Projeto e Implementação de um Processador Virtual de Comunicação

Lúcia Maria de Assumpção Drummond

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

Aprovada por:

Prof. Valmir C. Barbosa, Ph. D.
(presidente)

Prof. Edil Severiano T. Fernandes, Ph. D.

Prof. Júlio Aude Salek, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
AGOSTO DE 1990

DRUMMOND, LÚCIA MARIA DE ASSUMPÇÃO

Projeto e Implementação de um Processador Virtual de Comunicação [Rio de Janeiro] 1990

V, 77 p., 29.7 cm, (COPPE/UFRJ, M. Sc., ENGENHARIA DE SISTEMAS E COMPUTACÃO, 1990)

TESE – Universidade Federal do Rio de Janeiro, COPPE

1 – Processador 2 – Comunicação 3 – Sistemas Distribuídos

I. COPPE/UFRJ II. Título(Série).

A meus pais

Agradecimentos

Ao meu orientador Valmir, por quem tenho grande admiração, pelos exemplos de competência e dedicação, tão importantes para o meu amadurecimento acadêmico e profissional.

A meus pais, Lucinda e Francisco, principalmente, por terem estado sempre ao meu lado, me ensinando e me ajudando a lutar pela realização dos meus ideais.

À Cristina, pelos exemplos, apoio e amizade, fundamentais para realização do mestrado.

Ao Dimas e à Suzy pelo apoio e pela influência que tiveram na minha formação profissional, através de seus exemplos de dedicação e amor ao trabalho.

Ao meu primeiro professor de Sistemas Operacionais Nery pelo incentivo para realização do mestrado e pelas sugestões feitas na leitura da tese.

À Anna, Cláudia, Clícia, Delfim , Maria Cláudia e Rose pelo companheirismo demonstrado em todos os momentos em que trabalhamos juntos e pelas horas alegres e de descontração.

Ao Ricardo Citro pelo incentivo e sugestões feitas neste trabalho.

Aos colegas da UFF, que sempre me incentivaram e me apoiaram na realização do mestrado.

A todos que trabalham no NCP e na secretaria da COPPE/SISTEMAS por terem estado sempre prontos a ajudar quando necessário.

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

Projeto e Implementação de um Processador Virtual de Comunicação

Lúcia Maria de Assumpção Drummond

Agosto de 1990

Orientador: Valmir Carneiro Barbosa

Programa: Engenharia de Sistemas e Computação

Sistemas distribuídos do tipo MIMD para processamento paralelo representam, atualmente, uma forte tendência na área de arquitetura de computadores, como a grande quantidade de máquinas deste tipo no mercado indica. Contrariamente aos sistemas fortemente acoplados, onde o custo de comunicação entre processos resulta da disputa pelos mecanismos de interconexão, os sistemas distribuídos exigem que os processos alocados em processadores diferentes se comuniquem somente por troca de mensagens nos canais de comunicação. O custo de comunicação em sistemas distribuídos para processamento paralelo é comparável ao de processamento, o que sugere a necessidade de utilização de um processador especializado para executar funções relacionadas a comunicação.

O objetivo deste trabalho é apresentar o projeto e a implementação em *software* de um processador que executa operações de comunicação que são consideradas blocos de construção de algoritmos distribuídos. Dentre estas funções tem-se: troca de mensagens entre processadores, roteamento de mensagens e várias formas de difusão de mensagens largamente utilizadas em algoritmos distribuídos.

A implementação deste Processador Virtual de Comunicação foi feita na linguagem *Occam*, utilizando-se uma rede de *Transputers* para testes.

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

Project and Implementation of a Communication Virtual Processor

Lúcia Maria de Assumpção Drummond

August, 1990

Thesis Supervisor: Valmir C. Barbosa

Department: Programa de Engenharia de Sistemas e Computação

Distributed systems of the MIMD kind for parallel processing represent, today, a significant trend in computer architecture. Strong evidence of this fact is the increasing number of these machines currently available. Differently from strongly coupled systems, where the dispute for interconnection mechanisms increases communication costs, distributed systems require that processes allocated in different processors only communicate with each other by exchange of messages on the communication channels. The cost of communication in distributed systems for parallel processing is comparable to that of processing, thus recommending the use of specialized processors to perform functions related to communication.

This work presents the project and the *software* implementation of a processor that performs some communication operations, which can be seen as building blocks for distributed algorithms. Among these functions one finds: exchange of messages between nodes, message routing and several forms of group communication widely used in distributed algorithms. The implementation of this Communication

Virtual Processor was written in *Occam* language and the testes were performed on a Transputer network.

Índice

Lista de Figuras

Capítulo I

Introdução

Existe atualmente uma tendência de pesquisa e desenvolvimento de *hardware* para auxiliar a comunicação em máquinas paralelas do tipo MIMD com computação totalmente distribuída. Neste contexto, são apresentados nesta tese dois modelos de processadores de comunicação que visam melhorar o desempenho de tais máquinas. A seguir, são abordados alguns pontos relevantes para a apresentação do assunto.

I.1 Multicomputadores

Atualmente, está havendo um considerável aumento de interesse na área de processamento paralelo, sendo colocadas muitas máquinas com este tipo de processamento no mercado. Este interesse resultou da necessidade de se executar tarefas mais sofisticadas que exigem um melhor desempenho no tempo de processamento.

As máquinas do tipo MIMD fracamente acopladas (multicomputadores) têm merecido papel de destaque dentre os projetos de pesquisa nas universidades e nos centros de pesquisa. Estas são consideradas computadores paralelos de propósito geral que podem fornecer *speedups* para um conjunto diverso de aplicações. Dentre as razões para o desenvolvimento de multicomputadores, atualmente, pode-se citar: custo moderado, surgimento de poderosos microprocessadores (capazes de apresentar desempenho comparável a de um supercomputador se forem arranjados e programados para operar em conjunto) e modularidade, que possibilita a expansão incremental da arquitetura através da adição de módulos, permitindo, assim, que

aplicações pertencentes a diversas faixas de demanda computacional sejam atendidas a partir de um mesmo produto.

Algumas características presentes nos multicomputadores, que permitem distingui-los de outras organizações de processamento paralelo, conforme [13], são:

- Grande número de processadores, já que é economicamente viável projetar sistemas com centenas ou milhares de processadores devido ao baixo custo dos microprocessadores atualmente.
- Execução assíncrona, onde cada nó executa independentemente de outros nós e a sincronização entre os nós é baseada em primitivas de trocas de mensagens.
- Comunicação baseada em troca de mensagens. Ao contrário de organizações de computadores fortemente acoplados, onde os processadores se comunicam através de memória compartilhada, os nós de um multicomputador se comunicam somente por troca de mensagens.
- Computação de granularidade média, ou seja, a relação entre o tempo de computação e a soma dos tempos de comunicação e sincronização das tarefas não é tão baixa quanto nos sistemas SIMD (possuem granularidade fina) e nem tão alta quanto nos sistemas MIMD de memória compartilhada, onde cada processo possui o mesmo tamanho de um processo presente em um sistema operacional convencional (possuem granularidade grossa).

I.2 Processadores de Comunicação em Sistemas Distribuídos para Processamento Paralelo

Recentemente, algumas pequisas têm sido realizadas no sentido de fornecer suporte de *hardware* para melhorar o desempenho de multicomputadores. Estas propõem a separação de funções de processamento relacionadas com a aplicação, das funções de comunicação, utilizando-se um processador de comunicação para este fim [4] [12] [5].

Nos multicomputadores, a maior parte do tempo gasto em comunicação é atribuída à necessidade de rotear mensagens através de nós intermediários de um dado caminho fonte-destino e à própria comunicação física. Como, nestes sistemas de computação distribuída, o custo de comunicação é comparável ao de processamento, se as tarefas de roteamento forem de responsabilidade do mesmo processador que executa a aplicação, este processador pode se tornar um gargalo. Neste contexto, é possível verificar que existem alguns computadores concorrentes no mercado que incorporam um sistema de comunicação com um *hardware* para fazer o roteamento de mensagens, tais como os multicomputadores de segunda geração: IPSC/2 da INTEL [11] e SERIES 2010 da AMETEK [14]. Estas máquinas, ao contrário dos multicomputadores de primeira geração, utilizam uma forma de comunicação onde, na transmissão de mensagens por caminhos compostos por uma série de canais, não há interrupção dos processos que executam nos nós intermediários, pois as funções de roteamento são feitas por dispositivos dedicados exclusivamente a esta tarefa. No IPSC/2, por exemplo, um módulo de *hardware* "DIRECT CONNECT MODULE (DCM)" é utilizado para executar o roteamento de mensagens. Os DCM's são conectados para formar um hipercubo e utilizam uma rede *circuit-switched*, ou seja, estes criam circuitos, dinamicamente, compostos de uma série de canais que formam rotas únicas de nós fontes para nós destinos, passando por diversos roteadores intermediários associados com outros nós. Estes circuitos permanecem abertos durante as transmissões das mensagens, que não são armazenadas nos nós intermediários. Este mecanismo substitui o de troca de mensagens *store-and-forward* empregado no sistema IPSC original. O sistema do SERIES 2010 também utiliza dispositivos para roteamento de mensagens, que formam uma rede de duas dimensões, ou melhor, os nós são conectados através de uma matriz de circuitos integrados "AUTOMATIC MESSAGE ROUTING DEVICE (AMRD)". As mensagens nesta rede são transmitidas como uma seqüência de um ou mais pacotes que possuem informações relativas aos seus destinos. Desta forma, os AMRD's fazem os roteamentos destes pacotes até os seus destinos, sem interrrupções dos processos que estão sendo executados nos processadores intermediários.

Além da necessidade de um *hardware* para realizar o roteamento para auxiliar a comunicação, os aspectos vistos a seguir também devem ser considerados

no projeto de um processador de comunicação em sistemas distribuídos.

O projeto de um algoritmo distribuído envolve a especificação da computação a ser executada em cada nó do sistema, bem como da comunicação entre os diversos nós. A construção de algoritmos distribuídos corretos e confiáveis constitui, atualmente, um grande desafio para seus projetistas, já que estes não podem contar com o suporte existente para projeto de algoritmos centralizados, como provas de correção, estabelecimento de classes de complexidade e limites. Embora a compreensão dos fundamentos de sistemas distribuídos esteja ainda em um estágio inicial, muitos passos significativos têm sido dados, tendo-se hoje algoritmos distribuídos elegantes para resolver vários problemas que surgem no controle de sistemas distribuídos. Alguns destes algoritmos, embora complexos, são básicos o suficiente para serem considerados potenciais blocos de construção de alto nível no projeto de algoritmos de maior complexidade [7]. Da mesma forma, em um nível mais baixo, encontram-se também vários outros constituintes de algoritmos distribuídos que poderiam ser considerados blocos de construção. Estes são os procedimentos que tratam da comunicação ponto-a-ponto e de grupo em sistemas distribuídos. Tais procedimentos normalmente executam trocas de mensagens entre dois nós quaisquer e propagam mensagem entre os nós do sistema. Algumas formas utilizadas para propagação de mensagens em sistemas distribuidos, conforme [2], são:

- *Broadcast* para vizinhos, no qual um nó origem envia uma mesma mensagem para todos os seus vizinhos e pode receber confirmações destes em relação ao recebimento da mensagem, que é chamado de *broadcast* para vizinhos com realimentação.

- *Broadcast* em uma árvore geradora, que pressupõe a existência de uma árvore que cobre todo o sistema cuja raiz corresponde ao nó que origina o *broadcast.*

- *Broadcast* por enchente, onde o nó que origina o *broadcast* envia a mensagem para todos os seus vizinhos. Todos os outros nós fazem o mesmo ao receber a mensagem pela primeira vez. Há também *broadcast* por enchente com realimentação, onde, o nó originando o *broadcast* é informado quando a mensagem

atinge todos os outros nós do sistema.

- *Convergecast* em uma árvore geradora, que indica um fluxo de mensagens oposto ao de um *broadcast* em uma árvore geradora, ou seja, o *convergecast* é iniciado por todos os nós que constituem as folhas de uma árvore e cada nó não folha transmite a mensagem pelo seu único canal de saída da árvore quando recebem mensagens de todos os seus canais de entrada da árvore.

Estas operações podem sobrecarregar o sistema ou por serem complexas (no caso de *broadcasts* em árvores geradoras, por enchente e *convercast*) ou por serem executadas muito freqüentemente (no caso de trocas de mensagens e *broadcast* para vizinhos). Para evitar isso, em [2] é proposta a utilização de um processador de comunicação que execute tais funções, melhorando o desempenho do sistema. Desta forma, um nó da rede é decomposto em um processador hospedeiro e um processador de comunicação que executa estas funções sem conhecer a aplicação de nível mais alto que executa no processador hospedeiro.

Tendo em vista estas considerações, neste trabalho são apresentados o projeto e a implementação em *software* de um processador virtual de comunicação (CVP) para sistemas distribuídos. Este processador executa as operações descritas acima e possui alguns procedimentos adicionais (descritos sucintamente a seguir) que tornam sua utilização mais flexível e eficiente.

A transmissão de mensagens pode ser realizada de dois modos : *store-and-forward* e *circuit-switching*. Em ambos os modos pode haver *deadlock* de comunicação. Uma solução para prevenir tal problema é garantir que o algoritmo de roteamento seja livre de *deadlock*, ou seja, que não existam ciclos nas rotas utilizadas para transmissão de mensagens. Caso não se tenha tal algoritmo para roteamento, no modo de transporte *store-and-forward*, ainda se pode prevenir *deadlock* dividindo-se os *buffers* do processador em classes. Ou seja, é desejável que um processador de comunicação trate o problema de *deadlock* adequadamente para cada um dos modos de comunicação. Para resolver este problema, neste trabalho, foram implementados dois modelos de processadores de comunicação. O primeiro modelo (CVP I) atende aos modos de transporte *store-and-forward* e *circuit-switching* e exige que exista um

esquema de roteamento que previna *deadlock*. A este modelo foram acrescentadas funções de reservas e liberações de canais para um circuito e envio de mensagens por circuito, que podem ser utilizadas na troca de mensagens entre dois nós quaisquer ou na propagação de informação em uma árvore geradora (circuito em forma de árvore). O segundo modelo (CVP II), não exige um esquema de roteamento livre de *deadlock* e, por isso mesmo, não atende ao modo *circuit-switching*, apenas ao *store-and-forward*; entretanto, este modelo possui os *buffers* dividido em classes, garantindo a não ocorrência de *deadlock*. O CVP I e o CVP II em diversos aspectos funcionam da mesma forma e, por isso, no decorrer do trabalho, muitas vezes são tratados como um único processador, sendo referenciados simplesmente por CVP.

Este trabalho está organizado em cinco capítulos.

No Capítulo II, são especificadas as estruturas destes modelos com as descrições das unidades que os compõem e a discussão das diferenças e características comuns. As instruções executadas por estes processadores para realizarem as funções de troca de mensagens e propagação de informação, nos modos *store-and-forward* e *circuit-switching*, são descritas no Capítulo III, onde são também apresentados os algoritmos utilizados para implementá-las. Os possíveis tipos de *deadlock* nos métodos de transporte *store-and-forward* e *circuit-switching* e as soluções adotadas no CVP estão no Capítulo IV. Finalmente, no Capítulo V, se encontram as conclusões do trabalho.

No Apêndice A, são descritos o processador *Transputer*, a linguagem *Occam* e o sistema NCP-1, utilizados na implementação e testes destes processadores.

No Apêndice B, há um manual para utilização do CVPæ

Capítulo II

Descrição do Processador Virtual de Comunicação

O CVP é uma unidade dirigida por mensagens conectada a um processador hospedeiro e a outros CVP's por canais de comunicação bi-direcionais. Esta interconexão de processadores de comunicação fornece um "subsistema de comunicação" como é apresentado na Figura II.1, onde os quadrados represetam os hospedeiros, as circunferências representam os processdores virtuais de comunicação e a elipse tracejada determina o subsistema de comunicação.

Todas as mensagens que chegam a um nó são recebidas, tratadas e despachadas apropriadamente pelo CVP. Para realizar estas tarefas o CVP é composto das seguintes unidades: uma Unidade de Decisão (UD), uma Unidade de Gerência de Buffer (UGB) e para cada um dos canais uma Unidade Despachadora (UDES).

As mensagens que chegam ao CVP provêm ou do hospedeiro ao qual está conectado ou dos CVP's vizinhos. Estas mensagens, de um modo geral, são tratadas pela Unidade de Decisão que utiliza informações disponíveis em estruturas de dados do próprio CVP. A Unidade de Gerência de Buffer é a unidade responsável pelo armazenamento e recuperação de mensagens em um *buffer* do CVP. As Unidades Despachadoras são responsáveis pelo envio de mensagens para outros CVP's e hospedeiro e pelo controle de seus recebimentos.

