UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA

DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

A Estrutura de Hipercubo e os Autômatos Autônomos: Carregamento e Redistribuição Dinâmica de Trabalho

AUTOR: Noritsuna Furuya

ORIENTADOR: Prof. Dr. Paulo Cesar Bezerra

Tese apresentada à Faculdade de Engenharia Elétrica FEE–UNICAMP como parte dos requisitos exigidos para obtenção do título DOUTOR EM ENGENHARIA ELÉTRICA

Julho de 1990

Para Tsunao e Tsukie, meus pais.

Para Yolanda.

Agradecimentos

Ao Prof. Dr. Paulo Cesar Bezerra, pela orientação, estímulo e confiança.

Aos professores e funcionários do Departamento de Engenharia de Computação e Automação Industrial da Faculdade de Engenharia Elétrica da UNICAMP, pelo apoio e colaboração.

À CAPES, pela concessão da bolsa de Auxílio Deslocamento para custear as viagens a Campinas.

Ao professor Dr. Cláudio Luis de Amorim e ao pessoal do Programa de Engenharia de Sistemas de Computação da UFRJ, por permitir a utilização dos equipamentos para testes e pelas sugestões e esclarecimentos que foram de grande auxílio para a tese.

Ao professor Dr. Jan Frans Willem Slaets e ao pessoal do Laboratório de Instrumentação Eletrônica do Institudo de Física e Química de São Carlos da USP, por permitir a utilização dos equipamentos para testes.

Ao pessoal do Departamento de Computação da Universidade Federal de São Carlos, pelo apoio e concessão do afastamento parcial para desenvolver o Programa de Doutorado.

Ao colega José Tarcisio Costa Filho e ao pessoal do DMCSI da FEE - UNICAMP, pelo apoio e colaboração.

Ao Prof. Dr. Reginaldo Pallazo Jr. e à Maria Conceição Peres Young do Departamento de Comunicações pelo apoio e colaboração.

E a todos aqueles que colaboraram direta ou indiretamente, para a realização deste trabalho.

Resumo

O trabalho entitulado "A Estrutura de Hipercubo e os Autômatos Autônomos: Carregamento e Redistribuição Dinâmica de Trabalho" tem como objetivo a definição de uma metodologia para particionamento, distribuição inicial e redistribuição dinâmica de serviços entre processadores concorrentes estruturados num arranjo de hipercubo.

São considerados também os problemas cujos modelos são autômatos finitos autônomos, cujos grafos associados possam ser particionados em subgrafos isomorfos. Além disso, são apresentados estudos sobre a implementação dos algoritmos num ambiente com 4 Transputers. Ao final são feitas considerações sobre as restrições das linguagens C da 3L e OCCAM.

ÍNDICE

INTRODUÇÃO

Com o barateamento de custo e compactação de processadores, o multiprocessamento tem se tornado uma opção viável para muitas aplicações. Com isso, vários estudos sobre multiprocessamento têm sido feitos, procurando resolver problemas até então sem solução devido ao enorme volume de processamento.

Este trabalho segue esta tendência e tem como principal objetivo encontrar a melhor forma de implementar algoritmos para uma específica classe de problemas. Tem-se como proposta a identificação da classe de autômatos que possam ser implementados com alta eficiência numa estrutura de hipercubo, e a determinação de algoritmos para carregamento dinâmico de trabalho em processamento concorrente.

Está mais ou menos evidente que ainda não se tem a melhor arquitetura para todas as aplicações. Aliás, para muitas aplicações, o multiprocessamento ainda é bastante ineficiente. A melhor arquitetura para uma determinada aplicação pode ter desempenho medíocre em outra aplicação. Esta é a razão pela qual procurou-se solução para apenas uma classe de problemas. Para se distribuir trabalhos através da rede de processadores, é necessário que esta rede esteja organizada de tal forma que seja fácil identificar e localizar não só os processadores, mas também o trajeto das mensagens trocadas entre eles. Pouco antes de se iniciar este trabalho, foi proposta na Caltech uma arquitetura chamada hipercubo. Esta arquitetura não só satisfaz todos estes requisitos como também utiliza para a identificação das UCPs e definição de ligações o código binário. Isso é feito obedecendo o reticulado representativo da Álgebra de Boole, facilitando e agilizando a distribuição de tarefas e troca de mensagens. Esta arquitetura mostrou-se particularmente conveniente para resolver o problema proposto, de tal forma que foi escolhida para implementação. Os resultados obtidos foram satisfatórios, obtendo-se um "speed-up" 3 com o uso de 4 processadores para a classe de problemas propostos. O desempenho mostrou-se fortemente dependente da distribuição inicial de serviços. Ao se trabalhar sobre a segunda parte da proposta, que é a distribuição dinâmica de serviços, constatou-se que, embora as linguagens permitam fazer este tipo de aplicações, os compiladores ainda são deficientes para este nível de sofisticação. Entretanto, pode-se prever nítida melhora no desempenho quando os compiladores melhorarem em termos de potencial de uso das variáveis comuns, o que certamente deverá ocorrer num futuro próximo.

A seguir, apresenta-se a organização dos capítulos subsequentes.

No Capítulo 1 faz-se um estudo sobre multiprocessamento, arquitetura dos multiprocessadores, classificando-se e citando-se as vantagens e desvantagens.

No Capítulo 2 propõe-se um algoritmo para ser utilizado para uma classe específica de problemas.

No Capítulo 3 são feitos os estudos das máquinas mais adequadas ao problema.

No Capítulo 4 são apresentados o algoritmo implementado e seus resultados.

No Capítulo 5 estão as conclusões.

O apêndice constitui-se de quatro partes, sendo a primeira sobre os estudos feitos numa máquina VTM (Variable Topology Multicomputer) que é uma das arquiteturas em que se pode implementar o problema proposto. Na segunda parte tem-se o estudo feito numa máquina de barramento comum, onde foi possível perceber quanto é importante a infraestrutura do software básico. A terceita parte contém programas testes dos compiladores C Paralelo do 3L e o TDS (Transputer Development System) usando linguagem OCCAM. Finalmente, na última parte, tem-se a relação dos programas desenvolvidos.

Capítulo 1
MULTIPROCESSAMENTO

Desde que foi introduzido o computador digital na década de 50, o potencial do computador evoluiu contínuamente. Para cada nova geração que surgia, os computadores passavam a satisfazer novos requisitos e as velocidades que eram contadas em milisegundos passaram a microsegundos e, hoje, contam-se em nanosegundos. Esse mesmo ritmo se manteve em termos de barateamento do custo e aumento da capacidade de memória. Os processadores passaram a executar instruções cada vez mais poderosas e, atualmente, executam instruções de máquina que antes eram consideradas instruções de linguagem de alto nível.

Há muito tempo sentia-se a multiplicação do hardware como uma opção de aumentar a capacidade de processamento, sendo vista porém como uma opção cara e discutível, uma vez que tanto a UCP como a memória eram as partes mais caras do computador. Com o barateamento do hardware, mudou-se bastante a forma de se ver a multiplicação da UCP, que é hoje considerada como uma alternativa para resolver o problema de tempo crítico ou para os casos que requerem enorme volume de processamento. Muitas estruturas tempo real já procuram resolver problemas através de multiplicação das UCPs, de forma tal que muitos computadores digitais poderiam ser conectados para operar em paralelo e em cooperação mútua, procurando um objetivo final comum. É muito comum encontrar no processamento em tempo real os casos em que o tempo imposto pela aplicação em uma única máquina digital não é suficiente, mesmo nas mais poderosas.

Os primeiros multiprocessadores foram feitos em nível experimental: o PILOT em 1958, seguindo-se outros como o POLYMORPHIC em 1966, o SOLOMON [Paker 83], um "array" de processadores, e o já conhecido ILLIAC-IV [Soki 78] com 64 processadores montados em malhas bi-direcionais rígidas. Esses sistemas, porém, mostraram-se eficientes para certas tarefas específicas, mas eram inadequados para uso geral. Em 1971 foi montado o PEPE [Paker 83] que introduziu o paralelismo mais flexível, com cada atividade tratada por um processador separado, formando uma atividade simultânea em cooperação. Caso uma das atividades ficasse muito lenta, esta atividade seria decomposta em outras menores, até que se obtivesse a velocidade exigida. Um "host" seria responsável pelo controle global. Outro projeto de vulto foi o C.mmp [Soki 78][DeCe 89], na universidade de Carnegie-Mellon, que trabalha com 16 processadores e 16 módulos de memória, chaveados por "cross-bar". Posteriormente, a mesma universidade construiu o Cm, baseado no DEC LSI-11, usando o barramento paralelo hierarquizado. Houve também o PLURIBUS [Paker 83], baseado em barramentos múltiplos.

Com o advento dos microprocessadores mais poderosos, a construção de estruturas com multicomputadores ganhou um novo alento. Podem-se citar vários projetos de vulto, como: Micral M (na França), R2E (Universidade Kent, na Inglaterra), CUBY-M (Universidade de Manchester, na Inglaterra), POLYPROC (Universidade de Sussex, na Alemanha)

e SMS (Siemens na Alemanha) [Paker 83, DeCe 89].

Esses trabalhos identificaram diferentes tendências, variando com a aplicação. A primeira é a procura de um sistema de uso geral, com interconexão de grande número de processadores ligados de forma homogênea e facilmente identificáveis. A segunda é a de procurar sistemas de usos específicos, onde as ligações são feitas buscando otimizar o desempenho para uma aplicação particular. Uma terceira tendência é a de procurar uma arquitetura dinâmica onde os meios de comunicação se reorganizam de acordo com as necessidades do momento. A ela deu-se o nome de VTM (Variable Topology Multicomputer) [Paker 83, Alma89]. Portanto, o VTM pode ser considerado uma máquina multiprocessadora, com estrutura de acoplamento flexível e dinâmica entre os processadores e memórias. Um exemplo deste tipo de máquina é apresentado no Apêndice A.

1.1 – Classificação de Multiprocessadores.

Existem numerosas formas de classificar os processadores. Através de número de instruções e dados que executam (MIMD, SIMD,etc.), segundo técnicas de comunicações, e/ou pela configuração das conexões.

Tudo isso faz com que os multiprocessadores sejam divididos em classes muito variadas que sempre apresentam suas vantagens e desvantagens frente a diferentes aplicações. Assim, a melhor arquitetura para uma aplicação pode não ser a melhor para uma outra aplicação.

1.1.1 - Classificando pelo número de instruções e dados que executam.

Segundo o Almasi [Alma 89], os processadores paralelos podem ser separados em duas classes distintas, o SIMD (Single Instruction stream, Multiple Data stream) e o MIMD (Multiple Instruction stream, Multiple Data stream). O SIMD, como diz o nome, manipula vários dados com uma única instrução através de recursos de hardware. Estes processadores já têm classes de aplicações relativamente bem definidas, ou seja, se destinam a classes de problemas em que vários dados devem passar pelos mesmos cálculos, sem depender uns dos outros, mas certos autores não consideram estas arquiteturas como sendo multiprocessador. Mesmo na classe de SIMD podem-se manipular os dados de forma variada, tendo até o MSIMD (Multiple SIMD) [Alma 89]. O MIMD faz uso de vários processadores que, em muitos casos, têm arquiteturas semelhantes às dos processadores tradicionais (máquina de Von Neuman) [DeCe 89].

Existem dois grandes grupos de máquinas MIMD. O primeiro grupo caracteriza-se pelo forte acoplamento entre os processadores como na figura 1.1, onde os processadores podem acessar memória comum.

O segundo grupo é aquele em que cada processador tem sua própria memória independente e os dados são transferidos através de linhas de comunicação como na figura 1.2.

Processadores com Memória Comum

Esta classe de arquitetura é conhecida, também, por arquitetura com processadores fortemente acoplados. Tem a vantagem de poder transferir grande volume de dados em

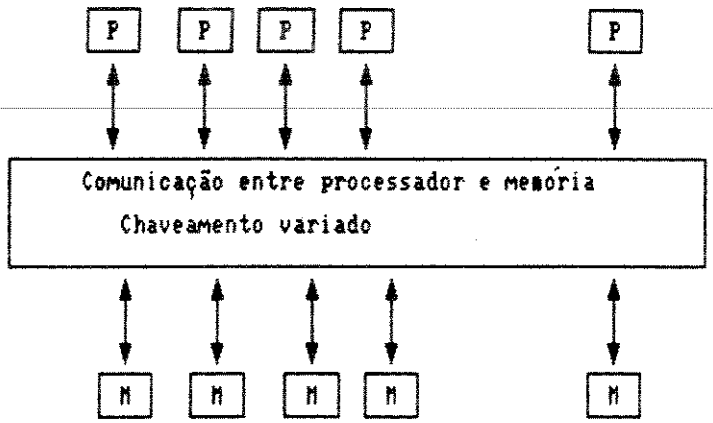


Fig. 1.1 – Vários processadores acessando a mesma memória

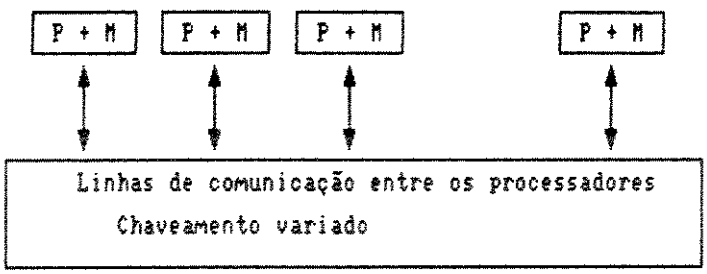


Fig. 1.2 – Processadores com memórias independentes

curto espaço de tempo pela transferência de posse da memória. Em alguns casos, como quando se faz uso da memória multiporta, a posse e acesso são simultâneos, podendo ocorrer apenas ligeiros atrasos de leitura. Na figura 1.3 pode-se ver um exemplo de acoplamento via memória "dual port" que, como diz o próprio nome, permite que dois processadores possam acessar um único bloco de memória. Nestas condições, automaticamente inibe o outro a acessar a memória. Este tipo de ligação envolve, além das linhas de dados, linhas de endereço e também linhas de controle, dificultando a expansividade e aumentando significativamente o custo.

Outra forma de vários processadores acessarem a memória é através do barramento comum como na figura 1.4.

Esta arquitetura permite aumentar o número de processadores sem aumentar em muito a fiação, porém apresenta duas grandes restrições. A primeira é como resolver a disputa dos processadores que podem tentar acessar o barramento. As soluções encontradas, seja de controle centralizado, seja de controle descentralizado, costumam ter perda de tempo na comutação. A segunda restrição é devido ao fato do barramento servir de "gargalo", pois quando um dos processadores acessá-lo a memória nenhum outro pode

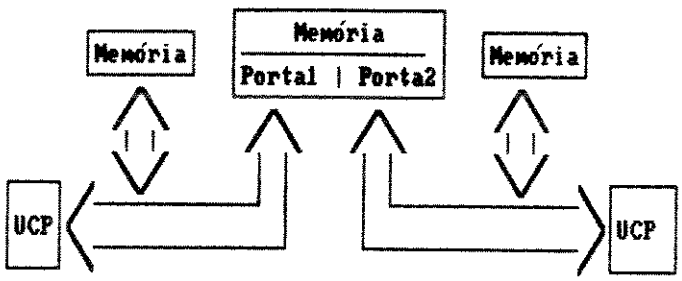


Fig. 1.3 – Ligação via memória "dual port"

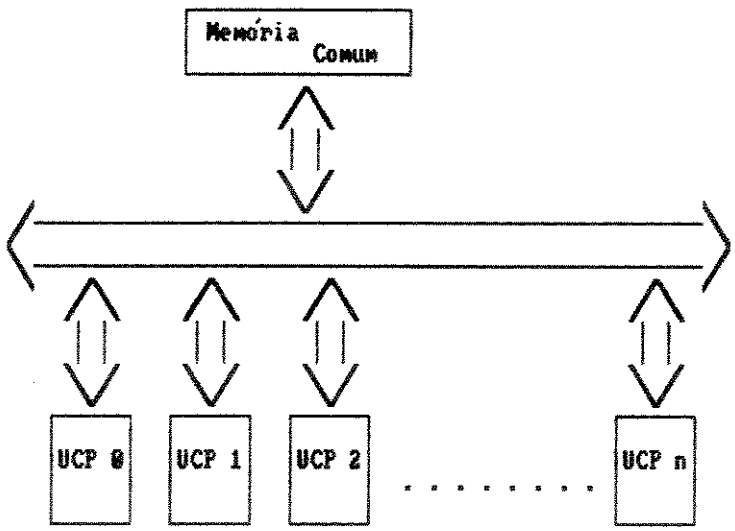


Fig. 1.4 – Ligação via barramento comum

acessá-lo. Existe uma variante do barramento comum que é o multibarramento comum, composto por vários barramentos comuns, mas o custo e o número de linhas também aumentam significativamente.

Uma forma de vários processadores poderem acessar a vários blocos de memórias é fazer uso do "cross-bus" como na figura 1.5. Mas, além do elevado custo e grande número de linhas envolvidas, o serviço de chaveamento deste "cross-bus" é complexo e a complexidade aumenta mais ainda quando se deseja maior flexibilidade de comutação ou se aumenta o número de processadores e blocos de memória.

Processadores com memória distribuída

Esta classe de arquitetura é também chamada de arquitetura com processadores fraca-

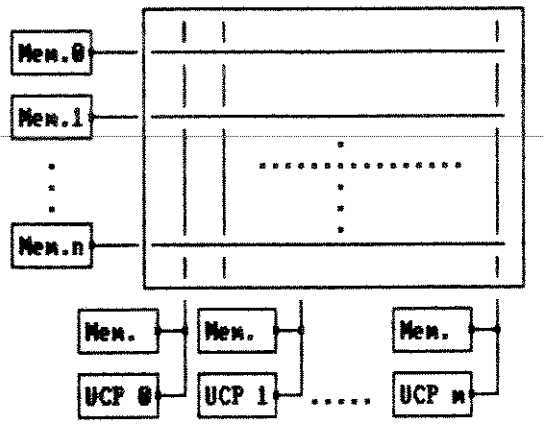


Fig. 1.5 – Arquitetura cross-bus

mente acoplados[DeCe 89] e a interação entre os programas é normalmente feita através de envio de mensagens, podendo envolver mecanismo de interrupção ou ADM (acesso direto a memória)[Hwang 85]. Caracteriza-se por ser eficiente quando a interação entre as tarefas são em pequenos volumes; além de diminuir os conflitos na memória, envolve menor quantidade de fios, podendo em muitos casos transferir a distâncias maiores. Normalmente o custo é menor do que o das arquiteturas com memória comum e possui melhor expansividade, tendo evoluído muito nos últimos anos.

Podem-se construir várias estruturas com processadores de memória distribuída, dependendo do número de portas de comunicação existentes em cada processador.

A estrutura mais simples e fácil de administrar é a linear, com apenas duas portas, como mostra a figura 1.6.

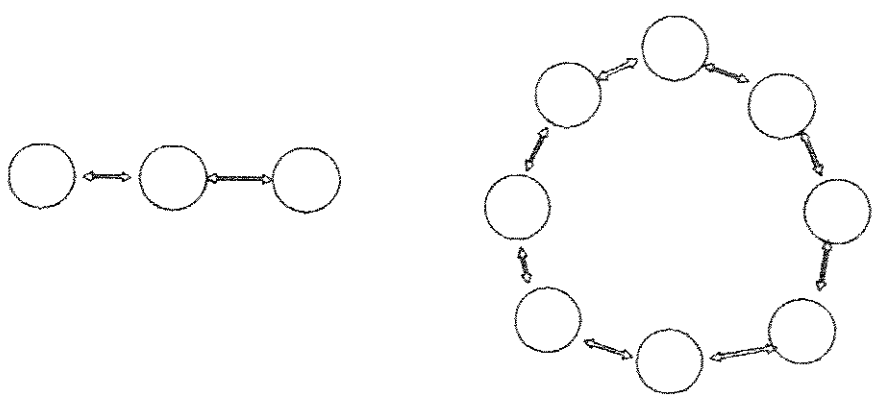


Fig. 1.6 – Exemplo de conexão linear

Aumentando-se para 3 o número de interfaces, tem-se a arquitetura em árvore binária, vista na figura 1.7. Se, por outro lado, o número de portas for de 4 a 8, podem-se montar

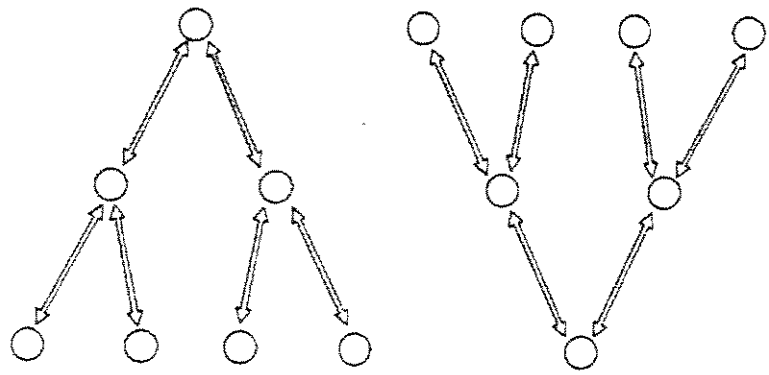


Fig. 1.7 – Arquitetura em árvore

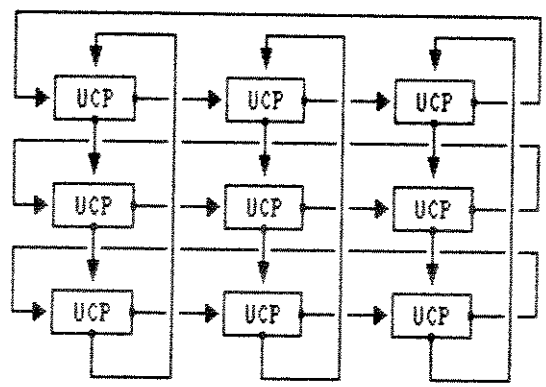


Fig. 1.8 – Arquitetura em malhas

"malhas" como mostra a figura 1.8.

Para os casos em que o número de portas é maior ou igual a três, é possível construir estruturas tridimensionais, como na Fig.1.9.

Em particular, se o número de portas é $log_2 N$, sendo N o número de processadores e uma potência de 2, pode-se construir um hipercubo com aspecto multidimensional como na figura 1.10.

Para se ter maior flexibilidade de ligação, pode-se também fazer o uso do "cross-bar",

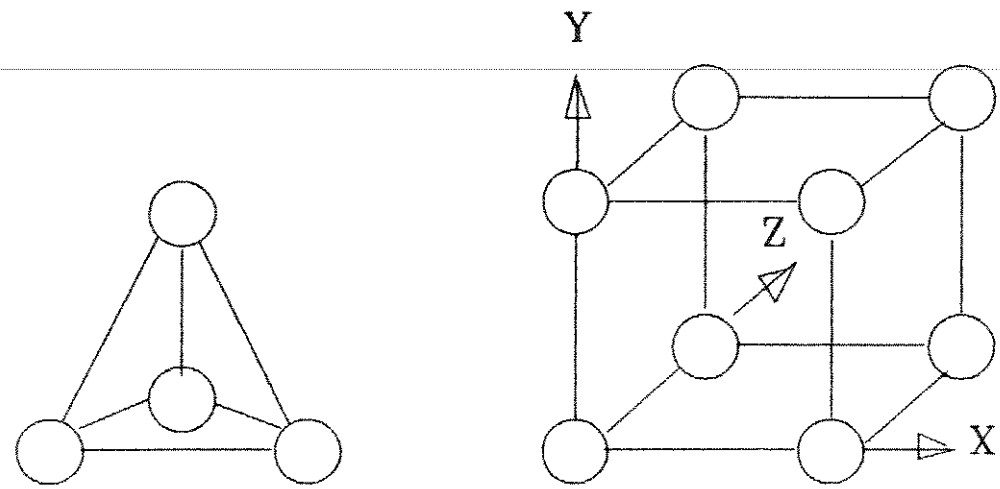


Fig. 1.9 – Arquitetura com figuras tridimensionais

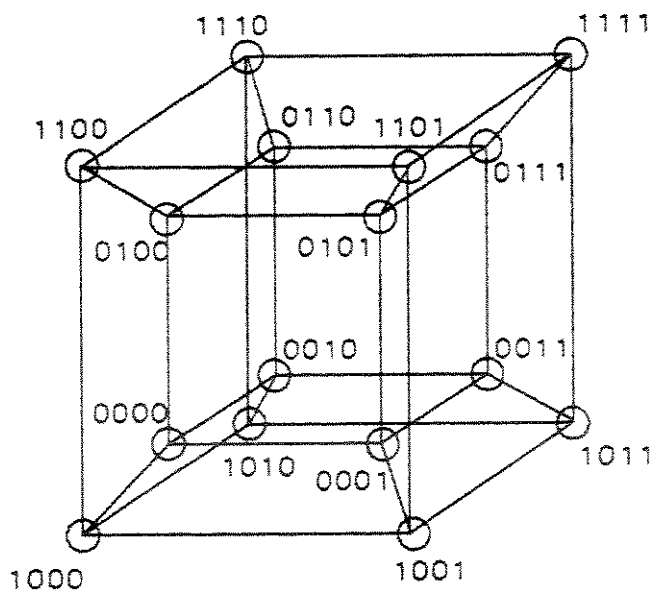


Fig. 1.10 – Arquitetura com figuras multidimensionais

ou circuitos comutadores [Wu84] como na figura 1.11.

Não é difícil perceber que, conforme a complexidade da estrutura aumenta, a tendência é aumentar também a complexidade do roteamento e o risco de erro, o que aumenta o processamento na administração do multiprocessador para que se possa evitar o envio desnecessário de mensagens de controle, diminuindo assim a possibilidade do multiprocessador entrar em "loop" ou em "dead-lock". A comutação dinâmica de um cross-bar exige um tempo significativo no processamento da comutação e administração e, portanto,

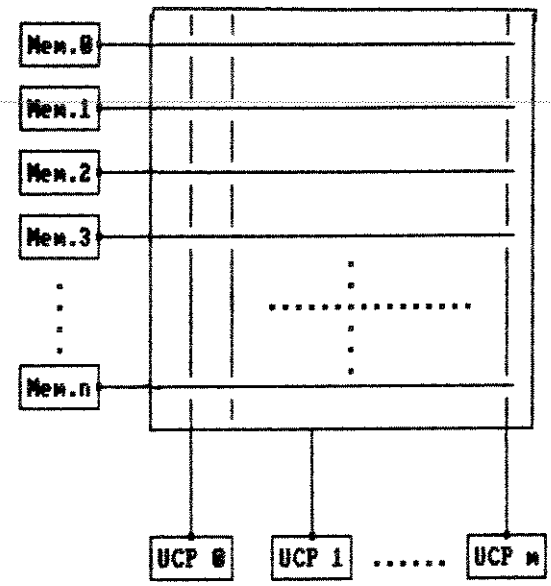


Fig. 1.11 – Arquitetura com "cross-bar" e circuitos comutadores

deve-se limitar o chaveamento.

1.1.2 - Classificando segundo localização do controle

O controle pode ser centralizado ou distribuído. O controle centralizado tem a vantagem de permitir soluções simples para problemas como conflitos e disputas mas, dependendo da arquitetura, pode ocorrer significativo aumento de mensagens de controle entre os processadores. Tem a desvantagem de que um problema na unidade onde está implementado o controle paralisa completamente o sistema.

O controle pode também ser feito tanto por hardware como por software. O controle por hardware tem a vantagem de maior velocidade mas, por outro lado, tende a ter maior custo e menor flexibilidade.

1.1.3 - Classificando segundo técnica de transmissão

A técnica de transmissão de sinais também varia muito. Além dos meios de comunicação, pode-se transmitir em paralelo ou serial, o que por sua vez pode-se multiplexar no tempo. A transmissão paralela permite maior velocidade de transferência de dados, mas aumenta também o custo, e o número linhas de comunicação dificulta a expansividade.

Uma linha pode também ser compartilhada por vários UCPs e assim ser ser multiplexada no tempo ou simplesmente enviar dados na forma de "broadcasting". Para organizar este tipo de transmissão usa-se muitas vezes a "passagem de bastão" [Ben 82].

1.2 - Fatores que influenciam na escolha de uma arquitetura

Quando se escolhe um multiprocessador, além da adequação para a aplicação, devem-se levar em conta muitos outros fatores como por exemplo o custo, a expansividade, a

facilidade de manuseio, a confiabilidade e também a disponibilidade. Em seguida, cada um deles é resumidamente analisado.

O custo costuma variar muito de arquitetura para arquitetura, recebendo ainda a influência do volume da produção. Em alguns casos o custo do multiprocessador aumenta exponencialmente com o número das UCPs; em outros casos a variação é próxima do linear.

A expansividade é outro fator importante. Além do fator custo, em certas arquiteturas a expansão pode tornar-se penosa ou como nos casos de barramento comum, o desempenho por unidade simplesmente pode cair conforme aumenta o número das UCPs até chegar a um ponto em que o aumento das UCPs não aumenta a velocidade final.

O volume de dados que os processos devem trocar e a freqüência de troca também vão influenciar na escolha. Troca de grandes quantidades de dados pode significar que a melhor escolha é usar uma arquitetura com memória comum. Se, porém, a troca de informações for freqüente, certas arquiteturas de memória comum também podem mostrar-se inadequadas. Em casos de troca esporádica de pequeno volume de dados é mais fácil e barato fazer o uso de linhas de comunicação, evitando assim o problema da disputa da memória comum que, geralmente, envolve problemas bastante complexos.

Homogeneidade e facilidade de determinação de rotas por parte do software é outro fator importante. Arquiteturas complexas podem dificultar significativamente a programação, além de reduzir a confiabilidade do software.

1.3 - Software

Os recursos de software disponíveis nos sistemas operacionais e linguagens fazem grande diferença para o usuário. Estes recursos podem variar desde a forma como o sistema carrega os processos através da rede, passando pelo apoio à troca de dados e sincronizações até completar com deputação e detecção de erros. Uma boa documentação tanto de hardware como de software também é muito importante. Trabalhar numa máquina sem o devido apoio por parte do sistema operacional e da linguagem significa fazer um esforço muito grande no software básico, ou fazer um software de aplicação limitada tanto na aplicação como na exploração dos recursos de hardware.

Capítulo 2

O PROBLEMA DA DISTRIBUIÇÃO DE TRABALHO

Em processamentos com processadores concorrentes existe uma fase anterior ao carregamento de programas e dados que é a divisão de tarefas entre os diversos processadores. O objetivo é dividir o trabalho (domínio do problema) entre os processadores, de maneira que nenhum trabalho deixe de ser executado ou que seja executado mais de uma vez. Além disso, o tempo total de execução deve ser minimizado.

Seja o problema de calcular uma função custo qualquer f, cujas variáveis sejam as mesmas que definam os estados de um autômato finito. Tal problema pode ser atacado de modo concorrente, através da partição do domínio de f. Supondo-se que o domínio de f tenha n elementos e que existam m processadores disponíveis, pode-se dividir igualmente o trabalho entre os m processadores, desde que n seja um múltiplo inteiro de m, $n = m \times p$. Desta forma, cada processador executará o trabalho de calcular os p valores de f, associados aos p elementos distintos do domínio de f,a ele assinalados.

