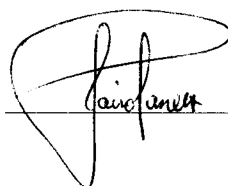
Paralelismo e Sincronização em Laços

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pelo Sr. EDUARDO VOIGT e aprovada pela Comissão Julgadora.

Campinas, 21 de Dezembro de 1991



Prof. Dr. JAIRO PANETTA (orientador)

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para a obtenção do título de MESTRE em Ciência da Computação

Agradecimentos

Ao meu orientador, Jairo, pelos incentivos, sugestões e constante apoio no decorrer da pesquisa que originou esta dissertação. Sua determinação sempre me motivou nos momentos mais difíceis (e difícei$).

Ao professor Catto, pelas valiosas sugestões que deram forma final a esta dissertação.

A todos os colegas do Projeto Computação Científica, pelos conselhos e discussões nestes últimos dois anos.

Por fim, quero agradecer e dedicar esta dissertação à minha família e à Raquel. Com sua presença e carinho, tudo fica mais fácil.

Resumo

A reestruturação de programas seqüenciais para processamento paralelo, em particular nos laços mais internos, mostra-se uma das formas mais promissoras para aumentar o desempenho destes programas. A exploração do paralelismo nestes laços requer uma análise das dependências de dados existentes entre as iterações e a inserção de sincronização apropriada para a execução paralela correta.

Motivados pela dificuldade encontrada pelos compiladores reestruturadores atuais, na detecção e inserção da sincronização, propõe-se, com esta dissertação, um esquema de paralelização e sincronização de laços mais internos que, atuando durante a execução, elimina anti-dependências e dependências de saída, resultando em laços com atribuições únicas. As dependências de fluxo restantes são garantidas por uma primitiva de sincronização apropriada, derivada do *full/empty bit*. Esta proposta de sincronização, diferentemente dos esquemas tradicionais, dispensa a detecção das dependências e o cálculo das distâncias entre os pontos de sincronização.

Conteúdo

1 Introdução **3**

1.1 Motivação 3

1.2 Tarefas e Problemas de um Compilador Reestruturador 4

1.3 Objetivo do Trabalho 5

1.4 Divisão do Trabalho 6

2 Primitivas de Sincronização **7**

2.1 Dependências de Dados 7

2.2 Sincronização 9

2.3 Primitivas de Sincronização 10

2.4 Propostas 13

2.4.1 Paradigmas de Utilização das Primitivas 13

2.4.2 *Test-and-set* 15

2.4.3 *Await/Advance* 16

2.4.4 *Full/empty bit* 18

2.4.5 *Fetch-and-add* 20

2.5 Razões para uma Nova Primitiva 22

2.5.1 Sincronização por Semáforos 23

2.5.2 Sincronização por Operações nos Dados 25

2.5.3 Sincronização por Controle de Acesso aos Dados 25

2.6 A Primitiva *Flowbit* 27

2.7 Teste Comparativo 28

2.8 Comentários Finais 31

3 Um Esquema de Sincronização para Laços Paralelos **32**

3.1 Esquema de Execução Paralela 32

3.2 Condições Suficientes para a Execução Correta de Laços Paralelos 34

3.3 Garantindo a Semântica em Programas com Atribuições Únicas 35

3.4 Dificuldades para Garantir a Semântica em Programas com Múltiplas Atribuições . . . 36
3.4.1 Anti-dependências e Dependências de Saída . . . 36
3.4.2 Deficiência na Detecção e Inserção da Sincronização . . . 37
3.5 Uma Proposta Inicial . . . 40
3.6 Uma Nova Proposta . . . 42
3.6.1 Revendo as Condições Básicas para Garantir a Semântica . . . 42
3.6.2 Reendereçamento de Escritas . . . 42
3.6.3 Reendereçamento de Leituras . . . 45
3.7 Considerações Sobre o Esquema . . . 49
3.8 Paralelismo . . . 50
3.9 Um Experimento . . . 55
3.9.1 Resultados . . . 56

4 Inserindo Especialização no Compilador **58**
4.1 Alternativas para Aumentar a Eficiência . . . 59
4.2 Análise de Dependências . . . 60
4.2.1 Limitações da Análise de Dependências Tradicional . . . 61
4.3 Previsão de Acessos Conflitantes . . . 62
4.3.1 Variáveis Escalares . . . 63
4.3.2 *Arrays* Unidimensionais . . . 65
4.4 Definição da Forma de Paralelização do Laço . . . 67
4.5 Considerações Finais . . . 73

5 Conclusões **74**

Capítulo 1

Introdução

1.1 Motivação

A rápida evolução da ciência experimental fez do computador o veículo primário no teste de teorias onde não é viável a construção de aparato físico experimental, como os modelos numéricos de previsão do tempo e estudo do universo.

A necessidade inegável de poder computacional, para simular tais modelos, motivou as comunidades acadêmica e industrial a buscarem novas soluções tecnológicas para aumentar a velocidade dos computadores existentes. A exploração do paralelismo intrínseco às aplicações científicas mostra-se uma das formas mais promissoras para se atingir tal objetivo. Para tanto, novas propostas de arquiteturas paralelas foram desenvolvidas, correspondendo a vários paradigmas de exploração de paralelismo.

Dentre estes paradigmas, destacam-se os computadores vetoriais (ex. CRAY) e as máquinas multiprocessadoras SIMD[1] e MIMD[2], na classificação de Flynn [1]. Esta última classe divide-se nas propostas de memória distribuída entre os processadores (ex. Hipercubo) e memória compartilhada (ex. Alliant FX/8).

A programação de tais máquinas carece, até o presente momento, de linguagens que permitam a expressão explícita de paralelismo e sejam universalmente aceitas [2]. Por outro lado, o enorme investimento dos últimos trinta anos na programação seqüencial de propósito geral, em particular programas científicos codificados em FORTRAN, justifica a existência de ferramentas que detectem e explorem o paralelismo implícito nestes programas, sem a necessidade de recodificação pelo programador. Estas ferramentas existem hoje na forma de compiladores reestruturadores.

[1] Single Instruction stream, Multiple Data stream

[2] Multiple Instruction stream, Multiple Data stream

A transformação de um programa seqüencial em um programa paralelo semanticamente equivalente exige do compilador a *detecção* do paralelismo intrínseco ao programa e a *exploração* deste paralelismo em uma determinada arquitetura. O mapeamento do paralelismo na arquitetura caracteriza uma forte dependência entre o compilador e a máquina alvo. Como exemplo, temos a exploração de paralelismo através da vetorização de comandos internos a um laço nas máquinas que fazem uso de instruções vetoriais.

Este trabalho limita-se ao estudo da detecção e exploração de paralelismo de granularidade média em *laços mais internos*[3] de programas FORTRAN, dado que estes laços são a fonte primária de paralelismo em programas [3,4]. O estudo será restrito a máquinas MIMD de memória central. O paralelismo será explorado pela execução simultânea de iterações de um mesmo laço em processadores distintos, onde cada iteração executa seqüencialmente em um processador. Esta proposta é adotada por máquinas como o Alliant FX/8 [5].

1.2 Tarefas e Problemas de um Compilador Reestruturador

O principal objetivo de um compilador reestruturador, quando aplicado a laços mais internos, é detectar e gerar código paralelo que apresente o máximo de simultaneidade na execução das iterações, preservando sempre a semântica original do programa seqüencial. A preservação da semântica é garantida por meio da inserção de código especial de *sincronização*, que seqüencializa a execução paralela nos trechos onde há possível conflito entre as iterações, seja entre o uso e atribuição de variáveis, seja no fluxo de controle. Este código especial é mapeado em *hardware* específico, caracterizando a dependência compilador–máquina alvo. As operações que compõem este código especial são denominadas *primitivas de sincronização*.

Um dos pontos críticos das primitivas utilizadas pelos compiladores atuais é que elas ainda são fortemente baseadas na aplicação de sincronização em sistemas operacionais e, como tal, são baseadas em eventos dissociados dos conflitos especificamente gerados pelas dependências dos dados. Semáforos são exemplos clássicos de primitivas deste tipo, utilizados na implementação de regiões críticas como forma de garantir o uso exclusivo de recursos do sistema. Situam-se nesta classe as primitivas *test-and-set*, *lock/unlock* e, até certo ponto, *await/advance*. Exceção a esta regra é a primitiva *full/empty bit*, voltada para referências aos dados na memória.

[3]Inner loops

Nos compiladores atuais, a detecção dos pontos de inserção das primitivas é fortemente baseada na análise, durante a compilação, do fluxo de controle e do fluxo de dados entre iterações. Em alguns compiladores, a inserção correta das primitivas requer ainda o cálculo da *distância* entre as iterações a serem sincronizadas.

Esta tarefa de detecção e inserção das primitivas de sincronização pelo compilador é dificultada por alguns problemas, relacionados abaixo:

1. Como a análise de fluxo de dados é feita sobre nomes de variáveis, e não sobre endereços, não é possível prever todas as prováveis dependências de dados durante a compilação. É possível que, durante a execução, mais de um nome referencie o mesmo endereço. Este problema, conhecido como *aliasing*, surge da passagem de parâmetros em subrotinas e do uso de ponteiros. O estudo desta limitação é uma forte motivação deste trabalho, dado que o uso de bibliotecas de subrotinas é comum em FORTRAN.

2. É indecidível, na fase de compilação, o mapeamento correto entre os pontos de origem e destino das dependências de dados em laços que apresentem vetores com índices não lineares. Estes casos implicam em distâncias variáveis ou desconhecidas, na fase de compilação, das iterações conflitantes. Laços com este comportamento executam seqüencialmente nas propostas atuais, dado que estas lidam apenas com distâncias fixas e conhecidas na fase de compilação.

3. O reuso de variáveis por mais de uma iteração gera conflitos de escrita na execução paralela, caracterizados como dependências de saída e anti-dependências. Este reuso implica em excesso de sincronização e, conseqüentemente, seqüencialização. Este problema é resolvido em parte, durante a compilação, pela mudança de nomes de variáveis[4].

1.3 Objetivo do Trabalho

Esta dissertação apresenta um esquema inédito de paralelização de laços mais internos em FORTRAN, que elimina escritas conflitantes através da criação de um endereço individual para cada escrita no laço. O esquema atua, durante a execução, no reendereçamento[5] das leituras e escritas conflitantes, em contraste com a técnica de mudança de nomes, executada durante a compilação.

---

[4]renaming

[5]readdressing

Esta reestruturação elimina dependências de saída e anti-dependências, resultando em laços com atribuições únicas, nos quais se manifestam apenas dependências de fluxo entre as iterações. Estas dependências são posteriormente sincronizadas por uma primitiva apropriada, resultante de modificações na *full/empty bit*. Esta sincronização, diferentemente dos esquemas tradicionais, dispensa a detecção e cálculo de distâncias entre os pontos de sincronização.

Em certos casos, é possível determinar, durante a compilação, que haverá independência nos acessos à memória entre duas referências no laço. A detecção destes casos requer a inserção de especialização no compilador na forma de uma análise de dependências tradicional. Esta análise deve guiar o compilador na geração de código paralelo mais eficiente, dispensando computações adicionais de sincronização durante a execução quando a independência for detectada.

A análise é também estendida a laços contendo referências a parâmetros de subrotinas. Como os endereços dos dados representados por estes parâmetros não podem ser definidos durante a compilação, são gerados testes que verificam as dependências e classificam, durante a execução, o laço original.

As propostas são simuladas em um mini-ambiente de programação paralela [6], constituído por um interpretador de um subconjunto da linguagem Pascal com extensões de paralelismo, um simulador de arquitetura paralela configurável e um conjunto de ferramentas para análise de desempenho.

1.4 Divisão do Trabalho

O capítulo **2** introduz dependências de dados e compara primitivas de sincronização. O capítulo **3** apresenta o esquema de paralelização e sincronização de laços mais internos. O capítulo **4** descreve a especialização necessária ao compilador para a geração de código paralelo mais eficiente. O capítulo **5** finaliza o trabalho com as conclusões obtidas e aponta possíveis trabalhos futuros.

Os resultados de simulações das propostas e estudos comparativos são apresentados individualmente nos capítulos correspondentes (**2** e **3**).

Capítulo 2

Primitivas de Sincronização

Este capítulo introduz o conceito de primitiva de sincronização e mostra sua utilidade na paralelização de programas seqüenciais. Algumas primitivas são apresentadas e comparadas. O capítulo culmina com a proposição de uma nova primitiva, utilizada em um esquema de sincronização de laços a ser descrito no próximo capítulo.

2.1 Dependências de Dados

A semântica de um programa, em uma máquina seqüencial tradicional, requer a execução dos comandos segundo o fluxo de controle. Estes comandos estão inter-relacionados pelo uso comum de dados armazenados em uma memória central. O fluxo dos dados entre estes comandos, via acessos à memória, determina uma relação de *dependência de dados*.

A análise de dependência de dados, nesta dissertação, será restrita a *blocos básicos*. Por definição [8], um bloco básico é uma seqüência de comandos consecutivos no qual o fluxo de controle passa do primeiro ao último comando sem possibilidade de parada de execução[1] ou desvios[2].

Portanto, entre dois comandos C_1 e C_2, localizados em um bloco básico e executados nesta seqüência, há três tipos possíveis de relações de dependência de dados [7]:

- **Dependência de fluxo:** ocorre quando um dado é computado (armazenado na memória) pelo comando C_1 e posteriormente lido pelo comando C_2. A notação equivalente é $C_1 \; \delta \; C_2$.

[1] halt

[2] branching

- **Anti-dependência:** ocorre quando um dado é lido pelo comando C_1 antes de ser novamente computado em C_2. A notação é $C_1\ \delta^-\ C_2$.

- **Dependência de saída:** surge da computação consecutiva de um mesmo dado por C_1 e C_2. A notação é $C_1\ \delta^\circ\ C_2$.

Estas relações impõem uma ordem de execução entre os comandos dependentes, determinando conseqüentemente uma ordem nos acessos à memória para leitura e alteração dos dados. A alteração desta ordem de acesso implica na modificação da semântica do programa.

Como exemplo, suponha o trecho de código abaixo:

$$
\begin{aligned}
C_1 &: a \leftarrow b + c \\
C_2 &: b \leftarrow a + 1 \\
C_3 &: a \leftarrow d + e \\
C_4 &: f \leftarrow b \times 3
\end{aligned}
$$

Este exemplo apresenta as seguintes relações de dependência:

$$
\begin{array}{ll}
C_1\ \delta\ C_2\ (\text{variável } a), & C_2\ \delta\ C_4\ (\text{variável } b) \\
C_1\ \delta^-\ C_2\ (\text{variável } b), & C_2\ \delta^-\ C_3\ (\text{variável } a) \\
C_1\ \delta^\circ\ C_3\ (\text{variável } a) &
\end{array}
$$

Se neste bloco a ordem de execução dos comandos C_2 e C_3 fosse invertida, a leitura da variável de nome *"a"* em C_2 retornaria o valor produzido em C_3, e não o valor produzido em C_1. A inversão destes comandos viola a anti-dependência entre C_2 e C_3 e altera a semântica do programa. Note que é permitida a inversão dos comandos C_3 e C_4 por não haver dependência entre eles.

Um compilador otimizador utiliza as relações de dependência para efetuar uma *análise do fluxo de dados*[8] de um programa. Esta análise, tradicionalmente feita sobre os nomes de variáveis, é a base para otimizações no uso de variáveis e alocação de registradores, como a detecção de variáveis vivas[3] e determinação do alcance de definições[4].

A transformação de um programa seqüencial em um programa paralelo é, a rigor, uma otimização feita por um compilador especializado, que utiliza as relações de dependência para determinar quais comandos são *independentes* no uso dos dados. Estes comandos, por não possuírem acessos comuns a dados na memória, podem ter sua ordem de execução alterada.

[3] live-variable analysis

[4] reaching definitions

Esta alteração, que se traduzia em inversão dos comandos nas máquinas seqüenciais, apresenta-se aos compiladores paralelizadores como uma forma de aumentar o desempenho do programa mediante a execução *simultânea* (paralela) desses comandos em processadores distintos.

2.2 Sincronização

A detecção de dependências indica ao compilador uma ordem de execução dos comandos no programa seqüencial a ser mantida no programa paralelo. Esta ordem implica na *seqüencialização* da execução dos comandos envolvidos em uma dependência. A *sincronização* é uma técnica que seqüencializa a execução dos comandos dependentes, mantendo o paralelismo no resto do programa. Através da seqüencialização dos trechos com dependências, a sincronização mantém a ordem original dos acessos à memória.

Para exemplificar a necessidade do sincronismo durante a execução paralela de trechos com dependências de dados, suponha o laço seqüencial abaixo:

do $i = 2, N$
    $A_i = A_{i-1} + 1$
enddo

Dados dois processadores, *P1* e *P2*, a paralelização deste laço poderia ser efetuada através da distribuição das iterações *pares* para o primeiro processador e das *ímpares* para o segundo, como no esquema abaixo, admitindo-se que A é um vetor de acesso comum:

P1	*P2*
do $i = 2, N, 2$	**do** $i = 3, N, 2$
$A_i = A_{i-1} + 1$	$A_i = A_{i-1} + 1$
enddo	**enddo**

Sendo *reg* um registrador arbitrário local a cada processador, a execução das iterações *2* e *3* pode ser traduzida na seguinte versão *assembly*:

P1	*P2*
c_1 : LOAD reg, A_1	c_4 : LOAD reg, A_2
c_2 : ADD reg, 1	c_5 : ADD reg, 1
c_3 : STORE A_2, reg	c_6 : STORE A_3, reg

Supondo que o valor inicial dos elementos A_1 e A_2 fosse zero, seria razoável esperar, após a execução das iterações *2* e *3*, que o valor final de A_3 fosse *2*. Isto pode não acontecer na execução paralela, pois há um ***entrelaçamento não determinístico*** na execução das instruções ***assembly*** devido a independência existente na execução dos comandos entre os processadores em uma máquina MIMD. Desta forma, poderíamos ter as seguintes seqüências de execução, dentre outras:

- $c_1\ c_2\ c_3\ c_4\ c_5\ c_6 \Rightarrow A_3 = 2$ (*correta*)
- $c_1\ c_4\ c_2\ c_5\ c_3\ c_6 \Rightarrow A_3 = 1$ (*incorreta*)

Este comportamento anômalo, que resulta em $A_3 = 1$, ocorre devido a violação da dependência de fluxo existente entre o armazenamento de A_i na iteração i (c_3) e sua leitura na iteração $i + 1$ (c_4). Esta alteração na ordem de acessos à memória é conhecida como ***condição de corrida***[5], na qual os processadores disputam a prioridade no acesso aos dados compartilhados.

