Primitivas de Sincronização para Máquinas Paralelas de Memória Compartilhada

Eduardo Voigt*
Jairo Panetta†

Instituto de Estudos Avançados (IEAv-CTA)
Rodovia dos Tamoios Km 5,5 Caixa Postal 6044
CEP 12231 São José dos Campos - SP
Tel: (0123) 413033 Ramais: 367 e 348
e-mail: CTA@BRFAPESP.BITNET

RESUMO

Este artigo critica o uso de semáforos como primitivas para sincronização de laços com dependências de dados em máquinas paralelas de memória compartilhada. Esta crítica é fundamentada na dificuldade, e por vezes impossibilidade, imposta ao compilador para o cálculo das distâncias entre iterações dependentes, além do custo adicional da manipulação dos semáforos.

São apresentadas vantagens no uso de primitivas voltadas à referência dos dados na memória, como a *full/empty bit*, que não exige o cálculo das distâncias em certos laços paralelos, dispensando a especialização do compilador. O artigo culmina com a proposta de uma simplificação da *full/empty bit*, adequada para programas com atribuições únicas.

ABSTRACT

This article criticizes the use of semaphores as synchronization primitives for parallel loops with data dependences, running on shared-memory multiprocessors. This criticism is based on the difficulty, and sometimes impossibility, imposed to the compiler in the calculation of distances between dependent iterations, besides the high cost of semaphore manipulation.

The article presents some advantages on the use of data oriented primitives, like the *full/empty bit*. For certain parallel loops, this primitive does not require the distance definition, minimizing compiler specialization. At last, we propose a new synchronization primitive, suited for single-assignment programs.

---

*MsC, UNICAMP.
†PhD, Purdue University.

1 INTRODUÇÃO

O principal objetivo de um compilador, quando da reestruturação de laços seqüenciais para processamento paralelo, 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 uma dependência entre o compilador e a 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 soluções oriundas de problemas de sincronização em sistemas operacionais e, como tal, são baseadas em eventos dissociados dos conflitos especificamente gerados pelas dependências dos dados. Semáforos[1] 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 da classe *wait/signal*(P e V)[1], *test-and-set*(TAS)[2] e, até certo ponto, *await/advance*[3]. Uma exceção a esta regra é a primitiva *full/empty bit*[4,5], voltada para referências aos dados na memória.

Este artigo aponta algumas desvantagens no uso de primitivas baseadas em semáforos, tais como a *TAS*, com relação ao uso de primitivas voltadas para referência aos dados, tais como a *full/empty bit*. Estas desvantagens concentram-se na dificuldade imposta ao compilador para o cálculo das distâncias entre iterações dependentes. Por muitas vezes, como será apresentado no tópico **3**, este cálculo não é passível de ser efetuado durante a compilação, forçando a seqüencialização dos laços quando a sincronização for efetuada por semáforos. Outra deficiência deste paradigma é o custo adicional da manipulação dos semáforos.

Será mostrado que, para certos laços paralelos, a utilização de uma primitiva como a *full/empty bit* garante a sincronização correta, dispensando a especialização necessária ao compilador para o cálculo das distâncias. O artigo culmina com a proposta de uma nova primitiva, baseada em uma simplificação da *full/empty bit*, adequada para programas com atribuições únicas.

O trabalho limita-se ao estudo da exploração de paralelismo de granularidade média em *laços mais internos*[1], dado que estes laços são a fonte primária de paralelismo em programas [6]. O estudo será restrito a máquinas MIMD de memória central. A exploração de paralelismo dar-se-á pela execução simultânea de iterações distintas de um laço em processadores distintos, onde cada iteração executa seqüencialmente em um processador. Esta proposta é adotada por máquinas como o Alliant FX/8 [3].

O tópico **2** introduz dependências de dados. O tópico **3** descreve algumas primitivas de sincronização, apresenta limitações na inserção destas primitivas em programas com atribuições únicas e propõe uma nova primitiva. O tópico **4** apresenta um experimento que avalia a eficiência das primitivas. O tópico **5** conclui o trabalho.

[1] Inner loops

2 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, neste trabalho, será restrita a *blocos básicos*. 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 $C_1\ \delta\ C_2$.
- **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 determinam um caminho crítico (de acessos à memória) em um grafo de dependências de dados entre os comandos do programa. Esta ordem seqüencial de acessos deve ser mantida pelo programa paralelo. A alteração desta ordem implica na modificação da semântica do programa.

