UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Multicomputador Nó//: Implementação de Primitivas Básicas de Comunicação e Avaliação de Desempenho

por

Valeria Alves da Silva

Dissertação submetida à Universidade Federal de Santa Catarina para obtenção do grau de Mestre em Ciência da Computação

Prof. Altamiro Amadeu Suzim, Dr.
Orientador

Florianópolis, maio de 1996

MULTICOMPUTADOR NÓ//: IMPLEMENTAÇÃO DE PRIMITIVAS BÁSICAS DE COMUNICAÇÃO E AVALIAÇÃO DE DESEMPENHO

VALERIA ALVES DA SILVA

ESTA DISSERTAÇÃO FOI JULGADA ADEQUADA PARA OBTENÇÃO DO TÍTULO DE

MESTRE EM CIÊNCIA DA COMPUTAÇÃO

ESPECIALIDADE SISTEMAS DE COMPUTAÇÃO E APROVADA EM SUA FORMA FINAL PELO PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO.

Prof. Murilo Silva de Camargo, Dr.
Coordenador do curso

BANCA EXAMINADORA:

Prof. Altamiro Amadeu Suzim, Dr.
Orientador

Prof. Paulo José de Freitas Filho, Dr.

Prof. Luis Fernando Friedrich, Dr.

Prof. Thadeu Botteri Corso, M.Sc.

A minha mãe por todo o seu amor.

RESUMO

Este trabalho está inserido no contexto do projeto do multicomputador Nó // e objetiva a concepção de um ambiente para programação paralela. Este projeto está fundamentado em um multicomputador com uma rede de interconexão dinâmica e um sistema de interrupção, através dos quais os nós de trabalho e o nó de controle se comunicam. Cada um dos nós que compõem o multicomputador possui um processador i486, memória RAM de 4 Mbytes e uma placa de comunicação. Esta, por sua vez, é responsável pela interface dos nós com a rede de interconexão e o sistema de interrupção. A rede de interconexão é composta de um comutador de conexões do tipo crossbar capaz de conectar dezesseis pares de nós simultâneamente. O nó de controle é responsável por gerenciar a configuração do crossbar, conforme as solicitações feitas pelos nós de trabalho. O multicomputador Nó // possui um sistema operacional que provê serviços para comunicação entre processos permitindo a exploração do paralelismo a nível de sistema e de aplicações. Este trabalho implementa as primitivas básicas de comunicação para a interface do hardware do multicomputador com as camadas do sistema operacional e avalia o desempenho da máquina através de um modelo de simulação. Este modelo simula o funcionamento da máquina, tornando possível aplicar cargas de trabalho semelhantes às utilizadas na camada de aplicação . A avaliação de desempenho apresenta a temporização do protocolo de comunicação, nas várias configurações - número de nós na máquina - e o tempo gasto na transferência de bytes.

ABSTRACT

This research is a part of the project Nó // (Paralell Node). The Nó // is defined as a mmulticomputer which uses a dynamic interconexion network with interruption lines based communication scheme. Every node of the multicomputer has the following configuration: i486 procesor, 4 Mbytes of RAM, and a communication interface that provides the interface between the interruption lines and the interconexion network. The interconexion network uses a crossbar, as interconexion switch, which may establish 16 point to point conections simultaneously. There is a control node which is responsable for managing.

The crossbar configuration following the request. At this time, there is on operating system kernel that provides interprocess communication (EPC) services in order to implement parallel aplications (system and user level).

This work presents: - The basics communication routines which will be used by the kernel in order to provide high level communications services. - A performance evaluation of the Nó // communication system.

SUMÁRIO

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

Estas primitivas utilizam as características de hardware da placa de comunicação para formecer estes serviços para a camada das chamadas do núcleo do sistema operacional CRUX.

A avaliação de desempenho do Nó // tem como objetivo obter algumas medidas de desempenho do Nó //, a partir da simulação dos tempos gastos na execução de um algoritmo paralelo. Estas medidas são: o tempo gasto com o protocolo de comunicação, o tempo gasto na transmissão de bytes na rede de comunicação, o tempo de espera por uma conexão com o incremento do número de nós de trabalho no Nó // e o ganho de performance obtido.

O trabalho está dividido em sete capítulos, apresentando os seguintes tópicos: as máquinas paralelas, os sistemas operacionais distribuídos, os algoritmos paralelos, o multicomputador Nó //, a implementação da primitivas básicas e a avaliação de desempenho.

O capítulo 2 aborda as máquinas paralelas. Define conceitos, apresenta classificações e explora os tipos principais de máquinas paralelas: os multiprocessadores e multicomputadores

O capítulo 3 apresenta as caraterísticas dos sistemas operacionais distribuídos como, por exemplo, o paradigma de comunicação entre processos e um núcleo em cada processador.

No capítulo 4 são mostrados alguns aspectos do processamento de algoritmos paralelos. Apresentando os paradigmas da programação paralelas e os conceitos fundamentais em processamento paralelo.

O capítulo 5 apresenta o multicomputador Nó //. Primeiramente é descrita a concepção do hardware e os componentes da máquina e, posteriormente, é descrito o sistema operacional CRUX, com suas características e primitivas.

No capítulo 6 é apresentada uma implementação da camada de comunicações de baixo nível para o Nó //, descrevendo todas as primitivas criadas.

No capítulo 7 descreve o modelo de simulação e os resultados obtidos na avaliação de desempenho do Nó //. A avaliação é conduzida a partir da simulação dos custos das comunicações no multicomputador Nó //.

CAPÍTULO 2

As Arquiteturas Paralelas

Devido ao crescente aumento da complexidade das aplicações computacionais estão surgindo novos modelos de arquiteturas, com o objetivo de aumentar o poder dos sistemas de computação. Nas tecnologias utilizada até hoje, baseadas no modelo de Von Neumann, soluções como aumento da velocidade, *pipeline* e acesso eficiente à memória e periféricos estão chegando ao seu limite. Devido a esse fato há uma tendência de emprego de novas arquiteturas. As arquiteturas baseadas em múltiplos processadores têm se mostrado uma opção para o aumento do poder computacional dos sistemas de computação. As máquinas paralelas podem ser classificadas pela interconexão de seus componentes, pelo tipo de controle dos processadores e pelo mecanismo de acesso a memória.

2.1 Classificação

Vários esquemas de classificação para sistemas de computação foram propostos. O esquema de classificação de Michael Flynn [FLY72] é o mais usado. Esta classificação é baseada no número de fluxos de instruções e de dados tratados simultaneamente pela máquina durante a execução do programa.

Denotando-se **ni** o número de fluxos de instruções e **nd** o número de fluxos de dados que podem ser processados ao mesmo tempo no computador. De acordo com Flynn, os sistemas de computação podem ser classificados em quatro classes distintas: SISD, SIMD, MISD, MIMD.

$\Rightarrow$ As máquinas SISD (*Single Instruction Single Data* $\Rightarrow$ **ni** = **nd** = **1**) incluem os computadores convencionais baseados no modelo de Von Neumann, compostos de uma unidade de processamento e uma unidade para decodificação e comando. As instruções e os dados são obtidos na memória através de um barramento, e executados seqüencialmente pela unidade de processamento (Figura 2.1).

CAPÍTULO 1

Introdução

O presente trabalho faz parte do projeto Nó // (lê-se nó paralelo), em desenvolvimento no Curso de Pós Graduação em Ciência da Computação da Universidade Federal de Santa Catarina, em cooperação com a Universidade Federal do Rio Grande do Sul e a Universidade Federal de Santa Maria. O projeto objetiva a concepção de um ambiente completo para programação paralela, construindo um multicomputador com rede de interconexão dinâmica, um sistema operacional e uma linguagem de programação paralela.

- O Nó //, é um multicomputador com rede de interconexão dinâmica que oferece grande flexibilidade para comunicação. Possui um mecanismo de comunicação extremamente simples que estabelece as conexões físicas, por demanda, através de canais diretos entre o nós processadores e forma um modelo específico de máquina paralela com memória distribuída. Foram propostos dois modelos de arquitetura para o multicomputador : Um modelo baseado em um barramento comum e um modelo baseado em um sistema de interrupção.

- O sistema operacional construído para o multicomputador Nó //, denominado CRUX ([MON95] e [ROD95]), tem como base o modelo de processos que se comunicam através da troca de mensagens e apresenta compatibilidade com sistema UNIX, a nível de interface de programação. O sistema foi implementado sobre um simulador do multicomputador com arquitetura baseada em barramento.

- A linguagem de programação implementada foi o SuperPascal [MER96]. Esta linguagem possui características para criação e a comunicação de processos e se adapta perfeitamente a topologia dinâmica do multicomputador Nó //.

Este trabalho tem como objetivos: implementar as primitivas básicas de comunicação e avaliar o desempenho do Nó //.

A implementação das primitivas básicas de comunicação tem como objetivo fornecer serviços para a troca de mensagens de pedidos entre os nós de trabalho e o nó de controle.

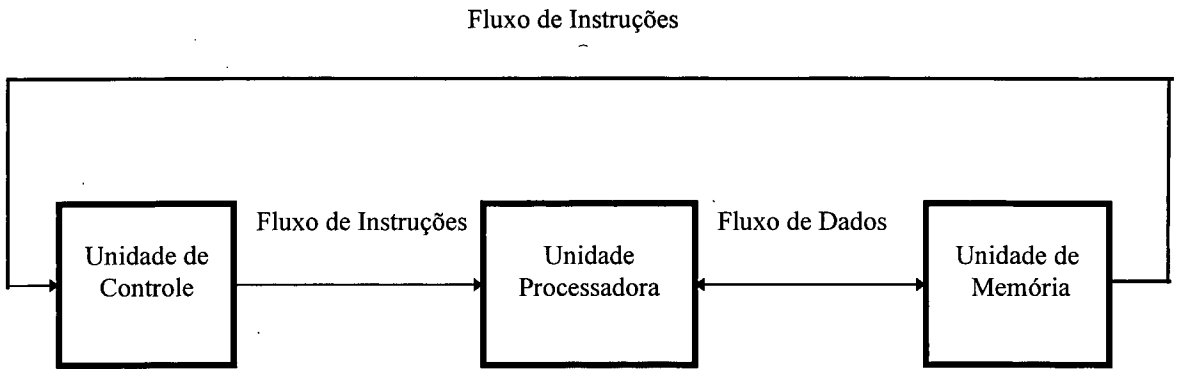


Figura 2.1 : Modelo de Máquinas SISD

⇒ As máquinas SIMD (*Single Instruction Multiple Data* ⇒ **ni** = 1, **nd** >1) possuem uma unidade para decodificação e comando de múltiplas unidades de processamento. Em tais computadores muitos dados podem ser acessados de uma memória privativa e processados simultaneamente por uma única instrução (Figura 2.2). O modelo SIMD pode variar em dois aspectos. Primeiro, em termos do número de processadores, que pode ser fixo ou não podendo ser inseridos mais processadores; segundo, em termos do mecanismo de roteamento entre os processadores, os quais podem se comunicar com cada outro por um barramento comum, onde todos os componentes da máquina estão ligados ou memória compartilhada ou por uma rede de interconexão.

⇒ Nas máquinas MISD (*Multiple Instruction Single Data* ⇒ **ni** > 1, **nd** = 1) cada unidade de processamento possui uma unidade para decodificação e comando. Desta forma, os processadores executam instruções diferentes sobre um mesmo conjunto de dados (Figura 2.3).

⇒ As máquinas MIMD (*Multiple Instruction Multiple Data* ⇒ **ni** > 1, **nd** > 1) são capazes de executar vários programas independentes simultaneamente. Elas possuem múltiplas unidades para decodificação e comando e múltiplas unidades de processamento que executam simultaneamente instruções diferentes sobre diferentes conjuntos de dados (Figura 2.4).

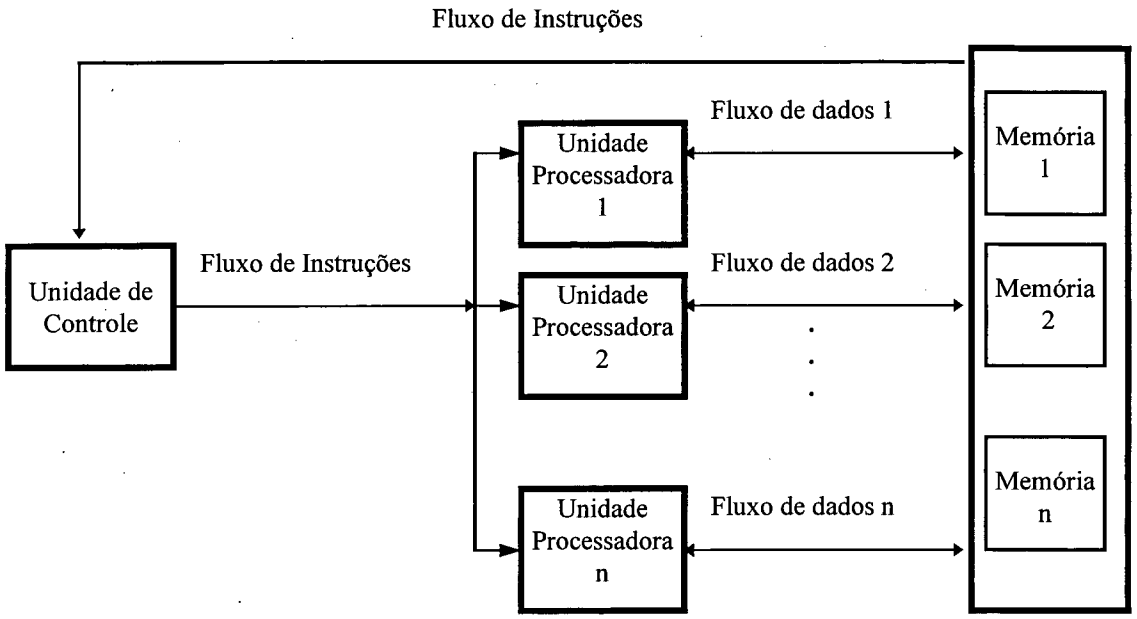


Figura 2.2 : Modelo de Máquinas SIMD

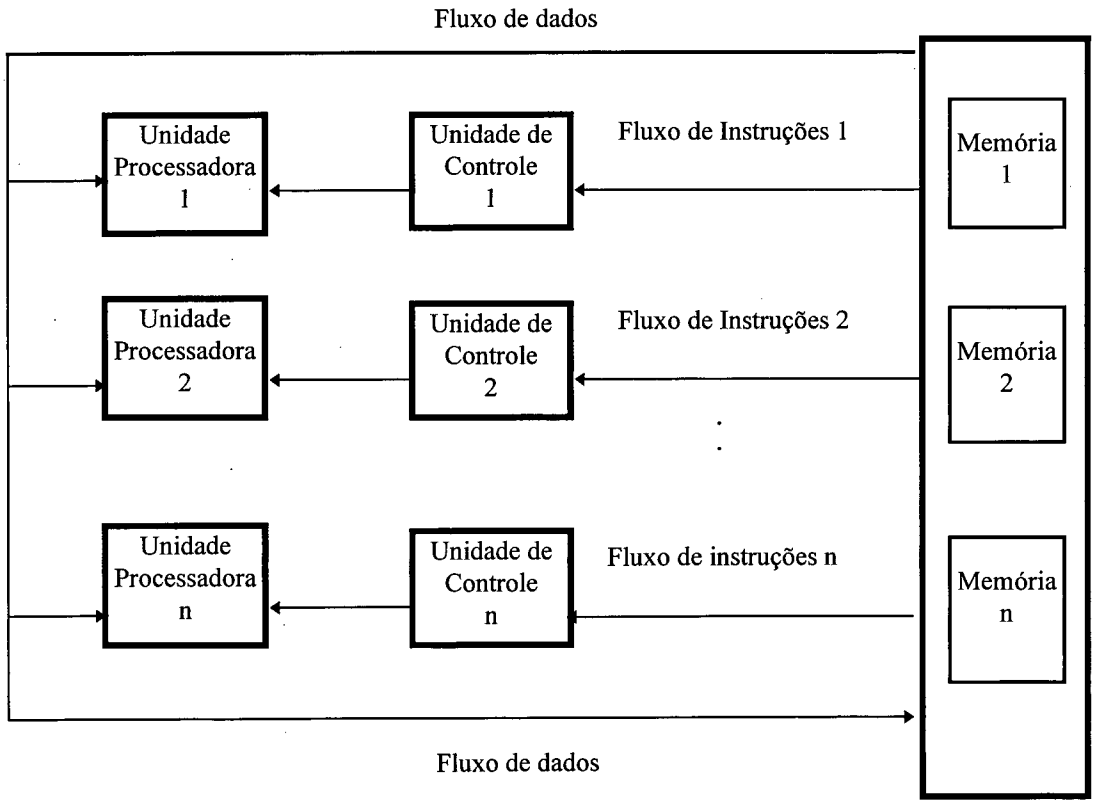


Figura 2.3 : Modelo de Máquinas MISD.

Com respeito à organização da memória, as máquinas MIMD se subdividem em multiprocessadores (memória compartilhada) e multicomputadores (memória distribuída).

O desempenho destes sistemas é fortemente afetado pelo volume, freqüência e velocidade de comunicação de dados entre os processadores.

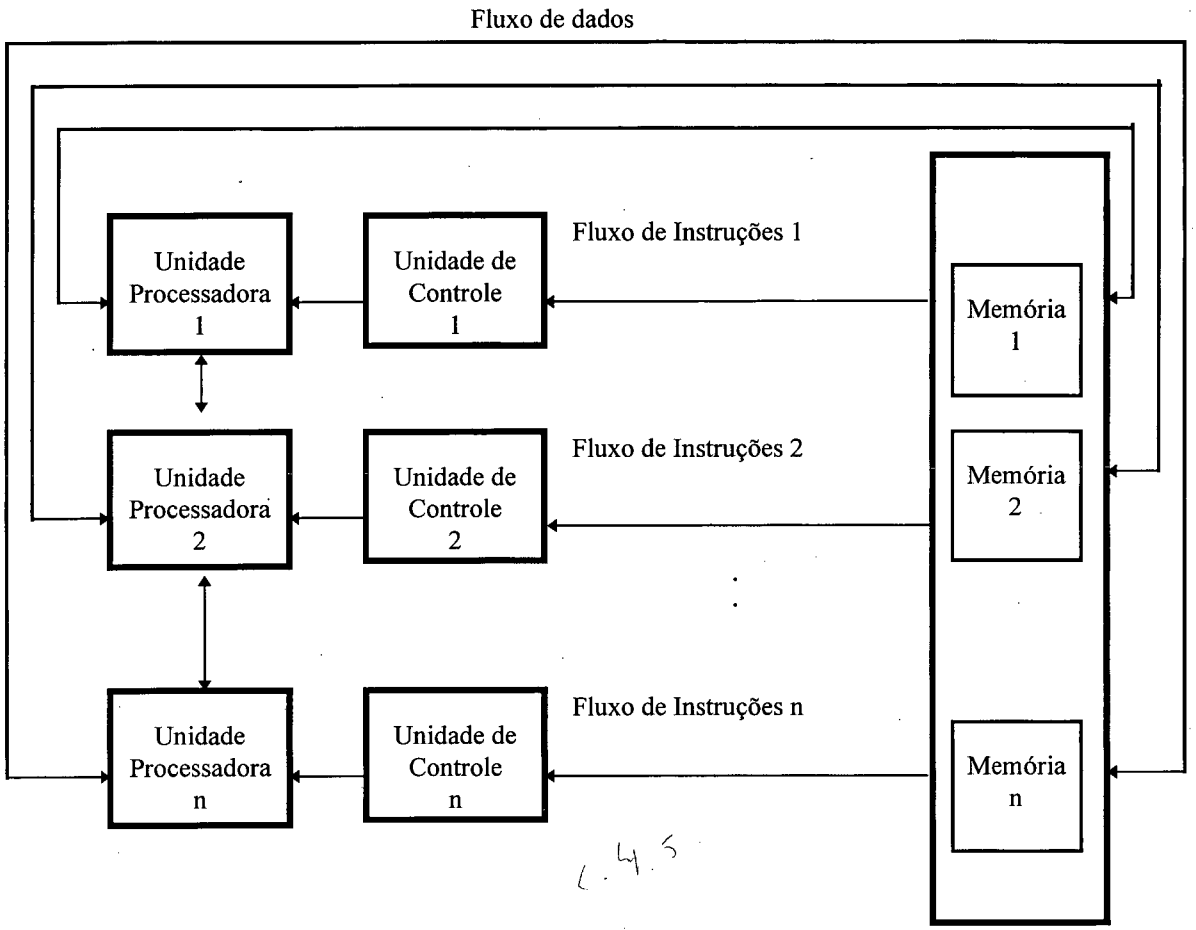


Figura 2.4 : Modelo de Máquinas MIMD

2.2 Multiprocessadores

Nos multiprocessadores todos os processadores utilizam uma memória comum como meio de comunicação entre os processadores. O custo envolvido no processo de chaveamento neste tipo de máquina é o fator dominante no seu desempenho. Estes sistemas são denominados fortemente acoplados

Os multiprocessadores são máquinas compostas de múltiplas unidades de processamento e uma memória compartilhada [AUS91]. Os Multiprocessadores são chamados de sistemas fortemente acoplados, pelo seu alto grau de compartilhamento. A seguir serão descritos três modelos de arquitetura de compartilhamento de memória. O modelo UMA (*uniform memory access*), o modelo NUMA (*nonuniform memory access*) e o modelo COMA (*cache-only memory architecture*).

2.2.1 O Modelo UMA

Em um multiprocessador UMA, a memória física é uniformemente compartilhada por todos os processadores, e dispositivos periféricos (Figura 2.5). Todos os processadores tem o mesmo tempo de acesso a memória de trabalho. O modelo UMA é conveniente para uso de propósitos gerais e aplicações *time-sharing* para múltiplos usuários.

2.2.2 O Modelo NUMA

Em um multiprocessador NUMA, o compartilhamento de memória tem os tempos de acesso que variam em função da localização da memória. A memória compartilhada é distribuída fisicamente por todos os processadores, e é chamada de memória local. O conjunto de memórias locais forma o espaço de endereçamento global acessível a todos os processadores. O tempo que um processador gasta para acessar um endereço em sua memória local é pequeno. Entretanto, o tempo que um processador gasta para acessar um endereço de memória de um processador remoto é acrescido de uma espera para o acesso à rede de interconexão (Figura 2.6).

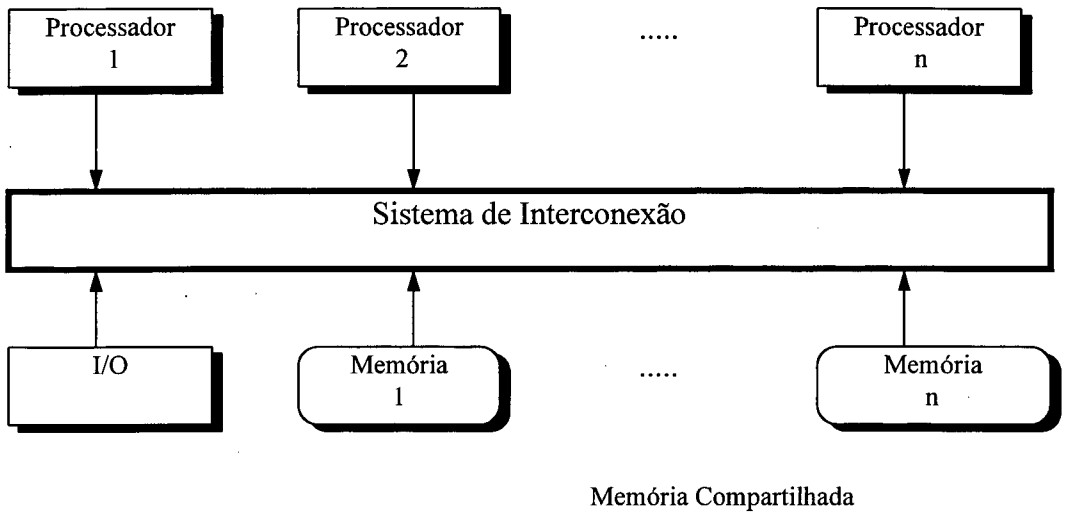


Figura 2.5 : Modelo **UMA** de Multiprocessador

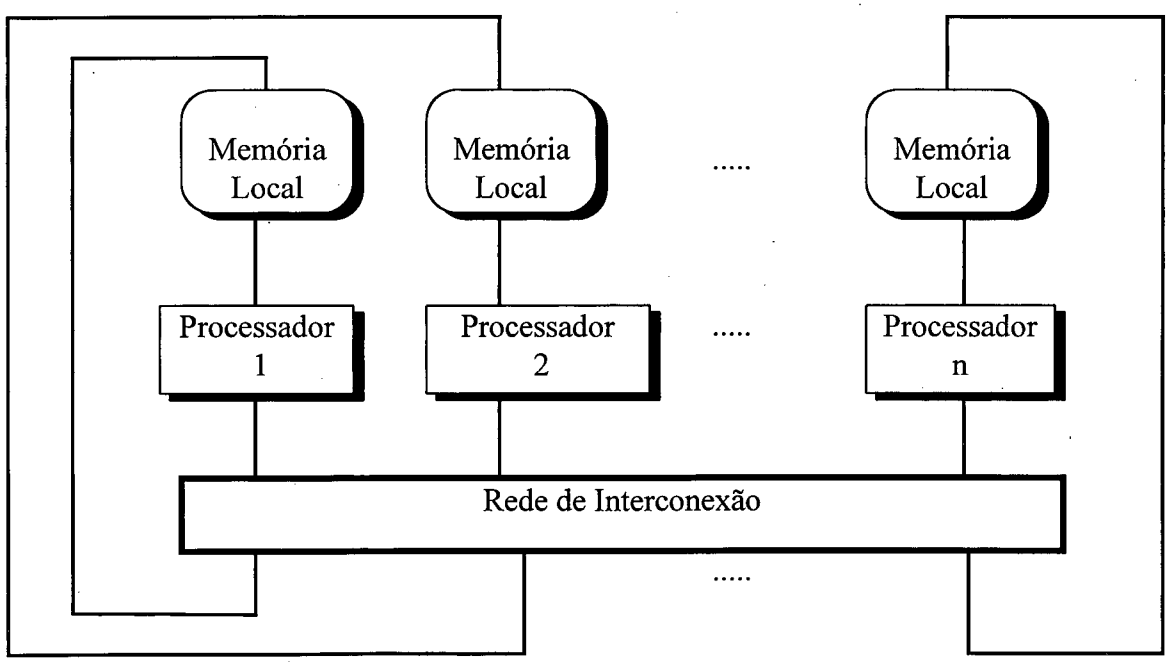


Figura 2.6 : Modelo **NUMA** de Multiprocessador

2.2.3 O Modelo COMA

O modelo COMA é um caso especial do modelo NUMA, no qual a memória principal distribuída é convertida em *caches*(depósito) de memória. Todos os depósitos formam um espaço de endereçamento global. O acesso aos depósitos remotos são assistidos por diretórios de depósito (Figura 2.7).

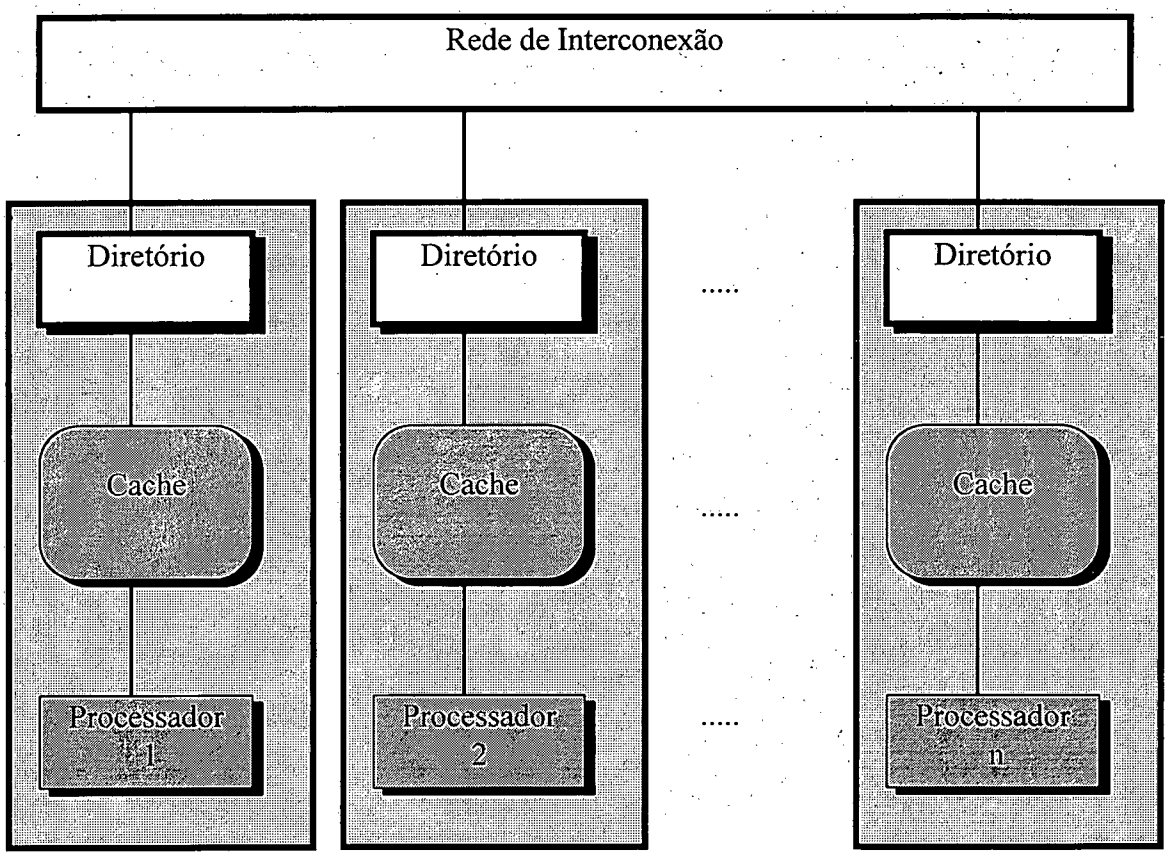


Figura 2.7 : Modelo **COMA** de Multiprocessador

2.3 - Multicomputadores

Nos multicomputadores cada processador possui sua própria memória e canais de comunicação. A comunicação entre os processadores é feita por meio de um sistema de troca de mensagens através de uma rede de interconexão. Estes sistemas são denominados

fracamente acoplados, e são sistemas eficientes quando a quantidade de interações entre os vários processos executados nos processadores é pequena.