Se o tempo de processamento para calcular o valor f for independente dos valores das variáveis da função e se todos os processadores começarem a trabalhar no mesmo instante, terminarão o trabalho simultaneamente. Por outro lado, se o tempo de processamento para o cálculo do valor de f depender dos valores das variáveis, o tempo total de processamento será igual ao tempo de processamento do processador de maior tempo de processamento. Nestas condições o "speedup" pode cair a níveis baixíssimos, o mesmo acontecendo com a eficiência. É bom observar que a complexidade do problema de distribuição de trabalho no exemplo aumenta se o número de elementos do domínio de f não for múltiplo inteiro do número de processadores.

Para problemas de grande complexidade de distribuição de trabalho, tem-se como opção fazer uma redistribuição de trabalho ao longo do processamento, denominada *redistribuição dinâmica de trabalho*, feita sem parar o processamento. Entretanto, mesmo para estes problemas, procura-se fazer uma distribuição inicial de trabalho balanceada.

2.1 A Redistribuição Dinâmica de Trabalho

Da mesma forma que a distribuição inicial de trabalho balanceada, a redistribuição dinâmica de trabalho apresenta uma série de dificuldades para ser implementada eficientemente. Entretanto, existem classes de problemas que permitem, de modo eficiente, o uso de metodologias de distribuição dinâmica de trabalho. Uma delas é a *classe de problemas de domínio discreto, cujos domínios possam ser particionados de tal forma que os elementos das partes sejam gerados por autômatos autônomos e isomorfos entre si.*

O particionamento do domínio garante que nenhum trabalho será executado mais de uma vez nem deixará de será executado. Os elementos das partes a serem gerados por autômatos autônomos possibilitam a diminuição do tamanho da mensagem de envio de

serviço de um processador para outro, pois basta enviar o estado inicial do autômato para ele assumir o processamento. O fato de os autômatos autônomos serem isomorfos entre si, por outro lado, possibilita, na prática, a distribuição de cópias de um único programa para cada processador. Isto porque, sendo os autômatos isomorfos entre si, eles diferem somente na nomeação dos estados, mantendo as mesmas funções de entrada e de saída.

Entre os problemas pertencentes à classe aqui discutida, encontram-se todos aqueles cujo grafos de transição dos estados dos autômatos sejam isomorfos a um hipercubo ou a um permutoedro. Fig 2.6 e fig 2.7 , respectivamente. A fig 2.1 apresenta o particionamento de um grafo em sub-grafos isomorfos entre si.

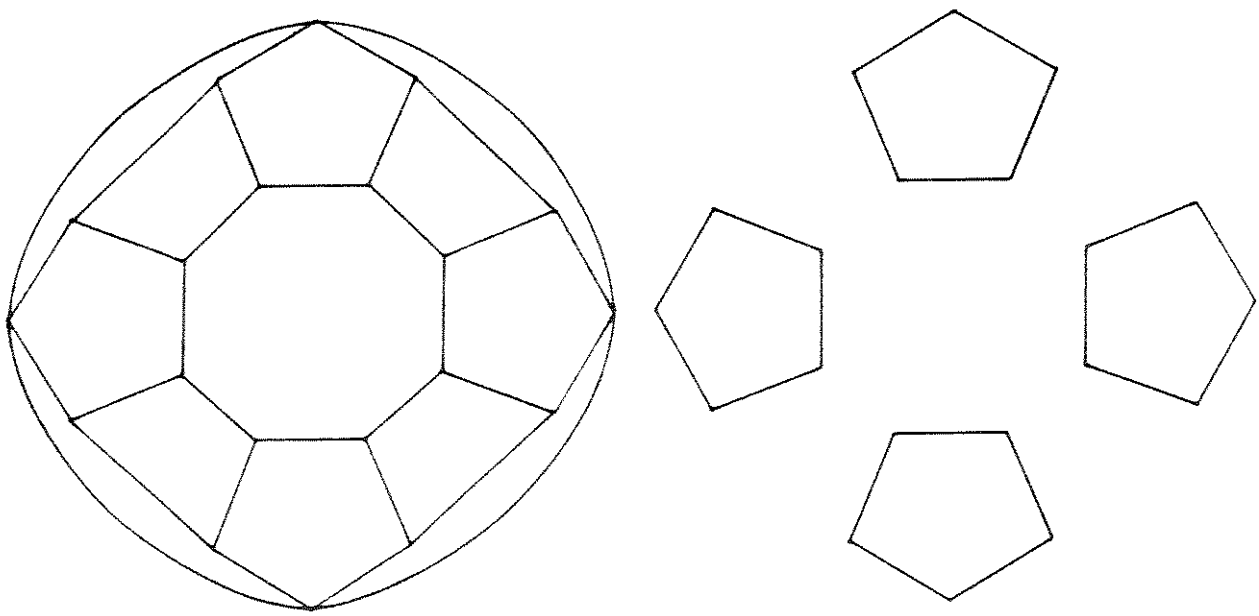


Fig. 2.1 – Grafo particionado em sub-grafos isomorfos entre si.

2.2 O Formalismo Envolvido

A Teoria do Autômato define um autômato como uma quíntupla $S =< I, Q, Z, \delta, \omega >$, onde I é o conjunto de símbolos de entrada permitidos, Q é o conjunto de estados do autômato, Z é o conjunto de símbolos de saída, ω é um mapeamento de $I \times Q$ em Q denominado função de saída. O autômato S é dito ser de *estados finitos* se o número de elementos de Q for finito. Por outro lado, se o conjunto I contiver um único elemento, diz-se que o autômato é autônomo [Booth 67].

Uma das formas de representação de autômatos é através do seu diagrama de transição, como exemplificado na fig.2.2. No exemplo, se o autômato estiver no estado q_0 e se a entrada for ativada com a_2, o autômato responde com uma saída 0 e passa ao estado q_0. Se, por outro lado, a entrada for ativada com a_1, o autômato responderá com uma saída 0 e mudará para o estado q_2. De maneira similar poderão ser entendidas as outras transições do exemplo.

Dois autômatos $S =< I, Q, Z, \delta_s, \omega_s >$ e $T =< I, U, Z, \delta_t, \omega_t >$ são ditos isomorfos se existe um mapeamento biunívoco r, de Q sobre U, tal que $\omega_s(i,q) = \omega_t(i, r(q))$ e $r(\delta_s(i,q)) = \delta_t(i, r(q))$ para todo $i \in I$ e todo $q \in Q$. O mapeamento r é chamado de Q sobre U, tal que $\delta_s(i,q) = \omega_t(i, r(q))$ e $r(\delta_s(i,q)) = \delta_t(i, r(q))$ para todo $i \in I$ e todo $q \in Q$. O mapeamento r é chamado de *isomorfismo grafo*. Decorre então que, se dois autômatos são isomorfos os seus diagramas de transição diferem somente por uma renomeação dos estados.

Um conceito importante, usado para verificar se dois autômatos são isomorfos, é o conceito de *seqüência de grafos de transição* expresso da seguinte forma[41]: Seja $S = < I, Q, Z, \delta, \omega >$ um autômato onde $I = [a_1, a_2, \ldots, a_n]$. A seqüência de grafos definidos pelos diagramas de transição de autômato S para cada uma das entradas a_i em separado, se constitui na seqüência de grafos de transição de S, escrita como:
$L(S) = ((Q, \delta_{a_1}), (Q, \delta_{a_2}), \ldots, (Q, \delta_{a_i}), \ldots, (Q, \delta_{a_n}))$.

O elemento $(Q.\delta_{a_i})$ da seqüência $L(S)$ representa o grafo de transição de todos elementos do Q para quando a entrada é a_i, sendo δ a função próximo estado. A fig.2.2 representa um diagrama de transição cuja seqüência de grafos é mostrada na fig.2.3.

Um teorema enunciado e demonstrado em Read [Read 72], no capítulo entitulado "Algebraic Isomorphism for Graphs of Automata", sugere um método para a verificação de isomorfismo entre dois autômatos. Enuncia o teorema:

Teorema. *Sejam dois autômatos S e S' pertencentes à classe de autômatos com o mesmo conjunto de símbolos de entrada, sendo $\phi(G_1, G_2, \ldots, G_k)$ e $\phi(G'_1, G'_2, \ldots, G'_i)$ as sequências de grafos de transição para o conjunto I de símbolos de entrada. O autômato S é isomorfo ao autômato S' se e somente se existir um isomorfismo grafo η tal que G_i seja isomorfo a G'_i sob η, para $i = 1, 2, 3, \ldots, k$.*

Desta forma, para se determinar se dois autômatos são isomorfos, determinam-se as seqüências grafos de cada um deles e procura-se um isomorfismo que satisfaça a todos os pares de elementos das seqüências grafos definidos pelas entradas em separado.

$S =< I, Q, Z, \delta, \omega >$

$Q = [q_0, q_1, q_2, q_3]$

$I = [a_1, a_2]$

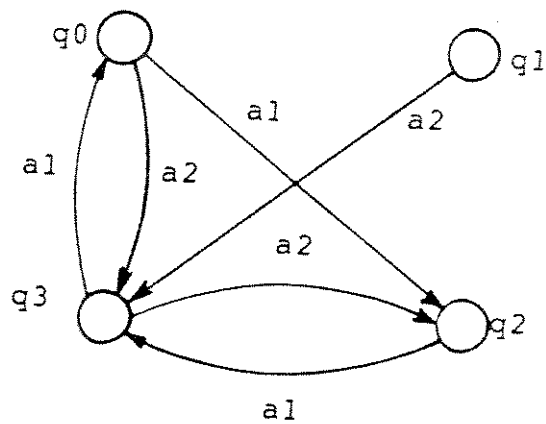


Fig 2.2 – Diagrama de transição associado a um autômato.

$L(S) =$

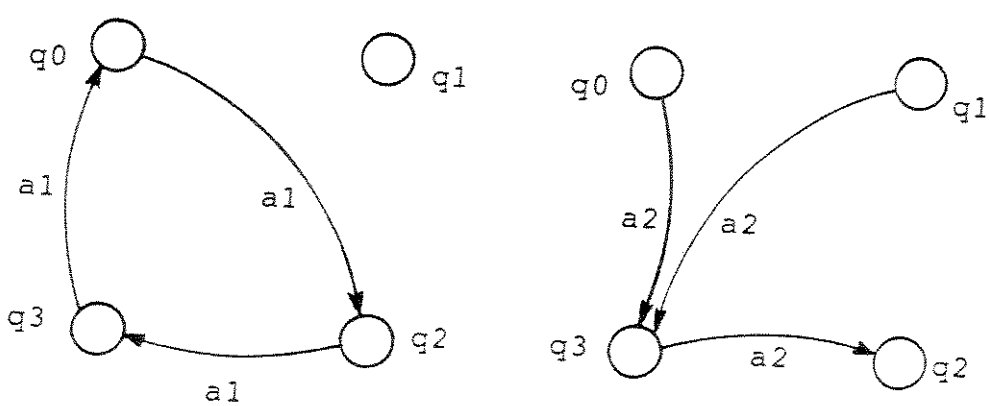


Fig 2.3 – Seqüência de grafo da fig 2.2.

2.3 A Metodologia "Quem Acabar Primeiro Ajuda os Outros"

A metodologia "Quem acabar primeiro ajuda os outros" é uma metodologia de administração distribuída onde cada processador administra a sua pilha de serviços e, quando detecta que a pilha atingiu um limite mínimo de serviços, envia uma mensagem aos processadores de estrutura pedindo serviços. Se algum processador da estrutura receber a mensagem e identificar que pode mandar serviço ao processador requisitante, o faz através de uma mensagem do tipo de envio de serviço. Aqui, o termo estrutura é usado no sentido de implementação física da topologia de ligações entre processadores concorrentes.

Existem três questões que devem ser consideradas sobre esta metodologia:

A primeira é sobre a hipótese de uma mensagem de pedido de serviço ser enviada em "broadcasting", o que poderia acarretar uma avalanche de serviços ao processador requisitante, porque não existe controle de quem deve enviar serviço ao requisitante.

A segunda questão que pode ser levantada é sobre o controle do término do processamento total, já que a administração é distribuída.

Finalmente, a terceira questão diz respeito à variação do pior caso do número de mensagens espalhadas entre os processadores da estrutura. Esta questão surge da suspeita de que as mensagens de pedidos e envios de serviços possam congestionar o tráfego nas ligações físicas da estrutura.

Todas estas três questões têm soluções elegantes para as estruturas de processadores que admitam pelo menos um ciclo hamiltoniano sobre elas.

A primeira questão é resolvida facilmente se uma mensagem de pedidos de serviço é enviada sobre um ciclo hamiltoniano. Assim, todos os processadores da estrutura podem receber a mensagem e, se um deles poder enviar serviço ao requisitante, a propagação da mensagem é inibida e o processador envia serviço. Entretanto, se a mensagem voltar ao requisitante, este fica informado de que, quando acabar o serviço relacionado na pilha de serviços, pode considerar o fim do seu processamento.

O controle de término de processamento total é feito através do envio de mensagem de controle sobre um ciclo hamiltoniano com início no processador "root" (processador ligado ao hospedeiro) e emitida logo no início do processamento. O processador "root" fica, então, esperando o retorno desta mensagem para definir o fim do processamento total. Cada processador da estrutura só propaga a mensagem de fim de processamento quando termina o seu processamento local, isto garante que, quando a mensagem voltar ao processador "root", todos os processadores já tenham terminado os seus respectivos processamentos.

O pior caso do número de mensagens espalhadas sobre os processadores da estrutura pode ser estudado considerando-se que só existem quatro tipos de mensagens: pedido de serviço, envio de serviço, mensagem de controle e mensagem de circulação de dados.

Os três primeiros tipos de mensagens citados já foram conceituados anteriormente, faltando somente o tipo de circulação de dados que se refere à troca de dados entre processadores. Em geral, estas mensagens trafegam sobre um ciclo hamiltoniano e fazem circular, sobre a estrutura, valores de variáveis que são comuns aos processos distribuídos entre os processadores. Por exemplo: máximo e mínimo de uma função que podem ser atualizados por qualquer um dos processadores.

Desta forma, o número total de mensagens M distribuídas entre os processadores da estrutura pode ser dado por:

$M = m_p + m_e + m_c + m_k$

Onde:

m_p = número de mensagens de pedido de serviço.
m_e = número de mensagens de envio de serviço.
m_c = número de mensagens de circulação de dados.
m_k = número de mensagens de controle.

Pode-se ver facilmente que o pior caso do número de mensagens espalhadas entre os processadores da estrutura ocorre numa situação em que todos os processadores pedem serviços ao mesmo tempo. Mesmo assim, tem-se na estrutura um número de mensagens de pedido de serviços igual ao número de processadores. Neste caso, o número de mensagens de envio de serviço é zero, porque se todos os processadores estão pedindo serviços, nenhum deles pode estar enviando serviço. Portanto, o número de total de mensagens passa a ser $M = N + m_c + m_k$. Onde N é o número de processadores na estrutura.

Considerando-se ainda o valor da densidade média de mensagens por linhas físicas (t_m) no pior caso, verifica-se que, para estruturas regulares com l_f pares de linhas físicas de entrada e saída por processador, o valor de t_m é dado por: $t_m = M/(N * l_f)$.

Numa estrutura hipercúbica [Seitz 85], o valor de t_m no pior caso é dado pela seguinte expressão:

$t_m = (N + m_c + m_k)/(N * log_2 N)$

Para se ter uma idéia real da ordem de grandeza de t_m numa estrutura hipercúbica, pode-se considerar o exemplo onde $N = 1024 m_c = 5 e m_k = 1$. Obtém-se:

$t_m = 1030/(1024 * 10) = 0.1006$.

Ou seja, no pior caso, em média, somente dez por cento das ligações físicas estariam sendo usadas.

2.4 A Implementação de Autômatos Dedicados a Metodologia "Quem Acabar Primeiro Ajuda os Outros"

A implementação de autômatos dedicados a metodologia "Quem acabar primeiro ajuda os outros" visa atender às necessidades de comunicação e controle entre processadores, bem como do processamento do problema de aplicação propriamente dito.

A comunicação e controle, englobam o recebimento, envio, interpretação e administração de mensagens. Em paralelo, o processamento do problema aplicativo diz respeito única e exclusivamente ao algoritmo que determina a solução do problema a ser resolvido.

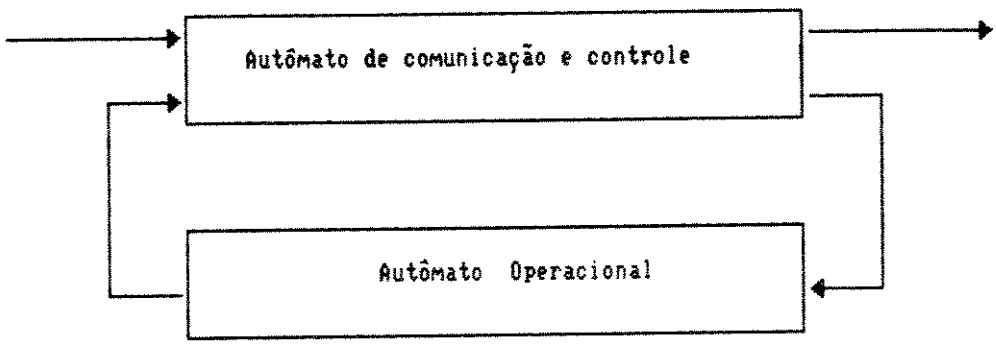


Fig. 2.4 – Diagrama esquemático da Cooperação entre os autômatos de comunicação e controle e o autômato operacional.

É natural, então, que se implementem em cada processador dois autômatos. O primeiro denomina-se *autômato de comunicação e controle* e o segundo, *autômato operacional* [Thay 84] e [Glush 65], apresentados na fig.2.4.

O autômato de comunicação e controle é responsável pela execução das seguintes tarefas:

a) Interpretação e execução dos comandos de uma linguagem de comunicação entre processadores, tendo um subconjunto de comandos que viabilizem a comunicação entre o autômato de comunicação e controle e o autômato operacional.
b) Administração de uma pilha de serviços, onde são alocados os dados correspondentes à seqüência de entradas válidas para o autômato operacional.
c) Administração de pilha de mensagem entre processadores.

Por outro lado, o autômato operacional é responsável pela execução das seguintes tarefas:

a) Interpretação e execução do subconjunto de comandos da linguagem de comunicação do autômato de comunicação citada no item anterior.
b) Execução do algoritmo que determina a solução do problema aplicativo.

Analisando-se as tarefas dos autômatos, verifica-se que o autômato de comunicação e controle independe da aplicação e que o autômato operacional tem parte que também independe da aplicação. Isto significa que, do ponto de vista da reusabilidade do software, o autômato de comunicação e controle é totalmente reutilizável, enquanto que o autômato operacional é parcialmente reutilizável.

2.5 Exemplo de Aplicação da Metodologia "Quem Acabar Primeiro Ajuda os Outros"

Como foi visto na seção 2.4, a metodologia proposta é mais fácil de ser aplicada em estrutura de processadores que permitem ciclos hamiltonianos. A estrutura hipercúbica é uma delas e será usada nos exemplos seguintes. A fig.2.5 ilustra uma estrutura hipercúbica de ordem 4, que se compõe de 16 processadores.

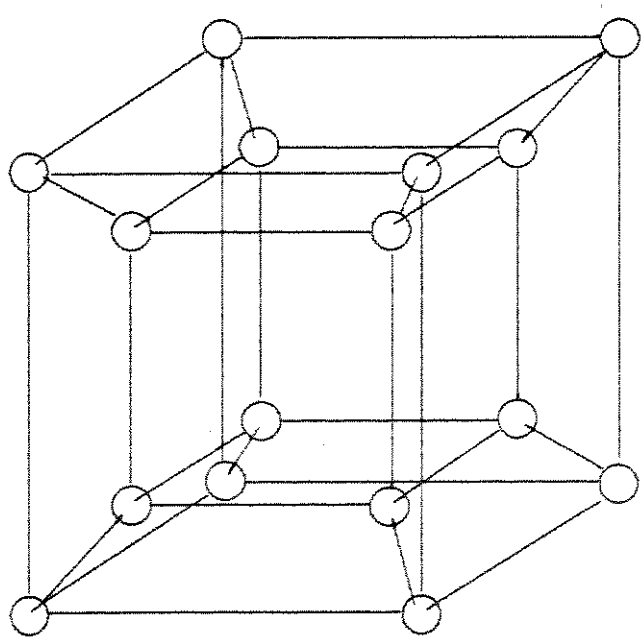


Fig. 2.5 – Estrutura hipercúbica de ordem quatro

2.5.1 Verificação de Tautologia

Diz-se que uma função booleana $f(x_1, x_2, \ldots, x_n)$ é tautológica se, para quaisquer valores das variáveis $x_1, x_2, \ldots, x_n$, o valor de f for sempre verdade.

Existem funções que são facilmente trabalhadas de maneiras a se demonstrar que são tautológicas. Entretanto, existem outras que requerem um grande esforço algébrico e cuja implementação no computador é de altíssima complexidade. Assim, em certos casos, a demonstração por exaustão, testando todas as possibilidades dos arranjos das variáveis de entrada, é uma solução para a determinação de tautologias, sendo que é limitada pela velocidade do processador utilizado. Esta opção é que será desenvolvida.

A divisão inicial de trabalho se baseia na propriedade de que todas as entradas permitidas da função f podem ser ordenadas por um hipercubo de ordem n [Birk 70]. Por sua vez, qualquer hipercubo pode ser particionado em sub-grafos isomorfos (hipercubos de ordem inferior). Portanto, na divisão inicial de trabalho, cada processador recebe um conjunto de hipercubos que definem seqüências de entradas para o cálculo de valores da

função f. Como o hipercubo permite um ciclo hamiltoniano, é necessário somente entregar a cada processador um vértice de cada hipercubo e implementar no autômato operacional um algoritmo que calcule a seqüência de vértices a serem percorridos.

Seja então uma função f com vinte variáveis de entrada e uma estrutura hipercúbica de processadores de ordem quatro, ou seja, com dezesseis processadores. Considere-se ainda que no autômato operacional esteja implementado um algoritmo que determine o caminho hamiltoniano sobre hipercubo de ordem oito. Portanto, tem-se: $2^{20} = 2^8 * 2^4 * 2^n$, onde 2^n é o número de hipercubos alocados para cada processador. Determina-se então h $= 8$.

No exemplo acima, a pilha inicial de serviços do autômato de comunicação e controle de um dado processador pode conter uma seqüência de serviços que, representada por números hexadecimais, tem o seguinte conteúdo: 000xx, 001xx,...,0FFxx, sendo xx qualquer número hexadecimal entre 00 e FF. Portanto, o autômato operacional, recebendo um dos elementos da pilha, gera uma seqüência de entradas para a função f. Se um dos valores de f for falso, então o processador envia uma mensagem de fim de processamento, devido à determinação de uma contradição.

É importante notar que, para um processador enviar trabalho para um outro processador, é necessário somente enviar parte do conteúdo de sua pilha de serviço.

2.5.2 Uma Solução para o Problema do Caixeiro Viajante.

Como existem várias instâncias de definição do problema do caixeiro viajante, a definição usada neste trabalho é explicitada a seguir: "Dado um conjunto de n cidades, sendo que cada uma delas se liga por estradas a todas as outras e conhecendo-se as distâncias entre todos os pares de cidades, pede-se calcular um caminho mínimo a ser percorrido por um caixeiro viajante, de maneira que ele passe somente uma vez por cada uma das cidades e volte à cidade de origem".

Considerando-se o enunciado do problema, é fácil provar que existem $(n-1)!/2$ caminhos hamiltonianos distintos. Pois, seja o conjunto de cidades $C = (c_1, c_2, \ldots, c_n)$ a serem consideradas no problema. Um caminho possível ao caixeiro, partindo da cidade c_n é dado por $c_n = (c_1, c_2, \ldots, c_n)$. Portanto, $(n-1)!$ representa o número de caminhos possíveis saindo da cidade c_n e voltando a cidade c_n. Entretanto, quando se leva em consideração a simetria do problema, ou seja, que a distância entre duas cidades independe da direção do percurso, encontra-se $(n-1)!/2$

Apesar de resolver o problema do caixeiro viajante somente para um número pequeno de cidades, a opção da busca por exaustão é, em certos casos, uma boa opção para se determinar a solução do problema, devido à simplicidade da implementação. No caso do processamento com processadores concorrentes, a solução do problema do caixeiro viajante pode também ser encaminhada sob uma administração distribuída, porque todas as permutações do conjunto de cidades a serem percorridas, podem ser ordenadas por um grafo, denominado permutoedro [Berge68], fig2.6. Além disso, como o permutoedro pode

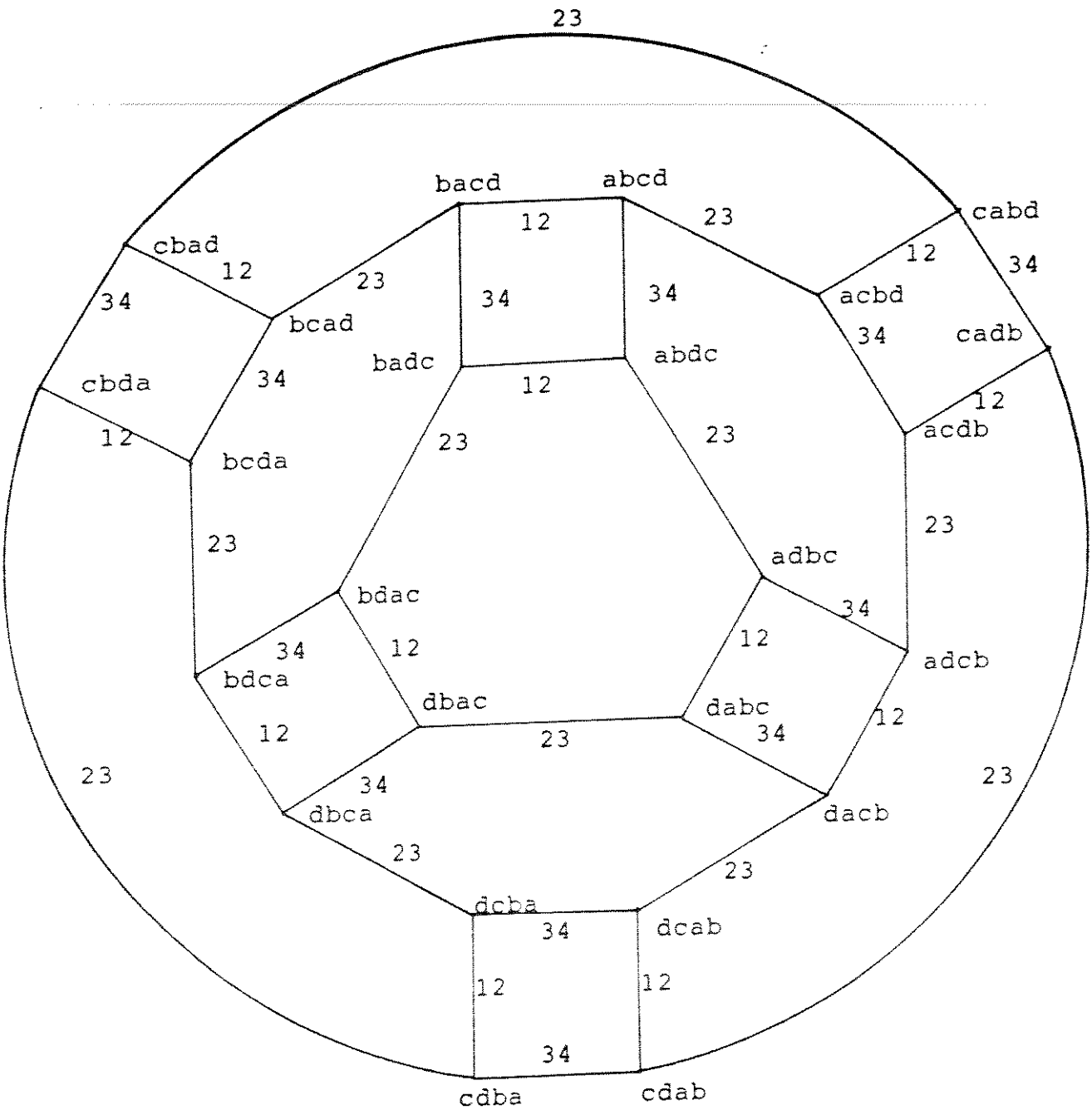


Fig. 2.6 – Permutoedro para quatro variáveis.

ser particionado em subgrafos isomorfos, exemplificado na fig.2.7, a metodologia proposta pode ser aplicada.

A implementação da solução do problema é feita de modo semelhante à implementação feita para o exemplo da determinação de tautologias, já que são satisfeitas as mesmas propriedades.

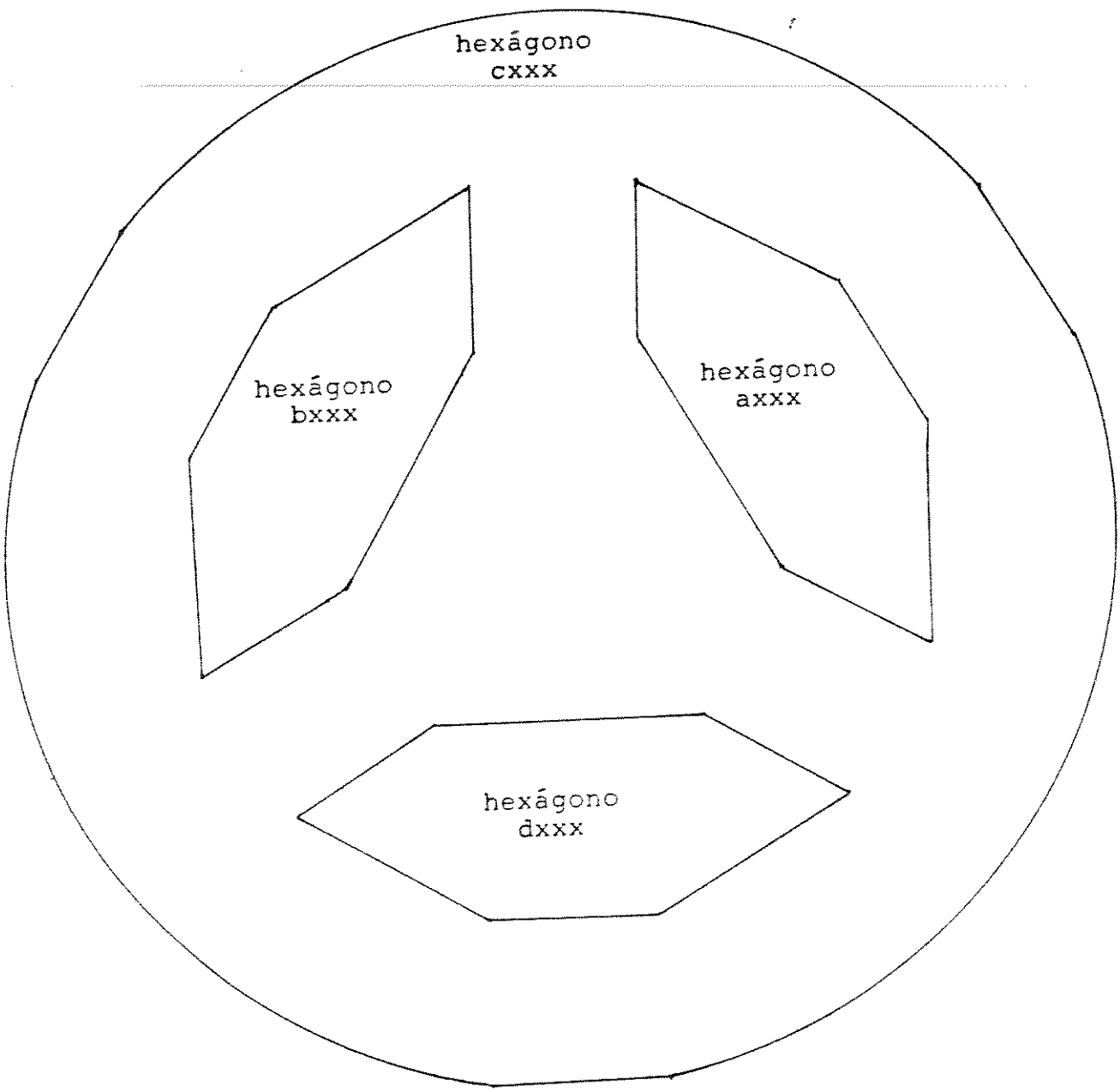


Fig. 2.7 – Permutoedro para quatro variáveis particionado em hexágonos.

