Definição 8.2: A partição da iteração de um laço aninhado L particionado pelo espaço $\Psi = span(\{\bar{t}_1, \bar{t}_2, \ldots, \bar{t}_u\})$, onde $\bar{t}_l \in R^n$, para $1 \leq l \leq u$, é denotado como $P_\Psi(I^n)$, é usado para particionar o espaço de iteração I^n sobre os blocos de iteração disjuntos $B_1, B_2, \ldots, B_q$, onde **q** é o número total de blocos particionados. Para cada bloco de iteração B_j, existe um ponto base $\bar{b}_j \in R^n$ e o bloco é dado pela expressão $B_j = \{\bar{i} \in I^n \mid \bar{i} = \bar{b}_j + a_1\bar{t}_1 + a_2\bar{t}_2 + \ldots + a_u\bar{t}_u,\ a_l \in R, \text{ para } 1 \leq l \leq u\}$, para $1 \leq j \leq \mathbf{q}$, onde:

$$I^n = \bigcup_{1 \le j \le q} B_j.$$

Se $dim(\Psi) = n$, então há a existência de apenas um bloco de iteração, o espaço de iteração inteiro I^n, enquanto aplicamos a partição da iteração $P_\Psi(I^n)$ para o laço L. Se $dim(\Psi) = 0$, então uma iteração é um bloco de iteração enquanto aplicamos a partição da iteração $P_\Psi(I^n)$ para o laço L.

Definição 8.3: Dada uma partição da iteração $P_\Psi(I^n)$, a partição dos dados do vetor A com todas as **(s)** referências, $A[H_A\bar{i} + \bar{c}_1], \ldots, A[H_A\bar{i} + \bar{c}_s]$, denotado como $P_\Psi(A)$, é a partição do espaço dos dados do vetor A sobre **q** blocos de dados B_1^A, B_2^A, ..., B_q^A. Para cada bloco de dados B_j^A corresponde um bloco de iteração B_j de $P_\Psi(I^n)$, para $1 \le j \le q$, então há a existência da seguinte condição:

$$B_j^A = \{A[\bar{a}] \mid \bar{a} = H_A\bar{i} + \bar{c}_l, \bar{i} \in B_j, \text{ para } 1 \le l \le s\}.$$

Considerando o exemplo 8.1, se $\Psi = span(\{(1,1)\})$ é escolhido como o espaço da partição da iteração $P_\Psi(I^2)$ no laço L1, então o espaço de iteração pode ser particionado sobre sete blocos de iteração, como mostrado na figura 8.2. Os pontos fechados por uma linha formam um bloco de iteração, e os pontos pontilhados representam os pontos base dos blocos correspondentes. Por exemplo, o ponto base $\bar{b}_5$ do bloco de iteração $B_5=\{\bar{i} \in I^2 \mid \bar{i} = \bar{b}_5 + a(1,1), \text{ para } 0 \le a \le 2\}$ é (2,1). Baseado na partição da iteração $P_\Psi(I^2)$, os vetores A, B, e C são particionados sobre os blocos de dados correspondentes pelo uso da respectiva partição de dados $P_\Psi(A)$, $P_\Psi(B)$, e $P_\Psi(C)$, como mostrado na figura 8.1.

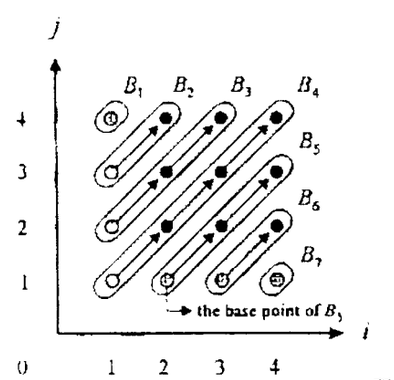


Figura 8.2. Particionamento do espaço de iteração do laço L1 sobre os blocos de iterações correspondentes

Exemplo 8.2: Considerando o laço aninhado L2,

```
FOR i = 1 to 4 do
        FOR j = 1 to 4 do
                S1 : A[i+j, i+j] = B[2i, j] * A[i+j-1, i+j];
                S2 : A[i+j-1, i+j-1] = B[2i -1, j-1] / 3;          (L2)
        END_FOR
END_FOR
```

Neste laço, as respectivas funções-referência dos vetores A e B são:

$$H_A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \text{ e } H_B = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}.$$

Os vetores-referência $\bar{r}_1$, entre A[i+j, i+j] e A[i+j-1, i+j-1], $\bar{r}_2$, entre A[i+j-1, i+j-1] e A[i+j-1, i+j], e $\bar{r}_3$, entre A[i+j-1, i+j] e A[i+j, i+j], do vetor A são (1,1), (0,-1), e (-1,0), respectivamente. O vetor-referência $\bar{r}_4$ do vetor B é (-1,0).

Considerando a equação $H_A\bar{i}_2 = \bar{r}_2$, ondeas duas iterações $\bar{i}_1$ e $\bar{i}_2$ podem acessar os mesmos elementos do vetor A se a equação $\bar{i}_2 - \bar{i}_1 = \bar{t}_2$ for satisfeita. Por não existir solução da equação $H_A\bar{i}_2 = \bar{r}_2$, então também não existe dependência de dados entre A[i+j-1, i+j-1] e A[i+j-1, i+j]. Contudo, resolvendo a equação $H_B\bar{i}_4=\bar{r}_4$ obtemos uma solução $\bar{i}_4 = (\frac{1}{2},1)$ que é impossível, pois não pertence a Z^2. Também não existe dependência de dados sobre o vetor B. O símbolo $0^d \in Z^d$ será denotado como um <u>vetor zero</u> onde cada componente é igual a 0.

Considerando a equação $H\bar{t} = \bar{r}$, queno caso especial em que $\bar{r} = 0^d$, o conjunto de soluções $\bar{t}$ da equação $H\bar{t} = 0^d$ é *Ker*(H), que é o espaço nulo de H. O vetor $\bar{t}$ indica a diferença entre duas iterações acessando os mesmos elementos, de acordo com Chen *et al.* [1994]. Por exemplo, *Ker*(H_A) é *span*({(1,-1)}) no laço L2. Sobre a referência A[i+j, i+j], o elemento A[4,4], é referenciado pela iteração (1,3), e pode ser referenciado novamente pelas iterações (1,3) + *span*({(1,-1)}), isto é, (2,2) e (3,1), do laço L2.

A seguir, discutiremos como escolher o melhor espaço para particionar o espaço de iteração e o espaço de dado sem que se tenha dados duplicados, tal que não existam comunicações inter-blocos e o paralelismo extraído seja tão grande quanto possível.

Definição 8.4: Em um laço aninhado L, se a função-referência H_A e as **(s)** referências, A[$H_A\bar{i} + \bar{c}_1$], ... , A[$H_A\bar{i} + \bar{c}_s$] para o vetor A existirem, então os vetores-referência são $\bar{r}_p = \bar{c}_j - \bar{c}_k$, para todos 1 $\leq j < k \leq s$ e $1 \leq p \leq \frac{s(s-1)}{2}$, e o espaço-referência do vetor A é:

$$\Psi_A = span(\beta \cup \{\bar{t}_1, \bar{t}_2, \ldots, \bar{t}_{\frac{s(s-1)}{2}}\}),$$

onde β é a base de $Ker(H_A)$, e $\bar{t}_j \in R^n$, para $1 \leq j \leq \frac{s(s-1)}{2}$, que satisfaz as seguintes condições:

1) $\bar{t}_j$ é uma solução particular da equação $H_A\bar{t} = \bar{r}_j$.

2) uma solução $\bar{t}' \in \bar{t}_j + Ker(H_A)$ existe, tal que $\bar{t}' \in Z^n$ e $\bar{t}' = \bar{i}_2 - \bar{i}_1$ onde $\bar{i}_1, \bar{i}_2 \in I^n$.

O espaço-referência representa as relações de todas as dependências entre as iterações. Para o vetor A, não existem dependências entre os blocos de iterações quando o espaço de iteração I^n é particionado com o espaço-referência Ψ_A. Isto ocorre porque todas as dependências de dados são consideradas em Ψ_A, tal que os dados acessados não são necessários entre os blocos de iteração. Em cada bloco de iteração, as iterações obedecem a ordem lexicográfica e são executadas preservando as relações de dependências do laço, segundo Chen *et al.* [1994].

Considerando o laço L2 do exemplo 8.2, o espaço-referência Ψ_A do vetor A é $span(\{(1,-1), (\frac{1}{2},\frac{1}{2})\})$, por que $Ker(H_A) = span(\{(1,-1)\})$ e existe uma solução particular $\bar{t}_1 = (\frac{1}{2},\frac{1}{2})$ da equação $H_A\bar{t} = \bar{r}_1$ que satisfaz as condições (1) e (2) da Definição 8.4. O espaço-referência Ψ_B do vetor B é $span(\phi)$, por que $Ker(H_B) = \{0^2\}$, e apenas a solução $\bar{t}_4 = (\frac{1}{2},1) \notin Z^2$ não satisfaz a condição (2) da Definição 8.4.

Na discussão acima, apenas a partição da iteração sem comunicação $P_{\Psi A}(I^n)$ e a partição dos dados $P_{\Psi A}(A)$ de um vetor A são considerados no laço aninhado. Sempre que particionamos o espaço de iteração, não apenas as referências que ocorrem nos vetores serão consideradas, mas também as referências que ocorrem entre os vetores no laço aninhado.

Dado um laço aninhado L com **k** variáveis-vetor, o espaço-referência Ψ_{Aj} será $span(X_j)$ do vetor A_j, para $1 \leq j \leq k$. Então $\Psi = span(X_1 \cup X_2 \cup \ldots \cup X_k)$ é o espaço partição para a partição sem comunicação dos vetores no laço L e sem dados duplicados. Todos os blocos de iteração particionados por $P_\Psi(I^n)$ podem ser corretamente executados em paralelo, de acordo com Chen *et al.* [1994].

Teorema 8.1: Dado um laço aninhado L com k variáveis-vetor, o espaço referencia Ψ_{Aj} será $span(X_j)$ de cada vetor A_j, para $1 \leq j \leq k$. Se $\Psi = span(X_1 \cup X_2 \cup \ldots \cup X_k)$, então Ψ é o espaço de

partição para o particionamento sem comunicação dos vetores A_j, para $1 \leq j \leq k$, **sem** dados duplicados, usando a partição da iteração $P_\Psi(I^n)$, de acordo com Chen *et al.* [1994].

Teorema 8.2: Dado um laço aninhado L com k variáveis-vetor, o espaço referencia reduzido Ψ^r_{Aj} será $span(X^r_j)$ de cada vetor A_j, para $1 \leq j \leq k$. Se $\Psi^r = span(X^r_1 \cup X^r_2 \cup \ldots \cup X^r_k)$, então Ψ^r é o espaço de partição para o particionamento sem comunicação dos vetores A_j, para $1 \leq j \leq k$, **com** dados duplicados, usando a partição da iteração $P_{\Psi^r}(I^n)$, de acordo com Chen *et al.* [1994].

Teorema 8.3: Dado um laço aninhado L com k variáveis-vetor, o espaço referencia mínimo $\Psi_{Aj}^{min\ r}$ será $span(X_j)$ de cada vetor A_j, para $1 \leq j \leq k$. Se $\Psi^{min\ r} = span(X^r_1 \cup X^r_2 \cup \ldots \cup X^r_k)$, então $\Psi^{min\ r}$ é o espaço de partição mínimo para o particionamento sem comunicação dos vetores A_j, para $1 \leq j \leq k$, **com** dados duplicados, usando a partição da iteração $P_{\Psi^{min\ r}}(I^n)$, de acordo com Chen *et al.* [1994].

Teorema 8.4: Dado um laço aninhado L com k variáveis-vetor, o espaço referencia reduzido mínimo Ψ_{Aj}^{min} será $span(X^r_j)$ de cada vetor A_j, para $1 \leq j \leq k$. Se $\Psi^{min} = span(X_1 \cup X_2 \cup \ldots \cup X_k)$, então Ψ^{min} é o espaço de partição mínimo para o particionamento sem comunicação dos vetores A_j, para $1 \leq j \leq k$, **sem** dados duplicados, usando a partição da iteração $P_{\Psi^{min}}(I^n)$, de acordo com Chen *et al.* [1994].

Pelo Teorema 8.1, quando $dim(\Psi) < n$, então existe paralelismo no laço L para a partição iteração $P_\Psi(I^n)$. Pela Definição 8.2, o valor de $dim(\Psi)$, é o máximo grau de paralelismo possível, de acordo com Chen *et al.* [1994].

Ilustramos a partição do vetor sem comunicação e sem dados duplicados, considerando o exemplo 8.1, onde os espaços-referência são $\Psi_A=\Psi_C=span(\{(1,1)\})$, e $\Psi_B=\{0^2\}$ para os respectivos vetores A, C, e B. Portanto, pelo Teorema 8.1, o espaço particionado é $\Psi=span(\{(1,1)\}) \cup \{(1,1)\} \cup \phi)$ para partição iteração sem comunicação $P_\Psi(I^2)$ para laço L1. Como $dim(\Psi) = 1$ (<2), então existe uma grande soma de paralelismo no laço L1. O resultado dos dados particionados e blocos de iteração no laço L1 é mostrado na figura 8.1 e figura 8.2, respectivamente.

Permitindo a duplicação dos dados para alguns elementos, pode tornar possível, que vários laços possam existir com uma grande soma de paralelismo, para a partição do vetor sem comunicação.

8.3.2. COM DADOS DUPLICADOS

Agora consideramos a partição do vetor sem comunicação e com dados duplicados; isto é, pode existir mais do que uma cópia de um elemento alocado na memória local dos processadores. Por causa do *overhead* de comunicação, consome-se mais tempo executando os programas em paralelo, e é proveitoso termos dados duplicados nos processadores, tal que um alto grau de paralelismo possa ser explorado; enquanto isto, as computações serão corretamente executadas sem comunicação.

A estratégia de dados duplicados, em comparação com uma de dados não duplicados, pode extrair mais paralelismo dos programas baseados em partição de vetor sem comunicação.

Definição 8.5: Se não existem dependências de fluxo em um vetor A, então o vetor A é chamado um vetor duplicável completamente; caso contrário, o vetor A é chamado um vetor duplicável parcialmente.

Observe que nos vetores duplicáveis completamente podem incorrer anti-dependências, dependências de saída, ou de entrada; entretanto, nos vetores duplicáveis parcialmente podem também incorrer as dependências de fluxo.

A seguir discute-se como escolher o melhor espaço para particionar o espaço de iteração e os vetores com dados duplicados, tal que não existam comunicações inter-blocos. Primeiro, examinamos os vetores duplicáveis completamente no laço L. Por não existirem dependências de fluxo no vetor A, os dados podem ser arbitrariamente distribuídos para cada processador, e o laço original pode ser corretamente executado em paralelo. Por tanto, o espaço-referência Ψ_A pode ser reduzido sobre *span*(ϕ) denotado como o espaço-referência reduzido Ψ^r_A. Isto é, Ψ^r_A é o sub-espaço de Ψ_A.

Para examinar os vetores duplicados parcialmente, assumimos que existem **p** dependências de fluxo em um vetor duplicável parcialmente A no laço L. O espaço-referência Ψ_A do vetor A pode ser reduzido sobre o espaço-referência reduzido $\Psi^r_A = (\beta \cup \{\bar{t}_1, \bar{t}_2, ..., \bar{t}_p\})$, onde β é a base de *Ker*(H_A) e $\bar{t}_j$, para $1 \leq j \leq$ **p**, são soluções particulares, que satisfazem as condições (1) e (2) na Definição 8.4.

A razão para o espaço-referência ser redutível, é que apenas as dependências de fluxo podem causar a transferência de dados entre as execuções das iterações. Isto é, apenas as dependências de

fluxo são necessárias considerar durante a execução de programas; contudo, as dependências de entrada, de saída e anti-dependências meramente determinam a precedência da execução das iterações, pois elas não causam transferências de dados, de acordo com Chen *et al.* [1994].

Agora consideramos como particionar o espaço de iteração, quando referências ocorrem entre todos os vetores em um laço aninhado. Pegando um laço aninhado L com k variáveis-vetor, o espaço-referência reduzido Ψ^r_{Aj} será $span(X^r_j)$ de cada um dos outros vetores duplicáveis parcialmente ou completamente A_j, para $1 \leq j \leq k$. No Teorema 8.2, temos que $\Psi^r = span(X^r_1 \cup X^r_2 \cup \ldots \cup X^r_k)$ é o espaço de partição para particionamento sem comunicação e com dados duplicados pelo uso da partição iteração $P_\Psi{}^r(I^n)$, de acordo com Chen *et al.* [1994].

Ilustramos o particionamento do vetor sem comunicação e com dados duplicados, considerando os exemplos anteriores. Primeiro, no exemplo 8.1 os espaços-referência reduzidos são $\Psi^r_A = span(\{(1,1)\})$ e $\Psi^r_B = \Psi^r_C = span(\phi)$ para os respectivos vetores A, B, e C. Por isso, pelo Teorema 8.2, o espaço de partição do laço L1 é $\Psi^r = span(\{(1,1)\} \cup \phi \cup \phi)$ para partição de iteração sem comunicação $P_\Psi{}^r(I^2)$. Para o laço L1, a estratégia com dados duplicados obtém os mesmos resultados que a estratégia sem dados duplicados. Isto é, o laço L1 não necessita de dados duplicados para acentuar o paralelismo.

Pelo Teorema 8.1, enquanto aplicamos a partição iteração $P_\Psi(I^2)$ para o laço L2, onde $\Psi = span(\{(1,-1), (\frac{1}{2},\frac{1}{2})\})$, o laço precisa ser executado seqüencialmente baseado na estratégia sem dados duplicados. Como ambos os vetores A e B no laço L2 são vetores duplicáveis completamente, o espaço particionado Ψ^r é $span(\phi)$ pelo Teorema 8.2. Enquanto aplicamos a partição iteração $P_\Psi{}^r(I^2)$ para o laço L2, este pode ser executado em paralelo.

As condições suficientes para o particionamento do vetor sem comunicação tem sido discutido e derivado aqui sem dados duplicados, correspondendo para os Teoremas 8.1 e 8.4, e com dados duplicados, correspondendo para os Teoremas 8.2 e 8.3.

8.4. TRANSFORMAÇÃO DO PROGRAMA

Agora descrevemos a transformação do programa de um laço aninhado particionado sem levar em conta o número de processadores.

Pelos Teoremas 8.1 a 8.4, podemos obter um laço aninhado com o espaço de particionamento $\Psi = span(X)$, onde X consiste de **g** vetores linearmente independentes; isto é,

$dim(\Psi) = g$. O laço aninhado particionado é então transformado sobre a forma de execução paralela, com k(=n-g) laços FORALL, apenas usando a partição iteração $P_\Psi(I^n)$, segundo Chen *et al.* [1994].

Pela projeção ortogonal, cada bloco iteração pode ser projetado em um o sub-espaço $Ker(\Psi)$. Isto implica que a base Q do $Ker(\Psi)$ pode portanto, ser usada para representar cada ponto transformado dos **k** laços FORALL; isto é, cada ponto indica um bloco de iteração particionado. Como $\boldsymbol{dim(Ker(\Psi)) = k}$ e $\boldsymbol{dim(\Psi) = g}$, então existe **k** índices FORALL mais externos e **g** índices mais internos no laço aninhado transformado, de acordo com Chen *et al.* [1994].

Primeiro, derivamos a base $Q = \{\bar{a}_i = (a_{i,1}, a_{i,2}, \ldots, a_{i,n}) \in Z^n | \gcd(a_{i,1}, a_{i,2}, \ldots, a_{i,n}) = 1$, para $1 \leq i \leq k\}$ para $Ker(\Psi)$. As operações elementares de linha são usadas na matriz $\begin{bmatrix} \bar{a}_1 \\ \vdots \\ \bar{a}_k \end{bmatrix}_{k \times n}$ para derivar a forma ***echelon*** linha $\begin{bmatrix} \bar{a}'_1 \\ \vdots \\ \bar{a}'_k \end{bmatrix}_{k \times n}$, onde $\bar{a}'_j$ é derivado apartir $\bar{a}_i$, onde $j = \sigma(i)$, e a função σ: $i \rightarrow j$ é uma permutação, para $1 \leq i,j \leq k$. A primeira posição do componente não zero de $\bar{a}'_j$ é y_i, para $1 \leq j \leq k$, e $y_j < y_{j+1}$, para $1 \leq j \leq k$.

Os novos índices $I'_1, I'_2, \ldots, I'_n$ podem ser obtidos apartir dos índices originais $I_1, I_2, \ldots, I_n$ mediante as seguintes equações:

$$I'_i = a_{\sigma^{-1}(j),1} I_1 + a_{\sigma^{-1}(j),2} I_2 + \ldots + a_{\sigma^{-1}(j),n} I_n, \qquad \text{Se } i = y_i, \text{ para } 1 \leq i \leq n \text{ e } 1 \leq j \leq k,$$

$$I'_i = I_i, \qquad \text{Se } i \neq y_j, \text{ para } 1 \leq i \leq n \text{ e } 1 \leq j \leq k, \qquad (1)$$

As relações inversas de (1) são derivadas como segue:

$$I_i = b_{j,1} I'_1 + b_{j,2} I'_2 + \ldots + b_{j,n} I'_n, \qquad \text{Se } i = y_j, \text{ para } 1 \leq i \leq n \text{ e } 1 \leq j \leq k,$$

$$\text{onde } b_{j,l} \in R, \text{ para } 1 \leq l \leq n; \qquad (2)$$

$$I_i = I'_i, \qquad \text{Se } i \neq y_j, \text{ para } 1 \leq i \leq n \text{ e } 1 \leq j \leq k.$$

Os limites inferiores l'_j e superiores u'_j dos k índices FORALL mais externos I'_{y_j}, para $1 \leq j \leq k$, podem ser calculados sobre a base de uma faixa dos índices originais e as equações (1) e (2). Para

executar todas as iterações dentro de um bloco de iteração, seguindo à ordem lexicográfica, os primeiros g índices I_{z_i}, para $1 \leq z_i \leq n$ e $1 \leq i \leq g$, são escolhidos como os índices mais internos. O I_{z_i} não pode ser expresso linearmente pelos índices $I'_{y_1}, \ldots, I'_{y_k}, I_{z_1}, \ldots, I_{z_{i-1}}$, para $1 \leq i \leq g$ e $z_j < z_{j+1}$, para $1 \leq j \leq g$, conforme Chen *et al.* [1994].

