UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E DE ESTATÍSTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# AMBIENTE PARA EXECUÇÃO DE PROGRAMAS PARALELOS ESCRITOS NA LINGUAGEM SUPERPASCAL EM UM MULTICOMPUTADOR COM REDE DE INTERCONEXÃO DINÂMICA

por

Carla Merkle

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

Prof. Thadeu Botteri Corso

Orientador

Florianópolis, fevereiro de 1996.

AMBIENTE PARA EXECUÇÃO DE PROGRAMAS PARALELOS
ESCRITOS NA LINGUAGEM SUPERPASCAL EM
UM MULTICOMPUTADOR COM REDE DE INTERCONEXÃO DINÂMICA

CARLA MERKLE

ESTA DISSERTAÇÃO FOI JULGADA ADEQUADA PARA A OBTENÇÃO DO TÍTULO DE

**MESTRE EM CIÊNCIA DA COMPUTAÇÃO**

NA ÁREA DE CONCENTRAÇÃO DE SISTEMAS DE COMPUTAÇÃO, SUB-ÁREA SISTEMAS OPERACIONAIS E APROVADA EM SUA FORMA FINAL PELO PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO.

Prof. Thadeu Botteri Corso, M.Sc.
Orientador

Prof. Rogério Cid Bastos, Dr.
Coordenador do Curso

BANCA EXAMINADORA :

Prof. Thadeu Botteri Corso, M.Sc. (Presidente)

Prof. Vitório Bruno Mazzola, Dr.

Prof. Luiz Fernando Jacintho Maia, Dr.

# AGRADECIMENTOS

Este é o resultado de um trabalho que contou com a colaboração de várias pessoas. Agradeço a todas elas e em especial ao meu orientador Thadeu pela paciência, sabedoria e inteligência dedicadas na orientação.

Um agradecimento ao colega Carlos Barros Montez pelas explicações e dicas. Agradeço ao Rodrigo pelo apoio enquanto esteve por aqui.

Um agradecimento também ao meu melhor amigo, por me motivar nos momentos necessários !

À minha família, pela paciência e apoio !

"Faça com que eles construam juntos uma torre, e eles se tornarão irmãos."

Saint-Exupéry

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Resumo

O desenvolvimento de programas paralelos como redes de processos comunicantes viabiliza a utilização da grande potência de processamento dos multicomputadores. Em um programa paralelo, a criação de processos e a comunicação entre eles pode produzir redes dinâmicas que devem se adaptar à topologia da máquina destino.

Este trabalho, dentro do contexto do projeto do multicomputador Nó Paralelo, ou Nó //, [COR93], tem como objetivo principal a implementação da linguagem paralela SuperPascal [HAN94a] nesse multicomputador, visando estender o emprego dessa máquina com uma linguagem de programação paralela. As principais características dessa linguagem são: simplicidade, segurança e portabilidade. Na linguagem SuperPascal, a criação de processos e a comunicação entre eles se adaptam adequadamente à topologia dinâmica do multicomputador Nó //.

# Abstract

The development of parallel programs as nets of communicating processes make feasible the use of the great processing power of multicomputers. In a parallel program, the creation of processes and the communication among them can produce dynamic nets that should match the target architecture topology.

This is a work in the context of the Nó Paralelo multicomputer project, or Nó // [COR93]. Its main purpose is the implementation of the SuperPascal parallel language [HAN94a] in this multicomputer, allowing parallel programming on this machine. The most important features of this language are : simplicity, security and portability. In SuperPascal, the creation of processes and the communication among them match adequately the Nó // multicomputer dynamic topology.

# 1. Introdução

O desenvolvimento de linguagens de programação paralela viabiliza a utilização da grande potência de processamento das máquinas paralelas. Desenvolver linguagens de programação efetivas para máquinas paralelas como multicomputadores continua sendo um desafio nos dias atuais [HAN94a]. Para a concretização de tais linguagens, é útil o suporte de sistemas operacionais para gerência do paralelismo, da sincronização e da comunicação entre os processos que irão executar nos diferentes nós da máquina e se comunicar através de canais providos pela rede de interconexão. A criação de processos e a comunicação entre eles pode produzir redes dinâmicas que devem se adaptar à topologia da máquina destino.

Na linguagem paralela SuperPascal [HAN94a], a criação de processos e a comunicação entre eles adapta-se adequadamente à topologia dinâmica do multicomputador Nó // (lê-se Nó Paralelo).

O projeto do multicomputador Nó // [COR93], em desenvolvimento no Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina, pelos grupos de pesquisa em arquitetura de computadores, sistemas operacionais e linguagens de programação, objetiva a concepção de um ambiente completo para programação paralela utilizando idéias simples e inovadoras para resolver os problemas clássicos.

Este trabalho tem como objetivo principal a implementação da linguagem SuperPascal no Nó // estendendo o seu emprego com uma linguagem de programação paralela. Utiliza-se como ambiente de implementação o simulador do multicomputador do Nó // e o sistema operacional *Crux* ([CAM95] e [MON95]).

O presente texto está dividido em seis capítulos. O capítulo 2 descreve máquinas paralelas e, em especial, os multicomputadores. O capítulo 3 apresenta algumas linguagens de programação com suas características de paralelismo, comunicação e ambientes de programação. O capítulo 4 descreve um simulador do multicomputador Nó // e o sistema operacional *Crux*, que constituem o ambiente efetivo para a implementação proposta neste trabalho. Finalmente, o capítulo 5 aborda a implementação da linguagem de programação SuperPascal no Nó // e o capítulo 6 descreve conclusões, contribuições e perspectivas futuras desse trabalho.

# 2. Máquinas Paralelas

A construção de máquinas paralelas exige a interconexão de processadores, memórias e canais de comunicação. As máquinas paralelas de uso geral podem ser divididas em três grandes grupos em função do grau de integração dos seus componentes [AUS91] : multiprocessadores, multicomputadores e redes locais.

Esses grupos se distinguem pelo grau e velocidade das interações possíveis entre seus componentes. Os **multiprocessadores** (ou máquinas paralelas com memória compartilhada) são computadores individuais com processadores que acessam memórias compartilhadas através de redes de interconexão dinâmicas. Os **multicomputadores** (ou máquinas paralelas com memória distribuída) são compostos de nós autônomos compostos de processadores, de memórias privativas e de canais de comunicação ligados através de redes de interconexão formadas por canais de comunicação bipontuais exclusivos. As **redes locais** são compostas de computadores independentes completos interconectados através de canais de comunicação compartilhados.

## 2.1. Multicomputadores

A conjunção das seguintes características permite distinguir os multicomputadores das outras máquinas paralelas [REE87]:

- *Grande número de elementos processadores homogêneos* - O baixo custo dos componentes e a facilidade de montagem permitem a construção de máquinas com grande quantidade de nós processadores.
- *Interação baseada em trocas de mensagens* - Os processos de um programa paralelo que executam em nós diferentes podem interagir somente através de trocas de mensagens.

- *Alta velocidade das comunicações* - Os canais dos multicomputadores constituem meios confiáveis de comunicação, apresentando velocidades de uma ordem de grandeza superior a das redes locais.
- *Granularidade média de processamento* - Os multicomputadores encorajam a exploração do paralelismo real pela decomposição de programas em vários processos cooperantes. Entretanto, uma granularidade muito fina pode se tornar inadequada pelo predomínio das comunicações.
- *Programação* - O desenvolvimento de programas paralelos compostos de vários processos cooperantes deve contar com linguagens, compiladores e sistemas operacionais adequados para explorar automaticamente o paralelismo fornecido pelo multicomputador.
- *Grande número de canais de comunicação bipontuais* - As redes de interconexão formam topologias regulares com muitos canais conectando aos pares os nós dessas máquinas.

As redes de interconexões são fator decisivo na determinação do desempenho geral dos multicomputadores pois, mesmo com altas velocidades de processamento dos nós, para a execução dos processos cooperantes, são necessárias trocas de mensagens pela rede de interconexão (figura 1). Dessa forma, o desempenho do multicomputador depende de uma combinação adequada da velocidade de processamento dos nós e da velocidade com que é possível trocar mensagens nas redes de interconexões.

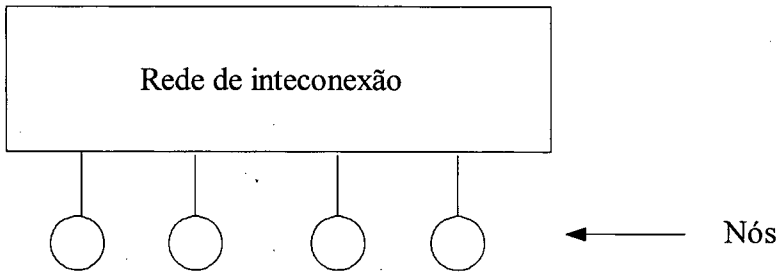


Figura 1 : Multicomputador.

## 2.2. Redes de Interconexão

Redes de interconexão são constituídas de entidades de *hardware* (canais de comunicação) e *software* (controle do estabelecimento dos canais) que são projetadas para facilitar a comunicação entre processos e processadores.

Um ambiente de computação paralela pode ser representado como na figura 2 [FEN81] :

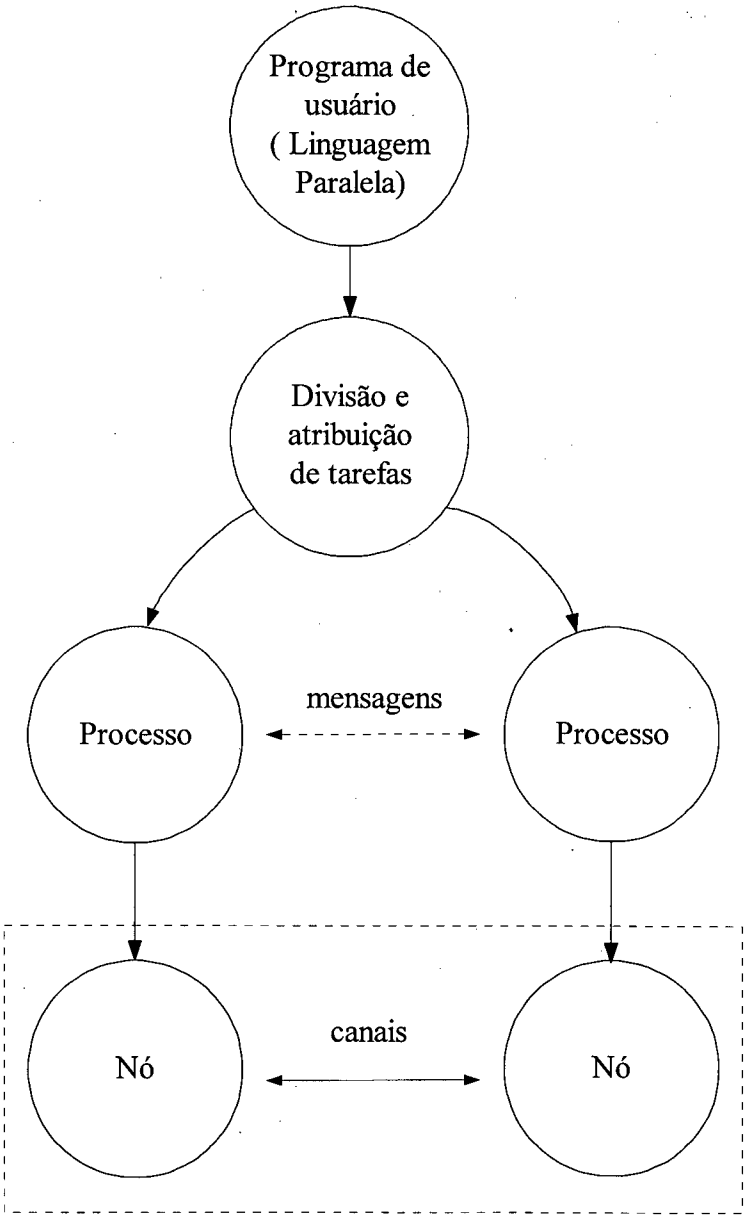


Figura 2 : Sistemas de processamento paralelo.

A rede de interconexão é o elemento principal de uma arquitetura paralela. No passado, essas redes serviam somente à comunicação de dados, resultado direto do conceito de trocas de mensagens no processamento distribuído. A tendência atual é de projetar redes de interconexão de tal forma que também funções a nível de sistema tais como alocação de recursos e sincronização possam ser facilmente implementadas usando facilidades da rede.

Redes de interconexão são consideradas **recursos** da mesma forma que processadores. Elas influenciam fortemente o projeto de algoritmos, linguagens, compiladores e escalonadores. Conseqüentemente, o problema do projeto de redes de interconexão efetivas torna-se cada vez mais crítico encorajando a busca de arquiteturas paralelas inovadoras.

Na seleção de uma arquitetura de rede de interconexão, algumas importantes decisões de projeto podem ser identificadas : topologia da rede, técnicas de reconfiguração, estratégia de controle, método de comutação e combinação entre algoritmo e arquitetura.

### √ Topologia da Rede de Interconexão

A topologia de uma rede de inteconexão pode ser representada por um grafo onde os vértices representam os nós processadores e as arestas representam canais de comunicação. A topologia das arquiteturas paralelas é um parâmetro de extrema importância por ter influência sobre o paralelismo a nível de *hardware*. As redes de interconexão bipontuais se dispõem de forma regular e podem ser agrupadas em duas categorias : estáticas e dinâmicas [FEN81].

Um grande número de propostas de redes de interconexão têm aparecido na literatura e uma enorme quantidade de pesquisa é centrada no projeto e análise dessas redes.

### √ Técnicas de Reconfiguração das Redes Dinâmicas

Técnicas de reconfiguração são usadas para alocar recursos de *hardware*, tais como nós processadores e canais de comunicação, para uma tarefa específica. Os recursos de *hardware* alocados são normalmente interconectados para formar uma topologia de rede adequada para a tarefa. A reconfiguração é executada quando uma tarefa solicita recursos através de um controlador do sistema. Uma técnica de reconfiguração eficiente deve encaixar várias topologias de rede com um pequeno *overhead*, enquanto a utilização de recursos permanece alta.

A técnica de reconfiguração é particularmente importante no escalonamento de sistemas de grande escala para computação de alto desempenho já que existe uma necessidade de combinar arquitetura e algoritmo e de alocar recursos de comunicação e processadores. A reconfiguração depende da habilidade da rede para adicionar novas conexões arbitrárias para uma rede com conexões existentes quaisquer. A rede pode permitir que qualquer conexão arbitrária seja estabelecida a qualquer instante ou o estabelecimento de alguma conexão pode ser bloqueado por uma conexão existente que usa um ponto de cruzamento que é necessário para a nova conexão. Dentre os principais termos que descrevem a habilidade das redes para estabelecer conexões pode-se ressaltar : não bloqueante e bloqueante.

Uma rede é **não bloqueante** se qualquer conexão desejada entre nós não utilizados pode ser estabelecida imediatamente sem interferência de quaisquer conexões existentes. Uma rede é **bloqueante** se existem grupos de conexões que impedem que conexões adicionais sejam estabelecidas entre nós não utilizados.

As classes de redes relevantes ao bloqueio são um resultado direto de várias topologias de rede onde a relação custo/desempenho principal é entre bloqueio e quantidade de pontos de cruzamento. Em geral, quanto mais pontos de cruzamento, mais rotas alternativas possíveis e menor a chance de bloqueio. Muito do esforço no estudo e desenvolvimento dessas topologias de rede referem-se à obtenção de alguma capacidade de conexão necessária com um número mínimo de pontos de cruzamento.