Um compilador reestruturador utiliza estas relações 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. Esta alteração traduz-se em possibilidade de paralelização por compiladores reestruturadores.

3 PRIMITIVAS DE SINCRONIZAÇÃO

Para garantir a execução exclusiva dos trechos com dependências de dados, os compiladores reestruturadores alteram o programa original, inserindo *código especial de sincronização*. As operações que compõem este código denominam-se *primitivas de sincronização*.

Como exemplo clássico de primitivas, podemos citar as operações sobre *semáforos*[1]. Um semáforo é uma variável inteira não negativa na qual atuam duas operações básicas: **P** e **V**[2].

Dado um semáforo s, a semântica das primitivas é:

```
P(s):   label: if s > 0 then          V(s):   s ← s + 1
                   s ← s − 1
               else
                   goto label
```

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.

[2]Estas primitivas são também denominadas WAIT e SIGNAL

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 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 compartilhada.

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.

3.1 Paradigmas de Utilização das Primitivas

Os tópicos seguintes apresentam primitivas de sincronização implementadas em máquinas paralelas MIMD de memória compartilhada. Estas primitivas, que obedecem às duas premissas básicas acima, podem ser englobadas em dois paradigmas distintos:

1. **Primitivas baseadas no 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 Sequent [2] e Alliant FX/8 [3], e envolve as primitivas *test-and-set* e o par *await/advance*, dentre outras.

2. **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*) [4,5], que associa a cada posição de memória um bit de controle de acesso.

3.2 *Test-and-set*

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

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

A *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 a *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.

O multiprocessador Sequent [2] 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.

3.3 *Await/Advance*

As primitivas *await/advance* foram propostas como forma básica de sincronização no mini-supercomputador Alliant FX/8 [3]. Sua finalidade é sincronizar dependência de dados entre as iterações em laços paralelizados automaticamente pelo compilador FORTRAN do Alliant [8].

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 ≤ i − d do;
}

ADVANCE (cs_n)
{
    while cs_n ≠ i do;
    cs_n ← cs_n + 1
}
```

onde:

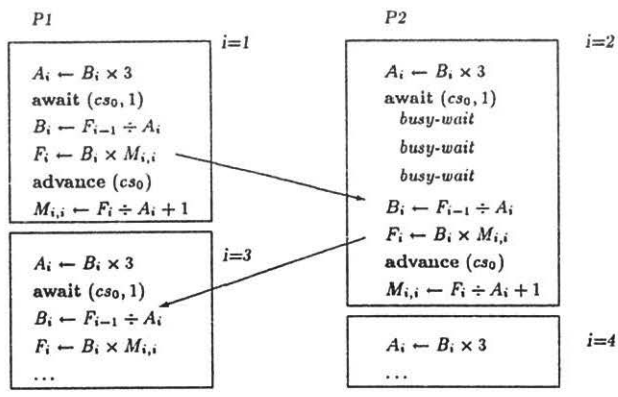
- cs_n é um semáforo compartilhado;
- i é a iteração corrente no processador que invoca *await* ou *advance*;
- d é a distância entre as iterações dependentes.

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.

3.4 Limitações à Sincronização por Semáforos

Por simplicidade de apresentação e para ordenar o raciocínio, este trabalho restringe a análise da sincronização à programas com *atribuições únicas*[3]. 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.*

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 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.

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*, vide tópico **3.2**). Isto ocorre porque um semáforo manipulado pela *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 a seguir:

[3] single-assignment