A principal forma de selecionar os **g** índices, é fazer um mapeamento apartir do espaço de iteração original para um novo espaço transformado. Similarmente, os respectivos limites inferiores l'_j e superiores u'_j, para $(k+1) \leq j \leq n$, dos índices mais internos I_{z_i}, para $1 \leq i \leq g$, podem ser derivados. Todos os limites inferiores e superiores podem ser determinados pelo método de transformação dos limites do laço proposto em [Wolf *et al.* 1991] Além disso, para determinar os valores dos índices originais, exceto para os **g** índices mais internos, as atribuições estendidas podem ser derivadas pelas equações (1) e (2) como segue:

$$I_i = c_{i,1} I'_{y_1} + \ldots + c_{i,k} I'_{y_k} + c_{i,k+1} I_1 + \ldots + c_{i,k+i-1} I_{i-1},$$

onde $i \neq z_j$ e $c_{i,l} \in R$, para $1 \leq l \leq (k+i-1)$, para $1 \leq i \leq n$ e $1 \leq j \leq g$.

Com base nas transformações acima, o novo laço transformado L' é:

```
FORALL I'_y1 = l'_1 to u'_1 do
  FORALL I'_y2 = l'_2 to u'_2 do
    ⋮
    FORALL I'_yk = l'_k to u'_k do
      FOR I_z1 = l'_k+1 to u'_k+1 do
        ⋮
        FOR I_zg = l'_n to u'_n do
            { Corpo do Laço Modificado }                    ( L' )
        END_FOR
        ⋮
      END_FOR
    END_FORALL
    ⋮
  END_FORALL
END_FORALL.
```

A transformação de um laço aninhado particionado para uma forma de execução paralela, será ilustrado com o exemplo seguinte.

Exemplo 8.3: Considerando um laço L3 com três aninhamentos.

```
FOR  i1 = 1 to 4 do
        FOR i2 = 1 to 4 do
                FOR i3 = 1 to 4 do
                        A[i1,i2,i3] = A[i1-1,i2+1,i3-1] + B[i1,i2,i3];          ( L3 )
                END_FOR
        END_FOR
END_FOR
```

Pela aplicação dos Teoremas 8.1 a 8.4, o espaço particionado mínimo do laço L3 é $\Psi = span(\{(1,-1,1)\})$. Isto é, o laço L3 não necessita de dados duplicados para herdar o paralelismo.

Primeiro, obtemos que a base do $\boldsymbol{Ker}(\Psi)$ é $Q=\{(q_1,q_2,q_3), (b_1,b_2,b_3)$, onde $(q_1,q_2,q_3) = (1,1,0)$ e $(b_1,b_2,b_3) = (-1,0,1)$, desde que $\boldsymbol{gcd}(q_1,q_2,q_3) = 1$ e $\boldsymbol{gcd}(b_1,b_2,b_3) = 1$.

Os novos índices i'_1, i'_2, e i'_3 serão $i'_1 = q_1 i_1 + q_2 i_2 + q_3 i_3 = i_1 + i_2$, $i'_2 = b_1 i_1 + b_2 i_2 + b_3 i_3 = -i_1 + i_3$, e $i'_3 = i_3$. As relações inversas são derivadas como $i_1 = -i'_2 + i'_3$, $i_2 = i'_1 + i'_2 - i'_3$, e $i_3 = i'_3$.

Como $\boldsymbol{dim}(\boldsymbol{Ker}(\Psi)) = 2$ e $\boldsymbol{dim}(\Psi) = 1$, então existe dois índices **FORALL** mais externos i'_1 e i'_2 e um índice mais interno i_1, respectivamente. O seguinte laço transformado L3' pode ser obtido através do uso da estratégia de transformação acima.

```
FORALL i'1 = 2 to 8 do
  FORALL i'2 = max(-3,-i'1 + 2) to min(3,-i'1 + 8) do
      FOR i1 = max(1, i'1 -4, -i'2 + 1) to min(4,i'1-1, -i'2+4) do
          E1: i2 = i'1 - i1;
          E2: i3 = i'2 + i1;
          A[i1,i2,i3] = A[i1-1,i2+1,i3-1] + B[i1,i2,i3];          (L3')
      END_FOR
  END_FORALL
END_FORALL
```

As atribuições E_1 e E_2 no laço L3' são as atribuições estendidas. Cada ponto do conjunto $\{(i'_1,i'_2) \mid 2 \le i'_1 \le 8 \text{ e } \max(-3, -i'_1 + 2) \le i'_2 \le \min(3, -i'_1 + 8)\}$ representa um bloco de iteração, e todos os pontos podem ser executados em paralelo sem comunicação inter-bloco.

Embora o número de processadores seja maior que o número de blocos de iteração particionados, cada bloco de iteração particionado corresponde a um elemento do conjunto $\{(I'_{y_1}, I'_{y_2}, \ldots, I'_{y_k}) \mid I'_j \le I'_{y_j} \le u'_j$, para $1 \le j \le k\}$, e os blocos de dados particionados de cada vetor podem ser distribuídos na memória local dos processadores correspondentes.

CAPÍTULO 9 - MÉTODO DE PARTICIONAMENTO & ROTULAÇÃO

O método apresentado neste capítulo permite, que obtenhamos a paralelização de laços, utilizando uma estrutura DO_ALL.

9.1. INTRODUÇÃO

Apresentamos duas técnicas, a de conversão de código e a de *rotulação*, que juntas formam o novo método. A primeira técnica transforma um laço aninhado em uma estrutura DO_ALL, e a segunda gera os *rótulos*.

9.2. NOTAÇÃO

Considerando um laço aninhado $L = (I_1, I_2, \ldots, I_n)(S_1, S_2, \ldots, S_S)$, onde I_i, para $1 \leq i \leq n$, é o índice e S_j, para $1 \leq j \leq s$, é um comando.

Um laço exemplo é um laço iteração onde os índices adquirem um valor particular, $I = \bar{i}$, com $\bar{i} = (i_1, i_2, \ldots, i_n)^T \in Z^n$. O $\bar{i}$ é o ponto de iteração do laço. O conjunto de índices I é o conjunto de pontos, $I = \{\bar{i}\}$, e constituem o espaço de iteração do laço. A nomenclatura $Sj(\bar{i})$ é a execução do comando Sj, onde o índice é $I = \bar{i}$. Sendo $P = (P_1, P_2, \ldots, P_q)$ uma partição do conjunto de índices I em subconjuntos P_k, para $1 \leq k \leq q$.

Primeiro buscamos a máxima partição do índice independente, $Pmáx = \{P_1, P_2, \ldots, P_{máx}\}$, tal que:

1) cada subconjunto P_k é um conjunto de dependências, isto é, onde todos os pontos de P_k estão conectados;
2) $P_i \cap P_j = 0$, para $i \neq j$, isto é, não existem dependências entre as iterações de diferentes conjuntos P_i e P_j.

Os conjuntos que satisfazem a condição 2 são chamados conjuntos de dependência máximos.

Segundo D'Hollander [1992], a cardinalidade da partição, $máx = |P_{máx}|$, é chamada de particionalidade do laço. Quando a máxima partição é estabelecida, as iterações nos diferentes subconjuntos P_k, para k = 1,..., *máx*, podem ser executadas em paralelo. Dentro de cada subconjunto as iterações estão conectadas e podem ser executadas em ordem lexicográfica.

O fato do espaço de iteração ser delimitado pelos limites do laço pode guiar para um particionamento mais distante, isto é, se duas iterações estão conectadas apenas através de pontos índices fora do espaço de iteração do laço I, então elas podem ser executadas em paralelo, segundo D'Hollander [1992].

O problema do particionamento pode ser resolvido se a distância da dependência entre as iterações for constante. Este é o caso para vetores com subscritos da forma (i+c), onde a constante **c** é conhecida quando o particionamento for alocado (no tempo de compilação ou no tempo de execução).

Os vetores dependência são armazenados na matriz dependência $D = [\bar{d}_1 \_ \bar{d}_2 \ldots \bar{d}_m]$, onde **m** é o número de vetores dependência.

A forma canônica do laço aninhado L é

DO I

$$\bar{S}(I) = f(\bar{S}(I - \bar{d}_1), \bar{S}(I - \bar{d}_2), \ldots, \bar{S}(I - \bar{d}_m)) \qquad (1)$$

END_DO

onde $\bar{S}(I)$ são as referências que representam a computação do corpo do laço, que é executado em ordem lexicográfica.

LEMA 9.1: Dois pontos índices, I^1 e I^2, são dependentes $(I^1 . \delta . I^2)$, se e somente se $I^1 - I^2 = D\bar{y}$, onde $\bar{y} = [y_1 y_2 \ldots y_m]^T \in Z^m$.

LEMA 9.2: Um conjunto de dependência máximo, $P_K = \{I\}$, para K=1,...,*máx*, é gerado por um dos elementos, $I_0^K = [I_{0,1}^K I_{0,2}^K \ldots I_{0,m}^K]$, usando a equação dependência

$$I = I_0^K + D\bar{y}, \bar{y} \in Z^m, \qquad (2)$$

onde I_0^K é chamado o rótulo do conjunto de dependência máximo P_K.

Corolário 9.1: Dois *Rótulos* conectados $I_0^1 . \delta . I_0^2$ geram o mesmo conjunto de dependência máximo.

Definido o conjunto *Rótulo* como o conjunto dos *rótulos* de todos os conjuntos de dependência, $L = \{I_0^K\}$. O conjunto *Rótulo* é completo se todos os pontos I tem um *rótulo*; o conjunto é próprio se todos os pontos I tem um único *Rótulo*. Um conjunto *Rótulo* próprio é necessariamente completo.

LEMA 9.3: Um conjunto *Rótulo* próprio gera a partição independente máxima.

9.3. TRANSFORMAÇÃO UNIMODULAR

Uma matriz unimodular é uma matriz quadrada de elementos inteiros, dos quais o determinante tem os valores +1, -1 ou 0. Uma matriz unimodular regular é uma matriz unimodular com o determinante de +1 ou -1. Utilizamos apenas matrizes unimodulares regulares. Se U é uma matriz unimodular, então o inverso U^{-1} também é unimodular. O produto de duas matrizes unimodular é também uma matriz unimodular, de acordo com D'Hollander [1992].

Definição 9.1: Uma Transformação Unimodular da matriz dependência $D_{n \times m}$ é a matriz $D^u_{nxm} = U_{nxn} . D_{nxm} . V_{mxm}$, onde U_{nxn} e V_{mxm} são matrizes unimodulares.

Definição 9.2: Uma Transformação Unimodular do conjunto de índices **I** é o conjunto dos índices **Y**=$\{Y | Y = UI, I \in Z^n\}$, onde U é unimodular.

Corolário 9.2: Como U^{-1} é unimodular, então existe um mapeamento entre o conjunto de índices **I** e **Y**. As Transformações Unimodulares são de interesse para a construção de partições paralelas porque:

1) a relação dependência (δ) é invariante sobre uma transformação unimodular;
2) existe uma transformação unimodular que reduz a matriz dependência para uma forma diagonal ou uma triangular;
3) a máxima partição independente de uma matriz diagonal ou triangular é conhecida.

No caso de matrizes não quadradas, usamos o termo diagonal e triangular para indicar matrizes a qual tem zeros fora da diagonal principal ou apenas em um dos lados da diagonal

principal, respectivamente. A notação $\delta(D)$ é usada para indicar o operador dependência com respeito para a matriz D.

O seguinte teorema mostra que a relação dependência (δ) é invariante sobre uma transformação unimodular.

Teorema 9.1: Sendo $Y=UI$ com $I \in Z^n$ e sendo U e V matrizes unimodulares. Os pontos I^1 e I^2 são conectados com respeito para matriz D, se e somente se o ponto transformado Y^1 e Y^2 são conectados com respeito para matrizes UD e UDV.

Como uma conseqüência do Corolário 9.2 e do Teorema 9.1, a Transformação Unimodular é Isomórfica e os conjuntos índices **I** e **Y** são isomórficos com respeito para com a dependência (δ). Deste modo, o particionamento do conjunto de índices **Y** produz automaticamente o particionamento do conjunto de índices **I**, pela transformação inversa do conjunto de dependências.

O corpo dos algoritmos de particionamento é a transformação unimodular da matriz dependência D sobre uma forma triangular ou diagonal. Os algoritmos são similares a um *Gaussian* ou a uma eliminação *Gauss-Jordan* para matrizes, por isso os elementos fora da diagonal são reduzidos usando uma combinação linear com o pivoteamento da linha ou coluna. A diferença é que apenas operações de inteiros podem ser usadas, segundo D'Hollander [1992].

Para simplificar os algoritmos, assumimos a matriz dependência $D_{n \times m}$ tem dimensões $m \geq n$. Quando $m < n$, D é estendido com $(n - m)$ vetores zero. Isto não reduz o paralelismo, por que um vetor dependência zero expressa $S(i)\ \delta\ S(i)$, isto é, o ciclo do laço **i** depende dele mesmo.

A Transformação Unimodular da matriz D é equivalente a uma sequência ordenada das seguintes operações elementares de linha ou coluna, conforme Wolf *et al.* [1991] e D'Hollander [1992]:

1) Troca de duas linhas ou colunas;
2) Multiplica uma linha ou coluna por (-1);
3) Adiciona um inteiro múltiplo de uma linha ou coluna para outro.

As matrizes unimodulares U e V, tal que $D^u = UDV$, são obtidas pela aplicação de sucessivas operações elementares de linha e coluna para as matrizes identidade I_u e I_v, respectivamente. O seguinte algoritmo busca uma matriz triangular inferior D^t, tal que $D^t = UDV$, para a matriz dependência não singular $D_{n \times m}$ com o *rank* $r = n$. As modificações para as matrizes dependências singulares são também apresentadas.

Algoritmo 9.1: Redução de D para uma forma triangular inferior (D^t)

1. Entrada: A matriz dependência D de n x m e o determinante **det** para uma sub-matriz arbitrária de *rank* r = n. O valor **det** é usado para limitar a magnitude dos valores intermediários (passo 4f).

2. Inicializa a matriz $V_{(m+n)x(m+n)} = I_{(m+n)x(m+n)}$, onde $I_{(m+n)x(m+n)}$ representa a matriz identidade.

3. Construção da matriz ampliada

$$A = [\ D\ !\ \det I_{n x m}\ ]_{n x (m+n)}$$

pela adição de **n** colunas de uma matriz quadrada de det *I* para a matriz $D_{n x m}$. $A_{\bullet j}$ representa a j-ésima coluna da matriz A.

4. **FOR** l = 1 **to** n **do**:

a. Procurar o elemento linha A_{lj} com o valor absoluto, tal que $j \geq l$ e $A_{lj} \neq 0$. Quando tal elemento não é encontrado, prosseguimos para o passo 4e.

b. Troca das colunas $A_{\bullet l}$ e $A_{\bullet j}$ para trazer o elemento A_{lj} sobre a diagonal. Trocando as colunas $V_{\bullet l}$ e $V_{\bullet j}$ para registrar a Transformação Unimodular.

c. **FOR** j =(l + 1) **to** (m + n) **do**: (redução de linha)

$A_{\bullet j} = A_{\bullet j} - \lfloor A_{lj} / A_{ll} \rfloor A_{\bullet l}$

$V_{\bullet j} = V_{\bullet j} - \lfloor V_{lj} / V_{ll} \rfloor V_{\bullet l}$

END_FOR j

d. Repetir apartir do passo 4a.

e. **IF** $A_{ll} < 0$ **THEN** (inverter os vetores dependência negativos)

$A_{\bullet l} = - A_{\bullet l}$

$V_{\bullet l} = - V_{\bullet l}$

END_IF

f. **FOR** j = l **to** (m + n) **do**:

FOR i = l **to** n **do**:

IF $|A_{ij}| >$ det **THEN** (reduzir números módulo det)

$A_{\bullet j} = A_{\bullet j} - \lfloor A_{ij} / \det \rfloor A_{\bullet i+m}$

$V_{\bullet j} = V_{\bullet j} - \lfloor V_{ij} / \det \rfloor V_{\bullet i+m}$

END_IF

END_FOR i

END_FOR j

END_FOR l

Como todas as operações são unimodulares, a nova matriz expressa as mesmas dependências pela virtude da invariância do Teorema 9.1.

Os passos 4a a 4d implicitamente executam o algoritmo *Eucledian* para calcular o grande divisor comum (**gcd**) das linhas (m+n-l+1) dos elementos $A_{l\,j}$, para $j \geq l$. O passo 4b eventualmente substitui **gcd**($A_{l\,l}$, ... ,$A_{l\,,m-n}$) sobre a diagonal e o passo 4c elimina os elementos $A_{l,l+1}$, ... , $A_{l,m+n}$.

Como um resultado do passo 4f, a magnitude do elementos $A_{i\,j}$ permanecem abaixo do **det**.

O algoritmo trabalha também para matrizes singulares ou quando o determinante não é conhecido, pela suposição de A=D, alterando o limite superior de (m+n) para **m** no passo 4c e omitindo o passo 4f.

Desde que as operações elementares de linha e coluna no Algoritmo 9.1 sejam unimodulares, pelo Teorema 9.1, a matriz D e a matriz transformada A = [D^t 0] descrevem as mesmas dependências (δ). As sucessivas transformações unimodulares são registradas na matriz V, tal que D^t = DV. Considerando rank(D), o *rank* da matriz dependência.

Por causa que $|V| = \pm 1$, temos r = *rank*(D) = *rank*(D^t) ≤ *min*(n,m). A matriz D^t contém r vetores linearmente independentes e (m-r) vetores linearmente dependentes. Quando r < n, é necessário uma transformação unimodular para transformar os (n-r) vetores linearmente dependentes em vetores nulos, conforme D'Hollander [1992] e isto é realizado pelo seguinte algoritmo.

Algoritmo 9.2: <u>Eliminação dos vetores linearmente dependentes</u>

```
1. Entrada: Matrizes D^t e V apartir do Algoritmo 9.1.
2. FOR l = 1 to min(n,m-1) do:
   IF D^t_ll = 0 THEN (elimina o vetor dependência D^t_*l)
     FOR i = l + 1 to n do:
        WHILE D^t_il ≠ 0 do: (elimina o elemento D^t_il)
          IF D^t_ii = 0 ou | D^t_ii |> D^t_il
                THEN (pela troca de colunas)
                        Troca colunas i e l de ambas as matrizes D^t e V.
                ELSE (pela redução da coluna)
                        D^t_*l = D^t_*l - ⌊ D^t_il / D^t_ii ⌋ D^t_*i
                        V_*l = V_*l - ⌊ D^t_il / D^t_ii ⌋ V_*i
          END_IF
        END_WHILE
     END_FOR i
   END_IF
 END_FOR l
```

O algoritmo seguinte reduz a matriz dependência D para a forma diagonal (D^d), e procura a matriz unimodular, tal que $D^d = UDV$.

Algoritmo 9.3: Redução da D para uma forma diagonal D^d

1. Entrada: A matriz dependência D de n x m e o determinante **det** de uma sub-matriz arbitraria de D de *rank* r = n.
2. Inicializa as matrizes U e V, $U_{n \times n} = I_{n \times n}$, $V_{(m+n) \times (m+n)} = I_{(m+n) \times (m+n)}$, onde $I_{l \times l}$ é igual a matriz identidade de dimensão l.
3. Construção da matriz ampliada $A = [\, D \mid \det I_{n \times n} \,]_{n \times (m+n)}$ pela adição de n colunas de uma matriz quadrada de det I para a matriz $D_{n \times m}$. A_{i*} e A_{*j} representam a i-ésima linha e a j-ésima coluna da matriz A, respectivamente.
4. **FOR** l = 1 **to** min(n,m) **do**:

a. Considerando a sub-matriz de A formada pelas linhas l, ..., n e as colunas l, ... ,m. Procurando o elemento A_{ij}, com o menor valor absoluto, tal que $i \geq l$, $j \geq l$ e $A_{ij} \neq 0$. Quando tal elemento não é encontrado, *Pare*.

b. Troca as linhas A_{l*} e A_{i*} e então as colunas A_{*l} e A_{*j} para trazer os elementos A_{lj} sobre a diagonal. Troca as linhas U_{l*} e U_{i*}, e as colunas V_{*l} e V_{*j} nas matrizes U e V, respectivamente.

c. **FOR** i = l + 1 **to** n **do**: (redução das colunas)

$A_{i*} = A_{i*} - \lfloor A_{li} / A_{ll} \rfloor A_{l*}$

$U_{i*} = U_{i*} - \lfloor U_{li} / U_{ll} \rfloor U_{l*}$

END_FOR i

d. **FOR** j = l + 1 **to** (m+n) **do**: (redução das linhas)

$A_{*j} = A_{*j} - \lfloor A_{lj} / A_{ll} \rfloor A_{*l}$

$V_{*j} = V_{*j} - \lfloor V_{lj} / V_{ll} \rfloor V_{*l}$

END_FOR j

e. Quando todos os elementos não diagonais na linha l e coluna l são zero, prosseguimos para o próximo l-ciclo(4), de outro modo repetimos apartir do passo 4a.

f. Realiza os passos 4e e 4f do Algoritmo 9.1

END_FOR l

9.4. MÁXIMA PARTIÇÃO INDEPENDENTE

De acordo com o Lema 9.3 a máxima partição independente é caracterizada pelos *rótulos*. Nesta seção um conjunto de *rótulos* é computado para a matriz dependência triangular. Então alguns *rótulos* são usados para gerar os conjuntos de dependência máximo para a matriz dependência original. Na seção 9.5 é mostrado como essas fórmulas podem ser usadas para converter laços DO aninhados em um aninhamento de laços DO_ALL.

9.4.1. SELEÇÃO DO *RÓTULO*

Considerando uma matriz dependência quadrada triangular superior $D'_{n \times n} = [\vec{d}'_1 \ldots \vec{d}'_n]$ com r = *rank*(D') ≤ n e tal que (n-r) colunas são zero. A matriz D^t é gerada pelos algoritmos da seção anterior. Se m > n, então as colunas de zeros n+1, n+2, ... ,m são desconsideradas para obter uma matriz quadrada.

Um conjunto de dependências máximo P_k com rótulo I^k_0 é descrito pelas equações (2):

$$
\begin{aligned}
I_1 &= I^k_{0,1} + d'_{11}y_1 \\
&\ldots \\
I_i &= I^k_{0,i} + d'_{i1}y_1 + \ldots + d'_{ii}y_i \\
&\ldots \\
I_n &= I^k_{0,n} + d'_{n1}y_1 + \ldots + d'_{ni}y_i + \ldots + d'_{nn}y_n.
\end{aligned}
\qquad (3)
$$

O *Rótulo* I^k_0 é calculado como segue:

- Para i = 1:

$$
I^k_{0,1} = \begin{cases} I_1 \bmod d'_{11} & \text{if } d'_{11} > 0 \\ I_1 & \text{if } d'_{11} = 0 \end{cases}
$$

$$
y_1 = \begin{cases} (I_1 - I^k_{0,1}) / d'_{11} & \text{if } d'_{11} > 0 \\ 0 & \text{if } d'_{11} = 0 \end{cases}
\qquad (4)
$$

- Para i = 2 até n:

$$
I^k_{0,1} = \begin{cases} (I_i - \sum_{j=1}^{i-1} d'_{ij} y_j) \bmod d'_{ii} & \text{if } d'_{ii} > 0 \\ I_i & \text{if } d'_{ii} = 0 \end{cases}
$$

$$y_i = \begin{cases} (I_i - \sum_{j=1}^{i-1} d'_{ij} y_j) / d'_{ii} & if\ d'_{ii} > 0 \\ 0 & if\ d'_{ii} = 0 \end{cases} \quad (5)$$

Teorema 9.2: O conjunto *rótulo* **L** = $\{I^k_0\}$ é próprio, isto é, cada ponto tem um único *rótulo*.