A sincronização surge como forma de controlar a execução dos comandos segundo as restrições impostas pelas dependências de dados. O trecho seqüencial determinado pela sincronização é definido como uma *região crítica*, onde a execução deve ser mutuamente exclusiva entre os processadores. A exclusão mútua garante que a seqüência de comandos internos à região crítica seja tratada como uma operação indivisível.

O exemplo da página anterior ilustra um caso onde a sincronização induz uma região crítica do tamanho do corpo do laço, forçando a seqüencialização de sua execução. Isto ocorre porque a execução do LOAD em *P2* deve aguardar a execução do STORE em *P1*. Um objetivo central dos compiladores que utilizam sincronização para explorar paralelismo em programas é diminuir ao máximo as regiões críticas, aumentando a fração de paralelismo explorável no programa.

2.3 Primitivas de Sincronização

Pode-se enumerar três passos de destaque, em um compilador reestruturador, na paralelização de programas seqüenciais:

1. **Criação do grafo de dependências** do programa, indicando as dependências existentes entre os comandos.

2. **Detecção dos trechos seriais**, ou seja, aqueles que possuem dependências de dados e que exigem sincronização para a execução paralela correta.

[5]Race condition

3. **Inserção de código de sincronização** para garantir a execução exclusiva de trechos seriais.

Este código especial para sincronização usualmente utiliza *hardware* específico, caracterizando uma dependência compilador–máquina. As operações que compõem este código especial denominam-se *primitivas de sincronização.*

Como exemplo clássico de primitivas, podemos citar as operações sobre *semáforos*[9]. Um semáforo é uma variável inteira não negativa na qual atuam duas operações básicas: **P** e **V**[6].

Dado um semáforo s, a semântica das primitivas é:

P(s): label: **if** $s > 0$ **then**
$s \leftarrow s - 1$
else
goto label

V(s): $s \leftarrow s + 1$

Sob certas condições, que serão enumeradas a seguir, os semáforos podem ser utilizados por um compilador para garantir o acesso exclusivo a regiões críticas, mediante a inserção de uma primitiva **P** na entrada da região crítica e de uma primitiva **V** ao final.

O trecho abaixo ilustra uma região crítica em um programa paralelo. A execução do trecho paralelo é feita simultaneamente por todos os processadores participantes da computação. Na região crítica a execução deve ser seqüencializada, isto é, apenas um processador *por vez* é habilitado a executar o trecho seqüencial.

trecho paralelo
...
P(s)
região crítica (trecho seqüencial)
V(s)
...
trecho paralelo

Supondo inicialmente que $s = 1$, o primeiro processador a executar a operação **P** torna-se apto a entrar na região crítica e impossibilita que outros processadores adentrem a região crítica através do decremento no semáforo ($s \leftarrow s - 1$). Os processadores desabilitados aguardam na operação **P**, mediante um teste contínuo no semáforo, até

[6]Estas primitivas são também denominadas WAIT e SIGNAL por outros autores

que o processador na região crítica execute a liberação do semáforo pela operação **V**. A execução correta deste trecho crítico por processadores paralelos é garantida por duas premissas básicas:

1. O semáforo s deve ser acessível a todos os processadores envolvidos na computação. A implementação do semáforo pode ser feita tanto em *hardware* especial como em uma posição de memória global.

2. As primitivas devem ser executadas *atomicamente*, ou seja, sem interrupção. Se as instruções de teste e decremento da primitiva **P** fossem feitas separadamente, outro processador poderia alterar o valor do semáforo entre o teste e o decremento, invalidando a operação. O exemplo abaixo ilustra esta hipótese, supondo-se que o valor inicial do semáforo seja *1*:

	P1		*P2*
c_1 :	**if** $s > 0$ **then**	c_3 :	**if** $s > 0$ **then**
c_2 :	$s \leftarrow s - 1$	c_4 :	$s \leftarrow s - 1$
	else ...		**else** ...

   Se a ordem de execução das instruções obedecesse à seqüência

$$c_1 \;\; c_3 \;\; c_2 \;\; c_4$$

   os dois processadores utilizariam o valor inicial de s (*1*) e estariam habilitados a entrar na região crítica, violando a exclusão mútua. Desta forma, as primitivas devem ser executadas de forma atômica, ou seja, *indivisível*. Em particular, na primitiva **P**, a seqüência teste e decremento deve ser executada indivisivelmente. Esta seqüência é usualmente implementada por uma instrução em *hardware*. A indivisibilidade na execução desta instrução, na leitura e alteração do valor do semáforo, é garantida pelo *hardware* de acesso à memória.

Um fator que afeta significativamente o custo computacional da operação de entrada na região crítica é a forma de implementação da espera pela liberação do semáforo. Há três formas básicas de implementação:

1. Através de um *teste contínuo*[7] sobre o semáforo. Este procedimento, exemplificado na semântica da operação **P** na página anterior, faz com que o processador permaneça testando o semáforo até conseguir a liberação. Esta técnica tem como

[7] spinning

desvantagem o excesso de requisições à memória no acesso ao semáforo, além de manter o processador ocupado[8] executando este teste. Seu grande atrativo é o baixo custo computacional, da ordem de poucos ciclos de máquina (acessos à memória).

2. Através de um *teste contínuo nas caches*. O uso de *caches* reduz significativamente os conflitos entre requisições paralelas no acesso à memória, permitindo ao processador executar o teste sobre uma cópia do semáforo, localmente em sua *cache* privada. Entretanto, todas as *caches* devem ser invalidadas quando da liberação do semáforo, implicando em excesso de requisições à memória para a obtenção do novo valor do semáforo.

3. Através da *suspensão da execução* dos processadores que não foram liberados pelo semáforo. Esta forma de espera ocorre freqüentemente em implementações das primitivas WAIT e SIGNAL. A principal desvantagem desta técnica é que a suspensão e liberação de processos acarretam elevado *custo computacional*, por exigir chamadas a funções do sistema operacional. Este alto custo impõe restrições ao suporte de sincronização pelo sistema operacional para exploração de paralelismo de granularidade média (laços) e baixa (operações), mostrando-se mais adequado à sincronização de blocos de granularidade alta (procedimentos).

2.4 Propostas

Este tópico apresenta um conjunto pequeno, porém representativo, de primitivas de sincronização implementadas em máquinas paralelas MIMD nos últimos anos. Todas as primitivas obedecem às duas premissas básicas apresentadas anteriormente: da atomicidade de execução e do compartilhamento do dado ou semáforo no qual a primitiva atua.

A apresentação das propostas é precedida de uma descrição dos paradigmas de utilização das primitivas quanto ao tipo de operação de sincronização efetuado sobre os dados.

2.4.1 Paradigmas de Utilização das Primitivas

O conjunto **P** e **V**, descrito no tópico anterior, é um exemplo de paradigma de primitiva que exige do compilador a inserção de um *controle sobre semáforos* para a sincronização

[8] busy-waiting

correta no acesso aos dados. Podemos destacar três paradigmas principais na utilização das primitivas pelos compiladores:

- Primitivas baseadas em **controle de semáforos**: o uso de primitivas desta classe é baseado na identificação de regiões críticas e na inserção de protocolos (comandos) de entrada e saída da região crítica. Este protocolos fazem uso de *semáforos* para o controle da execução de regiões críticas.

  Este paradigma de sincronismo é utilizado em praticamente todos os computadores MIMD com suporte de sincronização, tais como o Alliant FX/8 [5] e Sequent [10], e envolve as primitivas *await/advance*, *test-and-set*, dentre outras.

- Primitivas baseadas no **controle de acesso aos dados**: propostas existentes baseadas neste paradigma associam uma chave a cada variável na memória, utilizada no controle do acesso à variável pelos processadores. Este controle é feito de forma indivisível na chave particular de cada variável, dispensando o uso de semáforos para sincronizar acessos à memória. A tarefa do compilador, na inserção de primitivas deste paradigma, é localizar as variáveis que possuem dependências de dados e determinar os padrões de controle de acesso, segundo as relações de dependências.

  Um exemplo de primitiva nesta classe é o *full/empty bit*, implementado no HEP (*Heterogeneous Element Processor*) [11,12], que associa a cada posição de memória um bit de controle de acesso.

- Primitivas baseadas em **operações nos dados**: as primitivas desta classe têm como característica principal o fato de implementarem em *hardware* uma instrução que executa, de forma indivisível, uma seqüência *leitura — operação aritmética (ou booleana) — escrita* sobre uma variável compartilhada. A atomicidade desta seqüência de operações caracteriza um *encapsulamento* de uma região crítica em *hardware*. Este paradigma, como o anterior, dispensa o uso de semáforos por atuar diretamente na variável da computação original.

  Fetch-and-Add (FAA) [13,18] é um exemplo clássico neste paradigma, proposto como primitiva básica no NYU Ultracomputer [13]. Uma FAA executa uma soma em uma variável inteira de forma atômica, retornando seu valor anterior a soma. Outra maqúina que utiliza uma primitiva semelhante ao *FAA*, porém com uma gama maior de operações, é o RP3 [14] (IBM). O RP3 fornece uma primitiva *Fetch-and-ϕ*, onde ϕ é uma operação do tipo soma, mínimo, máximo, e–lógico, dentre outras.

Há também propostas híbridas, como no Cedar [15]. Nesta máquina, cada instrução de sincronização faz uma operação nos dados mediante um teste booleano de controle de acesso. Este multiprocessador, desenvolvido na Universidade de Illinois, adiciona uma chave (número inteiro) a cada posição de memória, além de um processador de sincronismo dedicado em cada módulo de memória. Este processador possui um repertório de cerca de 60 instruções [16] que executam, de forma indivisível, um teste de acesso e uma operação aritmética ou booleana sobre os dados.

2.4.2 *Test-and-set*

A primitiva *test-and-set* (*TAS*) tem a seguinte semântica:

TAS (*semáforo*)
{
  temp ← *semáforo*;
  semáforo ← 1;
  return *temp*
}

O *TAS* retorna o valor antigo do semáforo, atribuindo-lhe em seguida o valor *1*. A primitiva é usada na implementação de protocolos de entrada de regiões críticas. Supondo-se que o valor inicial de um semáforo s seja *0*, e admitindo-se que *0* seja o valor que libera a entrada na região crítica, é possível implementar um protocolo de entrada com a operação *TAS* da seguinte forma:

c_1 : **while** $TAS(s) = 1$ **do**;
  região crítica
c_2 : $s \leftarrow 0$

O primeiro processador a executar o *TAS* encontra o semáforo "aberto" (igual a zero) e fica habilitado a entrar na região crítica. Como a operação de leitura e atribuição é indivisível, o semáforo passa a ter o valor *1*, impedindo qualquer permissão de entrada posterior, até que a operação de liberação ($s \leftarrow 0$) seja executada.

Como foi visto no tópico *2.3*, a indivisibilidade da execução de uma primitiva **P** é normalmente garantida pela implementação da seqüência *teste–decremento* em uma instrução em *hardware*. Em particular, esta primitiva poderia ser implementada através de uma primitiva de nível inferior, como a instrução *assembly TAS*. O comando c_1 do exemplo acima ilustra esta possibilidade. Através do teste contínuo no resultado da

operação *TAS*, este comando implementa uma operação **P**. Em última instância, uma operação no nível de **P** pode ser utilizada na implementação de um novo comando de alto nível em uma linguagem de programação. Estes diversos níveis de abstração caracterizam uma *hierarquia de primitivas*.

O multiprocessador Sequent [10] é um exemplo de máquina paralela que fornece um conjunto de semáforos de 1 bit, implementados em *hardware* específico, e utiliza a primitiva *TAS* para implementar o protocolo de entrada (*lock*) de regiões críticas. A memória global possui uma cópia dos semáforos e a espera pela liberação é feita através de testes contínuos desta variável compartilhada nas *caches*. A liberação do semáforo é feita tanto na memória global quanto no *hardware* específico, garantindo consistência.

2.4.3 *Await/Advance*

As primitivas *await/advance* foram propostas como forma básica de sincronização no mini–supercomputador Alliant FX/8 [5]. Sua finalidade é sincronizar dependência de dados entre as iterações em laços paralelizados automaticamente pelo compilador FORTRAN do Alliant [17].

A rigor, *await/advance* fazem parte de um grupo de primitivas baseadas no controle de semáforos, porém de uma forma diferente do *TAS* no que se refere à ordem de entrada dos processadores na região crítica. Enquanto a primitiva *TAS* controla regiões críticas *não ordenadas*, no sentido de que a ordem de entrada dos processadores na região crítica é aleatória, as primitivas *await/advance* controlam regiões críticas *ordenadas*, pois há uma ordem de entrada dos processadores, usualmente definida por um teste efetuado sobre a *iteração* em execução no processador.

A semântica das primitivas é definida por:

AWAIT (cs_n, d)
{
  while $cs_n \leq i - d$ **do**;
}

ADVANCE (cs_n)
{
  while $cs_n \neq i$ **do**;
  $cs_n \leftarrow cs_n + 1$
}

onde:

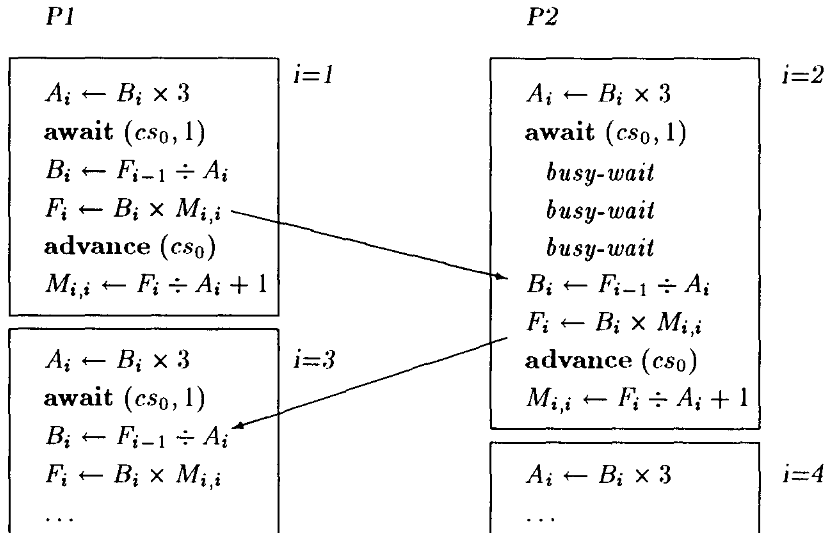
- cs_n é um semáforo compartilhado;
- i é a iteração corrente no processador que invoca *await* ou *advance*;
- d é a distância entre as iterações dependentes.

O compilador FORTRAN do Alliant detecta as dependências de dados entre as iterações e gera código vetorial e/ou paralelo com sincronização para explorar o paralelismo existente no laço. As primitivas do Alliant contam com suporte de *hardware* para sincronização, na forma de oito semáforos compartilhados ($cs_0 \rightarrow cs_7$) e registradores especiais para controlar a execução paralela do laço, como registradores locais aos processadores contendo a iteração em execução (que determinam o valor de i) e registradores de acesso comum contendo a última iteração executada, dentre outros.

A inserção das primitivas de sincronização é feita, a grosso modo, pelo seguinte algoritmo:

1. Detecta-se a *fonte da dependência*, isto é, o comando e a iteração que produzem um determinado dado.
2. Detecta-se o *destino da dependência*, isto é, o comando e a iteração que utilizam esse mesmo dado.
3. Determina-se a *distância* entre as iterações dependentes, para o teste de controle da entrada da região crítica. A distância é definida como a diferença entre a iteração que utiliza o dado e a iteração que produz este dado.
4. Seleciona-se um dos oito semáforos compartilhados e insere-se um *await* antes do destino da dependência e um *advance* após a fonte.

O exemplo a seguir ilustra a execução de um trecho de um laço que foi reestruturado para paralelismo e requer sincronização. Este laço é executado em uma máquina com dois processadores, onde o valor inicial de cs_0 é 1:



No exemplo acima, há uma dependência entre o dado produzido na atribuição a F_i e a leitura deste dado na iteração seguinte, pela leitura de F_{i-1}. Estas dependências de fluxo, indicadas pelas flechas, são sincronizadas pela primitiva *await* através do bloqueio da execução de um processador até o momento em que o processador anterior execute um *advance*, liberando a execução. A liberação ocorre de uma iteração *i* para a iteração *i+1*, caracterizando a ordem de entrada na região crítica. Observe que, apesar do trecho seqüencial, há paralelismo na execução do laço.

2.4.4 *Full/empty bit*

O HEP [11,12] foi uma proposta inovadora na exploração de paralelismo em programas. Foi manufaturado de 1982 a 1985 pela empresa americana Denelcor e pode ser caracterizado como um computador MIMD com memória compartilhada.

A sincronização no HEP é implementada através da adição de um *bit de estado*, conhecido como *full/empty bit*, a cada endereço de memória e a cada registrador compartilhado. Estes bits controlam o acesso aos dados em operações especiais de leitura e escrita. Dado que *empty* e *full* representam os possíveis estados destes *bits*, e que *reg* é um registrador local isento de *bit* de estado, a semântica das novas operações de leitura e escrita à memória é definida por:

```
$LOAD (variável, reg)
{
    while variável.bit = empty do;
    reg ← variável
    variável.bit ← empty
}

$STORE (reg, variável)
{
    while variável.bit = full do;
    variável ← reg
    variável.bit ← full
}
```

A operação de leitura ($LOAD) deve esperar até que o estado da variável seja *full*, quando então a leitura é executada, retornando ao estado *empty*. As operações $LOAD e $STORE, envolvendo o teste, a operação e a mudança de estado, são feitas de forma atômica.

