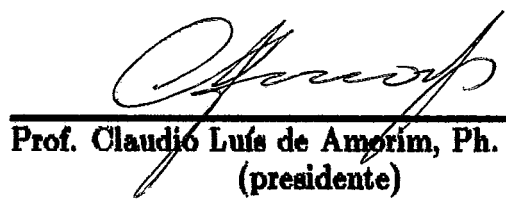
UMA BIBLIOTECA DE OPERAÇÕES VETORIAIS E MATRICIAIS PARALELAS PARA MULTIPROCESSADORES HIPERCÚBICOS BASEADOS EM TRANSPUTERS

Maria Clicia Stelling de Castro

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

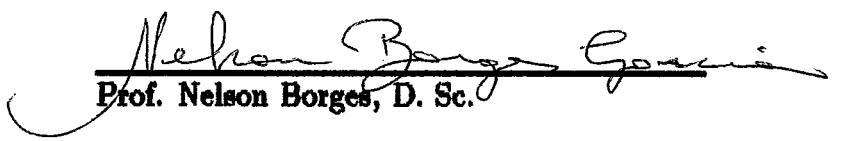
Aprovada por:



Prof. Claudio Luís de Amorim, Ph. D.
(presidente)

Prof. Valmir Carneiro Barbosa, Ph. D.

Prof. Eugenius Kaszkurewicz, D. Sc.



Prof. Nelson Borges, D. Sc.

RIO DE JANEIRO, RJ – BRASIL
MAIO DE 1991

CASTRO, MARIA CLICIA STELLING DE

Uma Biblioteca de Operações Vetoriais e Matriciais Paralelas para Multiprocessadores Hipercúbicos Baseados em *Transputers* [Rio de Janeiro] 1991

VI, 109 p., 29.7 cm, (COPPE/UFRJ, M. Sc., ENGENHARIA DE SISTEMAS E COMPUTAÇÃO, 1991)

TESE – Universidade Federal do Rio de Janeiro, COPPE

1 – Arquitetura de Computadores 2 – Processamento Paralelo

I. COPPE/UFRJ II. Título(Série).

Aos meus pais Maria José e Alcides

Agradecimentos

Agradeço aos meus familiares e amigos por todo apoio e carinho sempre presentes.

Em especial, agradeço ao professor Claudio Luís de Amorim pela sua orientação, apoio e estímulo.

Aos meus pais Maria José e Alcides,e meus irmãos Maria Clara e Dario por todo carinho e compreensão.

A professora Arzelina Mendes que com seu exemplo, seu carinho e sua dedicação me ensinou o valor do conhecimento.

Ao amigo Ildemar, pelo seu apoio e estímulo durante a realização desse trabalho.

Agradeço também aos funcionários técnicos e administrativos do Programa de Engenharia de Sistemas e Computação pela atenção e prestatividade.

Agradeço à Capes, ao CNPq, à Finep e à Sociedade Cutural e Beneficente Guilherme Guinle pelo apoio para a realização deste trabalho.

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M. Sc.)

Uma Biblioteca de Operações Vetoriais e Matriciais Paralelas para Multiprocessadores Hipercúbicos Baseados em *Transputers*

Maria Clicia Stelling de Castro

Maio de 1991

Orientador: Claudio Luís de Amorim
Programa: Engenharia de Sistemas e Computação

A solução de muitos problemas científicos e de engenharia requer uma significativa quantidade de computação de sistemas de equações. Os sistemas de computação paralelos, e em particular os de topologia hipercúbica, oferecem a perspectiva de um ganho de potência e de velocidade de processamento. Este é um ponto ainda em estudo e pesquisa.

Neste trabalho procuramos implementar uma biblioteca de operações paralelas para multiprocessadores hipercúbicos baseados em *transputers*, e que está relacionado ao modelo de Oliver Mcbryan e Erick Van de Velde.

O sucesso de projetos hipercúbicos está relacionado à possibilidade de implementação eficiente de algoritmos paralelos para uma extensa faixa de aplicações numéricas. A arquitetura hipercúbica tem características que permitem eficiente realização de outras topologias de interconexão apropriadas (grade, árvore e anel) dependendo da aplicação.

Nosso trabalho está concentrado no desenvolvimento de algoritmos

que são passos elementares de algoritmos maiores que solucionem, por exemplo, sistemas de equações elípticas, parabólicas e hiperbólicas encontrados em problemas das Engenharias, Física, Ciência da Computação entre outras.

As rotinas que formam a biblioteca de operações paralelas tratam do armazenamento de vetores e matrizes distribuídos de várias formas através dos processadores, da conversão entre esses tipos de armazenamento, de operações básicas da álgebra linear tais como produto interno, produtos de uma matriz por um vetor e de matrizes, e de operações de difusão e convergêngia de escalares, vetores e matrizes. A biblioteca opera tanto com matrizes densas quanto esparsas (matrizes de banda). As rotinas foram implementadas na linguagem *C Paralela* do *transputer.*

A aplicação de uma operação é realizada com os processadores chamando o procedimento individualmente com seus argumentos apropriados. O código executado em cada processador é o mesmo. Para a solução de um problema real, basta agrupar os procedimentos convenientes de modo a obter o algoritmo desejado, tendo a facilidade da modularidade para a depuração. A biblioteca de operações paralelas torna invisível a arquitetura da máquina e a comunicação entre os processadores realizada pelos procedimentos.

A finalidade é obter metodologias de programação portáveis de forma a facilitar o desenvolvimento de aplicações numéricas em hipercubos baseados em *transputers.*

Abstract of Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

A Library of Vector and Matrix Parallel Operations for Hypercube Multiprocessors Based on Transputers

Maria Clicia Stelling de Castro

May, 1991

Thesis Supervisor: Claudio Luís de Amorim
Department: Programa de Engenharia de Sistemas e Computação

The solution of many scientific and engineering problems requires a significant amount of computation of system of equations. Parallel computer systems, in particular hypercubes, open a perspective on high speedup and an increase of computing power. This point is still a subject of intensive research.

In this work we implement a library of parallel operations which has been developed for hypercube multiprocessors based on transputers. It is related to the Oliver Mcbryan and Erick Van de Velde's model.

The success of hypercube designs is related to the possibility of implementing parallel algorithms efficiently to a wide range of numerical applications. The hypercube architecture has caracteristics that make possible to embbed other topologies efficiently such as binary trees, hierarchies of rings and retangular grids, according to the applications.

Our work foccuses the development of algorithms that are elementary steps of larger algorithms, to solve elliptc, parabolic and hyperbolic equations which

occur in of Engineering, Phisical and Computing Science problems among others.

The routines in the library of parallel operations deal with distributed vector and matrix allocation, basic linear algebra operations such as inner products, matrix transpose and matrix-vector products and scalar, vector and matrix broadcasting. This library is efficient to both dense and sparse matrix (band matrix). The routines was implemented in *C Parallel* linguage of the transputer.

The use of a parallel operation is accomplished by having all of the processors calling the individual routines, with the appropriate arguments. The same code is executed in all the processors. To solve realistic problems, the routines must be grouped in order to obtain larger algorithms. This approach offers modularity that facilitates debugging. The library makes transparent the dependency on machine architecture and on the communication between processors.

Our goal is to obtain portable programming methodologies in order to simplify the development of numerical applications on Transputer based hypercubes.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo I

Introdução

A idéia de incluir paralelismo nos sistemas de computadores é antiga. Tanto a execução simultânea de diversas atividades pelo *hardware*, quanto a realização antecipada de algumas dessas atividades, são idéias empregadas em arquiteturas da atualidade, e incluem conceitos tais como busca antecipada de instruções (*lookahead*), superposição das fases de execução das instruções (*overlap*), múltiplas unidades funcionais, estágios de execução (*pipeline*), processamento vetorial e multiprocessadores.

O que estimula a computação paralela e distribuída é o desempenho, já que se diversos agentes estão operando é de se esperar que o tempo de processamento diminua; a tolerância a falhas, onde o componente defeituoso pode ser retirado do sistema sem que haja interrupção na operação; e a modularidade, que permite a expansão através da adição de mais processadores. Outras razões para a computação paralela foram o desenvolvimento das tecnologias empregadas no *hardware* (circuitos VLSI), o desenvolvimento de linguagens para programação paralela e pesquisas em algoritmos paralelos.

Existem diversas possibilidades de organizar uma coleção de processadores e produzir uma grande variedade de computadores paralelos com o objetivo de atingir melhor desempenho. Uma de particular sucesso é organização hipercúbica [1].

O sucesso de projetos hipercúbicos está relacionado à possibilidade de implementação eficiente de algoritmos paralelos para uma extensa faixa de aplicações

numéricas.

O trabalho desta tese está relacionado ao modelo de Oliver Mcbryan e Eric Van de Velde, que propuseram uma biblioteca de operações paralelas para multiprocessadores hipercúbicos. Mcbryan e Van de Velde utilizaram a biblioteca desenvolvida para implementar algoritmos paralelos para solucionar problemas relativos à dinâmica dos fluidos computacionais e realizaram algumas análises de desempenho empregando simuladores de máquinas hipercúbicas o Caltech Mark II da Caltech e o iPSC da Intel.

Inicialmente, empregamos um simulador de uma arquitetura hipercúbica, o simulador do iPSC2 da Intel, devido à lenta importação de uma placa *quadputer*. Este simulador utilizava um computador compatível com o sistema IBM - PC, e era executado sob o sistema operacional Xenix. Com a chegada da placa, este trabalho foi descontinuado, utilizamos um sistema *Quadputer* que é composto por quatro placas de *transputers* interconectadas como um cubo de dimensão dois.

Posteriormente utilizamos um multiprocessador de memória distribuída com topologia hipercúbica baseado em *transputers* que foi desenvolvido pela COPPE, o NCP I. Esse multiprocessador possui atualmente oito processadores.

Nosso trabalho está concentrado no desenvolvimento de algoritmos que são passos elementares de algoritmos maiores que solucionem, por exemplo, sistemas de equações elípticas, parabólicas e hiperbólicas encontrados em problemas das Engenharias, Física, Ciência da Computação entre outras.

Os algoritmos paralelos que formam a biblioteca de operações paralelas tratam do armazenamento de vetores e matrizes distribuídos de várias formas através dos processadores, da conversão entre esses tipos de armazenamento, de operações básicas da álgebra linear tais como produto interno, produtos de uma matriz por um vetor e de matrizes entre outras, e de operações de difusão e convergêngia de escalares, vetores e matrizes. A biblioteca opera tanto com matrizes densas quanto esparsas, em particular matrizes de banda.

A aplicação de uma operação é realizada com os processadores cha-

mando o procedimento individualmente com seus argumentos apropriados. O código executado em cada processador é o mesmo. Para a solução de um problema real, basta agrupar os procedimentos convenientes de modo a obter um algoritmo maior, tendo a facilidade da modularidade para a depuração. A biblioteca de operações paralelas torna invisível a dependência da arquitetura da máquina e a comunicação entre os processadores realizada pelos procedimentos. Toda metodologia utilizada na comunicação desenvolvida para a rede de transputers é interna as rotinas.

A finalidade é obter metodologias de programação portáveis de forma a facilitar o desenvolvimento de aplicações numéricas em hipercubos baseados em *transputers*. Pretendemos diminuir a dificuldade existente na programação paralela facilitando o uso na área de aplicação numéricas, reduzindo o esforço redundante de se reescrever algoritmos em geral. E ainda, proporcionar ferramentas de desenvolvimento para implementadores de algoritmos numéricos. O desenvolvimento de bibliotecas de álgebra linear portáveis pode ser relacionado a desenvolvimento tais como as rotinas BLAS [7] - [8].

O capítulo II descreve a arquitetura do multiprocessador hipercúbico baseado em *transputers* e a seqüência "Binary Reflected Gray Code" utilizada no implemento das topologias de grade, árvore e anel que são fundamentais aos algoritmos da biblioteca. A metodologia empregada para a troca de mensagens no multiprocessador hipercúbico baseado em *transputers* é relatada no capítulo III. O capítulo IV descreve as estruturas de dados que representam os vetores e as matrizes, as formas em que eles podem ser armazenados e os passos elementares que compõem a biblioteca de operações paralelas. A aplicação das operações da biblioteca para solucionar um sistema de equações lineares utilizando o método do gradiente conjugado, sua implementação e sua avaliação são mostrados no capítulo V. Os resultados finais e as possibilidades futuras são analisados no capítulo VI.

Capítulo II

Mapeamentos de Grades, Árvores e Anéis em Hipercubos

A biblioteca de operações paralelas foi desenvolvida para um multiprocessador de memória distribuída com topologia hipercúbica baseado em *transputers* (Rede de *transputers*). Esse capítulo descreve a arquitetura do multiprocessador hipercúbico e a seqüência "Binary Reflected Gray Code" (BRGC) utilizada no implemento das topologias de grade, árvore binária e anel nessa arquitetura, e que serão fundamentais para os algoritmos apresentados no capítulo IV.

II.1 Multiprocessador Hipercúbico Baseado em *Transputers*

Um multiprocessador com topologia hipercúbica binária [1] consiste de 2^D processadores independentes, que podem ser vistos espacialmente como se estivessem localizados nos vértices de um cubo de dimensão D cujas D arestas de ligação em cada vértice correspondem aos canais de comunicação com os vértices vizinhos (figura II.1).

Os processadores são identificados por números na faixa $[0, 2^D - 1]$, tal que a representação binária de D-dígitos de processadores adjacentes fisicamente diferem em somente um bit, como ilustrado na figura II.1. Os processadores não compartilham memória e se comunicam através da troca de mensagens.

O multiprocessador hipercúbico baseado em *transputers* utilizado para a validação e a avaliação de desempenho da biblioteca de operações paralelas foi o NCP I desenvolvido pela COPPE, que possui atualmente oito *transputers*, sendo então um hipercubo de dimensão três.

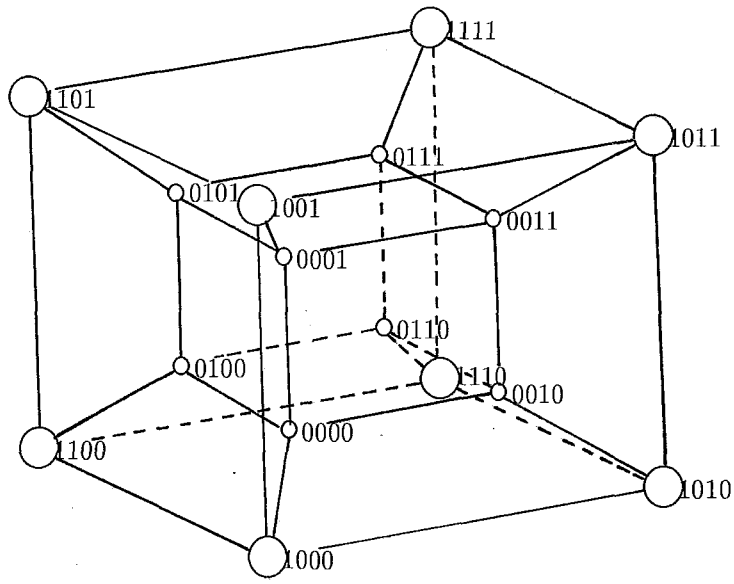


Figura II.1: Representação de um hipercubo de dimensão quatro

Cada processador individual é um *transputer* T800 [2] de 32 bits com 4K bytes de memória estática interna e 4G bytes de espaço total de endereçamento. Possui quatro ligações seriais bidirecionais com capacidade de 20M bits/s, para comunicação direta com outros quatro *transputers*, sendo a comunicação realizada de forma síncrona. Suporta também operações de ponto flutuante. Foi projetado inicialmente para dar suporte eficiente à liguagem *OCCAM*, sendo que já existem também disponíveis versões paralelas das linguagens *C*, *Pascal* e *Fortran*.

O multiprocessador hipercúbico NCP I utiliza um computador hospedeiro ligado ao processador zero, que atua como uma *interface* entre o usuário e o cubo, assim, todas as informações passam do computador hospedeiro ao processador zero através de um canal bidirecional de comunicação extra. O computador hospedeiro é um computador compatível com o sistema IBM PC.

II.2 A Seqüência "Binary Reflected Gray Code"

O código "Binary Reflected Gray Code" (BRGC) [4] tem uma importância fundamental no desenvolvimento da biblioteca de operações paralelas, pois é baseado nesta seqüência que são obtidas as topologias empregadas e distribuídos os dados através da rede hipercúbica.

Dependendo da aplicação, a interconexão entre os processadores deve ser organizada de forma a obter uma topologia apropriada. As topologias essenciais (grade, árvore e anel) aos algoritmos tratados posteriormente, são baseadas na seqüência "Binary Reflected Gray Code" (BRGC) para identificar os processadores [3]. A seqüência BRGC é definida recursivamente da seguinte forma:

- Com um bit a seqüência é 0 1;
- Com d-dígitos, o d-ésimo dígito é ativado e os $d-1$-dígitos restantes da seqüência são repetidos na ordem inversa. Assim com d-dígitos temos as seqüências para um, dois e três bits na tabela II.1:

1 bit	2 bits	3 bits	
0	00	000	
1	01	001	
	–	011	
	11	010	
	10	—	...
		110	
		111	
		101	
		100	

Tabela II.1: Geração da seqüência BRGC para um, dois e três bits

Essa seqüência apresenta propriedades convenientes a implementação das topologias de grade, árvore binária e anel:

- a seqüência é periódica;

- os elementos diferem em apenas um bit na seqüência binária;
- os elementos vizinhos na seqüência binária são vizinhos físicos no hipercubo e
- dois elementos que são uma potência de dois diferem em no máximo dois bits.

A periodicidade é útil no mapeamento do hipercubo em anel, pois os elementos dos extremos da seqüência também diferem em apenas um bit.

Se dois elementos diferem em apenas um bit, eles são vizinhos na seqüência e portanto no hipercubo (figura II.1), tendo então uma ligação física entre eles. Esse fato, auxilia no roteamento das mensagens (tempo gasto na comunicação) já que os mapeamentos de grade, árvore e anel são baseados nesta seqüência. Do mesmo modo, se dois elementos são uma potência de dois, eles diferem em no máximo dois bits e têem duas ligações físicas entre eles.

A seqüência BRGC para quatro bits onde podem ser verificadas essas propriedades é a seguinte.

0000 0001 0011 0010 0110 0111 0101 0100 1100 1101 1111 1110 1010 1011 1001 1000

A periodicidade pode ser notada entre os elementos 0000 e 1000 que diferindo apenas no bit 3, demonstram também a segunda propriedade. Esses mesmos elementos são ainda vizinhos na seqüência binária e no hipercubo (figura II.1) e como são uma potência de dois diferem apenas em um bit. A última propriedade fica mais clara com os elementos 0001 e 1000 que são uma potência de dois e diferem nos bits 0 e 3.

A tabela II.2 apresenta os procedimentos relativos a seqüência BRGC, onde pode-se obter a identificação do processador a partir de sua posição na seqüência ou o inverso, a posição dada a identificação.

Os algoritmos da seqüência BRGC são mostrados a seguir:

identificação = gray (posição)
p = identificação do processador segundo a seqüência BRGC;
$p = (posição \gg 1)$ exor *posição*;
Retorna p.

gray ()	retorna a identificação do processador dada a posição na seqüência
ginv ()	operação inversa, retorna a posicão na seqüência dada a identificação do processador

Tabela II.2: Rotinas da Seqüência "Binary Reflected Gray Code"

posição = ginv (identificação)

p = identificação do processador;
estado = 0;
D = dimensão do hipercubo;
Se $D <> 0$
  máscara = potência (2, (D - 1));
Para todo $i = 0$ a $i < D$
  Se *estado* $\neq 0$
    $p = p$ exor *máscara*;
  estado = p and *máscara*;
Retorna p.

Supondo um hipercubo de dimensão três, temos demonstrado na tabela II.3 a posição na seqüência BRGC e a identificação do elemento correspondente à posição.

Posição	Identificação
0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

Tabela II.3: Posição e identificação dos elementos segundo a seqüência BRGC

Então se desejarmos saber a posição na seqüência BRGC do processador cuja identificação é 100 obteremos como resposta a posição 7, utilizando o procedimento *ginv()*. E para se saber qual o identificação do processador que ocupa a posição 3 na seqüência utilizamos o procedimento *gray()* e obtemos como resposta 010.

II.3 Mapeamentos

Para tornar eficiente a aplicação da biblioteca de operações paralelas, são empregados os mapeamentos em grade, árvore e anel. Dependendo da aplicação o hipercubo é visto como uma dessas topologias.

A tabela II.4 é relativa aos procedimentos de mapeamento de uma grade, uma árvore binária e um anel.

grade ()	constrói uma grade bidimensional
árvore_binária ()	constrói uma árvore
anel ()	constrói anéis de níveis 0 a $D-1$, onde D é a dimensão do hipercubo

Tabela II.4: Rotinas de Mapeamento do Hipercubo

II.3.1 Mapeamento em grade

Algumas aplicações requerem que grades periódicas sejam mapeadas em cubos. A seqüência BRGC é conveniente para esse propósito. As grades em hipercubos são construídas a partir de grades unidimensionais, com conexão de vizinhos mais próximos. Para se obter grades de dimensões maiores, é necessário que cada dimensão seja mapeada adeqüadamente e use mapeamentos unidimensionais àqueles subcubos, como pode ser visto na figura II.2.

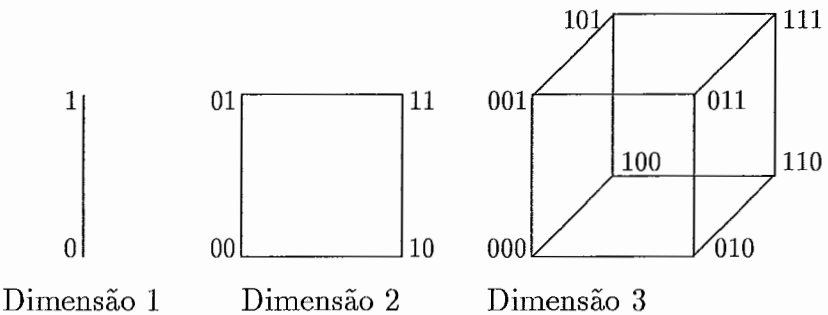


Figura II.2: Construção de uma grade de dimensão três

A estrutura de representação de uma grade bidimensional e o algoritmo de construção podem ser vistos a seguir.

estrutura em_grade {

processador ao norte

processador ao sul

processador ao leste

processador ao oeste

}

$$pt\_grade = grade\ ()$$

pt_grade = armazena área para a estrutura em grade;
x = posição do processador na direção x;
x_leste = identificação do processador $x + 1$;
x_oeste = identificação do processador $x - 1$;
y = posição do processador na direção y;
y_norte = identificação do processador $y - 1$;
y_sul = identificação do processador $y + 1$;
Retorna pt_grade.

Assim supondo um hipercubo de dimensão quatro, onde gostaríamos de ter uma grade (figura II.3) com quatro processadores na direção x e quatro na direção y, periódica em ambas as direções segundo a seqüência BRGC, teríamos para o processador $P_{(x,y)}$ cuja identificação é 0101 uma estrutura de dados preenchida da seguinte forma:

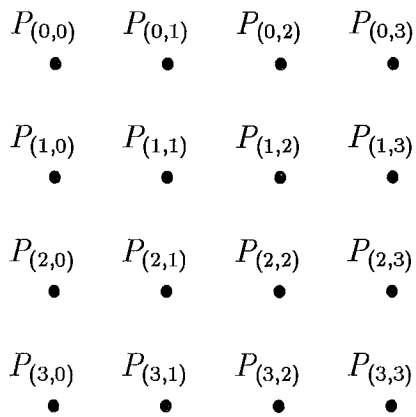


Figura II.3: Grade com quatro processadores na direção x e quatro na direção y

- processador ao norte = 0001;
- processador ao sul = 1101;
- processador ao leste = 0111;

- processador ao oeste = 0100.

A figura II.4 mostra uma grade bidimensional de um hipercubo de dimensão quatro com quatro processadores na direção x e quatro processadores na direção y.

