UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Algoritmos Paralelos para Alocação e Gerência de Processadores em Máquinas Multiprocessadoras Hipercúbicas

por

César A. F. De Rose

Dissertação submetida como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, dezembro de 1993

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

De Rose, César A. F.

Algoritmos Paralelos para Alocação e Gerência de Processadores em Máquinas Multiprocessadoras Hipercúbicas / César A. F. De Rose. – Porto Alegre: CPGCC da UFRGS, 1993.

154 p. : il.

Dissertação (mestrado)—Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1993. Orientador: Navaux, Philippe Olivier Alexandre

Dissertação: Arquitetura de Computadores, Processamento Paralelo
Alocação de Processadores, Algoritmos Paralelos, Máquinas Hipercúbicas

AGRADECIMENTOS

Agradeço ao professor Philippe O. A. Navaux, pela orientação deste trabalho e aconselhamentos no encaminhamento de minha vida acadêmica. A Paulo H. L. Fernandes, pelas discussões acerca de avaliação de desempenho de sistemas e pela cooperação do grupo ADMP, em especial do auxiliar de pesquisa Rudi Teodorowitsch, na confecção de ferramentas de simulação contidas neste trabalho. Aos componentes do grupo de processamento paralelo PROCPAR, em especial Gerson Cavalheiro, Ricardo Menna Barreto e Raul Ceretta Nunes, pelo apoio ao desenvolvimento de diversas etapas desta pesquisa. A CAPES, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, pelo auxílio financeiro.

SUMÁRIO

LISTA DE ABREVIATURAS

ASCII	*American Standard Code for Information Interchange* (código padrão americano para intercâmbio de informação)
CSP	*Communicating Sequential Processes* (processos seqüenciais comunicantes)
DAG	*Directed Acyclic Graph* (grafo acíclico direto)
EP	Elemento de Processamento
E/S	Entrada/Saída de dados
FL	*Free List* (estratégia de alocação)
GC	*Gray Codes* (estratégia de alocação)
MFLOPS	*Mega FLoat-point Operations Per Second* (milhão de operações ponto-flutuante por segundo)
MGC	*Multiple Gray Codes* (estratégia de alocação)
MIMD	*Multiple Instruction Flow Multiple Data Flow* (fluxo múltiplo de instruções para fluxo múltiplo de dados)
MIPS	*Mega Instructions Per Second* (milhão de instruções por segundo)
RAM	*Random Access Memory* (memória de acesso aleatório)
SIMD	*Single Instruction Flow Multiple Data Flow* (fluxo único de instruções para fluxo múltiplo de dados)
TC	*Tree Collapsing* (estratégia de alocação)
TRAM	*TRAnsputer Module* (módulo Transputer)
UCP	Unidade Central de Processamento
VLSI	*Very Large Scale Integration* (escala de intergação muito alta).

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Nos últimos anos, máquinas maciçamente paralelas, compostas de centenas de processadores, vem sendo estudadas como uma alternativa para a construção de supercomputadores. Neste novo conceito de processamento de dados, grandes velocidades são alcançadas através da cooperação entre os diversos elementos processadores na resolução de um problema.

Grande parte das máquinas maciçamente paralelas encontradas no mercado utilizam-se da topologia hipercúbica para a interconexão de seus múltiplos processadores, ou podem ser configuradas como tal. Uma alternativa interessante para o compartilhamento da capacidade de processamento destas máquinas é sua utilização como computador agregado a uma rede, servindo a diversos usuários [DUT 91]. Desta forma, a máquina hipercúbica se comporta como um banco de processadores, que permite que cada usuário aloque parte de seus processadores para seu uso pessoal. Isto resulta em um aumento no desempenho da rede ao nível de supercomputadores com um custo relativamente baixo e viabiliza a construção de máquinas hipercúbicas com altas dimensões, evitando que estas sejam sub-utilizadas.

Neste tipo de contexto, cabe ao sistema operacional atender as requisições dos usuários do hipercubo compartilhado de forma eficiente, a fim de evitar uma rápida fragmentação do cubo e de não exceder o tempo máximo de espera de uma determinada aplicação. Os algoritmos de gerência e alocação de processadores são responsáveis pela obtenção e controle de um ou mais processadores da máquina compartilhada para a execução de tarefas de um determinado usuário.

Neste trabalho são propostas versões paralelas dos principais algoritmos de alocação e gerência de processadores encontrados na literatura. Desta forma, se pretende diminuir o tempo de resposta dos algoritmos mais complexos, a ponto de viabilizar a sua utilização em um ambiente de compartilhamento

interativo. Como o algoritmo de alocação exerce a principal função do servidor de processadores, a utilização dos algoritmos mais complexos permite uma melhor utilização dos processadores da máquina compartilhada, resultando em uma melhora de desempenho da máquina paralela como um todo.

A partir dos algoritmos propostos é apresentada a definição de um servidor de processadores para o compartilhamento de uma máquina multiprocessadora hipercúbica em uma rede de estações de trabalho. Algumas funções deste servidor são implementadas por um protótipo denominado Sub-Cube RPC.

Com o objetivo de analisar o comportamento da rede de estações em relação a inclusão de um novo recurso a ser compartilhado, foi desenvolvido, juntamente com o grupo de Avaliação de Desempenho ADMP, um simulador para o ambiente SUN/UNIX. Através desta ferramenta e dos tempos de resposta obtidos pelo protótipo do servidor desenvolvido é possível avaliar o custo que o tráfego gerado pelo servidor adiciona à rede, sendo possível a manipulação de parâmetros da rede e do servidor.

Os resultados obtidos nas versões paralelas implementadas são comparados com o desempenho das versões seqüenciais. Para viabilizar esta comparação, todos os algoritmos sequenciais encontrados na literatura também foram implementados na linguagem "C" no ambiente alvo UNIX e encontram-se em anexo. As versões paralelas foram implementadas utilizando-se recursos da própria rede de estações, através de diretivas *socket*, e também em Transputers na linguagem C paralela. O protótipo do servidor de processadores foi implementado como um servidor RPC para uma rede de estações UNIX também na linguagem "C". A ferramenta de simulação para o funcionamento do servidor foi implementada na linguagem "C" e seu sistema de entrada de dados e visualização utiliza a interface X-Windows.

Com os resultados deste trabalho se pode ter uma boa idéia dos efeitos e das dificuldades encontradas na paralelização dos algoritmos de alocação

e gerência de processadores para máquinas Hipercúbicas. As informações contidas no trabalho auxiliam na melhoria do tempo de resposta dos algoritmos seqüenciais atuais e no desenvolvimento de novos algoritmos com mais recursos e ainda assim viáveis em ambientes interativos, graças a utilização de paralelismo.

O protótipo Sub-Cube RPC demonstra como os algoritmos estudados neste trabalho podem ser aplicados na construção de um servidor de processadores para máquinas multiprocessadas. O protótipo servirá como base para a implementação de um servidor semelhante no CPGCC/UFRGS, que colocará uma placa de Transputers à disposição da rede de estações do grupo de processamento paralelo.

PALAVRAS-CHAVE: Arquitetura de Computadores, Processamento Paralelo, Alocação de Processadores, Algoritmos Paralelos, Máquinas Hipercúbicas.

TITLE: "PARALLEL ALGORITHMS FOR PROCESSOR ALLOCATION IN HYPERCUBES"

ABSTRACT

In the last years massively parallel machines, build with hundreds of processors, are becoming an alternative for the construction of supercomputers. In this new concept of data processing, high performance is achieved by processor cooperation in the resolution of a problem.

A great part of the commercial massively parallel machines utilizes the hypercubic topology to interconnect their multiple processors, or may be configured as hypercubes. A very interesting alternative for sharing the processing power of this machines is their utilization as aggregated computer in a network, serving various users [DUT 91]. In such environment, the hypercube behaves like a processor server, permitting the users to allocate part of its processors for local use. This result in a enhancement in the performance of workstation networks to the level of supercomputers and allow higher dimension hypercubes to be better utilized.

In such environment the operating system is responsible for serving the users of a shared multiprocessor in a efficient way, not allowing a quick fragmentation of the hypercube and observing the maximal waiting time for the applications. The algorithms for processor allocation and management are responsible for obtention and control of one or more processors of the shared machine for the user's task execution.

In this study, parallel versions of the most important algorithms for processor allocation and management in hypercubes found in the literature are proposed. The intention with this paralelization is to achieved a better response

time of the more complex algorithms, making their use possible in a real time sharing environment.

Because the allocation is considered the most important part of the processor server, the utilization of more complex algorithms allows a better utilization of the shared processors, resulting in a performance increase of the parallel machine.

Based on the proposed algorithms, a processor server is defined for sharing a hypercubic multiprocessor in a workstation network. Some functions of this server are implemented in a prototype called Sub-Cube RPC.

To analyze the behavior of the network, in relation to the inclusion of this new shared resource, a simulator for the SUN/UNIX environment has been developed together with the Performance Evaluation Group ADMP. With this tool and with the response times of the developed server prototype, it is possible to evaluate the cost of the additional network traffic generated by the server, with the possibility to change parameters of the server and network.

The results obtained in the implemented parallel versions are compared with the performance of the sequential algorithms. To make this comparison possible all the sequential algorithms found in the literature are also implemented in the "C" language and can be found in annex.

The parallel versions were implemented using network resources, through the socket directive, and also using Transputers in parallel "C". The processor server prototype was implemented as a RPC server for an UNIX network, also in the "C" language. The simulation tool was coded in "C" and the I/O interface use the X-Windows protocol.

The results of this study may give a background about the effects and difficulties found in the paralelization of the allocation algorithms for the hypercubic machines. The information found in this study will help the operating system

designer to obtain a better response time of the sequential algorithms found in the literature and in the development of new and more complex algorithms that will be still practicable in a real time environment due to parallelism utilization.

The Sub-Cube RPC prototype demonstrates how the algorithms studied in this work can be applied in the construction of a processor server for multiprocessors. The prototype is the first step for the implementation of a similar server in the CPGCC/UFRGS that will share a Transputer board in a network of workstations from the parallel processing group.

KEYWORDS: Computer Architecture, Parallel Processing, Processor Allocation, Parallel Algorithms, Hypercubes.

1 INTRODUÇÃO

Com a diminuição no custo dos microprocessadores e com o desenvolvimento de tecnologias para sua interconexão em grande escala, máquinas maciçamente paralelas compostas de centenas de elementos processadores, se tornaram uma alternativa para a construção de supercomputadores.

Neste novo conceito de processamento de dados, grandes velocidades são alcançadas, através da cooperação entre os diversos elementos processadores na resolução de um problema. A forma como estes elementos processadores estão interligados afeta diretamente o desempenho do sistema como um todo, já que muitas vezes a comunicação entre eles se tornará necessária, tanto para a troca de valores intermediários, como para efeito de sincronização [HWA 85].

A interconexão hipercúbica é considerada hoje, uma das formas mais eficientes de interligação neste contexto, devido a sua ótima relação do número de elementos processadores com a distância máxima entre eles, e a sua grande flexibilidade [FEN 81]. Alguns exemplos de de máquinas que empregam esta interconexão são o Cosmic Cube [SEI 85], NCUBE [HAY 86], Mark III e Intel iPSC [PET 85].

Uma alternativa interessante para o compartilhamento da capacidade de processamento de uma máquina hipercúbica é sua utilização como computador agregado a uma rede, servindo a diversos usuários [DUT 91]. Desta forma, a máquina hipercúbica se comporta como um banco de processadores, que permite que cada usuário aloque parte de seus processadores para seu uso pessoal.

Em [TRI 90] é apresentado um ambiente de compartilhamento semelhante que se utiliza de uma malha de Transputers como banco de processadores. A interconexão Hipercúbica porém, é mais flexível, dando ao sistema uma maior

capacidade de reconfiguração e maior facilidade de gerência e alocação dos processadores [DER 93].

A característica de definição recursiva das estruturas hipercúbicas, ou seja, um hipercubo de grau n contém dois cubos de grau $n-1$, facilita este tipo de alocação, permitindo a construção de hipercubos com um grande número de nodos, que serão utilizados por diversos usuários na forma de cubos independentes de menor grau.

Este tipo de compartilhamento aumenta o desempenho de redes de computadores ao nível de supercomputadores com um custo relativamente baixo e viabiliza a construção de máquinas hipercúbicas com altas dimensões, evitando que estas sejam sub-utilizadas.

Os algoritmos de gerência e alocação de processadores são responsáveis pela obtenção e controle de um ou mais processadores da máquina multiprocessadora para a execução de tarefas de um determinado usuário. A precisão e o tempo de resposta destes algoritmos afetam diretamente a taxa de utilização dos processadores da máquina paralela compartilhada e conseqüentemente o seu desempenho como um todo.

Neste trabalho é feita uma revisão dos principais algoritmos de alocação e gerência de processadores para máquinas hipercúbicas encontrados na literatura. São analisadas formas de otimização destes algoritmos através de sua paralelização. Versões paralelas dos algoritmos são propostas e seu desempenho é comparado com os algoritmos seqüênciais.

A partir dos algoritmos propostos é apresentada a definição de um servidor de processadores para o compartilhamento de uma máquina multiprocessadora hipercúbica em uma rede de estações de trabalho. Algumas funções deste servidor são implementadas por um protótipo denominado Sub-Cube RPC.

Com o objetivo de analisar o comportamento da rede de estações, em relação a inclusão de um novo recurso a ser compartilhado, foi desenvolvido, juntamente com o grupo de Avaliação de Desempenho ADMP, um simulador para o ambiente SUN/UNIX. Através desta ferramenta, e dos tempos de resposta obtidos pelo protótipo do servidor desenvolvido, é possível avaliar o custo que o tráfego gerado pelo servidor adiciona à rede, sendo possível a manipulação de parâmetros da rede e do servidor.

Os objetivos principais deste trabalho são: i) implementar os algoritmos seqüenciais e as versões paralelas propostas no ambiente de compartilhamento definido, com o objetivo de avaliar o ganho obtido através da paralelização; ii) definir e iniciar a implementação de um servidor de processadores para o ambiente proposto; iii) fornecer, através das ferramentas desenvolvidas e das conclusões deste trabalho, subsídios para a implementação de uma rede de compartilhamento semelhante, no laboratório do CPGCC/UFRGS [1] com os recursos disponíveis.

A organização deste trabalho, composto por 7 capítulos, é vista a seguir. O capítulo 2 apresenta a definição e as principais características da topologia hipercúbica, definindo conceitos que serão utilizados ao longo do trabalho.

No capítulo 3 é definido o problema da gerência e alocação de processadores em máquinas multiprocessadoras, com o enfoque para máquinas hipercúbicas. Também é feita uma análise da evolução dos algoritmos encontrados na literatura para a resolução deste problema, juntamente com uma descrição dos principais algoritmos seqüenciais encontrados na literatura.

No capítulo 4 são descritos os algoritmos paralelos propostos, e avaliados os algoritmos paralelos até então encontrados na literatura.

No capítulo 5 é feita uma avaliação dos resultados obtidos com a implementação dos algoritmos no ambiente alvo. São fornecidos valores re-

[1] Curso de Pós-Graduação em Ciência da Computação/Universidade Federal do Rio Grande do Sul

ferentes ao desempenho dos algoritmos seqüenciais, dos algoritmos paralelos implementados com recursos da rede de estações, e dos algoritmos paralelos implementados em Transputers.

A aplicação dos algoritmos estudados na confecção de um servidor de processadores é descrita no capítulo 6. O protótipo Sub-Cube RPC é apresentado juntamente com a ferramenta desenvolvida para a simulação dos efeitos do servidor proposto na rede.

As conclusões do trabalho podem ser encontradas no capítulo 7, seguidas da bibliografia. Em anexo encontram-se as listagens das implementações seqüenciais e paralelas, do protótipo do servidor de processadores, e um *dump* de saída do simulador da rede.

2 A MÁQUINA HIPERCÚBICA: DEFINIÇÕES E CARACTERÍSTICAS TOPOLÓGICAS

Neste capítulo são apresentados alguns conceitos básicos e características da topologia hipercúbica que serão utilizadas ao longo deste trabalho.

Um estudo detalhado sobre máquinas multiprocessadoras hipercúbicas, abrangendo tanto o hardware quanto o software e contendo um estudo de casos pode ser encontrado em [DER 92].

2.1 Definições

Um computador hipercúbico, ou $n-$cúbico, é uma máquina multiprocessadora caracterizada pela presença de $N = 2^n$ processadores idênticos, interconectados como um cubo binário n-dimensional. Cada processador P_i forma um nodo (vértice) do cubo e pode ser visto como um computador independente, com sua própria UCP (unidade central de processamento) e memória local. P_i tem canais de comunicação direta com outros n processadores, seus vizinhos, que se localizam na extremidade oposta das arestas do cubo que pertencem ao vértice onde está situado. A enumeração destes nodos é feita com 2^n dígitos binários distintos, que representam o endereço de cada nodo. A distribuição dos endereços é feita de tal forma que todos os n vizinhos de um nodo tenham endereços que sejam diferentes do endereço deste nodo em exatamente uma posição de bit [HAY 86]. A figura 2.1 ilustra a topologia hipercúbica para um n até 4. Este valor n representa a dimensão, ou grau de um hipercubo. Convém ressaltar que um hipercubo zero-dimensional é um computador SISD convencional.

Um subcubo é um subconjunto de processadores do hipercubo que continuam respeitando a topologia hipercúbica, e desta forma mantém todas as

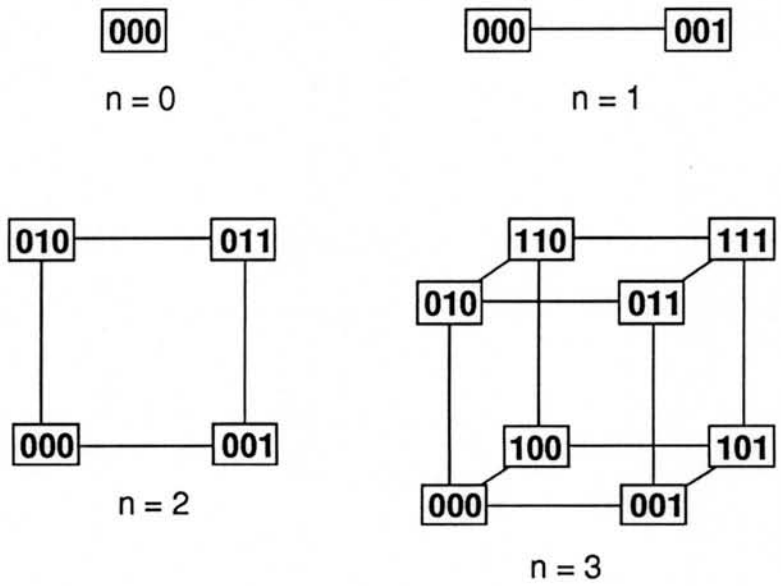


Figura 2.1: Hipercubo n-dimensional para um n = 0, 1, 2 e 3

suas características e propriedades. No hipercubo de grau 3 da figura 2.1 pode ser identificado, por exemplo, o subcubo de grau 2 formado pelos nodos (000) (001) (011) (010).

2.1.1 Lei de formação

Um hipercubo n-dimensional Q_n , para um $n > 2$, pode ser definido recursivamente em termos da operação de produto de grafos X, como está descrito em [HAR 88], onde $K_2 = Q_1$ é um grafo completo de 2 nodos:

$$Q_n = K_2 X Q_{n-1}$$

Como podemos ver na figura 2.1, Q_n é composto de duas cópias de Q_{n-1}. Cada nodo P_{0x} em uma das cópias de Q_{n-1} é vizinho de um nodo P_{1x} na outra cópia.

Uma outra forma de expressarmos a lei de formação de um hipercubo é através de permutação [HWA 85], onde N representa o número total de elementos:

$$P_i = \prod_{j=0}^{N-1} (jC_i(j))$$

Se aplicarmos esta fórmula para um hipercubo de dimensão 3 obteremos como resultado as permutações P_0 = (0;1) (2;3) (4;5) (6;7), P_1 = (0;2) (1;3) (2;4) (3;5) e P_2 = (0;4) (1;5) (2;6) (3;7), que representam todas as conexões necessárias entre os 8 ($2^n = 2^3 = N$) elementos processadores (EP's) para compor o cubo.

2.1.2 Formas de interconexão

Por forma de interconexão entende-se a forma de ligação física pela qual é implementada uma determinada topologia. Ou seja, a forma pela qual, através de placas e conectores, são interligados os processadores para respeitar um determinado padrão topológico.

No caso da topologia hipercúbica a interconexão dos processadores pode ser feita de duas formas [ABR 89]: direta e indireta.

direta a forma de interconexão direta é caracterizada pela ligação dos nodos (elementos processadores) através de conexões ponto-a-ponto. A rede de interconexão é estática e não pode ser reconfigurada durante a operação.

indireta na forma de interconexão indireta os processadores são conectados através de uma rede de chaves multinível. Esta rede de interconexão é dinâmica e pode ser reconfigurada durante a operação.

Na figura 2.2 podem ser vistos exemplos de interconexões diretas (a)(b)(c) e indiretas (d).

2.1.3 Formas de representação

A topologia hipercúbica pode ser representada graficamente de diversas formas. Cada uma delas tem por objetivo facilitar a visualização das propriedades deste tipo de interconexão sobre certas circunstâncias, como grande número de nodos ou sua interconexão indireta. A figura 2.2 apresenta as formas de representação mais encontradas na literatura.

A representação espacial figura 2.2(a) é utilizada para visualização dos diferentes planos de um hipercubo introduzindo a noção de volume. Este tipo de representação é geralmente utilizada para interconexões diretas pequenas, com dimensão em torno de 4. Representações espaciais de hipercubos com dimensão maior do que 4 são de difícil concepção e a identificação de um nodo e seus vizinhos já se torna complexa.

As representações em anel figura 2.2(b) e plana figura 2.2(c) permitem uma melhor visualização de um nodo e seus vizinhos em hipercubos de maior dimensão, e ressaltam o aspecto de simetria encontrado nesta topologia.

A representação multinível figura 2.2(d) é utilizada para interconexões hipercúbicas indiretas, que se utilizam de diversos níveis de chaves para efetuar a ligação de dois nodos. Esta forma de representação ressalta a disposição destas chaves e a forma que as mesmas estão ligadas aos nodos.

2.2 Características topológicas

A importância da arquitetura hipercúbica para as máquinas multiprocessadoras fracamente acopladas resulta de suas características topológicas extremamente favoráveis para este tipo de conexão. A seguir serão apresentadas algumas destas características, juntamente com comparações à outras topologias.

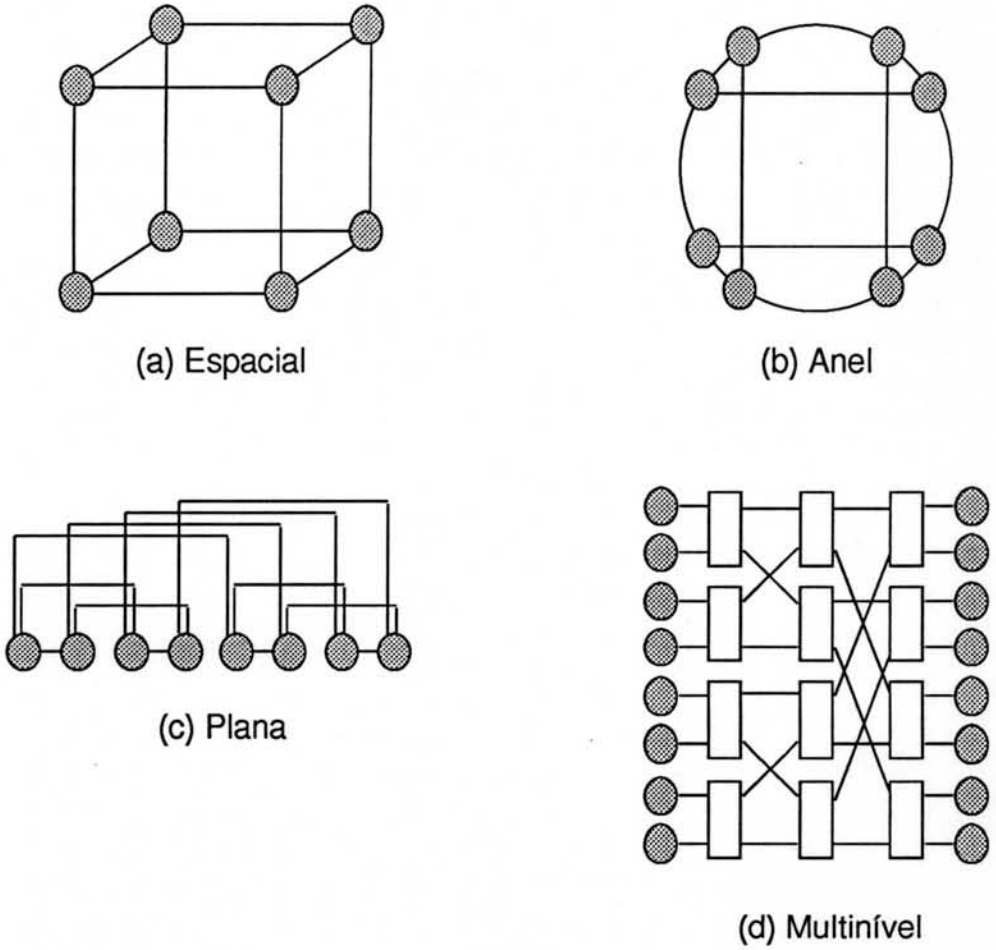


Figura 2.2: Diferentes formas de representação de um hipercubo de dimensão 3

2.2.1 Relação do número de nodos com o número de conexões

A relação entre o número de nodos de uma topologia e o número de conexões necessárias para sua interligação é um valor importante no que tange ao custo de sua implementação, principalmente quando o número de nodos se eleva. A expressão

$$(N/2) * n$$

calcula o número de ligações ponto-a-ponto necessárias para um hipercubo de $N = 2^n$ nodos.

2.2.2 Grau de um nodo

O grau de um nodo, também chamado de *fanout*, é o valor que expressa o número de vizinhos de um nodo. Este valor é muito importante na fase de implementação de uma topologia, pois como todas as ligações de um hipercubo são ponto-a-ponto, cada nodo terá um número de portas físicas de comunicação igual ao número de vizinhos.

O valor do grau de um nodo hipercúbico é igual a dimensão n do cubo em questão e pode ser expresso em função do número total de nodos N através da fórmula:

$$n = log_2 N$$

Na figura 2.3 podemos observar que o grau do nodo (000) é igual a 3, por se tratar da dimensão do hipercubo em questão. Isto significa que do nodo (000) partem três ligações ponto-a-ponto, uma para cada um de seus vizinhos (100) (010) (001). Por se tratar de uma estrutura regular, todos o nodos de um hipercubo possuem o mesmo grau.

2.2.3 Distância máxima entre nodos

A distância máxima entre nodos de uma topologia é expressa pelo número inteiro que representa a quantidade de ligações ponto-a-ponto necessárias para se efetuar a comunicação entre os dois nodos mais distantes da rede. Na teoria de grafos este valor também é chamado de diâmetro do grafo.

O hipercubo tem uma distância máxima entre nodos igual ao seu grau, ou seja

$$n = log_2 N$$

o que significa que a distância máxima entre dois nodos é relativamente curta, comparada com o tamanho da rede e com o número total de nodos. Este valor é maior que o diâmetro de um grafo totalmente conectado KN, que é igual a um, mas é alcançado com um grau de nodo de apenas $log_2 N$, em comparação com o grau $N-1$ de KN.

Em outras arquiteturas com um grau de nodo pequeno, como árvores, malhas e barramentos, encontramos uma distância máxima entre nodos maior que o hipercubo ($N^{\frac{1}{2}}$ nas malhas), ou a presença de algum canal de comunicação que se torna rapidamente o gargalo do sistema, como por exemplo, um barramento ou o canal de comunicação do nodo raiz de uma árvore [HAY 86].

2.2.4 Caminhos paralelos entre nodos

Considerando que os nodos processadores estão distribuídos pelos vértices do cubo e as conexões entre eles sobrepõem suas arestas, podemos constatar que existem caminhos paralelos de mesma distância entre dois nodos quaisquer, se eles não forem vizinhos. Estes caminhos paralelos (também chamados de *disjoint paths*) podem ser obtidos através de uma diferente ordem de variação sobre os eixos do cubo durante a operação de roteamento. Como cada bit do endereço de um nodo representa sua posição em um determinado eixo do cubo, podemos obter os diferentes caminhos paralelos entre dois nodos se alterarmos estes bits em ordem de seqüência diferentes.

Desta forma, se o endereço do nodo origem para o nodo destino difere por d bits, existem d caminhos paralelos entre eles de distância d [DOW 88]. Um exemplo desta propriedade está ilustrado na figura 2.3, onde são destacados os caminhos paralelos entre o nodo 0 e o nodo 7 para um hipercubo de grau 3.

Esta característica particular da estrutura de um cubo pode ser muito bem aproveitada quando se deseja balancear a carga de utilização dos caminhos.

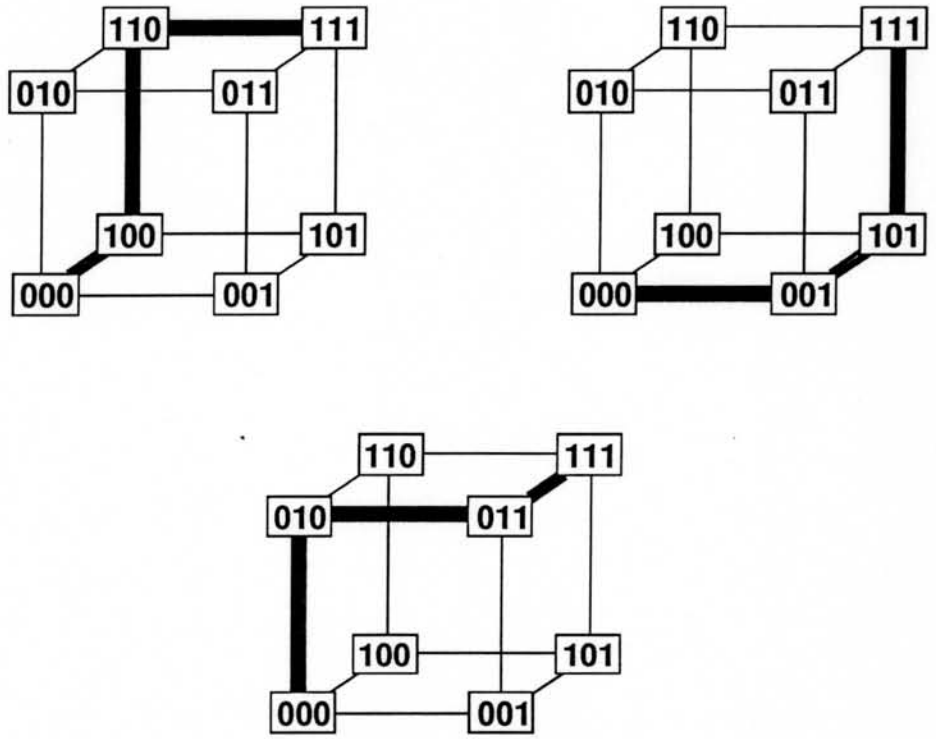


Figura 2.3: Diferentes caminhos paralelos em um hipercubo de grau 3

Um algoritmo de roteamento poderia avaliar, no caso de existirem caminhos paralelos, qual destes caminhos está menos sobrecarregado, e decidir pela utilização do mesmo em tempo de execução do algoritmo de roteamento. Este procedimento resultaria em um algoritmo de roteamento dinâmico, com balanceamento de carga entre os caminhos e tolerante a falhas, pois caminhos constatados defeituosos seriam evitados.

Outra utilização interessante destes caminhos paralelos seria no caso de se desejar uma maior taxa de transmissão entre dois nodos que não fossem vizinhos. Neste caso se poderia utilizar todos os caminhos paralelos disponíveis para uma transmissão simultânea [BHU 84].

2.2.5 Modo de operação

Um hipercubo pode operar tanto no modo SIMD (Single Instruction Multiple Data), como MIMD (Multiple Instruction Multiple Data) [DER 92]. No caso da operação SIMD, à máquina hospedeira é atribuído o papel de controlar a distribuição dos dados e o fornecimento das instruções para os nodos do cubo. Após o fornecimento das instruções, idênticas para todos os elementos processadores, o hospedeiro distribui os dados a serem operados entre os nodos, da melhor maneira possível, com o objetivo de diminuir ao máximo a comunicação entre os nodos. Após o término da operação por parte de cada nodo, todos retornam seus valores para a máquina hospedeira, que compõe e apresenta o resultado final.

No modo de operação MIMD, cada nodo da máquina hipercúbica executa uma parte do problema de forma totalmente independente. O fato de um nodo do hipercubo poder ser visto como uma máquina completa, UCP e memória local, viabiliza este modo de operação. Instruções e dados são divididos entre os processadores, como se estes fossem máquinas diferentes executando programas diferentes. Quanto melhor for o balanceamento de carga entre os nodos disponíveis, melhor será o desempenho deste modo de operação.

2.2.6 Metodologia de comutação

A metodologia de comutação entre os nodos de um hipercubo é a troca de mensagens. Isto significa que os nodos trocam informações através de pacotes de dados que navegam nas linhas ponto-a-ponto do cubo com o endereço ao qual são destinados. Cabe ao sistema operacional o roteamento destes pacotes através dos nodos até o seu endereço destino.

2.2.7 Particionamento do hipercubo

Uma característica que destaca a arquitetura hipercúbica dos outros modelos de interconexão de processadores é sua grande flexibilidade de utilização. Grande parte desta flexibilidade de utilização se deve ao fato deste tipo de arquitetura permitir a sobreposição de outras topologias e o particionamento de seus recursos.

Por particionamento entende-se a divisão lógica do cubo em partes independentes de tamanho variável. Cada uma destas partes pode executar uma tarefa específica, sendo a comunicação entre os seus processadores e o seu modo de operação totalmente independentes do restante do cubo.

Por se tratar de uma estrutura regular e poder ser definida recursivamente é possível obter, na operação de particionamento de um hipercubo, partes independentes de menor grau, que mantém todas as propriedades do cubo original. Um exemplo deste tipo de particionamento seria a divisão de um hipercubo K_4, de grau quatro, em dois subcubos de grau três, Ki_3 e Kj_3, ou até mesmo em quatro subcubos de grau 2, Ki_2, Kj_2, Kl_2 e Km_2.

Nos próximos ítens serão apresentadas algumas variações sobre a operação de particionamento.

2.2.7.1 Particionamento a nível de modo de operação

Como já foi visto na seção 2.2.5, a arquitetura hipercúbica permite o modo de operação SIMD e MIMD. A operação de particionamento pode ser utilizada neste caso para dividir o cubo em grupos de processadores com diferentes modos de operação. Como cada subcubo resultante da operação de particionamento é independente dos outros, não ocorreriam problemas de conflitos, sendo o controle efetuado pelo sistema operacional.

Um exemplo da utilização deste tipo de operação pode ser visto na figura 2.4, onde um cubo de grau 3 é particionado em dois subcubos de grau 2. Um destes subcubos opera no modo MIMD e o outro no modo SIMD, com o hospedeiro controlando a operação de seus nodos.

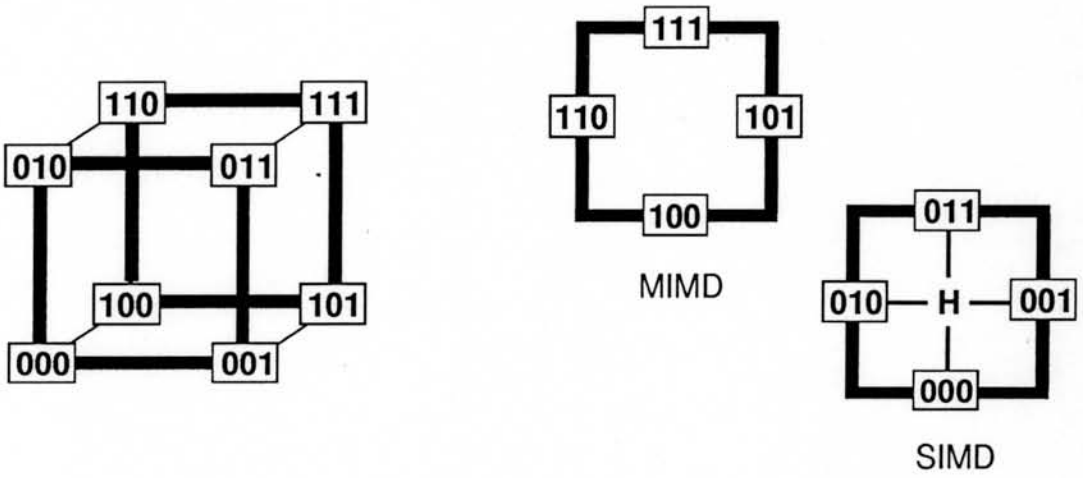


Figura 2.4: Particionamento de um hipercubo em dois subcubos de menor grau com diferentes modos de operação

2.2.7.2 Particionamento a nível de configuração

Como veremos com mais detalhes no final deste capítulo, o hipercubo possui uma grande facilidade em abrigar em sua estrutura original diversas outras topologias para a interconexão de processadores. Esta característica pode ser aliada à operação de particionamento para a obtenção de subcubos que se comportem como se seus processadores estivessem interligados, respeitando a uma outra forma de interconexão.

A figura 2.5 apresenta um exemplo deste tipo de operação de particionamento, com um hipercubo de grau 3 sendo particionado em duas estruturas menores, cada uma respeitando um determinado tipo de interconexão. No exemplo as interconexões resultantes foram uma árvore com três nodos (111), (110) e (101), e uma estrutura linear composta de quatro nodos (001), (000), (010) e (011). O nodo (100) ainda se encontra disponível após a operação de particionamento e

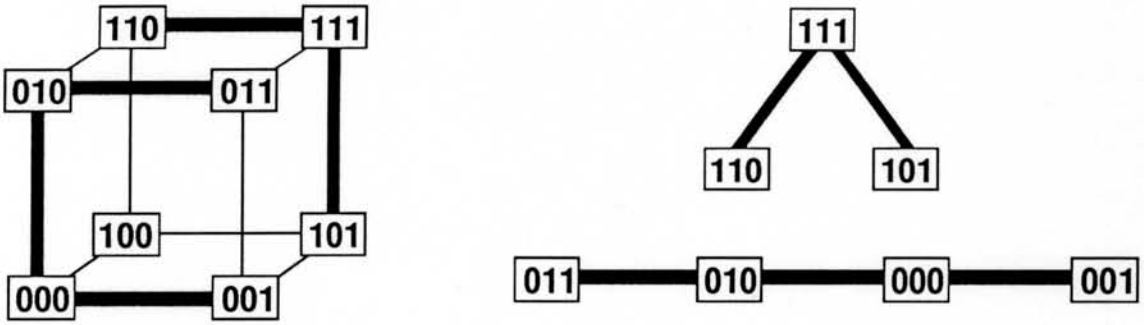


Figura 2.5: Particionamento de um hipercubo em subcubos com diferentes configurações topológicas

pode, por exemplo, ser alocado para substituir algum nodo no caso de ocorrerem falhas em qualquer um dos subcubos.

2.2.7.3 Particionamento de forma híbrida

Esta forma de utilização da operação de particionamento do hipercubo aproveita as duas possibilidades apresentadas anteriormente. Desta forma os subcubos resultantes da operação de particionamento podem operar em diferentes modos e se comportarem como outras formas de interconexão.

Uma aplicação deste tipo de particionamento seria a utilização de uma máquina hipercúbica agregada como banco de processadores auxiliares. Os usuários da máquina hospedeira poderiam então alocar subcubos deste banco, indicando o número de nodos, modo de operação e forma de interconexão que melhor se adaptassem a suas aplicações.

2.2.8 Sobreposição de topologias no cubo

A operação de sobreposição de outras topologias pode ser definida como um mapeamento de uma forma de interconexão qualquer sobre as conexões já disponíveis na arquitetura original. Quanto melhor for a capacidade de

sobreposição de outras topologias por parte de uma arquitetura, menor serão as distâncias entre os nodos da interconexão que se deseja sobrepor. A sobreposição de uma topologia totalmente conectada em um hipercubo com o mesmo número de nodos é possível, porém a distância entre os nodos ficará maior, pois serão utilizados caminhos alternativos que passam por vários nodos, para substituir os caminhos diretos não existentes. Já a sobreposição de um anel em um hipercubo com o mesmo número de nodos é feita sem prejuízo nenhum na distância entre seus nodos, pois as conexões necessárias são mapeadas sobre conexões diretas já existentes no hipercubo (seção 2.2.8.4).