Corolário 9.3: O conjunto *rótulo* **L** gera uma máxima partição independente.

Lema 9.4: As matrizes dependência D e D^t descrevem as dependências de alguns conjuntos índices **I**.

A cardinalidade do conjunto *Rótulo* é dada pelas equações (4) e (5), onde o i-ésimo componente do k-ésimo *rótulo* $I^k_{0,i}$ pode ter um valor inteiro arbitrário na seguinte faixa:

$$I^k_{0,i} \in \begin{cases} [0, d'_{ii} - 1] & if\ d'_{ii} > 0 \\ Z & if\ d'_{ii} = 0 \end{cases} \quad (6)$$

Como cada *rótulo* representa um conjunto diferente de dependências, o número de conjuntos paralelos são iguais a cardinalidade de $|\mathbf{L}|$. Isto é dado pela seguinte particionalidade do laço

$$|L| = \prod_{i=1}^{n} d'_{ii} \quad \text{se } rank(D) = n$$

$$|L| = \infty \quad \text{se } rank(D) < n \quad (7)$$

Observe que para as matrizes quadradas, as matrizes dependência D são não singulares, e a particionalidade é dada pelo $|\mathbf{L}|$ = **det** (D).

9.5. CONVERSÃO PARA CÓDIGO DO_ALL

Um laço paralelo é criado pelo circundaneamento do laço original com um aninhamento de laços DO_ALL. Estes laços geram os elementos de cada *rótulo* $I_o \in$ **L**. Dentro dos laços DO_ALL os paramentos do laço original são alterados pelos passos através das iterações com o mesmo *Rótulo* I_o em uma ordem lexicográfica.

A conversão considera os limites do laço normalizados 1 e N_i, para i = 1, ... , **n**, pelas seguintes mudanças no conjunto *rótulo* **L** e na matriz dependência D' [D'Hollander 1992]:

1) A primeira mudança torna os laços paralelos, apartir de uma instância de zero. Entretanto, os componentes zero $I_{0,i} = 0$ são alterados pelo $I_{0,i} = d'_{ii}$. Isto é, o *rótulo* $[0 \ldots 0]^T$ é trocado pelo *rótulo* equivalente $[d'_{ii} \ldots d'_{nn}]^T$ do mesmo conjunto dependência. Neste caso, os componentes do *rótulo* $I_{0,i}$ obtêm os valores no intervalo $[1, d'_{ii}]$ ao invés de $[0, d'_{ii}-1]$.

2) A segunda mudança evita uma singularidade no algoritmo de conversão quando *rank*(D')<**n**. Entretanto, os elementos zero da diagonal da matriz dependência D' são trocados pelos limites do laço na correspondente dimensão, isto é, $d'_{ii} = 0$ tornando-se $d'_{ii} = N_i$. Isto garante N_i conjuntos de dependência paralelos na dimensão i.

A conversão prossegue em dois passos. Primeiro, os $|\mathbf{L}|$ laços paralelos são criados pelo seguinte código:

```
DO_ALL I0,1 = 1,d'11
    DO_ALL I0,2 = 1,d'22
        ⋮
        DO_ALL I0,n = 1,d'nn
            { Corpo do Laço }                    (8)
        END_DO_ALL
        ⋮
    END_DO_ALL
END_DO_ALL
```

Então os limites do laço original são ajustado para gerar apenas as iterações com *rótulo* I_0, isto é, cujos índices satisfaçam as equações (3). O laço serial mais externo implementa as equações dependência $I_1 = I_{0,1} + d'_{11} y_1$. A distância entre os sucessivos I_1 valores é $d'_{11} \geq 1$ e o valor mínimo de $I_1 \geq 1$ é $I_{0,1}$. Isto gera

```
DO I1 = I0,1, N1, d'11
    y1 = (I1 - I0,1) / d'11                      (9)
END_DO
```

A i-ésima equação dependência lida é $I_i = I_{0,i} + \sum_{j=1}^{i-1} d'_{ij}\, y_j + d'_{ii}\, y_i$. As variáveis $I_{0,i}$ e y_j, para j = 1, ... ,i-1 são fixadas pelos laços antigos e somente as variáveis livres, y_i, causam variação em I_i com um valor positivo d'_{ii}. O valor mínimo de $I_i \geq 1$ que satisfaz esta equação é

$$I_{i,\min} = 1 + (I_{0,i} - 1 + \sum_{j=1}^{i-1} d'_{ij}\, y_j) \bmod d'_{ii} \qquad (10)$$

Entretanto, o i-ésimo laço DO serial interno torna-se

DO $I_i = I_{i,\min}$, N_i, d'_{ii}

$$y_i = (I_i - I_{0,i} - \sum_{j=1}^{i-1} d'_{ij}\, y_j) / d'_{ii} \qquad (11)$$

END_DO

Os *rótulos* gerados pelos laços DO_ALL protegem todos os conjuntos de dependências e os laços seriais criam as iterações com *rótulos* idênticos na ordem lexicográfica. Consequentemente, a faixa de iterações e a direção das dependências não é alterada, obedecendo o laço aninhado original, conforme D'Hollander [1992].

CAPÍTULO 10 - INTERFACE DE PASSAGEM DE MENSAGENS (MPI)

Neste capítulo, descrevemos a interface de passagem de mensagens (MPI - *Message Passing Interface*), utilizada como plataforma de comunicação entre os processadores.

10.1. INTRODUÇÃO

Processar uma tarefa paralelamente significa, de maneira simples, dividir esta tarefa de forma que ela execute em diversos processadores. Portanto, um programa paralelo pode ser visto como um conjunto de processos atuando em conjunto para resolver uma determinada tarefa.

Para que haja cooperação entre os diversos processos executando esta mesma tarefa, estes devem comunicar-se para que troquem informações e sincronizem-se. Um paradigma muito usado na implementação de programas paralelos, trata-se do ***message passing***, que oferece uma maneira de comunicação entre processos que não compartilham o mesmo espaço de memória.

10.2. DEFININDO *MESSAGE PASSING*

Arquiteturas MIMD, que relacionam as arquiteturas paralelas cujas unidades de processamento são independentes entre si, podem ser classificadas em relação a distribuição de memória entre os processadores. Classificam-se em:

- *Arquiteturas de memória centralizada*: os processadores compartilham uma única memória;
- *Arquiteturas de memória distribuída*: cada processador possui uma memória local. É interessante estender este conceito para que englobe também as redes de estações de trabalho, que apresentam a mesma natureza de distribuição de memória. Esquematicamente, pode-se representar tal modelo através da figura 10.1.

Como já foi citado, processadores trabalhando em conjunto em uma aplicação devem comunicar-se, de maneira que estes cooperem entre si. Quando se possui memória compartilhada, pode-se conseguir tal comunicação através de espaços de memória compartilhados entre os diversos

processos paralelos. Porém, em caso de memória distribuída, quando cada processador possui sua própria memória local, devem ser definidas primitivas explícitas que possibilitem que os processos se comuniquem, ou que possam acessar posições de memória não locais a ele.

Para esse fim, é definido um conjunto de primitivas que permitem que os processos troquem mensagens entre si, requerendo explicitamente dados de outros processadores. Estas primitivas de comunicação entre processos, caracterizam o paradigma ***message passing***.

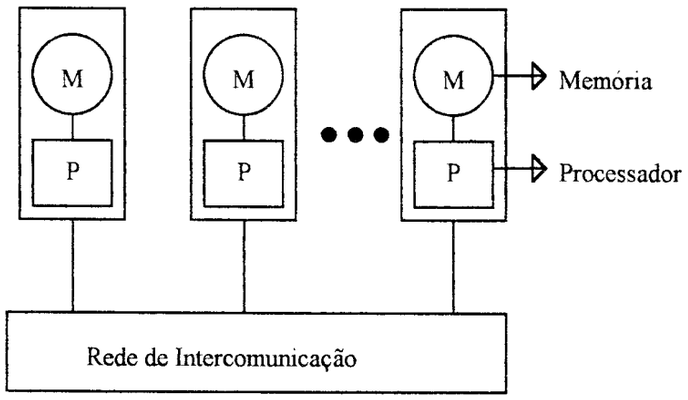


Figura 10.1. Modelo Genérico para Memória Distribuída

Segundo Quealy [], temos a seguinte definição:

"***Message Passing*** é um método para a comunicação de processos quando não há compartilhamento de memória. Ela é necessária porque:

- Memórias são locais aos processadores;
- Não há compartilhamento de variáveis;
- Única maneira de se conseguir dados de outras memórias."

O paradigma ***message passing*** tem-se tornado extremamente popular em tempos recentes, e podemos relacionar alguns fatores que justificam a sua grande aceitação:

- Programas ***message passing*** podem ser executados em uma grande variedade de plataformas, como arquiteturas paralelas de memória distribuída e redes de estações de trabalho. Apesar deste tipo de paradigma não se adaptar naturalmente a arquiteturas de memória centralizada, não há um fator que inviabilize a sua execução neste tipo de arquitetura;
- O paradigma é facilmente entendido e utilizado;

- Adequam-se naturalmente a arquiteturas escaláveis. Segundo McBryan [1994], a capacidade de aumentar o poder computacional de maneira razoavelmente proporcional ao aumento de componentes de um sistema, o que caracteriza arquiteturas e algoritmos escaláveis, trata-se de um fator chave para o desenvolvimento da computação paralela para que seja possível a conquista dos grandes desafios computacionais do final do século. Arquiteturas realmente escaláveis atualmente são as arquiteturas de memória distribuída, que são as arquiteturas em que melhor se caracteriza o paradigma ***message passing***.
- Não deve-se tornar obsoleto por redes mais rápidas ou arquiteturas que combinem memória compartilhada e memória distribuída. A algum nível, sempre será necessário utilizar-se de alguma maneira ***message passing***. [Dongarra *et al.* 1995]

10.3. *MESSAGE PASSING* EM AMBIENTES PARALELOS

Um programa ***message passing*** pode ser definido, de maneira simples, como um conjunto de programas seqüenciais, distribuídos em vários processadores, que se comunicam através de um conjunto limitado e bem definido de instruções. Tais instruções formam o *ambiente message passing*. Estas instruções são disponíveis através de uma *biblioteca message passing*, conforme McBryan [1994].

Não necessariamente os programas nos diversos processadores devem ser distintos. Pode-se utilizar (e na maior parte dos casos o é) o paradigma SPMD (*Single Program - Multiple Data*). Tal paradigma implica que seja distribuído pelos processadores o mesmo programa, e cada processador o execute de maneira independente, o que implica que diferentes partes deste programa são executados em cada um dos processadores. Quando se distribui programas distintos para os processadores, utiliza-se o paradigma MPMD (*Multiple Program - Multiple Data*).

Visto que, em um ambiente MPMD, os diferentes programas podem ser unidos em um só, excetuando-se um eventual gasto maior de memória em cada processador, praticamente não há perda de performance na utilização do paradigma SPMD. Ganha-se, nesse caso, na simplicidade de gerenciamento de um sistema deste tipo.

Um ambiente de programação ***message passing*** é então formado da seguinte maneira:

- uma linguagem seqüencial padrão, como C e Fortran, a qual será utilizada na programação dos programas seqüenciais nos processadores e
- a biblioteca ***message passing***

Os primeiros ambientes de programação ***message passing*** surgiram junto com as arquiteturas paralelas de memória distribuída. A medida que fabricantes lançavam novos modelos no mercado, lançavam seus respectivos ambientes de programação, que geralmente eram incompatíveis entre si por serem ligados a arquitetura que os gerou.

Esta especificidade arquitetura/ambiente tornava a portabilidade de programas entre sistemas diferentes de difícil, senão impossível, execução.

Para a solução deste tipo de problema, surgiram as chamadas **plataformas de portabilidade**. Nesse caso, define-se um ambiente e implementa-se em várias arquiteturas, possibilitando que programas possam ser portados de maneira fácil e direta. Além disso, estes tipos de plataformas adequam-se de maneira natural a ambientes heterogêneos, que são formados pela ligação de diferentes arquiteturas em um ambiente único. Podem ser citados como exemplos: EXPRESS, Linda, p4, PARMACS, ZipCode e principalmente o PVM. Este último, trata-se da plataforma que alcançou maior aceitação e atualmente, pode ser descrito como um padrão "de fato" de programação neste tipo de sistema.

Porém, o grande número de plataformas de portabilidade existentes gera um problema semelhante ao de ambientes específicos a arquiteturas. Além disso, grande parte das plataformas de portabilidade suportam apenas um subconjunto das características de determinadas arquiteturas, ocasionando o não uso de características importantes de algumas delas.

Levando-se em consideração estes aspectos dentre outros, iniciou-se um processo de padronização para plataformas de portabilidade, que agregou vários representantes de várias organizações, principalmente européias e americanas. Este padrão, que não é apoiado por nenhuma organização oficial, foi nomeado **MPI** (*Message Passing Interface*).

O **MPI** foi baseado nas melhores características de todas as plataformas de portabilidade, levando-se em consideração as características gerais das arquiteturas paralelas, de maneira que não se desperdiçasse as qualidades de cada arquitetura.

10.4. BIBLIOTECA *MESSAGE PASSING* GENÉRICA

Quealy [] descreve um conjunto de rotinas básicas que formam uma biblioteca ***message passing***. É importante ressaltar que a descrição apresentada é feita de maneira genérica, e o

conjunto de rotinas e as respectivas sintaxes podem variar de acordo com o ambiente de programação.

10.4.1. O QUE É UMA MENSAGEM

Antes de se definir cada uma das rotinas, é importante definir precisamente o que é uma mensagem, e as informações que se necessita saber sobre cada operação de transferência de mensagens entre processos.

Qualquer tipo de informação que deve ser transferida entre processos que não compartilhem memória, deve ser transportada explicitamente via mensagens. Nesse caso, pode-se dizer que o processo necessita de um dado que está fora do seu alcance, ou seja, não está armazenada em sua memória local. Portanto, uma transferência de mensagens pode ser entendida, de maneira geral, como um acesso a memória não local. Por exemplo, se um processo executando no processador *Pn*, necessita de um dado do processo executando no processador *P0*, então este dado será transferido explicitamente da memória local a *P0* e será copiado na memória local a *Pn*. Este modelo genérico é mostrado na figura 10.2.

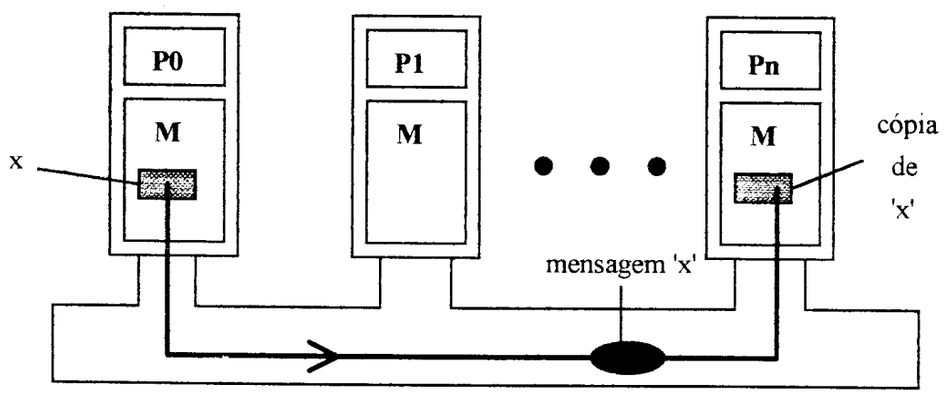


Figura 10.2. Transferência de uma Mensagem

Para uma operação de transferência de dados entre processos, há um conjunto de informações que devem ser controladas, de maneira que tal operação esteja completamente definida. Estas informações são:

- Qual processo está enviando a mensagem (processo fonte);

- Qual os dados que formam esta mensagem;
- Qual o tipo, ou tipos, dos dados (p. ex. *char*);
- Qual o tamanho da mensagem;
- Qual processo vai receber a mensagem (processo destino);
- Aonde os dados serão armazenados no processo receptor;
- Qual a quantidade de dados suportado pelo receptor, de maneira que não se envie dados que o receptor não esteja preparado para receber;

Estas informações devem ser supervisionadas pelo sistema que está gerenciando a transferência, e alguns desses dados devem ser anexados a mensagem, a fim de que o processo receptor possa reconhecer as mensagens e trata-las de forma conveniente.

Basicamente, uma mensagem deve ter o seguinte formato (figura 10.3):

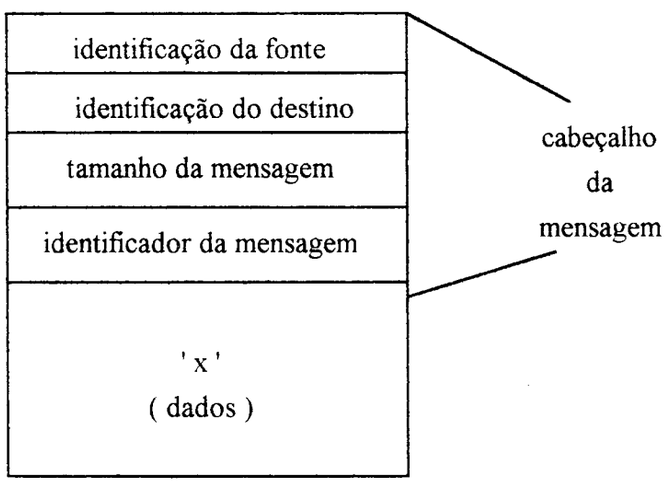


Figura 10.3. Mensagem Genérica

Uma informação que também deve ser anexada a uma mensagem, trata-se de um identificador, a partir do qual a mensagem possa ser selecionada pelo processo receptor para recebimento. O receptor pode utilizar outras informações, como o identificador do processo fonte, para a seleção de mensagens. Basta, por exemplo, que esta informação seja armazenada no identificador. Um exemplo simples é mostrado na figura 10.4, onde uma mensagem contendo a variável *p*, enviada pelo processo 0, é selecionado pela cadeia de caracteres "tipo" no processo 1.

Processo 0	Processo 1
p = ...	
send("tipo", p , 1)	*receive*("tipo", p)
*	*
*	*
*	*

Figura 10.4. Exemplo de Identificador de Mensagem

10.5. MPI - *MESSAGE PASSING INTERFACE*

O MPI trata-se de uma tentativa de padronização, independente de arquiteturas, para ambientes de programação ***message passing***. Como já foi apresentado (seção 10.3 - ***Message passing*** em ambientes paralelos), o MPI se presta a resolver alguns problemas relacionados as chamadas plataformas de portabilidade, como por exemplo:

- o grande número de plataformas existentes ocasiona restrições em relação a real portabilidade de programas;
- o mau aproveitamento de características de algumas arquiteturas.

Walker [1995] relaciona, de maneira mais geral, uma série de motivos que explicam a necessidade de um padrão para este tipo de sistema:

- **Portabilidade e facilidade de uso**: A medida que aumente a utilização do MPI, será possível portar transparentemente aplicações entre um grande número de plataformas;
- **Fornecer uma especificação precisa**: Desta maneira, fabricantes de *hardware* podem implementar eficientemente em sua máquinas um conjunto bem definido de rotinas. Similarmente, ferramentas podem ser construídas baseadas no MPI;
- **Necessidade de crescimento da indústria de *software* paralelo**: A existência de um padrão torna a criação de *software* paralelo (ferramentas, bibliotecas, aplicativos, etc.) por empresas independentes uma opção comercialmente viável;
- **Gerar um maior uso de computadores paralelos**: O crescimento da indústria de *software* paralelo, implica em maior difusão do uso de computadores paralelos.

Procurou-se aproveitar todas as melhores características de cada uma das plataformas de portabilidade, de maneira que todo o esforço e estudo que já havia sido despendido neste tipo de plataforma fosse aproveitado, levando-se também em consideração o aspecto eficiência em todas as arquiteturas. O problema da eficiência em várias arquiteturas complica-se quando se enxerga que determinadas operações são feitas de maneiras diferentes em diferentes arquiteturas e por diferentes protocolos.

Nesse caso, deve-se pensar várias maneiras de se fazer estas determinadas operações, de modo que não se desperdice qualidades intrínsecas a cada uma das arquiteturas. Por isso, algumas rotinas no MPI são implementadas de várias formas, o que gera uma certa complexidade. Porém, de certa maneira, tal problema é inevitável quando procura-se determinar um padrão que englobe diferentes tipos de plataformas. Operações coletivas recaem neste tipo de problema.

10.5.1. Comunicação no MPI

10.5.1.1. O que forma uma mensagem

Uma mensagem no MPI é definida como um vetor de elementos de um determinado tipo. Ao se enviar uma mensagem, deve-se indicar o primeiro elemento deste vetor e o número de elementos que o formam. Estes elementos devem ser de tipos iguais.

Mensagens no MPI são tipadas, de tal maneira que deve-se indicar na mensagem o tipo dos elementos sendo enviados. O MPI define tipos básicos, que são definidos de maneira análoga a linguagem de programação C. Alguns tipos são exemplificados na figura 10.5.

MPI_CHAR	caracter
MPI_INT	inteiro
MPI_FLOAT	ponto flutuante

Figura 10.5. Tipos Básicos no MPI

A obrigação de tipar-se mensagens vem da necessidade de se garantir a conversão entre sistemas heterogêneos.

O MPI permite que se crie tipos definidos pelo usuário, de maneira que se possa enviar mensagens compostas por elementos de tipos distintos, como por exemplo, estruturas (*structs*).

10.5.1.2. COMUNICAÇÃO PONTO-A-PONTO

Um processo MPI é definido por um grupo e um *rank* dentro deste grupo. Uma mensagem é definida por um identificador (que o MPI define como *tag*) e por um contexto. Portanto todas essas informações devem ser anexadas a uma mensagem, conforme Walker [1995].

