Avaliando Distribuições de Dados

Carlos H. B. da Silva[1]
Glauce H. S. Larangeira[2]
Paula Marisa da C. P. F. Maciel[3]
Jairo Panetta[4]

Sumário

Apresentamos uma ferramenta que avalia distribuições de dados em programas para máquinas de memória distribuída, por meio de simulações de sua execução. Argumentamos a importância da distribuição na eficiência da computação, fruto de seu impacto no escalonamento dos processadores e no volume de trocas de mensagens. Concluimos que a ferramenta, embora modesta, é extremamente útil para testar distribuições de dados.

Abstract

We present a tool that evaluates data distributions on programs for distributed memory parallel machines by simulating their execution. We argue that data distribution is central to computer efficiency, due to its importance on processor scheduling and message passing volume. We conclude that the tool, although modest, is very usefull on data distribution testing.

[1] Bacharel em Computação pela UFF
[2] Bacharel em Computação pela UFF
[3] Professora Assistente, Depto. de Computação, Universidade Federal Fluminense.
E-mail: paula@ufrj.cos.br
[4] Pesquisador, Instituto de Estudos Avançados, Centro Técnico Aeroespacial.
E-mail: panetta@ieav.cta.br

1. Introdução

Máquinas paralelas de memória distribuída são hoje uma realidade industrial. Fabricantes renomados produzem máquinas dessa classe com centenas e milhares de processadores, apresentando uma relação preço-benefício extremamente favorável. Entretanto, programar eficientemente tais máquinas ainda é uma tarefa difícil.

Uma dificuldade intrínseca à programação dessas máquinas é a distribuição de dados pelas memórias dos processadores. Essa obrigação existe apenas no modelo de memória distribuída, devido à visibilidade restrita dos dados (cada processador alcança, por seus próprios meios, apenas os dados contidos em sua memória, ou seja, em seu espaço de endereçamento) e acarreta pelo menos dois efeitos no programa. Primeiro, insere comunicação entre os processadores (por meio de comandos *send/receive*) quando um processador necessita de dados que não estão em sua memória, aumentando a possibilidade de erros no programa. Segundo, induz um escalonamento de comandos, que geralmente segue uma regra conhecida como *owner computes*, com forte influência na eficiência do programa: cada processador realiza apenas as atribuições que atualizam seus próprios objetos.

Conseqüentemente, a distribuição de dados interfere tanto na facilidade de programar essa classe de máquinas quanto na sua eficiência. Embora estes fatos fossem sobejamente conhecidos dos desenvolvedores de algoritmos para essa classe de máquinas, sua disseminação entre programadores ocorreu recentemente, quando tornou-se viável programar máquinas de memória distribuída utilizando-se um único espaço de endereçamento. Neste caso, a distribuição de dados deve ser explicitada e seu efeito é obvio.

Em conseqüência, encontram-se na literatura recente múltiplas propostas de ferramentas e métodos que auxiliam o programador a escolher uma distribuição de dados ([1], [2], [3]). É nesse contexto que propomos uma ferramenta que permite avaliar distribuições de forma extremamente simples e eficaz. Trata-se de um simulador que recebe na entrada um programa contendo um aninhamento de laços e uma distribuição dos objetos desses laços e produz o tempo de execução em cada processador e a fatia deste tempo devida à troca de mensagens. Desta forma, é possível rapidamente implementar e avaliar distribuições.

Este trabalho introduz a programação de máquinas de memória distribuída por espaço de endereçamento único (seção 2), descreve a ferramenta (seção 3), apresenta uma formulação para determinar distribuições livres de comunicação (seção 4) e mostra a utilidade da ferramenta ao avaliar algumas distribuições de dados (seção 5). Os exemplos permitem observar a utilidade da ferramenta e a importância da distribuição dos dados na eficiência do programa.

2 Programando em Espaço Único de Endereçamento

O exemplo a seguir monstra como programar máquinas de memória distribuída utilizando espaço único de endereçamento, bem como as implicações desse mecanismo na eficiência da execução.

