Uso do Comando *Doall* e Vetorização de Mensagens em Máquinas com Memória Distribuída

Alexandre da Costa Sena[1]
Aline de Paula Nascimento[2]
Paula Marisa da C. P. F. Maciel[3]
Jairo Panetta[4]
Claudio Amorim[5]

Sumário

Apresentamos um estudo, a partir de resultados obtidos, da distribuição das iterações do comando *doall* para gerar a carga de trabalho de cada processador em máquinas paralelas com memória distribuída. Essa implementação foi feita em paralelo com a vetorização de mensagens para otimizar parte da comunicação entre os processadores. Argumentamos a importância de uma nova técnica para geração da carga de trabalho de cada processador já que nem sempre a regra *owner computes*, amplamente usada, é justa. Concluímos que em certas aplicações a distribuição das iterações é importante para que processadores não fiquem ociosos, além de reduzir a quantidade de tempo gasta com troca de mensagens.

1. Introdução

Máquinas paralelas, cada vez mais usadas para aumentar a capacidade de execução de aplicações científicas, podem ser divididas, quanto à organização de sua memória, em máquinas paralelas com memória compartilhada e máquinas paralelas com memória distribuída. As máquinas com memória compartilhada são mais fáceis de programar, mas são extremamente caras e limitadas quanto ao número de processadores. Já as máquinas com memória distribuída possuem uma relação custo-desempenho bem melhor e são altamente escaláveis. Porém a programação nestas máquinas não é uma tarefa fácil e as maiores dificuldades enfrentadas pelos programadores estão diretamente ligadas à correção e eficiência dos programas. Entre essas dificuldades destaca-se a ausência de um espaço de endereçamento global, pois cada processador tem a sua memória local (seu próprio espaço de endereços), obrigando o programador a distribuir código e dados entre os nós e a inserir no programa fonte comandos que implementam troca de mensagens [1]. Isso torna tais programas mais dificeis de serem desenvolvidos e depurados. Assim, se existir muita comunicação entre os processadores, a eficiência da máquina se torna menor, o que deixa claro a dependência da *performance* em relação à distribuição dos dados. Por isso, é importante que se escolha um bom mapeamento dos dados entre os processadores para que o tempo gasto com comunicação seja minimizado [1,7].

Como atualmente existem muitas aplicações onde o uso de máquinas com memória distribuída é mais eficiente, algumas soluções foram propostas na tentativa

[1] Bacharel em Computação pela Universidade Federal Fluminense
[2] Bacharel em Computação pela Universidade Federal Fluminense
[3] Professora Assistente, Depto. de Computação, Universidade Federal Fluminense. Aluna de Doutorado COPPE Sistemas/UFRJ
E-mail: paula@cos.ufrj.br
[4] PhD, Pesquisador, Instituto de Estudos Avançados, Centro Técnico Aeroespacial
E-mail: panetta@ieav.cta.br
[5] PhD, Professor da COPPE Sistemas/UFRJ
E-mail: amorim@cos.ufrj.br

de minimizar as dificuldades e incentivar o uso dessas máquinas. O objetivo é permitir que os programas sejam escritos em um espaço de endereçamento único, o que significa que o programador não precisa se preocupar com a localização dos dados e, portanto, com nenhuma forma de comunicação. Cabe ao compilador a tarefa de traduzir esse espaço de endereçamento único num espaço de endereçamento múltiplo e de inserir no código gerado as primitivas que implementam a comunicação (*SEND* e *RECEIVE*). Além disso, ele deve também implementar a distribuição de dados especificada pelo programador [1,3].

A maioria dos modelos computacionais existentes segue a regra *owner computes* para estabelecer a carga de trabalho de cada processador. Segundo essa regra, cada processador executa apenas atribuições que atualizem os objetos armazenados em sua memória local [1,2], levando a um escalonamento de comandos muito dependente da distribuição de dados escolhida pelo programador. Dessa forma, a distribuição dos dados determina não só onde acontecerão as computações (carga de trabalho de cada processador), mas também onde há necessidade de comunicação.

O nosso objetivo neste trabalho é mostrar uma maneira diferente de definir a carga de trabalho de cada nó, não usando a regra *owner computes* [1]. A técnica se baseia na distribuição das iterações pertencentes a um *loop* paralelo do tipo *doall* entre os processadores existentes. Cada processador executa a computação que pertence às suas iterações, não importando se ele é dono dos elementos que são atualizados nos comandos de atribuição ou não. Esta técnica tem como meta obter uma melhor distribuição da carga de trabalho entre os processadores e mostrar que, conseqüentemente, pode ser obtida uma maior eficiência na execução dos programas.

Neste trabalho é feita uma descrição dos dois ambientes [6,8,9] utilizados, é apresentado o comando *doall* e são mostrados os testes e resultados obtidos (seção 2). Finalmente são apresentadas as conclusões (seção 3). Os testes e seus resultados nos permitiram observar que, para certas aplicações, é interessante que o usuário disponha de uma maneira alternativa de escalonar a carga de trabalho dos processadores.

2. Ambientes Utilizados e o Comando *Doall*

Um dos fatores que tornam a programação de máquinas com memória distribuída difícil é o fato dos programadores não terem uma visão global dos dados na memória local de cada processador. Os dados são distribuídos entre as memórias, e os processadores alcançam por seus próprios meios apenas os objetos contidos em sua memória local. O acesso a dados não locais só pode ser feito através de troca de mensagens.

Nos trabalhos [6,8,9] usados como base para as novas alterações, o modelo computacional adotado é conhecido como *Single Program Multiple Data* (SPMD), no qual o código gerado para todos os processadores é exatamente igual mudando apenas os dados que serão manipulados (lidos e escritos) em cada nó. Apesar do código ser o mesmo para cada processador, a carga de trabalho ou os comandos que cada um deles executará é definida pelos dados armazenados na memória local de cada unidade de processamento seguindo a regra *owner computes* [1]. Isso ocorre apenas em tempo de execução (*run-time resolution*) [3]. Assim, todos os processadores tentam executar todos os comandos de todas as iterações que compõem os comandos de repetição que formam o seu código. Porém, pela regra *owner computes*, os nós só realizam as atribuições que modificam os elementos contidos na sua memória local, embora possam utilizar elementos locais a outros processadores. Isto é para evitar que um processador tente escrever num elemento não local e gere resultados inconsistentes.