```
do i = 1, N
   while TAS(s_i) = 1 do;
   A_i = A_{i-1} + 1    (região crítica ordenada)
   s_{i+1} ← 0;
enddo
```

Supondo-se que os elementos do vetor s sejam inicialmente iguais a *1* e que $s_1 = 0$, a execução de uma iteração fica bloqueada até que a iteração anterior libere a região crítica, mediante a atribuição a s_{i+1}. 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 comandos e iterações de origem e destino das dependências. A rigor, a inserção das primitivas de sincronização exige o cálculo da *distância* entre as iterações dependentes. Como foi visto anteriormente, a distância é definida como a diferença entre as iterações destino e origem da dependência. No exemplo acima, a distância é 1.

No entanto, o mapeamento correto entre os pontos de origem e destino das dependências de dados é indecidível, *na fase de compilação*, 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.

Um estudo empírico [9] recente 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 cerca de 86% das dependências não serem constantes foi justamente a não linearidade dos índices dos *arrays*.

Portanto, o ponto crítico da sincronização por semáforos 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_{2i}, com $1 \leq i \leq 5$ devem ser inicializados com *0* (região crítica liberada) e os restantes com *1*.

O Alliant[3] 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, 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.

3.5 *Full/empty bit*

O HEP [4,5] 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:

$$\begin{aligned}
&\$\text{LOAD } (soma, reg_1)\\
&\text{LOAD } (soma_local, reg_2)\\
&reg_1 \leftarrow reg_1 + reg_2\\
&\$\text{STORE } (reg_1, soma)
\end{aligned}$$

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.

A sincronização pela *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 no tópico seguinte.

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.

3.6 Limitações à 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 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 e aumentando conseqüentemente o custo computacional devido à 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:

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

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.

3.7 A Primitiva *Flowbit*

Devido à incapacidade da *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*[10], baseada em uma simplificação da *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.

É importante observar que a primitiva #LOAD é semanticamente equivalente ao par $LOAD - FILL. No entanto, a *flowbit* sugere uma implementação mais barata e eficiente, além de possuir um custo operacional menor que a *full/empty bit*. Este ganho será verificado em um experimento no tópico seguinte, que faz uma comparação entre a TAS, a *full/empty bit* e a *flowbit*.

O compilador, com o uso da *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 a *full/empty* em programas com leituras únicas, surge agora em programas com múltiplas leituras e atribuições únicas.

4 UM EXPERIMENTO

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:

$g_1 \leftarrow dc_1$
$y_1 \leftarrow r_1$
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 : $y_i \leftarrow r_i + f_i \times y_{i-1}$
enddo

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:

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

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 [11] com a seguinte configuração:

- 4 processadores;
- memória entrelaçada[4] em 8 bancos;
- *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 [12]. 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

[4] interleaved

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 **3.6**, 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.

5 COMENTÁRIOS FINAIS

A crítica deste artigo ao uso de semáforos como primitivas de sincronização baseia-se no fato de que estas primitivas não são orientadas à sincronização de regiões críticas ordenadas e, para tal, exigem do compilador o cálculo das distâncias entre as iterações dependentes, e exigem ainda que esta distância seja fixa e conhecida durante a compilação. Estudos empíricos mostram que apenas uma pequena parte das distâncias em laços com dependências são fixas e conhecidas durante a compilação, limitando o uso de primitivas baseadas em semáforos.

Outro fator limitante é o custo e dificuldade na inserção destas primitivas, dado a necessidade de manipulação de vetores de semáforos e mapeamento dos índices às distâncias entre as iterações dependentes.

As primitivas baseadas nos acessos aos dados, como a *full/empty bit*, sobrepõem estas dificuldades por sincronizar diretamente na leitura e escrita de um dado na memória, dispensando o cálculo da distância. A principal limitação desta primitiva é que, para a sincronização correta, a ordem de acessos deve obedecer a uma seqüencia $STORE – $LOAD – $STORE – $LOAD – .... A alteração desta ordem pode levar o programa paralelo a um estado de *starvation*.

A primitiva *flowbit* aponta como uma solução potencialmente eficiente para programas com atribuições únicas e múltiplas leituras, por não exigir esta ordem de acessos para a sincronização correta, além de sugerir uma implementação mais barata.

Programas com atribuições únicas, contudo, não são o caso geral, dada a predominância de programas escritos em linguagens imperativas, como FORTRAN e C. No entanto, com certas restrições, é possível transformar laços seqüenciais com múltiplas atribuições em laços paralelos com atribuições únicas durante a execução do programa, habilitando a sincronização de tais laços com a *flowbit*. Tal reestruturação é descrita em uma publicação adicional[10].

Referências

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

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

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

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

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

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

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

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

[9] 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

[10] Eduardo Voigt, *Paralelismo e Sincronização em Laços* , Dissertação de Mestrado, UNICAMP, 1991

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

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