Como já dito no Capítulo I, foram implementados dois modelos de

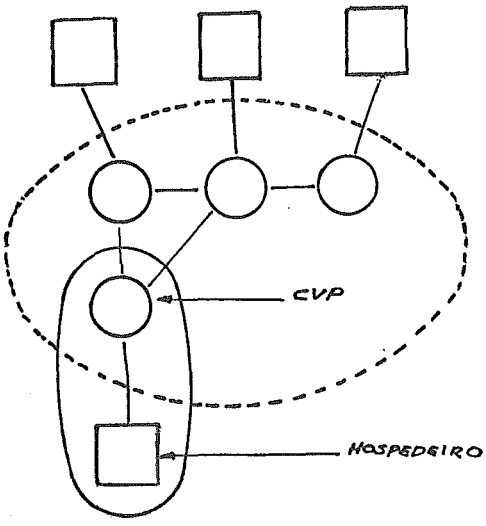

Figura II.1: Subsistema de Comunicação

processadores de comunicação. O CVP I, que atende aos modos de transporte *store-and-forward* e *circuit-switching*, exigindo um esquema de roteamento que previna *deadlock*, e o CVP II, que não exige um esquema de roteamento livre de *deadlock* e, por isso mesmo, não atende ao modo *circuit-switching*, apenas ao *store-and-forward*, tendo os *buffers* divididos em classes, para garantir a não ocorrência de *deadlock*.

II.1 Métodos de transporte *circuit-switching* e *store-and-forward*

Existem fundamentalmente dois métodos de transporte : *store-and-forward* e *circuit-switching*. No modo *circuit-switching* há um mecanismo que estabelece a rota para a mensagem e mantém esta reservada até que a transmissão termine. As desvantagens deste modo são a dificuldade de estabelecimento de uma rota em uma dada condição de carga e a perda de grande parte da largura de banda da rede quando a comunicação é feita em rajadas. A solução *circuit-switching*, porém, oferece vantagens. Por exemplo, as estratégias de armazenamento (*buffering*) são mais simples do que as utilizadas para redes empregando *store-and-forward*, pois, no modo *circuit-*

switching não há praticamente necessidade de armazenamento em processadores intermediários, ou melhor, somente alguns dígitos do fluxo ("*flits*") são armazenados em cada nó antes de serem transmitidos para o nó seguinte. Um *flit* é a menor unidade de informação que pode ser transmitida [6]. Conforme os *flits* fluem, a mensagem vai se espalhando pelos canais entre os nós fonte e destino. É possível que o primeiro *flit* de uma mensagem chegue ao nó destino antes do último *flit* da mensagem deixar o nó fonte. Como os *flits* não contêm informações de roteamento, os *flits* de uma mensagem devem permanecer em canais contíguos da rede e não podem ser intercalados com os de outras mensagens.

Em redes *store-and-forward*, as mensagens transmitidas de um nó para outro são temporariamente armazenadas na memória de nós intermediários e mais tarde encaminhadas para o nó final. Se uma mensagem passa por n canais, esta é armazenada temporariamente em n menos um nós. Este método de transporte evita o problema de perda de largura de banda, pois a aloca dinamicamente para mensagens, conforme estas fluem pela rede. Entretanto, o tempo gasto para roteamento é pior do que no *circuit-switching*, porque decisões de roteamento devem ser tomadas em cada processador de comunicação intermediário no caminho para cada mensagem enviada pela rede. Além disso, há um certo atraso em relação ao modo *circuit-switching*, como pode ser observado na Figura II.2. O CVP envia, no modo *store-and-forward*, mensagens inteiras de comprimentos variáveis.

II.2 Estrutura do CVP I

Os CVP's trocam basicamente mensagens que determinam operações de comunicação a serem executadas pela Unidade de Decisão. Estas operações definem o conjunto de instruções do CVP. Entretanto, além destas mensagens, algumas outras, com informações de *status*, descritas mais adiante, são também necessárias. Por isso, na conexão de CVP's existem, adicionalmente a um caminho primário de instruções,"inst.chan", dois outros caminhos, como se pode obsevar na Figura II.3.

Um deles, "st.chan", é utilizado para transmitir informações relativas ao controle de fluxo. Controle de fluxo é o mecanismo que regula a transmissão de

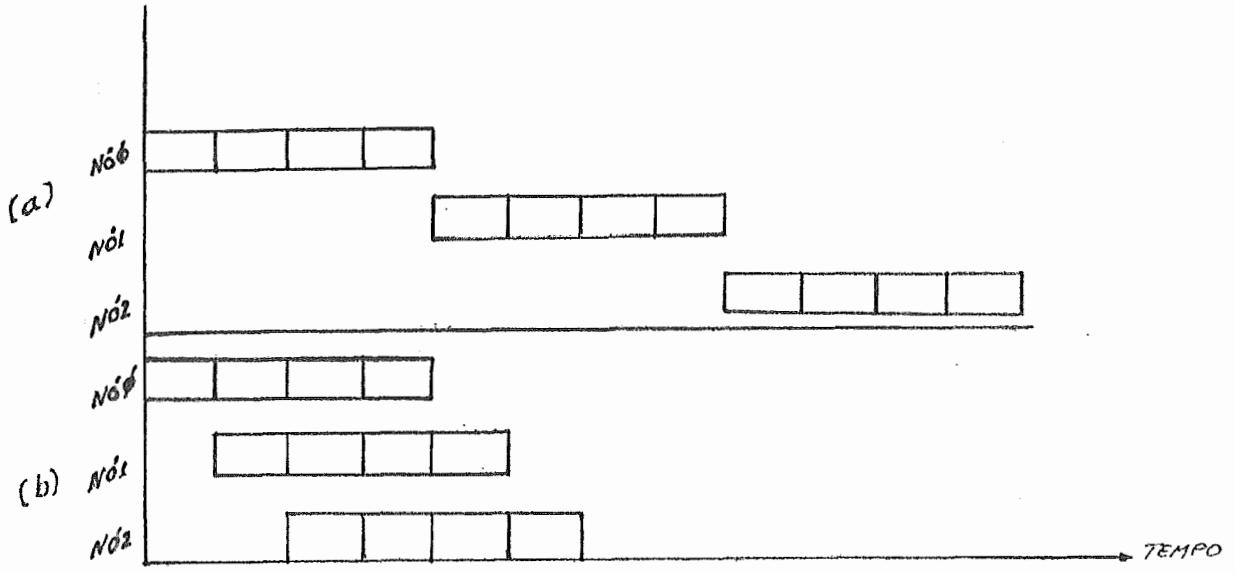

Figura II.2: Métodos store-and-forward (a) e circuit-switching (b)

mensagens por canais a fim de se evitar *overflow* de *buffer*. Desta forma, cada CVP, ao enviar uma mensagem, no modo *store-and-forward*, espera que o receptor retorne um sinal de controle indicando se esta foi aceita ou rejeitada, ou seja, se este possui espaço em *buffer* suficiente para armazená-la ou não. Um sinal de *ack* denota uma mensagem aceita, enquanto um *nack* denota uma mensagem rejeitada. Se um *nack* é retornado, outras retransmissões da mensagem são feitas até esta ser aceita. Tem-se, portanto, esta linha de controle separada para transmitir sinais de *ack* e *nack*. O terceiro caminho, "st.cir.chan", é utilizado para transmitir informações de *status* relativas ao estabelecimento de circuito. Quando um circuito é estabelecido, o nó destino do circuito inicia a transmissão de uma confirmação para o nó fonte por este terceiro caminho, ao longo do mesmo percurso do circuito em sentido oposto.

Na conexão do CVP I com o hospedeiro tem-se um único caminho no sentido hospedeiro-CVP e três caminhos no sentido contrário, como pode ser visto na Figura II.3.

O CVP recebe instruções do hospedeiro e no sentido contrário o hospedeiro, normalmente, recebe do CVP ao qual está conectado mensagens que foram

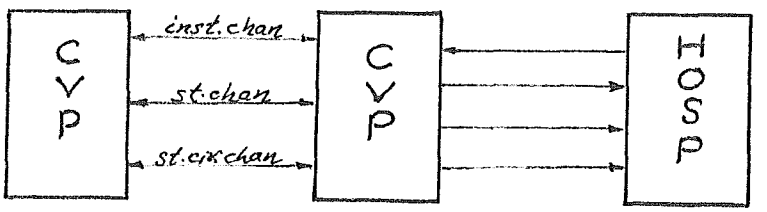

Figura II.3: Caminhos na conexão de CVP's

transmitidas no sistema através das instruções dos CVP's e que atingiram os seus destinos. Porém, o hospedeiro pode também receber do CVP informação relativa à aceitação ou recusa de instrução enviada para o CVP pelo hospedeiro e, ainda, informação relativa a um pedido de estabelecimento de circuito, como um aviso para o hospedeiro de que o circuito para um determinado destino já está estabelecido e pode ser usado.

No CVP, todas as mensagens são enviadas pelas Unidades Despachadoras e todas as mensagens são recebidas pela Unidade de Decisão, com exceção das que chegam via o caminho *st.chan*, que são recebidas pelas Unidades Despachadoras, como pode ser observado na Figura II.4. A Unidade de Decisão envia as mensagens a serem despachadas ou para a Unidade de Gerência de Buffer, no caso *store-and-forward*, ou diretamente para as Unidades Despachadoras, no caso *circuit-switching*. Da mesma forma, as Unidades Despachadoras enviam mensagens recuperadas de *buffers* via a Unidade de Gerência de Buffer ou mensagens recebidas diretamente da Unidade de Decisão, com exceção da Unidade Despachadora relativa ao canal que conecta o CVP ao hospedeiro (UDESH) que envia sempre mensagens recebidas diretamente da Unidade de Decisão.

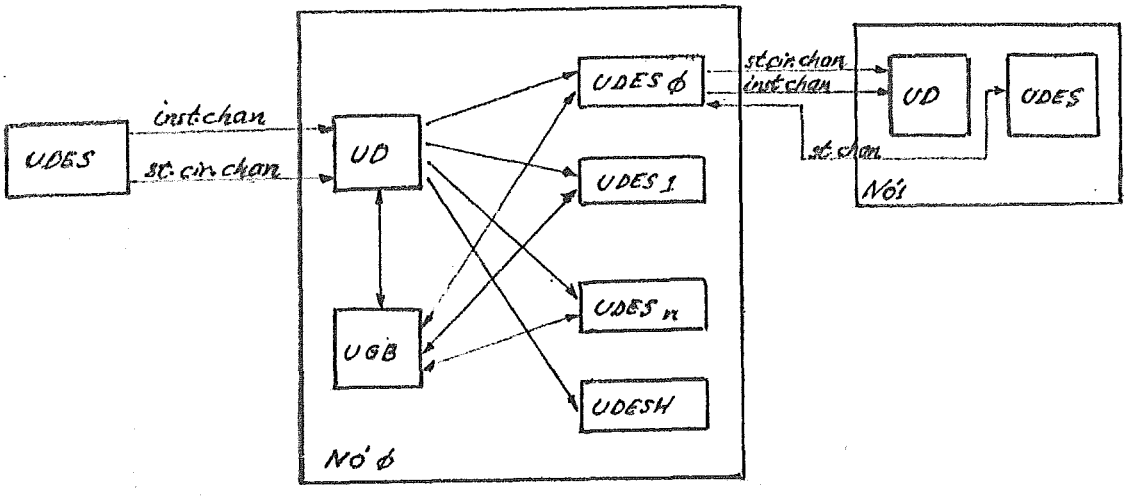

Figura II.4: Estrutura do CVP I

O algoritmo descrevendo o funcionamento do CVP pode ser visto na Figura II.5, onde todas as unidades executam paralelamente.

execute em paralelo :
  Unidade de Decisão
  Unidade de Gerência de Buffer
  para cada canal i do CVP execute em paralelo :
    Unidade Despachadora do canal i

Figura II.5: Algoritmo de funcionamento do CVP

II.2.1 Unidade de Decisão

A Unidade de Decisão recebe mensagens ou de outros CVP's ou do hospedeiro. Quando recebidas do hospedeiro estas mensagens são sempre instruções. Sendo recebidas de outros CVP's estas mensagens podem ser instruções ou confirmações de circuitos que estão sendo estabelecidos. No caso de instruções, a Unidade de Decisão pode necessitar de armazenamento de uma ou mais mensagens que serão posterior-

mente despachadas como resultado de suas execuções, se o modo de tranmissão é *store-and-forward.* Nesse caso, a Unidade de Decisão pede à Unidade de Gerência de Buffer para armazenar tais mensagens. Caso não haja *buffers* disponíveis, a Unidade de Decisão, ao receber esta informação da Unidade de Gerência de Buffer, envia um *nack* à Unidade Despachadora relativa ao canal pelo qual a mensagem foi recebida. Caso contrário, um *ack* é enviado.

Se, como conseqüência da execução de uma instrução, for necessário o armazenamento de outras instruções, a Unidade de Decisão também pede à Unidade de Gerência de Buffer a atualização de filas (existe uma por canal) de ponteiros para *buffers* com as mensagens a serem enviadas, como é descrito na seção II.2.3.

No modo *circuit-switching*, as instruções ou os *flits* recebidos pela Unidade de Decisão devem ser transmitidos diretamente para toda Unidade Despachadora associada a um canal de saída do circuito relacionado, sem necessidade de armazenamento em *buffer.* Existirão por CVP vários canais reservados para um mesmo cicuito quando este tiver a forma de uma árvore (ver Seção III.2).

As informações de confirmação de estabelecimento de circuito podem ser consideradas instruções a serem tratadas pela Unidade de Decisão pois estas resultam ou no envio de uma mensagem para o hospedeiro, caso este seja a fonte do pedido de estabelecimento de circuito ou no envio de mensagens do mesmo tipo para CVP's vizinhos, necessitando, portanto, de um certo tratamento.

A informação de confirmação de estabelecimento de circuito deve ser transmitida por um caminho diferente do caminho de instruções, para evitar seu bloqueio, caso os canais do percurso do cicuito em sentido contrário estejam reservados.

O algoritmo da Figura II.6 é executado ciclicamente. Neste, a Unidade de Decisão fica esperando o recebimento de instruções pelos canais do CVP, onde para cada canal, que o conecta a outro CVP, existem dois caminhos associados (*inst.chan* e *st.cir.chan*). Para cada um destes caminhos existe um procedimento relacionado que é executado quando a entrada referente a este caminho se torna disponível. Assim, as operações "recebendo" usadas no algoritmo não são necessa-

riamente executadas na ordem em que foram escritas e indicam, apenas, as operações que devem ser realizadas, caso sejam recebidas instruções por um destes caminhos. Ao receber uma instrução de outro CVP ou do hospedeiro pelo caminho *inst.chan*, a Unidade de Decisão basicamente executa as operações descritas no algoritmo (explicadas acima) para cada um dos modos *store-and-forward* e *circuit-switching*, onde a operação "executa instrução" é melhor especificada no Capítulo III.

```
recebendo instrução do hospedeiro ou de CVP's (inst.chan):
  se modo de transmissão store-and-forward então
     início
       envia pedido à UGB para gravar instrução
       recebe resposta da UGB
       se UGB gravou instrução em buffer então
          início
            envia pedido à UDES para enviar um ack
            executa instrução
            envia pedido à UGB para atualizar a fila correspondente
          fim
       senão
          envia pedido à UDES para enviar um nack
  senão (modo circuit-switching)
     início
       executa instrução
       envia pedido à UDES para enviar a instrução pelo canal do circuito correspondente
     fim
recebendo instrução com status de circuito de CVP's (st.cir.chan):
  executa instrução
  envia pedido à UDES para enviar a instrução de status em sentido oposto ao do circuito
```

Figura II.6: Algoritmo da Unidade de Decisão do CVP I

II.2.2 Unidades Despachadoras

Existe uma Unidade Despachadora para cada canal do CVP que pode conectá-lo a outro CVP ou ao hospedeiro. As Unidades Despachadoras para canais que interligam CVP's podem enviar instruções recebidas da Unidade de Gerência de Buffer (modo *store-and-forward*) ou da Unidade de Decisão (modo *circuit-switching*) Estas podem receber também da Unidade de Decisão mensagens para controle de

fluxo e mensagens de confirmação de circuitos que estão sendo estabelecidos.

A Unidade Despachadora para o canal que conecta o CVP ao hospedeiro recebe as mensagens diretamente da Unidade de Decisão.