A característica de permitir um bom mapeamento de outras interconexões sobre a sua própria estrutura é muito importante, pois permite um bom desempenho em um maior número de problemas. Isto porque os algoritmos paralelos para a solução eficiente de diversos problemas são dependentes da interconexão dos processadores da máquina onde são implementados. Quanto mais interconexões puderem ser mapeadas de forma eficaz em uma máquina, mais algoritmos poderão ser implementados eficientemente, e mais genérica esta máquina se tornará [SAA 88].

O hipercubo pode ser considerado uma máquina genérica devido a sua grande flexibilidade na sobreposição de outras topologias. Esta flexibilidade será comprovada nos próximos ítens com uma análise da capacidade de mapeamento do hipercubo em relação aos modelos de interconexão de processadores mais utilizados.

Estas operações de mapeamento serão utilizadas no capítulo 6, na definição da camada de mapeamento de topologias do servidor de processadores.

2.2.8.1 Conceito de expansão e dilatação

Para avaliar a qualidade do resultado obtido na operação de mapeamento de uma topologia sobre uma determinada arquitetura que se quer analisar, são encontrados com frequência na literatura os termos expansão e dilatação.

O termo expansão é definido como a razão entre o número total de nodos do hipercubo e o número de nodos do menor subcubo necessário para mapear a topologia desejada. Dilatação é definida como a distância máxima entre nodos adjacentes da topologia desejada quando mapeada no hipercubo. Ou seja, expansão mede o desperdício de nodos ou processadores no resultado da operação de mapeamento, enquanto dilatação mede o gasto em comunicação da sobreposição resultante [CHA 88].

No caso do mapeamento de uma interconexão de processadores totalmente conectada C_p sobre um hipercubo H_n, o resultado poderia ser expansão igual a 1 e dilatação igual a 3. Neste caso seria utilizado um hipercubo com o mesmo número de nodos que a topologia desejada ($2^n = p$) e distância máxima entre nodos adjacentes, que no caso da topologia totalmente conectada é o diâmetro do grafo; seria 3 pois é o diâmetro do grafo do hipercubo.

A relação expansão x dilatação não é sempre única. Em alguns casos é possível se obter uma queda no valor da dilatação com um aumento da arquitetura hospedeira e um consequente aumento do valor da expansão.

2.2.8.2 Códigos de gray

O conceito de circuito Hamiltoniano é extensamente utilizado na operação de mapeamento. Este tipo de circuito é representado por uma sequência de números binários de n-bits, de tal forma que dois números sucessivos tenham apenas um bit de diferença e todos os números binários com n bits sejam repre-

sentados. Seqüências binárias com estas propriedades são denominadas códigos de gray e vem sendo extensamente estudadas na teoria dos códigos [REI 77].

Existem diferentes formas de se gerar códigos de gray. O melhor método conhecido resulta no chamado código de gray refletido e é gerado da seguinte forma:

1. O algoritmo se inicia com a sequência de dois números binários de um bit, 0 e 1. G_1 é um código de gray de um bit.

$$G_1 = \{ 0 , 1 \}$$

2. Para construirmos uma sequência de dois bits, inserimos um 0 antes de cada número. Depois utilizaremos a sequência na ordem inversa e inserimos um 1 na frente de cada número.

$$G_2 = \{ 00 , 01 , 11 , 10 \}$$

3. O mesmo processo utilizado no passo acima é repetido para a construção de uma sequência de gray com 3 bits, e assim sucessivamente.

$$G_3 = \{ 000 , 001 , 011 , 010 , 110 , 111 , 101 , 100 \}$$

De forma genérica, denominando G_{ir} a sequência obtida revertendo a ordem da sequência G_i, e $0G_i$ ($1G_i$) a sequência obtida pela colocação de um 0 (1) na frente de cada elemento de G_i, então códigos de gray podem ser gerados recursivamente através da expressão:

$$G_{n+1} = 0G_n , 1G_{nr}$$

2.2.8.3 Sobreposição da topologia array linear

A topologia array linear pode ser sobreposta em um hipercubo sem maiores dificuldades com a obtenção do caso ótimo como resultado, ou seja,

expansão e dilatação igual a um. Os nodos do array linear $P_0, P_1, \ldots, P_i$ de tamanho arbitrário $i < 2^{n-1}$ podem ser mapeados facilmente nos nodos do hipercubo se utilizando de uma sequência de códigos de gray.

Dado um array linear de tamanho arbitrário i, o menor hipercubo no qual ele pode ser mapeado é o de dimensão $n = [log_2(i+1)]$.

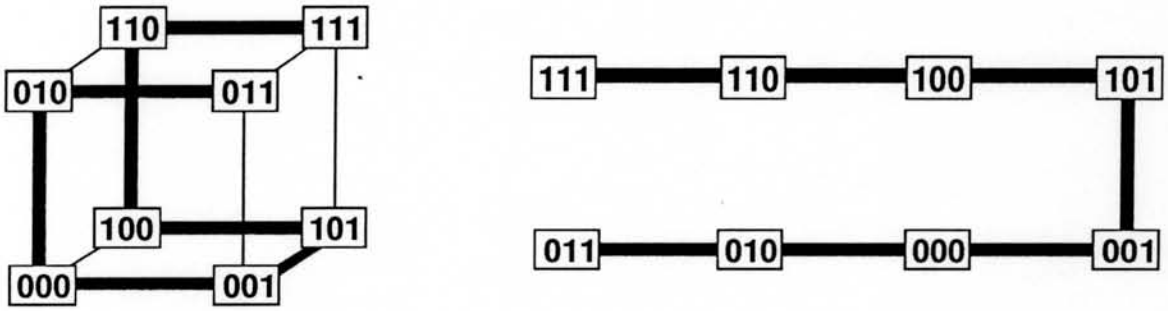


Figura 2.6: Sobreposição de uma topologia linear em um hipercubo de grau 3

A figura 2.6 apresenta a sobreposição de um array linear de 8 elementos sobre um hipercubo de grau 3. Como se pode notar, todos os nodos são aproveitados e a distância máxima entre os nodos adjacentes é mantida em um.

2.2.8.4 Sobreposição da topologia anel

Os códigos de gray também permitem que se efetuem mapeamentos de anéis, cujo número de nodos são potências de dois, em hipercubos. É importante observar, porém, que o mapeamento só é possível se o número de nodos i do anel que se deseja mapear for par, pois no hipercubo não encontramos ciclos ímpares [HAR 88]. O problema então se resume em encontrar um ciclo de tamanho i no hipercubo com $4 \leq i \leq 2^n$ e i sendo ímpar.

Seja $m = (i-2)/2$ representado por $G_{n-1}(m)$ o código de gray parcial com $(n-1)$ bits dos primeiros m elementos de G_{n-1}. Então o ciclo desejado pode ser mapeado nos nodos $\{0G_{n-1}(m), 1G_{n-1}(m)R\}$.

Assim concluímos que um anel com tamanho arbitrário i pode ser mapeado em um hipercubo, se i for par e $4 < i < 2^n$. A extensão e dilatação no caso do número de nodos do anel ser potência de dois fica igual a um.

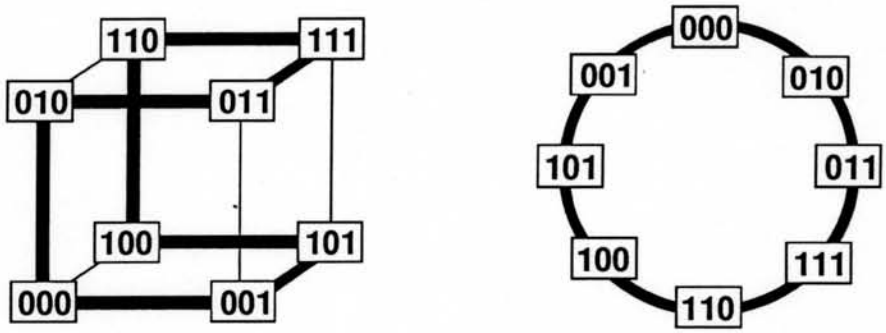


Figura 2.7: Sobreposição de uma topologia anel em um hipercubo de grau 3

A figura 2.7 apresenta o resultado do mapeamento de um anel de 8 nodos em um hipercubo de grau 3. Como o número de nodos do anel é potência de dois, não sobraram nodos após a operação de sobreposição e a distância máxima entre nodos adjacentes se manteve em um.

2.2.8.5 Sobreposição da topologia malha

A topologia malha (*mesh*), também conhecida como grade ou array n-dimensional, pode igualmente ser mapeada com facilidade no hipercubo. A figura 2.8 apresenta o mapeamento de uma grade 2x4 em um hipercubo de grau 3. O resultado desta operação de sobreposição obteve extensão e dilatação um [SAA 88].

A solução aplicada no problema do mapeamento de anéis no hipercubo com os códigos de gray pode ser estendida para as várias dimensões de uma grade.

Consideremos uma malha $m1xm2$ ... x md no espaço d-dimensional R^d com o tamanho em cada direção sendo potência de 2, isto é, $mi = 2^{pi}$. Sendo $n = p_1 + p_2 ... + p_d$, o problema consiste em mapear este array d-dimensional em

um hipercubo H^n. Observa-se que o número de nodos é suficiente para uma correspondência 1 : 1.

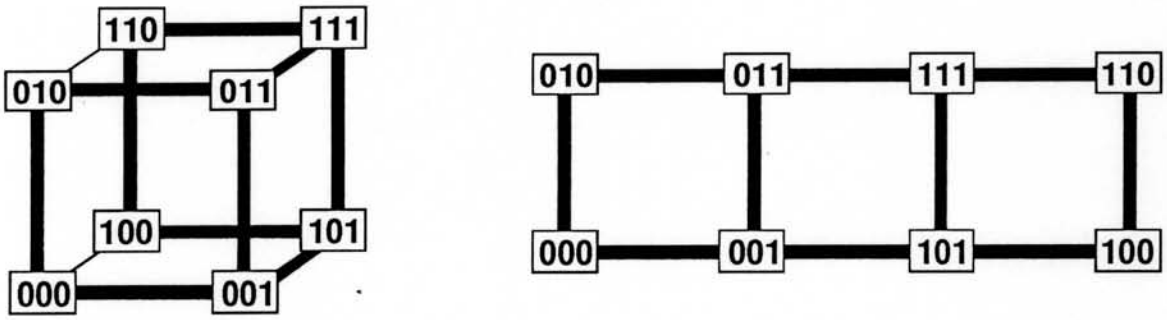


Figura 2.8: Sobreposição de uma topologia malha em um hipercubo de grau 3

A solução do problema de correspondência entre nodos do hipercubo com os nodos da grade pode ser melhor ilustrada com um exemplo. A figura 2.9 apresenta uma malha de $4x4$, com $d = 2$, $p_1 = 2$, $p_2 = 2$ e $n = p_1 + p_2 = 4$. Um número binário A, que representa o endereço dos nodos do 4-cubo onde queremos mapear esta malha, pode ser dividido em duas partes: os primeiros 2 bits e os últimos 2 bits, na forma

$$A = b_1 b_2 c_1 c_2$$

onde os bits b_i e c_j são 0 ou 1. Se escolhermos um código de gray de dois bits para cada uma das direções, o ponto (x_i, y_j) da malha será mapeado no nodo $b_1 b_2 c_1 c_2$ do cubo onde $b_1 b_2$ é um código gray para x_i e $c_1 c_2$ o código gray para y_i.

Generalizando para dimensões maiores, pode-se dizer que qualquer malha $m_1 x m_2$ ... x m_d de dimensão d com $m_i = 2^{pi}$ pode ser mapeada em um n-cubo com $n = p_1 + p_2$ ... $+ p_d$. A numeração dos pontos da grade tem que respeitar uma sequência de gray em todas as direções.

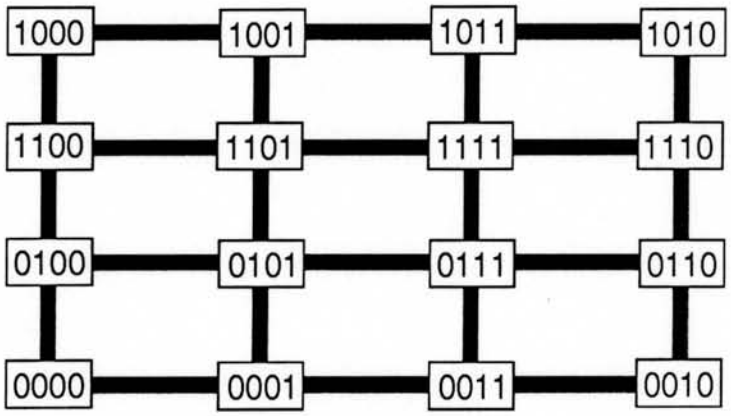


Figura 2.9: Sobreposição de uma topologia malha em um hipercubo de grau 4

2.2.8.6 Sobreposição da topologia árvore binária

Apesar de ser possível mapear uma árvore binária de dois níveis em um hipercubo de grau 2, esta afirmação não é mais verdadeira se aumentarmos o grau do hipercubo e o número de níveis da árvore. Para um $n > 3$ não é possível mapear uma árvore binária de n níveis em um hipercubo de grau n. Nestes casos, para mapear uma árvore binária de n níveis é necessário um hipercubo de grau $n + 1$. A extensão neste caso aumenta, pois sobrarão nodos que não serão utilizados, mas a dilatação se mantém em um. Na verdade, até duas árvores binárias de nível n podem ser mapeadas em um hipercubo de grau $n + 1$. O problema se encontra na ligação destas duas árvores a uma mesma raiz.

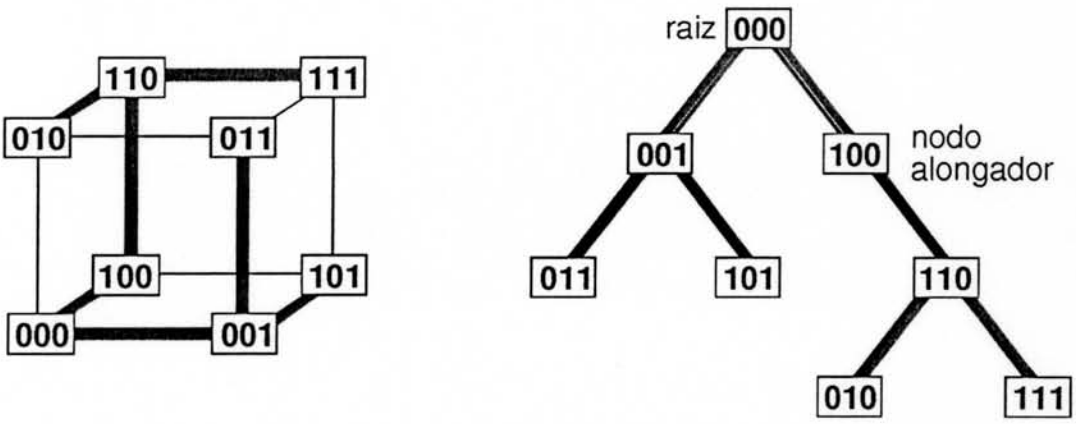


Figura 2.10: Sobreposição de uma topologia árvore binária em um hipercubo de grau 3. Na figura é destacada a sub-utilização de um dos nodos para possibilitar a conexão de um dos ramos

Este problema pode ser contornado se for acrescentado um nodo de alongamento para efetuar a conexão de uma destas árvores à raiz. Desta forma podemos mapear uma árvore de nível n em um hipercubo de grau n. A extensão fica em um, porém a dilatação aumenta para dois, pois existe um caminho entre a raiz e uma das sub-árvores que passa por um nodo intermediário, com a função apenas de roteamento [DES 86] (figura 2.10).

3 O PROBLEMA DA ALOCAÇÃO E GERÊNCIA DE PROCESSADORES

Neste capítulo será apresentado o procedimento de alocação de processadores em máquinas multiprocessadoras, e detalhada a fase específica deste procedimento na qual este trabalho se concentra. Após um histórico sobre a evolução dos algoritmos até hoje propostos para a solução deste problema, segue uma descrição dos principais algoritmos seqüenciais citados na literatura. No final deste capítulo pode ser encontrado um resumo das características dos algoritmos descritos.

3.1 Alocação de processadores

O processo de alocação de processadores em máquinas multiprocessadoras pode ser dividido em três etapas:

1. Geração do Grafo de Tarefas;
2. Mapeamento;
3. Obtenção dos Processadores.

As etapas acima representam todo o caminho que a aplicação paralela do usuário tem que seguir até poder ser executada nos processadores da máquina alvo. A figura 3.1 apresenta uma representação gráfica deste procedimento.

A primeira etapa recebe como entrada um conjunto de processos que compõem a aplicação paralela e gera um grafo de tarefas [ERC 90]. Neste grafo estão contidas todas as dependências entre as diferentes tarefas, que são expressas na aplicação através de troca de mensagens. Este grafo também contém

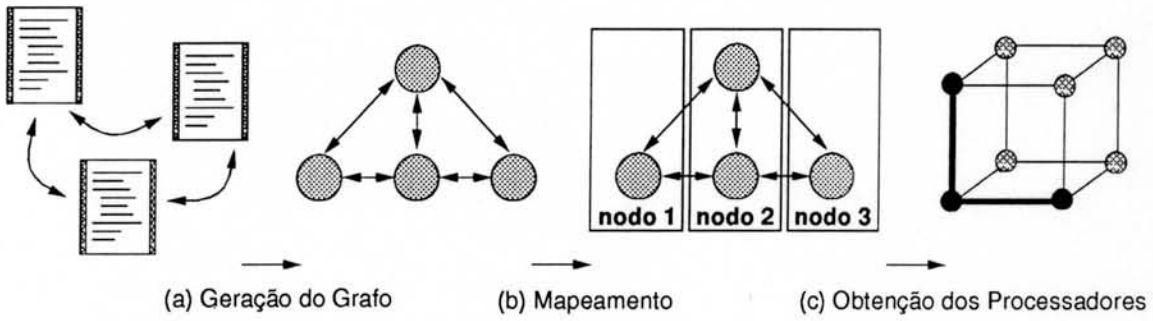


Figura 3.1: Procedimento de alocação de processadores

informações sobre as necessidades específicas de cada tarefa e uma avaliação de sua carga de trabalho.

O grafo de tarefas sofre, na segunda etapa, diversas alterações, baseado em informações sobre as características do ambiente alvo e em heurísticas para a obtenção de melhor desempenho. O grafo é transformado em um grafo de processadores que possa ser facilmente mapeado no ambiente alvo. Tarefas são acumuladas em um mesmo nodo para a economia de canais de comunicação, aumento da granulosidade das operações em um determinado nodo, e balanceamento da carga da máquina alvo. Desta forma se procura minimizar tanto o custo das comunicações entre as tarefas como o tempo total de execução. É nesta fase que é especificado o número de processadores que serão necessários para a execução da aplicação e a forma que devem estar conectados [RAM 88] [RAM 90].

A terceira etapa é responsável pela obtenção do número de processadores necessários na máquina paralela e pela garantia de uso exclusivo por parte da aplicação até a sua liberação. O grupo de processadores reservados deve respeitar a topologia escolhida na segunda fase do procedimento com a menor expansão e dilatação possíveis.

Os sistemas operacionais fornecidos com as máquinas hipercúbicas encontradas no mercado são ainda muito precários e não implementam todas as etapas do procedimento acima. Toda a parte de dimensionamento das tarefas, análise das comunicações entre elas e mapeamento em uma topologia disponível

no hipercubo, tem que ser feita manualmente pelo usuário (etapas 1 e 2). Apenas a terceira etapa é implementada através de ferramentas para a alocação e reserva de um subcubo para uma determinada aplicação. Devido a grande influência destas ferramentas na taxa de utilização e no desempenho da máquina compartilhada, se investe muito em pesquisas nesta área.

Outra característica importante desta terceira etapa da alocação de processadores, é que esta, ao contrário das outras duas, é necessária sempre que for executada a aplicação. Sendo assim, o tempo gasto nesta operação é somado ao tempo de execução da aplicação no cálculo do tempo total de execução da aplicação paralela. Uma redução no tempo de execução da terceira etapa leva conseqüentemente a uma redução no tempo total de execução da aplicação.

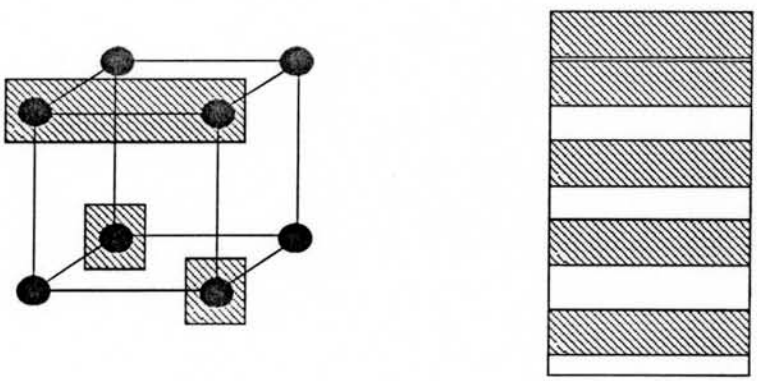
Este trabalho se concentra no estudo e na otimização de ferramentas de suporte para a terceira etapa do procedimento descrito acima. Sempre que, a partir de agora, for utilizado o termo alocação de processadores, estará sendo referida à etapa de obtenção dos mesmos. Esta é uma tendência seguida por diversos autores, como pode ser constatado na bibliografia deste trabalho [ALD 89] [CHE 87] [CHU 90] [KIM 89].

Existem duas modalidades de alocação de processadores: dinâmica (*on-line*) e estática (*off-line*). Na alocação estática o sistema operacional coleta um número suficiente de pedidos antes de iniciar a alocação propriamente dita. Desta forma é possível uma melhor compactação dos nodos alocados e conseqüentemente uma menor fragmentação do hipercubo. Porém, existe uma demora maior na resposta dos pedidos de alocação, que muitas vezes não pode ser suportada em sistemas de tempo real. Na alocação dinâmica os pedidos são aceitos ou negados imediatamente após a sua chegada, sem se levar em consideração os próximos pedidos. O tempo de resposta é melhor, mas o controle da fragmentação do cubo é mais complexo [DUT 91].

Todos os algoritmos estudados e propostos neste trabalho se ocupam com o problema da alocação dinâmica. Isto resulta do fato de que estes algoritmos serão utilizados para gerenciar os processadores de uma máquina multiprocessadora hipercúbica ligada a uma rede de estações de trabalho que compõem um ambiente de compartilhamento dinâmico.

A etapa de obtenção dos processadores parece, a primeira vista, a mais simples das três etapas relacionadas acima, chegando a parecer um problema trivial. Isto porém, não é bem verdade. Com a utilização de máquinas multiprocessadoras em ambientes multiusuário interativos, cresce a necessidade de um algoritmo de alocação e gerência de processadores que racionalize ao máximo a utilização dos processadores compartilhados. Também é fundamental uma alta capacidade de reconhecimento de subcubos para um melhor aproveitamento dos recursos compartilhados.

O problema da gerência de alocação de subcubos pode ser melhor compreendido se for feita uma analogia ao problema de alocação de memória (figura 3.2 e tabela 3.1). Inicialmente, com a memória totalmente livre, a alocação de segmentos de tamanhos diferentes é simples, sendo feita de forma seqüencial. Porém, ao longo do compartilhamento, segmentos vão sendo liberados após o uso e novos segmentos são requisitados.



(a) Gerenciamento de processadores no Hipercubo

(b) Gerenciamento de área em memórias

Figura 3.2: Problema de alocação em hipercubos (a) e em memórias (b)

Tabela 3.1: Comparação do problema da alocação em memórias e no hipercubo

	Memória	**Hipercubo**
Unidade de Compartilhamento	páginas	subcubos
Reserva	página	nodo
Fragmentação Interna	menor que página	não é potência de dois
Fragmentação Externa	área livre não contígua	nodos livres não vizinhos
Cresce Fragmentação	lacunas	desordem
Compactação	coleta	migração

O problema se complica devido à fragmentação da memória, ou seja, o surgimento de espaços livres entre áreas alocadas. O aproveitamento destes espaços, para atender a novas requisições, se torna necessário, aumentando a complexidade do procedimento de gerência. Sem este aproveitamento, a área compartilhada será utilizada de forma ineficiente e requisições serão rejeitadas, apesar de se encontrarem espaços disponíveis. No caso do compartilhamento dos nodos de um hipercubo, os nodos livres que não são reconhecidos pelo sistema operacional como subcubos são como as áreas de memória livre entre espaços alocados. O compartilhamento eficiente do hipercubo só ocorrerá se estes nodos puderem ser utilizados em novas requisições e se a fragmentação do hipercubo for mantida baixa.

Na figura 3.3 podemos observar diferentes resultados de alocação para uma mesma seqüência de requisições. O hipercubo (b) se encontra mais compacto que o hipercubo (c), e desta forma aceitaria a requisição de mais um 2-cubo. No hipercubo (c) este pedido seria negado pois não existe um 2-cubo disponível devido a sua fragmentação.

Como apenas subcubos da máquina compartilhada são alocados, para que sejam mantidas as propriedades da topologia hipercúbica, pode ocorrer durante o processo de alocação, tanto fragmentação interna como fragmentação externa.

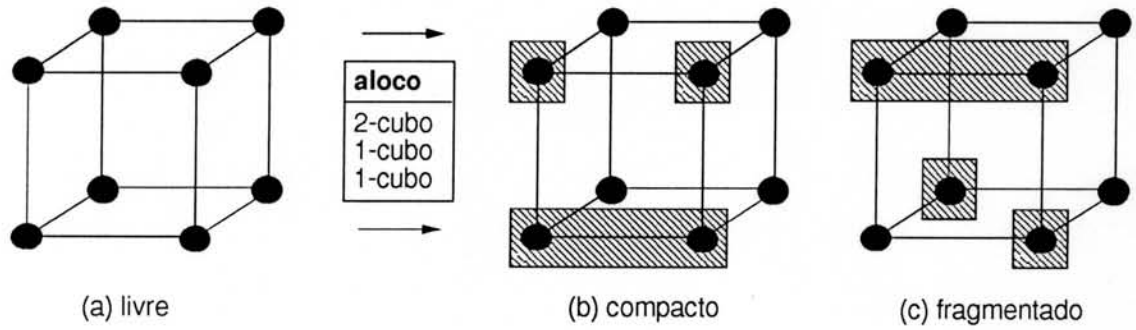


Figura 3.3: Fragmentação do hipercubo compartilhado

A fragmentação interna é caracterizada pelo fornecimento, por parte do algoritmo de alocação, de mais processadores do que de fato necessários para uma determinada aplicação. Isto ocorre pelo fato do algoritmo alocar o menor subcubo que contenha o número de processadores desejados pela aplicação. Como um subcubo possui sempre um número de processadores que seja potência de dois (0,1,2,4,8 ...), para um pedido de 5 processadores, por exemplo, seria alocado um subcubo de grau 3 com oito processadores. Os 3 processadores que não seriam empregados pela aplicação estariam marcados como alocados e não poderiam ser utilizados por nenhuma outra aplicação.

A fragmentação externa ocorre quando, após diversas operações de alocação e desalocação, existirem nodos livres para alocação, mas estes não formarem um subcubo. Sendo assim estes nodos não seriam alocados para nenhuma aplicação.

Existem portanto três possibilidades para que um pedido de alocação de subcubos seja negado:

- Não existe o número de processadores disponíveis para formar o menor subcubo que contenha os processadores pedidos por uma determinada aplicação;

- O número de processadores necessários para formar o menor subcubo que contenha os processadores pedidos por uma determinada aplicação está livre, mas estes processadores não formam um subcubo;
- O número de processadores necessários para formar o menor subcubo que contenha os processadores pedidos por uma determinada aplicação está livre, mas apesar de formarem este subcubo, não são reconhecidos como tal pelo algoritmo de alocação.

3.1.1 Fatores de avaliação dos algoritmos estudados

Após a análise detalhada do problema da alocação de processadores em máquinas hipercúbicas já é possível definir alguns fatores de avaliação para as soluções encontradas na literatura.

Ao longo da seção 3.2 serão feitas referências a estes fatores para avaliar a eficiência e eficácia dos algoritmos descritos.

Complexidade refere-se à quantidade de trabalho que é necessária por parte do algoritmo para que seja fornecido algum resultado. Este fator está relacionado com o tempo de resposta do algoritmo de forma diretamente proporcional em uma máquina seqüencial. Quanto mais complexo for o algoritmo, conseqüentemente mais tempo de processamento será necessário para que este forneça algum resultado.

Taxa de Reconhecimento refere-se à capacidade do algoritmo em reconhecer subcubos disponíveis dentre os nodos não alocados. Como todo o processo de alocação é baseado no fornecimento de subcubos, uma baixa taxa de reconhecimento aumenta o número de requisições negadas e resulta em uma pior taxa de utilização do hipercubo compartilhado.

Capacidade de Compactação refere-se à capacidade do algoritmo em manter o hipercubo compartilhado o mais compacto possível. Quanto mais compacto o hipercubo estiver, menos fragmentação existirá e mais subcubos livres serão reconhecidos. Alguns algoritmos possuem mecanismos de compactação implícitos no processo de alocação. Outros fornecem ferramentas de compactação que exigem uma parada no servidor para que possam ser executadas (explícitas).

Tendência a Fragmentação refere-se à velocidade com que o hipercubo compartilhado se fragmenta ao longo do tempo de execução do algoritmo. Esta velocidade está diretamente ligada com os mecanismos de compactação do algoritmo.

Funcionamento Interativo refere-se à possibilidade de utilização do algoritmo em um sistema interativo. Neste tipo de sistema o tempo de resposta do algoritmo tem que respeitar certos limites. Os algoritmos mais complexos não satisfazem estas condições.

Tolerância a Falhas refere-se à capacidade do algoritmo em lidar com eventuais defeitos em processadores durante o procedimento de alocação. Algoritmos com estruturas de alocação estáticas, geradas no início da execução do algoritmo e que não podem mais ser alteradas, não conseguem lidar com possíveis reconfigurações dos processadores.

Qualidade dos Resultados fator que indica a capacidade do algoritmo em fornecer bons resultados, ou seja, gerenciar o recurso compartilhado da melhor forma possível. Depende basicamente dos fatores taxa de reconhecimento, capacidade de compactação, e tendência a fragmentação. Se forem utilizados mecanismos eficazes para estas três áreas, o algoritmo certamente fornecerá bons resultados.

Com estes fatores pode-se facilmente definir o perfil do algoritmo ideal:

- Alta taxa de reconhecimento (de preferência 100%);
- Grande capacidade de compactação;
- Baixa tendência a fragamentação ao longo do tempo;
- Capacidade de tolerar reconfiguração dinâmica dos processadores em caso de falhas;
- Excelente qualidade dos resultados;
- Baixo tempo de resposta que respeite os limites impostos por um sistema de compartilhamento interativo.

Infelizmente as características iniciais deste algoritmo ideal são incompatíveis com a última. A inclusão de tantas facilidades como reconhecimento total, alta contenção da fragmentação e tolerância a falhas, requerem a inclusão de mecanismos específicos no algoritmo, tornando-o muito complexo e com um alto tempo de resposta. Este tempo de resposta, por sua vez, certamente não poderá ser tolerado por um sistema de compartilhamento interativo.

A figura 3.4 demonstra como se comportaram os fatores de avaliação Complexidade e Qualidade dos resultados durante a evolução dos algoritmos para a alocação e gerência de processadores.

Este trabalho propõe a utilização de paralelismo para que possa ser amenizada a relação direta identificada na figura 3.4 entre a qualidade dos resultados dos algoritmos de alocação e seu tempo de resposta.

3.1.2 A evolução dos algoritmos

Durante os últimos 20 anos, foram desenvolvidos algoritmos para a resolução do problema de alocação de processadores. A evolução destes algoritmos pode ser resumida nas 4 gerações de algoritmos descritas abaixo:

Figura 3.4: Relação entre os fatores Complexidade e Qualidade dos resultados durante a evolução dos algoritmos de alocação de processadores

1a Geração Os algoritmos são desenvolvidos para máquinas multiprocessadoras monousuário. São simples, com uma baixa taxa de reconhecimento (70%), mas muito rápidos. Não existe preocupação com a qualidade da alocação mas sim com o seu desempenho. As estruturas de dados utilizadas para a gerência dos processadores são listas indexadas. Os algoritmos BUDDY [PUR 70] e GC [CHE 87] pertencem a esta geração.

2a Geração Com a utilização de máquinas multiprocessadoras em ambientes multiusuário cresce a necessidade de uma alocação de processadores mais qualificada e uma melhor taxa de reconhecimento de subcubos, visando um melhor aproveitamento do recurso compartilhado. Aumenta a complexidade dos algoritmos para se atingir uma taxa de reconhecimento de 100% e conseqüentemente ocorre uma queda no desempenho. As estruturas de dados utilizadas são múltiplas listas indexadas. O algoritmo MGC [CHE 87] pertence a esta geração.

3a Geração Estudos visando um melhor aproveitamento dos processadores compartilhados aumentam a importância do conceito de fragmentação do hipercubo. Mecanismos para contenção da fragmentação são embutidos nos algoritmos, aumentando a sua complexidade de tal forma a comprometer a sua utilização em sistemas interativos. As estruturas

de dados utilizadas são árvores binárias e tabelas hash. Os algoritmos FL [KIM 91] e TC [CHU 90] pertencem a esta geração.

4a Geração A utilização de paralelismo nos algoritmos de alocação abre novos horizontes para as pesquisas nesta área. Além de possibilitar a utilização das técnicas da 3a geração em ambientes interativos, através de suas versões paralelas, são previstos novos algoritmos que sejam especialmente desenvolvidos para esta nova realidade. Desta forma se pretende viabilizar o desenvolvimento de algoritmos que forneçam resultados melhores com um tempo de resposta tolerável para o ambiente alvo.

Apesar de diversos autores já falarem em paralelismo e sugerirem formas de paralelizar seus algoritmos, pouco foi feito até agora no sentido de implementar algoritmos paralelos, identificar os problemas encontrados neste processo e avaliar os resultados obtidos. Também não são feitas referências claras de quais os recursos devem ser usados no processo de paralelização.

Este trabalho pretende preencher este vazio, forncendo elementos concretos para o estabelecimento e futuro desenvolvimento da 4a geração de algoritmos de alocação de processadores em máquinas hipercúbicas.

3.2 Algoritmos seqüenciais de gerência e alocação de subcubos

Nesta seção serão apresentados de forma superficial os principais algoritmos para alocação e gerência de processadores em máquinas hipercúbicas encontrados na literatura.

3.2.1 Algoritmo Buddy

A estratégia de gerência de alocação BUDDY [PUR 70] é utilizada pelo sistema operacional AXIS para o controle do ambiente multiusuário disponível nos hipercubos da série NCUBE [HAY 86].

O mecanismo de alocação utilizado atua sobre um vetor de bits numerado de 0 até $2^k - 1$ sendo k o grau do hipercubo. Cada posição deste vetor indica se o nodo em questão já está (1) ou não (0) alocado (figura 3.5a). Quando um j-cubo é requisitado, ocorre uma procura no vetor pelo menor inteiro m, de tal forma que todos os nodos entre m x 2^j até $((m+1)$ x $2^j) - 1$ estejam livres. Se estes nodos são encontrados, as respectivas posições no vetor são colocadas em 1 (figura 3.5b). Quando este subcubo for desalocado, as respectivas posições no vetor de bits são novamente colocadas em 0.

Indice	Alocado
000	0
001	0
010	0
011	0
100	0
101	0
110	0
111	0

(a) Estrutura inicial

Indice	Alocado
000	1
001	1
010	1
011	1
100	0
101	0
110	0
111	0

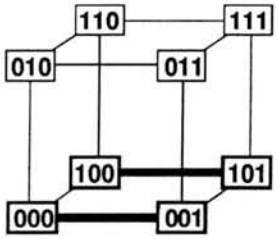
(b) Estrutura após a alocação de um 2-cubo

Figura 3.5: Estrutura de dados utilizada pelo algoritmo buddy antes (a) e depois (b) da alocação de um subcubo de grau 2 em um hipercubo de grau 3

A simplicidade do mecanismo de alocação de nodos resulta em um ótimo tempo de resposta, porém, com uma baixa capacidade de reconhecimento de subcubos disponíveis e rápida tendência a fragmentação.

A dificuldade em reconhecer os subcubos disponíveis pode ser vista em um exemplo. A figura 3.6 apresenta o estado de um hipercubo de grau 3 após diversas requisições de alocação e liberação de subcubos. Estão alocados

dois subcubos de grau 1 indicados por nodos pretos. Uma terceira requisição solicitando um subcubo de grau 2 seria rejeitada, apesar de podermos ver que existem 4 nodos disponíveis (nodos brancos) e que estes formam um subcubo.



(a) O 2-cubo (010,011,110,111) está disponível para alocação

Indice	Alocado
000	1
001	1
010	0
011	0
100	1
101	1
110	0
111	0

(b) A estrutura buddy correspondente não reconhece nenhum 2-cubo disponível

Figura 3.6: Exemplo da pouca capacidade de reconhecimento de subcubos por parte da estratégia buddy

3.2.2 Algoritmo Gray Codes

A estratégia de alocação Gray Codes GC [CHE 87] pode ser considerada uma evolução da técnica buddy. A técnica GC também se utiliza de uma lista de alocação para efetuar o controle dos nodos disponíveis, porém o mapeamento desta lista com os nodos do hipercubo é feito segundo os códigos de gray (seção 2.2.8.2). Normalmente o código de gray utilizado é o BRGC (Binary Reflected Gray Code).

A alocação de subcubos na lista de alocação é feita através da procura de 2^j nodos consecutivos na forma p até $p + 2^j - 1$, sendo k o grau do subcubo desejado e p o menor inteiro múltiplo de $2^j - 1$.

Foi provado em [CHE 87] que a estratégia GC pode reconhecer 2^{n-k+1} subcubos, sendo k o grau do subcubo, n o grau do hipercubo a ser compartilhado e $1 \leq k \leq n-1$, o que é o dobro de subcubos reconhecidos pela estratégia Buddy.

Esta melhora no resultado obtido pelo algoritmo GC advém da utilização dos códigos de gray como índices da estrutura de alocação (figura 3.7(b)). Desta forma são aproximados na lista de alocação os nodos que são vizinhos no hipercubo, permitindo que as sucessivas requisições de alocação possam ser mais compactadas que na estratégia Buddy, como pode ser visto na figura 3.7.

Indice	Alocado
0000	1 Q1
0001	1 Q3
0010	1 Q4
0011	0
0100	1 Q2
0101	1 Q2
0110	1 Q2
0111	1 Q2
1000	1 Q5
1001	1 Q5
1010	0
1011	0
1100	0
1101	0
1110	0
1111	0

(a) Estrutura buddy

Indice	Alocado
0000	1 Q1
0001	1 Q3
0011	1 Q2
0010	1 Q2
0110	1 Q2
0111	1 Q2
0101	1 Q4
0100	1 Q5
1100	1 Q5
1101	0
1111	0
1110	0
1010	0
1011	0
1001	0
1000	0

(b) Estrutura GC

Figura 3.7: Comparação entre o modo de operação da estratégia buddy (a) e da estratégia GC (b)

A explicação para esta aproximação baseia-se na definição dos códigos de gray e do conceito de vizinhança em hipercubos. Nas seqüenciais de gray, o antecessor e o sucessor de um número varia apenas em um bit, que é a mesma condição para a identificação de nodos vizinhos em um hipercubo (seção 2.1).

Como a utilização de códigos de gray tende a compactar sucessivas alocações de subcubos, a fragmentação acelerada do hipercubo é contida, resultando em um melhor compartilhamento dos processadores.

3.2.3 Algoritmo Multiple Gray Code

A estratégia GC pode reconhecer todos os subcubos disponíveis em um hipercubo compartilhado se forem utilizados múltiplos códigos de gray (MGC) [CHE 87]. A tabela 3.2 indica o número de códigos de gray necessários para uma taxa de reconhecimento de 100% em relação a dimensão do hipercubo a ser compartilhado.