Como exemplo de controle de acesso a uma variável, suponha uma computação onde vários processadores tentam atualizar um dado global (variável *soma*) com um dado local (variável *soma_local*). O trecho de código que executa esta operação é:

$$soma \leftarrow soma + soma_local$$

e corresponde à seguinte seqüência de comandos, onde reg_1 e reg_2 são registradores locais aos processadores:

```
$LOAD (soma, reg1)
LOAD (soma_local, reg2)
reg1 ← reg1 + reg2
$STORE (reg1, soma)
```

Supondo que o estado inicial da variável *soma* seja *full*, na execução paralela deste trecho somente um processador consegue executar o $LOAD, fazendo com que o estado da variável passe a *empty*. Os outros processadores esperam em $LOAD, pela liberação do acesso à variável, que acontece quando $STORE é executado e o estado da variável passa a *full*. Desta forma o HEP garante acesso exclusivo à variáveis compartilhadas.

A primitiva *full/empty bit* permite maior flexibilidade na sincronização, quando comparada com o controle de semáforos. Isto ocorre porque a sincronização opera sobre o

dado em si, e não sobre os comandos que utilizam o dado. Como exemplo desta flexibilidade, se a ordem de execução fosse alterada para

$$soma \leftarrow soma_local + soma$$

obter-se-ia, sem nenhuma otimização do compilador, uma região crítica 25% menor, pela execução do LOAD de *soma_local* antes da região crítica. Este *rearranjo de operações* melhora o desempenho do programa, por aumentar a fração do trecho paralelo, permitindo a exploração de paralelismo de granularidade mais fina.

O fato de controlar o acesso a um dado específico torna mais simples para o compilador, em certos casos, a tarefa de inserir primitivas de sincronismo. Em dependências de fluxo, bastaria localizar a variável que origina o conflito e iniciar seu estado com *empty*. As operações $LOAD e $STORE se encarregam, durante a execução do programa, de garantir automaticamente o acesso correto. Além disto, fica desnecessário o cálculo da *distância* envolvendo iterações conflitantes. Esta facilidade, contudo, ocorre apenas em casos especiais que serão descritos posteriormente.

A sincronização pelo *full/empty bit* exige atenção especial quando há duas referências iguais ao mesmo dado, como dois $LOAD consecutivos, por exemplo. O uso incorreto das primitivas pode levar o programa a um estado conhecido como *starvation*, onde o processador fica aguardando, durante toda a computação, uma mudança de estado da variável que não ocorrerá. Como exemplo, a computação de $a \leftarrow b + c \times b$ bloquearia a execução do programa se o compilador gerasse dois $LOAD consecutivos à variável b. Neste exemplo, em particular, um compilador mais "inteligente" poderia gerar um único acesso à memória para as duas referências a b. Contudo, este não é o caso geral, como será exemplificado mais a frente (página 26).

Para superar este problema, o HEP fornece operações indivisíveis nos bits de estado, como PURGE (faz $bit \leftarrow empty$) e FILL ($bit \leftarrow full$), que devem ser inseridas no código normal para garantir o acesso correto às variáveis.

2.4.5 *Fetch-and-add*

Fetch-and-add (*FAA*) é a primitiva básica de sincronização das máquinas NYU Ultracomputer[13] e IBM RP3[14]. Esta primitiva, como já foi visto anteriormente, encapsula em *hardware* uma região crítica que executa uma soma sobre uma variável compartilhada.

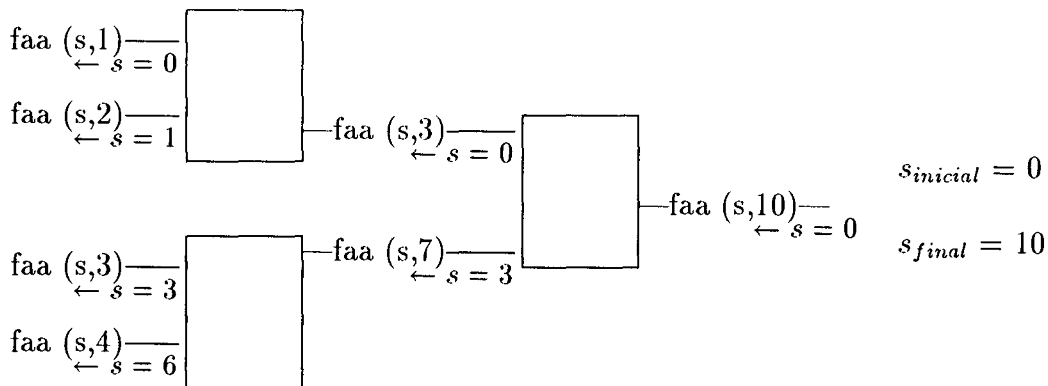
Sua semântica consiste em:

```
FAA (variável, valor)
{
    temp ← variável;
    variável ← variável + valor;
    return temp
}
```

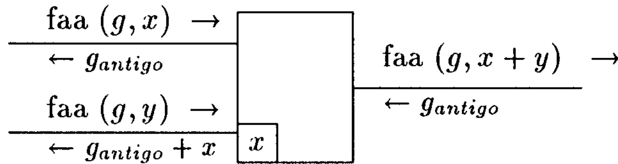
A grande motivação para o uso da primitiva é a possibilidade de agrupar várias requisições *FAA* simultâneas a uma mesma variável de memória em uma requisição única. Esta *combinação* de requisições permite reduzir o custo no acesso a variáveis compartilhadas, em comparação com a seqüencialização imposta nas propostas anteriores de primitivas.

A implementação eficiente do *FAA* exige, além da execução atômica da leitura e incremento, uma rede de comunicação entre os processadores e a memória que permita a combinação de operações *FAA*. Esta rede deve simular uma estrutura em árvore, onde as operações sejam combinadas de forma binária, produzindo uma requisição *FAA* final. O custo de **p** acessos à memória, quando da combinação dos acessos, é da ordem de $O(\log \mathbf{p})$, onde **p** representa o número de processadores. Este comportamento logarítmico surge da estrutura em árvore da rede combinacional.

Como exemplo, suponha a combinação de quatro requisições paralelas a uma variável compartilhada s:



A figura acima ilustra a propagação das *FAA* concorrentes e os valores de s que retornam, ao passarem pelos nós da rede. Cada retângulo representa um nó da rede. Este nós possuem *hardware* especial que executa a soma das requisições *FAA*, além de uma memória para armazenar os valores intermediários. A operação de um nó pode ser representada genericamente [18] por:



As setas "→" indicam o fluxo das requisições à memória, enquanto que as setas "←" indicam retorno dos dados da memória. A variável x dentro do nó caracteriza a memória interna ao nó.

Este exemplo ilustra a sincronização de requisições *FAA* paralelas à variável s através da combinação destes acessos à memória em um único acesso. A rigor, este acesso final representa uma região crítica. No entanto, ao invés de seqüencializar os acessos a região crítica, como no *test-and-set*, a primitiva *FAA* combina estes acessos, implicando em menor custo de sincronização.

A utilização da primitiva *FAA* torna-se mais atraente quando o número de processadores é alto. Um acesso seqüencial a uma variável compartilhada, por 512 processadores (como no RP3), levaria 512 unidades de tempo com uma sincronização *seqüencial*, ao invés de apenas 9 unidades de tempo[9] com a *FAA*.

2.5 Razões para uma Nova Primitiva

O objetivo deste tópico é avaliar as primitivas apresentadas anteriormente quanto a eficácia na sincronização de um programa com dependência de dados entre as iterações de um laço. Esta avaliação objetiva apresentar razões para a utilização de uma nova primitiva.

Por simplicidade de apresentação e para ordenar o raciocínio, serão avaliados programas com *atribuições únicas*[10]. Em laços paralelos com atribuições únicas, um dado é produzido de forma única por uma iteração específica e seu valor é lido em iterações subseqüentes, caracterizando relações de dependência de fluxo. Como há apenas uma escrita a cada endereço de memória, *não há dependências de saída e anti-dependências.*

[9] A rigor deve-se computar o tempo necessário para a volta dos dados na rede. Neste caso o custo de *FAA* combinados é de $2 \times \log \text{nproc}$.

[10] single-assignment

2.5.1 Sincronização por Semáforos

A sincronização da execução paralela de laços com atribuições únicas, na presença de dependências de fluxo, implica em uma ordem de execução das iterações. Esta ordem deve ser garantida nas propostas baseadas em semáforos (*await* e *test-and-set*) por meio da inserção de uma região crítica ordenada, seqüencializando a execução do trecho do laço entre o comando que produz um dado e o que utiliza este dado, como no exemplo da página 18.

A dificuldade na sincronização de tais laços com a primitiva *TAS* reside no fato de que a primitiva é voltada para a implementação de regiões críticas onde a ordem de entrada é aleatória (*regiões críticas não ordenadas*). Isto ocorre porque um semáforo manipulado pelo *TAS* assume apenas um valor booleano (*0* ou *1*), não permitindo um controle adicional que determine uma ordem específica de acesso à região crítica. A implementação de uma ordem de entrada exige a utilização de um *vetor de semáforos*, como no exemplo abaixo:

do $i = 1, N$
    ...
    while $TAS(s_{i-1}) = 1$ **do**;
    região crítica ordenada
    $s_i \leftarrow 0$;
    ...
enddo

Supondo-se que os elementos do vetor s sejam inicialmente iguais a *1* e que $s_0 = 0$, a execução de uma iteração fica bloqueada até que a iteração anterior libere a região crítica, mediante a atribuição a s_i. A execução correta das iterações é garantida porque a ordem de entrada na região crítica está vinculada ao índice do semáforo. No exemplo acima, a liberação da região crítica se faz da iteração *i* para a iteração *i + 1*.

Tal sincronização, além de exigir um número elevado de semáforos, impõe ao compilador um alto grau de especialização para a detecção dos pontos de origem e destino das dependências. O ponto crítico deste esquema é o *mapeamento das distâncias* entre as iterações dependentes e os índices dos semáforos que controlarão o acesso à região crítica.

Este esforço, contudo, pode por muitas vezes resultar em uma sincronização ineficiente, como mostra o exemplo a seguir:

```
do i = 1,8
    while TAS(s_2i) = 1 do;
    ... ← a_2i
    a_i+10 ← ...
    s_i+10 ← 0
    ...
enddo
```

O laço acima, já com a sincronização incluída, efetua acessos aos seguintes elementos do vetor a, de acordo com o número da iteração:

i	1	2	3	4	5	6	7	8
a_{2i}	2	4	6	8	10	12	14	16
a_{i+10}	11	12	13	14	15	16	17	18

Este laço apresenta uma relação de dependência de fluxo na qual a distância é *variável*, dificultando o trabalho do compilador. As dependências de fluxo têm origem nas escritas das iterações *2, 4* e *6*, e destino nas leituras das iterações *6, 7* e *8*. Outro agravante é o fato de que, apesar das iterações de *1* a *5* não possuírem relações de dependência de fluxo nas leituras a a_{2i}, há o custo da manutenção da região crítica e a dificuldade adicional de fornecer valores iniciais corretos aos semáforos. No exemplo, os semáforos s_i, com $2 \leq i \leq 10$ devem ser inicializados com *0* (região crítica liberada) e os restantes com *1*.

Este procedimento deve ser repetido para cada relação de dependência existente no laço. A otimização do uso de várias regiões críticas em um laço exige um grau maior de "inteligência" do compilador, pela necessidade de alterar a ordem dos comandos e eliminar sincronizações desnecessárias [7,19].

O Alliant foi a primeira proposta comercial a procurar *adaptar* o uso de semáforos à sincronização de laços que exigem *regiões críticas ordenadas* para garantir a dependência de fluxo entre as iterações. Isto ocorre porque um semáforo no Alliant está associado a um número de iteração e não a um valor booleano, como no *TAS*. O teste incluso nas primitivas *await/advance* compara este semáforo compartilhado com o número da iteração em execução no processador (vide a semântica das primitivas na página 16), permitindo um controle na ordem de entrada na região crítica.

Apesar do *hardware* especial para sincronização e da semântica mais apropriada das primitivas disponíveis, o compilador FORTRAN do Alliant ainda encontra muitas

limitações na inserção eficiente das primitivas. Estas dificuldades são similares às encontradas no *TAS* e surgem basicamente das limitações impostas pelo uso de semáforos na sincronização do fluxo dos dados.

2.5.2 Sincronização por Operações nos Dados

A primitiva *FAA* perde sua utilidade em programas com atribuições únicas, já que sua aplicação direta é na combinação de atualizações do tipo soma a uma variável compartilhada. A utilização do *FAA* como forma de garantir as dependências de fluxo de dados restringe-se, portanto, à implementação de protocolos de entrada e saída em regiões críticas, controlando o acesso ao semáforo.

Contudo, a primitiva perde em parte a vantagem da combinação, no controle de semáforos, já que a região crítica deve ser executada seqüencialmente. Um possível ganho, no uso da *FAA*, surge da diminuição do número de requisições à memória, implicando em uma liberação mais rápida da região crítica, porém este ganho depende de uma série de fatores, como o número de processadores e o tempo de acesso à memória pela rede combinacional, que não serão tratados neste trabalho.

A *FAA* mostra-se muito útil na sincronização exigida por operações de escalonamento[11] de processadores na execução de laços paralelos, em particular nas políticas de auto-escalonamento[12]. Esta operação propicia a utilização do *FAA* por constar de uma leitura e soma a uma variável global (índice compartilhado) de tipo inteiro.

2.5.3 Sincronização por Controle de Acesso aos Dados

O uso da primitiva *full/empty*, na sincronização de programas com atribuições únicas, será avaliado sob dois enfoques:

1. Programas com leituras únicas: nesta classe de programas, cada dado recebe acessos de um par único de referências, na forma STORE/LOAD. A inversão desta ordem de acesso implica em um erro semântico.

Supondo que cada posição de memória contém um bit adicional de estado e que no início do programa estes bits estão no estado *empty*, é possível afirmar que, se utilizarmos as primitivas $STORE e $LOAD, o programa estará corretamente sincronizado. Isto ocorre porque, se um dado está no estado *empty*, é impossível executarmos um $LOAD antes

[11] scheduling

[12] self-scheduling

de um $STORE. Como para cada dado nesta classe de programas o padrão de acesso corresponde a um único par $STORE/$LOAD, a sincronização está garantida.

O efeito deste esquema em um compilador reestruturador fica evidente. O trabalho de detecção e inserção das primitivas torna-se *nulo*, visto que a própria primitiva controla o acesso ao dado.

Programas nesta classe, contudo, não são realísticos. É comum em programas o reuso de um mesmo dado por meio de múltiplas leituras. Este reuso elimina a necessidade de recomputação do dado, diminuindo o tempo de execução do programa.

2. Programas com múltiplas leituras: o padrão de acesso a dados, em programas desta classe, obedece à seqüência STORE/LOAD/LOAD/..., com um número indeterminado de leituras. A sincronização, pelo uso da primitiva *full/empty bit*, não pode ser acoplada automaticamente como no caso anterior. A restrição surge da semântica da operação de $LOAD, que retorna o estado da variável para *empty* após a leitura. A segunda leitura ficará em *starvation*, aguardando que uma nova escrita passe o estado da variável para *full*. Isto nunca ocorrerá, pois o programa só possui uma escrita para cada variável.

Faz-se necessária a participação do compilador, através da inserção de operações especiais do tipo FILL, cuja função é colocar o estado da variável na condição *full*. Estas operações devem ser inseridas antes de cada $LOAD, exigindo um certo "grau de inteligência" do compilador na inserção dos $LOAD nos locais apropriados e aumentando conseqüentemente o custo computacional devido a inclusão de mais instruções.

Poder-se-ia argumentar que, a partir da segunda leitura, bastaria ao compilador gerar um LOAD normal, sem o teste de estado. Entretanto, nem sempre é possível determinar qual será o segundo LOAD, como mostra o seguinte exemplo, supondo que as iterações deste laço são executadas em processadores distintos:

$$
\begin{array}{l}
\textbf{do } i = 1, N \\
c_1 : \ a_i \leftarrow \ \ldots \\
c_2 : \ \ldots \leftarrow a_{i-1} + a_{i-2} \\
\textbf{enddo}
\end{array}
$$

O elemento a_2 possui os seguintes acessos:

- STORE na referência a_i, para $i = 2$
- LOAD na referência a_{i-1}, para $i = 3$
- LOAD na referência a_{i-2}, para $i = 4$

Devido à independência da execução dos processadores em uma máquina MIMD e a outros fatores, como conflitos no acesso à memória, a iteração *4* poderia executar as leituras do comando c_2 antes que a iteração *3* o fizesse, ou vice-versa. Esta possibilidade de alternação da ordem das leituras proíbe a geração do LOAD normal.

2.6 A Primitiva *Flowbit*

Devido à incapacidade do *full/empty bit* em garantir automaticamente as dependências de fluxo em programas com atribuições únicas e múltiplas leituras, propõe-se uma nova primitiva de sincronismo, denominada *flowbit*, baseada em uma simplificação do *full/empty bit*. Esta primitiva altera a semântica das instruções \$LOAD e \$STORE para:

\$LOAD (*variável*, *reg*)
{
    while *variável.bit* = *empty* **do**;
    reg ← *variável*
}

\$STORE (*reg*, *variável*)
{
    variável ← *reg*
    variável.bit ← *full*
}

A primitiva é apropriada para sincronização de programas com atribuições únicas, por dois motivos:

1. Foi eliminado o teste de estado antes do \$STORE, por ser desnecessário sincronizar duas escritas, já que o programa possui atribuições únicas.