2.6 Conclusões Sobre a Redistribuição de trabalho.

Considerando-se que não se tem uma estrutura de processadores que seja ótima para qualquer aplicação e que, da mesma forma, não existe metodologia de redistribuição de trabalho entre processadores concorrentes que seja adequada a qualquer aplicação, verificou-se neste trabalho o casamento de propriedades das estruturas de processadores com as propriedades do problema aplicativo, para se propor uma metodologia distribuída de redistribuição dinâmica de trabalho entre processadores concorrentes.

Os dois exemplos de aplicações apresentados, aparecem na solução de problemas nas áreas de pesquisa operacional, inteligência artificial e projeto de circuitos lógicos, o que mostra a abrangência da metodologia proposta.

Capítulo 3
ESCOLHA DA ARQUITETURA

Para implementar a proposta do capítulo anterior, não é difícil notar que as arquiteturas mais apropriadas são o VTM(Variable Topology Multicomputer)[Paker 83, DeCe 89] que permite adequar-se a várias topologias, e a arquitetura na forma do reticulado ou outra forma próxima a este. Assim, foram feitos estudos sobre VTM (no apêndice A tem-se o estudo e uma proposta de uma máquina VTM) e algumas máquinas com possibilidade de arquitetura em forma de reticulados. Foram realizados alguns testes com máquinas UNIX em forma de rede e com a máquina de barramento comum (no apêndice B tem-se o estudo sobre máquina de barramento comum).

Apesar de se ter chegado à conclusão de que máquinas VTM poderiam ser utilizadas para implementar o algoritmo proposto no capítulo 2, na época não foi possível encontrar nenhuma máquina deste tipo no país. Uma outra máquina que se apresentou promissora na época foi máquina com os Transputers (as primeiras máquinas estavam chegando ao País). Montando-as em forma de reticulados, poderiam servir para implementar o algoritmo. Após alguns estudos feitos sobre certos reticulados, encontramos um montado sobre álgebra de Boole chamado Hipercubo, que se mostrou muito prático e eficiente em termos de aplicação, sendo escolhido para fazer o teste de implementação. Seguem a explicação e estudos feitos com hipercubo.

3.1 Hipercubo

O Hipercubo é uma arquitetura para multiprocessamento apresentada em 1977 na Caltech, por Sally Browning e Bart Locanthi [Seitz 85], que propõe um prolongamento da Álgebra de Boole ao nível de arquitetura, o que facilita em muito a tarefa de identificação, análise, escolha de rota e processamento. Na figura 3.1 pode-se ver a organização em arquitetura 2N (2 dimensões binárias), tendo duas interfaces seriais por UCP. Aumentado este número de interface para 3, tem-se a arquitetura hipercubo 3N ou seja de dimensão 3, com 8 processadores comunicando-se entre si, o que pode ser visto na figura 3.2 . A arquitetura 4N já começa a ficar difícil de representar, mas pode-se ver na figura 3.3. Na figura 3.4 tem-se a arquitetura 5N com 32 processadores e 5 interfaces de entrada e saída serial por UCP, totalizando 80 linhas de comunicação, na verdade não é nada mais do que a figura 3.3 repetida duas vezes com cada processador ligado à sua imagem.

A arquitetura 6N com 64 processadores com 6 interfaces de entrada e saída serial em cada processador, totalizando 144 linhas de comunicação, pode ser vista na figura 3.5 e 3.6. A figura 3.5 é uma superposição de duas figuras 3.4, tendo cada processador ligado com a sua respectiva imagem.

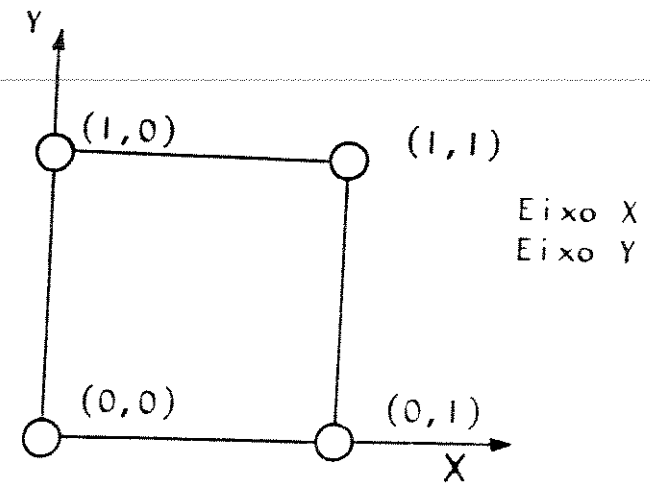


Fig. 3.1 – Hipercubo de dimensão 2N

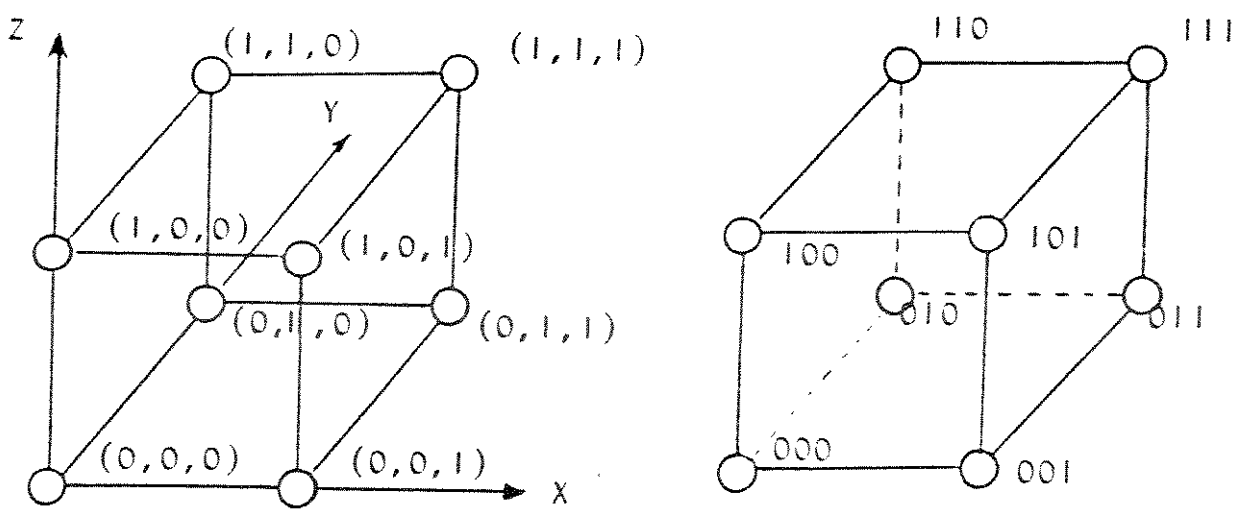


Fig. 3.2 – Hipercubo de arquitetura 3N

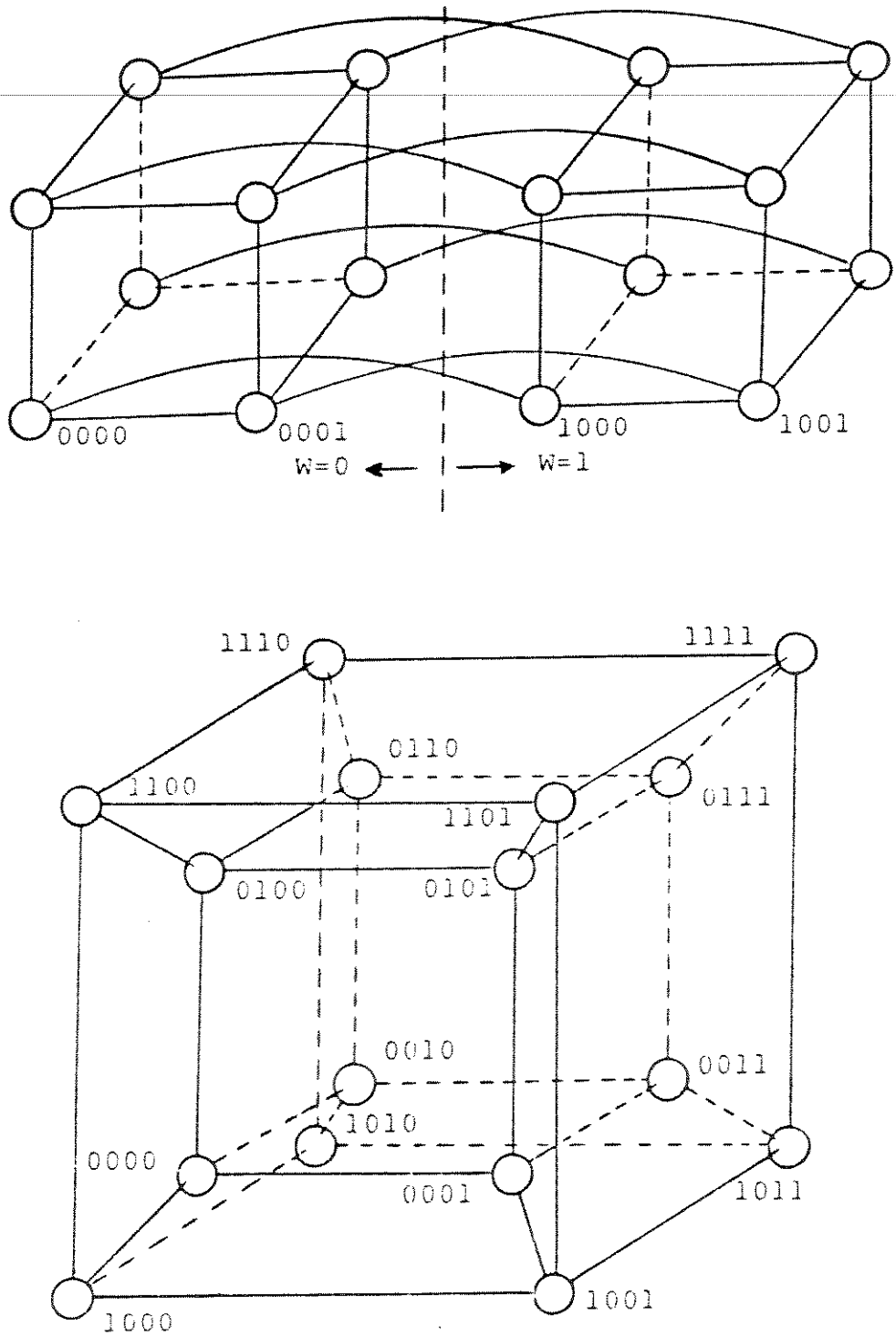


Fig. 3.3 – Hipercubo de dimensão 4N envolvendo 16 processadores e 32 linhas de comunicação

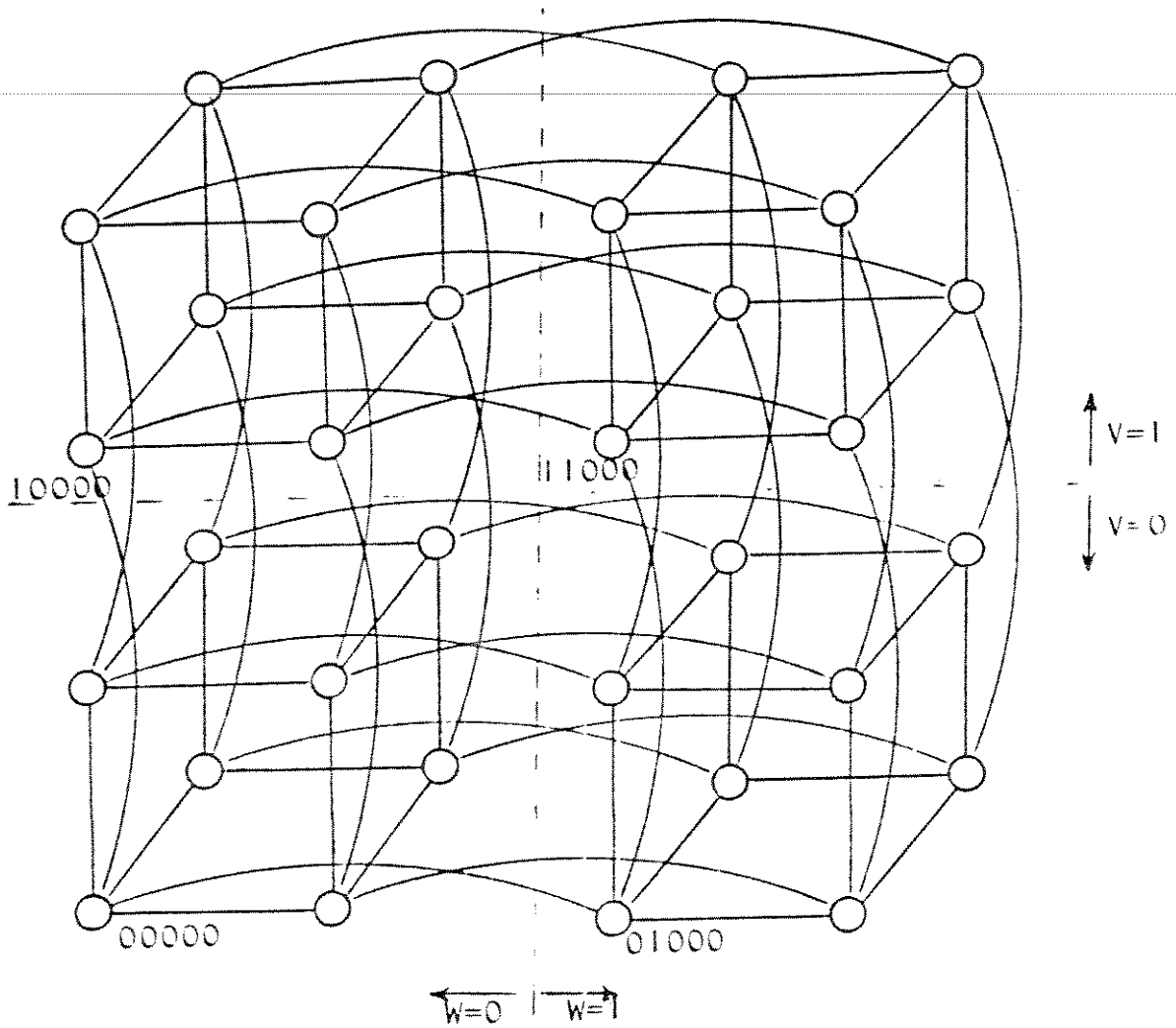


Fig. 3.4 – Hipercubo com dimensão 5N envolvendo 32 processadores e 80 linhas de comunicação

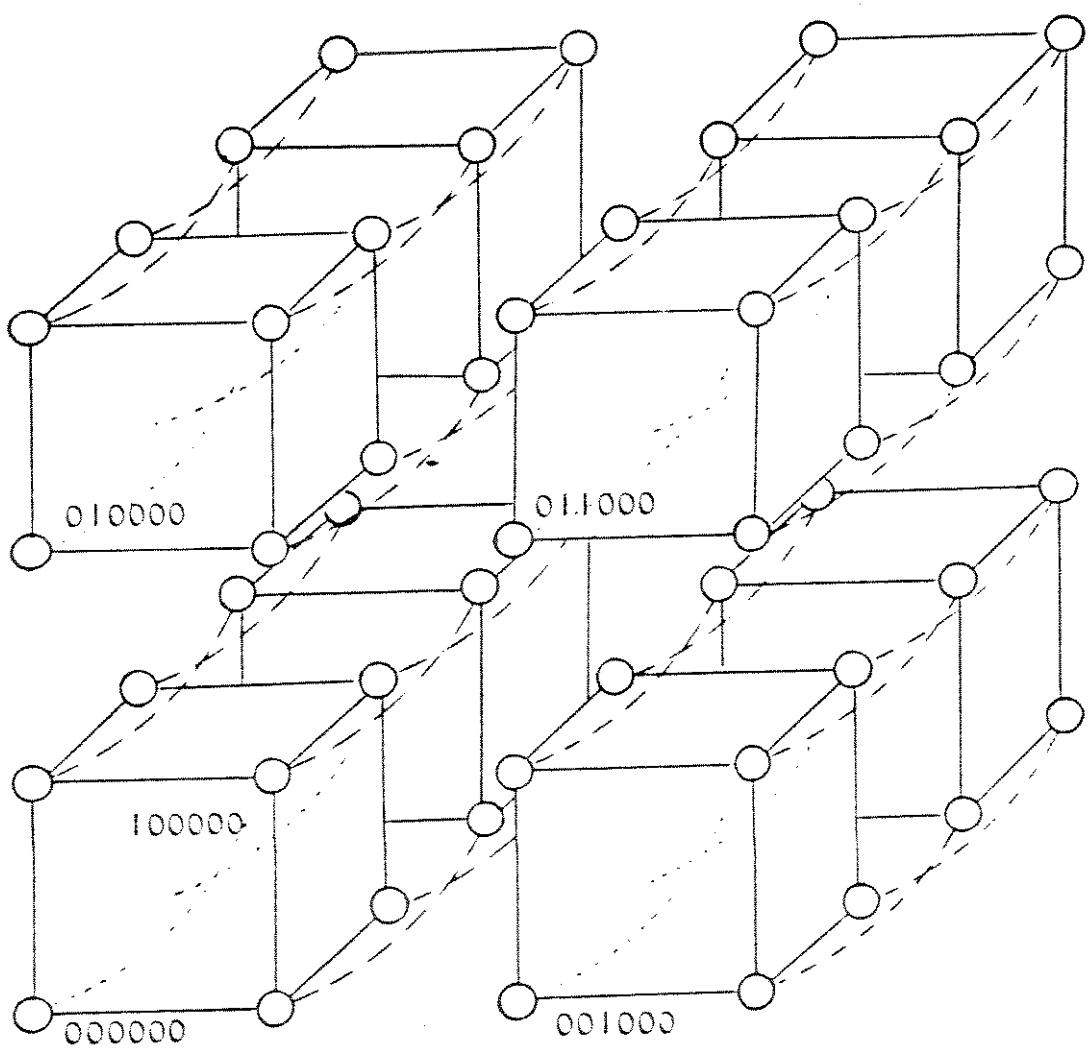


Fig. 3.5 – Hipercubo com dimensão 6N envolvendo 64 processadores e 144 linhas de comunicação

Na figura 3.5 estão representadas apenas as linhas que serão adicionadas em relação a 2 conjuntos de 5N (um conjunto no plano da frente e outro no fundo).

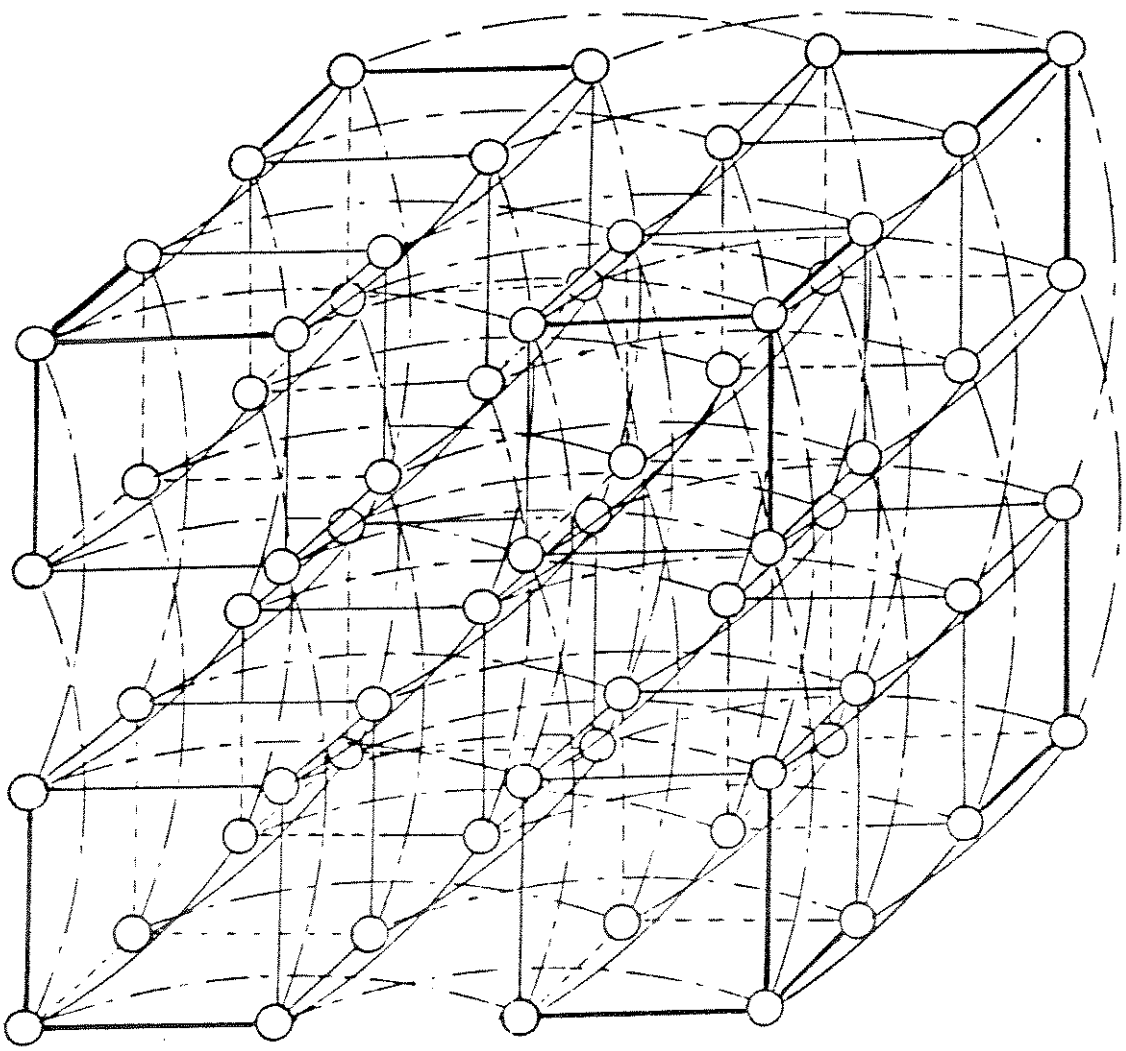


Fig. 3.6 – Hipercubo 6N com todas as ligações presentes

Conferindo as figuras, pode-se perceber com relativa facilidade em que endereço fica localizado cada processador. A seguir serão analisadas as propriedades e características desta arquitetura.

3.1.1 – Propriedades da arquitetura hipercubo

O hipercubo tem, entre outras, as seguintes propriedades:

i) Dois processadores vizinhos têm seus endereços diferenciados por apenas um bit.
Ex: O processador 010 é vizinho do 011 pois variou apenas um bit no lado direito. O processador 000 não é vizinho mais próximo do 011 pois varia de dois bits.

ii) Além de permitir identificar os vizinhos, a posição binária permite saber a distância que separa dois processadores. Portanto, o número de bits diferentes é o número de linhas que se necessita atravessar através da rota mais curta.

iii) A rota mais curta pode ser obtida de forma relativamente simples. Parte-se do endereço origem e modifica-se o mesmo um bit por vez até que vire o endereço destino.
Ex: Se o endereço origem for 0000 e o endereço destino for 0111, pode-se seguir a seguinte rota:
0000 - 0001 - 0011 - 0111 : são precisas 3 transmissões para se chegar ao destino.
0000 - 0100 - 0110 - 0111 seria outra alternativa.

iv) Procurar uma rota que não obedeça a essa regra significa fazer uma mudança desnecessária, exigindo correção posterior. Portanto, sair da rota mínima é aumentar a retransmissão em pelo menos 2 passos.

v) Para N bits diferentes entre endereços, tem-se N! caminhos mínimos.
Prova: Se para N bits diferentes, tem-se inicialmente N alternativas de bits a mudar, feita a primeira modificação, tem-se N−1 alternativas.
Em seguida tem-se N−2 alternativas.
Assim, para cada modificação vai-se diminuindo o número de alternativas até que acabe sobrando uma única, que é a final.
Assim, tem-se $N \times (N-1) \times (N-2) \times \cdots \times 1 = N!$ caminhos mínimos.

vi) O número de interfaces seriais cresce pouco em relação ao aumento de processadores ($log_2 N$).

vii) Tendo interfaces de reserva disponíveis, pode-se expandir com grande facilidade e de forma quase ilimitada.

viii) O fato do número de rotas alternativas crescer com a distância contitui-se numa clara vantagem tanto em termos de confiabilidade como em termos de velocidade.

ix) Por sua flexibilidade, é possível simular muitas outras arquiteturas de forma eficiente. Nas figuras 3.7 e 3.8, pode-se ver algumas das arquiteturas que o hipercubo pode simular [Pease 77, Seitz 85].

x) O sistema operacional neste tipo de arquitetura pode ter uma boa homogeneidade, precisando variar apenas o endereço que cada processador representa (que pode ser uma chave em "hardware").

xi) A simplicidade da rota permite que se trabalhe com um sistema operacional relativamente compacto.

xii) Em relação ao número de processadores envolvidos, o número de retransmissão é relativamente pequeno.

Estas propriedades, embora não resolvam todos os problemas do multiprocessamento, sem dúvida resolvem parte deles, principalmente na parte relacionada ao roteamento, que

costuma tornar-se complexa com o aumento de processadores, caso não se opte por uma técnica de anel ou barramento comum (que por sua vez tem suas desvantagens).

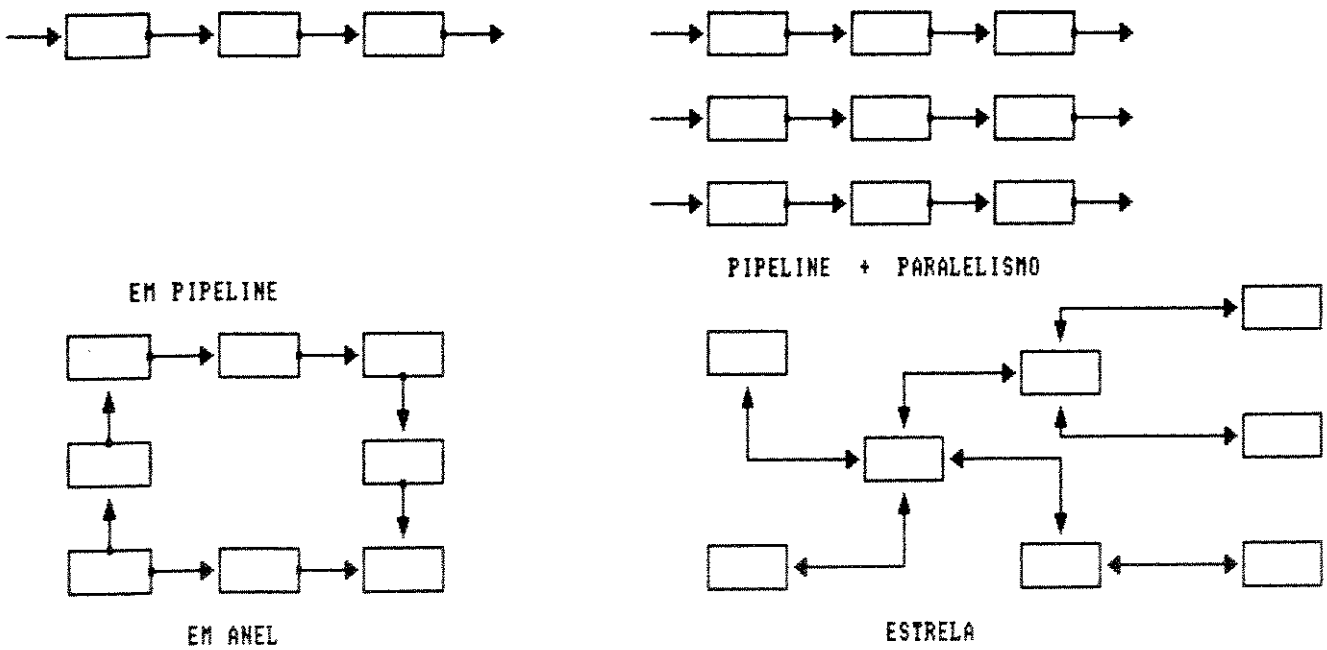


Fig. 3.7 – Alguns exemplos de simulações possíveis

Algumas desvantagens também podem ser detectadas. Entre elas, a possibilidade de congestionamento de certos caminhos num caso de comunicação muito intensa ou que envolva grande volume de dados. Uma outra é, sem dúvida, inerente à transmissão serial, ou seja, a velocidade de transmissão. A seguir comentam-se com mais detalhes estes problemas.

3.1.2 – Algumas questões a considerar

Como a transmissão paralela envolve grande número de fios, em estruturas com um grande número de processadores, ela se torna complexa e proibitiva. Por essa razão, muitos dos multiprocessadores usam a comunicação serial, que tem evoluído muito nos últimos anos, permitindo uma boa taxa de transmissão e confiabilidade tanto no "hardware" como no "software" [Gio 86].

Com a tecnologia de hoje pode-se atingir uma taxa de algumas dezenas de megabytes por segundo sem muita dificuldade. Isto significa que se pode enviar alguns bytes a cada microsegundo, o que para os microprocessadores de hoje é bastante aceitável, pois costuma-se gastar tempo para retirar/guardar o dado da interface, salvar/retirar da memória e verificar o "STATUS", no caso em que o processador não esteja equipado com "hardware" com ADM e detecção automático de erros.