Tabela 3.2: Número de códigos de gray necessários para reconhecimento total de subcubos na técnica MGC

Dimensão	4	5	6	7	8	9	10	11	12	13	14
Nodos	16	32	64	128	256	512	1024	2048	4096	8192	16384
Códigos	6	10	20	35	70	126	252	462	924	1716	3432

O procedimento de procura é o mesmo utilizado pelo algoritmo GC só que aqui tem que ser aplicado em todos os códigos seqüencialmente. Isto torna o algoritmo MGC mais complexo e o tempo de resposta maior, pois diversas tabelas teriam que ser consultadas. Além disto, a geração de múltiplos códigos de gray é demorada, e tem que ser feita previamente e armazenada em tabelas de consulta. Com esta geração estática, uma alteração na numeração dos nodos, causada, por exemplo, por uma reconfiguração após alguma falha, resultaria em um mau funcionamento do procedimento de gerência.

O princípio de funcionamento do algoritmo MGC baseia-se na utilização de diversos códigos de gray para aproximar os nodos do hipercubo de todas as formas necessárias, tornando possível que seja alcançada uma taxa de reconhecimento de 100%.

Na alocação de processadores, o procedimento de procura do algoritmo GC é aplicado em cada um dos códigos de gray, até que o subcubo desejado seja encontrado. Os bits da estrutura de alocação são então atualizados no código em que o subcubo foi encontrado e também em todos os outros. Na verdade, os múltiplos

códigos utilizados são todos imagens fiéis do mesmo mapa de alocação de bits do subcubo compartilhado. A única diferença entre eles reside na ordenação de suas tabelas de índice, como podemos ver na figura 3.8. Estas diferentes ordenações nos índices permitem que o mesmo algoritmo de procura seqüencial utilizado no algoritmo GC aplicado nos diferentes códigos, possa reconhecer um maior número de subcubos. O procedimento de alocação pode ser interrompido após o encontro do primeiro subcubo disponível, não sendo necessária a procura em todos os códigos. Caso após a procura em todos os códigos, não for encontrado um subcubo disponível, é possível afirmar com certeza que este não existe, por se tratar de um algoritmo com taxa de reconhecimento de 100%.

Indice GC_1	**Pid**	**Aloc.**	GC_2		GC_3		GC_4		GC_5		GC_6	
0	7	1	0	1	0	1	0	1	0	1	0	1
1	7	1	1	1	1	1	1	1	8	0	8	0
3	9	1	3	1	9	0	9	0	9	0	9	0
2	9	1	2	1	8	0	8	0	1	1	1	1
6	9	1	10	0	10	0	12	0	3	1	5	0
7	9	1	11	0	11	0	13	0	11	0	13	0
5	0	0	9	0	3	1	5	0	10	0	12	0
4	0	0	8	0	2	1	4	0	2	1	4	0
12	0	0	12	0	6	1	6	1	6	1	6	1
13	0	0	13	0	7	1	7	1	14	0	14	0
15	0	0	15	0	15	0	15	0	15	0	15	0
14	0	0	14	0	14	0	14	0	7	1	7	1
10	0	0	6	1	12	0	10	0	5	0	3	1
11	0	0	7	1	13	0	11	0	13	0	11	0
9	0	0	5	0	5	0	3	1	12	0	10	0
8	0	0	4	0	4	0	2	1	4	0	2	1

Figura 3.8: Estrutura de alocação MGC em um hipercubo de grau 4

Na desalocação de um subcubo, todos os códigos tem que ser varridos e os respectivos bits de alocação do subcubo em questão tem que ser alterados para 0.

A utilização desta estratégia não é viável em hipercubos de maior grau, devido ao rápido crescimento do número de códigos necessários para que seja mantida uma alta taxa de reconhecimento, como podemos ver na tabela 3.2.3.

3.2.4 Lista de Cubos Livres

A estratégia de alocação denominada lista de cubos livres (*Free List*) [KIM 91] efetua o controle dos nodos alocados através de listas de subcubos disponíveis no hipercubo compartilhado, uma lista para cada dimensão. Esta técnica é capaz de reconhecer todos os subcubos disponíveis, porém sua complexidade é extremamente alta, superior a $O(n^3)$.

A figura 3.9 apresenta um exemplo da estrutura de dados utilizada pela técnica de lista de cubos livres para um hipercubo de grau 4, demonstrando as alterações sofridas após operações de alocação e desalocação.

Uma requisição para um subcubo de dimensão k é atendida através da alocação do primeiro subcubo encontrado na lista de cubos livres de dimensão k. Caso esta lista esteja vazia, é procurado, em listas de dimensão maior, o primeiro subcubo disponível que tenha um grau maior do que k. Se não existir nenhum, a requisição é rejeitada, pois não existe subcubo desta dimensão disponível no momento. Caso seja encontrado um subcubo de grau maior do que k, este subcubo é decomposto em dois subcubos de dimensão menor, retirado de sua lista e os novos cubos são adicionados na lista de dimensão inferior (figura 3.9 (a)(c)). Este procedimento é repetido até que seja gerado um subcubo de dimensão k. Este subcubo então é alocado e retirado da lista de cubos livres.

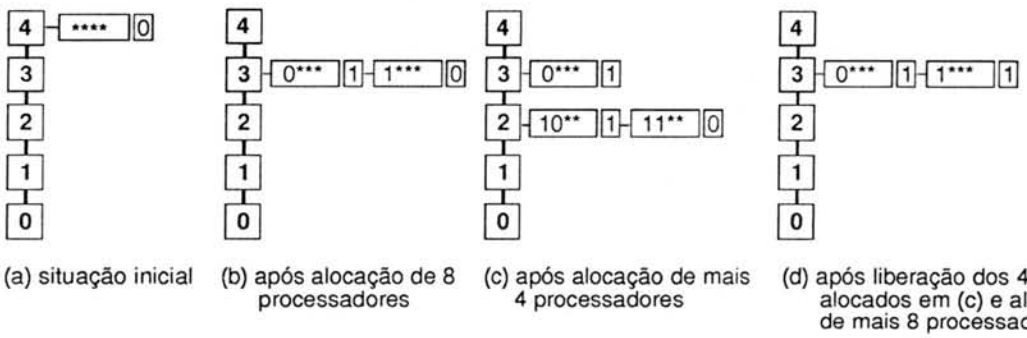


Figura 3.9: Atualização da estrutura utilizada pelo algoritmo FL

Apesar do procedimento de alocação ser relativamente simples, o processo de liberação de subcubos é muito complexo, pois quando um subcubo é liberado é necessário verificar se não é possível recompor algum subcubo de grau superior (figura 3.9 (d)). Este tipo de operação pode envolver todas as listas de subcubos livres e sua complexidade é proporcional ao tamanho do hipercubo a ser compartilhado.

Mesmo reconhecendo todos os subcubos disponíveis, a complexidade de $O(n^3)$ torna este método inviável para hipercubos de maior grau em um sistema de tempo real.

Em [DUT 91] é apresentada uma estratégia de alocação denominada MSS (Maximal Set of Subcubes), que segue o mesmo princípio de controle dos subcubos disponíveis em um hipercubo compartilhado. É feita uma análise mais detalhada do processo de desalocação, a fase mais complexa desta estratégia, e são propostas algumas alternativas.

3.2.5 Algoritmo Tree Collapsing

A estratégia de alocação denominada TC (*Tree Collapsing*) [CHU 90] é uma extensão da estratégia Buddy, capaz de detectar a totalidade dos subcubos disponíveis em um hipercubo compartilhado, sendo menos complexa que a estratégia de múltiplos códigos gray (MGC).

A idéia básica desta estratégia é de alterar a ordem dos nodos na lista de alocação da estratégia Buddy, gerando novas seqüências de códigos dinamicamente de acordo com a necessidade. Os nodos que estavam distantes na tabela de alocação e conseqüentemente não eram reconhecidos como subcubos, são aproximados através da consecutiva alteração nos ramos de uma árvore binária (*tree collapsing*) que representa o hipercubo compartilhado, permitindo assim seu reconhecimento (figura 3.10 (a)).

Esta estratégia não difere muito da aplicação dos diferentes códigos de gray para o reconhecimento completo dos subcubos possíveis de alocação. Porém, ao invés dos diferentes códigos, são utilizadas seqüências de nodos resultantes de diferentes mapeamentos de árvores binárias no hipercubo. No momento da alocação, estas seqüências são consultadas para que seja alcançada a capacidade total de reconhecimento de subcubos. A vantagem em relação ao método MGC é que a geração destas seqüências, através da operação de collapse em árvores binárias, é menos complexa que a geração de múltiplos códigos de gray, e pode ser feita em tempo real quando necessário.

A figura 3.10(a) apresenta a árvore binária denominada primária, que é o ponto de partida para a geração das seqüências de reconhecimento através da operação de collapse. É interessante ressaltar que o reconhecimento dos subcubos em uma árvore binária é feito através dos nodos que não são folhas. As folhas representam os processadores que compõem os subcubos. Se considerarmos, por exemplo, o nodo localizado na raiz da árvore binária da figura 3.10(a) é possível verificar que este nodo tem dois filhos, que por sua vez podem ser vistos como o nodo raiz de duas subárvores. Como a árvore da figura representa um hipercubo de grau 4, cada uma destas subárvores representa um subcubo de grau 3, e os seus nodos são identificados pelo número dos processadores associados às suas folhas. Quanto mais nos dirigirmos em direção às folhas da árvore, as subárvores serão cada vez menores e conseqüentemente subcubos de menor grau podem ser reconhecidos.

A operação de *collapse* é aplicada em um determinado nível da árvore binária, dependendo da dimensão do subcubo desejado, e consiste no entrelaçamento das subárvores de um determinado nível, gerando uma nova seqüência nos nodos associados às suas folhas. A figura 3.10 (b) demonstra o processo de entrelaçamento da operação de *collapse* no nível 1 da árvore primária.

No atendimento de um pedido de alocação, as folhas de uma árvore binária inicial são consultadas através de um algoritmo semelhante ao algoritmo

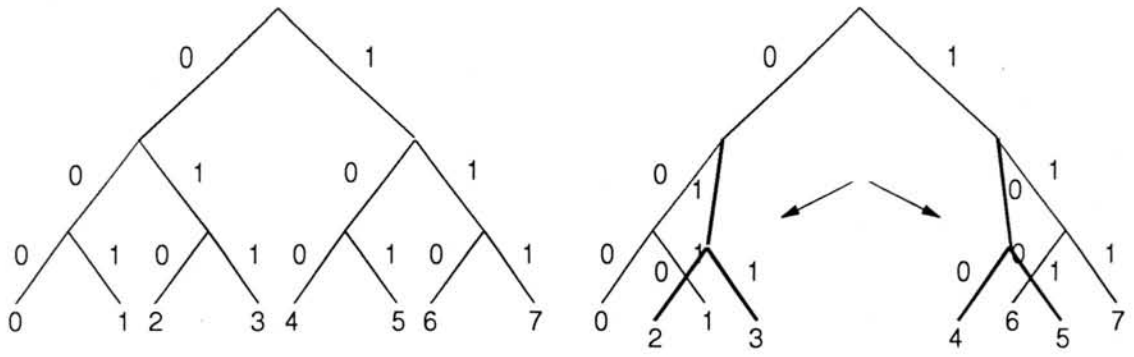


Figura 3.10: Estrutura utilizada pelo algoritmo TC

utilizado na estratégia Buddy. Caso a resposta da procura seja negativa, são efetuadas sucessivas operações de *collapse*, seguidas de uma nova consulta, até que seja encontrado um subcubo que atenda o pedido de alocação, ou que se esgote o número de tentativas do algoritmo. Como o algoritmo TC tem uma taxa de reconhecimento de 100%, o fato de se esgotar o número de tentativas do algoritmo indica que efetivamente não existe subcubo disponível que atenda o pedido de alocação.

O processo de desalocação de um subcubo se resume na alteração dos bits de alocação de seus nodos de 1 para 0. Esta alteração é feita através de uma varredura seqüencial nas folhas da árvore.

O algoritmo abaixo apresenta o procedimento de geração de todas as seqüências necessárias para o reconhecimento de todos os subcubos do hipercubo a ser compartilhado. Basicamente são geradas diferentes árvores binárias, através de todas as combinações possíveis de operações de *collapse* sobre a árvore primária. `Arvore_primaria`, `nova_arvore` e `pre_arvore` são árvores binárias, n é o grau do hipercubo a ser compartilhado e k o grau do subcubo desejado.

```
Procedure main ( n , k , arvore_primaria )
inicio
   se ( k == 0 ou k == n ) entao pare
   para i := n - k - 1 diminuindo ate 0
```

```
        executa collapse( nivel i, arvore_primaria, nova_arvore )
        executa novo_collapse( nivel i, 0, nova_arvore)
    fimpara
fim

Procedure novo_collapse ( pre_nivel , pre_step , pre_arvore )
inicio
    step := pre_step +1
    se step > k entao retorne
    senao
        para i := pre_level ate n - k - 1
            executa collapse( i , pre_arvore , nova_arvore )
            executa novo_collapse( i , step , nova_arvore )
        fimfor
    fimif
fim
```

O atendimento a um pedido de alocação deve ser feito seguindo os seguintes passos:

1. Inicialize k com o grau do subcubo desejado;
2. Procure no nível $n - k$ da árvore primária da esquerda para direita por um subcubo disponível (folhas da subárvore correspondente estão em 0). Se encontrar vá para o passo 4;
3. Execute o algoritmo *collapse* e procure um subcubo disponível no nível $n - k$ de cada árvore resultante, uma de cada vez. Se encontrar o subcubo vá para o passo 4. Senão vá para o passo 5;
4. Marque todas as folhas do subcubo encontrado com 1 e aloque o cubo. Pare;
5. Rejeite o pedido de alocação.

O processo de desalocação de um subcubo se resume na alteração dos bits de alocação de seus nodos de 1 para 0.

O cálculo das seqüências de reconhecimento pode ser executado com operações básicas de rotação e deslocamento, aumentando assim o desempenho do algoritmo de geração.

3.2.6 Resumo dos algoritmos apresentados

A tabela 3.3 apresenta um resumo das características dos algoritmos para alocação e gerência de processadores em máquinas hipercúbicas vistos até agora.

Uma descrição detalhada de alguns dos fatores analisados é fornecida na seção 3.1.1.

Tabela 3.3: Resumo das características encontradas nos algoritmos

	BUDDY	GC	MGC	TC	FL
Complexidade	$O(2^n)$	$O(2^n)$	$O((n/\lceil n/2 \rceil)2^n)$	$O(n2^n)$	$O(n^3)$
Uso de Memória	baixo	baixo	muito alto	médio	alto
Taxa de Reconhecimento	60%	70%	100%	100%	100%
Compactação	nenhuma	explícita	explícita	implícita	implícita
Fragmentação	rápida	rápida	rápida	média	lenta
Sistema Interativo	apto	apto	não-apto	não-apto	não-apto
Estrutura de dados	lista indexada	lista indexada	listas múltiplas	árvore	tabela hash
Qualidade dos resultados	ruim	razoável	bom	ótimo	ótimo

4 PARALELIZANDO OS ALGORITMOS

Como visto anteriormente, a operação de alocação e gerência de processadores tem uma influência muito grande no desempenho final de uma máquina multiprocessadora. Uma melhora no tempo de resposta destes algoritmos acarretaria uma melhora no desempenho global destas máquinas.

A paralelização dos algoritmos de alocação e gerência para máquinas hipercúbicas busca, não apenas esta melhora no desempenho (eficiência), mas principalmente possibilitar a utilização ,em um sistema interativo, de algoritmos que são considerados muito complexos para este tipo de ambiente. Com a utilização de algoritmos mais avançados é obtida uma melhora no resultado da operação de alocação (eficácia) e a máquina hipercúbica pode ser melhor compartilhada.

4.1 Versões paralelas dos algoritmos de alocação

Nas seções abaixo são apresentadas versões paralelas da maioria dos algoritmos de alocação e gerência de processadores em máquinas hipercúbicas descritos no capítulo 3. Os algoritmos escolhidos foram aqueles que de alguma forma indicavam a possibilidade de obtenção de ganho de desempenho com o procedimento de paralelização. Uma análise detalhada dos resultados obtidos pelas versões paralelas pode ser encontrada no capítulo 5.

4.1.1 Algoritmo Buddy/GC Paralelo

Na versão paralela do algoritmo BUDDY o processador da máquina hospedeira é considerado o mestre e os outros processadores os escravos, como

pode ser visto na figura 4.1. O mestre é responsável por receber os pedidos de alocação ou desalocação, distribuir o trabalho a ser feito entre os escravos, sincronizar as operações quando necessário, e responder aos pedidos.

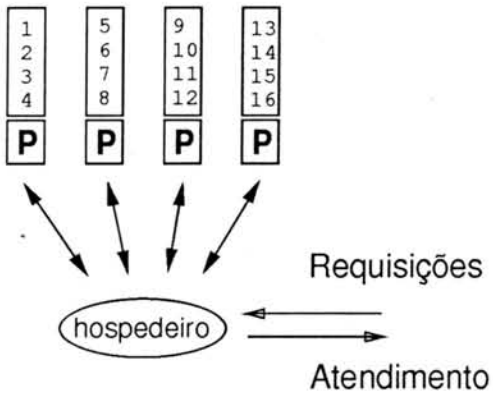


Figura 4.1: Modelo mestre-escravo da versão paralela do algoritmo buddy

O princípio básico deste algoritmo paralelo é a distribuição da lista de alocação entre os processadores escravos. Cada escravo gerencia a sua parte da lista alocando e desalocando processadores segundo orientação do mestre. Para não ficar subutilizado, o mestre também pode receber uma parte da lista de alocação.

Como a lista de alocação é varrida seqüencialmente no algoritmo seqüencial, tanto na alocação como na desalocação de processadores, ambos os procedimentos são otimizados quando as listas são divididas entre os processadores. A versão paralela do algoritmo BUDDY fica da seguinte forma:

- **Inicialização** A lista de alocação, estrutura utilizada para marcar os processadores alocados, é fragmentada e distribuída entre os processadores envolvidos.

- **Alocação Paralela**

1. o processador mestre recebe o pedido de alocação e envia para os processadores escravos o número de processadores desejados e a identificação do processo (pid) associado ao pedido;
2. mestre e escravos executam a operação de alocação em suas listas locais;
3. o primeiro processador que encontrar um resultado atualiza sua lista de alocação e envia o número dos processadores alocados para o processador mestre (se não o for). A atualização da lista de alocação só pode ser feita por um processador;
4. caso os escravos não encontrem livres o número de processadores desejados em suas listas, enviam um sinal NOT para o mestre;
5. para que a operação seja concluída pelo mestre este: (i) espera todos os escravos responderem com NOT e nega o pedido de alocação, (ii) espera a lista de nodos alocados de um de seus escravos ou, (iii) não espera por nada por ter ele mesmo encontrado os processadores desejados em sua lista local.

- **Desalocação Paralela**

1. mestre recebe pedido desalocação e envia pid a ser desalocado para escravos;
2. mestre e escravos liberam os nodos do pid recebido em suas listas locais;
3. não existe a necessidade de sincronização no final da operação, estando o mestre liberado para novos pedidos logo após o final da consulta a suas listas locais.

Como o processador mestre tem mais atribuições que os escravos, e conseqüentemente mais carga de trabalho, pode ser feito um balanceamento de carga estático antes do início das operações, atribuindo-se fragmentos maiores da lista de alocação aos escravos do que ao mestre.

O controle da alocação de todo o hipercubo compartilhado é feito no processador da máquina hospedeira (mestre) com apenas um bit. Se este mecanismo não fosse implementado não seria possível a alocação de todo o hipercubo compartilhado, pois sua lista de alocação está partida em pedaços e os nodos não seriam reconhecidos como vizinhos. Caso este algoritmo paralelo seja implementado em mais de 2 processadores, mecanismos semelhantes para a alocação de dimensões menores tem que ser implementados. Se, por exemplo, a lista de alocação de um hipercubo de grau k for dividida entre 4 processadores de forma igual, nenhum deles será capaz de alocar um subcubo com 2^{k-1} processadores.

O mesmo algoritmo paralelo descrito acima pode ser utilizado para a estratégia GC alterando-se as listas de alocação em cada processador (figura 4.2). Também é necessária uma pequena alteração na forma da procura dos nodos livres durante a fase de alocação de processadores, como no algoritmo GC (seção 3.2.2).

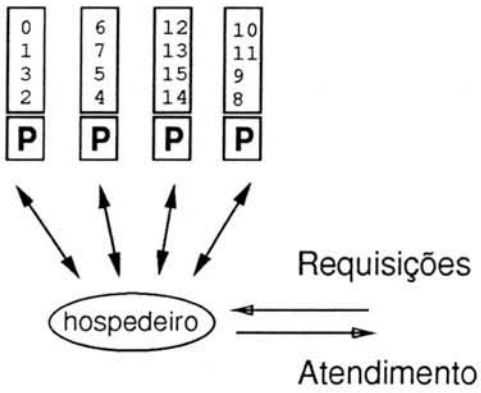


Figura 4.2: Modelo mestre-escravo da versão paralela do algoritmo GC

4.1.1.1 O problema da atualização das listas de alocação

Um problema comum às versões paralelas dos algoritmos BUDDY e GC é a atualização da lista de alocação quando da alocação de processadores. Como a lista de alocação foi dividida entre os processadores, cada um varre a sua lista local a procura de processadores livres para a alocação. No entanto, esta lista só pode

ser atualizada por um processador, para que não ocorram diversas alocações para um mesmo pedido. Sendo assim, o código responsável pela atualização desta lista pode ser visto como uma seção crítica e deve ser controlado por um mecanismo que garanta as seguintes condições:

- exclusividade mútua, ou seja, se um processo está na sua seção crítica, nenhum outro processo pode entrar na sua seção crítica;
- após a execução da seção crítica por um processo, nenhum outro processo pode executar a sua seção crítica.

O mecanismo de semáforos [SIL 91] foi escolhido para o controle desta seção crítica. Para que possa garantir as condições acima, foram feitas as seguintes alterações nas diretivas P e V:

```
init MUTEX = 1;

int P(MUTEX)
{
MUTEX = MUTEX - 1;
if ( MUTEX >= 0 )
   return(true);
else
   return(false);
}

V(MUTEX)
{
MUTEX = MUTEX + 1;
}
```

A diretiva P é chamada no início da seção crítica. Caso retorne 0 (*false*), a seção crítica já foi executada por outro processo e não deve ser executada. Caso retorne 1 (*true*), a seção pode ser executada. No final da execução da operação de

alocação, o mestre deve chamar a diretiva V para reinicializar o semáforo para o próximo pedido de alocação. Ambas as diretivas tem que ser implementadas de forma atômica.

O grafo de execução das versões paralelas dos algortimos BUDDY e GC com a utilização de semáforos para o controle da seção crítica, pode ser visto na figura 4.3.

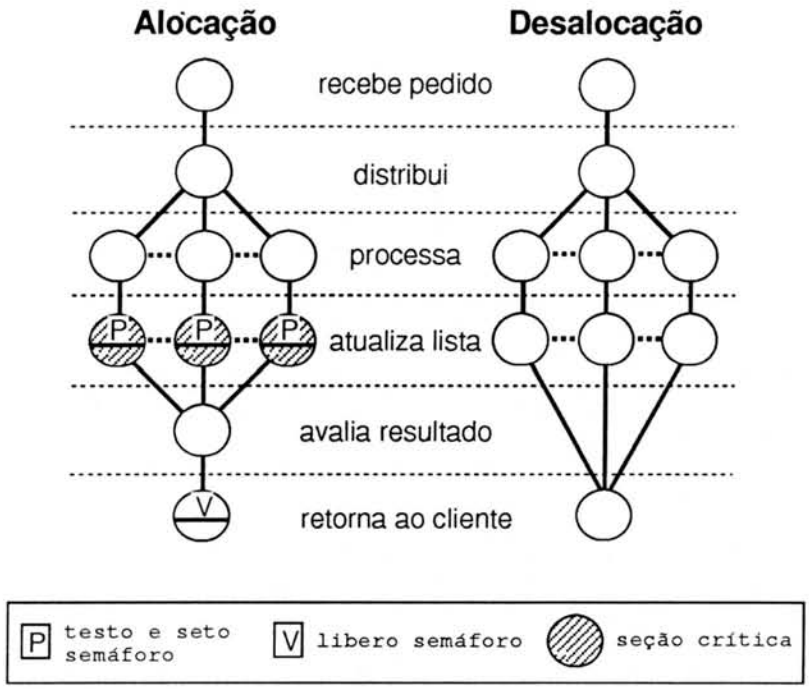


Figura 4.3: Grafo de execução das versões paralelas dos algoritmos BUDDY e GC

4.1.1.2 Otimização através de aceite ou negação imediatos

Grande parcela do tempo de execução das versões paralelas dos algoritmos BUDDY e GC é consumida na distribuição da tarefa a ser realizada entre os processadores envolvidos e na posterior coleta dos resultados.

Este processo de distribuição e coleta de resultados pode ser considerado tempo perdido se o resultado da operação for uma alocação na lista local ao processador mestre, ou a negação de um pedido de alocação.

Desta forma, mecanismos simples, que neguem ou aceitem pedidos de forma imediata por parte do mestre, sem que ocorra consulta aos escravos, aumentam consideravelmente o desempenho da versão paralela do algoritmo BUDDY e do algoritmo GC. Um exemplo deste tipo de mecanismo seria a negação imediata de um pedido de alocação de um número de processadores maior do que os que estão disponíveis no momento (controle local e remoto), ou a aceitação imediata de um pedido por parte do mestre quando o número de processadores alocados no hipercubo for muito pequeno e estiver disponível localmente.

É natural que muitos dos critérios utilizados para negar ou aceitar imediatamente uma requisição dependam das características da máquina onde o algoritmo foi implementado, do particionamento utilizado na distribuição da lista de alocação entre os processadores, e do perfil das requisições que são feitas no sistema.

Porém, alguns mecanismo genéricos podem ser definidos para a otimização deste algoritmo em qualquer sistema:

- **Negação Imediata**
  - O mestre possui um controle do número de processadores livres na máquina compartilhada. Um pedido maior do que o número de processadores livres é automaticamente negado;
  - O mestre possui um controle do número de processadores livres em cada uma das partes da lista de alocação. Processadores com listas de alocação que estão sem o número necessário de processadores livres para atender uma determinada requisição, nem chegam a ser ativados.
- **Aceite Imediato**
  - Como já vimos anteriormente, o controle das dimensões mais altas fica exclusivamente no mestre, na forma bits de alocação, devido a

divisão da lista de alocação. O mestre pode, desta forma, atender automaticamente requisições de hipercubos de grau alto;

- Com o controle do número de processadores livres em sua lista de alocação local o processador mestre pode atender localmente uma requisição, se existirem processadores livres para tal. A decisão de não ativar os processadores escravos, para atender um pedido localmente, deve ser feita apenas para requisições de hipercubos de grau pequeno (em relação ao tamanho da lista local) e com uma margem de segurança. O mestre não pode garantir que os processadores livres, apesar de suficientes, formem, ou sejam reconhecidos como um hipercubo pelo algoritmo de alocação.

4.1.2 Múltiplos Códigos de Gray Paralelo

Na versão paralela do algoritmo MGC é utilizado o mesmo modelo mestre-escravo descrito no algoritmo BUDDY paralelo.

A principal diferença é que, devido a necessidade do algoritmo MGC de trabalhar com várias seqüências de códigos de gray, na versão paralela cada processador recebe parte destes códigos para gerência, como podemos ver na figura 4.4.

O cálculo e a distribuição dos códigos de gray entre os processadores é feito *off-line* e não pode ser alterada durante a operação (estático). A versão paralela do algoritmo MGC fica da seguinte forma:

- **Inicialização** Os códigos de gray necessários para atingir a taxa de reconhecimento desejada são distribuídos igualmente entre os processadores envolvidos (mestre e escravos).

- **Alocação Paralela**

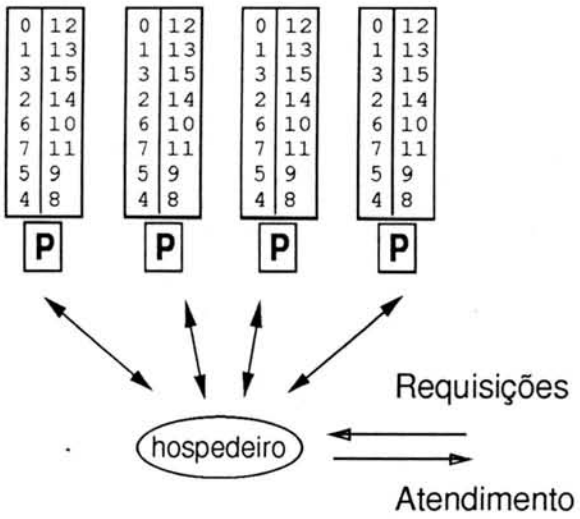


Figura 4.4: Modelo mestre-escravo da versão paralela do algoritmo MGC

1. o processador mestre recebe pedido de alocação e envia para os escravos o número de processadores desejados e a identificação do processo associado ao pedido;
2. mestre e escravos iniciam a procura por processadores livres em suas listas locais;
3. o primeiro processador que encontrar um resultado atualiza sua lista de alocação e envia o número dos processadores alocados para o processador mestre (se não o for);
4. caso os escravos não encontrem livres o número de processadores desejados em suas listas, enviam um sinal NOT para o mestre;
5. para que a operação seja concluida pelo mestre este: (i) espera todos os escravos responderem com NOT e nega o pedido de alocação, (ii) espera a lista de nodos alocados de um de seus escravos ou, (iii) não espera por nada por ter ele mesmo encontrado o número de processadores desejado em suas listas locais;
6. como as atualizações tem que ser feitas em todos os códigos, o processador mestre envia o índice dos processadores alocados para que os escravos possam atualizar suas listas locais.

- **Desalocação Paralela**

1. mestre recebe pedido desalocação e envia a identificação do processo a ser desalocado para escravos;
2. mestre e escravos liberam os nodos correspondentes a identificação do processo recebida, atualizando suas listas locais;
3. não existe a necessidade de sincronização no final da operação, estando o mestre liberado para novos pedidos logo após o final da consulta à suas listas locais.

Para resolver o problema da seção crítica no procedimento de atualização da lista de alocação foi utilizado o mesmo mecanismo de semáforos apresentado na seção 4.1.1.1.

O mesmo balanceamento de carga estático proposto no algoritmo BUDDY paralelo pode ser aplicado aqui se antes do início das operações forem atribuídos mais códigos de gray aos escravos do que ao mestre.

Os mesmos mecanismos de aceitação e negação imediata de pedidos propostos no algoritmo BUDDY paralelo também aumentam consideravelmente o desempenho do algoritmo MGC paralelo.

4.1.3 Tree Collapse Paralelo

Uma versão paralela do algoritmo de *Tree Collapse* é proposta por Chuang e Tzeng em [CHU 90]. Como a operação de *collapse* é bastante complexa e durante a fase de alocação é executada normalmente por diversas vezes, a versão paralela se baseia na execução desta operação por vários processadores em paralelo. A operação de desalocação é executada, devido a sua simplicidade, apenas no processador da máquina hospedeira.

Requisições de subcubos são gerenciadas pelo processador da máquina hospedeira, que mantém o vetor completo de bits de alocação. No recebimento de

um pedido de alocação, o processador hospedeiro fornece uma cópia do vetor de bits de alocação para cada processador envolvido, juntamente com a seqüência de operações que deve ser efetuada sobre ele. Os processadores envolvidos efetuam suas seqüências de operações e a procura pelo subcubo desejado em paralelo. O primeiro processador que encontrar, avisa o processador hospedeiro, que aborta a operação incompleta dos processadores restantes. O vetor de bits de alocação é então atualizado no processador hospedeiro. A mecânica de execução da versão paralela do algoritmo TC pode ser melhor compreendida com o auxílio da figura 4.5.

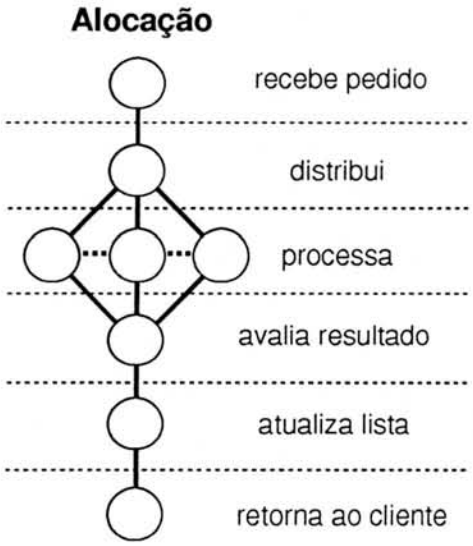


Figura 4.5: Modelo de execução da versão paralela do algoritmo TC

Como a lista de alocação é gerenciada apenas pelo processador mestre, não existe o problema da seção crítica na operação de atualização, não havendo necessidade da utilização de semáforos como nos algoritmos anteriores.

O número de processadores envolvidos no processo de alocação pode ser pré-determinado ou escolhido em tempo de execução. A segunda forma resulta em melhores resultados, pois evita que vários processdores sejam envolvidos em um procedimento de alocação simples, que poderia ser facilmente resolvido até mesmo pelo processador hospedeiro. O processamento necessário para uma

alocação pode ser estimado pelo processador hospedeiro com base na taxa de ocupação e na taxa de fragmentação do hipercubo compartilhado.

4.1.4 Lista de Cubos Livres Paralelo

Uma versão paralela do algoritmo de lista de cubos livres é proposta por Kim e Das em [KIM·91].

- **Inicialização** A lista de cubos livres para cada dimensão do hipercubo a ser compartilhado é mantida em um processador diferente. Para um hipercubo compartilhado de grau n são necessários $(n+1)$ processadores incluindo o processador da máquina hospedeira.

- **Alocação Paralela**

  1. o processador da máquina hospedeira envia o grau do subcubo desejado (k) para os processadores que mantém as listas de dimensão $\geq k$;
  2. todos os processadores dentre os acima que, possuirem um subcubo livre de dimensão m para um $m \geq k$ respondem a requisição enviando o subcubo disponível;
  3. o processador da máquina hospedeira escolhe o subcubo de dimensão mais próxima a k decompondo-o se necessário ($> k$) e enviando os subcubos decompostos para os respectivos processadores e já marcando o subcubo escolhido como alocado;
  4. o processador hospedeiro envia uma mensagem para que o processador da dimensão do subcubo que foi decomposto, se existir, o marque como alocado.

- **Desalocação Paralela**

1. o subcubo k desalocado é enviado e adicionado na lista do processador responsável pelos subcubos de dimensão k;
2. o processador da máquina hospedeira envia o subcubo k liberado para todos os processadores que mantém dimensões $> k$. Todos estes processadores geram possíveis subcubos, comparando suas listas com o subcubo recém liberado. Os subcubos i gerados são enviados para o processador responsável pela dimensão i;
3. cada nodo envia uma cópia da sua lista para o processador da dimensão acima. Se o processador que recebeu a mensagem estiver com sua lista vazia, a mensagem é redirecionada para o processador de dimensão acima. Ao mesmo tempo, cada processador gera subcubos sobrepostos, utilizando sua lista e os subcubos recebidos pelo vizinho de dimensão inferior. Se um dos subcubos recebidos não puder ser combinado, o processador o envia para o processador responsável pela lista de dimensão diretamente superior;
4. do processador que controla a lista da maior dimensão, é enviado um subcubo por vez para todos os outros processadores. Este passo é repetido sucessivamente pelos processadores responsáveis pelas dimensões menores, um a um;
5. cada processador decompõe seus subcubos que possuem nodos comuns com os subcubos recebidos. Os subcubos comuns são apagados e os subcubos restantes de menor dimensão são enviados para os processadores correspondentes para atualização. Os passos 4 e 5 podem ser feitos em paralelo.

Como podemos ver, o algoritmo de listas de cubos livres, como foi originalmente proposto por [KIM 91], tem uma alta complexidade e gera um grande número de mensagens entre os processadores, principalmente na fase de desalocação.

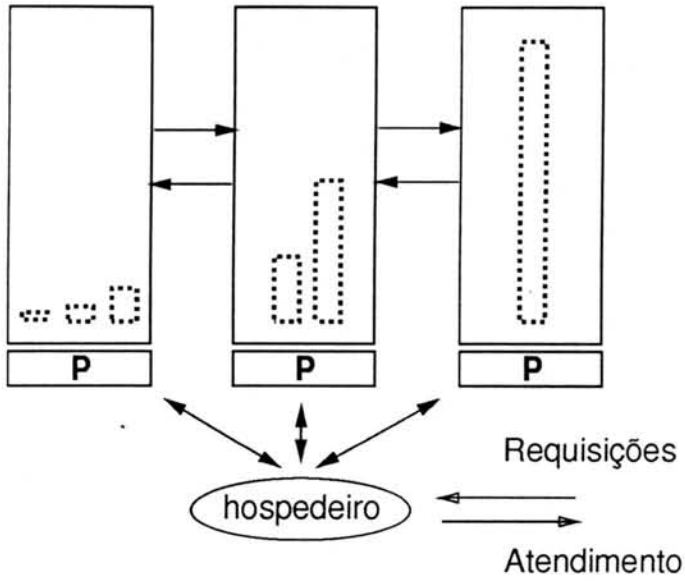


Figura 4.6: Modelo da versão paralela otimizada do algoritmo FL

Uma forma de diminuir este pico de tráfego entre processadores durante a fase de desalocação consiste no agrupamento de várias listas em um mesmo processador (figura 4.6). Desta forma, muitas das mensagens que seriam trocadas entre listas para sua atualização, são transformadas em processamento local.

Um balanceamento de carga estático pode ser feito com o agrupamento das listas de forma desigual entre os processadores. Como o tamanho máximo das listas varia de 1 até 2^k, sendo k a dimensão do hipercubo compartilhado, listas de menor tamanho (maior dimensão) podem ser agrupadas em maior número, enquanto as listas de menor dimensão (maior tamanho) ficam sob responsabilidade de um só processador.

4.2 Recursos utilizados na paralelização

A maioria das máquinas hipercúbicas encontradas no mercado são ligadas como máquinas agregadas em uma estação hospedeira. Esta estação hospe-

deira por sua vez está normalmente ligada a uma rede de estações, permitindo que a máquina multiprocessadora seja compartilhada por diversos usuários, evitando assim que seja subutilizada. Como os algoritmos de alocação e gerência rodam na máquina hospedeira, a paralelização pode ser feita tanto com processadores das estações vizinhas, como com processadores da máquina multiprocessadora compartilhada.

Os algoritmos propostos na seção 4 foram implementados com recursos da rede de estações e em uma placa com processadores Transputer. Ambos os ambientes são descritos nas seções abaixo.

4.2.1 Rede de estações

Uma rede de estações de trabalho pode ser vista, a nível funcional, como uma máquina multiprocessadora fracamente acoplada em grande escala. Os elementos processadores desta máquina seriam as estações, cada uma com seu processador e sua memória local, e a forma de interconexão entre eles seria um barramento, respeitando o protocolo ethernet [TAN 88] (figura 4.7).

O procedimento de alocação de processadores, normalmente executado apenas pela estação hospedeira da máquina Hipercúbica, pode ser distribuído entre as outras estações da rede. Desta forma pode se obter uma melhora no tempo de resposta graças a cooperação das diversas estações na execução deste procedimento.

Os algoritmos propostos na seção 4 não necessitam de nenhuma adaptação para execução neste ambiente. A distribuição dos processos é feita como indicado, ficando cada processo em uma estação. O processo mestre deve ficar na estação hospedeira.