2. A eliminação da passagem do estado para *empty*, após a leitura, permite que várias leituras ocorram sem a necessidade de preencher o estado com *full* entre elas. Como não há dependências entre duas leituras, por estas não modificarem o conteúdo da memória, a ordem de execução dos \$LOAD não precisa ser mantida.

O compilador, com o uso do *flowbit*, volta a controlar automaticamente a sincronização, no sentido de que o trabalho de detecção e inserção das primitivas é nulo. Esta condição, existente com o *full/empty* em programas com leituras únicas, surge agora em programas com múltiplas leituras e atribuições únicas.

2.7 Teste Comparativo

A opção da utilização de um paradigma de controle de acesso aos dados, como no *full/empty bit*, foi parcialmente justificada anteriormente quando comparada com a falta de flexibilidade e alto custo da sincronização baseada em semáforos, além da dificuldade imposta a um compilador reestruturador na detecção e inserção destas primitivas. Será apresentado em seguida um teste comparativo do desempenho das primitivas na sincronização de um laço com dependências de fluxo entre as iterações. Esta análise comparativa mostrar-se-á favorável à sincronização baseada nos controle de acesso aos dados, como a *flowbit*.

O experimento consiste na paralelização da resolução de sistemas de equações lineares, na forma $Ax = r$ onde x e r são vetores e A, a matriz de coeficientes, é tridiagonal, na forma:

$$A = \begin{bmatrix} dc_1 & ds_1 & & & \\ di_2 & dc_2 & ds_2 & & \\ & di_3 & dc_3 & ds_3 & \\ & & & \cdots & \\ & & & di_n & dc_n \end{bmatrix}$$

Ou seja, a matriz de coeficientes será representada por três vetores:

- ds, a diagonal superior da matriz;
- dc, a diagonal central;
- di, a diagonal inferior.

A resolução de tal sistema é feita pelo algoritmo da Eliminação de Gauss, nos passos a seguir:

passo 1 : *Decomposição* $A = LU$, onde L é triangular inferior e U é triangular superior. Esta decomposição transforma o problema de resolver um sistema tridiagonal no problema de resolver dois sistemas triangulares, pois $Ax = r$ é transformado em $LUx = r$, que é resolvido determinando-se y em $Ly = r$ e posteriormente x em $Ux = y$.

O código resultante é apresentado abaixo:

$$
\begin{array}{ll}
g_1 \leftarrow dc_1 & \\
y_1 \leftarrow r_1 & \\
\textbf{do } i = 2, N & \\
c1: & f_i \leftarrow -di_i \div g_{i-1} \\
c2: & g_i \leftarrow dc_i + f_i \times ds_{i-1} \\
c3: & y_i \leftarrow r_i + f_i \times y_{i-1} \\
\textbf{enddo} &
\end{array}
$$

Os comandos $c1$ e $c2$ efetuam a decomposição $A = LU$. O comando $c3$ efetua $y = L^{-1}r$. O vetor g representa a diagonal principal da matriz U.

passo 2 : resolução de $Ux = y$, através de uma *retrosubstituição* na matriz U, determinando o vetor de incógnitas (x). Este passo é efetuado a partir da última linha da matriz, onde temos uma equação com uma incógnita ($y_n \leftarrow g_n \times x_n$), permitindo a definição da incógnita x_n. A retrosubstituição implica em um laço com incremento negativo, efetuando o cálculo das incógnitas de $n-1$ até 1:.

$$
\begin{array}{l}
x_n \leftarrow y_n \div g_n \\
\textbf{do } i = N-1, 1, -1 \\
\quad x_i \leftarrow (y_i - ds_i \times x_{i+1}) \div g_i \\
\textbf{enddo}
\end{array}
$$

O ganho decorrente da paralelização deste sistema é normalmente muito baixo, devido a característica inerentemente seqüencial imposta pela dependência de fluxo entre iterações sucessivas, nas leituras a g_{i-1} e y_{i-1} no primeiro laço, e x_{i+1} no segundo. Estas dependências são sincronizadas por meio da inserção de duas regiões críticas no primeiro laço, a primeira envolvendo os comandos $c1$ e $c2$, e a segunda envolvendo o comando $c3$. No segundo laço a região crítica envolve, a princípio, todo o laço.

Para este experimento foi utilizada a versão do simulador descrita em [6] com a seguinte configuração:

- 4 processadores;
- memória entrelaçada[13] em 8 bancos;

[13]interleaved

- *bandwidth* do barramento: 2 requisições simultâneas;
- tempo de leitura/escrita: mínimo de 2 ciclos, influenciada por conflito de requisições no sistema de memória;
- tempo de multiplicação: 4 ciclos;
- tempo de divisão: 12 ciclos;
- tempo de adição: 2 ciclos;
- demais instruções: 1 ciclo.

Os tempos de execução das instruções sobre números ponto–flutuante (multiplicação, divisão e adição) são baseados no processador MIPS R2000/R3000 [25]. Os resultados para as primitivas *TAS*, *full/empty bit* e *flowbit* são apresentados na tabela abaixo, considerando-se N = 128:

	Seqüencial	*TAS*	*full/empty bit*	*flowbit*
número de ciclos	16452	15832	9329	8595
ganho	1	1.04	1.76	1.91

O paralelismo no primeiro laço decorre da sobreposição da execução da segunda região crítica em uma iteração i com a execução da primeira região crítica pela iteração $i+1$. O paralelismo no segundo laço decorre de um rearranjo da expressão, antecipando a operação de divisão para fora da região crítica. O segundo laço é rearranjado para:

do $i = N-1, 1, -1$
 $aux \leftarrow 1.0 \div g_i$
 $x_i \leftarrow aux \times (y_i - ds_i \times x_{i+1})$
enddo

As primitivas baseadas no controle de acesso aos dados, como o *full/empty bit* e o *flowbit*, apresentam maior desempenho que o *TAS*, na paralelização destes laços, por dois motivos:

1. O *TAS* implica em *alto custo de manutenção* de três regiões críticas, no que se refere aos testes e liberações de vetores de semáforos.

2. A sincronização pelas primitivas *full/empty bit* e *flowbit* implicam em regiões críticas menores, ou seja, apresentam um trecho seqüencial menor que os laços sincronizados por *TAS*. Isto ocorre porque a sincronização baseada em semáforos seqüencializa *todos os comandos* envolvidos na dependência de fluxo. A primitiva *full/empty bit* implica em um região crítica que compreende apenas o trecho entre as *referências* destino e fonte da dependência.

   Como exemplo, a primeira região crítica do primeiro laço, sincronizada pelo *TAS*, resulta em um trecho seqüencial compreendendo os dois primeiros comandos. Para o *full/empty bit*, a região crítica inicia-se na leitura de g_{i-1}, ou seja, as operações de leitura de di_i, inversão de sinal e indexação de g_{i-1} estão fora da região crítica e dentro do trecho paralelo. Poder-se-ia argumentar que este rearranjo também poderia ser efetuado pelo *TAS*, porém com o *full/empty bit* ele ocorre naturalmente.

O ganho apresentado pelo *flowbit* sobre o *full/empty bit*, neste caso, decorre da deficiência do *full/empty bit*, apontada no tópico *2.5.3*, quanto a múltiplas leituras sucessivas a um mesmo dado. Dado que os vetores g e y são lidos no primeiro laço, nas referências g_{i-1} e y_{i-1}, ao final deste laço o estado das variáveis é *empty*. Como o segundo laço utiliza novamente estes vetores, é necessário executar uma operação FILL nestes vetores, acarretando em custo adicional.

2.8 Comentários Finais

O resultado anterior aponta o *flowbit* como uma forma eficiente e de uso automático para sincronização em programas com atribuições únicas e múltiplas leituras. FORTRAN, contudo, não é uma linguagem de atribuições únicas. A possibilidade de reatribuição de variáveis viola esta condição e gera dependências de saída e anti-dependências no programa. O tratamento de tais dependências aumenta a complexidade da sincronização para todas as primitivas.

O capítulo seguinte apresenta uma proposta de paralelização e sincronização de laços FORTRAN que supera estas dificuldades através da eliminação de anti-dependências e dependências de saída durante a execução, colocando o laço na condição de atribuições únicas e múltiplas leituras, propício para a sincronização automática oferecida pelo *flowbit*.

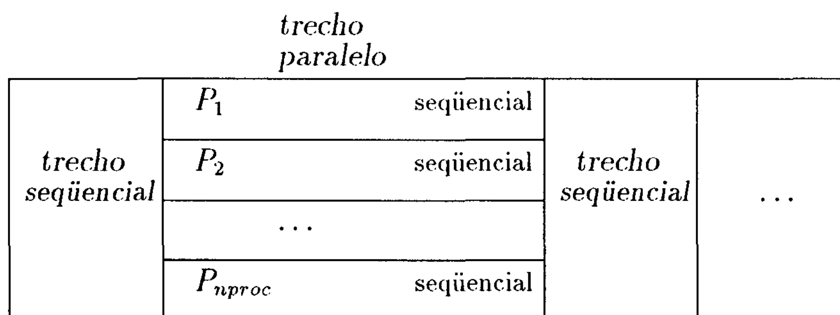
Capítulo 3

Um Esquema de Sincronização para Laços Paralelos

Este capítulo introduz condições que preservam a semântica de um programa seqüencial na sua reestruturação para permitir a exploração de paralelismo. Para garantir estas condições, no escopo de paralelização de laços, propõe-se um esquema inédito de sincronização, baseado no reendereçamento de escritas e leituras durante a execução, eliminando-se anti-dependências e dependências de saída. As dependências de fluxo restantes são sincronizadas pela primitiva *flowbit*.

3.1 Esquema de Execução Paralela

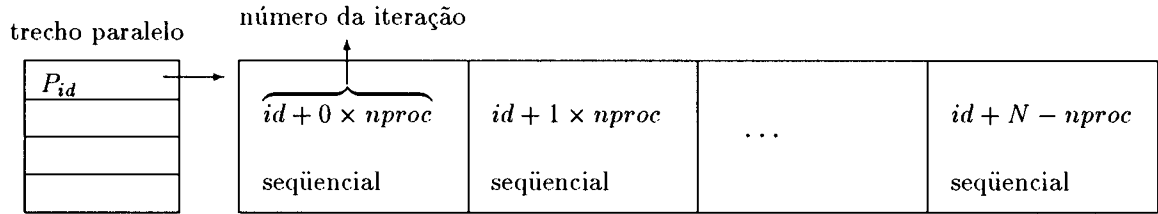
Esta dissertação limita-se ao estudo da paralelização de laços mais internos, ao nível de granularidade média. Nesta granularidade, o paralelismo é explorado através da execução simultânea de iterações distintas em processadores distintos, onde cada iteração é executada *seqüencialmente* em um processador. A figura abaixo esboça o esquema de paralelização de um programa:



nproc: número de processadores

P_{id}: identificação do processador $id = 1..nproc$

O trecho paralelo representa um laço mais interno. Este laço é dividido em partes seqüenciais, que são executadas simultaneamente por processadores distintos. Cada parte seqüencial corresponde a um conjunto de iterações, como no detalhe abaixo:



Supondo-se que o escalonamento das iterações aos processadores seja efetuado previamente durante a compilação[1], e ainda que $nproc = 8$ e N, para exemplificação, seja múltiplo de $nproc$ e igual a 64, o processador identificado por P_1 executaria seqüencialmente as iterações *1, 9, 17, 25, ..., 57.*

Exemplo:

$$
\begin{array}{l}
\textbf{do } i = 1,6 \\
\quad a_i \leftarrow y_i \times z_i \\
\quad y_i \leftarrow a_{i+1} \\
\textbf{enddo}
\end{array}
$$

Assumindo uma máquina com três processadores e uma política de pré-escalonamento, os processadores executariam as seguintes iterações:

P1	*P2*	*P3*
$i = 1$	$i = 2$	$i = 3$
$a_1 \leftarrow y_1 \times z_1$	$a_2 \leftarrow y_2 \times z_2$	$a_3 \leftarrow y_3 \times z_3$
$y_1 \leftarrow a_2$	$y_2 \leftarrow a_3$	$y_3 \leftarrow a_4$
$i = 4$	$i = 5$	$i = 6$
$a_4 \leftarrow y_4 \times z_4$	$a_5 \leftarrow y_5 \times z_5$	$a_6 \leftarrow y_6 \times z_6$
$y_4 \leftarrow a_5$	$y_5 \leftarrow a_6$	$y_6 \leftarrow a_7$

Dado que cada processador executa um conjunto de $\frac{N}{nproc}$ iterações e que qualquer iteração demanda o mesmo tempo de execução, o ganho de desempenho[2] decorrente

[1] pre-scheduling
[2] speed-up

da paralelização do laço é, a princípio, igual a *nproc*. Este ganho indica que o tempo de execução do programa paralelo é uma fração do tempo de execução do programa seqüencial equivalente, igual à $\frac{tempo\ sequencial}{nproc}$. No exemplo anterior, o paralelismo surge da execução simultânea das iterações *1*, *2* e *3*, em um primeiro instante, e das iterações *4*, *5* e *6*, posteriormente. O tempo de execução do programa paralelo é, a princípio, três vezes menor que o do programa seqüencial.

Este ganho é irreal na prática, devido a limitações e custos adicionais da execução paralela, como a seqüencialização imposta pela sincronização dos comandos (em iterações distintas) que possuem dependências de dados, o custo do escalonamento das iterações aos processadores, em particular nas políticas de auto-escalonamento[3], conflitos no barramento de dados e na memória devido a requisições de acesso paralelas, dentre outras.

Um dos objetivos de um compilador reestruturador, bem como da arquitetura de uma máquina paralela, é diminuir ao máximo o custo destas limitações, a fim de tornar o ganho de desempenho diretamente proporcional ao número de processadores.

3.2 Condições Suficientes para a Execução Correta de Laços Paralelos

A preservação da semântica seqüencial é uma condição básica para a reestruturação de trechos de programas. Admita que a execução do programa seqüencial seja uma função

$$f : memória \longrightarrow memória$$

onde *memória* representa o conjunto de endereços reservados ao programa seqüencial. Registradores não serão considerados.

Desconsiderando acessos à memórias secundárias, bem como comandos de entrada e saída, admita ainda que cada comando é função das suas leituras, na forma

$$g : endereços_leitura \longrightarrow endereço_escrita$$

A paralelização do programa seqüencial preserva a semântica se *computar a mesma função* que o programa original. Para isto, basta garantir que o estado da memória, ao término da execução paralela, seja idêntico ao do término da execução seqüencial. Por estado de memória idêntico entende-se que os *mesmos endereços* de memória utilizados no programa seqüencial possuem os *mesmos valores*, nas duas execuções.

Para tanto, basta garantir que a execução paralela atenda às seguintes condições:

[3]self-scheduling

1. *sejam alterados exatamente os mesmos endereços de memória que na execução seqüencial*;

2. *a última escrita a um endereço armazene o mesmo valor que na execução seqüencial.*

Uma forma de garantir estas duas condições, na execução paralela, é fazer com que:

1'. *escritas a um mesmo endereço sigam a ordem de execução seqüencial*;

2'. *leituras recebam o mesmo valor que na execução seqüencial*;

3'. *sejam executados os mesmos comandos que na execução seqüencial.*

A condição **2'** garante que os comandos executados computam e armazenam os mesmos valores que na execução seqüencial. Como **3'** garante que todos estes comandos são executados, de **2'** e **3'** temos que todos os comandos produzem os mesmos valores que a execução seqüencial e que todos estes resultados são armazenados. Resta garantir que o último resultado armazenado em cada endereço provém da mesma escrita nas duas execuções. A condição **1'** garante esta última escrita, e junto com **2'** e **3'**, garante a condição **2**. Observe que a condição **3'** garante **1**. Logo, o par **1-2** pode ser substituído por **1'**, **2'** e **3'**.

Supõe-se neste trabalho que o fluxo de controle é preservado na execução paralela por sincronização apropriada[4], garantindo a condição **3'**. Os tópicos seguintes limitam-se a garantir as condições **1'** e **2'**.

3.3 Garantindo a Semântica em Programas com Atribuições Únicas

Em programas com atribuições únicas, é possível garantir que o conteúdo de um endereço, após uma escrita, permanece inalterado durante toda a execução do programa, pois não há outra escrita ao mesmo endereço. *Eliminam-se, portanto, os conflitos de anti-dependências e dependências de saída*, pois estes estão vinculados ao reuso de endereços de memória.

Através da sincronização das dependências de fluxo restantes, por meio da primitiva *flowbit*, garante-se que as leituras aguardam a execução da escrita no endereço correspondente. Com as dependências de fluxo sincronizadas, garantem-se as condições **1'** e **2'**, pois:

[4] apesar de já contarmos com trabalhos neste sentido, não será apresentada sincronização de controle para manter a clareza da apresentação

a) Como só há uma escrita a um endereço de memória, esta é conseqüentemente a última escrita, garantindo a condição **1'**.

b) Se os acessos de leitura a um endereço estão sincronizados com o acesso de escrita, os valores recuperados são os mesmos que na execução seqüencial. Logo, **2'** está garantido.

3.4 Dificuldades para Garantir a Semântica em Programas com Múltiplas Atribuições

3.4.1 Anti-dependências e Dependências de Saída

A reatribuição de variáveis em um programa gera anti-dependências e dependências de saída, aumentando a complexidade da sincronização. Como exemplo desta dificuldade, suponha a sincronização do laço abaixo, nas referências à variável x, sabendo-se que os processadores *P1* e *P2* executam, respectivamente, as iterações *1* e *2*:

P1	*P2*
do	**do**
$x \leftarrow a_i + b$	$x \leftarrow a_i + b$
$\ldots \leftarrow x$	$\ldots \leftarrow x$
enddo	**enddo**

O esquema de execução paralelo garante que as leituras de x ocorrerão após as escritas da mesma iteração. Através da primitiva *full/empty bit*, garante-se ainda que os acessos a x obedecem a ordem STORE/LOAD/STORE/LOAD/..., impedindo que uma leitura de uma iteração utilize o valor de x escrito pela outra iteração. No entanto, é possível que, ao final da execução destas duas iterações, o valor de x seja o definido na iteração *1*, pois não há uma ordem obrigatória na execução dos $STORE entre as iterações. Esta possibilidade de inversão viola a condição **1'**, da ordem das escritas, e altera a semântica do programa.