É importante ressaltar a definição empregada pelo MPI para rotinas bloqueantes. Uma rotina bloqueante define que um processo será bloqueado até que se termine a operação de envio ou recepção de uma mensagem, como pode ser visto na figura 10.6:

Rotinas MPI
send() bloqueante
send() não bloqueante
receive() bloqueante
receive() não bloqueante

Figura 10.6. Rotinas de Comunicação Ponto-a-Ponto no MPI

A rotina **send()**, de maneira genérica, é definida por:

MPI_SEND (buf, count, tipo, dest, tag, comm)

onde:

buf : endereço inicial da mensagem

count: número de elementos da mensagem

tipo: tipo dos dados (pode ser um tipo definido pelo usuário)

dest: *rank* do processo destino

tag: identificador da mensagem

A maneira eficiente de implementação de uma rotina **send()**, depende de certa maneira do protocolo e da arquitetura sobre o qual o MPI está executando. Para garantir a eficiência em

qualquer plataforma, o MPI define vários modos de comunicação, que definem diferentes semânticas para a rotina. Todos os modos, apresentam versões bloqueantes e não bloqueantes.

Esses modos são:

- **síncrono**: utiliza uma semântica *rendezvous*, isto é, o transmissor deve esperar uma confirmação de recepção da mensagem;
- **padrão**: envia, sem se preocupar se o **receive()** correspondente foi ativado, e sem que a mensagem seja explicitamente *bufferizada* pelo MPI;
- ***bufferizado***: similar ao modo padrão, diferindo no fato de existirem *buffers* criados explicitamente pelo programador;
- ***ready***: similar ao modo padrão, excetuando-se que o **receive()** correspondente deve ter sido obrigatoriamente iniciado.

A rotina **receive()** é definida por:

MPI_RECV(buf, count, tipo, fonte, tag, comm, status)

onde:

buf : endereço inicial para o armazenamento dos dados recebidos

count: número máximo de elementos que podem ser recebidos

tipo: tipo do dado

fonte: *rank* do processo fonte

tag: identificador da mensagem

comm: *communicator*

status: informações sobre última mensagem

Uma mensagem é selecionada para recebimento pelo *rank* do processo que a enviou e pelo seu *tag* (respeitando-se o contexto). Esses dois valores podem ser *wild-cards*, isto é, pode-se utilizar rotinas que recebam de qualquer processo qualquer mensagem. Nesse caso, o programador pode utilizar-se dos valores retornados na variável *status*, que indica estes dois valores.

Já os *communicators* não aceitam valores *wild-card*, de tal maneira que não existe uma situação em que um processo pertença a vários contextos.

A rotina **receive()** é implementada apenas no modo padrão, bloqueante ou não bloqueante.

10.5.1.3. COMUNICAÇÃO COLETIVA

O MPI define vários tipos de rotinas coletivas, que são rotinas *broadcast*, de operações globais e de sincronização. Para rotinas *broadcast*, algumas variantes foram definidas: rotinas *gather*, *scather*, e combinação das duas anteriores. Uma representação genérica destas funções é apresentada na figura 10.7, conforme Dongarra *et al.* [1995].

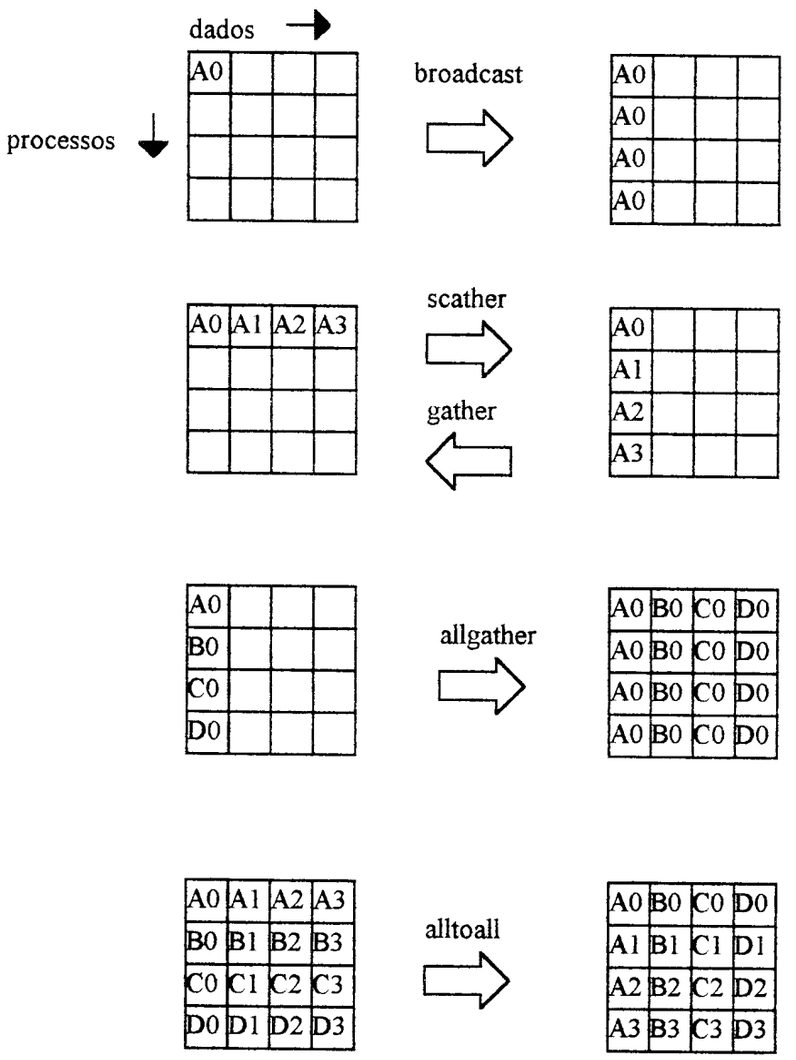


Figura 10.7. Variações sobre a Rotina *Broadcast*

Esta grande variedade de funções coletivas se explicam pelo fato de se garantir operações coletivas eficientes em todas as arquiteturas. Todas as operações coletivas no MPI (excetuando-se as de sincronização) são bloqueantes e executam no modo padrão.

CAPÍTULO 11 - IMPLEMENTAÇÃO

Os métodos de paralelização, apresentados nos capítulos 6, 7, 8 e 9, foram aplicados em 02 (dois) algoritmos de laços seqüenciais (**Anexo I**), o que gerou 09 algoritmos de laços paralelos (**Anexo II**). De cada algoritmo de laço original foram gerados 04 algoritmos paralelos, um para cada método, com exceção para o algoritmo Laço 2 que, pelo método Alocação de Dados, obteve-se 02 (dois) algoritmos paralelos (um de granularidade grossa e outro fina). Isto é devido ao fato de que este método permite a estratégia de termos dados duplicados ou não, na memória local dos processadores.

A conversão dos algoritmos de seqüencial para paralelo, foi efetuada manualmente, utilizando-se as regras de conversão, e observando-se as condições e restrições, de cada um dos métodos em questão, conforme apresentado nos capítulos 6, 7, 8 e 9.

A avaliação dos algoritmos paralelizados, foi realizada implementando-se os referidos algoritmos em linguagem "C" *padrão*, e adequados ao ambiente de passagem de mensagens, utilizando as rotinas da interface de passagem de mensagens (MPI), contidas na biblioteca "MPI.H".

Com isto, obtivemos 09 programas paralelos escritos em linguagem "C" *padrão* (**Anexo III**), os quais podem ser transportados para qualquer ambiente, desde que este aceite a linguagem "C" *padrão*, e também as rotinas de passagem de mensagem do padrão MPI.

Os programas foram compilados com o MPICC, que é um compilador próprio para a linguagem "C" com uso das rotinas do MPI, e executados com o MPIRUN, que estabelece todo o ambiente para a troca de mensagens utilizando a interface de passagens do MPI.

Nos programas foram utilizadas as rotinas de comunicação MPI do tipo bloqueante, ou seja, um processo (processador) origem que efetuou um ***Send*** (envio) só é liberado para continuar a execução, após o processo (processador) destino ter efetuado um ***Receive*** (recebido) correspondente, desta forma garantiu-se o sincronismo das operações.

Os programas paralelos foram adaptados à arquitetura *Mestre-Escravo*, sendo 01 (um) processador *Mestre* e os demais processadores *Escravos*.

Todos os programas-laço foram particionados, de modo a termos vários processos/processadores (*Escravo*) executando simultaneamente. Após a execução ter sido

completada, o processo *Mestre* obtém o resultado dos processos *Escravos*, reorganiza a matriz de dados e particiona novamente o laço, enviando para os processos *Escravos*, até que todas as iterações, que compõem o espaço de iteração do laço, tenham sido concluídas.

A simulação dos programas foi realizada em um sistema com processadores virtuais do tipo Sun4, na arquitetura *Mestre-Escravo*, utilizando o processamento paralelo simulado, pois utilizou-se apenas um processador físico, e considerando que cada processo (*Mestre* ou *Escravo*) tem seu próprio processador Sun4 virtual dedicado.

Na execução desta simulação foi utilizado um computador, com processador 486 DX/2 da Intel trabalhando a uma freqüência de 66Mhz, com 16 Mbytes de memória principal, executando o sistema operacional LINUX.

Antes dos programas paralelizados terem sido simulados na arquitetura com mais de um processador, eles foram executados seqüencialmente, de modo a comprovar sua execução, ou seja, verificar se geravam o mesmo resultado. Esta verificação foi feita comparando-se uma porção da matriz de dados gerada pelo laço original, executado seqüencialmente, com a mesma porção gerada pelo laço paralelizado, também executado seqüencialmente, e bem como o número de iterações realizadas em ambos os laços.

Comprovada a sua execução no modo seqüencial, os programas foram executados na arquitetura paralela simulada. E, para a conferencia da execução paralela, foi novamente comparada a porção da matriz resultado, com a mesma já obtida anteriormente mediante a execução seqüencial.

Como a proposta era a de fazer uma avaliação dos métodos, a fim de obtermos uma comparação entre eles, levando em conta a sua aplicabilidade em um processo computacional, desta forma a análise em tempo real é irrelevante, ou seja dispensável.

Em virtude dos programas paralelos terem sido executados na mesma máquina/ambiente, os resultados não são destoantes para efeito de comparação, já que foram submetidos às mesmas condições de execução.

As características dos laços são: o laço 1 é composto por três aninhamentos, e uma atribuição contendo 04 referências a uma matriz tridimensional, totalizando 4096 iterações; e o laço 2 que é composto de dois aninhamentos, e uma atribuição contendo 01 referências a uma matriz bi-dimensional, totalizando 256 iterações.

CAPÍTULO 12 - RESULTADOS

Os valores da tabela 12.1 e 12.2 são os valores médios, obtidos mediante a execução simulada, com 2, 3, 5, 7, 9 e 11 processadores.

O tempo total é o tempo gasto para a execução total do programa paralelo.O tempo gasto pelo processador *Mestre* para transmitir e receber os dados para/de os processadores *Escravos*, é o tempo de comunicação.O tempo de execução é a diferença entre o tempo total e o tempo de comunicação, ou seja, o tempo que o processador *Mestre* gasta para executar a partição do laço para os processadores *Escravos,* e para reorganizar os dados recebidos.

A quantidade de comunicação é o número de vezes que uma operação de comunicação ocorre na execução do programa.A quantidade de *Broadcast* é o número de vezes que uma operação de *Broadcast* é executada na execução paralela, isto ocorre quando o processador *Mestre* envia para todos os processadores *Escravos* o vetor/matriz de dados.Por sua vez, a quantidade de dados comunicados é a quantidade de bytes comunicados, que é a soma da quantidade de dados transmitidos com a quantidade de dados recebidos.

O número de processadores possíveis representa o número de processadores que podemos ter executando simultaneamente na arquitetura, para aquele laço paralelizado, conforme o método assim o permitir..

O *Slowdown* é a razão entre o tempo de execução seqüencial e o tempo de execução paralelo, ambos obtidos via simulação com processadores virtuais Sun4.O *Slowdown* Relativo corresponde á relação entre o tempo de execução com 2 processadores (um *Mestre* e outro *Escravo*) e o tempo de execução médio com **N** processadores.

Para o Laço 1 (Anexo I), o tempo de execução seqüencial é de **0,00323** segundos. O programa paralelo que melhor tem resultado é o que foi gerado pelo método de Particionamento & Rotulação, o qual teve um tempo total na simulação de **18,54463** segundos; **0,00883** segundos para o tempo de execução; são transmitidos **16540** bytes de dados entre os processadores em **156** comunicações do tipo bloqueante, como pode ser observado na tabela 12.1 abaixo.

Pelo método Particionamento & Rotulação temos que o processador *Mestre* executa 52 iterações do Laço, o que denota que poderíamos ter 52 processadores executando simultaneamente, ao invés dos que foram utilizados na simulação.

Métricas LAÇO 1	Hiperplano	Particona. & Rotulação	Transf. Unimodulares	Alocação de Dados	
				Com Duplic.	Sem Duplic.
Tempo Total	1.045,42631	**18,54463**	1.097,08092	1.080,73810	-
Tempo de Comunicação	1.044,98056	18,53580	1.096,61760	1.080,29277	-
Tempo de Execução	0,44575	**0,00883**	0,46331	0,44533	-
Quant. Comunicação	8.192	**156**	8.192	8.192	-
Quant. BroadCast	0	1	0	0	-
Quant. Dados Comun.	20.480	**16.540**	20.480	20.480	-
Quant.Dados Trans	16.384	156	16.384	16.384	-
Quant. Dados Rec.	4.096	16.384	4.096	4.096	-
Slowdown	0,00725	0,36415	0,00696	0,00732	-
No. Proc. Possíveis	4.096	**52**	4.096	4.096	-
Slowdown Relativo	1,00099	0,99766	1,00199	1,00286	-

Tabela 12.1. Dados Médios obtidos para o LAÇO 1, via simulação com 2,3,5,7,9,11 processadores virtuais

No caso do Laço 2 (Anexo II), o tempo de execução seqüencial é de **0,00016** segundos e os melhores resultados foram obtidos com os programas paralelizados pelos métodos de Particionamento & Rotulação e o de Alocação de Dados (Sem Dados Duplicados). Como pode ser visto na tabela 12.2, os resultados de ambos os métodos citados, tem um valor muito próximo, mas o método Alocação de Dados (Sem Dados Duplicados) tem o menor tempo de execução (**0,00330** segundos); o menor tempo médio de comunicação (**0,08313** segundos) e o menor número de iterações (**31**), enquanto que o método Particionamento & Rotulação o menor tempo total (**6,19776** segundos); a menor quantidade de comunicações entre os processadores (**64**) e a maior quantidade de bytes comunicados (**832**) entre o processador *Mestre* e os processadores *Escravos*.

Métricas LAÇO 2	Hiperplano	Particiona. & Rotulação	Transf. Unimodulares	Alocação de Dados	
				Com Duplic.	Sem Duplic.
Tempo Total	68,75948	**6,19776**	68,91596	67,68696	7,73453
Tempo de Comunicação	68,73349	6,19436	68,88980	67,66165	7,73123
Tempo de Execução	0,02598	0,00340	0,02616	0,02531	**0,00330**
Quant. Comunicação	512	**64**	512	512	93
Quant. BroadCast	0	1	0	0	1
Quant. Dados Comun.	**512**	**832**	**512**	**512**	799
Quant.Dados Trans	256	64	256	256	31
Quant. Dados Rec.	256	768	256	256	768
Slowdown	0,00427	0,03266	0,00419	0,00436	0,03280
No. Proc. Possíveis	256	32	256	256	**31**
Slowdown Relativo	1,00959	1,00894	1,00134	1,01468	0,98107

Tabela 12.2. Dados Médios obtidos para o LAÇO 2, via simulação com 2,3,5,7,9,11 processadores virtuais

CAPÍTULO 13 - CONCLUSÃO

13.1. DISCUSSÃO GERAL

A paralelização de laços é de extrema importância, pois tomam grande tempo de computação, de modo que quanto menor for o tempo despendido melhor. Isto acarreta um ganho em tempo de execução, que é o objetivo básico do processamento paralelo.

A paralelização de laços envolve duas fases distintas: primeira, extrair as dependências das iterações do laço, e depois aplicar o método de paralelização em questão, a fim de tornarmos os laços paralelos. Dependendo do método aplicado, podemos ou não utilizar todas as dependências extraídas do corpo do laço para proceder a partição, ou seja, secionar o espaço de iteração, de modo a executarmos simultaneamente várias porções, ver tabela 13.1, criando a frente de onda na execução do laço

Os métodos que exploram a granularidade fina, particionam o laço em pequenas porções (basicamente uma iteração do laço por vez), ao passo que os de granularidade grossa, fazem o particionamento em porções grandes (normalmente laços).

A adoção da arquitetura MIMD com memória distribuída, permite que utilizemos estes conceitos e sistema, e também o ambiente MPI, em uma arquitetura que é composta por computadores isolados interligados por um meio físico, formando uma rede de computadores.

O problema de adotarmos computadores interligados por rede, é que temos um aumento considerável na quantidade de informações que trafegam pela mesma. Isto é ocasionado pelo protocolo de comunicação e pelo controle de verificação de erros (bit de paridade, CRC, etc.), que ocasionam o aumento do *Overhead* de comunicação, sem contar com o problema de tráfego, dentre outros aspectos, e que não são pertinentes a este trabalho. Desta forma, a utilização de granularidade fina é desaconselhável, pois o número de comunicação é grande, e a quantidade de dados comunicados é pequena.

Outro fator que impossibilita esta utilização é que os computadores são bastante complexos para simplesmente fazerem operações aritméticas, como estas que estão envolvidas no corpo do laço, ocasionando um custo operacional alto.

Quando utilizamos paralelismo de granularidade grossa, saímos do pressuposto que os processadores tenham maior poder computacional, ou seja, podendo efetuar operações complexas. Eles devem tratar com laços e não somente com a(s) expressão(ões) aritmética(s) envolvidas no corpo do laço. Com paralelismo de granularidade fina, supomos que os processadores são menos complexos, ou seja, efetuam operações simples, tais como adição, subtração, multiplicação e divisão na(s) expressão(ões) aritmética(s) do corpo do laço.

A granularidade está intimamente relacionada com a arquitetura alvo (destino). É impossível explorarmos na sua totalidade a granularidade fina, se o número de processadores disponíveis no sistema for pequeno. Também é inviável adotarmos a granularidade grossa, se o número de processadores for grande, pois os processadores são simples, e também estaríamos subtilizando o sistema, pois teríamos processadores ociosos.

Deste modo, o método que seja generalista a ponto de explorar tanto a granularidade grossa, quanto a fina é altamente recomendável a sua aplicação em compiladores paralelizadores, pois torna o compilador flexível, podendo facilmente se adequar a arquitetura alvo (destino).

Na análise, o fator de maior influência foi que os métodos tem arquitetura-destino diferentes, e isto é em decorrência de explorarem granularidade de paralelismo diferentes e de termos aplicado sobre uma única arquitetura simulada.

Com base neste estudo podemos classificar os métodos em três categorias quanto a granularidade: métodos que exploram a granularidade fina, como é o caso do Hiperplano; métodos que exploram a granularidade grossa, como o Particionamento & Rotulação; e os que exploram ambas as granularidades grossa e fina, como é o caso do Transformações Unimodulares e o Alocação de Dados.

Em virtude dos resultados terem sido obtidos mediante a simulação, os valores apresentados são irreais e não expressam a realidade de uma execução em uma máquina paralela real, mas são utilizados como fator de comparação entre os métodos.

O número de processadores possíveis indica a quantidade de processadores necessários na arquitetura/máquina, de modo a explorarmos ao máximo o paralelismo. Desta forma, observamos que quanto menor for este número, menor será a granularidade, ou seja, temos uma granularidade grossa. Teremos uma maior granularidade, ou seja, uma granularidade fina, quando este número for grande.

É evidente que a quantidade de processadores necessários, de modo a podermos explorar ao máximo a paralelização, depende não somente do método, mas também do laço em questão, mais precisamente do número de iterações do laço e do nível de aninhamentos que este laço contenha.

Os parâmetros adotados, ou seja as métricas, foram obtidos da literatura, da análise dos métodos, e também da observação empírica da execução simultânea em ambiente de passagem de mensagens.

As métricas temporais foram obtidas pelo tempo de execução (processamento) do processador *Mestre* que é o responsável pelo controle da partição e da comunicação com os *Escravos*. Obs: o tempo de execução é o tempo de processamento realizado pelo processador *Mestre*, desconsiderando o tempo de comunicação entre os processadores *Mestre* e *Escravos* (transmissão e recepção dos dados).

Levando em consideração o tempo de execução para cada um dos métodos em função do número de processadores, verificamos que temos uma função linear (quase constante). Por outro lado, quando consideramos o tempo total (tempo de execução + tempo de comunicação), em função do número de processadores, temos uma função linear decrescente

Os valores da simulação para os métodos Hiperplano, Transformações Unimodulares e Alocação de Dados (Com Duplicação de Dados) são equivalentes e bastante grandes, significando que estes métodos são de granularidade fina, pois foram simulados em uma arquitetura com poucos processadores.

Já os métodos Particionamento & Rotulação e Alocação de Dados (Sem Dados Duplicados), apresentam valores pequenos, o que sugere que a arquitetura utilizada está em concordância com os métodos. Como temos poucos processadores na simulação, isto denota que estes métodos são úteis em granularidade grossa.

Os métodos que geram uma granularidade fina obtém um maior número de comunicação ocasionando um tempo de comunicação proporcional, o que acarreta em um tempo total maior. Por outro lado, temos os de granularidade grossa, que ocasionam o oposto, ou seja, um menor tempo de comunicação e consequentemente um menor tempo total.

Para explorarmos ao máximo o paralelismo obtido pela paralelização dos laços, é necessário que tenhamos na arquitetura o mesmo número de processadores, quanto for o valor do número de processadores possíveis.

13.2. DISCUSSÃO SOBRE O MÉTODO HIPERPLANO

O método hiperplano necessita que o laço seja composto de um índice mais externo, como índice que está ausente no corpo do laço. No caso em que temos um aninhamento no qual nenhum dos índices é ausente, simplesmente, adicionamos um índice mais externo que não tenha relação com a expressão do corpo do laço, ou seja, um que esteja ausente, tendo como limites superior e inferior o valor unitário 1.

Com esta suposição o resultado da computação, quando o laço é executado seqüencialmente não se altera, mas para a paralelização por meio do método hiperplano é imprescindível.

Após a paralelização, este índice ausente estará modificado, determinando a forma com a qual a execução das iterações será realizada.

O método utiliza as dependências, sob a forma de um sistema de equações, para a determinação dos coeficientes necessários ao mapeamento dos novos índices do laço. O sistema de equações terá tantas equações quanto forem os vetores dependência (distância), e o número de incógnitas (coeficientes) será o número de índices do laço, contando o índice que está ausente no corpo do laço. As equações são obtidas mediante os vetores distância, onde cada vetor gerará uma equação, e onde a magnitude e o sinal da dependência são obedecidos.