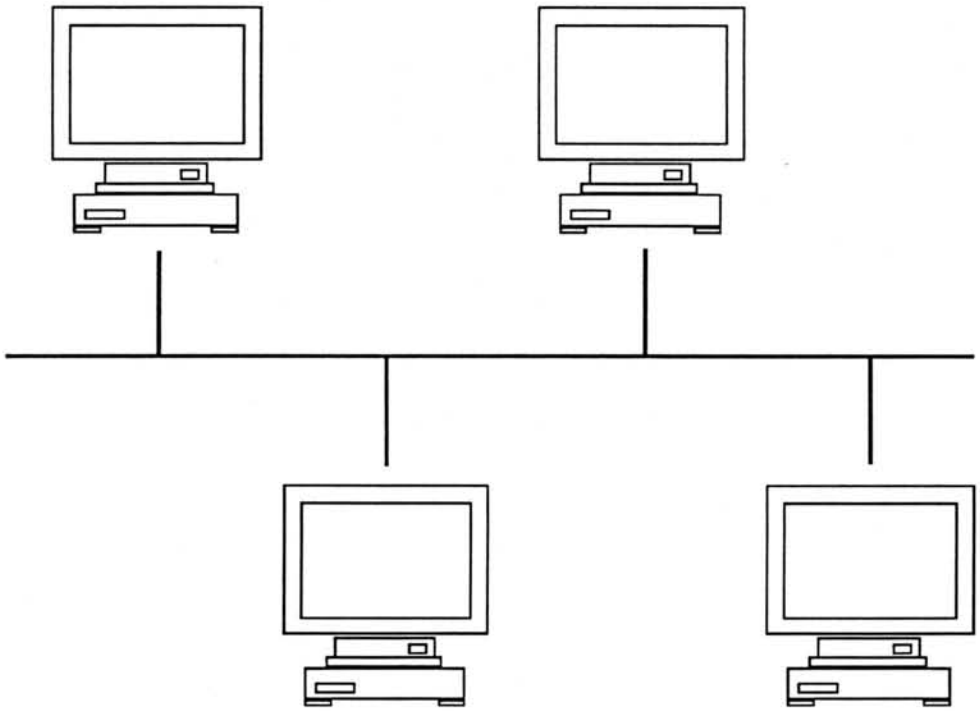


Figura 4.7: Rede de estações de trabalho

As diretivas de comunicação entre os processos são implementadas através do mecanismo de *sockets*. Este mecanismo implementa um canal bufferizado e bloqueante para leitura, entre dois processos da rede, independente da máquina onde estão executando [SUN 90]. Este tipo de canal se adapta perfeitamente ao conceito de diretiva de comunicação para o qual os algoritmos paralelos foram desenvolvidos.

Apesar de que processos localizados na mesma estação possam ser sincronizados através do mecanismo de semáforos, este mecanismo não está disponível para processos em diferentes estações. A solução encontrada foi a emulação do mecanismo de semáforos através de troca de mensagens. Isto é feito com a criação de um processo adicional que executa as funções de um semáforo para o ambiente distribuído, e é ativado por mensagens. Este processo foi denominado coordenador.

A figura 4.8 apresenta um diagrama da estrutura de comunicação utilizada nas implementações distribuídas dos algoritmos de alocação. O trabalho a

ser feito é distribuído pelo mestre entre os escravos, que após concluirem o trabalho retornam, ou o resultado da operação, ou seu estado. Nas versões em que é feita uma atualização da lista de alocação por parte dos escravos, estes devem consultar o processo coordenador, pois esta alteração só pode ser feita uma vez.

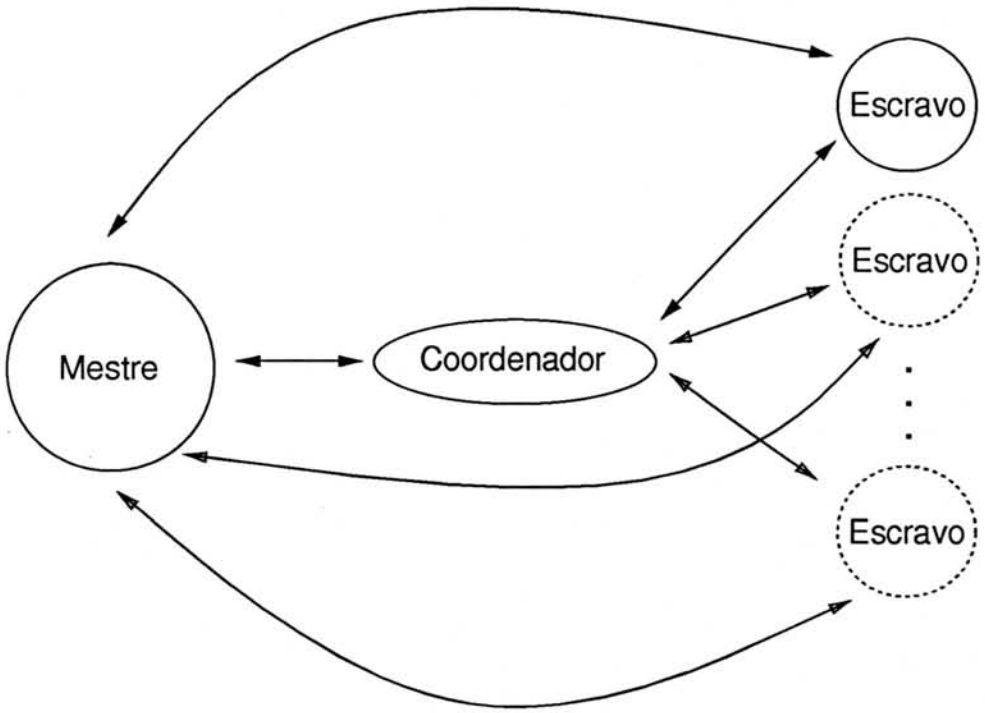


Figura 4.8: Diagrama da estrutura de comunicação utilizada nas versões distribuídas

As listagens dos algoritmos implementados para este ambiente podem ser encontradas no anexo A-2.1 e uma avaliação do seu desempemho na seção 5.2.1.

No entanto, é importante ressaltar que a principal diferença entre a rede de estações e uma máquina multiprocessadora, abstraindo-se a área física ocupada, está na velocidade da comunicação entre seus nodos. Em uma rede de estações o custo com comunicação é aproximadamente 25 vezes maior do que em uma máquina multiprocessadora. As medições do tempo de envio e recebimento de um caracter efetuadas nos dois ambientes indicaram um custo médio de 0.305

centésimos de segundo para a rede e 0.012 centésimos de segundo para a placa Transputer.

Como a comunicação entre os nodos é um fator que pesa muito no desempenho final de um programa paralelo, fica mais difícil a obtenção de desempenho na rede de estações, apesar de ambos os ambientes possuírem processadores com desempenho equivalente.

4.2.2 Placa com processadores Transputer

Uma segunda opção para a paralelização dos algoritmos de alocação é a utilização de processadores da máquina compartilhada. Desta forma, o processo mestre executa na estação hospedeira enquanto os escravos são carregados em processadores da máquina multiprocessadora agregada a esta estação. Para avaliar o desempenho resultante neste tipo de configuração, os algoritmos propostos foram implementados em uma placa com Transputers e o código resultante pode ser encontrado no anexo A-2.2.

Um Transputer é um dispositivo VLSI com memória, processador e links de comunicação para conexão direta com outros Transputers. Sistemas concorrentes podem ser construídos com uma coleção de Transputers que operam concorrentemente e comunicam-se através de seus links.

Uma máquina composta por Transputers é definida como uma arquitetura multiprocessadora (MIMD), fracamente acoplada (memória distribuída).

Existem no mercado Transputers de 16 bits (IMS T212 e IMS T222) e Transputers de 32 bits (IMS T414, IMS T425 e IMS T800). O IMS T800 possui internamente um processador de ponto flutuante de alta velocidade (FPU).

O Transputer apresenta algumas características que o tornam mais interessante em relação aos processadores convencionais. Uma delas é o aumento de velocidade conseguido com o uso da memória integrada no chip, junto com a CPU e a FPU (Floating Point Unit). Isso torna a execução de um programa muito mais rápida, pois diminui o tempo de acesso à memória para busca de instruções e operandos. Outro aumento de velocidade é conseguido com o uso dos links, pois dessa maneira os Transputers são interligados ponto-a-ponto, acabando com o problema causado pelo compartilhamento de fios.

As unidades internas dos Transputers são assimétricas, assim pode-se realizar uma operação em ponto flutuante na FPU em paralelo com uma comunicação através dos links e um processamento qualquer da CPU. Existe ainda um scheduler em microcódigo que compartilha o tempo do processador entre os diversos processos concorrentes.

O ambiente com processadores Transputer utilizado para avaliação dos algoritmos propostos foi a placa IMS B008 da INMOS [INM 90], que se utiliza de 4 processadores T800 (figura 4.9).

O T800 é um Transputer de 32 bits, com uma FPU de 64 bits (padrão IEEE 754), uma RAM estática de 4 Kbytes. Possui 4 links seriais de até 20 Mbits/segundo. Possui também instruções de movimento de blocos provendo alto desempenho no tratamento de gráficos. A figura 4.10 apresenta um diagrama de blocos deste processador.

Para a implementação dos algoritmos neste ambiente foi utilizada a linguagem "C" Paralela 3L da Inmos [INM 89]. As versões paralelas inplementadas na rede de estações foram facilmente portadas para esta linguagem devido as semelhanças encontradas nas diretivas básicas de comunicação.

Como não existe o mescanismo de semáforos disponível entre os processadores deste ambiente, a solução encontrada foi a emulação deste mecanismo

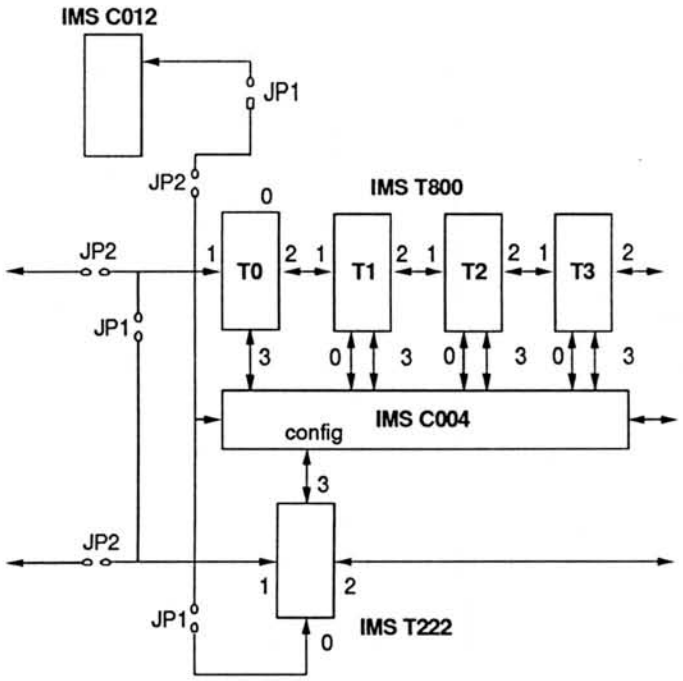


Figura 4.9: Diagrama de blocos da placa IMS B008 com processadores Transputer

através de troca de mensagens. Este procedimento é implementado por meio de um processo adicional que controla o semáforo, de forma análoga a solução utilizada na rede de estações.

Os resultados obtidos com a implementação das versões paralelas dos algoritmos de alocação na placa IMS B008, baseada em Transputers, podem ser encontrada na seção 5.2.2 e as respectivas listagens no anexo A-2.2.

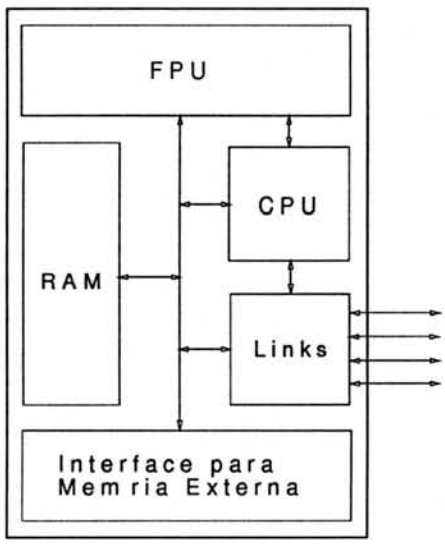


Figura 4.10: Diagrama de blocos do Transputer T800

5 AVALIAÇÃO DOS RESULTADOS OBTIDOS

Neste capítulo serão apresentados os resultados obtidos com a implementação das versões paralelas dos algoritmos de gerência e alocação de processadores para máquinas multiprocessadoras hipercúbicas.

Uma análise do funcionamento das versões paralelas é feita com o objetivo de compreender melhor os resultados obtidos e de identificar as dificuldades resultantes da paralelização.

5.1 O modelo de avaliação dos algoritmos

O modelo de simulação utilizado para comparação dos algoritmos de alocação encontrados na literatura e suas versões paralelas baseia-se em um gerador de requisições e um executor de requisições, que atende às mesmas através da execução dos algoritmos de alocação e gerência de processadores.

O gerador de requisições aceita parâmetros como tempo de simulação, número de clientes, tempo médio de alocação e número médio de processadores a serem requisitados. A partir dos dados de entrada são gerados registros com o número do pedido, tamanho do subcubo desejado, tempo de alocação , e tempo de desalocação (figura 5.1(a)).

O executor de requisições lê os registros gerados e atende os pedidos de acordo com a capacidade do algoritmo em teste, enfileirando as requisições que não possam ser atendidas. A política de escalonamento da fila de espera é FCFS. Durante a simulação, o executor de requisições monitora o funcionamento do algoritmo em termos do número de pedidos negados, tempo médio de espera na fila, tempo total da simulação, e taxa de utilização do hipercubo compartilhado

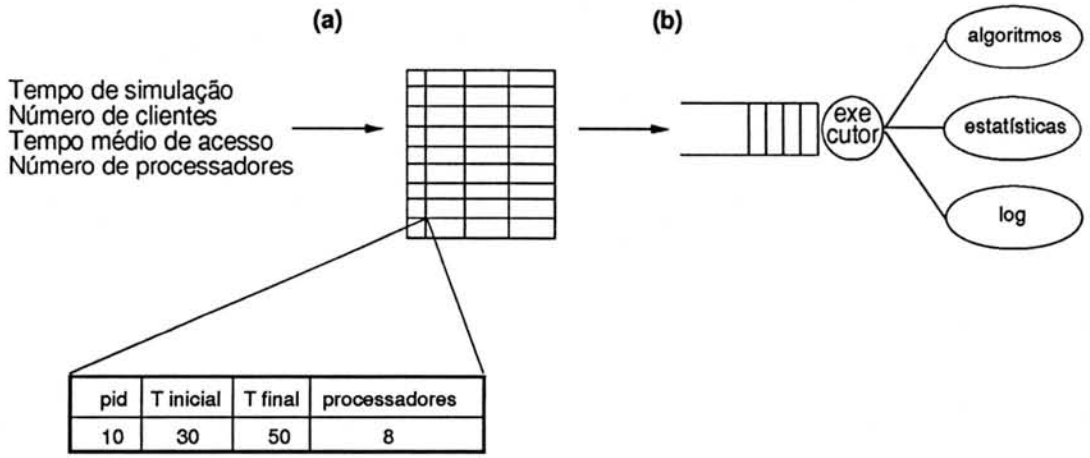


Figura 5.1: Modelo de simulação utilizado para comparação dos algoritmos

(5.1(b)). Além dos dados estatísticos também é gerado um arquivo de log, com o registro de todos os eventos que ocorreram durante a simulação.

Para fins de comparação foram implementadas versões seqüenciais dos algoritmos descritos no capítulo 2 e as versões paralelas propostas no capítulo 4.

5.2 Os resultados obtidos

Nas tabelas abaixo são apresentados os resultados obtidos pelos algoritmos de alocação e gerência de processadores implementados neste trabalho, nos diferentes ambientes de execução.

5.2.1 Na rede de estações

A tabela 5.1 apresenta uma comparação dos resultados obtidos pelas versões paralelas e as versões seqüenciais dos algoritmos de gerência e alocação

de processadores para hipercubos compartilhados de grau 8, 9, 10, 11, 12, 13 e 14, utilizando 2 estações de trabalho nas versões paralelas.

Os valores relativos a W_a e W_r referem-se ao tempo gasto pela pior alocação (*Worst Allocation*) e pior desalocação (*Worst Release*) respectivamente. Os valores relativos a tr referem-se ao tempo médio gasto com o atendimento de uma requisição.

Tabela 5.1: Resultados obtidos pelos algoritmos implementados na rede de estações (em centésimo de segundo)

	proc.	256	512	1024	2048	4096	8192	16384
BUDDY/GC	W_a	0.46	0.97	2.15	4.5	11.8	19.5	42.07
	W_r	0.05	0.09	0.16	0.32	0.63	1.26	2.69
	tr	0.06	0.08	0.22	0.55	1.17	2.16	3.84
BUDDY/GC ||	W_a	0.30	0.43	0.68	1.10	2.43	4.35	6.55
	W_r	0.12	0.17	0.27	0.41	0.72	1.36	2.64
	tr	3.36	3.41	3.49	3.61	3.69	3.95	4.89
MGC_{100}	W_a	76.97	390	3700	Falta de Memória			
	W_r	4.9	14	127.3				
	tr	50.35	167.6	3091				
MGC_{100} ||	W_a	40.51	123.8	1761	Falta de Memória			
	W_r	2.57	4.44	57.87				
	tr	26.5	89.5	1618				
TC	W_a	62.35	70.55	86.06	112.9	140	189	221.1
	W_r	0.05	0.09	0.16	0.32	0.63	1.26	2.63
	tr	9.25	13.15	19.86	26.59	32.88	47.68	54.83
TC ||	W_a	50.68	58.28	69.93	73.43	91.79	123.9	154.9
	W_r	0.12	0.17	0.27	0.41	0.72	1.36	2.64
	tr	10.02	11.23	13.08	15.25	17.86	27.02	30.8
FL	W_a	3.54	7.28	14.83	29.35	58.6	115.2	245.6
	W_r	11.5	25.65	75.95	195.2	501.6	1289.32	3314
	tr	9.75	24.87	58.46	149	365.2	858.3	2017

Para as implementações na rede de estações foi utilizada a linguagem "C" (SUN Microsystems cc) e os resultados são fornecidos em centésimos de segundo. O desempenho dos algoritmos foi medido utilizando-se uma estação

SUN Sparc Station 2 IPC como mestre e uma estação Sparc Station SLC *diskless* como escrava.

5.2.2 Na placa IMS B008

A tabela 5.2 apresenta uma comparação dos resultados obtidos pelas versões paralelas e as versões seqüenciais dos algoritmos de gerência e alocação numa placa com Transputers. As medições foram feitas para hipercubos compartilhados de grau 8, 9, 10, 11, 12, utilizando 2 processadores nas versões paralelas.

Os valores relativos a tr referem-se ao tempo médio gasto com o atendimento de uma requisição.

Tabela 5.2: Resultados obtidos pelos algoritmos implementados em Transputers (em centésimos de segundo)

	proc.	256	512	1024	2048	4096
BUDDY/GC	W_r	0.3	0.4	1.09	2.73	5.92
BUDDY/GC ||	W_r	1.95	2.2	2.6	3.2	3.6
MGC_{100}	W_r	249.2	855	14837	Falta de Memória	
MGC_{100} ||	W_r	146.4	518.1	9158	Falta de Memória	
TC	W_r	44.4	61.8	95.32	132.95	163.08
TC ||	W_r	23.36	32.97	53.8	76.5	94.43
FL	W_r	45.82	126.83	298.14	707.75	1811.13

Nas implementações para a placa IMS B008 com processadores Transputer foi utilizada a linguagem "C" paralela 3L [INM 89] e os resultados fornecidos são em centésimos de segundo. O desempenho dos algoritmos foi medido com um IBM PC-AT 386 como hospedeiro de uma placa equipada com 4 processadores Transputer T800.

5.3 Avaliação do desempenho dos algoritmos

Uma avaliação do desempenho das versões paralelas implementadas é apresentada nas seções abaixo, com o objetivo de facilitar a interpretação dos resultados obtidos.

5.3.1 Buddy/GC Paralelo

A fraca dependência entre as operações executadas nas listas de alocação distribuídas entre os processadores envolvidos mantém o fluxo de mensagem entre os processadores baixo e permite uma distribuição de trabalho simples e balanceada.

A operação de alocação tem um desempenho superior a desalocação devido a sua maior carga de trabalho (figura 5.2). Mesmo assim, em ambas as operações o trabalho a ser feito é muito pequeno para justificar os custos de comunicação quando do envolvimento de outro processador na operação.

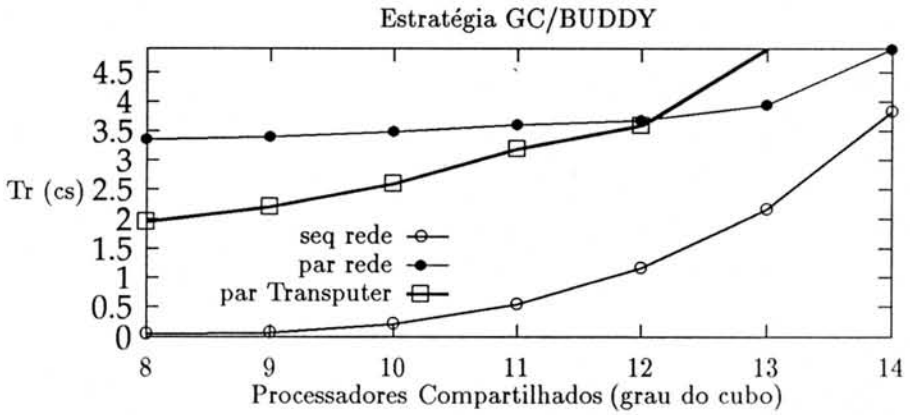


Figura 5.2: Resultados obtidos com a paralelização do algoritmo BUDDY/GC com o tempo de resposta em centésimos de segundo

O fluxo de mensagens durante a execução do algoritmo é muito baixo devido ao fato de só serem usadas mensagens para a distribuição do trabalho a ser feito, teste de acesso a seção crítica, e devolução dos resultados. As mensagens são de tamanho pequeno, variando de um a três bytes.

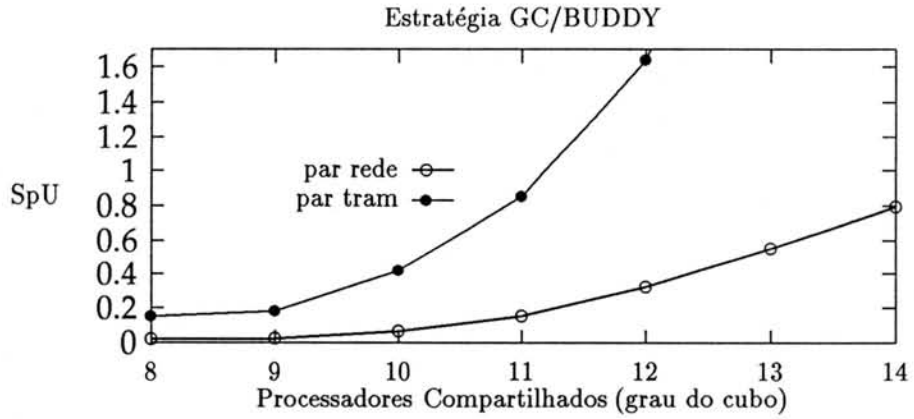


Figura 5.3: Resultados obtidos com a paralelização do algoritmo BUDDY/GC (speed–Up)

O desempenho do algoritmo depende muito da capacidade do processador mestre em detectar a necessidade ou não do envolvimento de mais processadores na operação requisitada. Com as diretivas propostas na seção 4.1.1.2 o algoritmo paralelo atingiu um *speed-up* de 0.79 em relação a versão seqüencial, com 2 estações de trabalho gerenciando um hipercubo de grau 14 (figura 5.3). O resultado na placa Transputer foi bem melhor, devido ao menor tempo consumido na operação de troca de mensagens. O algoritmo paralelo atingiu um *speed-up* de 1.64 com 2 processadores gerenciando um hipercubo de grau 12 (figura 5.3).

O problema da seção crítica 4.1.1.1 dificultou a obtenção de melhores resultados devido a necessidade de sincronização dos processos concorrentes e da geração de mensagens adicionais.

A diferença de desempenho dos algoritmos BUDDY e GC é mínima e não justificou um estudo separado. Os algoritmos apresentam características

de funcionamento muito semelhantes, possuindo diferenças apenas a nível de resultado (qualidade da alocação).

5.3.2 MGC Paralelo

A principal diferença no funcionamento da versão paralela do algoritmo BUDDY/GC e o algoritmo MGC é que no segundo caso a lista de alocação não é partilhada entre os processadores envolvidos. Cada processador recebe uma cópia desta lista, que após cada operação tem que ser atualizada. Esta atualização tem um custo significativo, pois aumenta o tempo de comunicação, principalmente quando são alocados subcubos de grande dimensão.

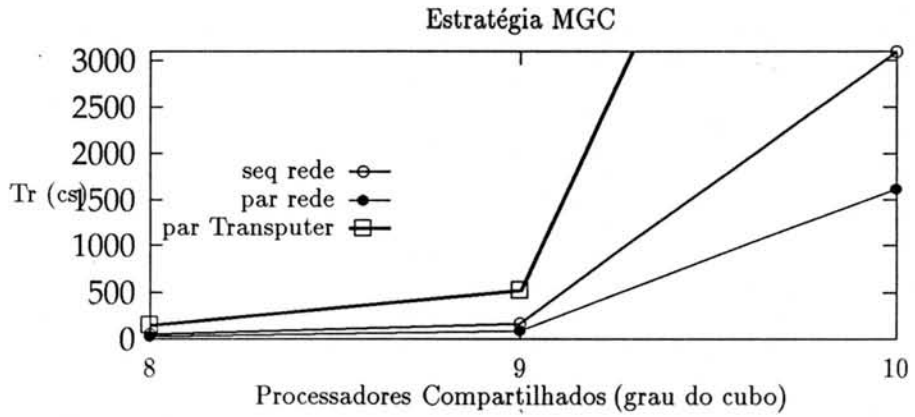


Figura 5.4: Resultados obtidos com a paralelização do algoritmo MGC com o tempo de resposta em centésimos de segundo

Por outro lado, cada nodo possui uma ou mais listas de códigos de gray para processar, o que aumenta consideravelmente a carga computacional de cada nodo, principalmente em hipercubos compartilhados de grande dimensão. Com este aumento de carga de trabalho a paralelização deste procedimento obtém melhores resultados do que a versão paralela do algoritmo BUDDY/GC (figura 5.4).

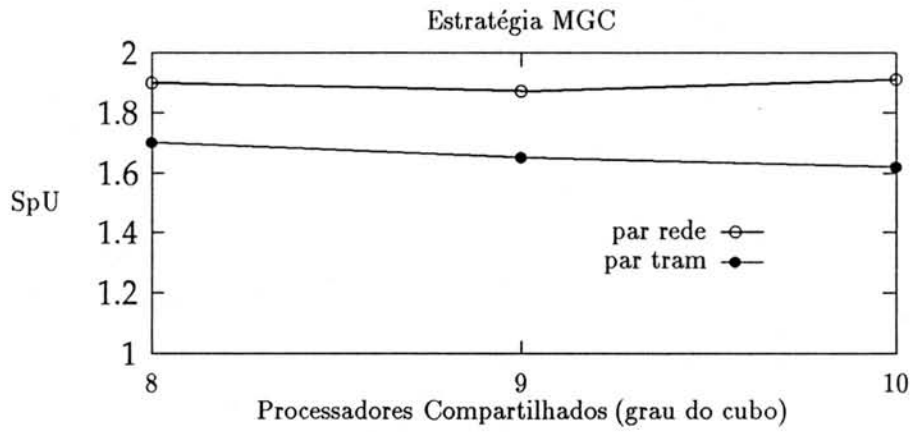


Figura 5.5: Resultados obtidos com a paralelização do algoritmo MGC (Speed–Up)

Foi obtido um *speed-up* de 1.89 com o algoritmo paralelo de gerência de um subcubo de grau 10 na rede de estações em relação ao algoritmo seqüencial. Na placa Transputer o algoritmo paralelo atingiu um *speed-up* de 1.65 em relação ao algoritmo seqüencial, com 2 processadores gerenciando um hipercubo de grau 12 (figura 5.5).

O desempenho da versão paralela do algoritmo MGC foi pior na placa com processadores Transputer devido a velocidade do processador T800, que é aproximadamente 5 vezes mais lento do que o processador SPARC da estação de trabalho utilizada. A a maior velocidade de comunicação entre os processadores da placa Transputer não foi suficiente para superar a defasagem de seu processador, devido a alta carga de trabalho imposta pelo algoritmo MGC.

O desempenho do algoritmo MGC na placa Transputer melhoraria consideravelmente, provavelmente superando até o desempenho nas estações de trabalho, se a placa b008 fosse equipada com um processador mais rápido, como por exemplo o T900.

5.3.3 Tree Collapse Paralelo

A operação de *Collapse*, sobre a qual se baseia o algortimo TC, tem uma alta complexidade e é chamada várias vezes durante uma operação de alocação, o que resulta em bons resultados quando este procedimento é paralelizado.

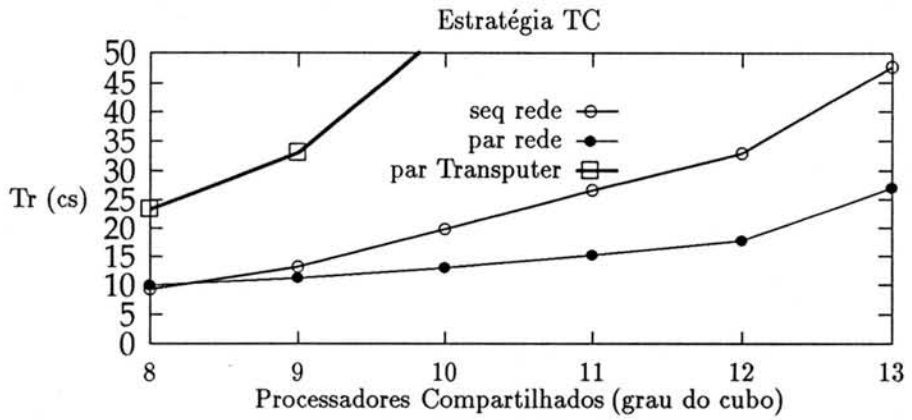


Figura 5.6: Resultados obtidos com a paralelização do algoritmo TC com o tempo de resposta em centésimos de segundo

A estrutura de dados que controla as alocações, porém, é copiada para cada processador, gerando mensagens grandes em hipercubos de alta dimensão. Em contrapartida, a atualização é feita apenas no processador mestre, o que torna o tratamento da seção crítica desnecessário neste algoritmo.

O principal problema do algoritmo é a grande variação da carga de trabalho de uma requisição de alocação. A capacidade do algoritmo paralelo de detectar quando a ativação de um nodo remoto é justificada pela carga de trabalho a ser feita, é fundamental para o bom desempenho da versão paralela (figura 5.6).

Na rede de estações o algoritmo paralelo obteve um *speed-up* de 1.63 em relação ao algoritmo seqüencial, na gerência de um subcubo de grau 12 com duas estações de trabalho. Na placa Transputer o algoritmo paralelo atingiu um

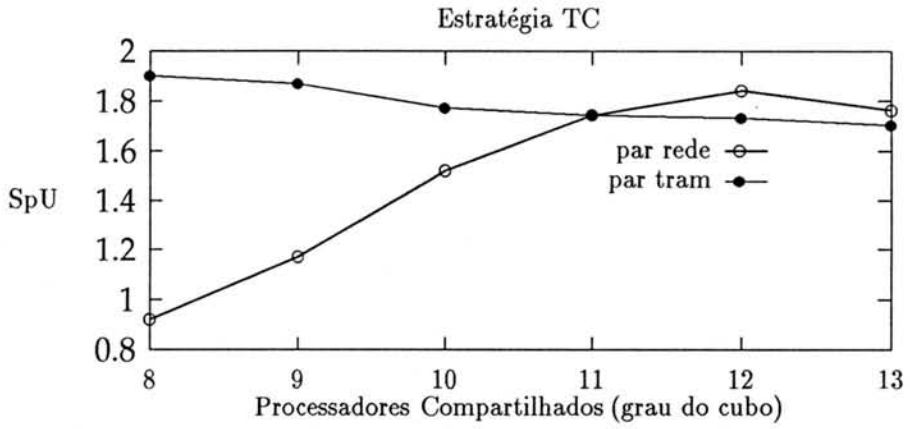


Figura 5.7: Resultados obtidos com a paralelização do algoritmo TC (Speed–Up)

speed-up de 1.79 com 2 processadores gerenciando um hipercubo de grau 12 (figura 5.7).

5.3.4 Free List Paralelo

Na tabela 5.1 não foram colocados os resultados obtidos pela versão paralela do algoritmo FL. Isto decorre do fato desta versão, proposta por Kim e Das em [KIM 91] ser muito complexa e gerar um tráfego de mensagens muito alto que não é suportado de forma alguma pelo ambiente de compartilhamento proposto. Nem mesmo as otimizações propostas na seção 3.4 tornaram esta versão paralela viável neste tipo de ambiente.

Este péssimo desempenho da versão paralela resulta da dificuldade de se paralelizar o algoritmo FL de forma eficiente. O algoritmo opera sobre uma estrutura de múltiplas listas que é tratada de forma seqüencial, existindo uma forte dependência entre as operações [KIM 91].

5.3.5 Comparação entre as versões paralelas

A figura 5.8 apresenta uma comparação do desempenho obtido pelas versões paralelas propostas e implementadas neste trabalho.

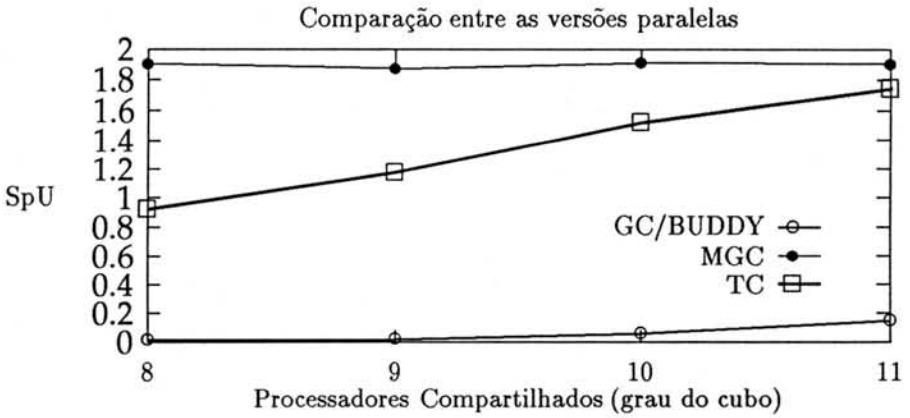


Figura 5.8: Comparação entre as versões paralelas implementadas na rede de estações (Speed–Up)

Os resultados apresentados na figura 5.8 são referentes as implementações paralelas dos algoritmos de alocação na rede de estações. É importante ressaltar que as versões paralelas que obtiveram melhor *speed-up* foram a MGC e TC, com 1.89 e 1.63 respectivamente (média para todas as dimensões medidas). Já com a versão paralela do algoritmo BUDDY/GC não foi possível superar o desempenho da versão seqüencial e o *speed-up* ficou em 0.79 (hipercubo de grau 14).

A decisão de qual versão paralela dos algortimos de alocação de processadores deve ser utilizada em um determinado sistema não deve ser tomada apenas analisando seu desempenho (figura 5.8). Como as propriedades de cada algoritmo foram mantidas no procedimento de paralelização a escolha deve ser tomada sobre a qualidade dos resultados fornecidos e características dos algoritmos seqüenciais. Após a escolha de um algoritmo de alocação que se enquadre nas necessidades do sistema em questão, deve ser utilizada na implementação a respectiva versão paralela para obtenção de um melhor desempenho.

6 APLICANDO OS ALGORITMOS DESENVOLVIDOS

Nesta seção são apresentadas aplicações que se baseiam nos algoritmos de alocação e gerência de processadores estudados e otimizados neste trabalho.

Primeiramente é apresentada a estrutura de um servidor de processadores para uma rede de estações de trabalho e descritos os serviços por ele fornecidos. Este servidor se utiliza dos algoritmos estudados neste trabalho para alocar e gerenciar os processadores de uma máquina paralela entre as diversas estações da rede.

A segunda aplicação apresentada é um simulador que se utiliza dos algoritmos de alocação para analisar o funcionamento do servidor de processadores em uma rede de estações que segue o protocolo ethernet.

6.1 Um servidor de processadores para uma rede de estações

O compartilhamento de uma máquina paralela em uma rede de estações é uma tendência que vem se acentuando nos últimos anos [TRI 90] [RIZ 92].

O objetivo deste compartilhamento é reduzir o custo de uma máquina paralela através de sua melhor utilização e aumentar o desempenho de redes de estações de trabalho a nível de supercomputadores.

Para que as estações de uma rede possam alocar processadores da máquina paralela para execução de seus processos é necessário que esta máquina seja conectada à rede através de uma estação hospedeira, e que nesta estação seja instalado um servidor de processadores.

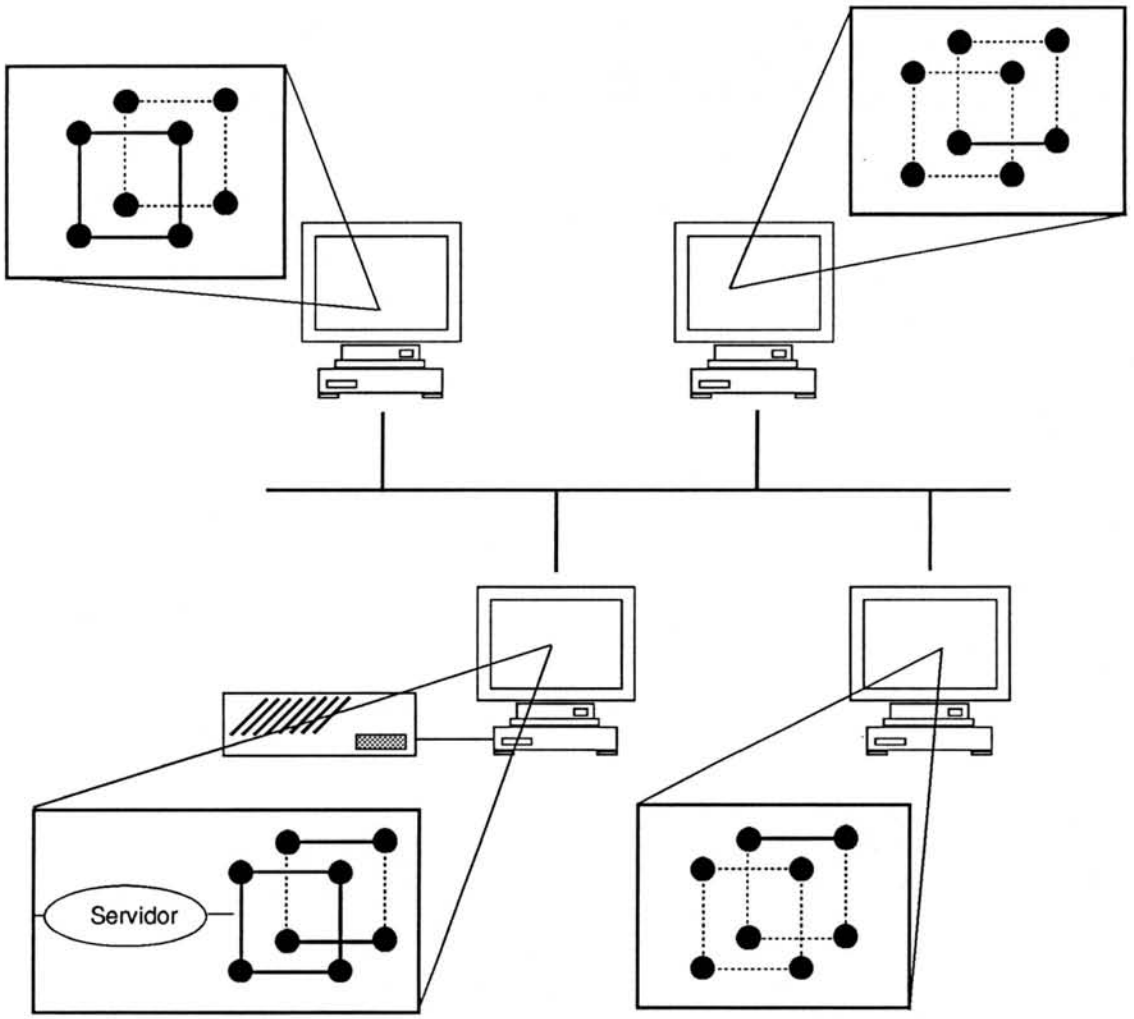


Figura 6.1: O ambiente de compartilhamento de uma máquina multiprocessadora

A figura 6.1 mostra como fica o ambiente de compartilhamento dos processadores de uma máquina paralela e a forma como a máquina é ligada a rede.