Quando as mensagens são enviadas no modo *store-and-forward*, as Unidades Despachadoras devem receber um *ack* ou um *nack* relativo à aceitação ou recusa da mensagem enviada e, então, pedir à Unidade de Gerência de Buffer a atualização da fila correspondente ao canal pelo qual a mensagem foi enviada, caso necessário. Ou seja, se uma mensagem é aceita esta deve ser retirada da fila da unidade de origem, se for recusada deve permanecer, a fim de que novas tentativas de envio sejam feitas. As informações de controle de fluxo devem se propagar livremente sem que haja bloqueios por possíveis reservas de circuitos, o que provocaria um atraso na Unidade Despachadora. Por causa disso, foi implementado o segundo caminho, *st.chan*, que permite a livre circulação destas mensagens.

O algoritmo de uma Unidade Despachadora que conecta o CVP a outro CVP via o canal i está na Figura II.7. Neste algoritmo é utilizada a variável *wait.ack*, que tem o valor TRUE caso a Unidade Despachadora esteja esperando por uma confirmação relativa a uma instrução enviada, e tem o valor FALSE, caso contrário. A UDES só pode fazer um pedido de recuperação de instrução na fila i à UGB se o valor de *wait.ack* for igual a FALSE. A UDES, como mostra o algoritmo, pode receber mensagens da UD, do CVP vizinho e da UGB. Ao executar as operações relacionadas ao recebimento de mensagens da UD e da UGB, o valor de *wait.ack* pode ser atualizado com TRUE. Quando isto ocorre, não deve ser feito um novo pedido à UGB porque ou esta ainda não enviou a resposta ou porque, tendo enviado a instrução recuperada da UGB, a UDES ainda não recebeu a confirmação do CVP vizinho. Assim, um novo pedido à UGB só pode ser feito quando uma confirmação é recebida ou quando a UGB responde com um sinal de fila vazia, o que significa que novas tentativas devem ser feitas. Este algoritmo é executado ciclicamente.

```
se ack.wait = FALSE então
   envia pedido à UGB para recuperação de instrução da fila do canal i
recebendo do canal i confirmação de mensagem recebida :
  envia pedido à UGB para atualizar a fila do canal i
  ack.wait := FALSE
recebendo pedido da UD :
  envia pelo canal i (pelo caminho apropriado) a mensagem recebida da UD
  ack.wait := TRUE
recebendo resposta da UGB em relação ao pedido feito :
  se instrução recuperada então
     início
       envia pelo canal i a instrução
       ack.wait := TRUE
     fim
  senão (fila vazia)
    ack.wait := FALSE
```

Figura II.7: Algoritmo da Unidade Despachadora do CVP I

II.2.3 Unidade de Gerência de Buffer

A Unidade de Gerência de Buffer é utilizada na transmissão de mensagens no modo *store-and-forward.* Esta é responsável pela localização de *buffer* disponível para armazenamento de uma mensagem recém recebida da Unidade de Decisão e pela localização da próxima mensagem que estiver esperando para ser transmitida via uma certa Unidade Despachadora. Estas mensagens são instruções a serem enviadas para outros CVP's. Os *buffers* do CVP I são divididos em duas partes. Uma, onde são armazenadas as mensagens recebidas do hospedeiro e outra, onde são armazenadas as mensagens recebidas de outros CVP's.

A Unidade de Gerência de Buffer mantém associadas a cada canal duas filas FIFO de ponteiros para *buffers.* A primeira, para mensagens que podem ser imediatamente enviadas pelo canal ("fila de envio") e a segunda, para mensagens que devem esperar pela liberação do canal, que se encontra reservado por algum circuito, para serem enviadas ("fila de espera"). Para cada fila é necessário apenas um ponteiro para um *buffer* físico, que é o primeiro da fila. Cada *buffer* físico tem associado também um ponteiro para o próximo *buffer* físico da fila.

As atualizações destas filas são feitas através de pedidos das Unidade de Decisão e Unidades Despachadoras à Unidade de Gerência de Buffer. A Unidade de Decisão faz um pedido de atualização destes ponteiros sempre que executa intruções que geram a necessidade do armazenamento de outras a serem, posteriormente, despachadas. Nesta situação, a fila de envio é utilizada quando o canal pelo qual a mensagem deve ser enviada não está reservado, caso contrário utiliza-se a fila de espera. Um pedido de uma Unidade Despachadora para recuperação de uma mensagem resulta na consulta à fila de envio do canal associado. Tendo seu pedido atendido, a Unidade Despachadora, ao enviar a mensagem, espera do CVP vizinho uma resposta de aceitação (*ack*) ou recusa (*nack*) desta. Caso a mensagem seja aceita, é feito um pedido de exclusão do ponteiro relativo à mensagem da fila. Caso a mensagem seja recusada, a Unidade Despachadora continua tentando enviá-la.

A fila de espera, associada a um certo canal, é atualizada quando este, estando reservado para um circuito, é liberado, através da execução de uma instrução específica para este fim. (descrita no Seção III.2). Quando isto ocorre, há a transferência de mensagens desta fila de espera, que não podiam ser enviadas porque o canal estava reservado por um circuito, para a fila de envio. É importante observar que, quando existe na fila de espera uma instrução de pedido de reserva de canal para um circuito, não é feita a transferência para a fila de envio das instruções subseqüentes a esta na fila de espera. Como é melhor explicado no Capítulo III, para cada instrução de pedido de reserva de canal é necessária a execução de uma instrução de liberação do canal, pois esta última permite que apenas as instruções executadas após o pedido correspondente de reserva de canal, mas antes de uma outra instrução de reserva do canal, sejam transferidas para a fila de envio.

As mensagens transmitidas podem ocupar mais de um *buffer* físico, o que implica, algumas vezes, na necessidade de encadeamento de *buffers* para armazenar uma mensagem inteira .

No algoritmo da Figura II.8, pode-se observar que a UGB espera pedidos da UD e das UDES's. Embora não representado no algoritmo, no pedido feito pela UD para inserção na fila, deve ser especificado a que canal a fila se refere e se a fila é de envio ou de espera, ou seja, se a atualização deve ser feita na fila de pontei-

ros para instruções que podem ser imediatamente despachadas pelo canal ou na fila em que as instruções devem esperar pela liberação do canal reservado por um dado circuito. Ao atender este pedido, a UGB insere em uma destas filas a identificação (ponteiro) do *buffer* alocado pela instrução. A operação "transfere instruções da fila de espera para a fila de envio" consiste na transferência dos ponteiros de *buffers* da fila de espera para a fila de envio, que deverá estar vazia, já que quando se usa a fila de espera se pressupõe que o canal relativo a esta fila foi reservado para um certo circuito, através de uma instrução, que foi a última a utilizar o canal antes de ser reservado para o circuito (ver Capítulo III). Ao atender um pedido de uma Unidade Despachadora para recuperação de uma instrução, a UGB envia uma cópia da primeira instrução da fila de envio para a Unidade Despachadora. A retirada deste elemento da fila, porém, só ocorre, quando a Unidade Despachadora, tendo recebido uma confirmação de aceitação da instrução enviada, pede à UGB para atualizar a fila.

```
recebendo pedido da UD :
  se pedido de gravação de instrução em buffer então
     início
       se buffer diponível então
          grava a instrução no buffer
       envia resposta à UD sobre operação realizada
     fim
  senão (pedido de atualização de fila)
    se pedido de inserção na fila então
       insere ponteiro do buffer na fila do canal correspondente
    senão (pedido de transferência)
      transfere instruções da fila de espera para a fila de envio
recebendo pedidos das UDES's :
  se pedido de recuparação de instrução do buffer então
     início
       recupera primeira instrução da fila do canal correspondente
       envia resposta à UDES com instrução recuperada ou sinal de fila vazia
     fim
  senão (pedido de atualização de fila)
    retira primeiro elemento da fila do canal correspondente
```

Figura II.8: Algoritmo da Unidade de Gerência de Buffer do CVP I

II.2.4 Estruturas de Dados

A relativa complexidade das operações a serem executadas pelo CVP exige que vários registradores e tabelas sejam manipulados. Antes de apresentar estas estruturas de dados, são descritos a seguir os parâmetros relevantes no projeto do processador:

nodes : número de nós no sistema. Assume-se que nós são numerados consecutivamente de 0 até *nodes* − 1 no sistema.

chans : número máximo de canais conectando um CVP a outros CVP's. Assume-se que canais são numerados consecutivamente de 0 a *chans* − 1 no CVP.

broadsnf : número máximo de *broadcasts* para vizinhos com realimentação que um nó pode executar antes de uma confirmação voltar.

trees : número máximo de árvores geradoras que podem coexistir simultaneamente no sistema com raiz em um dado nó. É usada em instruções para realizar *broadcast* e *convergecast* em árvores. Árvores são numeradas de 0 a *trees* − 1.

convs : número máximo de *convergecasts* em uma mesma árvore geradora do qual um nó pode estar participando simultaneamente.

floods : número máximo de *broadcasts* por enchente de uma mesma origem do qual um nó pode estar participando simultaneamente.

floodsfb : número máximo de *broadcasts* por enchente com realimentação de uma mesma origem do qual um nó pode estar participando simultaneamente.

max.len.msg : comprimento máximo de uma mensagem.

len.buf : comprimeto de um *buffer* físico.

num.buf : número de *buffers* físicos.

route : tabela com *nodes* entradas que armazena a estrutura de roteamento do sistema . A entrada *route*[*i*] desta tabela fornece o próximo canal no caminho para o nó *i*.

tree.broad : tabela de *chans* linhas e *trees* colunas utilizada para formar árvores geradoras, ou melhor, para uma entrada associada ao canal c e árvore t ($tree.broad[t, c]$), esta fornece o valor TRUE se o canal c é um canal de saída na árvore t ou FALSE, caso contrário.

tree.convin : tabela do mesmo tipo da anterior, onde, se um canal c é um canal de entrada em uma árvore de *convergecast* t, tem-se $tree.convin[t, c]$ TRUE, caso contrário, $tree.convin[t, c]$ FALSE.

tree.convout : também do mesmo tipo que as tabelas anteriores, indicando se um canal c é um canal de saída na árvore de *convergecast* t. Caso seja, tem-se $tree.convout[t, c]$ TRUE, caso contrário $tree.convout[t, c]$ é FALSE.

O único número que cada nó possui no sistema é sua identificação, armazenado no registrador *id.*

As estruturas de dados descritas a seguir são atualizadas pelo CVP durante a execução das instruções.

O controle de *broadcasts* para vizinhos com realimentação é feito com a tabela $broadnf$ com *chans* entradas, que fornece o número de mensagens esperadas para cada canal devido a *broadcasts* de mensagens com realimentação. Este valor é um inteiro entre 0 e *broadsnf.*

A fim de controlar as mensagens de *convergecast* que chegam ao CVP em uma mesma árvore geradora, é utilizada a tabela *convt*, de *trees* linhas e *chans* colunas. Se t é uma árvore e c um canal, $conv[t, c]$ só tem sentido se $tree.convin[t, c]$ for verdadeiro, neste caso esta contém o número de mensagens de *convergecast* ainda esperadas nesse canal. Este valor é um inteiro entre 0 e *convs.*

O controle de *broadcasts* simultâneos por enchente é conseguido com o auxílio da tabela $flood$ com *nodes* linhas e *chans* colunas. Para um nó i e um canal c o valor de $flood[i, c]$ é um inteiro de 0 a *floods*, e representa o número de mensagens esperadas no canal c devido a *broadcasts* por enchente iniciados no nó i.

No controle de *broadcasts* simultâneos por enchente com realimentação utilizamos duas tabelas. A primeira tabela, $floodfb$, possui *nodes* linhas e *chans* colunas e, analogamente ao caso anterior, se i é um nó e c é um canal, o valor $floodfb[i, c]$ representa o número de mensagens esperadas pelo canal c devido a *broadcasts* por enchente com realimentação iniciados em i. Por isso, este valor é um inteiro entre 0 e *floodsfb*. A outra tabela, *pending*, possui *nodes* linhas e *floodsfb* colunas, e é usada para indicar os canais nos quais as mensagens de realimentação devem ser enviadas para os vários *broadcasts*. Se i é um nó, as colunas da linha i são indexadas por dois ponteiros $first[i]$ e $last[i]$, que são incrementados circularmente na linha. O valor de $pending[i, first[i]]$ é o canal no qual uma mensagem de realimentação deve ser enviada quando o *broadcast* iniciado por i terminar, estando este *broadcast* previsto a terminar primeiro no CVP. Da mesma forma, $pending[i, last[i]]$ é atualizado com o canal no qual uma mensagem de realimentação deve ser enviada quando o *broadcast* iniciado por i terminar, estando este *broadcast* previsto para terminar por último.

Para controle de pedidos de reserva de um canal por circuitos existe a tabela *ask.counter* com *chans* entradas. Dado um canal c, a entrada $ask.counter[c]$ indica o número de pedidos de reservas deste canal executados por instruções de reservas de circuito.

A fim de controlar o envio de uma confirmação de estabelecimento de um circuito em forma de árvore são usadas as tabelas *tree.convin.circuit* e *tree.convout.circuit*. Estas atendem propósitos semelhantes aos das tabelas *tree.convin* e *tree.convout*. Da mesma forma, também é utilizada a tabela *conv.circuit* com propósitos análogos aos da tabela *conv*. Para controle do envio de uma confirmação de estabelecimento de um circuito simples usa-se a tabela *in.chan* com *nodes* linhas e *nodes* colunas. Dado um nó fonte do circuito, *source*, e um nó destino do circuito, *dest*, $in.chan[source, dest]$ fornece o canal a ser utilizado para o envio da confirmação. Estas tabelas são carregadas pelo CVP durante o estabelecimento do circuito.

II.3 Estrutura do CVP II

Este modelo trabalha basicamente da mesma forma que o anterior, não atendendo, entretanto, ao modo *circuit-switching*. Assim, este não possui o caminho *st.cir.chan* que, como já visto, transmite informação a respeito de circuitos sendo estabelecidos. Além disso, os *buffers* neste modelo são divididos em classes. Tendo em vista estas diferenças, são descritas a seguir as características das unidades do CVP II que distinguem os dois modelos, sem a repetição das características comuns a ambos, já explicadas na seção anterior.

II.3.1 Unidade de Decisão

A Unidade de Decisão deste modelo recebe apenas instruções, ou seja, como não existe neste modelo o caminho *st.cir.chan*, a Unidade de Decisão deve apenas tratar instruções recebidas pelo caminho *inst.chan*. O algoritmo desta unidade está na Figura II.9. As considerações relativas ao algoritmo da Unidade de Decisão do CVP I continuam válidas para este algoritmo, com exceção das relacionadas ao método de transporte *circuit-switching*.

```
recebendo instrução do hospedeiro ou de CVP's :
  início
    envia pedido à UGB para gravar instrução
    recebe resposta da UGB
    se UGB gravou instrução em buffer então
      início
        envia pedido à UDES para enviar um ack
        executa instrução
        envia pedido à UGB para atualizar a fila correspondente
      fim
    senão
      envia pedido à UDES para enviar um nack
  fim
```

Figura II.9: Algoritmo da Unidade de Decisão do CVP II

II.3.2 Unidades Despachadoras

Todas as instruções enviadas pelas Unidades Despachadoras deste modelo são recebidas da Unidade de Gerência de Buffer. Estas unidades ainda enviam mensagens de *status* para controle de *buffer* recebidas da Unidade de Decisão. Neste modelo quando a Unidade Despachadora recebe um *nack* de um CVP vizinho como resultado do envio de uma instrução, esta pede para Unidade de Gerência de Buffer colocar esta mensagem no final da fila. Ou seja, o fato de uma mensagem ser recusada não indica que não há espaço no CVP vizinho para qualquer mensagem da fila, já que estas mensagens podem ocupar classes diferentes de *buffers* no nó vizinho. A recusa de uma mensagem significa apenas que não existe espaço em uma determinada classe de *buffer*, mas que mensagens de classes diferentes podem ser enviadas. Por isso, as Unidades Despachadoras deste modelo continuam a enviar mensagens de classes diferentes da classe da mensagem recusada. As mensagens da mesma classe da mensagem recusada são atrasadas até que esta seja aceita, a fim de garantir que mensagens provenientes do mesmo nó fonte e com mesmo destino sejam recebidas na mesma ordem em que foram enviadas. O algoritmo desta unidade é semelhante ao da Uniade Despachadora do CVP I. Porém, a operação de pedido de atualização de fila na UDES do CVP I só ocorre quando esta unidade recebe uma confirmação de que a mensagem foi aceita, enquanto que no CVP II, mesmo quando uma mensagem é recusada, o pedido de atualização é feito, para que esta mensagem seja colocada no final da fila.

II.3.3 Unidade de Gerência de Buffer

A maior diferença na implementação dos dois modelos está no tratamento dos *buffers*. Neste modelo os *buffers* são divididos em classes. Toda mensagem recebida para armazenamento deve conter uma informação relativa à classe do *buffer* que permitirá saber onde esta deve, obrigatoriamente, ser mantida. Além disso, é necessária somente a fila de envio, já que não há mais a possibilidade de reserva de canais para circuitos.