Os dados não-locais devem ser lidos através da troca de mensagens entre os processadores envolvidos.

Fica claro que a distribuição de dados é de extrema importância, pois ela deve permitir que um nó faça o maior número possível de computações usando os seus dados locais, diminuindo a necessidade de comunicação entre nós. Além disso, uma distribuição de dados eficiente também implica numa carga de trabalho por nó mais equilibrada [1].

Para criarmos um ambiente onde o programador pudesse testar diferentes distribuições de dados e analisar a *performance* do seu programa desenvolvemos a ferramenta [6,8,9], composta por uma inteface, um módulo gerador de comunicação e um analisador de tempo de execução e de comunicação. O usuário tem acesso a diferentes telas onde pode especificar a distribuição dos *arrays* do programa fonte, verificar o código gerado para cada processador e o tempo que cada nó gastou na execução de comandos e em comunicação. O programa fonte é escrito numa versão Fortran simplificada,

Para especificar a decomposição e distribuição dos dados o programador deve usar os comandos **DECOMPOSE** e **DISTRIBUTE**. O comando **DECOMPOSE** particiona os *arrays* em linhas (I), colunas (J), blocos (I,J), linhas *round-robin* (RL), colunas *round-robin* (RC), diagonais (J-cte) ou antidiagonais (J+cte) [6,8,9] criando um outro *array* virtual. O comando **DISTRIBUTE** distribui o *array* virtual, alocando na memória local de cada processador uma ou mais linhas, colunas, blocos, diagonais ou antidiagonais, dependendo da decomposição escolhida. A utilização mais detalhada dos comandos **DECOMPOSE** e **DISTRIBUTE** pode ser vista em [6,8,9]. O exemplo a seguir mostra uma distribuição por coluna *round robin* (RC) do *array* A.

Decompose A(4,4) **into** *Colunarr* VA (1,2)
Distribute VA(i,j) = **RC**

O *array* A foi decomposto pelo comando **DECOMPOSE** em um *array* virtual VA com uma linha e duas colunas. A única linha de VA corresponde às quatro linhas de A e cada coluna de VA corresponde a duas colunas de A. Como a decomposição é do tipo *round-robin* a primeira coluna de VA conterá as colunas 1 e 3 de A e a segunda coluna de VA conterá as colunas 2 e 4. O comando **DISTRIBUTE** distribui cada coluna do *array* virtual VA entre cada processador.

2.1 O Comando *Doall*

O objetivo principal deste trabalho é implementar uma maneira diferente da regra *owner computes* para distribuir a carga de trabalho entre os processadores, na tentativa de se obterem resultados melhores. Nesta nova implementação os programas de entrada, escritos em Fortran simplificado, devem possuir um comando do tipo *doall*. Se existir apenas um *loop*, este passará a ser do tipo *doall*, e se existirem dois ou mais *loops* aninhados o mais externo deverá ser do tipo *doall*.

As iterações correspondentes ao comando *doall* são distribuídas entre os processadores ativos, definindo assim a carga de trabalho de cada nó. Além disso, os processadores podem atualizar elementos de *arrays*, mesmo que estes não estejam armazenados em sua memória local. Esses elementos não-locais modificados devem ser enviados para os processadores identificados como seus donos. Isso impõem uma

comunicação que não existia anteriormente envolvendo esses dados não-locais, situados no lado esquerdo dos comandos de atribuição. A nossa idéia é tentar compensar esse *overhead* com uma distribuição da carga de trabalho em cada processador mais justa. As iterações podem ser distribuídas de duas maneiras diferentes: por bloco e por *round robin*. Analisemos a distribuição das iterações com o exemplo a seguir.

→ Programa fonte :

```
real a(8,8)
real b(8,8)
doall 10 i = 1,7
   do 10 j = 1,8
10      a(i , j) = b(i + 1, j)
stop
end
```

- Supondo dois processadores (P_1 e P_2) e uma decomposição e distribuição de dados qualquer, a distribuição das iterações ficará da seguinte forma:

  - por bloco
    P_1 - executará as iterações 1,2,3,4;
    P_2 - executará as iterações 5,6,7;

  - por *round robin*
    P_1 - executará as iterações 1,3,5,7;
    P_2 - executará as iterações 2,4,6;

Cada processador executa as iterações a ele atribuídas e não mais é usada a regra *owner computes*. Como conseqüência, todos os processadores executarão quase o mesmo número de iterações. A escolha de uma boa decomposição e distribuição dos dados continua sendo fundamental, pois ela poderá aumentar ou diminuir consideravelmente a quantidade de comunicação entre os processadores. Com esta nova metodologia, cada processador passa a ser dono não só dos elementos armazenados na sua memória local, como também das iterações que ele executará. A carga de trabalho de cada processador é determinada levando-se em consideração os critérios a seguir:

- Se o processador que está executando é:

  → **Dono da iteração e de todos os elementos do lado direito, mas não é dono do lado esquerdo da atribuição**: o processador executa o comando do *loop* e guarda numa variável temporária o resultado da atribuição; a seguir, executa um *SEND* enviando o conteúdo dessa variável temporária para o processador dono do elemento.

→ **Dono da iteração, de todos os elementos do lado direito e do lado esquerdo da atribuição**: o processador apenas executa o comando, não havendo necessidade de comunicação.
→ **Dono da iteração e do lado esquerdo da atribuição, nas não é dono de pelo menos um elemento do lado direito**: o processador executa um *RECEIVE* para receber o dado não-local e depois executa os comandos que compõem a iteração.

→ **Dono da iteração, não é dono do lado esquerdo da atribuição nem de pelo menos um elemento do lado direito**: o processador executa um *RECEIVE* para receber o elemento do lado direito, executa o comando e executa um *SEND* para enviar o valor do lado esquerdo atualizado para o seu dono.