### √ Estratégia de Controle e Método de Comutação das Redes Estáticas

A estratégia de controle refere-se ao transporte de mensagens, roteamento e gerência de armazenamento temporário.

Roteamento pode ser implementado por dois métodos de comutação diferentes : comutação de circuitos e comutação de pacotes. Na comutação de circuitos um caminho físico é estabelecido entre uma fonte e um destino. Na comutação de pacotes, dados são colocados em um pacote e roteados através da rede de interconexão sem o estabelecimento de uma conexão física.

Na comutação de circuitos, uma vez estabelecida a conexão, dados podem ser transmitidos diretamente da entrada para a saída a altas velocidades dependentes somente das características elétricas do canal. Na comutação de pacotes há a necessidade de roteamento e de gerência de armazenamento temporário de pacotes enquanto eles estão a caminho do destino.

### √ Combinação entre Algoritmo e Arquitetura

O problema de mapeamento surge quando o padrão de comunicação do algoritmo paralelo difere da topologia da rede de interconexão do multicomputador [HWA87]. Mapeamento refere-se a forma utilizada para atribuir processos a recursos tais como os nós com o objetivo de minimizar o número total de passos de roteamento.

A exploração do paralelismo real provido pelos multicomputadores impõe a construção de programas paralelos sob a forma de redes de processos que se comunicam exclusivamente através de trocas de mensagens. Essas redes vêm sendo consideradas como um bom modelo de programação paralela e têm dado origem a várias linguagens como, por exemplo, Occam [INM84] e SuperPascal [HAN94a].

Conforme será apresentado com mais detalhes no capítulo 3, um programa Occam é formado por um número fixo de processos cujo padrão de comunicação é conhecido antes do início de sua execução e um programa SuperPascal pode ser formado por um número variável de processos cujo padrão de comunicação pode ser desconhecido antes do início da execução e sofrer alterações ao longo da mesma.

Com o objetivo de fornecer suporte às linguagens acima citadas, existem duas abordagens de combinação entre algoritmo e arquitetura paralela nos multicomputadores :

⇒ se todas as tarefas paralelas no processamento são conhecidas *a priori* elas podem ser estaticamente mapeadas nos nós da rede antes do processamento começar;

⇒ se novas tarefas iniciam e tarefas existentes terminam conforme o desenrolar do processamento, elas devem ser atribuídas dinamicamente aos nós da rede.

## 2.2.1. Redes de Interconexão Estáticas

As topologias estáticas apresentam canais ligando direta e estaticamente nós sem a possilidade de reconfiguração. A comunicação entre nós não diretamente ligados ocorre por intermédio de outros nós. Dois exemplos significativos de redes estáticas são a grelha quadrada e o hipercubo.

A **grelha quadrada** consiste de uma coleção de nós conectados como na figura 3. Ela possui $n^2$ nós, onde $n$ representa o número de nós em cada linha ou coluna. A figura 3 apresenta uma grelha quadrada com $n = 3$, também denominada grelha 3 x 3.

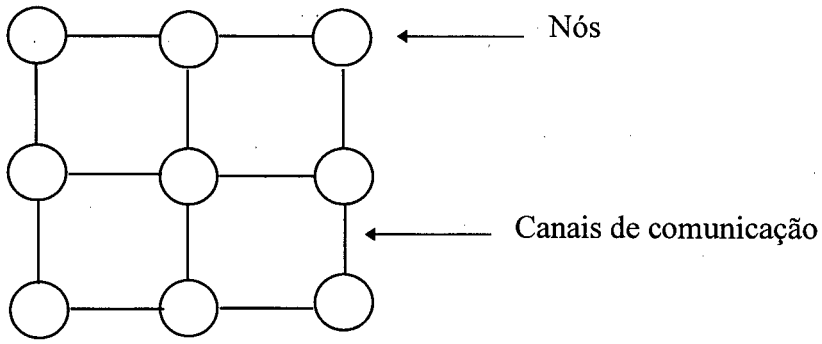


Figura 3 : Grelha quadrada 3 x 3.

O **hipercubo** ou $n$-cubo binário [REE87] é uma máquina cuja topologia é caracterizada pelo parâmetro $n$, denominado sua dimensão, que determina o número de ligações de cada nó (figura 4). Um hipercubo de dimensão $n$ possui $2^n$ nós. Um hipercubo de dimensão $n$+1 é obtido como a composição de dois hipercubos de dimensão $n$. Dessa forma um hipercubo de dimensão 0 possui um único nó, um de dimensão 1 é composto de dois nós, um de dimensão 2 é formado de quatro nós e assim por diante.

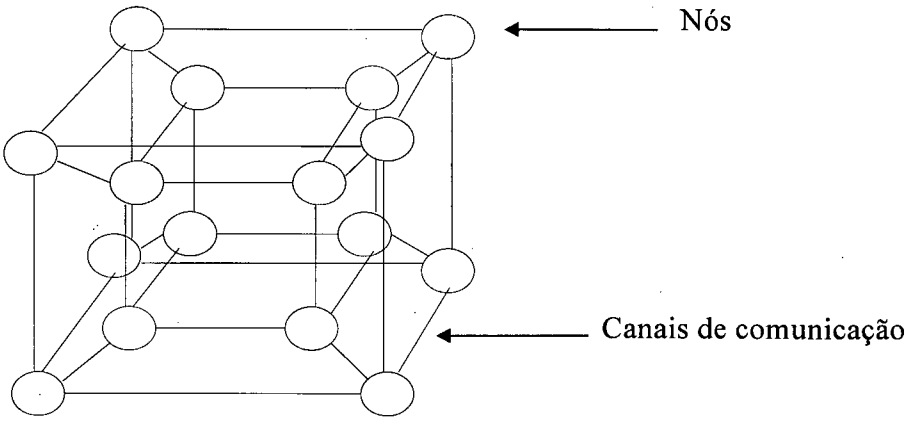


Figura 4 : Hipercubo de dimensão 4.

Alguns aspectos importantes a considerar nas topologias estáticas são : o tipo de aplicação para as quais elas são mais eficientes, a possibilidade de crescimento do número de nós e o comprimento máximo do caminho que uma mensagem percorre em cada topologia.

Aplicações específicas com seus respectivos padrões de comunicação sugerem a utilização de uma topologia determinada. Como exemplo de aplicação para a topologia grelha quadrada tem-se a multiplicação de matrizes.

Escalabilidade é a qualidade de uma máquina ser capaz de crescer em número de nós para aumentar sua potência de processamento. Essa é uma característica importante na maioria dos ambientes de computação onde a necessidade por capacidade de processamento tende a crescer com o passar do tempo de forma a exceder o que está disponível. A solução usual é abandonar o computador atual e comprar um novo. Um sistema escalável é capaz de crescer conforme crescem as necessidades dos usuários.

A topologia de uma máquina pode ter influência muito importante no fator escalabilidade. O número de canais por nó não se altera com o acréscimo de linhas e de colunas da grelha quadrada. Já o número de conexões de cada nó cresce com a dimensão do hipercubo. No projeto do *hardware* de qualquer topologia uma dimensão máxima sempre deve ser fixada. Por exemplo,

um nó possuindo 10 canais pode ser usado para construir qualquer hipercubo com até $2^{10}$ (1024) nós.

Tanto na grelha quadrada como no hipercubo, uma mensagem pode eventualmente percorrer vários nós para atingir seu destino. No entanto, a rota percorrida entre os nós mais distantes em ambas as topologias para o mesmo número de nós, definida como diâmetro da rede, é mais longa na grelha. Na grelha quadrada com $n$ nós em cada linha ou coluna, o diâmetro é igual a $2n$-2 [HWA93] enquanto no hipercubo é igual a sua dimensão.

Um dos aspectos que determina a flexibilidade das topologias é a possibilidade de executar uma aplicação feita para uma topologia diferente. Assim, pode-se executar eficientemente uma aplicação feita para a grelha quadrada em um hipercubo se o hipercubo contiver a grelha quadrada como sub-rede.

### 2.2.2. Redes de Interconexão Dinâmicas

Na topologia dinâmica, os canais dos nós não estão ligados diretamente uns aos outros mas utilizam um comutador de conexões para essa tarefa. As ligações podem ser reconfiguradas pelo ajuste dos elementos de comutação ativos da rede de interconexão.

Topologias da categoria dinâmica são classificadas em barramentos, redes multiestágio e *crossbar*.

√ **Barramentos** - são uma coleção de fios e conectores interconectando os vários nós de processamento de forma compartilhada (figura 5). Têm baixo custo mas possuem limitação na sua capacidade de transferência acarretando degradação de desempenho do sistema quando sobrecarregados.

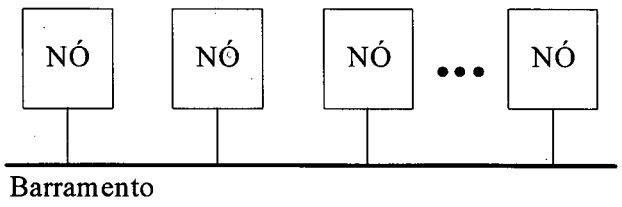


Figura 5 : Multicomputador baseado em barramento.

√ **Redes multiestágio** - são constituídos de vários estágios de pontos de cruzamento. Apenas o primeiro e o último estágios são conectados a nós de processamento. Pontos de cruzamento nos estágios intermediários são conectados a outros pontos de cruzamento. Vários estágios geralmente diminuem o número de pontos de cruzamento necessários para grandes redes. Vários estágios também aumentam o tempo para conectar dois nós que torna-se significativo em redes de muitos estágios. Um exemplo de rede multiestágio é a rede ômega na qual o tempo para conectar qualquer par de nós tem sempre a mesma duração (figura 6).

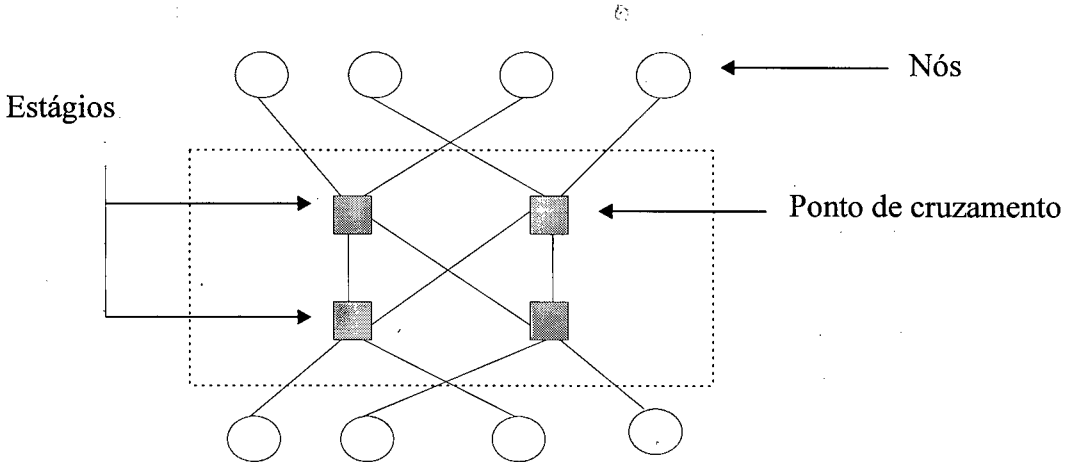


Figura 6 : Rede ômega 2 x 2.

√ ***Crossbar*** - Um *crossbar* é uma rede de interconexão que possui um ponto de cruzamento para cada par nó de entrada - nó de saída. Nessa rede, apenas um ponto de contato precisa ser fechado para estabelecer a conexão de qualquer par de nós. Ela é a única rede de comutação de circuitos verdadeiramente de estágio único [BRO83]. O *crossbar* com $n$ nós de entrada e $n$ nós de saída possui $n^2$ pontos de cruzamento (figura 7).

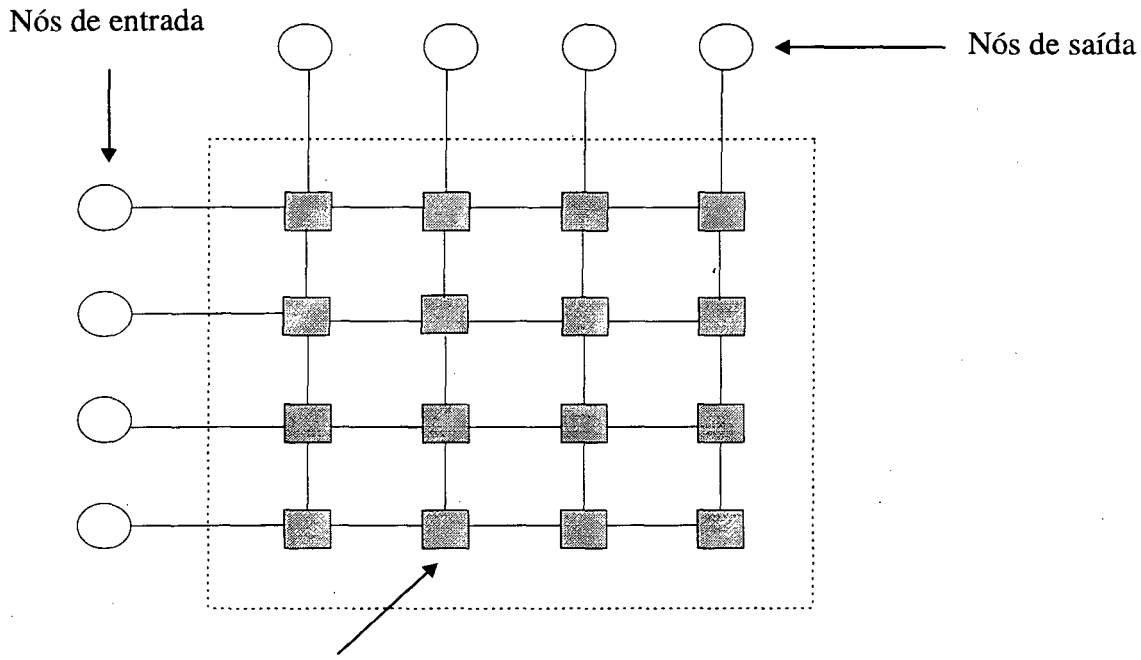


Figura 7 : *Crossbar* 4 x 4.

O *crossbar* é a rede de interconexão dinâmica mais poderosa [HWA93]. Ela é não bloqueante possuindo reduzido tempo para conexão de dois nós. Entretanto ela é de alto custo pelo grande número de pontos de cruzamento necessários.

"Custo" em redes de comutação é normalmente relacionado ao que algumas vezes é referido como "complexidade" ou número de elementos de comutação necessários. O custo de um *crossbar* pode ser medido pelo número de entradas multiplicado pelo número de saídas. Assim, o custo não está necessariamente relacionado ao preço, sendo antes uma medida para comparar a complexidade relativa das redes.

Quanto à escalabilidade é importante ressaltar que os avanços da tecnologia não podem ser ignorados. Apesar do estudo de redes de interconexão do tipo *crossbar* com tecnologia ótica já estar em desenvolvimento, o tempo necessário para comutar linhas de sinais óticos ainda é lento quando comparado com a tecnologia eletrônica atual. Entretanto, avanços nesse sentido podem viabilizar no futuro redes de grandes dimensões [VAR87].

### 2.2.3. Multicomputadores Comerciais

O aparecimento de multicomputadores comerciais demonstra o interesse no paradigma de trocas de mensagens para um ambiente de processamento paralelo. Sistemas classificados como pequenos e médios (até 1000 nós), dentro dos limites da tecnologia atual, usam normalmente as redes de interconexão estática.