Este exemplo mostra a dificuldade existente para garantir a ordem das escritas, quando há atribuições a um mesmo endereço em iterações distintas. No esquema de execução paralela em questão, no entanto, é possível garantir esta ordem em duas situações particulares:

- No trecho do programa fora de laços mais internos. Nestes trechos a execução é seqüencial, garantindo a ordem das escritas.

- Em laços onde as múltiplas atribuições a um mesmo endereço ocorrem *dentro da mesma iteração*. Como cada iteração é executada seqüencialmente em um processador, a ordem das atribuições será mantida e o laço executará corretamente em paralelo. Este caso retrata dependências de saída de *distância zero*, ou seja, dependências *intra-iteração*.

A condição crítica, portanto, é garantir a ordem dos acessos a um mesmo endereço quando executados em *iterações distintas*. Como exemplo, suponha o laço da página 33, com a mesma divisão das iterações entre os processadores:

do $i = 1, 6$
  $a_i \leftarrow y_i \times z_i$
  $y_i \leftarrow a_{i+1}$
enddo

P1	*P2*	*P3*
$i = 1$	$i = 2$	$i = 3$
$a_1 \leftarrow y_1 \times z_1$	$a_2 \leftarrow y_2 \times z_2$	$a_3 \leftarrow y_3 \times z_3$
$y_1 \leftarrow a_2$	$y_2 \leftarrow a_3$	$y_3 \leftarrow a_4$
$i = 4$	$i = 5$	$i = 6$
$a_4 \leftarrow y_4 \times z_4$	$a_5 \leftarrow y_5 \times z_5$	$a_6 \leftarrow y_6 \times z_6$
$y_4 \leftarrow a_5$	$y_5 \leftarrow a_6$	$y_6 \leftarrow a_7$

Como uma iteração executa *seqüencialmente* em um processador, a anti-dependência no acesso a y_i não precisa ser sincronizada, já que ela se manifesta *dentro de uma mesma iteração*. Entretanto, há conflito entre a leitura de a_{i+1} em uma iteração e a escrita em a_i na iteração seguinte, já que estas iterações executam simultaneamente em processadores distintos.

3.4.2 Deficiência na Detecção e Inserção da Sincronização

Outro fator que, aliado a anti-dependências e dependências de saída, dificulta a garantia da manutenção da semântica de um programa paralelo é a deficiência e por vezes impossibilidade de detecção das dependências e inserção de código de sincronização.

O capítulo anterior mostrou a deficiência da inserção das primitivas quando é exigido o cálculo da distância entre iterações dependentes. Quando esta distância é variável

ou desconhecida, estas propostas forçam a *execução seqüencial* dos laços nas máquinas paralelas. Um estudo empírico recente [20] sobre índices de *arrays* e dependências de dados, em bibliotecas de subrotinas FORTRAN, mostrou que somente uma pequena parte das dependências (13,65%) possui distância *constante*, que pode ser determinada durante a compilação.

O principal fator que levou 86% das dependências a possuírem distâncias variáveis ou desconhecidas na fase de compilação foi a *não linearidade* dos índices de *arrays*. Este problema surge na indexação de *arrays* com variáveis que não são funções lineares do índice do laço.

Para ilustrar a indecidibilidade do processo de detecção das dependências durante a compilação, observe os seguintes exemplos.

Exemplo 1

$$\begin{array}{ll} & \textbf{do } i = 1, N \\ c_1 : & \quad a_{b_i} \leftarrow a_{c_i} \\ & \textbf{enddo} \end{array}$$

Dado que b_i e c_i são vetores cujos valores são computados no próprio programa que contém o laço, os elementos que receberão acesso do vetor a só são definidos durante a execução. Portanto, o grafo de dependências deste laço deve prever todas as possíveis relações de dependências entre os acessos ao vetor a, que são:

$$c_1 \ \delta \ c_1 \qquad c_1 \ \delta^- \ c_1 \qquad c_1 \ \delta^\circ \ c_1$$

Outro agravante é que as distâncias entre dependências são *desconhecidas durante a compilação*, porque estão relacionadas aos valores de b_i e c_i. Estes valores podem, ainda, determinar distâncias variáveis. Os compiladores reestruturadores atuais, na presença de tais laços, forçam a seqüencialização da execução.

A falta de suporte adequado de sincronização impede que este laço execute em paralelo ainda que haja iterações independentes. Por exemplo, se os valores de b_i e c_i fossem iguais a i, não haveria dependência entre as iterações e o laço poderia ser paralelizado.

Exemplo 2

```
/* programa principal */
    ...
    call Vadd (x, y, z)
    ...
    call Vadd (x, x, z)
    ...
subroutine Vadd (a, b, c)
    do i = 1,n
        a_{i+1} ← b_i + c_i
    enddo
```

O laço na subrotina *Vadd*, que aparentemente não possui conflitos entre iterações, não pode ser paralelizado sem uma análise dos endereços dos parâmetros efetivos. A primeira chamada (**call** *Vadd(x,y,z)*) não exige sincronização na execução do laço, porque os parâmetros formais a, b e c referem vetores diferentes, no caso, x, y e z.

No entanto, a segunda chamada resulta em uma *dependência de fluxo*, originando-se na escrita em a_{i+1} para a leitura em b_i da iteração seguinte. Estes dois acessos atuam sobre o mesmo elemento do vetor x, já que este vetor é vinculado[5] aos parâmetros a e b na chamada da função. A paralelização deste laço pode alterar a semântica do programa por violar a condição **2'** (das leituras retornarem o valor original).

Estes exemplos mostram claramente que a deficiência na detecção das dependências surge porque os endereços utilizados no laço só são *definidos durante a execução*, logo é impossível determinar *a priori* se haverá conflito nos acessos à memória pelas iterações paralelas. No primeiro exemplo, os endereços do vetor a são definidos pelos *valores* contidos em b_i e c_i. No segundo exemplo, a vinculação[6] entre o vetor x e o parâmetro a também é efetuada durante a execução.

A limitação dos esquemas atuais, juntamente com alguns estudos empíricos [20], indicam a necessidade de um esquema que resolva as dependências durante a execução do programa. Este esquema deve resolver as anti-dependências e dependências de saída para garantir a ordem entre as escritas conflitantes a um mesmo endereço e sincronizar as dependências de fluxo para garantir que as leituras a um endereço retornem o mesmo valor que na execução seqüencial.

[5] bound

[6] binding

3.5 Uma Proposta Inicial

Uma primeira abordagem possível para garantir as condições **1'** e **2'** e preservar a semântica no programa paralelo, é fazer com que:

> *um acesso de escrita a um endereço aguarde até que sejam satisfeitos todos os acessos anteriores de escritas e leituras a este endereço, como na ordem de execução seqüencial.*

Este esquema, proposto em [21], garante as condições **1'** e **2'** através da *seqüencialização de acessos* ao mesmo endereço, pois:

a) Se uma escrita aguarda até que todas as anteriores sejam satisfeitas, garante-se a última escrita (**1'**).

b) Se uma escrita aguarda até que todas as leituras sejam satisfeitas, garante-se que as leituras retornam o valor armazenado pela escrita mais recente, como na ordem de execução original, garantindo a condição **2'**.

O fator crítico deste esquema é a definição da *ordem de acesso* a um determinado endereço. A ordem de acesso é uma ordem linear, α, de leituras e escritas a um endereço durante a execução do programa ($A_i \alpha A_j$ significa que o acesso A_i é executado antes que A_j). O esquema prega que, na execução paralela do programa, esta ordem de acessos deve ser mantida para garantir as dependências de dados (fluxo, anti e saída), com exceção que, entre duas escritas a um endereço, *leituras sucessivas* a este endereço podem ser executadas paralelamente, por não alterarem o conteúdo do endereço.

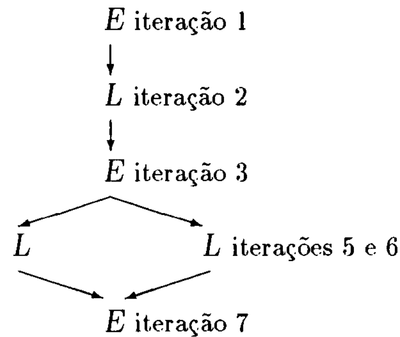
O esquema conta com primitivas adequadas (para o multiprocessador *Cedar*) de escrita e leitura. As primitivas, baseadas no controle de acesso aos dados, forçam a seqüencialização dos acessos segundo a ordem (α) definida para um determinado endereço. O laço abaixo ilustra como o esquema atua na sincronização dos acessos a um endereço:

do $i = 1, 7$
  $a_{b_i} \leftarrow a_{c_i}$
enddo

Admita os seguintes valores para os vetores b e c:

i	1	2	3	4	5	6	7
b_i	1	2	1	4	4	6	1
c_i	2	1	2	4	1	1	7

considerando apenas os acessos ao elemento a_1 e que E representa uma escrita e L uma leitura, a ordem de acesso a este elemento é definida por:



A execução deste laço em paralelo, segundo o esquema, deve respeitar a ordem de acessos ao elemento a_1, logo os acessos nas iterações *1, 2, 3, 5, 6* e *7* devem ser seqüencializados para respeitar as dependências de dados, com exceção das leituras nas iterações *5* e *6*, que podem ser paralelizadas.

Um dos grandes atrativos deste esquema é a possibilidade de efetuar o cálculo da ordem de acesso *durante a execução*, permitindo a exploração de paralelismo em laços onde a detecção tradicional força a seqüencialização. Além disso, por contar com primitivas de controle de acessos aos dados, o esquema não necessita do cálculo das distâncias para sincronizar as dependências.

No entanto, com o objetivo de garantir as dependências, este esquema é penalizado pela *seqüencialização* das escritas e leituras ao mesmo endereço. Esta seqüencialização, somada ao custo aparentemente alto do cálculo da ordem de acesso aos endereços, pode degradar significativamente o desempenho do laço paralelo.

3.6 Uma Nova Proposta

3.6.1 Revendo as Condições Básicas para Garantir a Semântica

Outra forma de preservar a semântica do programa (condição **2**) é garantir que, na execução paralela:

1". *A última escrita a um endereço armazene um valor computado pelo mesmo comando que na execução seqüencial.*

2". *As leituras recebam os valores armazenados pelas escritas mais recentes, como na ordem seqüencial.*

A condição **1"** garante que o último valor armazenado em um endereço é computado pelo mesmo comando em ambas as execuções. A condição **2"** garante que qualquer comando computa o mesmo valor que na execução seqüencial, pois utiliza os mesmos valores. Logo, o valor final do endereço, após a execução paralela, é o mesmo que da execução seqüencial e a semântica é preservada. Estas duas condições, portanto, substituem **1'** e **2'**, pois garantem a condição **2** original.

Os tópicos seguintes apresentam uma proposta de sincronização que garante as condições **1"** e **2"** através do *reendereçamento* de escritas e leituras de um laço. A proposta implica em um esquema particular de execução paralela, a ser descrito no tópico *3.8*.

3.6.2 Reendereçamento de Escritas

A condição **1"** diz que o valor final de um endereço deve ser idêntico para as duas execuções. A proposta anterior garantia o valor final de um endereço através da *seqüencialização* das escritas paralelas, como na ordem original de execução, evitando as *condições de corrida* ao mesmo endereço.

Esta seqüencialização é indesejável. Deseja-se obter um esquema que permita a execução paralela das escritas. Tal esquema é possível a partir de uma constatação importante:

- a condição **1"**, a rigor, *não impõe uma ordem de escritas*, mas apenas que o último *valor* computado para um determinado endereço seja igual ao da execução seqüencial.

Como a condição **1"** não impõe uma ordem de escritas, o esquema a ser proposto executará as escritas em paralelo. No entanto, para garantir a condição **1"** na presença de escritas paralelas a um mesmo endereço, é preciso que o esquema garanta que o *valor* computado no último comando de escrita a um endereço, na ordem de execução seqüencial, seja o último valor armazenado neste endereço na execução paralela. Para isto, é necessário que:

1. Identifique-se qual é a *última escrita* a um endereço na ordem de execução seqüencial.
2. Ao final da execução paralela, o *valor* computado por este comando de escrita esteja armazendo no endereço de memória correspondente.

Identificação da Última Escrita a um Endereço

A identificação da última escrita será feita através de uma *enumeração* das escritas efetivamente realizadas no laço, da seguinte forma:

1. *Linearizam-se* todas as escritas do laço, de forma que cada escrita em cada iteração contenha um número único.
2. Durante a execução do laço, efetua-se seqüencialmente o *cálculo dos endereços de escrita*, associando a este endereço o número linear da escrita definido anteriormente.

Desta forma, ao final do laço, é possível identificar, para cada endereço de escrita, qual foi a última escrita no laço a atualizar cada endereço. A implementação da enumeração é feita através de um vetor denominado *EMR* (*escritas mais recentes*) que, para cada posição de memória, contém o número da última escrita a esta posição (passo 2 acima).

Exemplo: Suponha o laço da página 40, com os mesmos valores de b_i e c_i. Neste laço, como cada iteração possui apenas uma escrita, a linearização acarreta que as escritas no laço sejam determinadas pelo número da iteração. No caso:

escrita	a_{b_1}	a_{b_2}	a_{b_3}	a_{b_4}	a_{b_5}	a_{b_6}	a_{b_7}
número	1	2	3	4	5	6	7

Desta forma, para o elemento a_1 (a_{b_1},a_{b_3},a_{b_7}), a enumeração seria definida por *1 – 3 – 7*, que são as escritas que alteram o conteúdo deste endereço. A última escrita no laço em a_1 é, portanto, a da iteração *7*.

Considerando os valores de b_i e c_i, o vetor *EMR* conteria, ao final do laço do exemplo anterior, os seguintes valores:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
EMR	7	2	-	5	-	6	-

Paralelização das Escritas

Dado que é possível determinar a última escrita a cada endereço, faz-se necessário definir um esquema que permita a execução paralela das escritas. O esquema a ser utilizado origina-se das seguintes constatações:

- A enumeração das escritas não obriga a realização das escritas seqüencialmente, e sim que o *cálculo dos endereços de escrita seja efetuado como na ordem de execução seqüencial.*

- A condição **1"** não obriga que os valores computados pelos comandos sejam armazenados na memória, *a não ser o último valor* computado para cada endereço. Reiterando que, por memória, entende-se o conjunto de endereços utilizados pelo programa seqüencial.

Com base nestas constatações, é proposta a reorientação das escritas a uma *"memória auxiliar"*. Esta reorientação exige o cálculo de um *novo endereço* na memória auxiliar, no qual as escritas serão efetuadas.

Nossa proposta é que estes novos endereços sejam um *mapeamento da enumeração* das escritas, ou seja, um endereço desta memória auxiliar seja representado pelo número da escrita no laço. Como a linearização implica em um número único para cada escrita no laço, as escritas são reorientadas a *endereços distintos* da memória auxiliar. Este esquema coloca o laço na condição de *atribuições únicas*, permitindo a execução paralela das escritas.

Supondo que o vetor *AUX* representa a memória auxiliar, a nova semântica de uma operação de escrita, incluindo a enumeração, passa a ser:

escreve *valor* **em** *endereço_escrita*:

$$EMR_{\text{endereço_escrita}} \leftarrow \mathit{número_da_escrita_no_laço}$$

$$AUX_{\text{número_da_escrita_no_laço}} \leftarrow \mathit{valor}$$

O laço abaixo ilustra a transformação das escritas no laço da página 40:

$$
\begin{array}{l}
\textbf{do } i = 1,7 \\
\quad EMR_{\text{endereço}(a_{b_i})} \leftarrow i \\
\quad AUX_i \leftarrow a_{c_i} \\
\textbf{enddo}
\end{array}
$$

As escritas originais aos elementos a_1, no laço acima, são efetuadas agora aos endereços AUX_1, AUX_3 e AUX_7. Como estes endereços são distintos, *eliminam-se as dependências de saída*, permitindo o paralelismo das escritas.

Atualização da Memória Original

Finalmente, para garantir a condição **1"**, basta atualizar, ao final do laço, os endereços da memória original. Como as escritas paralelas a cada endereço original são reorientadas à memória auxiliar, basta obter o *endereço* da memória auxiliar no qual foi executada a *última escrita* a um endereço original. Este endereço está contido no vetor EMR, já que, ao final do laço, a escrita mais recente a um endereço é a última escrita. A atualização dos endereços originais é feita pelo seguinte comando:

$$endereço_original \leftarrow AUX_{EMR_{\text{endereço_original}}}$$

Assumindo os valores de EMR da página 43, a atualização dos endereços finais do vetor a, no laço da página 40, seria:

$$
\begin{array}{l}
a_1 \leftarrow AUX_7 \\
a_2 \leftarrow AUX_2 \\
a_4 \leftarrow AUX_5 \\
a_6 \leftarrow AUX_6
\end{array}
$$

3.6.3 Reendereçamento de Leituras

A condição **2"** diz que as leituras devem utilizar o valor armazenado pela *escrita mais recente* ao endereço, como na ordem original. Contudo, no novo esquema, as escritas são reorientadas a novos endereços de uma memória auxiliar. Há portanto, dois problemas a serem resolvidos:

1. Definir qual é a *escrita mais recente*, na ordem de execução original.

2. Definir qual é o *novo endereço* desta escrita na memória auxiliar, para que a leitura seja também reorientada e utilize o valor correto.

Como a definição da escrita mais recente é feita *seqüencialmente* no vetor *EMR*, em um determinado ponto da execução do laço é possível definir, para um endereço, qual é esta escrita. Se os endereços das leituras efetivamente realizadas também forem calculados seqüencialmente, será possível definir, *até o momento desse cálculo*, qual foi a última escrita a este endereço.