→ **Não dono da iteração e dono do lado esquerdo e de todos os elementos do lado direito da atribuição**: o processador executa um *RECEIVE* para receber o valor atualizado do dado que foi modificado e executa um *SEND*, enviando os elementos do lado direito para o processador que executará a iteração.

Nos critérios apresentados fica clara a necessidade de comunicação envolvendo dados (lado esquerdo da atribuição) que não são atualizados pelos seus donos. A sintaxe dos comandos *SEND* e *RECEIVE* de variáveis não-locais será a mesma, com uma pequena alteração no *SEND*, já que, neste caso, será enviado o conteúdo de uma variável local temporária, e não um elemento de *array*. O processador dono do elemento modificado só receberá este valor quando executar um *RECEIVE* para este elemento. A seguir é mostrado um exemplo da utilização do comando *SEND* para enviar o conteúdo atualizado de uma variável não-local.

→ Processador P_1 : **SEND** (DEST = 2) W1 ⇒ End. Global: C[2,4]

→ Processador P_2 : **RECV** (ONLYSRC = 1) C[2,4]

P_1 executa um *SEND* enviando para P_2 o conteúdo da variável W1, que contém o valor atualizado do elemento C[2,4], o qual foi calculado na respectiva iteração. P_2 executa um *RECEIVE* correspondente para atualizar o valor da variável global C[2,4], local a ele.

As modificações propostas para a ferramenta foram implementadas para os dois ambientes descritos em [6,8,9], levando-se em conta as particularidades de cada um. A seguir são apresentados os dois ambientes assim como os testes realizados e os resultados. A entrada de cada teste inclui: o programa escrito em Fortran, a distribuição de dados e a escolha para a distribuição das iterações (bloco ou *round robin*). Para um mesmo programa de entrada foram testadas diferentes distribuições de dados e de iterações. As tabelas servem para compararmos os tempos obtidos com a versão original e com a versão modificada implementada neste trabalho. Em cada linha, podemos ver o tempo gasto por cada processador para executar e a percentagem desse tempo gasta com comunicação. Todas as medidas foram realizadas pela ferramenta [6,8,9].

2.2 Primeiro Ambiente

O primeiro ambiente é descrito nos trabalhos [6,8]. Ele representa a primeira implementação por nós realizada. A comunicação é inserida pelo módulo gerador de comunicação à medida que se torna necessária para satisfazer o acesso a dados não-locais. Dessa forma, os comandos *SEND* e *RECEIVE* se encontram espalhados pelo código gerado para cada processador. As alterações necessárias para a implementação da nova estratégia de distribuição da carga de trabalho se concentraram principalmente na implementação de mecanismos para a distribuição das iterações do comando *doall* entre os processadores. Além disso, para gerar a comunicação e a carga de trabalho de cada nó, passamos a considerar não só o processador dono dos objetos de dados como o processador dono de cada iteração. A seguir são mostrados os testes e os resultados obtidos para este ambiente.

Teste 1:

```
real A(8,8)
real B(8,8)
doall 10 i = 1,8
   do 10 j = 8,8
10    a(i,j) = b(i,j-3) + b(i,j-2)
stop
end
```

DECOMPOSE DISTRIBUTE	REGRA	P1	P2	P3	P4	P5	P6	P7	P8
VA(1,8) = J VB(8,1) = I	*OWNER COMPUTES*	10 u.t. 100%	10 u.t. 100%	10 u.t. 100%	10 u.t. 100%	10 u.t. 100%	10 u.t. 100%	10 u.t. 100%	43 u.t. 48%
	DOALL (em bloco)	11 u.t. 45%	11 u.t. 45%	11 u.t. 45%	11 u.t. 45%	11 u.t. 45%	11 u.t. 45%	11 u.t. 45%	24 u.t. 70%
	DOALL (em Round-Robin)	11 u.t. 45%	11 u.t. 45%	11 u.t. 45%	11 u.t. 45%	11 u.t. 45%	11 u.t. 45%	11 u.t. 45%	24 u.t. 70%
VA(1,4) = J VB(4,1) = I	*OWNER COMPUTES*	20 u.t. 100%	20 u.t. 100%	20 u.t. 100%	51 u.t. 45%				
	DOALL (em bloco)	22 u.t. 45%	22 u.t. 45%	22 u.t. 45%	39 u.t. 64%				
	DOALL (em Round-Robin)	65 u.t. 84%	75 u.t. 89%	94 u.t. 91%	101 u.t. 85%				
VA(8,1) = I VB(8,1) = RL	*OWNER COMPUTES*	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%
	DOALL (em bloco)	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%
	DOALL (em Round-Robin)	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%	6 u.t. 0%
VA(4,1) = I VB(4,1) = RL	*OWNER COMPUTES*	23 u.t. 60%	28 u.t. 85%	41 u.t. 82%	50 u.t. 80%				
	DOALL (em bloco)	23 u.t. 60%	28 u.t. 85%	41 u.t. 82%	50 u.t. 80%				
	DOALL (em Round-Robin)	23 u.t. 43%	24 u.t. 45%	36 u.t. 61%	43 u.t. 69%				
VA(1,4) = J VB(1,4) = J	*OWNER COMPUTES*	–	–	80 u.t. 100%	83 u.t. 61%				
	DOALL (em bloco)	28 u.t. 75%	48 u.t. 85%	82 u.t. 85%	85 u.t. 83%				
	DOALL (em Round-Robin)	59 u.t. 86%	69 u.t. 88%	82 u.t. 85%	85 u.t. 83%				

Teste 2:

```
real A(4,4)
real B(4,4)
doall 10 i = 1,3
  do 10 j = 4,4
10   a(i+1,j) = b(i,j-3) + b(i,j-2)
stop
end
```