Os Multicomputadores são máquinas compostas por várias unidades de controle e processamento (CPU), interligadas através de uma rede de interconexão. São considerados sistemas fracamente acoplados, pois, cada processador possui uma memória privativa, donde são obtidas as instruções e os dados para o processamento.

Os multicomputadores diferem das redes de computadores por terem canais de comunicação exclusivos para comunicações bipontuais. As redes de interconexão são o elemento principal para o desempenho de um multicomputador, podendo determinar o projeto do sistema operacional e aplicações para a máquina paralela.

2.3.1 - Tipos de Redes de Interconexão

As redes de interconexão podem ser divididas em dois tipos: as *redes de barramento*, onde existe um barramento comum a todos os nós processadores e as *redes bipontuais*, que possuem canais que fazem a conexão de pares de nós processadores. As redes bipontuais podem ser basicamente divididas em estáticas e dinâmicas [SIL94].

- **Redes Estáticas**

As redes estáticas se caracterizam por não terem capacidade de mudar sua configuração. A topologia ideal para uma rede estática seria *completamente interconectada* (Figura 2.5). Entretanto, esta topologia é inviável para um grande número de nós, pois cada nó deve ter um canal de comunicação com cada um dos outros nós da rede. A rede tipo *grelha* (Figura 2.9) é classificada como uma rede estática com topologia bidimensional. Neste caso, cada nó tem um canal de comunicação com o seu vizinho.

O principal problema nesta topologia é a comunicação entre nós não vizinhos e distantes. A mensagem terá que percorrer vários canais até alcançar seu destino. Sendo assim, esta topologia não é recomendada para grandes redes [TAN92].

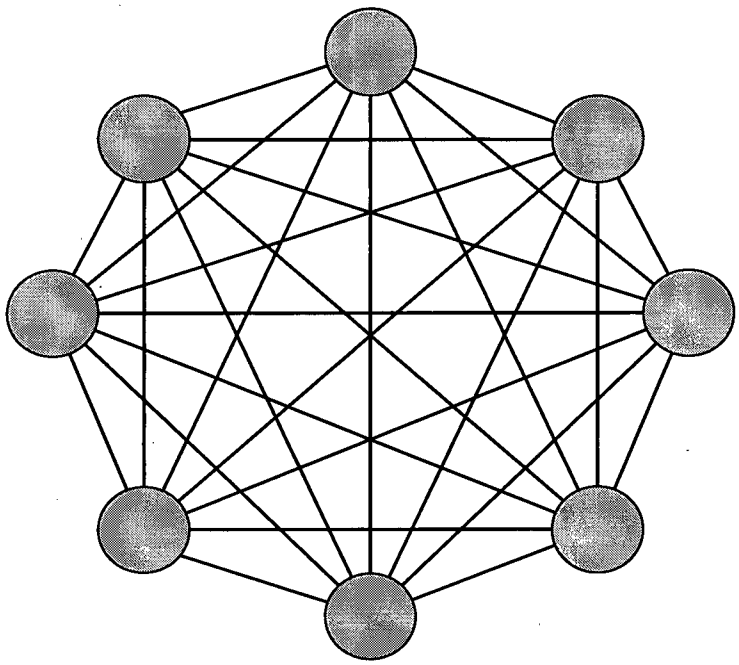


Figura 2.8 : Rede Estática do Tipo Completamente Interconectada.

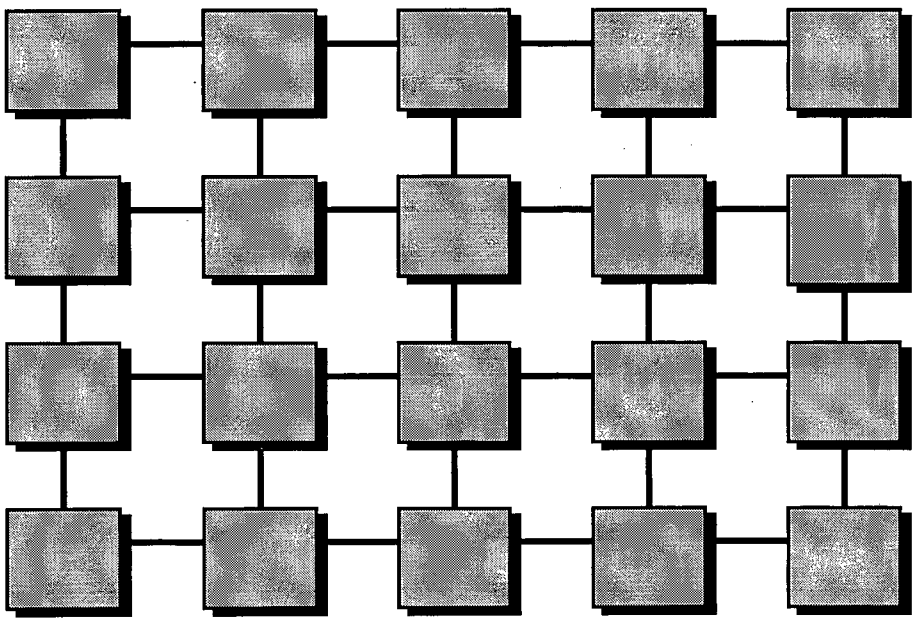


Figura 2.9 : Rede Estática do Tipo Grelha.

Outro tipo de rede estática é o *hipercubo* (Figura 2.10), que consiste de um cubo com dimensão n, onde são conectados 2^n nós. O custo de transmissão, no pior caso é igual a n. Em outras palavras, o número de nós percorridos no pior caso é igual a $\log_2 x$, onde x é o número de nós conectados na rede.

Embora o hipercubo proporcione um ganho significativo no custo de transmissão, o número de canais conectados à cada nó é igual à sua dimensão. Esta topologia possui um custo menor de transmissão que a grelha.

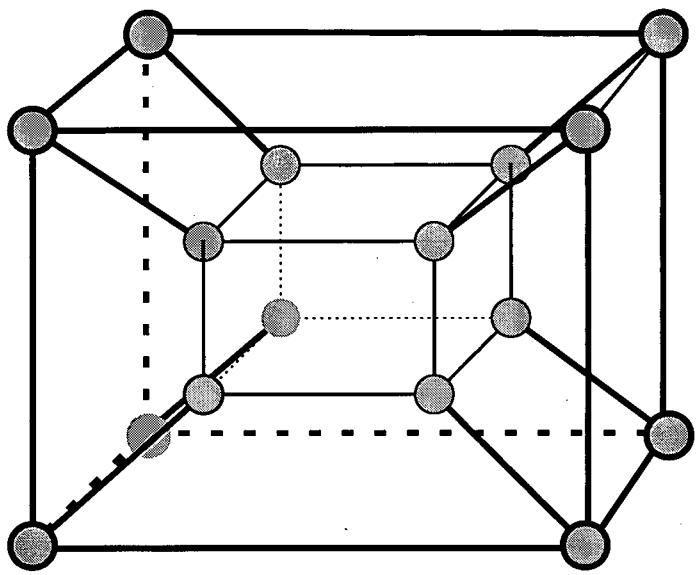


Figura 2.10 : Rede Estática do Tipo Hipercubo

- **Redes Dinâmicas**

As redes dinâmicas possuem a característica de poder alterar dinamicamente sua topologia através de comutadores de conexões. Assim, a máquina pode se configurar conforme a necessidade da aplicação. Dois modelos clássicos são as redes que utilizam *Crossbar* (Figura 2.11) e *Multi-Estágio*.

⇒ Em um dispositivo chamado *Crossbar*, existem várias linhas de entrada e de saída, onde no cruzamento de cada uma destas linhas há um comutador, que permite a conexão entre linhas de entrada e linhas de saída.

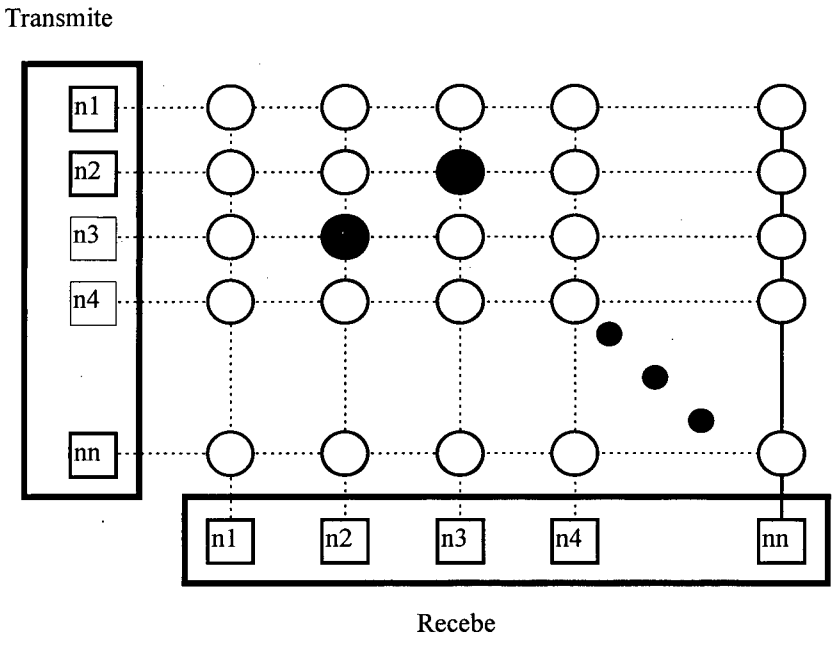


Figura 2.11 : Rede Dinâmica do Tipo *Crossbar*

⇒ As redes *Multi-Estágio* são formadas, normalmente, pela combinação de múltiplos elementos com capacidade de conectar, simultaneamente, duas entradas à duas saídas. A partir dessa combinação, é possível construir uma rede dinâmica que permite a conexão entre quaisquer elementos, como é mostrado na Figura 2.12, que exemplifica uma rede multi-estágio do tipo ômega.

Nas redes ômega, porém, a interconectividade simultânea entre todos os elementos é comprometida, pois cada conexão acarreta o bloqueio de certos comutadores, tornando impossível o estabelecimento de conexões que dependam destes comutadores. Este problema pode ser resolvido através da adição de comutadores e estágios, gerando outros tipos de redes multi-estágio como as redes Benes.

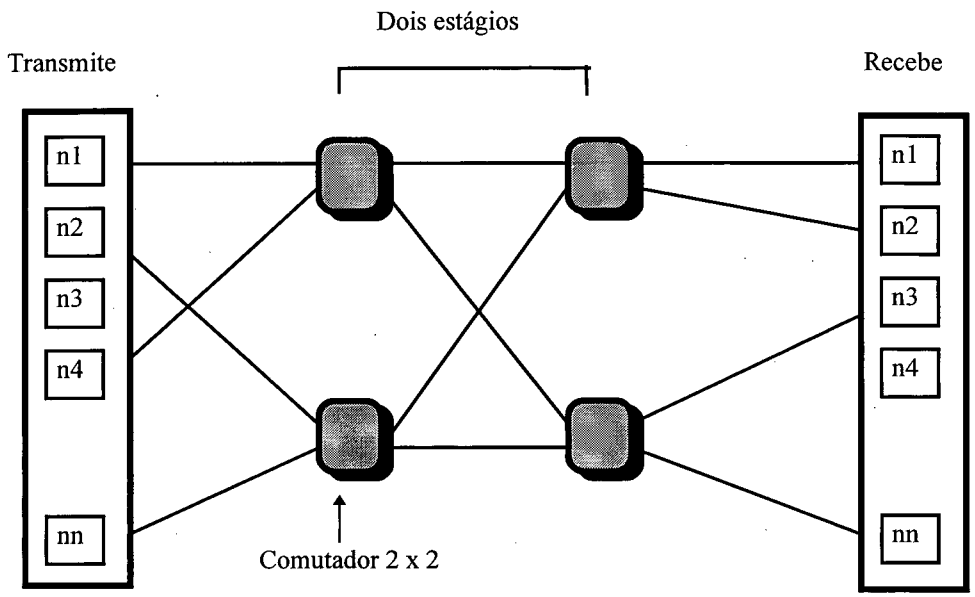


Figura 2.12 : Rede Multi-Estágio do Tipo Ômega

CAPÍTULO 3

Os Sistemas Operacionais Distribuídos

Um sistema de computação distribuído é uma coleção de computadores independentes que se apresentam para o usuário como um único computador [TAN95]. Como os sistemas operacionais distribuídos são implementados em processadores que possuem memória privativa, se faz necessário a implementação do mecanismo de troca de mensagens entre processos.

3.1 Comunicação entre Processos

Em ambientes distribuídos, o sistema operacional deve prover a comunicação uniforme entre processos independentemente do nó em que está, do nó que comanda as conexões, e da rede de conexão. Cada nó da máquina deve conter um micronúcleo residente que implementa serviços que suportem vários processos.

Um processo se comunica com outro pelo envio e recebimento de mensagens. Basicamente os serviços de comunicação entre processos são *send (mensagem)*, para o envio de uma mensagem a um processo destinatário, e *receive (mensagem)* para o recebimento de uma mensagem de um processo origem.

O projeto de primitivas de comunicação depende de decisões referentes à disciplina de comunicação, endereçamento, sincronismo, armazenamento temporário e fluxo. As disciplinas de comunicação são definidas como os padrões de troca de mensagens entre dois processos.

O endereçamento consiste na forma pela qual um processo se refere a outro, podendo ser feito o endereçamento direto, através de seus identificadores, ou endereçamento indireto envolvendo um elemento intermediário chamado caixa postal, para o qual os

processos emissores enviam uma mensagem e do qual os processos receptores obtém uma mensagem.

Quanto ao sincronismo a comunicação pode ser síncrona ou assíncrona. Na forma síncrona, as primitivas de comunicação implementam o bloqueio do processo emissor até que o processo receptor esteja apto a receber a mensagem. Na forma assíncrona, o processo emissor é desbloqueado após a mensagem ter sido copiada para o núcleo de transmissão e apenas o processo receptor permanece bloqueado até a chegada de uma mensagem.

O armazenamento temporário de mensagens é utilizado no modelo assíncrono, onde um processo receptor mais lento pode gerar um conjunto de mensagens pendentes. Este problema pode ser resolvido pelo núcleo gerindo filas de mensagens, uma por processo ou caixa postal, com um tamanho limitado. No caso de ocorrer esgotamento do espaço de armazenamento temporário, o processo emissor fica bloqueado até o processo receptor executar uma primitiva de recepção, liberando o espaço.

O fluxo de mensagens em uma comunicação pode ser unidirecional ou bidirecional. O fluxo unidirecional é normalmente utilizado em comunicações assíncronas, onde um processo emissor não permanece esperando por um resposta do processo receptor, não sendo necessário a comunicação no outro sentido. O fluxo bidirecional é utilizado em comunicações síncronas, normalmente aplicações cliente-servidor (descrito adiante), onde o processo servidor após receber e processar requisições de clientes, responde com o resultado da operação.

3.1.1 Disciplinas de comunicação

Existem vários tipos de disciplinas de comunicação, que são definidas como padrões de trocas de mensagens entre dois processos [COR93]. Essas disciplinas são utilizadas para facilitar implementações de comunicação adequadas para as aplicações. Embora existam vários tipos de disciplinas de comunicação, são tratados neste trabalho somente dois modelos, apresentados a seguir.

Modelo Produtor-Consumidor

No modelo produtor-consumidor, um processo que possui um canal unidirecional de emissão através do qual executa uma seqüência de envio de mensagens é chamado produtor. Um processo que possui um canal unidirecional de recepção através do qual executa uma seqüência de recepção de mensagens é chamado consumidor. Quando um canal de emissão de um produtor é conectado ao canal de recepção de um consumidor, desenvolve-se entre eles uma disciplina de comunicação do tipo produtor consumidor (Figura 3.1). Com o uso desta disciplina é possível construir um *pipeline* de processos.

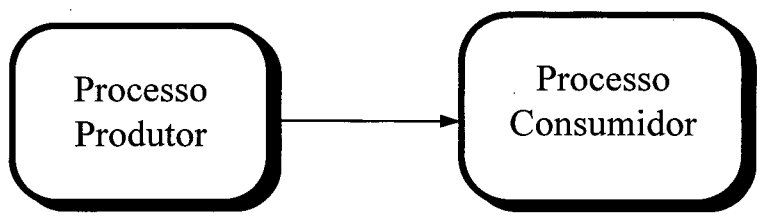


Figura 3.1 : Disciplina de Comunicação Produtor-Consumidor

Modelo Cliente-Servidor

No modelo cliente-servidor o sistema operacional pode ser dividido em módulos, cada um dos quais se encarrega de um serviço do sistema . Esses módulos podem ser distribuídos por vários processadores na máquina. Neste modelo, um processo servidor permanece aguardando pedidos de serviços, vindos de processos clientes na forma de mensagens. Estes processos, cliente e servidor, possuem um canal bidirecional para envio e recebimento de mensagens. Após o recebimento de uma requisição de serviço, o processo servidor analisa a mensagem, processa o pedido, retorna ao cliente o resultado da operação e volta a aguardar a chegada de outras requisições. Esse processo pode visualizado na Figura 3.2 para melhor entendimento.

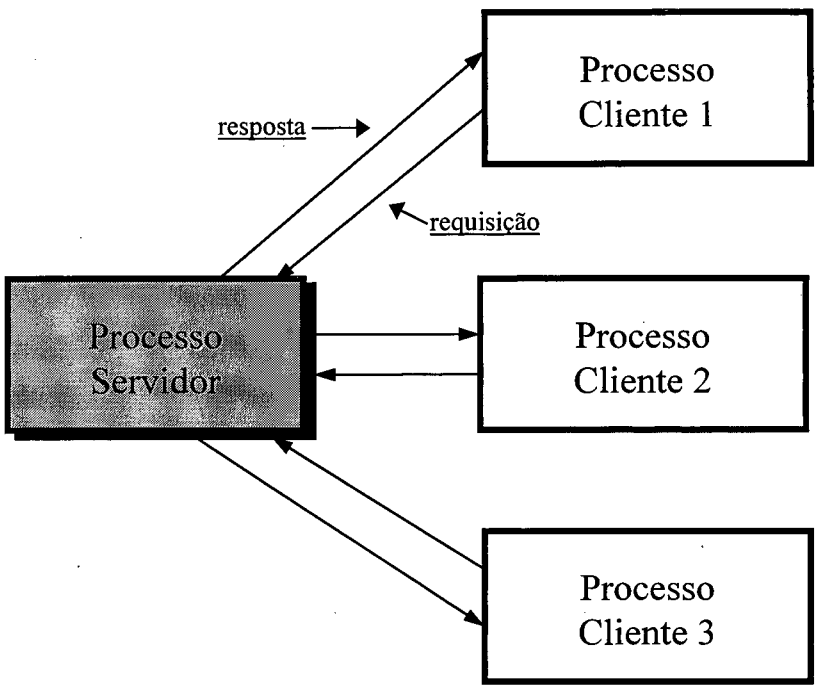


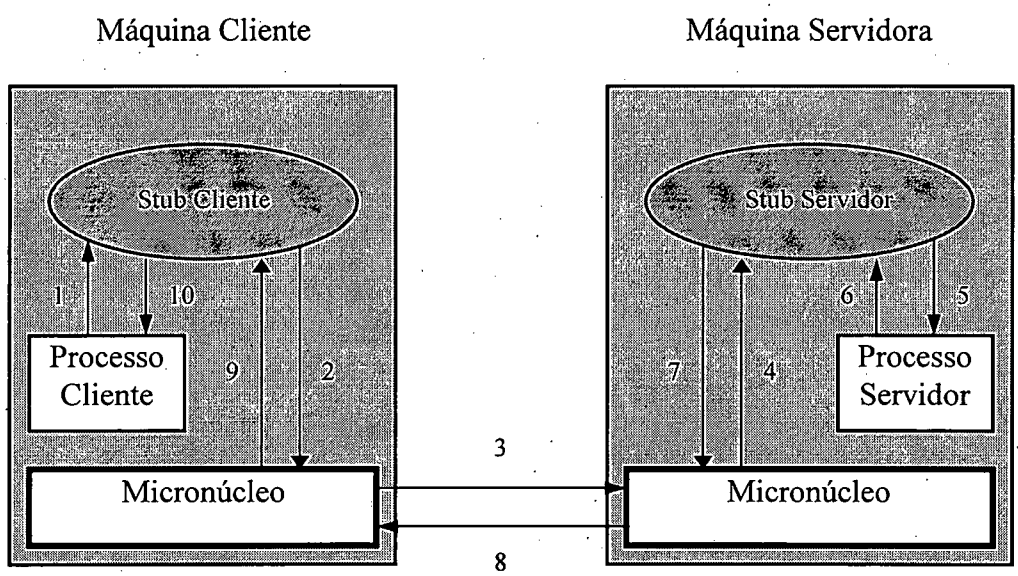
Figura 3.2 : Disciplina de Comunicação Cliente-Servidor

3.2 As Chamadas de Procedimento Remoto

A chamada de procedimento remoto é implementada sobre o modelo cliente-servidor, no qual processos usuários requisitam serviços de processos servidores. Como nos sistemas operacionais distribuídos os processos são colocados em processadores com memórias privativas e as requisições necessitam de comunicação entre processos. Chamadas de procedimento remoto (RPC), podem ocultar essas comunicações tornando a programação em ambientes distribuídos similar a de ambientes centralizados.

As aplicações, a fim de acessarem os serviços oferecidos pelo sistema, utilizam *stubs* clientes que criam mensagens, com informações sobre o tipo de serviço e seus parâmetros, para serem enviadas ao servidor (Figura 3.3) [COU88]. Quando o servidor recebe a requisição através de um *stub* servidor, atende a requisição e coloca o resultado da operação em um *stub* servidor. O *stub* servidor, por sua vez , coloca o resultado da operação

em uma mensagem enviada ao *stub* cliente. Assim, o processo cliente tem a impressão de ter executado um procedimento local, sem perceber as funções de envio, recepção, montagem e desmontagem de mensagens (Figura 3.3).



Legenda:

1. O processo cliente requisita serviço ao processo servidor fazendo uma chamada de procedimento.
2. O stub cliente empacota a requisição e seus parâmetros em uma mensagem e pede para o micronúcleo enviá-la.
3. O micronúcleo envia a mensagem.
4. Do outro lado da linha, o micronúcleo do nó onde se encontra o processo servidor recebe a mensagem e a envia para o stub servidor.
5. O stub servidor desempacota a mensagem, formando uma chamada de procedimento ao servidor.
6. O servidor executa o serviço e retorna o resultado para o stub servidor.
7. O stub servidor empacota a resposta em uma mensagem e pede para o micronúcleo enviá-la.
8. O micronúcleo envia a mensagem.
9. O micronúcleo do nó onde se encontra o processo cliente recebe a mensagem e a envia para o stub cliente.
10. O stub cliente desempacota a resposta e a retorna para o processo cliente.

Figura 3.3 : Chamada de Procedimento Remoto

3.3 O Micronúcleo

O micronúcleo é constituído de funções básicas que necessitam executar no modo supervisor, em espaço privilegiado, e que são capazes de fornecer mecanismos para implementar as abstrações de processos e suas comunicações. Essa concepção de núcleo se adapta perfeitamente ao modelo cliente-servidor que é adequado aos sistemas operacionais distribuídos [VAR94]. A seguir trataremos das funções básicas do micronúcleo de um sistema operacional distribuído enfocando as suas características.

3.3.1 Funções Básicas do Micronúcleo de um Sistema Operacional Distribuído

O micronúcleo possui uma estrutura compacta e é responsável por implementar algumas funções básicas em sistemas operacionais distribuídos. Essas funções são descritas a seguir.

- **Gerência de Interrupções e código dependente do processador**

O micronúcleo deve conter todo código dependente da máquina, fornecendo proteção ao sistema contra processos não privilegiados e tratando as interrupções ocorridas no sistema.

- **Gerência de Processos**

O micronúcleo deve fornecer mecanismos essenciais para gerenciamento de processos, como por exemplo, o escalonador de processos. Em sistemas operacionais modernos o micronúcleo implementa o conceito de *threads*. As *threads* são fluxos de execuções independentes no interior de um processo.

- **Comunicação entre Processos**

Como os sistemas operacionais distribuídos precisam da comunicação entre processos, no qual o paradigma é a troca de mensagens, o micronúcleo deve prover funções básicas desse tipo.

- **Gerência de Memória**

No interior do micronúcleo, normalmente é colocado o código que controla o hardware de gerenciamento de memória (paginação, segmentação e etc.). Apesar disso, a implementação das estratégias de gerenciamento da memória podem ficar fora do núcleo.

CAPÍTULO 4

Algoritmos Paralelos

Um algoritmo paralelo pode ser visto como um conjunto de tarefas independentes ou processos que podem ser executados concorrente e cooperativamente para resolver um dado problema. Durante a execução de um algoritmo paralelo diferentes processos interagem pela sincronização e troca de mensagens.

4.1 Complexidade de Algoritmos Paralelos

A complexidade de tempo de um algoritmo paralelo não pode ser determinado simplesmente pela contagem do número de operações elementares envolvidas na computação, como é o caso de algoritmos seqüenciais. Em um algoritmo paralelo um passo elementar refere-se a um conjunto de operações elementares que podem ser executados simultaneamente pelo conjunto de processadores. A complexidade de tempo de um algoritmo paralelo é determinada pelos passos elementares e o roteamento de dados, onde o tempo requerido para o roteamento de dados depende do padrão de interconexão entre os processadores [CHA92].

Algoritmos paralelos são executados por um conjunto de processadores cooperativamente e usualmente requer transferência de dados entre os processadores para execução. Assim, dois diferentes tipos de operações são envolvidas: a computação local de um processador e o roteamento dos dados entre os processadores.

O projeto de um algoritmo paralelo eficiente envolve a escolha apropriada de estrutura de dados, alocação de processadores e finalmente execução do algoritmo. Esses três aspectos juntos podem ser denominados como organização computacional. Algumas organizações computacionais que são freqüentemente usadas na solução de vários problemas em computadores paralelos são introduzidos como modelos de programação.

4.2 Modelos de Programação Paralela

Existe uma variedade de modelos para computação paralela e distribuída, incorporando diferentes aspectos ao poder de computação de cada processador e ao mecanismo de transferência de informação entre eles.

Modelos de programação paralela são abstrações de programas que fornecem ao programador uma visão transparente e simplificada do sistema de *hardware/software* do computador. São especificamente projetados para multiprocessadores e multicomputadores. Cinco modelos são caracterizados a seguir para esses computadores, que exploram o paralelismo segundo diferentes paradigmas .

4.2.1 Modelo de Memória Compartilhada

As unidades computacionais básicas em um programa paralelo são os processos, que correspondem ao desempenho de operações descritas nos segmentos de código [HWA93]. O nível de paralelismo de um algoritmo depende de como a comunicação entre os processos é implementada

O espaço de endereçamento dos processos pode ser compartilhado ou restrito. Para assegurar a ordenação na comunicação entre os processos, é utilizada um mecanismo de exclusão mútua, garantindo o acesso exclusivo a um objeto por um processo em um dado tempo. A seguir serão apresentados os aspectos do modelo de variáveis compartilhadas e suas soluções.

Comunicação Através de Variáveis Compartilhada

A programação em multiprocessadores é baseada no uso de variáveis compartilhadas em uma memória comum para a comunicação entre os processos. Para isso, é necessário o uso de uma memória compartilhada e o uso de um mecanismo de exclusão mútua entre os vários processos que acessam o mesmo conjunto de variáveis. Neste modelo, os pontos

importantes são o acesso protegido das regiões críticas, consistência da memória, a indivisibilidade das operações de memória, a sincronização , a implementação da estrutura de dados compartilhada, e técnicas para manipulação rápida de dados.

Uma região crítica é um segmento de código acessando as variáveis compartilhadas, e deve ser executado por um único processo de cada vez. Na região crítica, um processo não pode ser interrompido até terminar seu processamento na região crítica. Deve ser assegurado que pelo menos um processo tenha sucesso na entrada da região crítica a cada instante.

Um problema associado ao uso de uma região crítica está em seu tamanho. Se for demasiadamente grande, ele pode limitar o paralelismo devido a espera excessiva por parte dos processos. E por outro lado, se for pequeno, ele pode adicionar complexidade desnecessária de código.

Programação com variáveis compartilhadas requer uma operação atômica especial para a comunicação entre processos, construções específicas para a expressão do paralelismo, suporte de compilação para exploração do paralelismo, e um sistema operacional para escalonamento de eventos paralelos.

Multiprocessamento toma várias formas, dependendo do número de usuários e da granularidade da computação. Um multiprocessador pode ter uso na solução de um único problema grande ou na execução de múltiplos programas através dos processadores.

- **Multiprogramação**

Multiprogramação é definida como a existência de múltiplos programas executando independentemente em um único processador compartilhando o uso dos recursos do sistema. Um multiprocessador multiprogramado permite que múltiplos programas concorrentes compartilhem os processadores do sistema.

- **Multiprocessamento**

Se as comunicações entre os processadores são manuseadas a nível de instruções, o multiprocessador opera em modo MIMD (múltiplos instruções múltiplos dados). Se as comunicações entre os processadores são manuseadas a nível de programas, subrotinas ou

procedimentos, a máquina opera em modo MPMD (múltiplos programas sobre múltiplos fluxos de dados)

Em outra palavras, o multiprocessamento MIMD refere-se à granularidade fina e o multiprocessamento MPMD refere-se à granularidade grossa. Em ambos modos de multiprocessamento, as variáveis compartilhadas são usadas para realizar comunicação entre processos.

- **Multitarefa**

Um único programa pode ser particionado em múltiplas tarefas interrelacionadas, executadas concorrentemente em um multiprocessador. Um programa assim composto requer menos tempo de execução. A multitarefa é realizada com adição de códigos ao programa original para fornecer a sincronização das tarefas.

Somente quando o tempo gasto com a troca de mensagens é pequeno o multitarefa compensa. Nem todas as partes de um programa pode ser divididas em tarefas paralelas.