Neste modelo, entretanto, é fundamental haver um controle de en-

vio de mensagens a nível de classes. Este controle é realizado da seguinte forma: por canal existe para cada classe uma informação que é a identificação do *buffer* que a mensagem recusada ocupa, caso alguma mensagem daquela classe tenha sido recusada. Desta forma, quando uma mensagem é recusada, esta não só é colocada no final da fila, como também tem sua identificação guardada para uma classe específica. Quando a Unidade de Gerência de Buffer recebe um pedido de recuperação de uma mensagem para um certo canal, esta verifica, para este canal, se a classe da mensagem recuperada possui já alguma mensagem recusada. Caso possua, comparam-se as identificações dos *buffers* ocupados por estas mensagens. Se forem diferentes esta mensagem é colocada no final da fila, se iguais, significa que uma nova tentativa de envio da mensagem deve ser feita. Quando uma mensagem anteriormente recusada é aceita, a Unidade Despachadora pede à Unidade de Gerência de Buffer não só para retirá-la da fila de envio, como para invalidar a informação de que para aquela determinada classe existe uma mensagem recusada, permitindo, desta forma que uma mensagem subseqüente de mesma classe seja enviada normalmente. O algoritmo desta unidade está na Figura II.10. As diferenças existentes em relação à Unidade de Gerência de Buffer do CVP I se referem, basicamnete, ao pedido de recuperação de instrução do *buffer* e pedido de atualização de fila feitos pela Unidade Despachadora. Neste algoritmo, é utilizada a tabela *wait.buffer* que fornece, para um certo canal e uma dada classe, a identificação do buffer ocupado pela instrução recusada, caso exista. Assim, antes de se enviar uma instrução recuperada para a Unidade Despachadora, verifica-se se alguma mensagem de mesma classe já foi recusada quando enviada pelo mesmo canal. Esta tabela é atualizada quando uma mensagem é recusada ou quando uma mensagem, anteriormente recusada, é aceita No exemplo da Figura II.11, tem-se nove *buffers* divididos em três classes e duas filas para os canais zero e um. Inicialmente, em (a) a fila do canal zero é formada pelas mensagens que ocupam os *buffers* 2, 4, 5, 1 e 6. Se for recebido um *nack* como resposta ao envio da mensagem do *buffer* 2, tem-se a situação ilustrada em (b), onde os elementos na fila passam a ter a ordem 4, 5, 1, 6 e 2, e onde é armazenado em *wait.buffer* o valor 2. Sendo as mensagens 4 e 5 enviadas sem problemas, a fila fica com ponteiros para 1, 6 e 2. A mensagem do *buffer* 1 não é enviada, colocando-se este ponteiro no final da fila (6, 2 e 1), pois para esta classe

(zero) já existe um mensagem esperando. Após o envio da mensagem do *buffer* 6, envia-se novamente a mensagem do *buffer* 2 e, se esta for aceita, elimina-se este ponteiro da fila e invalida-se a informação correspondente na tabela *wait.buffer*.

```
recebendo pedido da UD :
  se pedido de gravação de instrução em buffer então
     início
       se buffer diponível então
          grava instrução no buffer
       envia resposta sobre operação realizada para UD
     fim
  senão (pedido de atualização de fila)
    insere ponteiro do buffer na fila do canal correspondente
para cada canal i do CVP
   recebendo pedidos da UDES do canal i:
     se pedido de recuparação de instrução do buffer então
        início
          recupera primeira instrução da fila
          se buffer ocupado pela instrução ≠ wait.buffer[i,classe do buffer da instrução] entã
             transfere instrução para o final da fila
          envia resposta à UDES com instrução recuperada ou sinal de pedido não atendido
        fim
     senão (pedido de atualização de fila)
       se pedido de retirada de elemento da fila então
          retira primeiro elemento da fila do canal correspondente
       senão (pedido de transferência)
         início
           transfere primeiro elemento da fila para o final
           wait.buffer[i,classe da primeira instrução da fila] := buffer ocupado pela instrução
         fim
```

Figura II.10: Algoritmo da Unidade de Gerência de Buffer do CVP II

II.3.4 Estruturas de Dados

Além dos parâmetros vistos no CVP I ainda são necessários os seguintes parâmetros no CVP II :

len.class : número de *buffers* por classe.

num.buf.class : número de classes de *buffers*.

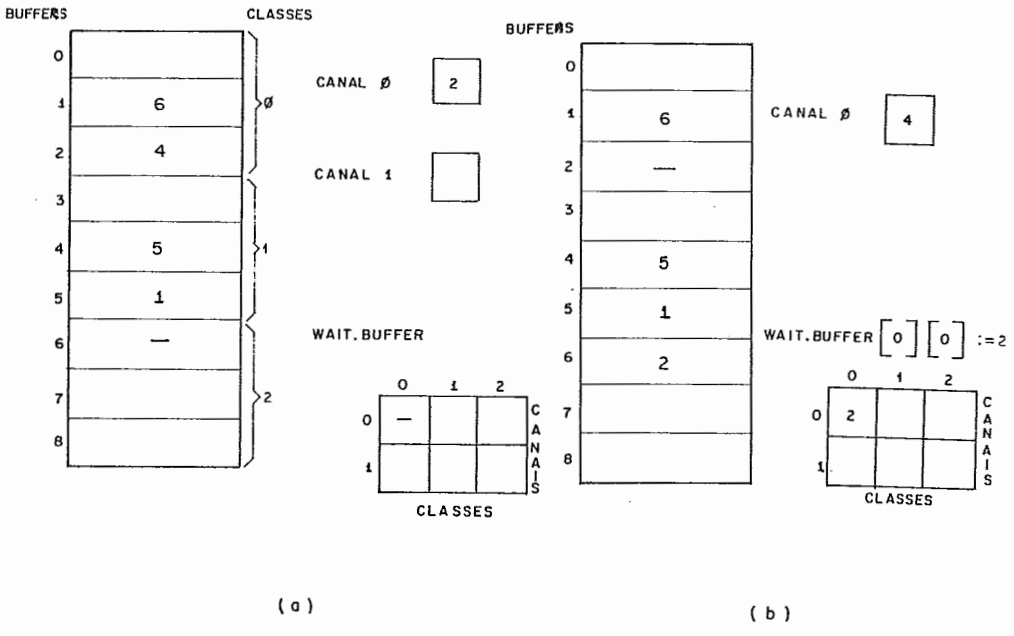

Figura II.11: Gerência de Buffers no CVP II

Em relação às estruturas de dados utilizadas pelas instruções de comunicação, todas as que existem para o CVP I continuam sendo necessárias no CVP II, com exeção das utilizadas para estabelecimento de circuito (*ask.counter*, *tree.convin.circuit*, *tree.convout.circuit*, *conv.circuit*, *in.chan*) .

Capítulo III

Instruções do Processador de Comunicação

As instruções do processador virtual de comunicação são executadas sem que sejam conhecidas as aplicações no nível mais alto executadas no processador hospedeiro. Estas instruções estão relacionadas aos esquemas de transporte *store-and-forward* e *circuit-switching*.

Os algoritmos das Unidades de Decisão, apresentados na Figura II.6 e II.9, descrevem as tarefas básicas necessárias na execução de qualquer instrução do CVP. No modo *store-and-forward* estas são: pedido à UGB para gravação de instrução em *buffer* e recebimento da resposta, pedido à UDES para envio de *ack* ou *nack*, a própria execução da instrução e pedido para UGB de atualização de fila. No modo *circuit-switching* tem-se a execução da instrução e o pedido à UDES para envio pelo circuito. Os algoritmos a seguir, embora especifiquem melhor a tarefa "executa instrução", incorporam também as outras tarefas, que são apresentadas, algumas vezes, de forma simplificada. Estas foram incluídas nos algoritmos a fim de fornecer descrições mais precisas das instruções, já que as execuções destas tarefas, normalmente, dependem de resultados de testes específicos presentes em cada uma das instruções.

III.1 Instruções no modo *store-and-forward*

Nesta seção são descritos os algoritmos utilizados pela Unidade de Decisão para execução das instruções transmitidas no modo *store-and-forward.* Estes algoritmos, da forma que estão apresentados, são válidos para ambos os modelos CVP I e CVP II, com exceção apenas dos algoritmos relativos às instruções de estabelecimento de circuito (ASK.CIRCUIT e ASK.VIRTUAL.TREE), que são executadas somente pelo CVP I .

Como já foi dito, a Unidade de Decisão recebe todas as instruções que chegam ao CVP. Quando estas são transmitidas no modo *store-and-forward* e quando o nó não é o destino final da instrução, a Unidade de Decisão verifica via Unidade de Gerência de Buffer se há disponibilidade de *buffer* para armazenar a instrução. Caso haja, é feito um pedido de envio de um sinal *ack* à Unidade Despachadora, referente ao canal pelo qual a instrução foi recebida, caso contrário, é feito um pedido de envio de um *nack.* Ambos os sinais, como já explicado, são enviados pelo caminho *st.chan.*

Algumas tarefas, comuns a todas as instruções, executadas pela Unidade de Decisão foram omitidas nestes algoritmos, a fim de se fornecer representações simples e claras das tarefas mais relevantes e que melhor caracterizam cada uma das instruções.

Assim, os testes de disponibilidade de *buffer* presentes nos algoritmos envolvem, na verdade, operações, não representadas, em que a Unidade de Decisão faz um pedido à Unidade de Gerência de Buffer para gravação da instrução em *buffer* e recebe desta um ponteiro do *buffer* utilizado para gravação da mensagem ou um sinal indicando que não há *buffer* disponível. No caso do CVP II o pedido é feito em relação a uma classe de *buffer* específica , na qual se pretende armazenar a mensagem, já no CVP I o pedido se refere a uma das duas partes em que o *buffer* é dividido. Da mesma forma, pode-se observar que são colocadas, nestes algoritmos, operações de envio de sinais *ack*, *nack* ou de mensagem para o processador hospedeiro, substituindo, os pedidos que a Unidade de Decisão realmente faz às Unidades Despachadoras para envio destas mensagens. Pois, como se sabe , somente Unidades

Despachadoras transmitem mensagens, que são recebidas da Unidade de Decisão ou recuperadas do *buffer* via Unidade de Gerência de Buffer.

Ainda devem ser consideradas as operações de atualização de filas. Como já exposto, a Unidade de Gerência de Buffer mantém filas para cada um dos canais com ponteiros para buffers com mensagens a serem transmitidas. No caso do CVP I tem-se duas filas para cada canal, no CVP II apenas uma fila por canal. Desta forma, a operação de atualização de filas envolve um pedido da Unidade de Decisão para a Unidade de Gerência de Buffer, precedido, no caso do CVP I, de um teste para se determinar qual das duas filas deve ser atualizada. Vale dizer que, no CVP I, caso o canal esteja reservado (o valor dado na tabela *ask.counter* relativa ao canal é diferente de zero), a Unidade de Decisão pede uma atualização da fila de espera, caso contrário, da fila de envio.

Ainda vale observar que instruções que objetivam reservar circuitos para transmissão de mensagens no modo *circuit-switching* utilizam o método de transporte *store-and-forward*, já que quando estas são executadas ainda não há circuito estabelecido.

1- SEND

Esta instrução realiza a troca de mensagens entre dois nós. Se N_i e N_j são os nós fonte e destino, respectivamente, e eles são conectados diretamente por um canal de comunicação, a troca de informação entre eles é direta. Se, por outro lado, eles não são vizinhos no sistema, é imprescindível que a informação seja roteada entre eles através de outros nós. Então, é necessário que haja informação de roteamento em todos os nós do sistema, indicando para onde enviar cada mensagem que chega. Para isso, é utilizada a tabela *route*. Esta instrução tem como operandos *dest* e *msg*, que correspondem ao nó destino e à mensagem a ser entregue. No algoritmo da Figura III.1, observa-se que, quando SEND chega ao destino ($dest = id$), *msg* é enviado para o hospedeiro; quando a instrução atinge um nó intermediário qualquer da rota, ela é trasnmitida pelo canal dado por *route*[*dest*]. A Figura III.2 ilustra a execução de SEND a partir do nó i até o nó k.

2- BROADN

```
se dest = id então
   início
     envia ack pelo canal de recebimento
     envia msg para o hospedeiro
   fim
senão
  início
    se buffer disponível então
       início
         envia ack pelo canal de recebimento
         atualiza fila do canal route[dest] com SEND
       fim
    senão
      envia nack pelo canal de recebimento
  fim
```

Figura III.1: Algoritmo da instrução SEND

Esta instrução faz *broadcast* de mensagens para vizinhos. Seu único operando é *msg*, a mensagem a ser transmitida. No algoritmo da Figura III.3, se o CVP recebe a instrução BROADN de seu hospedeiro, este a transmite para todos os CVP's vizinhos, caso contrário, envia para seu hospedeiro. Na Figura III.4 pode-se observar o *broadcast* para vizinhos a partir do nó i.

3- BROADNF

Bastante parecida com a instrução anterior, exceto que cada vizinho trasnmite de volta uma confirmação ao receber a mensagem. Esta requer os operandos *source* e *msg*, que são, respectivamente, o nó que origina o *broadcast* e a mensagem a ser enviada. Na Figura III.5, pode-se observar que o CVP que origina o *broadcast* mantém o controle do recebimento das confirmações de outros CVP's através de $broadsnf$, ou seja, este incrementa $broadsnf[c]$, quando envia a instrução por cada canal c. Um CVP qualquer, ao receber BROADNF de outro CVP, envia *msg* para o hospedeiro e envia a instrução de volta para o CVP que originou o *broadcast*. O CVP origem do *broadcast*, ao receber a instrução de outro CVP por um canal c, decrementa $broadsnf[c]$, comparando este valor com os demais valores de $broadsnf$ relativos aos outros canais. Se para todos os outros canais os valores de

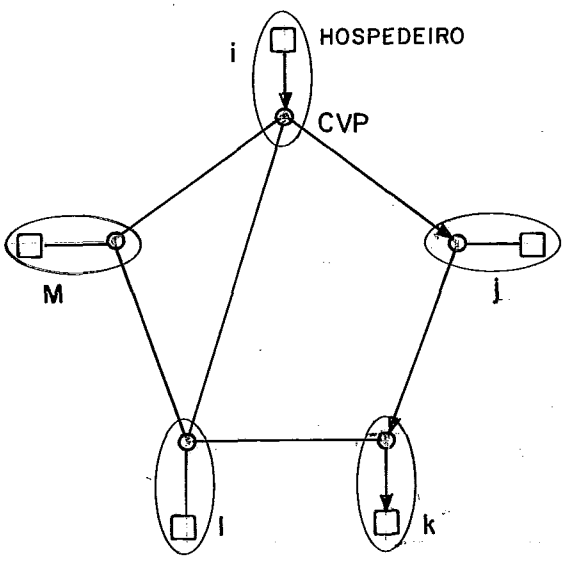

Figura III.2: Instrução SEND

broadsnf forem menores ou iguais ao valor de *broadsnf* relativo ao canal pelo qual a instrução foi recebida, este CVP envia *msg* para o hospedeiro, como uma aviso de que todos os vizinhos receberam a instrução. A Figura III.6 ilustra o *broadcast* para vizinhos originado no nó i que recebe de volta as instruções enviadas, como confirmações.