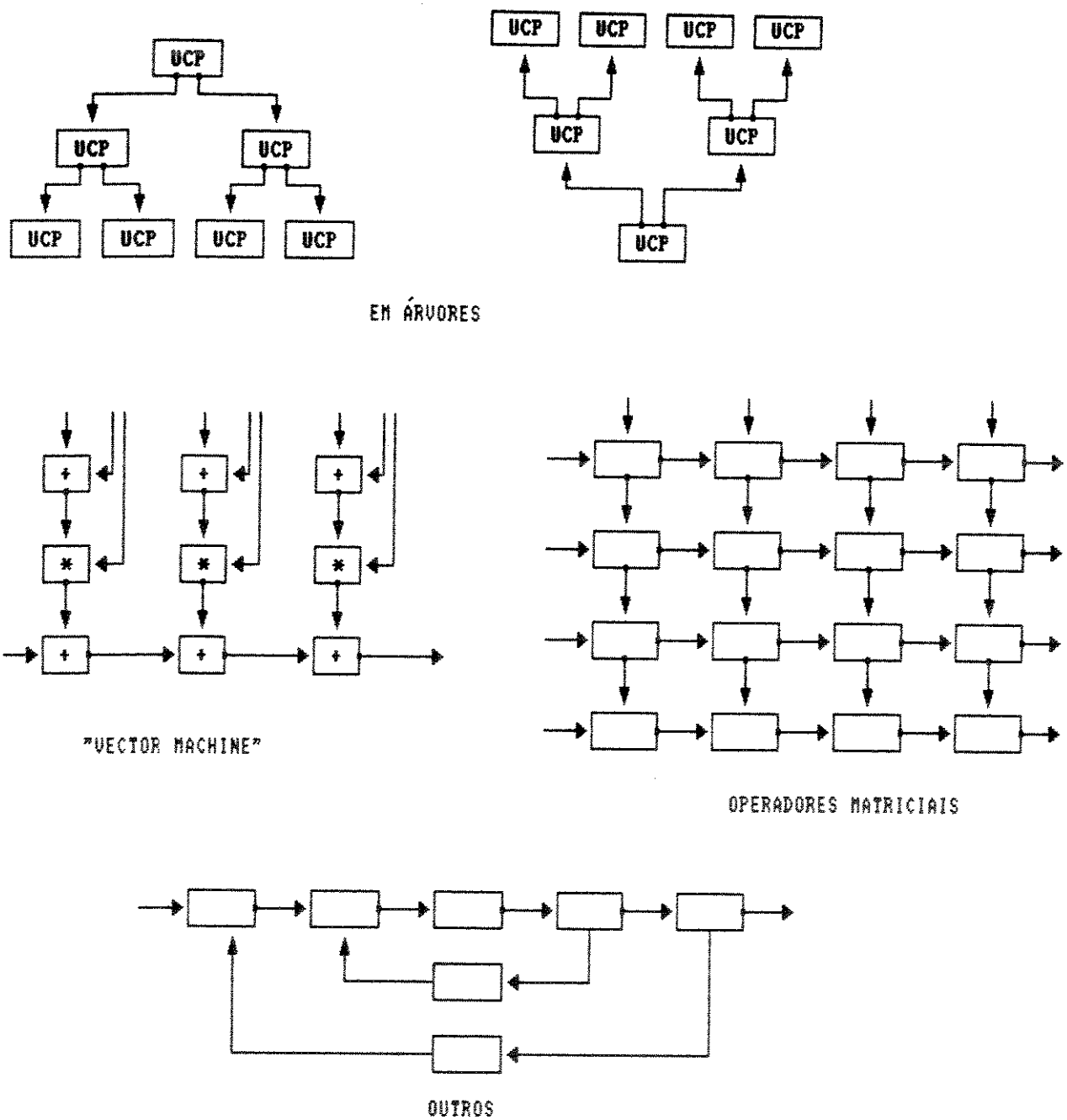


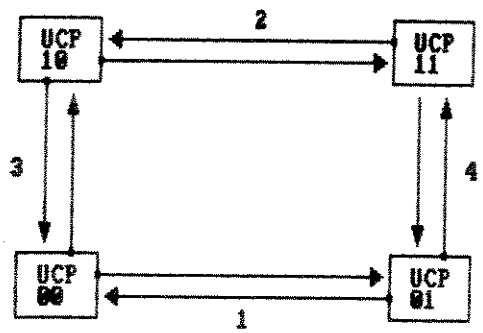
Fig. 3.8 – Mais exemplos de simulações possíveis

O estrangulamento de comunicação depende muito da aplicação que se faz mas, em muitos casos, pode-se minimizá-lo com soluções particulares de software.

Dentre as alternativas de minimização de estrangulamento pode-se citar :

i) Estabelecer uma regra para transmissão, de tal forma que as cargas sejam equilibradas em termos de probabilidade. Na figura 3.9 pode-se ver uma destas alternativas.
ii) Colocar um "software" sofisticado para minimizar os problemas. Existem padrões de comunicação já estabelecidos pelo ISO em níveis 3, 4 e 5 que podem resolver parte dos problemas como roteamento, compactação, etc.[Gioz 86].
iii) Estabelecer um outro caminho especializado em transmitir grandes "pacotes" de dados, deixando a fiação do hipercubo para dados de controles e pequenos "pacotes".

Esta última alternativa, sem dúvida, deve permitir melhor eficiência, e talvez seja a única alternativa para certas aplicações. Manter a fiação do hipercubo ainda se justifica, pois os canais de transmissão para grande volume de dados costumam enfrentar problemas de sincronismos e disputas entre os processadores, no que a fiação do hipercubo tem grande eficiência.




```
00 comunica com 11 pela rota 1
11 comunica com 00 pela rota 2
10 comunica com 01 pela rota 3
01 comunica com 10 pela rota 4
```

Fig. 3.9 – Uma regra alternativa para evitar estrangulamentos

3.2 Transputers

Transputer é o nome de um microprocessador pertencente a uma família de circuitos integrados VLSI projetados para multiprocessamento pela Inmos Ltd., Bristol, na Ingraterra, e que se tornou comercialmente utilizável em 1985. Embora o Transputer possua várias versões de UCPs, só serão emitidos comentários sobre o T800 [INM2-88, INM5-89], a versão mais popular no Brasil.

A UCP de um Transputer tem uma estrutura do tipo RISC (Reduced Instruction Set Computer) com 4Kbytes de RAM interna, além de "timer" e coprocessador. Porém o que mais caracteriza este microprocessador é o fato de possuir dentro do seu "chip" 4 entradas e saídas de linhas seriais de alta taxa de comunicação (20 Mbit/s), dotadas de controles capazes de fazer trabalhar cada linha de forma independente sem a intervenção do processador principal. Assim equipado, o Transputer pode se comunicar com outros Transputers vizinhos de forma bastante eficiente, bastando que, para isto, estejam ligadas fisicamente as entradas e saídas respectivas.

Por longos anos os projetistas estavam procurando uma arquitetura de processamento paralelo bastante flexível que permitisse uma ampla gama de aplicações e que, com a grande produção, diminuisse os custos. Pode-se dizer que de certa forma o Transputer alcançou este objetivo, embora não seja da forma que muitos esperavam. O transputer tem um custo relativamente reduzido em relação aos concorrentes e uma produção significativa, talvez numa quantidade nunca atingida na área de multiprocessamento.

Assim, pode-se construir, com Transputers, computadores de arquitetura variável. Esta flexibilidade topológica pode chegar ao nível dinâmico em tempo de execução; entretanto, é necessário que se desenvolvam programas de grande complexidade, controladores de topologia.

3.2.1 – Hardware

A família do Transputer constitui-se de vários tipos de UCP além de controladores, processadores gráficos, processadores de sinais, expansão de memórias, expansão de portas de entradas e saídas, chaves comutadoras de ligação, etc.

Entre as UCPs existentes podemos citar:

i) T212: Com processador de 16 bits, memória interna de 4Kbytes e 4 linhas de comunicação serial bidirecional[Nicole 89].
ii) T414: Com processador de 32 bits com dados e endereços multiplexados no tempo na parte externa[INM1 87, Nicole 89], memória interna de 2Kbytes e 4 linhas de comunicação serial bidirecional.
iii) T425: Com processador de 32 bits sem multiplexação, memória interna de 4Kbytes [Nicole 89] e 4 linhas de comunicação serial bidirecional.
iv) T800: Com processador de 32bits com dados e endereços multiplexados no tempo na parte externa[INM2 88, INM5 89], coprocessador de ponto flutuante de 64 bits, memória interna de 4Kbytes e 4 linhas de comunicação serial bidirecional.
v) T801: Com processador de 32 bits sem multiplexação, coprocessador de ponto flutuante de 64 bits[Nicole 89, INM5 89], memória interna de 4Kbytes e 4 linhas de comunicação serial bidirecional.
vi) H1 : Com processador vetorial de 32 bits[Nicole 89, Pou2 90], memória cache e 4 linhas de comunicação serial bidirecional.

A arquitetura do T800 é apresentada na figura 3.10. Sendo um RISC, o número de instruções desta UCP é pequena (103), assim como o número de registradores de trabalho comparado com um CISC (Complex Instruction Set Computer). As formas de endereçamento também são limitadas. Porém, o redirecionamento do esforço para alta velocidade permite que a maioria das instruções sejam executadas em poucos ciclos de relógio. Com isso a UCP com relógio de 20MHz trabalha com 10MIPS (Milhão de Instruções Por Segundo) com instruções de 32 bits, podendo endereçar até 4 gigabytes de memória. A multiplicação e divisão, porém, exigem 39 ciclos e o coprocessador de ponto flutuante que trabalha com 1,5 MFLOPS (Milhões de Operações em Ponto Flutuante por Segundo) vem aumentar a velocidade. Tudo isso num CI que consome menos de 0,5 Watts.

Segundo o fabricante, um Transputer tem o desempenho de aproximadamente 100 vezes o IBM-PC, 4,18 vezes o VAX 11/780/FPA e com 7 Transputers pode-se competir

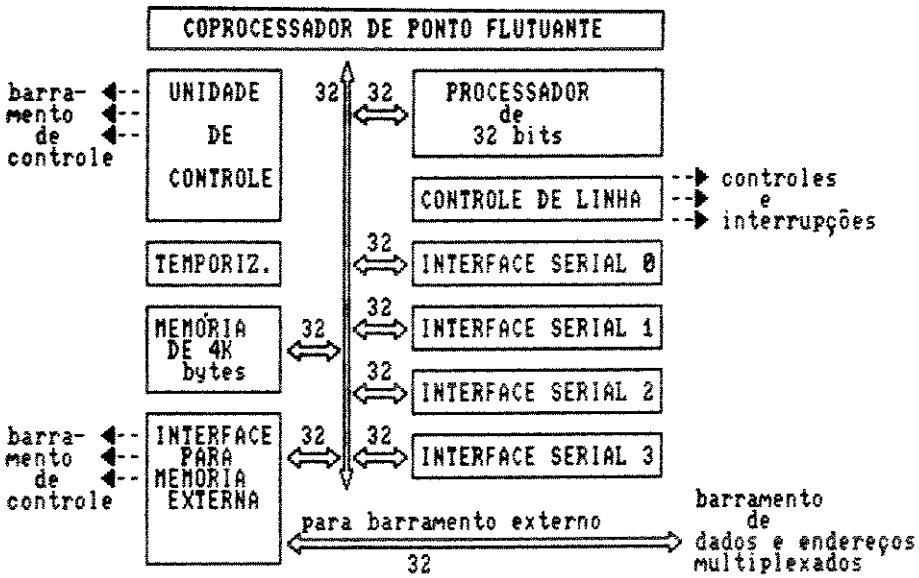


Fig. 3.10 – Diagrama de bloco da UCP T800

com CRAY-1S, mas existem no mercado Transputers que trabalham com 30Mhz, tendo portanto maior desempenho e, trabalhando a 70 graus Kelvin, já atingiram freqüências acima de 90 MHz.

As 4 entradas e as 4 saídas seriais podem transmitir com a taxa de 20 Mbit/s e, sem nenhuma linha auxiliar, podem trabalhar de forma síncrona. Para poder fazer isso, a INMOS adotou uma solução original: Enquanto que nas linhas seriais costumam ter 1 "start bit" para avisar o início de transmissão, no transputer tem-se 2 "start bits" podendo diferenciar a resposta de sincronismo com transmissão de dados como se pode ver na figura 3.11.

Esta forma de sincronismo simplificou muito a ligação dos fios entre pinos, de tal forma que, para se ligar um transputer no outro, basta ligar dois fios entre eles (e mais a terra em comum). Porém, a distância máxima permitida fica limitada a 30 cm caso não tenha repetidor (as chaves comutadoras possuem).

O circuito temporizador interno permite que o usuário não só faça medidas de tempo como permite que programe tempos de espera para processos de maior e menor prioridade de forma autônoma. Estes tempos são medidos em "ticks" que têm duração de .25 ms para processo de maior prioridade e 16 ms para processos de menor prioridade.

O mercado internacional já oferece grande quantidade de placas que usam Transputers

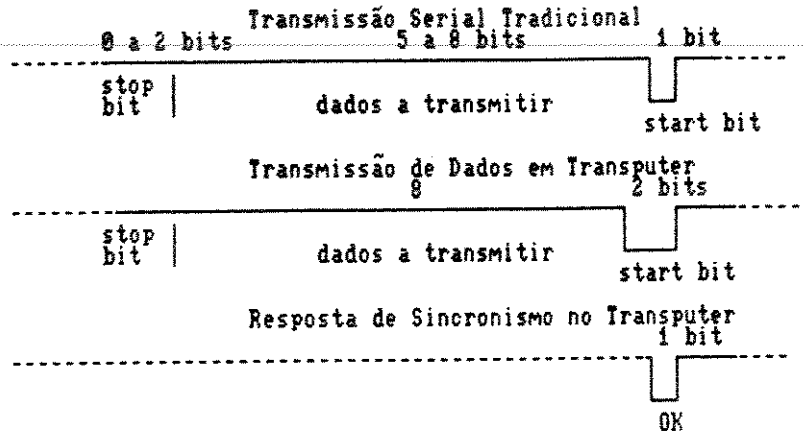


Fig. 3.11 – Forma de transmissão

e que podem ser classificadas em dois tipos; o primeiro é para sistemas projetados para trabalharem com ambientes de Transputers, e o segundo para serem conectados em barramentos de microcomputadores ou computadores de uso popular, trabalhando como hospedeiros. Hoje existem placas para serem conectadas com numerosos sistemas, podendo-se citar: IBM-PC, SUN-3, Mac-II, sistemas com barramentos VME e Computadores VAX.

O número de UCPs existentes nas placas variam de 1 a 17 UCPs, e a memória por UCP varia de algumas dezenas de Kbytes até 32 Mbytes.

3.2.2 – Software

Para computadores cujo processador principal é um Transputer, já está disponível um sistema operacional multi-usuário e multi-processamento UNIX chamado UNIX CORUS [Jose 85, Michel 89].

Para os usuários das placas com Transputers conectadas ao computador hospedeiro, o acesso aos periféricos e arquivos do hospedeiro, em geral é feito usando o próprio sistema operacional disponível. É o caso dos usuários das placas conectadas aos computadores tipo IBM-PC, que trabalham fazendo uso do sistema operacional MS-DOS. Porém, certas linguagens podem fazer uso de um ambiente integrado de software, incluindo editor, compilador, depurador, etc., de tal forma que quase não se percebe a interferência do hospedeiro.

Quanto às linguagens disponíveis, pode-se separá-las em dois grupos distintos:

No primeiro grupo pode-se classificar as linguagens seqüenciais tradicionais, acrescidas de recursos para processamentos paralelos e concorrentes. Tais recursos apresentam mecanismos bastante parecidos entre as linguagens , de tal forma que estudando uma delas, já se pode ter noção de como funcionam as outras linguagens desde que já se conheçam linguagens seqüenciais tradicionais. Pode-se classificar entre estas linguagens o C paralelo, o Fortran paralelo e o Pascal paralelo da empresa 3L. Existe ainda o Prolog seqüencial.

Todos fazem o uso de "threads", variáveis comuns, canais, semáforos e configuradores. Até as etapas de compilação têm semelhança. São os seguintes os nomes dos programas compiladores e configuradores:

C paralelo	Fortran Paralelo	Pascal
T8c	T8f	T8P
T8clink	T8flink	T8plink
T8ctask	.	.
T8cstask	.	.
Config	Config	Config
Afserver	Afserver	Afserver

No segundo grupo, classificam-se as linguagens projetadas especialmente para o paralelismo como é o caso da OCCAM.

A Inmos, fabricante do Transputer, está lançando a ADA, Parlog (Prolog paralelo) e está trabalhando uma nova linguagem chamado Haskell.

Quando se trata de linguagens paralelas, tem-se uma grande gama de alternativas de filosofia e implementação. Em alguns casos esta opção tem sido utilizada por ser mais promissora e em outros casos apenas por ser mais fácil implementar, e não é raro escolher opções a título de experiência, para verificação da aceitação, para melhoramentos e repotencializações posteriores. Pode-se citar, por exemplo, a forma de configurar (distribuir o programa através da rede). A decisão de como carregar e executar pode ser tomada em instantes diferentes, podendo ser na hora da compilação, na hora da carga (carregamento do programa através da rede), ou na hora da execução; sendo que a cada uma associam-se vantagens e desvantagens. Ao definir a configuração na hora da compilação, o usuário é obrigado a fornecer a topologia exata da rede antes da compilação e, além disso, tem a desvantagem de ser menos flexível entre as três opções. Tem-se, porém, a vantagem de ter um bom desempenho e ser fácil de implementar. A reconfiguração durante a execução permite grande flexibilidade ao programa, mas tem a desvantagem de penalizar em muito o desempenho. A reconfiguração durante a carga fica no meio termo entre os dois, em desempenho e flexibilidade.

O Transputer tem utilizado, preferencialmente, a técnica de definir a distribuição de processos na hora da configuração(etapa final de compilação), provavelmente por ser mais fácil de ser implementada. Existe exceção, que é a utilização da técnica "farm" (fazenda) que define a configuração no momento da carga. Utilizando esta técnica o usuário não precisa fornecer a topologia da rede, mas a aplicação é limitada somente para casos em que se utiliza um único mestre sendo todos os outros processos, os "workers" (trabalhadores), controlados diretamente pelo mestre. Deve ser definida também a forma de comunicação entre os processos.

Com relação ao protocolo de comunicação, o Transputer utiliza o processo seqüencial comunicante por ser bastante compatível com a comunicação por linhas seriais. Assim, os processos trocam informações entre si através de "canais" que são linhas seriais físicas, quando se trata de comunicação entre processos com UCPs diferentes, e um "buffer" de memória quando os processos estão numa única UCP. Este tipo de comunicação favorece a sincronização e confiabilidade, mas tem suas desvantagens como, por exemplo, o processo receptor ficar esperando a chegada de mensagens ou o transmissor esperar pela

disponibilidade do processo receptor fazendo com que, em alguns casos, perca-se tempo desnecessariamente.

Existe outra forma dos processos trocarem informacões entre si com mais eficiência quando estão na mesma UCP. Esta alternativa é a dos processos possuírem variáveis comuns entre eles. Assim, um processo pode analisar o estado do outro processo no instante em que desejar, permitindo maior liberdade. Isso, porém, envolve riscos. Vamos ver em seguida um tipo de erro muito comum neste caso:

Hipótese: Existem dois processos P1 e P2 e uma variável comum x.

Se os dois processos forem executados concorrentemente podem ocorrer os seguintes casos:

1) P1 lê valor de x
   P1 incrementa x
   .
   .
   .
   A UCP passa a executar P2
   P2 lê valor de x
   P2 incrementa x

   Neste casos, o x é incrementado duas vezes e o resultado é x+2.

2) P1 lê valor de x
   A UCP passa a executar P2
   P2 lê o valor de x
   P2 incrementa x
   .
   .
   .
   A UCP passa a executar P1
   P1 incrementa x

   Embora os dois processos tenham incrementado o valor do x, o resultado final é de x+1

O erro ocorreu porque a UCP comutou de processo na hora errada. Essa comutação ocorre sincronizada pelo temporizador e o usuário não tem controle sobre ele. Na verdade nunca se tem certeza de qual processo está na frente se não for sincronizado naquele instante. Basta um dos processos acessar um arquivo no disco, por exemplo, para defasar completamente o processo. Para defender-se deste tipo de erro, existe o conceito de semáforo. A utilização do semáforo, é bastante simples quando se tem apenas alguns dados em comum com poucos processos. Conforme aumenta o número de processos e de variáveis comuns o problema de administrar semáforos torna-se, em certas aplicações, um problema complexo, sendo que um erro no semáforo pode gerar problema sério de confiabilidade ou de tempo, podendo inclusive entrar em "deadlock" com facilidade.

O uso de variáveis comuns, por outro lado, tem a dificuldade da adminstração da concorrência ao acesso das variáveis. A linguagem OCCAM optou por permitir que, de dois ou mais processos concorrentes, somente um deles possa modificar o valor de uma variável

comum e os outros poderão somente ler o seu valor (o compilador acusa erro). No caso da linguagem C paralelo, o compilador simplesmente deixa, por conta do programador, o risco e a responsabilidade de cuidar dos erros que variáveis comuns possam gerar (o compilador nada acusa, mas podem ocorrer erros) e, neste caso, deve-se notar que se um programa foi executado uma vez sem problemas não quer dizer que não terá problemas na próxima vez, pois, a defasagem dos processos depende simplesmente do estado do temporizador, posição da cabeça do controlador de disco, tempo de resposta do usuário no teclado, etc.

A seguir, a linguagem C paralelo e a linguagem OCCAM são resumidamente analisadas.

O C paralelo

O compilador C que foi utilizado na implementação do hipercubo foi o C paralelo versão 2.0 da companhia 3L Ltda[3L 88]. Na parte seqüencial, esta linguagem difere muito pouco das linguagens C tradicionalmente utilizadas pelo MS-DOS. O próprio manual recomenda fazer o uso do livro C Programming Language de Kerninghan e Ritchie, pois obedece de forma bastante fiel a este livro. Os testes feitos vieram confirmar esta compatibilidade.

A biblioteca de subrotinas adicionais vem repotencializar a liguagem, sendo que muitas delas já são conhecidas como ascii.h, assert.h, ctype.h, dos.h, math.h, stdio.h, stdlib.h, string.h e time.h. Para dotar esta linguagem de recursos paralelos, foi preciso incluir nova biblioteca destinada ao paralelismo, podendo-se citar: chan.h e chanio.h que permitem o uso de canais de comunicação entre processos, net.h para implementar técnica "farm", sema.h para implementar semáforos, timer.h que manipula "timer" interno do Transputer, boot.h que permite ler ou gravar dados na memória dos Transputers vizinhos, thread.h que permite implementar e controlar processos paralelos a partir do processo principal. Acrescido destas bibliotecas e mais o programa configurador que permite distribuir os processos através da rede de Transputers, o C paralelo permite que os usuários da linguagem C possam explorar todo o conhecimento que já possuem na programação seqüencial, exigindo apenas que dominem os recursos e problemas adicionais que surgem com o paralelismo.

A comunicação entre os processos baseia-se em dois dos recursos tradicionais. O primeiro é o uso de variáveis comuns. Dois processos em paralelo possuem uma única variável comum ocupando o mesmo endereço da memória. Este recurso, porém, não pode ser utilizado quando os dois processos estão em Transputers diferentes, a não ser que tenham memória em comum. Para este caso, faz-se uso da técnica "processo sequencial comunicante [Hoare 78, Hoare 85, Kram 87]. Esta técnica se encaixa perfeitamente com o Trasnsputer que tem toda a estrutura baseada em linhas físicas seriais[INM1 85, INM2 88, INM4 89]. Estas "linhas" de comunicação recebem o nome de canais e, além de transferir dados, cumprem também o papel de sincronizador dos processos.

O programa configurador permite distribuir os processos através da rede. Como foi citado no capítulo anterior, o C paralelo usa preferencialmente o momento de compilação para definir a distribuição dos programas através da rede. Isto significa também que o usuário deverá conhecer a topologia da rede e definir a distribuição processo por processo, ligação por ligação (links) tal como será utilizado. Um erro nesta declaração pode acarretar resultados inesperados, além de que o programa não poderá ser utilizado em outra topologia

a não ser que modifique de novo as informações. Existe apenas um caso em que a topologia não precisa ser citada. É o caso em que se faz o uso da técnica "farm" [Poul 90, 3L 88](fazenda). Porém, esta técnica só pode ser utilizada quando se tem um único mestre e os outros são "workers", que são processos gêmeos controlados diretamente pelo mestre. Um rápido teste desta técnica, porém, mostrou ser ela pouco eficiente em desempenho, pois sacrifica parte do potencial de processamento para permitir uma certa "inteligência", além de ocupar mais memória.

Uma outra característica importante do configurador é permitir simular a existência de vários Transputers numa única UCP, pois aceita processos concorrentes.

Para que se possa explorar variáveis comuns, deve-se fazer o uso de "threads" que, no fundo, não são mais do que subrotinas que podem rodar em paralelo com o programa principal.

No capítulo anterior foram vistos alguns dos problemas que as variáveis comuns podem trazer, principalmente no C paralelo, no qual o compilador não detecta nenhum tipo de erro de variáveis comuns, deixando o cuidado e a responsabilidade inteiramente por conta do usuário.

O uso de variáveis comuns tem, porém, certas vantagens indiscutíveis em relação a canais. Uma delas é que se pode conhecer o estado do outros processos sem interferir ou ser interferido por estes. O segundo é que não se perde tempo nesta averiguação nem do processo que analisa, nem do processo analisado.

Se, em certas circunstâncias, o sincronismo e a comunicação são essenciais para o bom andamento do processo, existem casos em que o uso dos canais representa atrasos e queda de desempenho ou, mesmo, o aumento de complexidade do programa.

Para que se possa controlar os acessos nos momentos indesejáveis em variáveis comuns, foram criados os semáforos [Ben 82, 3L 88]. Se um dos processos aciona o semáforo antes de acessar o dado, o outro processo é obrigado a esperar o acesso à variável ou variáveis até que o primeiro termine a tarefa e libere o semáforo. O semáforo pode também ser utilizado como uma ferramenta de sincronismo dos processos.

A linguagem OCCAM

OCCAM é uma nova linguagem paralela desenvolvida para Transputer e o seu nome provém de um filósofo do século XIV, William Occam[Burns 88, INM3 88, INM5 89]. É uma linguagem baseada em Processo Sequencial Comunicante [Hoare 78, Hoare 88] que busca implementar o paralelismo da forma mais simples e amigável possível, procurando representar o paralelismo diretamente no programa.

A configuração é definida no momento da compilação e, portanto, o programador deve fornecer a topologia exata com antecedência.

A linguagem OCCAM utiliza preferencialmente a comunicação via canais, mas pode-se fazer o uso de variáveis comuns quando os processos são executados num mesmo processador. Porém, como foi colocado anteriormente, o compilador permite somente a um processo escrever em uma variável comum, e isto inclui também comandos como if x > y+1 onde y é variável comum.

Para apoiar o programador, existe um ambiente integrado de software chamado TDS (Transputer Development System)[INM6 89, INM7 88] equipado com editor, compilador,

depurador, etc., que permite trabalhar dentro dele sem depender diretamente dos comandos do sistema operacional hospedeiro. É organizado de forma bastante prática, permitindo que o usuário possa trabalhar com desenvoltura.

A linguagem OCCAM sem o TDS mostrou-se bastante pobre de bibliotecas e, embora possa usar processos concorrentes, trabalha somente com monoprocessamento. Além disso, precisa de vários passos de compilação mesmo para programas mais simples. A falta de bibliotecas resulta em programas longos e ineficientes e, por esta razão, o trabalho de implementação do hipercubo (Capítulo 4) foi baseado no sistema com TDS. Daqui para frente, quando se falar em OCCAM fica implícito que se trata da linguagem equipada com o TDS.

Em relação ao compilador C paralelo o compilador da linguagem OCCAM mostrou-se muito mais amigável e, além de detectar muito mais erros na compilação, é dotado de algumas mensagens de erro na execução. Além disso, a presença do depurador vem ajudar em muito o programador.

Por outro lado, o fato de trabalhar em sistema integrado exige que o usuário aprenda a fazer uso não só da linguagem, mas também do editor, depurador e todos os recursos do sistema como mudanças em arquivos, impressão, etc. Para o usuário familializar-se com o sistema, é necessário um período de aprendizado mais longo do que aprender somente a linguagem.

A linguagem também tem aspecto bastante diferente das linguagens tradicionais, o que exige um período de adaptação mesmo para programas seqüenciais.

Em seguida é comentada a organização da linguagem OCCAM.

O que mais diferencia a OCCAM das linguagens sequenciais é a presença dos "constructors" SEQ e PAR. Em outras palavras, no lugar de begin do pascal ou "{" do C existe o SEQ ou o PAR. Acionando-se o SEQ, os processos que vêm a seguir são processados de forma seqüencial. Se for o PAR Constructor acionado, os processos serão executados em paralelo, ou seja, concorrentemente.

Assim, se o programa for:

```
SEQ
  processo1
  processo2
  .
  .
  .
```

Primeiro será executado o processo1 e logo ao seu final será executado o processo2.

Um trecho de programa usando PAR será algo como:

```
PAR
  processo1
  processo2
```

Neste caso, o processo1 e o processo2 serão executados concorrentemente.

Tem-se também os "constructors" condicionais como o ALT, IF, WHILE. Sendo condicionais, este comandos devem vir acompanhados de condições. Existe, também, o "repli-

cadore" que utilizado junto com "constructor" pode gerar repetições de processos. Logo a seguir é explicado cada um destes constructors.

A identação, deslocamento da coluna, faz papel muito importante na OCCAM. É com ela que o compilador determina se um processo pertence a um "constructor" de um SEQ ou não. Esta forma de organizar o programa acaba formando uma estrutura bastante diferente da linguagem tradicional.

Embora a OCCAM manipule os IF, FOR, CASE e WHILE eles apresentam um conteúdo semântico diferente, o que vem dificultar a adaptação de um programador familiarizado com linguagens seqüenciais.

Eis alguns exemplos:

O conceito do IF:

```
IF
  condição1
    processo1
  condição2
    processo2
  TRUE
    SKIP
```

Neste caso, se as condições do IF forem satisfeitas, o programa executa o processo1. Se não forem satisfeitas as condições o programa continua normalmente sem executar o processo1 ou o processo2 . Porém se não existissem o TRUE e o SKIP e, nenhuma das condições fossem satisfeitas, seria gerada uma interrupção e o programa seria abortado.

O conceito do FOR:

```
SEQ x=5 FOR 7
  processo1
```

Aqui se tem um exemplo de replicador FOR. Deve-se alertar que o 7 é o número de vezes que o processo deve ser repetido e não o valor para o qual x deve parar. Assim o processo1 deve repetir para x=5 até x=11 inclusive, sendo incrementado de 1.

O conceito de ALT:

```
ALT
  condição1
    processo1
  condição2
    processo2
```

Aqui se tem o comando ALT que representa alternativa. A primeira condição que for satisfeita fará executar o respectivo processo. O outro processo será ignorado a não ser que o processo esteja em "loop".

Existem ainda o PLACED PAR e o PRI PAR. Enquanto o PAR ativa vários processos num único processador o PLACED PAR permite colocar os processos paralelos em determinados processadores. O PRI PAR permite ter processos com prioridades diferentes.

Infelizmente o Transputer possui só duas prioridades. Além disso a linguagem permite somente um processo de maior prioridade.

O PLACED PAR permite que os processos pré compilados possam ser carregados em determinados processadores e executados. Na verdade existem diversas formas de carregar programas no multiprocessador. A técnica utilizada fez uso do PROGRAM que carregava os processadores não "root" e, posteriormente, fazia a carga do .EXE, que era o programa do "root". Ao ordenar a execução do "root", este acionava os outros processos que esperavam a chegada de mensagens via canais.

As entradas e saídas dos canais são representados respectivamente por ? e !. Assim um comando "canal1 ! x" significa transmissão do conteúdo da variável x pelo canal de saída chamado canal1.