2.1 Expressando Distribuições

Suponha que o fragmento de programa seqüencial:

```
real a(8,8)
real b(8,8)
do i = 1,8
   do j = 4,8
      a(i,j) = b(i,j-1)
   end do
end do
```

deva ser executado em uma máquina paralela de memória distribuída. A tarefa inicial é distribuir os dados pelas memórias dos processadores. Suponha que cada coluna das matrizes a e b seja enviada a um processador. Para indicar essa distribuição dos dados, são necessários dois passos. Inicialmente, decompor estruturadamente os objetos, utilizando o comando

decompose a(8,8), b(8,8) **into** mask(1,8)

que particiona os objetos em oito trechos, cada um deles contendo uma coluna das matrizes originais. A seguir, distribuir as colunas da partição pelas memórias dos processadores, pelo comando

distribute mask(i,j) = j

que associa ao processador *j* (*j* = 1,8) o elemento (1, *j*) da partição mask, ou seja, a *j-ésima* coluna das matrizes *a* e *b*. Desta forma, *decompose* sobrepõe uma máscara ao objeto e *distribute* associa trechos da máscara aos processadores. O número de processadores é definido implicitamente, pelo número de elementos da máscara na direção distribuída.

Para distribuir as linhas do mesmo fragmento de programa, basta o par de comandos

decompose a(8,8), b(8,8) **into** mask(8,1)
distribute mask(i,j) = i

enquanto blocos de duas linhas por duas colunas são definidos por

decompose a(8,8), b(8,8) **into** mask(4,4)

e podem ser facilmente distribuídos por linhas ou colunas.

2.2 Explorando Paralelismo

O programa fonte original, descrito em um único espaço de endereçamento e munido de comandos de decomposição e distribuição dos dados deve ser automaticamente traduzido para múltiplos espaços de endereçamento. As regras básicas da tradução mais simples atualmente empregada são:

1. O mesmo código é gerado para todos os processadores. Como processadores distintos podem executar comandos distintos, cada comando é guardado por um teste que seleciona quais processadores irão executá-lo;
2. Comandos que alteram o fluxo de controle do programa são executados por todos os processadores;
3. Comandos de atribuição são executados apenas pelos processadores que guardam a variável a ser atribuída (*owner computes*).
4. Variáveis escalares são replicadas, ou seja, cada processador possui sua cópia de todas as variáveis escalares;
5. *Arrays* podem ser particionados pelas memórias dos processadores, de forma que cada elemento do *array* original é guardado por um único processador;

Por estas regras, atribuições a variáveis escalares são executadas por todos os processadores, enquanto atribuições a elementos de *arrays* distribuídos são executados apenas pelo processador que os guarda. Como os demais comandos são executados por todos os processadores, o paralelismo fica restrito às atribuições que tem como alvo *arrays* distribuídos.

Ainda mais, dado um programa, tanto o escalonamento das atribuições quanto o volume de troca de mensagens são funções exclusivas da distribuição de dados. Como a distribuição define as variáveis armazenadas em cada processador, define quais atribuições são executadas (o escalonamento) e onde estão os dados necessários a cada atribuição (a troca de mensagens).

Uma técnica simples (embora ineficiente) para implementar esse conjunto de regras é *run time resolution*. Nessa técnica, o código gerado é quem determina, durante a execução, qual processador guarda cada variável. Consequentemente, só durante a execução é possível determinar o escalonamento de comandos e o tráfego de mensagens.