4- BROADT

Esta instrução pressupõe a existência de uma árvore geradora no sistema (árvore que cobre todos os nós). A execução de BROADT permite que uma informação seja compartilhada por todo o sistema, através de uma árvore geradora, cuja raiz corresponde ao nó que origina o *broadcast.* Para isso, em cada CVP, é necessário que haja informação sobre quais os canais que saem do nó para formar a árvore. Este é o tipo de *broadcast* que envia o menor número de mensagens. Os operandos t e *msg* são necessários, sendo t a árvore na qual o *broadcast* deve ser feito e *msg* a mensagem a ser difundida. No algoritmo (Figura III.7), um CVP qualquer, ao receber BROADT, envia *msg* para seu hospedeiro e envia a instrução por todo canal c, para o qual o valor de $tree.broad[c]$ é igual a TRUE, indicando que o canal

```
se instrução recebida do hospedeiro então
   se buffer disponível então
      início
        envia ack pelo canal de recebimento
        para todo canal c faça
           atualiza fila do canal c com BROADN
      fim
   senão
     envia nack pelo canal de recebimento
senão
  início
    envia ack pelo canal de recebimento
    envia msg para o hospedeiro
  fim
```

Figura III.3: Algoritmo da instrução BROADN

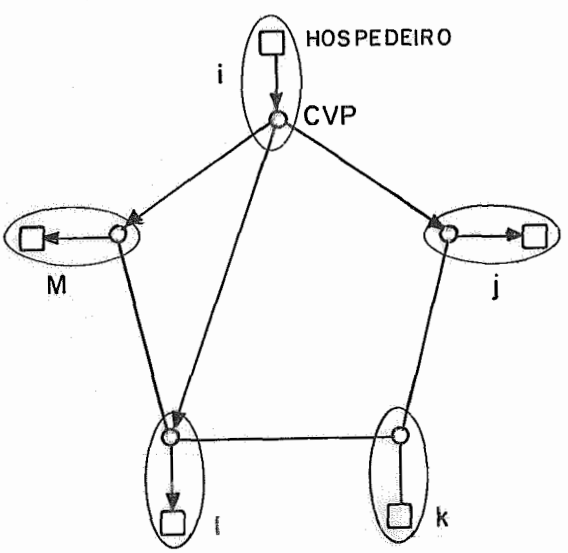

Figura III.4: Instrução BROADN

```
se instrução recebida do hospedeiro então
   início
     se buffer disponível então
        início
          envia ack pelo canal de recebimento;
          para todo canal c faça
             início
               incrementa broadsnf[c]
               atualiza fila do canal c com BROADNF
             fim
        fim
     senão
       envia nack pelo canal de recebimento
   fim
senão
  início
    se source ≠ id então
       início
         se buffer disponível então
            início
              envia ack pelo canal de recebimento
              envia msg para o hospedeiro
              atualiza fila do canal de recebimento com BROADNF
            fim
         senão
           envia nack pelo canal de recebimento
       fim
    senão
      início
        envia ack pelo canal de recebimento
        decrementa broadsnf[canalderecebimento]
        se broadsnf[c] ≤ broadsnf[canalderecebimento]
           para todo c ≠ canal de recebimento então
           envia msg para o hospedeiro
      fim
```

Figura III.5: Algoritmo da instrução BROADNF

c é um ramo de saída da árvore. Na Figura III.8, é apresentada a execução desta instrução a partir de uma árvore geradora de raiz no nó i.

5-CONVT

Como já dito no Capítulo I, o termo *convergecast* indica um fluxo de mensagens no sentido oposto ao do *broadcast*, ou seja, em direção a um nó a partir de todos os outros. Da mesma forma que o *broadcast* descrito acima, aqui também é necessária uma árvore geradora. O *convergecast* é iniciado por todas as folhas desta árvore assincronamente. Em qualquer nó não folha, a mensagem de *convergecast* é encaminhada no único canal de saída da árvore do nó, quando este tiver recebido as mensagens de *convergecast* de todos os canais de entrada da árvore, sendo necessário que exista informação local para manter este controle. Esta instrução requer os operandos t (a árvore de *convergecast* em uso) e *msg* (a mensagem). No algorimto (Figura III.9), observa-se que um CVP que recebe esta instrução do hospedeiro (nó folha) transmite-a pelo canal c, tal que $tree.convout[t, c]$ seja TRUE (canal de saída da árvore). Um CVP qualquer (nó não folha), ao receber esta instrução de outro CVP pela primeira vez (quando $conv[t,$ *canal de recebimeto*$]$ é igual a zero), deve incrementar $conv[t, c]$ para todo canal c cujo valor de $tree.convin[t, c]$ seja TRUE, caso existam outros nós pelos quais este CVP ainda espera instruções de CONVT. Desta forma, ao receber uma intrução CONVT subseqüente relativa à mesma árvore t por um canal c, $conv[t, c]$ é decrementado e é verificado se a instrução já foi recebida por todos os outros canais para os quais *tree.convin* é igual a TRUE (canais de entrada da árvore), ou seja é testado se para todo canal c, onde $tree.convin[t, c]$ é TRUE, o valor $conv[t, c]$ é menor ou igual ao valor de *conv* relativo ao canal de recebimento da instrução. Caso seja, *msg* é enviado para o hospedeiro e a instrução é enviada pelo canal c, tal que $tree.convout[t, c]$ seja TRUE. É ilustrada na Figura III.10 a execução de um *convergecast* a partir dos nós k e l.

6- BROADF

Esta instrução implementa *broadcast* de mensagens por enchente. *Broadcast* por enchente é potencialmente a forma mais rápida de *broadcast*, embora bastante onerosa para o sistema em termos do número de mensagens necessárias.

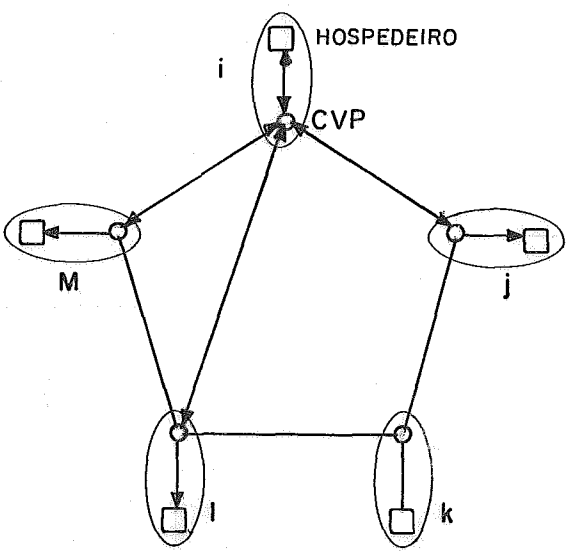

Figura III.6: Instrução BROADNF

se *buffer* disponível **então**
  início
    envia *ack* pelo canal de recebimento
    se instução não recebida do hospedeiro **então**
      envia *msg* para o hospedeiro
    para todo canal c tal que $tree.broad[t, c]$ seja TRUE **faça**
      atualiza fila do canal c com **BROADT**
  fim
senão
  envia *nack* pelo canal de recebimento

Figura III.7: Algoritmo da instrução BROADT

O nó que inicia o *broadcast* envia esta instrução para todos os seus vizinhos. Todos os outros nós fazem o mesmo ao receber a mensagem pela primeira vez. Para evitar que os nós permaneçam propagando, continuamente, mensagens relativas a um mesmo *broadcast*, são necessárias informações de controle nos nós. Seus operandos são *source* e *msg*, onde *source* é o nó que inicia o *broadcast* e *msg* é a mensagem a ser enviada. Na Figura III.11, o CVP ao receber esta instrução do hospedeiro a envia por todos os canais que o conectam a outros CVP's. Um CVP qualquer (não origem do *broadcast*) , ao receber esta instrução pela primeira vez, por um canal c, (quando então o valor $flood[source, c]$ é igual a 0), envia BROADF em todos os canais incrementando os valores da tabela $flood$ correspondentes a estes canais . Ao receber um BROADF subseqüente, este CVP decrementa a entrada de $flood$ relativa ao canal pelo qual esta instrução foi recebida, a fim de que os próximos *broadcasts* por enchente sejam tratados adequadamente, ou seja, para que possa ser detectado o início de um novo *broadcast* ($flood[source, c]$ igual a zero). O CVP que origina o *broadcast*, ao receber esta instrução de outro CVP, a ignora. É importante observar que esta instrução BROADF não permite início concorrente do mesmo *broadcast* por diversas fontes. Como exemplo, a Figura III.12 ilustra uma situação, onde o nó i inicia o *broadcast*, ou seja, este envia a instrução BROADF para os seus vizinhos j, l e m, que executam o mesmo procedimento ao receberem a instrução do nó i.

7- BROADFF

Esta instrução faz *broadcast* de mensagens por enchente com realimentação. Esta forma de *broadcast* possui todas as propriedades do mecanismo anterior e ainda permite que o nó origem saiba quando a informação atingiu todos os outros nós do sistema. O nó que inicia o *broadcast* envia a mensagem para todos os vizinhos. Qualquer outro nó recebendo esta pela primeira vez executa o mesmo procedimento, mas não envia para o vizinho do qual o nó recebeu a mensagem, por exemplo N_i. Ao receber uma confirmação de todos os vizinhos para os quais a mensagem foi enviada, uma confirmação é enviada para N_i. Quando o nó origem recebe a confirmação de todos os vizinhos, este sabe que todos os outros nós da rede receberam a mensagem. Os operandos desta instrução são *source* e *msg*, onde *source* é o nó origem e *msg* a mensagem sendo difundida. No algoritmo (Figura III.13 e

Figura III.14), um CVP ao receber esta instrução do hospedeiro envia-a por todo canal c que o conecta a outro CVP, incrementando $floodfb[source, c]$. Um outro CVP (estando conectado a outros CVP's), ao receber a primeira instrução BROADFF relativa a um novo *broadcast* (quando então a entrada na tabela $floodfb$ relativa ao canal pelo qual a instrução foi recebida e a *source* é igual a zero) , envia *msg* para o hospedeiro, incrementa $last[source]$, atualiza $pending[source, last[source]]$ com o canal pelo qual a instrução foi recebida e envia a instrução por todo canal c diferente do canal pelo qual a instrução foi recebida incrementando $floodfb[source, c]$. Ao receber um BROADFF subseqüente, este CVP decrementa a entrada na tabela $floodfb$ relativa ao canal pelo qual a instrução foi recebida e verifica se o valor na tabela $floodfb$ relativo ao canal pelo qual a instrução chegou é maior ou igual aos valores da tabela para os outros canais. Caso seja, a instrução BROADFF é enviada pelo canal dado por $pending[source, first[source]]$ e $first[source]$ é incrementado. Vale observar que quando uma instrução BROADFF chega por um canal c, para o qual $floodfb[c]$ é igual a zero, tem-se a execução de um novo *broacast* e, por isso, deve-se guardar em $pending[source, last[source]]$ (após icrementar $last[source]$) o canal de recebimento. O valor de $first[source]$ será um ponteiro para esta localização, quando todas as outras confirmações de *broadcats* anteriores já tiverem sido enviadas. É importante dizer que, para um CVP que possui apenas um vizinho, não é necessária a utilização da tabela *pending*, já que o canal pelo qual a confirmação deve ser enviada é sempre o mesmo. Na Figura III.15 é ilustrado um *broadcsat* iniciado pelo nó i, onde as linhas contínuas indicam as transmissões das instruções BROADFF como confirmações e as linhas tracejadas as transmissões devido à própria propagação das mensagens.

8- ASK.CIRCUIT

Esta instrução faz a reserva de um conjunto de canais, formando um circuito para um determinado destino. Os operandos desta instrução são *source* e *dest* que são respectivamente os nós origem e destino do circuito sendo estabelecido. Da mesma forma que na instrução SEND, é necessária informação de roteamento em todos os nós do sistema, indicando qual o canal a ser reservado. Para isso, também, é consultada a tabela *route*. A fim de impedir que um mesmo canal seja reservado

simultaneamente por mais de um circuito, é necessário manter controle do número de pedidos de reserva de um canal, o que é feito através da tabela *ask.counter*. Esta tabela, para cada canal, fornece um valor correspondente ao número de pedidos feitos para reserva deste. Desta forma, se este valor for igual a zero, sabe-se que o canal não foi reservado. Quando esta instrução é executada pelo nó destino do circuito, é iniciado o envio, pelo caminho de estado do circuito, *st.cir.chan*, da confirmação de circuito estebelecido, utilizando para determinação do caminho de volta a tabela *in.chan*. Esta tabela, conforme mostra o algoritmo da (Figura III.16), é carregada durante o estabelecimento do circuito e é utilizada, quando o CVP, tendo recebido uma confirmação de circuito estabelecido, desejar retransmiti-la pelo canal que o conecta ao outro nó vizinho do circuito, consultando, para obter este canal, $in.chan[source, dest]$. Na Figura III.17, o nó i envia a instrução para o nó k (destino do circuito) que transmite a confirmação de circuito estabelecido de volta pelo mesmo percurso.

9- ASK.VIRTUAL.TREE

Esta instrução pressupõe a existência de uma árvore cuja raiz corresponde ao nó que inicia o pedido. Em cada nó, é necessária informação sobre quais os canais que saem do nó para formar a árvore, o que é fornecido por *tree.broad*, que como já visto também é utilizado pela instrução BROADT. Esta instrução faz as reservas dos canais que formam a árvore geradora . Ao final, os nós folhas da árvore executam um *convergecast* com a confirmação da reserva. Estas mensagens de *convergecast* são enviadas pelo caminho de estado de circuito, *st.cir.chan*, e utilizam as tabelas *tree.convin.circuit* e *tree.convout.circuit*, que são carregadas durante o estabelecimento do circuito. Esta instrução possui um operando t que é a árvore na qual o circuito deve ser reservado. O algoritmo desta instrução é dado na Figura III.18. Na Figura III.19, o nó i inicia o estabelecimento do circuito e os nós k e l fazem o *convergecast* da confirmação de circuito estabelecido.

III.2 Instruções no modo *circuit-switching*

Estas instruções são executadas apenas pelo CVP I, que além de funcionar com o modo *store-and-forward*, funciona também com o modo de transporte *circuit-switching*. A Unidade de Decisão envia estas instruções diretamente para as Unidades Despachadoras para serem transmitidas pelos canais reservados pelo circuito já então estabelecido.

1- SEND.CIRCUIT

Esta instrução pressupõe a existência de um circuito pelo qual a mensagem deve ser enviada. Esta necessita do operando *dest*, que é o destino do circuito, e, quando recebida do hospedeiro, *msg*, a mensagem. Ao executá-la, a Unidade de Decisão determina o canal pelo qual os *flits* devem ser roteados. A partir daí esta recebe e envia *flits* até que seja detectado final de mensagem.(Figura III.20)

2- FREE.CIRCUIT

Esta instrução possibilita a liberação dos canais reservados pelo circuito e transfere as mensagens das filas de espera pelos canais para filas de mensagens a serem despachadas. Se houver pelo menos um pedido de reserva de canal pendente na fila de espera, a transferência ocorre até que se encontre esta instrução. Neste caso, as instruções seguintes ao pedido de reserva, na fila de espera, devem permanecer nesta, até que uma instrução de liberação do canal seja novamente executada. Esta instrução necessita do operando *dest*, que é o destino do circuito. Na Figura III.21, verifica-se que FREE.CIRCUIT utiliza a tabela *ask.counter* para manter o controle dos pedidos de reservas pendentes. A tabela *ask.counter* é fundamental para que a Unidade de Decisão possa fazer os pedidos de atualização das filas de espera e de envio para a Unidade de Gerência de Buffer apropriadamente. Ou seja, se um canal está reservado para algum circuito, a entrada na tabela *ask.counter* correspondente é diferente de zero, o que permite que a Unidade de Decisão, através de uma consulta à esta tabela, opte pela atualização da fila de espera ao invés da fila de envio. Vale dizer que, para cada instrução de pedido de reserva de um canal, deve ser executada uma instrução para liberação do canal, a fim de evitar que instruções

permaneçam indefinidamente na fila de espera.

3-SEND.TREE

Esta instrução pressupõe a existência de um circuito em forma de árvore pelo qual a mensagem deve ser enviada e necessita do operando *t*, que é a árvore do circuito, e, quando recebida do hospedeiro, *msg*, a mensagem. Ao executá-la, a Unidade de Decisão determina os canais pelos quais os *flits* devem ser roteados, recebendo e enviando *flits* até ser detectado final de mensagem. (Figura III.22)

4-FREE.VIRTUAL.TREE

Da mesma forma que FREE.CIRCUIT, esta instrução também possibilita a liberação dos canais reservados pelo circuito, que neste caso é uma árvore, e transfere as mensagens das filas de espera pelos canais para filas de mensagens a serem despachadas. Devendo-se considerar, como no caso anterior, os pedidos de reservas de canais pendentes. Esta instrução necessita do operando *t*, que é a árvore do circuito e, como mostra a Figura III.23, também utiliza a tabela *ask.counter*.