- ***Multithreading***

A concepção de *multithreading* é uma extensão da concepção de multitarefa e multiprocessamento. O núcleo do sistema operacional é estendido com propósito de permitir que múltiplas *threads* de processos leves compartilhem o mesmo espaço de endereçamento e executem no mesmo ou em diferentes processadores simultaneamente.

O propósito é explorar o paralelismo de grão fino na construção de multiprocessadores modernos com processadores de múltiplo-contexto ou processadores superescalares com múltiplas instruções. Conflitos de recursos são o maior problema a ser resolvido na arquitetura que utiliza *multithreads.*

O gerenciamento de memória e mecanismos de proteção especiais deve ser desenvolvidos para assegurar integridade e precisão de dados em operações de *threads* paralelos.

• **Particionamento e Replicação**

A meta do processamento paralelo é explorar paralelismo tanto quanto possível com o *overhead* pequeno. O particionamento de programas é uma técnica para decomposição de um programa grande e um conjunto de dados, em muitos pedaços pequenos para a execução paralela por múltiplos processadores. Técnicas de reestruturação de programas podem ser usadas para transformar programas seqüenciais em uma forma paralela mais conveniente para multiprocessadores. Estas transformações devem ser realizadas automaticamente pelo compilador. O particionamento é freqüentemente praticado em sistemas de multiprocessadores com memória compartilhada.

A replicação de programas refere-se a replicação do mesmo código de programa para execução paralela em múltiplos processadores sobre os diferentes conjuntos de dados. A replicação é mais conveniente para multicomputadores com troca de mensagens e memória distribuída.

• **Escalonamento e Sincronização**

Escalonamento de módulos de programa divididos em processadores paralelos são muito mais complicados que escalonamento de programas seqüenciais em um único processador. Existem dois tipos de escalonamento : o escalonamento estático possui a vantagem da simplicidade na execução da tarefa de escalonamento mas é difícil estabelecer um tempo preciso da execução de cada tarefa provocando a má utilização dos recursos do sistema, e o escalonamento dinâmico que requer chaveamento rápido de contexto, preempção, e muito mais suporte do sistema operacional. A vantagem do escalonamento dinâmico é melhor utilização dos recursos, mas com maior complexidade no escalonamento. Métodos estáticos e dinâmicos podem ser usados juntamente em um sofisticado sistema com multiprocessadores que demandem alta eficiência.

• **Proteção e Coerência de cache**

Além de manter a coerência da cache na memória hierárquica, os multiprocessadores devem assumir a consistência de dados entre os caches privadas e a memória compartilhada.

O problema de coerência em multicache demanda a invalidação ou validação antes de cada operação de escrita. Um sistema de memória é dito ser coerente se o valor retornado em uma instrução de leitura é sempre o valor armazenado pela última instrução de escrita na mesma localização de memória. A ordem de acesso nas caches e na memória principal fazem uma grande diferença nos resultados computacionais.

4.2 2 Modelo de Memória Distribuída

Dois processos residentes em diferentes nós processadores podem se comunicar pela troca de mensagens através de uma rede de interconexão. As mensagens podem ser instruções, dados, sincronização ou sinais de interrupção. O atraso na comunicação causado pela troca de mensagem é maior que o atraso causado pelo acesso de uma variável compartilhada em uma memória comum. Existem dois modelos de troca de mensagem : modelo de troca de mensagem síncrona e modelo de troca de mensagem assíncrona.

- **Troca de Mensagem Síncrona**

Visto que a memória não é compartilhada, não é necessário um mecanismo de exclusão mútua. A troca de mensagens síncronas pode sincronizar o processo remetente e o processo destinatário no tempo e espaço, que devem ser ligados por canais de comunicação físicos. Em geral, *buffers* não são usados nos canais de comunicação. Isso porque, as comunicações síncronas podem ser bloqueadas pela existência de canais ocupados ou com um erro, visto que, só é permitida a transmissão de uma mensagem em um dado momento, através de um canal de comunicação. Em outras palavras, se um processo está pronto para comunicar e o outro não está, o que está pronto deve ser bloqueado.

- **Troca de Mensagem Assíncrona**

As comunicações assíncronas não requerem que a mensagem enviada seja sincronizada no tempo e espaço. Buffers são freqüentemente usados nos canais, que resultam em troca de mensagens não bloqueantes, com o fornecimento de buffers. Este esquema é semelhante a um serviço postal usando caixas postais (buffers de canais) sem sincronização entre remetentes e destinatários.

Processos não bloqueantes podem ser realizados pela troca de mensagens assíncronas em que dois processos não tem que ser sincronizados também no tempo em espaço. O remetente tem permissão para enviar uma mensagem sem bloqueio, apesar do destinatário não estar pronto. Visto que buffers de canais são finitos, o remetente pode ser eventualmente bloqueado.

A questão crítica na programação deste modelo está na distribuição e na duplicação do código do programa e do conjunto de dados nos nós processadores. Um balanço entre tempo de computação e o tempo gasto na comunicação deve ser considerado.

A replicação de programa e distribuição de dados são usados em multicomputadores. Os processadores em um multicomputador são fracamente acoplados no sentido que eles não tem memória compartilhada. A troca de mensagem em um multicomputador é manipulada a nível de subprograma. Isso é porque o paralelismo explícito é mais atrativo para multicomputadores.

4.2.3 Modelo de Dados Paralelos

Com as operações de bloqueio em computadores SIMD, o código de dados paralelos é fácil de ser escrito e acompanhado passo a passo porque o paralelismo é manipulado explícitamente pela sincronização de hardware e fluxo de dados. Neste modelo existe a necessidade da distribuição prévia do conjunto de dados. Assim, a escolha das estruturas de dados paralelos e a interconexão entre elas (para facilitar operações de troca de dados) são pontos chaves. Em resumo, a programação de dados paralelos enfatizam computações locais e operações de roteamento de dados, e é aplicada para problemas de granularidade fina usando grelhas regulares e matrizes.

Paralelismo de dados pode ser implementado em computadores SIMD ou em multicomputadores SPMD (um programa sobre múltiplos fluxos de dados), dependendo do tamanho do grão e modo de operação adotado. O paralelismo de dados freqüentemente conduz para um alto grau de paralelismo envolvendo milhões de operações de dados concorrentemente. Isto é diferente do controle de paralelismo que oferece um baixo grau de paralelismo.

As comunicações entre elementos processadores são controladas diretamente pelo hardware. A execução de instruções sincronizadas e operações de roteamento de dados feitas por computadores SIMD é mais eficiente na exploração do paralelismo espacial em um *array* grande, grelhas, ou malhas de dados.

Em um programa SIMD, instruções escalares são diretamente executadas pela unidades de controle. Vetores de instrução são transmitidos para todos os elementos processadores. Vetores de operandos são carregados nos elementos processadores das memórias locais simultaneamente usando um endereço global com diferentes *offsets* nos registradores de índice locais. Vetores de armazenamento podem ser executados em uma maneira similar . Dados constantes podem ser transmitidos para todos os elementos processadores simultaneamente. As linguagens de programação SIMD devem ter um espaço de endereçamento global para o roteamento de dados explícito entre os elementos processadores.

O compilador deve separar o programa em componentes escalares e paralelos e integrar com o ambiente. Programas SIMD podem ser recompilados para arquitetura MIMD. A idéia é desenvolver um precompilador fonte para fonte para converter, por exemplo, programas de uma maquina de conexão executando em um multicomputador de troca de mensagem no modo SPMD.

Programas SPMD são uma classe especial de programas SIMD que enfatizam paralelismo de grão médio e sincronização a nível de subprograma menor que a nível de instrução. Neste sentido, o modelo de programação de dados paralelos se aplica a ambos computadores SIMD síncrono e computadores MIMD fracamente acoplados. A conversão de programas entre diferentes arquitetura de máquina é muito necessário para estender a portabilidade do *software*.

4.2.4 Modelo Orientado a Objeto

Neste modelo, objetos são criados e manipulados dinamicamente. O processamento é desempenhado pelo envio e recebimento de mensagens entre objetos. Modelos de

programação concorrente são constituídos de objetos de baixo nível tal como processos, filas e semáforos dentro de objetos de alto nível semelhantes a monitores e módulos de programa.

As abstrações de programas levam a modularidade de programas e reusabilidade de software, como é freqüentemente encontrado na programação orientada a objeto. Outras áreas que têm encorajado o crescimento da programação orientada a objeto são: desenvolvimento de ferramentas de CAD (projeto auxiliado por computador) e processadores de texto com recursos gráficos.

Objetos são entidades de programa que encapsulam dados e operações dentro de uma única unidade computacional. O desenvolvimento da programação orientada a objetos concorrente (COOP) fornece um modelo alternativo para computação concorrente em multiprocessadores ou em multicomputadores. Vários modelos de objetos diferem no comportamento interno dos objetos e como eles interagem com outros objetos.

Três padrões comuns de paralelismo tem sido encontrados na prática da programação orientada a objeto paralelo. O primeiro, a concorrência *pipeline,* onde. a computação é divida em um número de passos, chamados de segmentos. A saída de um segmento é a entrada do próximo segmento, sendo que cada segmento executa sua tarefa ao mesmo tempo. Sendo assim, a primeira execução do *pipeline* terá um tempo igual a soma dos tempos de execução em cada segmento. Como, executam paralelamente, assim que termina sua primeira execução o primeiro segmento envia o resultado ao segundo segmento, que inicia a sua primeira execução em paralelo com a segunda execução do primeiro segmento. O mesmo acontecendo com os segmentos seguintes. Desta forma, a segunda execução do *pipeline* terá um tempo de execução igual ao tempo de execução do segmento mais demorado no *pipeline.* Segundo, concorrência dividir e conquistar envolve a elaboração concorrente de subprogramas diferentes e a combinação de suas soluções para produzir a solução de todo o problema. Neste caso, não existe interação entre os procedimentos de solução dos subproblemas.

O terceiro padrão é chamado de solução de problema cooperativo. Um exemplo simples é o cálculo do caminho dinâmico (objetos computacionais) de muitos corpos físicos (objetos) sobre a influência mútua de campos gravitacionais. Neste caso, todos objetos

devem interagir entre si, e os resultados intermediários são armazenados em objetos e compartilhados pela troca de mensagens entre eles.

4.3 Conceitos Fundamentais em Processamento Paralelo

O processamento paralelo é definido por Quinn [QUI94] como uma espécie de processamento de informações que enfatiza a manipulação concorrente de dados pertencentes a um ou mais processos, que busca a solução de um problema simples. A seguir são apresentados vários conceitos e terminologias referentes ao processamento paralelo.

4.3.1 *Speed-up*

Speed-up é a razão entre o tempo necessário para o algoritmo seqüencial executar determinado processamento pelo tempo utilizado pelo correspondente algoritmo adaptado para uma máquina paralela.

$$Speed\text{-}up = \frac{\text{tempo seqüencial para executar uma tarefa}}{\text{tempo paralelo para executar uma tarefa}}$$

4.3.2 Latência

A latência de comunicação é definida como o tempo necessário para dois processadores se conectarem e estarem aptos para trocar dados entre si. A latência de comunicação pode ser classificada em latência de *hardware* e de *software*.

A latência de *hardware* é definida como o tempo necessário para dois processadores conectarem-se fisicamente.

A latência de *software* é definida como o tempo necessário para a transferência de zero *bytes* entre os processadores, ou seja, além de incluir a latência de *hardware*, aqui também é considerado o tempo necessário para a máquina acionar o *software* e os protocolos internos de transferência de dados entre os nós.

A latência de comunicação de uma máquina paralela é fixa e independe do tamanho da mensagem. Nos algoritmos paralelos deve-se procurar transferir o máximo possível de dados numa única operação de comunicação entre processadores, para diminuir o tempo da latência de comunicação.

4.3.3 Granularidade

⇒ **Granularidade de Hardware** - A granularidade de uma máquina paralela pode ser definida como a relação entre a sua capacidade de processamento, em *Mflops*, e a sua capacidade de comunicação entre os nós em MB/s.

$$\text{Granularidade} = \frac{\text{Capacidade de Processamento}}{\text{Capacidade de Comunicação}}$$

Quando máquinas paralelas apresentam um valor alto na relação é dita uma máquina de granularidade grossa, enquanto que, em casos em que apresentam o valor de relação baixo, a máquina é dita de granularidade fina. Quanto maior a granularidade de uma máquina paralela maior a dificuldade de se obter alto *speed-up* nos algoritmos nela processados.

⇒ **Granularidade de Software** - Os algoritmos desenvolvidos para máquinas paralelas também são classificados pela sua granularidade. Os algoritmos de granularidade grossa possuem tarefas grandes que podem ser processadas independentemente em paralelo e implicam em menor número de comunicações entre os processos e os algoritmos de granularidade fina possuem tarefas pequenas a serem processadas em paralelo que implicam em um maior número de comunicações.

4.3.4 Modelos de Programação

Os modelos de programação de uma máquina paralela estão relacionados com a maneira pela qual elas controlam os programas paralelos em andamento, e podem ser subdivididos em SPMD (*Single Program Multiple Data*) e Mestre/Escravo.

No modelo SPMD, não há controle centralizado e todos os processadores executam o mesmo programa, contribuindo para o processamento a ser realizado.

No modelo Mestre/Escravo o processo mestre centraliza o controle. Ele inicia os processos escravos, distribui o trabalho, sincroniza a comunicação e executa operações de I/O. São executados programas diferentes no processo mestre e nos processos escravos. O processador mestre pode ou não contribuir para o processamento do problema. Em geral, processadores escravos executam todo o processamento.

4.3.5 Escalabilidade

A escalabilidade de uma máquina paralela é a medida da sua capacidade de efetivamente utilizar um número crescente de processadores. Uma arquitetura é escalável se, com o aumento do número de processadores, ela permanece no mesmo patamar de desempenho.

CAPÍTULO 5

O Multicomputador Nó //

Este capítulo descreve o multicomputador Nó //, em desenvolvimento no Curso de Pós Graduação em Ciência da Computação da Universidade Federal de Santa Catarina, pelos grupos de pesquisa em sistema operacionais e arquitetura de computadores. O capítulo está dividido em duas seções. A primeira descreve o hardware do multicomputador proposto. A segunda descreve o sistema operacional desenvolvido para o multicomputador.

5.1 Descrição do Multicomputador Nó //

O multicomputador Nó // é composto por um conjunto de nós processadores interligados através de um comutador de conexões do tipo *crossbar*. Cada nó é formado por um processador i486 com uma memória privativa de até *4Mbytes* e por uma placa de interface para comunicação com outros nós. As conexões entre os nós são estabelecidas pelo *crossbar* dinamicamente, conforme a demanda do programa paralelo em execução. Um dos nós chamado nó de controle, tem a função de receber as requisições de serviço dos demais nós e atendê-las atuando sobre o *crossbar*.

Foram propostos dois modelos de arquitetura para o multicomputador Nó // : um modelo baseado em um barramento comum e um modelo baseado em um sistema de interrupção.

⇒**O Modelo de Arquitetura Baseado em um Barramento Comum** foi a primeira concepção do multicomputador Nó//. Ele consta de um conjunto de nós de trabalho ligados por meio de comutador de conexões e um barramento compartilhado. O comutador de conexões é manipulado durante o funcionamento normal da máquina pelo nó de controle, que por sua vez, utiliza um barramento de serviço e um *link* de configuração para determinar as necessidades do sistema e definir dinamicamente a estrutura da rede comunicação.

O nó de controle utiliza o barramento de serviço para realizar uma pesquisa seqüencial de modo a determinar os pedidos dos nós de trabalho. Esta pesquisa é feita através do questionamento pelo envio de um comando ao nó inquirido. Se este nó não deseja nenhum serviço, o nó de controle questiona o próximo nó de trabalho. Caso o nó questionado deseje um serviço, ele então envia uma requisição ao nó de controle. Após atender cada pedido, o nó de controle volta ao questionamento seqüencial dos nós de trabalho.

Se o pedido de serviço retornado por nó de trabalho for um pedido de conexão com outro nó, o nó de controle envia ao comutador de conexões a configuração para a conexão pedida. Se o nó solicitado já estiver conectado o pedido é colocado em uma fila de espera e nó de controle prossegue com o *polling*.

Esta arquitetura apresenta um desperdício de tempo na execução do *polling* pelo nó de controle uma vez que, se um nó de trabalho que acabou de ser questionado quiser fazer um pedido de serviço, terá que esperar todos os outros nós da seqüência serem questionados e atendidos, até chegar novamente a sua vez. Com o objetivo de melhorar esse mecanismo, foi proposta uma nova arquitetura, na qual o barramento de serviço é substituído por um sistema de interrupções.

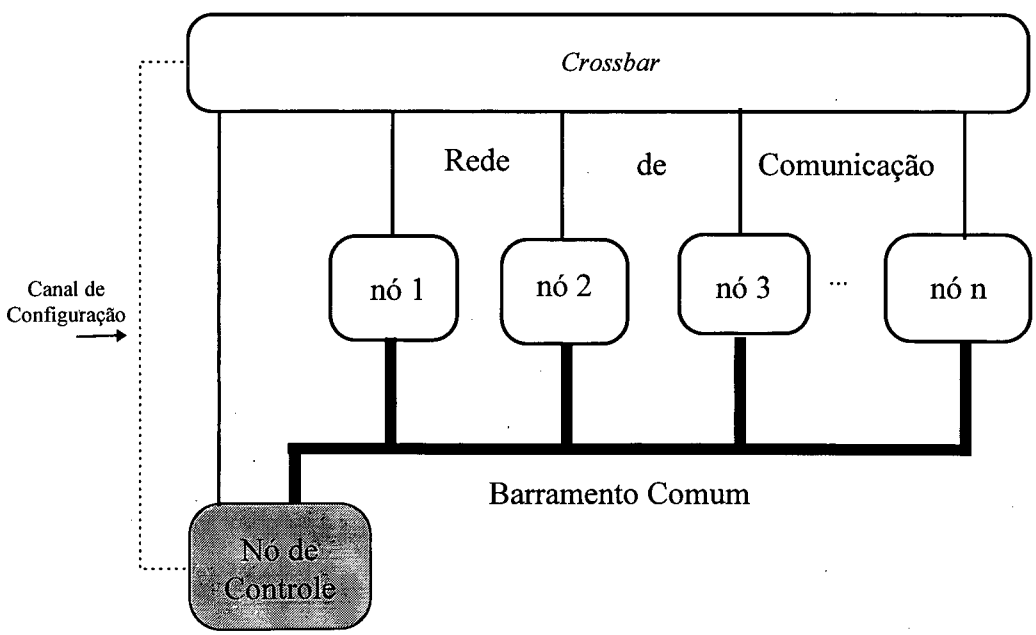


Figura 5.1: Arquitetura Baseada em um Barramento Comum

⇒**O Modelo de Arquitetura Baseada em um Sistema de Interrupção** é composto por linhas de interrupção exclusivas entre os nós de trabalho e o nó de controle (figura 5.1). Entre cada nó de trabalho e o nó de controle existe um par de linhas unidirecionais referidas por INTRtc e INTRct. A linha INTRtc é usada pelo Nó de Trabalho para interromper o Nó de Controle, enquanto que a linha INTRct é utilizada pelo nó de controle para interromper o nó de trabalho.

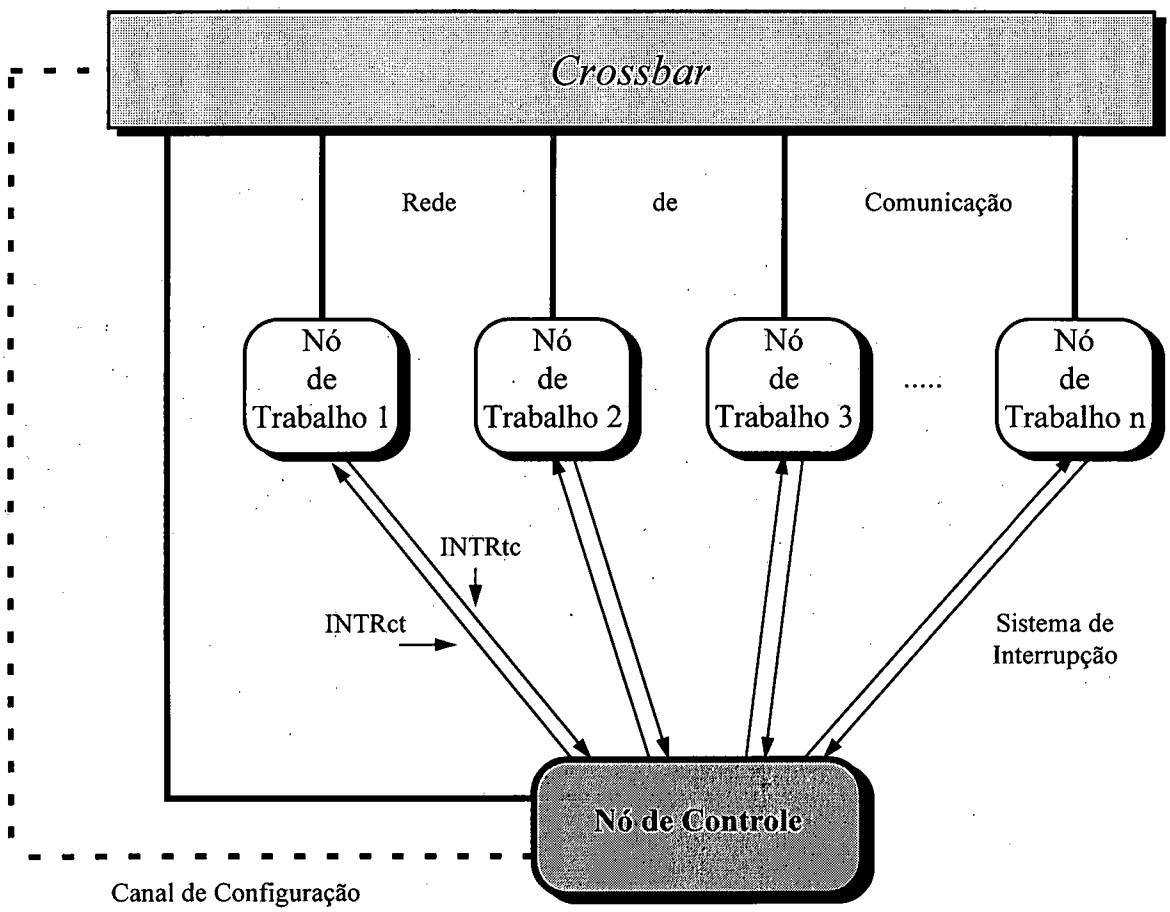


Figura 5.2 : A Arquitetura Baseada em um Sistema de Interrupção para o Multicomputador Nó//

As linhas de interrupção não permitem a transferência de dados. Elas servem para indicar a ocorrência de um evento. O nó de trabalho quando deseja enviar uma requisição de

serviço ao nó de controle, interrompe-o através da linha INTRtc. O nó de controle, por sua vez, conforme a prioridade, estabelece conexão via *crossbar* com o nó que gerou a interrupção. Assim acontece a troca de mensagens de controle como a requisição de serviço. Depois disto, a conexão é desfeita. O nó de controle avaliará a requisição e ao atendê-la ativará a linha de interrupção INTRct do nó de trabalho requisitante, confirmando a realização do serviço requisitado.

Todas as trocas de mensagens, seja de dados ou de controle, são efetuadas através da rede de comunicação. O atendimento a uma requisição tem seu tempo de atendimento reduzido, pois depende apenas do atendimento da interrupção e do chaveamento do *crossbar*. O que diminui o *overhead* associado à comunicação. O modelo baseado em um sistema de interrupções será a abordagem adotada para a construção do protótipo do multicomputador Nó //.

5.1.1 Componentes do Multicomputador Nó //

O multicomputador Nó // é dividido em duas partes : A rede de comunicação e o sistema de interrupção. Nesta sessão serão descritos os componentes do Multicomputador Nó // que formam estas partes.

5.1.1.1 Os Nós de Trabalho

O Multicomputador Nó // é constituído de vários nós com processadores i486 com memória RAM privativa de 4Mb, memória ROM (onde se encontra o micronúcleo), interface com a Rede de Comunicação (canais de comunicação) e interface com o Sistema de Interrupção (linhas de interrupção), como mostra a figura 5.3.

5.1.1.2 O Nó de Controle

O nó de controle é o responsável pelo controle da rede comunicação, no qual gerencia uma tabela de conexões onde estão contidos o estado atual de cada NT e realiza as conexões através de um canal de configuração existente no *crossbar*. E também é responsável pelo atendimento e resposta às interrupções geradas pelos nós de trabalho (figura.5.4).

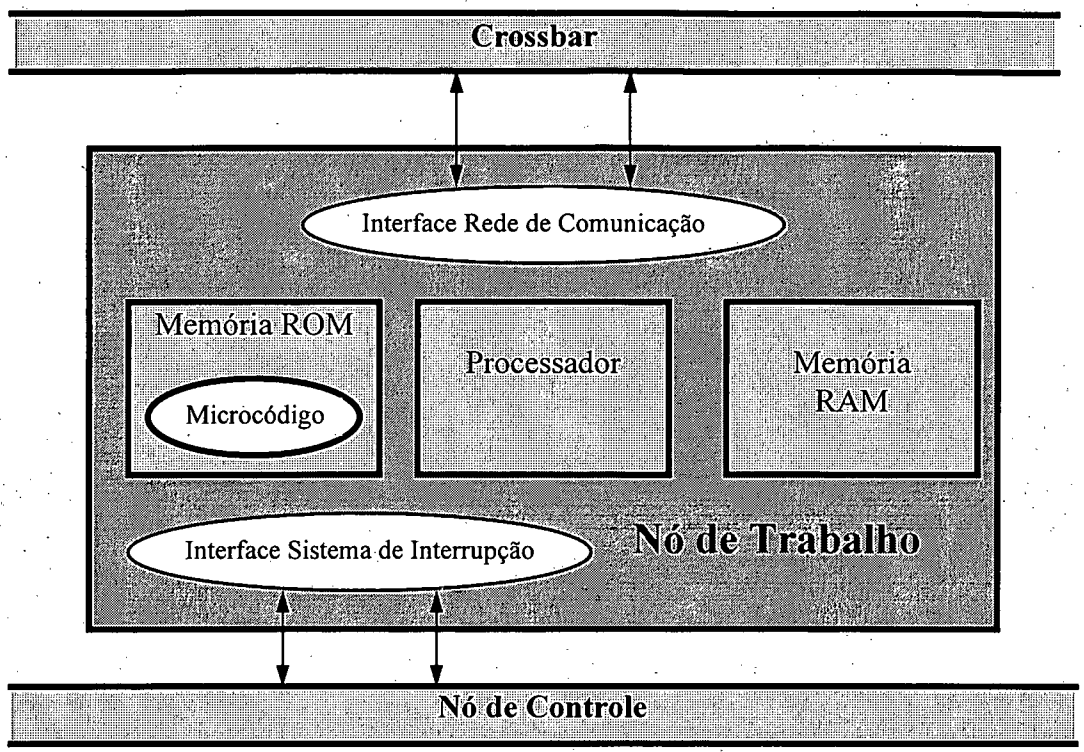


Figura 5.3: Estrutura do Nó de Trabalho

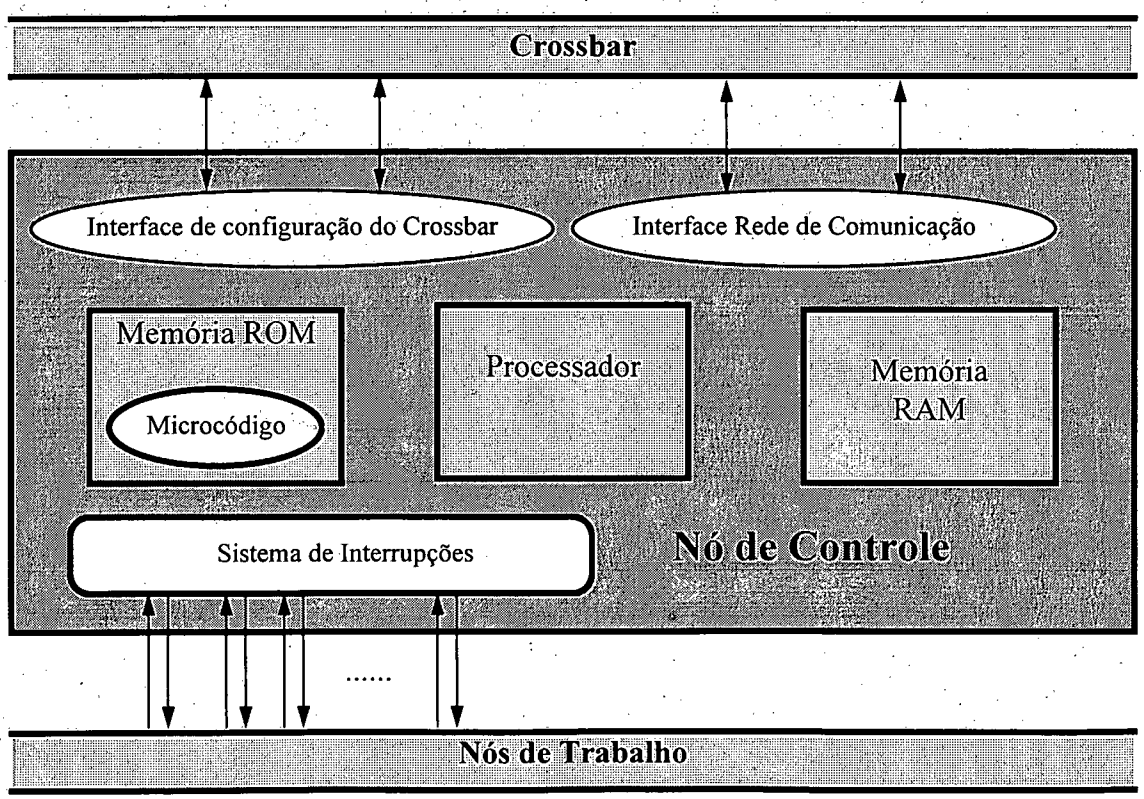


Figura 5.4 : Estrutura do Nó de Controle

5.1.1.3 O *Crossbar*

O comutador de conexões é constituído por um dispositivo de chaveamento chamado *Crossbar* IMS C004. O IMS C004 [INM89], figura 5.5 e 5.6, é um membro da família Transputer-INMOS. Constitui-se em um comutador de conexões programável projetado para fornecer um chaveamento completo entre 32 linhas de entrada e 32 linhas de saída.