DECOMPOSE DISTRIBUTE	REGRA	P1	P2	P3	P4
VA(1,4) = J VB(4,1) = I	*OWNER COMPUTES*	10 u.t. 100%	10 u.t. 100%	10 u.t. 100%	21 u.t. 61%
	DOALL (em bloco)	11 u.t. 45%	11 u.t. 45%	11 u.t. 45%	14 u.t. 92%
	DOALL (em Round-Robin)	11 u.t. 45%	11 u.t. 45%	11 u.t. 45%	14 u.t. 92%
VA(1,2) = J VB(2,1) = I	*OWNER COMPUTES*	20 u.t. 100%	29 u.t. 51%		
	DOALL (em bloco)	22 u.t. 45%	29 u.t. 72%		
	DOALL (em Round-Robin)	42 u.t. 76%	43 u.t. 86%		
VA(2,2) = I,J VB(4,1) = I	*OWNER COMPUTES*	10 u.t. 100%	10 u.t. 100%	23 u.t. 82%	26 u.t. 69%
	DOALL (em bloco)	11 u.t. 45%	11 u.t. 45%	23 u.t. 69%	24 u.t. 91%
	DOALL (em Round-Robin)	11 u.t. 45%	11 u.t. 45%	23 u.t. 69%	24 u.t. 91%
VA(4,1) = I VB(4,1) = RL	*OWNER COMPUTES*	10 u.t. 100%	23 u.t. 82%	36 u.t. 88%	39 u.t. 89%
	DOALL (em bloco)	11 u.t. 45%	23 u.t. 69%	35 u.t. 80%	36 u.t. 97%
	DOALL (em Round-Robin)	11 u.t. 45%	23 u.t. 69%	35 u.t. 80%	36 u.t. 97%
VA(2,1) = I VB(2,1) = RL	*OWNER COMPUTES*	16 u.t. 62%	19 u.t. 47%		
	DOALL (em bloco)	28 u.t. 67%	31 u.t. 83%		
	DOALL (em Round-Robin)	17 u.t. 29%	18 u.t. 61%		
VA(1,4) = J VB(1,4) = J	*OWNER COMPUTES*	15 u.t. 100%	15 u.t. 100%	–	19 u.t. 52%
	DOALL (em bloco)	23 u.t. 78%	33 u.t. 84%	41 u.t. 90%	42 u.t. 92%
	DOALL (em Round-Robin)	23 u.t. 78%	33 u.t. 84%	41 u.t. 90%	42 u.t. 92%
VA(2,2) = I,J VB(4,1) = I	*OWNER COMPUTES*	10 u.t. 100%	10 u.t. 100%	23 u.t. 82%	26 u.t. 69%
	DOALL (em bloco)	11 u.t. 45%	11 u.t. 45%	23 u.t. 69%	24 u.t. 91%
	DOALL (em Round-Robin)	11 u.t. 45%	11 u.t. 45%	23 u.t. 69%	24 u.t. 91%

Teste 3:

```
real A(8,8)
real B(8,8)
doall 10 i = 2,8
   do 10 j = 4,6
10    a(i,j-3) = b(i-1,j-1) + b(i-1,j-2) + b(i-1,j-3)
stop
end
```

DECOMPOSE DISTRIBUTE	REGRA	P1	P2	P3	P4	P5	P6	P7	P8
	OWNER COMPUTES	45 u.t. 100%	93 u.t. 83%	141 u.t. 89%	189 u.t. 92%	237 u.t. 93%	285 u.t. 94%	333 u.t. 95%	336 u.t. 95%
VA(8,1) = I VB(8,1) = I	*DOALL* (em bloco)	39 u.t. 38%	79 u.t. 65%	119 u.t. 77%	159 u.t. 83%	199 u.t. 86%	239 u.t. 88%	279 u.t. 90%	280 u.t. 98%
	DOALL (em Round-Robin)	39 u.t. 38%	79 u.t. 65%	119 u.t. 77%	159 u.t. 83%	199 u.t. 86%	239 u.t. 88%	279 u.t. 90%	280 u.t. 98%
	OWNER COMPUTES	69 u.t. 65%	141 u.t. 72%	213 u.t. 81%	240 u.t. 83%				
VA(4,1) = I VB(4,1) = I	*DOALL* (em bloco)	63 u.t. 23%	127 u.t. 59%	191 u.t. 73%	216 u.t. 87%				
	DOALL (em Round-Robin)	269 u.t. 85%	323 u.t. 90%	395 u.t. 90%	396 u.t. 85%				
	OWNER COMPUTES	117 u.t. 38%	192 u.t. 54%						
VA(2,1) = I VB(2,1) = I	*DOALL* (em bloco)	111 u.t. 13%	184 u.t. 59%						
	DOALL (em Round-Robin)	335 u.t. 75%	336 u.t. 82%						
	OWNER COMPUTES	105 u.t. 77%	120 u.t. 78%	125 u.t. 79%	45 u.t. 100%	45 u.t. 100%	45 u.t. 100%	45 u.t. 100%	–
VA(1,8) = J VB(8,1) = I	*DOALL* (em bloco)	75 u.t. 65%	88 u.t. 70%	95 u.t. 72%	39 u.t. 38%	39 u.t. 38%	39 u.t. 38%	39 u.t. 38%	–
	DOALL (em Round-Robin)	75 u.t. 65%	88 u.t. 70%	95 u.t. 72%	39 u.t. 38%	39 u.t. 38%	39 u.t. 38%	39 u.t. 38%	–
	OWNER COMPUTES	166 u.t. 61%	156 u.t. 79%	90 u.t. 100%	45 u.t. 100%				
VA(1,4) = J VB(4,1) = I	*DOALL* (em bloco)	126 u.t. 58%	130 u.t. 61%	78 u.t. 38%	39 u.t. 38%				
	DOALL (em Round-Robin)	259 u.t. 84%	269 u.t. 89%	268 u.t. 92%	247 u.t. 94%				
	OWNER COMPUTES	35 u.t. 100%	82 u.t. 86%	118 u.t. 88%	105 u.t. 89%	96 u.t. 89%	93 u.t. 92%	108 u.t. 93%	123 u.t. 94%
VA(8,1) = I VB(1,8) = J	*DOALL* (em bloco)	70 u.t. 85%	133 u.t. 87%	190 u.t. 90%	203 u.t. 92%	215 u.t. 94%	236 u.t. 96%	267 u.t. 96%	268 u.t. 98%
	DOALL (em Round-Robin)	70 u.t. 85%	133 u.t. 87%	190 u.t. 90%	203 u.t. 92%	215 u.t. 94%	236 u.t. 96%	267 u.t. 96%	268 u.t. 98%
	OWNER COMPUTES	118 u.t. 87%	175 u.t. 80%	138 u.t. 82%	178 u.t. 87%				
VA(4,1) = I VB(1,4) = J	*DOALL* (em bloco)	157 u.t. 79%	225 u.t. 83%	216 u.t. 88%	228 u.t. 95%				
	DOALL (em Round-Robin)	209 u.t. 86%	246 u.t. 84%	257 u.t. 89%	258 u.t. 94%				