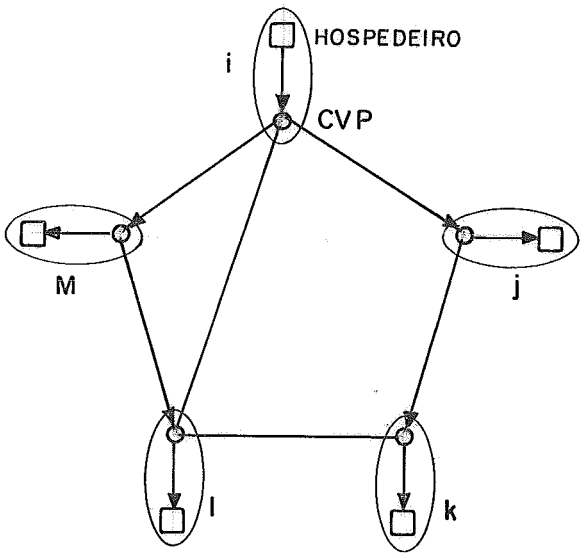

Figura III.8: Instrução BROADT

```
se instrução recebida do hospedeiro então
   se buffer disponível então
      início
        envia ack pelo canal de recebimento
        atualiza fila do canal c tal que tree.convout[t, c] seja TRUE com CONVT
      fim
   senão
     envia nack pelo canal de recebimento
senão
  se conv[t, canal de recebimento] = 0 então
     se para pelo menos um canal c ≠ canal de recebimento,
        tree.convin[t, c] = TRUE então
        início
          envia ack pelo canal de recebimento
          para cada canal c tal que tree.convin[t, c] seja TRUE faça
            incrementa conv[t, c]
     senão
       se buffer disponível então
          início
            envia ack pelo canal de recebimento
            envia msg para hospedeiro
            atualiza fila do canal c tal que tree.convout[t, c] seja TRUE com CONVT
       senão
         envia nack pelo canal de recebimento
  senão
    início
      decrementa conv[t, canal de recebimento]
      se para pelo menos um canal c, conv[t, c] > conv[t, canal de recebimento] então
         envia ack pelo canal de recebimento
      senão
        se buffer disponível então
           início
             envia ack pelo canal de recebimento
             envia msg para hospedeiro
             atualiza fila do canal c tal que tree.convout[t, c] seja TRUE com CONVT
           fim
        senão
          envia nack pelo canal de recebimento
    fim
```

Figura III.9: Algoritmo da instrução CONVT

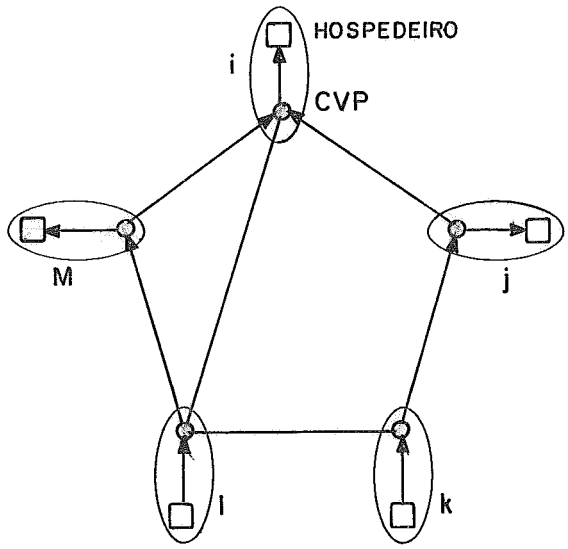

Figura III.10: Instrução CONVT

```
se instrução recebida do hospedeiro então
   início
     se buffer disponível então
        início
          envia ack pelo canal de recebimento
          para todo canal c faça
             atualiza fila do canal c com BROADF
        fim
     senão
       envia nack pelo canal de recebimento
   fim
senão
  início
    se source ≠ id então
       se flood[source, canal de recebimento] ≠ 0 então
          início
            envia ack pelo canal de recebimento
            decrementa flood[source, canal de recebimento]
          fim
       senão
         se buffer disponível então
            início
              envia ack pelo canal de recebimento
              envia msg para hospedeiro
              para todo canal c faça
                 incrementa flood[source, c]
                 atualiza fila do canal c com BROADF
            fim
         senão
           envia nack pelo canal de recebimento
    senão
      envia ack pelo canal de recebimento
  fim
```

Figura III.11: Algoritmo da instrução BROADF

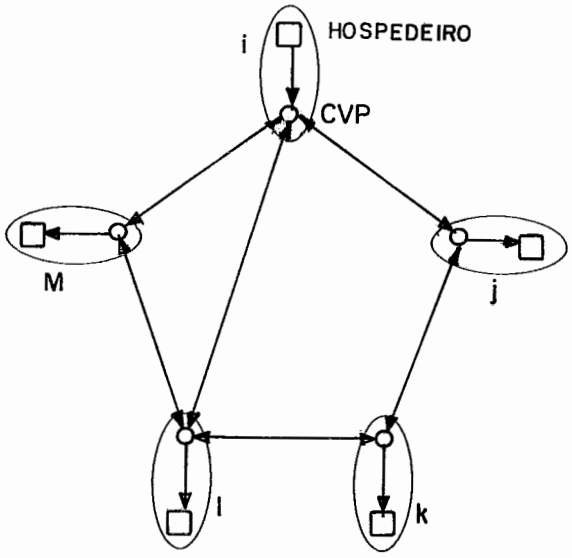

Figura III.12: Instrução BROADF

```
se instrução recebida do hospedeiro então
   início
     se buffer disponível então
        início
          envia ack pelo canal de recebimento
          para todo canal c faça
             início
               incrementa floodfb[source, c]
               atualiza fila do canal c com BROADFF
             fim
        fim
     senão
       envia nack pelo canal de recebimento
   fim
senão
  se source ≠ id então
     início
       se floodfb[source, i] = 0 então
          se não existem outros CVP's conectados então
             se buffer disponível então
                início
                  envia ack pelo canal de recebimento
                  envia msg para o hospedeiro
                  atualiza fila do canal de recebimento com BROADFF
                fim
             senão
               envia nack pelo canal de recebimento
```

Figura III.13: Algoritmo da instrução BROADFF

```
            senão (existem outros CVP's conectados)
              se buffer disponível então
                 início
                   envia ack pelo canal de recebimento
                   envia msg para o hospedeiro
                   last[source] := resto da divisão de (last[source] + 1) por floodsfb
                   pending[source, last[source]] := canal de recebimento
                   para todo canal c ≠ canal de recebimento faça
                      início
                        atualiza fila do canal c com BROADFF
                        incrementa floodfb[source, c]
                      fim
                 fim
              senão
                 envia nack pelo canal de recebimento
        senão (flood[source, i] ≠ 0)
          se para pelo menos um canal c,
             floodfb[source, c] > floodfb[source, canal de recebimento] então
             início
               decrementa floodbf[source, canal de recebimento]
               envia ack pelo canal de recebimento
             fim
          senão
            se buffer disponível então
               início
                 envia ack pelo canal de recebimento
                 decrementa floodbf[source, canal de recebimento]
                 atualiza fila do canal pending[source, first[source]] com BROADFF
                 incrementa first[source]
               fim
            senão
              envia nack pelo canal de recebimento
    fim
senão (source = id)
  início
    envia ack pelo canal de recebimento
    decrementa floodfb[source, canal de recebimento]
    se para todo canal c, floodfb[source, c] ≤ floodfb[source, canal de recebimento] então
       envia msg para hospedeiro
  fim
```

Figura III.14: Continuação do algoritmo da instrução BROADFF

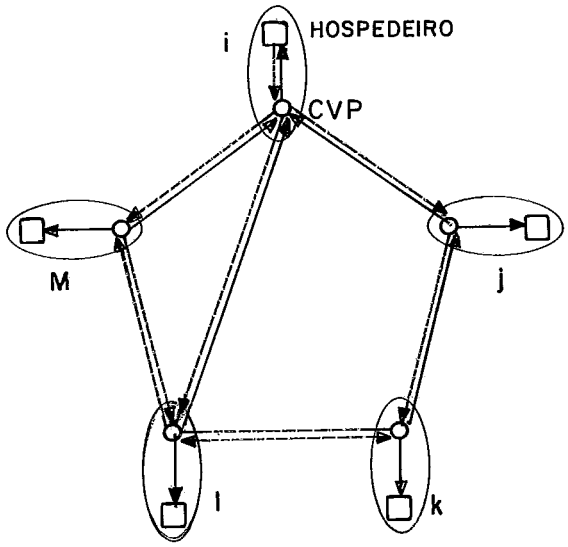

Figura III.15: Instrução BROADFF

se $dest \neq id$ **então**
  se *buffer* disponível
    início
      envia *ack* pelo canal de recebimento
      incrementa $ask.counter[route[dest]]$
      $in.chan[source, dest] :=$ *canal de recebimento*
      atualiza fila do canal $route[dest]$ com **ASK.CIRCUIT**
    fim
  senão
    envia *nack* pelo canal de recebimento
senão
  início
    envia *ack* pelo canal de recebimento
    executa o envio da confirmação de circuito reservado para *source*
  fim

Figura III.16: Algoritmo da instrução ASK.CIRCUIT

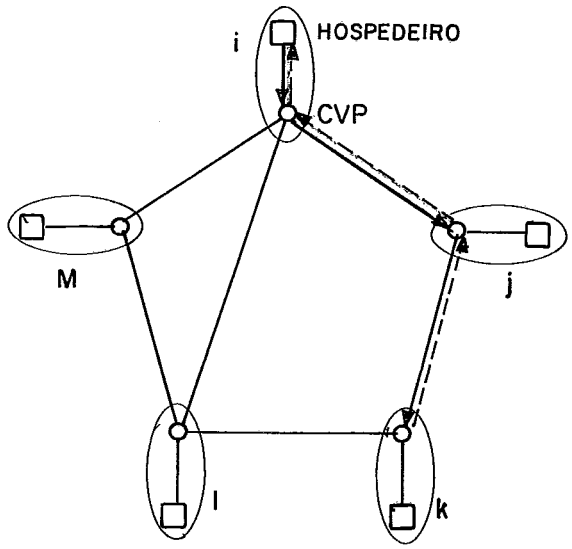

Figura III.17: Instrução ASK.CIRCUIT

se para pelo menos um canal c, *tree.broad[t,c]* = TRUE **então**
  se *buffer* disponível **então**
    início
      envia *ack* pelo canal de recebimento
      tree.convout.circuit[t, *canal de recebimento*] := TRUE
      para cada canal c tal que *tree.broad*[t, c] seja TRUE **faça**
        início
          incrementa *ask.counter*[c]
          atualiza fila relativa ao canal c com **ASK.VIRTUAL.TREE**
          tree.convin.circuit[t, c] := TRUE
        fim
    fim
  senão
    envia *nack* pelo canal de recebimento
senão
  início
    envia *ack* pelo canal de recebimento
    executa *convergecast* da confirmação de reserva de circuito de árvore

Figura III.18: Algoritmo da instrução ASK.VIRTUAL.TREE

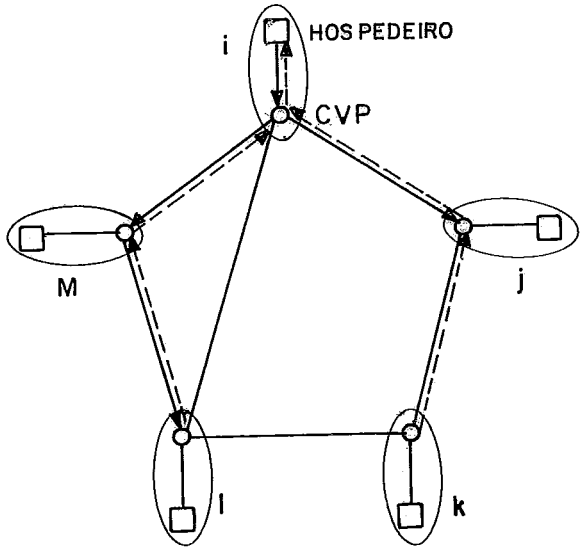

Figura III.19: Instrução ASK.VIRTUAL.TREE

```
se instrução recebida do hospedeiro então
   início
     envia SEND.CIRCUIT no canal route[dest]
     repita
       envia flits
     até final da mensagem
   fim
senão
  se dest ≠ id então
     início
       envia SEND.CIRCUIT no canal route[dest]
       repita
         recebe flits
         envia flits
       até final da mensagem
     fim
  senão
     início
       repita
         recebe flits
       até final da mensagem
       envia msg para o hospedeiro
     fim
```

Figura III.20: Algoritmo da instrução SEND.CIRCUIT

```
envia FREE.CIRCUIT no canal route[dest]
decrementa ask.counter[route[dest]]
se ask.counter[route[dest]] ≠ 0 então
   transfere instruções da fila de espera do canal route[dest] para a fila
     de envio até que a instrução seja ASK.CIRCUIT ou ASK.VIRTUAL.TREE
senão
  transfere todas as instruções da fila de espera do canal route[dest]
    para a fila de envio
```

Figura III.21: Algoritmo da instrução FREE.CIRCUIT

envia **SEND.VIRTUAL.TREE** em todos os canais c tal que $tree.broad[t, c]$ seja TRUE
se instrução recebida do hospedeiro **então**
  repita
    envia *flits*
  até final da mensagem
senão
  início
    repita
      recebe *flits*
      envia *flits* (se nó não folha)
    até final da mensagem
    envia *msg* para o hospedeiro
  fim

Figura III.22: Algoritmo da instrução SEND.VIRTUAL.TREE

para cada canal c tal que $treebroad[t, c]$ seja TRUE **faça**
  início
    envia **FREE.VIRTUAL.TREE** no canal c
    decrementa $ask.counter[c]$
    se $ask.counter[c] \neq 0$ **então**
      transfere instruções da fila de espera do canal c para a fila de envio
        até que a instrução seja **ASK.CIRCUIT** ou **ASK.VIRTUAL.TREE**
    senaõ
      transfere todas as instruções da fila de espera do canal c
        para a fila de envio
  fim

Figura III.23: Algoritmo da instrução FREE.VIRTUAL.TREE

Capítulo IV

Deadlocks de Comunicação

Ambos os modos de transmissão *store-and-forward* e *circuit-switching* são passíveis de ocorrência de *deadlock* de comunicação. No caso dos sistemas *store-and-forward*, *deadlock*s podem ocorrer quando o tráfego de mensagens fica impedido, porque dois ou mais nós esgotam todo o espaço de *buffer* disponível. Um ciclo é formado quando cada nó no sistema não pode transmitir sequer uma mensagem, porque não existem *buffers* disponíveis para receber a mensagem, e nenhum *buffer* pode ser liberado, porque as mensagens não podem ser transmitidas. A situação é ilustrada na Figura IV.1, onde são mostrados os processadores i, j, k e l. Se todos os *buffers* de j contêm mensagens destinadas a l, e assim por diante, a situação de *deadlock* está definida.

Em sistemas onde mensagens são transmitidas por *circuit-switching*, *deadlock* pode ocorrer quando uma rota fonte-destino não pode ser estabelecida porque uma das ligações necessárias está sendo utilizada por uma outra rota. Na Figura IV.2, se uma rota de i até l precisa ser estabelecida, e também uma de k a j, então *deadlock* pode surgir se a ligação de i' a j' estiver presa por uma das rotas enquanto a ligação de k' a l' estiver presa pela outra [1].