Para observar um exemplo de *run time resolution*, considere o código seqüencial abaixo:

```
real a(8,8)
real b(8,8)
do i = 1,8
   do j = 4,8
      a(i,j) = b(i,j-1)
   end do
end do
```

Admitindo a existência das seguintes funções:

own (x), que retorna se este processador guarda a variável de endereço global x;
owner (x), que retorna a identidade do processador que guarda a variável de endereço global x;

então, *run-time resolution* gera o pseudo-código a seguir, onde, por simplicidade de apresentação, foram mantidos os endereços globais:

```
do i = 1, 8
  do j = 4, 8
      /* se este processador guarda a variável desejada, então
                    envia dados para outros processadores: */
      if ( own ( b(i,j-1) ) and not own ( a(i,j) ) ) then
            send b(i,j-1) to owner ( a(i,j) )
      end if
      /* se este processador guarda o lado esquerdo da atribuição,
                    então receba dados dos outros processadores
                          e compute: */
      if ( own ( a(i,j) ) ) then
            if ( own ( b(i,j-1) ) ) then
                  a(i,j) = b(i,j-1)
            else
                  receive b_1 from owner ( b(i,j-1) )
                  a(i,j) = b_1
            end if
      end if
  end do
end do
```

Observe que cada processador decide, isoladamente, se deve ou não executar comandos no interior do laço. Desta forma, podem haver processadores que percorram todas as iterações e não executem nenhum comando. Da mesma forma, podem haver processadores que apenas enviem dados, ou ainda outros que recebam e computem. O papel de cada processador é função da computação em si e da distribuição. Por exemplo, na computação acima, distribuir a coluna *j* de cada objeto para o processador *j* ocasiona o tráfego e a carga a seguir:

Processador	*Envia*	*Recebe*	*Computa*
1	------	-------	-----
2	------	-------	-----
3	b(:,3)	-------	-----
4	b(:,4)	b(:,3)	a(:,4)
5	b(:,5)	b(:,4)	a(:,5)
6	b(:,6)	b(:,5)	a(:,6)
7	b(:,7)	b(:,6)	a(:,7)
8	------	b(:,7)	a(:,8)

Observe como a distribuição ocasiona grande variação na carga entre os processadores. Já se a linha *i* for armazenada no processador *i*, então esse processador computa a(*i*,4:8), o que requer b(*i*,3:7), dado armazenado nesse mesmo processador. Consequentemente, nessa distribuição todos os processadores realizam a mesma quantidade de atribuições e não há envio ou recepção de dados. Logo, a segunda distribuição é muito mais eficiente que a primeira.

A importância da distribuição de dados é patente. Ao ser explicitada pela programação em espaço único de endereçamento, sua interferencia na carga computacional de cada processador e na quantidade de comunicação entre

processadores torna-se imediata. A facilidade de alteração dessas características bem como a simplicidade da simulação projetada formam a motivação deste trabalho.

3 Descrição da Ferramenta

A ferramenta [4], baseada nos trabalhos [1] e [3], simula a execução de um programa, produzindo seu tempo de execução e a parcela deste tempo devida à comunicação entre os processadores. São suas características marcantes a facilidade de alteração da distribuição de dados e a velocidade de obtenção dos resultados.

3.1 A Estrutura Interna da Ferramenta

A estrutura da ferramenta encontra-se representada na Figura 1.

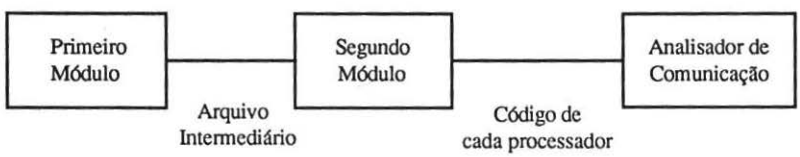

Figura 1. Estrutura da Ferramenta de Análise

O primeiro módulo traduz o programa original para uma forma intermediária ainda sob espaço de endereçamento único. O segundo módulo traduz para espaço de endereçamento múltiplo, ou seja, transforma endereços globais em locais e gera comandos de comunicação entre processadores. Já o analisador de comunicações simula a execução desse programa, atribuindo tempos de execução para cada instrução fonte para que o tempo total de execução, incluindo a comunicação, possa ser calculado precisamente.

O acesso aos módulos é feito unicamente através de uma interface com o usuário, descrita a seguir.