6.1.1 Os serviços básicos do servidor

Através da definição dos serviços básicos fornecidos pelo servidor de processadores pode se ter uma idéia mais clara da sua função no sistema:

Alocação e Gerência dos processadores pode ser considerado o núcleo do servidor. Executa o controle dos processadores da máquina compartilhada, identificando processadores livres para alocação, efetuando a reserva destes processadores para uma estação, e liberando estes processadores após o uso. São implementados pelos algoritmos de alocação e gerência estudados neste trabalho;

Gerenciamento dos pedidos efetua o tratamento dos pedidos para o servidor quando estes não puderem ser atendidos imediatamente (Ex: número de processadores desejados não se encontra disponível). Pode implementar uma fila de espera ou retornar o pedido com uma indicação para que seja refeito mais tarde;

Carga do código efetua a carga do código de um processo em um determinado processador já previamente alocado. Verifica se a estação que enviou o código executável do processo pode acessar o processador desejado;

Mapeamento de topologias efetua o mapeamento dos endereços físicos dos processadores da máquina compartilhada para os endereços lógicos utilizados pelas estações para acesso a estes processadores. Efetua também o mapeamento dos endereços físicos dos canais de comunicação dos processadores alocados pelas estações, em endereços lógicos que implementem diferentes topologias, como linear, anel, malha, etc., sempre que a rede de interconexão dos processadores permitir;

Interface com a rede é a camada mais externa do servidor. É responsável pela comunicação entre o servidor e as estações da rede. Implementa o recebimento de chamadas ao servidor e a ativação das funções para atendimento destas chamadas. Efetua o redirecionamento dos resultados fornecidos pelos processos alocados nos processadores da máquina compartilhada para as respectivas estações.

A figura 6.2 apresenta a estrutura do servidor com a identificação de suas camadas de serviços.

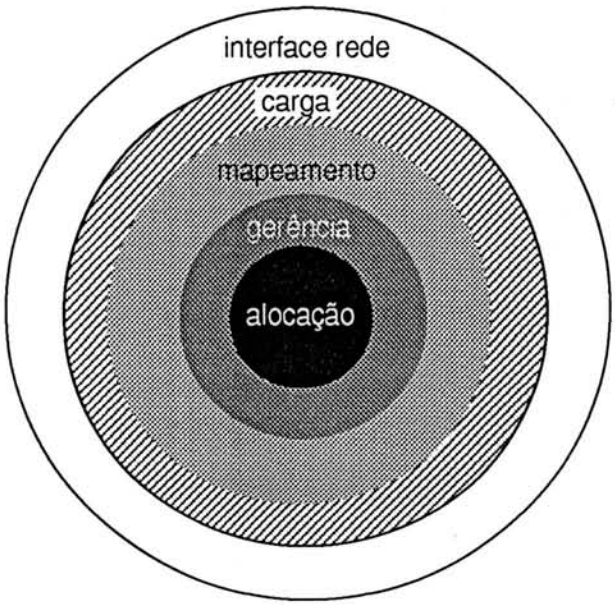


Figura 6.2: Estrutura do servidor com suas camadas de serviços

6.1.2 Alguns serviços adicionais

Existem alguns serviços que, apesar de não poderem ser considerados essenciais para o funcionamento do servidor, melhoram a qualidade dos resultados fornecidos. Outros serviços são interessantes para o gerente da rede, pois fornecem informações sobre o funcionamento do sistema e permitem que se interfira na escolha de algumas políticas de funcionamento do servidor.

Alguns destes serviços adicionais estão listados abaixo:

Gerência de processadores livres controla uma lista de processadores livres na máquina compartilhada e processadores que foram alocados mas não estão sendo usados (fragmentação interna). Estes processadores podem ser usados na paralelização de alguma função do servidor;

Tolerância a falhas identifica falhas nos processadores e interconexões da máquina compartilhada. Estes elementos são marcados como falhos e não são mais compartilhados até que sejam substituídos ou voltem a funcionar;

Monitoração do funcionamento cataloga todas as operações feitas pelo servidor em um arquivo de log, juntamente com o tempo e um resumo de seu estado atual (variáveis globais, elementos falhos, fila de espera, etc.);

Geração de estatísticas gera estatísticas sobre o funcionamento do sistema, como número de pedidos negados pelo servidor, taxa de utilização da máquina compartilhada, tempo médio de espera de uma requisição ao servidor, etc;

Escolha dos mecanismos coloca a disposição do servidor diversos mecanismos com características diferentes para a resolução de um mesmo problema. Desta forma é possível uma adaptação do servidor de acordo com as características dos usuários da rede (obtida dos dois ítens anteriores). Esta adaptação pode ser manual, através de interferência do gerente da rede, ou automática. Um exemplo deste serviço seria a possibilidade de se optar por diferentes algoritmos de alocação e gerência de processadores, de acordo com as necessidades do sistema.

6.1.3 As necessidades a nível de hardware

O modelo de servidor descrito acima pode trabalhar com qualquer tipo de máquina paralela que permita sua conexão a uma máquina hospedeira, que por sua vez esteja ligada a uma rede de outras máquinas.

O número de processadores e a forma de interconexão entre eles não afeta diretamente o funcionamento do servidor, mas pode dar, ou não, maior flexibilidade ao sistema. A máquina hipercúbica, por exemplo, permite o mapeamento de várias topologias como linear, anel, árvore, malha e hipercubo. Uma máquina

baseada em Transputers, com alguns canais de comunicação fixos, diminuiria o número de possíveis topologias.

Características como tipo de conexão da máquina compartilhada a seu hospedeiro, velocidade de processamento da máquina hospedeira e quantidade de memória, tipo de protocolo utilizado na rede, e tipo de ligação entre as máquinas da rede não afetam o funcionamento do servidor, mas apenas a qualidade de seus resultados.

6.1.4 Efeitos causados pelo servidor no funcionamento da rede

Por se tratar de um servidor que está localizado na máquina hospedeira e atende requisições segundo modelo cliente-servidor, o servidor de processadores gera um tráfego de mensagens concentrado entre esta estação e o restante da rede. Dependendo da configuração e carga da rede, este acrécimo de mensagens pode causar um colapso da rede.

Apesar da maioria das mensagens entre servidor e clientes ser de tamanho reduzido e não comprometer o funcionamento da rede, existem dois tipos de mensagens que podem ser grandes, dependendo do tipo de processamento enviado para a máquina compartilhada. Estas mensagens são o código executável para carga nos processadores da máquina compartilhada, e o arquivo com os resultados de saída destes processos.

Pelo fato de funcionar exclusivamente na máquina hospedeira e consumir tempo de processamento e de I/O, o servidor afeta consideravelmente o desempenho desta estação. Seria interessante que esta máquina fosse mais poderosa que as restantes, ou dedicada exclusivamente ao servidor de processadores.

6.1.5 O protótipo Sub-Cube RPC

Um protótipo do servidor de processadores denominado Sub-Cube RPC foi implementado com o objetivo de validar algumas das idéias propostas neste trabalho e de estruturar a base do servidor para desenvolvimento futuro.

O servidor foi codificado na linguagem "C" para uma rede de estações de trabalho SUN com sistema operacional Unix V. O mecanismo RPC (Remote Procedure Calls) foi utilizado para implementação do modelo mestre-escravo utilizado pelo servidor.

Parte do código do protótipo Sub-Cube RPC pode ser encontrado no anexo A-3.

6.1.5.1 Os serviços implementados

Seguindo os objetivos citados acima foi dada uma maior atenção aos serviços básicos de interface de rede, gerenciamento dos pedidos, e alocação e gerência de processadores. Com estes serviços já é possível analisar o comportamento do servidor na rede e avaliar o custo que implicará para o sistema.

Os serviços de mapeamento de topologias e carga do código, que são os serviços básicos que restaram, foram considerados de menor importância nesta fase inicial de desenvolvimento. Isto porque ambos os serviços dependem fortemente do hardware da máquina a ser compartilhada e só podem ser testados com a existência da mesma.

Dos serviços adicionais foram implementados a escolha de mecanismos e a monitoração do funcionamento. Desta forma foi possível testar as diversas políticas de alocação e gerenciamento de processadores com o servidor, e mo-

nitorar, através de um arquivo de log, o funcionamento de todo o sistema de compartilhamento.

O protótipo do servidor de processadores apresentou grande facilidade de acesso aos serviços de compartilhamento de recursos da máquina paralela.

A interface de rede implementada mostrou-se robusta e o gerenciamento de pedidos foi codificado de forma modular para permitir uma maior flexibilidade na inclusão e alteração dos serviços prestados pelo servidor.

6.2 O Simulador da rede local proposta

O simulador da rede local de compartilhamento proposta é uma ferramenta que foi desenvolvida com o auxílio do grupo ADMP para permitir que pudesse ser feita uma avaliação mais precisa do custo de um servidor de processadores totalmente operacional na rede de estações.

Com esta ferramenta, que foi desenvolvida em "C" para o ambiente X-WIndows, é possível dimensionar a rede de compartilhamento através de diferentes configurações da rede, do servidor de processadores, e da máquina paralela compartilhada.

Uma descrição detalhada desta ferramenta, incluindo os resultados fornecidos, interface com o usuário , e principalmente a forma com que são aplicados os algoritmos estudados neste trabalho, pode ser encontrada em [TEO 94].

7 CONCLUSÕES

Neste trabalho é analisada a utilização de técnicas de paralelização com o objetivo de otimizar o desempenho dos algoritmos de alocação e gerência de processadores para máquinas hipercúbicas encontrados na literatura.

Foram propostas versões paralelas dos algoritmos BUDDY [PUR 70], Códigos de Gray (GC) e Múltiplos Códigos de Gray (MGC) [CHE 87] , e sugeridas melhorias nas versões paralelas dos algoritmos *Tree Collapsing* (TC) [CHU 90] e Lista de Cubos Livres (FL) [KIM 91], propostas por seus respectivos autores.

Tanto os algoritmos seqüenciais, como as versões paralelas propostas, foram implementadas e seu desempenho foi avaliado através da simulação das condições do ambiente alvo desejado.

Como aplicação para os algoritmos propostos é definida a estrutura e os serviços de um servidor de processadores para compartilhar uma máquina multiprocessadora em uma rede de estações de trabalho.

Um protótipo do servidor de processadores denominado Sub-Cube RPC foi implementado com o objetivo de validar algumas idéias propostas no trabalho e de servir como base para o desenvolvimento deste projeto.

Os resultados demonstram que é possível se obter uma melhora no desempenho dos algoritmos com sua paralelização, principalmente quando a máquina compartilhada é de dimensão elevada (grau do hipercubo acima de 10).

O protótipo do servidor de processadores apresentou uma grande facilidade de acesso aos serviços de compartilhamento de recursos da máquina paralela. A estrutura implementada, baseada no modelo RPC (*Remote Procedure Call*), é modular e bastante flexível, podendo ser facilmente aperfeiçoada e adaptada às necessidades de outro ambiente de compartilhamento de processadores.

Dentre as dificuldades encontradas durante o processo de paralelização dos algoritmos de alocação, o alto custo da comunicação entre os processos paralelos é sem dúvida a mais significativa. Apesar de, na maioria dos casos, não existir grande dependência entre as operações que foram paralelizadas, a comunicação entre os processos é utilizada para distribuir as tarefas entre os processadores e para a troca de sinais de sincronização.

Sendo assim, a melhor opção para a questão de onde paralelizar os algoritmos de alocação é a utilização de processadores da máquina paralela. Nestes processadores, o custo com a comunicação é quase um quinto do tempo de comunicação entre as estações da rede. Esta diferença afeta diretamente o tempo de resposta dos algoritmos, resultando em um desempenho muito superior quando implementados nos processadores da máquina paralela.

Os resultados obtidos sustentam as afirmações acima e indicam que as versões paralelas dos algoritmos de alocação só se justificam a partir de hipercubos de grau 12 para implementações na rede de estações. Já para implementações nos processadores da máquina paralela são obtidos resultados satisfatórios para hipercubos a partir do grau 10.

Como o número de mensagens de distribuição de tarefas e de sincronização entre processos cresce à medida que aumentamos o número de processadores envolvidos na operação de paralelização, a maioria das versões paralelas propostas neste trabalho só é viável quando executadas por um pequeno número de processadores. Os resultados obtidos mostraram que na maioria dos algoritmos o melhor desempenho foi obtido com dois processadores.

A carga de trabalho de cada algoritmo é outro fator que afeta diretamente os resultados obtidos pelas versões paralelas. Os melhores resultados foram obtidos quando os algoritmos mais complexos são paralelizados em um pequeno número de processadores. Desta forma existe trabalho suficiente para

ser feito por vários processadores, valendo a pena distribuí-lo, mesmo com os altos custos de comunicação envolvidos.

Outro fator importante é que quanto mais alto o grau do hipercubo a ser compartilhado, mais custoso será o processo de alocação de processadores, aumentando assim a carga de trabalho a ser distribuído, e melhorando os resultados das versões paralelas.

Com as versões paralelas dos algoritmos mais complexos como MGC e TC foi possível melhorar o seu tempo de resposta, viabilizando estes algoritmos para sistemas compartilhados em tempo real.

A aplicação destes algoritmos neste tipo de sistema melhora a taxa de utilização da máquina multiprocessadora hipercúbica, permitindo um aproveitamento mais racional dos recursos compartilhados.

A figura 7.1 demonstra que foi possível, através da aplicação de paralelismo, amenizar a relação entre a qualidade dos resultados e o tempo de resposta dos algoritmos.

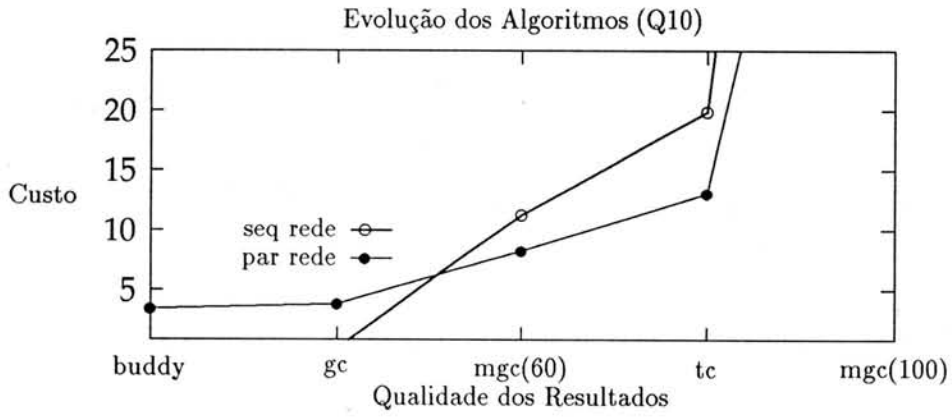


Figura 7.1: Relação dos fatores qualidade dos resultados e tempo de resposta para os algoritmos estudados

Com os resultados deste trabalho pode-se ter uma boa idéia dos efeitos e das dificuldades encontradas na paralelização dos algoritmos de alocação e gerência de processadores para máquinas Hipercúbicas. As informações contidas no trabalho auxiliam na melhoria do tempo de resposta dos algoritmos seqüenciais atuais e no desenvolvimento de novos algoritmos, específicos para execução em ambientes paralelos, com mais recursos e ainda assim viáveis em ambientes interativos graças a utilização de paralelismo.

O protótipo Sub-Cube RPC demonstra como os algoritmos estudados neste trabalho podem ser aplicados na construção de um servidor de processadores para máquinas multiprocessadas. O protótipo servirá como base para a implementação de um servidor semelhante no CPGCC/UFRGS, que colocará uma placa de Transputers a disposição da rede de estações do grupo de processamento paralelo.

A utilização de algoritmos avançados de alocação e gerência de processadores em máquinas multiprocessadoras de grau elevado em sistemas de tempo real, será apenas possível com a aplicação eficiente de técnicas de paralelização que reduzam consideravelmente a complexidade temporal destes algoritmos.

ANEXO A-1 LISTAGEM DOS ALGORITMOS SEQÜENCIAIS

As listagens neste anexo são implementações seqüenciais na linguagem "C" dos algoritmos para alocação e gerência de processadores em máquinas hipercúbicas descritos na seção 3.2.

A-1.1 buddy.c

```
/*********************************************/
/**   Rotinas para aloacacao e gerencia de   **/
/**  processadores em maquinas Hipercubicas  **/
/**       By Cesar De Rose - 29/05/92        **/
/**            Algoritmo: Buddy              **/
/*********************************************/

#define GRAU_CUBO 8
#define NUM_PROC  256

#define TRUE     1
#define FALSE    0

#include <stdio.h>

/* prototipes */

int  buddy_aloc   ();    /* int numero , int pid */
int  buddy_dealoc ();    /* int pid              */
void buddy_init   ();    /* void                 */
void buddy_mostra ();    /* void                 */

/* variaveis globais */

struct nodo_buddy {
       int processo;
       int alocado;
       };

struct nodo_buddy lista_buddy[NUM_PROC];

int p_aloc = 0; /* processadores ja alocados */

void main()
{
/* exemplo de execucao */

buddy_init();     /* inicializo estrutura de alocacao  */
buddy_aloc(8,5);  /* aloco 8 processadores com pid 5   */
buddy_mostra();   /* vejo estrutura de alocacao        */
buddy_dealoc(5);  /* desaloco pid 5                    */
buddy_mostra();   /* vejo estrutura de alocacao        */
}

void buddy_init()
{
/* inicializa lista buddy */

int i;

p_aloc = 0; /* todos os processadores livres */

for ( i = 0 ; i < NUM_PROC ; i++ )
    {
    lista_buddy[i].processo = 0;
    lista_buddy[i].alocado  = 0;
    }
}

void buddy_mostra()
{
/* mostra lista buddy */

int i;

printf("%d processadore(s) alocado(s).\n",p_aloc);

printf("\n         Lista Buddy\n\n");
for ( i = 0 ; i < NUM_PROC ; i++ )
    printf(" [%2d]   p: %2d    a:%2d \n",i,
    lista_buddy[i].processo,lista_buddy[i].alocado);
}

int  buddy_aloc   ( numero , pid )

int numero,
    pid;
{
/* aloco um hipercubo com "numero" processadores para */
/* este pid: retorno TRUE se a operacao obter sucesso */

int aloc_ok;
int bit_ok;
int i,t1,t2,m,dim, pot;

/* testo se existem processadores disponiveis */

if (numero > NUM_PROC-p_aloc) return FALSE;

/* encontro menor dimensao de cubo que contenha  */
/* "numero" processadores                        */

dim = 0;
t1 = 1;

while ( t1 < numero )
   {
   dim++;
   t1 *= 2;
   }

/* procuro menor "m" que satisfaca a equacao buddy */

m = 0;
aloc_ok = 0;

while (  ((((m+1)*t1)-1) < NUM_PROC) && !(aloc_ok) )
   {
   bit_ok = 1;
   t2 = m*t1;
   while ( (t2 <= (((m+1)*t1)-1)) && (bit_ok) )
      {
      if ( lista_buddy[t2].alocado != 0 )
         bit_ok = 0;
      t2++;
      }

   if ( bit_ok )
      aloc_ok = 1;
   else
      m++;
}

/* caso encontrado "m", aloco processadores */

if ( aloc_ok )
   for ( i = m*t1 ; i <= (((m+1)*t1)-1) ; i++ )
       {
       lista_buddy[i].alocado  = 1;
       lista_buddy[i].processo = pid;
       p_aloc += 1; /* mais um processador alocado */
       }

return(aloc_ok);
}

int  buddy_dealoc ( pid )

int pid;
{
/* libero processadores alocados pelo processo pid */
/* retorno numero de processadores desalocados     */

int i;
int pid_ok = 0;

for ( i = 0 ; i < NUM_PROC ; i++ )
    if ( lista_buddy[i].processo == pid )
       {
       lista_buddy[i].alocado  = 0;
       lista_buddy[i].processo = 0;
       p_aloc--;   /* processador liberado */
       pid_ok += 1;
       }

if ( pid_ok > 0)
   return(TRUE);
else
   return(FALSE);
}
```

A-1.2 gc.c

```
/*********************************************/
/**   Rotinas para aloacacao e gerencia de   **/
/**  processadores em maquinas Hipercubicas  **/
/**       By Cesar De Rose - 29/05/92        **/
/**            Algoritmo: GC                 **/
/*********************************************/

#define GRAU_CUBO 8
#define NUM_PROC  256
```

```
#define TRUE     1
#define FALSE    0

#include <stdio.h>

/* prototipes */

int  GC_aloc   ();    /* int numero , int pid */
int  GC_dealoc ();    /* int pid              */
void GC_init   ();    /* void                 */
void GC_mostra ();    /* void                 */

void GC_gera_brgc();          /* int * array */

/* variaveis globais */

struct nodo_GC {
       int node;
       int processo;
       int alocado;
       };

struct nodo_GC    lista_GC   [NUM_PROC];

int p_aloc = 0; /* processadores ja alocados */

void main()
{
/* exemplo de execucao */

GC_init();     /* inicializo estrutura de alocacao */
GC_aloc(8,5);  /* aloco 8 processadores com pid 5  */
GC_mostra();   /* vejo estrutura de alocacao       */
GC_dealoc(5);  /* desaloco pid 5                   */
GC_mostra();   /* vejo estrutura de alocacao       */
}

void GC_init()
{
/* inicializa lista GC */

int i;
int BRGC[NUM_PROC];

p_aloc = 0;
GC_gera_brgc(BRGC);

for ( i = 0 ; i < NUM_PROC ; i++ )
    {
    lista_GC[i].node     = BRGC[i];
    lista_GC[i].processo = 0;
    lista_GC[i].alocado  = 0;
    }
}

void GC_mostra()
{
/* mostra lista GC */

int i;

printf("%d processadore(s) alocado(s).\n",p_aloc);

printf("\n         Lista GC\n\n");
for ( i = 0 ; i < NUM_PROC ; i++ )
    printf(" [%2d]   p: %2d    a:%2d \n",lista_GC[i].node,
           lista_GC[i].processo,lista_GC[i].alocado);
}

int  GC_aloc   ( numero , pid )

int numero,
    pid;
{
/* aloco um hipercubo com "numero" processadores   */
/* o pid: retorno um numero positivo se a operacao */
/* obter sucesso                                   */

int aloc_ok;
int bit_ok;
int i,t2,m,dim, pot;
float t1;

/* testo se existem processadores disponiveis */

if (numero > NUM_PROC-p_aloc) return FALSE;

/* encontro menor dimensao de cubo que contenha */
/* "numero" processadores                       */

dim = 0;
t1 = 1;

while ( t1 < numero )
   {
   dim++;
   t1 *= 2;
   }

/* procuro menor "m" que satisfaca a equacao GC */

m = 0;
aloc_ok = 0;
t1/=2;

while (  ((((m+2)*t1)-1) < NUM_PROC) && !(aloc_ok) )
   {
   bit_ok = 1;
   t2 = m*t1;
   while ( (t2 <= (((m+2)*t1)-1)) && (bit_ok) )
      {
      if ( lista_GC[t2].alocado != 0 )
         bit_ok = 0;
      t2++;
      }

   if ( bit_ok )
      aloc_ok = 1;
   else
      m++;
}

/* caso encontrado "m", aloco processadores */

if ( aloc_ok )
   for ( i = m*t1 ; i <= (((m+2)*t1)-1) ; i++ )
       {
       lista_GC[i].alocado  = 1;
       lista_GC[i].processo = pid;
       p_aloc+=1;
       }

return(aloc_ok);
}

int  GC_dealoc ( pid )

int pid;
{
/* libero processadores alocados pelo processo pid */
/* retorno numero de processadores desalocados     */

int i;
int pid_ok = 0;

for ( i = 0 ; i < NUM_PROC ; i++ )
    if ( lista_GC[i].processo == pid )
       {
       lista_GC[i].alocado  = 0;
       lista_GC[i].processo = 0;
       p_aloc--;
       pid_ok += 1;
       }

if ( pid_ok > 0)
   return(TRUE);
else
   return(FALSE);
}

void GC_gera_brgc(array)

int * array;

{
int i,n;
int proc = NUM_PROC;
int ind_e,ind_l;

/* inicializo */

*(array+proc-2) = 0;
*(array+proc-1) = 1;
n = 2;

for ( i=1 ; i < GRAU_CUBO ; i++)
    {
   ind_l = proc-n;
   ind_e = proc - n *2;
   while( ind_l < proc )
      {
      if ( ind_l % 2 == 0 )
         {
         *(array+ind_e) = *(array+ind_l)<<1;
         ind_e++;
         *(array+ind_e) = (*(array+ind_l)<<1)+1;
```

```
        ind_e++;
        }
      else
        {
        *(array+ind_e) = (*(array+ind_l)<<1)+1;
        ind_e++;
        *(array+ind_e) = *(array+ind_l)<<1;
        ind_e++;
        }
      ind_l++;
      }
   n*=2;
    }
}
```

A-1.3 mgc.c

```
/***********************************************/
/**   Rotinas para aloacacao e gerencia de    **/
/**  processadores em maquinas Hipercubicas   **/
/**       By Cesar De Rose - 29/05/92         **/
/**            Algoritmo: MGC                 **/
/***********************************************/

#define GRAU_CUBO 4
#define NUM_PROC  16

#define TRUE      1
#define FALSE     0

#include <stdio.h>

/* prototipes */

int  MGC_aloc   ();     /* int numero , int pid */
int  MGC_dealoc ();     /* int pid              */
void MGC_init   ();     /* void                 */
void MGC_mostra ();     /* void                 */

void MGC_gera_brgc();           /* int * array */
int  MGC_fat();                 /* int n       */

/* variaveis globais */

struct nodo_MGC {
       int node[NUM_PROC];
       int processo[NUM_PROC];
       int alocado[NUM_PROC];
       };

struct nodo_MGC    lista_MGC   [NUM_PROC];
int ft[GRAU_CUBO];

int p_aloc = 0; /* processadores ja alocados */
int grays;      /* numero de gray codes utilizados */

void main()
{
/* exemplo de execucao */

MGC_init();     /* inicializo estrutura de alocacao */
MGC_aloc(8,5);  /* aloco 8 processadores com pid 5  */
MGC_mostra();   /* vejo estrutura de alocacao       */
MGC_dealoc(5);  /* desaloco pid 5                   */
MGC_mostra();   /* vejo estrutura de alocacao       */
}

void MGC_init()
{
/* inicializa lista MGC */

typedef int GC[NUM_PROC];
GC l_brgc[NUM_PROC];
int i,j;
int dist,masc;


grays = MGC_fat(GRAU_CUBO) / ( MGC_fat(GRAU_CUBO/2) *
        MGC_fat(GRAU_CUBO-(GRAU_CUBO/2)) );
p_aloc = 0;

for ( j=0 ; j< (grays*2) ; j+=2 )
    {
    /* inicializa ft */
    masc = 1;
    dist = j;
    for (i=0 ; i < GRAU_CUBO; i++)
        {
        ft[i] = ((dist)&(masc));
        masc<<=1;
        }

    MGC_gera_brgc(l_brgc[j/2]);
    }


for ( i = 0 ; i < NUM_PROC ; i++ )
    for (j =0 ; j< grays ; j++)
        {
        lista_MGC[i].node[j] = l_brgc[j][i];
        lista_MGC[i].alocado[j] = 0;
        lista_MGC[i].processo[j] = 0;
        }
}

void MGC_mostra()
{
/* mostra lista MGC */

int i,j;

printf("\n         Lista MGC\n\n");
for ( i = 0 ; i < NUM_PROC ; i++ )
    {
    printf(" [%2d]   p: %2d    a:%2d -- ",
           lista_MGC[i].node[0],lista_MGC[i].processo[0],
           lista_MGC[i].alocado[0]);

    for (j=1 ; j< grays ; j++ )
        printf("%2d(%d) ",lista_MGC[i].node[j],
               lista_MGC[i].alocado[j]);

    printf("\n");
    }

printf("\n\t %d processadore(s) alocado(s).\n",p_aloc);
}

int  MGC_aloc   ( numero , pid )

int numero,
    pid;
{
/* aloco um hipercubo com "numero" processadores */
/* para pid: retorno um numero positivo se a     */
/* operacao obter sucesso                        */

int aloc_ok;
int bit_ok;
int i,j,h,t2,m,dim, pot;
float t1;
int set = 0;  /* gray code que estou usando para */
              /* detectar subcubos */
int aux_t1;

/* testo se existem processadores disponiveis */

if (numero > NUM_PROC-p_aloc) return FALSE;

/* encontro menor dimensao de cubo que contenha */
/* "numero" processadores                       */

dim = 0;
t1 = 1;

while ( t1 < numero )
   {
   dim++;
   t1 *= 2;
   }

aux_t1 = t1;

/* procuro menor "m" que satisfaca a equacao MGC */

set = 0;
aloc_ok = 0;

while( (set< grays)&&(!aloc_ok) )
   {
   m = 0;
   t1 = aux_t1;
   t1/=2;
   aloc_ok = 0;

   while (  ((((m+2)*t1)-1) < NUM_PROC) && !(aloc_ok) )
      {
```

```
      bit_ok = 1;
      t2 = m*t1;
      while ( (t2 <= (((m+2)*t1)-1)) && (bit_ok) )
         {
        if ( lista_MGC[t2].alocado[set] != 0 )
            bit_ok = 0;
         t2++;
         }

      if ( bit_ok )
         aloc_ok = 1;
     else
         m++;
      }
   set++;
   }

set--;  /* acertar set, executou uma vez a mais */

/* caso encontrado "m", aloco processadores em todos */
/* os sets                                           */

if ( aloc_ok )
   for ( i = m*t1 ; i <= (((m+2)*t1)-1) ; i++ )
       {
       lista_MGC[i].alocado[set]  = 1;
       lista_MGC[i].processo[set] = pid;
       p_aloc+=1;

       /* nos outros sets pelo indice */

       for ( j=0 ; j< grays ; j++)
           for (h=0 ; h< NUM_PROC ; h++ )
              if ( lista_MGC[i].node[set] ==
                   lista_MGC[h].node[j] )
                 {
                 lista_MGC[h].alocado[j]  = 1;
                 lista_MGC[h].processo[j] = pid;
                 }

       }

return(aloc_ok);
}

int  MGC_dealoc ( pid )

int pid;
{
/* libero processadores alocados pelo processo pid */
/* retorno numero de processadores desalocados     */
/* tenho que desalocar em todos os sets            */

int i,j;
int pid_ok = 0;
int templ =0 ;

for ( i = 0 ; i < NUM_PROC ; i++ )
    for ( j=0 ; j< grays ; j++ )
        if ( lista_MGC[i].processo[j] == pid )
          {
          lista_MGC[i].alocado[j]  = 0;
          lista_MGC[i].processo[j] = 0;
          templ++;
          }

p_aloc-= templ/grays;
pid_ok-= templ/grays;

if ( templ > 0 )
   return(TRUE);
else
   return(FALSE);
}

void MGC_gera_brgc(array)

int * array;

{
int i,n;
int proc = NUM_PROC;
int ind_e,ind_l;
unsigned char pot;

/* inicializo */

*(array+proc-2) = 0;
*(array+proc-1) = 1;
n = 2;
pot = 1;

for ( i=1 ; i < GRAU_CUBO ; i++)
    {
       pot<<=1;
    ind_l = proc-n;
    ind_e = proc - n * 2;

    while( ind_l < proc )
       {
       if ( ind_l % 2 == 0 )
          {
                       if ( ft[i] == 0 )
                          {
             *(array+ind_e) = *(array+ind_l)<<1;
             ind_e++;
             *(array+ind_e) = (*(array+ind_l)<<1)+1;
             ind_e++;
                          }
                       else
                          {
             *(array+ind_e) = *(array+ind_l);
             ind_e++;
             *(array+ind_e) = (*(array+ind_l)|pot);
             ind_e++;
                          }
          }
       else
                {
                       if ( ft[i] == 0 )
                          {
             *(array+ind_e) = (*(array+ind_l)<<1)+1;
             ind_e++;
             *(array+ind_e) = *(array+ind_l)<<1;
             ind_e++;
                          }
                       else
                          {
                          *(array+ind_e) =
                          (*(array+ind_l)|pot);
             ind_e++;
             *(array+ind_e) = *(array+ind_l);
             ind_e++;
             }
          }
       ind_l++;
       }
    n*=2;
     }
}

int  MGC_fat(n)

int n;
{
/* calculo recursivamente o fatorial de n */

if ( n==1 )
   return(1);
else
   return(n*MGC_fat(n-1));
}
```

A-1.4 tc.c

```
/*********************************************/
/**   Rotinas para aloacacao e gerencia de  **/
/**  processadores em maquinas Hipercubicas **/
/**      By Cesar De Rose - 29/05/92        **/
/**            Algoritmo: TC                **/
/*********************************************/

#include <stdio.h>
#include <math.h>

#define GRAU_CUBO 4
#define NUM_PROC  16

#define TRUE      1
#define FALSE     0

int  TC_sub_collapse();  /* int n,int k,int pre_level, */
                         /* int pre_step,char *masc,   */
                         /* int pid                    */
int  TC_R();             /* char * masc,int i          */
int  TC_aloca_sub();     /* char masc,int grau,int pid */
int  TC_aloca_todo();    /* int pid                    */

int  TC_aloc   ();       /* int numero , int pid *
```

```
int  TC_dealoc ();       /* int pid              */
void TC_init   ();       /* void                 */
void TC_mostra ();       /* void                 */

void TC_inc_bit();       /* int pos            */
void TC_seta_bit();      /* int pos,int val    */
int  TC_le_bit();        /* int pos            */
int  TC_procura();       /* int * bits,int pid */
int  TC_pertence();      /* int * bits,int i   */

/* dados globais TC */

struct nodo_TC {
       int processo;
       int alocado;
       };

struct nodo_TC lista_TC[NUM_PROC];
int cont[GRAU_CUBO]; /* 0-3 */

int p_aloc = 0;

void main()
{
TC_init();
TC_mostra();
TC_aloc(2,1);
TC_mostra();
TC_dealoc(1);
}

int TC_R(masc,i)

unsigned int *masc;
int i;
{
/* efetuo a operacao R(rotate) a partir do bit i do */
/* byte enderecado por a: o bit mais significativo  */
/* de *(masc) e' o bit 0, o menos significativo o 7 */

unsigned int b,c;
int p,m;

if (i>GRAU_CUBO-2)
   return(FALSE);

/* salvo bit 0 de a em b */

b = *(masc);
b &= 1;

if (i>0)
   {
   /* efetuo mascara */

   c = *(masc)>>(GRAU_CUBO-i);
   c <<= (GRAU_CUBO-i);
   }

/* efetuo deslocamento */

*(masc) >>= 1;

/* recoloco bit 0 */

p = (int) pow((double)2,(double)((GRAU_CUBO-1)-i));

if (b)
   *(masc) |= p;
else
   *(masc) &= 255 - p;

/* recoloco mascara */

b =  (int) pow((double)2,(double)(GRAU_CUBO-1));
for ( m=0 ; m<i  ; m++)
    {
    /* acerto bits mais significativos de *masc */
    /* em c estao os i bits mais significativos */
    /* a serem copiados para masc               */

    if ( (c&b) > 0 )
       *(masc) |= (c&b);
    else
       *(masc) &= 255-b;

    b>>=1;
    }

return(TRUE);
}

int TC_aloca_sub ( masc, grau , pid )

unsigned int masc;
int grau,
    pid;
{
int i;
int suc;

/* conversao de mascara para bits */

for ( i=0 ; i < GRAU_CUBO ; i++ )
    if ( (masc & (int)pow((double)2,(double)i)) > 0  )
       cont[i] =  0;
    else
       cont[i] = -1;

/* vario alguns bits de cont segundo sequencia bit */

for (i=0 ; i < (int) pow((double)2,(double)
    (GRAU_CUBO-grau)) ; i++)
    {

    /* efetuo a procura */

    suc = TC_procura(cont,pid);
    if ( suc )
       return(TRUE);

    TC_inc_bit(1);
    if ( TC_le_bit(1) == 2)
       {
       TC_seta_bit(1,0);
       TC_inc_bit(2);

       if ( TC_le_bit(2) == 2 )
          {
          TC_seta_bit(2,0);
          TC_inc_bit(3);

          if ( TC_le_bit(3) == 2 )
             {
             TC_seta_bit(3,0);
             TC_inc_bit(4);

             if ( TC_le_bit(4) == 2 )
                {
                TC_seta_bit(4,0);
                TC_inc_bit(5);

                if ( TC_le_bit(5) == 2 )
                   {
                   TC_seta_bit(5,0);
                   TC_inc_bit(6);
                   }
              }
             }
          }
       }
    }
return(FALSE);
}

int  TC_dealoc ( pid )

int pid;
{
/* libero processadores alocados pelo processo pid */
/* retorno numero de processadores desalocados     */

int i;
int pid_ok = 0;

for ( i = 0 ; i < NUM_PROC ; i++ )
    if ( lista_TC[i].processo == pid )
       {
       lista_TC[i].alocado  = 0;
       lista_TC[i].processo = 0;
       pid_ok += 1;
       p_aloc--;
       }

return(pid_ok);
}

void TC_init()
{
/* inicializa lista TC */

int i;

p_aloc = 0;

for ( i = 0 ; i < NUM_PROC ; i++ )
```

```
    {
    lista_TC[i].processo = 0;
    lista_TC[i].alocado  = 0;
    }
}

void TC_mostra()
{
/* mostra lista TC */

int i;

printf("%d processadore(s) alocado(s).\n",p_aloc);

printf("\n         Lista TC\n\n");
for ( i = 0 ; i < NUM_PROC ; i++ )
    printf(" [%2d]   p: %2d    a:%2d \n",i,
            lista_TC[i].processo,lista_TC[i].alocado);
}

int  TC_aloc   ( numero , pid )

int numero,
    pid;

{
/* aloco numero processadores para processo pid */
/* retorna um numero positivo se a operacao for */
/* bem sucedida                                 */

unsigned int mascara;
int i,k,n;
int suc,dim,t1;
int m,aloc_ok,bit_ok,t2;

/* testo se existem processadores disponiveis */

if (numero > NUM_PROC-p_aloc) return FALSE;

/* encontro menor dimensao de cubo que contenha */
/* "numero" processadores                       */

dim = 0;
t1 = 1;

while ( t1 < numero )
   {
   dim++;
   t1 *= 2;
   }

/* acerto variaveis TC */

n = GRAU_CUBO;
k = dim;

if (k==0)
   {
   /* aloca primeiro nodo disponivel */

   for ( i=0 ; i < NUM_PROC ; i++ )
       if ( lista_TC[i].alocado == 0)
          {
          lista_TC[i].alocado = 1;
          lista_TC[i].processo = pid;
          p_aloc++;
          return(TRUE);
          }

   return(FALSE);
   }

if (k==n)
   {
   suc = TC_aloca_todo(pid);
   return(suc);
   }

/* primeiro procuro na primary tree - no nivel n-k */
/* algoritmo buddy */

/* procuro menor "m" que satisfaca a equacao buddy */

m = 0;
aloc_ok = 0;

while (  ((((m+1)*t1)-1) < NUM_PROC) && !(aloc_ok) )
   {
   bit_ok = 1;
   t2 = m*t1;
   while ( (t2 <= (((m+1)*t1)-1)) && (bit_ok) )
      {
      if ( lista_TC[t2].alocado != 0 )
         bit_ok = 0;
      t2++;
      }

   if ( bit_ok )
      aloc_ok = 1;
   else
      m++;
}

/* caso encontrado "m", aloco processadores */

if ( aloc_ok )
   {
   for ( i = m*t1 ; i <= (((m+1)*t1)-1) ; i++ )
       {
       lista_TC[i].alocado  = 1;
       lista_TC[i].processo = pid;
       p_aloc += 1; /* mais um processador alocado */
       }

   return(TRUE);
   }

/* caso nao encontrado executo algoritmo G */

for ( i=n-k-1 ; i>=0 ; i-- )
    {
    /* reinicializo mascara a cada nova interacao */

    mascara = 255;        /* ********00000000 */
    mascara = mascara<<k;   /* shift left */
    mascara &= 255;

    /* printf("Mascara: [%d] !!!\n",mascara); */

    if ( !TC_R(&mascara,i) )
       printf("\nErro na operacao de rotate: [%d] rot
              [%d] !!!\n",mascara,i);

    /* printf("Mascara: [%d] !!!\n",mascara); */

    suc = TC_aloca_sub(mascara,k,pid);
    if (suc)
       return(TRUE);

    suc = TC_sub_collapse(n,k,i,0,&mascara,pid);
    if (suc)
       return(TRUE);
    }

return(FALSE);
}

int TC_sub_collapse(n,k,pre_level,pre_step,masc,pid)

int n,
    k,
    pre_level,
    pre_step,
    pid;
unsigned int *masc;
{
int step,i;
int suc;
unsigned int mascara;

step = pre_step + 1;
if (step >= k)
   return(FALSE);
else
   for ( i = pre_level ; i <= n-k-1 ; i++ )
       {
       mascara = *(masc);

       if ( !TC_R(&mascara,step+i) )
       printf("\nErro na operacao de rotate: [%d] rot
              [%d] !!!\n",mascara,i);

       /* printf("Mascara: [%d] !!!\n",mascara); */

       suc = TC_aloca_sub(mascara,k,pid);
       if (suc)
          return(TRUE);

       suc = TC_sub_collapse(n,k,i,step,&mascara,pid);
       if (suc)
          return(TRUE);
       }
return(FALSE);
}

int  TC_aloca_todo(pid)
```

```
int pid;
{
/* tenta alocar todo o cubo compartilhado   */
/* retorna positivo se ok, negativo se erro */

int i;

for ( i=0 ; i < NUM_PROC ; i++ )
    if ( lista_TC[i].alocado == 1 )
       return(FALSE);

/* todos os nodos estao livres, efetuo alocacao */

for ( i=0 ; i < NUM_PROC ; i++ )
    {
    lista_TC[i].alocado  = 1;
    lista_TC[i].processo = pid;
    p_aloc++;
    }

return(TRUE);
}

void TC_inc_bit(pos)

int pos;
{
int ex;
int i;

ex = 0;
for(i=0 ; i<GRAU_CUBO ; i++)
   {
   if ( cont[i] != -1 )
      {
      ex++;
      if (ex==pos)
         {
         cont[i]+=1;
         break;
         }
      }
   }
}

void TC_seta_bit(pos,val)

int pos,
    val;
{
int ex;
int i;

ex = 0;
for(i=0 ; i<GRAU_CUBO ; i++)
   {
   if ( cont[i] != -1 )
      {
      ex++;
      if (ex==pos)
         {
         cont[i] = val;
         break;
         }
      }
   }
}

int TC_le_bit(pos)

int pos;
{
int ex;
int i;

ex = 0;
for(i=0 ; i<GRAU_CUBO ; i++)
   {
   if ( cont[i] != -1 )
      {
      ex++;
      if (ex==pos)
         {
         return(cont[i]);
         }
      }
   }
return(FALSE);
}

int TC_procura(bits,pid)

int * bits;
int pid;
{
/* procuro todos os nodos com bits iguais aos */
/* diferentes de -1: se todo subcubo livre,   */
/* aloco e retorno positivo. senao retorno -1 */

int i;

/* varro cada nodo contido no subcubo masc */

for ( i=0 ; i < NUM_PROC ; i++ )
    {
    if ( TC_pertence(bits,i) == 1  )
       {
       /* nodo pertence ao sub_cubo masc */
       /* testo se sta livre             */

       if (lista_TC[i].alocado == 1)
          {
          /* nodo ocupado     */
          /* sub_cubo ocupado */

          return(FALSE);
          }
       }
    }

/* aloco nodos que pertencem ao sub_cubo */

for ( i=0 ; i < NUM_PROC ; i++ )
    {
    if ( TC_pertence(bits,i) == 1 )
       {
       /* aloco nodo */

       lista_TC[i].alocado  = 1;
       lista_TC[i].processo = pid;
       p_aloc++;
       }

    }

return(TRUE);
}

int  TC_pertence(bits,i)

int * bits;
int i;
{
/* verifico se nodo pertence a subcubo indicado */
/* nos bits[GRAU_CUBO]: se pertencer retorno 1,  */
/* senao retorno negativo                        */

int j;

for ( j=0 ; j < GRAU_CUBO ; j++ )
    if (bits[j] != -1)
       {
          if ((i & (int)pow((double)2,(double)j)) !=
             (bits[j]*(int) pow((double)2,(double)j)) )
             return(FALSE);
       }

return(TRUE);
}
```