Muitos algoritmos de prevenção de *deadlock* têm sido desenvolvidos para redes de comunicação *store-and-forward*. A maior parte deles é baseada no conceito de um *pool* de *buffers* estruturado. Os *buffers* de mensagens em cada nó da rede são particionados em classes e o assinalamento de *buffers* a mensagens é restringido para definir uma ordem parcial nas classes de *buffers* que uma mensagem ocupa

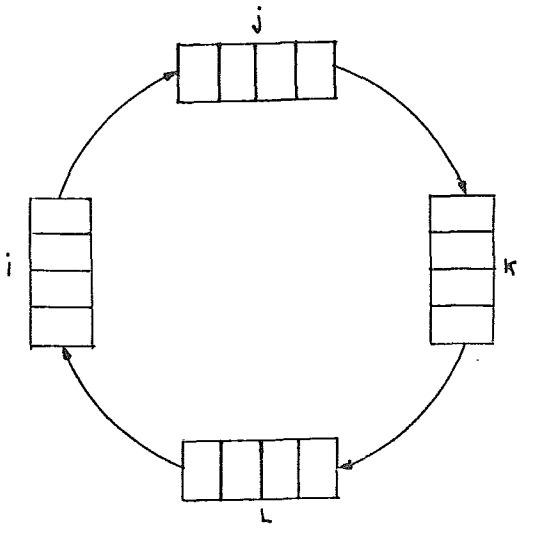

Figura IV.1: Deadlock de Comunicação no modo store-and-forward

durante o seu percurso. Por exemplo, assumindo que cada nó tenha no mínimo x *buffers*, onde x é o número máximo de canais percorridos por qualquer percurso no sistema, o *pool* de *buffers* do nó é particionado em x classes $0, 1, \cdots, x-1$. Uma mensagem chegando em um nó, tendo que passar ainda por i nós intermediários, deve ser colocada em um *buffer* de classe i. Consegue-se evitar *deadlock* porque todas as mensagens na rede podem ser entregues aos seus respectivos destinos, e todos os *buffers* que armazenam mensagens podem, desta forma, ser liberados, assim, todos os *buffers* da classe 0 podem ser liberados porque as mensagens que estes armazenam podem imediatamente ser transmitidas para seus destinos finais. Como todos os *buffers* da classe 0 podem ser liberados, *buffers* da classe 1 também podem ser liberados desalocando primeiro *buffers* da classe 0 e transmitindo mensagens da classe 1 para estes *buffers* da classe 0, quando então as mensagens podem deixar a rede. Aplicando este argumento recursivamente, é fácil ver que todas as mensagens na rede podem ser transmitidas para seus respectivos destinos. A principal desvantagem deste método é que redes muito grandes podem necessitar de muito espaço em *buffer*. O método implementado no CVP utiliza o algoritmo de "contagem de inversão de direção" [9] para determinação das classes de *buffers* que devem ser pe-

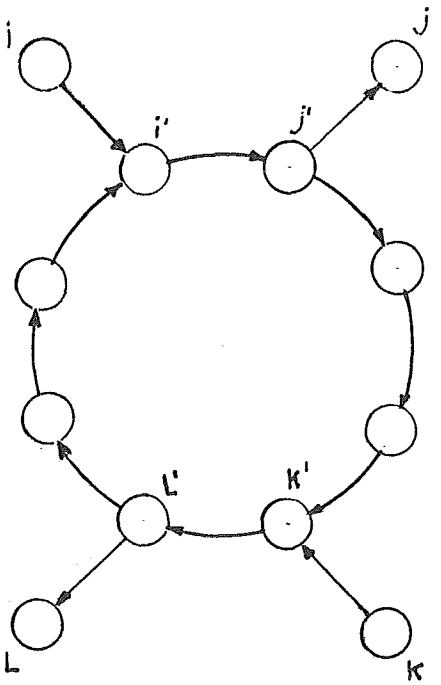

Figura IV.2: Deadlock de Comunicação no modo circuit-switching

didas sucessivamente por uma mensagem em um caminho na rede. Este algoritmo baseia-se na idéia de que, existindo números estáticos distintos associados aos nós da rede, a seqüência de números de nós que uma mensagem encontra sucessivamente no seu caminho através da rede, em geral, não cresce ou decresce monotonicamente. Ou seja, existirão "picos", que são nós cujos números são maiores do que os nós predecessores e sucessores na rota da mensagem; e "vales", que representam nós cujos identificadores são menores do que os nós adjacentes na rota. Quando uma mensagem entra no n-ésimo nó da sua rota, esta pede *buffers* da classe $npv(n)$, onde npv é o número de picos e vales encontrados no caminho, indicando o número de inversões de direção pelo qual uma rota já passou. Este é computado conhecendo-se $npv(n-1)$, que é concatenado à mensagem pelo nó anterior, e o número identificador deste n-ésimo nó. Para todas as mensagens nos nós origens o npv é igual a zero e considera-se neste primeiro nó da rota que a tendência da seqüência de números de nós é crescente. Desta forma, se $npv(n-1)$ for par e o n-ésimo nó da rota for maior do que o $(n-1)$-ésimo nó $(n > (n-1))$ ou se $npv(n-1)$ for ímpar e $n < (n-1)$, então tem-se $npv(n)$ igual a $npv(n-1)$, senão, se $npv(n-1)$ for par e $(n < (n-1)$, ou se $npv(n-1)$ for ímpar e $n > (n-1)$, então tem-se $npv(n)$ igual a $npv(n-1)$

mais um. Assim, por exemplo, se uma mensagem em uma dada rota passa pelos nós 5, 6, 1, 2 e 3, esta é alocada nos *buffers* de classes 0, 0, 1, 2 e 2, respectivamnete em cada um destes nós, onde o nó 6 é um pico e o nó 1 é um vale. Este método de classificação, na maior parte dos casos, requer um número menor de classes de *buffers* do que outros métodos, já que $npv(n)$ geralmente é menor do que n. O número total de picos e vales em qualquer rota é no máximo igual a duas vezes o número de vales mais um, já que picos e vales se alternam. Então, se K é um limite superior para o número de vales que podem ocorrer em qualquer rota, é necessário, em todo nó, um número de classes de *buffers* igual a duas vezes o valor de K mais um.

Esta solução não é aplicável a redes *circuit-switching*, já que *flits* de uma mensagem não podem ser intercalados com *flits* de outras mensagens. Estando o canal reservado, todos os *flits* de uma mensagem devem ser aceitos antes que quaisquer outros *flits* de outras mensagens sejam aceitos pelo canal. No roteamento *circuit-switching*, portanto, não é possível restringir o assinalamento de *buffers* a mensagens, como no caso anterior, para quebrar *deadlock*, ao invés disso, deve-se restringir o roteamento, o que também pode ser aplicado em redes *store-and-forward*.

No CVP é consultada a tabela *route* para roteamento de mensagens. Caso sejam executadas instruções no modo *circuit-switching* espera-se que esta tabela esteja inicializada apropriadamente a fim de se evitar *deadlock*. Os valores desta tabela poderiam ser dados por um algoritmo de roteamento livre de *deadlock*. Por exemplo, existe o algoritmo *e-cube* [8] para prevenir *deadlock* em rede hipercúbica, também chamada n-cúbica binária, que consiste de N (igual a 2^n) nós interconectados através de canais ponto-a-ponto, obedecendo a seguinte regra: dois nós cujos endereços binários diferem em exatamente uma posição de *bit* são conectados por um canal, onde cada posição de *bit* corresponde a uma dimensão da rede .Um exemplo de um hipercubo de dezesseis nós é mostrado na Figura IV.3) .

Assim, se um CVP estivesse conectado a outros CVP's formando uma rede hipercúbica, poder-se-ia se ter a tabela de roteamento inicializada através do algoritmo *e-cube*. Neste algoritmo, o endereço do nó destino da mensagem é comparado ao endereço do nó em que a mensagem se encontra. Qualquer uma das

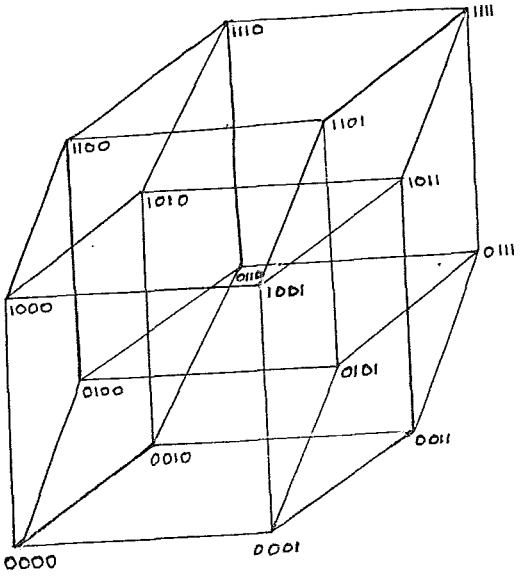

Figura IV.3: Um hipercubo de dezesseis nós

posições do *bit*, no qual o endereço destino se diferencie do endereço do nó, fornece uma dimensão válida para rotear a mensagem. Para garantir a não ocorrência de *deadlock*, quando existem múltiplos *bits* (dimensões) diferindo, escolhe-se a dimensão em uma ordem pré-determinada da direita para esquerda. Assim, se, por exemplo, o CVP de identificação zero (0000) recebe uma mensagem cujo destino é o CVP três (0011), este a transmite pelo canal zero, que é a posição do *bit* mais a direita diferente nos endereços zero e três. Quando o CVP um (0001) recebe esta mensagem, este, então, a envia para o CVP três (0011) via o canal um seguindo o mesmo critério.

Capítulo V

Conclusão

Um componente crítico de um computador concorrente é a sua rede de comunicação. Para muitos algoritmos distribuídos, a comunicação constitui uma limitação maior que o próprio processamento, observando-se que quanto mais fina a granularidade de um programa concorrente e maior o diâmetro da rede, mais mensagens são enviadas e menos instruções são executadas em resposta a cada mensagem, sendo fundamental a redução da latência de comunicação.

O CVP foi projetado com o objetivo de tornar mais eficiente a comunicação em multicomputadores, executando tarefas de comunicação anteriormente realizadas pelo mesmo processador que executava a aplicação do usuário. Como já visto, dentre estas tarefas, destacam-se, além do roteamento de mensagens (algumas máquinas concorrentes do mercado já possuem dispositivos específicos para executar esta operação), instruções de propagação de mensagens como *broadcast* para nós vizinhos, *broadcast* por enchente, *convergecast* e *broadcast* em árvore geradora, que são consideradas blocos de construção de algoritmos distribuídos.

O CVP foi implementado na linguagem OCCAM em ambiente TDS (*Transputer Development System*). Sua utilização é bastante simples, como pode ser verificado no manual do Apêndice B. Basicamente, a aplicação (hospedeiro) envia instruções para o CVP e recebe resultados deste, que podem ser mensagens que atingiram seus destinos, ou informações de *status* relativas às instruções enviadas.

O sistema NCP-1, descrito no Apêndice C, fornece uma rede de *trans-*

puters que foi utilizada para testes do CVP, onde, para cada *transputer* usado foram alocados um processo "hospedeiro" e um processo CVP. Como os CVP's se comunicam através de mais de um caminho (conforme apresentado no Capítulo II) e como cada CVP foi carregado em um *transputer* diferente, que está conectado a cada outro *transputer* por apenas um *link* físico, se tornou necessária a implementação de um multiplexador e um demultiplexador para cada uma dessas conexões. O processo multiplexador recebe uma mensagem por vez de um dos caminhos e a envia pelo *link*. Esta mensagem é recebida pelo demultiplexador, no *transputer* vizinho, que a envia para o CVP associado ao nó pelo caminho correspondente. Incluindo os processos multiplexadores e demultiplexadores, o código do CVP I ocupa uma área de 31 *Kbytes* e do CVP II uma área de 22 *Kbytes*.

Da forma que o CVP está disponível, tem-se, em um *transputer*, um processo hospedeiro e um processo CVP disputando um único processador. Portanto, quando um destes processos estiver executando, o outro permanecerá esperando, o que é uma situação indesejável. Assim, é fundamental que existam dois processadores. Um, para executar as aplicações do usuário, e outro, que execute as operações do CVP. Estas operações poderiam ser implementadas em um *chip*, construído com base no projeto do Processador Virtual de Comunicação, apresentado nesta tese.

Referências Bibliográficas

[1] BARBOSA, V., C., *Strategies for the Prevention of Communication Deadlocks in Distributed Parallel Programs*, IEEE Trans. on Software Engineering, aceito para publicação.

[2] BARBOSA, V., C. and, FRANÇA, F.,*Specification of a Communication Virtual Processor for Parallel Processing Systems*, EUROMICRO, pp. 511-518, Zürich, Switzerland (1988).

[3] BURNS, A., "Programming in Occam 2", Addison-Wesley Plubishing Company (1988).

[4] BUZZARD, G., and MUDGETUDGET, T., *High Performance Hypercube Communications*, vol-1, pp.600-609 in Proc. of the third Conference on Hypercube Concurrent Processors and Applications, California, USA (1988).

[5] DALLY, W. J. *et al*, *Architecture of a Message-Driven Processor*, pp. 189-196 in Proc. of the 14th Annual Int. Symp. on Comp. Arch., Pittsburgh, PA, USA (1987).

[6] DALLY, W. J., and SEITZ, C. L., *Deadlock-Free Message Routing in Multiprocessor Interconnection Networks*, IEEE Trans. on Computers, vol. C-36, no.5, pp.547-553, May 1987.

[7] GAFNI, E., *Perspectives on Distributed Network Protocols: A Case for Building Blocks*, MILCOM'86, USA (1986).

[8] GRUNWALD, D. C., and, REED, D. A., *Networks for Parallel Processors : Measurements and Prognostications*, vol-1, pp.600-608 in Proc. of the third

Conference on Hypercube Concurrent Processors and Applications, California, USA (1988).

[9] GÜNTHER, K. D., *Prevention of Deadlocks in Packet-Switched Data Transport Systems*, IEEE Trans. on Communications, vol. C-29, no.4, pp.512-524, April 1981.

[10] INMOS Limited, "TDS - Transputer Development System", Prentice Hall (1988).

[11] NUGENTS, S. F., *The IPSC/2 Direct-Connect Communications Technology*, vol-1, pp.51-60 in Proc. of the third Conference on Hypercube Concurrent Processors and Applications, California, USA (1988).

[12] RAMACHANDRAN, U., SOLOMON, M., and VERNOM M., *Hardware Support for Interprocess Communication* pp. 178-188 in Proc. of the 14th Annual Int. Symp. on Comp. Arch., Pittsburgh, PA, USA (1987).

[13] REED, D. A., and FUJIMOTO, R. M., "Message-Based Parallel Processing", The MIT Press Series in Scientific Computation (1987).

[14] "Series 2010 General Description", AMETEK Computer Research Division (1988).

Apêndice A

OCCAM e TRANSPUTER

A linguagem de programação *Occam* permite que uma aplicação seja descrita como uma coleção de processos que operam concorrentemente e se comunicam através de canais de comunicação. Para sistemas, onde existe uma relação de um para um entre processos *Occam* e elementos processadores, e entre canais *Occam* e canais entre elementos processadores, *Occam* é considerada uma linguagem de montagem. A maior parte das linguagens concorrentes foi projetada para fornecer concorrência simulada, ou seja, estes projetos foram baseados na necessidade de se compartilhar um computador entre muitas tarefas independentes. Por outro lado, o projeto da linguagem *Occam* foi baseado na necessidade de se usar muitos computadores se comunicando para executar uma única tarefa.

A tecnologia *VLSI* permite que um grande número de dispositivos idênticos sejam fabricados a baixo custo. Por esta razão torna-se atraente a implementação de um programa *Occam* usando vários componentes idênticos, cada um programado com um processo apropriado. O *transputer* é tal componente e foi projetado junto com o *Occam* pela INMOS.[3] Este é usado como um bloco de construção nos sistemas de processamento concorrente, já que implementa os conceitos de *Occam* de concorrência e comunicação. Desta forma, quando a linguagem *Occam* é usada para programar uma rede de *transputers*, cada *transputer* executa o processo alocado a este e a comunicação entre processos *Occam* em diferentes *transputers* é implementada diretamente por canais de *transputer*. O *Occam* pode ser usado também para programar um *transputer* individual, que então compartilha

seu tempo entre os processos concorrentes e implementa a comunicação de canais movendo dados na memória.

A família de *transputers* da INMOS consiste tipicamente de *chips VLSI* que contêm processador, memória e canais para conexão de *transputers*. O primeiro membro da família, o IMST414, apresentado em setembro de 1985, possui um processador de 32 *bits*, 2 *Kbytes* de memória no próprio *chip*, uma interface de memória externa de 32 *bits* e quatro canais para conexão com outros *transputers*. O último modelo na família de *transputers* da INMOS, o IMST800, pôde aumentar o desempenho de tal sistema oferecendo grande melhoria ao desempenho de ponto flutuante. Sua arquitetura é similar ao do IMST414. Entretanto, além da memória, canais, CPU e interface de memória externa, existe uma unidade de ponto flutuante microprogramada que opera concorrentemente sob o controle da CPU.

O *transputer* possui instruções especiais e *hardware* para fornecer máximo desempenho e ótimas implementações do modelo *Occam* de concorrência e comunicação. Assim, este possui um escalonador microprogramado que permite que vários processos sejam executados, compartilhando o tempo do processador e removendo a necessidade de um *kernel* em *software*. A qualquer momento, um processo concorrente pode estar ativo ou inativo. Se um processo estiver ativo, ele pode estar executando ou pode estar na fila de espera para ser executado. Se um processo estiver inativo, este pode estar esperando pelo recebimento ou envio por um canal ou pode estar esperando por um tempo específico, não consumindo tempo de processamento. A comunicação no *transputer*, também atendendo ao modelo *Occam*, só ocorre quando tanto o processo de entrada como o de saída estão prontos. Um canal entre dois processos que executam em um mesmo *transputer* é implementado por uma única palavra de memória; um canal entre processos executando em diferentes *transputers* é implementado por *links* ponto-a-ponto. Para fornecer comunicação sincronizada, cada mensagem deve ser confirmada. Conseqüentemente, um *link* precisa de no mínimo uma linha em cada direção. As duas linhas do *link* podem ser usadas para fornecer dois canais *Occam*, um em cada direção, onde *bytes* de dados e confirmações são multiplexados em cada linha.

Nas seções seguintes são apresentados os aspectos mais relevantes da

linguagem *Occam* e do ambiente de desenvolvimento de sistemas TDS.

A.1 Programação em OCCAM

O modelo básico para um program *Occam* é uma rede de processos se comunicando através de canais.

Programas *Occam* são construídos a partir de processos primitivos [3], que são :

1- Atribuição

$v := e$

Assinala à expressão à esquerda do símbolo de atribuição (:=) o valor da expressão à direita do símbolo.