O hiperplano permite duas formas de paralelização: Concorrente e Paralela (Simultânea). A diferença entre o modo de obtenção das duas formas, está na suposição feita nas para as equações na obtenção dos novos índices; para a concorrente as expressões devem ter resultados positivos e diferentes de zero, ou seja, valores maiores que zero; e para a simultânea, apenas resultados maiores ou iguais a zero.

13.3. DISCUSSÃO SOBRE O MÉTODO PARTICIONAMENTO & ROTULAÇÃO

Este método utiliza como base para o particionamento, as transformações unimodulares. Ele dispõe de um algoritmo para a eliminação dos vetores linearmente dependentes, o qual reduz a matriz dependência, facilitando a aplicação do método, obtendo um melhor resultado.

Os algoritmos, que compõem o método, trabalham com as dependências sob a forma de matrizes, e que são transformadas em matrizes triangular inferior, que possibilita a obtenção das equações para os índices.

Após obtermos a matriz triangular, usamos a seleção do rótulo para proceder a decomposição dos laços aninhados em laços DO_ALL.

Outra característica é que as iterações e as direções das dependências não são alteradas após a transformação do laço.

13.4. Discussão sobre o Método Transformações Unimodulares

Este método trabalha com as dependências sob a forma de matrizes e as transformações são realizadas, sobre o conjunto de vetores dependência, por meio das três transformações elementares:

- *Permutation*
- *Reversal*
- *Skewing*

Estas transformações podem ser aplicadas em qualquer ordem, até a obtenção de uma matriz com o mínimo ou nenhum elemento negativo e/ou os elementos com valores infinito positivo e negativo nos elementos mais internos no laço. Desta forma, teremos os laços externos paralelos gerando uma granularidade grossa.

Quando não tivermos elementos que tornem os laços seqüenciais e pudermos transformar todos os elementos negativos em elementos positivos por meio das transformações unimodulares, teremos uma granularidade fina. Portanto, teremos todos os laços paralelos, com exceção do mais externo.

Somente a aplicação das transformações unimodulares não garante o paralelismo nas iterações dos laços, mas também pela aplicação da transformação *Wavefront*, a qual gera a frente de onda.

Com a matriz transformação, que será o produto de todas as transformações inclusive a *Wavefront*, podemos alterar os limites do laço.

A transformação dos limites do laço, é realizada utilizando um procedimento que é composto por quatro passos:

- Extrair as desigualdades;
- Obter os valores de máximo e mínimo para cada um dos índices do laço;
- Transformação dos índices;
- Cálculo dos novos limites do laço.

13.5. DISCUSSÃO SOBRE O MÉTODO ALOCAÇÃO DE DADOS

Este método utiliza o conceito de referências geradas uniformemente, onde a função de transformação linear de um vetor é igual para todas as referências deste vetor, e isto deve ser válido em todas as expressões que compõem o corpo do laço.

O método avalia se existem dependências de fluxo, pois são elas que determinam se um vetor será duplicável parcial ou completamente, enquanto que as demais dependências (de entrada, de saída e anti-dependência) determinam apenas a precedência da execução das iterações. Esta definição é importante, pois é através dela que analisamos a transferência de dados entre os processadores.

A questão do vetor ser duplicável parcial ou completamente, está diretamente relacionada com os dois modos de paralelização: Sem duplicação de dados e Com duplicação de dados.

Quando um vetor for duplicável completamente, poderemos ter mais de uma cópia deste vetor (ou de um elemento) na memória local dos processadores, reduzindo desta forma a comunicação entre os processadores.

Por outro lado, quando este for duplicável parcialmente, apenas uma porção deste vetor poderá ser duplicada para os processadores, ou seja, temos mais que uma cópia de um mesmo elemento sobre a memória local dos processadores, o que ocasionará em comunicações entre os processadores para a atualização desses elementos.

A transformação do programa altera os índices do laço para a nova configuração paralela; e esta modificação dos índices do(s) laço(s) se fará tanto quanto forem o número de laços paralelos.

13.6. COMPARAÇÃO ENTRE OS MÉTODOS

Os Métodos Particionamento & Rotulação e Alocação de Dados (sem dados duplicados) são úteis quando a arquitetura alvo é fracamente acoplada, ou seja, os processadores tem grande poder computacional (complexos) e o número deles é pequeno, e há pouca comunicação; por outro lado, o Hiperplano e o Alocação de Dados (com duplicação de dados) é para sistemas fortemente acoplados, onde temos uma grande quantidade de processadores envolvidos e eles tem pequeno poder computacional, e normalmente realizam apenas operações aritméticas, e há um grande número de comunicações.

O método Transformações Unimodulares permite tanto granularidade grossa quanto fina, possibilitando a utilização em sistemas fraco e fortemente acoplados, conforme as dependências contidas no corpo do laço.

As análises/comparações efetuadas sobre os métodos estão na tabela 13.1, e onde também são apresentados alguns aspectos úteis na paralelização. Isto é importante para podermos escolher dentre os métodos abordados, o que melhor se adequa às características da arquitetura-destino (alvo), com base na granularidade, no número de processadores da arquitetura e na quantidade de comunicações.

Percebemos que os métodos que exploram a granularidade grossa, apresentam uma pequena quantidade de comunicação entre os processadores, como também um pequeno número de processadores; por outro lado, os de granularidade fina requerem um grande número de comunicação entre os processadores e também um número grande de processadores.

Se a arquitetura destino tiver um número grande de processadores, invariavelmente os métodos que exploram a granularidade fina apresentam melhores resultados; caso contrário, com um pequeno número de processadores uma granularidade grossa é aconselhável

Os métodos que exploram a granularidade grossa, tornam os laços mais externos paralelos e por outro lado os de granularidade fina tornam os laços mais internos ou até mesmo, em alguns casos, todos os laços paralelos.

Características	Hiperplano	Particionamento & Rotulação	Transform. Unimodulares	Alocação de Dados(Com/Sem)
Granularidade de Paralelismo	Fina	Grossa	Grossa/Fina	Grossa/Fina
Alteração do Corpo	Sim	Não	Sim	Sim
Alteração dos Índices	Sim	Sim	Sim	Sim
Laços Paralelos	Internos	Externos	Internos	Externos
Escalonador	Não	Sim	Não	Sim
Dependências Desconsideradas	Nenhuma	(0,0,...,0)	Lexicografica-mente Positivas (0,0,...,0) (0,'±',0)	(0,0,...,0)
Aninhamento (profundidade)	Todos	Todos	Todos	Todos
Direção das Dependências	Bidirecional	Bidirecional	Bidirecional	Bidirecional
Conversão DO_ALL	Sim	Sim	Sim	Sim
No. de Processadores	Grande	Pequeno	Pequeno/Grande	Pequeno/Grande
Quant. de Comunicações	Grande	Pequena	Pequena/Grande	Pequena/Grande

Tabela 13.1. Comparação entre os métodos

Todos os métodos estudados alteram os índices do laço para gerar a frente de onda, onde várias iterações do laço possam ser executadas simultaneamente.

Os métodos trabalham sobre o conjunto de vetores dependência, onde constam as dependências e as suas direções; deste modo analisando-as, por meio de suas condições/restrições, o

método irá transformar o laço, de modo a obter, um laço reestruturado, que explore a simultaneidade do laço.

Alguns métodos se utilizam de todas as dependências, ao passo que outros, desconsideram algumas que são irrelevantes, como por exemplo a dependência (0,0,...,0), que é desconsiderada nos métodos Particionamento & Rotulação, Transformações Unimodulares e Alocação de Dados.

Todos os métodos trabalham com as dependências bidirecionais, ou seja, com direções positivas e negativas nas dependências, representando dependências com relação a elementos anteriores e posteriores.

O método hiperplano permite um único modo de paralelização, que é a de granularidade fina, que torna o laço externo seqüencial e os demais laços paralelos. O Particionamento & Rotulação explora ao máximo a granularidade grossa, tornando na medida do possível os laços externos paralelos e os internos seqüenciais. O método Transformações Unimodulares explora tanto a granularidade grossa quanto fina, o qual permite uma maior flexibilidade no modo de paralelização do laço. E, por último, o Alocação de Dados que permite os dois modos de paralelização: o Com Duplicação dos dados e o Sem Duplicação dos dados, ou seja, explorando a paralelização de granularidade fina e grossa, respectivamente.

13.7. TRABALHOS FUTUROS

- **Extração/Análise das Dependências em Tempo de Execução** é útil quando não tivermos apenas expressões no corpo do laço, mas também condições ou dependências determináveis apenas durante a execução do programa, ou seja, em tempo de execução. É útil que haja esta análise, pois permite que seja obtida uma melhor paralelização do programa gerando um melhor paralelismo entre partes do programa.

- **Aplicação Efetiva dos Métodos em um Compilador Paralelizador** para comprovar sua eficácia na paralelização e o esforço computacional necessário para se conseguir a aplicação em compiladores paralelizadores. Além disso, pode ser obtido o tempo gasto pelo método, para conseguir transformar o programa seqüencial em paralelo. Estes itens, dentre outros, irão definir a aplicabilidade dos métodos em compiladores paralelizadores de tempo real.

ANEXOS

ANEXO I - ALGORITMOS DE LAÇOS SEQUENCIAIS

- Algoritmo Laço 1

```
DO I1 = 1, 16
 DO I2 = 1, 16
  DO I3 = 1, 16

      A(I1,I2,I3) = A(I1,I2,I3)
                  + A(I1-1,I2+2,I3-4)
                  - A(I1-2,I2-4,I3+1)
                  - A(I1,I2-4,I3-2);
  END_DO
 END_DO
END_DO
```

- Algoritmo Laço 2

```
DO I1 = 1, N1
 DO I2 = 1, N2

      A(I1,I2) = A(I1-2,I2-2) * 2;

 END_DO
END_DO
```

ANEXO II - ALGORITMOS DE LAÇOS PARALELOS

- **Algoritmos Paralelos** do Laço 1
  - Método Hiperplano
  - Método Particionamento & Rotulação
  - Método Transformações Unimodulares
  - Método Alocação de Dados - Com Duplicação de Dados

- **Algoritmo Paralelos** do Laço 2
  - Método Hiperplano
  - Método Particionamento & Rotulação
  - Método Transformações Unimodulares
  - Método Alocação de Dados - Com Duplicação de Dados
  - Método Alocação de Dados - Sem Duplicação de Dados

ALGORITMOS PARALELOS DO LAÇO 1

MÉTODO HIPERPLANO

```
DO Im'= 9, 54
        DO SIM FOR ALL (I1',I2',I3') ∈ {(I1,I2,I3) :
                1<= I1 <=1,
                1<= I2 <=16,
                1<= I3 <=16,
                1<= Im'-6I1-I2-I3 <=16 }

                exp = Im'-6I1'-I2'-I3';

                A(exp,I2',I3')  = A(exp,I2',I3')
                                  + A(exp-1,I2'+2,I3'-
4)
                                  - A(exp-2,I2'-
4,I3'+1)
                                  - A(exp,I2'-4,I3'-2);

        END_DO_SIM
END_DO
```

MÉTODO PARTICIONAMENTO & ROTULAÇÃO

```
DO_ALL I01 = 1, 1
DO_ALL I02 = 1, 4
DO_ALL I03 = 1, 13
        DO I1 = I01, 16, 1
                Y1 = (I1 - I01)/1
                I2MIN = 1 + (I02 - 1 +2*Y1) MOD 4
                DO I2 = I2MIN, 16, 4
                     Y2 = (I2 - I02 + 2*Y1)/4
                     I3MIN = 1 + (I03 - 1 + 4*Y1 +
2*Y2) MOD 13
                     DO I3 = I3MIN, 16, 13

                         A(I1,I2,I3) = A(I1,I2,I3)
                                  + A(I1-1,I2+2,I3-4)
                                  - A(I1-2,I2-4,I3+1)
                                  - A(I1,I2-4,I3-2);
                    END_DO
                END_DO
        END_DO
END_DO_ALL
END_DO_ALL
END_DO_ALL
```

MÉTODO TRANSFORMAÇÕES UNIMODULARES

```
DO I1` =  -5, -80
  DO_ALL I2' = MAX(-16,(I1`+3)/2)), MIN(-
1,(I1`+48)/2))
  DO_ALL I3` = MAX(-32,(-2*I2'+I1`+1)), MIN(-2,(-
2*I2`+I1`+16))

        I1 = -I2`;
        I2 = -2*I3`-2*I2`+I1`;
        I3 = I3`+ 2*I2`-I1`;

        A(I1,I2,I3) = A(I1,I2,I3)
                    + A(I1-1,I2+2,I3-4)
                    - A(I1-2,I2-4,I3+1)
                    - A(I1,I2-4,I3-2);
  END_DO_ALL
  END_DO_ALL
END_DO
```

MÉTODO ALOCAÇÃO DE DADOS - COM DUPLICAÇÃO DE DADOS

```
DO_ALL I' = (1+(A1 - (1 MOD P1)) MOD P1), 16 STEP P1
   DO_ALL J' = (1+(A2 - (1 MOD P2)) MOD P2), 16 STEP P2
      DO_ALL K' = (1+(A3 - (1 MOD P3)) MOD P3), 16 STEP
P3

                I1 = I';
                I2 = J';
                I3 = K';

                A(I1,I2,I3) = A(I1,I2,I3)
                        + A(I1-1,I2+2,I3-4)
                        - A(I1-2,I2-4,I3+1)
                        - A(I1,I2-4,I3-2);

      END_DO_ALL
   END_DO_ALL
END_DO_ALL
```

ALGORITMOS PARALELOS DO LAÇO 2

MÉTODO HIPERPLANO

```
DO Im'= 6, (4+N1+N2)
        DO SIM FOR ALL (I1',I2') E {(I1,I2) :
            1<= I1 <=1,
            1<= I2 <=N1,
            1<= Im'-4I1-I2 <=N2 }

            exp = Im'-4I1'-I2';

           A(I2',exp) = A(I2'- 2,exp -2) * 2;

        END_DO_SIM
END_DO
```

MÉTODO PARTICIONAMENTO & ROTULAÇÃO

```
DO_ALL I01 = 1, 2
DO_ALL I02 = 1, N2
        DO I1 = I01, N1, 2
                Y1 = (I1 - I01)/2
                I2MIN = 1 + (I02 - 1 + 2*Y1) MOD N2
                DO I2 = I2MIN, N2, N2

                        A(I1,I2) = A(I1-2,I2-2) * 2;

                END_DO
        END_DO
END_DO_ALL
END_DO_ALL
```

MÉTODO TRANSFORMAÇÕES UNIMODULARES

```
DO I1` =  1, 16
   DO_ALL I2' = MAX(-32,(-I1`-1)), MIN(-2,(-I1`-16))

                I1 = I1`;
                I2 = -I1` - I2`;

                A(I1,I2) = A(I1-2,I2-2) * 2;

   END_DO_ALL
END_DO
```

MÉTODO ALOCAÇÃO DE DADOS - COM DUPLICAÇÃO DE DADOS

```
DO_ALL I'= (1+(A1-(1 MOD P1)) MOD P1), 16 STEP P1
   DO_ALL J' = (1+(A2-(1 MOD P2)) MOD P2), 16 STEP P2

                I1 = I';
                I2 = J';
                A(I1,I2) = A(I1-2,I2-2) * 2;

   END_DO_ALL
END_DO_ALL
```

MÉTODO ALOCAÇÃO DE DADOS - SEM DUPLICAÇÃO DE DADOS

```
DO_ALL I'= 15, -15 STEP -1
        DO I1 = MAX(1,I'+1), MIN(16,I'+16)

                I2 = I1-I';
                A(I1,I2) = A(I1-2,I2-2) * 2;

        END_DO
END_DO_ALL
```

OU

```
DO_ALL I'= -15, 15 STEP 1
        DO I1 = MAX(1,-I'+1), MIN(16,-I'+16)

                I2 = I'+I1;
                A(I1,I2) = A(I1-2,I2-2) * 2;

        END_DO
END_DO_ALL
```

ANEXO III - LISTAGEM DOS PROGRAMAS PARALELOS

- **Programas Paralelos para o Laço 1**
  - Método Hiperplano
  - Método Particionamento & Rotulação
  - Método Transformações Unimodulares
  - Método Alocação de Dados
    - Com Duplicação dos Dados

- **Simulação para o Laço 2**
  - Método Hiperplano
  - Método Particionamento & Rotulação
  - Método Transformações Unimodulares
  - Método Alocação de Dados
    - Com Duplicação dos Dados
    - Sem Duplicação dos Dados

LAÇO 1

MÉTODO HIPERPLANO

```
/* LACO 1 - Metodo Hiperplano */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int p = 10;                          /* Numero de processos */
struct {                     /* Estrutura para passar armazenar os indices */
            int I1;
            int I2;
            int I3;
            int I4;
  } indice[10];

int expressao[4];
MPI_Status status;
int A[23][23][23];                   /* Matriz de calculo */
int retorno;                         /* Variavel para retorno da expressao
*/
main(argc, argv)
int argc;
char *argv[];
{
  int myrank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  if (myrank == 0)
     master();
  else
     slave();
  MPI_Finalize();
}
/* ---------------- */
/* Processo MASTER  */
/* ---------------- */
master()
{
  int Im,I1,I2,I3, cont,exp;
  int j,k;                           /* Indice para os processos */
  double start, finish;                          /* Variaveis de tempo */
  int qcom, qdados, qtrans, qrec;
  double scom, tcom;

 for (I1=0; I1 <= 22; I1++)                      /* Inicializa matriz com 1's
*/
            for (I2=0; I2 <= 22; I2++)
                        for (I3=0; I3 <= 22; I3++)
                                    A[I1][I2][I3] = 1;

   cont = 0;   j = 1;   tcom = 0.0;   qcom = 0;
   qdados = 0;   qtrans = 0;   qrec = 0;
   start = MPI_Wtime();
   for (Im=9; Im<= 54; Im++){
    for (I1 = 1; I1 <= 1; I1++)
    for (I2 = 1; I2 <= 16; I2++)
    for (I3 = 1; I3 <= 16; I3++){
            if (((Im-6*I1-I2-I3) >= 1) & ((Im-6*I1-I2-I3) <= 16))
            {
               exp = (Im - (6*I1) - I2 - I3);
              indice[j-1].I1 = Im;
              indice[j-1].I2 = I1;
              indice[j-1].I3 = I2;
              indice[j-1].I4 = I3;
              expressao[0] = A[exp+4][I2+4][I3+4];
              expressao[1] = A[exp-1+4][I2+2+4][I3-4+4];
              expressao[2] = A[exp-2+4][I2-4+4][I3+1+4];
              expressao[3] = A[exp+4][I2-4+4][I3-2+4];
              qcom++;
              qdados += 4;
              qtrans += 4;
              scom = MPI_Wtime();
               MPI_Send(expressao, 4, MPI_INT, j, 99,
MPI_COMM_WORLD);
              tcom +=MPI_Wtime() - scom;
              j++;
              cont ++;

              if (j>p){
                        for(j=1; j<=p; j++){
                          qcom++;
                          scom = MPI_Wtime();
                          MPI_Recv(&retorno, 1, MPI_INT, j, 99,
MPI_COMM_WORLD, &status);
                          tcom +=MPI_Wtime() - scom;
                          exp =  (indice[j-1].I1 - (6*indice[j-1].I2) -
indice[j-1].I3 - indice[j-1].I4);
                          A[exp+4][indice[j-1].I3+4][indice[j-1].I4+4] =
retorno;
                          qdados ++;
                          qrec ++;
                        }
                        j=1;
              }
            }
    }
   }
   k = j;
   for(j=1; j < k; j++) {
     qcom++;
     scom = MPI_Wtime();
     MPI_Recv(&retorno, 1, MPI_INT, j, 99, MPI_COMM_WORLD,
&status);
     tcom += MPI_Wtime() - scom;

     exp = (indice[j-1].I1 - 6*indice[j-1].I2)-indice[j-1].I3-indice[j-1].I4);
     A[exp+4][indice[j-1].I3+4][indice[j-1].I4+4] = retorno;
     qdados ++;
     qrec  ++;
   }
   finish = MPI_Wtime();
   expressao[0] = 1000;
   expressao[1] = 1000;
   expressao[2] = 1000;
   expressao[3] = 1000;
   for(j=1; j<=p ; j++) {
     MPI_Send(expressao, 4, MPI_INT, j, 99, MPI_COMM_WORLD);
   }
   printf("\n\nLACO 1 - Metodo Hiperplano \n\n");
   printf("\n\nFinalizando a execucao\n\n");
   printf("    Tempo total gasto        : %.7f\n",finish-start);
   printf("    Tempo de comunicacao     : %.7f\n",tcom);
   printf("    Tempo de execucao        : %.7f\n\n",(finish-start)-tcom);
   printf("    Quant. de Comunicacoes   : %d\",qcom);
   printf("    Quant. Dados Transmitidos : %d\n",qtrans);
   printf("    Quant. Dados Recebidos   : %d\n",qrec);
   printf("    Quant. Dados Comunicados  :
%d\n\n",qdados,qtrans+qrec);
   printf("    Tempo medio de Comunicacao: %.7f\n",tcom/qcom);
   printf("    No.de lacos (iteracoes)  : %d\n\n",cont);
   printf("\n\n\n");
   for(I1=9;I1 <= 9; I1++)
     for(I2=7; I2 <= 10; I2++)
            for(I3=1; I3 <= 16; I3++)
              printf("%d ",A[I1][I2][I3]);
   printf("\n\n\n");
 }
 /* ---------------- */
 /* Processo SLAVE    */
 /* ---------------- */
 slave()
 {
   do {
    MPI_Recv(expressao, 4, MPI_INT, 0, 99, MPI_COMM_WORLD,
&status);
   if ((expressao[0] != 1000) && (expressao[1] != 1000)) {
            retorno = expressao[0] + expressao[1] - expressao[2] -
expressao[3];
            MPI_Send(&retorno, 1, MPI_INT, 0, 99,
MPI_COMM_WORLD);
    }
   }
   while ((expressao[0] != 1000) && (expressao[1] != 1000));
 }
```