O IMS C004 pode chavear *links* (canais) operando a qualquer uma das velocidades, 10M*bits*/seg ou 20M*bits*/seg. Ele introduz , em média, apenas um tempo de atraso de 1.75 *bit* no sinal. Diversas chaves *crossbar* podem ser conectadas em cascata sem perda na integridade do sinal, e podem ser usadas para construir redes reconfiguráveis de tamanho arbitrário. O *crossbar* é programado através de um canal bidirecional serial denominado *link* de configuração.

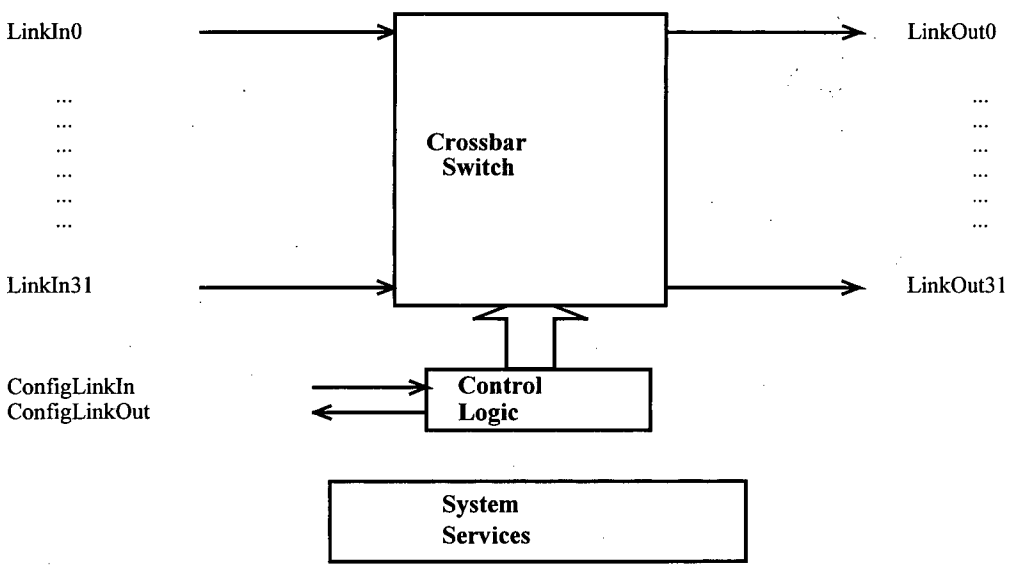


figura 5.5 - Diagrama de blocos básico do IMS C004

O IMS C004 é organizado internamente em um conjunto de multiplexadores de 32-por-1. Cada um destes multiplexadores está associado à um registrador (*latch*) de seis *bits,* dos quais cinco *bits* servem para selecionar uma das entradas (0 a 31) como fonte de dados para a saída correspondente. O sexto *bit* é usado para conectar e desconectar a saída. Estes registradores podem ser escritos e lidos por mensagens enviadas no *link* de configuração

através das linhas **ConfigLinkIn** e **ConfigLinkOut**. A saída de cada multiplexador é sincronizada com um *clock* interno de alta velocidade. Esta sincronização introduz, em média, um tempo de atraso de 1.75 *bit,* conforme já citado.

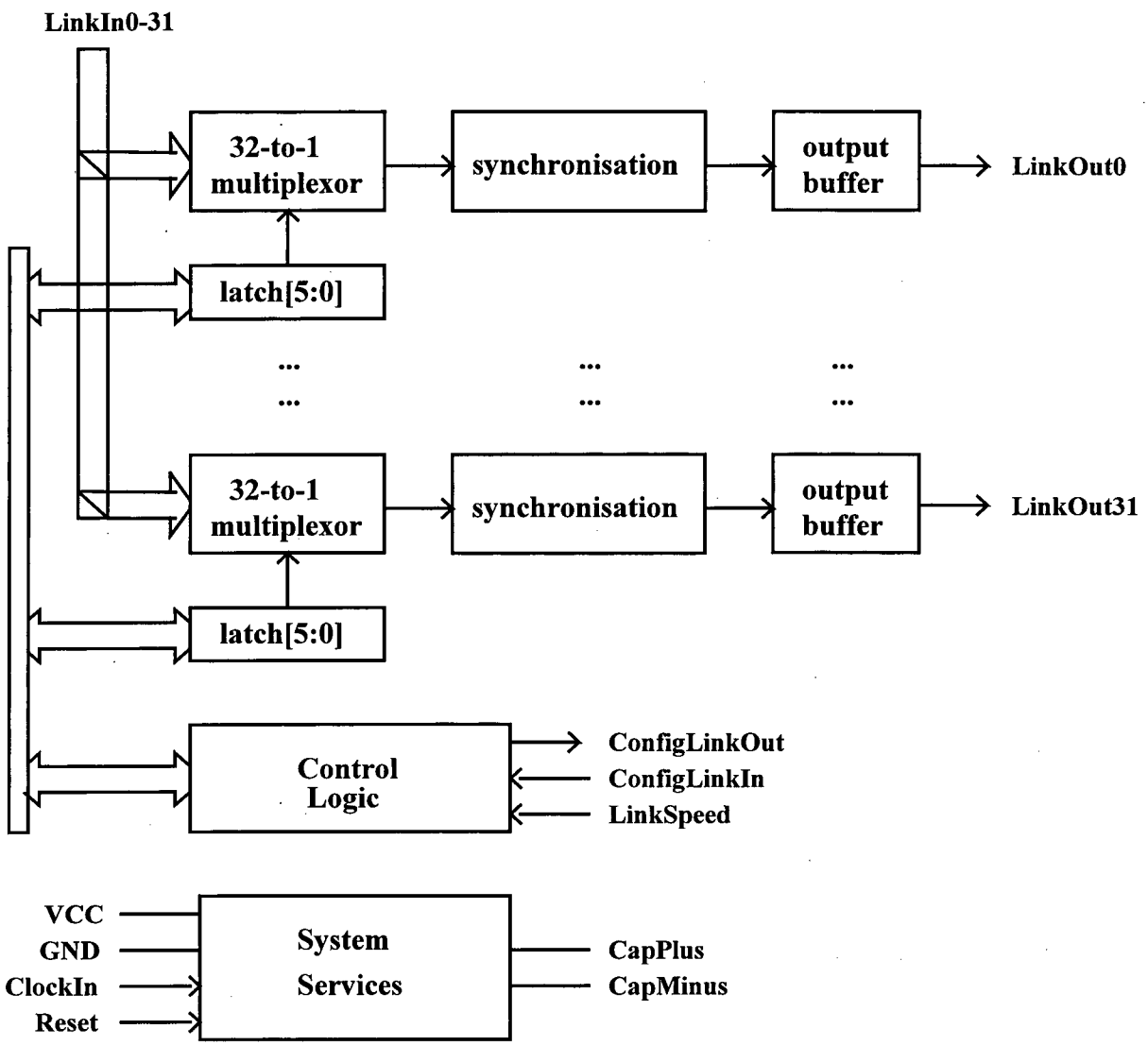


figura 5.6 - Diagrama de blocos do IMS C004

Cada entrada e saída é identificada por um número na faixa de 0 a 31. As mensagens de configuração consistem de um, dois ou três *bytes*, e são enviadas ao *crossbar* através do *link* de configuração, conforme a tabela 1.

Tabela 1 - As mensagens de configuração do IMS C004.

Mensagem de configuração	**Função**
[0] [input] [output]	Conecta **input** a **output.**
[1] [link1] [link2]	Conecta **link1** a **link2** ligando a entrada de **link1** à saída de **link2** e a entrada de **link2** à saída de **link1**.
[2] [output]	Questiona a qual entrada **output** está conectada. O IMS C004 responde com o número da entrada. O *bit* mais significativo deste *byte* indica se a saída está conectada (*bit* setado em um) ou desconectada (*bit* setado em zero).
[3]	Este *byte* de comando deve ser enviado ao final de cada seqüência de configuração, a qual define uma conexão. O IMS C004 então está pronto para receber dados nas entradas conectadas.
[4]	Reinicializa o *Crossbar*. Todas as saídas são desconectadas e levadas ao nível baixo. Isto também acontece quando um sinal de **Reset** é aplicado ao IMS C004.
[5] [output]	Saída **output** é desconectada e levada ao nível zero.
[6] [link1] [link2]	Desconecta a saída de **link1** e a saída de **link2.**

5.1.1.4 Os Canais de Comunicação

Os canais de comunicação utilizados são implementados através de *Link Adaptors* C011 [INM89] que é um membro da família *Transputer-INMOS*. Este componente estabelece uma comunicação *full-duplex* com microprocessadores padrão e com outros sub-sistemas, realizando a conversão de um canal serial bidirecional em uma interface paralela de oito *bits*, e vice-versa.

Todos os produtos INMOS que usam *links* de comunicação, independentemente do tipo de dispositivo, suportam uma freqüência de comunicação padrão de 10 M*bits*/seg; muitos produtos também operam em 20 M*bits*/seg. O *link* IMS C011 pode rodar a qualquer uma destas duas taxas de transferência. A recepção de dados é assíncrona, permitindo que a comunicação seja independente da fase do *clock*.

O *link adaptor* pode ser operado em dois modos. No Modo 1 o IMS C011 estabelece a conversão de um canal bidirecional serial em duas interfaces paralelas de oito *bit*s, uma de entrada e outra de saída, sendo que cada uma delas possui um par de linhas de controle para realização do *handshake* de comunicação.

No Modo 2 o IMS C011 fornece uma interface entre um canal bidirecional serial e um sistema de barramento de microprocessador. Registradores de *status* e de dados para as portas de entrada e saída podem ser acessados através de uma interface bidirecional de oito *bit*s. Há ainda duas saídas de interrupção, uma para indicar dados de entrada disponível e outra para *buffer* de saída vazio. Neste projeto, o IMS C011 é utilizado apenas no Modo 1 de operação, conforme a Figura 5.7.

O *handshake* de comunicação através da porta de saída se dá como é descrito a seguir. Um dado recebido através da linha de entrada **LinkIn** é registrado e apresentado nas linhas de saída **Q0-7**; o *link adaptor* então torna **QValid** alto iniciando o *handshake*. Depois que o dado é lido pelo periférico conectado à saída paralela, o mesmo deve setar **QAck** em nível lógico 1. O IMS C011 então envia a mensagem de reconhecimento ACK através da linha de saída serial **LinkOut**, indicando que transferência do *byte* foi efetivada, e resseta **QValid** levando-o para o nível baixo, completando o *handshake*. O periférico deve ressetar **QAck** tornando-o novamente zero.

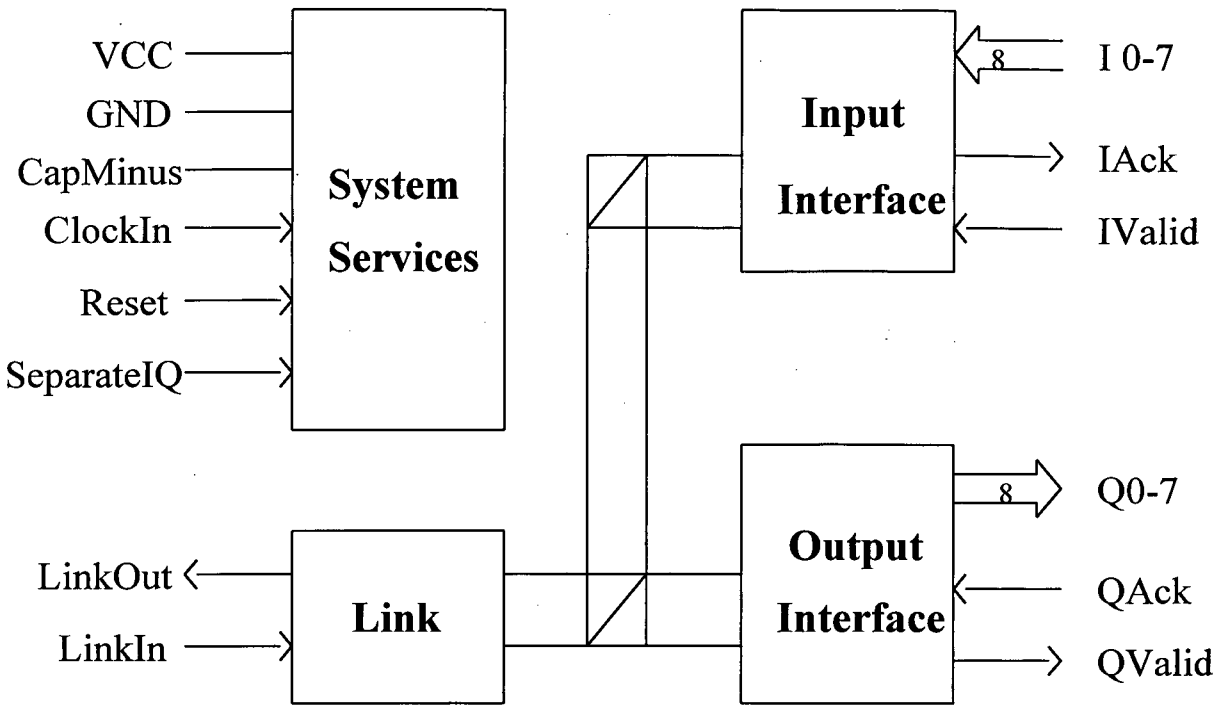


Figura 5.7 - Digrama de Blocos do IMS C011 Operando no Modo 1.

A comunicação através da porta de entrada obedece um *handshake* semelhante ao descrito no parágrafo anterior. Quando um periférico conectado à entrada paralela **I0-7** deseja enviar um *byte*, deve setar **IValid** alto de modo a iniciar o *handshake*. O *link adaptor* então transmite o dado presente em **I0-7** através da linha de saída serial **LinkOut**. Quando a mensagem de reconhecimento ACK é recebida na linha de entrada **LinkIn**, o IMS C011 seta **IAck** alto. Para completar o *handshake*, o periférico deve tornar **IValid** novamente baixo. O *link adaptor* irá ressetar **IAck** levando-o à zero. Um novo dado não pode ser apresentado às linhas **I0-7** enquanto **IAck** não retornar para o nível baixo.

5.1.1.5 Protocolo de Comunicação Serial *Transputer Link*

Este protocolo estabelece a forma de comunicação entre produtos INMOS [INM89] utilizando-se *links* (canais) seriais bidirecionais. Cada um destes *links* possui um canal de

entrada e um canal de saída, através dos quais é feita a transferência de dados. Um *byte* enviado por um *link* é reconhecido na entrada do mesmo *link*, assim, cada linha pode transportar dados e informações de controle.

O canal de saída apresenta-se normalmente em nível baixo. Cada *byte* (Figura 5.8) é transmitido como um *start bit* alto, seguido de um *bit* alto, oito *bits* de dado e um *stop bit* baixo. O *bit* menos significativo do dado é o primeiro a ser enviado. Depois da transmissão de cada *byte* o módulo transmissor espera pelo reconhecimento (*Ack - acknowledge*), que é transmitido pelo receptor do *byte*, consiste de um *start bit* alto seguido de um *bit* zero. Esta informação de controle é recebida pelo canal de entrada do módulo transmissor e informa que o módulo receptor, que recebeu o *byte* transmitido, está pronto para uma receber um novo *byte*.

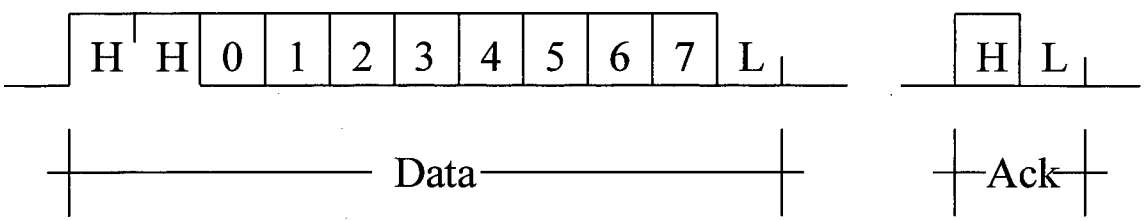


Figura 5.8 : Protocolo de Comunicação : Mensagens de Dado e de Reconhecimento

5.1.1.6. Estrutura de Hardware das Placas de Interface

As interfaces da rede de comunicação, do sistema de interrupção e a de configuração do Crossbar (pertencente apenas ao nó de controle) estão localizadas e uma placa compatível com o barramento PC-AT. A essência das interfaces de com rede de comunicação e configuração do Crossbar está no Link Adaptor IMS C011, o qual fornece um link serial para transferência de dados em alta velocidade. A seguir serão apresentadas as estruturas das placas de interface para os nós de trabalho e de controle, servem para implementar a arquitetura do Nó //.

 # Placa de Interface dos nós de trabalho

A interface da placa com o barramento PC-AT, é responsável pela comunicação entre o microprocessador e as outras interfaces. A interface com a rede de comunicação é utilizada para comunicação entre os nós de trabalho e entre o nó de controle e os nós de trabalho. As comunicações se dão através de conexões estabelecidas pelo Crossbar. A interface com o sistema de interrupção gera uma interrupção (INTRtc) que é enviada ao nó de controle para caracterizar o desejo de solicitação de um pedido, e detecta a chegada de uma interrupção gerada pelo nó de controle (INTRct) (Figura 5.9).

 # Placa de Interface do nó de controle

A placa de interface do nó de controle é semelhante a placa dos nós de trabalho, acrescentando a esta uma interface com o canal de configuração do Crossbar, responsável pela configuração da rede (Figura 5.10).

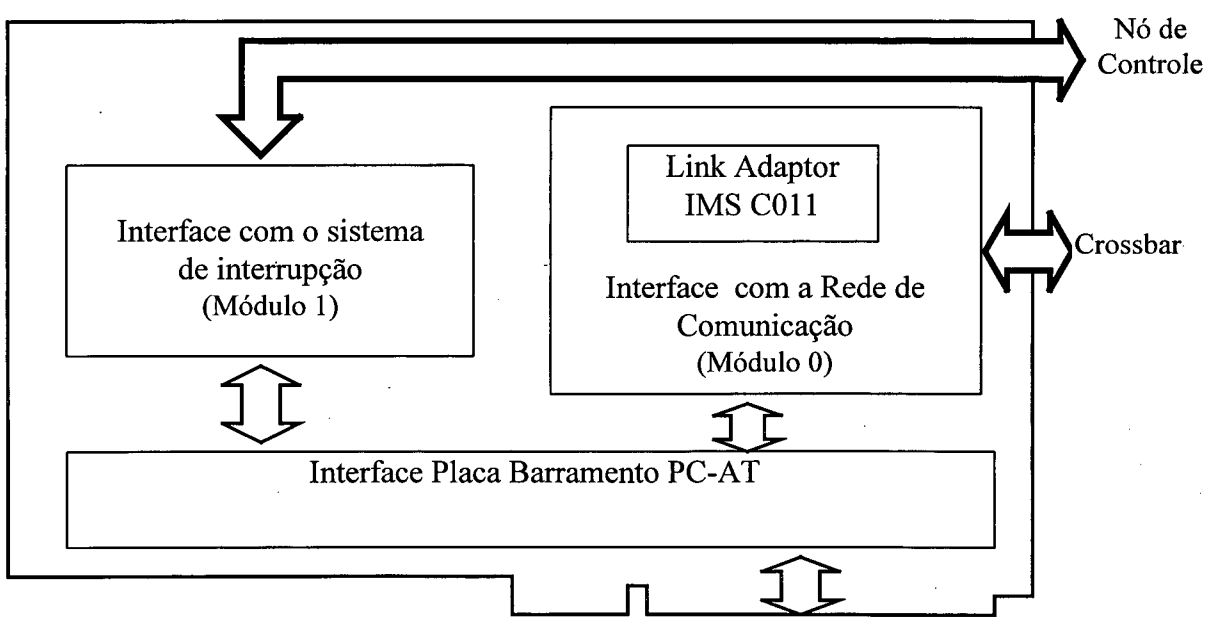


Figura 5.9 : Diagrama de Blocos da Placa de Interface do Nó de Trabalho

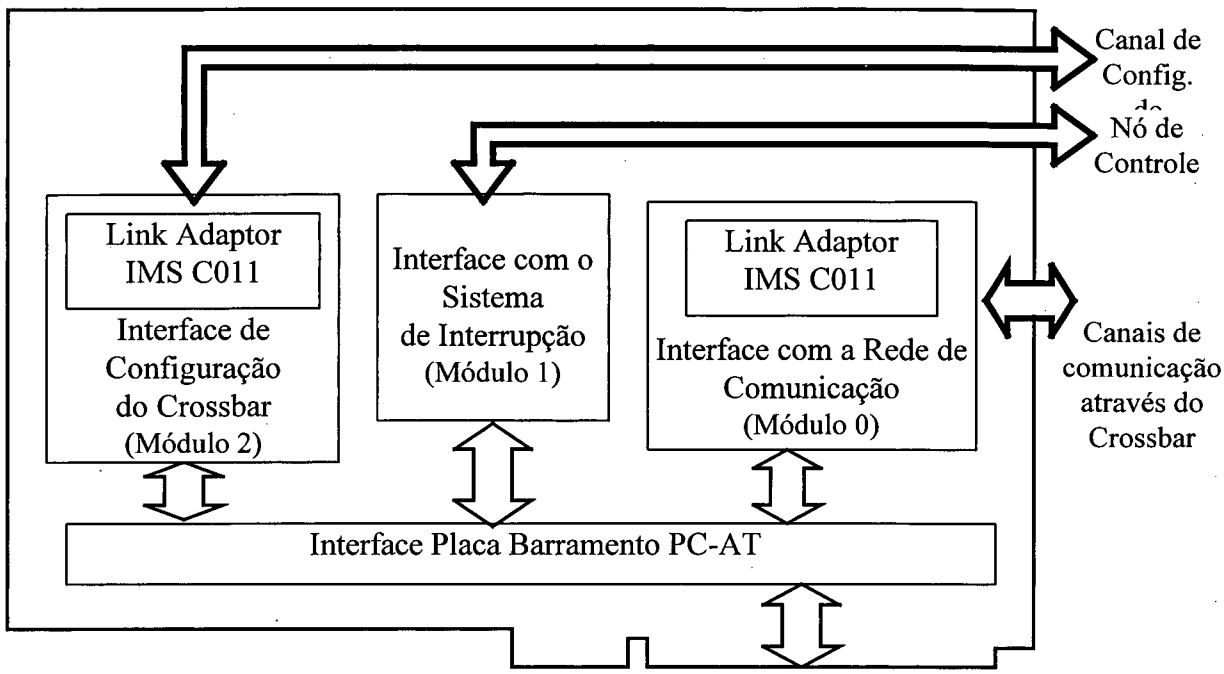


Figura 5.10 : Diagrama de Blocos da Placa de Interface do Nó de Controle

5.2 O Sistema Operacional CRUX

O sistema operacional do multicomputador Nó // é denominado CRUX. O CRUX surgiu a partir da idéia de combinar o modelo de processos que se comunicam exclusivamente por trocas de mensagens com a arquitetura dinâmica do multicomputador Nó //, sendo compatível com o sistema UNIX. Ele é constituído de um micronúcleo distribuído, de biblioteca de funções de acesso às primitivas do sistema e de um servidor.

Não são aplicadas técnicas de gerenciamento de memória, como paginação ou segmentação, pois considera-se a memória como um recurso abundante. Na gerência de processos não há o escalonamento, pois existe apenas um processo por nó e este cabe em sua memória privativa. O sistema operacional CRUX, descrito em [MON95] e [ROD95], foi implementado para executar na arquitetura baseada em barramento comum e apoia-se sobre as idéias descritas nas próximas seções. Seguindo a tendência dos sistemas operacionais modernos (*Amoeba*, *Mach*, *Chorus*), o CRUX é formado por duas peças principais : um micronúcleo e um conjunto de processos servidores (Figura 5.11).

A Comunicação entre os processos no Nó // será efetuada através de canais bipontuais, com endereçamento direto, de maneira síncrona [COR93]. A partir desse modelo simples, pretende-se construir mecanismos mais aprimorados, como caixas postais ou comunicação assíncrona.

Os serviços de sistema no CRUX são implementados através de dois processos servidores: o servidor de processos e o servidor de arquivos. Os processos de aplicação acessam estes serviços através de uma camada de *stubs*, que provê uma interface de programação compatível com a do sistema UNIX.

5.2.1 Processos

Os processos CRUX são programas que estão em execução em cada um dos nós de trabalho. Um processo (Figura 5.12) colocado na memória de um nó é composto pelo [MON95] seu espaço de endereçamento, dados, pilha, valores dos registradores e outras informações necessárias para sua execução. A área de código consiste em uma seqüência de

padrões de bytes que o processador interpreta como instruções de máquina. Área de dados corresponde à seção de dados inicializados ou não no arquivo executável. Área de pilha é criada no início e pode ser alterada dinamicamente durante a execução do programa. Os processos CRUX criados e enviados para um nó disponível que o recebe e o "instala". O código responsável por receber e instalar o processo e seu descritor na memória é o carregador de processos. O sistema operacional possui uma estrutura na memória de cada nó para armazenar o contexto do processo que é denominado descritor de processos local. Esta estrutura armazena localmente diversas informações que servem para descrever o processo para o sistema operacional.

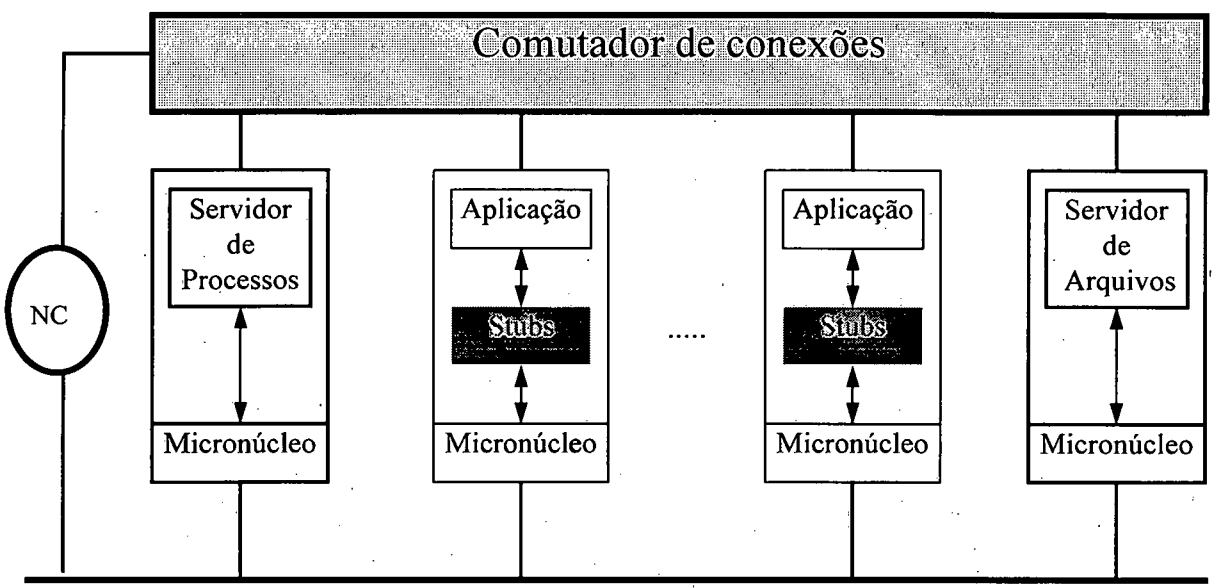


Figura 5.11 : A Arquitetura do CRUX

5.2.2 - As Camadas do Sistema CRUX

A visão completa do sistema CRUX é apresentada na Figura 5.13. Ele está estruturado de forma que cada camada utiliza serviços da camada imediatamente inferior para oferecer serviços a camada imediatamente superior.

Figura 5.12 : Um processo CRUX

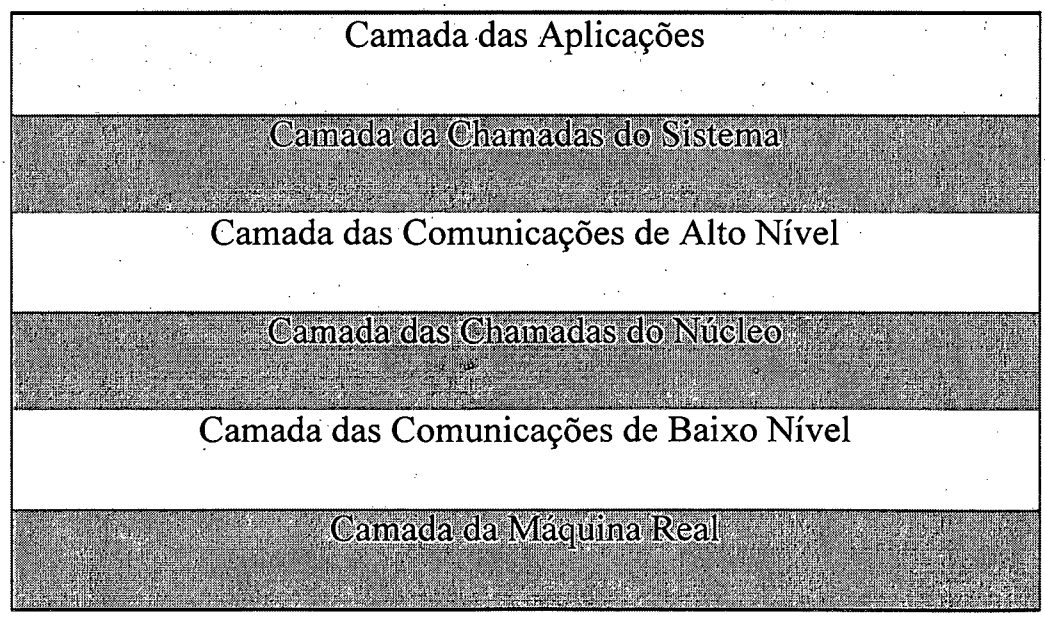


Figura 5.13 : As Camadas do Sistema CRUX

A camada de Chamada do Sistema fornece serviços equivalentes aos serviços fornecidos pelo sistema UNIX, acrescidos de serviços de comunicação do modelo das redes de processos comunicantes. A camada das comunicações de Alto Nível oferece serviços para o transporte de mensagens volumosas entre os nós de trabalho através da rede de comunicação. A camada das Chamadas de Núcleo fornece os serviços para o estabelecimento e o cancelamento de conexões no crossbar, além de alocação e da liberação de nós de trabalho. A camada das comunicações de Baixo Nível fornece serviços para a troca de mensagens de serviços entre os nós de trabalho e o nó de controle. A camada das Aplicações refere-se a aplicações desenvolvidas sobre o sistema e a da Máquina Real refere-se aos recursos materiais do multicomputador Nó //.