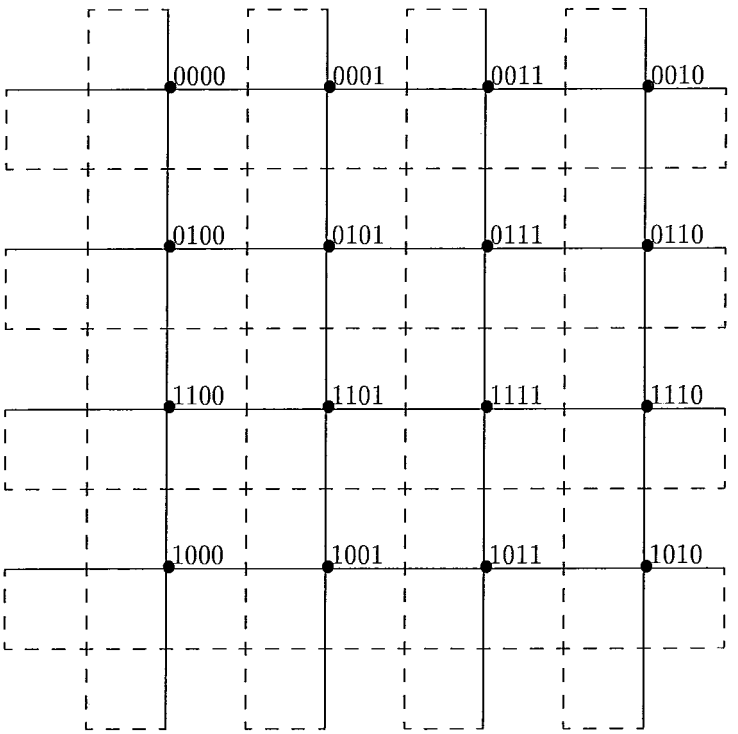


Figura II.4: Grade bidimensional de um hipercubo de dimensão quatro

II.3.2 Mapeamento em árvore

Entre as diversas maneiras de se construir uma árvore em um hipercubo, as mais comumente utilizadas são árvore binária ou árvore D-ária, onde D é a dimensão do cubo.

Uma árvore binária balanceada pode ser mapeada num hipercubo considerando o fato que um hipercubo de dimensão D pode ser construído com dois outros cubos de dimensão $D-1$ com cada processador correspondente ligado por um canal extra como foi ilustrado na figura II.1. Numerando então os dois cubos de dimensão $D-1$ como $p_0, \ldots, p_0 + 2^{D-1} - 1$, e $p_0 + 2^{D-1}, \ldots, p_0 + 2^D - 1$ respectivamente, e conectando o processador p_0 do primeiro subcubo com o processador

$p_0 + 2^{D-1}$ do segundo subcubo, a árvore binária é definida recursivamente como tendo o processador com numeração mais baixa como o raiz, e os cubos de dimensão $D-1$ como suas subárvores esquerda e direita. A tabela II.5 mostra a construção das subárvores direita e esquerda.

Dimensão	Subárvore		Raiz	
	Direita $p_0, \ldots, p_0 + 2^{D-1} - 1$	Esquerda $p_0 + 2^{D-1}, \ldots, p_0 + 2^D - 1$	Direita	Esquerda
1	0	1	0	1
2	0,1	2,3	0	2
3	$0, \ldots, 3$	$4, \ldots, 7$	0	4
4	$0, \ldots, 7$	$8, \ldots, 15$	0	8

Tabela II.5: Subárvores direita e esquerda para formação de uma arvóre binária

A figura II.5 ilustra esta árvore binária.

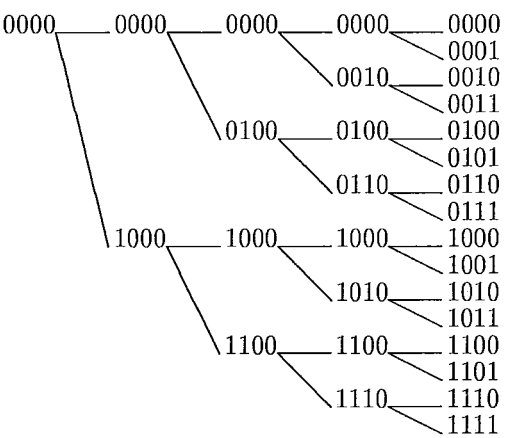


Figura II.5: Árvore binária balanceada para um hipercubo de dimensão quatro

Esta é somente uma árvore lógica já que alguns processadores ocorrem diversas vezes na árvore. Por exemplo, o processador 0000 ocorre D vezes, sendo a raiz de D subárvores. Cada processador está localizado exatamente uma vez em cada lado da árvore.

Uma alternativa é o mapeamento da rede hipercúbica numa árvore D-ária não balanceada.

No mapeamento em árvore, a rede hipercúbica é representada como uma árvore D-ária não balanceada, onde D é a dimensão do cubo. As árvores são formadas pela definição dos filhos de cada processador ou do pai de cada processador (nodo).

O filho de um nodo são os processadores cujos números de identificação são obtidos ativando-se cada bit de menor ordem na representação binária de D-dígitos. Por exemplo, na figura II.6 o nodo 000 tem como filhos 001, 010 e 100.

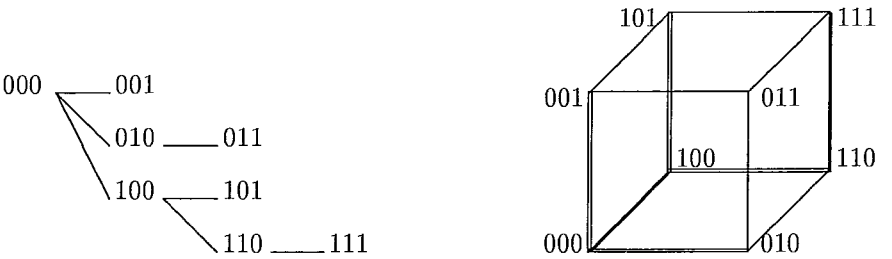


Figura II.6: Hipercubo mapeado em árvore de dimensão três

O pai de um nodo é obtido com uma operação inversa, isto é, desativando-se o bit de menor ordem na representação binária de D-dígitos do seu número de identificação. Essa árvore pode ser vista na figura II.6.

Essas duas árvores lógicas são duas maneiras diferentes de representar o mesmo conjunto de conexões. Portanto, no implemento do mapeamento em árvore em um hipercubo empregamos uma árvore D-ária.

O processador zero foi escolhido como o processador raiz por ser mais conveniente, já que este está conectado ao hospedeiro e realiza as funções de entrada e saída no hipercubo baseado em *transputers*, como citado anteriormente na seção II.1.

A seguir são mostrados a estrutura que representa uma árvore binária e o seu algoritmo de construção.

```
estrutura em_árvore {
        processador pai
        ponteiro para os processadores filhos
        }
```

pt_árvore = árvore_binária ()

pt_árvore = armazena área para a estrutura em árvore;
Se tem filhos
  pt_filhos = identificação dos processadores filhos;
Se *processador* $\neq$ *RAIZ*
  pai = identificação do processador pai;
Retorna *pt_árvore*.

Supondo então um hipercubo de dimensão três, teríamos a estrutura do processador cuja identificação é 100 preenchida com os seguintes valores:

- processador pai = 000;
- ponteiro para os processadores filhos = endereço de memória onde está armazenado num *array* os filhos 101 e 110.

II.3.3 Mapeamento em anel

É possível mapear uma estrutura de anel num hipercubo associando a cada processador a sua posição segundo a seqüência BRGC. Esta seqüência possui a propriedade de que vizinhos nesta ordenação lógica (vizinhos lógicos) são também vizinhos no hipercubo (vizinhos físicos). Os termos distância lógica e distância física são empregados sob o mesmo critério. Essa propriedade é ilustrada na figura II.7.

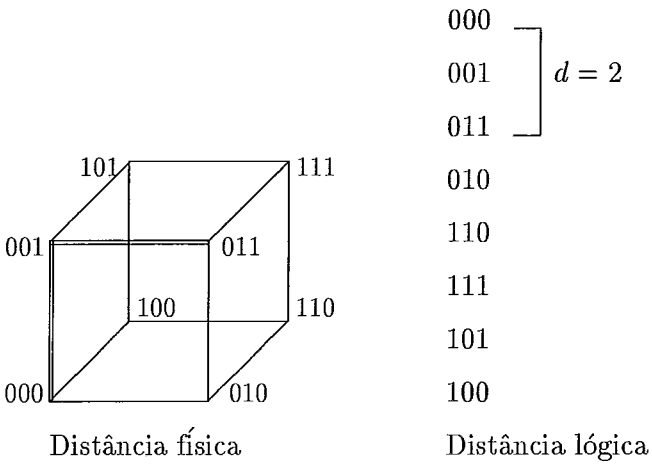


Figura II.7: Distâncias lógicas e físicas em um hipercubo de dimensão três

Em todo anel com distância lógica 2^d, onde $d = 1, \ldots, D-1$ e sendo D a dimensão do cubo, os processadores estão a uma distância física no máximo igual a 2 (Como ressaltado na figura II.7). Cada valor de d é um nível de subanel que pode ser criado no hipercubo. E todo processador está em exatamente um subanel de nível d.

Um subanel de distância lógica 2^d é composto por 2^{D-d+1} processadores, então existem $2^D/2^{D-d+1} = 2^{d-1}$ subanéis cobrindo o hipercubo no nível d. Assim para um cubo de dimensão D igual a três temos:

- níveis de subanéis possíveis $d = 1, 2$;
- nível $d = 1$;
  - distância lógica $2^d = 2$;
  - quantidade de subanéis cobrindo o hipercubo $2^{d-1} = 1$;
  - distância física = 1;
- nível $d = 2$;
  - distância lógica $2^d = 4$;
  - quantidade de subanéis cobrindo o hipercubo $2^{d-1} = 2$;
  - distância física = 2;

A figura II.8 mostra as estruturas de anéis lógicos e físicos que podem ser construídos em um hipercubo de dimensão três.

A estrutura que representa os níveis dos anéis e o algoritmo de construção dos mesmos são apresentados a seguir.

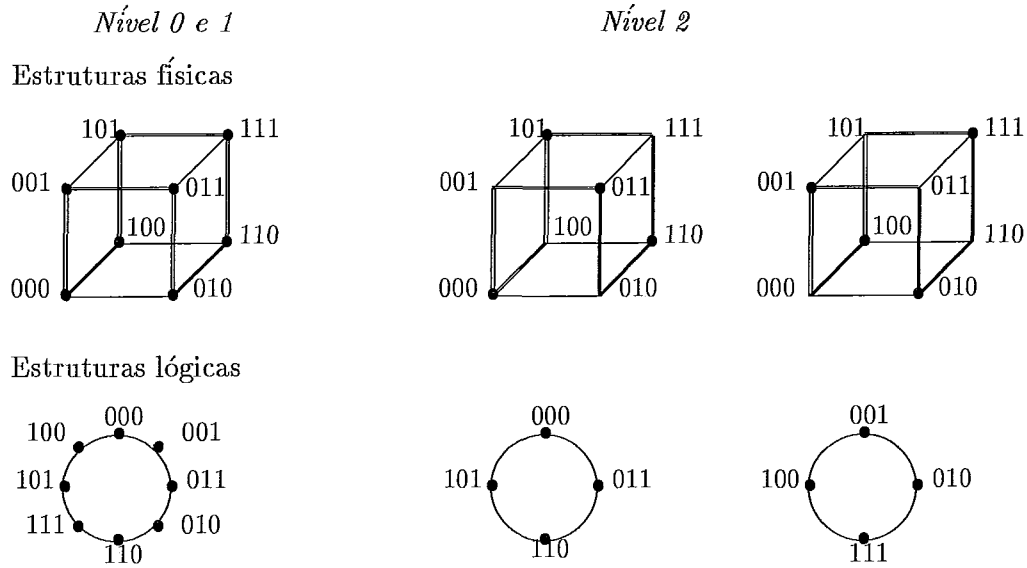


Figura II.8: Construção de anéis em um hipercubo de dimensão três

estrutura em_anel {
processador à direita no anel
processador à esquerda no anel
}

anel ()

Para todo *nível* = 0 a *nível* < D
posição = posição do processador à direita no *nível*;
p_direita = identificação do processador na posição à direita;
posição = posição do processador à esquerda no *nível*;
p_esquerda = identificação do processador na posição à esquerda;

Após a chamada dessa rotina de construção de anel temos, por exemplo, no processador cuja identificação é 000 de um hipercubo de dimensão três, uma estrutura preenchida da seguinte forma:

- no nível 0
  - processador à direita = 001;
  - processador à esquerda = 100;
- no nível 1
  - processador à direita = 001;
  - processador à esquerda = 100;
- no nível 2
  - processador à direita = 011;
  - processador à esquerda = 101;

II.4 Resumo

O multiprocessador hipercúbico baseado em *transputers* utilizado foi o NCP I desenvolvido pela COPPE que tem atualmente disponível oito *transputers* T800.

A seqüência BRGC apresentada além de fundamental para as topologias de grade, árvore e anel, também é de grande importância na distribuição, convergência e armazenamento dos dados nos processadores do sistema hipercúbico.

Os mapeamentos vistos (grade, árvore e anel) são utilizados internamente aos procedimentos que necessitam de comunicação entre os processadores.

Os procedimentos que utilizam a seqüência BRGC e os mapeamentos de grade, árvore e anel serão vistos no capítulo IV.

Capítulo III

Comunicação no Transputer

Alguns procedimentos da biblioteca de operações paralelas necessitam de comunicação entre os processadores. Toda comunicação realizada pela biblioteca de operações é feita internamente aos procedimentos tornando-a invisível. Nela está contida toda a dependência da arquitetura da rede hipercúbica. Esse capítulo relata a metodologia empregada para a troca de mensagens no multiprocessador hipercúbico baseado em *transputers*.

III.1 Os Processos

Um sistema complexo geralmente é composto por um conjunto de objetos ou dados que definem o escopo de um problema computacional. O primeiro passo na computação paralela consiste em decompor este sistema em partes (processos concorrentes) que operem de forma cooperativa para se obter uma solução em menor tempo para o problema.

Uma característica da biblioteca de operações paralelas é que cada processador recebe o mesmo código de programa, e se comunica através da troca de mensagens.

O multiprocessador hipercúbico baseado em *transputers* é uma coleção de elementos de processamento idênticos (como visto na seção II.1), chamados processadores ou nodos. Cada processador está relacionado a um único número de identificação que o diferencia dos demais. Uma vez que cada processador recebe a mesma cópia do programa (um único processo), o número de identificação é muito importante para se referenciar a um processador ou processo específico dentro da rede, pois é dado ao processo a mesma identificação do processador. Desta forma cada processador executa as mesmas instruções sobre um conjunto de dados diferentes, tornando o hipercubo, abstratamente, uma máquina virtual do tipo *Single-Instruction Multiple-Data* (SIMD).

No multiprocessador hipercúbico baseado em *transputers* não existe a facilidade direta para a entrada e saída paralela dos dados. Isso é feito de modo indireto através do processador raiz. Desse modo encontramos a necessidade de se criar um processo especial chamado processo de controle que manipula a entrada e saída de dados. Por causa de sua função especial, o processo de controle tem pouca realização computacional, tratando somente da captação, da difusão e da convergência dos dados. Este processo reside no processador raiz que está ligado ao computador hospedeiro. Todos os processos podem se referir ao processo de controle. Este também tem seu próprio número

de identificação cujo valor é igual a quantidade de processadores participantes da rede. Assim, supondo que o hipercubo tenha dimensão três, este é composto por $(2^3 = 8)$ oito processadores, cujos processadores têm números de identificação na faixa de $[0 \ldots 7]$, e o processo de controle recebe como identificador o número oito.

Os programas de aplicação neste modelo correspondem então a criação de dois processos: um de controle, que manipula a entrada e saída dos dados, e um processo igual para todos os nodos, que realizam a computação do problema propriamente dito. Esses dois processos se comunicam de maneira coordenada. A figura III.1 mostra o modo como estão interligados todos os processos em um hipercubo de dimensão três.

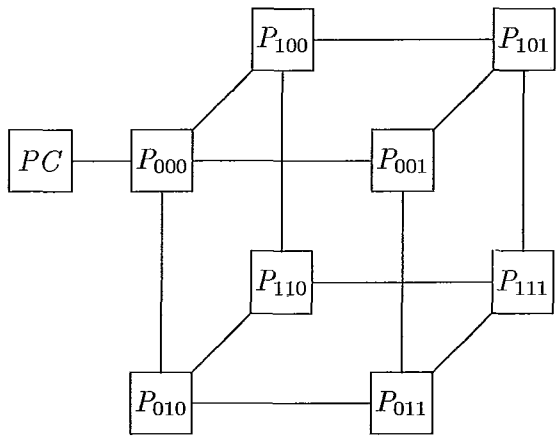


Figura III.1: Processos em um hipercubo de dimensão três

III.2 Os Canais de Comunicação

Toda comunicação entre os processadores é realizada pela troca de mensagens através de canais de comunicação. Assim, os canais são o único meio de comunicação entre os processadores. Cada canal realiza a comunicação ponto-a-ponto entre dois processadores e tem associado a ele um número de identificação. Uma maneira conveniente de distingüí-los é estabelecer uma relação entre os números de identificação dos processadores nos extremos do canal, já que estes são únicos e diferem em apenas um bit no seu número de identificação. Para tanto, utilizamos como identificador do canal o número do bit diferente entre os números de identificação dos nodos. A tabela III.1 mostra os identificadores dos canais de comunicação do processador 100 com seus processadores vizinhos em um hipercubo de dimensão três.

A figura III.2 mostra todos os canais de ligação entre os processadores de um hipercubo de dimensão três. A comunicação entre o processo de controle e o processo residente no processador raiz se dá através do canal com identificação sempre igual a dimensão do hipercubo adicionado de uma unidade. O canal com identificação igual a dimensão é deixado para a identificação dos processos como um resultado de implementação como pode ser visto no apêndice A.

Processadores	Canal
101	0
110	1
000	2

Tabela III.1: Identificadores dos canais de comunicação do processador 100 com seus vizinhos

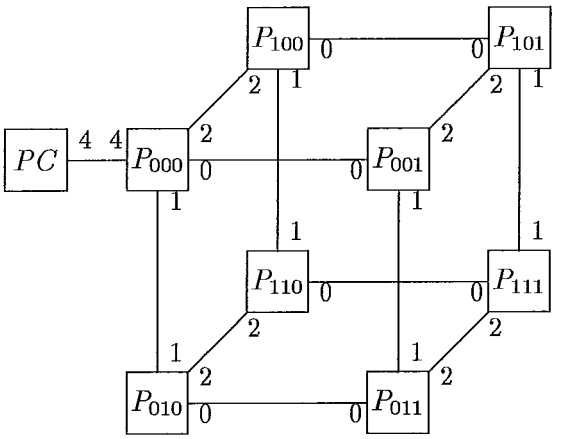


Figura III.2: Canais de ligação em um hipercubo de dimensão três

III.3 Roteamento

O multiprocessador hipercúbico baseado em *tranputers* possui um sistema de comunicação ponto-a-ponto e síncrono. A comunicação ponto-a-ponto nos leva a necessidade das mensagens precisarem ser enviadas ao longo de rotas que passam por outros processadores que não são sua origem ou seu destino. O sincronismo faz com que uma mensagem só possa ser enviada a um processador se este estiver preparado para recebê-la. Estas duas características do sistema de comunicação pode levar a uma situação de *deadlock* quando os processos não estiverem corretamente coordenados. Para que não haja *deadlock* na comunicação é necessário então se estabelecer um caminho para o envio e a recepção de mensagens de modo síncrono e coordenado.

O método de roteamento empregado nas comunicações realizadas internamente aos procedimentos da biblioteca de operações paralelas utiliza o roteamento *E-Cube*[6], que é considerado um método de caminho mais curto (*shortest path*). Se os processadores fonte e destino estão distantes K ligações, a mensagem passará exatamente por K processadores. A topologia hipercúbica binária tem uma rede de interconexão rica, com $K!$ rotas entre quaisquer dois processadores que estão distantes K ligações. Essas rotas formam uma árvore de busca de altura K com uma quantidade de folhas $K!$. A figura III.3 mostra uma árvore de busca de todas as rotas possíveis do processador zero ao processador sete em um hipercubo de dimensão três.

O caminho destacado é selecionado pelo método de roteamento *E-Cube* para enviar uma mensagem do processador zero ao processador sete. O roteamento de-

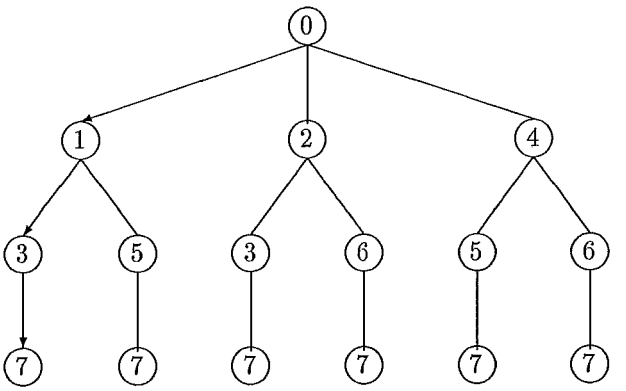


Figura III.3: Árvore de busca em um hipercubo de dimensões três

termina um caminho fixo, livre de *deadlock* entre quaisquer dois processadores usando a topologia da rede hipercúbica.

Embora qualquer rota possa ser utilizada, o roteamento *E-Cube* restringe o caminho escolhido dependendo da identificação dos processadores envolvidos na comunicação. O caminho é estabelecido trocando o bit diferente de mais baixa ordem entre os números de identificação dos processadores fonte e destino. O algoritmo *E-Cube* é mostrado a seguir:

p = identificação do processador;
p_destino = identificação do processador destino;
p_receptor = identificação do próximo processador na rota;
Se p = *p_destino*
    Libera mensagem;
    Retorna flag indicativa de sucesso;
Caso contrário
    p_receptor = troca o bit menos significativo diferente entre os números de identificação de p e *p_destino*;
    Envia mensagem ao *p_receptor*;
    Se mensagem enviada
        Retorna flag indicativa de sucesso;
    Caso contrário
        Retorna flag indicativa de erro.

Assim, para se transferir uma mensagem do processador fonte 000 ao processador destino 111 em um hipercubo de dimensão três, temos o seguinte caminho dado pela tabela III.2.

A quantidade de bits que diferem entre os números de identificação dos processadores de dois nodos dá a distância entre os nodos.

Processador Fonte	Processador Destino	Próximo Processador
000	111	001
001	111	011
011	111	111

Tabela III.2: Roteamento de uma mensagem do processador 000 ao processador 111

III.4 Envio e Recepção de Mensagens

Como mencionado anteriormente a comunicação entre os processadores é realizada pelo envio de mensagens através dos canais de comunicação que conectam pares de processadores. Toda comunicação entre processadores deve ter operações recíprocas envolvendo transmissão e recepção coordenadas. Estes procedimentos de transmissão e recepção devem ser complementares.

O procedimento criado para transmitir uma mensagem formata um identificador da mensagem com informações sobre os processadores fonte e destino e tamanho da mensagem. Transfere os dados da mensagem para um *buffer*, define o próximo processador na rota estabelecida e o canal pelo qual a mensagem deve ser enviada. Envia o identificador da mensagem seguida então da mensagem propriamente dita. Por fim, libera o *buffer* utilizado para armazenar a mensagem.

O procedimento de recepção complementar ao de transmissão, recebe inicialmente o identificador da mensagem e a mensagem em um *buffer*. Verifica se a mensagem chegou ao destino retransmitindo-a, se necessário, a um outro processador no caminho estabelecido pelo procedimento de roteamento. Depois de recebida a mensagem libera o *buffer* utilizado na transferência.

A tabela III.3 mostra as rotinas implementadas que combinadas realizam a troca de mensagens no sistema *transputer*. O apêndice A apresenta, passo a passo, a implementação e os algoritmos dessas rotinas.

canal_de_escrita ()	canal por onde a mensagem deve ser enviada
canal_de_leitura ()	canal por onde a mensagem deve ser recebida
roteamento ()	próximo processador a receber a mensagem
envia_mensagem ()	
recebe_mensagem ()	

Tabela III.3: Rotinas Relativas à Comunicação

III.5 Resumo

Na solução de problemas utilizando a biblioteca de operações paralelas cada processador deve receber um único processo. Esse está relacionado a um número de identificação que o diferencia dos demais. Além desses processos, existe um processo de controle que reside no processador raiz, que manipula a entrada e a saída dos dados.

Toda comunicação entre processadores é realizada pela troca de mensagens através de canais de comunicação que também são diferenciados por números de identificação.

A comunicação entre processadores é realizada através de rotas pré-estabelecidas, baseadas no algoritmo de roteamento *E-Cube*.

Os procedimentos para a troca de mensagens são síncronos e coordenados para evitar *deadlock*.

Capítulo IV

Organização de Memória

As várias operações que compõem a biblioteca de operações paralelas são realizadas com todos os processadores chamando simultaneamente a rotina apropriada com seus argumentos e operando sobre os seus segmentos locais, ignorando a existência de seus vizinhos na rede. O tamanho do vetor ou a dimensão da matriz deve ser maior ou pelo menos igual ao número de processadores participantes da rede para que todos tenham pelo menos um elemento do vetor ou da matriz. Neste capítulo descreveremos as estruturas de dados que representam os vetores e as matrizes, as formas em que eles podem ser armazenados e os passos elementares que compõem a biblioteca de operações paralelas.