A história dos multicomputadores comerciais inicia em 1985 com o Cosmic Cube do Departamento de Ciências da Computação CalTech no Instituto de Tecnologia da Califórnia. Essa máquina utilizava processadores Intel 8086/8087 com memória privativa conectados através de uma rede de interconexão estática do tipo hipercubo.

Seguindo o sucesso do Cosmic Cube, outras empresas rapidamente iniciaram a venda de multicomputadores configurados como hipercubos. Assim, a Intel desenvolveu o iPSC e a Ametek desenvolveu o System/14, ambos baseados no grupo de processadores Intel 80286/80287.

Tanto iPSC como o System/14 usam um microprocessador existente agrupado com lógica adicional para gerir a comunicação. Entretanto, a comunicação pode ser integrada ao próprio processador como é o caso do Transputer da Inmos onde um único *chip* contém um processador de 32 bits, 2K bytes de memória, interface para memória externa de 32 bits e quatro canais de comunicação *full-duplex*. Conectando-se Transputers, pode-se construir grandes multicomputadores com mínimo suporte adicional de *hardware*. Utilizando o Transputer como bloco de construção, a empresa Floating Point Systems desenvolveu os hipercubos chamados T Series.

## 2.3. A Arquitetura do Nó //

O projeto Nó // [COR93] em desenvolvimento no Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina, pelos grupos de pesquisa em arquitetura de computadores, sistemas operacionais e linguagens de programação, objetiva a concepção de um ambiente completo para programação paralela.

### 2.3.1. Configuração geral

A arquitetura da máquina paralela projetada é a de um multicomputador (figura 8 [CAM95]) que consta de um conjunto de nós conectados entre si por intermédio de dois dispositivos distintos: um comutador de conexões de tipo *crossbar* e um barramento de serviço, caracterizando um sistema fracamente acoplado, típico para aplicações de granularidade média.

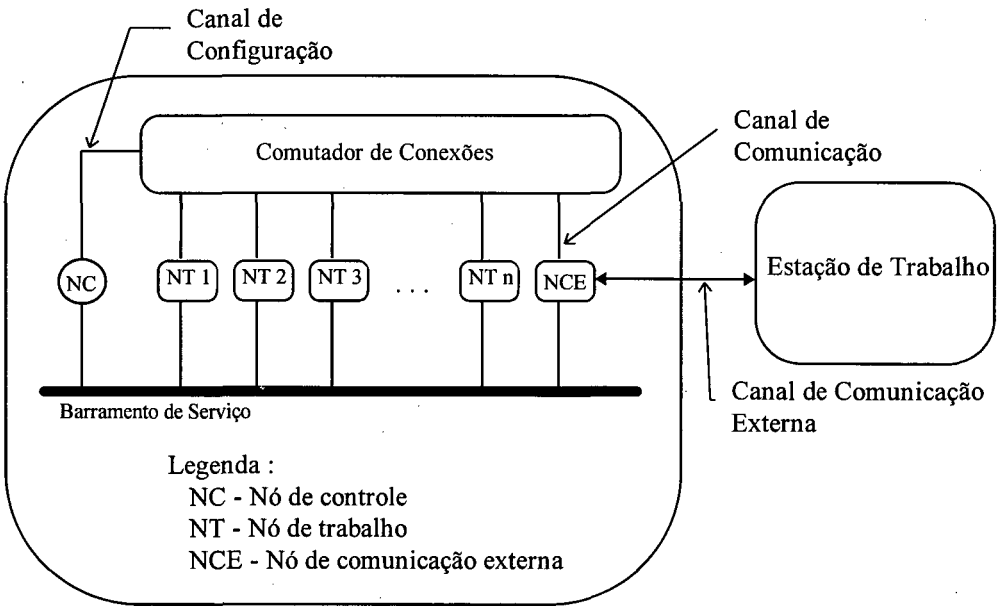


Figura 8 : A arquitetura do Nó //.

A maior parte das decisões para a concepção do ambiente para programação paralela se apoiam sobre as seguintes características do modelo de multicomputador considerado:

- Um número expressivo de nós (de 16 a 64).
- Um número moderado de canais por nó (4, por exemplo).
- Um comutador de conexões de grande dimensão (de 64x64 a 256x256).
- Um barramento de serviço de alto desempenho (talvez múltiplo).

Prevê-se ligar vários exemplares dessas máquinas como nós em redes locais (donde o nome Nó //, que deve ser lido como Nó Paralelo) mas inicialmente ela será tratada como uma máquina autônoma a ser utilizada como uma estação de trabalho individual de grande potência de processamento.

Quanto a sua flexibilidade, com a configuração de 4 canais por nó, é possível mapear, por exemplo, topologias estáticas como a grelha quadrada e o hipercubo de dimensão 4 nessa rede de interconexão dinâmica.

### 2.3.2. Protótipo inicial

O primeiro protótipo desse modelo de multicomputador deverá possuir :

- 32 nós constituídos de processadores i486 da Intel, com memória privativa de 2 Mbytes;
- 1 canal de comunicação por nó;
- Um *crossbar* COO4 da INMOS de dimensão 32x32;
- Um barramento de serviço.

Nesse protótipo, um dos nós de trabalho será conectado a uma estação de trabalho que fornecerá as ferramentas para o desenvolvimento do projeto (através de seus editores, compiladores, etc.) e os equipamentos periféricos. Ele deve ser entendido como uma simplificação do modelo geral descrito anteriormente que preserva suas qualidades básicas.

## √ Nós

Cada nó possui um processador, memória RAM privativa e uma pequena memória ROM, com seu respectivo microcódigo. Possui ainda um canal de comunicação bidirecional conectado ao *comutador de conexões*. O *barramento de serviço* funciona como um canal adicional de comunicação e controle compartilhado por todos os nós (figura 9 [MON95]).

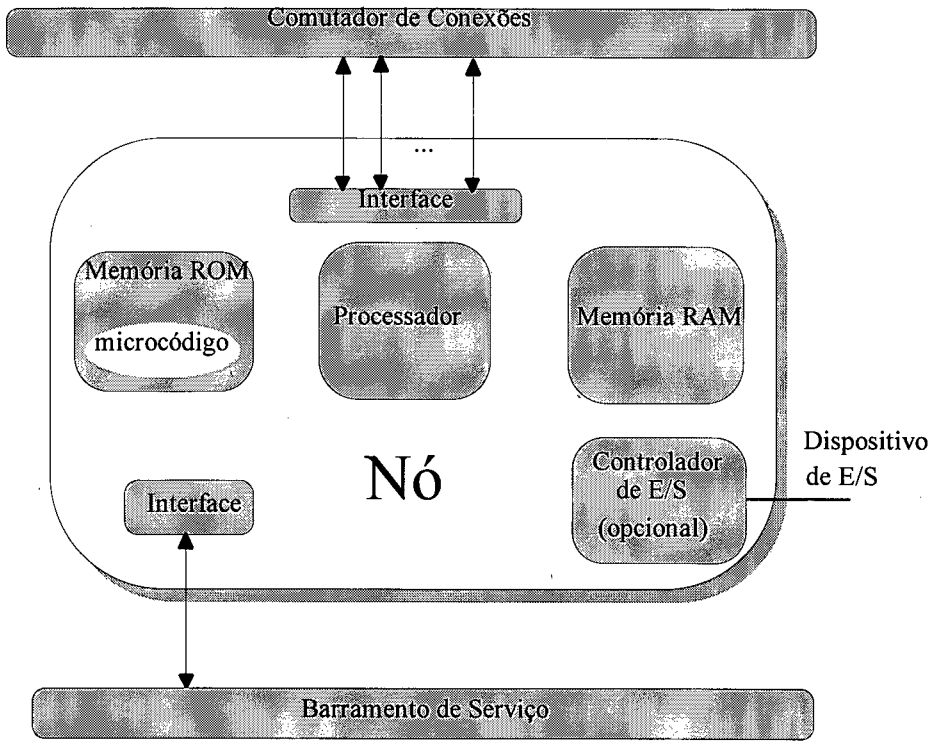


Figura 9 : A estrutura interna de um nó.

Dos 32 nós do protótipo do Nó //, 30 NTs (nós de trabalho) estão disponíveis para aplicações e 2 desempenham funções especiais : NC (nó de controle) e NCE (nó de comunicação externa).

O NC é o responsável pelo controle das conexões no Nó //, comandando o comutador de conexões conforme os pedidos dos NTs, através do canal de configuração do C004. Os pedidos, bem como os resultados, são transmitidos através do barramento de serviço.

O NCE efetua a comunicação externa ao Nó //. Possui, além do canal ligado ao *crossbar* e ao barramento de serviço, um canal de comunicação externo ligado à estação de trabalho. Através desse canal, a estação de trabalho efetua a carga inicial do Nó //.

## √ Canais

Os canais utilizados serão implementados através de *link adapters* C011 da INMOS. Cada canal é formado por duas linhas unidirecionais (uma em cada direção, permitindo transmissões do tipo *full-duplex*), bipontuais, operando a uma velocidade de 20 Mbits/sec.

## √ O comutador de conexões

O comutador de conexões do Nó // será do tipo *crossbar* C004 da INMOS de dimensão 32x32, próprio para a construção de redes de interconexão dinâmicas, provendo conexões entre canais do tipo *link adapters* C011. Através do C004, é possível estabelecer conexão entre quaisquer canais, sendo possível manter até 16 conexões bidirecionais simultâneas.

Além dos 32 canais utilizados para comunicação entre os processadores, o C004 possui um canal especial, chamado de canal de configuração, que no Nó // estará conectado ao nó de controle (NC), usado para programação do *crossbar*.

Basicamente, os comandos reconhecidos pelo C004 são :

- √ Conectar um par de canais;
- √ Desconectar um par de canais;

- √ Obter o número do canal conectado a um determinado canal;
- √ Inicializar o *crossbar*, colocando-o em um estado onde todas as conexões estão desfeitas.

## √ O barramento de serviço

O barramento de serviço é um barramento de 16 bits, bidirecional, altamente confiável, compartilhado por todos os processadores, que servirá para a transmissão de pequenas mensagens entre o NC e os NTs, sendo de uso tão somente do *software* de controle da máquina, com o objetivo de transportar requisições de serviço dos NTs ao NC.

Desse modo, este barramento não servirá para comunicação entre NTs, que deverão interagir por meio do *crossbar*. Essa restrição ao uso do barramento de serviço será imposta pela interface ao barramento serviço, que será de dois tipos :

- √ A interface dos nós de trabalho (NT), que poderá enviar/receber mensagens somente para/do NC, sendo utilizado pelos NTs para transmissão de pedidos ao NC, e recepção dos resultados das operações;
- √ A interface do NC, que poderá enviar/receber mensagens para/de um NT qualquer, sendo utilizado pelo NC para recepção dos pedidos dos NTs e envio dos resultados das operações.

A utilização combinada do barramento de serviço e do comutador de conexões permite considerar o estabelecimento por demanda (dinâmico) de canais para trocas de mensagens diretas entre nós como uma alternativa em relação às redes estáticas. Isso porque a comunicação pelo barramento de serviço pode ser usada como um meio confiável para a transmissão de requisições de conexão e desconexão de canais e o estabelecimento de uma conexão individual através do comutador de conexões é uma operação muito rápida que pode ser efetuada a qualquer momento sem afetar outras conexões já existentes. Dessa forma, canais de comunicação são estabelecidos à medida de necessidade dos algoritmos, evoluindo juntamente com a execução dos programas paralelos.

## √ A inicialização

O processo de inicialização do Nó // é suportado por um conjunto de ROMs com programas especiais. Cada nó possui uma ROM, cujo código será executado no momento de sua inicialização, sendo o responsável pela carga do nó. Este código pode ser de três tipos, variando de acordo com a função de cada nó.

O nó NCE é o responsável por receber todo o código de inicialização do Nó //. Este código é recebido por um canal de comunicação externo, ligado à uma estação hospedeira. O código da ROM do nó NCE segue o seguinte algoritmo :

- √ Espera pelo canal externo o tamanho do programa a ser carregado no NCE;
- √ Recebe o programa, carregando-o em determinada posição de memória;
- √ Executa o programa recebido.

Todos os NTs possuem a mesma ROM, que executa o seguinte algoritmo:

- √ Espera pelo canal ligado ao *crossbar* o tamanho do programa a ser carregado;
- √ Recebe o programa, carregando-o em determinada posição de memória;
- √ Executa o programa recebido.

O NC é equipado com uma ROM, que contém todo o código necessário para o seu funcionamento (serviços de conexão e alocação). Desse modo, o NC executa um código imutável, não necessitando de carga através dos canais ou de outros dispositivos.

O estado do Nó // após a etapa de inicialização é o seguinte :

- √ O nó NCE espera, pelo canal externo, o programa para carregar o sistema.
- √ O NC espera, pelo barramento de serviço, pedidos de conexão ou alocação.
- √ Os NTs esperam, pelos canais ligados ao *crossbar*, os programas de sistema e das aplicações.

O multicomputador concebido para compor o ambiente de programação paralela proposto concretiza alguns aspectos expressos em 1979, por Howard Jay Siegel [SIE79] : "sistemas de processamento paralelo reconfiguráveis tornam-se cada vez mais úteis na medida que diminui o custo de *hardware* e aumenta o conhecimento sobre exploração do paralelismo em tarefas. Além disso, redes de interconexão devem ser reestruturáveis sob controle de *software* com controle da rede acessível quando necessário".

# 3. Linguagens para programação paralela

Um **modelo de programação** paralela é um paradigma para a expressão de processos e das interações entre eles [AND91]. Os dois modelos gerais de programação paralela são :

- √ compartilhamento de memória : os processos de um programa paralelo cooperam utilizando dados compartilhados;
- √ troca de mensagens : os processos de um programa paralelo cooperam utilizando dados transferidos entre eles na forma de mensagens.

O tipo de máquina tem forte influência sobre esses modelos. Assim, multiprocessadores induzem a utilização de cooperação por compartilhamento de memória e multicomputadores impõem a utilização de cooperação por trocas de mensagens.

Um modelo de programação paralela se concretiza através da concepção de uma nova linguagem (como Occam, Joyce e SuperPascal), por extensões incorporadas a uma linguagem clássica (como C Concorrente) ou através de uma linguagem clássica enriquecida pela utilização de bibliotecas paralelas (como C com bibliotecas MPI ou PVM [GEI94] ).

Novas linguagens de programação vêm sendo desenvolvidas para expressar processos concorrentes cujos objetivos vão desde aplicações científicas até a construção de sistemas operacionais [HAN93a]. As linguagens paralelas necessitam do suporte de um sistema operacional fisicamente distribuído sobre os vários processadores para gerência do paralelismo, da sincronização e da comunicação entre processos.

## 3.1. Paralelismo

Os **processos** podem conter um único ou vários **fluxos de controle**. Vários fluxos de controle podem simplificar a programação de diversas aplicações pela combinação de execução seqüencial com chamadas de sistema bloqueantes [TAN92].

O paralelismo pode ser expresso de forma explícita e implícita. Com o **paralelismo explícito,** o programador é responsável pela expressão do paralelismo no seu programa, representando processos e suas interações. O **paralelismo implícito**, ao contrário, consiste na produção de código paralelo extraído pelo compilador a partir de um programa seqüencial.

Utilizando paralelismo explícito, o programador pode produzir programas paralelos mais eficientes por ter conhecimento da aplicação, embora o uso de construções paralelas acrescente um grau de dificuldade para a produção e correção de programas. Com o uso do paralelismo implícito, os usuários podem escrever seus programas em linguagens de programação seqüenciais e a análise do programa fonte para encontrar partes que podem ser executadas em paralelo é de responsabilidade do compilador.