MÉTODO PARTICIONAMENTO & ROTULAÇÃO

```
/* LACO 1 - Metodo Particionamento e Rotulação */

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int p = 10;                         /* Numero de processos */
int indice[3];
struct {                    /* Estrutura para receber os valores alterados */
            int I1;
            int I2;
            int I3;
            int Val;
  } retorno[80];

int Qr;                     /* Quant. de dados retornados por iteracao     */
MPI_Status status;
int A[23][23][23];                  /* Matriz de calculo */

main(argc, argv)
int argc;
char *argv[];
{
  int myrank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  if (myrank == 0)
            master();
  else
            slave();
  MPI_Finalize();
}

/* ---------------- */
/* Processo MASTER  */
/* ---------------- */
master()
{
  int I01,I02,I03,I1,I2,I3, cont;
  int j,i,k;                                /* Indice para os
processos */
  int cont2;
  double start, finish;                     /* Variaveis de tempo */
  int qcom,qbcast,qdados,qtrans,qrec;
  double scom,tcom;

  for (I1=0; I1 <= 22; I1++)                /* Inicializa matriz com 1's
*/
            for (I2=0; I2 <= 22; I2++)
                        for (I3=0; I3 <= 22; I3++)
                                    A[I1][I2][I3] = 1;

  cont  = 0;
  j     = 1;
  qcom  = 0;
  qbcast = 0;
  qdados = 0;
  qtrans = 0;
  qrec  = 0;

  start = MPI_Wtime();
  MPI_Bcast(A, 12167, MPI_INT, 0, MPI_COMM_WORLD);

  tcom = MPI_Wtime()-start;
  qbcast ++;
  qdados += 12127;

  for (I01 = 1; I01 <= 1; I01++)
  for (I02 = 1; I02 <= 4; I02++)
  for (I03 = 1; I03 <= 13; I03++){
              indice[0] = I01;
              indice[1] = I02;
              indice[2] = I03;
        qcom ++;
        scom = MPI_Wtime();
        MPI_Send(indice, 3, MPI_INT, j, 99, MPI_COMM_WORLD);
        tcom +=MPI_Wtime()-scom;
        qdados += 3;
        qtrans += 3;
              cont++;
              j++;
              if (j>p){
                        for(j=1; j<=p; j++){
                          qcom += 2;
                          scom = MPI_Wtime();
                          MPI_Recv(&Qr, 1, MPI_INT, j, 99,
MPI_COMM_WORLD, &status);
                          MPI_Recv(retorno, (Qr*4), MPI_INT, j, 99,
MPI_COMM_WORLD, &status);
                          tcom += MPI_Wtime() - scom;
                          qdados += Qr*4;
                          qrec += Qr*4;

                          for (i=0; i<Qr; i++){
                            A[retorno[i].I1][retorno[i].I2][retorno[i].I3] =
retorno[i].Val;
                          }
                        }
                        j=1;
              }
  }

  for(k=1; k<j; k++) {
    qcom +=2;
    scom = MPI_Wtime();
    MPI_Recv(&Qr, 1, MPI_INT, k, 99, MPI_COMM_WORLD, &status);
    MPI_Recv(retorno, (Qr*4), MPI_INT, k, 99, MPI_COMM_WORLD,
&status);
    tcom +=MPI_Wtime()-scom;
    qdados += Qr*4;
    qrec += Qr*4;
    for (i=0; i<Qr; i++) {
            A[retorno[i].I1][retorno[i].I2][retorno[i].I3] = retorno[i].Val;
    }
  }

  finish = MPI_Wtime();
  indice[0] = 1000;
  indice[1] = 1000;
  indice[2] = 1000;

  for(k=1; k<=p; k++) {
    MPI_Send(indice, 3, MPI_INT, k, 99, MPI_COMM_WORLD);
  }

  printf("\n\nLACO 1 - Metodo Particao e rotulação \n\n");
  printf("\n\nFinalizando a execucao\n\n");
  printf("   Tempo Total gasto          : %.7f\n",finish-start);
  printf("   Tempo de Comunicacao       : %.7f\n",tcom);
  printf("   Tempo de Execucao          : %.7f\n\n",(finish-start)-tcom);
  printf("   Quant. de Comunicacoes     : %d\n",qcom);
  printf("   Quant. de Bcast            : %d\n",qbcast);
  printf("   Quant. Dados Transmitidos  : %d\n",qtrans);
  printf("   Quant. Dados Recebidos     : %d\n",qrec);
  printf("   Quant. Dados Comunicados   : %d
%d\n\n",qdados,qtrans+qrec);
  printf("   Tempo Medio de Comunicacao :
%.7f\n",tcom/(qcom+qbcast));
  printf("   No. de lacos (iteracoes)   : %d\n\n",cont);
  printf("   No. de Processadores       : %d\n\n",p+1);

  for(i=9;i<= 9;i++)
    for(j=7; j<=10; j++)
      for(k=1; k<=16; k++)
        printf("%d ",A[i][j][k]);

  printf("\n\n\n");
}
/* ---------------- */
/* Processo SLAVE   */
/* ---------------- */
slave()
{
 int j,I2MIN,I3MIN;
 int I01,I02,I03,I1,I2,I3,I1,I2,I3;
 float Y1,Y2;
 MPI_Bcast(A, 12167, MPI_INT, 0, MPI_COMM_WORLD);

 do {
```

```
  MPI_Recv(indice, 3, MPI_INT, 0, 99, MPI_COMM_WORLD, &status);

  if ((indice[0] != 1000) && (indice[1] != 1000)) {

    I01 = indice[0];
    I02 = indice[1];
    I03 = indice[2];
    j=0;
    for (I1=I01; I1 <= 16;I1++){
            Y1 = (float)((I1 - I01)/1);
            I2MIN = 1 + (I02 - 1 +(int)(2*Y1)) % 4;
            for (I2 = I2MIN; I2 <= 16; I2 += 4){
                    Y2 = (float)((I2 - I02 + 2*Y1)/4);
                    I3MIN = 1 + (I03 - 1 + (int)(4*Y1) +(int)(2*Y2)) %
13;
                    for (I3 = I3MIN; I3<=16; I3 += 13){

                      I1 = I1 + 4;
                      I2 = I2 + 4;
                      I3 = I3 + 4;
                              A[I1][I2][I3] = A[I1][I2][I3] + A[I1-1][I2+2][I3-4] -
A[I1-2][I2-4][I3+1] - A[I1][I2-4][I3-2];
                              retorno[j].I1 = I1;
                              retorno[j].I2 = I2;
                              retorno[j].I3 = I3;
                              retorno[j].Val = A[I1][I2][I3];
                              j++;
                            }
                  }
    }
  }

  Qr = j;

  MPI_Send(&Qr, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
  MPI_Send(retorno, (Qr*4), MPI_INT, 0, 99, MPI_COMM_WORLD);

 }
 while ((indice[0] != 1000) && (indice[1] != 1000));
}
```

MÉTODO TRANSFORMAÇÕES UNIMODULARES

```
/* LACO 1 - Metodo Transformacoes Unimodulares */

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int p = 10;                          /* Numero de processos */

struct {                     /* Estrutura para passar armazenar os indices */
          int I1;
          int I2;
          int I3;
  } indice[10];

int expressao[4];

MPI_Status status;

int A[25][25][25];                   /* Matriz de calculo */

int retorno;                         /* Variavel para retorno da expressao
*/

main(argc, argv)
int argc;
char *argv[];
{
  int myrank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

  if (myrank == 0)
     master();
  else
     slave();

  MPI_Finalize();
}

int MAX(a,b)
int a,b;
   {
          if (a > b) return a;
          else return b;
   }

int MIN(a,b)
int a,b;
   {
          if (a < b) return a;
          else return b;
   }

/* ---------------- */
/* Processo MASTER  */
/* ---------------- */
master()
{
  int I1,I2,I3, cont;
  int k,j,x,y;                       /* Indice para os processos */
  double start, finish;                        /* Variaveis de tempo */
  double tcom, scom;
  int qcom, qdados, qtrans, qrec;

  for (I1=0; I1 <= 24; I1++)                   /* Inicializa matriz com 1's
*/
          for (I2=0; I2 <= 24; I2++)
        for(I3=0; I3<= 24;I3++)
                    A[I1][I2][I3] = 1;

  cont = 0;  scom = 0;  tcom = 0;  qcom = 0;  qdados = 0;  qtrans = 0;
qrec = 0;  j = 1;

  start = MPI_Wtime();
  for (I1 = -5; I1 >= -80; I1--){
    if(((I1+3)%2==0.0) || ((I1+48)%2==0.0)) {

    if(MAX(-16,(I1+3)/2) > MIN(-1,(I1+48)/2)) x = -1; else x=1;
    for (I2 = MAX(-16,(I1+3)/2); I2 <=  MIN(-1,(I1+48)/2); I2 =I2 + x) {
            if (MAX(-32,(I1-(2*I2)+1)) > MIN(-2,(I1-(2*I2)+16))) y= -1;
else y=1;
            for (I3 = MAX(-32,(I1-(2*I2)+1)); I3<=MIN(-2,(I1-(2*I2)+16));
I3=I3+y)
            {
              indice[j-1].I3 = -I1+(2*I2)+I3;
              indice[j-1].I2 = I1-(2*I2) - (2*I3);
              indice[j-1].I1 = -I2;

              if(indice[j-1].I1>=1 & indice[j-1].I2>=1 & indice[j-1].I3>=1 &
indice[j-1].I1<=16 & indice[j-1].I2<=16                                    &
indice[j-1].I3<=16) {

              expressao[0] = A[indice[j-1].I1+4][indice[j-1].I2+4][indice[j-
1].I3+4];
              expressao[1] = A[indice[j-1].I1-1+4][indice[j-
1].I2+2+4][indice[j-1].I3-4+4];
              expressao[2] = A[indice[j-1].I1-2+4][indice[j-1].I2-
4+4][indice[j-1].I3+1+4];
              expressao[3] = A[indice[j-1].I1+4][indice[j-1].I2-
4+4][indice[j-1].I3-2+4];

              scom = MPI_Wtime();
              MPI_Send(expressao, 4, MPI_INT, j, 99,
MPI_COMM_WORLD);
              tcom += MPI_Wtime() - scom;
              qcom ++;
              qdados ++;
              qtrans ++;
               j++;
              cont ++;

              if (j>p){
                      for(j=1; j<=p; j++){
                        scom = MPI_Wtime();
```

```
                    MPI_Recv(&retorno, 1, MPI_INT, j, 99,
MPI_COMM_WORLD, &status);
                    tcom += MPI_Wtime() - scom;
                    qcom ++;
                    A[indice[j-1].I1+4][indice[j-1].I2+4][indice[j-
1].I3+4] = retorno;

                    qdados ++;
                    qrec ++;
                  }
                  j=1;
            }
            }
          }
    }
   }
  }

  k = j;
  for(j=1; j<k; j++) {
    scom = MPI_Wtime();
    MPI_Recv(&retorno, 1, MPI_INT, j, 99, MPI_COMM_WORLD,
&status);
    tcom += MPI_Wtime() - scom;
    qcom ++;
    A[indice[j-1].I1+4][indice[j-1].I2+4][indice[j-1].I3+4] = retorno;
    qdados ++;
    qrec ++;
  }

  finish = MPI_Wtime();

  expressao[0] = 1000;
  expressao[1] = 1000;
  expressao[2] = 1000;
  expressao[3] = 1000;

  for(j=1 ; j<=p; j++) {
    MPI_Send(expressao, 4, MPI_INT, j, 99, MPI_COMM_WORLD);
  }

  printf("\n\nLACO 1 - Metodo Transformacoes Unimodulares\n\n");
  printf(" \n Granularidade Fina \n\n");
  printf("\n\nFinalizando a execucao\n\n");
  printf("   Tempo Total gasto          : %.7f\n",finish-start);
  printf("   Tempo de Comunicacao       : %.7f\n",tcom);
  printf("   Tempo de Execucao          : %.7f\n\n",finish - start - tcom);
  printf("   Quant. de Comunicacoes     : %d\n",qcom);
  printf("   Quant. Dados Transmitidos  : %d\n",qtrans);
  printf("   Quant. Dados Recebidos     : %d\n",qrec);
  printf("   Quant. Dados Comunicados   : %d
%d\n\n",qdados,qtrans+qrec);
  printf("   Tempo Medio de Comunicacao : %.7f\n",tcom/qcom);
  printf("   No. de lacos (iteracao)    : %d\n",cont);
  printf("   No. de Processadores       : %d\n",p+1);
  printf("\n\n\n");

  for(I1=9; I1 <=9; I1++)
    for(I2=7; I2 <=10; I2++)
      for(I3=1; I3<=16;I3++)
      printf("%d ",A[I1][I2][I3]);
  printf("\n\n\n");

}
/* ---------------- */
/* Processo SLAVE     */
/* ---------------- */
slave()
{
  do {
    MPI_Recv(expressao, 4, MPI_INT, 0, 99, MPI_COMM_WORLD,
&status);
    if ((expressao[0] != 1000) & (expressao[2] != 1000)) {
            retorno = expressao[0] + expressao[1] - expressao[2] -
expressao[3];
            MPI_Send(&retorno, 1, MPI_INT, 0, 99,
MPI_COMM_WORLD);
    }
  } while ((expressao[0] != 1000) & (expressao[2] != 1000));
}
```

Método Alocação de Dados - Com Duplicação de Dados

```
/* LACO 1 - Metodo Communication - Free ( Com dados duplicados ) */

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int p = 10;                    /* Numero de processos */

struct {                /* Estrutura para passar armazenar os indices */
        int I1;
        int I2;
        int I3;
  } indice[10];

int expressao[4];

MPI_Status status;

int A[23][23][23];             /* Matriz de calculo */

int retorno;                   /* Variavel para retorno da expressao
*/

main(argc, argv)
int argc;
char **argv;
{
  int myrank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

  if (myrank == 0)
     master();
  else
     slave();

  MPI_Finalize();
}

/* ---------------- */
/* Processo MASTER  */
/* ---------------- */

master()
{
  int I1,I2,I3, cont;
  int k,j;                     /* Indice para os processos */

  int p1,p2,p3;
  int b1,b2,b3;

  double start, finish;                /* Variaveis de tempo */
  int qcom, qdados,qtrans,qrec;
  double scom,tcom;

  for (I1=0; I1 <= 22; I1++)           /* Inicializa matriz com 1's
*/
          for (I2=0; I2 <= 22; I2++)
                  for (I3=0; I3 <= 22; I3++)
                          A[I1][I2][I3] = 1;

  cont = 0;
  j = 1;
  qcom = 0;
  qdados = 0;
  qtrans = 0;
  qrec = 0;

  p1 = 1;
  p2 = 1;
  p3 = 1;

  b1 = p1-1;
  b2 = p2-1;
  b3 = p3-1;
```

```
  start = MPI_Wtime();

  for (I1 = (1+(b1-(1 % p1)) % p1); I1 <= 16; I1+=p1)
  for (I2 = (1+(b2-(1 % p2)) % p2); I2 <= 16; I2+=p2)
  for (I3 = (1+(b3-(1 % p3)) % p3); I3 <= 16; I3+=p3){

            indice[j-1].I1 = I1;
            indice[j-1].I2 = I2;
            indice[j-1].I3 = I3;

            expressao[0] = A[I1+4][I2+4][I3+4];
            expressao[1] = A[I1-1+4][I2+2+4][I3-4+4];
            expressao[2] = A[I1-2+4][I2-4+4][I3+1+4];
            expressao[3] = A[I1+4][I2-4+4][I3-2+4];

      qcom ++;
      scom = MPI_Wtime();
      MPI_Send(expressao, 4, MPI_INT, j, 99, MPI_COMM_WORLD);
      tcom += MPI_Wtime() - scom;
      qdados += 4;
      qtrans += 4;

      j++;
      cont ++;

            if (j>p){
                        for(j=1; j<=p; j++){
                          qcom ++;
                          scom = MPI_Wtime();
                          MPI_Recv(&retorno, 1, MPI_INT, j, 99,
MPI_COMM_WORLD, &status);
                          tcom += MPI_Wtime() - scom;
                          A[indice[j-1].I1+4][indice[j-1].I2+4][indice[j-
1].I3+4] = retorno;
                          qdados ++;
                          qrec ++;
                        }
                        j=1;
            }
  }
  k = j;
  for(j=1; j<k; j++) {
    qcom++;
    scom = MPI_Wtime();
    MPI_Recv(&retorno, 1, MPI_INT, j, 99, MPI_COMM_WORLD,
&status);
    tcom += MPI_Wtime() - scom;
    A[indice[j-1].I1+4][indice[j-1].I2+4][indice[j-1].I3+4] = retorno;
    qdados ++;
    qrec ++;
  }
  finish = MPI_Wtime();

  expressao[0] = 1000;
  expressao[1] = 1000;
  expressao[2] = 1000;
  expressao[3] = 1000;

  for (j=1; j<=p; j++) {
    MPI_Send(expressao, 3, MPI_INT, j, 99, MPI_COMM_WORLD);
  }

  printf("\n\nLACO 1 - Metodo Communication - Free\n\n");

  printf("\n\nFinalizando a execucao\n\n");
  printf("   Tempo Total gasto          : %.7f\n",finish-start);
  printf("   Tempo de comunicacao       : %.7f\n",tcom);
  printf("   Tempo de Execucao          : %.7f\n\n",finish-start-tcom);
  printf("   Quant. de Comunicacoes     : %d\n",qcom);
  printf("   Quant. Dados Transmitidos  : %d\n",qtrans);
  printf("   Quant. Dados Recebidos     : %d\n",qrec);
  printf("   Quant. Dados Comunicados   :
%d\n\n",qdados,qtrans+qrec);
  printf("   Tempo  Medio de Comunicacao : %.7f\n",tcom/qcom);
  printf("   No. de lacos (iteracoes)    : %d\n\n",cont);
  printf("   No. de Processadores        : %d\n\n",p+1);
  printf("\n\n\n");

  for(I1=9; I1<=9; I1++)
    for(I2=7; I2<=10; I2++)
      for(I3=1; I3<=16; I3++)
        printf("%d ",A[I1][I2][I3]);

  printf("\n\n\n");
}

/* ------------------- */
/* Processo SLAVE      */
/* ------------------- */

slave()
{
  do {

    MPI_Recv(expressao, 4, MPI_INT, 0, 99, MPI_COMM_WORLD,
&status);

    if ((expressao[0] != 1000) && (expressao[1] != 1000)){

            retorno = expressao[0] + expressao[1] - expressao[2] -
expressao[3];

      MPI_Send(&retorno, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
    }
  }
  while ((expressao[0] != 1000) && (expressao[1] != 1000));
}
```

LAÇO 2

MÉTODO HIPERPLANO

```
/* LACO 2 - Metodo Hiperplano */

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int p = 11;                          /* Numero de processos */

struct {                  /* Estrutura para passar armazenar os indices */
          int I1;
          int I2;
          int I3;
  } indice[15];

int expressao;

MPI_Status status;

int A[19][19];                                /* Matriz de calculo */

int retorno;                         /* Variavel para retorno da expressao
*/

main(argc, argv)
int argc;
char *argv[];
{

  int myrank;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

  if (myrank == 0)
     master();
  else
     slave();

  MPI_Finalize();
}

/* ---------------- */
/* Processo MASTER  */
/* ---------------- */
master()
{
  int Im,I1,I2, cont,exp;
  int k,j;                           /* Indice para os processos */
  double start, finish;                         /* Variaveis de tempo */
  double tcom, scom;
  int qcom, qdados, qtrans, qrec;

  for (I1=0; I1 <= 18; I1++)                    /* Inicializa matriz com 1's
*/
          for (I2=0; I2 <= 18; I2++)
                    A[I1][I2] = 1;

  cont = 0;
  scom = 0;
  tcom = 0;
  qcom = 0;
  qdados = 0;
  qtrans = 0;
  qrec = 0;

  j = 1;

  start = MPI_Wtime();

  for (Im=4; Im<= 34; Im++){
   for (I1 = 1; I1 <= 1; I1++)
   for (I2 = 1; I2 <= 16; I2++){
          if (((Im-2*I1-I2) >= 1) & ((Im-2*I1-I2) <= 16))
          {
                    exp = (Im - (2*I1) - I2);

                    indice[j-1].I1 = Im;
                    indice[j-1].I2 = I1;
                    indice[j-1].I3 = I2;

                    expressao = A[exp-2+2][I2-2+2];

                    scom = MPI_Wtime();
                    MPI_Send(&expressao, 1, MPI_INT, j, 99,
MPI_COMM_WORLD);
                    tcom += MPI_Wtime() - scom;
                    qcom ++;
                    qdados ++;
                    qtrans ++;

                    j++;
                    cont ++;

                    if (j>p){
                              for(j=1; j<=p; j++){
                                scom = MPI_Wtime();
                                MPI_Recv(&retorno, 1, MPI_INT, j, 99,
MPI_COMM_WORLD, &status);
                                tcom += MPI_Wtime() - scom;
                                qcom ++;
                                exp = (indice[j-1].I1 - (2*indice[j-1].I2) - indice[j-
1].I3);
                                A[exp+2][indice[j-1].I3+2] = retorno;
                                qdados ++;
                                qrec ++;
                              }
                              j=1;
                    }
          }
   }
  }
  k = j;
  for(j=1; j<k; j++) {
    scom = MPI_Wtime();
    MPI_Recv(&retorno, 1, MPI_INT, j, 99, MPI_COMM_WORLD,
&status);
    tcom += MPI_Wtime() - scom;
    qcom ++;
    exp = (indice[j-1].I1 - (2*indice[j-1].I2) - indice[j-1].I3);
    A[exp+2][indice[j-1].I3+2] = retorno;
    qdados ++;
    qrec ++;
  }

  finish = MPI_Wtime();

  expressao = 1000;

  for(j=1 ; j<=p; j++) {
    MPI_Send(&expressao, 1, MPI_INT, j, 99, MPI_COMM_WORLD);
  }

  printf("\n\nLACO 2 - Metodo Hiperplano\n\n");

  printf("\n\nFinalizando a execucao\n\n");

  printf("   Tempo Total gasto          : %.7f\n",finish-start);
  printf("   Tempo de Comunicacao       : %.7f\n",tcom);
  printf("   Tempo de Execucao          : %.7f\n\n",finish - start - tcom);
  printf("   Quant. de Comunicacoes     : %d\n",qcom);
  printf("   Quant. Dados Transmitidos  : %d\n",qtrans);
  printf("   Quant. Dados Recebidos     : %d\n",qrec);
  printf("   Quant. Dados Comunicados   : %d
%d\n\n",qdados,qtrans+qrec);
  printf("   Tempo Medio de Comunicacao : %.7f\n",tcom/qcom);
  printf("   No. de lacos (iteracao)    : %d\n",cont);
  printf("   No. de Processadores       : %d\n",p+1);
  printf("\n\n\n");

  for(I1=5; I1 <=10; I1++)
```

```
    for(l2=5; l2 <=10; l2++)
      printf("%d ",A[l1][l2]);
  printf("\n\n\n");
}
/* ---------------- */
/* Processo SLAVE     */
/* ---------------- */
slave()
{
  do {
    MPI_Recv(&expressao, 1, MPI_INT, 0, 99, MPI_COMM_WORLD,
&status);
    if (expressao != 1000) {
            retorno = expressao * 2;

            MPI_Send(&retorno, 1, MPI_INT, 0, 99,
MPI_COMM_WORLD);
    }
  } while (expressao != 1000);
}
```