3.2 A Interface

O objetivo da interface ([3], [5]) é auxiliar a busca de distribuições, automatizando partes do processo de experimentação. Dado um programa fonte, a interface provê um mecanismo eficiente para especificar uma distribuição e analisar a comunicação induzida, realimentando a escolha de nova distribuição. Observe que esse processo não altera o programa fonte.

A interface é composta por três telas. A primeira permite a definição da decomposição, por meio dos comandos **decompose** e **distribute**. A segunda controla a navegação pelo sistema enquanto a última reporta os resultados da simulação. Estes são o tempo total gasto na execução de instruções (em unidades de tempo), o tempo total gasto especificamente em instruções *send* e *receive* e o percentual do tempo total gasto em comunicação.

Há ainda outras telas e mensagens (por exemplo, solicitando o nome do programa fonte) que são extremamente úteis mas irrelevantes no contexto deste

trabalho. Em essência, a análise de distribuições é efetuada gerando a distribuição, simulando sua execução e analisando a comunicação gerada. A ferramenta é construida de forma a permitir a análise de múltiplas distribuições para um mesmo programa.

4 Distribuições Ótimas

Uma conseqüência natural da existência da ferramenta é a procura por distribuições de dados ótimas com relação à comunicação, ou seja, distribuições que não acarretem comunicação entre os processadores. Apresentamos a seguir um método para encontrar tais distribuições, publicado em [4]. Ele será utilizado para gerar as distribuições ótimas nos exemplos de uso da ferramenta.

Distribuir um *array* entre processadores equivale a encontrar uma partição do espaço de índices desse *array* e atribuir cada trecho da partição a um processador. Uma partição do espaço de índices de um *array* bidimensional $a[i,j]$ é uma família de equações na forma

$$\alpha i + \beta j = c$$

com α e β fixos. Por exemplo, a família de equações $0i + 1j = c$ associa à cada valor de c uma coluna da matriz original. Essa coluna pode ser associada ao processador c, se houver o mesmo número de processadores e colunas, ou ainda ao processador $c/2$, caso o número de colunas for o dobro do número de processadores.

Encontrar distribuições de dois *arrays* que não acarretem comunicação entre os processadores significa encontrar partições correspondentes, ou seja, partições dos *arrays* tais que percorrer um único trecho da partição do primeiro *array* implica, obrigatoriamente, em percorrer um único trecho da partição do segundo *array*. Por exemplo, trechos que estejam mapeados no mesmo processador.

Suponha, a seguir, o laço abaixo:

```
do i = 1, n
  do j = 1, n
    a[i, j] ← b[i', j']
  end do
end do
```

com i' e j' funções de i e j na forma:

$$\begin{aligned} i' &= a_{11}i + a_{12}j + a_{10} \\ j' &= a_{21}i + a_{22}j + a_{20} \end{aligned} \qquad (1)$$

Procuramos partições correspondentes para a e b, ou seja, funções na forma $\alpha i + \beta j = c$ para o *array* a e $\alpha' i' + \beta' j' = c'$ para o *array* b de tal forma que todos os pares (i,j) que geram um único valor c na primeira partição são mapeados, por (1), em pares (i',j') que geram um único valor c' na segunda partição.

Para tanto, basta impor que as duas funções sejam idênticas, ou seja, obtenham os mesmos resultados para os mesmos valores de (i,j). Substituindo (1) na expressão da partição de b, obtemos

$$\alpha'(a_{11}i + a_{12}j + a_{10}) + \beta'(a_{21}i + a_{22}j + a_{20}) = c'$$

que deve ser idêntica à partição de *a*. Para tanto, basta identificar os coeficientes de *i*, *j* e do têrmo independente, gerando o sistema:

$$\begin{bmatrix} a_{11} & a_{21} & 0 \\ a_{12} & a_{22} & 0 \\ -a_{10} & -a_{20} & 1 \end{bmatrix} \begin{bmatrix} \alpha' \\ \beta' \\ c' \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \\ c \end{bmatrix}$$