IV.1 Rotinas Vetoriais Básicas

Todos os processadores recebem uma cópia da biblioteca de operações paralelas e do programa de aplicação, bem como os vetores distribuídos adeqüadamente. As rotinas são chamadas simultaneamente por todos os processadores e operam sobre os seus segmentos locais, ignorando a existência de seus vizinhos na rede.

Os vetores são representados por uma estrutura de dados que contém todas as informações necessárias à sua manipulação dentro das rotinas. Esta estrutura é mostrada a seguir:

estrutura vetorial {
tipo = tipo de armazenamento vetorial;
h = tamanho do segmento vetorial local ao processador;
tamanho = tamanho total do vetor;
pt_início = ponteiro do início da área de dados;
pt_fim = ponteiro do fim da área de dados;
pt_antes = ponteiro auxiliar de deslocamento;
pt_depois = ponteiro auxiliar de deslocamento;
}

O campo *tipo* especifica a forma de armazenamento do vetor. Vetores com diferentes tipos de armazenamento podem ser misturados livremente, quando necessário as conversões apropriadas podem ser aplicadas. Os dois tipos existentes são *SIMPLE* e *SHIFT*, cuja definição pode ser vista na seção IV.1.1.

Os campos h e *tamanho* são relativos ao tamanho do segmento local ao processador e ao tamanho total do vetor. Os outros campos são ponteiros para facilitar a manipulação da área de dados.

Algumas rotinas vetoriais não utilizam comunicação entre os processadores e são trivialmente paralelizáveis. Fazem parte deste grupo as rotinas de armazenamento e de liberação de memória, conversão entre os tipos de armazenamento, cópia vetorial, soma e multiplicação entre vetores e escalares. A operação correspondente é realizada em cada processador sobre o segmento local. Desse grupo somente o procedimento de armazenamento vetorial será apresentado detalhadamente neste capítulo. As rotinas onde há a necessidade de comunicação, como a de deslocamento vetorial e a de produto interno entre outras, serão vistas em detalhes nas seções seguintes. As rotinas vetoriais e suas funções são apresentadas na tabela IV.1. Todos os procedimentos vetoriais estão descritos no apêndice B.

aloca_vetor	armazena segmento local de um vetor
deleta_vetor	libera memória atribuída a um vetor
converte_vetor	troca o tipo de armazenamento
vetor_nulo	$u_i = 0$
copia_vetor	$u_i = v_i$
soma_vet_a_esc_vet	$u_i = u_i + sv_i$
soma_esc_vet_a_vet	$u_i = su_i + v_i$
soma_vet_a_vet_vet	$u_i = u_i + v_i w_i$
desloca_vetor	$u_i = u_{(i-s) \bmod N}$
desloca_vetor_ótimo	$u_i = u_{(i-s) \bmod N}$
máximo_vetor	$max_{0 \leq i < N} u_i$
soma_vetor	$\sum_{i=0}^{N-1} u_i$
produto_interno	$\sum_{i=0}^{N-1} u_i v_i$

Tabela IV.1: Rotinas Vetoriais

IV.1.1 Armazenamento Vetorial

Para distribuir vetores aos P processadores da rede hipercúbica, consideramos somente aqueles vetores cujo tamanho é igual ou maior a P. Assim todos os processadores receberão pelo menos um elemento do vetor.

Um vetor distribuído desta maneira é chamado vetor distribuído e todos os vetores do mesmo tamanho serão sempre distribuídos da mesma forma aos processadores, isto é, segmentos de vetores diferentes residindo no mesmo processador têm o mesmo tamanho. A fórmula para a distribuição de um vetor de tamanho N aos P processadores que compõem a rede hipercúbica é:

$$N = hP + r, \quad 0 \leq r < P.$$

Essa fórmula garante que cada processador tenha um segmento de tamanho igual a pelo menos $h = N/P$ elementos do vetor, caso o tamanho do vetor seja um múltiplo do número de processadores. Se o tamanho do vetor não for um múltiplo do número de processadores, então aos r primeiros processadores na seqüência BRGC (secão II.2) serão atribuídos os segmentos de tamanho $h + 1$ elementos e aos $P - r$ processadores restantes serão atribuídos segmentos de tamanho h elementos.

Existem dois tipos de representação vetorial que são chamados tipo *SIMPLE* e *SHIFT*. Na forma *SIMPLE*, cada segmento vetorial é armazenado em um *array* de tamanho igual a h ou $h+1$. No formato *SHIFT*, cada segmento local é armazenado em um *buffer* de tamanho $2h$ dentro do qual será permitido deslocamentos vetoriais como serão vistos mais adiante na seção IV.1.2. Inicialmente o segmento é centralizado no *buffer*.

As figuras IV.1 e IV.2 demonstram como um vetor de tamanho total igual a vinte elementos pode ser distribuído em um hipercubo de dimensão três, tanto para o tipo *SIMPLE* quanto para o tipo *SHIFT*.

$h = N/P$	$r = N \bmod P$
$h = 20/8$	$r = 20 \bmod 8$
$h = 2$	$r = 4$

Os quatro primeiros processadores na seqüência BRGC recebem três elementos do vetor e os outros quatro restantes recebem dois elementos.

IV.1.2 Deslocamento Vetorial

Esta operação é essencial uma vez que várias outras operações a utilizam. A operação de deslocamento vetorial consiste em deslocar cada elemento do vetor distribuído m vezes, enviando m elementos para o processador seguinte e recebendo m elementos do processador precedente. Para esta operação é necessário que cada processador contenha pelo menos um elemento do vetor.

Existem dois procedimentos para realizar o deslocamento vetorial, o *desloca_vetor* e o *desloca_vetor_ótimo*. Esses procedimentos vêem a rede hipercúbica como uma hierarquia de anéis e exigem que o vetor envolvido na operação esteja armazenado

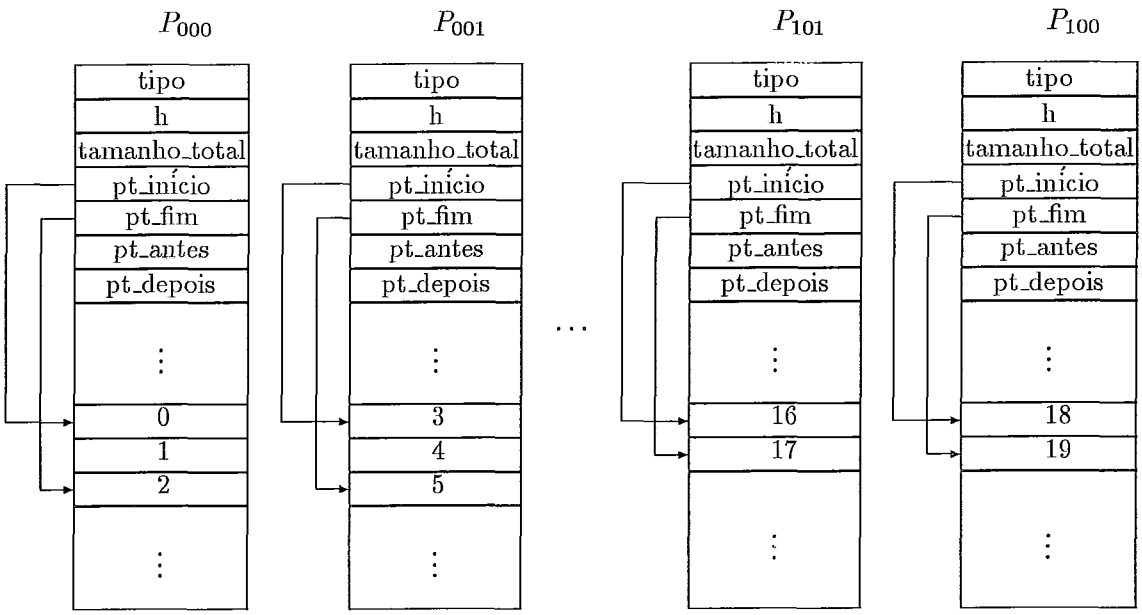


Figura IV.1: Vetor distribuído do tipo *SIMPLE* em um hipercubo de dimensão três

no formato *SHIFT*, isto é, tenha uma memória extra em cada processador, um *buffer* de extensão $h/2$ para cada lado do segmento.

• **Procedimento** *desloca_vetor*

O procedimento *desloca_vetor* é utilizado quando o tamanho total do vetor não é um múltiplo do número de processadores. Para deslocamentos pequenos como o de um elemento, os deslocamentos são feitos no anel de nível 0 e os vetores então em vez de serem deslocados são simplesmente deixados no lugar e, adiciona-se o novo elemento vindo do processador vizinho na posição apropriada do vetor local. O ponteiro do início do vetor na estrutura de dados correspondente é incrementado para cada deslocamento. Após $h/2$ deslocamentos elementares, se faz necessária uma operação para recolocar o vetor no seu ponto inicial em cada processador. A colocação na posição inicial é automaticamente detectada e realizada quando o ponteiro auxiliar de deslocamento alcança uma das extremidades do *buffer*.

As figuras IV.3, IV.4 e IV.5 ilustram as memórias dos processadores durante a operação de deslocamento de um elemento em um vetor distribuído de tamanho total dez em um hipercubo de dimensão dois. A figura IV.3 ilustra a área de dados da memória de cada processador com a posição inicial dos elementos do vetor distribuído.

A figura IV.4 mostra o novo elemento vindo do processador vizinho no anel e adicionado na posição apropriada.

A figura IV.5 ilustra a área de dados da memória de cada processador com a posição final do vetor com seus elementos já centralizados.

• **Procedimento** *desloca_vetor_ótimo*

A operação de deslocamento vetorial otimizada é utilizada somente quando o vetor é um múltiplo do número de processadores ($N = hP$). Se consistir de muitos des-

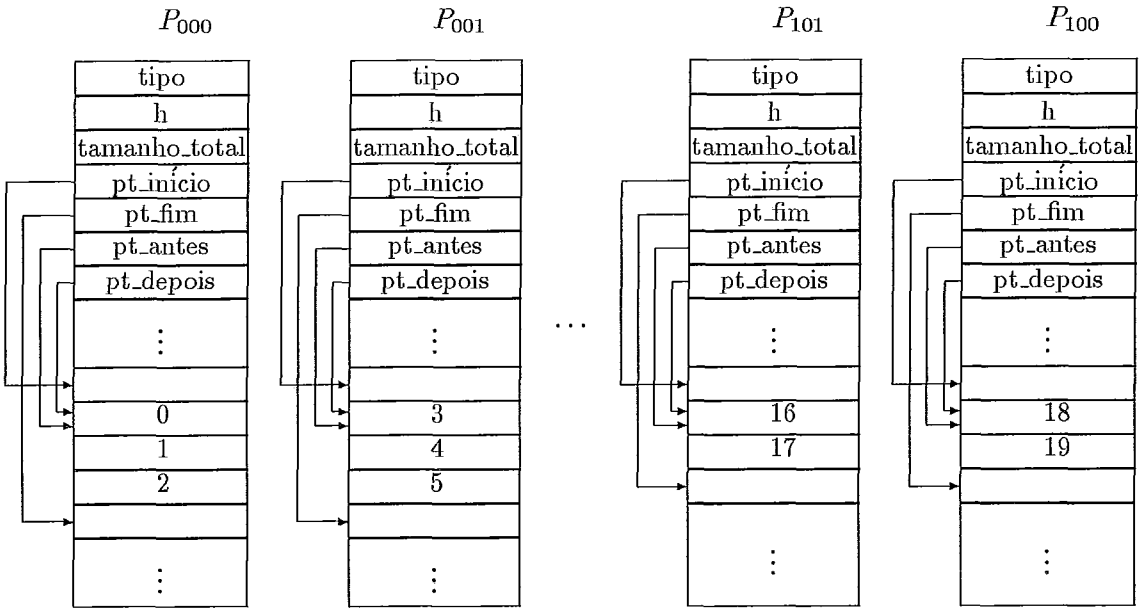


Figura IV.2: Vetor distribuído do tipo *SHIFT* em um hipercubo de dimensão três

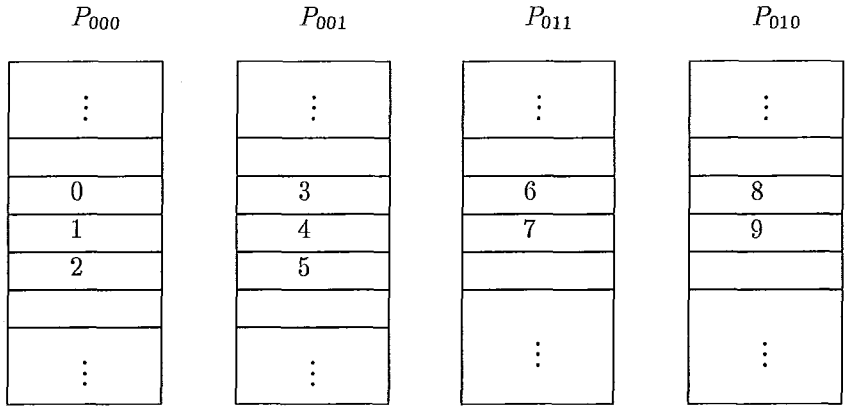


Figura IV.3: Posição inicial

locamentos de um elemento em sucessão, esta operação pode ser feita mais eficientemente utilizando a hierarquia de anéis em vários níveis ao invés de só no nível 0 como visto anteriormente em *desloca_vetor*.

Uma grande quantidade de deslocamentos m, pode ser dividida em quantidade de deslocamentos de segmento m_s e em quantidade de deslocamentos de elementos m_e. Então podemos escrever que

$$m = m_s h + m_e, \quad 0 \leq m_e < h,$$

onde h é o tamanho do segmento local ao processador, igual em todos os processadores já que $N = hP$.

Uma operação física de comunicação de transferência de elementos e transferência de segmentos aos processadores vizinhos seguem o mesmo critério. Então um deslocamento de segmento resulta em cada processador transmitir o seu segmento local completo ao processador seguinte no anel e receber um segmento completo do processador precedente no anel.

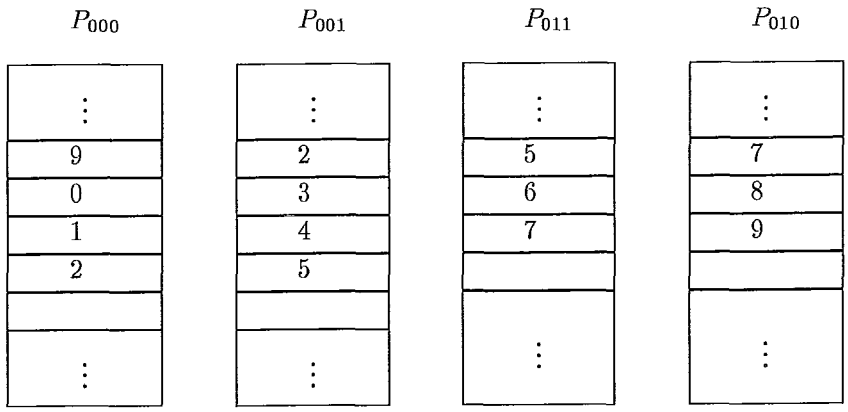


Figura IV.4: Deslocamento de elemento

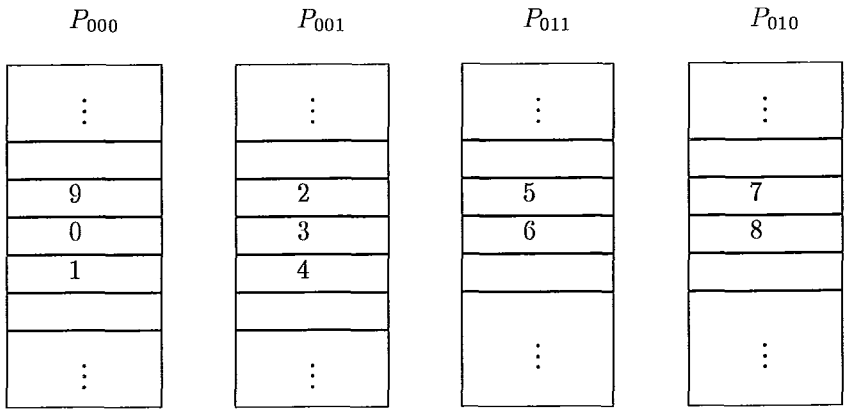


Figura IV.5: Vetor deslocado

Se o deslocamento m é muito grande podemos ter então $m_s \geq P$. Se $m_s \geq P$ podemos reduzir o deslocamento de segmentos fazendo $m_s = m_s \bmod P$, uma vez que deslocar de P segmentos é equivalente a um deslocamento de $N = hP$ elementos e cujo efeito é de não haver nenhum deslocamento. Uma outra redução é possível observando quando $m_s > P/2$. Nesse caso, é melhor deslocar os segmentos na direção oposta $m_s - P$ vezes. Se todas as reduções forem feitas em m_s então teremos deslocamentos somente na faixa de $-P/2+1, \ldots, P/2$.

Os m_e deslocamentos de elementos restantes são feitos como descrito anteriormente em *desloca_vetor*.

Para aumentar a velocidade dos deslocamentos de segmentos empregamos a hierarquia de anéis. Considere a representação binária de m_s dada por $b_{D-1} \ldots b_1 b_0$. Para cada $i > 0$ tal que b_i está ativado, necessitamos realizar 2^i deslocamentos de segmento, que reduz-se a apenas um deslocamento de segmento no subanel lógico de nível i. Como visto na seção II.3.3, isto é feito com no máximo dois deslocamentos físicos no subanel de nível i. Isto tem o efeito de realizar 2^i deslocamentos de segmento levando o tempo de somente duas transferências físicas de segmento. O bit 0 é uma excessão porque quando ativo requer somente um deslocamento no anel principal, levando o tempo de uma transferência de segmento. Assim no pior caso teremos todos os $D-1$ bits em m_s ativos.

Como exemplo, suponhamos um cubo de dimensão dois, um vetor de tamanho vinte e um deslocamento de 77 vezes. O procedimento então seria:

tamanho do segmento local	deslocamento de segmentos	deslocamento de elementos
$h = N/P$	$m = m_s h + m_e$	$m_e = m \bmod h$
$h = 20/4$	$m_s = 77/5$	$m_e = 77 \bmod 5$
$h = 5$	$m_s = 15$	$m_e = 2$

Realizando as reduções no deslocamento de segmentos temos:

$m_s \geq P$	$m_s > P/2$
$m_s = m_s \bmod P$	$m_s = m_s - P$
$m_s = 15 \bmod 4$	$m_s = 3 - 4$
$m_s = 3$	$m_s = -1$

Resumindo, o deslocamento total foi subdividido em um deslocamento de segmento na direção oposta mais dois deslocamentos de elementos individuais. A figuras IV.6, IV.7, IV.8 e IV.9 demonstram os passos realizados no deslocamento. A figura IV.6 mostra a área de dados da memória de cada processador com a posição inicial dos elementos do vetor distribuído.

Na figura IV.7 é dada a posição dos elementos do vetor após o deslocamento de um segmento na direção oposta. O deslocamento foi realizado no nível 0 do anel, já que a representação binária correspondente é 01, e somente o bit 0 está ativado.

A figura IV.8 mostra a transferência dos dois elementos ao processador vizinho no anel.

A figura IV.9 apresenta então a posição final dos elementos do vetor distribuído.

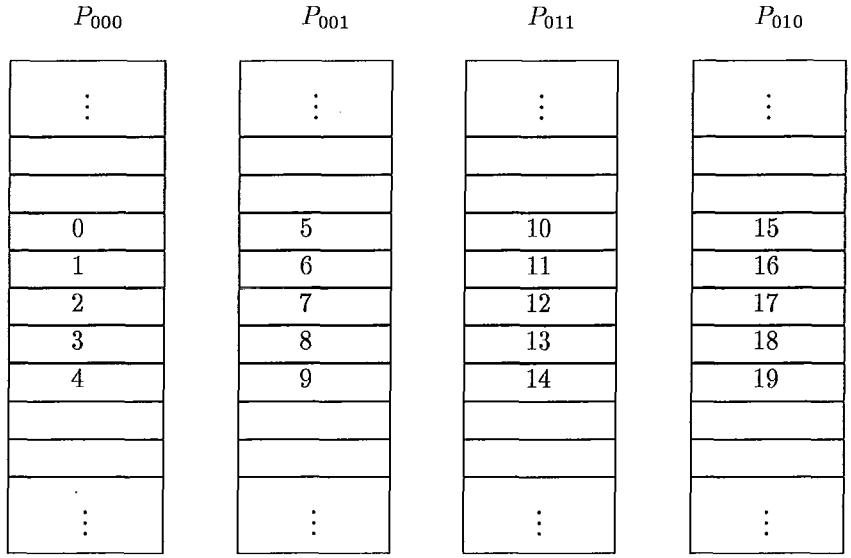


Figura IV.6: Posição inicial

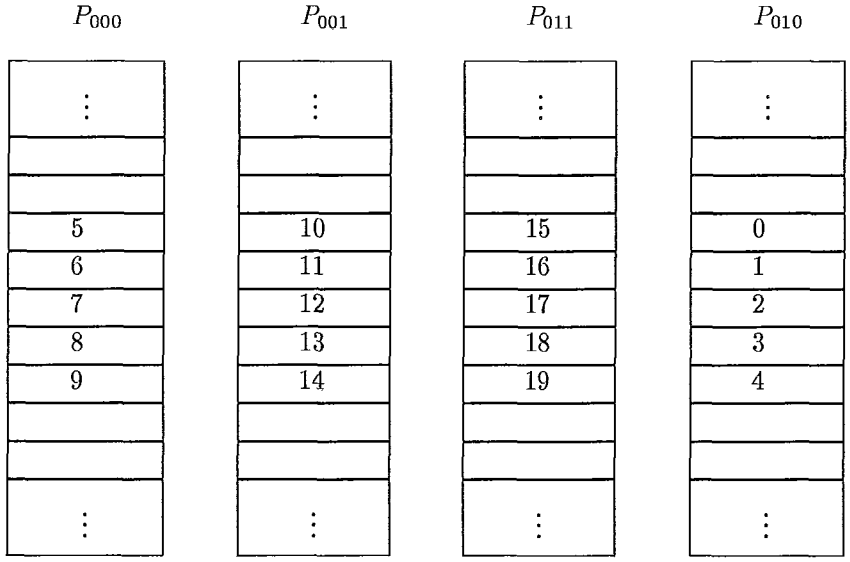


Figura IV.7: Deslocamento de segmento

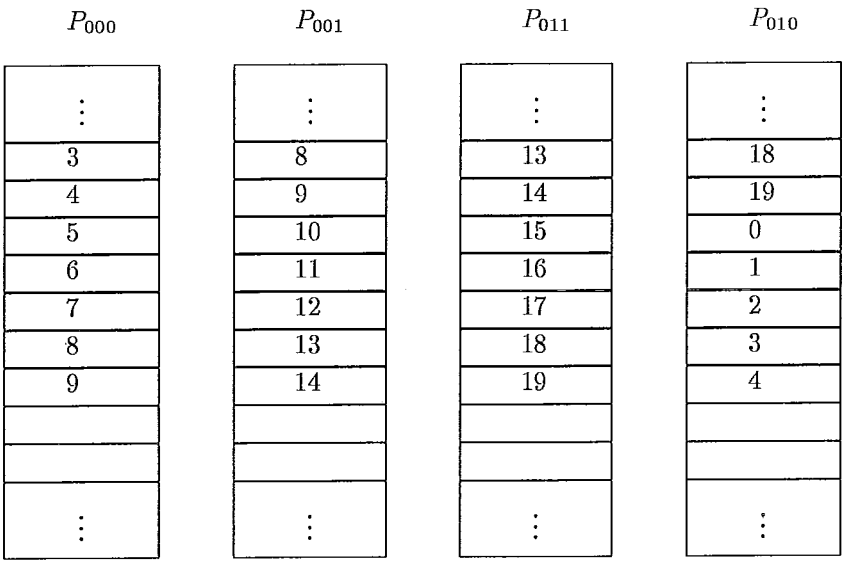


Figura IV.8: Deslocamento de elemento

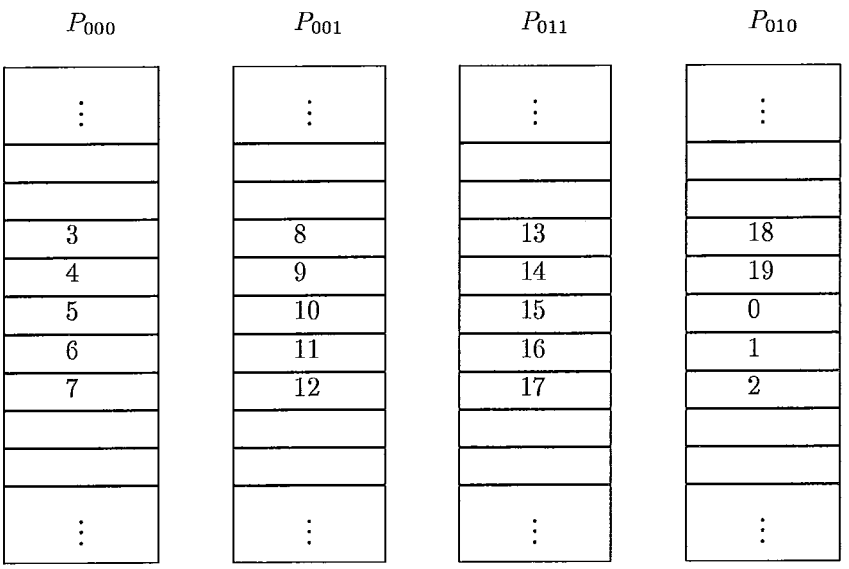


Figura IV.9: Vetor deslocado

IV.1.3 Elemento Máximo de um Vetor Distribuído

Para se obter o elemento máximo em um vetor basta que todos os processadores simultaneamente utilizem o procedimento *máximo_vetor*. Ao final desta operação todos os processadores terão o valor do elemento máximo do vetor distribuído. Nenhum processador é distingüido nesta semântica.