A-1.5 fl.c

```
/*************************************************/
/**   Rotinas para aloacacao e gerencia de    **/
/**  processadores em maquinas Hipercubicas   **/
/**       By Cesar De Rose - 29/05/92         **/
/**            Algoritmo: FL                  **/
/*************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define GRAU_CUBO 6
#define NUM_PROC  64

#define TRUE 1
```

```
#define FALSE 0

int  FL_aloc   ();     /* int numero , int pid */
int  FL_dealoc ();     /* int pid              */
void FL_init   ();     /* void                 */
void FL_mostra ();     /* void                 */
void FL_free   ();     /* void                 */

int FL_procs();            /* int dim */

int irmao();             /* nodo_FL * A, nodo_FL *B,  */
                         /* int grau                  */
void compacta();         /* nodo_FL * obj , int nivel */
void split();            /* nodo_FL * obj , int level */
struct nodo_FL * join(); /* nodo_FL * obj ,           */
                         /* nodo_FL * obj1 ,int level,*/
                         /* int offset                */

/* dados globais FL */

struct nodo_FL{
       int dim[GRAU_CUBO]; /* 0-(GRAU_CUBO-1) */
       int processo;
       int alocado;
       struct nodo_FL * next;
       };

struct nodo_FL * lista_FL[GRAU_CUBO+1]; /* 0-GRAU_CUBO */
int p_aloc = 0;

void main()
{
int sera;
long tempo = 0;
int i;

FL_init();

/* atrolha */

for ( i=0 ; i< (NUM_PROC/2) ; i++ )
     FL_aloc(1,i);
FL_mostra();

tempo = clock();
FL_aloc(1,999);
 tempo = clock() - tempo;
FL_mostra();

FL_dealoc(999);
FL_mostra();

FL_free();
printf("\n Tempo: %ld\n",tempo);
}

void FL_init()
{
/* Inicializo estrutura */

int i,j;
struct nodo_FL *aux;

p_aloc = 0;

for (i=0 ; i <= GRAU_CUBO ; i++)
    lista_FL[i] = NULL;

/* incluo primeiro nodo */

aux = malloc(sizeof(struct nodo_FL));
aux->processo = 0;
aux->alocado  = 0;
for (j=0 ; j < GRAU_CUBO ; j++)
    aux->dim[j] = -1;          /* -1 = wildcard(*) */

aux->next = NULL;
lista_FL[GRAU_CUBO] = aux;
}

void FL_free()
{
/* libero memoria alocada */

int i;
struct nodo_FL *aux,*next;

for (i=0 ; i < GRAU_CUBO ; i++)
    {
    next = lista_FL[i];
    if (next)
       {
       lista_FL[i] = NULL;
       while ( next )
          {
          aux = next;
          next = aux->next;
          free(aux);
          }
       }
    }
}

void FL_mostra()
{
/* mostra lista_FL */

int i,j;
struct nodo_FL *aux;

printf("%d processador(es) alocado(s).\n",p_aloc);

printf("\n          Estrutura FL \n\n");
for (i=GRAU_CUBO ; i >= 0 ; i--)
    {
    printf("Fila [%d]",i);
    aux = lista_FL[i];
    if (aux)
       {
       while ( aux )
          {
          printf(" A[%d] P[%d] ",aux->alocado,
                 aux->processo);
          for (j=GRAU_CUBO-1 ; j>=0 ; j-- )
              if ( aux->dim[j] == -1)
                 printf("*");
              else
                 printf("%d",aux->dim[j]);

          aux = aux->next;
          }
       printf("\n");
       }
    else
       printf("vazia \n");
    }
}

void split ( obj,level)

struct nodo_FL * obj;
int level;

{
int i;
int offset;
struct nodo_FL * aux;

/* crio nodos na lista level-1 */

aux = lista_FL[level-1];
if (aux)
   {
   while (aux->next)
        aux = aux -> next;
   aux->next = malloc(sizeof(struct nodo_FL));
   aux = aux -> next;
   }
else
   {
   aux = malloc(sizeof(struct nodo_FL));
   lista_FL[level-1] = aux;
   }

/* encontro offset */

offset = GRAU_CUBO;
for ( i=GRAU_CUBO-1 ; i>=0 ; i-- )
    if (obj->dim[i] == -1)
       {
       offset = i;
       break;
       }

/* acerto lo nodo */

aux -> processo = 0;
aux -> alocado = 0;
for (i=0 ; i<GRAU_CUBO ; i++)
    aux -> dim[i] = obj -> dim[i];
aux->dim[offset]++;

/* adiciono +1 nodo */

aux->next = malloc(sizeof(struct nodo_FL));
aux = aux->next;
```

```
aux -> next= NULL;
aux->processo = 0;
aux->alocado =0;
for (i=0 ; i<GRAU_CUBO ; i++)
    aux->dim[i] = obj->dim[i];
aux->dim[offset]+=2;

/* retiro nodo que originou split */

aux = lista_FL[level];

if (aux==obj)
   lista_FL[level] = obj->next;
else
   {
while (aux->next != obj)
   aux=aux->next;
aux->next = obj->next;
   }

free(obj);
}

struct nodo_FL * join( obj, obj1, level, offset)

struct nodo_FL * obj;
struct nodo_FL * obj1;
int level,offset;

{

int i;
struct nodo_FL * aux;
struct nodo_FL * new;

/* crio nodo novo na lista level+1 */

aux = lista_FL[level+1];
if (aux)
   {
   while(aux->next)
      aux = aux->next;

   aux->next = malloc(sizeof(struct nodo_FL));
   aux=aux->next;
   }
else
   {
   aux = malloc(sizeof(struct nodo_FL));
   lista_FL[level+1] = aux;
   }

aux->processo = 0;
aux -> alocado = 0;
aux -> next = NULL;
new = aux;

for ( i=0 ; i<GRAU_CUBO ; i++)
    aux->dim[i] = obj->dim[i];

aux->dim[offset] = -1;

/* retiro obj e obj1 */

aux = lista_FL[level];
if (aux==obj)
   lista_FL[level] = obj->next;
else
   {
   while(aux->next != obj )
      aux = aux->next;
   aux->next = obj->next;
   free(obj);
   }

aux = lista_FL[level];
if (aux==obj1)
   lista_FL[level] = obj1->next;
else
   {
   while(aux->next != obj1 )
      aux = aux->next;
   aux->next = obj1->next;
   free(obj1);
   }

return(new);
}

int FL_aloc (numero, pid)

int numero,pid;

{
/* retorno TRUE se alocacao foi realizada, FALSE se */
/* nao foi                                          */

int i,achou;
struct nodo_FL * aux;
struct nodo_FL * queb;
int grau,t1;

/* testo se existem processadores disponiveis */

if (numero > NUM_PROC-p_aloc) return FALSE;

/* acho grau do subcubo desejado */

grau = 0;
t1 = 1;

while ( t1 < numero )
   {
   grau++;
   t1 *= 2;
   }

/* procuro cubo desejado na respectiva lista */

aux = lista_FL[grau];
while(aux)
   {
   if (aux->alocado == 0)
      {
      /* aloco subcubo */

      aux->alocado = 1;
      aux->processo = pid;
      p_aloc+= t1;
      return(TRUE);
      }
   aux=aux->next;
   }

/* nao existe no nivel. Procuro o subcubo mais proximo */
/* em um nivel superior que esteja livre para quebrar  */
/* em menores                                          */

achou = 0;
i = grau+1;
while( (i<=GRAU_CUBO) && (!achou) )
   {
   aux = lista_FL[i];
   while(aux && !achou)
      {
      if (aux->alocado == 0)
         {
         achou = i;
         queb = aux;
         }
      aux = aux -> next;
      }
   i++;
   }

/* se encontrei subcubo ivre, quebro ate o nivel */
/* desejado: se nao encontrei nao posso alocar.  */
/* Subcubo nao disponivel                        */

if (achou)
   {
   /* quebro ate grau */

   for (i=achou ; i>grau ; i--)
       {
       split(queb,i);
       queb = lista_FL[i-1];
       while(queb->next->next)
          queb = queb ->next;
       }
   }
else
   return(FALSE);

/* aloco no nivel */

aux = lista_FL[grau];
while(aux)
   {
   if (aux->alocado == 0)
      {
      /* aloco subcubo */

      aux->alocado = 1;
      aux->processo = pid;
      p_aloc+=t1;
```

```
      return(TRUE);
      }
   aux=aux->next;
   }

return(FALSE);
}

int irmao( A, B, grau)

int *A;
int *B;
int grau;
{
/* compara dois subcubos para ver se sao derivados de */
/* um subcubo de grau n+1                             */

int i;
for ( i=0 ; i < grau ; i++ )
    if ( *(A+GRAU_CUBO-1-i) != *(B+GRAU_CUBO-1-i) )
       return(FALSE);

return(TRUE);
}

int FL_dealoc( pid )

int pid;
{
/* desaloco subcubo alocado por este processo: chamo */
/* rotina compacta para reagrupar subcubos livres    */

int i,achou;
struct nodo_FL * aux;

achou = 0;
i = GRAU_CUBO;
while( (!achou)&&(i>=0) )
   {
   aux = lista_FL[i];
   while( (aux)&&(!achou) )
      {
      if (aux->processo == pid)
         {
         aux->alocado = 0;
         aux->processo = 0;

         p_aloc-= FL_procs(i);  /* i = grau do cubo */
                                /* desalocado */
         compacta(aux,i);
         return(TRUE);
         }
      aux = aux->next;
      }
   i--;
   }
return(FALSE);
}

void compacta( obj, nivel )

int nivel;
struct nodo_FL * obj;
{
struct nodo_FL *aux;
struct nodo_FL *aux1;

/* vejo se irmao liberou */

aux = lista_FL[nivel];
while(aux)
   {
   if (aux==obj)
      aux = aux->next;

   if (aux)
      if (aux->alocado == 0 )
         if ( irmao(aux,obj,GRAU_CUBO-1-nivel) )
            {
            aux1 = join(aux,obj,nivel,nivel);

            if (nivel<GRAU_CUBO)
               compacta(aux1,nivel+1);
            }

    if (aux)
       aux = aux->next;
    }

return;  /* irmao nao esta na fila */
}

int FL_procs( dim )

int dim;
{
/* retorno numero de processadores desta dimensao */

int t1 = 1;
int d = 0;

while ( d < dim )
   {
   d++;
   t1 *= 2;
   }

return(t1);
}
```

ANEXO A-2 LISTAGEM DOS ALGORITMOS PARALELOS

As listagens neste anexo são implementações paralelas na linguagem "C" dos algoritmos para alocação e gerência de processadores em máquinas hipercúbicas descritos no capítulo 4.

Os algoritmos foram paralelizados de duas formas: utilizando os recursos da rede de estações e utilizando Transputers. Na versão para a rede de estações a paralelização é implementada através de diretivas para a comunicação entre processos em diferentes estações (*sockets*) [SUN 90]. Nos Transputers foi utilizada a linguagem "C" Paralela 3L da Inmos [INM 89].

A-2.1 Algoritmos para a rede de estações

A-2.1.1 coord.c

```
/* Esqueleto para implementacao distribuida  */
/* dos algoritmos de alocacao    By De Rose  */
/*  Coordenador da secao critica            */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <sys/time.h>

#define SOCKET_M  2223 /* numero da porta que conecta */
                       /* com o escravo               */
#define SOCKET_S  2221 /* numero da porta que conecta */
                       /* com o coordenador           */

#define P  1
#define V  2
#define EOT 0

/* prototipos */

int Criosock_bind_aceito(); /* recebe porta,retorna   */
                            /* hand                   */
int respondo_mesg();        /* retorna hand */

/* variaveis globais */

int master_sock,slave_sock;
int arg;
struct timeval timeout;
fd_set  read_socket;
int sel_ret;
int semafor = 1;

void main(void)
{
int end;

/* aceito conexao do mestre */

master_sock = Criosock_bind_aceito(SOCKET_M);

/* aceito conexao do escravo */

slave_sock = Criosock_bind_aceito(SOCKET_S);

printf("\n(C) Conexoes estabelecidas !!!");

/* pooling entre as sockets de mestre e escravo */

/* defino time outs do pooling (select) */

timeout.tv_sec = 518400;   /* 6 dias */
timeout.tv_usec = 0;

end = 0;

while (!end)
    {
    /* faco select */

    FD_ZERO(&read_socket);

    FD_SET(master_sock,&read_socket);
    FD_SET(slave_sock,&read_socket);

    sel_ret = select (FD_SETSIZE,&read_socket,(fd_set *)
                      0,(fd_set *) 0,&timeout);

    if ( sel_ret <= 0 )
       {
       printf("\n(C) Erro na operacao de select !!!\n");
       exit(3);
       }


   if (FD_ISSET(master_sock, &read_socket))
       {
       /* recebi comunicacao do mestre */

       printf("\n(C) Recebi comunicacao MASTER.");

       end = respondo_mesg(master_sock);
       }


   if (FD_ISSET(slave_sock, &read_socket))
       {
       /* recebi comunicacao do escravo */

       printf("\n(C) Recebi comunicacao SLAVE.");

       end = respondo_mesg(slave_sock);
       }

    }

/* fecho sockets */

close(master_sock);
close(slave_sock);

printf("\n(C) Corto Conexao.\n");
}

int Criosock_bind_aceito(porta)
int porta;

{
struct sockaddr_in server;
int hand;
int flag = 1;

/* crio socket */

if ( (hand = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
   {
   perror("Opening stream socket");
   exit(1);
   }

setsockopt(hand,SOL_SOCKET,SO_REUSEADDR,(char *)
          &flag,sizeof flag);

/* faco bind */

 server.sin_family = AF_INET;
 server.sin_addr.s_addr = INADDR_ANY;
 server.sin_port = porta;

 if (bind(hand,(struct sockaddr *) &server,sizeof(server))
     < 0)
    {
    perror("Binding stream socket");
    exit(1);
    }

/* listen especifica o numero de conexoes possiveis */

 listen(hand,2);

/* accept extrai primeira conexao da lista de */
/* conexoes pendentes                         */

 hand = accept(hand, (struct sockaddr *) 0, (int *) 0);

 if (hand == -1) {
    perror("accept");
    exit(1);
    }

return(hand);
}

int respondo_mesg(hand)
int hand;
{
unsigned char token[1];
unsigned char tok[1];
int end;

token[0] = 0;
end = 0;

/* recebo mensagem */

tok[0] = 0;
read(hand,tok,1);

/* processo mesangem */
```

```
switch(tok[0])
   {
   case P:  printf("\n(C) Recebi oper P.");
            semafor -= 1;
            if (semafor >= 0)
               {
               /* return OK */
               token[0] = 1;
               }
            else
               {
               /* return NO */
               token[0] = 0;
               }

            /* envio mensagem */

            write(hand,token,1);
            break;

   case V:  /* operacao V */

            printf("\n(C) Recebi oper V.");
            semafor = 1;
            break;

   case EOT: printf("\n(C) Recebi oper EOT.");
             end = 1;
   }
return(end);
}
```

A-2.1.2 budy_par_M.c

```
/* Algoritmo Buddy - Versao Distribuida */
/* By De Rose   processo MESTRE         */

#define NUM_PROC 16

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <sys/time.h>

#define SOCKET_S  2222 /* numero da porta que conecta */
                       /* com o escravo               */
#define SOCKET_C  2223 /* numero da porta que conecta */
                       /* com o coordenador           */
#define P  1
#define V  2
#define EOT 0

#define TRUE     1
#define FALSE    0

/* prototipos */

int Criosock_conecto();   /* envio maquina escravo */
                          /* e porta, retorna hand */

int  buddy_aloc   ();     /* int numero , int pid */
int  buddy_dealoc ();     /* int pid              */
void buddy_init   ();     /* void                 */
void buddy_mostra ();     /* void                 */


/* variaveis globais */

int slave_sock,coord_sock;
int arg;
unsigned char token[3];
unsigned char tok[3];

struct nodo_buddy {
       int processo;
       int alocado;
       };


struct nodo_buddy lista_buddy[NUM_PROC/2];
struct nodo_buddy todo;  /* controle de todo o cubo */
int p_aloc = 0;          /* processadores ja alocados */

/* argv[1] - maquina onde se encontra o escravo     */
/* argv[2] - maquina onde se encontra o coordenador */

main(argc,argv)

int argc; char *argv[];
{
 unsigned int i;
 struct timeval      tp;
 struct timezone     tz;
 double              tmp,
                     tempo_inicial, tempo_final,
                     tempo_total;
int numero;
int ret_code;

/* inicializo socket */

token[0] = 0;

/* peco conexao SOCKET_C para o coordenador  */
/* que ja espera                             */

coord_sock = Criosock_conecto(argv[1],SOCKET_C);

/* peco conexao SOCKET_S para o escravo que ja espera */

slave_sock = Criosock_conecto(argv[2],SOCKET_S);

/* conexao com escravo estabelecida */

printf("\n(M) Comunicacao estabelecida !!!");

buddy_init();

/* pego tempo inicial */

gettimeofday ( &tp, &tz);
tmp      = tp.tv_usec;
tempo_inicial =  ( tmp / 1000000) + tp.tv_sec;

/* Efetuo Alocacao */

ret_code = buddy_aloc(8,9);

if (ret_code > 0)
   printf("\n(M) Aloquei os processadores.");
else
   printf("\n(M) Alocacao falhou !!!");

buddy_mostra();

ret_code = buddy_dealoc(9);

if (ret_code > 0)
   printf("\n(M) Desaloquei os processadores.");
else
   printf("\n(M) Desalocacao falhou. Pid nao encontrado
          !!!");

buddy_mostra();

/* pego tempo final */

gettimeofday ( &tp, &tz);
tmp       = tp.tv_usec;
tempo_final  = ( tmp / 1000000) + tp.tv_sec;

/* calculo diferenca */

tempo_total = tempo_final - tempo_inicial;
printf("\n(M) Tempo total = %f\n",tempo_total);

/* desativo escravo */

printf("\n(M) Desativo escravo.");

token[0] = 3;
write(slave_sock,token,1);

/* desativo coordenador */

printf("\n(M) Desativo coordenador.");

token[0] = EOT;
write(coord_sock,token,1);

/* fecho sockets */

close(slave_sock);
close(coord_sock);
```

```
printf("\n(M) Cortei conexao !!!\n");
}

int Criosock_conecto(host,porta)
char *host;
int porta;

{
 struct sockaddr_in server;
 struct hostent *hp, *gethostbyname();
 int hand;
 int flag = 1;

/* crio socket */

if ( (hand = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
   {
   perror("Opening stream socket");
   exit(1);
   }

setsockopt(hand,SOL_SOCKET,SO_REUSEADDR,(char *)
           &flag,sizeof flag);

/* tento conexao com maquina remota */

 server.sin_family = AF_INET;
 if ( (hp = (struct hostent *) gethostbyname(host))
       == 0)
    {
    fprintf(stderr,"%s: unknownhost\0", host);
    exit(2);
    }
 bcopy( (char *)hp->h_addr, (char *)&server.sin_addr,
        hp->h_length);
 server.sin_port = porta;

if (connect(hand, (struct sockaddr *) &server,
    sizeof(server)) < 0)
    {
    perror("connect");
    exit(1);
    }

return(hand);
}

/* rotinas buddy */

void buddy_init()
{
/* inicializa lista buddy */

int i;

p_aloc = 0;
todo.processo = 0;
todo.alocado = 0;

for ( i = 0 ; i < NUM_PROC/2 ; i++ )
    {
    lista_buddy[i].processo = 0;
    lista_buddy[i].alocado  = 0;
    }
}

void buddy_mostra()
{
/* mostra lista buddy */

int i;

printf("\n         Lista Buddy Local\n\n");
for ( i = 0 ; i < NUM_PROC/2 ; i++ )
    printf(" [%2d]   p: %2d    a:%2d \n",i,
          lista_buddy[i].processo,lista_buddy[i].alocado);

printf("\nTodo Cubo [%d].",todo.alocado);
}

int  buddy_aloc   ( numero , pid )

int numero,
    pid;
{
/* aloco um hipercubo com "numero" processadores */
/* para pid                                      */
/* retorno TRUE se a operacao obter sucesso      */

int aloc_ok = 0;
int bit_ok;
int i,t1,t2,m,dim, pot;

/* testo se existem processadores disponiveis */

if (todo.alocado) return FALSE;

if (numero == NUM_PROC)
   {
   todo.alocado = 1;
   todo.processo = pid;
   return(TRUE);
   }

/* disparo procura remota */

token[0] = 1;
token[1] = numero;
token[2] = pid;

printf("\n(M) Envio num = %d pid = %d para alocar !",
       token[1],token[2]);

write(slave_sock,token,3);

/* vejo se local tem processadores disponiveis para */
/* atender                                          */


if (numero < ((NUM_PROC/2)-p_aloc) )
{

/* encontro menor dimensao de cubo que contenha */
/* "numero" processadores                       */

dim = 0;
t1 = 1;

while ( t1 < numero )
   {
   dim++;
   t1 *= 2;
   }

/* procuro menor "m" que satisfaca a equacao buddy */

m = 0;
aloc_ok = 0;

while (  ((((m+1)*t1)-1) < (NUM_PROC/2)) && !(aloc_ok))
   {
   bit_ok = 1;
   t2 = m*t1;
   while ( (t2 <= (((m+1)*t1)-1)) && (bit_ok) )
      {
      if ( lista_buddy[t2].alocado != 0 )
         bit_ok = 0;
      t2++;
      }

   if ( bit_ok )
      aloc_ok = 1;
   else
      m++;
}

/* caso encontrado "m", tento alocar processadores */

if ( aloc_ok )
   {
   /* faco consulta a coordenador - secao critica */

   token[0] = P;
   write(coord_sock,token,1);

   printf("\n(M) Testando coordenador sobre SC.");

   /* espero resposta */

   read(coord_sock,tok,1);

   printf("\n(M) Resposta coord Byte = %d.",tok[0]);

   if (tok[0])
      for ( i = m*t1 ; i <= (((m+1)*t1)-1) ; i++ )
          {
          lista_buddy[i].alocado  = 1;
          lista_buddy[i].processo = pid;
          p_aloc += 1;
          }
    else
       aloc_ok = 0;
   }

   }
```

```
else
   aloc_ok = 0;   /* nao posso alocar local */

if (tok[0])
   printf("\nLocal alocou legal (pid=%d) !!!",pid);
else
   printf("\nLocal NAO alocou legal (pid=%d) !!!",pid);


/* recebo ok ou negado do remoto */

read(slave_sock,tok,1);

if (tok[0])
   {
   aloc_ok = 1;
    printf("\nRemoto encontrou !!!");
   }
else
   printf("\nRemoto NAO encontrou !!!");

if (!aloc_ok)   /* local negou, pego ret code remoto */
   aloc_ok = tok[0];

/* libero coordenador - secao critica */

printf("\nLibero Secao critica !!!");
token[0] = V;
write(coord_sock,token,1);

return(aloc_ok);
}

int  buddy_dealoc ( pid )

int pid;
{
/* libero processadores alocados pelo processo pid */
/* retorno numero de processadores desalocados     */
/* efetuo desalocacao remota */

int i;
int pid_ok = 0;

if (pid == todo.processo)
   {
   todo.processo = 0;
   todo.alocado = 0;
   return(TRUE);
   }

/* envio pid para nodo remoto */

token[0] = 2;
token[1] = pid;
token[2] = 0;

printf("\n(M) Envio pid para remoto desalocar = %d
       !",token[1]);

write(slave_sock,token,3);

/* nao preciso esperar resposta */

/* efetuo processamento local */

for ( i = 0 ; i < (NUM_PROC/2) ; i++ )
    if ( lista_buddy[i].processo == pid )
       {
       lista_buddy[i].alocado  = 0;
       lista_buddy[i].processo = 0;
       pid_ok += 1;
       p_aloc --;
       }

if (pid_ok > 0)
   return(TRUE);
else
   return(FALSE);
}
```

A-2.1.3 budy_par_S.c

```
/* Algoritmo Buddy - Versao Distribuida */
/* By De Rose  processo (ESCRAVO)       */

#define NUM_PROC 16

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>

#define SOCKET_M  2222 /* numero da porta que conecta */
                       /* com o escravo               */
#define SOCKET_C  2221 /* numero da porta que conecta */
                       /* com o coordenador           */
#define P  1

#define TRUE     1
#define FALSE    0

/* prototipos */

int Criosock_bind_aceito(); /* envio porta, recebo */
                            /* hand                */
int Criosock_conecto();     /* envio maquina e porta */
                            /* recebo hand           */

int  buddy_aloc   ();  /* int numero , int pid */
int  buddy_dealoc ();  /* int pid              */
void buddy_mostra ();  /* void                 */
void buddy_init   ();  /* void                 */

/* variaveis globais */

int master_sock,coord_sock;
int arg;
unsigned char token[3];
unsigned char tok[3];
int numero;

struct nodo_buddy {
       int processo;
       int alocado;
       };


struct nodo_buddy lista_buddy[NUM_PROC/2];

int p_aloc = 0;   /* processadores ja alocados */

/* argv[1] - maquina onde se encontra o coordenador */

main(argc,argv)

int argc; char *argv[];
{
int ret_code;
int pid,end;

token[0] = 0;

/* aceito cenexao do mestre */

master_sock = Criosock_bind_aceito(SOCKET_M);

/* tento conexao com o coordenador */

coord_sock = Criosock_conecto(argv[1],SOCKET_C);

printf("\n(S) Conexao estabelecida !!!");

/* inicializo algoritmo */

buddy_init();

/* atendo pedidos do mestre */

end = 0;

while (!end)
   {

printf("\n(S) Espero mensagem.");

tok[0] = 0;
read(master_sock,tok,3);

/* processo mesangem */

switch( tok[0] )
   {
   case 1:       /* efetuo alocacao */

        printf("\n(S) Chegou pedido Aloc !!!");
```

```
        /* acerto parametros */

        numero = tok[1];
        pid = tok[2];

        ret_code = buddy_aloc(numero,pid);

        if (ret_code > 0)
           {
           printf("\n(S) Aloquei os processadores.");
           token[0] = 1;
           write(master_sock,token,1);
           buddy_mostra();
           }
        else
           {
           printf("\n(S) Alocacao falhou !!!");
           token[0] = 0;
           write(master_sock,token,1);
           }

        break;

   case 2:      /* efetuo desalocacao */

        printf("\n(S) Chegou pedido Desaloc !!!");

        pid = tok[1];

        ret_code = buddy_dealoc(pid);

        if (ret_code > 0)
           {
           printf("\n(S) Desaloquei os processadores.");
           token[0] = 1;
           write(master_sock,token,1);
           }
        else
           {
           printf("\n(S) Desalocacao falhou !!!");
           token[0] = 0;
           write(master_sock,token,1);
           }

        break;

   case 3:      /* corto conexao, saindo do loop */

        end = 1;
        printf("\nSai do Loop (mesg 3) !!!");
        buddy_mostra ();
    }

   }

/* fecho sockets */

close(master_sock);
close(coord_sock);

printf("\n(S) Corto Conexao.\n");
}

int Criosock_bind_aceito(porta)
int porta;

{
struct sockaddr_in server;
int hand;
int flag = 1;

/* crio socket */

if ( (hand = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
   {
   perror("Opening stream socket");
   exit(1);
   }

setsockopt(hand,SOL_SOCKET,SO_REUSEADDR,(char *)
          &flag,sizeof flag);

/* faco bind */

 server.sin_family = AF_INET;
 server.sin_addr.s_addr = INADDR_ANY;
 server.sin_port = porta;

 if (bind(hand,(struct sockaddr *) &server,
         sizeof(server)) < 0)
    {
    perror("Binding stream socket");
    exit(1);
    }

/* listen especifica o numero de conexoes possiveis */

listen(hand,1);

/* accept extrai primeira conexao da lista de */
/* conexoes pendentes                         */

 hand = accept(hand, (struct sockaddr *) 0, (int *) 0);

 if (hand == -1) {
    perror("accept");
    exit(1);
    }

return(hand);
}

int Criosock_conecto(host,porta)
char *host;
int porta;

{
int hand;
int flag = 1;

struct sockaddr_in server;
struct hostent *hp, *gethostbyname();

/* crio socket */

if ( (hand = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
   {
   perror("Opening stream socket");
   exit(1);
   }

setsockopt(hand,SOL_SOCKET,SO_REUSEADDR,(char *)
          &flag,sizeof flag);

/* tento conexao com maquina remota */

 server.sin_family = AF_INET;
 if ( (hp = (struct hostent *) gethostbyname(host))
      == 0)
    {
    fprintf(stderr,"%s: unknownhost\0", host);
    exit(2);
    }
 bcopy( (char *)hp->h_addr, (char *)&server.sin_addr,
        hp->h_length);
 server.sin_port = porta;

if (connect(hand, (struct sockaddr *) &server,
    sizeof(server)) < 0)
    {
    perror("connect");
    exit(1);
    }

return(hand);
}

/* rotinas buddy */

void buddy_init()
{
/* inicializa lista buddy */

int i;

p_aloc = 0;

for ( i = 0 ; i < (NUM_PROC/2) ; i++ )
    {
    lista_buddy[i].processo = 0;
    lista_buddy[i].alocado  = 0;
    }
}

void buddy_mostra()
{
/* mostra lista buddy */

int i;

printf("\n          Lista Buddy Local\n\n");
for ( i = NUM_PROC/2 ; i < NUM_PROC ; i++ )
    printf(" |%2d|   p: %2d    a:%2d \n",i,
          lista_buddy[i].processo,lista_buddy[i].alocado);

}
```

```
int  buddy_aloc   ( numero , pid )

int numero,
    pid;
{
/* aloco um hipercubo com "numero" processadores */
/* para pid                                      */
/* retorno TRUE se a operacao obter sucesso      */

int aloc_ok;
int bit_ok;
int i,t1,t2,m,dim, pot;

/* testo se existem processadores livres */

if (numero < ((NUM_PROC/2)-p_aloc) )
{

/* encontro menor dimensao de cubo que contenha */
/* "numero" processadores                       */

dim = 0;
t1 = 1;

while ( t1 < numero )
   {
   dim++;
   t1 *= 2;
   }

/* procuro menor "m" que satisfaca a equacao buddy */

m = 0;
aloc_ok = 0;

while (  ((((m+1)*t1)-1) < (NUM_PROC/2)) && !(aloc_ok))
   {
   bit_ok = 1;
   t2 = m*t1;
   while ( (t2 <= (((m+1)*t1)-1)) && (bit_ok) )
      {
      if ( lista_buddy[t2].alocado != 0 )
         bit_ok = 0;
      t2++;
      }

   if ( bit_ok )
      aloc_ok = 1;
   else
      m++;
}

/* caso encontrado "m", tento alocar processadores */

if ( aloc_ok )
   {
   /* faco consulta a coordenador - secao critica */

   token[0] = P;
   write(coord_sock,token,1);

   printf("\n(S) Testando coordenador sobre SC.");

   /* espero resposta */
   read(coord_sock,tok,1);

   printf("\n(S) Resposta coord Byte = %d.",tok[0]);

   if (tok[0])
      {
      for ( i = m*t1 ; i <= (((m+1)*t1)-1) ; i++ )
          {
          lista_buddy[i].alocado  = 1;
          lista_buddy[i].processo = pid;
          p_aloc += 1;
          }
      printf("\n(S) Local alocou legal (pid=%d)
              !!!",pid);
      return(aloc_ok);
      }
   }

} /* end if */

printf("\n(S) Local NAO alocou legal (pid=%d) !!!",pid);
return(FALSE);
}

int  buddy_dealoc ( pid )

int pid;
{

/* libero processadores alocados pelo processo pid */
/* retorno numero de processadores desalocados     */

int i;
int pid_ok = 0;

/* efetuo processamento local */

for ( i = 0 ; i < (NUM_PROC/2) ; i++ )
    if ( lista_buddy[i].processo == pid )
       {
       lista_buddy[i].alocado  = 0;
       lista_buddy[i].processo = 0;
       pid_ok += 1;
       p_aloc --;
       }

if (pid_ok > 0)
   return(TRUE);
else
   return(FALSE);
}
```