O **processamento** paralelo de programas em um multicomputador requer a utilização de técnicas de decomposição [HWA93] para dividir, mapear e garantir o equilíbrio da carga computacional e estruturas de dados.

## 3.2. Sincronização

A sincronização estabelece restrições na ordem das operações dos processos interativos. Considerando-se duas operações pertencentes a processos diferentes, se alguma sincronização é necessária entre elas então, certamente, ela se enquadra em um dos dois tipos básicos :

- √ **as operações não devem ser executadas simultaneamente** : sincronização necessária para implementar acesso exclusivo a recursos compartilhados;

√ **uma operação deve ser executada após o término da outra** : sincronização necessária quando os processos cooperam entre si executando etapas seqüenciais de um mesmo serviço.

Os modelos de programação paralela influenciam os mecanismos de sincronização que podem ser utilizados pelas linguagens de programação. O modelo de compartilhamento de memória possibilita o uso de mecanismos de sincronização como **semáforos** e **monitores**. No modelo de **troca de mensagens**, os mecanismos de sincronização estão implicitamente associados aos problemas de comunicação.

## 3.3. Comunicação por troca de mensagens

Mecanismos de comunicação por troca de mensagens são necessários para prover o transporte de dados entre processos. Mensagens fluem entre processos através de canais de comunicação e seguem determinados padrões de interação. Um canal de comunicação é uma abstração que fornece um caminho para as trocas de mensagens [AND91]. O modelo de programação paralela por troca de mensagens utiliza primitivas para o envio e recepção de mensagens.

Para implementar canais de comunicação as seguintes características são importantes [PET85] :

√ **capacidade** : Determina se o canal possui fila de mensagens e qual o seu comprimento.

⇒ nula - não possui fila;

⇒ limitada - possui fila com comprimento máximo definido;

⇒ ilimitada - possui fila com comprimento potencialmente infinito.

√ **sentido** : Caracteriza a direção do fluxo das mensagens.

⇒ unidirecional - as mensagens podem ser transferidas em um único sentido entre dois processos;

⇒ bidirecional - as mensagens podem ser transferidas em ambos sentidos entre dois processos.

√ **acesso** : Caracteriza o número de processos que podem enviar ou receber mensagens através do canal.

⇒ exclusivo - o canal está associado a exatamente dois processos;

⇒ compartilhado - o canal pode estar associado a mais de dois processos.

√ **mensagens** :

⇒ tamanho - as mensagens podem ter tamanho fixo ou variável;

⇒ tipos de dados - os tipos de dados contidos nas mensagens podem ser considerados ou desconsiderados.

√ **designação** : Caracteriza a forma de identificar os processos envolvidos na comunicação.

⇒ direta - Designação direta ocorre quando cada processo que deseja enviar ou receber uma mensagem deve identificar explicitamente o receptor ou emissor na comunicação. O canal de comunicação é exclusivo para cada par de processos e é estabelecido de forma implícita com sentido bidirecional. A designação direta subdivide-se em simétrica ou assimétrica. Designação simétrica ocorre quando os processos envolvidos se identificam mutuamente. Na designação assimétrica, apenas o emissor designa o receptor que não precisa conhecer o emissor.

⇒ indireta - Na designação indireta o envio ou recepção de uma mensagem deve identificar o canal para onde as mensagens são enviadas ou de onde são recebidas. O canal pode estar associado a mais de dois processos e essa associação é feita de forma explícita com sentido unidirecional ou bidirecional. O canal de comunicação na designação indireta também é conhecido como caixa-postal.

- √ **criação e destruição de canais** : Os canais de comunicação, assim como os processos, podem ser criados de forma estática ou dinâmica em uma linguagem paralela. Na criação estática os canais são conhecidos em tempo de compilação e existem durante toda execução do programa paralelo. Na criação dinâmica eles só são conhecidos no decorrer da execução e podem ser criados e destruídos através de primitivas específicas.

- √ **propriedade** : Os canais podem pertencer aos seus processos criadores ou ao sistema operacional. Quando o canal pertence a um processo ele existe enquanto o seu criador existir. Se o canal pertence ao sistema operacional, privilégios de posse podem ser atribuídos a processos através de chamadas de sistema apropriadas.

- √ **ligação** : Para ocorrer a comunicação entre processos é necessário que eles estejam ligados. Ligação entre processos significa que os processos envolvidos na comunicação conhecem a identificação dos canais de comunicação que estão sendo utilizados [LIM94].

### 3.3.1. Mecanismos de comunicação

Os mecanismos de comunicação provêm trocas de mensagens entre processos que diferem conforme o modelo de programação da linguagem e a maneira de agrupar as características dos canais de comunicação.

- √ ***Comunicação síncrona*** - Em uma comunicação síncrona, o canal serve ao transporte de mensagens entre os dois processos envolvidos. Os canais não possuem capacidade de armazenamento e as comunicações requerem a participação simultânea dos processos envolvidos. Esse tipo de mecanismo também provê a sincronização dos processos.

√ ***Comunicação assíncrona*** - Em uma comunicação assíncrona, o canal serve ao transporte bidirecional de mensagens entre os processos envolvidos. Os canais possuem capacidade de armazenamento e as comunicações não requerem a participação simultânea dos processos envolvidos.

## 3.3.2. Disciplinas de comunicação

Disciplinas de comunicação são definidas como padrões de trocas de mensagens entre dois processos [COR93]. Cada tipo de disciplina de comunicação pode ser útil para resolver problemas de programação específicos. Embora possam ser definidos vários tipos de disciplinas, considera-se a seguir apenas duas : produtor-consumidor e cliente-servidor.

√ ***Disciplina 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 de tipo produtor-consumidor (figura 10).

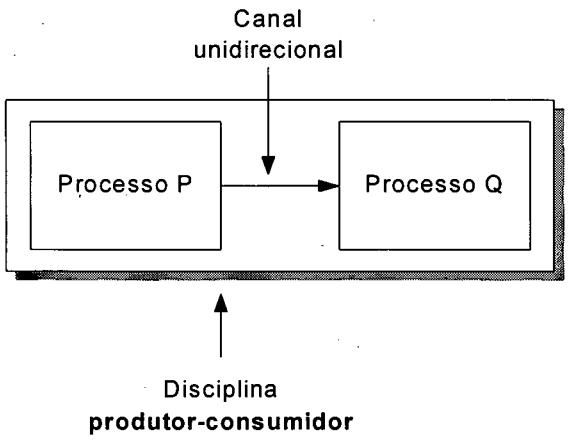


Figura 10 : Disciplina produtor-consumidor.

Com a utilização da disciplina de comunicação produtor-consumidor pode-se construir um ***pipeline*** de processos (figura 11).

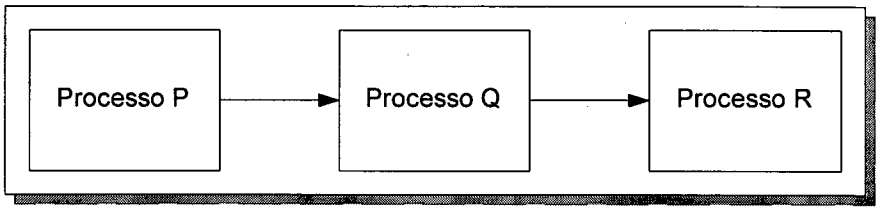


Figura 11 : *Pipeline* de processos.

√ ***Disciplina cliente-servidor***

Um processo que possui um canal bidirecional através do qual executa uma seqüência de envio de uma mensagem e recepção de outra é chamado cliente. Um processo que possui um canal bidirecional através do qual executa uma seqüência de recepção de uma mensagem e envio de outra é chamado de servidor. Quando o canal bidirecional de um cliente é conectado ao canal bidirecional de um servidor, desenvolve-se entre eles uma disciplina de comunicação de tipo cliente-servidor [COR93] (figura 12).

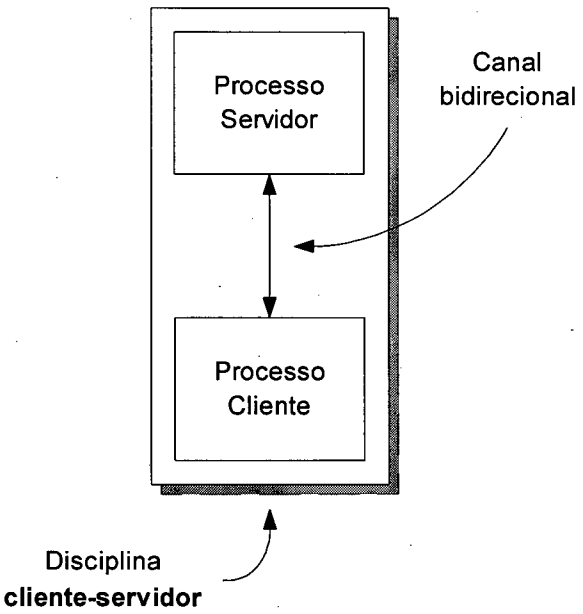


Figura 12 : Disciplina cliente-servidor.

Em geral um servidor atende vários clientes, o que pode ser observado na figura 13.

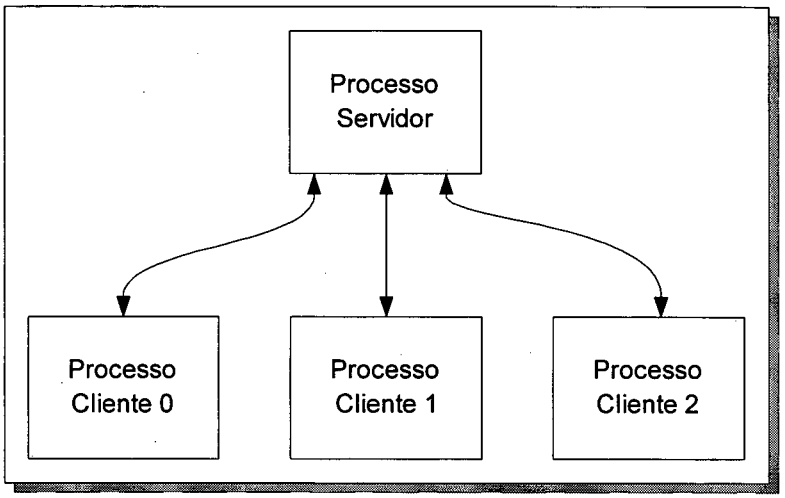


Figura 13 : Processo servidor com vários clientes.

## 3.4. Ambientes de programação paralela

Um **ambiente de programação paralela** é concretizado por ferramentas para desenvolver e executar programas. Distingue-se dois componentes básicos em um ambiente de programação paralela [COS93] : ambiente de produção de programas e ambiente de execução de programas.

- √ **Ambiente de produção de programas** : É composto por uma interface de usuário onde existe ao menos um editor de textos e um compilador de uma linguagem paralela.
- √ **Ambiente de execução de programas** : É composto pelos serviços exigidos para suportar o modelo de programação paralela : mapeamento, sincronização, comunicação, gerência de memória e gerência de processos.

## 3.5. Linguagens paralelas

As linguagens de programação paralela apresentadas neste trabalho são CSP, Occam, Joyce e SuperPascal, todas orientadas à troca de mensagens. Todas essas linguagens geram programas paralelos na forma de redes de processos que se comunicam exclusivamente por troca de mensagens. O interesse pelo estudo dessas linguagens deriva da semelhança entre as redes lógicas dos programas e as redes físicas dos multicomputadores.

Para ilustrar a programação em cada uma delas, utiliza-se o exemplo do algoritmo de Miller-Rabin, descrito em [HAN94a], utilizado para testar se um número é primo. O algoritmo executa $p$ testes probabilísticos do mesmo número inteiro simultaneamente através de $p$ processos. Cada teste prova que o número é composto (não primo) ou falha sem provar nada.

O algoritmo executa operações aritméticas em números naturais de comprimento variável representados por vetores de $w$ posições (mais um dígito de *overflow*). Cada teste inicializa um gerador de números aleatórios com uma semente distinta. O processamento paralelo é organizado como uma rede em anel formada por um processo *master* e um *pipeline* de processos conectados por canais de comunicação (figura 14).

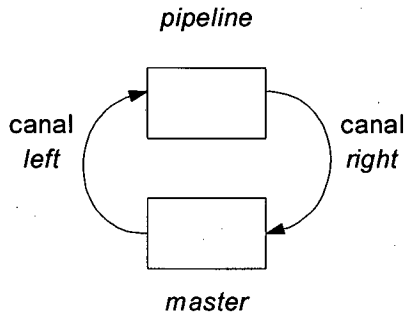


Figura 14 : Rede de processos em anel.

O *pipeline* consiste de $p$ processos *Nodes* paralelos conectados por $p+1$ canais de comunicação (figura 15).

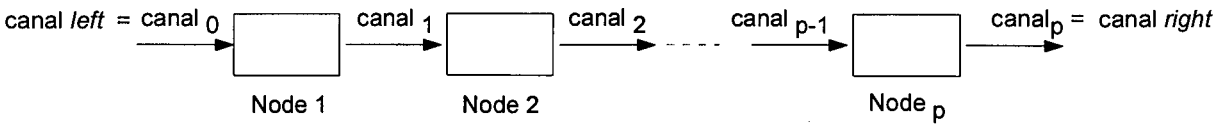


Figura 15 : *Pipeline*.

O processo *master* envia um número através do *pipeline* e recebe *p* valores lógicos do *pipeline* (os valores lógicos são os resultados de *p* testes independentes executados em paralelo pelos processos *Nodes*).

### 3.5.1. CSP

"O projeto de uma linguagem de programação sempre é um compromisso no qual o bom projetista deve levar em consideração o nível de abstração desejado, a arquitetura da máquina destino e as aplicações propostas" [Hoare, citado por Brinch Hansen em HAN93a, p. 49].

CSP (***Communicating Sequential Processes***) é uma linguagem baseada no modelo de troca de mensagens que foi proposta por Hoare em 1978 [HOA78]. Embora tendo influenciado o projeto de outras linguagens paralelas, CSP é essencialmente um projeto de linguagem sem implementação. As principais motivações para o seu desenvolvimento foram :

- √ Introduzir comandos básicos de entrada e saída na própria linguagem de programação para prover comunicação entre processos paralelos;
- √ Introduzir um comando paralelo para especificar execução paralela de processos seqüenciais;
- √ Prover comunicação síncrona direta entre processos;
- √ Utilizar os comandos guardados de Dijkstra associados aos comandos de entrada para introduzir e controlar o não-determinismo.

A seguir, tem-se o exemplo do algoritmo de Miller-Rabin em CSP.

```
RING = [ pipeline :: PIPELINE || master :: MASTER ]

PIPELINE = [ Node( 1 ) || Node( i : 2..p-1 ) :: NODE || Node( p ) ]

Node( 1 ) = [ a : integer;
              composite : boolean;

              master ? a;
              Node( 2 ) ! a;
              ... executa procedimento test ( a, i, composite ) ...
              Node( 2 ) ! composite;
        ]

NODE = [      a : integer;
              composite : boolean;

              Node( i ) ? a;
              [  i < p → Node( i+1 ) ! a  ];
              ... executa procedimento test ( a, i, composite ) ...
              Node( i+1 ) ! composite;
              j : integer; j := 1;
              * [  j <= i-1 → Node( i ) ? composite;
                   Node( i+1 ) ! composite;
                   j := j + 1 ];
        ]

Node( p ) = [ a : integer;
              composite : boolean;

              Node ( p ) ? a;
              master ! a;
              ... executa procedimento test ( a, i, composite ) ...
              master ! composite;
              j : integer; j := 1;
              * [  j <= p-1 → Node( p ) ? composite;
                   master  ! composite;
                   j := j + 1 ];
        ]
```

```
MASTER = [ a : integer;
           composite : boolean;
           ... a := valor...

           Node( 1 ) ! a;
           prime : boolean;
           prime := true;
           i : integer; i := 1;
           * [  prime; i <= p → Node( p ) ? composite;
                i := i + 1;
                [  composite → prime := false ]
             ]
         ]
```

Algoritmo 1 : Miller-Rabin em CSP.