A implementação naturalmente distingüe os processadores. A operação realiza comunicação entre os processadores. Inicialmente cada processador encontra o seu elemento máximo do segmento local. Uma árvore D-ária é mapeada à rede hipercúbica, com o processador 0 como raiz (como visto na seção II.3.2). Os processadores se comunicam então com o pai e com os filhos e realizam novas comparações até que todos obtenham o mesmo resultado. O procedimento *máximo_vetor* tem o seu algoritmo mostrado a seguir.

D = dimensão do hipercubo;
pt_árvore = árvore_binária ();
máximo = máximo_vetor_local (pte_vetor);
Para $i = 0$ a $i < D$ faça
    Se tem filhos
        Recebe *máximo* dos filhos;
    Se tem pai
        Envia *máximo* ao pai;
    máximo = maior elemento entre os máximos local e dos filhos;
Para $i = 0$ a $i < D$ faça
    Se tem filhos
        Envia *máximo* aos filhos;
    Se tem pai
        Recebe *máximo* do pai;
Retorna *máximo*.

O custo dessa implementação é $O(N/P)+O(logP)$. Todas as comparações dos segmentos locais são realizadas em paralelo em um tempo $O(N/P)$, proporcional ao tamanho do segmento local $h = N/P$. As comparações parciais sobre uma árvore D-ária podem ser computadas em um tempo D, se a unidade de tempo consiste numa comparação de um número real mais a comunicação deste número ao seu vizinho no hipercubo.

Para comprovar esta afirmação note que uma árvore D-ária não balanceada com raiz no nodo 000..0 contém $(D-1)$-ária,$D-2$-ária,...,0-ária subárvores com seus filhos 100..0, 010..0, 001..0, respectivamente. Pelo princípio de indução assumimos que o resultado é verdadeiro para cada subárvore de $0, 1, \ldots, (D-1)$. A hipótese é verdade para $D = 0$ e $D = 1$. Pela hipótese de indução, a comparação do nodo 100..0 chegará ao 000..0 em um tempo $D-1$, do nodo 010..0 em um tempo $D-2$ e assim por diante. Cada comparação parcial pode ser realizada em 000..0 enquanto chegam as próximas comparações parciais. Assim todas as comparações serão completadas em um tempo D, completando a indução. A difusão do resultado aos processadores individuais seguem os passos acima e requerem um tempo $O(D)$.

A figura IV.10 e a figura IV.11 mostram o mapeamento em árvore empregado internamente à rotina e os passos do procedimento para achar o elemento máximo do vetor distribuído, respectivamente, em um hipercubo de dimensão dois.

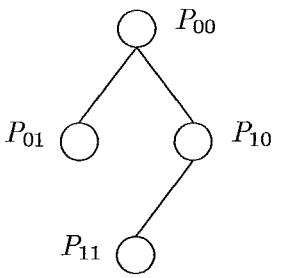


Figura IV.10: Mapeamento em árvore em um hipercubo de dimensão dois

	P_{00}	P_{01}	P_{11}	P_{10}
	max_0	max_1	max_3	max_2
1	max_1 max_2			max_3
2	$max_{0,1,2}$	max_1	max_3	$max_{2,3}$
1	max_1 $max_{2,3}$			max_3
2	$max_{0,1,2,3}$	max_1	max_3	$max_{2,3}$
3		$max_{0,1,2,3}$	$max_{2,3}$	$max_{0,1,2,3}$
3		$max_{0,1,2,3}$	$max_{0,1,2,3}$	$max_{0,1,2,3}$

1 Comunicação dos valores máximos parciais entre pais e filhos.
2 Cálculo do novo valor.
3 Envio do valor final aos filhos.

Figura IV.11: Elemento máximo de um vetor

IV.1.4 Soma e Produto Interno

De forma semelhante ao procedimento *máximo_vetor* podemos especificar as operações de somar todos os elementos de um vetor (*soma_vetor*) e de produto interno (*produto_interno*). Esses procedimentos utilizam a mesma topologia para troca de mensagens vendo o hipercubo como uma árvore D-ária. Trabalham inicialmente sobre os seus segmentos locais e comunicam seus dados parciais até que todos obtenham o mesmo resultado, ou o somatório de todos os elementos do vetor distribuído ou o produto interno.

IV.2 Rotinas Matriciais

Assim como os vetores, as matrizes são representadas por estruturas de dados que contém informações necessárias à sua manipulação pelas rotinas. Esta estrutura é definida a seguir:

estrutura matricial {
tipo = tipo de armazenamento matricial;
linha = quantidade de linhas local ao processador;
coluna = quantidade de colunas local ao processador;
linha_total = quantidade total de linhas da matriz;
coluna_total = quantidade total de colunas da matriz;
lin_esparsa = quantidade de linhas não nulas da matriz;
pt_lin_esparsa = ponteiro para *array* de linhas não nulas;
col_esparsa = quantidade de colunas não nulas da matriz;
pt_col_esparsa = ponteiro para *array* de colunas não nulas;
diag_esparsa = quantidade de diagonais não nulas da matriz;
pt_diag_esparsa = ponteiro para *array* de diagonais não nulas;
p_x = quantidade de processadores na direção x;
p_y = quantidade de processadores na direção y;
per_x = periodicidade na direção x;
per_y = periodicidade na direção y;
b = limite simulado;
pt_matriz = ponteiro da área de dados onde está armazenada a matriz;
}

Esta estrutura permite que matrizes com diferentes formas de armazenamento sejam manipuladas consistentemente, uma vez que cada processador tem disponível a representação dos dados com os quais está trabalhando. Existem seis tipos diferentes de armazenamento matricial chamados *LINHA_CONTÍGUA*, *LINHA_DISTRIBUÍDA*, *COLUNA_CONTÍGUA*, *COLUNA_DISTRIBUÍDA*, *DIAGONAL* e *ÁREA*. Estes tipos de armazenamento serão exemplificados na seção IV.2.1.

Os campos *lin_esparsa*, *col_esparsa* e *diag_esparsa* são indicadores da esparsidade da matriz. Se o valor contido nesses campos for diferente de zero significa que a matriz é esparsa e contém a quantidade de linhas, colunas ou diagonais não nulas indicadas. Os campos *pt_lin_esparsa*, *pt_col_esparsa* e *pt_diag_esparsa* são ponteiros para os *arrays* de linhas, colunas ou diagonais não nulas. A forma como as matrizes esparsas são armazenadas é adeqüada à classe de matrizes cuja estrutura de esparsidade tem a forma de matrizes de banda, onde os elementos não nulos estão ao redor da diagonal principal.

Para o tipo de armazenamento por *ÁREA* (por bloco) as informações de esparsidade são ignoradas, porém contém outras informações relevantes obtidas através do procedimento *parâmetros_de_área* que indicam a periodicidade nas direções x e y, o limite simulado que envolve cada submatriz entre outras.

A tabela IV.2 apresenta os procedimentos matriciais que estão disponíveis na biblioteca de operações paralelas.

O apêndice C apresenta em detalhes a implementação dos prodecimentos matriciais.

aloca_matriz	armazena segmento local de uma matriz
deleta_matriz	libera memória de uma matriz
converte_matriz	troca o tipo de armazenamento matricial
transposição_por_deslocamento	transposição matricial
transposição_por_bloco	transposição matricial
matriz_vetor	multiplicação de uma matriz por um vetor
matriz_matriz	multiplicação de matrizes
rank1_update	$A \leftarrow A + xy^T$
shuffle	troca os limites de área com os vizinhos

Tabela IV.2: Rotinas Matriciais

IV.2.1 Armazenamento Matricial

A maneira mais simples de se armazenar uma matriz é orientá-la por linhas ou por colunas. Existem seis tipos diferentes disponíveis de armazenamento matricial: *LINHA_CONTÍGUA*, *LINHA_DISTRIBUÍDA*, *COLUNA_CONTÍGUA*, *COLUNA_DISTRIBUÍDA*, *DIAGONAL* e *ÁREA*. A forma como são realizados estes armazenamentos são mostrados a seguir.

- *LINHA_CONTÍGUA*

Para armazenar uma matriz por linhas contíguas dividi-se a quantidade de linhas da matriz pela quantidade de processadores existentes na rede hipercúbica. Assim uma matriz $A_{N \times N}$ é decomposta em:

$$N = hP + r, \quad 0 \leq r < P.$$

Cada linha é armazenada inteiramente em um processador. Aos r primeiros processadores na seqüência BRGC (seção II.2) são atribuídas $h+1$ linhas consecutivas da matriz, que é armazenada contiguamente como uma matriz $A_{(h+1) \times N}$. Aos $P-r$ processadores restantes são atribuídas h linhas consecutivas, armazenadas contiguamente em uma matriz $A_{h \times N}$. A seguir é monstrado como uma matriz $A_{10 \times 10}$ é armazenada em um hipercubo de dimensão dois.

$$\begin{array}{ll} h = N/P & r = N \bmod P \\ h = 10/4 & r = 10 \bmod 4 \\ h = 2 & r = 2 \end{array}$$

Aos dois primeiros processadores serão atribuídas três linhas consecutivas a cada um e aos outros dois restantes serão atribuídas duas linhas consecutivas.

	a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	a_{06}	a_{07}	a_{08}	a_{09}
P_{00}	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
P_{01}	a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
	a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
	a_{60}	a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
P_{11}	a_{70}	a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
	a_{80}	a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
P_{10}	a_{90}	a_{91}	a_{92}	a_{93}	a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

- *LINHA_DISTRIBUÍDA*

Na representação de uma matriz por linhas distribuídas, cada linha é armazenada como um vetor distribuído, com segmentos em cada processador de linhas consecutivas armazenadas contiguamente. Dessa forma uma matriz $A_{N\times N}$ é decomposta em matrizes $A_{N\times(h+1)}$ ou $A_{N\times h}$. Neste tipo de armazenamento cada processador tem uma coluna inteira, embora não contiguamente. A seguir é mostrada uma matriz $A_{10\times10}$ armazenada em um hipercubo de dimensão dois.

P_{00}			P_{01}			P_{11}		P_{10}	
a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	a_{06}	a_{07}	a_{08}	a_{09}
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
a_{60}	a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
a_{70}	a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
a_{80}	a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
a_{90}	a_{91}	a_{92}	a_{93}	a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

- *COLUNA_CONTÍGUA*

A distribuição de uma matriz por colunas contíguas segue o mesmo critério em relação as colunas que a distribuição por linhas contíguas. Cada processador tem colunas inteiras armazenadas contiguamente. Assim, a matriz é armazenada em submatrizes $A_{N\times(h+1)}$ ou $A_{N\times h}$. A seguir é mostrada uma matriz $A_{10\times10}$ distribuída por colunas contíguas.

P_{00}			P_{01}			P_{11}		P_{10}	
a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	a_{06}	a_{07}	a_{08}	a_{09}
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
a_{60}	a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
a_{70}	a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
a_{80}	a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
a_{90}	a_{91}	a_{92}	a_{93}	a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

Os elementos contidos em cada processador são iguais tanto no tipo *LINHA_DISTRIBUÍDA* quanto no tipo *COLUNA_CONTÍGUA*, porém são organizados de modo diferente na memória como no exemplo ilustrado na figura IV.12, que mostra uma parte da memória do processador zero com uma matriz 10 x 10 em um cubo de dimensão dois.

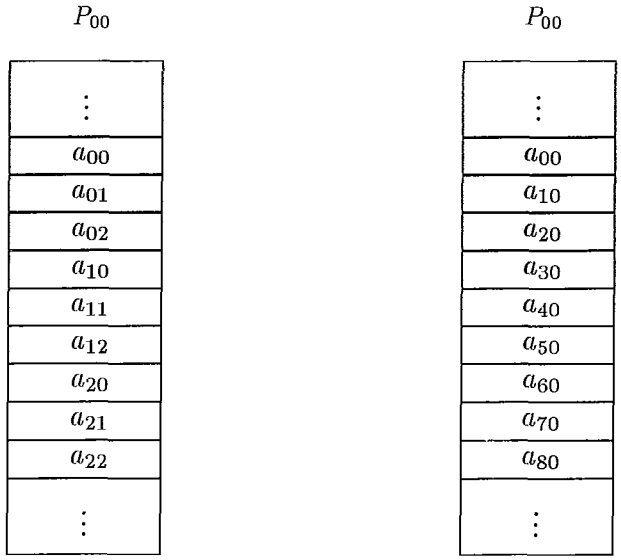


Figura IV.12: *LINHA_DISTRIBUÍDA* e *COLUNA_CONTÍGUA*

- *COLUNA_DISTRIBUÍDA*

Este tipo de armazenamento também segue o mesmo critério utilizado para distribuir uma matriz por linhas distribuídas em relação as colunas distribuídas. Desse modo temos então uma matriz $A_{N\times N}$ armazenada em submatrizes $A_{(h+1)\times N}$ ou $A_{h\times N}$. A seguir é mostrada esta distribuição.

	a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	a_{06}	a_{07}	a_{08}	a_{09}
P_{00}	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
P_{01}	a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
	a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
	a_{60}	a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
P_{11}	a_{70}	a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
	a_{80}	a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
P_{10}	a_{90}	a_{91}	a_{92}	a_{93}	a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

Da mesma forma como ocorre com os tipos *LINHA_DISTRIBUÍDA* e *COLUNA_CONTÍGUA*, os elementos contidos em cada processador são iguais tanto no tipo *LINHA_CONTÍGUA* quanto no tipo *COLUNA_DISTRIBUÍDA*, porém são organizados de modo diferente na memória como no exemplo ilustrado na figura IV.13. Nesse exemplo, é ilustrado parte da memória do processador zero com uma matriz 10 x 10 em um cubo de dimensão dois.

- *DIAGONAL*

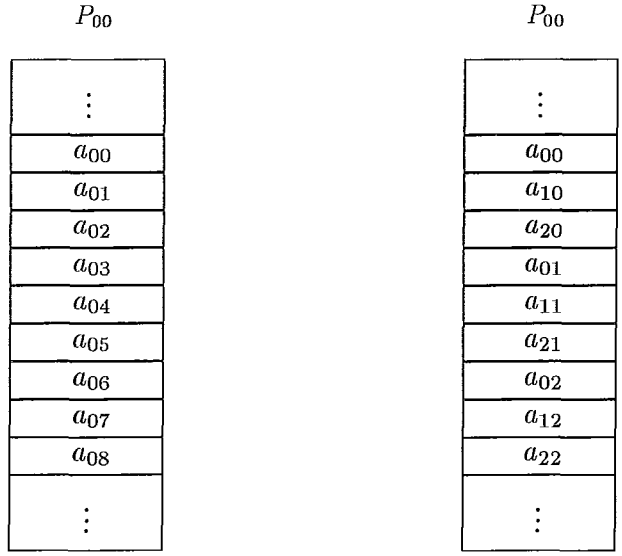


Figura IV.13: *LINHA_CONTÍGUA* e *COLUNA_DISTRIBUÍDA*

Esta forma de armazenamento é especialmente conveniente para matrizes que tem uma pequena largura de banda ou com algumas poucas diagonais não nulas. Consideremos uma matriz $A_{N\times N}$ com linhas indexadas por i, colunas por j e ambos i e j na faixa de $0,\ldots,N-1$. A k-ésima diagonal de A, onde k está na faixa de $-(N-1),\ldots,(N-1)$, consiste daqueles elementos A_{ij} tais que $j-i=k$. Esta diagonal contém $N-\mid k \mid$ elementos. A seguir construimos D_m diagonais estendidas indexadas por m na faixa de $0,\ldots,(N-1)$ da seguinte forma. Para cada $k>0$, consideramos as diagonais k e $k-N$ que têm, respectivamente $N-k$ e k elementos. Assim, é possível construir um vetor de tamanho N, que é indexado por $m=k$ e chamado de diagonal estendida, consistindo da diagonal $k-N$ nos primeiros k componentes do vetor e diagonal k nos últimos $N-k$ componentes. Uma matriz completa é representada unicamente por N diagonais estendidas de tamanho N.

É conveniente introduzir uma matriz D cujas linhas são as diagonais estendidas de A (ordenadas pelo índice m introduzido acima). Podemos descrever então a matriz D como uma transformação de índice da matriz A.

$$A_{i,j} = D_{(j-i)\mathrm{mod}N,j}.$$

A matriz A e a matriz D podem ser vistas a seguir.

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix} \qquad D = \begin{pmatrix} a_{00} & a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{50} & a_{01} & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{40} & a_{51} & a_{02} & a_{13} & a_{24} & a_{35} \\ a_{30} & a_{41} & a_{52} & a_{03} & a_{14} & a_{25} \\ a_{20} & a_{31} & a_{42} & a_{53} & a_{04} & a_{15} \\ a_{10} & a_{21} & a_{32} & a_{43} & a_{54} & a_{05} \end{pmatrix}$$

A forma diagonal de uma matriz é armazenada representando cada diagonal estendida como um vetor distribuído. Assim uma outra matriz $D_{10\times 10}$ é armazenada por linhas distribuídas como pode ser visto a seguir.

P_{00}			P_{01}			P_{11}		P_{10}	
a_{00}	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}	a_{88}	a_{99}
a_{90}	a_{01}	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{67}	a_{78}	a_{89}
a_{80}	a_{91}	a_{02}	a_{13}	a_{24}	a_{35}	a_{46}	a_{57}	a_{68}	a_{79}
a_{70}	a_{81}	a_{92}	a_{03}	a_{14}	a_{25}	a_{36}	a_{47}	a_{58}	a_{69}
a_{60}	a_{71}	a_{82}	a_{93}	a_{04}	a_{15}	a_{26}	a_{37}	a_{48}	a_{59}
a_{50}	a_{61}	a_{72}	a_{83}	a_{94}	a_{05}	a_{16}	a_{27}	a_{38}	a_{49}
a_{40}	a_{51}	a_{62}	a_{73}	a_{84}	a_{95}	a_{06}	a_{17}	a_{28}	a_{39}
a_{30}	a_{41}	a_{52}	a_{63}	a_{74}	a_{85}	a_{96}	a_{07}	a_{18}	a_{29}
a_{20}	a_{31}	a_{42}	a_{53}	a_{64}	a_{75}	a_{86}	a_{97}	a_{08}	a_{19}
a_{10}	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}	a_{87}	a_{98}	a_{09}

É importante notar que podemos converter a forma de armazenamento de linhas distribuídas ou colunas contíguas para a forma diagonal sem qualquer comunicação entre os processadores, já que estas formas de armazenamento contém colunas inteiras em cada processador e a j-ésima coluna da matriz D é uma permutação da j-ésima coluna da matriz A. Então se a matriz A estiver na forma *LINHA_DISTRIBUÍDA* ou *COLUNA_CONTÍGUA*, a conversão de A para a forma *DIAGONAL* D envolve somente permutações nas colunas internas a cada processador. A figura IV.14 ilustra a organização dos elementos desses três tipos de distribuição matricial no processador zero para uma matriz 4×4 em um cubo de dimensão dois.

P_0		P_1	
a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

P_0	P_0	P_0
⋮	⋮	⋮
a_{00}	a_{00}	a_{00}
a_{01}	a_{10}	a_{11}
a_{10}	a_{20}	a_{30}
a_{11}	a_{30}	a_{01}
a_{20}	a_{01}	a_{20}
a_{21}	a_{11}	a_{31}
a_{30}	a_{21}	a_{10}
a_{31}	a_{31}	a_{21}
⋮	⋮	⋮

Figura IV.14: *LINHA_DISTRIBUÍDA*, *COLUNA_CONTÍGUA* e *DIAGONAL*

- *ÁREA*

A forma de distribuição de uma matriz por área consiste em atribuir uma

matriz de dimensões $N_x \times N_y$ em forma de blocos retangulares aos processadores da rede hipercúbica vistos como uma grade lógica $p_x \times p_y$. Cada bloco é envolvido por um limite simulado de largura b linhas e colunas. Os limites permitem que a matriz seja tratada como se armazenada em uma memória global, as operações realizadas têm a extensão de raio b. A figura IV.15 mostra uma distribuição típica de matriz por *ÁREA*, na qual as submatrizes são quadradas e o limite simulado escolhido é um.

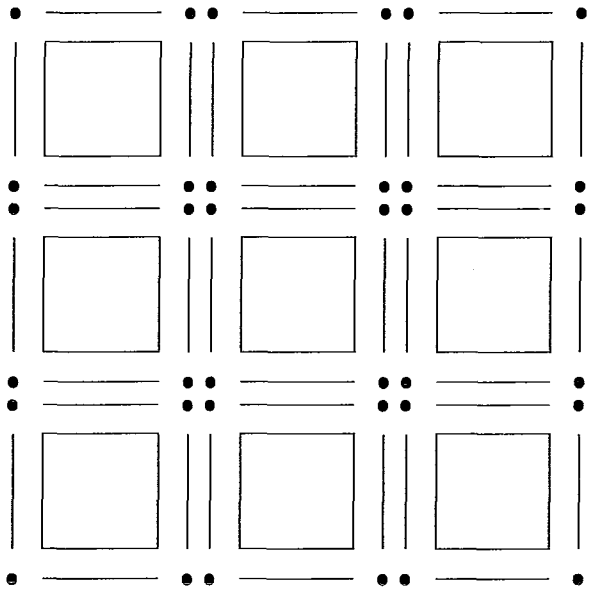


Figura IV.15: Distribuição de uma matriz por área com limite simulado ($b = 1$)

Antes de um armazenamento de matrizes do tipo ÁREA, deve-se utilizar o procedimento *parâmetros_de_área* para especificar as dimensões da grade lógica e o limite simulado além da periodicidade desejados. A chamada deste procedimento é feita da seguinte forma:

parâmetros_de_área (p_x, p_y, per_x, per_y, b).

Este procedimento armazena as informações que são consultadas no mapeamento do hipercubo em uma grade lógica utilizando somente conexões de vizinhos mais próximos (seção II.3.1). A grade é periódica na direção x ou y de acordo com os parâmetros *per_x* ou *per_y*. Chamadas subseqüentes à rotina de armazenamento matricial utilizam as informações mantidas pela última utilização do procedimento, sendo estas informações copiadas e armazenadas na estrutura de dados que representa a matriz que é retornada pela rotina de armazenamento matricial (*aloca_matriz*).

Assumiremos, por simplicidade, que podemos fatorar o número de processadores da rede hipercúbica em $P = p_x p_y$ e que as dimensões N_x e N_y da matriz A são múltiplas de p_x e p_y, sendo então: $N_x = n_x p_x$ e $N_y = n_y p_y$. Os processadores podem ser indexados por um índice de grade P_g, sendo $g = (i, j)$ com $0 \leq i < p_y$ e $0 \leq j < p_x$. A matriz A é então conceitualmente dividida em P submatrizes A_g retangulares iguais de tamanho $n_x \times n_y$, cada uma atribuída a um processador separado. O elemento matricial $a_{I,J}$ é atribuído ao processador $P_{I/n_y, J/n_x}$. A rotina *aloca_matriz* armazena memória para os *arrays* e os limites simulados. Cada submatriz é centralizada em uma matriz armazenada de tamanho $(n_x + 2b) \times (n_y + 2b)$. Assim o bloco A_g é indexado como $A_{g;i,j}$, onde $-b \leq i < n_y + b$ e $-b \leq j < n_x + b$. As linhas e as colunas na faixa $[-b, -1]$ e $[n_x, n_x + b - 1]$ ou $[n_y, n_y + b - 1]$ correspondem as linhas e colunas de limite simulado. O propósito dos pontos de limite simulado é permitir que alguns elementos

matriciais armazenados em processadores vizinhos estejam disponíveis localmente.

