Transformando Laços Seqüenciais em Laços Paralelos com Atribuição Única Durante a Execução

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

Laços mais internos são fontes promissoras para a exploração de paralelismo em 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 neste artigo um esquema inédito 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 (*flowbit*), que, quando aplicada a laços com atribuições únicas, dispensa a detecção das dependências e o cálculo das distâncias para a sincronização correta.

ABSTRACT

Inner loops are a major source of parallelism in programs. Parallelism exploitation in such loops requires data dependence analysis for the insertion of proper synchronization to enforce correct execution.

Motivated by the difficulty and defficiency of current reestructuring compilers, on the detection and insertion of synchronization primitives, this work proposes a new parallelization and synchronization scheme that eliminates anti- and ouput-dependences at execution time, resulting in single-assignment loops. The remaining flow-dependences are enforced by a proper synchronization primitive (*flowbit*), that does not require data dependence and distance computation for correct synchronization of single-assignment programs.

---

*MsC, UNICAMP.
†PhD, Purdue University.

1 INTRODUÇÃO

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*[1] 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.

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.

Este artigo apresenta um esquema inédito de paralelização de laços mais internos, que elimina os problemas acima citados por meio de um reendereçamento[1], durante a execução do laço, das leituras e escritas conflitantes, em contraste com a técnica de mudança de nomes[2][2], executada durante a compilação.

Esta reestruturação eliminará dependências de saída e anti-dependências através da geração de um endereço individual para cada uma das escritas conflitantes, 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(*flowbit*[3]), resultante de modificações na *full/empty bit*[4]. Esta sincronização, diferentemente dos esquemas tradicionais, dispensa a detecção e cálculo de distâncias entre os pontos de sincronização.

O esquema compara favoravelmente com esquemas similares, tais como [5] e [6], por impedir excesso de sequencialização, como em [5] e evitar alocação dinâmica de memória, como em [6]. Ambos os esquemas exigem ainda a determinação do padrão de acesso a cada posição de memória referida no laço. Tal algoritmo, além de ter um custo de execução relativamente alto, nem sempre pode ser paralelizado.

O item 2 introduz um esquema de execução paralela de laços. O item 3 apresenta as condições suficientes para a preservação da semântica na paralelização de um programa e traz alguns problemas na detecção e exploração do paralelismo em laços. O item 4 apresenta o esquema de reestruturação e sincronização de laços mais internos. O item 5 faz alguns comentários sobre o esquema, finalizando o trabalho com um experimento no item 6.

2 ESQUEMA DE EXECUÇÃO PARALELA

Este trabalho limita-se ao estudo da paralelização de laços mais internos, ao nível de granularidade média, em máquinas paralelas com memória central. 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.

---

[1] readdressing

[2] renaming

Exemplo:

```
do i = 1,6
   a_i ← y_i × z_i
   y_i ← a_{i+1}
enddo
```

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[3] decorrente da paralelização do laço é, em tese, igual a *nproc*. 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[4], conflitos no barramento de dados e na memória devido a requisições de acesso paralelas, dentre outras.

3 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

$$f : \textit{memória} \rightarrow \textit{memória}$$

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

$$g : \textit{endereços\_leitura} \rightarrow \textit{endereço\_escrita}$$

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] speed-up

[4] 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[5], garantindo a condição 3'. Os tópicos seguintes limitam-se ao estudo de como garantir as condições 1' e 2'.

3.1 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.

Admita que as dependências de fluxo são sincronizadas pela primitiva *flowbit*[3]. Esta primitiva garante, em programas com atribuições únicas, que leituras a um endereço ocorrerão somente após a escrita ao endereço correspondente. Com as dependências de fluxo sincronizadas, garantem-se as condições 1' e 2', pois:

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.

3.2 DIFICULDADES PARA GARANTIR A SEMÂNTICA EM PROGRAMAS COM MÚLTIPLAS ATRIBUIÇÕES

ANTI-DEPENDÊNCIAS E DEPENDÊNCIAS DE SAÍDA

A reatribuição de variáveis em um programa gera anti-dependências e dependências de saída, aumentando a complexidade do compilador na detecção de dependências e inserção de sincronização. No esquema de execução paralela em questão é possível garantir a execução correta do programa em duas situações particulares:

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

- Em trechos 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*.

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.