### *Paralelismo*

Um processo consiste de um nome, variáveis locais e uma lista de comandos seqüenciais. Comandos simples são subdivididos em : atribuição, entrada e saída. Comandos estruturados são subdivididos em : paralelos, alternativos e repetitivos. Comandos alternativos e repetitivos utilizam os comandos guardados de Dijkstra associados aos comandos de entrada.

CSP utiliza o paralelismo explícito fornecendo um único comando paralelo ( || ) para criar um número fixo de processos paralelos. No algoritmo 1, por exemplo, tem-se os seguintes processos: pipeline, master e $p$ processos Node. A notação Node( i : 2..p-1 ) :: NODE equivale a $p$-2 processos Node.

A notação expressãoLógica; comandoDeEntrada → listaDeComandos representa um comando guardado. Comandos guardados são compostos por expressões lógicas e comandos de entrada, seguidos de uma lista de comandos.

A notação [ comandoGuardado □ comandoGuardado □ ... ] representa um comando alternativo. O término de um comando alternativo ocorre após a execução de um único comando guardado. A notação * comandoAlternativo representa um comando repetitivo. O comando repetitivo representa a execução repetida do comando alternativo componente. O término de um comando repetitivo ocorre quando todas as guardas falham.

### *Comunicação*

Um programa paralelo CSP contém uma coleção de processos que evoluem concorrentemente e se comunicam através de canais.

Os processos se comunicam através de comandos de entrada ( ? ) e de saída ( ! ). O processo emissor designa o processo receptor e fornece o valor a ser enviado. O processo receptor designa o processo emissor e fornece a variável para a qual o valor será atribuído. O mecanismo de comunicação utilizado é a comunicação síncrona direta e a associação dos processos com os canais é feita de forma implícita.

Um mesmo canal pode transportar diferentes tipos de dados. No algoritmo 1, o processo Node( i ) executa o comando de saída Node( i ) ! a para enviar uma mensagem de tipo integer e o comando de saída Node( i ) ! composite para enviar uma mensagem de tipo boolean.

Além da transferência de dados simples exemplificada no algoritmo 1, os canais podem transportar dados estruturados utilizando-se construtores. Por exemplo, no comando Node( i ) ! msg(a, composite), a mensagem transportada pelo canal contém dois tipos diferentes de dados associados pelo construtor msg : um integer e um boolean. Um construtor vazio como x( ), pode ser utilizado apenas para sincronizar dois processos sem transferir dados.

### *Ambiente de programação paralela*

Como CSP é essencialmente um projeto de linguagem, não possui nenhum ambiente de programação efetivo.

### *Resumo das características de CSP*

Pode-se comentar que CSP possui as seguintes limitações :

√ Processos definidos estaticamente;

√ Comunicação por troca de mensagens com designação direta simétrica;

√ Falta de recursividade;

√ Guardas de saída não admitidos nos comandos guardados.

A tabela 1 resume as características dos canais em CSP.

Tabela 1 - Características dos canais em CSP.

| Característica | Tipo |
|---|---|
| capacidade | zero |
| sentido | bidirecional |
| acesso | exclusivo |
| mensagens | tamanho variável |
| designação | direta simétrica |
| criação e destruição | estática |
| propriedade | processos são proprietários |

### 3.5.2. Occam

Occam é uma linguagem derivada de CSP, projetada para a programação do Transputer da Inmos, tendo sua primeira versão aparecido em 1984 [INM84]. O Transputer possui grande capacidade de processamento e comunicação fornecendo rápido chaveamento de processos e atendimento a interrupções. A relação do *hardware* com a linguagem é muito próxima pois funções da linguagem possuem operadores correspondentes no *hardware*.

Occam tem aplicações para processamento de sinais, processamento de imagens, controle de processos, simulação, processamento em tempo real e análise numérica [BAL89]. Suas características estáticas fornecem alta eficiência para esses tipos de aplicações. Occam2 é uma extensão da linguagem Occam que surgiu em 1988 [BUR88].

A seguir, tem-se o exemplo do algoritmo de Miller-Rabin em Occam.

```
PROTOCOL channel IS BOOL, INT :
INT a :
BOOL prime :
CHAN OF channel left, right :
PROC Node ( INT i, CHAN OF channel left, right )
  INT a, j :
  BOOL composite :
  SEQ
    left ? a
    IF i < p
```

```
            right ! a
          TRUE
            SKIP :
        ... executa procedimento test ( a, i, composite ) ...
        right ! composite
        SEQ j = 1 FOR i - 1
          SEQ
            left ? composite
            right ! composite
:
PROC pipeline ( CHAN OF channel left, right )
  [p+1] CHAN OF channel c :
  PAR
    Node( 1, left, c[ 1 ] )
    Node( p, c[ p ], right )
    PAR i = 2 FOR p-1
      Node( i, c[ i-1 ], c[ i ] )
:
PROC master ( INT a, BOOL prime, CHAN OF channel left, right )
  INT i :
  BOOL composite :
  SEQ
    left ! a
    prime := TRUE
    SEQ i = 1 FOR p
      SEQ
        right ? composite
        IF composite = TRUE
          prime := FALSE
        TRUE
          SKIP
:
PAR
  pipeline( left, right )
  master( a, prime, left, right )
:
```

Algoritmo 2 : Miller-Rabin em Occam.

### *Paralelismo*

Toda instrução Occam é considerada um processo. Occam possui cinco processos primitivos : processos de atribuição ( := ), de entrada ( ? ), de saída ( ! ), nulo (SKIP) e de parada (STOP). Os processos primitivos podem ser combinados para expressar comportamentos mais complexos através dos construtores da linguagem. Os construtores de Occam são : SEQ, PAR, WHILE, IF, CASE e ALT.

Compete ao programador indicar explicitamente se processos serão combinados em seqüência através do construtor SEQ ou em paralelo através do construtor PAR.

É possível aplicar replicadores FOR aos construtores SEQ, PAR, ALT, IF com o objetivo de replicar o processo componente. Um exemplo de aplicação de replicador pode ser visto no algoritmo 2, no procedimento pipeline, para produzir $p$-2 processos Node.

### *Comunicação*

Um programa paralelo Occam contém uma coleção de processos que evoluem concorrentemente e se comunicam através de canais.

O mecanismo de comunicação utilizado envolve trocas de mensagens síncronas indiretas. Esse mecanismo de comunicação é implementado no próprio *hardware* dos Transputers de forma altamente eficiente. Os canais lógicos de Occam podem ser associados aos canais físicos de comunicação dos Transputers.

A declaração CHAN OF descreve canais pelos quais trafegam apenas dados tipados. A declaração PROTOCOL descreve o formato das mensagens compostas por grupos de tipos de dados. No algoritmo 2, os canais left e right são canais de comunicação através dos quais são transferidas mensagens com protocolo simples composto por dois tipos de dados : lógico (BOOL) e numérico (INT).

### *Ambiente de programação paralela*

O ambiente de programação paralela para Occam2, CSA Transputer Education Kit [CSA90], é um conjunto de ferramentas para possibilitar programação de Transputers compatíveis com o modelo T400. Ele permite que programas paralelos sejam desenvolvidos em máquinas hospedeiras para serem executados em um único Transputer ou em redes de Transputers.

A versão descrita do ambiente para a linguagem Occam2 executa sobre o sistema operacional DOS em um IBM-PC e é composto por :

- √ **Ambiente de produção de programas** : Esse ambiente é representado pelo compilador *occam*, ferramenta para a criação de bibliotecas de código chamada *ilibr*, ferramenta para a ligação de programas com bibliotecas e outros códigos compilados chamada *ilink*, ferramenta *iboot* que produz código executável para apenas um Transputer e a ferramenta *iconf* que produz código executável e dados de carga para uma configuração específica de uma rede de Transputers. O editor utilizado para escrever o programa Occam pode ser qualquer editor disponível sobre o DOS;
- √ **Ambiente de execução de programas** : Esse ambiente é representado pela ferramenta *iserver* que carrega os programas nos Transputers usando o sistema de arquivos da máquina hospedeira e fornece suporte em tempo de execução para interface com a máquina hospedeira.

### *Resumo das características de Occam*

A sintaxe de um programa Occam possui características especiais como as seguintes :

- √ cada processo primitivo, construtor e declaração devem ocupar uma linha;
- √ a linguagem impõe endentação como forma de determinar o início e fim de novos processos.

Pode-se comentar que Occam possui as seguintes limitações :

- √ Falta de recursividade para funções e procedimentos;

√ Vetores de processos devem ter tamanhos constantes;

√ Instruções simples devem ser escritas em linhas separadas;

√ Não define mecanismos de entrada e saída como parte da linguagem;

√ Não permite alocação dinâmica de variáveis.

A tabela 2 resume as características dos canais em Occam.

Tabela 2 - Características dos canais em Occam.

| Característica | Tipo |
|---|---|
| capacidade | zero |
| sentido | unidirecional |
| acesso | exclusivo |
| mensagens | tamanho variável |
| designação | indireta |
| criação e destruição | estática |
| propriedade | processos são proprietários |

### 3.5.3. Joyce

Joyce é uma linguagem derivada de CSP com notação baseada em Pascal. Ela foi projetada por Brinch Hansen em 1987 [HAN87] para o ensino de princípios e técnicas de programação distribuída. A remoção de várias restrições do CSP também motivou o seu desenvolvimento. Para isso, foi introduzido :

√ Comunicação síncrona com designação indireta;

√ Compartilhamento de canais;

√ Variáveis de tipo porta (*port*) para acesso aos canais;

√ Alfabetos de canal;

√ Recursividade;

√ Uso de instruções de saída nas guardas.

A seguir, tem-se o exemplo do algoritmo de Miller-Rabin em Joyce.

```
type number = array [ 0..w ] of integer;

const p = ... número de processadores ...
type channel = [ log(boolean), num(integer) ];

agent ring ( );
var a : number;
    prime : boolean;
    left, right : channel;

        agent Node ( i : integer; left, right : channel );
        var a : number;
            j : integer;
            composite : boolean;
        begin
          left ? num( a );
          if i < p then right ! num ( a );
          ... executa procedimento test ( a, i, composite ) ...
          right ! log(composite);
          j := 1;
          while j <= i - 1 do
            begin
              left ? log( composite );
              right ! log( composite );
              j := j + 1;
            end;
        end;

        agent pipeline ( left, right : channel );
        type row = array [ 1..p-1 ] of channel;
        var c : row;
            i : integer;
        begin
          i := 1;
          while i < p do
            begin
              +c[ i ];
               i := i + 1;
            end;
          Node(1, left, c[1]);
          Node(p, c[ p-1 ], right);
          i := 2;
          while i < p do
```

```
            begin
               Node( i, c[ i - 1 ], c[ i ] );
               i := i + 1;
            end;
      end;

      agent master ( a : number; prime : boolean; left, right : channel );
      var i : integer;
          composite : boolean;
      begin
         left ! num( a );
         prime := true;
         i := 1;
         while i <= p do
            begin
               right ? log( composite );
               if composite then
                 prime := false;
               i := i + 1;
            end;
      end;
begin
   +left;
   +right;
   a := ... valor do número a ser testado ...
   prime := true;
   pipeline (left, right);
   master (a, prime, left, right);
end;
```

Algoritmo 3 : Miller-Rabin em Joyce.

### *Paralelismo*

Um programa Joyce consiste de procedimentos que definem processos paralelos conhecidos como agentes. A execução de um programa origina uma árvore hierárquica de processos criados e destruídos de forma dinâmica a partir de um processo inicial único que é automaticamente criado quando um programa inicia. Os processos são criados com sintaxe semelhante a uma chamada de procedimento. Observa-se no algoritmo 3 a existência do processo inicial ring que cria dois processos filhos pipeline e master. O processo pipeline por sua vez cria $p$ processos Node. Têm-se dessa forma $p$+3 processos evoluindo em paralelo.

Como o número total de processos existentes é conhecido somente em tempo de execução, o mapeamento de processos em uma implementação do ambiente Joyce para uma máquina paralela deve ser feito de forma dinâmica.

### *Comunicação*

Um programa paralelo Joyce contém uma coleção de processos que evoluem concorrentemente e se comunicam através de canais. O mecanismo de comunicação utilizado é a comunicação síncrona indireta.

Processos se comunicam através de símbolos transmitidos por canais. Cada canal possui um alfabeto que define o conjunto fixo de símbolos disjuntos que ele pode transportar. Um símbolo pode transportar uma mensagem de um tipo fixo. Tanto o alfabeto de símbolos quanto os tipos que estes podem transportar são conhecidos na definição do canal. Um canal é considerado como um dispositivo de comunicação compartilhado porque pode ser utilizado por dois ou mais processos embora as comunicações ocorram entre dois processos de cada vez.

Para fazer referência ao canal, são utilizadas variáveis do tipo *port*. A definição `type TipoCanal = [ símb(integer) ]`, por exemplo, define um alfabeto de canal consistindo do símbolo `símb` que transporta uma mensagem de tipo `integer`. Um processo acessa um canal através de uma variável local do tipo *port*, como `c : TipoCanal`. Quando um processo executa a instrução `+c`, um novo canal com o alfabeto definido pelo tipo `TipoCanal` é criado e um ponteiro para o canal é atribuído à variável `c`.

Uma das estruturas de dados utilizadas para implementar canais é a fila de símbolos. Existe um par de filas para cada símbolo no alfabeto do canal : uma fila de processos emissores (fila de saída) do símbolo e uma fila de processos receptores (fila de entrada) do símbolo. As filas funcionam para registrar pedidos de comunicação entre os processos que compartilham o canal. A presença de duas filas com pedidos de comunicação por parte dos processos possibilita o transporte bidirecional de mensagens através do canal bem como o seu compartilhamento.

Dois processos devem executar instruções de entrada ( ? ) e saída ( ! ) para concretizar uma comunicação. No algoritmo 3, o processo **master** executa a instrução de saída **left ! num(a)** para enviar a mensagem e o processo **Node** executa a instrução **left ? num(a)** para receber a mensagem.

A instrução *poll* é semelhante à instrução alternativa de CSP. Entretanto, na linguagem Joyce, podem ser utilizados tanto guardas de entrada como de saída.

Para ilustrar a utilidade dos canais compartilhados, fez-se a seguinte alteração no algoritmo 3 : canais compartilhados **left** e **right** são utilizados para realizar a tarefa de comunicação dos **Nodes** com o processo **master**. Dessa forma, ao invés do valor da variável **composite** (*p* valores), que representa o resultado parcial feito por cada processo **Node**, ser transportado através de um **pipeline** até chegar ao processo **master**, são utilizados os canais compartilhados para realizar essa tarefa. Economiza-se assim *p*-1 canais de comunicação.

A nova situação é representada na figura 16 e o algoritmo correspondente é o algoritmo 4.