Esta definição, para uma leitura na forma **lê de** *endereço_leitura*, é feita por um acesso ao vetor *EMR*, como ilustrado abaixo:

$$\textit{última_escrita} \leftarrow EMR_{\text{endereço_leitura}}$$

A vantagem deste esquema é que a enumeração que define a última escrita também define o *novo endereço* desta escrita na memória auxiliar. Portanto, a definição da escrita mais recente também define automaticamente o endereço desta escrita, que é, a rigor, o *endereço onde a leitura deve ser efetuada*.

Este esquema resolve os dois problemas mencionados anteriormente. A nova semântica de uma operação de leitura passa a ser:

leitura de *endereço_leitura*:
$\quad \textit{novo_endereço_leitura} \leftarrow EMR_{\text{endereço_leitura}}$
$\quad$ **lê de** $AUX_{\text{novo_endereço_leitura}}$

Este esquema resolve o reendereçamento de leituras cujas escritas mais recentes são executadas *dentro do laço*. Quando a escrita mais recente ocorre fora do laço, ou seja, no *trecho de execução seqüencial*, a leitura deve utilizar o valor contido na memória original. Ao obter o número da escrita mais recente, portanto, deve ser possível à leitura identificar que esta escrita foi efetuada no trecho seqüencial. Para tanto, os valores iniciais do vetor *EMR* devem ser menores que o *limite inferior* do laço, simbolizando o trecho seqüencial.

Os laços a seguir ilustram o reendereçamento completo das leituras (já prevendo escritas fora do laço) e das escritas, inclusive com a atualização final da memória original, para o laço da página 40:

```
do i = 1,7
    novo_end_leitura ← EMR_endereço(a_{c_i})
    EMR_endereço(a_{b_i}) ← i
    if novo_end_leitura = 0 then            /* EMR fora do laço */
        AUX_i ← a_{c_i}
    else                                    /* EMR no laço */
        AUX_i ← AUX_{novo_end_leitura}
    endif
enddo

do i = 1,7
    if EMR_endereço(a_{b_i}) = i then
        a_{b_i} ← AUX_i
    endif
enddo
```

Exemplo: dado os valores de b_i e c_i, como no exemplo do tópico *3.5*:

i	1	2	3	4	5	6	7
b_i	1	2	1	4	4	6	1
c_i	2	1	2	4	1	1	7

o *trace* abaixo ilustra, para o primeiro laço, os valores do vetor *EMR* e os novos endereços de leitura (indicando a reorientação), ao final da execução de cada iteração:

- inicialmente:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
EMR	0	0	0	0	0	0	0

- após a iteração *1*:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
EMR	1	0	0	0	0	0	0

escreve a_{b_1} em AUX_1

novo_endereço_leitura = 0 $\Rightarrow$ lê a_{c_1} de a_2

- após a iteração *2*:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
EMR	1	2	0	0	0	0	0

escreve a_{b_2} em AUX_2

novo_endereço_leitura = 1 $\Rightarrow$ lê a_{c_2} de AUX_1

- após a iteração *3*:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
EMR	3	2	0	0	0	0	0

escreve a_{b_3} em AUX_3

novo_endereço_leitura = 2 $\Rightarrow$ lê a_{c_3} de AUX_2

- após a iteração *4*:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
EMR	3	2	0	4	0	0	0

escreve a_{b_4} em AUX_4

novo_endereço_leitura = 0 $\Rightarrow$ lê a_{c_4} de a_4

- após a iteração *5*:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
EMR	3	2	0	5	0	0	0

escreve a_{b_5} em AUX_5

novo_endereço_leitura = 3 $\Rightarrow$ lê a_{c_5} de AUX_3

- após a iteração *6*:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
EMR	3	2	0	5	0	6	0

escreve a_{b_6} em AUX_6

novo_endereço_leitura = 3 $\Rightarrow$ lê a_{c_6} de AUX_3

- após a iteração *7*:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7
EMR	7	2	0	5	0	6	0

escreve a_{b_7} em AUX_7

novo_endereço_leitura = 0 $\Rightarrow$ lê a_{c_7} de a_7

Repare que o esquema proposto reorienta as escritas a a_{b_1} (em AUX_1) e a_{b_3} (em AUX_3), embora ambas escrevam, na execução seqüencial, em a_1. Repare também que as leituras de a_{c_2} e a_{c_5}, ambas de a_1, são reorientadas para AUX_1 e AUX_3.

Como múltiplas escritas ao mesmo endereço são reorientadas para endereços distintos e as leituras utilizam apenas o endereço da escrita mais recente, *eliminam-se anti-dependências*. As dependências de fluxo restantes, como na proposta de atribuições únicas, são garantidas através da primitiva *flowbit*, nas leituras e escritas à memória auxiliar. Este laço, contudo, ainda não está na forma paralela. Uma proposta de paralelismo é discutida no tópico *3.8*.

Cabe aqui uma observação sobre a forma de implementação da atualização da memória original. Como esta atualização é feita após a execução do laço e utiliza os endereços originais de escrita (no exemplo anterior, **if** $EMR_{\text{endereço}(a_{b_i})} = i$) para determinar a última

escrita a este endereço, a implementação atual não comporta laços onde os índices de *arrays* são *alterados dentro do laço*. Exemplo:

$$a_{b_i} \leftarrow \ldots$$
$$b_{c_i} \leftarrow \ldots$$

Neste exemplo, se b_i fosse alterado no laço, após sua leitura em a_{b_i}, não seria possível obter seu valor original na atualização da memória (**if** $EMR_{\text{endereço}(a_{b_i})} = i$). Em razão disto, restringimos o estudo a laços onde os índices de vetores não são alterados no laço.

3.7 Considerações Sobre o Esquema

O reendereçamento das escritas e leituras constitui a essência do novo esquema de sincronização. A aplicação deste reendereçamento em um laço, aliado a sincronização de fluxo pela primitiva *flowbit*, fornecem um esquema de sincronização potencialmente mais eficiente que o esquema de [21], devido ao maior grau de paralelismo na execução do laço, além de apresentar uma série de vantagens sobre esquemas tradicionais de detecção e inserção de sincronização, enumeradas abaixo:

1. O esquema de sincronização proposto *elimina anti-dependências* e *dependências de saída*. Com isso, escritas ao mesmo endereço podem ser paralelizadas, já que o *laço* passa a ser constituído de *atribuições únicas*. Este esquema apresenta mais paralelismo que a proposta baseada na ordem de acessos, descrita no tópico *3.5*, pois esta última faz com que uma escrita aguarde até que sejam executados todos os acessos anteriores ao endereço, implicando em excesso de *seqüencialização*.

   No exemplo da página 40, o esquema inicial impõe a seqüencialização dos acessos ao endereço de a_1, nas iterações *1, 2, 3, 5, 6* e *7*, pois, além de sincronizar as dependências de saída nas iterações *1, 3* e *7*, deve-se garantir as anti-dependências entre as iterações 2 e 3, entre 5 e 7, e 6 e 7.

   No novo esquema, todos os acessos das iterações *1, 2, 3, 5, 6* e *7* podem ser executados em paralelo, aumentando o ganho de desempenho. As dependências de fluxo, para o elemento a_1, das iterações *1* e *3*, para *2* e *5*, são garantidas nos dois esquemas por sincronização apropriada.

2. A inserção da primitiva de sincronização no novo esquema é muito simplificada. Como se manifestam apenas dependências de fluxo, basta inserir as primitivas $STORE e $LOAD (*flowbit*) nos acessos à memória auxiliar, sem *nenhum código adicional* para sincronização.

3. O novo esquema *dispensa o cálculo das distâncias* entre iterações dependentes. A distância é necessária em esquemas tradicionais para determinar as iterações de origem e destino da dependência. No novo esquema, como as dependências de fluxo são sincronizadas pelo *flowbit*, que é uma primitiva baseada no controle de acesso aos dados, o cálculo da distância é desnecessário.

4. Como se manifestam apenas dependências de fluxo, e dado que as leituras são *emparelhadas* com as escritas mais recentes, através do reendereçamento das leituras, o esquema *dispensa a detecção de dependências*, já que esta detecção é implicitamente executada pelo cálculo dos novos endereços.

   Outra vantagem é que, como o cálculo de novos endereços é feito *durante a execução* do laço, o esquema sincroniza dependências que não podem ser detectadas pelos esquemas atuais, como aquelas exemplificadas no tópico *3.4.2*.

Há vários pontos que ainda estão sob investigação para a definição de uma proposta eficiente de implementação. Um destes pontos é o *tamanho dos vetores* AUX e EMR. Como AUX é indexado pela enumeração das escritas em um laço, seu tamanho só pode ser calculado em tempo de compilação quando o limite superior do laço também for conhecido. Uma possível solução seria dividir o laço em laços menores que comportassem um tamanho pré-definido de AUX. Como o vetor EMR é indexado pelos endereços de memória utilizados no programa seqüencial, seu tamanho deve ser proporcional a memória. Estamos estudando formas de diminuir o tamanho deste vetor, para uma implementação mais eficiente.

Outro ponto crítico a ser resolvido é a paralelização de um laço neste esquema respeitando a *ordem de execução seqüencial* para o cálculo dos endereços de escritas e leituras. O tópico seguinte traz uma proposta neste sentido.

3.8 Paralelismo

Este tópico propõe uma forma de paralelizar um laço sincronizado pelo esquema em questão, respeitando a seqüencialização exigida pelo cálculo dos endereços de leitura e escrita. Supõe-se que as dependências de fluxo estão garantidas por meio da primitiva *flowbit*.

Para ilustrar a proposta, supõe-se que uma iteração consiste em um bloco básico [8], só com atribuições. Temos, portanto, um laço do tipo:

do

atr_1

atr_2

atr_3

enddo

Atr_i é uma atribuição na forma:

$$e_esq \leftarrow f(e_dir_1, e_dir_2, \ldots, e_dir_n)$$

onde:

- e_esq é o endereço onde é executada a escrita;
- $e_dir_{1..n}$ são os endereços de leitura.

Supondo que todos os acessos à memória no laço possam ser conflitantes, o cálculo dos novos endereços deve ser feito para todas as leituras e escritas do laço. Este cálculo é agrupado, para cada comando de atribuição, em uma região chamada *CNE* (cálculo de novos endereços), executada antes da atribuição. Esta região executa duas tarefas:

1. O *reendereçamento das leituras* aos novos endereços da memória auxiliar.
2. O *reendereçamento das escritas* à memória auxiliar, juntamente com a *enumeração das escritas*, já que este número representa o novo endereço da escrita.

Desta forma, uma atribuição passa a ter a seguinte semântica:

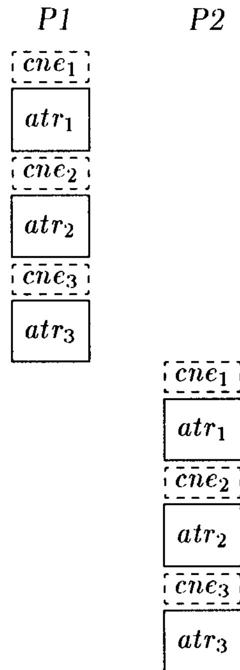
$$\left.\begin{array}{l} novo_end_leitura_{1..n} \leftarrow EMR_{e_dir_{1..n}} \\ EMR_{e_esq} \leftarrow número_da_escrita \end{array}\right\} CNE$$

$$atr$$

onde:

- *CNE* é o cálculo dos novos endereços para leituras e escritas;
- *atr* é o código original da atribuição com os endereços calculados acima.

Supondo-se, para efeito de ilustração, o uso de dois processadores (*P1* e *P2*) e que o tempo de execução do *CNE* seja metade do tempo de execução da atribuição, a figura a seguir ilustra o esquema inicial de paralelização das iterações de um laço:



O esquema de sincronização exige que o cálculo dos novos endereços seja efetuado como na ordem seqüencial de execução. Note que no esboço anterior esta ordem é garantida em duas situações:

1. *Dentro de uma iteração*, pois o esquema de execução apresentado no tópico *3.1* implica na seqüencialização de uma iteração em um processador.

2. *Entre as iterações* em processadores distintos, pela própria seqüencialização imposta neste esboço inicial.

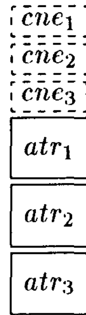
Este esboço não apresenta paralelismo na execução. Há casos muito particulares nos quais esta deve ser a forma de execução. Um destes casos pode ser exemplificado abaixo:

do $i = 1, N$
 $a_{b_i} \leftarrow \ldots$
 $\ldots$
 $b_{i+1} \leftarrow \ldots$
enddo

Neste exemplo, a seqüencialização ocorre porque o *cne* do acesso a a_{b_i} exige o *valor* de b_i para a indexação, valor este produzido pelo último comando da iteração anterior.

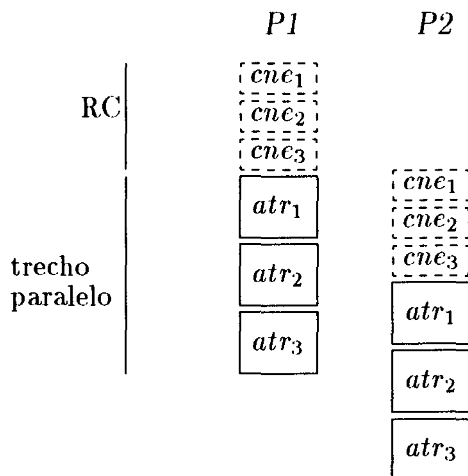
Acreditamos que existam poucos laços que se encaixam nesta condição. Um estudo mais aprofundado é necessário, porém não temos no momento o aparato experimental (simulador mais acurado, conjunto de programas científicos representativos, gerador de código FORTRAN, etc) necessário para tal estudo.

No caso geral, é possível agrupar o *cne* das atribuições em um bloco único, no início do laço. Para tanto, é feita uma inversão dos comandos de *cne* com as atribuições anteriores. Este rearranjo implica em uma iteração na seguinte forma:



Para garantir a execução paralela correta de um laço, com as iterações na forma acima, é necessário garantir que o *cne* das iterações seja executado *seqüencialmente*. Para isto, será inserida uma *região crítica* (*RC*) ordenada, que compreende o trecho entre o primeiro e o último *cne* no laço. A região crítica é ordenada porque a liberação é feita de uma iteração *i* para uma iteração consecutiva ($i + 1$).

Esta seqüencialização, imposta pela região crítica, implica no atraso do início da execução de uma iteração até que a iteração anterior termine o cálculo dos novos endereços de *todos* os seus comandos. O paralelismo é explorado pela sobreposição de uma fração das iterações, como no esboço a seguir:



Este esboço retrata a menor região crítica possível para uma iteração, dentro desta proposta de paralelismo, resultante da *união dos CNEs* para todos os comandos em que há possível conflito. Os laço a seguir ilustram a reestruturação do laço da página 40 para o esquema de sincronização, já com a exploração de paralelismo, onde *reg1*, *reg2* e *novo_endereço_leitura* são registradores auxiliares, locais a cada processador:

do $i = 1, 7$
 $reg1 \leftarrow b_i$
 $reg2 \leftarrow c_i$
 RC
  $novo_end_leitura \leftarrow EMR_{\text{endereço}(a_{reg2})}$
  $EMR_{\text{endereço}(a_{reg1})} \leftarrow i$
 endRC
 if $novo_end_leitura = 0$ /* *EMR* fora do laço */
  $AUX_i \leftarrow a_{reg2}$
 else /* *EMR* no laço */
  $AUX_i \leftarrow AUX_{novo_end_leitura}$
 endif
enddo

do $i = 1, 7$
 if $EMR_{\text{endereço}(a_{b_i})} = i$ **then**
  $a_{b_i} \leftarrow AUX_i$
 endif
enddo

O próximo capítulo desta dissertação apresenta um conjunto de melhorias que visam diminuir ainda mais a região crítica, através da *eliminação dos CNE desnecessários*, aumentando a fração de paralelismo e, conseqüentemente, o desempenho do laço paralelo.

3.9 Um Experimento

Este tópico apresenta os resultados da simulação do esquema de sincronização sobre variações do seguinte laço:

do $i = 1, 128$

$a_{b_i} \leftarrow a_{c_i}$

enddo

Determinamos que este laço é de *grão 0*. Por grão definimos toda computação excedente à atribuição $a_{b_i} \leftarrow a_{c_i}$ pertencente ao corpo do laço. O método utilizado neste experimento foi variar a granularidade do corpo do laço mediante a inserção de comandos de atribuição na forma $x_i \leftarrow y_i \times z_i$, onde cada comando representa um grão. Desta forma, laços de grão *1* e *2* são definidos como:

grão 1	grão 2
$a_{b_i} \leftarrow y_i \times z_i$	$a_{b_i} \leftarrow y_i \times z_i$
$x_i \leftarrow a_{c_i}$	$x_i \leftarrow y_i \times z_i$
	$x_i \leftarrow a_{c_i}$

O esquema, neste experimento, fará o reendereçamento de escritas e leituras apenas para os acessos ao vetor a. Esta redução do número de *CNE* executados é uma forma de otimização a ser descrita no próximo capítulo.

Para realizar este experimento foi utilizada uma nova versão do simulador, ainda em fase de documentação, que comporta uma arquitetura de memória mais completa. A configuração da arquitetura para simulação foi:

- 8 processadores;
- memória entrelaçada em 8 bancos;
- *bandwidth* do barramento: 3 acessos simultâneos;
- tempo de acesso à memória para leitura: mínimo de 4 ciclos;

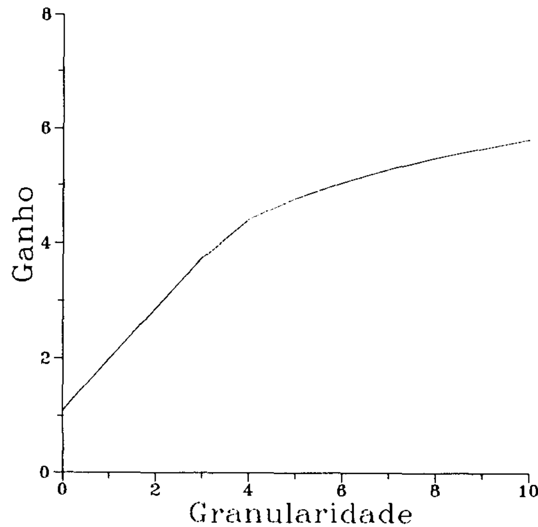
- tempo de multiplicação: 4 ciclos;
- tempo de acesso a registradores: 1 ciclo.