Capítulo 4

IMPLEMENTAÇÃO

A redistribuição dinâmica de trabalho entre processadores concorrentes é entendida como a tarefa de se redistribuir o trabalho entre os processadores durante o processamento, visando uma melhor eficiência computacional. Esta é uma tarefa que depende do modelamento escolhido para o problema a ser resolvido e da estrutura de interconexão dos processadores.

Sem dúvida, o ideal para manter uma eficiência computacional alta seria uma estrutura que permitisse a solicitação e o envio de dados de um processador a outro com um baixo custo e rápida comunicação. Isso exige da estrutura que a solicitação de tarefa seja endereçada a todos os processadores e que os dados sejam enviados através do caminho mais curto entre dois processadores. Por outro lado, o modelamento deve proporcionar que o número de dados relativos à tarefa seja minimizado para diminuir o comprimento da mensagem de envio de tarefas de um processador para outro e para que o comprimento das mensagens de solicitação de tarefa seja também minimizado.

Um outro aspecto a ser considerado na redistribuição dinâmica de trabalho é a administração da ocupação dos processadores, ou seja, como saber quais processadores estão em via de terminarem seu trabalho e quais são aqueles que ainda têm trabalho suficiente para dividir com outros.

Considerando que a administração centralizada de trabalho acarreta um tráfego adicional de mensagens na estrutura de interconexão dos processadores, propoẽ-se uma administração distribuída. Cada processador toma conta do seu nível de ocupação e, quando detecta a proximidade do final de seu trabalho, envia uma mensagem aos outros processadores solicitando que lhe sejam enviadas novas tarefas.

É a filosofia da redistribuição de trabalho que tem base na premissa de que "quem acabar primeiro ajuda os outros".

4.1 O Hipercubo e a Redistribuição de Trabalho

A interconexão de processadores usando a estrutura de hipercubica tem propriedades que facilitam o envio de mensagem de um processador a todos os outros ("broadcast") e a determinação do caminho mínimo entre dois processadores.

O envio de mensagem do tipo "broadcast" pode ser implementado de pelo menos duas maneiras no hipercubo. A primeira, usando a transmissão do tipo Gray[Lang 81] conforme mostra a fig.4.1 e, a outra, através da transmissão tipo árvore, mostrada na fig.4.2.

A determinação do caminho mínimo a ser percorrido por uma mensagem requer somente que a mensagem possua no seu cabeçalho o código do vértice do hipercubo ao qual ela se destina. Isto é suficiente porque, a cada retransmissão da mensagem, o processador

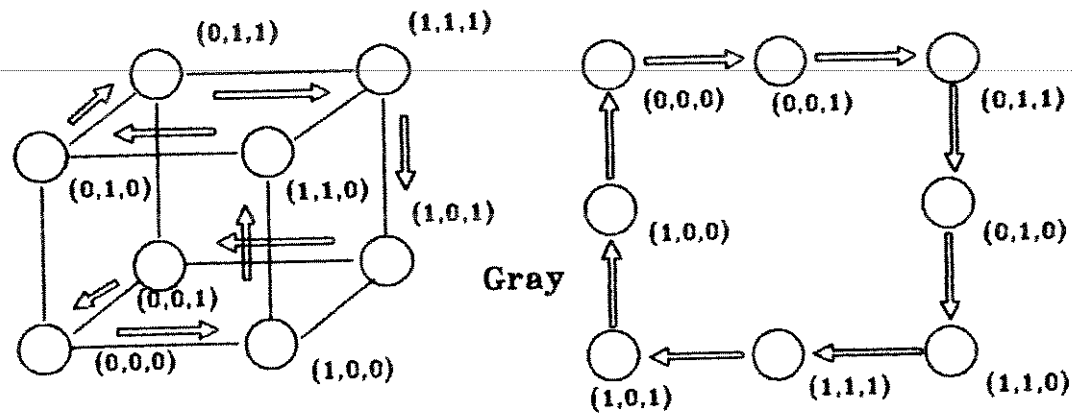


Fig. 4.1 – Transmissão de mensagem usando o código Gray

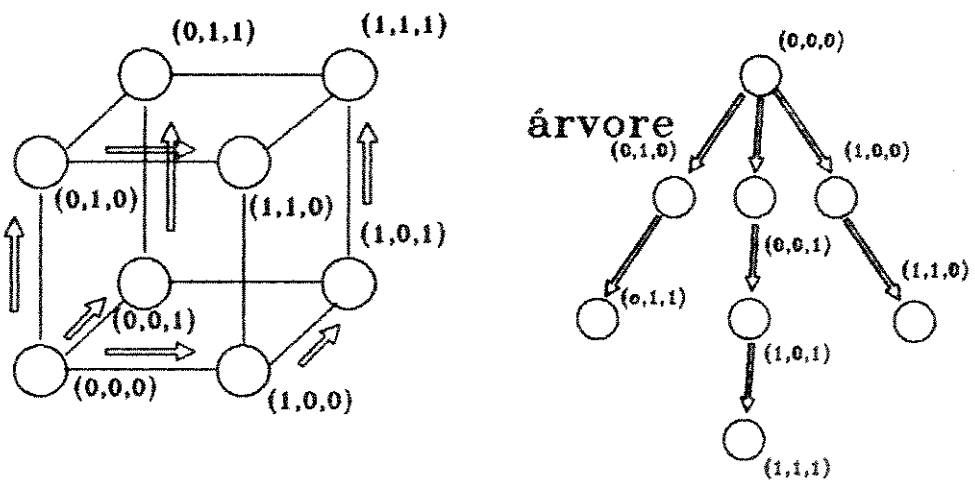


Fig. 4.2 – Transmissão de mensagem usando árvore

que retransmite identifica quais dos processadores vizinhos a ele têm distâncias "Hamming" menores com relação ao processador destinatário para, então, escolher um deles e retransmitir a mensagem. Este procedimento leva necessariamente ao caminho mínimo, medido em número de retransmissões da mensagem. A fig.4.3 apresenta um caminho mínimo percorrido por uma mensagem sobre um hipercubo de ordem 3.

Assim, para que se implemente sobre um hipercubo uma administração distribuída de trabalho, basta que se implementem os recursos para envio de mensagens, em árvore, código Gray e pelo caminho mínimo. Por sua vez, as mensagens devem conter o código do destinatário, o código do remetente, um identificador do tipo de mensagem e o tamanho da mensagem em bytes.

Com os recursos acima, os processadores poderão ser carregados, inicialmente, por transmissões do tipo árvore, pedir tarefas com transmissões em Gray e receber ou enviar

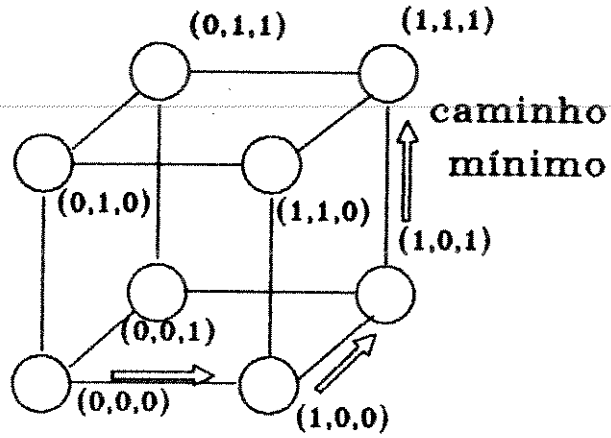


Fig. 4.3 – Transmissão sobre um caminho mínimo

tarefas por transmissões de caminho mínimo.

A transmissão em árvore otimiza a distribuição inicial de trabalho, já que a retransmissão vai se expandindo à proporção em que a árvore vai sendo percorrida. Assim, tem-se um tempo bem menor do que se o carregamento fosse feito em código Gray, por exemplo.

A transmissão em código Gray garante que a mensagem enviada possa percorrer todos os processadores e voltar ao ponto de partida. Ela é ideal, então, para solicitação de trabalho pois, se ao longo do caminho encontrar algum processador que possa enviar tarefas ao processador solicitante, a sua retransmissão pode ser interrompida e a solicitação atendida. Caso contrário, se não encontrar um processador que possa atender à solicitação, volta ao solicitante, que interpreta o evento como não existindo nenhum processador na estrutura que possa lhe enviar mais tarefas e, portanto, pode encerrar o seu trabalho.

Finalmente, a transmissão por caminho mínimo deve ser usada para garantir uma maior velocidade de transmissão de mensagens de trabalho entre dois processadores.

O término do trabalho total deve ser supervisionado pelo processador raiz ("root") que, no início, envia uma mensagem em transmissão Gray que só é retransmitida por um processador quando ele acaba de realizar o seu trabalho e não existe um outro processador que lhe possa enviar mais tarefa. Quando o processador raiz receber de volta esta mensagem, significa que o trabalho total está terminado e o resultado final é enviado para o terminal de vídeo.

Identificam-se, na descrição acima, alguns tipos de mensagens, por exemplo: carga inicial de trabalho, solicitação de tarefa, envio de tarefa e mensagem de controle do término total. Outros tipos de mensagens têm importância na supervisão do processamento total; pode-se citar as mensagens de erro e o retorno de mensagem por pane numa rota do hipercubo.

Nas aplicações em que, por exemplo, se procura encontrar o valor mínimo de uma certa variável, pode-se fazer com que fique circulando sobre o hipercubo uma mensagem que contenha o valor mínimo atual da variável. Portanto, poder-se-ia criar um código particular para este tipo de mensagem. Isto é somente um exemplo, entre muitos, que se

pode formular, mostrando que a atribuição dos códigos das mensagens depende também da aplicação e não somente da estrutura do hipercubo.

Na fig.4.4. é apresentado um diagrama de estados que resume a evolução do processamento total, vista de um dos processadores.

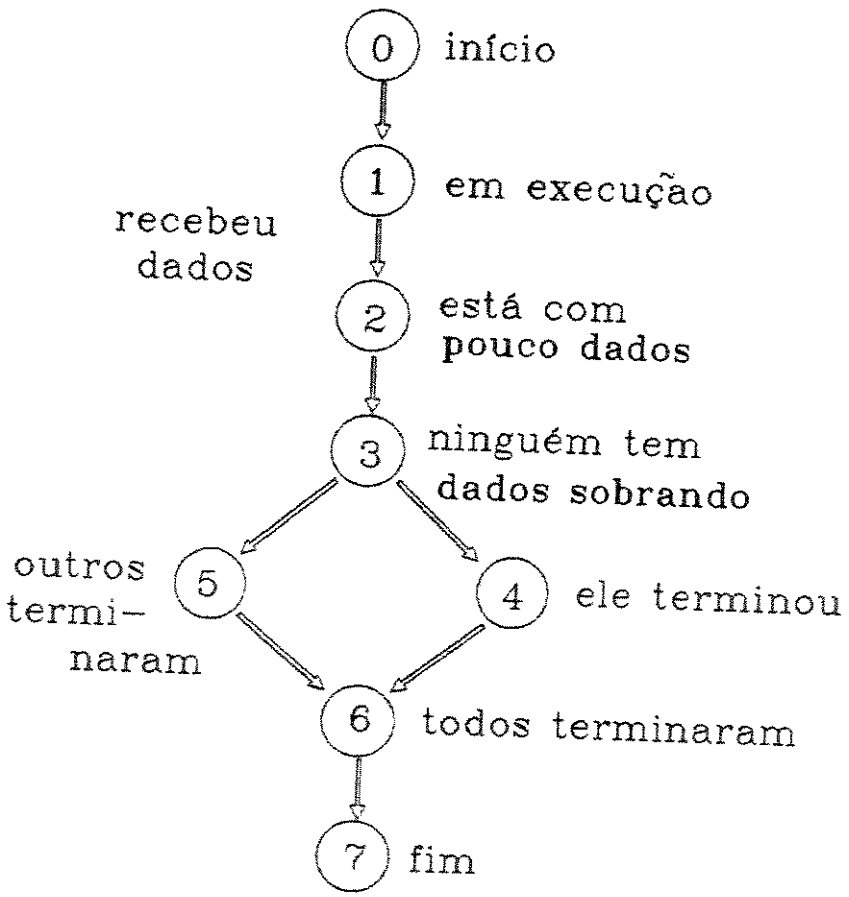


Fig. 4.4 – Diagrama de estado resumo da evolução do processamento total.

4.2. O particionamento de trabalho e a redistribuição dinâmica

O particionamento de trabalho foi abordado no Capítulo 2, onde se examinaram propriedades de autômatos que modelam implementações de soluções de problemas. Estes autômatos tinham grafos associados que permitiam ser particionados em sub-grafos disjuntos, o que garantia uma distribuição de trabalho de tal forma que nenhuma tarefa fosse executada mais de uma vez. Além disso, no caso dos autômatos autônomos, a mensagem de envio de trabalho fica resumida à identificação do estado inicial que o sub-autômato deve assumir para desenvolver o trabalho.

Como se viu no item anterior, a estrutura de interconexão de hipercubo é bastante adequada para hospedar um sistema, capaz de administrar de forma distribuída um trabalho de processadores concorrentes.

Finalmente, propõe-se que, quando da implementação da solução de um problema utilizando-se processadores concorrentes interligados numa estrutura hipercubica sejam inicialmente analisados os seguintes aspectos:

i) Se o problema permite um modelamento que facilite a distribuição inicial de trabalho via partição em trabalhos menores.
ii) Se o modelamento escolhido permite ou não uma administração distribuída.

Estes dois aspectos são vitais para o sucesso na implementação de soluções de problemas usando o hipercubo.

Os capítulos seguintes tratam dos estudos feitos para a implemantação de hipercubo utilizando recursos de máquinas existentes no Departamento de Computação e Automação Industrial da Faculdade de Engenharia Elétrica da Unicamp, bem como de implementações de programas em ambientes com "Transputers" com recursos dos laboratórios do Programa de Engenharia de Sistemas de Computação da Universidade Federal do Rio de Janeiro e Laboratório de Instrumentação Eletrônica do Instituto de Física e Química da USP, campus São Carlos.

4.3 Usando Transputers

Equipado com linhas de comunicação de alta velocidade, o Transputer é bastante adequado à arquitetura hipercúbica. A primeira dúvida que surge, porém, é o número de linhas seriais disponíveis. Se cada UCP estiver equipado somente com 4 linhas seriais, como montar um hipercubo com dimensão acima de 4? Quando uma das linhas se comunica com o "Host", a situação fica pior ainda, pois dispõe-se apenas de 3 linhas, o que permite montar uma estrutura cúbica até dimensão 3 com 8 processadores. Esta restrição, porém, só é válida se se consider que um nó só pode ser montado com um único processador. No entanto, se o programa de cada nó for divisível em mais de uma UCP, pode-se contar com muito mais linhas de saída por nó, como pode ser visto na figura 4.5, onde dois processadores dentro de um único nó permitem haver 6 linhas bidirecionais proporcionando 64 nós envolvendo 128 processadores. Na figura 4.6, onde o nó envolve 4 processadores, permite-se ter 8 linhas bidirecionais implementadas num hipercubo de dimensão 8 com 256 nós e 1024 processadores.

Desta forma, conforme é aumentado o número de UCPs por nó, pode-se aumentar a dimensão do hipercubo.

Fazendo-se uso do mais recente Transputer H1 [Nicole 89, Pou2 90], o problema do número de linhas de entrada e saída já está contornado através do uso de canais virtuais e do circuito integrado comutador de linhas C104 e novos comutadores que estão sendo pesquisados [Pou2 90, May 90]. O canal virtual possibilita a comunicação com numerosos processadores embora use somente 4 canais reais, permitindo montar um hipercubo com 65K processadores, ainda que, em certos casos, a multiplexação deixe a comunicação bem mais lenta.

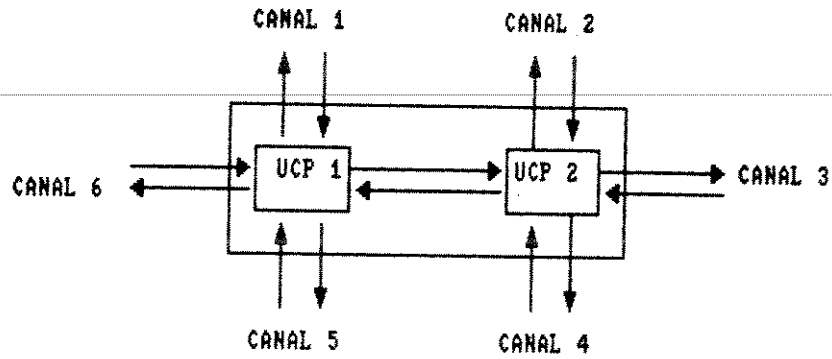


Fig. 4.5 – Colocando-se dois processadores dentro do nó,

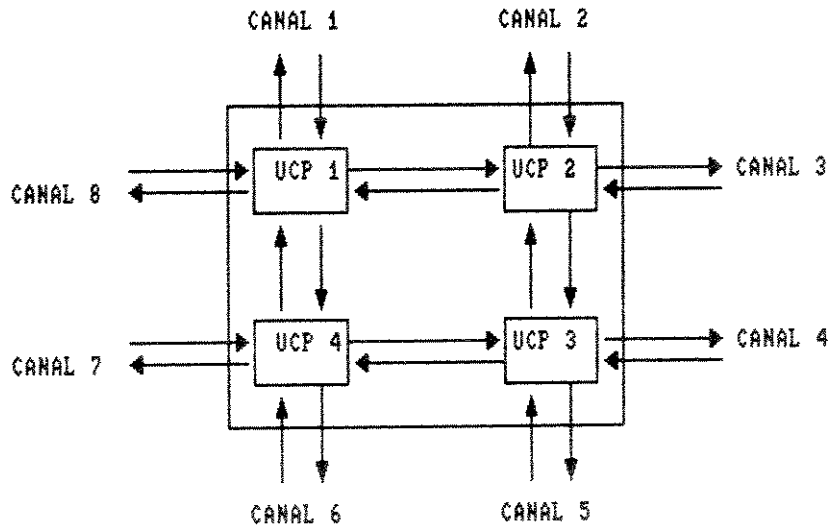


Fig. 4.6 – Nó formado por 4 processadores e 8 canais

4.4 Usando C Paralela

A implementação do hipercubo, usando a C paralelo da 3L, foi feita com relativa facilidade após o domínio da linguagem que, por sua vez, se mostrou bastante simples. Para uma boa distribuição de tarefas, nem sempre foi obedecida a seqüência lógica. Explicando com mais detalhes, se temos 16 tarefas e 4 processadores, o processador 0 recebe as tarefas 0, 7, 8 e 15; assim se a tarefa 0 for a mais trabalhosa, das quatro primeiras, em seguida ele processa a tarefa 7 que é a menos trabalhosa dos quatro que se seguem.

Para inicializar, o processador "root" (aquele ligado ao IBM-PC) interage com o usuário e recebe os devidos parâmetros, que repassa para os outros três processadores, sequindo-se a execução. Ao terminar, o processador "root" envia um aviso pelo caminho

Hamiltoniano(Código Gray), que circula por todos os processadores junto com o melhor resultado obtido. O processador seguinte, ao receber a mensagem, confere o resultado recebido com o seu resultado; se o resultado dele for pior que o recebido, retransmite o valor que recebeu; se o resultado dele for melhor, prossegue a retransmissão, mas com o seu resultado. Assim, quando a mensagem retorna ao "root", este tem condição de saber o melhor resultado global, que é impresso no vídeo. A retransmissão é feita através do autômato paralelo para melhor rendimento do autômato de cálculo. O resultado foi satisfatório, obtendo-se um "speedup" de 3 com relativa facilidade.

Nos testes feitos, verificou-se que um Transputer rodando em "monoprocessamento" trabalhando com números inteiros de 32 bits executa em 2 segundos um programa que demora mais de 3 minutos no IBM-PC (fazendo o uso de inteiros com 16 bits) o que equivale a dizer que roda quase 100 vezes mais rápido, superando até mesmo os microcomputadores com UCP 80386. Foi testado o multiprocessamento fazendo o uso da distribuição inicial do Capítulo 2 e o tempo de processamento caiu de 2 segundos para 0,7 segundos com 4 processadores, o que equivale a dizer que se obteve perto de 71% de aproveitamento na primeira tentativa com o problema da mochila ("Knapsack")[13].

Um algoritmo de redistribuição dinâmica de serviço aplicado ao mesmo problema é de supor que melhore significativamente o desempenho global. Porém, ao começar a implementar, o primeiro problema que surgiu foi que a linguagem C paralelo utilizada não tinha instruções que permitissem vigiar a chegada de dados por mais de uma porta por vez. Fazer o uso de comando chan_in_word_t que vigia a porta por um determinado tempo estabelecido pelo usuário, convertendo em um tipo de varredura multiplexada no tempo, significaria inevitável sobrecarga de processamento, tendo o agravante de que o dado enviado provavelmente não seria coletado imediatamente. Assim, pensou-se em colocar um processo vigiando e manipulando cada porta de entrada e saída mas, para coordenar tudo isso, era necessário que o processo coordenador pudesse ter a condição de vigiar todos os processos de entrada e saída. Portanto, procurou-se fazer uso da variáveis comuns, ou seja, fazer uso de memória comum entre os processos concorrentes. Ao fazer o teste constatou-se que, embora a linguagem não proiba fazer o uso da memória comum, o compilador utilizado não permite que dois processos com variáveis comuns sejam executados ao mesmo tempo, mesmo que se tome cuidado para evitar "colisões". Ao tentar fazer o uso simultâneo, o processador simplesmente aborta o programa, sem nada avisar. Ao constatar-se que sem esse recurso o desempenho cairia significativamente, não tinha mais sentido tentar implementar a redistribuição dinâmica de serviços. O primeiro e o segundo programa do apêndice C demonstram claramente a existência deste problema.

Constatada esta deficiência no compilador, procurou-se outra alternativa de linguagem ou do compilador. As linguagens Fortran Paralelo e Pascal Paralelo eram pouco promissoras pelo fato de terem seguido quase a mesma filosofia e, também de terem sido implementadas pela mesma empresa. Assim, a alternativa que restou foi fazer uso da nova linguagem OCCAM citada no capítulo anterior.

4.5 Usando OCCAM

A linguagem OCCAM utilizada foi a TDS versão 2.0 IMS 701 C do INMOS. Como foi citado no Capítulo 3, o TDS é um ambiente integrado de software, equipado com editor,

compilador, depurador e outros recursos mais, podendo trabalhar sem notar a presença do sistema operacional do IBM-PC. A presença do comando ALT, que permite vigiar mais de um canal simultaneamente, parecia tornar a linguagem muito mais promissora para implementar o algoritmo proposto. O livro de Burns [Burns 88] cita claramente que nesta linguagem é proibido que mais de um processo grave numa memória comum, mas ele pode ser lido por mais de um processo. A proibição de dois processos gravarem numa mesma variável comum atrapalha, mas não inviabiliza a aplicação. Assim, foram iniciados a implementação e o teste, e foi com surpresa que se notou a mesma deficiência do C Paralelo neste compilador. Isto é, ao fazer uso das variáveis comuns, embora somente um dos processos esteja gravando quando os dois processos entram en ação simultaneamente, o programa interrompe a execução, e os testes com "debugador" mostravam nítido problema de organização de biblioteca.

Os programas que confirmaram esta deficiência estão no Apêndice C juntamente com os resultados do teste.

Constatada esta deficiência, ficou claro que um dos importantes recursos do processo concorrente, que é variável comum, não pode ser utilizado como meio de comunicação entre o processo de cálculo e o processo de controle. Isso representa uma séria restrição que limita significativamente o desempenho e o algoritmo, pois o uso de canais entre processos envolve, inevitavelmente, esperas que comprometem o desempenho.

Os testes com distribuição inicial de serviços também funcionaram bem nesta linguagem, e os resultados foram ora a favor do C Paralelo, ora a favor da OCCAM, dependendo da aplicação. Em seu todo, a Occam mostrou-se mais eficiente na parte que envolve paralelismo, troca de mensagens e sincronizações, enquanto que a linguagem C Paralelo mostrou-se mais eficiente como linguagem em si, principalmente na parte que envolve portas de entradas e saídas como vídeo teclado ou discos.

Capítulo 5

CONCLUSÕES

Quando do início deste trabalho, o que se tinha como objetivo primário era encontrar algoritmos adequados para a solução de classes específicas de problemas existentes na engenharia. Após a escolha de classes e a proposição de algoritmos, iniciaram-se os testes de implementação. Devido à existência de uma máquina com quatro placas CPU's com microprocessadores M68000 (o Homuk), no Departamento de Computação e Automação Industrial da FEE da Unicamp, partiu-se para o estudo do software e hardware disponíveis a fim de se iniciar a implementação do hipercubo. Com o decorrer do trabalho, notou-se a existência de um grande vazio no software do Homuk e tentou-se superá-lo com a utilização do Cadmus (máquina com processador 68010), que conta com mais recursos de software. A importância desta primeira experiência foi a de mostrar qual o conjunto de recursos de comunicação entre CPU's, necessários num hipercubo, para a viabilização do carregamento inicial de trabalho, alimentação e coleta de dados. Esta experiência está registrada no apêndice B.

No instante em que se passou a discutir a redistribuição dinâmica de trabalho, sentiu-se que seriam necessários recursos adicionais de comunicação que permitissem um tráfego mais intenso de mensagens sobre a estrutura como, por exemplo, um buffer de entrada e saída que não fosse administrado diretamente pela CPU. Tais recursos não tiveram iniciadas as implementações porque surgiu a opção de se usar uma placa com quatro Transputers. O teste da distribuição inicial de serviços nos Transputers pode ser considerado bom, conforme foi citado no Capítulo 4. Todavia, ficou claro também que a redistribuição dinâmica de serviços iria exigir muito mais do hardware e, principalmente, do software.

Após exaustivos testes enfrentando dificuldades como documentação pobre do hardware e omissão no software, chegou-se à conclusão de que os compiladores das linguagens de alto nível testadas (C da 3L e OCCAM) não apresentam recursos que viabilizem uma redistribuição dinâmica de trabalho eficiente. O que falta nestes compiladores são recursos mais eficientes na administração de variáveis comuns a processos dentro da mesma CPU e/ou uma melhor administração da entrada e saída como, por exemplo a interrupção programada.

Devido a uma documentação ainda muito pobre das duas linguagens, o que impediu maior intimidade com a máquina, gastou-se bastante tempo e fez-se uma quantidade enorme de testes para que se concluisse a real falta de recursos adequados para a implementação dos procedimentos de redistribuição dinâmica de trabalho. Os testes estão citados no apêndice C.

Quanto ao aspecto do assunto título deste trabalho, o estudo dos autômatos na distribuição inicial de trabalho e redistribuição dinâmica, teve-se sucesso total, o que era de se esperar. Isto porque, desde que se restrinja a classe de autômatos àqueles cujos grafos

associados possam ser particionados em subgrafos com caminhos hamiltonianos, garante-se que nenhum procesador vá executar um trabalho já executado por outro e que exista um algoritmo que fornece a sequência de estados a serem percorridos.

Dentro da classe de autômatos, com as restrições impostas no trabalho, estão aqueles com grafos que representam os reticulados ("lattice") da álgebra de Boole e do permutoedro. Estes dois autômatos são usados em Cálculo Proposicional, Projetos de Circuitos Digitais, Roteamento de Ligações em Circuitos Integrados e Impressos, Teoria de Códigos ("Knapsack") e todos os problemas de otimização modelados pelo Problema do Caixeiro Viajante.

O que poderia ser enfatisada nestas conclusões é que o princípio do "Quem acabar primeiro vai ajudar o outro", como filosofia de redistribuição dinâmica de trabalho entre processadores, é o que se pode ter de mais natural quando se busca uma solução para distribuição balanceada de tarefas. Ele, entretanto, exige que haja recursos para se manter um tráfego intenso nas linhas de comunicação e que se tenha recursos adicionais para a administração da análise e redistribuição de mensagens, sem que a própria CPU tenha que intervir no processo. Em resumo, tem que existir um administrador de canal eficiente.

Ficam registrados aqui as deficiências e méritos notados nos compiladores utilizados.

O C Paralelo versão 2.0 da companhia 3L Ltda [3L 88].

O mérito é a parte seqüencial ser uma linguagem relativamente poderosa e rápida, e bastante conhecida pelos programadores.

As deficiências são:

i) O detector de erros tanto do compilador como do configurador é bastante limitado, detectando apenas os erros normalmente encontrados em programas sequenciais, tais como erros de sintaxe, uso de variáveis não declaradas ou erros com chaves, colchetes ou aspas.

ii) Para multiprocessamento, muitos erros foram ignorados, tais como: erros no comandos de comunicação entre processos, erros de manipulação de semáforos e variáveis comuns; eles foram completamente ignorados na compilação, resultando em erros de execução.

iii) Eventuais erros de execução também são ignorados. Quando surge problema de erro na execução o sistema simplesmente se bloqueia ou gera resultados errados ou inesperados. Isto não só dificulta a correção do programa como reduz a confiabilidade do software.

iv) Muitas das limitações não são citadas nos manuais como é o caso do número máximo de canais e "threads" que podemos ter ou acionar ao mesmo tempo. Só se descobrem estes limites quando acidentalmente se ultrapassam os 4 canais permitidos.

v) O semáforo é controlado com certa segurança enquanto existe um número muito limitado de variáveis comuns. Conforme aumenta o número de processos e de variáveis comuns o controle dos semáforos passa a aumentar em complexidade chegando rapidamente a um ponto em que o usuário tem dificuldade de ter o domínio da situação. O mais grave de tudo isso é que um erro na manipulação do semáforo tem efeito sério na confiabilidade, desempenho e continuidade do programa.

vi) As variáveis locais fazem o uso de memórias internas ao "chip", para trabalhar com maior velocidade. Porém, um eventual estouro destas memórias de apenas 4K bytes nem sempre resulta em mensagem de erro, exigindo do usuário muito cuidado com este

detalhe. É possível fazer o uso de memória externa para variáveis locais, mas deve-se avisar o compilador no momento da compilação. Isto não parece um problema difícil de ser resolvido, pois o OCCAM usa a memória interna da mesma forma mas, quando ocorre o "estouro" da memória interna, usa a memória externa como extensão, automaticamente.

vii) Ao mandar ler o dado de um canal o processo fica esperando a chegada de dados, o que é muito bom para o sincronismo. Porém, isto significa que, para vigiar as quatro portas de entrada, deve-se gerar quatro processos paralelos, pois o uso de multiplexação resulta não só em sobrecarga de processamento como pode resultar em uma resposta mais lenta. Uma vez que o hardware tem recursos de interrupção, deveria dar ao software condições de explorar melhor esses recursos. Este problema fica diretamente relacionado com a observação 5. Ao criar quatro processos em paralelo por UPC para vigiar as porta de entrada, já se tem 5 processos em paralelo que, para um bom funcionamento, devem ter variáveis em comum e, também, semáforos. Um comando do tipo "alt" (conf.[Burns 88, Demi 89, INM3 88, INM6 89, INM7 88]), que vigia muitas portas simultaneamente com o uso das interrupções existentes na OCCAM, faz muita falta aqui.

viii) A possibilidade de se ter um processo com maior prioridade não ajuda muito a vigilância dos canais, pois os canais são 4 entradas e 4 saídas e a prioridade só tem dois níveis. A observação 10 vem complementar este problema.

ix) Certas instruções mostraram-se problemáticas. Por exemplo, no uso do "^" (ou exclusivo) o compilador não detecta erros, mas quando essa instrução é executada apresenta problemas. O mesmo ocorre quando se usam instruções IF , WHILE o SWHICH com lógicas muito complexas e longas.

x) Embora canais de comunicação tenham alto desempenho, notou-se que existe o risco de "overrun" (sobreposição de comando de entrada), pois o processador também tem alto desempenho. Deve-se tomar cuidado para evitá-los. Na verdade o compilador poderia resolver este problema (bastaria consultar o status do canal), mas isso inapelavelmente resultaria em atraso na transmissão. Aqui sente-se muito a falta de um "buffer" ou "fifo", que provavelmente não foi colocado pela limitação do espaço no chip.

xi) Os testes mostraram que, ao se acionar simultaneamente dois processos que tenham variáveis comuns, o sistema apresenta resultados imprevisíveis. Tudo indica que, como um dos processos ocupa a área da variável, o outro não consegue fazer uso desta. Isso limita em muito a flexibilidade e o potencial da linguagem.