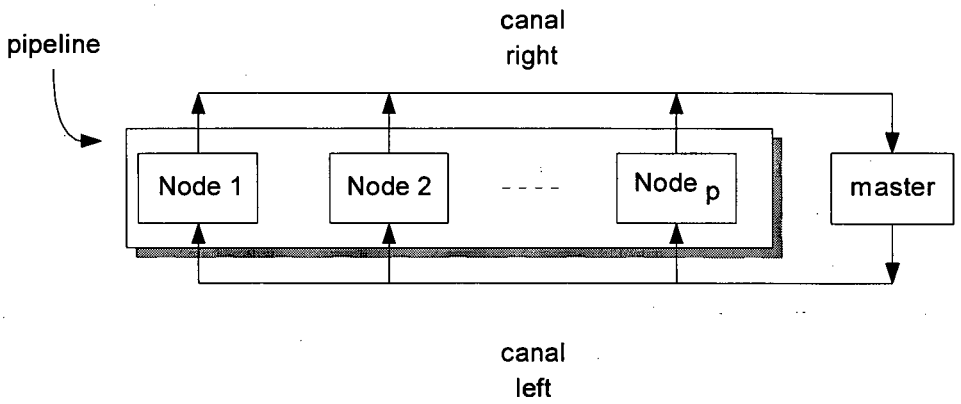


Figura 16 : Rede de processos com canais compartilhados.

```
type number = array [ 0..w ] of integer;

const p = ... número de processadores ...
type channel = [ log(boolean), num(integer) ];

agent ring ( );
var a : number;
    prime : boolean;
    left, right : channel;
```

```
        agent Node ( i : integer; left, right : channel );
        var a : number;
            j : integer;
            composite : boolean;
        begin
          left ? num( a );
          ... executa procedimento test ( a, i, composite ) ...
          right ! log(composite);
        end;
        agent pipeline ( left, right : channel );
        var i : integer;
        begin
          i := 1;
          while i <= p do
            begin
              Node( i, left, right );
              i := i + 1;
            end;
        end;
        agent master ( a : number; prime : boolean; left, right : channel );
        var i : integer;
            composite : boolean;
        begin
          prime := true;
          i := 1;
          while i <= p do
            begin
              left ! num( a );
              right ? log( composite );
              if composite then
                prime := false;
              i := i + 1;
            end;
        end;
begin
  +left;
  +right;
  a := ... valor do número a ser testado ...
  prime := true;
  pipeline (left, right);
  master (a, prime, left, right);
end;
```

Algoritmo 4 : Miller-Rabin em Joyce com canais compartilhados.

### *Ambiente de programação paralela*

Os ambientes de programação paralela construídos para a linguagem Joyce são três : o primeiro ambiente para IBM-PC [HAN87b], o segundo para um hipercubo [AND89] e o terceiro para um multiprocessador [HAN89b].

**Ambiente para IBM-PC**

O ambiente para IBM-PC é composto por :

√ **Ambiente de produção de programas** : Esse ambiente é representado pelo compilador Joyce que é um compilador de três passos escrito em Pascal. A compilação é dividida em análise léxica (*scanner*), análise sintática (*parser*) e a montagem do código objeto (*assembler*). Essas fases em conjunto compõem um compilador que gera código portável interpretado por um núcleo Joyce escrito em linguagem *Assembly*.

√ **Ambiente de execução de programas** : Esse ambiente é representado pelo núcleo Joyce que implementa a gerência de comunicação, de memória e de processos.

Esse ambiente para um processador único foi implementado com idéias simples e eficientes para solidificar os conceitos da linguagem.

**Ambiente para um multicomputador com rede de interconexão estática**

O multicomputador utilizado para esse ambiente foi o hipercubo binário iPSC da Intel. Cada nó dessa máquina é conectado aos vizinhos através de canais de comunicação Ethernet. Existe um nó gerente (*cube manager*) que é um microcomputador Intel 80286 utilizado para fornecer entrada/saída e ambiente de produção de programas.

√ **Ambiente de produção de programas** : Esse ambiente é representado pelo compilador Joyce que é um compilador de quatro passos escrito em C gerando código portável. O compilador é executado no nó gerente.

√ **Ambiente de execução de programas** : Esse ambiente é representado pelo núcleo Joyce composto por um interpretador escrito em *Assembly* e por rotinas de comunicação e escalonamento escritas em C. Cópias do código portável são carregadas pelo núcleo em cada nó do hipercubo. Também existe o módulo de interface com o usuário presente no nó gerente apenas quando um programa Joyce está sendo executado. A interface representa tarefa do sistema operacional sob o ponto de vista do programa.

Esse ambiente para um hipercubo mostrou-se ineficiente para a execução de programas Joyce principalmente porque a criação de processos e a comunicação entre eles produz redes dinâmicas que se adaptam com dificuldade à topologia estática do multicomputador.

**Ambiente para um multiprocessador**

O multiprocessador utilizado para esse ambiente foi o Encore Multimax 320 que possui 18 processadores NS32332 e um barramento compartilhado que conecta os processadores com a memória compartilhada de 128 Mbytes.

√ **Ambiente de produção de programas** : O compilador utilizado pelo multiprocessador é semelhante ao utilizado pelo ambiente para IBM-PC quanto a sua forma de compilação. Existe a interface Unix Umax 4.2, uma versão Multimax para o sistema Unix de Berkeley. Inicialmente, um usuário se comunica com um único processo Unix chamado processo mestre. Quando o usuário decide executar um programa Joyce em $p$ processadores, o processo mestre cria $p$ processos Unix adicionais conhecidos como processadores Joyce.

**Ambiente de execução de programas** : Esse ambiente é representado pelo núcleo para o multiprocessador. Esse núcleo, similar ao núcleo do IBM-PC, adiciona tarefas como equilíbrio de carga e *locks* (os *locks* são bloqueios para garantir a exclusão mútua no acesso às estruturas de dados compartilhadas do núcleo). Esse ambiente para multiprocessador foi uma tentativa para eliminar os problemas encontrados na implementação no hipercubo.

## *Resumo das características de Joyce*

Pode-se comentar que Joyce possui as seguintes limitações :

√ Um processo não pode acessar variáveis globais;

√ Uma mensagem não pode incluir referências a canais;

√ Dois processos não podem se comunicar através do *polling* de um mesmo canal.

A primeira simplificação é a única considerada realmente importante [HAN93a, p. 41].

A tabela 3 resume as características dos canais em Joyce.

Tabela 3 - Características dos canais em Joyce.

| Característica | Tipo |
|---|---|
| capacidade | zero |
| sentido | bidirecional |
| acesso | compartilhado |
| mensagens | tamanho variável |
| designação | indireta |
| criação e destruição | dinâmica em tempo de execução |
| propriedade | processos são proprietários |

### 3.5.4. SuperPascal

SuperPascal é uma linguagem que estende um subconjunto do Pascal com instruções determinísticas para criação de processos paralelos e para trocas de mensagens síncronas. As características de paralelismo são baseadas principalmente em Occam2 estendida pela inclusão de vetores de processos dinâmicos e processos paralelos recursivos. O projeto da linguagem SuperPascal utiliza ainda características de linguagens como CSP e Joyce, e foi motivado por dois objetivos [HAN94c] :

- √ Simplicidade : Criar uma linguagem de programação elegante para a computação científica paralela adicionando uma quantidade mínima de conceitos à linguagem Pascal;
- √ Segurança : Impor restrições adicionais nos conceitos de programação de Pascal para permitir que um compilador verifique que processos paralelos são disjuntos, i.e., que eles atualizam somente conjuntos disjuntos de variáveis.

A simplicidade foi atingida estendendo Pascal apenas com instruções para criação de processos e trocas de mensagens síncronas para comunicação entre eles. A segurança foi garantida pela imposição de restrições adicionais nos procedimentos e funções e omitindo algumas características do Pascal.

A seguir, tem-se o exemplo do algoritmo de Miller-Rabin em SuperPascal.

```
type number = array [ 0..w ] of integer;

const p = ... número de processadores ...
type channel = *( boolean, number );

procedure master ( a : number; var prime : boolean; left, right : channel );
var i : integer;
    composite : boolean;
begin
  send( left, a );
  prime := true;
  for i := 1 to p do
    begin
```

```
        receive( right, composite );
        if composite then
          prime := false;
      end;
end;

procedure node ( i : integer; left, right : channel );
var a : number;
      j : integer;
      composite : boolean;
begin
   receive ( left, a );
   if i < p then send ( right, a );
   ... executa procedimento test ( a, i, composite ) ...
   send ( right, composite );
   for j := 1 to i - 1 do
     begin
        receive ( left, composite );
        send ( right, composite );
      end;
end;

procedure pipeline ( left, right : channel );
type row = array [ 0..p ] of channel;
var c : row;
     i : integer;
begin
   c[ 0 ] := left;
   c[ p ] := right;
   for i := 1 to p - 1 do
     open( c[ i ] );
   forall i := 1 to p do
     node ( i, c[ i-1 ], c[ i ] );
end;

procedure ring ( a : number; var prime : boolean );
var left,
     right : channel;
begin
   open( left, right );
   parallel
      pipeline( left, right ) |
      master( a, prime, left, right )
   end
end;
```

Algoritmo 5 : Miller-Rabin em SuperPascal.

### *Paralelismo*

As características de paralelismo da linguagem SuperPascal são representadas pelas instruções *parallel* e *forall* para criação de processos paralelos. No algoritmo 5, por exemplo, o procedimento ring utiliza a instrução *parallel* para explicitar que os procedimentos pipeline e master são executados em paralelo. O procedimento pipeline utiliza a instrução *forall* para explicitar que *p* processos node são executados em paralelo.

Existe criação dinâmica de processos através da recursividade e da possibilidade do número de processos de uma instrução *forall* ser definido em tempo de execução. Dessa forma, o mapeamento de processos, em uma implementação do ambiente SuperPascal para uma máquina paralela, deve ser feito de forma dinâmica.

### *Comunicação*

Um programa SuperPascal contém uma coleção de processos que evoluem concorrentemente e se comunicam através de canais. O mecanismo de comunicação utilizado é a comunicação síncrona indireta.

Processos se comunicam através de valores chamados mensagens transmitidos por meio de entidades chamadas canais. Processos criam canais dinamicamente e os acessam utilizando variáveis que fazem referência aos canais. Uma vez criados, os canais existem até o término do programa.

Um canal suporta mensagens de tipos diferentes, o que pode ser observado no algoritmo 5, com a declaração type channel = *( boolean, number ) indicando que um canal desse tipo pode transportar um valor de tipo boolean ou um valor de tipo number.

Os procedimentos necessários para a troca de mensagens são : *open*, *send* e *receive*. *Open* é a instrução para criação de canais, *send* e *receive* são as instruções para o envio e recepção de mensagens, respectivamente.

Dois processos devem executar instruções de envio e recepção de mensagens para concretizar uma comunicação. No algoritmo 5, o processo master executa a instrução de envio de mensagens send( left, a ) e o processo node executa a instrução receive ( left, a ) para a recepção dessa mensagem.

### *Ambiente de programação paralela*

Foi construído um ambiente de programação paralela portável para a linguagem SuperPascal em uma estação de trabalho Sun sobre o Unix. Pode-se classificar o ambiente da seguinte forma :

- √ **Ambiente de produção de programas** : Esse ambiente é representado pelo compilador SuperPascal chamado *sc*.
- √ **Ambiente de execução de programas** : Esse ambiente é representado pelo interpretador chamado *sr* que é responsável pela execução de programas SuperPascal.

O ambiente SuperPascal é utilizado para desenvolver programas portáveis para problemas usuais na ciência da computação. Também é uma tentativa de simplificar a tarefa de programação para cientistas que, geralmente, estão mais preocupados com os resultados numéricos do que com o aprendizado de programação em um ambiente de programação paralela já que esse aprendizado costuma ser difícil [HAN94a].

### *Resumo das características de SuperPascal*

Pode-se comentar que SuperPascal possui as seguintes limitações :

- √ Parâmetros reais da chamada de procedimento ou função e variáveis globais utilizadas no seu corpo não podem ser sinônimos;
- √ Procedimentos e funções recursivas não podem usar variáveis globais;
- √ Funções não podem atualizar variáveis globais e não podem utilizar parâmetros por referência ou procedimentos como parâmetros;
- √ Procedimentos e funções não podem usar procedimentos e funções como parâmetros;

√ Declarações *forward* de procedimentos e funções não podem ser utilizadas;

√ Tipo *pointer* é omitido;

√ Instruções *goto* e *label* são omitidos.

Não existem conceitos como comandos guardados bem como não-determinismo.

A tabela 4 resume as características dos canais em SuperPascal.

Tabela 4 - Características dos canais em SuperPascal.

| Característica | Tipo |
|---|---|
| capacidade | zero |
| sentido | bidirecional |
| acesso | exclusivo |
| mensagens | tamanho variável |
| designação | indireta |
| criação e destruição | dinâmica em tempo de execução |
| propriedade | processos são proprietários |

## 3.6. Principais características das linguagens citadas

Todas as linguagens de programação paralela descritas são baseadas em troca de mensagens, sendo especialmente adequadas para aplicações em máquinas paralelas como multicomputadores.

CSP pode ser considerada o fundamento teórico de Occam, Joyce e SuperPascal no que diz respeito ao paralelismo e à comunicação síncrona.

Occam, por ser uma linguagem projetada especificamente para a programação de um tipo de *hardware* (Transputers), fornece alta eficiência na execução dos programas por possuir operadores em *hardware* para executar funções da linguagem de forma direta.

Joyce, mesmo sendo uma linguagem projetada para o ensino de programação distribuída, introduziu idéias novas como o compartilhamento de canais de comunicação síncronos.

SuperPascal, projetada para a elaboração de algoritmos científicos, é uma linguagem simples e segura, adequada para aplicações que seguem a disciplina produtor-consumidor.

## Características dos canais

Os canais de todas as linguagens descritas possuem capacidade zero, i.e., utilizam o mecanismo de comunicação síncrona. Os canais de CSP, Joyce e SuperPascal possuem sentido bidirecional para o tráfego das mensagens enquanto que os de Occam têm sentido unidirecional.

O acesso aos canais só é compartilhado na linguagem Joyce. Nas demais linguagens descritas eles são de acesso exclusivo dos processos por eles conectados.

O tamanho das mensagens é variável em todas as linguagens, i.e., podem ser transferidas mensagens de tipos diferentes através do mesmo canal.

A designação é direta simétrica em CSP e indireta nas demais linguagens descritas.

A criação e destruição de canais é estática em CSP e Occam, e dinâmica em Joyce e SuperPascal. Os canais pertencem aos processos em todas essas linguagens.

# 4. Ambiente de execução

Este capítulo descreve um simulador do multicomputador Nó // e o sistema operacional *Crux*, com micronúcleo distribuído e fundamentado no modelo cliente-servidor ([CAM95] e [MON95]).

O simulador do multicomputador Nó // e o sistema operacional *Crux* constituem o ambiente efetivo para a implementação da linguagem de programação SuperPascal proposta neste trabalho.

## 4.1. Simulador

Para permitir o desenvolvimento de todos os componentes do projeto Nó // (*hardware* e *software*) de forma simultânea, a equipe de sistemas operacionais optou por desenvolver um simulador para o multicomputador.

O simulador para o Nó // consiste basicamente de um programa para uma máquina monoprocessadora do tipo PC i486, que simula os elementos básicos do multicomputador : os nós, os canais de comunicação, as redes de interconexão (barramento de serviço, *crossbar*) e o processo de inicialização da máquina (ROMs).

Além disso, o simulador está conectado à uma estação de trabalho hospedeira através de uma linha serial RS232, onde deve executar a primeira versão do sistema de arquivos do *Crux*. O ambiente em que está inserido o simulador é mostrado na figura 17.