2.3 Segundo Ambiente

Este ambiente [9] apresenta características diferentes do anterior, principalmente no que se refere à colocação dos comandos de comunicação. O modelo adotado, proposto por Charles Koelbel, Joel Saltz, Piyush Mehrotra e Harry Berryman [2], tem o objetivo de realizar uma busca antecipada das variáveis não locais que um processador necessita para a execução dos comandos que compõem um *loop*. Conhecendo previamente que variáveis não-locais serão usadas, é possível sobreposição de uma fase de comunicação da qual todos os processadores participam, com uma fase de execução de iterações que apenas utilizam variáveis locais. Estas iterações formam um conjunto chamado de iterações locais. Após todas as iterações locais terem sido executadas pelos processadores, passa-se para uma terceira fase que envolve a execução das iterações que usam variáveis não locais, as quais já foram recebidas e armazenadas em *buffers* de entrada.

Para a implementação deste modelo, cada *loop* do programa de entrada é desdobrado em dois *loops* chamados de **INSPETOR** e **EXECUTOR**. A principal função do **INSPETOR** é gerar uma lista (*Receive_List*) composta por todas as variáveis que devem ser buscadas antecipadamente nas memórias dos outros processadores, isto é, as variáveis não-locais. Após gerada a *Receive_List* de cada processador, estes a enviam para os demais processadores para que eles possam gerar cada qual a sua *Send_List*. Na *Send_List* são colocados todos os elementos que devem ser enviados para os outros processadores. Esta é uma etapa crítica pois envolve uma fase de comunicação pesada entre todos os processadores. Além dessas duas listas, o **INSPETOR** cria duas outras para cada nó, que são: a *Local_Index_List* para guardar as iterações locais e a *Non_Local_Index_List* para guardar as iterações não-locais. O **EXECUTOR** utiliza as listas geradas pelo **INSPETOR** para a troca de dados entre os processadores e para a execução das iterações locais e não-locais.

As alterações necessárias para a implementação da nova estratégia de distribuição da carga de trabalho neste segundo ambiente [9,11] não foram tão simples já que não se restringiram apenas à distribuição das iterações. Elas incluiram também: a inserção de comandos de comunicação *SEND* e *RECEIVE* entre os comandos de atribuição, uma pequena alteração da sintaxe do comando *SEND* e uma otimização envolvendo a troca de mensagens.

Na implementação original [9] toda a comunicação era agrupada e gerada a partir das listas construídas pelo **INSPETOR**, existindo duas fases de comunicação: uma de envio de variáveis locais (*SEND*'s) e outra de recebimento de variáveis não-locais (*RECEIVE*'s). A etapa de comunicação para o envio de dados não-locais se sobrepunha à execução das iterações locais. Com a permissão de escrita em variáveis não-locais e o envio das mesmas, depois de atualizadas, para seus respectivos donos, uma parte da comunicação terá de ocorrer durante a execução dos comandos de atribuição que atualizam esses elementos não-locais. Essa comunicação não pode ser antecipada, pois causaria uma inconsistência nos resultados obtidos já que os valores a serem enviados e recebidos só estão disponíveis após a sua atualização na iteração correta.

O comando *SEND* teve a sua sintaxe ligeiramente modificada. O objetivo é mostrar se o envio de uma variável temporária foi realizado numa iteração local ou numa não-local. Uma variável temporária é do tipo **WL** quando o cáculo e o envio são numa iteração local e é do tipo **W** quando essas duas operações ocorrem numa iteração não-local.

A comunicação foi otimizada através da **vetorização de mensagens** [4,5], a qual consiste em agrupar em um único comando de *SEND* ou *RECEIVE* um conjunto de elementos cujo destino ou origem, respectivamente, seja o mesmo processador. Isso foi implementado neste ambiente a partir das listas geradas pelo **INSPETOR** e pelo fato de que grande parte da comunicação gerada está agrupada nas duas fases já descritas. Os *SEND*'s e *RECEIVE*'s que não pertencerem à *Send_List* ou à *Receive_List*, os quais correspondem ao envio e recebimento de variáveis não-locais, não foram vetorizados. Estes comandos não se encontram agrupados e devem seguir uma ordem pré-determinada. Como passou a existir duas classes de comunicação (vetorizada e não vetorizada), os comandos *SEND* e *RECV* representam o envio e o recebimento de um único dado e os comandos *SENDR* e *RECVR* representam o envio e recebimento de um conjunto de dados. Através dos resultados obtidos nos testes esperamos poder demonstrar a vantagem da vetorização. O exemplo a seguir mostra como a vetorização de mensagens foi implementada.