3.9.1 Resultados

A tabela abaixo expõe os resultados da simulação deste laço, variando-se o grão de 0 a 10. A última coluna indica o ganho de desempenho resultante da paralelização:

Grão	**Tempo Seqüencial**	**Tempo Paralelo**	**Ganho**
0	3722	3378	1.10
1	6793	3407	2.00
2	9865	3423	2.88
3	12983	3448	3.75
4	16009	3622	4.42
5	19081	4000	4.77
6	22153	4388	5.05
7	25225	4774	5.28
8	28297	5164	5.48
9	31369	5549	5.65
10	34441	5927	5.81

O gráfico a seguir ilustra a curva de ganho de desempenho em função da granularidade do laço:



O objetivo desta simulação é validar o esquema quanto a capacidade de exploração de paralelismo em um laço *seqüencializado* por esquemas tradicionais de sincronização e exploração de paralelismo (vide tópico *3.4*). Como o esquema seqüencializa apenas o cálculo dos novos endereços, eliminando anti-dependências e dependências de saída, há uma fração do laço que permite a exploração do paralelismo, resultando no ganho ilustrado pelo gráfico acima.

Os resultados foram medidos incluindo o tempo de inicialização do vetor *EMR*, o laço sincronizado e a atualização final dos endereços de memória original. Este três laços são totalmente paralelizáveis. O programa utilizado na simulação é basicamente o do exemplo da página 54, acrescido de inicialização.

O objetivo deste experimento não é avaliar o esquema quanto a eficiência da implementação utilizada. A simulação é conservadora no sentido de não contar com possível *hardware* especializado para a implementação do esquema[7]. A possibilidade de contar com *hardware* especial, como uma memória especial para *AUX* e *EMR*, ainda está sendo investigada. Acreditamos que este *hardware* aumentará o ganho, principalmente nas granularidades mais baixas.

[7]com exceção da região crítica ordenada, implementada como um barramento especial de sinalização de 1 bit entre os processadores e executada mediante instruções *send* e *receive*

Capítulo 4

Inserindo Especialização no Compilador

O capítulo anterior apresentou um esquema de sincronização de laços que preserva a semântica seqüencial, quando da paralelização da execução das iterações, *independente* de quais relações de dependência se manifestam e de quais iterações estão envolvidas.

Esta independência é conseguida através de duas características básicas do esquema de sincronização:

1. *generalidade de aplicação*: assumindo que o fluxo de controle é sincronizado apropriadamente, esta generalidade é conseguida através da "monitoração" dos endereços utilizados durante a execução;

2. *segurança*: surge da *super-estimação* das dependências, caracterizando uma abordagem *conservadora*, ou seja, o esquema assume que todos os acessos podem ser conflitantes e executa incondicionalmente o cálculo dos novos endereços para todas as leituras e escritas no laço.

Uma das principais vantagens deste esquema é que, apesar destas características, ele *não exige especialização* do compilador. Esta especialização é usualmente caracterizada, em compiladores reestruturadores, pela presença da *análise de dependências* e do *cálculo das distâncias* para inserção das primitivas de sincronização.

No entanto, para conseguir tal generalidade de aplicação com segurança, o esquema impõe computações adicionais, como o cálculo de novos endereços e a atualização da memória original, dentre outras. Esta computação gera um *custo adicional* que diminui o ganho esperado na paralelização do laço.

O objetivo deste capítulo é propor um esquema híbrido para exploração de paralelismo em laços que, por meio da inserção de especialização adequada no compilador, antecipe a análise do laço a ser paralelizado, permitindo utilizar o esquema de sincronização de forma menos conservadora.

Este capítulo não trata esta análise exaustivamente, mas indica um caminho que parece conduzir a um equilíbrio adequado entre o custo adicional do esquema e a especialização necessária ao compilador para reduzir tal custo. É necessário avaliar, experimentalmente, a gama de aplicabilidade da implementação e o benefício obtido.

4.1 Alternativas para Aumentar a Eficiência

Para facilitar o entendimento do trabalho, um laço paralelizado pelo esquema passa a ser denominado *dosync*. Há duas formas imediatas para aumentar a eficiência de laços *dosync*:

1. Inserir *hardware adicional* para a implementação do esquema, como, por exemplo, uma memória auxiliar para o reendereçamento das escritas e leituras. Propostas neste sentido não serão avaliadas neste trabalho.

2. Reduzir o custo adicional através da *eliminação* da computação de cálculo de novos endereços. Esta melhoria sugere que, ao invés de super-estimar as dependências, um laço *dosync não efetue* o cálculo dos novos endereços para os acessos à memória que *comprovadamente* sejam independentes de outros acessos.

Vejamos alguns casos particulares de (2):

- Quando *algumas* referências no laço são comprovadamente independentes. Este caso é exemplificado pelos laços de granularidade maior que zero no experimento do tópico *3.9*. Os acessos a x, y e z, se executados no corpo do programa principal, são todos independentes entre si e entre os acessos aos vetores a, b e c, dispensando o cálculo dos novos endereços para x, y e z.

  O ganho obtido advém da diminuição da região crítica, aumentando, conseqüentemente, a fração de paralelismo nas iterações. Este caso implica na execução de um *dosync* com melhorias no código gerado.

- Quando *todas* as referências forem comprovadamente independentes. Neste caso, todo o esquema de sincronização do *dosync* é *dispensável*, já que não há dependências a serem sincronizadas. Como exemplo, suponha o laço abaixo, lo-

calizado no corpo do programa principal. Este laço não possui dependências entre as iterações, pois os acessos são a vetores diferentes:

$$
\begin{array}{l}
\textbf{do } i = 1, N \\
\quad c_i \leftarrow a_i + b_i \\
\textbf{enddo}
\end{array}
$$

Laços com estas características serão explorados como laços paralelos do tipo *doall* [7], constituíndo a maior fonte de paralelismo em programas.

- Quando *há dependências*, porém previsíveis em *tempo de compilação* e com possibilidade de inserção das primitivas de sincronização. Esta situação também dispensa o uso do esquema pois as dependências são mapeadas sem a necessidade de computação adicional durante a execução. Isto ocorre tipicamente em laços com dependências de distância fixa, como no laço da página 60.

  Laços com estas características são denominados *doacross* [22]. Embora estes laços contem com esquemas próprios de execução paralela, eles não serão distinguidos, neste trabalho, como esquemas especiais, sendo tratados como *dosync*.

Os dois primeiros casos, *dosync* melhorado e *doall*, requerem especialização do compilador, visando detectar acessos independentes. Esta especialização é proposta na forma de uma *análise de dependências* tradicional, tratada na literatura por vários autores [7,8,23]. A partir desta análise, se for seguro, o compilador gera código para *doall*. Caso contrário, gera código para *dosync*.

4.2 Análise de Dependências

Conforme descrito no capítulo 2, um compilador reestruturador tradicional gera o *grafo de dependências* de um laço a partir da *análise de dependências* das variáveis utilizadas neste laço.

Esta análise é determinada a partir de relações entre os conjuntos de variáveis *consumidas*(lidas) e *produzidas*(escritas) por dois comandos no laço. Estes conjuntos são definidos [7] como:

- *IN*: conjunto de variáveis escalares ou elementos de *array* cujos *valores* são utilizados pelo comando.

- *OUT*: conjunto de variáveis cujos *valores* são alterados pelo comando.

Dados dois comandos, S_1 e S_2, com a execução de S_1 precedendo a execução de S_2, as relações de dependência do início do capítulo *2* implicam nas seguintes relações entre os conjuntos:

- se *fluxo* $\Rightarrow$ $OUT(S_1) \cap IN(S_2) \neq \emptyset$
- se *anti* $\Rightarrow$ $IN(S_1) \cap OUT(S_2) \neq \emptyset$
- se *saída* $\Rightarrow$ $OUT(S_1) \cap OUT(S_2) \neq \emptyset$

4.2.1 Limitações da Análise de Dependências Tradicional

A principal limitação da análise de dependências surge do fato dos conjuntos *IN* e *OUT* tratarem dos *valores* das variáveis, lidos ou alterados na memória.

Dado que uma variável é definida por uma quádrupla [24], constituída de:

- um nome;
- um endereço;
- um valor;
- um conjunto de atributos.

o *valor* de uma variável é determinado pelo *conteúdo de seu endereço*. Esta correspondência entre o endereço da variável e seu valor determina que a análise dos conjuntos *IN* e *OUT* está diretamente relacionada com a *análise dos endereços* utilizados durante a execução.

Como a análise de dependências tradicional é baseada em *nomes* de variáveis, esta análise será *limitada* aos casos onde os endereços utilizados na execução puderem ser determinados univocamente pelo *nome* da variável na fase de compilação. Esta limitação impede a análise de laços como o da página 38, pois os endereços referidos são função da indexação de *valores* definidos durante a execução.

Em FORTRAN, a análise de dependências é limitada a:

1. *Variáveis escalares*, de escopo global ou local.
2. *Arrays*, de escopo global ou local, quando a indexação for uma função linear do índice do laço.

Parâmetros constituem uma limitação à análise, dado que o *endereço referenciado* por um parâmetro é a rigor um *valor* que este parâmetro recebe na chamada de um procedimento, pois em FORTRAN a passagem de parâmetros é feita apenas por referência. Um exemplo desta limitação ocorre no procedimento *Vadd*, no laço da página 39. Este problema, conhecido como *aliasing*, requer uma abordagem conservadora na análise de dependências, ou seja, os acessos no laço devem ser considerados dependentes, impedindo a paralelização.

Nossa proposta é a inserção de especialização na forma de uma análise de dependências tradicional, acrescida de testes resolvidos durante a execução, para a detecção de dependências envolvendo parâmetros. O compilador, auxiliado por estes testes, deverá ser capaz de apontar uma forma segura e eficiente de explorar paralelismo em laços.

4.3 Previsão de Acessos Conflitantes

A previsão dos acessos conflitantes, resultantes da análise de dependências acrescida dos testes sobre parâmetros, será discutida em duas etapas: análise de variáveis *escalares* e análise de *arrays* unidimensionais.

Os resultados destas análises serão armazenados, para efeito de ilustração, em *tabelas de conflito*. Estas tabelas contém as informações, coletadas durante a compilação, para cada par de referências existente em um laço. Como a análise deve prever os conflitos de dependências entre duas escritas (*saída*) e entre escritas e leituras (*anti* e *fluxo*), serão definidas duas tabelas:

1. Tabela de *escrita(E)* × *escrita(E)*, para *dependências de saída*. Exemplo para um laço com duas escritas por iteração:

	E_1	E_2
E_1		
E_2	$\otimes$	

O símbolo $\otimes$ indica que esta entrada não precisa ser analisada, pela simetria entre dependências de saída em comandos de um laço. A diagonal principal da matriz indica o resultado da análise de dependências de uma escrita com relação a ela mesma. Por exemplo, a escrita a a_{b_i} em uma iteração i pode ter dependência de saída para esta mesma escrita em outra iteração. A entrada $E_1 \times E_2$ contém o resultado da análise de dependências entre as duas referências de escrita, em qualquer par de iterações.

2. Tabela de *escrita(E)* × *leitura(L)*, para *dependências de fluxo* e *anti-dependências*. Exemplo para um laço com duas escritas e quatro leituras por iteração:

	L_1	L_2	L_3	L_4
E_1				
E_2				

   Cada entrada da tabela indica o tipo de conflito existente entre as referências de leitura e escrita em qualquer par de iterações.

Note que não há análise de conflitos para duas leituras, por não originarem dependências.

4.3.1 Variáveis Escalares

Há quatro possíveis entradas nas tabelas de conflito, para variáveis escalares. As duas primeiras tratam de variáveis com endereços iguais, conhecidos na fase de compilação (globais ou locais), ou de duas referências ao mesmo parâmetro formal:

- LOCAL: indica, para duas referências à mesma variável, que o valor atribuído à variável não será utilizado em outra iteração.

  Exemplo:

  $$a \leftarrow \ldots$$
  $$\ldots \leftarrow a$$

  Este caso retrata iterações onde a escrita à variável antecede lexicamente a leitura. A entrada LOCAL propõe que a variável escalar seja alocada localmente aos processadores, ou seja, cada processador terá sua *instância da variável.* Como a execução das iterações em um processador é seqüencial, este caso não requer sincronização.

  Para preservar a semântica, ao final do laço paralelo deve-se atualizar a variável escalar com o valor da variável local do processador que executou a última iteração do laço.

- SYNC: indica, para duas referências a mesma variável, que o valor atribuído à variável em uma iteração será utilizado na iteração posterior.

  Exemplo:

$$\ldots \leftarrow a$$
$$a \leftarrow \ldots$$

Este caso implica na leitura de uma variável cujo valor foi produzido na iteração anterior e surge quando a leitura antecede lexicamente a escrita na iteração. Como já foi apontado anteriormente, laços com dependências entre iterações serão sincronizados por *dosync*.

Restam dois casos extremos: quando comprovadamente não há conflitos entre as iterações e quando é impossível, na compilação, detectar a existência de conflitos. Estes casos são indicados, respectivamente, pelas seguintes entradas:

- NOSYNC: indica que não haverá conflito entre as referências, por um dos seguintes motivos:

  1. Os endereços acessados, previsíveis na compilação, são diferentes (*nomes* diferentes para variáveis globais ou locais).
  2. Uma referência for um parâmetro e a outra for uma variável de escopo local ao procedimento. Como o parâmetro só pode endereçar uma variável externa ao procedimento, não haverá conflito.

- POSS: indica que há *possível* conflito entre as referências. É utilizada nos casos onde:

  1. Uma das referências é um parâmetro formal de um procedimento e a outra possua escopo global.
  2. Duas referências à parâmetros formais com nomes diferentes.

  Nestes casos, não é possível prever na fase de compilação se há dependências entre as duas referências, pois os endereços referidos pelos parâmetros *só são conhecidos em tempo de execução*.

  Nossa proposta é que, ao encontrar entradas POSS, o compilador gere um teste a ser *avaliado em tempo de execução*. Este teste, descrito abaixo, determinará se há conflito:

  – para o primeiro caso (um parâmetro, uma variável global):

    se endereço da variável global = endereço referido pelo parâmetro

- para o segundo caso (dois parâmetros):

  se endereço referido pelo parâmetro 1 = endereço referido pelo parâmetro 2

4.3.2 *Arrays* Unidimensionais

- SYNC: indica duas situações particulares. Na primeira, *haverá dependência* entre as iterações, quando do acesso à memória pelas referências em questão. Esta dependência é previsível porque os endereços base (nomes) dos *arrays* (variáveis ou parâmetros) são iguais e a indexação é função linear do índice do laço. Este caso detecta:

  1. Dependência de saída entre duas escritas. Exemplo:

     $$a_i \leftarrow \ldots$$
     $$a_{i+1} \leftarrow \ldots$$

  2. Dependência de fluxo ou anti-dependência entre uma escrita e uma leitura. Exemplo de dependência de fluxo:

     $$a_i \leftarrow a_{i-1} \ \ldots$$

  A segunda situação envolve referências a *arrays* cuja indexação não é função linear do índice do laço. Este caso detecta:

  1. Possível dependência de saída cíclica para uma escrita. Surge quando o índice do *array* é uma função de mapeamento *vários-para-um*[1]. Neste caso, várias instâncias da mesma referência em diferentes iterações podem fazer acesso o mesmo elemento.

     Exemplo:

     $$a_{b_i} \leftarrow \ldots$$

  2. Índices não lineares para uma leitura, havendo uma escrita ao mesmo *array*. Este caso implica em possíveis dependências de fluxo e/ou anti-dependências, definidas apenas durante a execução pelo *valor* das variáveis que compõem o índice da leitura. Exemplo:

     $$a_i \leftarrow a_{c_i} \ \ldots$$

[1] many-to-one mapping

- NOSYNC: indica que não haverá conflito entre as iterações nas referências analisadas. Ocorre em três casos particulares:

  1. Quando os *arrays* são variáveis de escopo global ou local e possuem nomes diferentes.
  2. Quando os *arrays* (variáveis ou parâmetros) possuem o mesmo nome, porém a indexação determina dependências *dentro de uma mesma iteração*. Exemplo:

     $$a_i \leftarrow \ldots$$
     $$\ldots \leftarrow a_i$$

  3. Quando uma referência for a um *array* de escopo local e a outra for a um *array* parâmetro.

- POSS: indica, da mesma forma que nas variáveis escalares, que não foi possível determinar as dependências porque as referências envolvem parâmetros, cujo endereço de referência não pode ser determinado em tempo de compilação.

  A rigor, o endereço do parâmetro efetivo determina o endereço inicial do *array* visível ao parâmetro formal. Este endereço pode corresponder ao endereço base (ex. **call** *P(a)*) ou a um endereço intermediário (ex. **call** $P(a_5)$).

  Considerando o primeiro caso, da passagem de endereços base, dadas duas referências, sendo:

  - um parâmetro e um *array* de escopo global;
  - dois parâmetros de nomes diferentes;

  o compilador inicialmente efetua uma análise da indexação destas referências. Se os índices forem iguais, não haverá dependências entre as iterações quaisquer que sejam os endereços base dos parâmetros efetivos. Este caso implica em uma entrada NOSYNC na tabela de conflito.