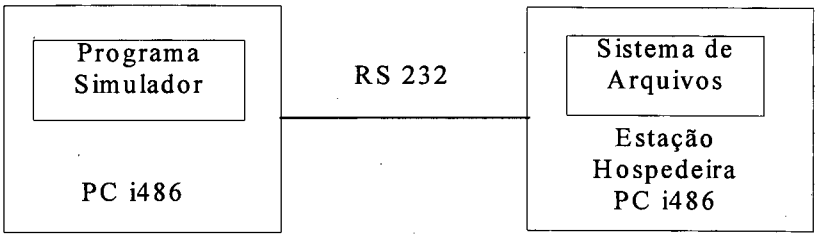


Figura 17 : Simulador do Nó // conectado a uma estação de trabalho.

Uma tarefa disparada na estação de trabalho pode gerar processos que podem ser alocados em diferentes nós do Nó //.

O simulador foi construído através da modificação dos seguintes componentes do sistema operacional XINU [COM84] :

√ **gerente de processos** : o paralelismo entre os nós é simulado através de processos;

√ **mecanismos de comunicação entre processos** : para simular a comunicação entre os nós, tanto pelos canais quanto pelo barramento de serviço;

√ **gerente de memória** : para simular as memórias privativas dos nós (figura 18);

√ **chamadas ao sistema** : para simular as chamadas à máquina e ao sistema operacional.

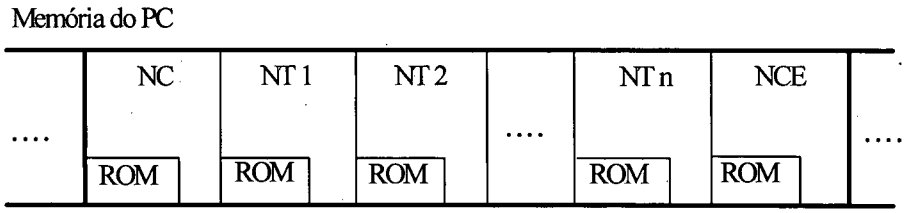


Figura 18 : Memória do PC dividida entre os nós simulados.

## 4.2. O Sistema *Crux*

Os sistemas operacionais são responsáveis pela administração dos recursos físicos e lógicos dos computadores. Eles podem ser vistos globalmente como montagens de sub-sistemas concebidos como servidores construídos em torno de cada tipo de recurso para aplicar métodos de utilização específicos. O **sistema operacional *Crux*** bem como cada um de seus sub-sistemas pode ser visto tanto como uma **rede de processos comunicantes** quanto como uma **hierarquia de camadas de *software*** [COR93]. O sistema operacional ***Crux***, descrito em [CAM95] e [MON95], desenvolvido sobre o simulador descrito em 4.1., apoia-se sobre essas duas idéias complementares.

### 4.2.1. Processos *Crux*

Processos *Crux* são programas em execução nos nós do Nó //. Um processo (figura 19) é constituído pelo seu espaço de endereçamento, dados, pilha, valores dos registradores e outras informações necessárias para sua execução [MON95].

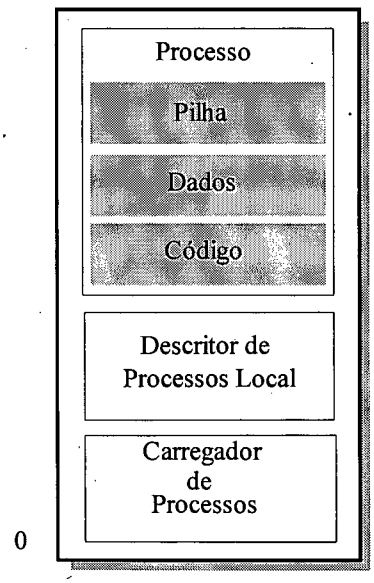


Figura 19 : Um processo colocado na memória de um nó.

Conforme pode ser observado na figura 19, existem três componentes principais descritos a seguir:

√ **Carregador de Processos**

Processos *Crux* são criados através da chamada *fork*, que é uma chamada de sistema compatível com UNIX. Neste caso, o processo criado é enviado 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.

√ **Descritor de Processos Local**

O sistema operacional possui uma estrutura na memória de cada nó própria para armazenar o contexto do processo. Essa estrutura, denominada *descritor de processos local*, armazena localmente diversas informações que servem para descrever o processo para o sistema operacional.

√ **Processo**

Um processo colocado na memória é composto de seu código, sua área de dados e pilha. A área de código consiste em uma seqüência de padrões de *bytes* que o processador interpreta como instruções de máquina. A área de dados corresponde à seção de dados inicializados e não inicializados existentes no arquivo executável. A área de pilha é criada e pode ser alterada dinamicamente durante a execução do programa.

Na implementação inicial do *Crux* utilizou-se um formato de programa que não necessita de relocação de seus endereços durante sua carga. Seu formato é semelhante aos dos programas .COM existentes nos sistemas MS-DOS. Essa característica permitiu uma grande simplificação nas tarefas dos carregadores de processos.

### 4.2.2. As camadas do sistema

A visão completa do sistema *Crux* como uma hierarquia de camadas é mostrada na figura 20.

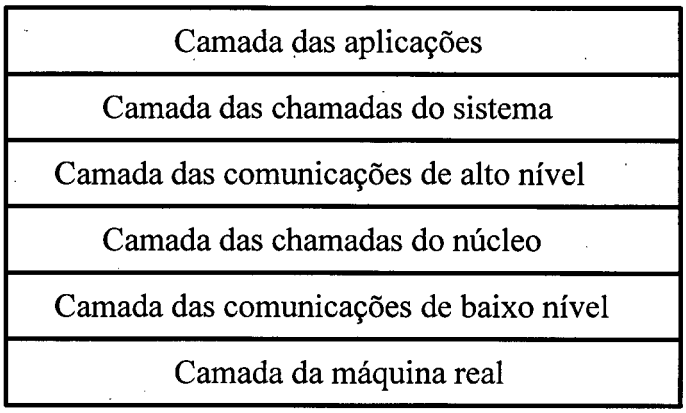


Figura 20 : As camadas do sistema.

As camadas das comunicações de baixo nível, das chamadas do núcleo e das comunicações de alto nível compõem o **micronúcleo distribuído** (figura 21) ou rede de serviços do núcleo do sistema operacional ***Crux***. A camada das chamadas de sistema é acessada através de procedimentos intermediários (*stubs*) relativos aos serviços habituais do sistema operacional e dos serviços de comunicação por troca de mensagens.

Essas camadas são examinadas a seguir em ordem descendente, com exceção da camada das aplicações (constituída das aplicações desenvolvidas sobre o ambiente proposto) e da camada da máquina real (constituída pelo multicomputador Nó //) [COR93].

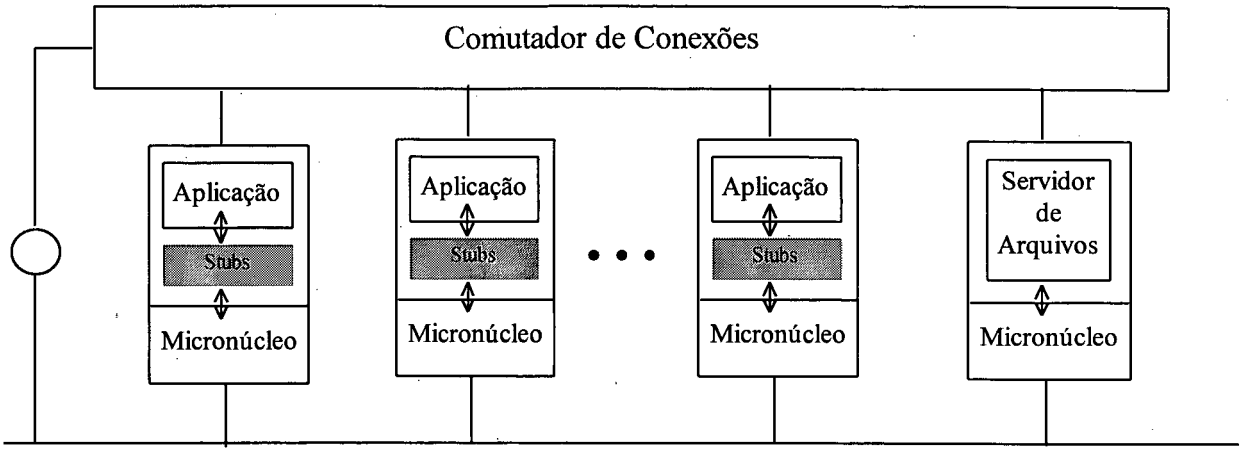


Figura 21 : Arquitetura do *Crux*.

## √ A camada das chamadas do sistema

Os serviços do sistema operacional são oferecidos através de uma interface de programação de tipo Unix. Ela é estendida, com serviços de comunicação síncrona direta, para criar um ambiente para programação paralela.

## √ A camada das comunicações de alto nível

Na implantação da interface de programação Unix, através de chamadas de procedimentos remotos, as trocas de mensagens entre os intermediários dependem de um serviço de comunicação. O objetivo da camada das comunicações de alto nível é de prover esses serviços que servem ao transporte de mensagens volumosas transitando pelos canais da rede de interconexão dinâmica.

Comunicações síncronas diretas podem ser efetuadas através dos seguintes operadores :

- ***Send*** **(process, message, length)**

  Um processo colocado em um nó qualquer solicita, através do operador *Send,* o envio de uma mensagem de endereço *message* e de tamanho *length* a outro processo *process* colocado em outro nó.

- ***Receive*** **(process, message, length)**

  Um processo colocado em um nó qualquer solicita, através do operador *Receive,* a recepção de uma mensagem no endereço *message,* de tamanho máximo *length,* de outro processo *process* colocado em outro nó.

- ***ReceiveAny*** **(process, message, length)**

  Um processo colocado em um nó qualquer solicita, através do operador *ReceiveAny,* a recepção de uma mensagem no endereço *message,* de tamanho máximo *length,* de outro processo cuja identificação é devolvida em *process*.

No caso do operador *ReceiveAny*, a política de escolha de uma comunicação entre as alternativas possíveis é uma decisão de implementação cuja execução fica a cargo do núcleo do sistema operacional.

Todos os três operadores são utilizados nas comunicações que seguem uma disciplina de tipo cliente-servidor. Assim, um cliente executa o operador *Send* (para o envio de uma requisição de serviço) seguido, imediatamente, de *Receive* (para a recepção da resposta) e um servidor executa o operador *ReceiveAny* (para a recepção de uma requisição de serviço) e *Send* (para o envio da resposta). Dois desses operadores respondem de forma imediata às exigências das comunicações simétricas que caracterizam uma disciplina de tipo produtor-consumidor. Assim, um produtor executa o operador *Send* (para o envio de uma mensagem) e um consumidor executa o operador *Receive* (para a recepção de uma mensagem).

## √ A camada das chamadas do núcleo

Os serviços oferecidos pelo núcleo se referem ao estabelecimento e rompimento de canais da rede de interconexão dinâmica necessárias às comunicações de alto nível e à alocação e liberação de nós de trabalho associadas ao mecanismo de criação dinâmica de processos.

A realização das comunicações de alto nível se apoia sobre os seguintes operadores de conexão e desconexão de canais:

- ***Connect*** **(node)**

  Um processo colocado em um nó qualquer, solicita através do operador *Connect,* sua conexão através de um canal direto ao nó *node*.

- ***ConnectAny*** **(node)**

  Um processo colocado em um nó qualquer solicita, através do operador *ConnectAny,* sua conexão através de um canal direto a um nó qualquer cuja identificação é devolvida em *node*.

- ***Disconnect*** **(node)**

  Um processo colocado em um nó qualquer solicita, através do operador *Disconnect,* a desconexão do canal direto que o conecta ao nó *node*.

Os operadores *Connect*, *ConnectAny* e *Disconnect* são usados pelos operadores da camada das comunicações de alto nível.

Para permitir a criação e a remoção de processos, são necessários os seguintes operadores para a alocação e liberação de nós de trabalho:

- ***Allocate*** **(node)**

  Um processo colocado em um nó qualquer solicita, através do operador *Allocate,* a alocação do nó de trabalho *node*.

– ***AllocateAny*** **(node)**

Um processo colocado em um nó qualquer solicita, através do operador *AllocateAny,* a alocação de um nó de trabalho livre qualquer cuja identificação é devolvida em *node.*

– ***Deallocate*** **()**

Um processo colocado em um nó qualquer solicita, através do operador *Deallocate,* a liberação do nó de trabalho por ele ocupado.

Os operadores *Allocate*, *AllocateAny* e *Deallocate* são necessários ao mecanismo de criação dinâmica de processos oferecido pela interface do sistema Unix. Eles são usados somente na implementação da camada das chamadas do sistema.

Os operadores de conexão e desconexão envolvem a aplicação de ações sobre o comutador de conexões. Os operadores de alocação e liberação de nós de trabalho envolvem a manipulação de sinais de controle do barramento de serviço. Como esses dispositivos só podem ser comandados pelo nó de controle, o núcleo do sistema operacional é colocado obrigatoriamente sobre esse nó.

## √ A camada das comunicações de baixo nível

O objetivo da camada das comunicações de baixo nível é de prover serviços de comunicação, que permitem trocas de mensagens compactas, para as chamadas de procedimento remotos através do barramento de serviço.

As comunicações pelo barramento de serviço tem sempre o nó de controle como origem ou destino e, todas elas, se desenvolvem sobre o comando desse nó. Para suportar essas comunicações, operadores diferentes, para o nó de controle e para os nós de trabalho, evidenciam a assimetria no seu comportamento:

- ***W_SendReceive*** **(request, reply)**

  Um processo colocado em um nó de trabalho qualquer solicita, através do operador *W_SendReceive,* o envio ao nó de controle de uma mensagem de requisição de serviço *request* e a recepção de uma mensagem de resposta em *reply*.

- ***C_ReceiveAny*** **(node, request)**

  O processo colocado no nó de controle solicita, através do operador *C_ReceiveAny,* a recepção de uma mensagem de requisição de serviço em *request* de um nó de trabalho qualquer cuja identificação é devolvida em *node*.

- ***C_Send*** **(node, reply)**

  O processo colocado no nó de controle solicita, através do operador *C_Send,* o envio de uma mensagem de resposta *reply* a um nó de trabalho identificado por *node*.

Os operadores são orientados exclusivamente para a disciplina de comunicação de tipo cliente-servidor. Assim, um cliente executa o operador *W_SendReceive* (para o envio de uma requisição de serviço e a recepção da resposta) e um servidor executa o operador *C_ReceiveAny* (para a recepção de uma requisição de serviço) e *C_Send* (para o envio da resposta).

No caso do operador *C_ReceiveAny*, a política de escolha de uma comunicação entre as alternativas possíveis é uma decisão de implementação cuja execução fica a cargo do núcleo do sistema operacional .

### 4.2.3. O servidor *Crux*

Em sua primeira versão, o sistema *Crux* considera o Nó // apenas como um *pool* de nós processadores, estando os dispositivos de entrada e saída (teclado, monitor, disco, etc.) e o sistema de arquivos, localizados na máquina hospedeira. A máquina hospedeira estará executando o sistema Linux, responsável por prover acesso aos dispositivos e ao sistema de arquivos.

Os processos *Crux* acessam os dispositivos e arquivos através de chamadas de procedimentos remotos, que fornecem uma interface compatível com a do sistema UNIX, tornando transparente a comunicação com a máquina hospedeira.

A interface entre o Nó // e a máquina hospedeira (figura 22) será realizada por dois processos : o processo representante do **servidor de arquivos** (RSA) e o processo de acesso ao sistema Linux (AL).