ou seja, dada uma partição de *a*, o sistema acima gera uma partição correspondente em *b*. Por exemplo, o laço

```
do i = 1, n
  do j = 1, n
     a[i,j] ← b[i-1,j]
  end do
end do
```

gera o sistema

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha' \\ \beta' \\ c' \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \\ c \end{bmatrix}$$

que apresenta solução $\alpha' = \alpha$, $\beta' = \beta$ e $c' = c - \alpha$. Supondo n processadores e distribuindo uma linha de *a* para cada processador (o que significa $\alpha = 1$ e $\beta = 0$) a solução do sistema é $\alpha' = 1$, $\beta' = 0$ e $c' = c - 1$, o que significa distribuir uma linha de *b* para cada processador ($\alpha' = 1$ e $\beta' = 0$) de tal forma que se a linha *i* de *a* está no processador *c* então a linha $i - 1$ de *b* está no mesmo processador (pois $c' = c - 1$). Uma outra distribuição possível é atribuir uma coluna de *a* para cada processador (o que significa $\alpha = 0$ e $\beta = 1$). Nesse caso, a solução implica em atribuir a mesma coluna de *b* para cada processador (pois $\alpha' = 1$, $\beta' = 1$ e $c' = c$).

Ainda mais importante, a existência ou não da partição equivalente em *b* independe da particular partição adotada em *a*, visto que a existência da solução do sistema depende apenas da matriz dos coeficientes e não dos valores de α e β. Logo, dada a relação entre os índices de *a* e *b*, é possível determinar se existem partições correspondentes.

5. Medidas Realizadas

Avaliamos, a seguir, distribuições de dados utilizando a ferramenta recém apresentada. Para cada programa fonte apresentamos o tempo de execução e a parcela desse tempo utilizada em comunicação sob diversas distribuições de dados. Nas tabelas abaixo, a notação *va(1,4)=j* representa a decomposição e a distribuição efetuadas no *array a*. Significa que *a* foi decomposto em um *array* virtual *va* de uma linha e quatro colunas e distribuído por colunas. Já *vb(4,1)=i* significa que o array *b* foi decomposto em um *array* virtual *vb* com quatro linhas e uma coluna e distribuído por linhas.

O primeiro programa testado possui distribuição ótima imediata:

```
real a(8,8)
real b(8,8)
do i = 1,8
  do j = 4,8
       a(i,j) = b(i,j-3) + b(i,j-2)
  end do
end do
```

Basta observar que, ao distribuir as linhas de *a* e *b*, todos os dados necessários a cada iteração estão no mesmo processador. Os resultados da simulação demonstram claramente esse fato:

DECOMPOSE DISTRIBUTE	P1	P2	P3	P4	P5	P6	P7	P8
va(1,1)=i vb(1,1)=i	120 ut 0%							
va(2,1)=i vb(2,1)=i	60 ut 0%	60 ut 0%						
va(8,1)=i vb(8,1)=i	15 ut 0%	15 ut 0%	15 ut 0%	15 ut 0%	15 ut 0%	15 ut 0%	15 ut 0%	15 ut 0%
va(1,2)=j vb(1,2)=j	224 ut 89%	230 ut 58%						
va(1,8)=j vb(1,8)=j	40 ut 100%	80 ut 100%	80 ut 100%	109 ut 91%	114 ut 92%	111 ut 83%	112 ut 85%	117 ut 86%
va(8,1)=i vb(1,8)=j	57 ut 85%	102 ut 92%	114 ut 91%	123 ut 90%	129 ut 89%	122 ut 87%	122 ut 88%	132 ut 89%

A execução seqüencial demanda 120 unidades de tempo. Inicialmente, particionou-se os *arrays* por linha e variou-se o número de processadores. Com dois processadores observa-se que o tempo de execução é a metade do seqüencial. Com oito processadores, o tempo de execução é um oitavo do seqüencial, ou seja, o ganho é ótimo nos dois casos (igual ao número de processadores) e não há qualquer troca de mensagens (0%) entre os processadores.