A inserção das primitivas de sincronização, pelos compiladores reestruturadores atuais, exigem o cálculo das distâncias entre as iterações dependentes e ainda que estas distâncias sejam fixas e conhecidas durante a compilação. Quando esta distância é variável ou desconhecida, estas propostas forçam a *execução seqüencial* dos laços nas máquinas paralelas. Um estudo empírico recente [7] 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

```
        do i = 1, N
c1 :        a_{b_i} ← a_{c_i}
        enddo
```

Dado que b_i e c_i são vetores cujos valores são computados no próprio programa que contém o laço, os elementos do vetor a que receberão acessos 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.

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 existam 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[6] 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 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.

Tais limitações 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.

4 UMA PROPOSTA DE REESTRUTURAÇÃO DE LAÇOS SEQÜENCIAIS

Com o objetivo de transpor as dificuldades apontadas no tópico anterior, propomos um esquema de reestruturação de laços seqüenciais para laços paralelos que atua durante a execução para resolver acessos conflitantes à memória. Este esquema coloca o laço na condição de atribuições únicas. Sincronizando as dependências de fluxo remanescentes com a primitiva *flowbit*, é possível garantir que a semântica do laço seqüencial será preservada, como foi visto no tópico 3.1.

Esta reestruturação compreeende duas fases principais, de **reendereçamento das escritas** e **reendereçamento das leituras**. A descrição das fases da reestruturação será exemplificada, passo a passo, sob o laço abaixo, doravante denominado *laço exemplo*:

```
do i = 1, N
   a_{b_i} ← a_{c_i}
enddo
```

para o qual, assume-se N=7 e 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

4.1 REENDEREÇAMENTO DE ESCRITAS

Esta fase efetuará a paralelização das escritas conflitantes, eliminando dependências de saída e anti-dependências, através da geração de um endereço individual para cada escrita conflitante no laço. Esta fase é dividida em três passos:

[6] bound

- **Linearização das escritas:** enumera todas as escritas do laço, identificando qual é a última escrita, na ordem de execução seqüencial, a cada endereço de memória alterado no laço.
- **Paralelização das escritas:** reorienta as escritas conflitantes a endereços individuais de uma memória auxiliar, eliminando dependências de saída e anti-dependências.
- **Atualização da memória original:** atualiza as posições de memória originais com os mesmos valores contidos nos endereços auxiliares.

LINEARIZAÇÃO DAS ESCRITAS

A identificação da última escrita a cada endereço de memória alterado no laço será feita através da *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: No laço exemplo, 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 exemplo, 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.*
- Como foi visto no tópico 3, a preservação da semântica implica que, ao final da execução paralela, os mesmos endereços contenham os mesmos valores que na execução seqüencial. Portanto, deve-se

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

Esta condição não obriga que os valores computados pelos comandos sejam armazenados na memória, *a não ser o último valor* computado para cada endereço. Reiteramos 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*:
$EMR_{\text{endereço\_escrita}} \leftarrow$ *número_da_escrita_no_laço*
$AUX_{\text{número\_da\_escrita\_no\_laço}} \leftarrow$ *valor*

O laço abaixo ilustra a transformação das escritas no laço exemplo:

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

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 que o laço paralelo preserve a semântica do laço seqüencial, no que se refere as escritas conflitantes, 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 anterior, a atualização dos endereços finais do vetor a, no laço exemplo, seria:

$a_1 \leftarrow AUX_7$
$a_2 \leftarrow AUX_2$
$a_4 \leftarrow AUX_5$
$a_6 \leftarrow AUX_6$

4.2 REENDEREÇAMENTO DE LEITURAS

A rigor, as leituras devem utilizar o valor armazenado pela *escrita mais recente* a um 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*:
$\quad$ *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 exemplo:

```
do i = 1,7
   novo_end_leitura ← EMR_endereço(a_ei)
   EMR_endereço(a_bi) ← i
   if novo_end_leitura = 0 then          /* EMR fora do laço */
      AUX_i ← a_ei
   else                                  /* EMR no laço */
      AUX_i ← AUX_novo_end_leitura
   endif
enddo

do i = 1,7
   if EMR_endereço(a_bi) = i then
      a_bi ← AUX_i
   endif
enddo
```

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_3}, 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 a seguir.

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 um determinado elemento do vetor b fosse alterado no laço pela referência b_{c_i}, após sua leitura em a_{b_i}, não seria possível obter seu valor original na atualização da memória (if $EMR_{\text{endereço}(a_{b_i})} = i$). Logo, esta proposta é restrita a laços onde os índices de vetores não são alterados no laço.