A-2.1.4 mgc_par_M.c

```
/* Algoritmo MGC - Versao Distribuida */
/* By De Rose   (MESTRE)              */

#define GRAU_CUBO 8
#define NUM_PROC  256

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <sys/time.h>

#define SOCKET_S  2222  /* numero da socket que conecta */
                        /* com o escravo                */
#define SOCKET_C  2223  /* numero da socket que conecta */
                        /* com o coordenador            */
#define P  1
#define V  2
#define EOT 0

#define TRUE     1
#define FALSE    0

/* prototipos */

int Criosock_conecto(); /* maquina onde peco conexao */
                        /* e porta                   */

int  MGC_aloc   ();  /* int numero , int pid */
int  MGC_dealoc ();  /* int pid              */
void MGC_init   ();  /* void                 */
void MGC_mostra ();  /* void                 */

void MGC_gera_brgc(); /* int * array */
int  MGC_fat();       /* int n */


/* variaveis globais */

int slave_sock,coord_sock;
int arg;
unsigned char token[3];
unsigned char tok[3];

struct nodo_MGC {
       int node[NUM_PROC];
       int processo[NUM_PROC];
       int alocado[NUM_PROC];
       };

struct nodo_alocado {
       int proc;
       int pid;
       };

struct nodo_MGC    lista_MGC   [NUM_PROC];
struct nodo_alocado alocados[NUM_PROC];
int ft[GRAU_CUBO];
```

```
int p_aloc = 0; /* processadores ja alocados */
int grays;      /* numero de gray codes utilizados */

/* argv[1] - maquina onde se encontra o escravo     */
/* argv[2] - maquina onde se encontra o coordenador */

main(argc,argv)

int argc; char *argv[];
{
 unsigned int i;
 struct timeval      tp;
 struct timezone     tz;
 double              tmp,
                     tempo_inicial, tempo_final,
                     tempo_total;
int numero;
int ret_code;

/* inicializo socket */

token[0] = 0;

/* peco conexao SOCKET_C para o coord que ja espera */

coord_sock = Criosock_conecto(argv[1],SOCKET_C);

/* peco conexao SOCKET_S para o escravo que ja espera */

slave_sock = Criosock_conecto(argv[2],SOCKET_S);

/* conexao com escravo estabelecida */

printf("\n(M) Comunicacao estabelecida !!!\n");

MGC_init();

printf("\n(S) Codigos Inicializados !!!\n");

/* pego tempo inicial */

gettimeofday ( &tp, &tz);
tmp      = tp.tv_usec;
tempo_inicial =  ( tmp / 1000000) + tp.tv_sec;

/* Efetuo Alocacao */

ret_code = MGC_aloc(4,9);

if (ret_code > 0)
   printf("\n(M) Aloquei os processadores.\n");
else
   printf("\n(M) Alocacao falhou !!!\n");

MGC_mostra();

ret_code = MGC_dealoc(9);

if (ret_code > 0)
   printf("\n(M) Desaloquei os processadores.\n");
else
   printf("\n(M) Desalocacao falhou. Pid nao
           encontrado !!!\n");

MGC_mostra();

/* pego tempo final */

gettimeofday ( &tp, &tz);
tmp       = tp.tv_usec;
tempo_final  = ( tmp / 1000000) + tp.tv_sec;

/* calculo diferenca */

tempo_total = tempo_final - tempo_inicial;
printf("\n(M) Tempo total = %f\n",tempo_total);

/* desativo escravo */

printf("\n(M) Desativo escravo. \n");

token[0] = 3;
write(slave_sock,token,1);

/* desativo coordenador */

printf("\n(M) Desativo coordenador. \n");

token[0] = EOT;
write(coord_sock,token,1);

/* fecho sockets */

close(slave_sock);
close(coord_sock);

printf("\n(M) Cortei conexao !!!\n");
}

int Criosock_conecto(host,porta)
char *host;
int porta;

{
 struct sockaddr_in server;
 struct hostent *hp, *gethostbyname();
 int hand;
int flag =1;

/* crio socket */

if ( (hand = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
   {
   perror("Opening stream socket");
   exit(1);
   }

setsockopt(hand,SOL_SOCKET,SO_REUSEADDR,(char *)
          &flag,sizeof flag);

/* tento conexao com maquina remota */

 server.sin_family = AF_INET;
 if ( (hp = (struct hostent *) gethostbyname(host)) == 0)
    {
    fprintf(stderr,"%s: unknownhost\0", host);
    exit(2);
    }
 bcopy( (char *)hp->h_addr, (char *)&server.sin_addr,
       hp->h_length);
 server.sin_port = porta;

if (connect(hand, (struct sockaddr *) &server,
    sizeof(server)) < 0)
    {
    perror("connect");
    exit(1);
    }

return(hand);
}

/* rotinas MGC */

void MGC_init()
{
/* inicializa lista MGC */

typedef int GC[NUM_PROC];
GC l_brgc[NUM_PROC];
int i,j;
int dist,masc;


grays = MGC_fat(GRAU_CUBO) / ( MGC_fat(GRAU_CUBO/2) *
        MGC_fat(GRAU_CUBO-(GRAU_CUBO/2)) );
p_aloc = 0;

for ( j=0 ; j< (grays) ; j+=2 )
    {
    /* inicializa ft */

    masc = 1;
    dist = j;
    for (i=0 ; i < GRAU_CUBO; i++)
        {
        ft[i] = ((dist)&(masc));
        masc<<=1;
        }

    MGC_gera_brgc(l_brgc[j/2]);
    }


for ( i = 0 ; i < NUM_PROC ; i++ )
    for (j =0 ; j< (grays/2) ; j++)
        {
        lista_MGC[i].node[j] = l_brgc[j][i];
        lista_MGC[i].alocado[j] = 0;
        lista_MGC[i].processo[j] = 0;
        }
}

void MGC_mostra()
{
```

```
/* mostra lista MGC */

int i,j;

printf("\n          Lista MGC\n\n");
for ( i = 0 ; i < NUM_PROC ; i++ )
    {
    printf(" [%2d]   p: %2d    a:%2d -- "
           ,lista_MGC[i].node[0],
            lista_MGC[i].processo[0],
            lista_MGC[i].alocado[0]);

    for (j=1 ; j< (grays/2) ; j++ )
        printf("%2d(%d) "
          ,lista_MGC[i].node[j],lista_MGC[i].alocado[j]);

    printf("\n");
    }

printf("\n\t %d processadore(s) alocado(s).\n",p_aloc);
}

int  MGC_aloc   ( numero , pid )

int numero,
    pid;
{
/* aloco hipercubo com "numero" processadores para pid */
/* retorno um numero positivo se operacao OK           */

int aloc_ok;
int bit_ok;
int i,j,h,t2,m,dim, pot;
float t1;
int set = 0; /* gray code que estou usando */
int aux_t1,indice;

/* testo se existem processadores disponiveis */

if (numero > NUM_PROC-p_aloc) return FALSE;

/* disparo procura remota */

token[0] = 1;
token[1] = numero;
token[2] = pid;
token[3] = 0;

printf("Envio num = %d pid = %d para alocar !\n",
        token[1],token[2]);

write(slave_sock,token,3);

/* encontro menor dimensao de cubo que contenha */
/* "numero" processadores                       */

dim = 0;
t1 = 1;

while ( t1 < numero )
   {
   dim++;
   t1 *= 2;
   }

aux_t1 = t1;

/* procuro menor "m" que satisfaca a equacao MGC */

set = 0;
aloc_ok = 0;

while( (set< (grays/2) )&&(!aloc_ok) )
   {
   m = 0;
   t1 = aux_t1;
   t1/=2;
   aloc_ok = 0;

   while (  ((((m+2)*t1)-1) < NUM_PROC) && !(aloc_ok) )
      {
      bit_ok = 1;
      t2 = m*t1;
      while ( (t2 <= (((m+2)*t1)-1)) && (bit_ok) )
         {
        if ( lista_MGC[t2].alocado[set] != 0 )
            bit_ok = 0;
         t2++;
         }

      if ( bit_ok )
         aloc_ok = 1;
      else
            m++;
         }
      set++;
      }

set--;  /* acertar set, executou uma vez a mais */

/* caso encontrado "m", aloco processadores em todos */
/* os sets                                           */

if ( aloc_ok )
   {
   /* faco consulta a coordenador - secao critica */

   token[0] = P;
   write(coord_sock,token,1);

   printf("\n(M) Testando coordenador sobre SC.\n");

   /* espero resposta */

   read(coord_sock,tok,1);

   printf("\n(M) Resposta coord Byte = %d\n.",tok[0]);

   if (tok[0])
      {
      indice = 0;
      for ( i = m*t1 ; i <= (((m+2)*t1)-1) ; i++ )
          {
          /* atualizo array para enviar remoto */

          alocados[indice].proc = i;
          alocados[indice] .pid = pid;
          indice++;

          lista_MGC[i].alocado[set]  = 1;
          lista_MGC[i].processo[set] = pid;
          p_aloc+=1;

          /* nos outros sets pelo indice */

          for ( j=0 ; j< (grays/2) ; j++)
              for (h=0 ; h< NUM_PROC ; h++ )
                 if ( lista_MGC[i].node[set] ==
                      lista_MGC[h].node[j] )
                    {
                    lista_MGC[h].alocado[j]  = 1;
                    lista_MGC[h].processo[j] = pid;
                    }

          }

       }
   }

if (tok[0])
   {
   printf("\nLocal alocou legal (pid=%d) !!!\n",pid);

   /* aviso escravo que aloquei */

   token[0] = 1;
   write(slave_sock,token,1);

   /* envio lista de procs aloc para remoto atualizar */

   printf("\n Envio lista procs alocados para
           remoto !!!\n");

   for (j=0 ; j< numero ; j++ )
       {
       token[0] = alocados[j].proc;
       token[1] = alocados[j].pid;
       write(slave_sock,token,2);
       }

   }
else
   {
   printf("\nLocal NAO alocou legal (pid=%d) !!!\n",pid);

   /* aviso escravo que nao aloquei */

   token[0] = 0;
   write(slave_sock,token,1);
   }

/* recebo ok ou negado do remoto */

read(slave_sock,tok,1);

if (tok[0])
```

```
    {
    aloc_ok = 1;
    printf("\nRemoto encontrou !!!\n");

    /* recebo lista de procs alocados pelo remoto para */
    /* atualizacao                                     */

    printf("\n Recebo lista procs alocados remoto !!!\n");

    for (j=0 ; j< numero ; j++ )
        {
        read(slave_sock,tok,2);
        alocados[j].proc = tok[0];
        alocados[j].pid  = tok[1];
        }

    /* atualizo estrutura local */

    for ( i = 0; i < numero ; i++ )
        {
        lista_MGC[alocados[i].proc].alocado[set]  = 1;
        lista_MGC[alocados[i].proc].processo[set] =
                                    alocados[i].pid;

        /* nos outros sets pelo indice */

        for ( j=0 ; j< grays/2 ; j++)
            for (h=0 ; h< NUM_PROC ; h++ )
               if ( lista_MGC[alocados[i].proc].
                    node[set] ==
                    lista_MGC[h].node[j] )
                  {
                  lista_MGC[h].alocado[j]  = 1;
                  lista_MGC[h].processo[j] =
                              alocados[i].pid;
                  }

        }
    }
else
    printf("\nRemoto NAO encontrou !!!\n");

/* libero coordenador - secao critica */

printf("\nLibero Secao critica !!!\n");
token[0] = V;
write(coord_sock,token,1);

return(aloc_ok);
}

int  MGC_dealoc ( pid )

int pid;
{
/* libero processadores alocados pelo processo pid */
/* retorno numero de processadores desalocados     */
/* tenho que desalocar em todos os sets */

int i,j;
int pid_ok = 0;
int temp1 =0 ;

/* envio pid para nodo remoto */

token[0] = 2;
token[1] = pid;
token[2] = 0;

printf("Envio pid para remoto desalocar = %d !\n",
        token[1]);

write(slave_sock,token,3);

/* nao preciso esperar resposta */

for ( i = 0 ; i < NUM_PROC ; i++ )
    for ( j=0 ; j< (grays/2) ; j++ )
        if ( lista_MGC[i].processo[j] == pid )
          {
          lista_MGC[i].alocado[j]  = 0;
          lista_MGC[i].processo[j] = 0;
          temp1++;
          }

p_aloc-= temp1/(grays/2);
pid_ok-= temp1/(grays/2);

if ( temp1 > 0 )
   return(TRUE);
else
   return(FALSE);
}

void MGC_gera_brgc(array)

int * array;

{
int i,n;
int proc = NUM_PROC;
int ind_e,ind_l;
unsigned char pot;

/* inicializo */

*(array+proc-2) = 0;
*(array+proc-1) = 1;
n = 2;
pot = 1;

for ( i=1 ; i < GRAU_CUBO ; i++)
    {
        pot<<=1;
   ind_l = proc-n;
   ind_e = proc - n * 2;

   while( ind_l < proc )
      {
      if ( ind_l % 2 == 0 )
         {
                        if ( ft[i] == 0 )
                           {
            *(array+ind_e) = *(array+ind_l)<<1;
            ind_e++;
            *(array+ind_e) = (*(array+ind_l)<<1)+1;
            ind_e++;
                           }
                        else
                           {
            *(array+ind_e) = *(array+ind_l);
            ind_e++;
            *(array+ind_e) = (*(array+ind_l)|pot);
            ind_e++;
                           }
         }
      else
                {
                        if ( ft[i] == 0 )
                           {
            *(array+ind_e) = (*(array+ind_l)<<1)+1;
            ind_e++;
            *(array+ind_e) = *(array+ind_l)<<1;
            ind_e++;
                           }
                        else
                           {
                           *(array+ind_e) =
                            (*(array+ind_l)|pot);
            ind_e++;
            *(array+ind_e) = *(array+ind_l);
            ind_e++;
                           }
         }
      ind_l++;
      }
   n*=2;
    }
}

int  MGC_fat(n)

int n;
{
/* calculo recursivamente o fatorial de n */

if ( n==1 )
   return(1);
else
   return(n*MGC_fat(n-1));
}
```

A-2.1.5 mgc_par_S.c

```
/* Algoritmo MGC - Versao Distribuida */
/* By De Rose  (ESCRAVO)              */

#define GRAU_CUBO 8
```

```
#define NUM_PROC  256

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>

#define SOCKET_M  2222 /* numero socket que conecta */
                       /* com o escravo             */
#define SOCKET_C  2221 /* numero socket que conecta */
                       /* com o coordenador         */
#define P  1

#define TRUE     1
#define FALSE    0

/* prototipos */

int Criosock_bind_aceito(); /* parametros: porta     */
int Criosock_conecto();     /* parametros: maquina e */
                            /* porta                 */

int  MGC_aloc   ();   /* int numero , int pid */
int  MGC_dealoc ();   /* int pid              */
void MGC_init   ();   /* void                 */
void MGC_mostra ();   /* void                 */

void MGC_gera_brgc();  /* int * array */
int  MGC_fat();        /* int n       */

/* variaveis globais */

int master_sock,coord_sock;
int arg;
unsigned char token[3];
unsigned char tok[3];
int numero;

struct nodo_MGC {
       int node[NUM_PROC];
       int processo[NUM_PROC];
       int alocado[NUM_PROC];
       };

struct nodo_alocado {
       int proc;
       int pid;
       };

struct nodo_MGC    lista_MGC   [NUM_PROC];
struct nodo_alocado alocados[NUM_PROC];
int ft[GRAU_CUBO];

int p_aloc = 0; /* processadores ja alocados       */
int grays;      /* numero de gray codes utilizados */

/* argv[1] - maquina onde se encontra o mestre      */
/* argv[2] - maquina onde se encontra o coordenador */

main(argc,argv)

int argc; char *argv[];
{
int ret_code;
int pid,end,j;

token[0] = 0;

/* aceito cenexao do mestre */

master_sock = Criosock_bind_aceito(SOCKET_M);

/* tento conexao com o coordenador */

coord_sock = Criosock_conecto(argv[1],SOCKET_C);

printf("\n(S) Conexao estabelecida !!!\n");

MGC_init();

printf("\n(S) Codigos Inicializados !!!\n");

/* atendo pedidos do mestre */

end = 0;

while (!end)
   {

printf("\n(S) Espero mensagem.\n");

tok[0] = 0;
read(master_sock,tok,3);

/* processo mesangem */

switch( tok[0] )
   {
   case 1:      /* efetuo alocacao */

        printf("\n(S) Chegou pedido Aloc !!!\n");

        /* acerto parametros */

        numero = tok[1];
        pid = tok[2];

        ret_code = MGC_aloc(numero,pid);

        if (ret_code > 0)
           {
           printf("\n(S) Aloquei [%d]
                   processadores.\n",ret_code);
           token[0] = 1;
           write(master_sock,token,1);

           MGC_mostra();
           }
        else
           {
           printf("\n(S) Alocacao falhou !!!\n");
           token[0] = 0;
           write(master_sock,token,1);
           }

       break;

   case 2:      /* efetuo desalocacao */

        printf("\n(S) Chegou pedido Desaloc !!!\n");

        pid = tok[1];

        ret_code = MGC_dealoc(pid);

        if (ret_code > 0)
           {
           printf("\n(S) Desaloquei [%d]
                   processadores.\n",ret_code);
           token[0] = 1;
           write(master_sock,token,1);
           }
        else
           {
           printf("\n(S) Desalocacao falhou !!!\n");
           token[0] = 0;
           write(master_sock,token,1);
           }

        break;

   case 3:      /* corto conexao, saindo do loop */

        end = 1;
        printf("\nSai do Loop (mesg 3) !!!\n");

        MGC_mostra ();
    }

   }

/* fecho sockets */

close(master_sock);
close(coord_sock);

printf("\n(S) Corto Conexao.\n");
}

int Criosock_bind_aceito(porta)
int porta;

{
struct sockaddr_in server;
int hand;
int flag = 1;

/* crio socket */

if ( (hand = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
   {
   perror("Opening stream socket");
   exit(1);
   }
```

```
setsockopt(hand,SOL_SOCKET,SO_REUSEADDR,(char *)
          &flag,sizeof flag);

/* faco bind */

 server.sin_family = AF_INET;
 server.sin_addr.s_addr = INADDR_ANY;
 server.sin_port = porta;

 if (bind(hand,(struct sockaddr *) &server,
         sizeof(server)) < 0)
    {
    perror("Binding stream socket");
    exit(1);
    }

/* listen especifica o numero de conexoes possiveis */

listen(hand,1);

/* accept extrai primeira conexao da lista de */
/* conexoes pendentes                         */

 hand = accept(hand, (struct sockaddr *) 0, (int *) 0);

 if (hand == -1)
    {
    perror("accept");
    exit(1);
    }

return(hand);
}

int Criosock_conecto(host,porta)
char *host;
int porta;

{
int hand;
int flag = 1;

struct sockaddr_in server;
struct hostent *hp, *gethostbyname();

/* crio socket */

if ( (hand = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
   {
   perror("Opening stream socket");
   exit(1);
   }

setsockopt(hand,SOL_SOCKET,SO_REUSEADDR,(char *)
          &flag,sizeof flag);

/* tento conexao com maquina remota */

 server.sin_family = AF_INET;
 if ( (hp = (struct hostent *) gethostbyname(host))
         == 0)
    {
    fprintf(stderr,"%s: unknownhost\0", host);
    exit(2);
    }
 bcopy( (char *)hp->h_addr, (char *)&server.sin_addr,
       hp->h_length);
 server.sin_port = porta;

if (connect(hand, (struct sockaddr *) &server,
                  sizeof(server)) < 0)
    {
    perror("connect");
    exit(1);
    }

return(hand);
}

/* rotinas MGC */

void MGC_init()
{
/* inicializa lista MGC */

typedef int GC[NUM_PROC];
GC l_brgc[NUM_PROC];
int i,j;
int dist,masc;


grays = MGC_fat(GRAU_CUBO) / ( MGC_fat(GRAU_CUBO/2) *
        MGC_fat(GRAU_CUBO-(GRAU_CUBO/2)) );
p_aloc = 0;

for ( j= grays ; j< (grays) ; j+=2 )
    {
    /* inicializa ft */

    masc = 1;
    dist = j;
    for (i=0 ; i < GRAU_CUBO; i++)
        {
        ft[i] = ((dist)&(masc));
        masc<<=1;
        }

    MGC_gera_brgc(l_brgc[j/2]);
    }


for ( i = 0 ; i < NUM_PROC ; i++ )
    for (j =0 ; j< (grays/2) ; j++)
        {
        lista_MGC[i].node[j] = l_brgc[j][i];
        lista_MGC[i].alocado[j] = 0;
        lista_MGC[i].processo[j] = 0;
        }
}

void MGC_mostra()
{
/* mostra lista MGC */

int i,j;

printf("\n         Lista MGC\n\n");
for ( i = 0 ; i < NUM_PROC ; i++ )
    {
    printf(" [%2d]   p: %2d    a:%2d -- ",
        lista_MGC[i].node[0],lista_MGC[i].processo[0],
        lista_MGC[i].alocado[0]);

    for (j=1 ; j< (grays/2) ; j++ )
        printf("%2d(%d) ",lista_MGC[i].node[j],
                          lista_MGC[i].alocado[j]);

    printf("\n");
    }

printf("\n\t %d processadore(s) alocado(s).\n",p_aloc);
}

int  MGC_aloc   ( numero , pid )

int numero,
    pid;
{
/* aloco um hipercubo com "numero" processadores */
/* para pid; retorno um numero positivo se a     */
/* operacao  obter sucesso                       */

int aloc_ok;
int bit_ok;
int i,j,h,t2,m,dim, pot;
float t1;
int set = 0; /* gray code que estou usando */
int aux_t1,indice;

/* encontro menor dimensao de cubo que contenha */
/* "numero" processadores                       */

dim = 0;
t1 = 1;

while ( t1 < numero )
   {
   dim++;
   t1 *= 2;
   }

aux_t1 = t1;

/* procuro menor "m" que satisfaca a equacao MGC */

set = 0;
aloc_ok = 0;

while( (set< (grays/2) )&&(!aloc_ok) )
   {
   m = 0;
   t1 = aux_t1;
   t1/=2;
   aloc_ok = 0;
```

```
   while (  ((((m+2)*t1)-1) < NUM_PROC) && !(aloc_ok) )
      {
      bit_ok = 1;
      t2 = m*t1;
      while ( (t2 <= (((m+2)*t1)-1)) && (bit_ok) )
         {
        if ( lista_MGC[t2].alocado[set] != 0 )
            bit_ok = 0;
         t2++;
         }

      if ( bit_ok )
         aloc_ok = 1;
     else
         m++;
     }
   set++;
   }

set--;  /* acertar set, executou uma vez a mais */

/* caso encontrado "m", aloco processadores em todos */
/* os sets                                           */

if ( aloc_ok )
   {
   /* faco consulta a coordenador - secao critica */

   token[0] = P;
   write(coord_sock,token,1);

   printf("\n(S) Testando coordenador sobre SC.\n");

   /* espero resposta */

   read(coord_sock,tok,1);

   printf("\n(S) Resposta coord Byte = %d\n.",tok[0]);

   if (tok[0])
     {
     indice = 0;

     for ( i = m*t1 ; i <= (((m+2)*t1)-1) ; i++ )
         {
         /* atualizo array para enviar remoto */

         alocados[indice].proc = i;
         alocados[indice] .pid = pid;
         indice++;

         lista_MGC[i].alocado[set]  = 1;
         lista_MGC[i].processo[set] = pid;
         p_aloc+=1;

         /* nos outros sets pelo indice */

         for ( j=0 ; j< (grays/2) ; j++)
            for (h=0 ; h< NUM_PROC ; h++ )
                if ( lista_MGC[i].node[set] ==
                     lista_MGC[h].node[j] )
                 {
                 lista_MGC[h].alocado[j]  = 1;
                 lista_MGC[h].processo[j] = pid;
                 }
         }

     }
   }

if (tok[0])
   {
   printf("\n(S) Local alocou legal (pid=%d) !!!\n",pid);

   /* envio lista de procs alocados para remoto */
   /* atualizar                                 */

   printf("\n Envio lista procs alocados para remoto
           !!!\n");

   for (j=0 ; j< numero ; j++ )
       {
       token[0] = alocados[j].proc;
       token[1] = alocados[j].pid;
       write(master_sock,token,2);
       }

   return(aloc_ok);
   }
else
   {
   printf("\n(S) Local NAO alocou legal (pid=%d)
           !!!\n",pid);

   /* verifico se mestre alocou */

   read(master_sock,tok,1);

   if (tok[0])
      {
      printf("\nMestre encontrou !!!\n");

      /* recebo lista de procs alocados pelo remoto */
      /* para atualizacao                           */

      printf("\n Recebo lista procs alocados
              remoto !!!\n");

      for (j=0 ; j< numero ; j++ )
          {
          read(master_sock,tok,2);
          alocados[j].proc = tok[0];
          alocados[j].pid  = tok[1];
          }

       /* atualizo estrutura local */

       for ( i = 0; i < numero ; i++ )
          {
          lista_MGC[alocados[i].proc].alocado[set]  = 1;
          lista_MGC[alocados[i].proc].processo[set] =
                                  alocados[i].pid;

          /* nos outros sets pelo indice */

          for ( j=0 ; j< grays/2 ; j++)
             for (h=0 ; h< NUM_PROC ; h++ )
                 if ( lista_MGC[alocados[i].proc].
                      node[set] ==
                      lista_MGC[h].node[j] )
                    {
                    lista_MGC[h].alocado[j]  = 1;
                    lista_MGC[h].processo[j] =
                                  alocados[i].pid;
                    }
          }
       }
     else
       printf("\nMestre NAO encontrou !!!\n");

   return(FALSE);
   }

}

int  MGC_dealoc ( pid )

int pid;
{
/* libero processadores alocados pelo processo pid */
/* retorno numero de processadores desalocados     */
/* tenho que desalocar em todos os sets            */

int i,j;
int pid_ok = 0;
int templ =0 ;

for ( i = 0 ; i < NUM_PROC ; i++ )
    for ( j=0 ; j< (grays/2) ; j++ )
        if ( lista_MGC[i].processo[j] == pid )
          {
          lista_MGC[i].alocado[j]  = 0;
          lista_MGC[i].processo[j] = 0;
          templ++;
          }

p_aloc-= templ/(grays/2);
pid_ok-= templ/(grays/2);

if ( templ > 0 )
   return(TRUE);
else
   return(FALSE);
}

void MGC_gera_brgc(array)

int * array;

{
int i,n;
int proc = NUM_PROC;
int ind_e,ind_l;
unsigned char pot;

/* inicializo */
```

```
*(array+proc-2) = 0;
*(array+proc-1) = 1;
n = 2;
pot = 1;

for ( i=1 ; i < GRAU_CUBO ; i++)
    {
        pot<<=1;
   ind_l = proc-n;
   ind_e = proc - n * 2;

   while( ind_l < proc )
      {
      if ( ind_l % 2 == 0 )
         {
                        if ( ft[i] == 0 )
                           {
            *(array+ind_e) = *(array+ind_l)<<1;
            ind_e++;
            *(array+ind_e) = (*(array+ind_l)<<1)+1;
            ind_e++;
                           }
                        else
                           {
            *(array+ind_e) = *(array+ind_l);
            ind_e++;
            *(array+ind_e) = (*(array+ind_l)|pot);
            ind_e++;
                           }
         }
      else
                  {
                        if ( ft[i] == 0 )
                           {
            *(array+ind_e) = (*(array+ind_l)<<1)+1;
            ind_e++;
            *(array+ind_e) = *(array+ind_l)<<1;
            ind_e++;
                           }
                        else
                           {
                           *(array+ind_e) =
                            (*(array+ind_l)|pot);
            ind_e++;
            *(array+ind_e) = *(array+ind_l);
            ind_e++;
            }
         }
      ind_l++;
      }
   n*=2;
    }
}

int  MGC_fat(n)

int n;
{
/* calculo recursivamente o fatorial de n */

if ( n==1 )
   return(1);
else
   return(n*MGC_fat(n-1));
}
```

A-2.1.6 tc_par_M.c

```
/* Algoritmo tc - Versao Distribuida */
/* By De Rose   processo MESTRE      */
/*        VERSAO PRELIMINAR          */

#define NUM_PROC 16

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <sys/time.h>

#define SOCKET_S  2222 /* numero da porta que conecta */
                       /* com o escravo               */
#define SOCKET_C  2223 /* numero da porta que conecta */
                       /* com o coordenador           */
#define P  1
#define V  2
#define EOT 0

#define TRUE     1
#define FALSE    0

/* prototipos */

int Criosock_conecto();   /* envio maquina escravo */
                          /* e porta, retorna hand */

int  TC_sub_collapse();  /* int n,int k,int pre_level, */
                         /* int pre_step,char *masc,   */
                         /* int pid                    */
int  TC_R();             /* char * masc,int i          */
int  TC_aloca_sub();     /* char masc,int grau,int pid */
int  TC_aloca_todo();    /* int pid                    */

int  TC_aloc   ();       /* int numero , int pid */
int  TC_dealoc ();       /* int pid              */
void TC_init   ();       /* void                 */
void TC_mostra ();       /* void                 */

void TC_inc_bit();       /* int pos            */
void TC_seta_bit();      /* int pos,int val    */
int  TC_le_bit();        /* int pos            */
int  TC_procura();       /* int * bits,int pid */
int  TC_pertence();      /* int * bits,int i   */

/* dados globais TC */

struct nodo_TC {
       int processo;
       int alocado;
       };

struct nodo_TC lista_TC[NUM_PROC];
int cont[GRAU_CUBO]; /* 0-3 */

int slave_sock,coord_sock;
int arg;
unsigned char token[3];
unsigned char tok[3];

int p_aloc = 0;          /* processadores ja alocados */


/* argv[1] - maquina onde se encontra o escravo     */
/* argv[2] - maquina onde se encontra o coordenador */

main(argc,argv)

int argc; char *argv[];
{
 unsigned int i;
 struct timeval      tp;
 struct timezone     tz;
 double              tmp,
                     tempo_inicial, tempo_final,
                     tempo_total;
int numero;
int ret_code;

/* inicializo socket */

token[0] = 0;

/* peco conexao SOCKET_C para o coordenador  */
/* que ja espera                             */

coord_sock = Criosock_conecto(argv[1],SOCKET_C);

/* peco conexao SOCKET_S para o escravo que ja espera */

slave_sock = Criosock_conecto(argv[2],SOCKET_S);

/* conexao com escravo estabelecida */

printf("\n(M) Comunicacao estabelecida !!!");

buddy_init();

/* pego tempo inicial */

gettimeofday ( &tp, &tz);
tmp      = tp.tv_usec;
tempo_inicial =  ( tmp / 1000000) + tp.tv_sec;

/* Efetuo Alocacao */

TC_init();
```

```
ret_code = TC_aloc(2,1);

if (ret_code > 0)
   printf("\n(M) Aloquei os processadores.");
else
   printf("\n(M) Alocacao falhou !!!");

TC_mostra();

ret_code = TC_dealoc(1);

if (ret_code > 0)
   printf("\n(M) Desaloquei os processadores.");
else
   printf("\n(M) Desalocacao falhou. Pid nao encontrado
           !!!");
TC_mostra();

/* pego tempo final */

gettimeofday ( &tp, &tz);
tmp       = tp.tv_usec;
tempo_final  = ( tmp / 1000000) + tp.tv_sec;

/* calculo diferenca */

tempo_total = tempo_final - tempo_inicial;
printf("\n(M) Tempo total = %f\n",tempo_total);

/* desativo escravo */

printf("\n(M) Desativo escravo.");

token[0] = 3;
write(slave_sock,token,1);

/* desativo coordenador */

printf("\n(M) Desativo coordenador.");

token[0] = EOT;
write(coord_sock,token,1);

/* fecho sockets */

close(slave_sock);
close(coord_sock);

printf("\n(M) Cortei conexao !!!\n");
}

int Criosock_conecto(host,porta)
char *host;
int porta;

{
 struct sockaddr_in server;
 struct hostent *hp, *gethostbyname();
 int hand;
 int flag = 1;

/* crio socket */

if ( (hand = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
   {
   perror("Opening stream socket");
   exit(1);
   }

setsockopt(hand,SOL_SOCKET,SO_REUSEADDR,(char *)
           &flag,sizeof flag);

/* tento conexao com maquina remota */

 server.sin_family = AF_INET;
 if ( (hp = (struct hostent *) gethostbyname(host))
        == 0)
    {
    fprintf(stderr,"%s: unknownhost\0", host);
    exit(2);
    }
 bcopy( (char *)hp->h_addr, (char *)&server.sin_addr,
        hp->h_length);
 server.sin_port = porta;

if (connect(hand, (struct sockaddr *) &server,
    sizeof(server)) < 0)
    {
    perror("connect");
    exit(1);
    }

return(hand);
}

/* rotinas tc */

int TC_R(masc,i)

unsigned int *masc;
int i;   '
{
/* efetuo a operacao R(rotate) a partir do bit i do */
/* byte enderecado por a: o bit mais significativo  */
/* de *(masc) e' o bit 0, o menos significativo o 7 */

unsigned int b,c;
int p,m;

if (i>GRAU_CUBO-2)
   return(FALSE);

/* salvo bit 0 de a em b */

b = *(masc);
b &= 1;

if (i>0)
   {
   /* efetuo mascara */

   c = *(masc)>>(GRAU_CUBO-i);
   c <<= (GRAU_CUBO-i);
   }

/* efetuo deslocamento */

*(masc) >>= 1;

/* recoloco bit 0 */

p = (int) pow((double)2,(double)((GRAU_CUBO-1)-i));

if (b)
   *(masc) |= p;
else
   *(masc) &= 255 - p;

/* recoloco mascara */

b =  (int) pow((double)2,(double)(GRAU_CUBO-1));
for ( m=0 ; m<i  ; m++)
    {
    /* acerto bits mais significativos de *masc */
    /* em c estao os i bits mais significativos */
    /* a serem copiados para masc               */

    if ( (c&b) > 0 )
       *(masc) |= (c&b);
    else
       *(masc) &= 255-b;

    b>>=1;
    }

return(TRUE);
}

int TC_aloca_sub ( masc, grau , pid )

unsigned int masc;
int grau,
    pid;
{
int i;
int suc;

/* conversao de mascara para bits */

for ( i=0 ; i < GRAU_CUBO ; i++ )
    if ( (masc & (int)pow((double)2,(double)i)) > 0  )
       cont[i] =  0;
    else
       cont[i] = -1;

/* vario alguns bits de cont segundo sequencia bit */

for (i=0 ; i < (int) pow((double)2,(double)
    (GRAU_CUBO-grau)) ; i++)
    {

    /* efetuo a procura */

    suc = TC_procura(cont,pid);
    if ( suc )
       return(TRUE);
```

```
    TC_inc_bit(1);
    if ( TC_le_bit(1) == 2)
       {
       TC_seta_bit(1,0);
       TC_inc_bit(2);

       if ( TC_le_bit(2) == 2 )
          {
          TC_seta_bit(2,0);
          TC_inc_bit(3);

          if ( TC_le_bit(3) == 2 )
             {
             TC_seta_bit(3,0);
             TC_inc_bit(4);

             if ( TC_le_bit(4) == 2 )
                {
                TC_seta_bit(4,0);
                TC_inc_bit(5);

                if ( TC_le_bit(5) == 2 )
                   {
                   TC_seta_bit(5,0);
                   TC_inc_bit(6);
                   }
           }
             }
          }
       }
    }
return(FALSE);
}

int  TC_dealoc ( pid )

int pid;
{
/* libero processadores alocados pelo processo pid */
/* retorno numero de processadores desalocados     */

int i;
int pid_ok = 0;

for ( i = 0 ; i < NUM_PROC ; i++ )
    if ( lista_TC[i].processo == pid )
       {
       lista_TC[i].alocado  = 0;
       lista_TC[i].processo = 0;
       pid_ok += 1;
       p_aloc--;
       }

return(pid_ok);
}

void TC_init()
{
/* inicializa lista TC */

int i;

p_aloc = 0;

for ( i = 0 ; i < NUM_PROC ; i++ )
    {
    lista_TC[i].processo = 0;
    lista_TC[i].alocado  = 0;
    }
}

void TC_mostra()
{
/* mostra lista TC */

int i;

printf("%d processadore(s) alocado(s).\n",p_aloc);

printf("\n          Lista TC\n\n");
for ( i = 0 ; i < NUM_PROC ; i++ )
    printf(" [%2d]   p: %2d    a:%2d \n",i,
            lista_TC[i].processo,lista_TC[i].alocado);
}

int  TC_aloc   ( numero , pid )

int numero,
    pid;

{
/* aloco numero processadores para processo pid */
/* retorna um numero positivo se a operacao for */
/* bem sucedida                                 */

unsigned int mascara;
int i,k,n;
int suc,dim,t1;
int m,aloc_ok,bit_ok,t2;

/* testo se existem processadores disponiveis */

if (numero > NUM_PROC-p_aloc) return FALSE;

/* encontro menor dimensao de cubo que contenha */
/* "numero" processadores                       */

dim = 0;
t1 = 1;

while ( t1 < numero )
   {
   dim++;
   t1 *= 2;
   }

/* acerto variaveis TC */

n = GRAU_CUBO;
k = dim;

if (k==0)
   {
   /* aloca primeiro nodo disponivel */

   for ( i=0 ; i < NUM_PROC ; i++ )
       if ( lista_TC[i].alocado == 0)
          {
          lista_TC[i].alocado = 1;
          lista_TC[i].processo = pid;
          p_aloc++;
          return(TRUE);
          }

   return(FALSE);
   }

if (k==n)
   {
   suc = TC_aloca_todo(pid);
   return(suc);
   }

/* primeiro procuro na primary tree - no nivel n-k */
/* algoritmo buddy */

/* procuro menor "m" que satisfaca a equacao buddy */

m = 0;
aloc_ok = 0;

while (  ((((m+1)*t1)-1) < NUM_PROC) && !(aloc_ok) )
   {
   bit_ok = 1;
   t2 = m*t1;
   while ( (t2 <= (((m+1)*t1)-1)) && (bit_ok) )
      {
      if ( lista_TC[t2].alocado != 0 )
         bit_ok = 0;
      t2++;
      }

   if ( bit_ok )
      aloc_ok = 1;
   else
      m++;
}

/* caso encontrado "m", aloco processadores */

if ( aloc_ok )
   {
   for ( i = m*t1 ; i <= (((m+1)*t1)-1) ; i++ )
       {
       lista_TC[i].alocado  = 1;
       lista_TC[i].processo = pid;
       p_aloc += 1; /* mais um processador alocado */
       }

   return(TRUE);
   }

/* caso nao encontrado executo algoritmo G */

for ( i=n-k-1 ; i>=0 ; i-- )
```

```
    {
    /* reinicializo mascara a cada nova interacao */

    mascara = 255;        /* ********00000000 */
    mascara = mascara<<k;   /* shift left */
    mascara &= 255;

    /* printf("Mascara: [%d] !!!\n",mascara); */

    if ( !TC_R(&mascara,i) )
       printf("\nErro na operacao de rotate: [%d] rot
             [%d] !!!\n",mascara,i);

    /* printf("Mascara: [%d] !!!\n",mascara); */

    suc = TC_aloca_sub(mascara,k,pid);
    if (suc)
       return(TRUE);

    /* disparo remoto */

    if (odd(i))
       {
       /* envio para escravo */

       token[0] = mascara;
       token[1] = n;
       token[2] = pid;
       write(slave_sock,token,2);
       }
    else
       {
       suc = TC_sub_collapse(n,k,i,0,&mascara,pid);
       if (suc)
          return(TRUE);
       }
    }

return(FALSE);
}

int TC_sub_collapse(n,k,pre_level,pre_step,masc,pid)

int n,
    k,
    pre_level,
    pre_step,
    pid;
unsigned int *masc;
{
int step,i;
int suc;
unsigned int mascara;

step = pre_step + 1;
if (step >= k)
   return(FALSE);
else
   for ( i = pre_level ; i <= n-k-1 ; i++ )
       {
       mascara = *(masc);

       if ( !TC_R(&mascara,step+i) )
       printf("\nErro na operacao de rotate: [%d] rot
             [%d] !!!\n",mascara,i);

       /* printf("Mascara: [%d] !!!\n",mascara); */

       suc = TC_aloca_sub(mascara,k,pid);
       if (suc)
          return(TRUE);

       suc = TC_sub_collapse(n,k,i,step,&mascara,pid);
       if (suc)
          return(TRUE);
       }
return(FALSE);
}

int  TC_aloca_todo(pid)

int pid;
{
/* tenta alocar todo o cubo compartilhado   */
/* retorna positivo se ok, negativo se erro */

int i;

for ( i=0 ; i < NUM_PROC ; i++ )
    if ( lista_TC[i].alocado == 1 )
       return(FALSE);

/* todos os nodos estao livres, efetuo alocacao */

for ( i=0 ; i < NUM_PROC ; i++ )
    {
    lista_TC[i].alocado  = 1;
    lista_TC[i].processo = pid;
    p_aloc++;
    }

return(TRUE);
}

void TC_inc_bit(pos)

int pos;
{
int ex;
int i;

ex = 0;
for(i=0 ; i<GRAU_CUBO ; i++)
   {
   if ( cont[i] != -1 )
      {
      ex++;
      if (ex==pos)
         {
         cont[i]+=1;
         break;
         }
      }
   }
}

void TC_seta_bit(pos,val)

int pos,
    val;
{
int ex;
int i;

ex = 0;
for(i=0 ; i<GRAU_CUBO ; i++)
   {
   if ( cont[i] != -1 )
      {
      ex++;
      if (ex==pos)
         {
         cont[i] = val;
         break;
         }
      }
   }
}

int TC_le_bit(pos)

int pos;
{
int ex;
int i;

ex = 0;
for(i=0 ; i<GRAU_CUBO ; i++)
   {
   if ( cont[i] != -1 )
      {
      ex++;
      if (ex==pos)
         {
         return(cont[i]);
         }
      }
   }
return(FALSE);
}

int TC_procura(bits,pid)

int * bits;
int pid;
{
/* procuro todos os nodos com bits iguais aos */
/* diferentes de -1: se todo subcubo livre,   */
/* aloco e retorno positivo. senao retorno -1 */

int i;

/* varro cada nodo contido no subcubo masc */

for ( i=0 ; i < NUM_PROC ; i++ )
    {
```

```
    if ( TC_pertence(bits,i) == 1  )
       {
       /* nodo pertence ao sub_cubo masc */
       /* testo se sta livre             */

       if (lista_TC[i].alocado == 1)
          {
          /* nodo ocupado     */
          /* sub_cubo ocupado */

          return(FALSE);
          }
       }
    }

/* aloco nodos que pertencem ao sub_cubo */

for ( i=0 ; i < NUM_PROC ; i++ )
    {
    if ( TC_pertence(bits,i) == 1 )
       {
       /* aloco nodo */

       lista_TC[i].alocado  = 1;
       lista_TC[i].processo = pid;
       p_aloc++;
       }

    }

return(TRUE);
}

int  TC_pertence(bits,i)

int * bits;
int i;
{
/* verifico se nodo pertence a subcubo indicado */
/* nos bits[GRAU_CUBO]: se pertencer retorno 1,  */
/* senao retorno negativo                        */

int j;

for ( j=0 ; j < GRAU_CUBO ; j++ )
    if (bits[j] != -1)
       {
          if ((i & (int)pow((double)2,(double)j)) !=
             (bits[j]*(int) pow((double)2,(double)j)) )
             return(FALSE);
       }

return(TRUE);
}
```

A-2.1.7 tc_par_S.c

```
/* Algoritmo tc - Versao Distribuida */
/* By De Rose  processo (ESCRAVO)    */
/*         VERSAO PRELIMINAR         */

#define NUM_PROC 16

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>

#define SOCKET_M  2222 /* numero da porta que conecta */
                       /* com o escravo               */
#define SOCKET_C  2221 /* numero da porta que conecta */
                       /* com o coordenador           */
#define P  1

#define TRUE     1
#define FALSE    0

/* prototipos */

int Criosock_bind_aceito(); /* envio porta, recebo */
                            /* hand                */
int Criosock_conecto();     /* envio maquina e porta */
                            /* recebo hand           */

/* variaveis globais */

int master_sock,coord_sock;
int arg;
unsigned char token[3];
unsigned char tok[3];
int numero;


int p_aloc = 0;   /* processadores ja alocados */

/* argv[1] - maquina onde se encontra o coordenador */

main(argc,argv)

int argc; char *argv[];
{
int ret_code;
int pid,end;

token[0] = 0;

/* aceito cenexao do mestre */

master_sock = Criosock_bind_aceito(SOCKET_M);

/* tento conexao com o coordenador */

coord_sock = Criosock_conecto(argv[1],SOCKET_C);

printf("\n(S) Conexao estabelecida !!!");

/* inicializo algoritmo */

TC_init();

/* atendo pedidos do mestre */

end = 0;

while (!end)
   {

printf("\n(S) Espero mensagem.");

tok[0] = 0;
read(master_sock,tok,3);

/* processo mesangem */

switch( tok[0] )
   {
   case 1:      /* efetuo alocacao */

        printf("\n(S) Chegou pedido Aloc !!!");

        /* acerto parametros */

        numero = tok[1];
        pid = tok[2];

        ret_code = TC_aloc(numero,pid);
       if (ret_code > 0)
          {
          printf("\n(S) Aloquei os processadores.");
          token[0] = 1;
          write(master_sock,token,1);
          TC_mostra();
          }
        else
          {
          printf("\n(S) Alocacao falhou !!!");
          token[0] = 0;
          write(master_sock,token,1);
          }

        break;

   case 2:      /* efetuo desalocacao */

        printf("\n(S) Chegou pedido Desaloc !!!");

        pid = tok[1];

        ret_code = TC_dealoc(pid);

        if (ret_code > 0)
          {
          printf("\n(S) Desaloquei os processadores.");
          token[0] = 1;
          write(master_sock,token,1);
          }
```

```
        else
           {
           printf("\n(S) Desalocacao falhou !!!");
           token[0] = 0;
           write(master_sock,token,1);
           }

        break;

   case 3:      /* corto conexao, saindo do loop */

        end = 1;
        printf("\nSai do Loop (mesg 3) !!!");
        TC_mostra ();
    }

   }

/* fecho sockets */
close(master_sock);
close(coord_sock);

printf("\n(S) Corto Conexao.\n");
}

int Criosock_bind_aceito(porta)
int porta;

{
struct sockaddr_in server;
int hand;
int flag = 1;

/* crio socket */

if ( (hand = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
   {
   perror("Opening stream socket");
   exit(1);
   }

setsockopt(hand,SOL_SOCKET,SO_REUSEADDR,(char *)
           &flag,sizeof flag);

/* faco bind */

 server.sin_family = AF_INET;
 server.sin_addr.s_addr = INADDR_ANY;
 server.sin_port = porta;

 if (bind(hand,(struct sockaddr *) &server,
          sizeof(server)) < 0)
    {
    perror("Binding stream socket");
    exit(1);
    }

/* listen especifica o numero de conexoes possiveis */

listen(hand,1);

/* accept extrai primeira conexao da lista de */
/* conexoes pendentes                         */

 hand = accept(hand, (struct sockaddr *) 0, (int *) 0);

 if (hand == -1) {
    perror("accept");
    exit(1);
    }

return(hand);
}

int Criosock_conecto(host,porta)
char *host;
int porta;

{
int hand;
int flag = 1;

struct sockaddr_in server;
struct hostent *hp, *gethostbyname();

/* crio socket */

if ( (hand = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
   {
   perror("Opening stream socket");
   exit(1);
   }

setsockopt(hand,SOL_SOCKET,SO_REUSEADDR,(char *)
           &flag,sizeof flag);

/* tento conexao com maquina remota */

 server.sin_family = AF_INET;
 if ( (hp = (struct hostent *) gethostbyname(host))
       == 0)
    {
    fprintf(stderr,"%s: unknownhost\0", host);
    exit(2);
    }
 bcopy( (char *)hp->h_addr, (char *)&server.sin_addr,
        hp->h_length);
 server.sin_port = porta;

if (connect(hand, (struct sockaddr *) &server,
    sizeof(server)) < 0)
    {
    perror("connect");
    exit(1);
    }

return(hand);
}

/* rotinas tc */
```

A-2.2 Algoritmos para Transputers

Neste anexo é apresentado um exemplo de como fica a implementação das versões paralelas dos algoritmos de alocação de processadores em Transputers.