2- Comunicação

A comunicação entre processos em *Occam*, é feita através de canais de comunicação, os quais são de capacidade zero, unidirecionais e fornecem comunicação ponto-a-ponto entre dois processos concorrentes. O formato e tipo de comunicação em um canal é especificado por um protocolo de canal dado na declaração do canal.

Para executar a comunicação, existem os processos primitivos de entrada e saída :

$c\ !\ v$

Transmite a variável v para o canal c. A saída espera até que o valor tenha sido recebido por uma entrada correspondente.

$c\ ?\ v$

Recebe um valor do canal c e o assinala a variável v. A entrada espera até que um valor seja recebido.

A comunicação em um canal só pode ocorrer quando tanto o processo de entrada quanto o de saída estão prontos, ou seja, a comunicação é sincronizada.

3- SKIP e STOP

O processo primitivo SKIP é um processo cuja execução não provoca nenhum efeito e termina.

A execução do STOP não provoca nenhum efeito, mas nunca termina, impedindo que o processo que o executou prossiga.

Os processos primitivos são combinados para formar um processo maior, chamado construção, que começa com uma palavra chave *Occam* a qual indica como os processos componentes são combinados.

Estas construções são descritas a seguir.

Construção SEQ

Os processos devem ser executados um depois do outro. Esta construção termina quando seu último processo termina. A seguir a execução sequencial dos processos P_1 e P_2 :

SEQ
  P_1
  P_2

Construção PAR

Os processos devem ser executados em paralelo, terminando apenas quando todos os processos componentes terminarem. No exemplo a seguir P_2 e P_3 executam em paralelo.

SEQ
  P_1
  PAR
    P_2
    P_3
  P_4

Construção WHILE

Os processos são executados repetidamente enquanto uma determinada condição for verdadeira. No exemplo a seguir, P_1 e P_2 são executados até que v_1 seja menor que v_2.

WHILE $v_1 \geq v_2$
  SEQ
    P_1
    P_2

Construção IF

IF pode receber qualquer número de processos, cada um deles precedidos por um teste. Somente um dos processos será realmente executado e este será o primeiro (na ordem em que foram escritos) cujo teste for verdadeiro. No exemplo a seguir $cond_1$,..., $cond_n$ são expressões lógicas e a expressão TRUE faz com que o processo associado P_0 seja executado, caso nenhuma das expressões lógicas anteriores seja verdadeira.

IF
  $cond_1$
    P_1
  ...
  $cond_n$

P_n

TRUE

P_0

Construção CASE

CASE pode receber qualquer número de processos, cada um com uma lista de uma ou mais expressões colocada antes deste. Somente um dos processos será executado, e este será o primeiro (novamente, na ordem que eles foram escritos) com uma expressão que tem o mesmo valor de uma variável utilizada para seleção. No exemplo a seguir, caso V não seja igual a $v1$,...,vn, P_0 será executado.

CASE V

$v1$

P_1

...

vn

P_n

ELSE

P_0

Construção ALT

Como foi visto, IF permite que escolhas sejam feitas em consonância com os valores de expressões condicionais e CASE de acordo com o valor de uma variável. Com ALT pode-se fazer escolhas consoantes os estados de canais. ALT tem como cada alternativa um processo de entrada seguido por um processo a ser executado. ALT assiste a todos os processos de entrada e executa o processo associado com a primeira entrada a se tornar disponível, onde, somente um processo será executado.

ALT

$c1?$ $v1$

P_1

...

$cn?\ vn$

P_n

A.2 Transputer Development System (TDS)

O TDS é um sistema de desenvolvimento integrado que pode ser usado para desenvolver programas *Occam* para uma rede de *transputers*. Este consiste de um *plug* na placa do *transputer* para um IBM-PC e um *software* de desenvolvimento que executa na placa do *transputer*. Esta combinação fornece um ambiente no qual programas podem ser desenvolvidos, compilados e executados. Programas podem também ser desenvolvidos e compilados no TDS para serem executados em uma rede de *transputers*, sendo o código carregado na rede a partir do TDS.

A principal interface com o sistema é um editor; logo que o sistema é inicializado, o usuário é colocado em um ambiente de edição, e assim, a edição do programa, compilação e execução podem ser executadas dentro deste próprio ambiente. A interface com o editor é baseada no conceito de "*folding*". As operações de *folding* permitem que seja dada ao texto uma estrutura hierárquica que reflete a estrutura do programa sendo desenvolvido. Desta forma, através deste editor de *fold* é possível agrupar blocos de linhas.

Para compilar um programa *Occam*, duas condições devem ser atendidas. A primeira, é a gravação do *fold* que contém o programa fonte em um arquivo separado, criando-se um "*filed fold*". A segunda, é a criação de um *fold* de compilação que contém o *fold* com o programa, que é utilizado, então, pelo compilador. O tipo de *fold* de compilação indica que tipo de unidade de compilação este contém. Assim, por exemplo, o tipo SC ("*separate compilation unit*"), normalmente, não possui um programa completo, apenas alguns procedimentos e funções e usualmente está dentro de uma outra unidade de compilação; o tipo PROGRAM contém informação de configuração que possibilita a carga do sistema desenvolvido em uma rede de *trans-*

puters; e o tipo EXE é um processo *Occam*, a ser executado no próprio ambiente TDS, que pode acessar canais que se comunicam com a tela e o teclado.

Para configurar uma rede de *transputers* é necessário colocar os procedimentos a serem carregados nos *transputers* em SC's. Estas SC's devem, então, ser agrupadas em um *filed fold*, ao qual é atribuído o tipo PROGRAM. Neste *fold* PROGRAM são colocados os comandos que descrevem as interconexões entre os processadores e que chamam os procedimentos a serem executados em cada processador [10].

A.3 O Sistema NCP-1

O sistema NCP-1 é uma máquina de memória distribuída interligada segundo um cubo binário de dimensão quatro. O sistema como um todo é constituído de dois componentes. O primeiro é um hospedeiro, atualmente um IBM-PC ou AT, que oferece acesso a disco, fita ou terminal gráfico e atua como um dispositivo de entrada e saída do cubo. O segundo é o NCP-1, um cubo (atualmente de dimensaão três) que possui nós idênticos de processamento conectados ponto-a-ponto através de *links* bidirecionais de comunicação. Cada um destes nós é constituído por dois módulos, sendo o primeiro o módulo de processamento e comunicação (já disponível) e o segundo o módulo vetorial.

O módulo de processamento e comunicação, possui um microprocessador transputer T-800 operando a 20MHz. O T-800 tem registradores de 32 *bits*, quatro *links* bi-direcionais de 20 Mbits/sec, memória interna RAM estática de 4 *Kbytes* e uma unidade de ponto flutuante. Como o T-800 é um processador de 32 *bits*, as suas linhas de endereços e dados são multiplexadas possibilitando um espaço total de endereçamento de memória de 4 *Gbytes*. Além do T-800, no módulo, existem 64 *Kbytes* de memória EPROM e 4 *Mbytes* de memória RAM dinâmica. Em EPROM ficam residentes os dados de configuração do *transputer* e as rotinas de diagnóstico, de tratamento de interrupção e de tratamento de erros internos. Como o *transputer* tem quatro *links* de comunicação serial, formados por um canal de entrada e outro de saída, pode-se obter no máximo configurações de dimensões iguais

a quatro. Portanto, para permitir configurações com dimensões maiores, o módulo de processamento e comunicação é dotado de um "*crossbar-switch*". Este dispositivo, além de oferecer uma interligação com mais vértices no cubo, possibilita a reconfiguração da rede, permitindo a utilização de diferentes topologias.

O módulo vetorial possui um microprocessador Intel i860 e ainda não está diponível no NCP-1.

Documentações mais detalhadas deste sistema estão sendo desenvolvidas pelos pesquisadores da COPPE/UFRJ envolvidos com o projeto desta máquina.

Apêndice B

Manual de Utilização do CVP

O CVP foi implementado como um processo que recebe e envia mensagens por canais que o conectam a outros CVP's e ao hospedeiro.

Para cada processo hospedeiro existe um processo CVP correspondente executando em paralelo. Da forma implementada, um processo hospedeiro (implementado em *Occam*) e um processo CVP executam em um mesmo *transputer*, e o CVP pode se comunicar com outros CVP's alocados em *transputers* diferentes. Para executar o CVP é necessário chamar o procedimento "com.proc" que necessita dos seguintes parâmetros : *id* (identificação do nó) e os canais que são usados para conectar o CVP aos seus vizinhos. Estes canais utilizam o protocolo "G.PROTOC" definido nesta implementação. Associado ao CVP existe um *fold*, chamado *constants*, com os parâmetros necessários para sua execução (descritos nas Seções II.2.4 e II.3.4) que devem ser atualizados pelo usuário. O processo hospedeiro envia instruções para o CVP pelo canal "host.ud" e recebe resultados deste pelos canais "host.dis.msg", "host.dis.st" e "host.dis.stcir" (este último no caso do CVP II apenas). Os protocolos usados por estes canais são : "INST", "P.MSG", "STAT" e "STAT.CIR ", respectivamente, e estão definidos no *fold channels*, juntamente com as declarações destes canais. Portanto, para utilizar o CVP é necessário que o processo, que executa em um *transputer*, declare os canais que o conectam a outros *transputers* com o protocolo "G.PROTOC" (que está no *fold Protocols*); defina os os parâmetros do CVP (disponíveis no *fold constants*); defina os canais utilizados na comunicação entre o CVP e o hospedeiro (disponíveis no *fold chan-*

nels); defina o próprio CVP que está no procedimento com.proc (disponível no *fold CVP*); e, finalmente, execute em paralelo o processo hospedeiro e o procedimento com.proc, passando os parâmetros adequadamente. Esta estrutura está disponível, no *file* PC.TOP, bastando ao usuário fazer a substituição do *file host*, relativo ao processo hospedeiro, pelo seu próprio *file* e as atualizações dos parâmetros no *file constants*, se necessário. Exemplos de utilizações do CVP são colocados na Seção B.1.

Após o envio de cada instrução no modo *store-and-forward*, o processo hospedeiro deve esperar um sinal do CVP, que pode ser um *ack*, caso a instrução tenha sido aceita, ou um *nack*, caso a instrução tenha sido recusada por falta de espaço em *buffer*. Este sinal é recebido pelo canal host.dis.st. O hospedeiro também recebe mensagens (transmitidas no sistema através de uma das instruções do CVP) pelo canal host.dis.msg e, no caso do CVP I, confirmações de circuitos estabelecidos pelo canal host.dis.stcir.

Nas próximas seções, são especificados os procedimentos necessários para utilização do CVP I e do CVP II.

B.1 Utilização do CVP I

Se o processo hospedeiro estiver conectado ao CVP I, este pode enviar as seguintes instruções : SEND, BROADF, BROADFF, BROADT, CONVT, BROADN, BROADNF, ASK.CIRCUIT, ASK.VIRTUL.TREE, SEND.CIRCUIT, SEND.VIRTUAL.TREE, FREE.CIRCUIT e FREE.VIRTUAL.TREE.

A seguir, são colocados alguns exemplos que mostram como o CVP I deve ser utilizado para execuções destas instruções.

EX1: Envio da mensagem "teste" do nó zero para o nó três.

O processo hospedeiro do nó zero deve executar primeiro:

host.ud ! send;3;5::"teste"

Desta forma, o nó zero "pede" para o CVP I executar a instrução SEND cujos operandos são o destino, que é o nó três, e a própria mensagem "teste", que possui comprimento de cinco *bytes*.

Depois de enviar a instrução para o CVP I, o processo hospedeiro recebe na variável *ans* um *ack* ou um *nack* (como mostrado a seguir), o que lhe permitirá saber se a instrução foi aceita ou recusada.

host.dis.st ? ans

O processo hospedeiro do nó três deve executar :

host.dis.msg? len::msg

Desta forma, este recebe na variável *msg* a mensagem enviada pelo nó zero.

EX2: *Broadcsat* por enchente da mensagem "teste" a partir do nó zero.

O hospedeiro do nó zero deve executar:

host.ud ! broadf; 0;5::"teste"
host.dis.st ? ans

Todos os outros processos hospedeiros do sistema devem executar a instrução a seguir para receber a mensagem "teste":

host.dis.msg ? len::msg

Se fosse executado o *broadcast* por enchente com realimentação (host.ud! broadff; 0;5::"teste"), o hospedeiro do nó zero deveria receber ainda, caso a instrução tivesse sido aceita, a própria mensagem, como uma indicação de que o *broadcast* terminou :

host.dis.msg ? len::msg

EX3 : *Broadcast* da mensagem "teste"em uma árvore geradora, identificada pelo valor um, cuja raiz é o nó zero.

O hospedeiro do nó zero deve executar :

host.ud ! broadt; 1;5::"teste"
host.dis.st ? ans

Todos os outros nós do sistema devem esperar o recebimento da mensagem :

host.dis.msg? len::msg

EX4 : *Convergecast* em uma árvore geradora, identificada pelo valor um, da mensagem "teste".

Os nós que são folhas da árvore devem executar :

host.ud ! convt; 1;5::"teste"
host.dis.st ? ans

Todos os outros nós do sistema devem esperar o recebimento da mensagem, da mesma forma que a apresentada anteriormente.

EX5 : *Broadcast* para os nós vizinhos da mensagem "teste" a partir do nó zero.

O hospedeiro do nó zero deve executar :

host.ud ! broadn;5::"teste"
host.dis.st ? ans

Todos os nós vizinhos do nó zero devem esperar o recebimento da mensagem.

Se fosse executado o *broadcast* para vizinhos com realimentação (host.ud! broadnf;0;5::"teste"), o hospedeiro do nó zero deveria receber ainda, caso a instrução tivesse sido aceita, a mensagem, como uma indicação de que o *broadcast* terminou .

EX6: Envio da mensagem "teste" do nó zero para o nó três, através de um circuito estabelecido entre estes nós.

O processo hospedeiro do nó zero deve primeiro estabelecer o circuito do nó zero ao nó três, Da mesma forma que nas instruções anteriores, o hospedeiro pode receber um *ack* ou um *nack* caso a instrução tenha sido aceita, ou recusada, como apresentado a seguir:

host.ud ! ask.circuit;0;3
host.dis.st ? ans

Caso a instrução tenha sido aceita pelo CVP I, o processo hospedeiro do nó zero deve receber uma confirmação de circuito estabelecido :

host.dis.stcir ? ans

Tendo recebido esta confirmação, o processo hospedeiro do nó zero pode enviar mensagens pelo circuito, da seguinte forma:

host.ud ! send.circuit;0;3;5::"teste"

E o nó três deve esperar o recebimento da mensagem :

host.dis.msg ? len::msg

Para liberar o circuito o processo hospedeiro do nó zero deve executar:

host.ud ! free.circuit;3

EX7: Envio da mensagem "teste" por um circuito em forma de árvore, identificada pelo valor um, formado a partir do nó zero.

O processo hospedeiro do nó zero deve primeiro estabelecer o circuito. Da mesma forma que nas instruções anteriores, o hospedeiro pode receber um *ack* ou um *nack*:

host.ud ! ask.virtual.tree;1
host.dis.st ? ans

Caso a instrução tenha sido aceita pelo CVP I, o processo hospedeiro deve receber uma confirmação de circuito estabelecido :

host.dis.stcir ? ans

Tendo recebido esta confirmação, o processo hospedeiro do nó zero pode enviar mensagens pelo circuito, da seguinte forma:

host.ud ! send.virtual.tree;1;5::"teste"

E todos os outros nós do sistema devem esperar o recebimento da mensagem :

host.dis.msg ? len::msg

Para liberar o circuito o processo hospedeiro do nó zero deve executar:

host.ud ! free.virtual.tree;1

B.2 Utilização do CVP II

Se o processo hospedeiro estiver conectado ao CVP II, este pode enviar as seguintes instruções : SEND, BROADF, BROADFF, BROADT, CONVT, BROADN, BROADNF. Ou seja, todas as instruções do CVP I que atendem ao modo *store-and-forward.* A única diferença em relação ao CVP I é que cada uma destas instruções deve ter um parâmetro adicional que permite a escolha de uma classe de *buffer* no CVP II para armazená-la (como explicado no Capítulo IV).

Desta forma, o processo hospedeiro , ao utilizar o CVP II, deve acrescentar a cada uma das instruções uma indicação da classe onde esta deve ser mantida. Conforme o algoritmo apresentado no Capítulo IV, a classe que a mensagem deve ocupar no nó origem é a zero. Assim, no primeiro exemplo da seção anterior, o hospedeiro deveria executar : host.ud ! send;0;3;5::"teste", onde zero indica a classe de *buffer* a ser ocupada pela instrução.

Em todas as outras instruções, também acrescenta-se um segundo parâmetro com o valor zero.