Segue-se a análise de aspectos positivos e negativos do compilador da linguagem OCCAM usando DTS versão 2.0 IMS 701 C do INMOS Corp..

Os aspectos positivos:

i) A linguagem permite manipular e visualizar o paralelismo diretamente no programa, facilitando o programador.

ii) Está dotada de detectores de erros melhores que a C paralelo, facilitando em muito o programador.

iii) Embora limitada, está equipada com mensagens de erros em execução.

iv) O TDS possui depurador que vem facilitar em muito a correção do programa.

v) A linguagem vem facilitar em muito a parte de envio e recepção de mensagens e controles de sincronismo, de tal forma que a sobrecarga do programa e do programador na parte de controle fica menor que em relação a linguagens seqüenciais adaptadas.
vi) A presença de "constructors" como ALT, PAR e FOR vem facilitar em muito o programador, permitindo simplificar a parte de controle e sincronismo.

Os aspectos negativos:

i) O sistema testado mostrou-se mais limitado em comandos de manipulação de periféricos, aumentando o tamanho dos programas.
ii) Em seu todo, o programa em OCCAM fica maior do que em C paralelo pois, embora economize na parte de controle e sincronização, nas outras partes não apresentou o potencial de uma linguagem como C paralelo.
iii) Ao utilizar multiprocessamento, é freqüente o sistema TDS bloquear-se de tal forma que nem o "Control-Alt-Del" permite reiniciar o sistema. Somente desligando e religando o computador é que o sistema TDS é recarregado. Isto nunca ocorreu com C paralelo. Nestas condições de nada adianta um depurador.
iv) A limitação imposta a variáveis comuns, embora venha a aumentar a confiabilidade, está restringindo um importante recurso para troca de informações para processos concorrentes.
v) O configurador que acompanha a OCCAM mostrou-se extremamente ineficiente quando se trata de multiprocessamento. Uma determinada arquitetura só é aceita se a declaração de UCPs, canais e programas obedece a certa lógica, o que não é citado no manual. Caso esta lógica não seja obedecida, o TDS simplesmente não consegue carregar os programas através da rede.
vi) Embora o manual avise que para dois processos concorrentes não pode haver dois processos gravando numa mesma variável comum, os testes mostraram que, mesmo nos casos em que somente um dos processos grava, também surgem graves problemas, caso sejam acionados ao mesmo tempo, como mostram os testes no apêndice C.

O que ficou em evidência, também, é que no que se refere a paralelismo, sincronismo e comunicação entre os processos, seja o C Paralelo, Fortran Paralelo, Pascal Paralelo ou Occam, estas linguagens estão ainda trabalhando em nível de máquina, sendo que os usuários devem visualizar e manipular com bastante cuidado a programação.

Finalmente, é importante frisar que, nos próximos anos, haverá um crescimento acentuado do processamento concorrente entre processadores, já que existe um barateamento do hardware e um crescimento no potencial das linguagens de alto nível. Um outro aspecto importante é que as estruturas do tipo MIMD ("Multiple Instruction Multiple Data") com arquiteturas de CPU's do tipo RISC ("Reduced Instruction Set Computer") serão predominantemente usadas em processamentos concorrentes de uso mais geral, que exigem reconfiguração da estrutura da rede de interconexão dos processadores. As estruturas do tipo SIMD ("Single Instruction Multiple Data") deverão ficar restritas às máquinas vetoriais.

Existe ainda a grande expectativa causada pela filosofia de desenvolvimento de linguagens orientadas a objetos [Taka 88] que facilita a compreensão dos processos concorrentes,

portanto deverá encabeçar a família das metodologias de desenvolvimentos de sistemas concorrentes num futuro próximo.

Apêndice A

SISTEMA COM ARQUITETURA RECONFIGURÁVEL

No Capítulo 1, mencionou-se uma arquitetura chamada VTM (Variable Topology Multicomputer)[DeCe 89, Park 83], ou seja, um computador com arquitetura reconfigurável. Neste apêndice é apresentada uma proposta feita durante a implementação em hardware de um acumulador do projeto PAE [Furu 84], onde foram desenvolvidos testes de uso de memórias PRON's como chaveadores de sinais. O problema de chaveamento de sinais tem grande importância na implantação de arquiteturas concorrentes em que se usam roteadores específicos [May 90, May&T 90] para rotearem mensagens entre processadores.

Durante a implementação em hardware de um acumulador do projeto PAE [Furu 84] o problema de chaveamento para operações de deslocamento era crítico. O acumulador precisava deslocar tanto para a direita como para a esquerda, com entrada de um ou zero, além de permitir que circulasse tanto para a direita como para a esquerda. A implementação inicial envolvia vários C.I.'s de tecnologia TTL. Percebeu-se, então, que todo o circuito poderia ser implementado com uma única PROM devidamente programada [Furu 84], como é apresentado na figura A.1.

Como pode ser visto na figura A.1, bits de endereço passaram a funcionar como entrada de dados e, a saída continuou a funcionar como saída de dados.

Explicando com mais detalhes: os endereços mais significativos (C,D e E) passaram a funcionar como decodificadores. Por exemplo, se C, D e E forem 0, 1 e 1, então a PROM seleciona o deslocamento para a esquerda com entrada do sinal "0". Neste caso, sejam quais forem os sinais que venham nos endereços A e B, a saída S4 deve manter sinal "0". Para que isso aconteça, basta que se programe a PROM com valores "0" para os endereços 00000, 00001, 00010 e 00011. Se C, D e E forem 1, 1 e 0, então o acumulador deve circular para a esquerda. Neste caso, o sinal do D11 que entra no endereço B deve sair para o Sr(Shift right). Para que isso ocorra, basta que se programe a PROM com valor de B na saída S4, ou seja, o "bit" 4 terá o valor 0 para os endereços 11000 e 11001, e terá o valor 1 para endereços 11010 e 11011.

Naturalmente, junto com estes sinais de saída, saem os sinais S6 e S7 que vão controlar o deslocamento do registrador de deslocamento SN 74194[INT 81]. Como pode ser visto no exemplo, os sinais no "bit" de endereço A ou B podem "sair" em qualquer uma das saídas da memória, conforme se programe a PROM. Programada devidamente, esta unidade de memória não somente pode se multiplexar, mas também fazer sair o sinal em várias saídas simultaneamente ou mesmo realizar funções lógicas.

Utilizando unidades de memórias com maior área de endereçamento, pode-se ter maior número de entradas. Uma EPROM com 4K de endereçamento que possui 12 bits de endereço pode ser utilizada com 4 deles sendo utilizado para chaveamento, podendo chavear oito entradas e oito saídas (se tiver 8 saidas) de dezesseis formas diferentes. A memória

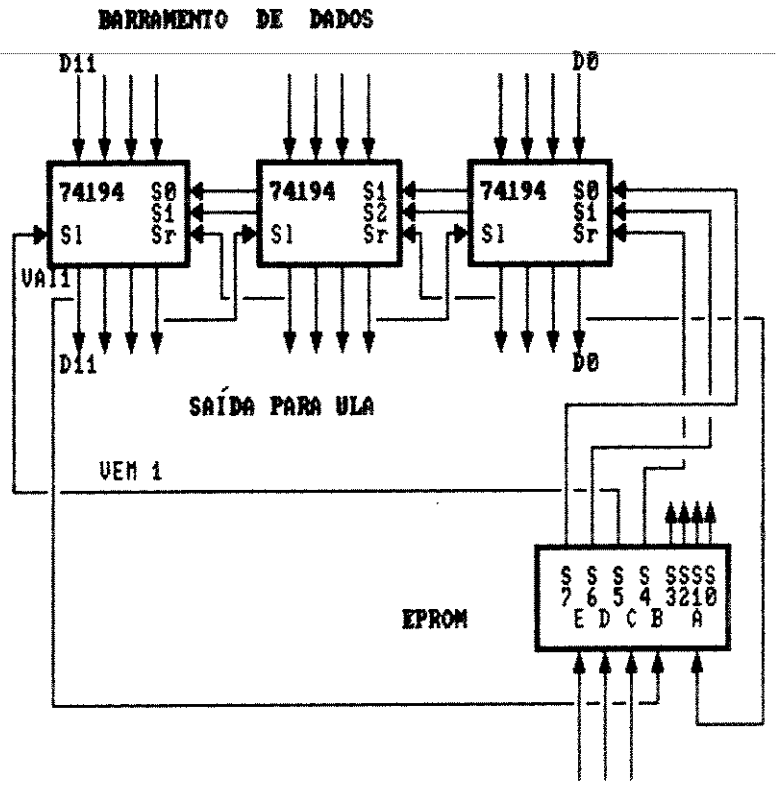


Fig. A.1 – Exemplo de uma PROM sendo utilizado como chave

ficaria como na figura A.2.

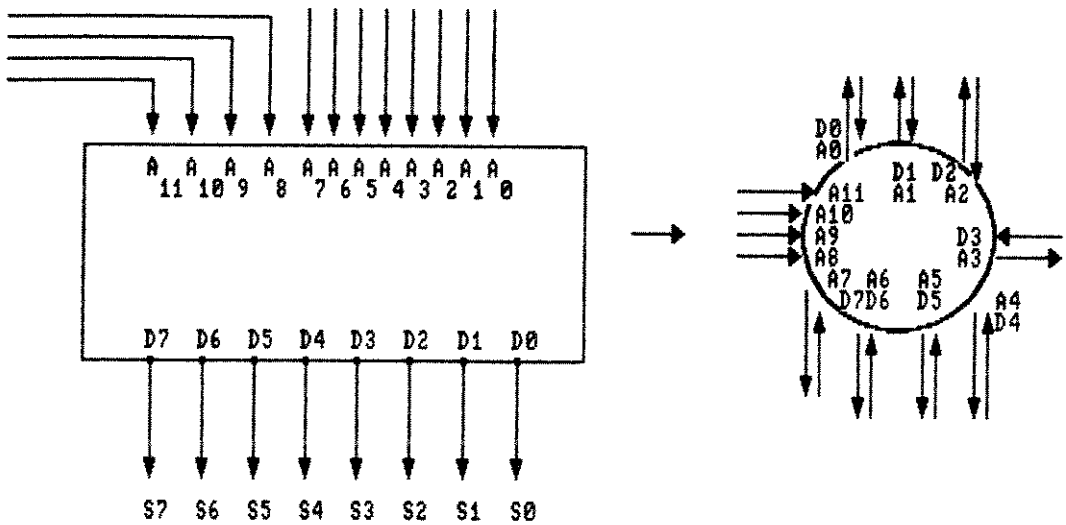


Fig. A.2 – Uma memória sendo utilizado como chave

As oito entradas e saídas podem ser colocadas em paralelo, formando palavras de quantos "bits" forem necessários. Naturalmente, neste caso as EPROMs devem ser programadas de forma idêntica e controladas pelo mesmo sinal de controle (endereços).

Quando se desejar maior flexibilidade de comutação, além dos limites das EPROMs, deve-se usar memórias RAMs que tem a vantagem de ser reprogramáveis. É exigido, entretanto, um circuito extra ligado a um dos processadores do sistema para permitir a reprogramação.

O esquema seria o da figura A.3.

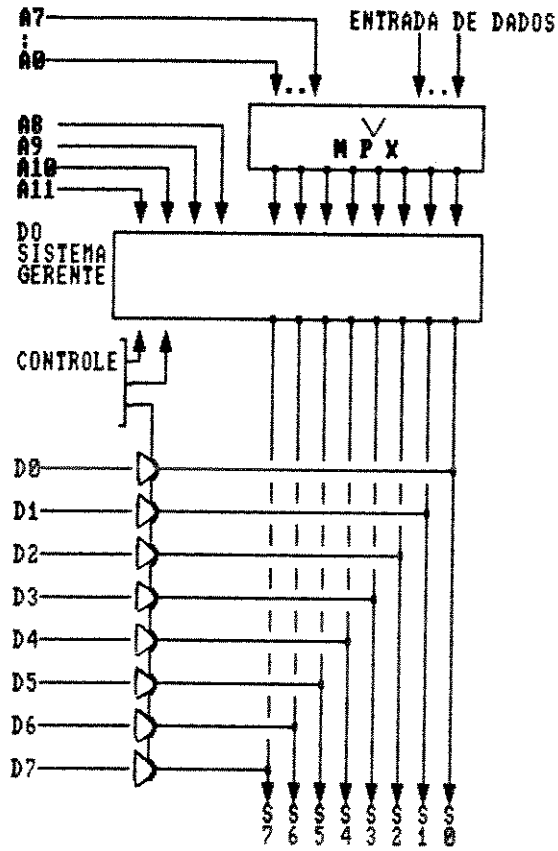


Fig. A.3 – Um circuito com comutador de memória

Representando-se o circuito anterior como uma esfera, a ligação com os computadores fazendo o uso desta "chave" ficaria como na figura A.4.

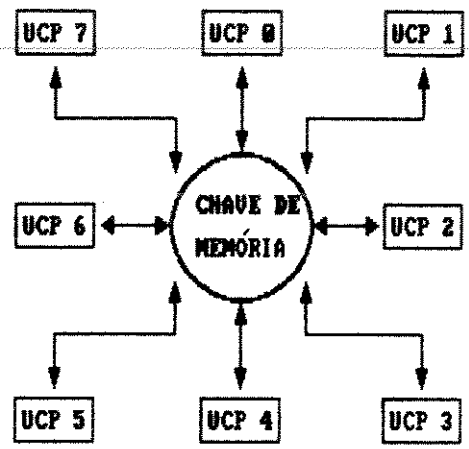


Fig. A.4 – O comutador com os processadores ligados

Conforme a programação, pode-se ligar qualquer um dos processadores a qualquer outro, ou mesmo, um dos processadores comunicando-se com todos os outros, permitindo ter uma arquitetura reconfigurável para várias formas, como pode ser visto na figura A.5.

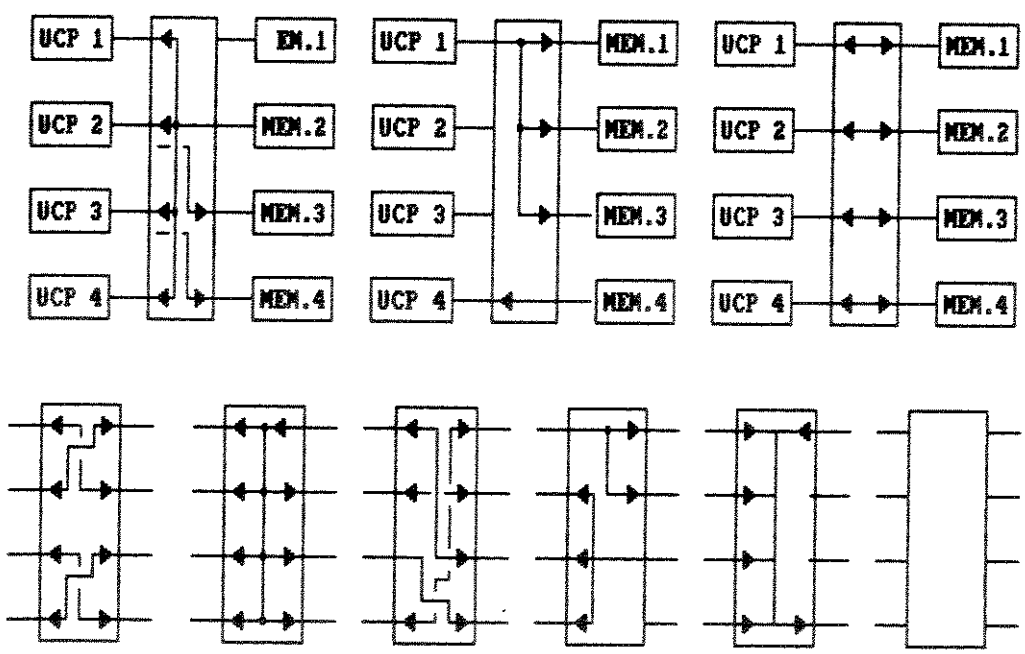


Fig. A.5 – Algumas alternativas da configuração

Neste caso, além de enviar os sinais normalmente, tem-se a alternativa de se enviar sinais negados, ou mesmo, formando lógicas entre os sinais a enviar.

Quando se desejar maior expansão, pode-se ligar mais de um conjunto de memórias

como na figura A.6.

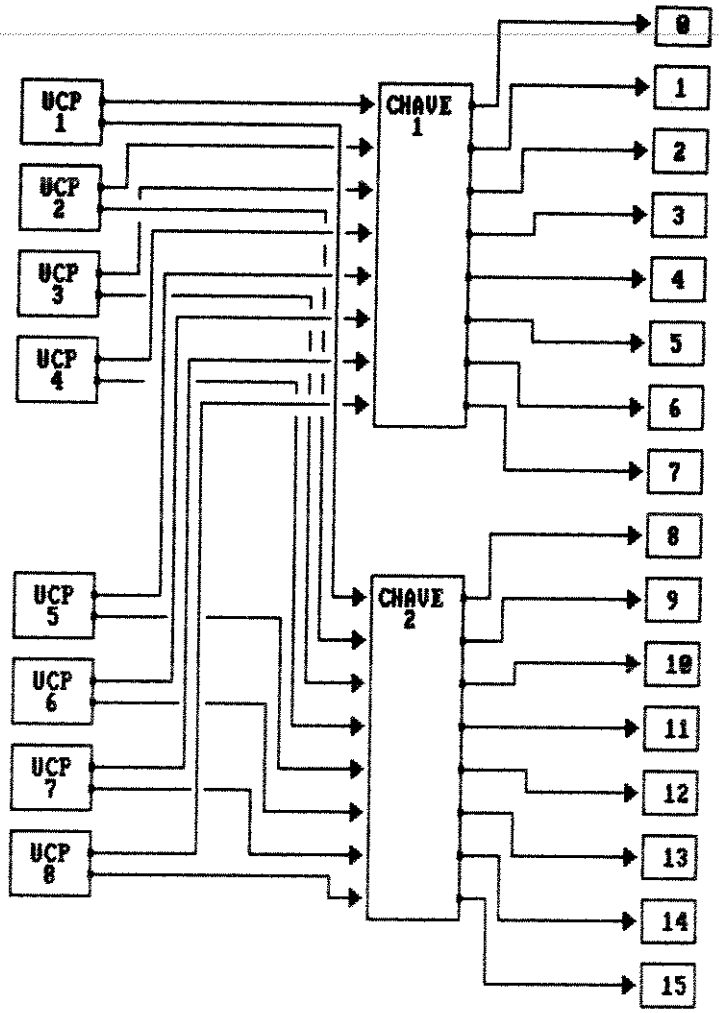


Fig. A.6 – Dobrando o número de linhas

O acréscimo de mais uma chave de memória permite que mais de um processador se comunique a outro determinado processador.

A.1 – Dificuldades existentes no uso de EPROM como comutadores

Naturalmente pode-se encontrar desvantagens e dificuldades na utilização de memória como comutadores de linhas. Pode-se citar entre elas:

i) Atraso da memória. O tempo de atraso do sinal na memória é bem maior do que nas portas lógicas. Portanto, utilizando-se uma memória, tem-se que levar em conta este atraso.

ii) Possível instabilidade de sinal nas mudanças de endereços das memórias. O fabricante não garante a estabilidade de sinal enquanto o endereço esta sendo comutado. Existem até casos de memórias que não fornecem os dados de saída enquanto não se chaveia o sinal de "ativa a saída".

iii) Expansividade. Até um determinado número de pinos de saída, os comutadores podem

expandir facilmente, mantendo ligações ponto a ponto entre quaisquer processadores. Entretanto, existem restrições para se manter esta propriedade quando aumenta o número de processadores.

A.2 – Algumas aplicações de EPROMS como comutadores

Entre possíveis aplicações, pode-se citar um chaveamento de rede de processadores, chaveamento de um multiprocessador, ou chaveamento de bancos de memórias. Assim, os processadores com configuração da figura A.7 se comunicam como se estivessem ligados de acordo com a configuração apresentada na figura A.8.

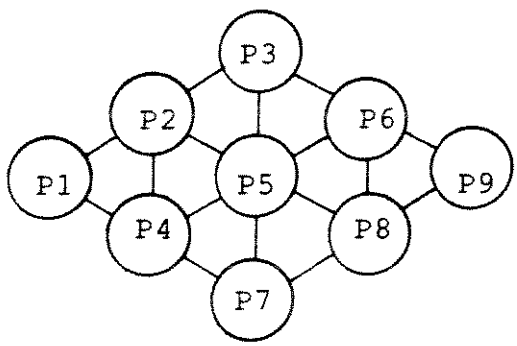


Fig. A.7 – Processadores ligados hexagonalmente

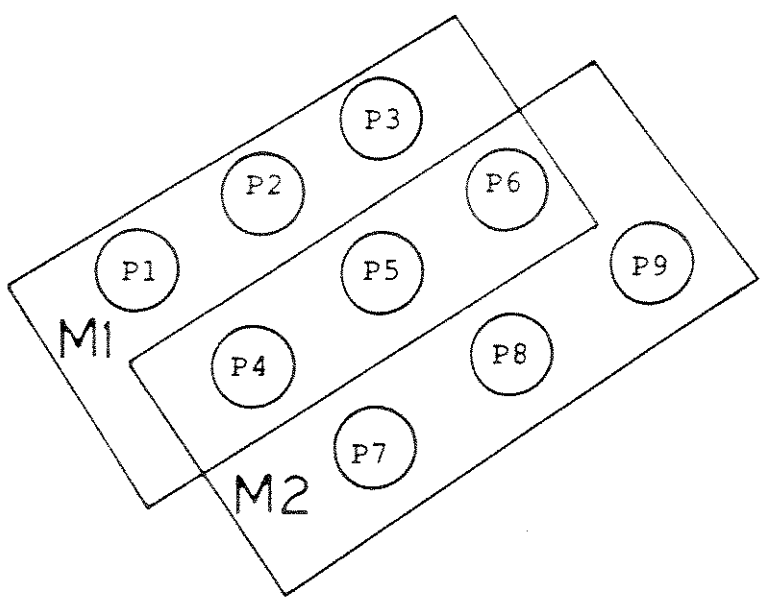


Fig. A.8 – Processadores ligados com memórias

Como pode ser visto, tem-se uma grande economia de interfaces em cada processador, da mesma forma que a fiação se torna muito mais simples.

Numa conexão como Hipercubo [Seitz 85] poder-se-ia utilizar algo como a figura A.9.

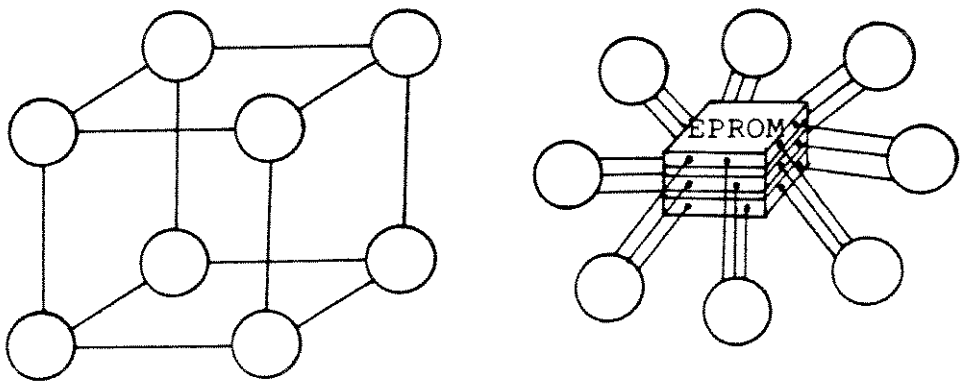


Fig. A.9 – Cada processador pode interagir com vizinhos próximos

Prosseguindo na busca de arquitetura flexível e mais completa, foi testada uma unidade com 8 entradas e 16 saídas, como na figura A.10. O resultado foi a configuração da figura A.11 onde todas as 16 UCPs podem intercomunicar-se simultaneamente.

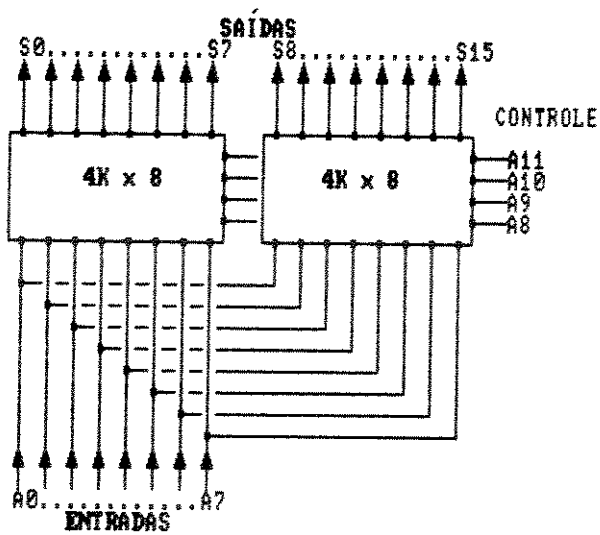


Fig. A.10 – Usando duas memórias de 4K X 8 e 16 saídas

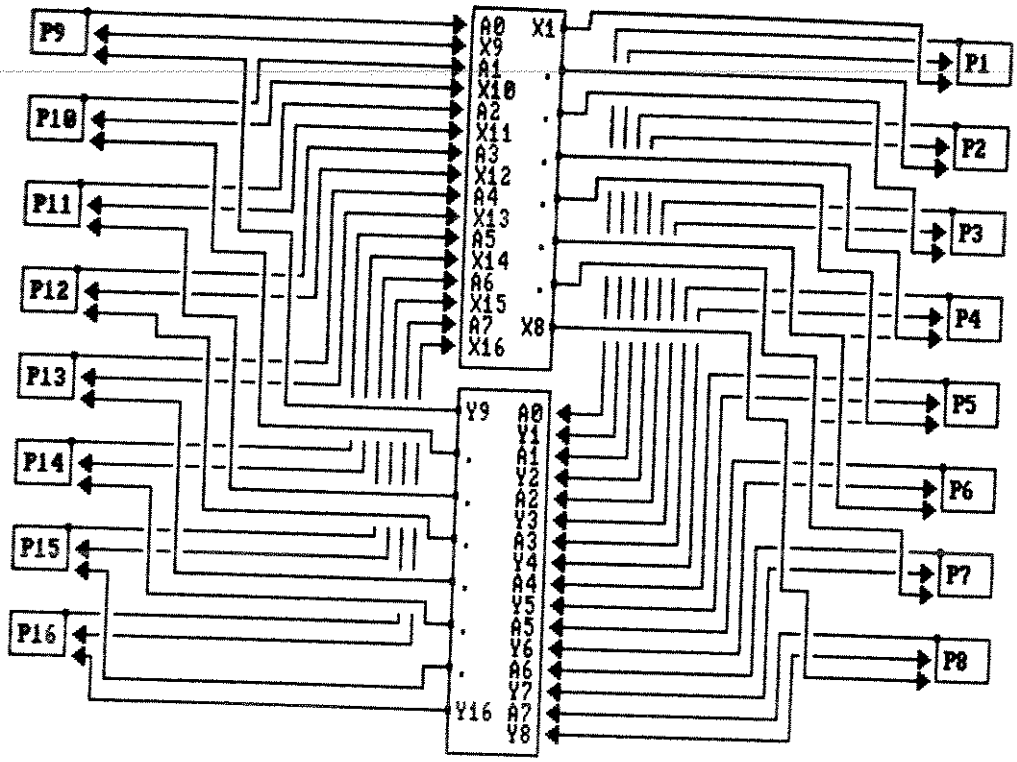


Fig. A.11 – Comunicação com 16 processadores simultaneamente

Estendendo-se a idéia, tem-se a figura A.12 onde 24 processadores podem trocar informações simultaneamente.

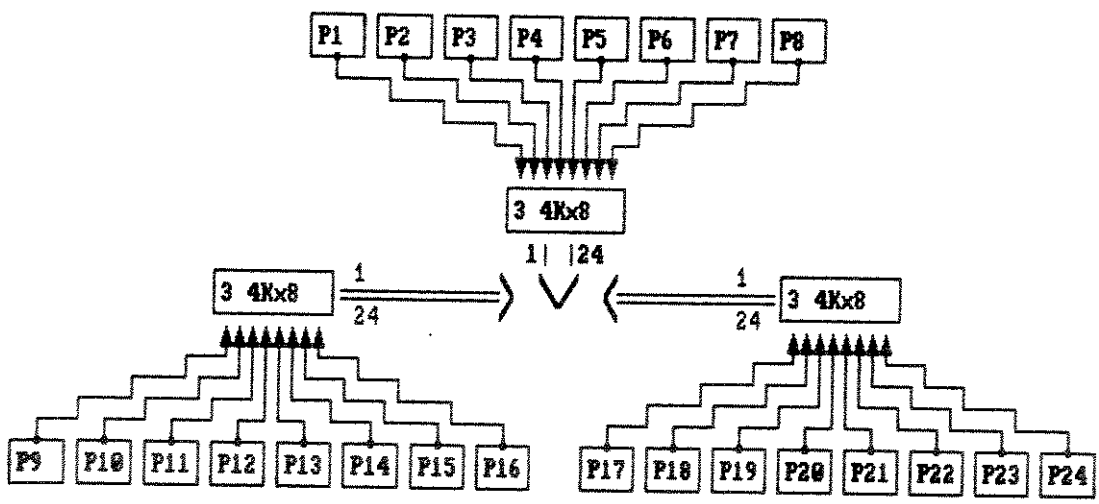


Fig. A.12 – Comunicação com nove unidades de memória de 4K×8, e 24 processadores

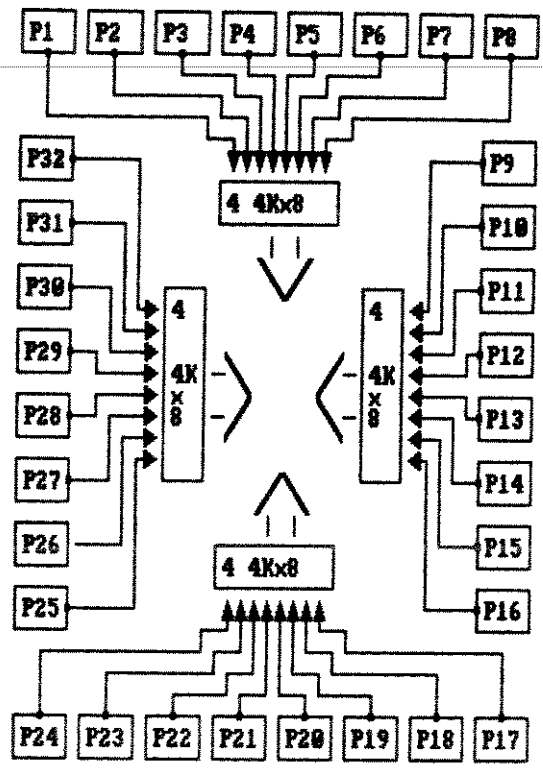


Fig. A.13 – 32 processadores ligados entre si

Assim, utilizando-se as memórias, como foi explicado, para cada $8n$ processadores precisa-se de n^2 memórias de 4K endereços e 8 bits de saída, precisando de n interfaces para cada processador. Caso se queira comunicar com 8 bits em paralelo, deve-se multiplicar este número por oito. Para verificar o comportamento das memórias utilizadas nestas condições, foram feitos testes utilizando memória RAM P2114 A-4 [INT 81] da Intel.