  Havendo possibilidade de conflito devido à diferença nos índices, o compilador gera um dos seguintes testes, a ser avaliado em tempo de execução antes do laço:

  - para uma variável e um parâmetro:

    se endereço base do array = endereço referido pelo parâmetro

- para dois parâmetros:

  se endereço referido pelo parâmetro 1 = endereço referido pelo parâmetro 2

O segundo caso, da passagem de endereços intermediários, é mais difícil de ser tratado, pois uma referência ao parâmetro formal no procedimento P deve considerar o deslocamento no parâmetro efetivo. Uma referência ao parâmetro formal com índice 1, no exemplo (**call** $P(a_5)$), acessaria o elemento a_5. Há uma proposta, ainda em estudo, para este caso. Esta proposta, que não será tratada neste trabalho, envolve a *passagem do endereço base* junto com o endereço inicial. Por exemplo, (**call** $P(a, a_5)$). Com os endereços base conhecidos, o compilador geraria testes envolvendo a *base* e o *deslocamento inicial* para determinar, durante a execução, se há dependências nas referências.

Com as tabelas de conflitos preenchidas, o compilador pode determinar a forma mais eficiente e segura de execução paralela do laço. Esta decisão é descrita no tópico a seguir.

4.4 Definição da Forma de Paralelização do Laço

Ao final da análise de dependências, as tabelas de conflitos guiam o compilador na decisão da melhor forma de execução paralela para o laço. Trataremos de três possibilidades:

1. Se há *pelo menos uma entrada* SYNC, então haverão dependências na execução, previsíveis na compilação. Neste caso, o paralelismo é explorado por *dosync* e a geração de código para as referências no laço é guiada pelo seguinte algoritmo:

   - efetua-se uma pesquisa, para cada referência individualmente, das suas entradas nas tabelas de conflito. Para uma escrita, percorre-se a linha e coluna da referência, na tabela $(E \times E)$, e a linha da referência na tabela $(E \times L)$. Para uma leitura, percorre-se na tabela $(E \times L)$ a coluna referente a leitura. Com base nestas entradas, define-se a geração de código para a referência, com três possibilidades:

     - se há pelo menos uma entrada SYNC, gera-se código de cálculo de novo endereço (*cne*) para a referência;
     - se há *apenas* entradas NOSYNC e/ou LOCAL, gera-se código normal, sem *cne*;
     - se há uma entrada POSS, gera-se *cne condicional*.

A última possibilidade, de geração de *cne condicional* determina que o *cne* será executado se pelo menos um dos testes gerados pela entrada POSS, para a referência em questão, for verdadeiro.

Os exemplos a seguir ilustram laços envolvendo as três propostas de código descritas anteriormente. Nestes exemplos, o índice do laço encaixa-se como LOCAL, dado que seu valor inicial é definido antes de qualquer leitura na iteração, e será sempre alocado localmente aos processadores. Supõe-se, para manter a coerência com o capítulo *3*, que os índices dos *arrays* não são alterados dentro do laço, com exceção do índice do laço.

Para simplificar a apresentação, ilustraremos somente o laço de reendereçamento de escritas e leituras, ignorando o laço final de atualização da memória original.

- **Exemplo 1**:

$$
\begin{array}{l}
\textbf{do } i = 1, 128 \\
\quad a_{b_i} \leftarrow x_i \times y_i \\
\quad z_i \leftarrow a_{c_i} \\
\textbf{enddo}
\end{array}
$$

onde todas as variáveis são de escopo global. As tabelas de conflitos seriam definidas por:

	a	z
a	SYNC	NOSYNC
z	$\otimes$	NOSYNC

	b	c	x	y	a
a	NOSYNC	NOSYNC	NOSYNC	NOSYNC	SYNC
z	NOSYNC	NOSYNC	NOSYNC	NOSYNC	NOSYNC

código resultante:

```
do i = 1,128
    reg1 ← b_i
    reg2 ← c_i
    RC
        EMR_endereço(a_reg1) ← i
        reg3 ← EMR_endereço(a_reg2)
    endRC
    AUX_i ← x_i × y_i
    if reg3 = 0
        z_i ← a_reg2
    else
        z_i ← AUX_reg3
    endif
enddo
```

- **Exemplo 2:**

```
do i = 1,128
    a_{b_i} ← ...
    ... ← a_{b_i}
enddo
```

tabelas de conflito[2]:

	a
a	SYNC

	a
a	NOSYNC

Este caso implicaria na execução do *cne* apenas para a escrita, já que há possível dependência de saída entre as iterações. A entrada NOSYNC para a leitura a a_{b_i} indica que não há necessidade de geração de *cne* para o acesso ao dado. Contudo, este caso retrata uma situação onde a decisão de geração de código para uma leitura *não é ortogonal* à decisão da geração de código para uma escrita. Como a escrita será reorientada a AUX, a leitura também deve ser reorientada a AUX, porém *sem a geração* de *cne* para esta leitura, já que é possível determinar na compilação que o acesso de leitura utilizará o valor armazenado pela escrita da mesma iteração.

[2]as referências indicadas por "..." serão desconsideradas para facilitar a exemplificação

código gerado:

```
do i = 1,128
    RC
        EMR_endereço(a_{b_i}) ← i
    endRC
    AUX_i ← ...
    ... ← AUX_i
enddo
```

- **Exemplo 3**:

```
subroutine P(b)
    do i = 1,N
        a_i ← ...
        a_{i+1} ← ...
        ... ← b_{i-1}
    enddo
```

onde a é um vetor de escopo global ao procedimento. As tabelas de conflito são definidas como:

	a_i	a_{i+1}
a_i	NOSYNC	SYNC
a_{i+1}	$\otimes$	NOSYNC

	b
a_i	POSS
a_{i+1}	POSS

Neste caso deve ser gerado o cálculo de novos endereços para as escritas ao vetor a, para sincronizar as dependências de saída. Além disso, é gerado um teste para a entrada POSS, que determinará a execução do *cne* para a referência de leitura b_{i-1}, já que os índices das escritas e da leitura são diferentes.

Código gerado:

$flag \leftarrow endereço\ base\ (a) = endereço\ referido\ (b)$

do $i = 1, N$
  $reg1 \leftarrow i \times 2$
  RC
    $EMR_{\text{endereço}(a_i)} \leftarrow reg1 - 1$
    $EMR_{\text{endereço}(a_{i+1})} \leftarrow reg1$
    if *flag* **then**
      $reg2 \leftarrow EMR_{\text{endereço}(b_{i-1})}$
    endif
  endRC
  $AUX_{reg1-1} \leftarrow \ldots$
  $AUX_{reg1} \leftarrow \ldots$
  if (**not** *flag*) **or** $(reg2 = 0)$ **then**
    $\ldots \leftarrow b_{i-1}$
  else
    $\ldots \leftarrow AUX_{reg2}$
  endif
enddo

2. Quando só há entradas NOSYNC e LOCAL, é gerado um *doall*, pois em tempo de compilação foi detectado que não haverá conflito nos acessos à memória. Considerando o laço da página 60, a tabela abaixo mostra o desempenho quando o laço é executado pelo *dosync* com a computação adicional de *cne* para *todas* as referências no laço e o ganho decorrente da paralelização simples, sem nenhum código adicional de sincronização, pelo *doall*:

dosync	*doall*
0.65	6.63

3. se não há entradas SYNC, porém há *pelo menos uma entrada* POSS, não é possível determinar em tempo de compilação se o laço contém dependências, já que entre as referências envolvidas nas entradas POSS pelo menos uma é parâmetro, cujo endereço de referência só é conhecido durante a execução.

   O compilador deve gerar os dois laços, *doall* e *dosync*, e determinar durante a execução, com base nos testes gerados pelas entradas POSS, qual o laço a ser

executado. Se pelo menos um destes testes resultar em conflito, executa-se o *dosync*, já melhorado pela análise de quais *cne* serão necessários. Se nenhum teste for verdadeiro (resultar em conflito), executa-se o *doall*.

Como exemplo, suponha o trecho de programa, já discutido anteriormente no capítulo *3*:

```
/* programa principal */
    ...
    call Vadd (x, y, z)
    ...
    call Vadd (x, x, z)
    ...
subroutine Vadd (a, b, c)
    do i = 1,n
        a_{i+1} ← b_i + c_i
    enddo
```

tabelas de conflito:

	a
a	NOSYNC

	b	c
a_i	POSS	POSS

código gerado:

```
flagb ← endereço referido (a) = endereço referido (b)
flagc ← endereço referido (a) = endereço referido (c)

if flagb or flagc then
    dosync com cne condicionais
else
    doall
endif
```

Para a chamada *Vadd (x, y, z)*, os endereços referidos são diferentes e executa-se o *doall*. Na segunda chamada, *Vadd (x, x, z)*, há conflito entre a e b, determinando a execução do *dosync*, com cálculo de novos endereços para as referências conflitantes. Note que a referência c não executará o *cne* pois o *flagc* determina que não haverá conflito entre a escrita a a e a leitura de c.

4.5 Considerações Finais

Com base nas limitações apontadas ao processo tradicional de análise de dependências, este capítulo indicou um esquema inicial para paralelização e sincronização de laços, que envolve a análise tradicional de dependências pelo compilador acrescida de técnicas utilizadas durante a execução para detecção e sincronização das dependências.

A efetiva utilização deste esquema requer uma análise mais rigorosa da geração de código a partir de decisões sobre as tabelas de conflito. Esta análise deverá considerar outras formas de paralelização de laços, como o *doacross*, e prever casos onde a decisão do código a ser gerado não é ortogonal a duas referências distintas, como no exemplo 2, da página 69.

Capítulo 5

Conclusões

A reestruturação de programas seqüenciais para processamento paralelo mostra-se uma das formas mais promissoras para aumentar o desempenho destes programas. Esta dissertação procurou analisar este processo de reestruturação no escopo de paralelização e sincronização, para máquinas MIMD de memória compartilhada, de laços mais internos em programas FORTRAN, dado que estes são a maior fonte de paralelismo em programas.

A validação desta reestruturação é determinada pela preservação da semântica do programa seqüencial original. Esta semântica é função da seqüência de acessos de leitura e alteração de dados na memória, imposta pelo programa seqüencial, determinando relações de dependência entre os comandos do programa. Para garantir esta semântica, o programa paralelo deve respeitar as dependências entre os acessos à memória, como na ordem de execução seqüencial.

As arquiteturas paralelas contam com primitivas de sincronização, usualmente implementadas em *hardware* especial, que, por garantir acessos indivisíveis a blocos ou endereços da memória, permitem a implementação de esquemas de sincronização que garantem as dependências impostas pelo programa seqüencial.

Estes esquemas, contudo, são normalmente baseados no controle dos acessos à memória por meio de semáforos, dissociados dos conflitos especificamente gerados pelas dependências de dados. O uso correto destes semáforos exige o cálculo da distância entre as iterações dependentes, assumindo ainda que esta distância mantém um valor fixo durante a execução do laço. Estudos mostram que este comportamento constante nem sempre ocorre, levando estes esquemas a seqüencializarem tais laços.

A partir da análise desta ineficiência, esta dissertação propôs uma primitiva de sincronização (*flowbit*), baseada no controle de acesso aos dados, que sincroniza eficientemente as dependências de fluxo em programas com atribuições únicas e múltiplas

leituras, sem exigir do compilador o cálculo de dependências ou distâncias.

A reutilização de variáveis, na forma de múltiplas escritas a um mesmo endereço, gera anti-dependências e dependências de saída entre as iterações. Estas dependências dificultam a sincronização para todas as primitivas e impedem o uso do *flowbit*.

Com base nestas dificuldades, o capítulo *3* apresentou um esquema de sincronização de laços que, através de um reendereçamento das leituras e escritas durante a execução, elimina as anti-dependências e dependências de saída em laços, colocando-os na condição de atribuições únicas, propícios para o uso da primitiva *flowbit*. A reestruturação de laços por este esquema preserva a semântica do programa original, pois garante que os endereços alterados no programa seqüencial possuem os mesmos valores, quando da execução paralela.

Este esquema, contudo, impõe um custo adicional que diminui o ganho de desempenho resultante da paralelização do laço. Este custo surge basicamente do reendereçamento seqüencial das escritas e leituras no laço, implicando em uma região crítica entre as iterações. Acreditamos que, com um suporte mais eficiente de simulação de arquiteturas paralelas, seja possível medir com mais precisão este custo e indicar *hardware* especial que o diminua.

No entanto, este custo surge porque o esquema, por não exigir nenhuma especialização do compilador, é conservador e assume que todas as referências podem ser conflitantes na execução paralela do laço. Esta super-estimação das dependências implica no reendereçamento de todas as referências no laço.

Acreditamos que, por meio de uma combinação da análise de dependências tradicional com o esquema de sincronização atuando durante a execução, seja possível determinar uma forma mais eficiente e segura de execução paralela. O aumento da eficiência ocorre da eliminação do custo adicional de reendereçamento das referências cuja independência no acesso à memória possa ser comprovada na compilação, reiterando que apenas as dependências entre iterações devem ser sincronizadas.

Se este processo determinar, na fase de compilação, independência para todas as referências no laço, paraleliza-se o laço normalmente, sem código de sincronização. Caso contrário, aplica-se o esquema de sincronização durante a execução do laço, porém dispensando o reendereçamento para referências independentes.

Por fim, o capítulo *4* mostrou que a análise de dependências tradicional é limitada aos casos onde os endereços das variáveis analisadas são conhecidos durante a compilação e a indexação de *arrays* é função linear do índice do laço. Em casos como o uso de parâmetros, onde os endereços só são conhecidos na execução, propõe-se a geração de testes que, durante a execução, determinem a presença de endereços conflitantes. A

decisão final do laço paralelo a ser executado resultará destes testes.

Trabalhos futuros incluem uma análise da utilização do esquema de sincronização (*dosync*), quando aplicado a um conjunto de programas científicos representativos. Para tanto, além da seleção dos programas, faz-se necessário a construção de ferramentas de simulação e análise de desempenho mais realísticas.

Outra frente de trabalho imediata é a definição de uma proposta final de especialização no compilador, a partir do trabalho inicial apresentado no capítulo *4*. Faz-se necessário uma análise mais rigorosa da geração de código paralelo a partir de decisões sobre as tabelas de conflito, considerando outras formas de paralelização do laço, como *doacross*.

Bibliografia

[1] Michael J. Flynn, *Some Computers Organizations and Their Effectiveness*, IEEE Transactions on Computers, **C-21**, 2, 1972

[2] Alan H. Karp, *Programming for Parallelism*, IEEE Computer, maio 1987

[3] David J. Kuck et al., *The Effects of Program Reestructuring, Algorithm Change and Architecture Choice on Program Performance*, Proceedings of the 1984 International Conference on Parallel Processing, IEEE Computer Society Press, agosto 1984

[4] Donald E. Knuth, *An Empirical Study of FORTRAN Programs*, Software – Practice and Experience, **1**, 1, 1971

[5] *Alliant Product Summary*, Alliant Computer System Corporation, janeiro 1985

[6] Eduardo Voigt, *CP, Um Simulador de Paralelismo. Manual do Usuário*, Documento Interno, Instituto de Estudos Avançados (IEAv), CTA, novembro 1990

[7] Michael Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989

[8] Alfred V. Aho, Ravi Sethi e Jeffrey D. Ullmann, *Compilers Principles, Techniques and Tools*, Addison Wesley, 1986

[9] Edsger W. Djikstra, *The Structure of the "THE"-Multiprogramming System*, Communications of the ACM, **11**, 5, 1968

[10] Anita Osterhaug, *editor*, *Guide to Parallel Programming on Sequent Computer Systems*, Prentice Hall, 1989

[11] Harry F. Jordan, *HEP Architecture, Programming and Performance*, Parallel MIMD Computation: The HEP Supercomputer and its Applications, MIT Press, 1985

[12] Burton Smith, *The Architecture of the HEP*, Parallel MIMD Computation: The HEP Supercomputer and its Applications, MIT Press, 1985

[13] Allan Gottlieb et al., *The NYU Ultracomputer, Designing an MIMD Shared Memory Parallel Computer*, IEEE Transactions on Computers, **C-32**, 2, 1983

[14] Frederica Darema, *Applications Environment for the IBM Research Parallel Processor Prototype (RP3)*, Proceedings of the First International Conference on Supercomputing, Grécia 1987

[15] David J. Kuck, Edward S. Davidson, Duncan H. Lawrie e Ahmed H. Sameh, *Parallel Supercomputing Today and the Cedar Approach*, Science, **231**, fevereiro 1986

[16] David Pointer e Greg Jaxon, *Cedar Synchronization Processor Instruction Set Reference*, Relatório Interno, Center of Supercomputing Research and Development (CSRD), Urbana, Illinois, julho 1990

[17] *Concurrent FORTRAN Programming Manual*, Alliant Computer System Corporation, novembro 1984

[18] Allan Gottlieb e Clyde P. Kruskal, *Coordinating Parallel Processors: A Partial Unification*, Computing Architecture News, outubro 1982

[19] Samuel P. Midkiff e David A. Padua, *Compiler Algorithms for Synchronization*, IEEE Transactions on Computers, **C-36**, 12, 1987

[20] Z. Shen, Z. Li e Pen-Chung Yew, *An Empirical Study of FORTRAN Programs for Parallelizing Compilers*, IEEE Transactions on Parallel and Distributed Computers, **1**, 3, 1990

[21] Peiyi Tang, Pen-Chung Yew e Chuan-Qi Zhu, *Compilers Techniques for Data Synchronization in Nested Parallel Loops*, ACM International Conference on Supercomputing, 1990

[22] Ron G. Cytron, *Compile-Time Scheduling and Optimization For Asynchronous Machines* , Tese de Doutorado, Center of Supercomputing Research and Development (CSRD), Urbana, Illinois, outubro 1984

[23] Constantine D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988

[24] Ellis Horowitz, *Fundamentals of Programming Languages*, Springer-Verlag, 1984

[25] Gerry Kane, *MIPS R2000 RISC Architecture*, Prentice Hall, 1987