IV.2.2 Operação Shuffle

A operação *shuffle* é utilizada para trocar as linhas e/ou colunas externas de uma matriz armazenada na forma por *ÁREA* (por bloco) com os processadores vizinhos no hipercubo. Ela atualiza os elementos externos da submatriz local ao processador. Para esta, operação a topologia empregada é de uma grade. A operação é realizada com o seguinte procedimento

shuffle (pte_matriz, largura, lados)

Somente as linhas e colunas externas especificadas pelo parâmetro *largura* são trocadas. Embora possa ser especificada qualquer largura, geralmente ela é escolhida de tal forma que coincida com o limite simulado de largura b. O parâmetro *lados* especifica quais os lados envolvidos na operação. Existem quatro constantes simbólicas *NORTE*, *SUL*, *LESTE* e *OESTE* com os valores 1, 2, 4 e 8, respectivamente, que identificam os lados da submatriz. A constante simbólica dos lados é então obtida realizando uma operação **or** entre as constantes simbólicas, podendo ser especificado qualquer conjunto de lados. Por exemplo, para se trocar os lados *LESTE* e *OESTE* o parâmetro *lados* deve ter o valor 12 (4 + 8 correspondentes aos lados leste e oeste respectivamente).

É importante notar que a operação envolvendo os lados *LESTE* e *OESTE* é realizada com elementos não contíguos, já que as submatrizes são armazenadas por linhas. Para ilustrar este ponto, suponha uma matriz $A_{8\times8}$ armazenada por *ÁREA* em um cubo de dimensão dois, com uma grade lógica com dois processadores na direção x e dois processadores na direção y (como visto anteriormente na seção II.3.1) e um limite simulado igual a um.

	P_{00}			P_{01}			
a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	a_{06}	a_{07}
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}
a_{60}	a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}
a_{70}	a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}
	P_{10}				P_{11}		

a_{77}	a_{70}	a_{71}	a_{72}	a_{73}	a_{74}
a_{07}	a_{00}	a_{01}	a_{02}	a_{03}	a_{04}
a_{17}	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}
a_{27}	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}
a_{37}	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}
a_{47}	a_{40}	a_{41}	a_{42}	a_{43}	a_{44}

Os elementos dos lados leste e oeste são armazenados em posições não contíguas na memória.

Em princípio a operação *shuffle* é proporcional em custo ao perímetro de uma subgrade. A operação pode ser minimizada se as subgrades forem escolhidas próximas a subgrades quadradas. A figura IV.16 apresenta mapeamentos de grades lógicas 8 × 4, 16 × 2 e 32 × 1 em um hipercubo de dimensão cinco. Esses mapeamentos têm respectivamente, perímetros 24, 36 e 66.

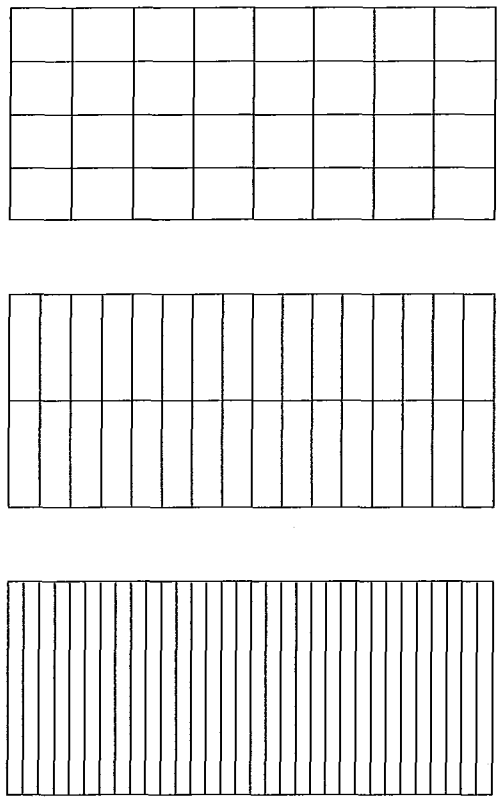


Figura IV.16: Atribuição de blocos de área iguais com perímetros diferentes

IV.2.3 Multiplicação entre uma Matriz e um Vetor

A multiplicação de uma matriz por um vetor quando a matriz está distribuída por linhas é trivial. Neste caso a multiplicação se resume em um conjunto de produtos internos padrão de vetores distribuídos entre as linhas da matriz e o vetor. O tempo necessário para esta operação é $O(N^2/P + logP)$, onde o $O(N^2/P$ se refere ao cálculo local a cada processador e $O(logP)$ é referente à difusão do resultado do produto interno para todos os processadores.

A representação de uma matriz na forma diagonal, combinada com as operações de deslocamento vetorial e soma entre vetores, permite que a multiplicação seja realizada eficientemente e de um modo altamente portável. Consideremos primeiro o caso da matriz densa A onde desejamos computar o produto $Y = AX$:

$$\begin{aligned} Y_i &= \sum_{j=0}^{N-1} A_{i,j} X_j \\ &= \sum_{j=0}^{N-1} D_{(j-i) \bmod N, j} X_j \end{aligned} \tag{IV.1}$$

fazendo $m = (j - i) \bmod N$ e $j = (i + m) \bmod N$ temos:

$$Y_i = \sum_{m=0}^{N-1} D_{m,(i+m) \bmod N} X_{(i+m) \bmod N}. \quad \text{(IV.2)}$$

Para escrevermos a equação acima na forma vetorial indicamos a m-ésima diagonal de A na forma diagonal por D_m, o deslocamento do vetor X m vezes por X^m e o produto vetorial de dois vetores ($(u \underline{*} v)_i = u_i v_i$) por $\underline{*}$. Dessa forma obtemos:

$$Y = \sum_{m=0}^{N-1} D_m^m \underline{*} X^m. \quad \text{(IV.3)}$$

No m-ésimo termo desta soma, ambos D_m e X são deslocados m vezes. Ao invés de deslocar ambos as diagonais e o vetor X pela mesma quantidade, podemos deslocar o vetor resultante Y por m na direção oposta. Dessa forma obtemos um algoritmo mais eficiente com apenas um deslocamento vetorial. Isto leva a forma final do algoritmo *matriz_vetor*:

$Y = 0$
Para $m = 0, \ldots, N-1$ faça
    $Y = Y + D_m \underline{*} X$
    $Y = Y^{-1}$

O primeiro comando é a operação de criação de um vetor nulo. Dentro do *loop*, o comando inicial é a operação padrão de *soma_vet_a_vct_vet*, o segundo comando é uma operação de deslocamento vetorial. A paralelização dessas operações já foi mencionada anteriormente na seção IV.1. Assim para paralelizar este procedimento é suficiente armazenar as diagonais estendidas D_m, X e Y como vetores distribuídos, sendo que o vetor Y necessita estar no formato *SHIFT*, uma vez que este sofrerá deslocamentos sucessivos. O tempo de computação da operação *matriz_vetor* é da ordem $O(N^2/P)$ operações aritméticas, para uma matriz densa, e um tempo da ordem $O(N)$ para N operações de deslocamento do vetor Y. O *overhead* de comunicação (tempo gasto nos deslocamentos), será desprezível se comparado com a computação de matrizes muito grandes, definidas como aquelas em que a dimensão é muito maior que o número de processadores ($N \gg P$).

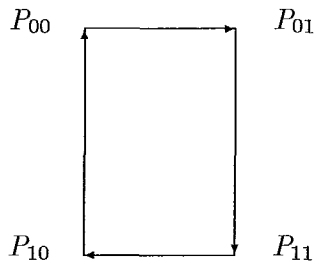
No caso de matrizes esparsas podemos reduzir o número de operações aritméticas já que é desnecessário multiplicar as diagonais nulas. Entretanto, os deslocamentos ainda devem ser feitos para todas as diagonais. O custo de comunicação permanece de N operações de deslocamento, embora muitos deles possam ser agrupados em deslocamentos maiores.

A figura IV.17 mostra os passos do algoritmo para uma multiplicação de uma matriz por um vetor num cubo de duas dimensões.

IV.2.4 Multiplicação entre matrizes

Na multiplicação de matrizes a forma diagonal também é muito conveniente. Consideremos a operação:

$$C \leftarrow C + A * B,$$



$$Y = \sum_{j=0}^{N-1} A_{i,j} X_j$$

$$A_{4\times4} = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \qquad X = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

$$Y = \begin{pmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{pmatrix}$$

	P_{00}	P_{01}	P_{11}	P_{10}
D	a m i e	f b n j	k g c o	p l h d
X	x	y	z	w
Y	0	0	0	0
1	$0 + ax$	$0 + fy$	$0 + kz$	$0 + pw$
2	pw	ax	fy	kz
1	$mx + pw$	$ax + by$	$fy + gz$	$kz + lw$
2	$kz + lw$	$mx + pw$	$ax + by$	$fy + gz$
1	$ix + kz + lw$	$mx + ny + pw$	$ax + by + cz$	$fy + gz + dw$
2	$fy + gz + dw$	$ix + kz + lw$	$mx + ny + pw$	$ax + by + cz$
1	$ex + fy + gz + dw$	$ix + jy + kz + lw$	$mx + ny + oz + pw$	$ax + by + cz + dw$
2	$ax + by + cz + dw$	$ex + fy + gz + dw$	$ix + jy + kz + lw$	$mx + ny + oz + pw$

1 Operação *soma_vet_a_vet_vet*
2 Operação *desloca_vetor*

Figura IV.17: Multiplicação de uma matriz por um vetor

onde A, B e C são matrizes quadradas de dimensão N. Para descrevermos o algoritmo usaremos uma notação simplificada na qual caracteres minúsculos se referem a elementos da matriz e caracteres maiúsculos se referem a elementos da forma diagonal. Assim, temos que:

$$a_{i,j} = A_{(j-i) \bmod N, j}.$$

Omitiremos também por simplicidade todos os índices de módulo N, então a fórmula acima reduz-se a:

$$a_{i,j} = A_{(j-i),j}.$$

Como na seção IV.2.3, usaremos letras subscritas para representar as diagonais estendidas e a notação de letras superscritas para deslocamentos vetoriais com $\underline{*}$ para os produtos entre vetores.

A fórmula clássica de multiplicação matricial com estas notações torna-se:

$$C_{j-i,j} = \sum_{k=0}^{N-1} A_{k-i,k} B_{j-k,j}. \tag{IV.4}$$

Substituindo $p = j - i$, e $q = j - k$ temos:

$$C_{p,j} = \sum_{q=0}^{N-1} A_{p-q,j-q} B_{q,j}. \tag{IV.5}$$

Essa equação pode ser escrita na forma vetorial, usando as diagonais de A e B e o produto vetorial. Assim esta fórmula reduz-se a:

$$C_p = \sum_{q=0}^{N-1} A^{-q}_{p-q,} \underline{*} B_q. \tag{IV.6}$$

Dessa forma, torna-se fácil mostrar que o seguinte algoritmo implementa a operação de multiplicação de matrizes utilizando somente operações paralelas:

Para $q = 0, \ldots, N-1$ faça
  Para $p = 0, \ldots, N-1$ faça
    $C_p \leftarrow C_p + A_{p-q} \underline{*} B_q$
    $A_{p-q} \leftarrow A^{-1}_{p-q}$

O primeiro comando do *loop* é o procedimento de *soma_vet_a_vet_vet*, que é trivialmente paralelizável. A segunda linha de comando é uma operação de deslocamento vetorial. Note que, após todas as operações, todas as matrizes, incluindo a matriz A, estão em suas formas originais. Somente as diagonais de A necessitam ser armazenadas como vetores distribuídos do tipo *SHIFT*, pois nelas serão realizados sucessivos deslocamentos. As diagonais de C e B devem ser armazenadas no formato *SIMPLE*.

Como no procedimento de multiplicação de uma matriz por um vetor, podemos também agrupar deslocamentos maiores quando as matrizes A ou B forem esparsas, e reduzir a quantidade de operações aritméticas, uma vez que não é necessário a multiplicação de diagonais nulas.

Com matrizes densas a operação de multiplicação realiza uma operação de comunicação (tempo para transferência de uma variável real) para cada $2N$ operações de ponto flutuante.

IV.2.5 Procedimento Rank One Update

Como nos procedimentos de multiplicação entre uma matriz e um vetor e entre matrizes, a forma diagonal também é muito conveniente para a implementação da operação:

$$A \leftarrow A + xy^T.$$

Os elementos da matriz $B = xy^T$ são $b_{i,j} = x_i y_j$. Isto implica que a forma diagonal de B tem elementos $B_{k,j} = x_{j-k} y_j$, onde usamos as notações já mencionadas na seção IV.2.4. Deste modo podemos expressar B em termos de deslocamento vetorial e produtos de elementos vetoriais como:

$$B_k = x^{-k} \underline{*} y. \tag{IV.7}$$

O algoritmo para a operação *rank_one_update* tem então a seguinte forma:

Para $k = 0, \ldots, N-1$
$A_k \leftarrow A_k + x \underline{*} y$
$x \leftarrow x^{-1}$

A primeira linha de comando é uma operação de *soma_vet_a_vet_vet*, e a segunda é uma operação de deslocamento vetorial. Todas as operações aritméticas são paralelizáveis e o algoritmo realiza uma operação de deslocamento para cada $2N$ operações de ponto flutuante.

A figura IV.18 mostra como se comporta a memória durante a operação *rank_one_update*.

IV.2.6 Transposição Matricial

Esta operação converte uma matriz distribuída por linhas em uma matriz distribuída por colunas. Existem dois algoritmos para transposição matricial. O primeiro algoritmo utiliza a forma diagonal e operações de deslocamento vetorial, por isso denominamos o algoritmo de transposição por deslocamento. Para este algoritmo ser eficiente, é essencial que seja utilizado o algoritmo de deslocamento vetorial otimizado, visto na seção IV.1.2. O algoritmo somente deve ser empregado quanto a dimensão da matriz for um múltiplo do número de processadores que compõem a rede hipercúbica. O segundo algoritmo chamamos de transposição por bloco. Ele utiliza a forma de armazenamento por linhas contíguas e pode ser usado com matrizes de qualquer dimensão.

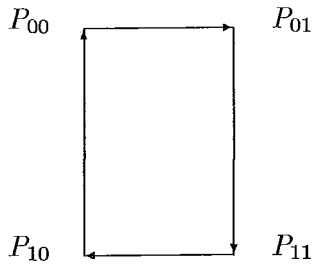
- Transposição por deslocamento

O elemento $A_{i,j}$ da matriz A é equilavente ao elemento $D_{(j-i) \bmod N, j}$ da matriz diagonal D. A transposta A^T da matriz A, tem a forma diagonal DT tal que

$$DT_{(j-i) \bmod N, j} = A^T_{i,j} = A_{j,i} = D_{(i-j) \bmod N, i}. \tag{IV.8}$$

Disso temos que

$$DT_{k,j} = D_{N-k,i} = D_{N-k,(j-k) \bmod N}. \tag{IV.9}$$



$$A = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \qquad x = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \qquad y^T = \begin{pmatrix} q \\ r \\ s \\ t \end{pmatrix}$$

$$A = \begin{pmatrix} a+xq & b+xr & c+xs & d+xt \\ e+yq & f+yr & g+ys & h+yt \\ i+zq & j+zr & k+zs & l+zt \\ m+wq & n+wr & o+ws & p+wt \end{pmatrix}$$

	P_{00}	P_{01}	P_{11}	P_{10}
D	a	f	k	p
	m	b	g	l
	i	n	c	h
	e	j	o	d
X	x	y	z	w
Y	q	r	s	t
1	$a+xq$	$f+yr$	$k+zs$	$p+wt$
2	w	x	y	z
1	$m+wq$	$b+xr$	$g+ys$	$l+zt$
2	z	w	x	y
1	$i+zq$	$n+wr$	$c+xs$	$h+yt$
2	y	z	w	x
1	$e+yq$	$j+zr$	$o+ws$	$d+xt$
2	x	y	z	w

1 Operação *soma_vet_a_vet_vet*
2 Operação *desloca_vetor*

Figura IV.18: Operação *rank one update*

Isso significa que a k-ésima diagonal de DT é a $(N-k)$-ésima diagonal de D, deslocada por k. Note que uma vez que $k > N/2$ é mais eficiente e equivalente a deslocar por $N-k$.

Na implementação desse algoritmo de transposição utilizamos a matriz armazenada na forma *LINHA_DISTRIBUÍDA*, ao invés de armazená-la na forma diagonal. Utilizamos dois vetores do tipo *SHIFT*, que no k-ésimo passo, guardam as diagonais estendidas k e $N-k$ da matriz A que são respectivamente deslocados por k e $-k$. Uma vez realizados estes deslocamentos, as diagonais são copiadas de volta a matriz intercaladas corretamente.

• Transposição por bloco

Este algoritmo não necessita que a dimensão da matriz seja um múltiplo da quantidade de processadores da rede hipercúbica. Uma matriz $A_{2\times 2}$ pode ser transposta pela troca dos seus elementos superior direito e inferior esquerdo. Isso leva ao seguinte algoritmo serial para transpor uma matriz:

Transposição (M):

$$M = \begin{bmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{bmatrix}$$

$M_{01} \leftrightarrow M_{10}$

Transposição(M_{00}) *Transposição*(M_{01})

Transposição(M_{10}) *Transposição*(M_{11})

Uma matriz com dimensão $N = 2^L$ pode ser transposta recursivamente aplicando este procedimento através de L níveis. No nível final todas as matrizes são de dimensão 1×1 e trivialmente transpostas. A figura IV.19 mostra os dois primeiros níveis deste procedimento, onde as setas representam os blocos trocados.

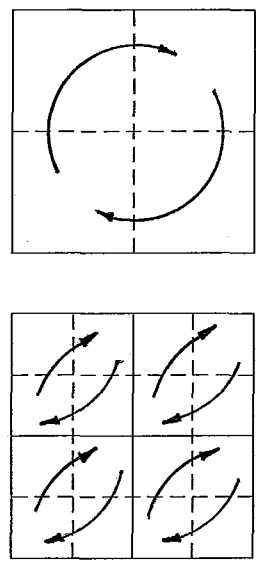


Figura IV.19: Representação esquemática da transposição por bloco

Para implementar este algoritmo em um hipercubo, a matriz M deve estar

armazenada por linhas contíguas. Para tornar mais simples o seu entendimento assumimos que a dimensão de M é divisível pelo número de processadores ($P = 2^D$). Neste algoritmo é empregada a ordenação natural dos processadores. A intertroca do subbloco superior direito e inferior esquerdo no nível L envolve a comunicação entre processadores com números de identificação diferindo de 2^{D-L-1}, mesmo estando os processadores a uma distância física igual a um. A figura IV.20 ilustra os passos do algoritmo de transposição por bloco em um hipercubo de dimensão dois com uma matriz $A_{4\times4}$.

$$A = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \qquad A^T = \begin{pmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{pmatrix}$$

	P_{00}	P_{01}	P_{11}	P_{10}
	$a\ b\ c\ d$	$e\ f\ g\ h$	$i\ j\ k\ l$	$m\ n\ o\ p$
1	$a\ b \mid c\ d$	$e\ f \mid g\ h$	$i\ j \mid k\ l$	$m\ n \mid o\ p$
2	$a\ b$ $m\ n$	$e\ f$ $i\ j$	$g\ h$ $k\ l$	$c\ d$ $o\ p$
1	$a \mid b$ $m \mid n$	$e \mid f$ $i \mid j$	$g \mid h$ $k \mid l$	$c \mid d$ $o \mid p$
2	a e i m	b f j n	c g k o	d h l p

1 Seleção dos dados
2 Troca de mensagens

Figura IV.20: Transposição matricial por bloco

IV.3 Procedimentos de Difusão e Convergência

Um dos passos fundamentais é a distribuição e o armazenamento dos dados iniciais. Estes são feitos antes do início da computação. Os dados devem ser divididos em partes e armazenados na memória local de cada processador.

Como no multiprocessador hipercúbico baseado em *transputers* não existe a facilidade direta para a entrada e saída paralela dos dados, isto é realizado de modo indireto através do processo de controle residente no processador raiz que está ligado ao computador hospedeiro.

Os programas de aplicação resultam então em diversos processos que devem ter uma comunicação de maneira coordenada, para o envio e a recepção dos dados para/do hipercubo.

Os procedimentos de difusão distribuem um escalar, um vetor, uma matriz ou uma mensagem qualquer através dos processadores do hipercubo. Os vetores e ma-

trizes são distribuídos segundo a seqüência BRGC aos processadores e os dados divididos dependendo da forma de armazenamento indicada como visto nas seções IV.1.1 e IV.2.1. O processo de controle faz a divisão e distribuição dos dados. Os processos dos nodos têm procedimentos coordenados para a recepção desses dados.

Os procedimentos de convergência recebem um escalar, um vetor, uma matriz ou uma mensagem dos processadores do hipercubo. Tratam de reconstituir segundo a mesma seqüência BRGC e divisão, o vetor, a matriz ou a mensagem recebidos. O processo de convergência também é realizado de modo coordenado entre o processo de controle e os processos dos nodos.

Os procedimentos implementados de difusão e convergência são mostradas na tabela IV.3.

envia_escalar_ao_cubo	difusão e convergência do
recebe_escalar_do_cubo	processo de controle
envia_vetor_ao_cubo	
recebe_vetor_do_cubo	
envia_matriz_ao_cubo	
recebe_matriz_do_cubo	
envia_mensagem_ao_cubo	
recebe_mensagem_do_cubo	
envia_escalar_ao_controle	difusão e convergência dos
recebe_escalar_do_controle	processos dos nodos
envia_vetor_ao_controle	
recebe_vetor_do_controle	
envia_matriz_ao_controle	
recebe_matriz_do_controle	
envia_mensagem_ao_controle	
recebe_mensagem_do_controle	

Tabela IV.3: Rotinas Relativas à Difusão e Convergência

IV.4 Resumo

Todos os procedimentos que fazem parte da biblioteca de operações paralelas trabalham sobre os seus segmentos locais ignorando a existência de seus vizinhos na rede hipercúbica. Toda troca de informações necessária é realizada internamente a cada procedimento, sendo portanto transparente ao usuário. Cada operação é realizada através de uma chamada ao procedimento pertinente utilizando argumentos apropriados.

Os vetores e matrizes são representados por estruturas de dados que contém informações necessárias a sua manipulação dentro dos procedimentos. Os vetores têm dois tipos diferentes de armazenamento e as matrizes podem ser armazenadas de seis modos distintos, podendo ser densas ou esparsas.

A biblioteca de operações paralelas inclue operações para armazenamento vetorial e matricial, operações básicas da álgebra linear tais como produto interno, transposição matricial, produto entre matrizes, difusão e convergência de dados entre outras.

Capítulo V

Aplicação em Sistemas Lineares

A solução de sistemas de equações lineares é fundamental para inúmeras aplicações numéricas, encontradas em problemas das Engenharias, Física, Meteorologia entre outras. Para demonstrar a simplicidade e o potencial de aplicação da biblioteca de operações paralelas, procuramos solucionar um sistema de equações lineares algébricas utilizando o método do gradiente conjugado. Embora a solução de sistemas de equações lineares já tenha sido amplamente realizada através deste e de vários outros métodos. Este capítulo descreve o método do gradiente conjugado e sua implementação utilizando a biblioteca de operações paralelas, e realiza uma avaliação comparativa com matrizes de dimensões diferentes. A avaliação foi realizada utilizando o multiprocessador hipercúbico NCP I da COPPE.

V.1 O Método do Gradiente Conjugado

Para uma matriz $A(n,n)$, simétrica e definida positiva, a solução do sistema linear algébrico [9]

$$Ax = b \tag{V.1}$$

é equivalente ao seguinte problema de otimização sem restrições

$$min\ Q(x) = \frac{1}{2}x^t Ax - b^t x. \tag{V.2}$$

A função $Q(x)$ tem um mínimo global onde o seu gradiente

$$\nabla Q(x) = Ax - b \tag{V.3}$$

é zero.

A forma geral dos métodos iterativos de minimização é dada por [9]

$$x^{k+1} = x^k - \alpha^k d^k, \quad k = 1, 2, \ldots \tag{V.4}$$

onde:
d^k é a direção vetorial de minimização e
α^k é o escalar que define a distância de movimento na direção d^k.

A forma de escolha de α^k e d^k, define uma grande variedade de métodos, entre eles, o método do gradiente conjugado [9] empregado neste trabalho.