O resultado foi bem mais animador do que o esperado. As perturbações nas saídas com mudanças de endereços não ocorreram, e o circuito funcionou bem até a freqüência de operação de 4 MHz, bem acima da especificação do fabricante para o uso normal. Acima desta freqüência as perturbações não foram testadas.

Assim, chega-se à conclusão de que memórias semicondutoras, pela sua alta compactação e flexibilidade, podem ter aplicações muito mais diversificadas do que as previstas inicialmente.

APÊNDICE B

TENTATIVA DE UTILIZAÇÃO DO HOMUK PARA MONTAR O HIPERCUBO

Existe na Faculdade de Engenharia Elétrica da UNICAMP (FEE) uma máquina multiprocessadora denominada HOMUK [ELT 83, Enca 84], com barramento comum e as placas de UCP obedecendo padrão VME [VME 85].

Atualmente o HOMUK possui 4 placas de UCP, sendo que uma das placas possui: 1 UCP M68000, 1 Mbytes de memória RAM, 32K bytes de memória ROM, controlador de driver floppy de 8"(normalmente controla até 4 drivers), 4 entradas e saídas seriais (usando SCC do Zilog - taxa máxima de transmissão de 1Mbps), saída para barramento VME e saída para barramento comum.

As outras 3 placas possuem: 1 UCP M68000, 256k bytes de memória RAM, 32K bytes de memória ROM, controlador de driver floppy de 8"(normalmente controla até 4 drivers), 2 entradas e saídas seriais (usando SCC do Zilog - taxa máxima de transmissão é de 1Mbps), saída para barramento VME e saída para barramento comum.

O HOMUK está equipado com 2 winchesters de 20 Mbytes e 1 driver de floppy de 8". A placa de UCP possui dois conectores tipo VME com 96 pinos em 3 filas de 32 pinos por conector. O conector superior destina-se a barramento VME. O conector inferior tem três filas de pinos, sendo que uma das filas é utilizada por barramento comum, a outra, para conectar saída para os drivers do floppy,e a última é utilizada para as duas entradas e saídas seriais. Assim equipado, o Homuk tem a capacidade não só de carregar os programas em cada UCP, como os UCPs podem trocar informações entre si através de barramento comum. Embora não esteja equipado para comunicar-se com linhas seriais especiais, os SCC (chip de comunicação serial) da Zilog, são capazes de transferir os dados com uma taxa de 100K bits por segundo, permitido fazer teste de hipercubo com dimensão 2 (4 UCPs).

B.1 – Hardware e software do HOMUK

Como foi dito anteriormente, o HOMUK está equipado com 4 UCPs montadas sobre placas EUROCOM E3-1.3/68K, com dois conectores do tipo VME, sendo que o primeiro conector obedece ao barramento padrão VME e o outro serve como saídas para o controlador de disco floppy e duas entradas e saídas seriais.

A primeira placa é dotada dos seguintes recursos: uma UCP tipo MC68000 com relógio comutável de 8MHz e 12.5MHz, interface para barramento padrão VME, 1 Mbytes de RAM, 32 Kbytes de ROM (Comutável para 16k ou 64K bytes por "jumper"), unidade controladora de discos "floppy",(podendo controlar até 4 unidades de disco de $5\frac{1}{4}$ ou 8 polegadas, em simples ou dupla face, e densidade simples ou dupla). quatro entradas e saídas seriais (com taxa de transmissão programável que, dependendo da placa que for ligada, pode obedecer ao padrão RS232c, RS422 ou RS423, atualmente estão ligadas placa padrão RS232c) e relógio para tempo real, dotado de bateria, (com capacidade de marcar segundos, minutos, hora, dia, mês, ano e dia da semana, dotado também de alarme para acionar a interrupção, além de temporizador que pode gerar interrupções a cada 19.5 mS).

As outras três placas são similares, exceto que possuem apenas 256 Kbytes de memória e duas entradas e saídas seriais. Na figura B.1 pode-se ver o aspecto geral da placa E3– 1.3/68K. Assim, pode-se contar com placas de interface padrão RS232C, RS422 ou RS423, placa de interface do barramento padrão VME para o barramento comum, placa controladora de discos rígidos, etc. Outras placas são oferecidas pelo fabricante como placas de apoio para UCP.

A Figura B.2 mostra como ficam organizados os conectores padrão VME de 96 pinos com 32 pinos por coluna, onde o conector superior destina-se ao barramento padrão VME e o conector inferior está alocado para barramento comum.

A saída serial não é compatível com o padrão RS232. Para conversão é utilizada uma placa adaptadora equipada com C.I.s MC1488 e MC1489 da Motorola que converte o sinal de 0V e 5V para +12V e -12V como na figura B.3. Para distâncias curtas, pode-se pensar em dispensar tal placa, permitindo assim uma maior taxa de transmissão, mais adequada ao teste de multiprocessamento.

Com esses recursos, pode-se expandir o sistema para uma configuração de multiprocessamento, podendo-se inclusive contar com o apoio de placas de outros fabricantes que obedeçam ao padrão VME comercializados no mercado. Assim, o sistema consegue ter uma configuração como a da figura B.4 onde o barramento comum serve de meio de comunicação.

O fabricante do HOMUK oferece opções de compra de UCPs escravas para barramento VME, o que permitiria uma arquitetura, como na figura B.5, formando uma estrutura hierarquizada.

Os endereços da memória estão organizados como na tabela B.1 com os endereços distintos para controle, portas de entradas e saídas, EPROMs, memória local e acessos a barramento padrão VME.

Com relação a software, o HOMUK é pobre, no momento,como se pode ver na descrição sequencial do uso do software básico disponível atualmente no HOMUK da FEE – UNICAMP.

Um monitor residente em EPROM é acionado logo que a máquina é ligada. Este

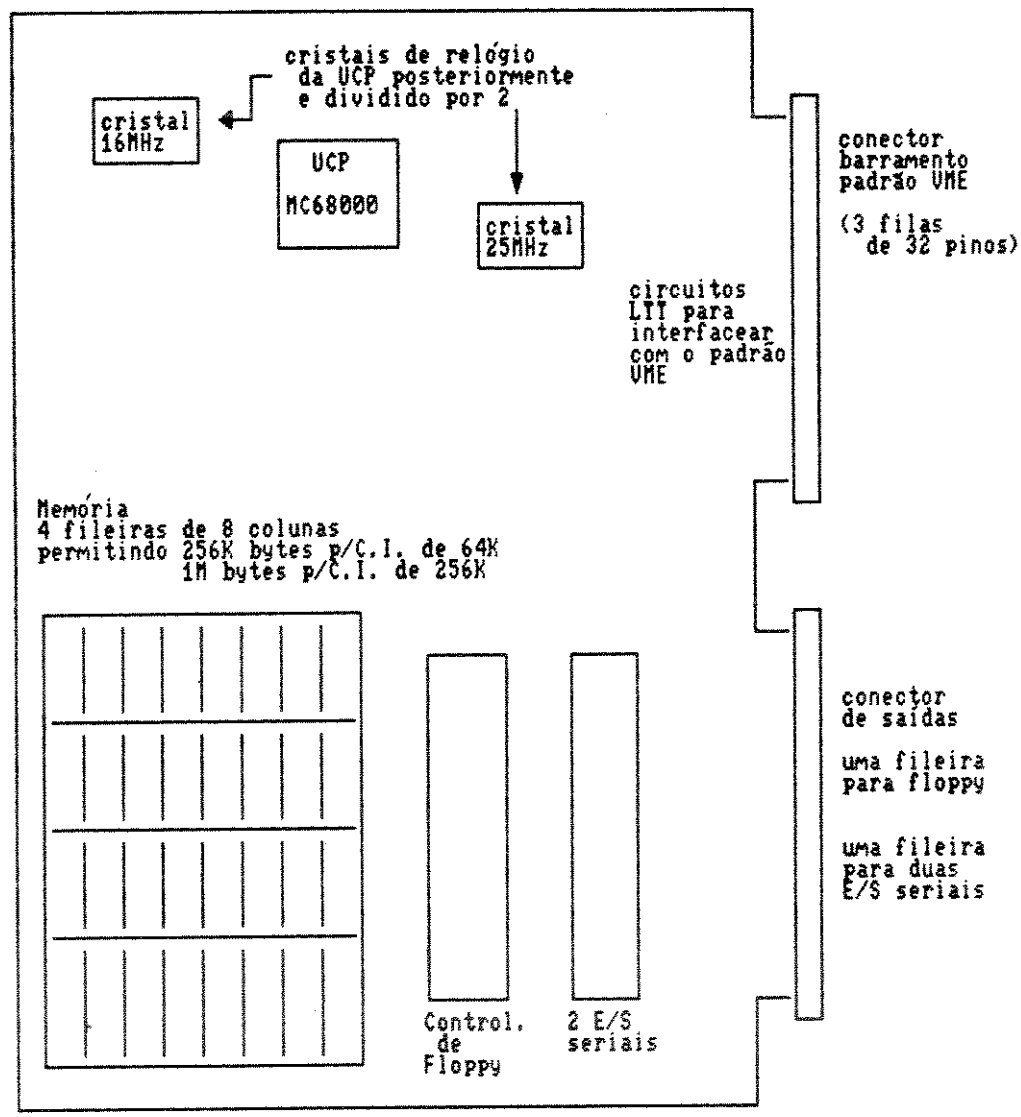


FIG B.1 – Aspecto Geral da Placa UCP

monitor conta com um assembler e desassembler e comandos de controle da memória e entrada e saída, além da carga do sistema operacional("boot"). Com isso pode-se carregar os sistemas operacionais como HTS ou CAD/1.

O HTS ou Higher Time Sharing Operating System é um sistema operacional desenvolvido para Univac V70 e conta com recursos para administrar tarefas de multiusuário em tempo partilhado, alocação dinâmica de memória, relocação dinâmica de programas, subrotinas recursivas e reentrantes e "Overlaying" automático de programas. Porém, este sistema operacional relativamente sofisticado conta com poucas linguagens e programas utilitários disponíveis. Na parte de linguagens, conta apenas com uma versão melhorada

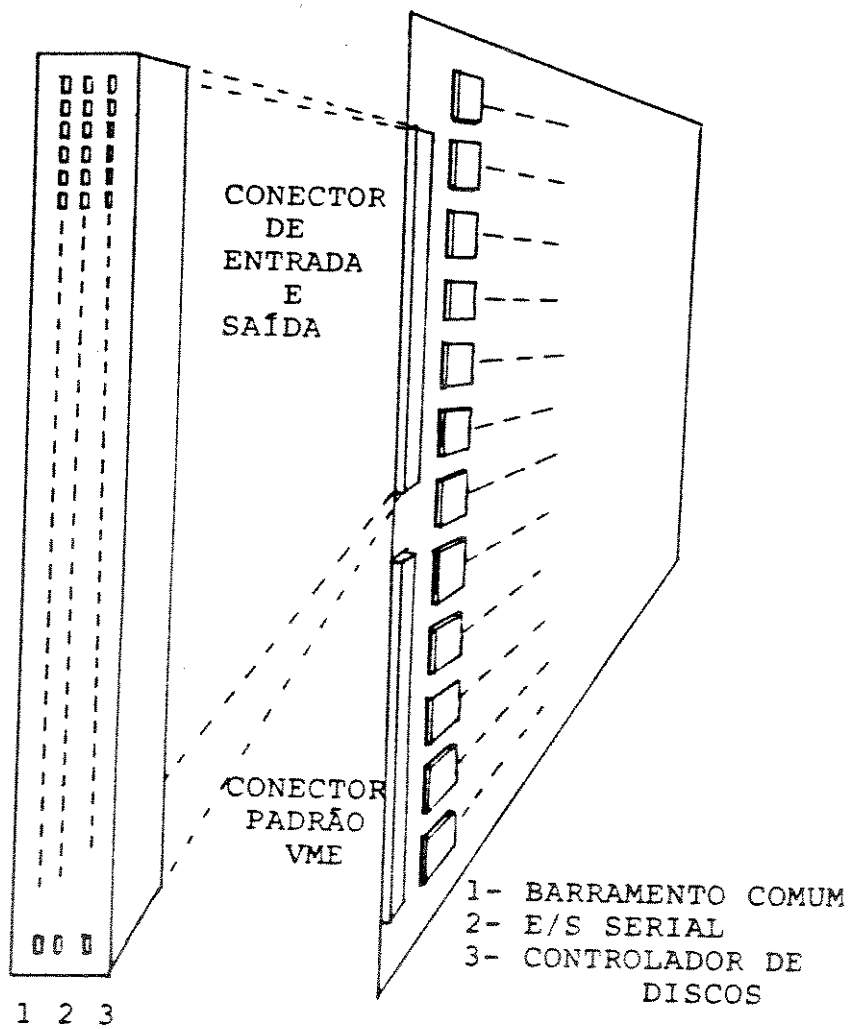


Fig. B.2 – Conectores padrão VME de 96 pinos

do FORTRAN IV e assembler DAS.

O outro sistema operacional disponível é o CAD/1 ou Flexos Operating System (versão 4.0), que conta com recursos gráficos prevendo inclusive o uso de plotters, tablets e "work-station" GKS para multiprocessamento. Em termos de linguagem, o CAD/1 continua bastante pobre, podendo contar apenas com uma variante do FORTRAN 66 expandido com recursos gráficos e "multitask" além de assembler do MC68000.

O sistema disponível para multiprocessamento, mostrou-se muito pouco confiável tanto na transferência de dados como nas transferências de programas, não tendo utilização prática. Além disso, para muitas aplicações, os 256K bytes de memória mostravam-se

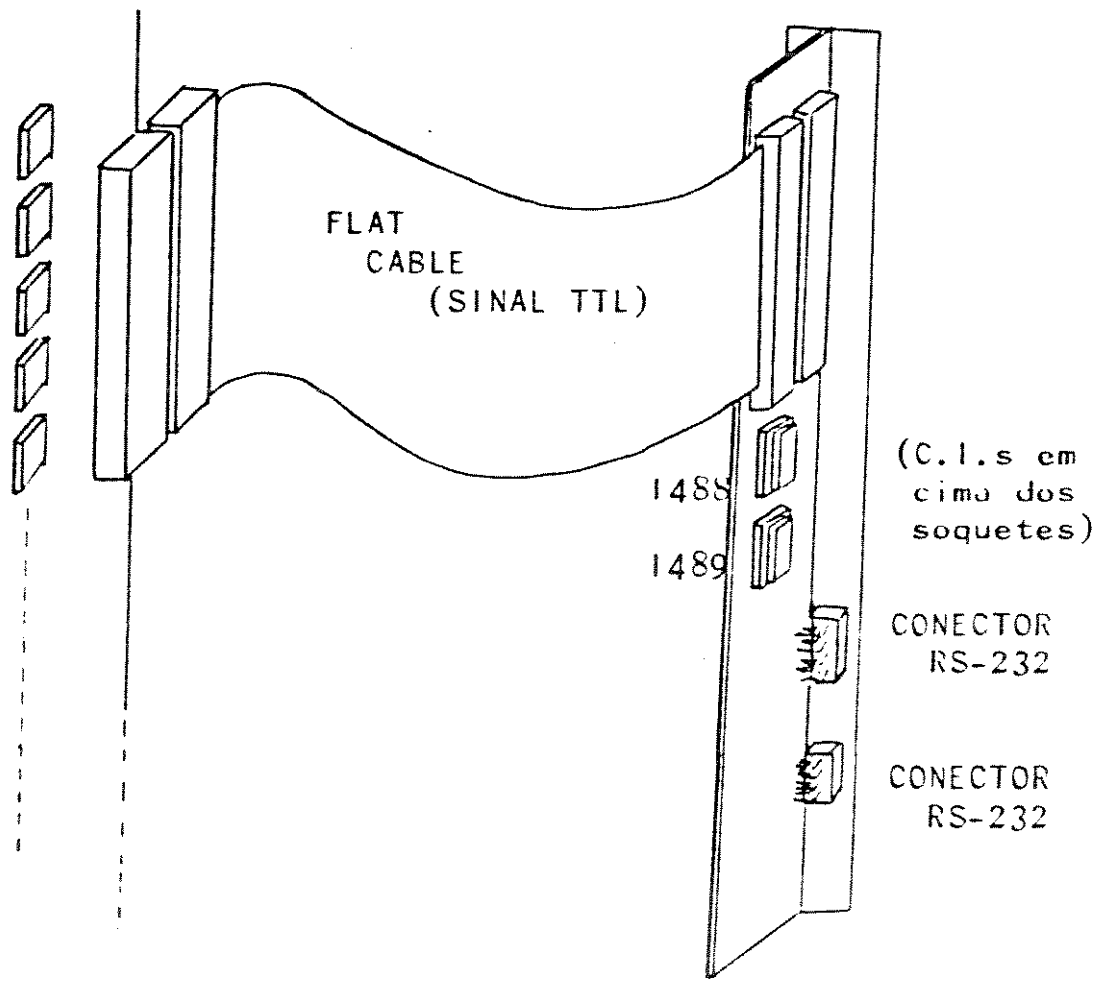


Fig B.3 – Placa de Interface Padrão RS-232

muito limitados e, assim, pensou-se em editar e compilar os programas num outro computador, chamado CADMUS, com mais recursos de software, e transferir através de linhas seriais para cada UCP do HOMUK, comutando-se através de chaves.

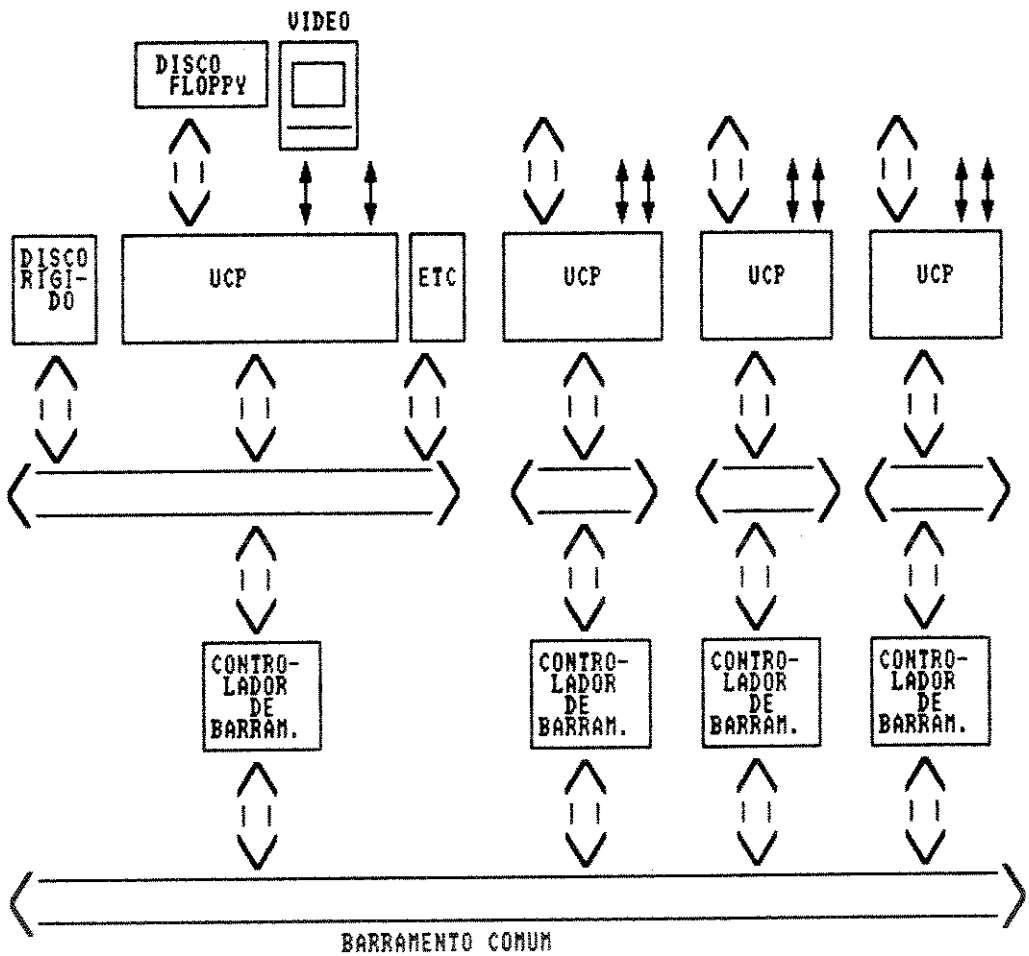


Fig. B.4 – Sistema multiprocessamento com barramento comum para comunicação

B.2 - Hardware e software do CADMUS

CADMUS é um sistema multiusuário baseado na UCP MC68010, com grande quantidade de recursos de hardware e software, disponível no mercado internacional.

Tendo software compatível com DEC Software e usando um sistema operacional MUNIX que é na verdade um UNIX Versão v.5 sob licença, com farta bibliografia disponível e mundialmente conhecida, este sistema operacional oferece, além de recursos, bastante software de apoio [Chri1 87,Chri2 87,PCS 84].

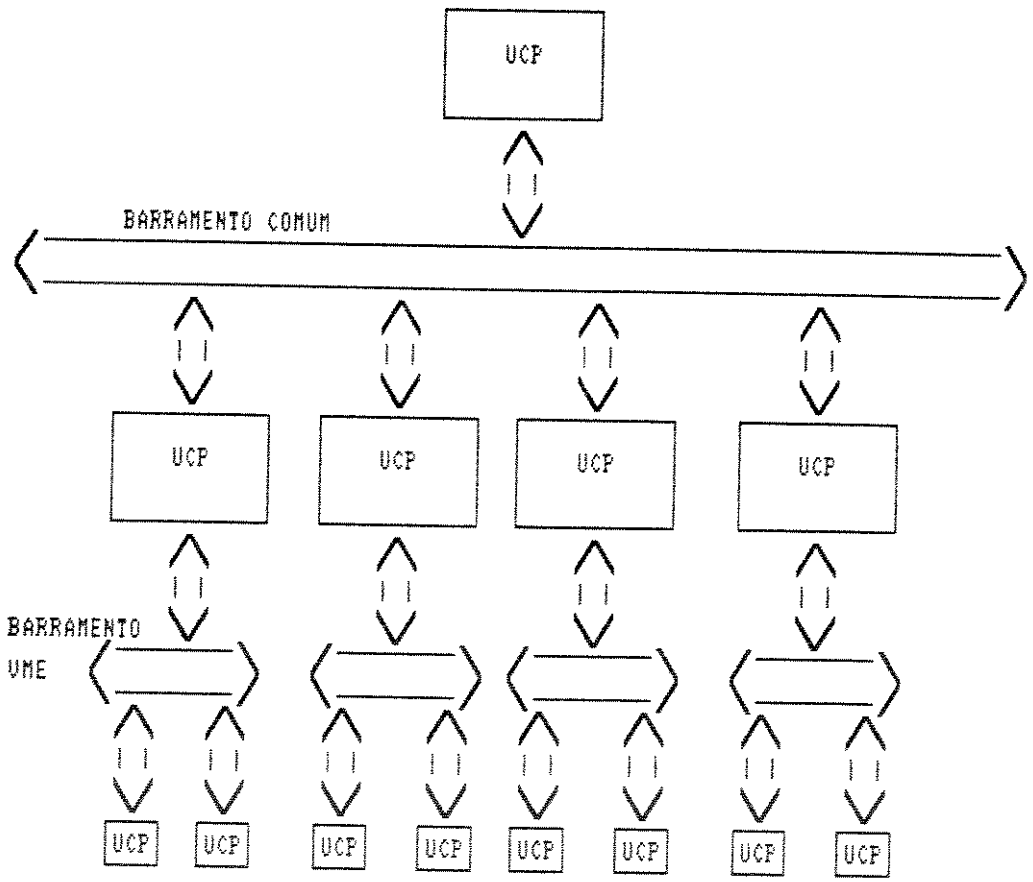


Fig. B.5 – Sistema com Estrutura Hierarquizada

As saídas das interfaces são eletrônica e mecânicamente compatíveis com os populares LSI-11, LSI-11/2 e LSI11/23 do DEC, permitindo interligar ampla gama de periféricos disponíveis no mercado[PCS 85, Prata 87].

Sendo o CADMUS uma máquina rica em software, ao contrário do HOMUK, achou-se conveniente ligar o CADMUS como um "host" do HOMUK, de tal forma que pudesse conciliar grande quantidade de software disponível no CADMUS com os preocessadores disponíveis no HOMUK, para que estes processadores pudessem compartilhar linguagens e programas que o CADMUS oferece[INM2 88, INT 81].

O CADMUS usa uma UCP MC68010 com 10MHz de relógio e 2Mbytes de memória,

CONTROLE DO BARRAMENTO VME	FF FFFF	RESEVADO PARA CONTROLE VME-BUS	ÁREA GLOBAL	HL Xddd
	FC FFFF			XH Xddd
	FB FFFF	RESERVADO PARA ENTRADA/SAÍDA		
	FA 7FFF	E/S LOCAL	ÁREA LOCAL	
	FA 0000	RESERVADO		
	F8 FFFF	EPROM DE 32kX16		
	F8 7FFF	EPROM DE 16KX16		
EPROM	F8 3FFF	EPROM DE 8KX16		
	F8 0007	VETOR RESET EPROM		
	F7 FFFF			
ACESSO A BARR.VME		ACESSO PARA O BARRAMENTO PADRÃO VME	ÁREA GLOBAL	XH Xddd
	0F FFFF			
		RAM DE 1MB		
	03 FFFF			
RAM 256K		RAM DE 256KB	ÁREA LOCAL	
	00 0007			
	00 0000	VETOR DE RESET		

OBS: X-IRRELEVANTE
ddd-FC0 a FC2

Tabela B.1 – Organização da memória no HOMUK

expansível até 4 Mbytes, tem memória virtual e possibilidade de implementar coprocessadores de ponto flutuante. Possui dois barramentos de dados, sendo que um deles é equipado com um controlador micro-programável, além de ADM, de tal forma que a UCP não precisa ficar esperando uma entrada ou saída de dados. O CADMUS tem a seguinte configuração: UCP MC68010 com relógio de 10MHz, MMU (Memory Management Unit) que ajuda no manuseio da memória possibilitando o uso da memória virtual, relógio em tempo real, interrupção vetorizada com 4 níveis de prioridades, 8 interfaces de entrada e saída serial (expansível para 16) todas reconfiguráveis para padrão RS-232, RS-422 ou RS-423 através de "jumpers". Tem compatibilidade mecânica e eletrônica com interfaces do LSI-11, LSI-11/2 e LSI-11/23 do DEC, taxa programável entre 150 a 38.400 bauds, interfaces disponíveis para ampla gama de controladores de discos rígidos, discos flexíveis, plotters, impressoras, impressoras laser, saídas para redes locais, tablados, etc. disponíveis no mercado mundial e fonte de alimentação que suporta uma queda de tensão de até 10 ms.

O esquema geral pode ser visto na figura B.6 .

Como foi mencionado anteriormente, o CADMUS trabalha com sistema operacional MUNIX (uma variante do UNIX v.5, sob licença) [Prata 86, Prata 85] com farta bibliografia e compatível com muitas máquinas, principalmente com o DEC Software, contando

com grande quantidade de software, como linguagens, programas de apoio e aplicativos diversos[PCS 84, PCS 85].

O MUNIX conta com duas versões: uma com memória virtual e outra com memória real. O uso da versão com memória virtual torna o sistema mais lento, de tal forma que o fabricante recomenda fazer o uso da versão com memória virtual somente quando necessário.

Conta com as seguintes linguagens: Assembler (a68:MIT), COBOL, PASCAL, FORTRAN 77, PROLOG, LISP, C, APL e SNOBOL.

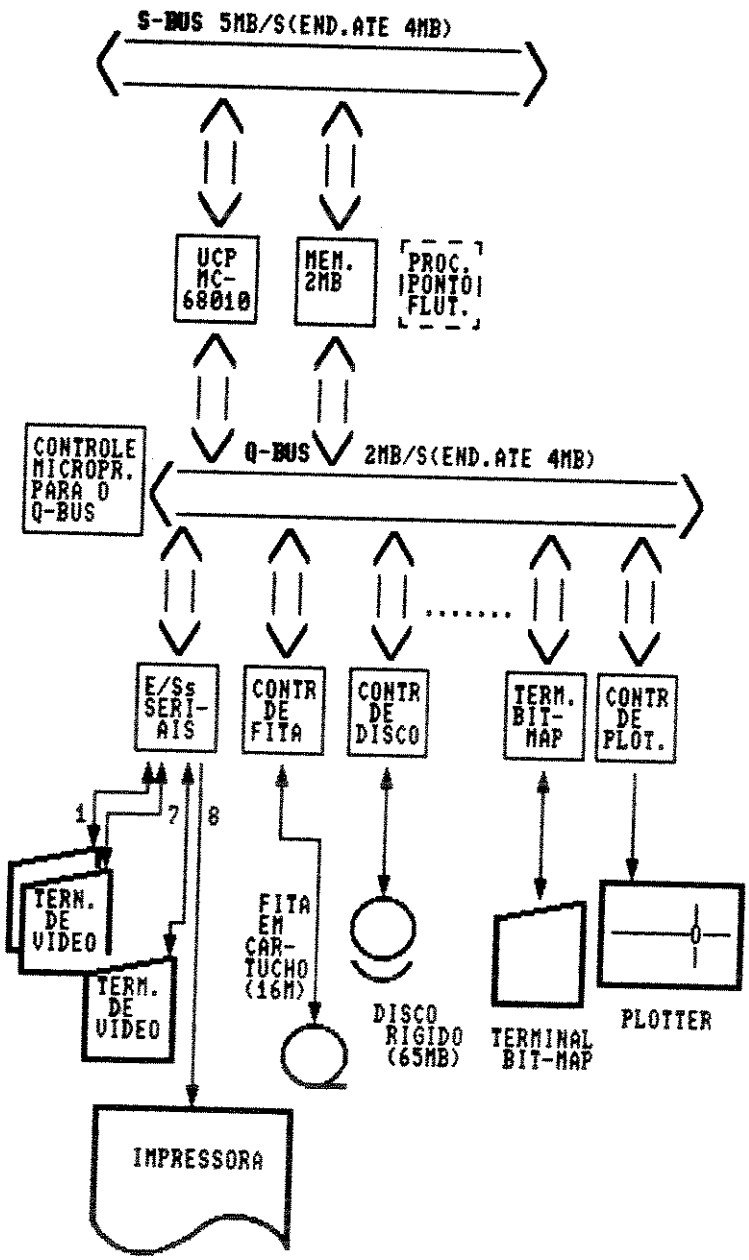


Fig. .6 – Esquema geral do CADMUS 9.200 com duplo barramento.