5.2.3 Micronúcleo CRUX

O sistema operacional do Nó // é fundamentado sobre um micronúcleo que provê serviços para comunicação entre processadores e alocação de processadores. As primitivas de comunicação entre processadores são síncronas, com endereçamento direto, permitindo envio e recepção de mensagens de tamanhos variáveis. Cada nó processador da máquina terá um micronúcleo em uma memória ROM. O micronúcleo do CRUX foi estruturado em um esquema de três camadas, onde as camadas inferiores fornecem serviços às superiores.(Figura 5.14).

O Micronúcleo está dividido em três camadas básicas; gerência do barramento de serviço, gerência de nós e conexões e gerência e comunicações de processos. Estas camadas são apresentadas a seguir:

◊ Camada de Gerência do Barramento de Serviço

O objetivo desta camada é prover serviços de comunicações, que permitem trocas de mensagens compactas, para as chamadas de procedimento remoto através do barramento de serviço.

W_SendReceive (request,reply)	Nó de trabalho envia mensagem ao nó de controle e aguarda resposta.
C_ReceiveAny (node,request)	Nó de controle aguarda uma mensagem.
C_Send (node,reply)	Nó de controle envia ao nó de trabalho *node* uma mensagem.

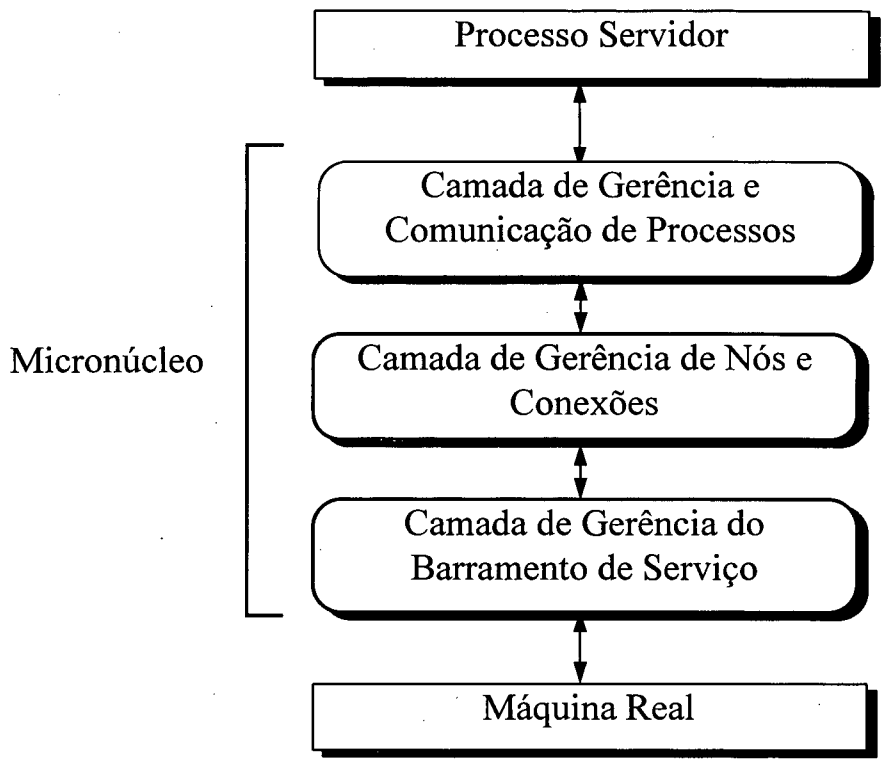


Figura 5.14 : Micronúcleo do CRUX

◊ **Camada de Gerência de Nós e Conexões**

Esta camada oferece serviços de conexão e alocação de nós, utilizando as funções da camada inferior. As funções desta camada são :

Allocate(nid) :	Aloca o nó com endereço nid.
AllocateAny(nid) :	Aloca um nó qualquer.

Deallocate()	Desaloca o nó solicitante.
Connect(nid)	Pede conexão com o nó nid.
ConnectAny()	Pede conexão com um nó qualquer.
Disconnect()	Pede desconexão do nó conectado.

◊ **Camada de Gerência e Comunicação de Processos**

Esta camada oferece serviços aos processos externos ao micronúcleo. Na seção de gerência de processos existem três funções para criar e remover processos. As interações entre os processos existentes no sistema seguirão na grande maioria das vezes a disciplina de comunicação cliente-servidor.

CreateProcess(nid)	Cria um processo no nó nid.
CreateAnyProcess()	Cria um processo num nó qualquer.
RemoveProcess()	Remove o processo do nó corrente.
Send(nid)	Envia uma mensagem para o nó nid.
Receive(nid)	Aguarda uma mensagem do nó nid.
ReceiveAny()	Aguarda uma mensagem de um nó qualquer.

5.2.4 Biblioteca CRUX

Esta biblioteca possui funções que podem ser ligadas com os programas de usuário. Através desta biblioteca os processos acessam o conjunto das chamadas de sistema CRUX. A maioria das chamadas são atendidas pelo servidor CRUX. Essas chamadas envolvem uma comunicação cliente-servidor utilizando serviços oferecidos pela camada de comunicação entre processos existentes no micronúcleo.

5.2.5 Servidor CRUX

O servidor CRUX é o processo do sistema, responsável por gerenciar as funções de processos UNIX e funções de sistema de arquivos UNIX. As funções de processos UNIX consistem em requisições relacionadas a gerenciamento de processos (*fork*, *exit*, *wait* e *exec*), para os quais o servidor CRUX mantém uma tabela de processos, onde são armazenadas as informações dos processos existentes no sistema. As funções de sistema de arquivos (*open, close, lseek, read e write*) também são tratadas pelo servidor CRUX. Cada processo pode manter vários arquivos abertos, cujos descritores são armazenados na tabela de processos mantida pelo servidor CRUX.

CAPÍTULO 6

A Implementação das Primitivas Básicas de Comunicação

A camada de comunicações de baixo nível do Nó // é responsável por prover comunicações confiáveis e eficientes entre pares de nós da máquina, conectados fisicamente através de canais de comunicação [FRO96]. As primitivas que compõe a camada das comunicações de baixo nível do sistema operacional CRUX foram implementadas para suportar um modelo de arquitetura baseada em um barramento comum. As primitivas implementadas neste trabalho foram desenvolvidas para o modelo de arquitetura baseada em um sistema de interrupções.

A camada de comunicações de baixo nível implementa primitivas de serviço para comunicação entre os nós e para configuração do *crossbar* pelo nó de controle (NC), sendo responsável pela gerência de conexões físicas entre os nós da máquina. Estas primitivas estão localizadas no *micro-kernel* de cada nó do multicomputador. As comunicações entre os nós da máquina são realizadas por meio de conexões estabelecidas no *crossbar* segundo o protocolo descrito a seguir.

Um nó de trabalho (NT), quando deseja o atendimento de um serviço de conexão, deve interromper o NC através da linha apropriada. Este, por sua vez, executa uma rotina de tratamento da interrupção, identificando o nó requisitante e estabelecendo uma conexão via *crossbar* com o NT. Então, o NC envia uma mensagem de consulta ao NT, que retorna com a sua requisição de serviço. A conexão física é realizada pelo NC, que configura o *crossbar* e interrompe os NTs conectados, habilitando-os a iniciar a comunicação. A Figura 6.1 resume o protocolo descrito acima.

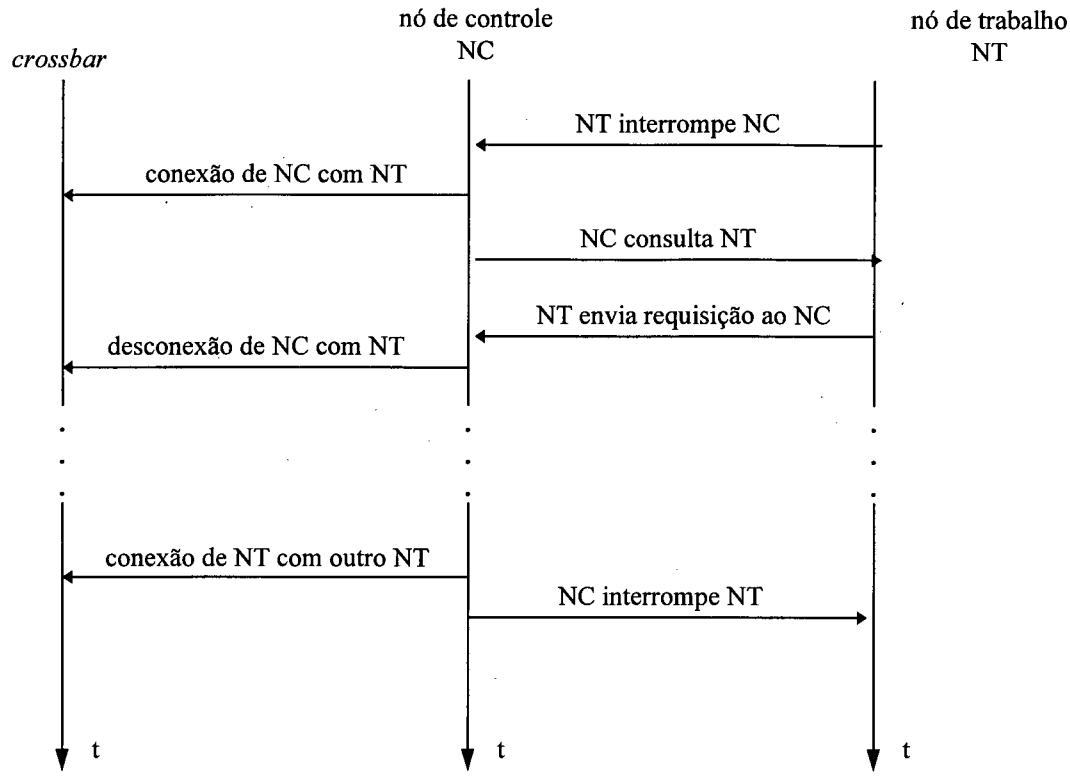


Figura 6.1 - Protocolo da Camada de Comunicações Baixo Nível

6.1 As Primitivas

O conjunto de primitivas implementadas na camada de comunicações de baixo nível permite a comunicação entre os nós da máquina, suportando o uso de diferentes tipos de protocolos de comunicação. Este conjunto é apresentado através da descrição funcional das primitivas, conforme segue:

A rotina de tratamento de interrupções *C_ReceiveIntAny*

INTRtc	*INTRct*
CB_Conecta (nó in, nó out)	*CB_Desconecta (nó in, nó out)*
*CB_Send (*msg, msLength)*	*CB_Recv (*msg, mslength)*

⇒ A Rotina de Tratamento de Interrupções de Comunicação

Esta rotina, e também as primitivas *INTRct*, *CB_Conecta* e *CB_Desconecta*, só estão disponíveis no NC. A função desta rotina é atender as interrupções geradas pelos NTs. A rotina de tratamento das interrupções de comunicação fica instalada no NC e assim que ocorre uma interrupção, o processador lê o vetor de interrupções onde ela está localizada e dá inicio ao tratamento da interrupção de comunicação.

Quando um NT envia uma interrupção significa que ele deseja fazer um pedido de conexão. Assim que é acionada a rotina de tratamento de interrupções de comunicação, *C_ReceiveIntAny,* mascara as interrupções e realiza uma leitura no módulo 1 da placa de comunicação para identificar o nó que gerou a interrupção. Após identificar o NT a rotina consulta a tabela de conexões. Se o estado do NT é conectado, configura o *crossbar* para executar a desconexão do NT. Senão envia uma mensagem de configuração do *crossbar*, para conectar o NC ao NT que gerou a interrupção, envia uma mensagem de consulta utilizando a primitiva CB_Send e aguarda a chegada da requisição do NT. Quando *C_ReceiveIntAny* recebe a mensagem, analisa o pedido e consulta a tabela de conexões para verificar se o pedido pode ser atendido. Se não puder, o pedido é colocado em uma fila de espera, até que a conexão possa ser feita. Se puder ser atendido, a rotina envia uma mensagem de configuração do *crossbar* para fornecer a conexão desejada. Após a confirmação da configuração realizada, envia uma interrupção de comunicação do tipo *INTRct*, para ambos os NTs (o que realizará o envio e o que realizará o recebimento) informando ao que o pedido foi atendido, e desmascara as interrupções atendendo a próxima interrupção, se houver.

Quando um NT envia uma *INTRtc,* fica aguardando um ciclo de escrita no módulo 1 da placa de comunicação para enviar a sua requisição. Após o envio da requisição fica aguardando a chegada de uma *INTRct*. No término da utilização da conexão o nó trabalho envia novamente uma *INTRtc*. A rotina *C_ReceiveIntAny* ao atender a interrupção consulta a tabela de conexões, constata que o NT está conectado, então realiza a desconexão e atualiza a tabela de conexões.

O NC gerencia a tabela de conexões, esta tabela contém o estado atual de cada um dos NTs. Quando a rotina de tratamento de interrupções recebe uma *INTRtc*, primeiro identifica o nó que gerou a interrupção, depois consulta a tabela de conexões para saber em que estado se

encontra o NT. O NT pode estar no estado conectado, desconectado ou não alocado. No estado conectado, o nó está realizando a emissão ou recepção de mensagens. No estado desconectado, o nó pode estar aguardando a confirmação do atendimento a um pedido de serviço, ou pode estar processando. No estado não alocado ele se encontra sem nenhum processo alocado.

O *hardware* e as primitivas fornecem flexibilidade para a implementação de diferentes políticas para a comunicação. Uma política é por exemplo, estabelecer uma conexão apenas quando os dois NTs querem se conectar, por exemplo, quando um **nó y** envia um pedido de conexão com um **nó x** e este envia um pedido de conexão com o **nó y.** Neste caso, a conexão só é realizada quando chegar os dois pedidos. Um outro exemplo de política, estabelece a conexão independente da existência dos dois pedidos de conexão, um **nó y** envia uma pedido de conexão com um **nó x**, o NC então verifica o estado do **nó x** e se estiver livre, envia uma interrupção ao **nó x** para avisá-lo que o **nó y** deseja se conectar com ele. Esta política cobre situações em que os nós podem ter prioridade para estabelecer uma conexão com outros nós. Por exemplo, se o **nó y** possui prioridade e deseja uma conexão com o **nó x**, o NC envia uma interrupção ao **nó x,** independentemente do seu estado, para que ele se prepare para a conexão. Se o nó x se encontra em um estado de espera de uma confirmação do atendimento a um pedido solicitado (*INTRct*), o NT se preparará para receber ou enviar uma mensagem através da rede de comunicação. Como não foi realizada a conexão, pois a interrupção não ocorreu com o objetivo de confirmar uma conexão, o nó não conseguirá enviar ou receber e então perceberá que o NC deseja falar com ele. Portanto, podem ser implementadas políticas que se adptem melhor ao comportamento do sistema desejado.

⟹ A Interrupção de comunicação *INTRtc* :

Esta interrupção é utilizada quando o NT deseja fazer um pedido de conexão ao NC. Esta primitiva gera um sinal de interrupção no NC através da escrita de um *byte* no endereço do módulo 1 da placa de interface de comunicação localizada no NT. O módulo 1 é responsável pela interface com o sistema de interrupção contendo a lógica necessária para o envio de um pulso baixo através de uma linha física conectada ao sistema de recepção de interrupções (módulo 1) na placa de interface de comunicação do NC. Este sistema de

recepção de interrupções fornece uma lógica para geração de uma interrupção, enviando o sinal para uma entrada IRQ onde é então percebida pelo processador. Assim que reconhece a interrupção, o processador lê o vetor de interrupções e dá inicio ao tratamento da interrupção. Após o envio de uma interrupção o módulo 1 do NT fica aguardando o recebimento de uma interrupção do NC (*INTRct*) que caracteriza que o NT está conectado ao NT requisitante e que já pode enviar a sua mensagem de pedido de serviço, através do módulo, responsável pela interface com o sistema de comunicação via *crossbar*. O conteúdo do *byte* a ser escrito no endereço do módulo 1 é irrelevante, pois a lógica do *hardware* do módulo 1 precisa apenas de um ciclo de escrita para gerar o sinal de interrupção.

Essa primitiva é acionada pela camada do núcleo do sistema operacional.

⇒ A Interrupção de Comunicação *INTRct* :

Esta primitiva é utilizada quando o NC deseja enviar uma interrupção a um ou mais NTs. O NC pode enviar esta interrupção para confirmar o atendimento a um pedido ou para solicitar que o nó pare o que está fazendo e se prepare para receber uma mensagem do mesmo. Após gerar uma *INTRtc* o NT não pode gerar outra até a recepção de uma *INTRct*. A primitiva *INTRct* gera uma interrupção de forma análoga a *INTRtc*, escrevendo o endereço do NT no módulo 1 da placa de interface de comunicação do NC.

O primeiro protótipo do Nó// possui oito NTs e a placa de interface de comunicação do NC possui uma conexão com cada linha de interrupção que vem dos NTs. Para enviar uma interrupção a um NT, o NC escreve um *byte* no endereço do módulo 1. Este *byte* possui o valor 1 no *bit* da posição onde está ligada a linha de interrupção do nó de trabalho no barramento de dados (D0 a D7) e o *bit* 0 nos demais *bit*s. Como ilustra a Figura 6.2.

Tabela 2 : Linhas de interrupção conectadas no barramento de dados

NT1	NT2	NT3	NT4	NT5	NT6	NT7	NT8
D0	D1	D2	D3	D4	D5	D6	D7

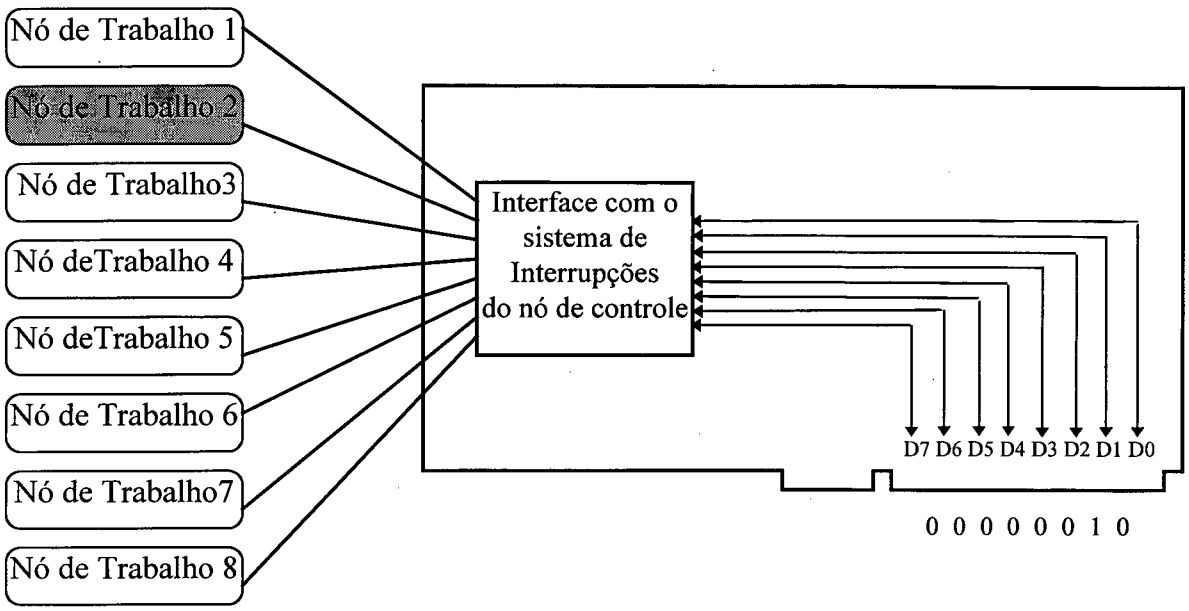


Figura 6.2 : Envio e uma *INTRct* para o NT 2.

Quando deseja enviar uma interrupção a mais de um NT ao mesmo tempo, o NC escreve o *bit* 1 nas posições do barramento de dados correspondentes aos nós que deseja interromper. A lógica do módulo 1 interpreta esse *bit* 1 e envia um sinal para a lógica do módulo 1 da placa do NT, que caracteriza a interrupção.

⇒A primitiva *CB_Conecta* (*nó out, nó in*)

A primitiva *CB_Conecta*, que somente está disponível no NC, é acionada pela rotina de tratamento de interrupções *C_ReceiveIntAny* quando necessita de uma conexão entre o NC e um dos NTs ou uma conexão entre dois NTs. Esta primitiva recebe dois parâmetros: o nó que deseja ser conectado e o nó a que deseja ser conectado. Na placa do NC existe a interface de configuração do *crossbar* (módulo 2) que utiliza o protocolo de comunicação serial Transputer-Link (Figura 5.6) para transferir os *bytes* necessários para a configuração do *crossbar*. Quando a primitiva *CB_Conecta* é acionada, ela monta a mensagem de

configuração e escreve os *bytes* no endereço do módulo 2. O módulo 2 possui um Link Adaptor que recebe dados paralelos e os envia serialmente (como descrito no capítulo 5).

A tabela 1 do capítulo 5 descreve as mensagens de configuração do *crossbar* detalhadamente. O primeiro *byte* contém o modo de operação o Nó // utilizamos o modos de operação 1 que conecta link1 a link2 ligando a entrada de link1 à saída de link2 e a entrada de link2 à saída de link1, e modo de operação 6 (utilizado na primitiva *CB_Desconecta*) que desconecta a saída de *link1* e a saída de *link2*. O segundo e terceiro *byte* devem conter os links a serem conectados, que devem estar na faixa de 0 à 31, que é a capacidade máxima do *crossbar* IMS C004. O quarto *byte* é enviado ao final da configuração para confirmar que a conexão foi realizada pelo *crossbar*.

⇒ A primitiva *CB_Desconecta* (*nó out, nó in*)

Esta primitiva funciona de forma análoga a primitiva *CB_Conecta*. A única diferença está no primeiro *byte* da mensagem de configuração que corresponde ao modo de operação. Aqui, é enviado um *byte* com o valor 6 para caracterizar uma desconexão.

⇒ A Primitiva *CB_Send* (msg, mslength*)**

Esta primitiva é responsável pelo envio de mensagens através da rede de comunicação utilizando o *crossbar*. A placa de comunicação tanto do NC quanto dos NTs possuem uma interface com a rede de comunicação que é denominado módulo 0. A transferência das mensagens é feita através das conexões configuradas no *crossbar* e de *Link Adaptors* para enviar e receber os *bytes,* utilizando o protocolo de comunicação serial *Transputer-Link* (descrito anteriormente). Esta primitiva recebe a mensagem contida em **msg* e o tamanho *length* da mensagem e realiza a escrita de um *byte* no endereço do módulo 0 do primeiro *byte* da mensagem e aguarda o permissão para escrever o próximo *byte*, até que a messagem termine. Quando um nó executa um *CB_Send* o outro nó que está conectado a ele deve executar um *CB_Recv*, que realiza a leitura de um *byte* no módulo 0, escrito pelo nó que executou *CB_Send.*

⇒ **A primitiva *CB_Recv* (**msg, mslength*)**

Esta primitiva funciona de maneira semelhante a primitiva *CB_Send*. A única diferença é que realiza uma leitura no endereço do módulo 0 ao invés de uma escrita. A primitiva recebe a mensagem **msg* e o seu tamanho *length* realizando assim a leitura dos *bytes* escritos pelo outro nó conectado a ele.

⇒ **Exemplo de protocolo de comunicação**

O protocolo de comunicação apresentado a seguir utiliza as primitivas da camada de baixo nível, adotando uma política de atendimento da conexão somente quando os dois concordam na conexão.

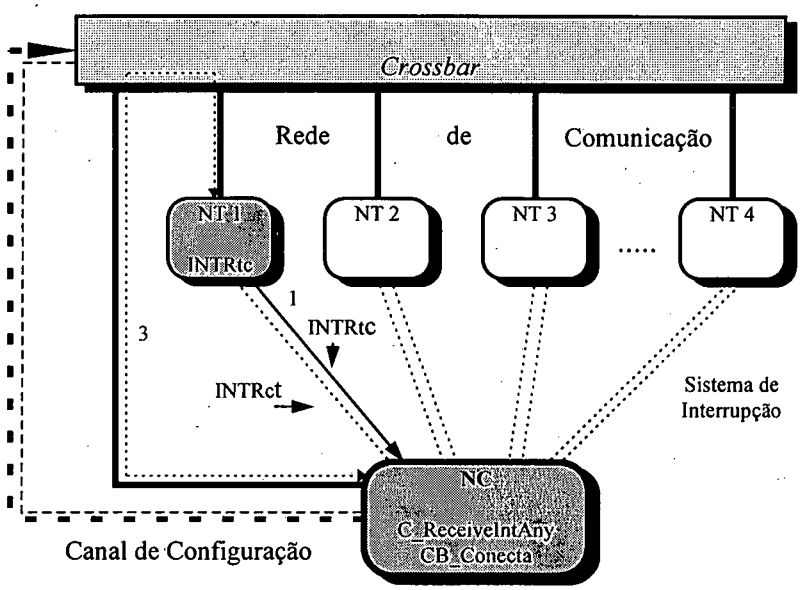


Figura 6.3 : NT envia pedido de serviço

- 1. O NT1 envia uma interrupção *INTRtc* para o NC.
- 2. A rotina de tratamento *C_ReceiveIntAny* detecta a interrupção identifica, o NT1 e chama a primitiva *CB_Conecta* para enviar a configuração para o *crossbar* conectando o NC ao NT que gerou a interrupção.

- 3. A conexão é estabelecida.

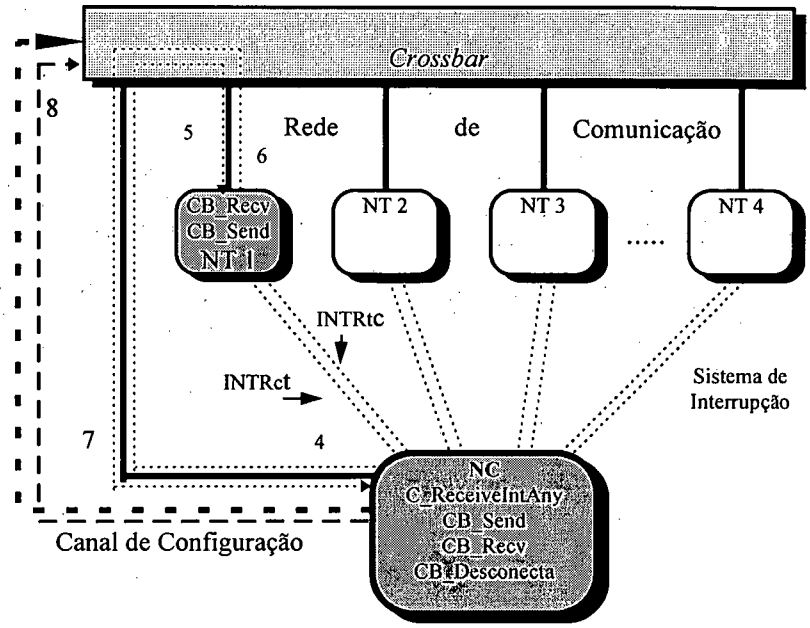


Figura 6.4 : NC se conecta ao NT que envia seu pedido

- 4 . Após estar estabelecida a conexão o NC envia uma mensagem do tipo "o que queres ?" utilizando a primitiva *CB_Send.*

- 5. O NT1 então realiza uma primitiva *CB_Recv* para receber a mensagem do NC.

- 6 . O NT1 aciona a primitiva *CB_Send* para enviar o seu pedido de conexão com o NT3.

- 7. No NC a rotina *C_ReceiveIntAny* recebe a mensagem e chama *CB_Desconecta* para cancelar a conexão com o NT1 e analisa a tabela de conexões para verificar se pode atender o pedido

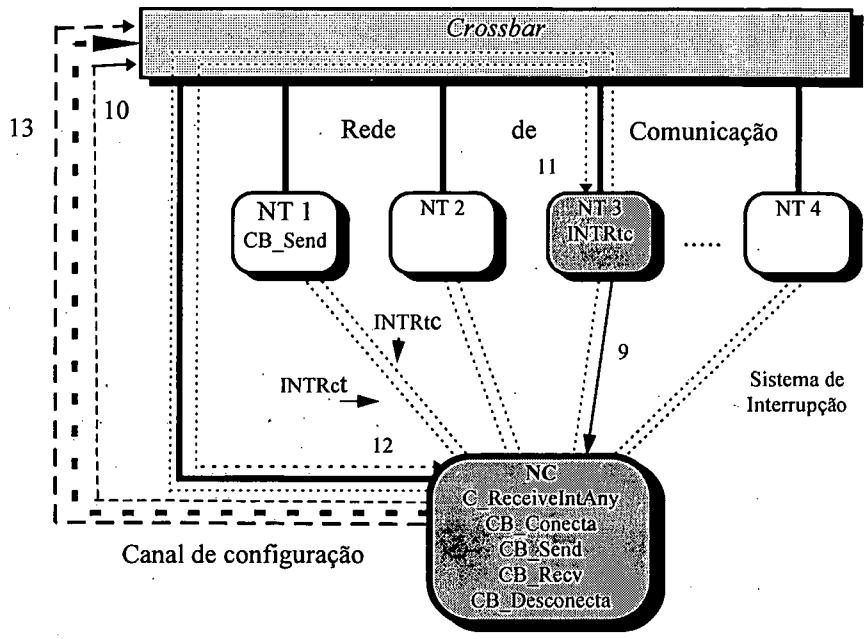


Figura 6.5 : NT3 solicita requisição de serviço

- 9. O NT 3 envia uma *INTRtc* solicitando uma conexão.
- 10. A rotina *C_ReceiveIntAny* identifica NT3 e chama *CB_Conecta* para configurar o *crossbar*.
- 11. Assim que está conectada *C_ReceiveIntAny* envia *CB_Send* com a mensagem "o que queres?" e o NT3 recebe.
- 12. O nó trabalho envia sua mensagem de pedido de conexão com o NT1, ao NC.
- 13. O NC recebe o pedido através da primitiva *CB_Recv* se desconecta do NT3 e analisa o pedido.