Desse modo o problema de minimizar a função $Q(x)$ é transformado em sucessivos problemas do tipo minimizar $Q(x)$ na direção d^k de modo que α^k satisfaça a seguinte equação:

$$Q(x^k - \alpha^k d^k) = min_\alpha \, Q(x^k - \alpha d^k). \tag{V.5}$$

Para x^k e d^k especificados e suprimindo o superescrito k, pode-se escrever a seguinte função em α:

$$q(\alpha) = Q(x - \alpha d) = \frac{1}{2}(x - \alpha d)^t A(x - \alpha d) - b^t(x - \alpha d) \tag{V.6}$$

ou

$$q(\alpha) = \frac{1}{2}d^t A d\alpha^2 - d^t(Ax - b)\alpha + \frac{1}{2}x^t(Ax - 2b). \tag{V.7}$$

A função quadrática $q(\alpha)$ é minimizada quando sua primeira derivada for zero, ou seja:

$$\frac{dq(\alpha)}{d\alpha} = d^t A d\alpha - d^t(Ax - b) = 0. \tag{V.8}$$

Resolvendo-se a equação para α, temos:

$$\alpha = \frac{d^t(Ax - b)}{d^t A d}. \tag{V.9}$$

No método gradiente conjugado as direções de minimização unidirecionais são geradas a partir da direção do gradiente, sucedidas por direções conjugadas de modo que satisfaçam a seguinte equação:

$$(d^i)^t A d^j = 0, \quad i \neq j. \tag{V.10}$$

A direção do gradiente é obtida tendo a função objetivo

$$Q(x) = \frac{1}{2}x^t A x - b^t x \tag{V.11}$$

definindo d^k tal que:

$$d^k = -\nabla Q(x^k) - (Ax^k - b). \tag{V.12}$$

Então, o algoritmo do gradiente conjugado quando aplicado na resolução de um sistema linear algébrico segue os seguintes passos:

1. Especificação de uma solução inicial x^0;
2. Cálculo da direção negativa do gradiente

$$r^0 = b - Ax; \tag{V.13}$$

3. Inicialização da direção vetorial

$$d^0 = r^0; \tag{V.14}$$

4. Enquanto o resíduo

$$\left\| r^{k+1} \right\|_2^2 \tag{V.15}$$

for maior que a precisão ε especificada, calcular

$$\alpha^k = -\frac{\langle r^k . r^k \rangle}{\langle d^k . A d^k \rangle} \tag{V.16}$$

$$x^{k+1} = x^k - \alpha^k d^k \tag{V.17}$$

$$r^{k+1} = r^k + \alpha^k A d^k; \tag{V.18}$$

5. Cálculo do novo resíduo

$$\left\| r^{k+1} \right\|_2^2 \tag{V.19}$$

6. Se o resíduo for maior que a precisão ε, então calcula a nova direção conjugada d^{k+1}

$$\beta^k = \frac{\langle r^{k+1} . r^{k+1} \rangle}{\langle r^k . r^k \rangle} \tag{V.20}$$

$$d^{k+1} = r^{k+1} + \beta^k d^k \tag{V.21}$$

7. Retorna ao passo quatro.

Definido o método a ser empregado para a resolução de um sistema linear algébrico, é necessário captar e distribuir os dados, a serem manipulados, através de todos os processadores na rede e coletar os resultados finais obtidos.

V.2 Implementação do Método do Gradiente Conjugado

Para solucionar o sistema de equações lineares, foram definidos dois processos. Um processo de controle que realiza as operações de entrada e saída dos dados e um processo que efetivamente calcula a solução segundo o algoritmo do gradiente conjugado visto anteriormente. Esses dois processos, chamados *processo de controle* e *processo dos nodos*, se comunicam de maneira coordenada.

O algoritmo do processo de controle (figura V.1) realiza a captação dos dados, a distribuição desses para todos os processadores da rede e coleta os dados obtidos na solução. A descrição do algoritmo do processo de controle é mostrada a seguir:

Processo de Controle

```
leitura_matricial (pta_A, linha, coluna);
leiruta_vetorial (pta_b, tamanho);
leitura_vetorial (pta_x0, tamanho);
envia_matriz_ao_cubo (pte_A, tipo);
envia_vetor_ao_cubo (pte_b);
envia_vetor_ao_cubo (pte_x0);
pte_x = recebe_vetor_do_cubo ();
```

Figura V.1: Uso da biblioteca de operações no Processo de Controle

1. Para se obter os dados iniciais armazenados em arquivos, chamamos as rotinas de leitura tendo como argumentos os ponteiros dos arquivos, e a dimensão da matriz ou o tamanho do vetor;

   - para a leitura da matriz temos:
     $pte\_A = leitura\_matricial(pta\_A, linha, coluna)$;
   - e para a leitura dos vetores temos:
     $pte\_b = leitura\_vetorial(pta\_b, N)$;
     $pte\_x^0 = leitura\_vetorial(pta\_x^0, N)$;

   Essas rotinas retornam um ponteiro para a estrutura de dados que representam a matriz e os vetores.

2. A distribuição da matriz e dos vetores aos processadores que compõem o hipercubo, é realizada pelas rotinas de difusão;

   - os argumentos da rotina que distribui uma matriz são o ponteiro da estrutura de dados e o tipo de armazenamento conveniente à aplicação (seção IV.2.1);
     $envia\_matriz\_ao\_cubo(pte\_A, tipo)$;
   - o argumento da rotina de distribuição vetorial é o ponteiro da estrutura de dados;
     $envia\_vetor\_ao\_cubo(pte\_b)$;
     $envia\_vetor\_ao\_cubo(pte\_x^0)$;

3. Para receber o vetor solução chamamos a rotina de convergência vetorial que não contém argumentos.
   $pte\_x = recebe\_vetor\_do\_cubo()$;

Assim, no processo de controle utilizamos cinco rotinas diferentes da biblioteca de operações paralelas. Duas para leitura matricial e vetorial, duas para o envio da matriz e dos vetores, e uma para recepção do vetor solução. Pode-se notar a ausência de qualquer preocupação com a arquitetura ou o modo como foram distribuídos e recebido a matriz e os vetores empregados na solução.

O algoritmo dos processos dos nodos está demonstrado a seguir e é dividido em duas partes, uma de recepção e envio dos dados em relação ao processo de controle (figura V.2) e outra do método do gradiente conjugado propriamente dito (figura V.3). A modularidade empregada é para facilitar a depuração do algoritmo.

Processo dos Nodos

```
pte_A = recebe_matriz_do_controle ();
pte_b = recebe_vetor_do_controle (tipo);
pte_x^0 = recebe_vetor_do_controle (tipo);
pte_x = gradiente (pte_A, pte_b, pte_x^0);
envia_vetor_ao_controle (pte_x);
```

Figura V.2: Uso da biblioteca de operações no Processo dos Nodos

1. Para receber a matriz e os vetores do processo de controle utilizamos as rotinas de recepção;

- para a recepção matricial não há argumentos;

  $pte\_A = recebe\_matriz\_do\_controle()$;

- a recepção de vetores tem como argumento o tipo de armazenamento vetorial;

  $pte\_b = recebe\_vetor\_do\_controle(tipo)$;

  $pte\_x^0 = recebe\_vetor\_do\_controle(tipo)$;

Essas rotinas retornam as estruturas de dados que representam a matriz e os vetores.

2. O cálculo do gradiente conjugado é realizado por um módulo separado para facilitar a depuração, cujos argumentos são os ponteiros das estruturas da matriz e dos vetores;

   $pte\_x = gradiente(pte\_A, pte\_b, pte\_x^0)$;

   Esse módulo retorna o ponteiro do vetor solução.

3. Finalmente para enviar o vetor solução ao processo de controle utilizamos a rotina correspondente, tendo como argumento o ponteiro da estrutura de dados do vetor.

   $envia\_vetor\_ao\_controle(pte\_x)$;

Aqui foram empregados três diferentes passos elementares da biblioteca, dois correspondentes a recepção da matriz e dos vetores, e um de envio do vetor solução ao processo de controle.

Aplicando agora as rotinas da biblioteca de operações paralelas para compor o algoritmo do gradiente conjugado, temos:

Gradiente Conjugado

```
pte_r^k = matriz_vetor (pte_A, pte_x^k);
soma_esc_vet_a_vet (pte_A, pte_b, -1);
pte_d^k = copia_vetor (pte_r^k);
n^k1 = produto_interno (pte_r^k, pte_r^k);
Enquanto n^k > ε and flag = false
  n^k = n^k1;
  pte_Ad^k =matriz_vetor (pte_A, pte_d^k);
  d^k = produto_interno (pte_d^k, pte_Ad^k);
  α^k = -n^k / d^k;
  soma_vet_a_esc_vet (pte_x^k, pte_d^k, -α^k);
  soma_vet_a_esc_vet (pte_r^k, pte_Ad^k, α^k);
  n^k1 = produto_interno (pte_r^k, pte_r^k);
  Se n^k1 ≤ ε
    flagp = true;
  Caso contrário
    d^k = n^k;
    β^k = n^k1 / d^k;
    soma_esc_vet_a_vet (pte_d^k, pte_r^k, β^k);
Retorna pte_x^k.
```

Figura V.3: Uso da biblioteca de operações no Gradiente

1. A especificação da solução inicial x^0 foi passada como argumento para a rotina e a precisão ε foi especificada com um valor fixo, interno à rotina;

2. O cálculo da direção negativa do gradiente da equação V.13 é realizado com dois passos elementares:

- primeiro é feito o produto entre a matriz A e o vetor solução inicial x^0, tendo como argumentos os ponteiros das estruturas de dados da matriz e do vetor;

  $pte\_r^0 = matriz\_vetor(pte\_A, pte\_x^0)$;

  ao retornarmos dessa rotina temos $r^0 = Ax^0$;

- a seguir chamamos a rotina de soma vetorial apropriada, cujos argumentos são as estruturas do vetor obtido pelo primeiro passo e do vetor b, e o escalar -1 para a subtração;

  $soma\_esc\_vet\_a\_vet(pte\_r^0, pte\_b, -1)$;

Ao término obtemos então r^0 da equação V.13;

3. A inicialização da direção vetorial d^0 é conseguida com outro passo elementar que copia um vetor, enviando como argumento o vetor r^0;

   $pte\_d^0 = copia\_vetor(pte\_r^0)$;

   Assim temos então a equação V.14;

4. O resíduo da equação V.15 é calculado através do produto interno, que recebe como argumentos a estrutura do vetor r^k

   $n^{k+1} = produto\_interno(pte\_r^0, pte\_r^0)$;

5. Para as sucessivas iterações verificamos se o resíduo obtido é maior que a precisão especificada, não sendo atendida a condição, calcula-se:

   - a distância de movimento na direção d^k que corresponde a equação V.16. São realizados uma atribuição a variável

     $n^k = n^{k+1}$;

     que corresponde ao numerador e mais duas chamadas aos passos elementares para se obter o denominador. Primeiro para se ter o produto entre a matriz A e o vetor d^k e depois o produto interno entre esse resultado e o vetor d^k. Temos assim:

     $pte\_Ad^k = matriz\_vetor(pte\_A, pte\_d^k)$;
     $d^k = produto\_interno(pte\_d^k, pte\_Ad^k)$;
     $\alpha^k = -n^k/d^k$;

   - Para o cálculo do novo vetor solução (equação V.17), utilizamos a rotina de soma vetorial tendo como argumentos as estruturas dos vetors x^k e d^k e o escalar α^k;

     $soma\_vet\_a\_esc\_vet(pte\_x^k, pte\_d^k, -\alpha^k)$;

   - A equação V.18 também é resolvida com a rotina de soma vetorial agora tendo como argumentos os vetores r^k, d^k e o escalar α^k

     $soma\_vet\_a\_esc\_vet(pte\_r^k, pte\_Ad^k, \alpha)$;

6. Para o cálculo do novo resíduo procedemos como da forma anterior empregando o produto vetorial, agora tendo o vetor r^k, calculado no passo acima, como argumento;

   $n^{k+1} = produto\_interno(pte\_r^k, pte\_r^k)$;

7. Verificamos novamente se o resíduo atende a precisão dada, ou em caso contrário, calculamos a nova direção conjugada através da soma vetorial e do valor de β^k da seguinte forma:

- a equação V.20 é obtida com os resultados já calculados para o numerador de α^k, agora como denominador, e do novo resíduo n^{k+1} como numerador;
  $\beta^k = n^{k+1}/n^k$;
- e para a nova direção conjugada (V.21) enviamos como argumentos as estruturas dos vetores d^k e r^k e o escalar β^k;
  $soma\_esc\_vet\_a\_vet(pte\_d^k, pte\_r^k, \beta^k)$;

Dessa forma, o algoritmo do gradiente conjugado foi implementado com cinco passos elementares da biblioteca de operações paralelas:

- Cópia de um vetor ($u = v$);
- Produto de uma matriz por um vetor ($v = Au$);
- Produto interno ($\sum_{i=0}^{N-1} u_i v_i$);
- Soma do produto de um escalar por um vetor com outro vetor ($u = su + v$) e
- Soma de um vetor com o produto de um escalar por outro vetor ($u = u + sv$).

O apêndice D apresenta a implementação dos processos de controle e dos nodos.

V.3 Avaliação de Desempenho

Para validar e realizar uma avaliação inicial de desempenho da biblioteca de operações paralelas, empregamos o algoritmo do gradiente conjugado para solucionar um sistema linear algébrico ($Ax = b$).

O desempenho foi considerado em relação ao ganho de velocidade e em relação à eficiência.

O ganho de velocidade (*speedup*) pode ser definido como a relação entre o tempo consumido na execução do algoritmo em um único processador, com o tempo de execução quando for utilizado o mesmo algoritmo paralelizado em n processadores. Assim temos:

$$S_n = \frac{T_1}{T_n}, \tag{V.22}$$

onde
S_n é o ganho de velocidade para n processadores;
T_1 o tempo de processamento do algoritmo em um único processador e
T_n o tempo de processamento do algoritmo com n processadores.

A eficiência é definida como a relação entre o ganho de velocidade e o número de processadores. Dessa forma, temos:

$$\eta_n = \frac{S_n}{n} \tag{V.23}$$

onde
η_n é a eficiência do paralelismo com a utilização de n processadores;
S_n é o ganho de velocidade para n processadores e
n é o número de processadores.

Os dados utilizados foram obtidos de um sistema de equações, onde cada incógnita ou equação está associado a um nó da rede elétrica num sistema de fluxo de potência. Para as redes elétricas usuais, a matriz apresenta um elevado grau de esparsidade. Os únicos elementos que não são nulos estão na diagonal principal e nas posições correspondentes às conexões físicas entre os nós da rede elétrica. Este tipo de esparsidade não é aproveitado pela biblioteca de operações paralelas (que visa matrizes esparsas de banda) que trata a matriz então como se fosse praticamente uma matriz densa.

Os parâmetros que foram variados no programa para o estudo do seu comportamento são os seguintes:

- n - número de equações (dimensão do sistema);
- p - quantidade de processadores da rede hipercúbica;
- ε - precisão.

A comparação foi realizada com um sistema mínimo de dimensão igual a oito (hipotético), um sistema de 45 barras e dois sistemas de 118 barras [10]. Num dos sistemas de 118 barras, referido como 118*, a condição inicial está próxima da sua solução e no outro, a condição inicial está distante da sua solução. A quantidade de processadores variou de $1,\ldots,8$ (dimensão zero a três). As precisões empregas foram 1.0E-4, 1.0E-6 e 1.0E-8. Em todos os casos, os vetores iniciais utilizados foram vetores nulos.

A tabela V.1 mostra os tempos de execução em segundos, obtidos com um *transputer* e precisão igual a 1.0E-4, 1.0E-6 e 1.0E-8.

Tempo de Execução (s)			
Sistema	ε = 1.0E-4	ε = 1.0E-6	ε = 1.0E-8
8	0.0240	0.0270	0.0301
45	2.2045	2.6203	2.8283
118	26.7024	31.4714	34.7414
118*	1.7244	4.5953	16.0796

Tabela V.1: Tempo de execução para um *transputer* com precisões iguais a 1.0E-4, 1.0E-6 e 1.0E-8

A tabela V.1 mostra que os tempos de execução aumentaram de acordo com a dimensão do sistema em teste e da precisão exigida. A finalidade dos dois sistemas de 118 barras é ilustrar quanto o tempo de execução pode variar na blblioteca dependendo do valor da condição inicial dada. Como o sistema 118* tem o vetor inicial mais próximo da solução, seu tempo de execução foi bem menor.

A tabela V.2, a tabela V.3 e a tabela V.4 mostram os tempos de execução obtidos com dois, quatro e oito *transputers* respectivamente, utilizando a precisão igual a

Tempo de Execução (s)			
Sistema	ε = 1.0E-4	ε = 1.0E-6	ε = 1.0E-8
8	0.0631	0.0711	0.0791
45	2.7266	3.0800	3.4841
118	19.6627	23.2565	25.3710
118*	1.2692	3.3837	11.6289

Tabela V.2: Tempo de execução para dois *transputers* com precisão igual a 1.0E-4, 1.0E-6 e 1.0E-8

1.0E-4, 1.0E-6 e 1.0E-8. O tempo de execução foi obtido somando o tempo de manipulação dos dados e o tempo de comunicação entre os processadores.

Tempo de Execução (s)			
Sistema	ε = 1.0E-4	ε = 1.0E-6	ε = 1.0E-8
8	0.0909	0.1025	0.1142
45	2.4392	2.7616	3.1302
118	14.1501	16.5848	18.2588
118*	0.9113	2.4330	8.3678

Tabela V.3: Tempo de execução para quatro *transputers* com precisão igual a 1.0E-4, 1.0E-6 e 1.0E-8

Tempo de Execução (s)			
Sistema	ε = 1.0E-4	ε = 1.0E-6	ε = 1.0E-8
8	0.1406	0.1588	0.1771
45	2.5095	2.8890	3.2686
118	11.9325	14.1383	15.5656
118*	0.7732	2.0707	7.1315

Tabela V.4: Tempo de execução para oito *transputers* com precisão igual a 1.0E-4, 1.0E-6 e 1.0E-8

Comparando o tempo de execução entre dois, quatro e oito *transputers* (tabelas V.2, V.3 e V.4, respectivamente) dos sistemas testados em relação a um *transputer*, verifica-se um aumento no tempo de execução para os sistemas de 8 e 45 barras. Para sistemas com dimensões pequenas, ainda é melhor o algoritmo seqüencial. Nos sistemas de 118 barras, o tempo de execução foi menor nos sistemas de dois, quatro e oito *transputers* do que no caso seqüencial.

A tabela V.5, a tabela V.6 e a tabela V.7 mostram o ganho de velocidade (*speedup*) e a eficiência para uma rede com dois, quatro e oito *transputers*, respectivamente.

Sistema	S	η
8	0.38	19%
45	0.82	41.2%
118	1.36	68%
118*	1.37	68.3%

Tabela V.5: Ganho de velocidade e eficiência para dois *transputers*

Sistema	S	η
8	0.26	6.5%
45	0.92	23%
118	1.90	47.5%
118*	1.90	47.5%

Tabela V.6: Ganho de velocidade e eficiência para quatro *transputers*

Sistema	S	η
8	0.17	2.1%
45	0.87	10.9%
118	2.23	27.9%
118*	2.24	28%

Tabela V.7: Ganho de velocidade e eficiência para oito *transputers*

O melhor ganho de velocidade (2.24) foi obtido no sistema de 118 barras em oito *transputers*, porém a melhor eficiência (68%) foi alcançada no sistema com dois *transputers*. Esses dados demonstram a influência da comunicação entre os processadores no resultado final quando as dimensões do sistema são pequenas em relação a comunicação total realizada.

Para sistemas de dimensões maiores, espera-se que o tempo de comunicação se torne um fator não dominante no resultado final, quando a quantidade de processamento dos dados locais a cada processador for maior que o tempo de comunicação entre os processadores.

Numa comparação feita, buscamos uma implementação paralela para o algoritmo do gradiente conjugado. A implementação realizada por Cabral[11] utiliza o método do gradiente conjugado pré-condicionado paralelo e otimizado. Neste caso foi obtido uma relação de tempo de aproximadamente um para dez para a implementação o que representa um bom resultado, considerando que a biblioteca não está otimizada, a diferença de tempo de implementação (dias × meses), além da redundância de esforço de programação. É certo também que os valores dos ganhos de velocidade e de eficiência encontrados não podem ser diretamente comparados , pois os métodos implementados são diferentes e os tempos de comunicação foram apenas estimados, já que Cabral utilizou um simulador de uma máquina hipercúbica, o iPSC/860 da Intel sobre um PC 386. Além disso, esse algoritmo trata especialmente a esparsidade da matriz e utiliza o pré-condicionamento.

V.4 Resumo

Para validar e avaliar comparativamente a biblioteca de operações paralelas, empregamos o método do gradiente conjugado para solucionar um sistema linear algébrico.

A forma de implementação a partir dos passos elementares da biblioteca demonstram a sua simplicidade e a facilidade de depuração.

Para sistemas com dimensões pequenas, ainda é melhor a utilização do algoritmo seqüencial. Os resultados obtidos demonstram a influência do tempo de comunicação dos dados em sistemas de dimensões pequenas.

Para sistemas de dimensões maiores, espera-se que o tempo de comunicação se torne um fator não dominante no resultado final.

Capítulo VI

Conclusões

Neste trabalho realizamos a implementação, a validação e a avaliação de desempenho de uma biblioteca de operações paralelas que são passos elementares de algoritmos maiores. A biblioteca trata do armazenamento de vetores e matrizes, de operações básicas da álgebra linear, e de operações de difusão e convergência de dados. Este trabalho está relacionado ao modelo desenvolvido por Oliver Mcbryan e Eric Van de Velde [3].

A implementação de algoritmos a partir dos passos elementares da biblioteca de operações paralelas é um procedimento simples e possui a facilidade da modularidade para a depuração.

Toda a dependência da arquitetura da máquina e toda comunicação realizada entre os processadores é tornada invisível ao usuário, o que facilita a disseminação do uso de máquinas paralelas e aplicações numéricas.

O tempo de execução depende das características do problema a ser solucionado e da comunicação entre os processadores. O problema deve ter o tempo de processamento dos dados bem maior que o tempo de comunicação.

A vantagem de se utilizar a biblioteca é a facilidade de programação de algoritmos paralelos. E os resultados obtidos são promissores em relação ao tempo de execução.

Para otimização da biblioteca de operações paralelas, sugerimos como continuação desse trabalho, a avaliação de cada rotina em particular para definir parâmetros como:

- a partir de que quantidade de dados é mais conveniente utilizar esta operação da biblioteca de operações paralelas ?
- qual a melhor relação entre quantidade de processamento dos dados locais em relação a comunicação realizada ?
- qual a dimensão do cubo mais apropriada para uma determinada massa de dados ?

O prosseguimento desse trabalho deverá ser feito ainda, utilizando o sistema iPSC/860 da Intel, que foi recentemente instalado no Núcleo de Computação Paralela da COPPE/UFRJ. O iPSC representa a terceira geração da topologia hipercúbica da Intel.

Portando a biblioteca de operações paralelas para esta outra máquina, novas avaliações serão realizadas para comparação dos tempos obtidos.

Uma outra extensão do trabalho será portar a biblioteca de operações paralelas para ser utilizada sob o sistema operacional Helios, que possui ferramentas onde programas podem ser gerados a partir de seguimentos de códigos em *Fortran* e *C*, possibilitando uma utilização mais ampla em diversos algoritmos.

Referências Bibliográficas

[1] Hwang, Kai e Briggs, Fayé A., "Computer Architecture and Parallel Processing", McGraw Hill, Inc 1984.

[2] "User Manual", Transtech Devices LTD TSB44/48.

[3] McBryan, Oliver A. e Velde, Eric F. Van de, "Hypercube Algorithms and Implementations", SIAM J. Sci, Statistic Comput. 8 (1987) pp 227-287.

[4] Gilbert, E. N., "Gray Codes and Paths on n-Cube", Bell System Tech. J., 37 (1958), p 915.

[5] "Parallel C User Guide", 3L LTD, 1988.

[6] Dirk C. e Reed, Daniel A., "Networks For Parallel Processor Measurements and Prognostications", ACM 1988, pp 610 - 619.

[7] Lawson, C. L., Hanson, R. J., Kincaid, D. R. e Krogh, F. T., "Basic Linear Algebra Subprograms for Fortran Usage" ACM Transactions on Mathematical Software, Vol. 5, No 3, September 1979, pp 308-323.

[8] Dongarra, Jack J., Du Croz, Jeremy, Hammarling, Sven e Hanson, Richard J. "An Extended Set of Fortran Basic Linear Algebra Subprograms", ACM Transactions on Mathematical Software, Vol. 14, No 1, March 1988, pp 1-17.

[9] Ortega, James M., "Introduction to Parallel and Vector Solution of Linear Systems", Plenum Press, 1988.

[10] Athay, T., Sherkey, J. P., Podmore, R., Virmani, S. e Puech, C. (1980) "Transient Energy Stability Analysis", Conference on Sistem Engineering for Power: Emergency Operating State Control - Section IV, Davos, Switzerland 1979, also U.S. Dept. of Energy Publication no. Conf. 790904, Pl.

[11] Cabral, Roberto G., "Avaliação de Desempenho do Método dos Gradientes Conjugados em Multiprocessadores com Aquitetura Hipercúbica", Tese M.Sc. Engenharia Eleétrica, COPPE / UFRJ, 1991.

Apêndice A

Comunicação no Transputer

Neste apêndice são mostrados em detalhes os procedimentos empregados na realização da comunicação entre os processadores. A implementação dos procedimentos de comunicação dependem da definição de um arquivo de configuração que descreve todo o sistema hipercúbico. O arquivo de configuração é utilizado por uma ferramenta chamada configurador de propósito geral que é fornecida com o compilador da linguagem *C Paralela* do *transputer* para gerar programas executáveis.