Programas Gráficos disponíveis: GKS, PLOT10, GRAPH, etc.

Editores de texto disponíveis: MED (editor orientado por tela), ED (editor orientado por linha), VI (editor orientado por tela), etc.

Aplicativos diversos disponíveis: manual residente, calendário, softwares de Comunicação (X25, Kermit, MUNIX/NET,etc.), criptografia, calculadora de mesa, mail (correspondência entre os usuários), etc.

O sistema operacional permite processamentos concorrentes para multiusuários, contando inclusive com comandos poderosos como o SHELL [PCS 85, Prata 86, Prata 85].

B.3 - Ligação CADMUS - HOMUK

Mesmo o HOMUK tendo como UCP um MC68000 da Motorola e o CADMUS a UCP MC68010 que, excluindo algumas instruções com memória virtual (que normalmente não são utilizadas no CADMUS), possuem todas as instruções compatíveis entre si, possibilitando a utilização de programas executáveis gerados pelo CADMUS no HOMUK, as duas máquinas são completamente incompatíveis em termos de sistemas operacionais, o que torna muito difícil a comunicação entre elas .

Poder-se-ia transferir os programas e os dados de várias formas, mas cada método apresentava suas vantagens e desvantagens, de tal forma que foi necessário fazer um estudo mais detalhado para decidir qual era o mais conveniente. Foram consideradas as seguintes opções:

i) Comunicar-se utilizando somente programação a nível de linguagem de máquina.
ii) Utilização de linguagem de máquina aproveitando-se também as subrotinas residentes na EPROM (o BIOS).
iii) Utilização de linguagem de máquina, mas explorando os recursos disponíveis nos sistemas operacionais.
iv) Utilização de linguagens de alto nível, de tal forma que as comunicações ficassem implícitas nos comandos de entrada e saídas de dados.
v) Fazer o uso de programas de comunicação prontos como o Kermit ou X-25.
vi) Utilização em ambos de um sistema operacional multi-UCP, onde a distribuição de tarefas estivesse implícita nos comandos e nas linguagens.

B.3.1 – Comunicação usando somente linguagens de máquina

Uma vez que os dois sistemas são diferentes, os programas devem ser distintos na parte de entrada e saída de dados. Poder-se-ia gerar subrotinas distintas para cada sistema, que cuidassem da entrada e saída de dados, para que o resto do programa continuasse sendo comum e compatível. Porém, além de trabalhoso, o fato de programar em linguagem de máquina priva de explorar a vantagem do CADMUS ser muito rico em software, pois aproveita-se muito pouco do que estava disponível. Pode-se pensar em misturar linguagens de alto nível chamando subrotinas em linguagens de máquina, mas a incompatibilidade de sistemas operacionais limita em muito os recursos disponíveis em linguagens de alto nível, pois estas costumam explorar constantemente os recursos oferecidos pelos sistemas operacionais.

Assim, o uso da linguagem de máquina mostrou-se viável mas bastante trabalhoso.

B.3.2 – Uso da linguagem de máquina com BIOS

Dentro das EPROMs residentes nas máquinas costuma haver subrotinas prontas de entradas e saídas de dados que o monitor utiliza, que podem ser usadas juntas com programas em linguagens de máquina, facilitando em muito a programação. Uma vez que as duas máquinas são incompatíveis em termos de sistemas operacionais, tanto o BIOS como as entradas e saídas de dados nas duas máquinas são diferentes, deve-se chamar de forma diferenciada mas, reduz em muito a tarefa de programação.

Quanto às comunicações entre as UCPs do HOMUK, são sempre compatíveis, pois possuem o mesmo BIOS.

B.3.3 – Uso da linguagem de máquina com recursos de S.Os

Utilizar as funções do sistema operacional não difere muito do caso anterior, mesmo porque em muitos casos o sistema operacional aproveita as subrotinas residentes nas EPROMs.

Caso os dois sistemas operacionais sejam compatíveis, o programa de um sistema pode ser transferido ao outro sistema com total compatibilidade. Porém, caso os sistemas operacionais sejam incompatíveis, torna-se muito difícil uma total portabilidade de uma máquina à outra. Neste último caso, a única forma de compatibilizar o programa de um sistema com outro seria ter o conhecimento das chamadas das subrotinas dos sistemas operacionais, adaptando-as ao novo sistema operacional[Blair 85].

B.3.4 - Comunicação utilizando linguagens de alto nível

Como as linguagens de alto nível costumam fazer uso de recursos oferecidos pelo sistema operacional, um programa compilado num sistema operacional não costuma ser executado em outros sistemas operacionais, a não ser que sejam compatíveis entre si.

Assim, uma das poucas formas de executar um programa de um sistema em outro, seria transferir o programa em forma de fonte e recompilá-lo no novo sistema operacional, tomando cuidado com algumas variações possíveis (ex: as entradas e saídas de dados podem ser diferentes). Para tal operação é necessário ter um programa de transferência de arquivos ou compatibilidade na formatação de discos flexíveis ou fitas.

No caso do CADMUS e HOMUK esta tranferência é bastante difícil pois, além de não ter programas de comunicação compatíveis, o CADMUS está equipado com fitas em cartuchos e discos rígidos, enquanto o HOMUK possui discos flexíveis e discos rígidos. Assim, não se pode transferir os dados nem via fita, nem via discos flexíveis, não tendo também programas de comunicação serial compatível.

Mesmo que consiga transferir os programas fonte, como os únicos compiladores que o HOMUK tem disponível, o FORTRAN e assembler, o CADMUS tem muito pouco em que ajudar o HOMUK.

Certas linguagens, como a linguagem C, foram projetadas tendo como preocupação maior a portabilidade e facilidade de controle do sistema. Nesse caso pode-se pensar, ainda, em portabilidade, contanto que não se faça uso de certos "includes" que acionam as entradas e saídas de dados ou recursos do sistema operacional(como o acesso a disco), uma vez que o HOMUK não dispõe de compilador da linguagem C.

B.3.5 - Uso de programa de comunicação

Muitos programas de comunicação com complexidade e sofisticação variadas foram feitos para diversos sistemas. Caso se tivesse um destes programas disponíveis em ambas as máquinas, se transferiria com certa facilidade o programa de um sistema para o outro.

O CADMUS é muito rico neste tipo de software, tendo quase todos os programas de comunicação mais populares nos dias de hoje. Porém, o HOMUK dispõe somente de um simples programa de "DUMP" de memória, próprio dele, não permitindo explorar os recursos oferecidos pelo CADMUS.

Foram feitos alguns estudos para o uso do Kermit [Prata 87] e constatou-se que a fonte deste programa foi feita em C, usando-se vários "includes" de tal forma que não foi possível implementar no HOMUK. Aliás, a maioria dos programas de comunicação está sendo feita em linguagem C ou em assembler. No primeiro caso tem-se alguns programas fontes mas não se tem compilador no HOMUK e, no segundo caso, exigiria um trabalho bastante árduo de adaptação ao novo sistema.

B.3.6 - Utilização de sistemas operacionais ou linguagens multi-UCPs

O HOMUK possui um sistema operacional chamado Flexos Operating System com recursos gráficos incluindo o uso de "workstation GKS" para multiprocessador. Para que isso funcione, porém, antes de mais nada deve estar funcionando o circuito de barramento comum, mas o barramento comum mostrou-se pouco confiável trabalhando com este sistema.

Quanto a linguagens de multiprocessamento, existem algumas linguagens desenvolvidas recentemente para alguns sistemas, mas não estão disponíveis nem no HOMUK nem no CADMUS. Portanto, o uso de tais recursos de software pode ser considerado inviável no momento.

B.3.7 - Conclusão

Após a análise destes dados chegou-se à conclusão de que o processamento paralelo, ou o multiprocessamento usando o CADMUS como "host" do HOMUK, seria muito ineficiente e bastante restrito em termos de aplicação. Para superar estas dificuldades, ter-se-ia que investir brutalmente em software básico, o que não era o objetivo deste trabalho. Na época do estudo da utilização do CADMUS como "host" do HOMUK, surgiram no Brasil os primeiros equipamentos com "Transputers" e linguagens de alto nível para multiprocessamento. Devido às dificuldades acima descritas, em comparação com os recursos já disponíveis nas placas de transputers, resolveu-se continuar o trabalho em equipamentos com Transputers.

Apêndice C
PROGRAMAS TESTES

Para analisar o potencial do uso de variáveis comuns na linguagem C paralelo versão 2.0 da companhia 3L Ltda[46] e linguagem OCCAM do TDS versão 2.0 IMS 701 C do INMOS , foram realizados vários testes, que podem ser sintetizados nos quatro programas testes apresentados a seguir. Todos os programas utilizam variáveis comuns, mas o primeiro e o terceiro, manipulam tais variáveis não concorrentemente, isto é, quando uma variável está sendo acessada por um processo, outros processos não fazem uso da mesma. No segundo e no quarto, mais de um processo acessa variáveis comuns concorrentemente. Os programas e os respectivos resultados são apresentados a seguir:

Programa 1

```
#include<stdio.h>                    /* carrega e/s padrão */
#include<thread.h>                   /* carrega recursos de thread */
#include<sema.h>                     /* instala semáforos */

int sinc1, resposta;
SEMA *sem1, *sem2;                   /* teremos dois semáforos */

main()                               /* programa principal */
  {
    int i, j;
    extern void proc1();
    sema_init(sem1,0);               /* semáforo sem1 fechado */
    sema_init(sem2,0);               /* semáforo sem2 fechado */
    i=1; sinc1=0;
    thread_create(proc1,1024,0);     /* ativa o thread proc1 */
    i=1; sinc1=0;
    printf("Iniciando o programa principal \n");
    while (i<=4)
      {
      printf("Tecle o número do processo");
      i=getnum;
      j=0;
      sinc1=i;
      sema_signal(&sem1);            /* libera o sem1 */
      sema_wait(&sem2);              /* espera o sem2 ser liberado */
      printf("Enviei mensagem para proc1 e recebi %d\n",resposta);
```

```
    }
  }

void proc1()                          /* processo em paralelo */
{
  while (sinc1<4)
    {
    sema_wait(&sem1);                 /* espera sem1 ser liberado */
    resposta=sinc1;
    sema_signal(&sem2);               /* libera sem2 */
    }
}

getnum()
{
  char s[80];
  get(s);
  return(atoi(s));
}
```

O programa1 funcionou perfeitamente. Este programa, embora faça uso de variáveis comuns, só executa um processo por vez.

Inicialmente, o programa principal ajusta as variáveis sem que o "thread" proc1 seja ativado. Logo em seguida, o proc1 é ativado, mas ele fica esperando o semáforo sem1 ser liberado. O processo principal lê o dado do teclado e, em seguida, ao mesmo tempo em que libera o semáforo para ativar o proc1, ele para, esperando que o proc1 libere o semáforo sem2. O proc1 libera o semáforo sem2 e pára, esperando que o processo principal libere o semáforo sem1, e assim por diante, até que o usuário tecle o valor maior ou igual a 4, quando os dois processos terminam.

Em outras palavras, no Programa 1 os dois processos funcionam alternadamente de tal forma que nunca disputam a posse das variáveis comuns.

Progama 2

```
#include<stdio.h>                     /* carrega e/s padrão */
#include<thread.h>                    /* carrega recursos de thread */

int sinc1, resposta;

main()                                /* programa principal */
  {
    int i, j;
    extern void proc1();
    i=1; sinc=0                       /* ajusta as variáveis */
    thread_create(proc1,1024,0);      /* ativa o thread e prossegue */
    printf("Iniciando o programa principal \n");
```

```
    while (i<=4)
      {
      printf("Tecle o número do processo");
      i=getnum;
      j=0;
      if (i==1)
        {
        sinc1=1;
        while (sinc1==1) j++
        printf("Enviei para proc1 e recebi %d\n",resposta);
        }
      }
    }

void proc1()                              /* thread em paralelo */
{
  while (sinc1<4)
    {
    if (sinc1==1)
      {
      resposta = 1;
      sinc1 = 0;
      }
    }
}

getnum()
{
  char s[80];
  get(s);
  return(atoi(s));
}
```

O Programa 2 também trabalha com dois processos, porém com a diferença de que os dois são ativados ao mesmo tempo. Nestas condições, o programa mal inicia a execução e o Tansputer já bloqueia, sem sequer transmitir a primeira mensagem. Como se pode ver, dois processos com variáveis comuns ativados ao mesmo tempo não são permitidos.

Dick Pountain, em uma recente publicação na revista BYTE (janeiro de 1990), traz resultados de testes com a nova linguagem Par.C [36]. Segundo o autor, esta linguagem melhorou em alguns pontos. Por exemplo, agora a nova versão possui o comando "alt" igual ao OCCAM, o que aumenta o potencial de uso. Vem também equipado com o comando "par", que simplifica a geração e visualização de processos paralelos. Por outro lado, o mesmo artigo alerta que esta nova versão continua fazendo péssimo aproveitamento de memória pois carrega todos os processos em todas as UCPs e ativa aqueles que cada UCP

utiliza. Por outro lado, segundo anúncios publicitários do fabricante, esta nova versão possui um depurador, o que vem resolver alguns problemas.

A conclusão a que se chega é que, mesmo na versão mais nova, muitas das limitações detectadas nos testes ainda permanecem.

Foram apresentados anteriormente os programas testes de variáveis comuns para a linguagem C paralelo. Testes na mesma linha foram feitos para o OCCAM e os resultados podem ser sintetizados nos dois programas (Programa3 e Programa4) comentados em seguida.

Como no outro teste, o primeiro programa usa variáveis comuns que são tratadas não concorrentemente, e o segundo, tenta tratar concorrentemente. Os resultados, em ambos os casos, são muito semelhantes aos obtidos com a linguagem C paralelo.

Programa 3

```
                                        --indica comentários
#USE userio                             --carrega biblioteca de e/s
INT x1,x2,x3,x4,y1,y2,y3,y4,kchar:      --declara inteiros
PAR                                     --processos serão paralelos
  SEQ                                   --processo principal sequencial
    x1:=0
    WHILE x1 < 4
      SEQ
        read.char(keyboard,kchar)
        read.int(keyboard,x1,kchar)     --lê valor de x1 do teclado
        IF
          x1 = 1                        --se x1 for 1
            SEQ
              saida1 ! x1               --envia x1 para esc1
              entra1 ? y1               --recebe y1 do esc1
              write.int(screen,y1,4)    --e coloca na tela
          x1 = 2                        --se x1 for 2
            SEQ
              saida2 ! x1               --envia x1 para esc2
              entra2 ? y1
              write.int(screen,y1,4)
          x1 = 3                        --se x1 for 3
            SEQ
              saida3 ! x1               --envia x1 para esc3
              entra3 ? y1
              write.int(screen,y1,4)
          x1 >= 4                       --se x1 for maior ou igual a 4
            SEQ
              write.full.string(screen,"Terminando esc1*c*n")
              saida1 ! x1               --envia x1 para esc1 (acaba)
              entra1 ! y1               --espera resposta
```

```
            write.full.string(screen,"Terminando esc2*c*n")
            saida2 ! x1              --envia x1 para esc2
            entra2 ! y1              --espera resposta
            write.full.string(screen,"Terminando esc3*c*n")
            saida3 ! x1              --envia x1 para esc3
            entra3 ! y1              --espera resposta
  write.full.string(screen,"Terminando o Principal*c*n ")
  read.char(keyboard,kchar)
SEQ
  x2:=1                              --processo esc1
  WHILE x2 = 1                       --enquanto x2 = 1
    SEQ
      saida1 ? x2                    --lê novo valor de x2
      y2 := 11
      entra1 ! y2                    --e responde ao principal
SEQ
  x3:=2                              --processo esc2
  WHILE x3 = 2
    SEQ
      saida2 ? x3
      y3 := 22
      entra2 ! y3
SEQ
  x4:=3                              --processo esc3
  WHILE x4 = 3
    SEQ
      saida3 ? x4
      y4 := 33
      entra3 ! y4
```

O programa acima tem 4 processos em paralelo com nomes de principal, esc1, esc2 e esc3. O processo principal lê o número do teclado e envia para os respectivos processos; se for digitado um 1 é enviado 1 para o esc1, que responderá para o processo de nome principal com valor 11 listado na tela, e assim por diante. Caso o número seja maior ou igual a 4, o principal vai enviar este número para o primeiro processo que, antes de terminar, vai responder ao principal. Enquanto não vier a resposta, o processo principal fica esperando pela resposta. Em seguida faz-se o mesmo tratamento com o esc2 e com o esc3.

Como pode ser visto neste programa, os processos funcionam alternadamente. Em nenhum momento mais de um processo entra em ação ao mesmo tempo, a não ser os comandos de sincronismo.

Em seguida é mostrado o que ocorre ao serem acionados simultaneamente mais de um processo.

Programa4

```
                                        --indica comentários
#USE userio                             --carrega biblioteca de e/s
CHAN OF INT entra1,entra2,entra3,saida1,saida2,saida3:
INT x1,x2,x3,x4,y1,y2,y3,y4,kchar:      --declara inteiros
PAR                                     --processos serão paralelos
  SEQ                                   --processo principal sequencial
    x1:=0
    WHILE x1 < 4
      SEQ
        read.char(keyboard,kchar)
        read.int(keyboard,x1,kchar)     --lê valor de x1 do teclado
        IF
          x1 = 1                        --se x1 for 1
            SEQ
              saida1 ! x1               --envia x1 para esc1
              entra1 ? y1               --recebe y1 do esc1
              write.int(screen,y1,4)    --e coloca na tela
          x1 = 2                        --se x1 for 2
            SEQ
              saida2 ! x1               --envia x1 para esc2
              entra2 ? y1
              write.int(screen,y1,4)
          x1 = 3                        --se x1 for 3
            SEQ
              saida3 ! x1               --envia x1 para esc3
              entra3 ? y1
              write.int(screen,y1,4)
          x1 >= 4                       --se x1 for maior ou igual a 4
            SEQ
              saida1 ! x1               --envia x1 para esc1 (acaba)
              saida2 ! x1               --envia x1 para esc2
              saida3 ! x1               --envia x1 para esc3
    write.full.string(screen,"Terminando o Principal*c*n ")
    read.char(keyboard,kchar)
  SEQ
    x2:=1                               --processo esc1
    WHILE x2 = 1                        --enquanto x2 = 1
      SEQ
        saida1 ? x2                     --lê novo valor de x2
        y2 := 11
        IF
          x2 = 1                        --se x2 = 1
            entra1 !y2                  --responde ao principal
          TRUE
```

```
          SKIP
SEQ
  x3:=2                              --processo esc2
  WHILE x3 = 2
    SEQ
      saida2 ? x3
      y3 := 22
      IF
        x3 = 2                       --se x3 = 2
          entra2 ! y3                --responde
        TRUE
          SKIP
SEQ
  x4:=3                              --processo esc3
  WHILE x4 = 3
    SEQ
      saida3 ? x4
      y4 := 33
      IF
        x4 = 3                       --se x4 = 3
          entra3 ! y4                --responde
        TRUE
          SKIP
```

Este programa é parecido com o anterior, exceto em um dos detalhes: o programa principal ao receber dado maior ou igual a 4 tenta terminar todos os processos ao mesmo tempo. Entretanto, durante a sua execução, ao receber mensagem maior ou igual a 4, interrompeu a execução e emitiu a mensagem:

"Transputer system error flag set"

Ao fazer uso do depurador, o mesmo simplesmente apresentava no vídeo a seguinte mensagem:

"Error: Location not in program or a library"

Inserindo-se mensagens ao longo do programa, constatou-se que o problema ocorre no exato momento em que o programa principal envia mensagem ao esc1 e tenta prosseguir a execução em paralelo.

O teste mostra que, embora o "constructor" seja denominado PAR no OCCAM, os processos não conseguem utilizar variáveis comuns concorrentemente, o que limita significativamente o potencial de multiprocessamento, pois as variáveis comuns são importantes recursos de comunicação entre os processos.

APÊNDICE D

O disquete que acompanha este trabalho contém as últimas versões dos programas que implementavam os algoritmos relativos ao problema do "caixeiro viajante", a "determinação de tautologia" e o Problema de "mochila" ("knapsack").

É importante frisar que estes programas não funcionaram integralmente, porque foi através deles que se verificou a falta de recursos das linguagens em relação à implementação dos procedimentos de redistribuição dinâmica de trabalho. Os testes demonstrativos desta falta de recursos foram apresentados no Capítulo 4 e apêndice C.

Seque-se a relação de arquivos.

1 – MOCHILA.CFG : Configurador do Programa Mochila em C paralelo.
2 – MOCHILAM.C : Programa mestre que roda no processador principal do Programa Mochila em C paralelo.
3 – MOCHILAE.C : Programa escravo que roda nos processadores auxiliares do Programa Mochila em C paralelo.
4 – TAUTOLOG.CFG: Configurador do Programa Determinação de Taulologia em C paralelo.
5 – TAUTOLM.C : Programa mestre que roda no processador principal do Programa Determinação de Tautologia em C paralelo.
6 – TAUTOLE.C : Programa escravo que roda nos processadores auxiliares do Programa Determinação de Tautologia em C paralelo.
7 – CAIXEIRO.CFG: Configurador do Programa Caixeiro Viajante em C paralelo.
8 – CAIXM.C : Programa mestre que roda no processador principal do Programa Caixeiro Viajante em C paralelo.
9 – CAIXE.C : Programa escravo que roda nos processadores auxiliares do Programa Caixeiro Viajante em C paralelo.
10 – CAIXEIRO.OC : Programa Caixeiro Viajante escrito em OCCAM usando TDS.
11 – TAUTOLOG.OC : Programa Determinação de Tautologia escrito em OCCAM usando TDS.

REFERÊNCIAS BIBLIOGRÁFICAS

[Alma 89] ALMASI, G.; Gottlieb, A.; *Highly Parallel Computing*, Bejamin/Cummings Publishing Co. Inc., New York, **1989**.

[Asbu 85] ASBURY, Ray; FRISON, Steven G.; ROTH, Thomas, *Concurrent Computer Ideal for Inherently Parallel Problems*, Computer Design, September 1, **1985**, pp. 99–107.

[Ben 82] BEN ARI, M.; *Principles of Concurrent Programming* , Prentice-Hall Intenational Inc., **1982**.

[Berge 68] BERGE, C.; *Principes de Combinatoire* Dunod, Paris **1968**, pp. 91–123.

[Birk 67] Birkhoff, G.; Bartee Thomas C.; *Modern Applied Algebra*, Mc Graw-Hill, New York, **1970**.

[Blair 85] BLAIR, Gordom S., MALONE, Mariani; *A Critique of Unix Software*, Practice and Experience, Vol 15, December **1985**, pp. 1125–1139.

[Booth 67] BOOTH, Taylor L.; *Sequencial Machine and Automata Theory*, John Willey & Sons, 1967.

[Burns 88] BURNS, Alan; *Programming in OCCAM 2*, Addison-Wesley Publis. Co., Workingham England, **1988**.

[Chris1 87] CHRISTIAN, Kaare; *Entendendo o Sistema UNIX*, Vol.1, Editora Campos Ltda., Rio de Janeiro, **1987**.

[Chris2 87] CHRISTIAN, Kaare; *Tópicos Avançados do Sistema UNIX*, Vol.2, Editora Campos Ltda., Rio de Janeiro, **1987**.

[DeCe 89] DeCegama, A.L.; *The Technology of Parallel Processing* Prentice-Hall International Inc., Englewood Cliffs, **1989**.

[Demi 89] DEMIRALP, S. Vedat; *Parallel Processing with OCCAM and Transputer*, Texto do Seminário ETHOS em Florianópolis em 14 a 16 de Julho de **1989**.

[Dun 90] DUNCAN, Ralph; *A Survey of Parallel Computer Architectures*, Computer, IEEE Computer Society, February **1990**, Vol. **23**, no. **2**.

[Elt 83] ELTEC, ELEKTRONIC GmbH; *Eltec 68K — System*, July **1983**.

[Enca 84] ENCARNAÇÃO, J., GOBEL, M., LINDNER, ING. R.; *HOMUK - Ein Homegener Multiprozessorkern für Verteilte Graphische Systeme*, Technische Hochschule Darmstadt, Fachbereich Informatik, June **1984**.

[Fox 88] FOX, Geoffrey C.; *Solving Problems on Concurrent Processors General Techniques and Regular Problems*, vol.1, Prentice-Hall International Inc., **1988**.

[Furu 84] FURUYA, N.; *PAE — Processador de Auxílio ao Ensino*, tese de Mestrado apresentado ao Instituto de Ciências Matemáticas de São Carlos, da Universidade de São Paulo, em dezembro de **1984**.

[Furu 89] FURUYA, N.; *Experience in parallel C*, Trabalho apresentado no congresso OCCAM USER GROUP, Florianópolis - Sta Catarina, **1989**.

[Gaj 85] GAJSKI, Daniel, D., PEIR, Jih Kwon; *Essential Issues in Multiprocessor Systems*, Computer, June **1985**, pp. 9–27.

[Gio 86] GIOZZA, W. Ferreira, ARAUJO, Jose F. M., MOURA, José A.B., SAUVÉ, Jacques Philippe; *Redes Locais de Computadores*, McGraw-Hill, São Paulo, **1986**.

[Glush 65] Glushkov, V.; *Automata Theory and Formal Microprogram Transformation*, Cibernetica, Voll, pp.1-9, **1965**.

[Hoare 78] HOARE, C.A.R.; *Communicating Sequential Process*, Communicating of the ACM., August **1978**; vol 21, no. 8, pp. 666–677.

[Hoare 85] HOARE, C.A.R.; *Communicating Sequential Process*, Prentice-Hall International Inc., UK, **1985**.

[Hwang 85] Hwang, K.; Briggs F.A.; *Computer Architecture and Parallel Processing*, Mc Graw-Hill, New York, **1985**.

[INM1 87] INMOS Corp.; *IMS T414 transputer*, Manual do INMOS Corp., **1987**.

[INM1 88] INMOS Corp.; *IMS T800 transputer*, Manual do INMOS Corp., **1988**.

[INM3 88] INMOS Corp.; *OCCAM 2 Reference Manual*, Prentice Hall, New York, **1988**.

[INM4 89] INMOS Corp.; *The Transputer Application Notebook*, Manual do INMOS Corp., Redwood Burn Limited, Trowbridge, **1989**.

[INM5 89] INMOS Corp.; *The Transputer Databook*, Manual do INMOS Corp., Bristol, UK, **1989**.

[INM6 89] INMOS Corp.; *The Transputer Databook and iq Systems Databook*, Manual do INMOS Corp., Bristol, UK, **1989**.

[INM7 88] INMOS Corp.; *TRANSPUTER DEVELOPMENT SYSTEM*, Prentice Hall, New York, **1988**.

[INT 81] INTEL Corp.; *Microprocessor Component Catalog*, Manual do Intel publicado em **1981**.

[Jose 85] JOSEPH, Mathai, PRASAD,V.R., NATARAJAN,N.; *A Multiprocessor Operating System*, Prentice/Hall International, Rio de Janeiro, **1985**.

[Kram 87] KRAMER J., MAGEE J., SLOMAN M.; *The Conic Toolkit for Building Distributed Systems*, IEEE PROCEEDINGS, Vol 134, nº 2, March **1987**.

[Lang 81] LANGDON, Glen G.Jr, FREGNI, Edson; *Projeto de Computadores Digitais*, Editora Edgard Blücher Ltda, 2a edição **1981**.

[May 90] May David; *Towards General Purpose Parallel Computers*, Folha solta, 5 de Abril **1990**.

[May&T 90] May, David; Thompson, Peter; *Transputers and Routers: Components for Concurrent Machines*, Folha solta, 9 de Maio de **1990**.

[VME 85] Manual do VME (folha solta).

[Michel 89] MICHEL, Gien; *Arquitecture des Systemes Répartis — Une Nouvelle Génération d' UNIX*, Texto do Seminário ETHOS em Florianópolis em 14 a 16 de Julho de **1989**.

[Nico 89] NICOLE, Denis A.; LLOYD, E. Keith; *Switching Networks for Transputer Links*, Texto do Seminário ETHOS em Florianópolis, de 14 a 16 de Julho de **1989**.

[Nije 78] NIJENHUIS, Albert; WILF, Herbert S.; *Combinatorial Algorithms for Computer and Calculators*, Academic Press, New York, **1978**, pp. 666–677.

[Parker 83] PAKER,Y.; *Multi-Microprocessor Systems*, Academic Press Inc., New York, **1983**.

[PCS 84] PCS – Periphere Computer System Gmbh; *MUNIX MANUAL*, Munique, **1984**.

[PCS 85] PCS – Periphere Computer System Gmbh; *MUNIX-MUE INTRODUCTION*, Munique, **1985**.

[Pease 77] PEASE, Marshall; *The Indirect Binary N-Cube Microprocessor Array*, IEEE Transaction on Computers, May **1977**, Vol. 26, no. 5, pp. 458–473.

[Peck 62] PECK J.E.L., SCHRACK; *Algorithm 86 Permute*, Communication of the ACM, April **1962**, Vol. 5, no.4, pp. 208–210.

[Polla 85] POLLARA F.; *Concurrent Viterbi Algorithm on a Hipercube*, Caltech Report, C3, pp. 208, **1985**.

[Pou1 90] POUNTAIN, Dick; *Configuring Parallel Programs*, Byte, Mc Graw Hill Publication, Jan. **1990**, pp. 327–334.

[Pou2 90] POUNTAIN, DICK; *Virtual Channels: The Next Generation of Transputers*, Byte, Mc Graw Hill Publication, April **1990** pp. 53–64.

[Prata 86] PRATA, Stephem; *ADVANCED UNIX- A PROGRAMMER'S GUIDE*, 2a edição, Howard W.Sams & Co, Indianapolis **1986**.

[Prata 87] PRATA, Stephem; *UNIX COMMUNICATION*, Howard W. Sams Co, Indianapolis *1987*.

[Prata 85] PRATA, Stephem, WAIDE, Martin; *UNIX SYSTEM V PRIMER*, Howard W.Sams Co., Indianapolis **1985**.

[Read 72] READ, Ronald C.; *Graph Theory and Computing* , Academic Press, **1972**.

[Seits 85] SEITZ, Charles L.; *The Cosmic Cube*, Communication of the ACM, Vol 28, no. 1, Jan. **1985**, pp. 22–33.

[Soki 78] SOKI, Nobuo et alii; *"Microcomputadores e suas utilizações " (em japonês)*, Sociedade Científica de Transmissão Eletrônica, 2a edição, Tokio, Japão, Julho de **1978**.

[Thay 84] THAYSE, A.; *Functions and Boolean Matriz Factorizarion*, Lectures Notes in Computer Science, pp.175, Spring-Verlag, **1984**.

[Taka 88] TAKAHASHI, Tadao; *Introdução a Programação Orientada a Objetos*, Ebai, Rio de Janeiro, **1988**.

[Trot 62] TROTTER, H.F.; *Algorithm 115 PERM*, Communication of the ACM, Vol. 5, no. 8, **1962**, pp. 434–435.

[Wu 84] Wu, C.; Feng T.; *Interconnection Networks for Parallel and Distributed Processing*, The Corrnputer Society of the IEEE, Piscataway, N.J., **1984**.

[3L 88] 3L Ltd; *Parallel C User Guide*, Manual 3L, Scotland, **1988**.