4.3 PARALELIZAÇÃO DO ESQUEMA

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:

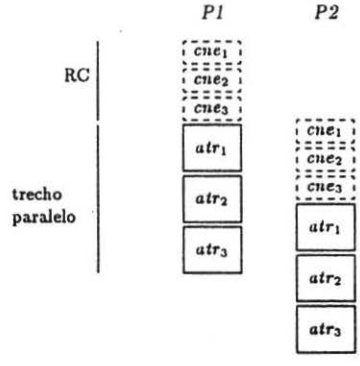
- *CNE* é o cálculo dos novos endereços para leituras e escritas;
- *atr* é o código original da atribuição com os endereços calculados acima.

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 uma iteração reestruturada para o esquema:

Com poucas restrições [3], 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 exemplo 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. Assume-se que as dependências de fluxo serão sincronizadas pela primitiva *flowbit*.

```
do i = 1,7
   reg1 ← b_i
   reg2 ← c_i
   RC
      novo_end_leitura ← EMR_endereço(a_reg2)
      EMR_endereço(a_reg1) ← i
   endRC
   if novo_end_leitura = 0          /* EMR fora do laço */
      AUX_i ← a_reg2
   else                             /* EMR no laço */
      AUX_i ← AUX_novo_end_leitura
   endif
enddo

do i = 1,7
   if EMR_endereço(a_bi) = i then
      a_bi ← AUX_i
   endif
enddo
```

5 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 eficiente, dado que:

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*. Como exemplo, todos os acessos das iterações *1, 2, 3, 5, 6* e *7* do laço exemplo 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 por sincronização apropriada.

2. A inserção da primitiva de sincronização no 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 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 **3.2**

Acreditamos que este esquema seja mais eficiente que esquemas com o mesmo propósito, tais como [5] e [6]. A proposta de Yew at al.[5] implica em excesso de sequencialização, por sincronizar todas as

dependências de saída e anti-dependências. A proposta de V.P.Krothapalli e P.Sadayappan [6] implica no alto custo de alocação dinâmica de memória para a eliminação de dependências de saída. É importante mencionar que nenhuma destas propostas trata o problema de *aliasing*.

Ambos os esquemas utilizam um algoritmo para determinação do padrão de acessos a variáveis no laço. Em particular, a proposta de [6] utiliza um algoritmo paralelo que faz uso da primitiva *FAA*[8] para determinação do número e ordem de acesso às posições de memória. O fato de um cálculo determinístico, como o cômputo da ordem de acessos, ser efetuado com uma primitiva não determinística, como a *FAA*, pode inibir a paralelização de tal algoritmo.

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 durante a 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.

6 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} ← 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 $x_i \leftarrow y_i \times x_i$, onde cada comando representa um grão. Desta forma, laços de grão *1* e *2* são definidos como:

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

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 descrita em [3].

Para realizar este experimento, foi utilizado um simulador[9], com a seguinte configuração:

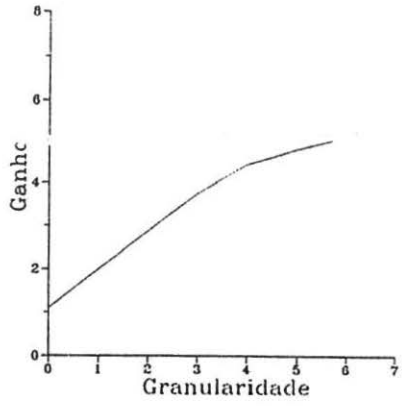
- 8 processadores;
- memória entrelaçada em 8 bancos;
- *bandwidth* do barramento: 3 acessos simultâneos;
- tempo de acesso à memória para leitura: mínimo de 4 ciclos;
- tempo de multiplicação: 4 ciclos; tempo de acesso a registradores: 1 ciclo.

6.1 Resultados

A tabela abaixo expõe os resultados da simulação deste laço, variando-se o grão de 0 a 7. 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

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. 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 resultante da reestruturação do laço exemplo, 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 imple-

mentação 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.

Referências

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

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

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

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

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

[6] V.P Krothapalli e P.Sadayappan, *An Approach to Synchronization for Parallel Computing*, Proceedings of ACM International Conference on Supercomputing, 1988

[7] 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

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

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

[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*