→ Listas de *SEND*'s e *RECEIVE*'s de P_i antes da vetorização:

---------- LISTA DE SEND's----------
SEND(DEST = 1) B[1,1] ⇒ End. Global: B[1,2]
SEND(DEST = 1) B[2,1] ⇒ End. Global: B[2,2] (Tempo total = 20 ut)
SEND(DEST = 3) B[3,1] ⇒ End. Global: B[3,2]
SEND(DEST = 3) B[4,1] ⇒ End. Global: B[4,2]

----------LISTA DE RECEIVE's----------
RECV (ONLYSRC = 3) B[2,3] → buf (1)
RECV (ONLYSRC = 3) B[3,3] → buf (2)

→Listas de *SEND*'s e *RECEIVE*'s de P_i depois da vetorização:

---------- LISTA DE SEND's ----------
SENDR(DEST = 1) (B[1,1], B[2,1]) ⇒ End. Global: B[1,2], B[2,2]
SENDR(DEST = 3) (B[3,1], B[4,1]) ⇒ End. Global: B[3,2], B[4,2]
(Tempo total = 10 ut)

----------LISTA DE RECEIVE's----------
RECVR (ONLYSRC = 3) B[2,3], B[3,3] → buf (1), buf(2)

A seguir são apresentados os testes e os resultados obtidos para este ambiente, assim como uma legenda para explicar os tipos de distribuições testadas. As tabelas obtidas devem ser analisadas para se verificar se houve ou não melhora com a implementação da nova técnica de geração de carga de trabalho para cada processador. É importante esclarecer que, para que as duas versões deste segundo ambiente pudessem ser comparadas, em termos de tempo de execução, foi implementada também a vetorização de mensagens no trabalho desenvolvido por [9]. Além disso, os testes foram feitos com o tamanho das mensagens vetorizadas restrito a no máximo 5 elementos.

Teste 1:

```
real A(8,8)
real B(8,8)
doall 10 i = 1,8
  do 10 j = 8,8
10   a(i,j) = b(i,j-3) + b(i,j-2)
stop
end
```

DECOMPOSE DISTRIBUTE	REGRA	P1	P2	P3	P4	P5	P6	P7	P8
VA(1,8) = J VB(8,1) = I	*OWNER COMPUTES*	11 u.t. 100% 6 u.t.	16 u.t. 100% 11 u.t.	21 u.t. 100% 16 u.t.	26 u.t. 100% 21 u.t.	31 u.t. 100% 26 u.t.	36 u.t. 100% 31 u.t.	41 u.t. 100% 36 u.t.	90 u.t. 46% 35 u.t.
	DOALL (em bloco)	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	32 u.t. 37% 0 u.t.
	DOALL (em Round-Robin)	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	32 u.t. 37% 0 u.t.
VA(1,4) = J VB(4,1) = I	*OWNER COMPUTES*	11 u.t. 100% 6 u.t.	16 u.t. 100% 11 u.t.	21 u.t. 100% 16 u.t.	66 u.t. 27% 15 u.t.				
	DOALL (em bloco)	22 u.t. 45% 0 u.t.	22 u.t. 45% 0 u.t.	22 u.t. 45% 0 u.t.	37 u.t. 35% 0 u.t.				
	DOALL (em Round-Robin)	29 u.t. 75% 6 u.t.	46 u.t. 73% 12 u.t.	46 u.t. 73% 12 u.t.	55 u.t. 65% 11 u.t.				
VA(4,1) = I VB(4,1) = RL	*OWNER COMPUTES*	19 u.t. 63% 6 u.t.	36 u.t. 66% 12 u.t.	36 u.t. 66% 12 u.t.	24 u.t. 70% 11 u.t.				
	DOALL (em bloco)	19 u.t. 63% 6 u.t.	36 u.t. 66% 12 u.t.	36 u.t. 66% 12 u.t.	24 u.t. 70% 11 u.t.				
	DOALL (em Round-Robin)	20 u.t. 30% 0 u.t.	28 u.t. 42% 0 u.t.	28 u.t. 42% 0 u.t.	25 u.t. 44% 0 u.t.				
VA(1,8) = J VB(1,8) = J	*OWNER COMPUTES*	–	–	–	–	21 u.t. 100% 11 u.t.	31 u.t. 100% 21 u.t.	–	81 u.t. 40% 20 u.t.
	DOALL (em bloco)	28 u.t. 78% 10 u.t.	33 u.t. 81% 10 u.t.	38 u.t. 84% 10 u.t.	43 u.t. 86% 10 u.t.	59 u.t. 89% 12 u.t.	64 u.t. 90% 17 u.t.	53 u.t. 88% 10 u.t.	74 u.t. 72% 10 u.t.
	DOALL (em Round-Robin)	28 u.t. 78% 10 u.t.	33 u.t. 81% 10 u.t.	38 u.t. 84% 10 u.t.	43 u.t. 86% 10 u.t.	59 u.t. 89% 12 u.t.	64 u.t. 90% 17 u.t.	53 u.t. 88% 10 u.t.	74 u.t. 72% 10 u.t.
VA(1,4) = J VB(1,4) = J	*OWNER COMPUTES*	–	–	41 u.t. 100% 21 u.t.	89 u.t. 46% 20 u.t.				
	DOALL (em bloco)	33 u.t. 63% 5 u.t.	38 u.t. 68% 5 u.t.	33 u.t. 100% 8 u.t.	60 u.t. 60% 5 u.t.				
	DOALL (em Round-Robin)	33 u.t. 63% 5 u.t.	38 u.t. 68% 5 u.t.	33 u.t. 100% 8 u.t.	60 u.t. 60% 5 u.t.				

Teste 2:

```
real A(8,8)
real B(8,8)
doall 10 i = 1,4
  do 10 j = 7,8
10   a(i,j-2) = b(i,j) + a(i+2,j-1)
stop
end
```