Tentou-se, a seguir, distribuir os dados por colunas. Com dois processadores, o tempo de execução praticamente dobrou com relação ao tempo seqüencial, e 89% do tempo foi dedicado à comunicação. Com oito processadores o tempo de execução foi aproximadamente igual ao tempo seqüencial e os três primeiros processadores realizaram apenas comunicação. Isto ocorre por dois fatos, originados na distribuição por colunas: o aumento da necessidade de comunicação (pois os dados de uma iteração em *j* residem em outras iterações, provavelmente em outros processadores) e o péssimo escalonamento (pois o laço interno possui cinco iterações que não são equitativamente particionadas entre os processadores). Por exemplo, com dois processadores, o primeiro computa apenas uma iteração enquanto o segundo computa quatro.

O segundo teste não possui particionamento ótimo imediato, pois há comunicação tanto no particionamento por linhas quanto por colunas:

```
real a(8,8)
real b(8,8)
do i = 2,6
   do j = 3,6
        a(i,j) = b(i+1,j+2) + b(i+2,j+1)
   end do
end do
```

Os resultados experimentais demonstram a dificuldade de particionamento. Não há ganho nas distribuições convencionais (distribuindo os dois objetos por linhas ou por colunas simultaneamente) e há ganho mínimo nas distribuições alternativas (quando um objeto é distribuido por linhas e o outro por colunas).

DECOMPOSE DISTRIBUTE	P1	P2	P3	P4	P5	P6	P7	P8
va(1,1)=i vb(1,1)=i	60 ut 0%							
va(1,8)=j vb(1,8)=j	0 ut 0%	18 ut 66%	28 ut 78%	48 ut 85%	64 ut 89%	72 ut 88%	50 ut 100%	25 ut 100%
va(8,1)=i vb(8,1)=i	0 ut 0%	44 ut 90%	64 ut 93%	84 ut 95%	84 ut 95%	84 ut 95%	40 ut 100%	20 ut 100%
va(1,8)=j vb(8,1)=i	0 ut 0%	33 ut 65%	43 ut 81%	52 ut 92%	52 ut 92%	52 ut 92%	40 ut 100%	20 ut 100%
va(8,1)=i vb(1,8)=j	0 ut 0%	23 ut 65%	43 ut 81%	52 ut 92%	52 ut 92%	52 ut 92%	50 ut 100%	25 ut 100%

Entretanto, há particionamentos que não geram comunicações. O método desenvolvido em [4] sugere particionar os *arrays* *a* e *b* em anti-diagonais, que são famílias de linhas paralelas de equação $i+j=cte$. A partição sem comunicação é obtida atribuindo ao mesmo processador os elementos $i+j=k$ de *a* e $i+j=k+3$ de *b*. Infelizmente, este resultado não pode ser testado, pois a implementação atual da ferramenta não suporta tais decomposições.

O terceiro teste é uma transposição de matrizes.

```
real a(8,8)
real b(8,8)
do i = 1,8
   do j = 1,8
        a(i,j) = b(j,i)
   end do
end do
```

Os tempos de execução abaixo demonstram que não há ganho ao escolher uma distribuição convencional (colunas, no exemplo abaixo) mesmo operando com oito processadores. Só há ganho para uma distribuição não convencional (linhas de uma matriz e colunas de outra):

DECOMPOSE DISTRIBUTE	P1	P2	P3	P4	P5	P6	P7	P8
va(1,1)=i vb(1,1)=i	128 ut 0%							
va(1,2)=j vb(1,2)=j	220 ut 70%	226 ut 71%						
va(1,8)=j vb(1,8)=j	175 ut 92%	180 ut 91%	185 ut 91%	190 ut 91%	195 ut 91%	200 ut 92%	205 ut 92%	205 ut 92%
va(2,1)=i vb(1,2)=j	64 ut 0%	64 ut 0%						
va(8,1)=i vb(1,8)=j	16 ut 0%	16 ut 0%	16 ut 0%	16 ut 0%	16 ut 0%	16 ut 0%	16 ut 0%	16 ut 0%
va(1,8)=j vb(8,1)=i	16 ut 0%	16 ut 0%	16 ut 0%	16 ut 0%	16 ut 0%	16 ut 0%	16 ut 0%	16 ut 0%

A distribuição que gera comunicação nula foi gerada pela formulação [4]. Nesse caso, há ganho ótimo (fator de oito em oito processadores).