Método Particionamento & Rotulação

```
/* LACO 2 - Metodo Particao e Rotulação */

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int p = 7;                          /* Numero de processos */

int indice[2];

struct {            /* Estrutura para receber os valores alterados */
            int l1;
            int l2;
            int Val;
  } retorno[8];


MPI_Status status;

int A[19][19];                              /* Matriz de calculo */

main(argc, argv)
int argc;
char *argv[];
{

  int myrank;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

  if (myrank == 0)
            master();
  else
            slave();

  MPI_Finalize();
}

/* ---------------- */
/* Processo MASTER  */
/* ---------------- */
master()
{
  int l01,l02,l1,l2, cont;
  int k,j,i;                        /* Indice para os processos */
  double start, finish;                       /* Variaveis de tempo */
  double tcom, scom;
  int qcom,qbcast, qdados,qtrans,qrec;

  for (l1=0; l1 <= 18; l1++)                  /* Inicializa matriz com 1's
*/
            for (l2=0; l2 <= 18; l2++)
                        A[l1][l2] = 1;

  cont=0;
  scom=0;
  tcom=0;
  qcom=0;
  qbcast=0;
  j = 1;
  qdados = 0;
  qtrans = 0;
  qrec = 0;

  start = MPI_Wtime();
  MPI_Bcast(A, 361, MPI_INT, 0, MPI_COMM_WORLD);
  tcom += MPI_Wtime() - start;
  qbcast ++;
  qdados += 361;

  for (l01 = 1; l01 <= 2; l01++)
  for (l02 = 1; l02 <= 16; l02++)
            {
              indice[0] = l01;
              indice[1] = l02;

              scom = MPI_Wtime();
              MPI_Send(indice, 2, MPI_INT, j, 99,
MPI_COMM_WORLD);
              tcom += MPI_Wtime() - scom;
              qcom ++;
              qdados += 2;
              qtrans += 2;

              j++;
              cont ++;

              if (j>p){
                        for(j=1; j<=p; j++){
                          scom = MPI_Wtime();
                          MPI_Recv(retorno, 24, MPI_INT, j, 99,
MPI_COMM_WORLD, &status);
                          tcom += MPI_Wtime() - scom;
                          qcom ++;
                          qdados += 24;
                          qrec += 24;
                          for (i=0; i<=7; i++){
                            A[retorno[i].l1][retorno[i].l2] = retorno[i].Val;
                          }
                        }
                        j=1;
              }
  }

  k = j;
  for(j=1; j< k ;j++) {
    scom = MPI_Wtime();
    MPI_Recv(retorno,24,MPI_INT,j,99, MPI_COMM_WORLD, &status);
    tcom += MPI_Wtime() - scom;
    qcom ++;
    qdados += 24;
    qrec += 24;
    for(i=0; i<=7; i++) {
            A[retorno[i].l1][retorno[i].l2] = retorno[i].Val;
    }
  }

  finish = MPI_Wtime();

  indice[0] = 1000;
  indice[1] = 1000;

  for(j=1; j<=p; j++) {
    MPI_Send(indice, 2, MPI_INT, j, 99, MPI_COMM_WORLD);
  }

  printf("\n\nLACO 2 - Metodo Particao e Rotulação\n\n");
  printf("\n\nFinalizando a execucao\n\n");
  printf("   Tempo Total gasto        : %.7f\n",finish-start);
  printf("   Tempo de Comunicacao     : %.7f\n",tcom);
  printf("   Tempo de Execucao        : %.7f\n\n",finish-start-tcom);
  printf("   Quant. de Comunicacao    : %d\n",qcom);
  printf("   Quant. de Bcast          : %d\n",qbcast);
```

```
  printf("   Quant. Dados Transmitidos  : %d\n",qtrans);
  printf("   Quant. Dados Recebidos     : %d\n",qrec);
  printf("   Quant. Dados Comunicados   : %d
%d\n\n",qdados,qtrans+qrec);
  printf("   Tempo Medio de Comunicacao :
%.7f\n",tcom/(qcom+qbcast));
  printf("   No. de Lacos (iteracao)    : %d\n",cont);
  printf("   No. de Processadores       : %d\n",p+1);
  printf("\n\n\n");

  for(I1=5; I1 <=10; I1++)
    for(I2=5; I2 <=10; I2++)
      printf("%d ",A[I1][I2]);

  printf("\n\n\n");

}
/* ------------------ */
/* Processo SLAVE     */
/* ------------------ */
slave()
{
int j;
int I01,I02,I1,I2,I1,I2;
int Y1,I2MIN;
MPI_Bcast(A,361, MPI_INT, 0, MPI_COMM_WORLD);
do {

  MPI_Recv(indice, 2, MPI_INT, 0, 99, MPI_COMM_WORLD, &status);

  if ((indice[0] != 1000) && (indice[1] != 1000)){

    I01 = indice[0];
    I02 = indice[1];
    j=0;

    for (I1=I01; I1 <= 16;I1+=2){
            Y1 = (I1 - I01)/2;
            I2MIN = 1 + (I02-1 + (2*Y1)) % 16;

            for (I2 = I2MIN; I2 <= 16; I2+=16){

                    I1 = I1 + 2;
                    I2 = I2 + 2;

                    A[I1][I2] = A[I1-2][I2-2] * 2;
                    retorno[j].I1 = I1;
                    retorno[j].I2 = I2;
                    retorno[j].Val = A[I1][I2];
                    j++;
            }
    }

    MPI_Send(retorno, (j*3), MPI_INT, 0, 99, MPI_COMM_WORLD);
  }
} while ((indice[0] != 1000) && (indice[1] != 1000));
}
```

MÉTODO TRANSFORMAÇÕES UNIMODULARES

```
/* LACO 2 - Metodo Transformacoes Unimodulares  */

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int p = 10;                          /* Numero de processos */

struct {                  /* Estrutura para passar armazenar os indices */
            int I1;
            int I2;
  } indice[10];

int expressao;

MPI_Status status;

int A[19][19];                                /* Matriz de calculo */

int retorno;                         /* Variavel para retorno da expressao
*/

main(argc, argv)
int argc;
char *argv[];

{

  int myrank;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

  if (myrank == 0)
     master();
  else
     slave();

  MPI_Finalize();
}

int MAX(a,b)
int a,b;
   {
            if (a > b) return a;
            else return b;
   }

int MIN(a,b)
int a,b;
   {
            if (a < b) return a;
            else return b;
   }


/* ------------------ */
/* Processo MASTER  */
/* ------------------ */
master()
{
  int I1,I2, cont;
  int k,j,x;                          /* Indice para os processos */
  double start, finish;                          /* Variaveis de tempo */
  double tcom, scom;
  int qcom, qdados, qtrans, qrec;

  for (I1=0; I1 <= 18; I1++)                     /* Inicializa matriz com 1's
*/
            for (I2=0; I2 <= 18; I2++)
                    A[I1][I2] = 1;

  cont = 0;
  scom = 0;
  tcom = 0;
  qcom = 0;
  qdados = 0;
  qtrans = 0;
  qrec = 0;

  j = 1;

  start = MPI_Wtime();
  for (I1 = 1; I1 <= 16; I1++) {
   if (MAX(-32,-I1-16) > MIN(-2,-I1-1)) x= -1; else x=1;
   for (I2 = MAX(-32,-I1-16); I2<=MIN(-2,-I1-1); I2=I2+x)
            {
              indice[j-1].I1 = I1;
              indice[j-1].I2 = -I1-I2;

              expressao = A[indice[j-1].I1-2+2][indice[j-1].I2-2+2];

              scom = MPI_Wtime();
              MPI_Send(&expressao, 1, MPI_INT, j, 99,
MPI_COMM_WORLD);
              tcom += MPI_Wtime() - scom;
              qcom ++;
              qdados ++;
              qtrans ++;

              j++;
```

```
            cont ++;

            if (j>p){
                    for(j=1; j<=p; j++){
                      scom = MPI_Wtime();
                      MPI_Recv(&retorno, 1, MPI_INT, j, 99,
MPI_COMM_WORLD, &status);
                      tcom += MPI_Wtime() - scom;
                      qcom ++;
                      A[indice[j-1].I1+2][indice[j-1].I2+2] = retorno;
                      qdados ++;
                      qrec ++;
                    }
                    j=1;
            }
          }
  }

  k = j;
  for(j=1; j<k; j++) {
    scom = MPI_Wtime();
    MPI_Recv(&retorno, 1, MPI_INT, j, 99, MPI_COMM_WORLD,
&status);
    tcom += MPI_Wtime() - scom;
    qcom ++;
    A[indice[j-1].I1+2][indice[j-1].I2+2] = retorno;
    qdados ++;
    qrec ++;
  }

  finish = MPI_Wtime();

  expressao = 1000;

  for(j=1 ; j<=p; j++) {
    MPI_Send(&expressao, 1, MPI_INT, j, 99, MPI_COMM_WORLD);
  }

  printf("\n\nLACO 2 - Metodo Transformacoes Unimodulares \n\n");

  printf(" \n Granularidade Fina \n\n");

  printf("\n\nFinalizando a execucao\n\n");
  printf("   Tempo Total gasto        : %.7f\n",finish-start);
  printf("   Tempo de Comunicacao     : %.7f\n",tcom);
  printf("   Tempo de Execucao        : %.7f\n\n",finish - start - tcom);
  printf("   Quant. de Comunicacoes   : %d\n",qcom);
  printf("   Quant. Dados Transmitidos : %d\n",qtrans);
  printf("   Quant. Dados Recebidos   : %d\n",qrec);
  printf("   Quant. Dados Comunicados  : %d
%d\n\n",qdados,qtrans+qrec);
  printf("   Tempo Medio de Comunicacao : %.7f\n",tcom/qcom);
  printf("   No. de lacos (iteracao)   : %d\n",cont);
  printf("   No. de Processadores      : %d\n",p+1);
  printf("\n\n\n");

  for(I1=5; I1 <=10; I1++)
    for(I2=5; I2 <=10; I2++)
      printf("%d ",A[I1][I2]);
  printf("\n\n\n");

}

/* ---------------- */
/* Processo SLAVE    */
/* ---------------- */
slave()
{
  do {
    MPI_Recv(&expressao, 1, MPI_INT, 0, 99, MPI_COMM_WORLD,
&status);
    if (expressao != 1000) {
            retorno = expressao * 2;

            MPI_Send(&retorno, 1, MPI_INT, 0, 99,
MPI_COMM_WORLD);
    }
  } while (expressao != 1000);
}
```

Método Alocação de Dados - Sem Duplicação de Dados

```
/* LACO 2 - Metodo Communcation - Free ( Sem dados duplicados   )*/

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int p = 10;                          /* Numero de processos */

int indice;

struct {                  /* Estrutura para receber os valores alterados */
          int I1;
          int I2;
          int Val;
  } retorno[16];

int Qr;                   /* Quant. de dados retornados apos iteracao  */


MPI_Status status;

int A[19][19];                                /* Matriz de calculo */

main(argc, argv)
int argc;
char *argv[];
{

  int myrank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  if (myrank == 0)
          master();
  else
          slave();

  MPI_Finalize();
}

int MAX(a,b)
int a,b;
  {
          if (a > b) return a;
          else return b;
  }

int MIN(a,b)
int a,b;
  {
          if (a < b) return a;
          else return b;
  }

/* ---------------- */
/* Processo MASTER  */
/* ---------------- */
master()
{
  int I01,I1,I2, cont;
  int k,j,i;                          /* Indice para os processos */
  double start, finish;                          /* Variaveis de tempo */
  double tcom, scom;
  int qcom, qbcast, qdados,qtrans,qrec;

  for (I1=0; I1 <= 18; I1++)                     /* Inicializa matriz com 1's
*/
          for (I2=0; I2 <= 18; I2++)
                  A[I1][I2] = 1;

  cont=0;
  tcom=0;
  scom=0;
  qcom=0;
  qbcast=0;
  j = 1;
  qdados=0;
  qtrans = 0;
  qrec = 0;
```

```
  start = MPI_Wtime();

  MPI_Bcast(A, 361, MPI_INT, 0, MPI_COMM_WORLD);
  tcom += MPI_Wtime() - start;
  qbcast ++;
  qdados += 361;

  for (I01 = 15; I01 >= -15; I01--)
  {
            indice = I01;

        scom = MPI_Wtime();
            MPI_Send(&indice, 1, MPI_INT, j, 99,
MPI_COMM_WORLD);
        tcom += MPI_Wtime() - scom;
        qcom ++;
        qdados ++;
        qtrans ++;
        cont ++;
            j++;

            if (j>p){
                        for(j=1; j<=p; j++){
                          scom = MPI_Wtime();
                          MPI_Recv(&Qr, 1, MPI_INT, j, 99,
MPI_COMM_WORLD, &status);
                          MPI_Recv(retorno, (Qr*3), MPI_INT, j, 99,
MPI_COMM_WORLD, &status);
                          tcom += MPI_Wtime() - scom;
                          qcom += 2;
                          qdados += (Qr*3);
                          qrec += Qr*3;
                          for (i=0; i<=Qr; i++){
                            A[retorno[i].I1][retorno[i].I2] = retorno[i].Val;
                          }
                        }
                        j=1;
            }
  }
  k=j;
  for(j=1; j < k; j++) {
    scom = MPI_Wtime();
    MPI_Recv(&Qr, 1, MPI_INT, j, 99, MPI_COMM_WORLD, &status);
    MPI_Recv(retorno, (Qr*3), MPI_INT, j, 99, MPI_COMM_WORLD,
&status);
    tcom += MPI_Wtime() - scom;
    qcom += 2;
    qdados += (Qr*3);
    qrec += Qr*3;
    for(i=0; i<=Qr; i++) {
            A[retorno[i].I1][retorno[i].I2] = retorno[i].Val;
    }
  }

  finish = MPI_Wtime();

  indice = 1000;

  for(j=1; j<=p; j++) {
    MPI_Send(&indice, 1, MPI_INT, j, 99, MPI_COMM_WORLD);
  }

  printf("\n\nLACO 2 - Metodo Communication - Free");
  printf("\n\n                                    Sem dados duplicados");

  printf("\n\nFinalizando a execucao\n\n");

  printf("   Tempo Total gasto          : %.7f\n",finish-start);
  printf("   Tempo de Comunicacao       : %.7f\n",tcom);
  printf("   Tempo de Execucao          : %.7f\n\n",finish-start-tcom);
  printf("   Quant. de Comunicacoes     : %d\n",qcom);
  printf("   Quant. de Bcast            : %d\n",qbcast);
  printf("   Quant. Dados Transmitidos  : %d\n",qtrans);
  printf("   Quant. Dados Recebidos     : %d\n",qrec);
  printf("   Quant. Dados Comunicados   : %d %d\n\n",qdados,
qtrans+qrec);
  printf("   Tempo Medio de Comunicacao :
%.7f\n",tcom/(qcom+qbcast));
  printf("   No. de Lacos (iteracao)    : %d\n",cont);
  printf("   No. de Processadores       : %d\n",p+1);
  printf("\n\n\n");

  for(I1=5; I1 <=10; I1++)
    for(I2 = 5; I2 <=10; I2++)
      printf("%d ",A[I1][I2]);

  printf("\n\n\n");
}


/* ------------------- */
/* Processo SLAVE      */
/* ------------------- */
slave()
{
int j;
int I01,I1,I1,I2;

MPI_Bcast(A, 361, MPI_INT, 0, MPI_COMM_WORLD);

do {

  MPI_Recv(&indice, 1, MPI_INT, 0, 99, MPI_COMM_WORLD, &status);

  if (indice != 1000) {

    I01 = indice;

    j=0;

    for (I1=MAX(1,I01+1); I1 <= MIN(16,I01+16);I1++){

            I1 = I1 + 2;
            I2 = I1 - I01 + 2;

            A[I1][I2] = A[I1-2][I2-2] * 2;

            retorno[j].I1 = I1;
            retorno[j].I2 = I2;
            retorno[j].Val = A[I1][I2];
            j++;
    }
    Qr = j;
    MPI_Send(&Qr, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
    MPI_Send(retorno, (j*3), MPI_INT, 0, 99, MPI_COMM_WORLD);
  }
} while (indice != 1000);
}
```

MÉTODO ALOCAÇÃO DE DADOS - COM DUPLICAÇÃO DE DADOS

```
/* LACO 2 - Metodo Communication - Free  */
/*                    Com dados duplicados
            */

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int p = 10;                    /* Numero de processos */

struct {                /* Estrutura para passar armazenar os indices */
            int I1;
            int I2;
      } indice[10];

int expressao;

MPI_Status status;

int A[23][23];                                  /* Matriz de calculo */

int retorno;                    /* Variavel para retorno da expressao
*/
```

```
main(argc, argv)
int argc;
char **argv;
{

  int myrank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  if (myrank == 0)
     master();
  else
     slave();

  MPI_Finalize();
}

/* ---------------- */
/* Processo MASTER  */
/* ---------------- */
master()
{
  int I1,I2, cont;
  int j,k;                          /* Indice para os processos */

  int p1,p2;
  int b1,b2;

  double start, finish;                     /* Variaveis de tempo */
  double tcom, scom;
  int qcom, qdados, qtrans, qrec;

  for (I1=0; I1 <= 22; I1++)                /* Inicializa matriz com 1's
*/
            for (I2=0; I2 <= 22; I2++)
                        A[I1][I2] = 1;

  cont = 0;
  qcom = 0;
  scom =0;
  tcom =0;
  j = 1;
  qdados=0;
  qtrans = 0;
  qrec = 0;

  p1 = 1;
  p2 = 1;

  b1 = p1-1;
  b2 = p2-1;

  start = MPI_Wtime();

  for (I1 = (1+(b1-(1 % p1)) % p1); I1 <= 16; I1+=p1)
  for (I2 = (1+(b2-(1 % p2)) % p2); I2 <= 16; I2+=p2){

            indice[j-1].I1 = I1;
            indice[j-1].I2 = I2;

            expressao = A[I1-2+2][I2-2+2];

       scom = MPI_Wtime();
            MPI_Send(&expressao, 1, MPI_INT, j, 99,
MPI_COMM_WORLD);
       tcom += MPI_Wtime() - scom;
       qcom ++;
       qdados ++;
       qtrans ++;
       j++;
       cont ++;

            if (j>p){
                        for(j=1; j<=p; j++){
                          scom = MPI_Wtime();
                          MPI_Recv(&retorno, 1, MPI_INT, j, 99,
MPI_COMM_WORLD, &status);
                          tcom += MPI_Wtime() - scom;
                          qcom ++;
                          A[indice[j-1].I1+2][indice[j-1].I2+2] = retorno;
                          qdados ++;
                          qrec ++;
                        }
                        j=1;
            }
  }

  k = j;
  for(j=1; j < k; j++) {
    scom = MPI_Wtime();
    MPI_Recv(&retorno, 1, MPI_INT, j, 99, MPI_COMM_WORLD,
&status);
    tcom += MPI_Wtime() - scom;
    qcom ++;
    A[indice[j-1].I1+2][indice[j-1].I2+2] = retorno;
    qdados ++;
    qrec ++;
  }

  finish = MPI_Wtime();

  expressao = 1000;

  for(j=1; j<=p; j++) {
    MPI_Send(&expressao, 1, MPI_INT, j, 99, MPI_COMM_WORLD);
  }

  printf("\n\nLACO 2 - Metodo Communication - Free ");
  printf("\n\n                                        Com dados
duplicados\n\n");

  printf("\n\nFinalizando a execucao\n\n");
  printf("   Tempo Total gasto          : %.7f\n",finish-start);
  printf("   Tempo de Comunicacao       : %.7f\n",tcom);
  printf("   Tempo de Execucao          : %.7f\n\n",finish-start-tcom);
  printf("   Quant. de Comunicaoes      : %d\n",qcom);
  printf("   Quant. Dados Transmitidos  : %d\n",qtrans);
  printf("   Quant. Dados Recebidos     : %d\n",qrec);
  printf("   Quant. Dados Comunicadoes  : %d %d\n\n",qdados,
qtrans+qrec);
  printf("   Tempo Medio de Comunicacao : %.7f\n",tcom/qcom);
  printf("   No.de lacos (iteracao)     : %d\n",cont);
  printf("   No. de Processadores       : %d\n",p+1);
  printf("\n\n\n");

  for(I1 = 5;I1 <=10 ; I1++)
    for(I2 = 5; I2 <=10; I2++)
      printf("%d ",A[I1][I2]);

  printf("\n\n\n");

}


/* ---------------- */
/* Processo SLAVE   */
/* ---------------- */
slave()
{
 do {

   MPI_Recv(&expressao, 1, MPI_INT, 0, 99, MPI_COMM_WORLD,
&status);

   if (expressao != 1000) {

            retorno = expressao * 2;

            MPI_Send(&retorno, 1, MPI_INT, 0, 99,
MPI_COMM_WORLD);
   }
 } while (expressao != 1000);
}
```

ANEXO IV - TABELAS COM OS RESULTADOS DA SIMULAÇÃO

- **Simulação para o Laço 1**
  - Método Hiperplano
  - Método Particionamento & Rotulação
  - Método Transformações Unimodulares
  - Método Alocação de Dados
    - Com Duplicação dos Dados

- **Simulação para o Laço 2**
  - Método Hiperplano
  - Método Particionamento & Rotulação
  - Método Transformações Unimodulares
  - Método Alocação de Dados
    - Com Duplicação dos Dados
    - Sem Duplicação dos Dados

SIMULAÇÃO PARA O LAÇO 1

MÉTODO HIPERPLANO

Método Hiperplano	Número de Processadores						Média
	2	3	5	7	9	11	
Tempo Total	894,3603	1.095,3361	1.075,0639	1.040,4902	1.021,0471	995,1943	1.045,4263
Tempo de Comun.	893,9136	1.094,8729	1.074,6305	1.040,0460	1.020,6066	994,7468	1.044,9806
Tempo de Exec.	0,4467	0,4631	0,4334	0,4442	0,4405	0,4475	0,4457
Quant. Comun.	8192	8192	8192	8192	8192	8192	8192
Quant. Bcast	0	0	0	0	0	0	0
Quant. Dados	20480	20480	20480	20480	20480	20480	20480
Quant.Dados Trans	16384	16384	16384	16384	16384	16384	16384
Quant. Dados Rec.	4096	4096	4096	4096	4096	4096	4096
Slowdown	0,0072	0,0070	0,0075	0,0073	0,0073	0,0072	0,0072
No. Proc. Possíveis	4096	4096	4096	4096	4096	4096	4096
Slowdown Relativo		1,0367	0,9358	1,0250	0,9916	1,0158	1,0010