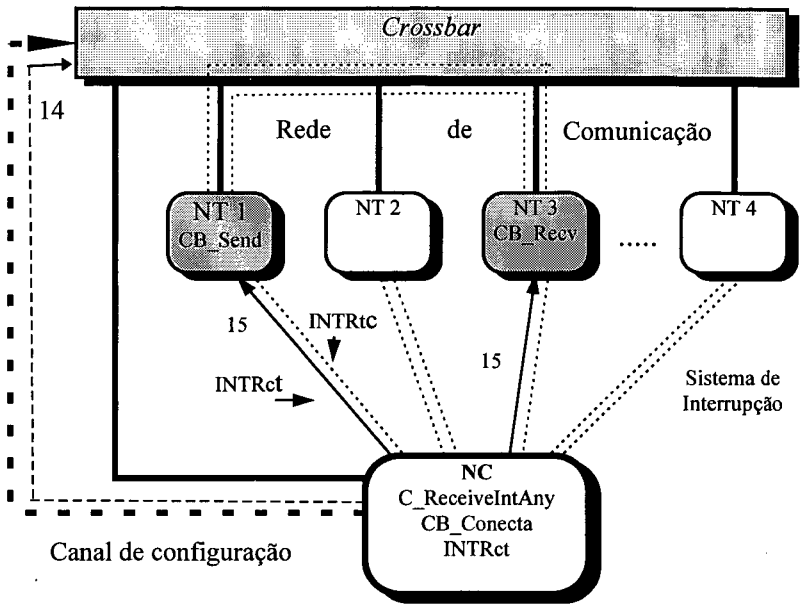


Figura 6.6 : NC realiza a conexão entre NT1 e NT3.

- 14. Depois de consultar a tabela de conexões o NC constata que a conexão pode ser realizada e então chama *CB_Conecta* para configurar a conexão.
- 15. Assim que estão conectados os dois nós recebem uma interrupção *INTRct* ao mesmo tempo para avisá-los que podem iniciar a comunicação.

CAPÍTULO 7

A Avaliação de Desempenho do Nó //

A simulação é uma das mais poderosas ferramentas de análise disponíveis para o projeto e operação de sistemas ou processos complexos. Em um mundo altamente competitivo a simulação torna-se uma ferramenta poderosa para o planejamento, projeto e o controle de sistemas [PEG95].

O objetivo da avaliação é obter informações sobre a capacidade da máquina e medidas como tempo gasto com o protocolo de comunicação, com aumento do número de nós e consequentemente o número de comunicações qual o *overhead* gerado, o custo de transmissão por *byte* e assim por diante. Obtendo essas respostas podemos traçar um perfil para o multicomputador Nó //.

7.1 A Simulação

Nesta parte, são apresentados os aspectos relativos ao modelo de simulação para a máquina baseada em um sistema de interrupções. Estes aspectos são definidos, conforme [JAI91], como aqueles necessários a uma avaliação de desempenho.

- **Definição do sistema :** o objetivo da avaliação é determinar o desempenho do multicomputador Nó // baseado em uma arquitetura com um sistema de interrupções. Obtendo os dados sobre o *overhead* causado na comunicação, o tempo gasto no protocolo e o tempo gasto para transmissão de *byte*s.
- **Medidas :** serão levantadas as seguintes medidas:

  Taxas de ocupação do NC e *crossbar*, tempos gasto para uma comunicação, custo de transmissão de *byte*s, porcentagem de tempo comunicando e processando e o tempo de espera por uma comunicação

- **Parâmetros :** a definição de cada experimento é feita através da divisão da carga de trabalho, do número de NTs utilizados, tamanho das mensagens e quantidade de comunicações.
- **Fatores :** O experimentos realizados baseiam-se nos seguintes fatores:

  # Número de NTs : 2, 4, 6 e 8;

  # Velocidade do *Crossbar* : 10M*bits*/s;

  # Tamanho das mensagens pode ser definido pela variável TAM ou pré definida como segue:

  mensagens curtas: de 24 a 72 *bytes*;

  mensagens de I/O, tamanho médio de 1 *Kbytes*;

  mensagens de carga do programa, de 100 a 3000 K*bytes*

- **Carga de Trabalho :** As carga de trabalho são geradas conforme aplicação desejada, uma parte do modelo foi estruturada para suportar várias mudanças na quantidade de processamento e comunicação

- **Questões :** busca-se obter as medidas do desempenho do nó //, aplicando os tempos fornecidos pelos fabricantes dos componentes de hardware, nas atividades por eles desenvolvidas. Também conhecer o custo do protocolo de comunicações na arquitetura baseada em um sistema de interrupções e o overhead gerado pela comunicação na disputa pelo NC e pelo nó com quem deseja se comunicar.

O software de simulação utilizado é o Arena versão 1.25 da empresa *Systems Modeling Corporation*, número de série 9410807. O Arena contém vários recursos para simulação e inclui facilidades como um ambiente de programação gráfica, recursos de animação, possibilidade de uso da linguagem de simulação SIMAN V (da mesma empresa) e relatórios de resultados.

7.2 O Modelo do Nó //

A modelagem do *hardware* seguiu a descrição da arquitetura de máquina baseada em um sistema de interrupções. Tal modelagem é formada pelos seguintes componentes [FIL96]:

- **Nó de Controle:**

  ⇒ Todo pedido de conexão com o *crossbar* é processado pelo Nó de Controle;

  ⇒ O tempo para o atendimento desses pedidos foi definido pelos tempos fornecidos pelo manual do processador e representa o tempo de reconhecimento de uma interrupção.

  ⇒ O nó de controle é definido como um RESOURCE que possui capacidade unitária, isto é, atendimento de um pedido ou requisição (entidade) por vez.

- **Nós de Trabalho:**

  ⇒ São responsáveis pela geração de interrupções e pelo envio de requisições de serviço.

  ⇒ As mensagens que trafegam durante as comunicações podem ser de vários tamanhos, a serem informados na variável TAM, ou podem ser de três tipos pré definidos:

  Mensagens curtas, com tamanho de 24 a 72 *bytes*;

  Mensagens de I/O, com tamanho médio de 1 *KByte*;

  Mensagens de carga de programa, com tamanho de 50 a 100 *KBytes*.

- ***Crossbar:***

  ⇒ Pode operar na velocidade de 10 ou 20 M*bits*/s, neste trabalho consideramos apenas a velocidade utilizada no protótipo 10 M*bits*/s.

  ⇒ Possui capacidade de 16 conexões simultaneamente entre pares de nós.

7.3 O Modelo Seqüencial

Foi desenvolvido um modelo para um computador IBM PC AT 486 com uma freqüência de 66 Mhz, executando uma aplicação seqüencial. Esta aplicação realiza um processamento com inúmeras variáveis e repete esse processamento muitas vezes. Os tempos utilizados no modelo foram obtidos com a medida de tempo de execução da aplicação em uma máquina real.

7.4 Os Experimentos

Foram implementados três modelos: um modelo do Nó // com a arquitetura baseada em um sistema de interrupções, um modelo de execução de um algoritmo seqüencial e um modelo de execução de um algoritmo paralelo no Nó // para a realização dos experimentos.

Primeiro foram realizados seis experimentos utilizando o modelo do nó // com a arquitetura baseada em um sistema de interrupções, que adota uma política de somente realizar uma conexão se os dois Nts estiverem de acordo, variando-se apenas o tamanho das mensagens transferidas pelos NTs. O objetivo destes experimentos é encontrar o tempo gasto com o protocolo de comunicação e tempo gasto para a transferência de um *byte* na arquitetura baseada em um sistema de interrupções

Segundo, foi realizado um experimento com modelo seqüencial, de uma aplicação para o cálculo de 1.000 variáveis. Este valor será comparado com os tempos obtidos com a execução do modelo paralelo com 2 a 16 NTs.

Terceiro, a aplicação foi modelada paralelamente, utilizando um nó mestre que distribui o trabalho entre os NTs. O nó designado como mestre instala o programa e distribui as variáveis nos Nts. Durante o processamento os NTs realizam duas comunicações a cada iteração e ao final retorna o resultado ao mestre. O tempo de processamento do algoritmo seqüencial foi dividido entre o número de nós ativos na máquina. A cada comunicação feita pelos Nts eles trocam 500 *bytes* de mensagem. No primeiro experimento desta série foi realizado utilizando um nó como o mestre e dois nós escravos. O tempo do processamento sequencial foi dividido por dois e distribuído entre os dois nós. A aplicação começa com o nó mestre dividindo o processamento carregando o programa em cada um dos nós e enviando

500 variáveis para cada um calcular. O tamanho da mensagem para carregar o programa foi considerado 1000 K*bytes* e os dados 2000 *Mbytes*. Durante a execução os Nts precisam trocar informações, o escravo NT 1 deve enviar os valores que encontrou para o escravo NT 2 e vice e versa.. Foram considerados dois casos; o melhor caso, quando os pedidos de conexão não acontecem ao mesmo tempo, assim não espera para o acesso ao nó de controle e nem espera para a liberação de um nó para a conexão e todos os nós gastam o mesmo tempo na comunicação . E o pior caso, quando os pedidos de conexão ocorrem ao mesmo tempo. Foram realizados dezesseis experimentos com os números pares de nós nos melhores e piores casos.

Quarto, foram realizados quatro experimentos semelhantes aos descritos acima mudando apenas o número de comunicações entre os nós de trabalho de duas para quatro a cada processamento e o número de NTs ativos na máquina que chegou ao máximo de oito.

7.5 Conclusões

As considerações feitas a seguir apresentam medidas de desempenho no multicomputador Nó //. O tempo para a transferência de mensagens pela rede de comunicação, o tempo necessário para a execução do protocolo de comunicação na arquitetura baseada em um sistema de interrupções, o aumento na performance da execução de uma aplicação sendo executada paralelamente e o tempo gasto com as comunicações que são realizadas durante a execução.

7.5.1 Descrição dos Resultados

Os números resultantes dos seis primeiros experimentos realizados são descritos a seguir: Encontrou-se um tempo de 3.45 μs para a transferência de cada byte pela rede de comunicação, onde são gastos 0.24 μs para escrever o byte no barramento, 3.0 μs para transferir este byte pelo *crossbar*, utilizando o protocolo de comunicação serial transputer link, até outro NT e 0.21 μs para realizar a leitura de um byte, quando o mesmo chega do outro lado. E um tempo de 88,7 μs na utilização do protocolo de comunicação da arquitetura baseada em um sistema de interrupções, onde;

41.40 μs são gastos com o envio de mensagens de comando entre o NC e os NTs para a realização de uma conexão. As mensagens de comando podem ser de 2 *bytes* quando são enviadas do NC para os NTs e de 4 *bytes* quando são enviadas dos NTs para o NC. No

protocolo a cada pedido são trocadas uma mensagem de 2 *bytes* e um de 4 *bytes*. Como para acontecer a conexão é necessário que os dois nós peçam, deve ocorrer um segundo envio de mensagem de pedido por parte do outro NT, então devem ocorrer a troca de 12 *bytes* de comando (duas mensagens de 2 *bytes* e duas de 4 *bytes*). Com o custo de tempo da transferência de 3.45 μs por *byte*, chegamos a um tempo de 41,40 μs.

1,61 μs são gastos no envio das interrupções INTRtc e INTRct. No protocolo, mencionado no item anterior, devem ser geradas sete interrupções. Duas INTRtc de cada nó conectado, um para pedir conexão e outro para pedir desconexão. E três INTRct, um é enviado aos dois NTs ao mesmo tempo para avisar que a conexão está feita e as outras duas, uma para cada NT, para avisar que a desconexão foi realizada. O tempo para a geração de uma interrupção é de 0.23 μs vezes sete 1.61 μs.

1.60 μs são gastos na recepção das interrupções INTRtc enviadas pelos Nts, para o envio de um pedido de conexão. No protocolo ocorrem duas INTRtc de cada NT, um para pedir a conexão e outro para pedir a desconexão. O custo de tempo para a geração de cada interrupção é de 0.40 μs, multiplicados por quatro é igual a 1.60 μs.

2.0 μs para a consulta ,manipulação e atualização da tabela de configuração.

1.0 μs na realização da desconexão dos Nts.

1.0 μs na chamada Send da camada de comunicações de alto nível.

1.0 μ na chamada Receive da camada de comunicações de alto nível.

0.1 μs em uma chamada da camada das chamadas dos sistema.

O resultado do experimento utilizando a modelagem de uma aplicação seqüencial, chegamos a um total de tempo de execução de 550.000 μs.

Os resultados obtidos com os experimentos utilizando o modelo do Nó // são mostrados na tabela 3. Analizando os resultados podemos fazer as seguintes conclusões:

Como podemos observar, com o aumento do nós de trabalho ocorre o aumento no tempo gasto nas comunicações, mas ocorre também a queda no tempo do processamento.

Número de nós	casos	Tempo total (μs)	Tempo de processa-mento (μs)	TP/TT	Tempo de Comunicação (μs)	TC/TT	Total de bytes transferidos	Total de bytes de controle
2	melhor caso	308.061	275.000	89%	33.061	11%	12.000	60
2	pior caso	316.645	275.000	87%	41.645	13%	12.000	60
4	melhor caso	177.568	137.000	77%	40.568	23%	17.000	144
4	pior caso	188.070	137.000	73%	51.070	27%	17.000	144
6	melhor caso	141.256	91.850	65%	49.406	35%	22.016	228
6	pior caso	156.353	91.850	59%	64.503	41%	22.016	228
8	melhor caso	121.052	68.750	57%	52.302	43%	23.500	270
8	pior caso	135.092	68.750	51%	66.342	49%	23.500	270
10	melhor caso	123.023	55.000	45%	68.023	55%	31.500	378
10	pior caso	152.566	55.000	36%	97.566	64%	31.500	378
12	melhor caso	138.633	45.833	33%	92.800	67%	38.500	462
12	pior caso	173.463	45.833	26%	127.630	74%	38.500	462
14	melhor caso	145.460	39.285	27%	106.175	73%	45.500	546
14	pior caso	178.688	39.285	22%	139.403	78%	45.500	546
16	melhor caso	163.570	34.375	21%	129.195	79%	51.500	642
16	pior caso	196.722	34.375	17%	162.347	83%	51.500	642

Tabela 3: Valores obtidos com a simulação no Nó // com duas comunicações por NT.

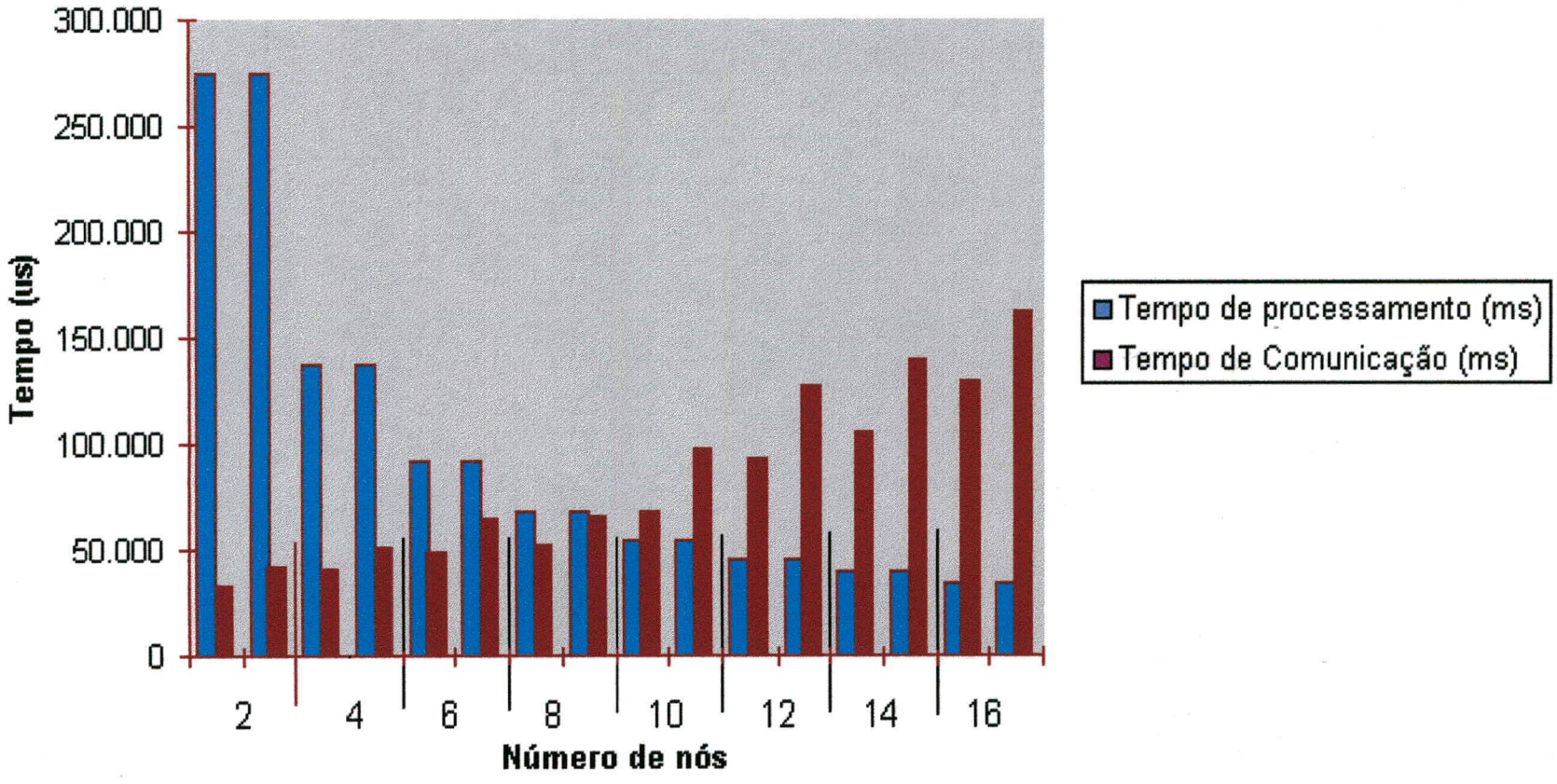


Figura 7.1: Aumento do tempo gasto em comunicação com o acréscimo de NTs

A aplicação modelada sequencialmente, como visto anteriormente, utiliza um tempo total de processamento de 550.000μs. O Nó // conseguiu um bom desempenho na utilização de até oito nós de trabalho, acima de oito nós o tempo de processamento é menor que o tempo gasto para as comunicações no Nó //. Portanto, perde performance.

Na figura 7.1 podemos visualizar melhor o crescimento do tempo gasto nas comunicações com o aumento do número de nós processando no Nó //. A aplicação modelada tem um tempo de processamento em que o Nó // obtém uma boa performance até a utilização de oito NTs. Acima deste número a espera pela realização de uma comunicação deteriora a performance. Se aplicação necessitasse de um tempo de processamento maior, que alcançasse um tempo superior ao mínimo para a realização de uma troca de mensagens (que é equivalente a 88.7 μs) mais o tempo de espera no sistema de comunicação para a realização da comunicação, obteríamos uma performance maior com a utilização de um número maior de Nts. O tempo de espera no sistema também depende do número de NTs ativos no Nó //, pois quanto mais NTs ativos mais comunicação precisam ser realizadas, o que implica em mais trabalho para o NC e como ele atende a um NT de cada vez isto pode acarretar uma espera razoável. Portanto, a escolha do número ideal de NTs para o processamento de uma aplicação, implica diretamente no desempenho da máquina.

Podemos dizer que temos uma aplicação executada seqüencialmente em um tempo de 550.000μs. Esta mesma aplicação é divida em duas e executada paralelamente, teoricamente teríamos a execução em um tempo de 275.000μs, obtendo um aumento ideal de performance de 50 % na execução da aplicação. No Nó // esta aplicação é executada por dois nós paralelamente em um tempo de 308.061 μs, obtendo um aumento da performance de 44.5 % na execução da Aplicação (tabela 4).

Com a utilização de dois nós de trabalho, o tempo de processamento é dividido por dois, portanto há a redução de cinqüenta por cento do tempo de execução do algoritmo. Na utilização de quatro nós de trabalho, o tempo de processamento é dividido por quatro. Assim, o tempo de execução do algoritmo é equivalente a vinte e cinco por cento do tempo de execução serial portanto ocorre o aumento de setenta e cinco por cento na performance de execução do algoritmo. Ocorrendo de forma semelhante com o aumento gradativo de nós de trabalho. Como pode-se observar na tabela 4, a simulação da execução do algoritmo no Nó // obteve um aumento real próximo do aumento ideal da performance até a utilização de oito

NTs. Com a utilização de mais de oito Nts o Nó // perde performance devido ao fato de que com o aumento do número de nós há consequentemente diminuição do tempo de processamento. Para ser realizada uma comunicação é necessário a espera da realização do protocolo de comunicação, portanto se o tempo de que um NT fica processando for menor que o tempo que utiliza para realização uma comunicação, ele vai se encontrar a maior parte do tempo a espera da realização de uma comunicação.

Nos últimos experimentos realizados, observou-se o aumento no tempo total de execução da aplicação com o aumento do número de comunicações por NT. O multicomputador Nó // perde performance com o aumento do número de comunicações por processamento. A tabela 5 mostra os resultados obtidos com a utilização de até oito NTs levando em conta o pior caso que pode ocorrer. Comparando os resultados da tabela 5 com a tabela 4 é observado um aumento gradual no tempo de execução da aplicação em relação ao número de NTs.

Tabela 4 : Aumento na performance de execução da aplicação

Número de Nós de Trabalho	Aumento Ideal da Performance	Aumento Real da Performance no Nó //
2 NTs	50.0 %	44.5 %
4 NTs	75.0 %	68.0 %
6 NTs	83.3 %	74.3 %
8 NTs	87.5 %	78.0 %
10 NTs	90.0 %	77.0 %
12 NTs	91.6 %	74.7 %
14 NTs	92.8 %	73.5 %
16 NTs	93.7 %	70.9 %

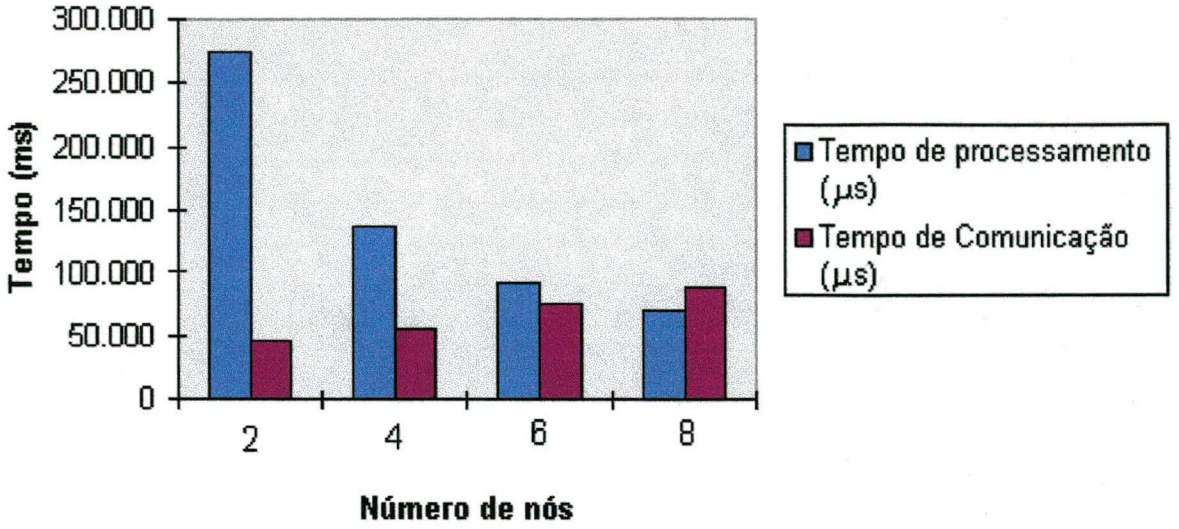


Figura 7.2: Aumento do tempo gasto na execução total com a realização de quatro comunicações por NT.

Número de nós	Tempo total (μs)	Tempo de processa-mento (μs)	TP/TT	Tempo de Comunicação (μs)	TC/TT	Total de bytes transferidos	Total de bytes de controle
2	320.127	275.000	86%	45.127	14%	13.000	84
4	191.621	137.000	71%	54.621	29%	19.000	192
6	166.945	91.850	55%	75.095	45%	27.016	324
8	156.659	68.750	44%	87.909	56%	39.000	558

Legenda: TP/TT = Tempo de Processamento divido por Tempo Total de execução

TC/TT = Tempo de Comunicação divido pelo Tempo Total de execução

Tabela 5: Resultados obtidos realizando quatro comunicações por NT.

A ocupação do nó de controle não é significante. Nos experimentos com quatro comunicações por NT com utilização de 16 Nts o *crossbar* chegou a uma taxa de ocupação de 80%. O *crossbar* pode realizar até dezesseis conexões simultaneamente, como só existem dezesseis NTs, na maioria das vezes não apresentou restrições pois não ultrapassa sua capacidade.

CAPÍTULO 8

Conclusões

O projeto Nó // visa a construção de um ambiente paralelo em um multicomputador com rede interconexão dinâmica, um sistema operacional e uma linguagem de programção paralela. O multicomputador é composto de vários nós, denominados nós de trabalho, que são ligados entre si através de uma rede de comunicação. A rede de comunicação é composta de comutador de conexões capaz de realizar 16 conexões simultâneamente. O comutador de conexões é configurado dinâmicamente por um nó denominado nó de controle. Os nós de trabalho são ligados ao nó de controle através do sistema de interrupções. A qual utilizam para pedirem uma conexão com o nó de controle para enviarem um pedido de conexão com outro nó de trabalho.

Das contribuições do trabalho duas são mais significativas. Uma contribuição é prática implementando as primitivas básicas de comunicacão e outra é teórica avaliando o desempenho do Nó //.

As primitivas implementadas compõe a camada de comunicação de baixo nível do sistema operacional CRUX e o *software* básico para o *hardware* do Nó //. O conjunto das primitivas implementadas juntamente com o *hardware* apresentam as seguintes características: flexibilidade para implementação de políticas no estabelecimento de conexões, suporte a aplicações de tempo real e possibilidade de políticas de endereçamento indireto. A camada de comunicação de baixo nível está fundamentada em uma rotina de tratamento de interrupções que gerencia todas as atividades desta camada. Esta rotina está localizada no nó de controle que é o nó responsável por configurar o crossbar para realizar as conexões físicas.

Na avaliação do desempenho, foi desenvolvido um modelo de simulação para o Nó // onde que estão inseridos o sistema operacional e o *hardware*. O modelo foi desenvolvido na linguagem ARENA/SIMAN V, que possui uma interface gráfica amigável relatórios detalhados e recursos de animação. Utilizando este modelo foram realizados vários

experimentos variando-se as características dinâmicas do Nó //. Utilizando a simulação podemos avaliar as medidas da performance da máquina, como: o custo do seu protocolo de comunicação que utiliza um sistema de interrupções para gerenciar o atendimento de pedidos de conexão. A taxa de transmissão de *bytes* pela rede de comunicação. As medidas encontradas são as primeiras avaliações feitas sobre o Nó // baseado em um sistema de interrupção e irão contribuir para a busca de uma melhor arquitetura para o multicomputador. As primeiras medidas de desempenho também demonstraram que o Nó // possui um bom desempenho na simulação de uma aplicação que necessita de duas comunicações durante um processamento de 550 milisegundos.

Atualmente o projeto se encontra no seguinte estado: já foram desenvolvido um interpretador para linguagem paralela super pascal, o sistema operacional CRUX, implementado em quase toda sua totalidade e o projeto do *hardware*.

8.1 Sugestões para Trabalhos Futuros

O conjunto de primitivas básicas de comunicação implementado pode ser otimizado utilizando a simulação para achar o melhor caminho para o protocolo de comunicações.

O modelo de simulação desenvolvido pode ser melhorado implementando uma modelagem mais genérica das aplicações existentes. Devem ainda, serem feitos experimentos das várias combinações possíveis da configuração da máquina. Por exemplo, experimentos com base no aumento da quantidade de comunicações em cada NT, na variação do tempo de processamento e na utilização de mensagens de tamanhos variados.

Avaliar o desempenho da modelagem de uma arquitetura híbrida utilizando um sistema de barramento comum e um sistema de interrupção ao mesmo tempo.

REFERÊNCIAS BIBLIOGRÁFICAS

[AUS91] Austin, P. & Murray, K. & Wellings, K., *The Design of an Operating System for a Scaable Parallel Computing Engine, Software* - Practice and Experience, Vol 21, p. 989-1013, outubro de 1991.

[BER89] Bertsekas, Dimitri P., Tsitsiklis, J. M., *Parallel and Distributed compution, Numerical Methods.* Prentice - Hall, 1989.

[CAM95] Campos, R. A., *Um Sistema Operacional fundamentado no Modelo Cliente-Servidor e um Simulador Multiprogramado de Multicomputador.* Dissertação de Mestrado, CPGCC-UFSC, Florianópolis, 1995.

[CHA92] Chaudhuri, P., *Parallel Algoritms Design and Analysis*, Prentice-Hall, 1992.

[COR93] Corso, T. B., *Ambiente para Programação Paralela em Multicomputador,* Relatório Técnico CPGCC-UFSC n.1, novembro de 1993.

[COU88] Coulouris, G. F. & Dollimore, J., *Distributed Systems: Concepts and Design,* Addison-Wesley, 1988.

[FEN81] Feng, T., *A Survey of interconnection Networks,* Computer, p. 12-27, dezembro de 1981.

[FIL96] Filho, Paulo J. Freitas, C. Merkle, C. A. .Zeferino, V. A. Silva., *A Simulation Model for the Comparison of Two Multicomputer Arquitetures,* in the 1996 Summer Computer Simulation Conference, Portland - USA, Julho de 1996.

[FLY72] Flynn, M. J., *Some Computer Organizations and Their Effectiveness.* IEEE Trans. on Computers, vol. C-21, p. 948-960, Setembro de 1972.

[FRO96] Fröhlich, A. A., C. A. Zeferio, V. A. Silva., *Process Comunication in Nó//,* International Conference on Information Systems Analysis and Synthesis, Orlando - USA, julho de 1996.