No quarto teste, a formulação [4] indica que não há distribuição livre de comunicação. As distribuições testadas aumentam o tempo de execução em relação ao tempo gasto por um único processador.

```
real a(4,4)
real b(4,4)
do i = 2,4
   do j = 2,4
        a(i,j) = b(i-2,j-1) + b(i-1,j-1) + b(i-1,j-2)
   end do
end do
```

DECOMPOSE DISTRIBUTE	P1	P2	P3	P4
va(1,1)=i vb(1,1)=i	16 ut 0%			
va(1,4)=j vb(1,4)=j	10 ut 100%	30 ut 100%	38 ut 86%	41 ut 85%
va(4,1)=i vb(4,1)=i	10 ut 100%	30 ut 100%	42 ut 85%	44 ut 84%
va(1,4)=j vb(4,1)=i	10 ut 100%	30 ut 100%	39 ut 82%	41 ut 80%
va(4,1)=i vb(1,4)=j	10 ut 100%	30 ut 100%	27 ut 74%	32 ut 84%

5 Conclusões

Este trabalho é um primeiro passo no estudo de compiladores para máquinas com memória distribuída, por permitir estimar o impacto de uma distribuição de dados em um aninhamento de laços. É claro que o problema da distribuição de dados, da comunicação e do escalonamento associados é muito mais grave. Basta observar que um mesmo objeto pode comparecer em diversos laços em um mesmo programa e que uma distribuição adequada para um laço pode ser péssima para outro.

Os laços e os resultados da seção anterior permitem verificar que a formulação matemática proposta em [4] produz distribuições inesperadas, com ótimos resultados. Melhor ainda, as distribuições são obtidas automaticamente, o que estimula o uso da formulação em um compilador. Entretanto, a formulação opera em um único laço e produz distribuições que, embora ótimas quanto à comunicação, podem não ser ótimas quanto à carga computacional.

A utilidade da ferramenta é patente. Direções óbvias de trabalho são a extensão do simulador para tratar programas com múltiplos laços e distribuições não convencionais. Deseja-se, também, aferir o simulador contrastando seus resultados com execuções em máquinas paralelas reais.

6 Bibliografia

[1] Callahan, D. & Kennedy, K., *Compiling Programs for Distributed Memory Multiprocessors*, Technical Report, Department of Computer Science, Rice University, 1988.

[2] Wholey, S., "Automatic Data Mapping for Distributed-Memory Parallel Computers", *Proceedings of the International Conference on Supercomputing*, Washington, DC, 1992, pp. 25-34.

[3] Balasundaram, V. & Fox ,G. & Kennedy,K. & Kremer, U. "An Interactive Environment for Data Partitioning and Distribution", *Proceedings of the Fifth Distributed Memory Computing Conference*, Charleston, S. Carolina, 1990.

[4] Ramanujam, J., Sadayappan, P., "Compile-Time Techniques for Data Distribution in Distributed Memory Machines", *IEEE Transactions on Parallel and Distributed Systems* (2)4, October 1991, pp. 472-482.

[5] Silva, C.H.B., Larangeira, G.H.S. *Implementação de um Algoritmo para Distribuição de Dados num Sistema com Memória Distribuída e de uma Ferramenta para Análise da Dsitribuição de Dados*, Projeto de Final de Curso, Depto. de Computação, Univerisdade Federal Fluminense, Janeiro, 1994.