A.1 Arquivo de Configuração

Para programas escritos na linguagem *C Paralela* do *transputer* existem ferramentas de programação fornecidas com o compilador que são utilizadas na prática. Uma dessas ferramentas é o configurador de propósito geral [5] que é um programa preparado para atender a um arquivo de configuração que descreve o sistema a ser utilizado e que produz arquivos executáveis para vários *transputers* e vários processos.

O arquivo de configuração deve conter toda a estrutura da rede de interconexão hipercúbica tais como a declaração dos processadores participantes, os canais de ligação física entre eles, a distribuição dos processos (tarefas) no sistema e suas ligações lógicas.

O primeiro passo a ser realizado é a configuração física da rede hipercúbica. As conexões físicas possíveis para um sistema básico com quatro *transputers* pode ser vista na figura A.1.

O *transputer* conectado ao computador hospedeiro (H) é chamado raiz (T0) da rede e toda a comunicação entre o hospedeiro e o sistema é realizada através dele.

Para a construção de um cubo de dimensão dois admissível no sistema básico (figura A.1) é necessário que o arquivo de configuração contenha as seguintes declarações relacionadas ao sistema físico:

- declaração de todos os processadores participantes da rede através do comando *processor*, como no exemplo a seguir:
  processor host

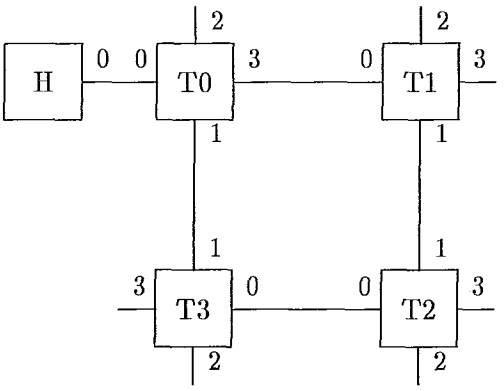


Figura A.1: Representação de um sistema básico com quatro *transputers*

processor raiz
⋮

- declaração das ligações físicas entre os processadores pelo comando *wire* exemplificado a seguir:
  wire ? host[0] raiz[0]
  wire ? raiz[3] nodo1[0]
  ⋮

Assim como o sistema físico, o arquivo de configuração deve conter todas as informações sobre o sistema lógico tais como a declaração das tarefas, a distribuição das mesmas aos processadores, suas quantidades de canais de entrada e saída para comunicação com outras tarefas, arquivo onde se encontra a tarefa a ser executada, tamanho e estratégia para armazenamento de memória, sua identificação e especificação de suas ligações lógicas com as outras tarefas.

Então para se declarar uma tarefa utiliza-se o comando *task* da seguinte forma:
task afserver ins = 2 outs = 2
task tarefa0 ins = 2 outs = 2 file = arq.b4 data = 10k
⋮

Os parâmetros *ins* e *outs* designam a quantidade de canais lógicos de entrada e saída, respectivamente, para comunicação entre as tarefas. O parâmetro *file* diz qual o arquivo onde se encontra a tarefa0 e o parâmetro *data* o tamanho da área de memória que pode ser usada pela tarefa.

O processador onde deve ser executada a tarefa é dado através do comando *place* como no seguinte exemplo:
place afserver host
place filter raiz
place tarefa0 raiz
⋮

Para estabelecer os canais lógicos de comunicação entre as tarefas é utili-

zado o comando *connect*. Se o canal for bidirecional, então devem ser declaradas as duas conexões como mostrado a seguir:
connect ? afserver[0] filter[0]
connect ? filter[0] afserver[0]
⋮

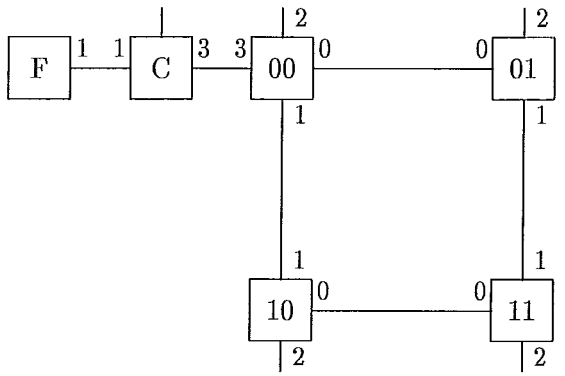
As tarefas empregadas neste exemplo têm um papel fundamental no sistema. A tarefa *afserver* se encontra num arquivo executável (.exe) do MS-DOS e é executada pelo computador hospedeiro (PC) em paralelo ao sistema *transputer*. Atua como um arquivo servidor, carregando os arquivos executáveis nos *transputers* e realizando a comunicação de entrada e saída dos dados com as tarefas executadas no sistema *transputer*. É através do canal bidirecional 0 que ela se comunica com a tarefa *filter*.

A tarefa *filter* é executada no processador raiz e compatibiliza os diferentes protocolos de comunicação entre a tarefa *afserver* e as tarefas executadas nos *transputers*. Deve ser interligada ao *afserver* pelo canal bidirecional 0 e ao processo que realiza a entrada e saída dos dados (processo de controle) pelo canal bidirecional 1.

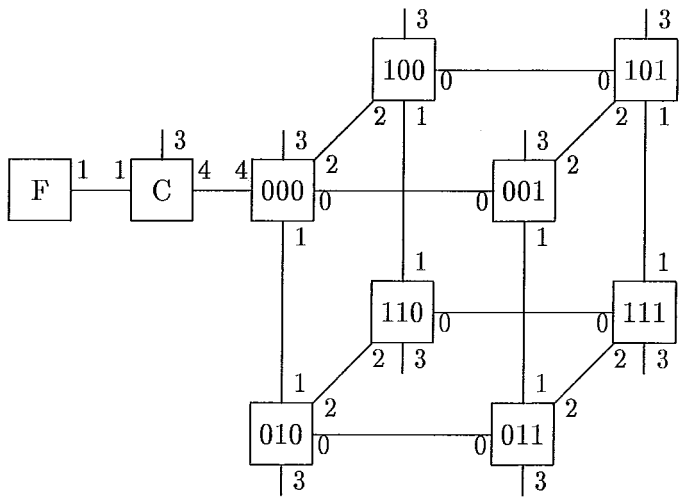
Para a construção de um sistema hipercúbico de dimensão D, sendo D igual ou maior a dois foram definidos os seguintes parâmetros lógicos:

- a identificação do processo de controle será sempre igual a P, onde P é a quantidade de processadores que formam a rede hipercúbica;
- a identificação dos outros processos é igual a identificação física dos processadores na rede, uma vez que todos os processadores só receberão um processo para ser executado;
- identificação dos processos realizada através dos canais lógicos D, onde D é a dimensão do cubo;
- ligação entre o processo do raiz e o processo de controle pelos canais lógicos $D + 1$ para ambos;
- ligação entre os outros processos de cada processador variam de 0 a $D - 1$, dependendo do bit de diferença entre os números de identificação dos processos, como no exemplo a seguir:
  ligação entre 000 e 010 canal lógico 1;
  ligação entre 000 e 100 canal lógico 2.

A figura A.2 mostra estes parâmetros para cubos de dimensão dois e três, onde a tarefa *filter* é identificada pela letra F, o processo de controle pelo C e os demais processos pelos números de identificação dos processadores.



Dimensão 2



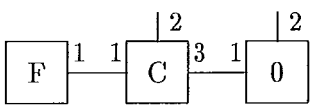
Dimensão 3

Figura A.2: Hipercubos lógicos de dimensões dois e três

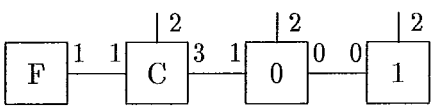
Essa especificação é válida para qualquer dimensão maior ou igual a dois. Para a construção de um sistema com dimensão igual a zero ou um foi adotada uma solução diferente das demais dimensões. Existe uma exigência na implementação da linguagem *C Paralela* que requer que o canal lógico 0 esteja livre no processo que realiza a entrada e saída dos dados (processo de controle), pois é através dele que a biblioteca padrão da linguagem *C* realiza a entrada e saída, e o canal lógico 1 é a ligação deste mesmo processo com a tarefa *filter* que compatibiliza os protocolos de comunicação como foi visto anteriormente. Então para as dimensões zero e um adotou-se as seguintes definições:

- a identificação do processo de controle será sempre igual a P, onde P é a quantidade de processadores que formam a rede hipercúbica;
- a identificação dos outros processos é igual a identificação física dos processadores na rede, uma vez que todos os processadores só receberão um processo para ser executado;
- identificação dos processos deve ser realizada através do canal lógico 2;
- ligação entre o processo do raiz e o processo de controle deve ser feita pelos canais lógicos 1 e 3 respectivamente, deixando assim os canais 0 e 1 livres no processo de controle.

Essas especificações para as dimensões zero e um podem ser vistas na figura A.3.



Dimensão 0



Dimensão 1

Figura A.3: Hipercubos lógicos de dimensões zero e um

A.2 Rotinas de Comunicação

As rotinas da biblioteca de operações paralelas que necessitam de comunicação entre os processadores, utilizam as rotinas da tabela A.1 que combinadas realizam a troca de mensagens no sistema *transputer*. Essas rotinas foram implementadas levando-se em consideração os parâmetros definidos na seção A.1.

canal_de_escrita ()	canal por onde a mensagem deve ser enviada
canal_de_leitura ()	canal por onde a mensagem deve ser recebida
roteamento ()	próximo processador a receber a mensagem
envia_mensagem ()	
recebe_mensagem ()	

Tabela A.1: Rotinas Relativas à Comunicação

A.2.1 Rotina de *roteamento*

A comunicação entre dois processos deve ser realizada de maneira coordenada. Para que não haja *deadlock* na comunicação é necessário se estabelecer um caminho para o envio e a recepção de mensagens. Para tanto o algoritmo de roteamento empregado utiliza o roteamento *E-Cube*[6]. O algoritmo é demonstrado a seguir e uma chamada a rotina de roteamento indica para qual processador deverá ser enviada a mensagem. O argumento dessa rotina é o processador destino da mensagem.

p_receptor = roteamento (p_destino)

p = identificação do processo;
Se p = processo de controle
  p_receptor = identificação do processo raiz;
Se p = processo do raiz
  Se *p_destino* = processo de controle
    p_receptor = identificação do processo de controle;
  Caso contrário
    Se p = *p_destino*
      p_receptor = *p_destino*;
    Caso contrário
      p_receptor = troca o bit menos significativo diferente entre
      os números de identificação de p e *p_destino*;
Se p = quaisquer outros processos;
  Se p = *p_destino*
    p_receptor = *p_destino*;
  Caso contrário
    p_receptor = troca o bit menos significativo diferente entre
    os números de identificação de p e *p_destino*;
Retorna *p_receptor*.

Assim, supondo um hipercubo de dimensão três em que uma mensagem deva ser enviada do processador cuja identificação é 000 ao processador 011, temos o seguinte caminho:

- 000 envia para 001 - processsador 000 trocou o bit 0 no seu número de identificação;
- 001 envia para 011 - processador 001 trocou o bit 1 no seu número de identificação.

Todos os demais exemplos empregados neste apêndice para demonstrar as rotinas utilizadas na comunicação supõem um hipercubo de dimensão três e a troca de mensagens entre os processadores 000 e 011.

A.2.2 Rotina *canal_de_escrita*

No *transputer* toda comunicação é realizada através da troca de mensagens pelos canais lógicos declarados no arquivo de configuração (seção A.1). Os canais pelos quais as mensagens devem ser enviadas durante a comunicação (aqui chamados canais de escrita) são obtidos com uma chamada a rotina

canal = canal_de_escrita(p_destino)

que retorna o valor do canal segundo o roteamento *E-Cube* e o arquivo de configuração. O algoritmo para a obtenção do canal é mostrado a seguir.

p = identificação do processo;
D = dimensão do cubo;
Se p = processo de controle
  Se $D = 0$ ou 1
    canal = 3;
  Caso contrário
    canal = $D + 1$;
Se p = processo do raiz
  Se *p_destino* = processo de controle
    Se $D = 0$ ou 1
      canal = 3;
    Caso contrário
      canal = $D + 1$;
  Caso contrário
    canal = bit menos significativo diferente entre os números
    de identificação de p e *p_destino*;
Se p = qualquer outro processo
  canal = bit menos significativo diferente entre os números
  de identificação de p e *p_destino*;
Retorna *canal*.

O canal pelo qual a mensagem do processador 000 deve ser enviada ao processador 011 é o canal 0, que corresponde ao bit menos significativo diferente entre os seus números de identificação.

A.2.3 Rotina *canal_de_leitura*

Assim como a rotina *canal_de_escrita* que define os canais de escrita envolvidos na comunicação, a rotina *canal_de_leitura* define os canais pelos quais as mensagens devem ser recebidas durante a comunicação (chamados canais de leitura) segundo o roteamento e o arquivo de configuração utilizados. A chamada da rotina é

canal = canal_de_leitura(p_fonte)

cujo argumento é o processador de onde a mensagem virá e o seu algoritmo é apresentado a seguir.

p = identificação do processo;
D = dimensão do cubo;
Se p = processo de controle
  Se $D = 0$ ou 1
    $canal = 3$;
  Caso contrário
    $canal = D + 1$;
Se p = processo do raiz
  Se p_fonte = processo de controle
    Se $D = 0$ ou 1
      $canal = 3$;
    Caso contrário
      $canal = D + 1$;
  Caso contrário
    $canal$ = bit mais significativo diferente entre os números
    de identificação de p e p_fonte;
Se p = qualquer outro processo
  $canal$ = bit mais significativo diferente entre os números
  de identificação de p e p_fonte;
Retorna $canal$.

O canal pelo qual o processador 011 deve receber a mensagem do processador 000 é o canal 1, que corresponde ao bit mais significativo diferente entre os seus números de identificação 000 e 011.

A.2.4 Rotina *envia_mensagem*

Para a troca de mensagens entre processadores é utilizada a rotina *envia_mensagem* que, tendo como parâmetros o identificador do processo para onde a mensagem deve ser enviada, seu o tamanho e o ponteiro do local de memória onde ela se encontra, transfere a mensagem para o próximo processador no caminho estabelecido pela rotina de roteamento descrita na seção A.2.1. A chamada da rotina *envia_mensagem* e o seu algoritmo podem ser vistos a seguir.

envia_mensagem (p_destino, tam_buf, pt_buf)

Armazena área de memória para transferência da mensagem;
Transfere dados apontados pelo *pt_buf* para a área armazenada;
Formata identificador da mensagem a ser enviada;
$p\_receptor = roteamento(p\_destino)$;
$canal = canal\_de\_escrita(p\_receptor)$;
Envia identificador da mensagem;
Envia mensagem;
Libera área de transferência armazenada;
Retorna tam_buf.

A.2.5 Rotina *recebe_mensagem*

A rotina *recebe_mensagem* é complementar a rotina *envia_mensagem*, cujos parâmetros especificam o tamanho da mensagem, o local onde deve ser armazenada e o canal pelo qual deve ser recebida. A sua chamada é

recebe_mensagem (tam_buf, pt_buf, can)

e que ao ser recebida a mensagem, verifica seu identificador, e se chegou ao seu destino ou se deve ser encaminhada a um novo processador. A seguir é mostrado o seu algoritmo.

Armazena área de memória para o identificador da mensagem;
Armazena área de memória para transferência da mensagem;
Recebe identificador da mensagem;
Recebe mensagem;
p = identificação do processo;
Se $p \neq p\_destino$
    $p\_receptor = roteamento(p\_destino)$;
    $canal = canal\_de\_escrita(p\_receptor)$;
    Envia identificador da mensagem;
    Envia mensagem;
Caso contrário
    Transfere dados recebidos para a área apontada pelo *pt_buf*;
Libera área de memória do identificador da mensagem;
Libera área de transferência armazenada;
Retorna p_fonte.

Apêndice B

Rotinas Vetoriais Básicas

Este apêndice apresenta todos os procedimentos vetoriais básicos implementados.

Todos os processadores recebem uma cópia da biblioteca de operações paralelas e do programa a ser executado, bem como os vetores distribuídos adequadamente. As rotinas são chamadas simultaneamente por todos os processadores que operam sobre os seus segmentos locais ignorando a existência de seus vizinhos na rede.

As rotinas vetoriais e seus propósitos podem ser vistos na tabela B.1.

aloca_vetor	armazena segmento local de um vetor
deleta_vetor	libera memória armazenada de um vetor
converte_vetor	troca o tipo de armazenamento
vetor_nulo	$u_i = 0$
copia_vetor	$u_i = v_i$
soma_vet_a_esc_vet	$u_i = u_i + sv_i$
soma_esc_vet_a_vet	$u_i = su_i + v_i$
soma_vet_a_vet_vet	$u_i = u_i + v_i w_i$
desloca_vetor	$u_i = u_{(i-s) \bmod N}$
desloca_vetor_ótimo	$u_i = u_{(i-s) \bmod N}$
máximo_vetor	$max_{0 \leq i < N} u_i$
soma_vetor	$\sum_{i=0}^{N-1} u_i$
produto_interno	$\sum_{i=0}^{N-1} u_i v_i$

Tabela B.1: Rotinas Vetoriais

B.1 Rotina *aloca_vetor*

Esta rotina reserva uma área de memória para um vetor através da chamada

pte_vetor = aloca_vetor, tipo

cujos parâmetros são o tamanho total do vetor N e a tipo de armazenamento vetorial *tipo*. Existem dois tipos de representação vetorial que são chamados tipo *SIMPLE* e *SHIFT*. Ela retorna um ponteiro para a estrutura de dados que representa o vetor armazenado.

O algoritmo da rotina *aloca_vetor* é mostrado a seguir:

pte_vetor = armazena área para a estrutura de dados que
representa o vetor;
N = tamanho total do vetor a ser distribuído;
P = processadores existentes na rede hipercúbica;
posição = posição do processador na seqüência BRGC;
$h = N/P$;
$r = N \bmod P$;
Se *tipo* = *SIMPLE*
  Se $r \neq 0$ e *posição* $< r$
    $tam\_buf = h + 1$;
  Caso contrário
    $tam\_buf = h$;
Se *tipo* = *SHIFT*
  Se $r \neq 0$ e *posição* $< r$
    $tam\_buf = 2h + 1$;
  Caso contrário
    $tam\_buf = 2h$;
pt_vetor = armazena área de dados;
Preenche a estrutura de dados convenientemente;
Retorna *pte_vetor*.

B.2 Rotina *deleta_vetor*

Este procedimento é bastante simples, pois libera a área de memória ocupada pelo vetor. Ele tem como argumento o ponteiro da estrutura que representa o vetor que se quer liberar. A sua chamada e o seu algoritmo são mostrados a seguir.

deleta_vetor (pte_vetor)

Libera área do segmento local ao processador;
Libera área da estrutura de dados que representa o vetor.

B.3 Rotina *converte_vetor*

A rotina *converte_vetor* troca o tipo de armazenamento vetorial para o tipo conveniente à aplicação. Existem operações que necessitam de um determinado tipo de armazenamento, como exemplo temos o deslocamento vetorial que necessita que o vetor esteja armazenado no formato *SHIFT*. Suponhamos que este estivesse inicialmente no formato *SIMPLE* e necessitássemos realizar um deslocamento, antes teríamos que utilizar a rotina

converte_vetor(pte_vetor)

tendo como argumento o ponteiro da estrutura que representa o vetor para realizar a conversão necessária.

Esta rotina é realizada sem comunicação entre os processadores, calcula o novo tamanho do *array* onde será armazenado o vetor, transfere os dados para a nova área e preenche a estrutura de representação do vetor convenientemente. O seu algoritmo é apresentado a seguir.

h = tamanho do segmento local ao processador;
r = quantidade de processadores que receberão $h + 1$ elementos;
posição = posição do processador na seqüência BRGC;
Se *tipo* = *SHIFT*
  $tam\_buf = h$;
Se *tipo* = *SIMPLE*
  Se $r \neq 0$ e *posição* $< r$;
    $tam\_buf = 2h + 1$;
  Caso contrário
    $tam\_buf = 2h$;
pt_vet = armazena nova área de dados;
Transfere dados para a nova área;
Atualiza estrutura de dados do vetor.

B.4 Rotina *vetor_nulo*

A rotina *vetor_nulo* tem como argumentos o tamanho total do vetor e o tipo de armazenamento vetorial desejado. Ela cria um vetor nulo distribuído entre todos os processadores e como as rotinas anteriores não troca mensagens entre os processadores. Então com a chamada

pte_vetor = vetor_nulo (N, tipo)

temos em *pte_vetor* o ponteiro da estrutura de dados de um vetor nulo. O seu algoritmo é demonstrado a seguir.

pte_vetor = aloca_vetor (N, tipo);
Preenche com valores nulos a área de dados;
Retorna *pte_vetor*.

B.5 Rotina *copia_vetor*

Nesta rotina não há comunicação entre os processadores. Ela cria um vetor igual ao vetor passado como parâmetro na chamada

pte_vetor = copia_vetor (pte_vet)

pte_vetor = armazena área para a estrutura de dados do novo vetor;
pt_vetor = armazena área para os dados do novo vetor;
Preenche a estrutura de dados do novo vetor;
Copia a área de dados para o novo vetor;
Retorna *pte_vetor* do novo vetor.

B.6 Rotinas de soma e multiplicação de vetores por escalares

Nestes procedimentos o código serial é executado por todos os processadores simultaneamente e não há comunicação entre os processadores. Para o procedimento que realiza a operação

$$u_i = u_i + ev_i$$

passamos como parâmetros as estruturas de dados dos dois vetores e o escalar envolvidos na operação, assim temos:

soma_vet_a_esc_vet (pte_u, pte_v, escalar).

h = tamanho do segmento local a cada processador;
Para $i = 0$ a $i < h$ faça
    $u_i = u_i + escalar\ v_i$;

Da mesma forma para a operação

$$u_i = eu_i + v_i$$

enviamos como parâmetros as estruturas de dados dos vetores e o escalar envolvidos na operação, então temos:

soma_esc_vet_a_vet (pte_u, pte_v, escalar)

h = tamanho do segmento local a cada processador;
Para $i = 0$ a $i < h$ faça
    $u_i = escalar\ u_i + v_i$;

A terceira e última operação de soma envolvendo vetores é

$$u_i = u_i + v_i w_i$$

que tem passados como parâmetros as estruturas dos três vetores que participam da operação, assim temos:

soma_vet_a_vet_vet (pte_u, pte_v, pte_w)

h = tamanho do segmento local a cada processador;
Para $i = 0$ a $i < h$ faça
    $u_i = u_i + v_i\ w_i$;

B.7 Rotinas de deslocamento vetorial

Existem duas rotinas implementadas para realizar o deslocamento vetorial, a *desloca_vetor* e a *desloca_vetor_ótimo*, ambas tem como argumento o ponteiro da estrutura do vetor a ser deslocado e a quantidade de deslocamentos desejada.