[GUP95] Gupta, A. and Kumar, V., "Performance and Scalability of Preconditioned Conjugate Gradient Methods on Parallel Computers ", IEEE Trans. on Parallel and Distributed Systems, Vol. 6, No. 5, May, 1995.

[IEE92] IEEE Committee Report, "Parallel Processing in Power Systems Computation", IEEE Transactions on Power System, Vol. 7, PP. 629-638, May 1992.

[HAN73] Hansen, B., *The Operating System Principles,* Prentice Hall, 1973

[HWA93] Hwaig, Kai, *Advanced Conpter Arquiteture : Parallelism, Scalability, Programmability. McGraw-Hill, 1993.*

[MON95] Montez, Carlos Barros, *Sistema Operacional com Micronúcleo Distribuído e um Simulador para um Multicomputador com Rede de Interconexão Dinâmica,* Dissertção de Mestrado, CPGCC-UFSC, Florianópolis, 1995.

[MUL88] Mullender, S. J., *Distrubuted Operanting Systems: State-of-the-Art and Future Directions,* Proc. of the EUTECO 88, Viena, p. 57-66, 1988.

[QUI95] Quinn, M.J., *Parallel Computing - Theory and Practice*, McGraw-Hill, 1995.

[REE87] Reed, D.A.,Fujimoto, R.M., *Multicomputer Networks: Mesage-Based Parallel Processing.* MIT Press, 1987.

[SIL94] Siberschatz, A. P. & Galvin, *Operating Systems Concepts*, Addison Wesley, 1994.

[TAY89] Taylor, S., *Parallel Logic Programming Techniques,* Printice-Hall, p 89, 1989

[TAN92] Tanenbaum, A. S., *Operating Systems: Design and Implementation,* Printice-Hall, 1987.

[VAR94] Varhal, P. D., *Small Kernels Hit it Big,* Byte, p. 119-128, janeiro de 1994

[ZEF95] Zeferino, C. A., Lucke, H.A.H., Silva, V.A., *Um multicomputador com Sistema experimental de comunição.* In. VII Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho, Canela-RS, julho de 1995.

[ZIM90] Zima, Hans, *Supercompilers for Parallel and Vector Computers,* Addison-Wesley, 1990.

[ZOR92] Zorpette, G., "The power of Paralelism", IEEE Spectrum, PP. 28-33, September, 1992.

ANEXO I

Pseudocódigo das Primitivas Básicas de Comunicação

Rotina de tratamento de interrupções de comunicação (*C_ReceiveIntAny)*

Inicializar rotina *C_ReceiveIntAny* (detecta a chegada de uma interrupção de communicação)
Mascarar a ocorrência de interrupções
Ler o conteúdo do endereço do módulo 1 (sistema de tratamento de interrupções)
Atribuir o valor a variável noINT

PROCURAR noINT
REPETIR ATÉ no igual noINT
    Ler estado no
    SE conectado
    ENTÃO chama a rotina *CB_Disconecta(no, no_a_que_esta_conectado)*
    SENÃO SE estado for desconectado logicamente
        ENTÃO atualiza estado para desconectado
        SENÃO chama a rotina *CB_Conecta (no de controle, noINT)*
        FIMSE
    FIMSE
FIMREPETIR

Primitiva CB_Conecta (*no_out, no_in*)

Inicializar a rotina (receber os parâmetros)
Atribuir o valor 1 a variável operação
Atribuir o endereço do módulo 2 a variável port
CHAMAR a função MONTA_CONFIG (no_origem, no_destino, operação)
CHAMAR a função ENVIA_MENSAGEM (mensgconfig, length, port)
Terminar rotina

FUNÇÃO MONTA_CONFIG (*no_origem, no_destino, operação*)

(Inicializar a função para montar mensagem de cofiguração do *crossbar)*
Declarar a estrutura mensgconfig: tipoperação, no_out, no_in
Receber os parâmetros no_origem, no_destino, operação
Atribuir a variável mensgconfig.tipoperação o valor de operação
Atribuir a variável mensgconfig.no_out o valor de no_origem
Atribuir a variavel mensgconfig.no_in o valor de no_destino
Terminar função

```
FUNÇÃO ENVIA_MENSAGEM (*msg, length, port)

(Inicializar função que envia mensagens através do módulo especificado)
Colocar ponteiro trans no inicio da mensagem
ENQUANTO length for > que zero
        SE obeFLAG for verdadeiro
                escrever o primeiro byte da mensagem no endereço apontado por port
                Decrementar o ponteiro trans
        FIMSE
Terminar função
```

Primitiva CB_Send (*msg, length)

```
(Inicializar rotina para enviar mensagens através da rede de comunicação (módulo 0))
Atribuir o valor endereço do módulo 0 a variável port
CHAMA função ENVIA_MENSAGEM (*msg, length, port)
Terminar rotina
```

Primitiva CB_Recv (*msg, length)

```
(Inicializar a rotina que recebe mensagens através da rede de comunicação)
Atribuir o valor endereço do módulo 0 a variável port
CHAMA função RECEBE_MENSAGEM ( *msg, lenght, port)
Terminar rotina
```

```
FUNÇÃO RECEBE_MENSAGEM (*msg, length, port)

(Inicializar função que recebe mensagens através do módulo especificado)
Colocar ponteiro trans no inicio da mensagem
ENQUANTO length for > que zero
        SE ibfFLAG for verdadeiro
                Ler o primeiro byte da mensagem no endereço apontado por port
                Incrementar o ponteiro trans
        FIMSE
Terminar função
```

Primitiva CB_Desconecta (*no_out, no_in)*

Inicializar a rotina (receber os parâmetros)
Atribuir o valor 6 a variável operação
Atribuir o endereço do módulo 2 a variável port
CHAMAR a função MONTA_CONFIG (no_origem, no_destino, operação)
CHAMAR a função ENVIA_MENSAGEM (mensgconfig, length, port)
Terminar rotina

A Interrupção de Comunicação INTRct

(Inicialização da rotina para a geração de uma interrupção para um NT)
Escrever no endereço do módulo 1 o valor da variável noINT
Terminar a rotina

A Interrupção de Comunicação INTRtc

(Inicialização da rotina para a geração de uma interrupção para um NC)
Escrever no endereço do módulo 1 o valor da variável de identificação do nó (noid)
Terminar a rotina

ANEXO II

A Linguagem SIMAN V

A.1 CONCEITOS BÁSICOS

Abaixo, seguem alguns conceitos de modelagem extraídos do capítulo 3 - *Basic Modelling Concepts* - do livro [Pegden90]:

- **Modelo:** é uma descrição funcional dos componentes do sistema e de suas interações.
- **Experimento:** define as condições experimentais (tempo de execução, condições iniciais,...) sob a qual o modelo é executado para gerar uma saída específica.
- **Entidade:** representa qualquer pessoa, objeto ou coisa - seja real ou imaginária - que movimenta-se através do sistema, causando mudanças no seu estado.
- **Atributos:** são as características de um tipo de entidade. Dentro de um dado sistema podem existir muitos tipos de entidades e cada tipo pode ter atributos específicos e únicos.
- **Processo:** representa a seqüência de operações ou atividades através das quais as entidades se movem.
- **OBS:** Entidades são dinâmicas; a entrada e a saída da entidade no/do modelo correspondem, respectivamente, à chegada e à partida no/do sistema. O número de entidades no modelo muda cada vez que uma nova entidade entra no modelo ou uma entidade existente sai do modelo. Já os processos são estáticos e são ativados pela chegada de entidades.
- **Diagramas de blocos:** é um fluxo gráfico linear e *top-down* que descreve o processo através do qual as entidades movem-se no sistema. O diagrama de blocos é construído como uma seqüência de blocos, cujas formas indicam sua função geral. O seqüenciamento dos blocos é mostrados por setas, as quais representam o fluxo de entidades de bloco para bloco.
- **Tipos de blocos básicos:** existem dez tipos de blocos básicos, sendo que os três primeiros são blocos multifuncionais, isto é, cada um representa um grupo de funções similares para modelagem de processos. Os blocos básicos são: HOLD,

TRANSFER, OPERATION, QUEUE, BRANCH, PICKQ, QPICK, MATCH, SELECT e STATION.

- **Nomes dos blocos funcionais:** cada bloco funcional possui um nome que corresponde a uma função de modelagem específica em SIMAN e consiste de um verbo sugestivo da função do bloco (EX.: CREATE para criação de entidades, DELAY para atrasos de entidades,...).

- **Operandos dos blocos funcionais:** cada bloco modela uma função geral dentro do processo. O controle da operação exata de um bloco é feito pela especificação de seus operandos. O número de operandos para cada bloco, bem como o significado de cada operando, depende do bloco em particular. Alguns operandos são opcionais e possuem valores preestabelecidos (*default*), outros não. Os operandos podem ser: constantes, variáveis, atributos, variáveis randômicas, expressões, condições e nomes simbólicos, conforme segue:

- **Constantes:** podem ser do tipo inteiro ou real (com ou sem o expoente E).

- **Variáveis:** referem-se ao conjunto de valores modificáveis que caracterizam os componentes do sistema como um todo, ou seja, não se referem às características das entidades individuais que movem-se através do sistema. Existem dois tipos de variáveis: "variáveis de propósito especial", que têm significado pré-definido em SIMAN, e "variáveis de propósito geral", cujo significado atribuído baseia-se no processo sendo modelado.

- **Atributos:** caracterizam entidades e movem-se com elas. Assim como para as variáveis, existem "atributos de propósito especial" e "atributos de propósito geral", os quais são atribuídos às entidades conforme o processo particular para o qual se desenvolve o modelo. Em muitos casos é conveniente especificar um operando de um bloco como um atributo de uma entidade que passa através do bloco, de modo que a operação do mesmo dependa desse atributo.

- **Variáveis randômicas:** são especificadas por distribuições de probabilidade, que possuem um ou mais parâmetros (média, desvio padrão,...) associados, conforme a lista abaixo:

Distribuição	Abreviação	Parâmetros
Beta	BETA	(Alpha1, Alpha2)
Continous	CONT	(CumP1, Val1, CumP2, Val2,...)
...	...	...
Exponential	EXPO	(Mean)
...	...	...
Uniform	UNIF	(Min, Max)
Weibull	WEIB	(Beta, Alpha)

- **Expressões e condições:** para a maioria dos operandos numéricos pode-se especificar uma expressão formada por uma ou mais constantes, atributos, variáveis ou variáveis randômicas. Expressões são formadas utilizando operadores aritméticos padrão (+ , - , / , * e ** para exponenciação) além de funções matemáticas fornecidas pelo SIMAN. Já as condições são formadas pela combinação de duas expressões que utilizam operadores relacionais (< , > , = = , < > , > = e < =). As expressões e condições são avaliadas em SIMAN usando a seguinte ordem de prioridades:

  1. Avaliação dentro de parênteses
  2. Operadores aritméticos (i. ** ; ii. * e / ; iii. + e -)
  3. Operadores relacionais
  4. Operadores lógicos (i. .AND. ; ii. .OR.)

- **Nomes Simbólicos:** são utilizados para referenciar vários tipos de objetos (recursos, filas,...) dentro do modelo. Cada objeto é identificado pelo nome simbólico ou pelo seu número correspondente.

A.2 Um subconjunto inicial de blocos

Esta parte são apresentados os blocos CREATE, QUEUE, SEIZE, DELAY, RELEASE, COUNT, ASSIGN, BRANCH e TALLY, utilizados para definir o modelo de simulação de um sistema qualquer.

A.2.1 CREATE - entrando entidades no modelo

Este bloco modela a chegada seqüencial de entidades dentro de um modelo de acordo com um padrão específico.

CREATE,BatchSize,Offset: Interval,MaxBatches;

- **BatchSize** é o número de entidades que o bloco deve fazer entrar em cada ponto da seqüência de chegadas. (default = 1)
- **Offset** é o tempo entre o início da simulação e a chegada do primeiro ponto na seqüência de chegadas. (default = 0)
- **Interval** é o atraso entre a chegada de sucessivos pontos após o primeiro. (default = ∞)
- **MaxBatches** é o número máximo de pontos de chegada. (default = ∞)

A.2.2 QUEUE - fornecendo espaço de espera para entidades

Este bloco modela um espaço de espera para entidades cujo movimento através do modelo tenha sido suspenso baseado no estado do sistema. Esse tipo de atraso é referenciado em SIMAN por atraso de estado.

QUEUE,QueueID,Capacity,BalkLabel;

- **QueueID** é o nome ou número exclusivos utilizados para referenciar a fila.
- **Capacity** é o número de entidades que podem residir simultaneamente em um bloco QUEUE. (default = ∞)
- **BalkLabel** define o caminho de desvio para as entidades que chegam à fila quando a capacidade da mesma está esgotada e é negada a entrada de novas entidades. Se esse parâmetro não é especificado, então uma entidade que chega a um bloco QUEUE cheio é automaticamente destruída, ou seja, abandona o modelo.

NOTAS:

1. **NQ(QueueID)** obtém o comprimento corrente da fila com identificador QueueID, ou seja, o número de entidades residindo no bloco QUEUE.

A.2.3 SEIZE - alocando recursos para entidades

Este bloco modela o atraso relacionado a alocação de recursos. O termo genérico "recursos" define um ou mais "objetos" idênticos, chamados "unidades de recurso", que podem ser alocados por uma entidade. O número de unidades de recurso idênticas correspondentes a um recurso específico é chamado "capacidade de recurso". Cada unidade de recurso individual tem seu estado: "ocupado" ou "ocioso". Recursos em SIMAN são nomeados e numerados.

SEIZE,Pr: ResName,Qty: repeats;

- **Pr** é um número inteiro que estabelece a prioridade para alocação de entidades esperando o mesmo recurso. Os blocos QUEUE-SEIZE são examinados para alocação das unidades de recurso liberadas mais recentemente com base na regra *low value first (LVF).*
- **ResName** define qual recurso está sendo requisitado pela entidade.
- **Qty** é o número de unidades de recursos alocadas para a entidade. (default = 1)
- **repeats** indica que o último segmento pode ser repetido de modo a permitir a alocação de mais de um recurso diferente ao mesmo bloco SEIZE.

NOTAS:

1. As unidades de recurso são alocadas sempre para a primeira entidade da fila de espera.
2. Não há limites para o número de recursos diferentes em um mesmo bloco SEIZE.
3. **NR(ResourceID)** indica o número atual de unidades de recurso ocupadas para o identificador de recurso ResourceID, que pode ser um número ou um nome simbólico.
4. **MR(ResourceID)** indica o número de unidades de recurso (ociosas ou ocupadas) no modelo.
5. O número, nome e capacidade de cada recurso deve ser definido no experimento pelo uso do elemento RESOURCES.

6. O bloco SEIZE é na verdade um bloco Hold e portanto deve ser precedido por um bloco QUEUE.

A.2.4 DELAY - representando atrasos de tempo

Este bloco modela os atrasos relacionados às atividades que consomem tempo, como por exemplo: máquinas, inspeções,...

DELAY: Duration,StorID;

- **Duration** é a quantidade de tempo simulado que cada entidade requer para passar através do bloco DELAY.
- **StorID** (identificador do meio de armazenamento) fornece um mecanismo para coleta estatística do número de entidades residindo em um ou mais blocos DELAY.

NOTAS:

1. Não há limite para o número de entidades que podem ser simultaneamente atrasadas pelo bloco DELAY.

A.2.5 RELEASE - liberando recursos

Este bloco fornece mecanismos para liberação dos recursos alocados por uma entidade.

RELEASE: ResName,Qty: repeats;

- **ResName** é o nome do recurso a ser liberado.
- **Qty** é a quantidade de unidades de recurso a serem liberadas. (default = 1)
- **repeats** indica que o último segmento pode ser repetido de modo a permitir a liberação de mais de um recurso diferente pelo mesmo bloco RELEASE.

NOTAS:

1. Quando uma entidade chega a um bloco RELEASE o número especificado de unidades para cada recurso muda do estado "ocupado" para o estado "*ocioso*".

A.2.6 COUNT - contando eventos

Este bloco fornece mecanismos para contagem do número de ocorrências de algum evento. Cada operação de contagem em SIMAN usa um contador, o qual tem um valor associado que pode ser incrementado ou decrementado quando uma entidade passa através do bloco COUNT.

COUNT: CounterID,Increment;

- **CounterID** é o nome ou o número utilizado para referenciar o contador.
- **Increment** é o incremento associado ao contador e pode ser positivo ou negativo. (default = 1)

NOTAS:

1. Um dado contador pode ser referenciado por um ou mais blocos COUNT.
2. Cada contador tem associado um limite; se o valor corrente de qualquer contador atinge ou excede seu limite, a simulação é automaticamente terminada pelo SIMAN.
3. Os números, nomes e limites associados aos contadores são definidos externamente no elemento COUNTERS do experimento.
4. **NC(CounterID)** fornece o valor corrente do contador especificado por CounterID.

A.2.7 ASSIGN - alterando valores de atributos e de variáveis de propósito geral

Este bloco permite a alteração dos valores de atributos e de variáveis de propósito geral durante a execução do modelo.

ASSIGN: Variable = Value: repeats;

- **Variable** é uma referência a uma variável ou atributo de propósito geral (ou de propósito especial atribuível pelo usuário).
- **Value** é o valor a ser atribuído à variável ou atributo.
- **repeats** indica que o último segmento pode ser repetido de modo a permitir a alteração do valor de mais de uma variável ou atributo pelo mesmo bloco ASSIGN.

NOTAS:

1. As variáveis do modelo são globais e os atributos são locais a cada entidade específica.
2. Quando uma atribuição é feita a uma variável, seu novo valor pode ser referenciado por qualquer entidade do modelo.
3. Quando uma atribuição é feita a uma atributo, o valor do atributo é atualizado somente para a entidade passando através do bloco ASSIGN.

A.2.8 BRANCH - direcionando o fluxo de entidades entre os blocos

Este bloco fornece uma capacidade de ramificação generalizada para direcionar o fluxo de entidades.

BRANCH, MaxTake:

WITH, Probability, Label:

IF, Condition, Label:

ELSE, Label:

ALWAYS, Label;

- **MaxTake** define o número máximo de ramos que cada entidade pode selecionar.
- **WITH, Probability, Label** fornece uma seleção randômica do ramo com base em uma probabilidade especificada. Esse tipo de ramo é chamado de "ramo probabilístico".
- **IF, Condition, Label** permite uma seleção baseada em uma condição lógica envolvendo variáveis e atributos. Esse tipo de ramo é chamado de "ramo condicional".
- **ELSE, Label: ALWAYS, Label** é uma condição nula que leva o ramo a ser selecionado, mesmo que um ramo anterior já tenha sido selecionado. Esse tipo de ramo é chamado de "ramo determinístico".

NOTAS:

1. Se mais de um ramo é selecionado, a entidade é enviada através do primeiro bloco selecionado e clones dessa entidade são criados e enviados através dos demais blocos selecionados.
2. Quando vários ramos probabilísticos emanam do mesmo bloco BRANCH, eles são considerados mutuamente exclusivos, isto é, a soma das probabilidades de todos os ramos no bloco BRANCH não pode exceder 1.
3. Em ramos determinísticos, a condição é especificada como ALWAYS se o ramo deve ser selecionado a cada chegada ao bloco.
4. Em ramos determinísticos, a condição é especificada como ELSE se o ramo é somente dado selecionado quando MaxTake ramos anteriores não tenham sido selecionados.

A.2.9 TALLY - registrando dados observacionais

Este bloco registra um valor a cada chegada de uma entidade no bloco.

TALLY: TallyID, Value;

- **TallyID** especifica o registro de conta (*tally*) para o qual as observações serão adicionadas.
- **Value** especifica que valor é registrado a cada chegada de uma entidade ao bloco TALLY. Este valor pode ser um variável SIMAN, variáveis definidas pelo usuário ou uma expressão envolvendo uma ou mais variáveis.

NOTAS:

1. O bloco TALLY é normalmente utilizado para registrar o tempo requerido por uma entidade para mover-se entre dois pontos no modelo. Isso é feito pelo uso do modificador MARK(AttributeID), adicionado a qualquer bloco do modelo. Esse modificador armazena em AttributeID o tempo em que cada entidade chega ao bloco marcado. O intervalo de tempo gasto no percurso de interesse é então

determinado pelo uso da opção INTERVAL(AttributeID) como operando **Value** em um bloco TALLY subseqüente. INTERVAL pode ser abreviado por INT.

2. O bloco TALLY é também normalmente utilizado para registrar o tempo entre sucessivas chegadas a um bloco. Isso é feito pela especificação do operando **Value** do bloco TALLY como BETWEEN (ou BET).

A.3 Um subconjunto inicial de elementos de experimento

São mostrados, nesta seção, os elementos PROJECT, DISCRETE, QUEUES, RESOURCES, COUNTERS, REPLICATE, ATTRIBUTES, VARIABLES, DSTATS, TALLIES e STORAGES, utilizados para descrever o experimento a ser realizado sobre um modelo de simulação.

A.3.1 PROJECT - descrevendo o projeto da simulação

Este elemento descreve o projeto da simulação e habilita a geração automática de um relatório ao final de cada replicação da simulação. O relatório consiste de um sumário estatístico do comportamento das variáveis selecionadas pelo modelador.

PROJECT, Title, Analyst, Date;

- **Title** é o título do projeto; pode ter um tamanho de até 24 caracteres alfanuméricos.
- **Analyst** é o nome do analista; pode ter um tamanho de até 24 caracteres alfanuméricos.
- **Date** é a data do projeto; pode ser escrita nos formatos mês/dia/ano ou dia/mês/ano, conforme o relógio do computador.

A.3.2 DISCRETE - limitando o número de entidades

Este elemento limita o número de entidades correntes no modelo. Este número é dado pela diferença entre o número de entidades que tenham entrado no modelo e o número de entidades que tenham saído do modelo. Seu valor default é função do tamanho do modelo, do número de atributos alocados a cada entidade e da memória total disponível.

DISCRETE, Entities;

- **Entities** é o número máximo de entidades correntes e deve ser menor que o valor default.

A.3.3 QUEUES - descrevendo filas

Este elemento define informações sobre as filas do modelo. Estas informações incluem os números, nomes e regras de ordenação das filas, as quais determinam a ordem na qual as entidades que chegam devem esperar na fila.

QUEUES: Number, Name, Ranking: repeats;

- **Number** é o número associado à fila. (opcional)
- **Name** é o nome associado à fila.
- **Ranking** define o mecanismo pelo qual é estabelecida a posição relativa de cada entidade esperando na fila. Os mecanismos base são: FIFO, LIFO, LVF(AtributeID) e HVF(AtributeID). (default = FIFO)
- **repeats** indica que o último segmento pode ser repetido de modo a definir mais de uma fila.

A.3.4 RESOURCES - descrevendo recursos

Este elemento fornece informações sobre os recursos do modelo, incluindo o número, o nome e a capacidade do recurso.

RESOURCES: Number, Name, Capacity: repeats;

- **Number** é o número associado ao recurso. (opcional)
- **Name** é o nome associado ao recurso.
- **Capacity** é a capacidade do recurso.
- **repeats** indica que o último segmento pode ser repetido de modo a definir mais de um recurso.

A.3.5 COUNTERS - descrevendo contadores

Este elemento fornece informações sobre os contadores do modelo: número, nome limite, opção de reinicialização e nome do arquivo de saída.

COUNTERS: Number, Name, Limit, InitOpt, OutFile: repeats;

- **Number** é o número associado ao contador. (opcional)
- **Name** é o nome associado ao contador.
- **Limit** é o valor máximo do contador, o qual, quando atingido ou ultrapassado, leva ao término da execução. (default = ∞)
- **InitOpt** controla a inicialização dos contadores, quando construindo múltiplas replicações do modelo. Se YES, os contadores são resetados no início de cada nova replicação. Se NO, os contadores acumulam seus valores a cada nova replicação.
- **OutFile** especifica o arquivo de saída no qual é armazenada a história completa dos valores do contador. (opcional)
- **repeats** indica que o último segmento pode ser repetido de modo a definir mais de um contador.

A.3.6 REPLICATE - controlando replicações

Este elemento consiste de um único segmento de linha contendo operandos para especificar o número de replicações, o tempo de início de cada replicação, o comprimento máximo das replicações, a opção de inicialização do sistema, a opção de inicialização da estatística e o período de preparação (*warm-up*) da estatística.

REPLICATE, NumReps, BeginTime, Length, InitSys, InitStats,WarmUp;

- **NumReps** é o número de replicações consecutivas da simulação que serão executadas.
- **BeginTime** é o valor do tempo inicial da primeira replicação.
- **Length** é o comprimento máximo da replicação; limita a duração de cada replicação da simulação.

- **InitSys** (YES/NO) se o sistema é inicializado, seu estado, no começo de cada replicação, é reinicializado para o mesmo estado inicial da primeira replicação, de modo que cada estado envolve as mesmas condições iniciais. Se o sistema não é inicializado, o estado inicial da replicação seguinte é dado pelo estado final da replicação anterior e cada replicação é simplesmente a continuação da anterior. (default = YES)
- **InitStats** (YES/NO) se as estatísticas são inicializadas, todos os valores estatísticos registrados para replicação anterior são descartados antes do início da replicação seguinte e o relatório gerado ao final de cada replicação reflete somente os valores registrados para a replicação anterior. Se as estatísticas não são inicializadas, cada relatório é baseado nas observações registradas da replicação corrente e das replicações anteriores. (default = YES)
- **WarmUp** é utilizado para deslocar, ou atrasar, o tempo no qual as estatísticas são inicializadas. Se o sistema é inicializado (InitSys = YES), então o período de preparação é aplicado a cada replicação do modelo e o comprimento de cada replicação é reduzido por esta quantidade. Se o sistema não é inicializado, então o período de preparação é aplicado antes do início da primeira replicação e o comprimento de cada replicação mantém-se inalterado.

A.3.7 ATTRIBUTES e VARIABLES - descrevendo atributos e variáveis de propósito geral

Estes elementos fornecem informação geral sobre os nomes simbólicos e propriedades de atributos e variáveis.

ATTRIBUTES: Number, Name(Index), Value, ... : repeats;

VARIABLES: Number, Name(Index), Value, ... : repeats;

- **Number** é o número do atributo ou variável. (opcional)
- **Name** é o nome do atributo ou variável. Pode ter um índice que define um arranjo unidimensional (Ex.: Vetor(10)) ou bidimensional (Ex.: Matriz(3,4)).

- **Value** é o valor de inicialização de atributo ou variável. Pode constituir uma lista de valores de inicialização para arranjos.
- **repeats** indica que o último segmento pode ser repetido de modo a definir mais de um atributo ou variável.

A.3.8 DSTATS - registrando dados dependentes do tempo

Este elemento leva ao registro automático de estatísticas sobre uma ou mais variáveis dependentes do tempo durante a execução da simulação.

DSTATS: Number, Expression, Name, OutFile: repeats;

- **Number** é o numero associado ao DSTATS. (opcional)
- **Expression** é uma expressão envolvendo uma ou mais variáveis com valores definidos sobre o tempo. A expressão também pode envolver varáveis definidas pelo usuário.
- **Name** é o nome associado ao DSTATS
- **OutFile** é o arquivo de saída no qual é escrita a história completa da simulação.
- **repeats** indica que o último segmento pode ser repetido de modo a definir mais de um DSTATS.

NOTAS:

1. Duas variáveis DSTAT comumente especificadas são **NQ(QueueID)** e **NR(ResourceID)**, que correspondem, respectivamente, ao número de entidades em uma fila e ao número de unidades de recurso ocupadas.

A.3.9 TALLIES - descrevendo os registros de conta

Este elemento fornece informação descritiva sobre os registros de conta (*tallies*) do modelo.

TALLIES: Number, Name, OutFile: repeats;

- **Number** é o numero associado ao registro de conta. (opcional)
- **Name** é o nome associado ao registro de conta.
- **OutFile** é o arquivo de saída opcional no qual é escrita cada observação individual obtida para o registro.
- **repeats** indica que o último segmento pode ser repetido de modo a definir mais de um registro de conta.

A.3.10 STORAGES - descrevendo armazéns

Este elemento define os nome simbólicos para cada um dos armazéns do modelo. Tais armazéns podem ser associados com blocos DELAY e fornecem uma contagem do número de entidades em tais blocos.

STORAGES: Number, Name: repeats;

- **Number** é o número associado ao armazém. (opcional)
- **Name** é o nome associado ao armazém
- **repeats** indica que o último segmento pode ser repetido de modo a definir mais de um armazém.

NOTAS:

1. **NSTO(StorageID)** define o número atual de entidades no armazém identificado por StorageID.

ANEXO III

Aspectos da Modelagem

A.1 ASPECTOS DA MODELAGEM

√ A estratégia da modelagem descreve quais recursos do *software* Arena foram utilizados para modelar o *hardware,* o *software* e os processos que ocorrem no ambiente Nó//.

- O NC e o *Crossbar* foram definidos como recursos (RESOURCES).
- Os Nós de Trabalho foram definidos como um conjunto de recursos (SETS). A definição do número de NTs da configuração da máquina simulada é feita através da alocação do mesmo.
- As chamadas de sistema foram divididas em dois grandes grupos: as chamadas regulares (*CALL*) e as chamadas irregulares (*READ*, *WRITE*, *FORK*, *EXECVE*, *WAIT*, *EXIT*). Essas sete chamadas são geradoras de entidades dentro do modelo. Com isso, é possível simular sistemas de comportamentos diferentes pela alteração da freqüência de geração dessas chamadas. Todas as chamadas são enviadas para o módulo ROTA, que se encarrega de executar a seqüência adequada.
- As chamadas de alto nível (*Send*, *Receive* e *ReceiveAny*), do núcleo (*Connect*, *ConnectAny*, *Disconnect*, *Allocate* e *Deallocate*) e de baixo nível (*INTRtc*, *C_ReceiveIntAny*, *INTRct*) foram implementadas como estações (STATIONS).
- Para cada chamada de sistema existe um caminho dentro das estações definido por uma seqüência (SEQUENCES).
- O controle sobre os NTs é feito utilizando-se a tabela (TABCOMUNIC), definida como um conjunto de recurso (SETS). TABCOMUNIC define se os nós estão ou não se comunicando pelo *Crossbar*.
- Na fila Forigemcom ficam armazenados os nós que não conseguiram se conectar com o nó que desejava conexão, eles ficam aguardando até o nó ficar disponível.