O arquivo `dist.cfg` é utilizado para configurar a alocação dos processos mestre e escravo na placa e indicar como devem ser conectados.

Como as ùnicas diferenças em relação ao código do anexo A-2.1 são as chamadas as funções de comunicação e as rotinas de tempo, foi incluído neste anexo apenas a implementação do algoritmo `budy_par` para exemplificar estas alterações.

A unidade de tempo medida pelas versões paralelas para a placa Transputer é *tick's* de CPU do processador mestre. A conversão para centésimos de segundo é feita através da divisão deste valor por 156,25 (1 *tick* de CPU de um processador Transputer rodando em baixa prioridade equivale a $64x10^{-6}$ segundos).

A-2.2.1 dist.cfg

```
Processor Host
Processor Root
Processor P1

Wire ? Host[0] Root[0]
Wire ? Root[2] P1[1]

Task Iserver  Ins=1 Outs=1
Task Filter   Ins=2 Outs=2 Data=10K
Task master   Ins=3 Outs=3 File="master.b8" Data=500K
Task slave    Ins=2 Outs=2 File="slave.b8"  Data=500k
Task mux      Ins=2 Outs=2 File="mux.b8"    Data=500k
Task coord    Ins=1 Outs=2 File="coord.b8"  Data=500k

Connect ? Iserver[0]  Filter[0]
Connect ? Filter[0]   Iserver[0]
Connect ? Filter[1]   master[1]
Connect ? master[1]   Filter[1]
Connect ? slave[0]    master[0]
Connect ? master[0]   slave[0]
Connect ? master[2]   mux[0]
Connect ? coord[0]    master[2]
Connect ? slave[1]    mux[1]
Connect ? coord[1]    slave[1]
Connect ? mux[0]      coord[0]

Place Iserver  Host
Place Filter   Root
Place master   Root
Place slave    P1
Place coord    P1
Place mux      P1
```

A-2.2.2 coord.c

```
/* Esqueleto para implementacao distribuida  */
/* dos algoritmos de alocacao    By De Rose  */
/*  Coordenador da secao critica (TRAM)      */

#include <stdio.h>
#include <chan.h>

#define P  1
#define V  2
#define EOT 0

/* prototipos */

int respondo_mesg();        /* retorna hand */

/* variaveis globais */

int arg;
int sel_ret;
int semafor = 1;

main(argc,argv,envp,in_ports,ins,out_ports,outs)

int argc,ins,outs;
char *argv[],*envp[];
CHAN *in_ports[],*out_ports[];
{
int end;

/* pooling entre as sockets de mestre e escravo */

end = 0;

while (!end)
    {
    /* faco select */

    if   if (FD_ISSET(master_sock, &read_socket))
      {
      /* recebi comunicacao do mestre */

      printf("\n(C) Recebi comunicacao MASTER.");

      end = respondo_mesg(master_sock);
      }

    if (FD_ISSET(slave_sock, &read_socket))
       {
       /* recebi comunicacao do escravo */

       printf("\n(C) Recebi comunicacao SLAVE.");

       end = respondo_mesg(slave_sock);
       }

    }
}

int respondo_mesg(hand)
int hand;
{
unsigned char token[1];
unsigned char tok[2];
int end;

token[0] = 0;
end = 0;

/* recebo mensagem */

tok[0] = tok[1] = 0;
chan_in_byte(&tok[0],in_ports[0]);  /* operacao */
chan_in_byte(&tok[1],in_ports[0]);  /* maquina  */

/* processo mesangem */

switch(tok[0])
   {
   case P:  printf("\n(C) Recebi oper P.");
            semafor -= 1;
            if (semafor >= 0)
               {
               /* return OK */
               token[0] = 1;
               }
            else
               {
               /* return NO */
               token[0] = 0;
               }

            /* envio mensagem */

            chan_out_byte(token[0],out_port[tok[1]]);
            break;

   case V:  /* operacao V */

            printf("\n(C) Recebi oper V.");
            semafor = 1;
            break;

   case EOT: printf("\n(C) Recebi oper EOT.");
             end = 1;
   }
return(end);
}
```

A-2.2.3 mux.c

```
/* Esqueleto para implementacao distribuida  */
/* dos algoritmos de alocacao    By De Rose  */
/* Multiplexador de entradas (select TRAM)   */

#include <stdio.h>
#include <chan.h>
#include <sema.h>
#include <thread.h>

#define P  1
#define V  2
#define EOT 0

/* variaveis globais */

char buf[1024];
SEMA buf free;     /* controle de acesso ao buffer */
CHAN **in_p,**out_p; /* ponteiros para as portas   */


main(argc,argv,envp,in_ports,ins,out_ports,outs)
```

```
int argc,ins,outs;
char *argv[],*envp[];
CHAN *in_ports[],*out_ports[];
{
extern void receive();
int i;
sema_init(&buf_free,1)
in_p = out_ports;

/* crio as duas threads para ler as portas de */
/* entrada 0 e 1                              */

for(i=0; i<2; i++)
   thread_create(receive,50*sizeof(int),1,i);
}

void receive(i)
int i;
{
int msglen;

for(;;)
   {
   /* recebo do canal i */

   chan_in_byte(&msglen,in_p[i]);
   sema_wait(&buf_free);  /* protejo a secao critica */

   chan_in_message(msglen,&buf[0],in_pi[i]);
   chan_out_word(msglen,out_p[0]);

   /* envio na porta 0 */

   chan_out_message(msglen,&buf[0],out_p[0]);

   sema_signal(&buf_free);

}
```

A-2.2.4 master.c

```
/* Algoritmo Buddy - Versao Distribuida TRAM */
/* By De Rose   processo MESTRE              */

#define NUM_PROC 16

#include <stdio.h>
#include <string.h>
#include <timer.h>
#include <chan.h>

#define P  1
#define V  2
#define EOT 0


#define TRUE     1
#define FALSE    0

/* prototipos */

int  buddy_aloc   ();     /* int numero , int pid */
int  buddy_dealoc ();     /* int pid              */
void buddy_init   ();     /* void                 */
void buddy_mostra ();     /* void                 */


/* variaveis globais */

int arg;
unsigned char token[3];
unsigned char tok[3];

struct nodo_buddy {
       int processo;
       int alocado;
       };

struct nodo_buddy lista_buddy[NUM_PROC/2];
struct nodo_buddy todo;  /* controle de todo o cubo */
int p_aloc = 0;          /* processadores ja alocados */

main(argc,argv,envp,in_ports,ins,out_ports,outs)

int argc,ins,outs;
char *argv[],*envp[];
CHAN *in_ports[],*out_ports[];
{
unsigned int i;
int      tmp, i_tempo, f_tempo;
int numero;
int ret_code;

token[0] = 0;
buddy_init();

/* pego tempo inicial */

i_tempo = timer_now();
printf("Tempo inicial = %d.\n",i_tempo);

/* Efetuo Alocacao */

ret_code = buddy_aloc(8,9);

if (ret_code > 0)
   printf("\n(M) Aloquei os processadores.");
else
   printf("\n(M) Alocacao falhou !!!");

buddy_mostra();

ret_code = buddy_dealoc(9);

if (ret_code > 0)
   printf("\n(M) Desaloquei os processadores.");
else
   printf("\n(M) Desalocacao falhou. Pid nao encontrado
          !!!");

buddy_mostra();

/* pego tempo final */

f_tempo = timer_now();
printf("Tempo final = %d.\n",f_tempo);

/* calculo diferenca */

printf("Mestre se despedindo !!!\n");
printf("Rodei por %d Tranputer ticks.\n",f_tempo-i_tempo);

/* desativo escravo */

printf("\n(M) Desativo escravo.");
token[0] = 3;
chan_out_byte(token[0],out_port[0]);

/* desativo coordenador */

printf("\n(M) Desativo coordenador.");
token[0] = EOT;
chan_out_byte(token[0],out_port[2]);}

/* rotinas buddy */

void buddy_init()
{
/* inicializa lista buddy */
int i;

p_aloc = 0;
todo.processo = 0;
todo.alocado = 0;

for ( i = 0 ; i < NUM_PROC/2 ; i++ )
    {
    lista_buddy[i].processo = 0;
    lista_buddy[i].alocado  = 0;
    }
}

void buddy_mostra()
{
/* mostra lista buddy */

int i;

printf("\n         Lista Buddy Local\n\n");
for ( i = 0 ; i < NUM_PROC/2 ; i++ )
    printf(" [%2d]   p: %2d    a:%2d \n",i,
             lista_buddy[i].processo,
             lista_buddy[i].alocado);

printf("\nTodo Cubo [%d].",todo.alocado);
}

int  buddy_aloc   ( numero , pid )
```

```
int numero,
    pid;
{
/* aloco um hipercubo com "numero" processadores */
/* para pid                                      */
/* retorno TRUE se a operacao obter sucesso      */

int aloc_ok = 0;
int bit_ok;
int i,t1,t2,m,dim, pot;

/* testo se existem processadores disponiveis */

if (todo.alocado) return FALSE;

if (numero == NUM_PROC)
   {
   todo.alocado = 1;
   todo.processo = pid;
   return(TRUE);
   }

/* disparo procura remota */

token[0] = 1;
token[1] = numero;
token[2] = pid;

printf("\n(M) Envio num = %d pid = %d para alocar !",
        token[1],token[2]);

chan_out_byte(token[0],out_port[0]);
chan_out_byte(token[1],out_port[0]);
chan_out_byte(token[2],out_port[0]);

/* vejo se local tem processadores disponiveis para */
/* atender                                          */

if (numero < ((NUM_PROC/2)-p_aloc) )
{

/* encontro menor dimensao de cubo que contenha */
/* "numero" processadores                       */

dim = 0;
t1 = 1;

while ( t1 < numero )
   {
   dim++;
   t1 *= 2;
   }

/* procuro menor "m" que satisfaca a equacao buddy */

m = 0;
aloc_ok = 0;

while (  ((((m+1)*t1)-1) < (NUM_PROC/2)) && !(aloc_ok))
   {
   bit_ok = 1;
   t2 = m*t1;
   while ( (t2 <= (((m+1)*t1)-1)) && (bit_ok) )
      {
      if ( lista_buddy[t2].alocado != 0 )
         bit_ok = 0;
      t2++;
      }

   if ( bit_ok )
      aloc_ok = 1;
   else
      m++;
}

/* caso encontrado "m", tento alocar processadores */

if ( aloc_ok )
   {
   /* faco consulta a coordenador - secao critica */

   token[0] = P;
   token[1] = 0; /* sou mestre */
   chan_out_byte(token[0],out_port[2]);
   chan_out_byte(token[1],out_port[2]);

   printf("\n(M) Testando coordenador sobre SC.");

   /* espero resposta */

   chan_in_byte(&tok[0],in_ports[2]);

   printf("\n(M) Resposta coord Byte = %d.",tok[0]);

   if (tok[0])
      for ( i = m*t1 ; i <= (((m+1)*t1)-1) ; i++ )
          {
          lista_buddy[i].alocado  = 1;
          lista_buddy[i].processo = pid;
          p_aloc += 1;
          }
    else
       aloc_ok = 0;
   }

   }
else
   aloc_ok = 0;   /* nao posso alocar local */

if (tok[0])
   printf("\nLocal alocou legal (pid=%d) !!!",pid);
else
   printf("\nLocal NAO alocou legal (pid=%d) !!!",pid);


/* recebo ok ou negado do remoto */

chan_in_byte(&tok[0],in_ports[0]);

if (tok[0])
   {
   aloc_ok = 1;
    printf("\nRemoto encontrou !!!");
   }
else
   printf("\nRemoto NAO encontrou !!!");

if (!aloc_ok)   /* local negou, pego ret code remoto */
   aloc_ok = tok[0];

/* libero coordenador - secao critica */

printf("\nLibero Secao critica !!!");
token[0] = V;
token[1] = 0; /* null */
chan_out_byte(token[0],out_port[2]);
chan_out_byte(token[1],out_port[2]);

return(aloc_ok);
}

int  buddy_dealoc ( pid )

int pid;
{
/* libero processadores alocados pelo processo pid */
/* retorno numero de processadores desalocados     */
/* efetuo desalocacao remota */

int i;
int pid_ok = 0;

if (pid == todo.processo)
   {
   todo.processo = 0;
   todo.alocado = 0;
   return(TRUE);
   }

/* envio pid para nodo remoto */

token[0] = 2;
token[1] = pid;
token[1] = 0;

printf("\n(M) Envio pid para remoto desalocar = %d
       !",token[1]);

chan_out_byte(token[0],out_port[0]);
chan_out_byte(token[1],out_port[0]);
chan_out_byte(token[2],out_port[0]);

/* nao preciso esperar resposta */
/* efetuo processamento local */

for ( i = 0 ; i < (NUM_PROC/2) ; i++ )
    if ( lista_buddy[i].processo == pid )
       {
       lista_buddy[i].alocado  = 0;
       lista_buddy[i].processo = 0;
       pid_ok += 1;
       p_aloc --;
      }

if (pid_ok > 0)
```

```
   return(TRUE);
else
   return(FALSE);
}
```

A-2.2.5 slave.c

```
/* Algoritmo Buddy - Versao Distribuida TRAM */
/* By De Rose  processo (ESCRAVO)            */

#define NUM_PROC 16

#include <stdio.h>
#include <string.h>
#include <timer.h>
#include <chan.h>

#define P  1

#define TRUE     1
#define FALSE    0

/* prototipos */

int  buddy_aloc   ();  /* int numero , int pid */
int  buddy_dealoc ();  /* int pid              */
void buddy_mostra ();  /* void                 */
void buddy_init   ();  /* void                 */

/* variaveis globais */

int arg;
unsigned char token[3];
unsigned char tok[3];
int numero;

struct nodo_buddy {
       int processo;
       int alocado;
       };


struct nodo_buddy lista_buddy[NUM_PROC/2];

int p_aloc = 0;   /* processadores ja alocados */

main(argc,argv,envp,in_ports,ins,out_ports,outs)

int argc,ins,outs;
char *argv[],*envp[];
CHAN *in_ports[],*out_ports[];
{
int ret_code;
int pid,end;

token[0] = 0;
buddy_init();

/* atendo pedidos do mestre */

end = 0;

while (!end)
   {

printf("\n(S) Espero mensagem.");

tok[0] = 0;
chan_in_byte(&tok[0],in_ports[0]);
chan_in_byte(&tok[1],in_ports[0]);
chan_in_byte(&tok[2],in_ports[0]);

/* processo mesangem */

switch( tok[0] )
   {
   case 1:      /* efetuo alocacao */

        printf("\n(S) Chegou pedido Aloc !!!");

        /* acerto parametros */

        numero = tok[1];
        pid = tok[2];

        ret_code = buddy_aloc(numero,pid);

        if (ret_code > 0)
           {
           printf("\n(S) Aloquei os processadores.");
           token[0] = 1;
           chan_out_byte(token[0],out_port[0]);
           buddy_mostra();
           }
        else
           {
           printf("\n(S) Alocacao falhou !!!");
           token[0] = 0;
           chan_out_byte(token[0],out_port[0]);
           }

        break;

   case 2:      /* efetuo desalocacao */

        printf("\n(S) Chegou pedido Desaloc !!!");

        pid = tok[1];

        ret_code = buddy_dealoc(pid);

        if (ret_code > 0)
           {
           printf("\n(S) Desaloquei os processadores.");
           token[0] = 1;
           chan_out_byte(token[0],out_port[0]);
           }
        else
           {
           printf("\n(S) Desalocacao falhou !!!");
           token[0] = 0;
           chan_out_byte(token[0],out_port[0]);
           }

        break;

   case 3:      /* corto conexao, saindo do loop */

        end = 1;
        printf("\nSai do Loop (mesg 3) !!!");
        buddy_mostra ();
    }

   }
}

/* rotinas buddy */

void buddy_init()
{
/* inicializa lista buddy */

int i;

p_aloc = 0;

for ( i = 0 ; i < (NUM_PROC/2) ; i++ )
    {
    lista_buddy[i].processo = 0;
    lista_buddy[i].alocado  = 0;
    }
}

void buddy_mostra()
{
/* mostra lista buddy */

int i;

printf("\n         Lista Buddy Local\n\n");
for ( i = NUM_PROC/2 ; i < NUM_PROC ; i++ )
    printf(" [%2d]   p: %2d    a:%2d \n",i,
           lista_buddy[i].processo,
           lista_buddy[i].alocado);
}

int  buddy_aloc   ( numero , pid )

int numero,
    pid;
{
/* aloco um hipercubo com "numero" processadores */
/* para pid                                      */
/* retorno TRUE se a operacao obter sucesso      */

int aloc_ok;
int bit_ok;
int i,t1,t2,m,dim, pot;
```

```
/* testo se existem processadores livres */

if (numero < ((NUM_PROC/2)-p_aloc) )
{

/* encontro menor dimensao de cubo que contenha */
/* "numero" processadores                       */

dim = 0;
t1 = 1;

while ( t1 < numero )
   {
   dim++;
   t1 *= 2;
   }

/* procuro menor "m" que satisfaca a equacao buddy */

m = 0;
aloc_ok = 0;

while (  ((((m+1)*t1)-1) < (NUM_PROC/2)) && !(aloc_ok))
   {
   bit_ok = 1;
   t2 = m*t1;
   while ( (t2 <= (((m+1)*t1)-1)) && (bit_ok) )
      {
      if ( lista_buddy[t2].alocado != 0 )
         bit_ok = 0;
      t2++;
      }

   if ( bit_ok )
      aloc_ok = 1;
   else
      m++;
}

/* caso encontrado "m", tento alocar processadores */

if ( aloc_ok )
   {
   /* faco consulta a coordenador - secao critica */

   token[0] = P;
   token[1] = 1;  /* sou escravo */
   chan_out_byte(token[0],out_port[1]);
   chan_out_byte(token[1],out_port[1]);

   printf("\n(S) Testando coordenador sobre SC.");

   /* espero resposta */

   chan_in_byte(&tok[0],in_ports[1]);

   printf("\n(S) Resposta coord Byte = %d.",tok[0]);

   if (tok[0])
      {
      for ( i = m*t1 ; i <= (((m+1)*t1)-1) ; i++ )
          {
          lista_buddy[i].alocado  = 1;
          lista_buddy[i].processo = pid;
          p_aloc += 1;
          }
      printf("\n(S) Local alocou legal (pid=%d)
              !!!",pid);
      return(aloc_ok);
      }
   }

} /* end if */
printf("\n(S) Local NAO alocou legal (pid=%d) !!!",pid);
return(FALSE);
}

int  buddy_dealoc ( pid )

int pid;
{
/* libero processadores alocados pelo processo pid */
/* retorno numero de processadores desalocados     */

int i;
int pid_ok = 0;

/* efetuo processamento local */

for ( i = 0 ; i < (NUM_PROC/2) ; i++ )
    if ( lista_buddy[i].processo == pid )
       {
       lista_buddy[i].alocado  = 0;
       lista_buddy[i].processo = 0;
       pid_ok += 1;
       p_aloc --;
       }

if (pid_ok > 0)
   return(TRUE);
else
   return(FALSE);
}
```

ANEXO A-3 LISTAGEM DO PROTÓTIPO SUB-CUBE RPC

Neste anexo é apresentada uma listagem parcial do servidor Sub-Cube RPC juntamente com o código de um respectivo cliente. O código foi simplificado e as rotinas de atendimento foram retiradas (podem ser as do anexo A ou B) com o propósito de demonstrar a estrutura de um servidor RPC [SUN 90].

A-3.1 p_server.c

```
/*********************************************/
/** O Servidor de Processadores Sub-Cube RPC **/
/**                                         **/
/**      By Cesar De Rose - 29/05/92        **/
/**                                         **/
/*********************************************/

/* Servidor de processadores    */
/* By Cesar De Rose 10/10/92    */

#include <stdio.h>
#include <math.h>
#include <rpc/rpc.h>
#include "svc_def.h"

#define GRAU_CUBO  6
#define NUM_PROC   64

/* operations types */

#define INIT   0
#define ALOC   1
#define DEALOC 2
#define SHOW   3
#define FREE   4  /* somente usado pelo algoritmo FL */
                  /* (alocacao dinamica)             */

/* prototipes */

int  buddy_aloc   ();     /* int numero , int pid */
int  buddy_dealoc ();     /* int pid              */
void buddy_init   ();     /* void                 */
void buddy_mostra ();     /* void                 */

int  GC_aloc   ();        /* int numero , int pid */
int  GC_dealoc ();        /* int pid              */
void GC_init   ();        /* void                 */
void GC_mostra ();        /* void                 */

void GC_gera_brgc();      /* int * array */

/* variaveis globais */

struct nodo_buddy {
       int processo;
       int alocado;
       };

struct nodo_GC {
       int node;
       int processo;
       int alocado;
       };

struct nodo_buddy lista_buddy[NUM_PROC];
struct nodo_GC    lista_GC   [NUM_PROC];

int p_aloc = 0; /* processadores ja alocados       */

/* codigo do servidor */

main()
{
SVCXPRT *transp;
void dispatch();

/* handle de transporte */

transp = svcudp_create(RPC_ANYSOCK);

(void)pmap_unset(SERVER_PROG,SERVER_VERS_1);

/* registro do servidor no mapeador de portas */

svc_register(transp,SERVER_PROG,SERVER_VERS_1,dispatch,
             IPPROTO_UDP);

printf("O SERVIDOR ESPERA POR MENSAGENS\n");
svc_run();
}

void dispatch(rqstp,transp)
struct svc_req *rqstp;
SVCXPRT        *transp;
{
int  idn[3]; /* 0-2 */
char gradep;
int status;

/* chegou algo */

printf("Chegou um pedido ... \n");

switch(rqstp->rq_proc)
  {
  case BUDDY:
     printf("Algoritmo BUDDY \n");
     svc_getargs(transp,xdr_par_vector,idn);
     switch(idn[0])
       {
       case INIT:
          printf("Efetuo inicializacao da estrutura.\n");
          buddy_init();
          status = TRUE;  /* sempre OK */
          break;

       case ALOC:
          printf("Efetuo a alocacao de %d procs com
                  pid %d.\n",idn[1],idn[2]);
          status = buddy_aloc(idn[1],idn[2]);
          break;

        case DEALOC:
          printf("Efetuo a desalocacao do pid %d.\n",
                  idn[1]);
          status = buddy_dealoc (idn[1]);
          break;

        case SHOW:
          printf("Mostro lista buddy\n");
          buddy_mostra ();
          status = TRUE; /* sempre ok */
          break;

        default:
          printf("Funcao %d nao existente \n",idn[0]);
          status = FALSE;
       }
     svc_sendreply(transp,xdr_int,&status);
     printf("Mandei reply para o cliente ... \n");
     break;

  case GRAYCODE:
     printf("Algoritmo GC \n");
     svc_getargs(transp,xdr_par_vector,idn);
     switch(idn[0])
       {
       case INIT:
         printf("Efetuo inicializacao da estrutura.\n");
         GC_init();
         status = TRUE;  /* sempre OK */
         break;

       case ALOC:
         printf("Efetuo a alocacao de %d procs com pid
                 %d.\n",idn[1],idn[2]);
         status = GC_aloc(idn[1],idn[2]);
         break;

       case DEALOC:
         printf("Efetuo a desalocacao do pid %d.\n",
                 idn[1]);
         status = GC_dealoc (idn[1]);
         break;

       case SHOW:
         printf("Mostro lista gc\n");
         GC_mostra ();
         status = TRUE; /* sempre ok */
         break;

       default:
         printf("Funcao %d nao existente \n",idn[0]);
         status = FALSE;
       }
     svc_sendreply(transp,xdr_int,&status);
     printf("Mandei reply para o cliente ... \n");
     break;

  default:
     printf("Classe do servico nao reconhecida !!!\n");
            svcerr_noproc(transp);
     break;
  }

}

/*------------------- rotinas buddy -------------------*/
/*--------------------- rotinas GC --------------------*/
```

A-3.2 p_client.c

```
/*************************************************/
/***  Cliente para o servidor de processadores ***/
/***                                           ***/
/***      By Cesar De Rose - 12/10/92          ***/
/*************************************************/

#include <stdio.h>
#include <rpc/rpc.h>

#include <sys/time.h>
#include <sys/resource.h>

#include "svc_def.h"

struct timeval timeout = {25,0};

main(argc,argv)
int argc;
char *argv[];
{
CLIENT *clnt_handlep;
enum clnt_stat status;
int opr_status;
int ret_code, i;
int param[3];            /* 0-2 */
struct rusage dados;

if ( (argc<3)||(argc>5) )
   {
   printf("\nUso: %s maquina operacao [parametro1]
          [parametro2]\n\n",argv[0]);

   printf("Operacoes deste cliente:   codigo:   parame
          tros: \n");
   printf("-----------------------------------------
          --------\n");
   printf("mgc_init                    0
          nenhum \n");
   printf("mgc_aloc                    1
          [proc] [pid] \n");
   printf("mgc_dealoc                  2
          [pid]\n");
   printf("mgc_mostra                  3
          nenhum\n\n");

   return(1);
   }

/* faco controle da linha de comando */

param[0] = atoi(argv[2]);
param[1] = 0;
param[2] = 0;

switch(param[0])
   {
   case 0:
      if ( argc>3 )
         {
         printf("Operacao de Inicializacao MGC (0)
                nao necessita parametros !\n");
         printf("Uso: %s maquina operacao\n",argv[0]);
         return(1);
         }
      else
         {
         printf("Executando operacao de Inicializacao
                MGC (0) ...\n");
         }
      break;

   case 1:
      if ( argc<5 )              /* alocacao */
         {
         printf("Operacao de Alocacao MGC (1)
                necessita dois parametros !\n");
         printf("Uso: %s maquina operacao
                n_processadores pid\n",argv[0]);
         return(1);
         }
      else
         {
         param[1] = atoi(argv[3]);
         param[2] = atoi(argv[4]);
         printf("Executando operacao de Alocacao
                MGC de %d processadores com pid
                %d ...\n",param[1],param[2]);
         }
      break;

   case 2:
      if ( argc!=4 )             /* desalocacao */
         {
         printf("Operacao de Desalocacao MGC (2)
                se utiliza de um parametro !\n");
         printf("Uso: %s maquina operacao pid\n",argv[0]);
         return(1);
         }
      else
         {
         param[1] = atoi(argv[3]);
         printf("Executando operacao de
                desalocacao MGC do pid %d ...\n",param[1]);
         }
      break;

   case 3:
      if ( argc>3 )
         {
         printf("Operacao Mostra Estrutura MGC
                (3) nao necessita parametros !\n");
         printf("Uso: %s maquina operacao\n",argv[0]);
         return(1);
         }
      else
         {
         printf("Executando operacao Mostra
                Estrutura MGC (3) ...\n");
         }
      break;

   default:
      printf("Operacao %d inexistente !!!\n",param[0]);
      printf("Operacoes MGC: (0) Inicializacao,
             (1) Alocacao, (2) Desalocacao, (3)
             Mostra Estrutura. \n");
      return(1);
   }

/* pego um handle pro cliente */

clnt_handlep = clnt_create(argv[1],SERVER_PROG,
               SERVER_VERS_1,"udp");

if (!clnt_handlep)
   clnt_perror(clnt_handlep,argv[1]);

/* efetuo RPC call para o server */

printf("Faco Chamada ao servidor \n");

status = clnt_call(clnt_handlep,MGRAYCODE,xdr_par_vector
         ,param,xdr_int,&opr_status,timeout);

if (status == RPC_SUCCESS)
   {
   if(!opr_status)
     {
     printf("Erro no retorno da funcao %d !!!\n",
            param[0]);
     ret_code=0;
     }
   else
     printf("Sucesso na execucao da funcao !\n");

   }
else clnt_perror(clnt_handlep,argv[1]);

clnt_destroy(clnt_handlep);

getrusage(RUSAGE_SELF, &dados);

printf("Tempo decorrido (seg/u_seg): %d:%03d \n",
       dados.ru_utime.tv_sec+dados.ru_stime.tv_sec,
       dados.ru_utime.tv_usec+dados.ru_stime.tv_usec);

return(ret_code);
}
```

BIBLIOGRAFIA

[ABR 89] ABRAHAM, Seth; PADMANABHAN, K. Performance of the direct binary n-cube network for multiprocessors. **IEEE Transactions on Computers**, New York, v. 38, n. 7, p. 1000-1011, July 1989.

[ALD 89] AL-DHELAAN, A.; BOSE, B. A New Strategy for Processor Allocation in an N-cube Multiprocessor. In: PHOENIX CONFERENCE COMPUTER AND COMMUNICATIONS, , Mar., 1989. **Proceedings...** New York:IEEE, 1989. p. 114-118.

[BHU 84] BHUYAN, L.; AGRAWAL, D. Generalized hypercube and hyperbus structures for a computer network. **IEEE Transactions on Computers**, New York, v. 33, n. 4, p. 323-333, 1984.

[CHA 88] CHAN, M.; SHIN, F. On Embedding Rectangular Grids in Hypercubes. **IEEE Transactions on Computers**, New York, v. 37, n. 10, p. 1285-1288, Oct. 1988.

[CHE 86] CHEN, M. S.; SHIN, S.G. Embedment of interacting task modules into a hypercube multiprocessor. In: HYPERCUBE CONFERENCE, 2., Oct. 1986. **Proceedings...** 1986. p. 121-129.

[CHE 87] CHEN, M.; SHIN, K. Processor Allocation in an n-cube multiprocessor using Gray codes. **IEEE Transactions on Computers**, New York, v. 36, n. 12, p. 1396-1407, Dec. 1987.

[CHE 89] CHEN, W.; STALLMANN, M.; GHERINGER, E. Hypercube Embedding Heuristics: An Evaluation. **International Journal of Parallel Programming** , New York, v. 18, n. 6, p. 505-549, Dec. 1989.

[CHE 90] CHEN, M.; SHIN, K. Subcube allocation and task migration in hypercube multiprocessors. **IEEE Transactions on Computers**, New York, v. 39, n. 9, p. 1146-1155, Sept. 1990.

[CHU 90] CHUANG, P.; TZENG, N. Dynamic Processor Allocation in Hypercube Computers. In: ANNUAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 17., May 28-31, 1990, Seattle. **Proceedings...** Washington:IEEE, 1990. 377 p. p. 40-49.

[CHU 92] CHUANG, P.; TZENG, N. A Fast Recognition–Complete Processor Allocation Strategy for Hypercube Computers. **IEEE Transactions on Computers**, New York, v. 41, n. 4, p. 467-479, Apr. 1992.

[DER 92] DE ROSE, César. **As várias faces de um cubo. Um estudo crítico sobre arquiteturas hipercúbicas.** Porto Alegre: CPGCC da UFRGS, 1992. 189 p. (Trabalho Individual, 259)

[DER 93] DE ROSE, César; NAVAUX, P. Algoritmos Paralelos para alocação de processadores em máquinas multiprocessadoras hipercúbicas. In: 5. SIMPOSIO BRASILEIRO DE ARQUITETURAS DE COMPUTADORES – PROCESSAMENTO DE ALTO DESEMPENHO (SBAC–PAD), 7-10 set. 1993, Florianópolis. **Anais ...** Florianópolis:SBC, 1993. 775 p. p. 459-474.

[DES 86] DESPHANDE, S.; JENEVEIN, R. Scalability of a Binary Tree on a Hypercube. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Aug. 19-22, 1986, University Park, PE. **Proceedings...** New York:IEEE, 1986. 1051 p. p. 661-668.

[DOW 88] DOWD, P.; JABBOUR, K. Spanning Multiaccess Channel Hypercube Computer Interconnection. **IEEE Transactions on Computers**, New York, v. 37, n. 9, p. 1137-1142, Sept. 1988.

[DUT 91] DUTT, S.; HAYES, P. Subcube allocation in Hypercube Computers. **IEEE Transactions on Computers**, New York, v. 40, n. 3, p. 341-352, Mar. 1991.

[ERC 90] ERCAL, F.; RAMANUJAM, J.; SADAYAPPAN, P. Task allocation onto a hypercube by recursive mincut bipartitioning. **Journal of Parallel and Distributed Computing**, v. 10, p. 35-44, 1990.

[FEN 81] FENG, T. A Survey of Interconection Networks. **IEEE Computer** , New York, v. 14, n. 12, p. 12-27, Dec. 1981.

[HAY 86] HAYES, J.; MUDGE, T.; STOUT. Architecture of a Hypercube Supercomputer. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Aug. 19-22, 1986, University Park, PE. **Proceedings...** New York:IEEE, 1986. 1051 p. p. 653-660.

[HAR 88] HARARY, F.; HAYES, J.P.; WU, H.J. A survey of the theory of hypercube graphs. **Comput. Math. Appl** , Great Britain, v. 15, n. 4, p. 277-289, 1988.

[HUA 89] HUANG, C.; JUANG, J. A Partial Compaction Scheme for Processor Allocation in Hypercube Multiprocessors. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 1990. **Proceedings...** New York:IEEE, 1989. p. 211-217.

[HWA 85] HWANG, K.; BRIGGS, A. **Computer Architecture and Parallel Processing.** New York:McGraw-Hill, 1985. 846 p.

[INM 89] INMOS Limited. **3L Parallel C User Guide.** Bristol: INMOS Limited, 1989. 271 p.

[INM 90] INMOS Limited. **IMS B008 User Guide and Reference Manual.** Bristol: INMOS Limited, 1990. 104 p.

[KIM 89] KIM, J.; DAS, R.; LIN W. A processor Allocation Scheme for Hypercube Computers. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, ,Aug. 1989. **Proceedings...** New York:IEEE, 1989. p. 231-238.

[KIM 91] KIM, J.; DAS, R.; LIN W. A Top-Down Processor Allocation Scheme for Hypercube Computers. **IEEE Transactions on Parallel and Distributed Systems,** New York, v. 2, n. 1, p. 20-30, Jan. 1991.

[PET 85] PETERSON, J.; TUAZON, J.; PNIEL, M.; LIBERMAN, D. The Mark III Hypercube-Emsemble Concurrent Computer. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING ,Aug. 20-23, 1985, St. Charles. **Proceedings...** New York:IEEE, 1985. 868 p. p. 71-73.

[PUR 70] PURDOM, P.W. Jr.; STIGLER, M. Statical properties of the buddy system. **Journal of the Association for Computing Machinery,** New York, v. 17, n. 4, p. 683-697, Oct. 1970.

[RAM 88] RAMANUJAM, J.; ERCAL, F.; SADAYAPPAN, P. Task allocation by simulated annealing. In: INT. CONF. ON SUPERCOMPUTING (ICS), Jan. 4-8, 1988, St. Malo, France. **Proceedings...** New York:ACM, 1988. 679 p.

[RAM 90] RAMANUJAM, J.; ERCAL, F.; SADAYAPPAN, P. Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning. **Journal of Parallel and Distributed Computing**, v. 10, p. 35-44, 1990.

[REI 77] REINGOLD, M.; NIEVERGELD, J.; DEO, N. **Combinatorial Algorithms.** Englewood Cliffs, NJ: Prentice-Hall, 1977.

[RIZ 92] RIZZO, L. PCserver: Networking PC-hosted Transputers. In: TRANSPUTERS '92, 1992. **Proceedings...** Amsterdam:IOS Press, 1992. p. 158-171.

[SAA 88] SAAD, Y.; SCHULTZ, M. Topological Properties of Hypercubes. **IEEE Transactions on Computers**, New York, v. 37, n. 7, p. 867-872, July 1988.

[SEI 85] SEITZ, C. The Cosmic Cube. **Communications of the ACM** , New York, v. 28, n. 1, p. 22-33, Jan. 1985.

[SIL 91] SILBERSCHATZ, A.; PETERSON, J.; GALVIN, P. **Operating System Concepts.** 3th ed. Reading:Addison-Wesley, 1991.

[SUN 90] SUN Microsystems. **Network Programming Guide**. Mountain View:SUN, 1990. 356 p.

[SUN 90] SUN Microsystems. **Remote Procedure Call Programming Guide** Mountain View:SUN, 1990.

[TAN 88] TANENBAUM, A. **Computer Networks.** 2nd ed. Englewood Cliffs:Prentice-Hall, 1988. 658 p.

[TEO 94] TEODOROWITSCH, R.; DE ROSE, C. **Um simulador para o compartilhamento de uma máquina Hipercúbica em uma rede de estações.** Porto Alegre:CPGCC da UFRGS, 1994. (Relatório Interno, a ser publicado)

[TRI 90] TRINDADE Jr., O.; SANTANA, M. J. Um Servidor de Processamento Paralelo Baseado em Transputers – Requisitos e Definição. In: 3. SIMPOSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES – PROCESSAMENTO DE ALTO DESEMPENHO (SBAC — PAD), 7-9 nov. 1990, Rio de Janeiro. **Proceedings.** Rio de Janeiro. SBC/PUC RJ, 1990. 374 p. p. 225-237.

[WAN 91] WANG, H.; YANG, Q. Prime Cube Graph Approach for Processor Allocation in Hypercube Multiprocessors. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 1991. **Proceedings...** New York:IEEE, 1991. p. 25-32.

[WEI 90] WEIL, F.; JAMIESON, L.; DELP, E. An Analysis of Fixed-Assignement Hypercube Partitioning. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Aug. 1990. **Proceedings...** 1990.

Informática
UFRGS

Algoritmos paralelos para a gerência e alocação de processadores em máquinas multiprocessadoras hipercúbicas.

Dissertação apresentada aos Senhores:

Prof. Dr. Dalcidio Moraes Claudio

Prof. Dr. Philippe Olivier Alexandre Navaux

Prof. Dr. Virgílio F. Almeida (UFMG)

Prof. Dr. Wolfgang Pandikow

Vista e permitida a impressão.
Porto Alegre, 21/01/94.

Prof. Dr. Philippe Olivier Alexandre Navaux,
Orientador.

Prof. Dr. Ricardo A. da L. Reis,
Coordenador do Curso de Pós-Graduação
em Ciência da Computação.