DECOMPOSE DISTRIBUTE	REGRA	P1	P2	P3	P4	P5	P6	P7	P8
VA(8,1) = I VB(8,1) = I	*OWNER COMPUTES*	23 u.t. 47% 5 u.t.	23 u.t. 47% 5 u.t.	24 u.t. 50% 6 u.t.	24 u.t. 50% 6 u.t.	11 u.t. 100% 6 u.t.	11 u.t. 100% 6 u.t.	–	–
	DOALL (em bloco)	23 u.t. 47% 5 u.t.	23 u.t. 47% 5 u.t.	24 u.t. 50% 6 u.t.	24 u.t. 50% 6 u.t.	11 u.t. 100% 6 u.t.	11 u.t. 100% 6 u.t.	–	–
	DOALL (em Round-Robin)	23 u.t. 47% 5 u.t.	23 u.t. 47% 5 u.t.	24 u.t. 50% 6 u.t.	24 u.t. 50% 6 u.t.	11 u.t. 100% 6 u.t.	11 u.t. 100% 6 u.t.	–	–
VA(4,1) = I VB(4,1) = I	*OWNER COMPUTES*	35 u.t. 31% 5 u.t.	36 u.t. 33% 6 u.t.	11 u.t. 100% 6 u.t.	–				
	DOALL (em bloco)	47 u.t. 65% 6 u.t.	58 u.t. 65% 8 u.t.	44 u.t. 72% 11 u.t.	49 u.t. 75% 10 u.t.				
	DOALL (em Round-Robin)	47 u.t. 65% 6 u.t.	58 u.t. 65% 8 u.t.	44 u.t. 72% 11 u.t.	49 u.t. 75% 10 u.t.				
VA(1,8) = RC VB(1,8) = RC	*OWNER COMPUTES*	–	–	–	–	41 u.t. 41% 10 u.t.	46 u.t. 47% 11 u.t.	21 u.t. 100% 11 u.t.	11 u.t. 100% 6 u.t.
	DOALL (em bloco)	45 u.t. 73% 15 u.t.	50 u.t. 76% 15 u.t.	55 u.t. 78% 15 u.t.	60 u.t. 80% 15 u.t.	37 u.t. 78% 0 u.t.	62 u.t. 87% 14 u.t.	29 u.t. 100% 9 u.t.	39 u.t. 100% 19 u.t.
	DOALL (em Round-Robin)	45 u.t. 73% 15 u.t.	50 u.t. 76% 15 u.t.	55 u.t. 78% 15 u.t.	60 u.t. 80% 15 u.t.	37 u.t. 78% 0 u.t.	62 u.t. 87% 14 u.t.	29 u.t. 100% 9 u.t.	39 u.t. 100% 19 u.t.
VA(1,8) = J VB(8,1) = I	*OWNER COMPUTES*	17 u.t. 100% 7 u.t.	27 u.t. 100% 17 u.t.	32 u.t. 100% 22 u.t.	37 u.t. 100% 27 u.t.	59 u.t. 59% 25 u.t.	65 u.t. 63% 26 u.t.	16 u.t. 100% 11 u.t.	–
	DOALL (em bloco)	39 u.t. 69% 10 u.t.	44 u.t. 72% 10 u.t.	49 u.t. 75% 10 u.t.	54 u.t. 77% 10 u.t.	36 u.t. 77% 0 u.t.	56 u.t. 85% 9 u.t.	34 u.t. 100% 14 u.t.	–
	DOALL (em Round-Robin)	39 u.t. 69% 10 u.t.	44 u.t. 72% 10 u.t.	49 u.t. 75% 10 u.t.	54 u.t. 77% 10 u.t.	36 u.t. 77% 0 u.t.	56 u.t. 85% 9 u.t.	34 u.t. 100% 14 u.t.	–
VA(1,4) = J VB(4,1) = I	*OWNER COMPUTES*	11 u.t. 100% 6 u.t.	21 u.t. 100% 16 u.t.	71 u.t. 32% 15 u.t.	16 u.t. 100% 11 u.t.				
	DOALL (em bloco)	40 u.t. 70% 11 u.t.	52 u.t. 76% 17 u.t.	69 u.t. 65% 13 u.t.	56 u.t. 78% 17 u.t.				
	DOALL (em Round-Robin)	40 u.t. 70% 11 u.t.	52 u.t. 76% 17 u.t.	69 u.t. 65% 13 u.t.	56 u.t. 78% 17 u.t.				

Teste 3:

```
real A(4,4)
real B(4,4)
real C(4,4)
doall 10 i = 1,4
  do 10 j = 4,4
10   c(i,j) = b(i,j) * a(i,j)
stop
end
```

DECOMPOSE DISTRIBUTE	REGRA	P1	P2	P3	P4
	OWNER COMPUTES	17 u.t. 64% 5 u.t.	22 u.t. 72% 5 u.t.	27 u.t. 77% 5 u.t.	23 u.t. 100% 8 u.t.
VA(4,1) = I VB(1,4) = J VC(4,1) = RL	*DOALL* (em bloco)	17 u.t. 64% 5 u.t.	22 u.t. 72% 5 u.t.	27 u.t. 77% 5 u.t.	23 u.t. 100% 8 u.t.
	DOALL (em Round-Robin)	17 u.t. 64% 5 u.t.	22 u.t. 72% 5 u.t.	27 u.t. 77% 5 u.t.	23 u.t. 100% 8 u.t.
	OWNER COMPUTES	11 u.t. 100% 6 u.t.	16 u.t. 100% 11 u.t.	30 u.t. 60% 11 u.t.	18 u.t. 33% 5 u.t.
VA(4,1) = I VB(4,1) = I VC(2,2) = I,J	*DOALL* (em bloco)	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	17 u.t. 41% 0 u.t.	14 u.t. 42% 0 u.t.
	DOALL (em Round-Robin)	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	17 u.t. 41% 0 u.t.	14 u.t. 42% 0 u.t.
	OWNER COMPUTES	11 u.t. 100% 6 u.t.	16 u.t. 100% 11 u.t.	21 u.t. 100% 16 u.t.	42 u.t. 42% 15 u.t.
VA(1,4) = J VB(4,1) = I VC(1,4) = RC	*DOALL* (em bloco)	22 u.t. 64% 5 u.t.	27 u.t. 77% 5 u.t.	32 u.t. 81% 5 u.t.	32 u.t. 81% 8 u.t.
	DOALL (em Round-Robin)	22 u.t. 64% 5 u.t.	27 u.t. 77% 5 u.t.	32 u.t. 81% 5 u.t.	32 u.t. 81% 8 u.t.
	OWNER COMPUTES	11 u.t. 100% 6 u.t.	16 u.t. 100% 11 u.t.	21 u.t. 100% 16 u.t.	42 u.t. 42% 15 u.t.
VA(4,1) = RL VB(4,1) = RL VC(1,4) = J	*DOALL* (em bloco)	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	20 u.t. 40% 0 u.t.
	DOALL (em Round-Robin)	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	11 u.t. 45% 0 u.t.	20 u.t. 40% 0 u.t.