B.7.1 Rotina *desloca_vetor*

A rotina *desloca_vetor* é utilizada quando o tamanho total do vetor não é um múltiplo do número de processadores. O algoritmo da rotina de deslocamento vetorial é mostrado a seguir.

desloca_vetor(pte_vetor, m)

Se *tipo* $\neq$ *SHIFT*
  converte_vetor (pte_vetor);
anel();
n = quantidade de elementos transferidos;
Se $m > 0$
  p_direita = *estrutura_de_anel[0].direita*;
  p_esquerda = *estrutura_de_anel[0].esquerda*;
  canal = *canal_de_leitura(p_esquerda)*;
  $n = 0$;
  Enquanto $i < m$ faça
    Posiciona ponteiros auxiliares de deslocamento;
    Envia elemento ao *p_direita*;
    Recebe elemento do *p_esquerda* através do *canal*;
    Incrementa n;
    Se ponteiro auxiliar de deslocamento chegou ao final
    do *buffer*
      centraliza_vetor (pte_vetor, direção, n);
      Posiciona ponteiros auxiliares de deslocamento;
      $n = 0$;
Caso contrário
  p_direita = estrutura_de_anel[0].direita;
  p_esquerda = estrutura_de_anel[0].esquerda;
  canal = *canal_de_leitura(p_direita)*;
  $n = 0$;
  Enquanto $i <$ *abs(m)* faça
    Posiciona ponteiros auxiliares de deslocamento;
    Envia elemento ao *p_esquerda*;
    Recebe elemento do *p_direita* através do *canal*;
    Incrementa n;
    Se ponteiro auxiliar de deslocamento chegou ao final
    do *buffer*
      centraliza_vetor (pte_vetor, direção, n);
      Posiciona ponteiros auxiliares de deslocamento;
      $n = 0$;
Se $n > 0$
  centraliza_vetor (pte_vetor, direção, n);

B.7.2 Rotina *desloca_vetor_ótimo*

Esta operação de deslocamento vetorial é utilizada somente quando o vetor é um múltiplo do número de processadores ($N = hP$). O algoritmo *desloca_vetor_ótimo* pode ser visto a seguir.

desloca_vetor_ótimo(pte_vetor, m)

Se *tipo* $\neq$ *SHIFT*
  converte_vetor (pte_vetor);
anel();
h = tamanho do segmento vetorial local ao processador;
P = quantidade de processadores existentes no hipercubo;
$m\_segmento = m/h$;
$m\_elemento = m \bmod m\_segmento$;
Se $m\_segmento \geq P$
  $m\_segmento = m\_segmento \bmod P$;
Se $m\_segmento > P/2$
  $m\_segmento = m\_segmento - P$;
Para todos os níveis i ativados em $m\_segmento$ faça
  Se $m\_segmento > 0$
    $p\_direita$ = estrutura_de_anel[i].direita;
    $p\_esquerda$ = estrutura_de_anel[i].esquerda;
    canal = *canal_de_leitura(p_esquerda)*;
    Posiciona ponteiros auxiliares de deslocamento;
  Caso contrário
    $p\_direita$ = estrutura_de_anel[i].direita;
    $p\_esquerda$ = estrutura_de_anel[i].esquerda;
    canal = *canal_de_leitura(p_direita)*;
    Posiciona ponteiros auxiliares de deslocamento;
  Envia segmento ao processador vizinho no anel de nível i;
  Recebe segmento do processador vizinho no anel de nível i
  através do *canal*;
Se $m\_elemento \neq 0$
  desloca_vetor (pte_vetor, m_elemento);

B.8 Rotina *máximo_vetor*

Para obter o elemento máximo em um vetor basta que todos os processadores simultaneamente utilizem a rotina

máximo = máximo_vetor (pte_vetor)

tendo como argumento o ponteiro da estrutura do vetor. Ao retornar desta rotina todos os processadores terão o valor do elemento máximo do vetor distribuído. A rotina *máximo_vetor* tem o seu algoritmo mostrado a seguir.

D = dimensão do hipercubo;
pt_árvore = árvore_binária ();
máximo = máximo_vetor_local (pte_vetor);
Para $i = 0$ a $i < D$ faça
    Se tem filhos
        Recebe *máximo* dos filhos;
    Se tem pai
        Envia *máximo* ao pai;
    máximo = maior elemento entre os máximos local e dos filhos;
Para $i = 0$ a $i < D$ faça
    Se tem filhos
        Envia *máximo* aos filhos;
    Se tem pai
        Recebe *máximo* do pai;
Retorna *máximo*.

B.9 Rotina *soma_vetor*

Domesmo modo que o procedimento para encontrar o elemento máximo de um vetor, no final da operação de somar todos os elementos de um vetor, todos os processadores terão o mesmo resultado da soma. A sua chamada e o seu algoritmo são apresentados a seguir.

soma = soma_vetor(pte_vetor)

D = dimensão do hipercubo;
pt_árvore = árvore_binária ();
minha_soma = soma_vetor_local (pte_vetor);
Para $i = 0$ a $i < D$ faça
    Se tem filhos
        Recebe soma dos filhos;
    Se tem pai
        Envia *minha_soma* ao pai;
    soma = minha_soma+ soma dos filhos;
Para $i = 0$ a $i < D$ faça
    Se tem filhos
        Envia *soma* aos filhos;
    Se tem pai
        Recebe *soma* do pai;
Retorna *soma*.

B.10 Rotina *produto_interno*

O produto interno é obtido com o procedimento

produto_interno = produto_interno(pte_x,pte_y)

cujos argumentos são os ponteiros das estruturas dos vetores. Ao final dessa operação todos os processadores terão o mesmo resultado. O seu algoritmo pode ser visto a seguir.

D = dimensão do hipercubo;
pt_árvore = árvore_binária ();
meu_produto_interno = produto_interno_local (pte_vetor);
Para $i = 0$ a $i < D$ faça
    Se tem filhos
        Recebe produto interno dos filhos;
    Se tem pai
        Envia *meu_produto_interno* ao pai;
    produto_interno = meu_produto_interno+ produtos internos dos filhos;
Para $i = 0$ a $i < D$ faça
    Se tem filhos
        Envia *produto_interno* aos filhos;
    Se tem pai
        Recebe *produto_interno* do pai;
Retorna *produto_interno*.

Apêndice C

Rotinas Matriciais

Este apêndice mostra detalhadamente todos os algoritmos matriciais implementados que estão disponíveis na biblioteca de operações paralelas.

A tabela C.1 apresenta as rotinas matriciais disponíveis na biblioteca de operações paralelas.

aloca_matriz	armazena segmento local de uma matriz
deleta_matriz	libera memória de uma matriz
converte_matriz	troca o tipo de armazenamento matricial
transposição_por_deslocamento	transposição matricial
transposição_por_bloco	transposição matricial
matriz_vetor	multiplicação de uma matriz por um vetor
matriz_matriz	multiplicação de matrizes
rank1_update	$A \leftarrow A + xy^T$
shuffle	troca os limites de área com os vizinhos

Tabela C.1: Rotinas Matriciais

C.1 Rotina *aloca_matriz*

O armazenamento de uma matriz é realizado pelo procedimento

pte_matriz = aloca_matriz (pte_mat, tipo)

que recebe como parâmetros uma estrutura descritiva da matriz com as informações sobre a dimensão e esparsidade da matriz e o tipo de armazenamento matricial. Existem seis tipos diferentes disponíveis de armazenamento matricial: *LINHA_CONTÍGUA*, *LINHA_DISTRIBUÍDA*, *COLUNA_CONTÍGUA*, *COLUNA_DISTRIBUÍDA*, *DIAGONAL* e *ÁREA*. Ao final da operação de armazenamento matricial é retornado um ponteiro para a estrutura de dados que representa a matriz. O algoritmo de armazenamento matricial é mostrado a seguir:

tipo = tipo de armazenamento matricial;
Caso *tipo*
  LINHA_CONTÍGUA :
  pte_matriz = aloc_tipo_0_3 (pte_mat, tipo);
  LINHA_DISTRIBUÍDA :
  pte_matriz = aloc_tipo_1_2_4 (pte_mat, tipo);
  COLUNA_CONTÍGUA :
  pte_matriz = aloc_tipo_1_2_4 (pte_mat, tipo);
  COLUNA_DISTRIBUÍDA :
  pte_matriz = aloc_tipo_0_3 (pte_mat, tipo);
  DIAGONAL :
  pte_matriz = aloc_tipo_1_2_4 (pte_mat, tipo);
  ÁREA :
  pte_matriz = aloc_tipo_5 (pte_mat, tipo);
Retorna *pte_matriz*;

aloc_tipo_0_3
pte_matriz = armazena área para a estrutura de dados que representa a matriz;
linha = quantidade total de linhas da matriz;
coluna = quantidade total de colunas da matriz;
P = processadores existentes na rede hipercúbica;
posição = posição do processador na seqüência BRGC;
$h = linha/P$;
$r = linha \bmod P$;
Caso *tipo*
    LINHA_CONTÍGUA :
    Se *linha_esparsa* = 0
        Se $r \neq 0$ e *posição* $< r$
            $h = h + 1$;
        $tam\_buf = h * coluna$;
    Caso contrário
        n = quantidade de linhas não nulas;
        $tam\_buf = n * coluna$;
    COLUNA_DISTRIBUÍDA :
    Se $r \neq 0$ e *posição* $< r$
            $h = h + 1$;
    Se *coluna_esparsa* = 0
        $tam\_buf = h * coluna$;
    Caso contrário
        n = quantidade de colunas não nulas;
        $tam\_buf = h * n$;
pt_matriz = armazena área de dados;
Preenche a estrutura de dados convenientemente;
Retorna *pte_matriz*.

aloc_tipo_1_2_4
pte_matriz = armazena área para a estrutura de dados que representa a matriz;
linha = quantidade total de linhas da matriz;
coluna = quantidade total de colunas da matriz;
P = processadores existentes na rede hipercúbica;
posição = posição do processador na seqüência BRGC;
$h = coluna/P$;
$r = coluna \bmod P$;
Caso *tipo*
LINHA_DISTRIBUÍDA :
Se $r \neq 0$ e *posição* $< r$
$h = h + 1$;
Se *linha_esparsa* $= 0$
tam_buf $= h * linha$;
Caso contrário
n = quantidade de linhas não nulas;
tam_buf $= h * n$;
COLUNA_CONTÍGUA :
Se *coluna_esparsa* $= 0$
Se $r \neq 0$ e *posição* $< r$
$h = h + 1$;
tam_buf $= h * linha$;
Caso contrário
n = quantidade de colunas não nulas;
tam_buf $= n * linha$;
DIAGONAL :
Se $r \neq 0$ e *posição* $< r$
$h = h + 1$;
Se *diagonal_esparsa* $= 0$
tam_buf $= h * linha$;
Caso contrário
n = quantidade de diagonais não nulas;
tam_buf $= h * n$;
pt_matriz = armazena área de dados;
Preenche a estrutura de dados convenientemente;
Retorna *pte_matriz*.

aloc_tipo_5
pte_matriz = armazena área para a estrutura de dados que representa a matriz;
linha = quantidade total de linhas da matriz;
coluna = quantidade total de colunas da matriz;
P = processadores existentes na rede hipercúbica;
posição = posição do processador na seqüência BRGC;
$nx = coluna/P$;
$rx = coluna \bmod P$;
$ny = linha/P$;
$ry = linha \bmod P$;
b = limite simulado;
x = posição do processador na direção x na grade;
y = posição do processador na direção y na grade;
Se $rx \neq 0$ e $x < rx$
    $nx = nx + 1$;
Se $ry \neq 0$ e $y < ry$
    $ny = ny + 1$;
$tam\_buf = (nx + 2b) * (ny + 2b)$;
pt_matriz = armazena área de dados;
Preenche a estrutura de dados convenientemente;
Retorna *pte_matriz*.

C.2 Rotina *deleta_matriz*

Esta rotina

deleta_matriz (pte_matriz)

libera a área de memória armazenada para uma matriz. O seu algoritmo é bastante simples e é apresentado a seguir:

Libera área do segmento matricial local ao processador;
Libera área da estrutura de dados que representa a matriz.

C.3 Operação *shuffle*

A operação *shuffle* atualiza os elementos externos de uma submatriz pela troca das linhas e/ou colunas externas com os processadores vizinhos no hipercubo. A operação é realizada com o seguinte procedimento

shuffle (pte_matriz, largura, lados)

Somente as linhas e colunas externas especificadas pelo parâmetro *largura* são trocadas. O parâmetro *lados* especifica quais os lados envolvidos na operação. Existem quatro constantes simbólicas *NORTE*, *SUL*, *LESTE* e *OESTE* que identificam os lados da submatriz que serão trocados realizando uma operação **or** entre elas.

tipo = tipo de armazenamento matricial;
Se *tipo* $\neq$ *ÁREA*
    converte_matriz (ÁREA, pte_matriz);
ny = quantidade de linhas locais ao processador;
nx = quantidade de colunas locais ao processador;
b = limite simulado;
linha = $ny + 2b$;
coluna = $nx + 2b$;
Se *lado* = *NORTE* ativado
    Armazena *buffer* de transferência;
    Transfere dados para o *buffer*;
    Envia dados para o *NORTE*;
    Recebe dados do *SUL*;
    Libera *buffer* de transferência;
Se *lado* = *SUL* ativado
    Armazena *buffer* de transferência;
    Transfere dados para o *buffer*;
    Envia dados para o *SUL*;
    Recebe dados do *NORTE*;
    Libera *buffer* de transferência;
Se *lado* = *LESTE* ativado
    Armazena *buffer* de transferência;
    Transfere dados para o *buffer*;
    Envia dados para o *LESTE*;
    Recebe dados do *OESTE*;
    Libera *buffer* de transferência;
Se *lado* = *OESTE* ativado
    Armazena *buffer* de transferência;
    Transfere dados para o *buffer*;
    Envia dados para o *OESTE*;
    Recebe dados do *LESTE*;
    Libera *buffer* de transferência;

C.4 Multiplicação *matriz_vetor*

A multiplicação de uma matriz por um vetor é obtida com o procedimento

pte_vetor_resultante = matriz_vetor (pte_matriz, pte_vetor)

cujos argumentos são os ponteiros das estruturas da matriz e do vetor envolvidos na operação. Ao final desta operação é retornado um ponteiro para a estrutura de dados do vetor resultante. O seu algoritmo é mostrado a seguir:

Se *tipo* $\neq$ *SIMPLE*
  converte_vetor (pte_x);
Se *tipo* $\neq$ *DIAGONAL*
  converte_matriz (DIAGONAL, pte_A);
N = tamanho total do vetor;
pte_y = vetor_nulo (N, SHIFT);
pte_Dm = aloca_vetor (N, SIMPLE);
Se *linha_esparsa* = 0
  Para $m = 0$ a $m <$ total de linhas da matriz faça
    pte_Dm = linha[m];
    soma_vet_a_vet_vet (pte_y, pte_Dm, pte_x);
    desloca_vetor (pte_y, 1);
Caso contrário
  Para $m = 0$ a $m <$ total de linhas da matriz faça
    n = contador de linhas nulas;
    Se m não é nula
      pte_Dm = linha[m];
      soma_vet_a_vet_vet (pte_y, pte_Dm, pte_x);
      $n = 1$;
    Caso contrário
      $n = n + 1$;
    desloca_vetor (pte_y, n);
deleta_vetor (pte_Dm);
Retorna *pte_y*.

C.5 Multiplicação *matriz_matriz*

O procedimento

pte_matriz_resultante = matriz_matriz (pte_A, pte_B)

cujos parâmetros enviados são as estruturas de dados das matrizes, realiza a multiplicação entre elas. A matriz resultante desta operação é conhecida por um ponteiro enviado como resultado final da operação. O seu algoritmo é mostrado a seguir:

Se *tipo* $\neq$ *DIAGONAL*
  converte_matriz (DIAGONAL, pte_A);
Se *tipo* $\neq$ *DIAGONAL*
  converte_matriz (DIAGONAL, pte_B);
pte_C = armazena área para a matriz C resultante;
pte_DAm = armazena matriz A vetores distribuídos do tipo *SHIFT*;
pte_DBm = armazena matriz B vetores distribuídos do tipo *SIMPLE*;
pte_DCm = armazena matriz C vetores distribuídos do tipo *SIMPLE*;
Se *linha_esparsa* da matriz $A = 0$
  Se *linha_esparsa* da matriz $B = 0$
    Para $q = 0$ a $q <$ total de linhas da matriz B faça
      Para $i = 0$ a $i <$ total de linhas da matriz A faça
        $p = (i + q)$ mod *linha*;
        pte_DC = pte_DCm[p];
        pte_DA = pte_DAm[i];
        pte_DB = pte_DBm[q];
        soma_vet_a_vet_vet (pte_DC, pte_DA, pte_DB);
        desloca_vetor (pte_DA, 1);
  Caso contrário
    Para $q = 0$ a $q <$ total de linhas da matriz B faça
      b = contador de linhas nulas da matriz B;
      Se q não é nula
        Para $i = 0$ a $i <$ total de linhas da matriz A faça
          $p = (i + q)$ mod *linha*;
          pte_DC = pte_DCm[p];
          pte_DA = pte_DAm[i];
          pte_DB = pte_DBm[q];
          soma_vet_a_vet_vet (pte_DC, pte_DA, pte_DB);
          $b = 1$;
      Caso contrário
        $b = b + 1$;
      desloca_vetor (pte_DA, b);

Caso contrário
    Se *linha_esparsa* da matriz $B = 0$
        Para $q = 0$ a $q <$ total de linhas da matriz B faça
            Para $i = 0$ a $i <$ total de linhas da matriz A faça
                Sei não é nula
                    $p = (i + q)$ mod *linha*;
                    pte_DC = *pte_DCm[p]*;
                    pte_DA = *pte_DAm[i]*;
                    pte_DB = *pte_DBm[q]*;
                    soma_vet_a_vet_vet (pte_DC, pte_DA, pte_DB);
                    desloca_vetor (pte_DA, 1);
    Caso contrário
        Para $q = 0$ a $q <$ total de linhas da matriz B faça
            b = contador de linhas nulas da matriz B;
            Se q não é nula
                Para $i = 0$ a $i <$ total de linhas da matriz A faça
                    Se i não 'e nula
                        $p = (i + q)$ mod *linha*;
                        pte_DC = *pte_DCm[p]*;
                        pte_DA = *pte_DAm[i]*;
                        pte_DB = *pte_DBm[q]*;
                        soma_vet_a_vet_vet (pte_DC, pte_DA, pte_DB);
                        $b = 1$;
            Caso contrário
                $b = b + 1$;
            desloca_vetor (pte_DA, b);
deleta_matriz (pte_DAm);
deleta_matriz (pte_DBm);
Retorna *pte_C*.

C.6 Rotina *rank1_update*

A operação *rank one update* é realizada com o seguinte procedimento

rank1_update (pte_A, pte_x, pte_y)

cujos argumentos são a matriz e os vetores envolvidos na operação. O seu algoritmo é apresentado a seguir:

Se *tipo* $\neq$ *SHIFT*
  converte_vetor (pte_x);
Se *tipo* $\neq$ *SIMPLE*
  converte_vetor (pte_y);
Se *tipo* $\neq$ *DIAGONAL*
  converte_matriz (DIAGONAL, pte_A);
N = tamanho total do vetor;
pte_DA = aloca_vetor (N, SIMPLE);
Para $m = 0$ a $m <$ total de linhas da matriz faça
  Se *diagonal_esparsa* $= 0$
    pte_DA $= 0$;
  Caso contrário
    pte_DA = linha[m];
  soma_vet_a_vet_vet (pte_DA, pte_x, pte_y);
  desloca_vetor (pte_x, 1);
deleta_vetor (pte_DA);

C.7 Rotina *transposição_por_deslocamento*

Esse procedimento utiliza a forma diagonal e operações de deslocamento vetorial. Somente deve ser utilizado quanto a dimensão da matriz for um múltiplo do número de processadores que compõem a rede hipercúbica. Assim temos

pte_matriz = transposição_por_deslocamento (pte_A)

cujo argumento é o ponteiro da estrutura de dados que representa a matriz e retorna um ponteiro para a estrutura de dados da matriz transposta de A. A seguir é mostrado o seu algoritmo.

Se *tipo* $\neq$ *LINHA_DISTRIBUÍDA*
  converte_matriz (LINHA_DISTRIBUÍDA, pte_matriz);
N = tamanho do vetor igual a quantidade de colunas da matriz;
pte_v1 = aloca_vetor (N, SHIFT);
pte_v2 = aloca_vetor (N, SHIFT);
pte_matriz = ponteiro da matriz transposta;
Para $m = 0$ a $m <$ quantidade de linhas da matriz faça
  $K = (N - m) \bmod N$;
  Para $i = 0$ a $i <$ quantidades de linhas faça
    Para $j = 0$ a $j <$ quantidade de colunas locais ao
    processador faça
      $x = j - i$;
      Se linha_esparsa = 0
        Se $(x + N) \bmod N) = K$
          Se $x > 0$
            Preenche vetor v1;
          Caso contrário
            Preenche vetor v2;
        Caso contrário
          Se i não é nula
            Se $(x + N) \bmod N) = K$
              Se $x > 0$
                Preenche vetor v1;
              Caso contrário
                Preenche vetor v2;
  desloca_vetor_ótimo (pte_v1, m);
  desloca_vetor_ótimo (pte_v2, m);
deleta_vetor (pte_v1);
deleta_vetor (pte_v2);
Preenche estrutura de dados convenientemente;
Retorna *pte_matriz.*

C.8 Rotina *transposição_por_bloco*

A rotina

pte_matriz = transposição_por_bloco (pte_A)

é uma outra forma de se obter a transposição matricial. Seu argumento é o ponteiro da estrutura de dados da matriz. Utiliza a forma de armazenamento por linhas contíguas e pode ser usada com matrizes de qualquer dimensão. Retorna também um ponteiro para

a matriz transposta de A. O seu algoritmo é mostrado a seguir:

Se *tipo* $\neq$ *LINHA_CONTÍGUA*
    converte_matriz (LINHA_CONTÍGUA, pte_A);
linha = quantidade de linhas da matriz;
coluna = quantidade de colunas da matriz;
D = dimensão do hipercubo;
p = identificação do processo;
pte_matriz = ponteiro da matriz tranposta;
Para $L = 0$ a $L < D$ faça
    diferença = potência (2, (D-L-1));
    nodo = p **exor** *diferença*;
    Calcula tamanho da nova área de dados;
    Armazena *buffer* de envio e recepção;
    Envia dados para *nodo*;
    Recebe dados do *nodo*;
Preenche estrutura de dados convenientemente;
Retorna *pte_matriz.*

Apêndice D

Aplicação em Sistemas Lineares

Neste apêndice são mostrados os procedimentos empregados na solução de um sistema linear, utilizando as rotinas da biblioteca de operações paralelas. O método empregado foi o do gradiente conjugado. No implemento, a solução foi dividida em dois processos: o processo de controle e o processo dos nodos.

O processo de controle realiza as operações de captação, distribuição e coleta dos dados utilizados. O seu algoritmo pode ser visto a seguir:

Processo de Controle

linha = quantidade de linhas da matriz;
coluna = quantidade de colunas da matriz;
pta_A = ponteiro do arquivo da matriz A;
pta_b = ponteiro do arquivo do vetor b;
pta_x^0 = ponteiro do arquivo do vetor solução inicial x^0;
pte_A = ponteiro da estrutura da matriz A;
pte_b = ponteiro da estrutura do vetor b;
pte_x^0 = ponteiro da estrutura do vetor solução inicial x^0;
pte_x = ponteiro da estrutura do vetor solução x;
pte_A = leitura_matricial (pta_A, linha, coluna);
pte_b = leitura_vetorial (pta_b, tamanho);
pte_x^0 = *leitura_vetorial* (*pta_x*^{0},*tamanho*);
envia_matriz_ao_cubo (pte_A, DIAGONAL);
envia_vetor_ao_cubo (pte_b);
envia_vetor_ao_cubo (*pte_x*^{0});
pte_x = recebe_vetor_do_cubo ();

O processo dos nodos está demonstrado a seguir e é dividido em duas partes, uma para recepção e envio dos dados em relação ao processo de controle e outra do método do gradiente conjugado propriamente dito. A modularidade foi empregada para facilitar a depuração do algoritmo.

Processo dos Nodos

pte_A = ponteiro da estrutura da matriz A;
pte_b = ponteiro da estrutura do vetor b;
pte_x^0 = ponteiro da estrutura do vetor solução inicial x^0;
pte_x = ponteiro da estrutura do vetor solução x;
pte_A = *recebe_matriz_do_controle ()*;
pte_b = *recebe_vetor_do_controle (SIMPLE)*;
pte_x^0 = *recebe_vetor_do_controle (SIMPLE)*;
pte_x = *gradiente* (pte_A, pte_b, pte_x^0);
Se pte_x não for nulo
    envia_vetor_ao_controle (pte_x);

Gradiente

pte_A = ponteiro da estrutura da matriz A;
pte_b = ponteiro da estrutura do vetor b;
pte_x^k = ponteiro da estrutura do vetor solução,
valor inicial igual a x^0;
pte_r^k = ponteiro da estrutura do vetor residual;
pte_d^k = ponteiro da estrutura do vetor direção;
pte_Ad^k = ponteiro da estrutura do vetor resultante de $A * d^k$;
α = escalar;
β = escalar;
n^k = numerador de α;
n^{k1} = numerador de β;
d^k = denominador;
ε = resíduo;
flagp = flag de parada;
pte_r^k = *matriz_vetor* (pte_A, pte_x^k);
soma_esc_vet_a_vet $(pte\_r^k, pte\_b, -1.0)$;
pte_d^k = *copia_vetor* (pte_r^k);
n^{k1} = *produto_interno* (pte_r^k, pte_r^k);
Enquanto $n^{k1} > \varepsilon$ and *flagp* = *FALSE*
    $n^k = n^{k1}$;
    pte_Ad^k = *matriz_vetor* (pte_A, pte_d^k);
    d^k = *produto_interno* (pte_d^k, pte_Ad^k);
    $\alpha^k = -n^k / d^k$;
    soma_vet_a_esc_vet $(pte\_x^k, pte\_d^k, -\alpha^k)$;
    soma_vet_a_esc_vet $(pte\_r^k, pte\_Ad^k, \alpha^k)$;
    n^{k1} = *produto_interno* (pte_r^k, pte_r^k);
    Se $n^{k1} \leq \varepsilon$
        flagp = *TRUE*;
    Caso contrário
        $d^k = n^k$;
        $\beta^k = n^{k1} / d^k$;
        soma_esc_vet_a_vet $(pte\_d^k, pte\_r^k, \beta^k)$;
Retorna pte_x^k;