√ Para modelar as camadas de comunicação descritas no capítulo 5, foi utilizado o conceito de Estações (*Stations*) fornecido pela linguagem ARENA/SIMAN V. Abaixo são descritas cada chamada de sistema com suas respectivas decomposições em chamadas de comunicação dos níveis inferiores. A unidade de tempo adotada foi o μs (microsegundo).

Chamadas de sistema:

chamadas regulares (denominadas "*Call*");

Read (leitura de arquivos);

Write (escrita de arquivos);

Fork (criação de processos);

Exit (término do processo);

Execve (execução de código);

Wait (processo pai provoca término do processo filho).

Comunicações de alto nível:

Send;

Receive;

ReceiveAny.

Chamadas do núcleo:

Connect;

ConnectAny;

Disconnect;

Allocate;

Deallocate.

Comunicações de baixo nível:

CB_Recv;

CB_Send;

CB_Sendcomm;

CB_Recvcomm

CB_Sendcommc

CB_Recvcommc

INTRtc;

C_ReceiveIntAny;

INTRct.

A.2 Estações

As estações descrevem chamadas da Camada de Sistema, a qual é decomposta nas outras chamadas das camadas inferiores, sendo essa seqüência definida em SEQUENCES .

Call: chamadas de sistema regulares. Segue a seqüência **SCALL** que é decomposta no seguinte caminho de estações:

Send&Connect&INTRtc&AlocancC_ReceiveIntAny&C_Sendcommc&C_Recvcommc&cC_Sendcom&C_Revccomm&CB_Desconecta&**Receive&**Connect&INTRtc&AlocancC_ReceiveIntAny&C_Sendcommc&C_Recvcommc&cC_Sendcomm&C_Recvcomm&CB_Desconecta&Conexão&CB_Conecta&INTRct&Desalocanc&CB_Send&CB_Recv&Disconnect&INTRtc&Alocanc&CB_Desconecta&INTRctDesalocanc&Disconect&INTRtcAlocanc&INTRctDesalocanc&Final&.

***Fork*:** criação de processos. Segue a seqüência **SFORK** que é decomposta no seguinte caminho de estações:

Send&Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Send&CB_Desconecta&CB_Conecta&INTRct&CB_Send&INTRtc&C_ReceiveIntAny&C_Send&Disconnect&CB_Desconecta&INTRct&**Receive&**Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Recv&CB_Desconecta&CB_Conecta&INTRct&CB_Send&INTRtc&C_Recei

veIntAny&C_Send&Disconnect&CB_Desconecta&INTRct&**Alocate&**INTRtc&C_ReceiveI
ntAny&C_Send&CB_Conecta&CB_Send&CB_Desconecta&CB_Conecta&INTRct&CB_Se
nd&INTRtc&C_ReceiveIntAny&C_Send&Disconnect&CB_Desconecta&INTRct&Final:

***Read*:** leitura de arquivos. Segue a seqüência **SREAD** que é decomposta no seguinte caminho de estações:

Send&Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Send&CB_Desco
necta&CB_Conecta&INTRct&CB_Send&INTRtc&C_ReceiveIntAny&C_Send&Disconnect
&CB_Desconecta&INTRct&CB_Desconecta&INTRct&Final:**Receive&**Aux1&EsperaRSA&
Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Recv&INTRct&CB_Sen
d&INTRtc&C_ReceiveIntAny&C_Send&Disconnect&CB_Desconecta&INTRct&**Receive&**
Aux1&EsperaRSA&Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Rec
v&CB_Desconecta&CB_Conecta&INTRct&CB_Send&INTRtc&C_ReceiveIntAny&C_Sen
d&Disconnect&

***Write*:** escrita de arquivos. Segue a seqüência **SWRITE** que é decomposta no seguinte caminho de estações:

Send&Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Send&CB_Desco
necta&CB_Conecta&INTRct&CB_Send&INTRtc&C_ReceiveIntAny&C_Send&Disconnect
&CB_DesconectaINTRct&**Send&**Aux1&Connect&INTRtc&C_ReceiveIntAny&C_Send&C
B_Conecta&CB_Send&CB_Desconecta&CB_Conecta&INTRct&CB_Send&INTRtc&C_Re
ceiveIntAny&C_Send&DisconnectCB_Desconecta&INTRct&**Receive&**Connect&INTRtc&
C_ReceiveIntAny&C_Send&CB_Conecta&CB_Recv&CB_Desconecta&CB_Conecta&INT
Rct&CB_Send&INTRtc&C_ReceiveIntAny&C_Send&Disconnect&CB_Desconecta&INTR
ct&Final:

***Fork*:** criação de processos. Segue a seqüência **SFORK** que é decomposta no seguinte caminho de estações:

Send&Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Send&CB_Desco
necta&CB_Conecta&INTRct&CB_Send&INTRtc&C_ReceiveIntAny&C_Send&Disconnect
&CB_Desconecta&INTRct&**Receive&**Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_
Conecta&CB_Recv&CB_Desconecta&CB_Conecta&INTRct&CB_Send&INTRtc&C_Recei
veIntAny&C_Send&Disconnect&CB_Desconecta&INTRct&**Alocate&**INTRtc&C_ReceiveI
ntAny&C_Send&CB_Conecta&CB_SendCB_Desconecta&CB_Conecta&INTRct&CB_Send
&INTRtc&C_ReceiveIntAny&C_Send&Disconnet&CB_Desconecta&INTRct&Final:

***Exit*:** término de processo. Segue a seqüência **SEXIT** que é decomposta no seguinte caminho de estações:

FilaExit&**Send&**Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Send&
CB_Desconecta&CB_Conecta&INTRct&CB_Send&INTRtc&C_ReceiveIntAny&C_Send&
Disconnect&CB_Desconecta&INTRct&**Receive&**Connect&INTRtc&C_ReceiveIntAny&C_
Send&CB_Conecta&CB_Recv&CB_Desconecta&CB_Conecta&INTRct&CB_Send&INTRt
c&C_ReceiveIntAny&C_Send&Disconnect&CB_Desconecta&INTRct&**Send&**Aux1&Conn
ect&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Send&CB_Desconecta&CB_
Conecta&INTRct&CB_Send&INTRtc&C_ReceiveIntAny&C_Send&Disconnect&CB_Desc
onecta&INTRct&**Deallocate&**INTRtc&C_ReceiveIntAny&C_Send&Final:

***Exec*:** execução de código. Segue a seqüência **SEXEC** que é decomposta no seguinte caminho de estações:

Send&Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Send&CB_Desco
necta&CB_Conecta&INTRct&CB_Send&INTRtc&C_ReceiveIntAny&C_Send&Disconnect
&CB_DesconectaINTRct&**Receive&**Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_C
onecta&CB_Recv&CB_Desconecta&CB_Conecta&INTRct&CB_Send&INTRtc&C_Receiv
eIntAny&C_Send&Disconnect&CB_Desconecta&INTRct&**Receive&**Aux1&EsperaRSA&C
onnect&INTRtc&C_ReceiveIntAny&C_SendCB_Conecta&CB_Recv&CB_Desconecta&CB
_Conecta&INTRct&CB_Send&INTRtc&C_ReceiveInAny&C_Send&Disconnect&CB_Desc
onecta&INTRct&Final:

Wait: processo pai requisita término do processo filho. Segue a seqüência **SWAIT** que é decomposta no seguinte caminho de estações:

FilaWait&**Send&**Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Send&CB_Desconecta&CB_Conecta&INTRct&CB_Send&INTRtc&C_ReceiveIntAny&C_Send&Disconnect&CB_Desconecta&INTRct&**Receive&**Connect&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Recv&CB_Desconecta&CB_Conecta&INTRct&CB_Send&INTRtc&C_ReceiveIntAny&C_Send&Disconnect&CB_Desconecta&INTRct&**ReceiveAny&**Aux1&ConnectAny&INTRtc&C_ReceiveIntAny&C_Send&CB_Conecta&CB_Recv&CB_Desconecta&CB_Conecta&INTRct&CB_Send&INTRtc&C_ReceiveIntAny&C_Send&Disconnect&CB_Desconecta&INTRct&Final:

A.3 Descrição das estações

As estações criadas possuem a mesma denominação das primitivas utilizadas no sistema operacional, como descrito no capítulo 5. Abaixo segue uma breve descrição das funções realizadas em cada estação criada no modelo:

CALL: seleciona nó origem da chamada, seleciona nó destino da chamada e seleciona seqüência SCALL.

READ: seleciona nó origem da chamada, estabelece nó destino como RSA, seleciona seqüência SREAD.

WRITE: seleciona nó origem da chamada, estabelece nó destino como RSA e seleciona seqüência SWRITE.

FORK: seleciona nó origem da chamada, estabelece nó destino (nó não alocado) e seleciona seqüência SFORK.

EXIT: seleciona nó origem (exceto nó INIT), estabelece nó destino como RSA e seleciona seqüência SEXIT.

EXECVE: seleciona nó origem, estabelece nó destino como RSA e seleciona seqüência SEXEC.

WAIT: seleciona nó origem (se existirem filhos), estabelece nó destino como RSA e seleciona seqüência SWAIT.

Send: define o tamanho da mensagem TAM e espera tempo de 1 μs.

Receive: define o tamanho da mensagem TAM e espera tempo de 1 μs.

ReceiveAny: define o tamanho da mensagem: TAM e espera tempo de 1 μs.

Connect: aloca origem e destino na tabela de comunicação TABPROC e espera tempo de 2μs.

ConnectAny: aloca origem e destino na tabela de comunicação TABPROC e espera tempo de 2 μs.

Disconnect: libera nós origem e destino na tabela de comunicação TABPROC e espera tempo de 1μs.

Allocate: define tamanho da mensagem: TAMCARREG = 300 + TAMDESCRITOR=200, pesquisa um nó livre a ser alocado (destino), aloca nó destino na tabela de processamento TABPROC e aloca nós origem e destino na tabela de comunicação TABCOMUNIC.

Deallocate: libera nó origem na tabela de processamento TABPROC.

CB_Recv: aguarda um tempo para recepção de dados pelo *Crossbar,* que é em função do tamanho da mensagem.

CB_Send: aguarda um tempo para transmissão de dados pelo *Crossbar.* Esse tempo é em função do produto do tamanho da mensagem (TAM) pelo tempo de escrita de um *byte* no barramento (0.24μs).

CB_Sendcomm: Aguarda um tempo para transmissão de uma mensagem de comando. O tempo para transmissão desta mensagem é o produto do seu tamanho pelo tempo de escrita no barramento (idem ao item anterior).

CB_Recvcomm: Aguarda um tempo para recepção de uma mensagem de comando. Esse tempo é em função do produto do tamanho da mensagem pelo tempo de leitura no barramento (0.21μs).

INTRtc: aguarda um tempo para gerar a interrupção que equivale a 0.23 μs e Aloca o NCONTR (NC).

C_ReceiveIntAny: Depois que a entidade aloca o NCONTR, ela entra nesta *station* aguarda o tempo de detecção de uma interrupção. Este tempo é equivalente a 0.4 μs, onde 0.19 μs são necessários para o processador detectar uma interrupção e realizar a leitura do vetor de interrupções e 0.21 μs são para leitura no módulo 1 para a identificação do NT gerador da INTRtc

INTRct: avisa ao NTs que a sua requisição de serviço foi atendida, seja ela estabelecimento ou cancelamento de conexão, e aguarda o tempo para sua geração e equivalente a 0.23 μs.

Conexão: aloca os recursos de índice origem e destino na TABCOMUNIC e aguarda 2 μs para manipulação e atualização da tabela de configuração.

Desconexão: libera os recursos de índice origem e destino na TABCOMUNIC e aguarda 2 μs.

CB_Conecta: aloca e espera o tempo de envio de 4 *bytes* de configuração do *Crossbar,* O envio desta mensagem de configuração segue o protocolo de comunicação serial Transputer *Link*, onde o tempo para transmissão de quatro *bytes* leva 13 μs.

CB_Desconecta: desaloca *Crossbar*.

Final: armazena estatísticas;

A.1 VISÃO DO MODELO NO ARENA

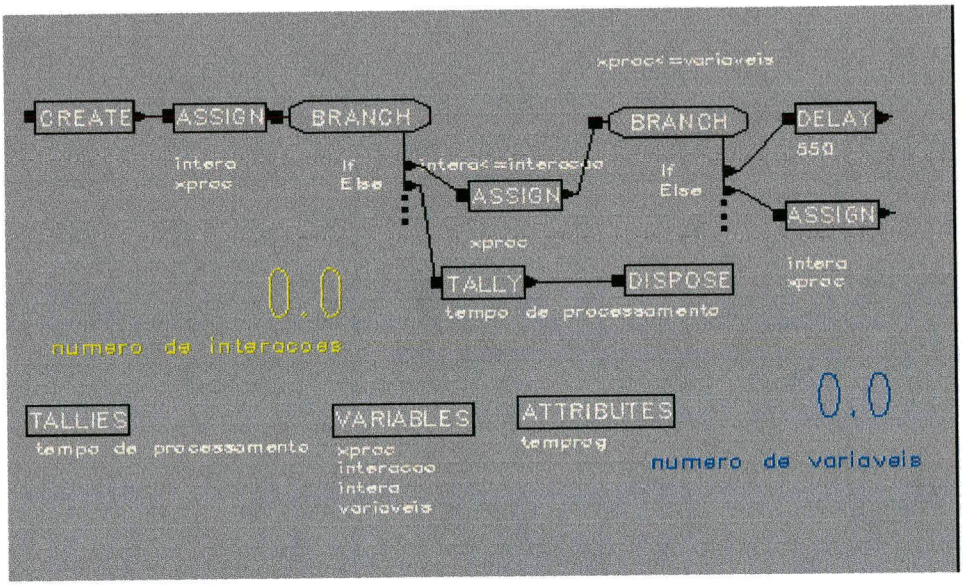


Figura 1: Visão do modelo seqüencial

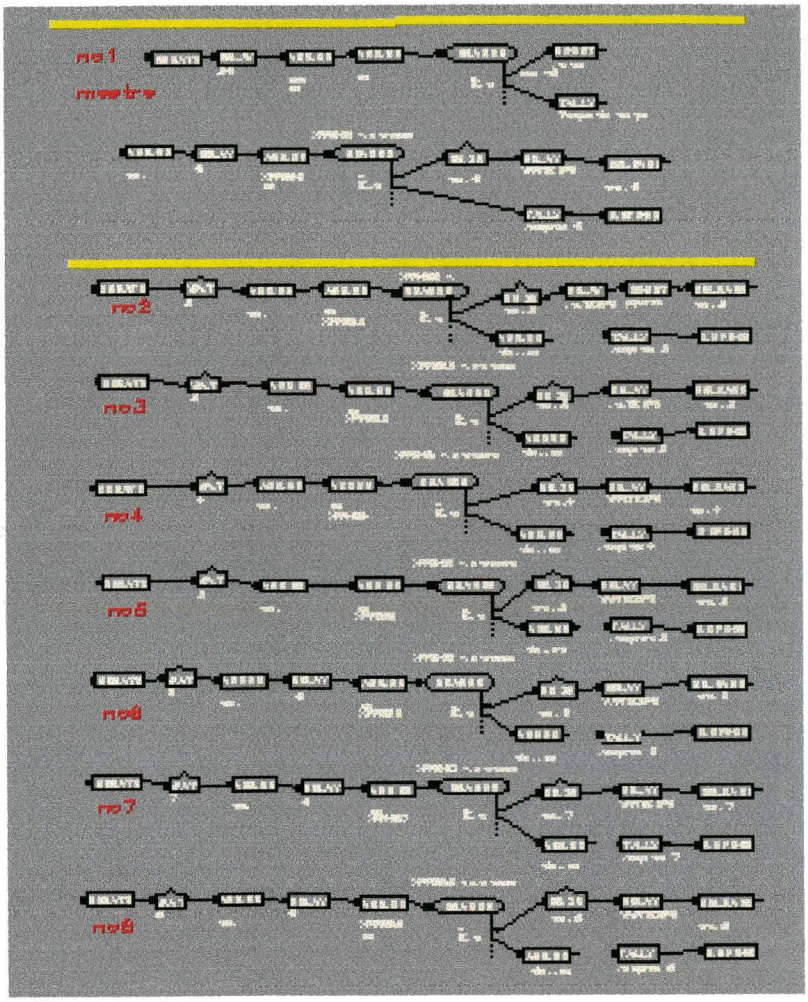


Figura 2: Visão da modelagem da aplicação

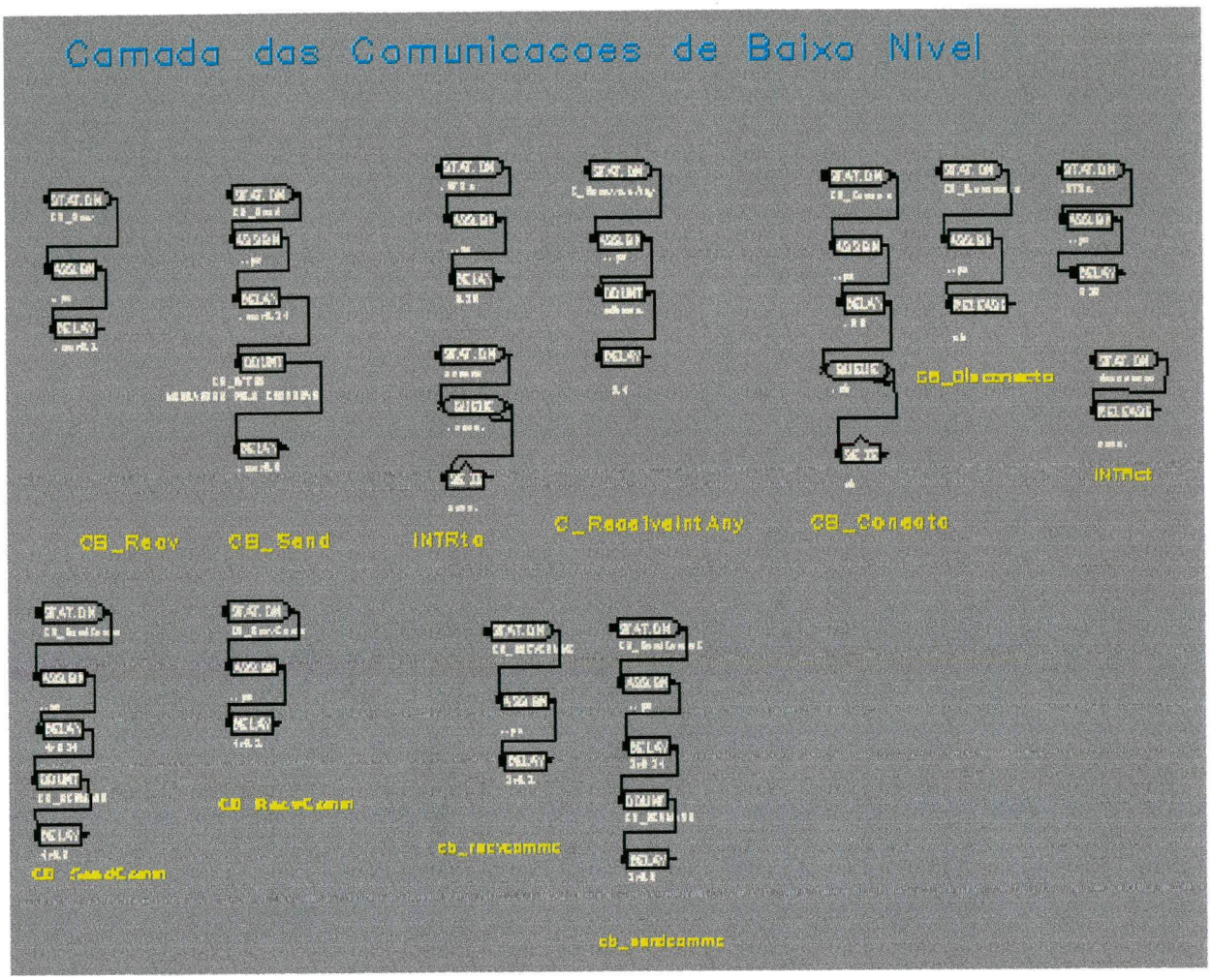


Figura 3: Visão da modelagem da camada de baixo nível.

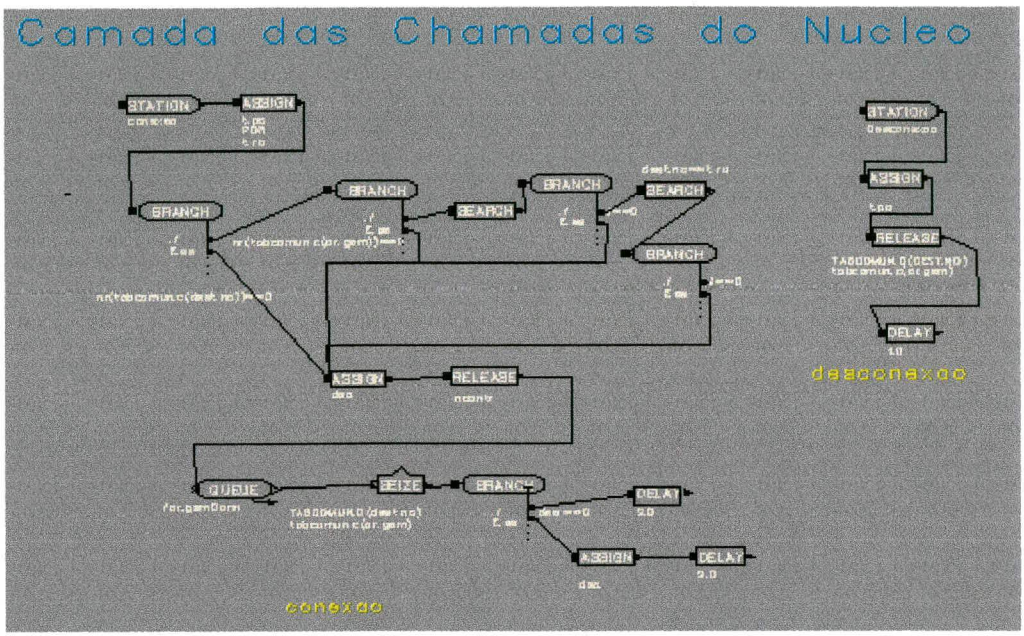


Figura 4: Visão da modelagem da camada das chamadas do núcleo.

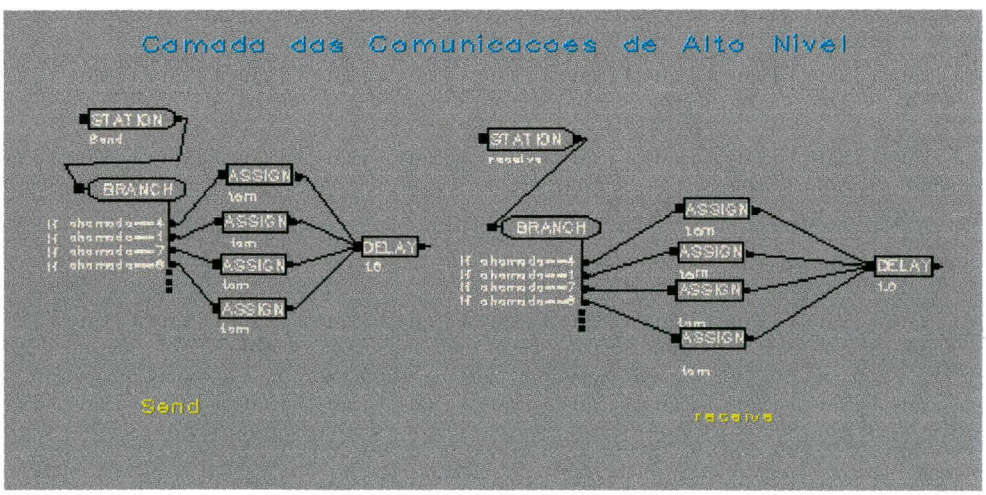


Figura 5: Visão da modelagem da camada das comunicações de alto nível.

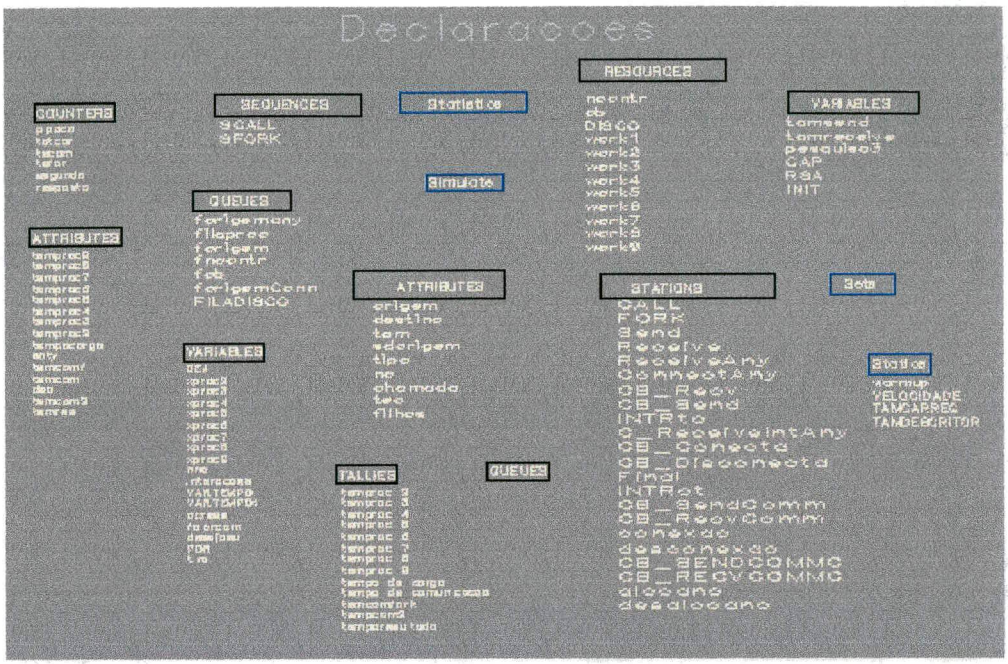


Figura 6: Visão das declarações do modelo.

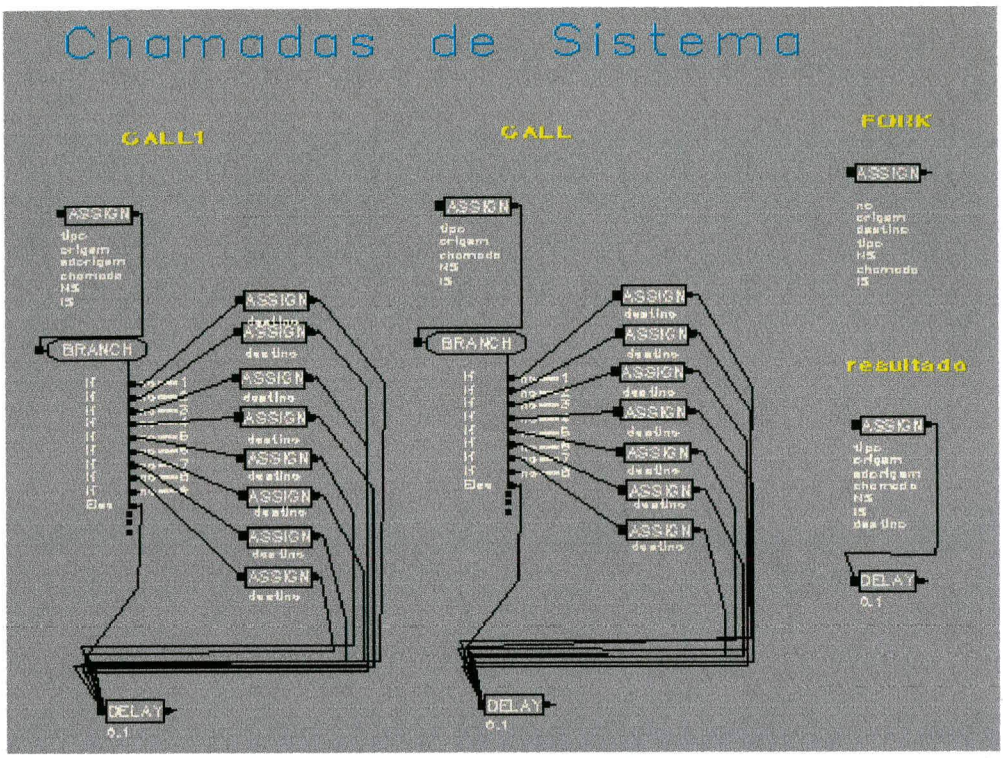


Figura 7: Visão da modelagem das chamadas do sistema.

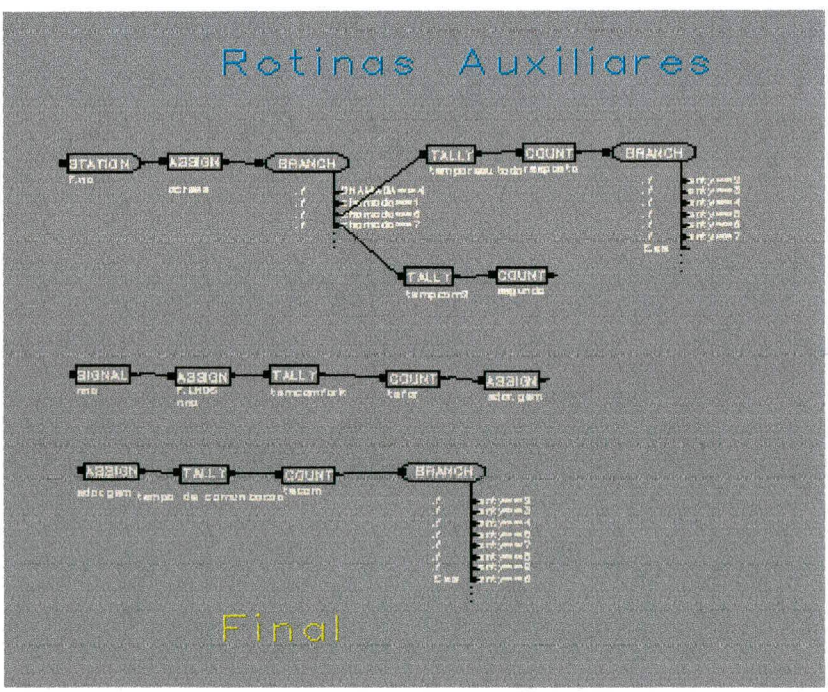


Figura 8: Visão da modelagem das rotinas auxiliares.