3. Conclusões

A maioria dos trabalhos presentes na literatura atual [1,2,6,8,9] apresenta a regra *owner computes* ou algumas variações dela, como *almost owner computes* [3], como solução para a geração da carga de trabalho e de comunicação de cada

processador envolvido na execução de um programa. O nosso objetivo era estudar, implementar e testar uma nova regra, a qual envolve a distribuição das iterações de um *loop* paralelo (*doall*) entre os processadores. Com essa regra, cada processador passa a ser dono dos elementos de dados alocados na sua memória local e das iterações a ele atribuídas. Além disso, é permitido que os processadores escrevam em elementos não-locais, introduzindo uma nova classe de comunicação que envolve esses dados não-locais, os quais devem ser enviados para seus respectivos donos.

Os resultados obtidos nos levam a concluir que os nossos objetivos iniciais foram plenamente alcançados e, por isso, resolvemos destacar alguns aspectos importantes para cada um dos dois ambientes utilizados ao longo deste trabalho.

Como mencionado, o uso da regra *doall* permite a escrita em variáveis não-locais a um processador. Sempre que isto acontece é inserida no código de cada processador uma comunicação extra envolvendo os elementos do lado esquerdo das atribuições. Para compensar essa comunicação que não existia com o uso da regra *owner computes*, deve se levar em consideração a quantidade de comunicação envolvendo elementos do lado direito que é possível minimizar. Outra grande mudança que o uso da regra *doall* proporciona, a qual pode ser vista como uma melhora, é a distribuição das iterações entre os processadores, que leva a uma melhor distribuição da carga de trabalho entre os mesmos. Isto foi constatado sempre que o número das iterações foi maior ou igual que o número de processadores.

No primeiro ambiente, sempre que o uso da regra *doall* conseguiu minimizar a comunicação que existia com o uso da regra *owner computes* em relação à nova comunicação inserida, os resultados obtidos foram vantajosos. É claro portanto, que se o uso da regra *doall* em um determinado programa só inserir nova massa de comunicação os tempos de execução gasto pelos processadores serão bem maiores.

No segundo ambiente, com a vetorização das mensagens implementadas, o que sem dúvida reduz consideravelmente os tempos de comunicação entre os processadores, as conclusões não são tão simples como as do primeiro ambiente. Isto acontece porque, mesmo quando a massa de comunicação envolvendo elementos do lado direito é retirada ou reduzida, e a nova comunicação inserida envolvendo os elementos do lado esquerdo é mínima, temos que levar em conta a quantidade de mensagens vetorizadas que foram retiradas (que antes envolviam os elementos do lado direito) e a quantidade de mensagens não vetorizadas (que envolvem elementos do lado esquerdo) que foram inseridas. Assim, neste ambiente, para um bom uso da regra *doall*, deve se considerar a quantidade de vetorização que deixará de existir, o tempo que se ganhará na construção das *Send_List* e, dessa forma, tentar compensar a comunicação que envolve os elementos do lado esquerdo das atribuições.

Para os dois ambientes observamos que é importante para um bom desempenho, que a distribuição dos elementos de *array* combine com a distribuição das iterações pelos processadores.

Devemos destacar que, a regra *doall* sempre obtém uma distribuição da carga de trabalho mais justa entre os processadores. Porém, em relação ao tempo de execução gasto pelos processadores, essa regra pode ou não trazer vantagens. Isto dependerá do tipo da distribuição de dados utilizada, decisão que é deixada, por enquanto, na mão do programador. É interessante que as duas regras estejam disponíveis num ambiente do gênero por nós criado, já que as vantagens e desvantagens de ambas dependem bastante da própria aplicação.

4. Referências Bibliográficas

[1] Callahan, D. & Kennedy, K. *Compiling Programs for Distributed Memory Multiprocessors*. Departament of Computer Science, Rice University, Houston, Texas, 1988.

[2] Koelbel, C. & Mehrotra, P. & Saltz, J. & Berryman, H. *Parallel Loops on Distributed Machines*. ICASE, Nasa, Langely Research Center, Hampton.

[3] A. Rogers & K. Pingali *Compiling for Distributed Memory Architectures*. IEEE Transactions on Parallel and Distributed Systems, Vol. 5, N. 3, março de 1994.

[4] J. Li, M. Chen. *Compiling Communication-Efficient Programs for Massively Parallel Machines.*, IEEE, 1991.

[5] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka. *Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results*. ACM,1993.

[6] C. Batista & G. Larangeira. *Um Algoritmo para o Problema da Distribuição de Dados em Multiprocessadores*. Projeto de Final de Curso, Depto. de Computação, Universidade Federal Fluminense, dezembro de 1993.

[7 C.H.B. da Silva, G.H.S. Larangeira, P.M.C.P.F Maciel & J. Panetta *Avaliando Distribuições de Dados*. SBAC, Caxambú, agosto de 1994.

[8] Miranda, M. *Técnicas para Distribuição de Dados em Máquinas com Memória Distribuída*. Projeto de Final de Curso, Depto. de Computação, Universidade Federal Fluminense, agosto de 1994.

[9] Brazil, W. *Um Algoritmo para Minimizar o Problema da Comunicação entre Processadores em Máquinas com Memória Distribuída*. Projeto de Final de Curso, Depto. de Computação, Universidade Federal Fluminense, agosto de 1994.

[10] W. Brazil, M. Miranda, P.M.C.P.F.Maciel, J. Panetta. *Distribuições de Dados e Compilações para Máquinas com Memória Distribuída*. Apresentado no SUPERCOMP 94, Porto Alegre.

[11] Alexandre da Costa Sena, Aline de Paula Nascimento. *Distribuição da Carga de Trabalho e Vetorização de Mensagens entre Processadores na Ferramenta MMDUFF*. Projeto de Final de Curso, Depto. de Computação, Universidade Federal Fluminense, janeiro de 1995.