Tabela V.1. Resultados da Simulação do laço 1 - Método Hiperplano

MÉTODO PARTICIONAMENTO & ROTULAÇÃO

Método Part. & Rotul.	Número de Processadores						Média
	2	3	5	7	9	11	
Tempo Total	0,6203	18,6384	16,9787	18,3611	21,3195	17,4254	18,5446
Tempo de Comun.	0,6112	18,6293	16,9696	18,3524	21,3111	17,4165	18,5358
Tempo de Exec.	0,0091	0,0091	0,0091	0,0087	0,0084	0,0090	0,0088
Quant. Comun.	156	156	156	156	156	156	156
Quant. Bcast	1	1	1	1	1	1	1
Quant. Dados	16540	16540	16540	16540	16540	16540	16540
Quant.Dados Trans	156	156	156	156	156	156	156
Quant. Dados Rec.	16384	16384	16384	16384	16384	16384	16384
Slowdown	0,3542	0,3553	0,3565	0,3720	0,3868	0,3600	0,3642
No. Proc. Possíveis	52	52	52	52	52	52	52
Slowdown Relativo		0,9969	0,9967	0,9583	0,9618	1,0746	0,9977

Tabela V.2. Resultados da Simulação do laço 1 - Método Particionamento & Rotulação

MÉTODO TRANSFORMAÇÕES UNIMODULARES

Método Unimodular	Número de Processadores						Média
	2	3	5	7	9	11	
Tempo Total	639,5772	1.272,9490	1.097,1631	1.076,4709	1.029,0665	1.009,7552	1.097,0809
Tempo de Comun.	639,1064	1.272,4977	1.096,7070	1.076,0101	1.028,5929	1.009,2803	1.096,6176
Tempo de Exec.	0,4708	0,4512	0,4560	0,4608	0,4736	0,4749	0,4633
Quant. Comun.	8192	8192	8192	8192	8192	8192	8192
Quant. Bcast	0	0	0	0	0	0	0
Quant. Dados	20480	20480	20480	20480	20480	20480	20480
Quant.Dados Trans	16384	16384	16384	16384	16384	16384	16384
Quant. Dados Rec.	4096	4096	4096	4096	4096	4096	4096
Slowdown	0,0069	0,0072	0,0071	0,0070	0,0068	0,0068	0,0070
No. Proc. Possíveis	4096	4096	4096	4096	4096	4096	4096
Slowdown Relativo		0,9584	1,0106	1,0105	1,0277	1,0027	1,0020

Tabela V.3. Resultados da Simulação do laço 1 - Método Transformações Unimodular

MÉTODO ALOCAÇÃO DE DADOS - COM DUPLICAÇÃO DOS DADOS

Método Com Duplicação	Número de Processadores						Média
	2	3	5	7	9	11	
Tempo Total	1.004,7774	1.194,3942	1.118,7970	1.085,3885	1.016,2700	988,8408	1.080,7381
Tempo de Comun.	1.004,3531	1.193,9277	1.118,3582	1.084,9382	1.015,8261	988,4136	1.080,2928
Tempo de Exec.	0,4243	0,4666	0,4388	0,4503	0,4439	0,4271	0,4453
Quant. Comun.	8192	8192	8192	8192	8192	8192	8192
Quant. Bcast	0	0	0	0	0	0	0
Quant. Dados	20480	20480	20480	20480	20480	20480	20480
Quant.Dados Trans	16384	16384	16384	16384	16384	16384	16384
Quant. Dados Rec.	4096	4096	4096	4096	4096	4096	4096
Slowdown	0,0076	0,0069	0,0074	0,0072	0,0073	0,0076	0,0073
No. Proc. Possíveis	4096	4096	4096	4096	4096	4096	4096
Slowdown Relativo		1,0997	0,9405	1,0262	0,9859	0,9621	1,0029

Tabela V.4. Resultados da Simulação do laço 1 - Método Alocação de Dados - Com Duplicação dos Dados

SIMULAÇÃO PARA O LAÇO 2

MÉTODO HIPERPLANO

Método Hiperplano	Número de Processadores						Média
	2	3	5	7	9	11	
Tempo Total	1,2444	85,9556	70,4662	65,8465	61,4509	60,0782	68,7595
Tempo de Comun.	1,2195	85,9296	70,4408	65,8211	61,4238	60,0522	68,7335
Tempo de Exec.	0,0249	0,0259	0,0254	0,0254	0,0271	0,0261	0,0260
Quant. Comun.	512	512	512	512	512	512	512
Quant. Bcast	0	0	0	0	0	0	0
Quant. Dados	512	512	512	512	512	512	512
Quant.Dados Trans	256	256	256	256	256	256	256
Quant. Dados Rec.	256	256	256	256	256	256	256
Slowdown	0,0044	0,0042	0,0043	0,0043	0,0041	0,0042	0,0043
No. Proc. Possíveis	256	256	256	256	256	256	256
Slowdown Relativo		1,0409	0,9798	1,0003	1,0646	0,9623	1,0096

Tabela V.5. Resultados da Simulação do laço 2 - Método Hiperplano

MÉTODO PARTICIONAMENTO & ROTULAÇÃO

Método Part. & Rotul.	Número de Processadores						Média
	2	3	5	7	9	11	
Tempo Total	0,1470	8,4299	7,4134	6,4380	4,9784	3,7291	6,1978
Tempo de Comun.	0,1438	8,4265	7,4099	6,4345	4,9751	3,7258	6,1944
Tempo de Exec.	0,0032	0,0034	0,0035	0,0034	0,0033	0,0034	0,0034
Quant. Comun.	64	64	64	64	64	64	64
Quant. Bcast	1	1	1	1	1	1	1
Quant. Dados	832	832	832	832	832	832	832
Quant.Dados Trans	64	64	64	64	64	64	64
Quant. Dados Rec.	768	768	768	768	768	768	768
Slowdown	0,0341	0,0326	0,0317	0,0320	0,0329	0,0327	0,0327
No. Proc. Possíveis	32	32	32	32	32	32	32
Slowdown Relativo		1,0474	1,0275	0,9896	0,9732	1,0069	1,0089

Tabela V.6. Resultados da Simulação do laço 2 - Método Particionamento & Rotulação

MÉTODO TRANSFORMAÇÕES UNIMODULARES

Método Unimodular	Número de Processadores						Média
	2	3	5	7	9	11	
Tempo Total	54,2649	85,0789	69,5947	65,6956	64,4396	59,7710	68,9160
Tempo de Comun.	54,2381	85,0528	69,5683	65,6701	64,4137	59,7441	68,8898
Tempo de Exec.	0,0267	0,0260	0,0264	0,0255	0,0260	0,0269	0,0262
Quant. Comun.	512	512	512	512	512	512	512
Quant. Bcast	0	0	0	0	0	0	0
Quant. Dados	512	512	512	512	512	512	512
Quant.Dados Trans	256	256	256	256	256	256	256
Quant. Dados Rec.	256	256	256	256	256	256	256
Slowdown	0,0041	0,0042	0,0042	0,0043	0,0042	0,0041	0,0042
No. Proc. Possíveis	256	256	256	256	256	256	256
Slowdown Relativo		0,9735	1,0155	0,9641	1,0188	1,0348	1,0013

Tabela V.7. Resultados da Simulação do laço 2 - Método Transformações Unimodular

MÉTODO ALOCAÇÃO DE DADOS - COM DUPLICAÇÃO DOS DADOS

Método Com Duplicação	Número de Processadores						Média
	2	3	5	7	9	11	
Tempo Total	25,8088	76,3050	68,7124	65,0599	64,5789	63,7786	67,6870
Tempo de Comun.	25,7837	76,2802	68,6876	65,0350	64,5539	63,7516	67,6616
Tempo de Exec.	0,0251	0,0249	0,0249	0,0249	0,0250	0,0269	0,0253
Quant. Comun.	512	512	512	512	512	512	512
Quant. Bcast	0	0	0	0	0	0	0
Quant. Dados	512	512	512	512	512	512	512
Quant.Dados Trans	256	256	256	256	256	256	256
Quant. Dados Rec.	256	256	256	256	256	256	256
Slowdown	0,0044	0,0044	0,0044	0,0044	0,0044	0,0041	0,0044
No. Proc. Possíveis	256	256	256	256	256	256	256
Slowdown Relativo		0,9897	1,0003	1,0014	1,0028	1,0792	1,0147

Tabela V.8. Resultados da Simulação do laço 2 - Método Alocação de Dados - Com Duplicação dos Dados

MÉTODO ALOCAÇÃO DE DADOS - SEM DUPLICAÇÃO DOS DADOS

Método Sem Duplicação	Número de Processadores						Média
	2	3	5	7	9	11	
Tempo Total	0,1962	10,3926	8,7279	7,0762	6,5706	5,9054	7,7345
Tempo de Comun.	0,1926	10,3894	8,7246	7,0729	6,5673	5,9021	7,7312
Tempo de Exec.	0,0037	0,0032	0,0033	0,0033	0,0033	0,0033	0,0033
Quant. Comun.	93	93	93	93	93	93	93
Quant. Bcast	1	1	1	1	1	1	1
Quant. Dados	799	799	799	799	799	799	799
Quant.Dados Trans	31	31	31	31	31	31	31
Quant. Dados Rec.	768	768	768	768	768	768	768
Slowdown	0,0300	0,0339	0,0329	0,0337	0,0331	0,0333	0,0328
No. Proc. Possíveis	31	31	31	31	31	31	31
Slowdown Relativo		0,8851	1,0321	0,9755	1,0175	0,9952	0,9811

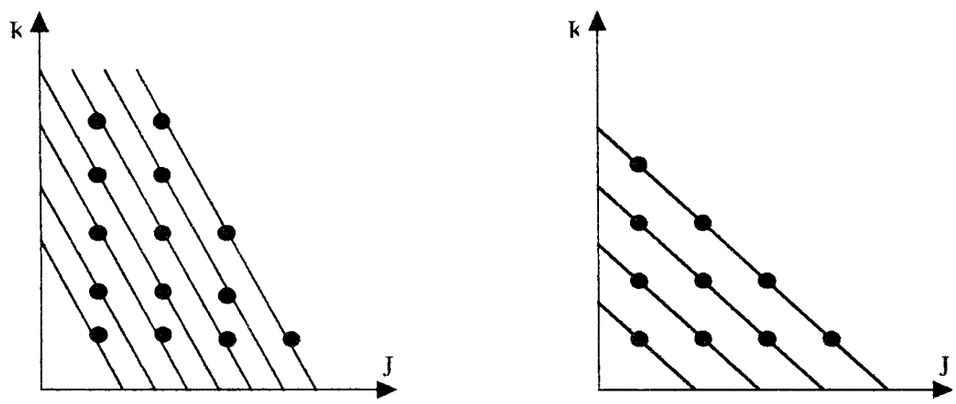
Tabela V.9. Resultados da Simulação do laço 2 - Método Alocação de Dados - Sem Duplicação dos Dados

GLOSSÁRIO

- **Referência** a uma variável, é a aparição desta variável em um comando.

- **Espaço de Índices** é o espaço multi-dimensional definido pelos índices do aninhamento.

- **Frente de Onda** é definida como sendo o plano definido pelos índices de um laço a cada passo, onde os planos podem ser executados simultaneamente, por exemplo na figura abaixo, temos dois exemplos de frente de onda.



2M+N-2 Iterações M+N-1 Iterações

- **Tupla** é um par ordenado de objetos. Escrevemos <x,y> para denotar o par x e y nesta ordem. Note que x pode ser igual a y.

- **Ordem Lexicográfica** é uma relação de ordenação entre vetores.

- **Combinação Linear** de um conjunto de vetores, uma combinação $x = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_m v_m$, onde cada α_i é um número real.

- **Hiperplano** é a combinação linear de índices representa um plano em um espaço multi-dimensional.

- **Linearmente Independente** Um conjunto de vetores é linearmente independente se existe uma combinação não linear com um ou mais coeficientes não zero que produza o vetor zero.

- **Espaço de Iteração** O subconjunto de uma ***Lattice*** de inteiros que corresponde às iterações do laço aninhado.

- ***Lattice*** é um subconjunto de Z^n que é gerado por uma combinação inteira de um conjunto de vetores.

- **Vetor Índice** é um vetor onde os elementos são os índices de um laço aninhado.

- **Vetor Dependência** é um vetor onde os elementos são as dependências de dados.

- **Ordem Topológica** é uma ordem ***traversal*** dos nós em um DAG tal que todos os predecessores de um nó são visitados antes do próprio nó.

- **Vetores Lexicograficamente Positivos** Ver Lexicograficamente positivos.

- **Matriz Triangular Inferior** é uma matriz quadrada com elementos zero acima da diagonal.

- ***Fork*** é a ativação de outras tarefas paralelas ou processadores paralelos.

- **Granularidade** Definido no Capítulo 2

- **Forma Canônica** são laços aninhados permutável completamente.

- **Permutável Completamente** são laços que podem ser combinados entre si.

- **Espaços Ortogonais** são espaços formados por dois planos perpendiculares, formando um ângulo de 90° entre si.

- **DAG** Grafo Orientado Aciclico, um grafo que contêm nenhum ciclo.

- **Vetor Distância** é o vetor diferença entre os vetores iteração das iterações origem e destino de uma relação de dependência.

- **Vetores Direção** é um vetor ordenação entre os vetores iteração de iterações origem e destino das relações de dependência.

- **Produto Ponto** de dois vetores x^n e y^n, o somatório $\sum_{i=1}^{n} x_i . y_i$, denotado por x.y.

- **Espaço Nulo** de uma matriz nxm, o subespaço de R^m que é mapeado sobre um vetor zero pela aplicação da matriz.

- **Lexicograficamente Não-Negativas** quando tivermos um vetor v, tal que $0 \prec v$; isto é, a primeira entrada não zero de v, se for, é positiva.

- **Lexicograficamente Positivas** quando tivermos um vetor v tal que $0 \prec v$; isto é, a primeira entrada não zero de v é positiva.

- **Transformação Linear** é uma transformação que muda os elementos, por meio de uma combinação linear destes elementos.

- **Dimensionalidade** de um subespaço, o número de vetores linearmente independentes em um conjunto de vetores essa pequena extensão o subespaço, ou o número de vetores em uma base para o subespaço.

- **Laços *Serializing*** são os laços com componentes dependência ambos $+\infty$ e $-\infty$.

- **Matriz Singular** é uma matriz quadrada que não tem inversa; isto é, uma matriz quadrada na qual vetores coluna ou vetores linha são linearmente dependentes.

- **Matriz Diagonal** é uma matriz onde todos os elementos são zero, exceto na diagonal.

- **Semi Plano** é uma fatia de um plano.

- **Matriz Identidade** é uma matriz com uns na diagonal e zeros nos demais elementos, escrita como I; para alguns vetores V^n, o produto $I^{nxn}V^n = V^n$.

- **Matriz Unimodular** é uma matriz inteira com determinante igual a ± 1.

- **GCD** *Greatest Common Divisor*, é o grande inteiro positivo que eventualmente divide dois ou mais outros inteiros.

- **Predecessor** em um gráfico direto, um nó X conectado com um nó Y por uma *EDGE*.

- ***Edge*** é uma tupla <X,Y> onde X e Y são os vértice de um gráfico; isto é usualmente escrito como $X \rightarrow Y$.

- **Vetor Zero** é um vetor com todos os elementos zero.

REFERÊNCIAS BIBLIOGRÁFICAS

[Blume *et al.* 1992] - BLUME, William & EIGENMANN, Rudolf, **"Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs"**. IEEE Transactions on Parallel and Distributed Systems. November 1992. Vol. 3. No. 6. Pág. 643 - 56.

[Cann 1993] - CANN D.C.; **"Sisal 1.2: A brief introduction and tutorial"**. Lawrence Livermore National Laboratory. 1993.

[Chang 1995] - CHANG, Pohua P.; **"The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors"**. IEEE Transactions on Computers. March 1995. Vol 44. Nº 3. Pág. 353-70.

[Chen *et al.* 1994] - CHEN, Tzung-Shi & SHEU, Jang-Ping., **"Communication-Free Data Allocation Techniques for Parallelizing Compilers on Multicomputers"** . IEEE Transactions on Parallel and Distributed Systems. September 1994. Vol. 5. No. 9. Pág. 924 - 38.

[Darte *et al.* 1994] - DARTE, Alain & ROBERT, Yves., **"Constructive Methods for Scheduling Uniform Loop Nests"**. IEEE Transactions on Parallel and Distributed Systems. August 1994. Vol. 5. No. 8. Pág. 814 - 22.

[D'Hollander 1992] - D'HOLLANDER, Erik H., **"Partitioning and Labeling of Loops by Unimodular Transformations"** . IEEE Transactions on Parallel and Distributed Systems. July 1992. Vol. 3. No. 4. Pág. 465 - 76.

[Dongarra *et al.* 1995] - DONGARRA, J.J.; OTTO, S. W.; Snir, M. & WALKER, D. **"An introduction to the MPI Standard"**. A ser publicado na *Communications of the ACM*. January 1995.

[Eigenmann *et al.* 1992] - EIGENMANN, R., HOEFLINGER, J., LI, Z. & PADUA, D., **"Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs"**. Springer-Verlag. 1992. Pág. 65-83.

[Gentzsch 1990] - GENTZSCH, W., **"Performance Evaluation for Shared-Memory Parallel Computers"**. Parallel Computing 89. North-Holland: Elsevier Science Publishers B.V., 1990. Pág. 3 - 4.

[Girkar *et al.* 1992] - GIRKAR, Milind. & POLYCHRONOPOULOS, Constantine D., **"Automatic Extraction of Functional Parallelism from Ordinary Programs"**. IEEE Transactions on Parallel and Distributed Systems. March 1992. Vol. 3. No. 2. Pág. 166 - 78.

[Hilhorst *et al.* 1987] - HILHORST, D. & BRAAM, H.D.A., SIPS, H.J., **"Some Methods for the Automatic Parallelization of Programs; the Hyperplane Method and its Application on the Delft Parallel Processor"**. Algorithms and Applications on Vector and Parallel Computers. North - Holland: Elsevier Science Publishers B.V. 1987. Pág. 47 - 73.

[Hockey *et al.* 1988] - HOCKEY R.W., JESSHOPE C.R.; **"Parallel Computer 2"**. Adam Hilger. 1988.

[Hwang 1984] - HWANG, K., **"Advanced Parallel Processing with Supercomputer Architectures"**, Proceeding of the IEEE, vol. 75, n. 10, October. 84. Pág. 1348-79.

[Hwang *et al.* 1985] - HWANG, Kai. & BRIGGS, Fayé A. **"Computer Architecture and Parallel Processing"**. McGraw-Hill International Editions. 1985. Pág. 6, 533-56, 613-42.

[Ibbett *et al.* 1989] - IBBETT, R.N. & TOPHAN, N.P. **"Architecture of High Performance Computers II"**, 1989. Pág. 1-5, 83-108, 141-68.

[Lee 1995] - LEE, Gyungho, **"Parallelizing Iterative Loops wiht Conditional Branching"**. IEEE Transactions on Parallel & Distributed Systems. February 1995. Vol 6. Nº 2. Pág. 185-9.

[Lu *et al.* 1992] - LU, L.-C. & CHEN, M., **"Parallelizing Loops with Indirect Array References or Pointers"** . Springer-Verlag. 1992. Pág. 201 - 17.

[McBryan 1994] - MCBRYAN, O. A. **"An overview of message passing environments"**. Parallel Computing 20. March 1994. Pág. 417-44.

[Mokarzel *et al.* 1988] - MOKARZEL, Fábio Carneiro. & PANETTA, Jairo., **"Reestruturação Automática de Programas Sequenciais para Processamento Paralelo"** . II Simpósio Brasileiro de Arquitetura de Computadores - Processamento Paralelo (II -SBAC-PP) - Anais Vol 1. Setembro de 1988. Pág. 5.B.1.1 - 7.

[MPI Forum 1994] - **"Message Passing Interface Standard"**. MPI Forum. 1994.

[Navaux 1988] - NAVAUX, Philippe O.A. **"Introdução ao Processamento Paralelo"** . II Simpósito Brasileiro de Arquitetura de Computadores - Processamento Paralelo (II - SBAC-PP) - Anais Vol 1. Setembro de 1988. Pág. I.1 - 14.

[NEC 1993] - **"SX World"** . NEC Corporation. Summer 1993. Número 12.

[Padua *et al.* 1986] - PADUA, D.A. & WOLFE, M.j., **"Advanced Compiler Otimizations for Supercomputers"**, Communications of the ACM, v. 29, n. 12, December 86. Pág. 1184 -201.

[Perrott 1990] - PERROTT, R.H., **"Parallel Languages and Parallel Programming"** . Parallel Computing 89. North-Holland: Elsevier Science Publishers B.V., 1990. Pág. 47 - 58.

[Pountain *et al.* 1988] - POUNTAIN D., MAY D.;**"A tutorial introduction to occam programming"**.BSP. 1988.

[Quealy] - QUEALY, A. **"An introduction to Message Passing"**. slides. NASA Lewis Research Center.

[Saltz *et al.* 1991] - SALTZ, Joel H. & MIRCHANDENEY, Ravi., CROWLEY, Kay., **"Run-Time Parallelization and Scheduling of Loops"** . IEEE Transactions on Computers. May 1991. Vol. 40. No. 5. Pág. 603 - 11.

[Tanenbaum 1992] - TANENBAUM, Andrew S. **"Organização Estruturada de Computadores"**, Rio de Janeiro: Prentice/Hall, 1992. Pág. 386-420.

[Tzen *et al.* 1993] - TZEN, Ten H. & NI, Lionel M., **"Dependence Uniformatization: A Loop Parallelization Technique"**. IEEE Transactions on Parallel and Distributed Systems. May 1993. Vol. 4. No. 5. Pág. 547 - 58.

[Walker 1995] - WALKER, D.W. **"MPI: From Fundamentals To Applications"**. Oak Ridge National Laboratory. April 1995.

[Wolf *et al.* 1991] - WOLF, Michael E. & LAM, Monica S., **"A loop Transformation Theory and an Algorithm to Maximize Parallelism"** . IEEE Transactions on Parallel and Distributed Systems. October 1991. Vol. 2. No. 4. Pág. 452 - 71.

[Wolfe 1992] - WOLFE, Michael., **"New Program Restructuring Technology"**. Springer-Verlag. 1992. Pág. 13 - 36.

[Wolfe *et al.* 1992] - WOLFE, Michael. & TSENG, Chau-Wen., **"The Power Test for Data Dependence"** . IEEE Transactions on Parallel and Distributed Systems. September 1992. Vol. 3. No. 5. Pág. 591 - 601.

[Wolfe 1996] - WOLFE, Michael., **"High Performance Compilers for Parallel Computing"**. New Jersey: Addison-Wesley. 1996.

[Zhiyuan *et al.* 1990] - ZHIYUAN, Li. & YEW, Pen-Chung., ZHU, Chuan-Qi., **"An Efficient Data Dependence Analysis for Parallelizing Compilers"** . IEEE Transactions on Parallel and Distributed Systems. January 1990. Vol. 1. No. 1. Pág. 26 - 34.