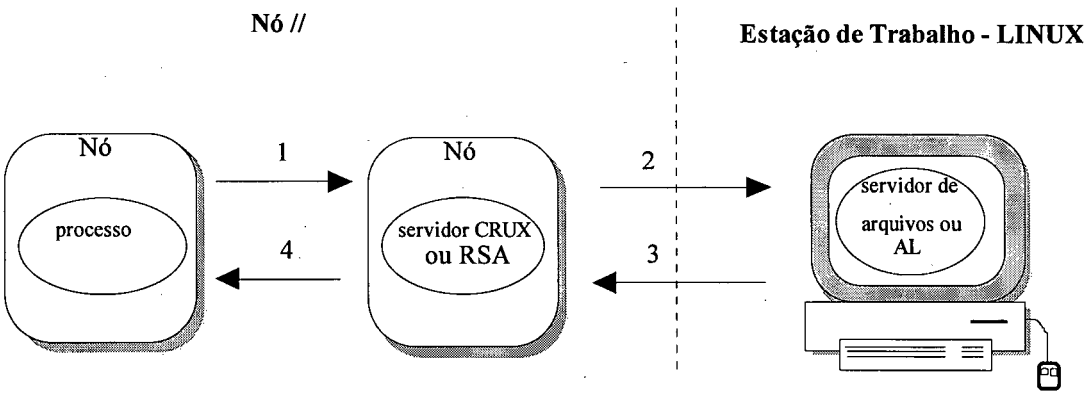


Figura 22 : Utilização de um servidor de arquivos externo ao Nó //.

O RSA consiste de um processo *Crux*, que está localizado no NCE, e recebe todas as requisições endereçadas ao sistema Linux, enviando-as à máquina hospedeira, pelo canal externo. Do ponto de vista dos processos *Crux*, o RSA constitui um servidor de arquivos, que executa todos os serviços recebidos, muito embora, na realidade, as requisições sejam passadas à máquina hospedeira para execução.

O RSA executa, basicamente, os seguintes passos :

√ Recebe de algum processo *Crux*, uma requisição de serviço endereçada à máquina hospedeira;

√ Envia a requisição à máquina hospedeira, através do canal externo;

√ Permanece esperando pela resposta do serviço;

√ Recebe, pelo canal externo, o resultado do serviço;

√ Retorna o resultado do serviço ao processo *Crux* requisitante.

O processo AL consiste de um processo Linux, que está localizado na máquina hospedeira, responsável por processar as requisições enviadas pelo RSA, através da execução da chamada Linux correspondente ao pedido. O AL executa, basicamente, os seguintes passos :

√ Recebe do RSA uma requisição de serviço, através do canal externo;

√ Analisa a mensagem recebida e processa o pedido através da execução da chamada Linux correspondente;

√ Monta uma mensagem de resposta, enviando-a ao RSA através do canal externo.

Resumindo, os passos de uma chamada ao sistema de arquivos (figura 22) :

√ Um processo requisita um serviço ao servidor *Crux* fazendo uma chamada de procedimento remoto (1);

√ O servidor constata que a chamada se refere a serviços de arquivo e faz uma chamada de procedimento remoto ao servidor de arquivos (2);

√ O servidor de arquivos recebe a mensagem, executa a tarefa e responde ao servidor *Crux* (3);

√ Este último recebe a resposta e envia ao processo que requisitou o serviço (4).

# 5. Implementação da linguagem SuperPascal no Nó //

Este capítulo descreve a implementação da linguagem SuperPascal no Nó //. O trabalho aqui proposto visa complementar o ambiente de execução descrito no capítulo 4 (simulador e sistema operacional) com uma linguagem de programação paralela.

## 5.1. Motivações para a escolha da linguagem SuperPascal

Das linguagens descritas no capítulo 3, somente foram consideradas Occam, Joyce e SuperPascal (apesar de sua importância teórica, CSP é apenas um projeto de linguagem sem implementação).

Dentre os motivos para a escolha da linguagem SuperPascal, considerou-se os seguintes :

√ **A disponibilidade e boa documentação do código fonte** : O ambiente para programação paralela para Occam, apresentado no item 3.5.2, possui boa documentação mas o código fonte não está disponível nem para o ambiente de produção e nem para o de execução de programas. Joyce e SuperPascal possuem boa documentação e o código fonte está disponível tanto para o ambiente de produção (compilador) quanto para o de execução (interpretador) de programas.

√ **A similaridade entre os modelos de programação da linguagem e de execução do multicomputador** : Em Occam, os processos e canais de um programa paralelo podem ser determinados em tempo de compilação. Em Joyce e SuperPascal, tanto processos quanto canais podem ser criados dinamicamente, em tempo de execução. Dessa forma, Joyce e SuperPascal possuem modelos de programação mais próximos dos multicomputadores com redes de interconexão dinâmicas do que Occam.

√ **Portabilidade do ambiente de execução da linguagem** : O ambiente de execução de Occam possui portabilidade restrita, possuindo maior afinidade com redes de Transputers. Os ambientes de execução de Joyce e SuperPascal, representados por interpretadores apresentam grande portabilidade. Entretanto, o interpretador SuperPascal foi escrito em Pascal e o de Joyce foi escrito em linguagem *Assembly*. Dessa forma, o ambiente de execução da linguagem SuperPascal possui maior portabilidade do que o ambiente da linguagem Joyce.

## 5.2. Máquina SuperPascal

O compilador SuperPascal gera código para um computador específico chamado **máquina SuperPascal**. O código executado por esse computador é conhecido como **código SuperPascal**.

A máquina SuperPascal é uma máquina de pilha. Sua memória é constituída por uma seqüência de números inteiros. Os elementos da memória são conhecidos como **palavras** e os índices referentes a cada elemento são os **endereços**. As palavras armazenam o código de um programa SuperPascal com suas instruções e suas variáveis. O código que tem tamanho fixo, é colocado no início da memória. O restante da memória é utilizada como pilha de variáveis. Durante a execução de instruções a pilha também armazena resultados temporários. A máquina possui quatro registradores de índice :

√ registrador p : registrador do programa - contém o endereço da instrução atual;

√ registrador b : registrador base - usado para acessar variáveis;

√ registrador s : registrador de pilha - armazena o endereço do topo da pilha do processo;

√ registrador t : registrador de início de memória livre - armazena o endereço da última posição de memória ocupada.

A representação da memória de uma máquina SuperPascal aparece na figura 23.

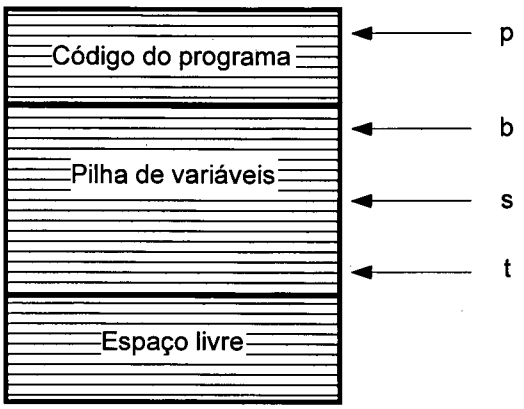


Figura 23 : Memória da máquina SuperPascal.

A máquina SuperPascal é ideal para a linguagem SuperPascal porque :

√ os operadores do código SuperPascal correspondem diretamente aos conceitos da linguagem;

√ o código SuperPascal de um programa tem praticamente a mesma sintaxe do próprio programa fonte. Conseqüentemente, o gerador de código do compilador pode ser uma extensão do analisador sintático;

√ uma máquina SuperPascal implementada em *hardware* poderia executar programas SuperPascal com maior velocidade do que a maioria dos computadores tradicionais.

Como não se dispõe de uma máquina SuperPascal implementada em *hardware*, ela pode ser simulada através de um **interpretador SuperPascal**. O ambiente de programação é composto por dois componentes independentes : um ambiente de produção de programas (o compilador) e um ambiente de execução (o interpretador).

Uma desvantagem da interpretação do código é a baixa velocidade de execução. Uma vantagem é a maior **portabilidade** : o *software* pode ser transportado de um computador para outro reescrevendo o interpretador para a máquina destino.

## 5.3. O código SuperPascal

Existem 111 instruções na máquina SuperPascal que são compostas por um código de operação e um número variável de operandos. Tanto os códigos de operação quanto os operandos são representados por números inteiros. Assim, o código produzido pelo compilador SuperPascal é constituído por uma seqüência de números inteiros. Os códigos de operação podem ser observados na figura 24.

```
{ operation parts }
minoperation = 0;  maxoperation = 110;
abs2 = 0;          absint2 = 1;          add2 = 2;          addreal2 = 3;
and2 = 4;          arctan2 = 5;          assign2 = 6;       assume2 = 7;
case2 = 8;         checkio2 = 9;         chr2 = 10;         cos2 = 11;
divide2 = 12;      divreal2 = 13;        do2 = 14;          downto2 = 15;
endall2 = 16;      enddown2 = 17;        endio2 = 18;       endparallel2 = 19;
endproc2 = 20;     endprocess2 = 21;     endprog2 = 22;     endto2 = 23;
eof2 = 24;         eoln2 = 25;           eqord2 = 26;       eqreal2 = 27;
eqstring2 = 28;    equal2 = 29;          exp2 = 30;         field2 = 31;
float2 = 32;       floatleft2 = 33;      for2 = 34;         forall2 = 35;
goto2 = 36;        grord2 = 37;          grreal2 = 38;      grstring2 = 39;
index2 = 40;       ln2 = 41;             lsord2 = 42;       lsreal2 = 43;
lsstring2 = 44;    minus2 = 45;          minusreal2 = 46;   modulo2 = 47;
multiply2 = 48;    multreal2 = 49;       neord2 = 50;       nereal2 = 51;
nestring2 = 52;    ngord2 = 53;          ngreal2 = 54;      ngstring2 = 55;
nlord2 = 56;       nlreal2 = 57;         nlstring2 = 58;    not2 = 59;
notequal2 = 60;    odd2 = 61;            open2 = 62;        or2 = 63;
ordconst2 = 64;    parallel2 = 65;       pred2 = 66;        proccall2 = 67;
procedure2 = 68;   process2 = 69;        program2 = 70;     read2 = 71;
readint2 = 72;     readln2 = 73;         readreal2 = 74;    realconst2 = 75;
receive2 = 76;     result2 = 77;         round2 = 78;       send2 = 79;
sin2 = 80;         sqr2 = 81;            sqrint2 = 82;      sqrt2 = 83;
stringconst2 = 84; subreal2 = 85;        subtract2 = 86;    succ2 = 87;
to2 = 88;          trunc2 = 89;          value2 = 90;       variable2 = 91;
varparam2 = 92;    write2 = 93;          writebool2 = 94;   writeint2 = 95;
writeln2 = 96;     writereal2 = 97;      writestring2 = 98; globalcall2 = 99;
globalvalue2 = 100; globalvar2 = 101;    localreal2 = 102;  localvalue2 = 103;
localvar2 = 104;   ordassign2 = 105;     ordvalue2 = 106;   realassign2 = 107;
realvalue2 = 108;  defaddr2 = 109;       defarg2 = 110;
```

Figura 24 : Códigos de operação SuperPascal.

Por exemplo, o código gerado para o algoritmo simples da figura 25 é apresentado na figura 26-a. Substituindo os códigos de operações numéricos da figura 26-a pelos mnemônicos encontrados na figura 24, colocando os argumentos entre parênteses e numerando a seqüência de execução do programa, o código SuperPascal fica mais legível (figura 26-b) :

```
program Simples;
  { ------------------------------------------
    Um produtor e um consumidor.
   ------------------------------------------}
  const
    k = 9;

  type
    tipoCanal = *(integer);

  var
   canal : tipoCanal;

procedure produtor;
  begin
   send(canal, k);
  end;

  procedure consumidor;
  var
   msg : integer;
  begin
   receive(canal,msg);
  end;

  begin
   open(canal);
   parallel
    produtor |
    consumidor
   end
  end.
```

Figura 25 : Texto de um programa simples.

```
70 1 1 5 44 1
 68 2 0 0 7 7 15
100 4
 9 16
 64 9
 79 1 1 16
 18
 20
 68 3 0 1 7 7 20
100 4
 9 23
104 4
 76 1 1 23
 18
 20
104 4
 62 27
 65
 69 4 4 10 29
 99 -48
 21 15 29
 69 5 5 10 30
 99 -39
 21 5 31
 19 32
 22
```

(a)

```
1.  program2(1, 1, 5, 44, 1)
15. procedure2(2, 0, 0, 7, 7, 15)
16. globalvalue2(4)
17. checkio2(16)
18. ordconst2(9)
19. send2(1, 1, 16)
20. endio2
21. endproc2
9.  procedure2(3, 0, 1, 7, 7, 20)
10. globalvalue2(4)
11. checkio2(23)
12. localvar2(4)
13. receive2(1, 1, 23)
23. endio2
24. endproc2
2.  localvar2(4)
3.  open2(27)
4.  parallel2
5.  process2(4, 4, 10, 29)
14. globalcall2(-48)
22. endprocess2(15, 29)
6.  process2(5, 5, 10, 30)
8.  globalcall2(-39)
25. endprocess2(5, 31)
7.  endparallel2(32)
26. endprog2
```

(b)

Figura 26 : Código para o texto do programa da figura 25.

## 5.4. Implementação da linguagem SuperPascal em máquinas monoprocessadoras sobre o sistema Unix

O interpretador original da máquina SuperPascal [HAN94b] permite a execução de programas SuperPascal em estações de trabalho Sun. Para o sistema Unix, o interpretador é visto como um único processo seqüencial. Programas paralelos podem ser executados porque o próprio interpretador implementa internamente o escalonamento dos processos SuperPascal. Entretanto, como a linguagem foi projetada para ser implementada em multicomputadores, não houve preocupação em elaborar um algoritmo de escalonamento justo e eficiente para esse ambiente monoprocessado.

As instruções *parallel* e *forall* em um programa fonte na linguagem SuperPascal, produzem, respectivamente, as instruções parallel2 e forall2 da máquina SuperPascal. Essas instruções implementam o mecanismo de criação dinâmica de processos.

Processos SuperPascal são compostos por uma área de código e uma área de pilha. A área de pilha é chamada **registro de ativação** e é alocada na criação de um processo, armazenando seus parâmetros, seu contexto (registradores p, b , s e endereço de retorno), suas variáveis locais e suas variáveis temporárias.

A estrutura de dados utilizada para implementar o escalonador de processos é uma pilha de processos prontos (*ready*) para executar. Cada processo empilhado armazena os registradores p, b e s além do endereço do processo pronto anterior na pilha.

Dois procedimentos do interpretador são responsáveis pela gerência do escalonamento de processos : o procedimento ***activate*** e o procedimento ***select***.

O procedimento *activate* é responsável por empilhar os processos prontos para executar e é executado nas seguintes instruções da máquina SuperPascal : process2, forall2, send2 e receive2. Em cada uma dessas instruções, o procedimento *activate* tem uma função específica :

- √ process2 : empilha o processo que está sendo criado pela instrução process2;
- √ forall2 : empilha todos os processos criados pela instrução forall2;
- √ send2 : empilha o processo bloqueado na recepção de uma mensagem pelo canal em questão, caso ele exista;
- √ receive2 : empilha o processo bloqueado no envio de uma mensagem pelo canal em questão, caso ele exista;

O escalonamento, ou a seleção de um novo processo da pilha de processos prontos é não-preemptivo, realizado através do procedimento *select*, executado nas seguintes instruções da máquina SuperPascal : endparallel2, endprocess2, forall2, endall2, send2 e receive2. Essas instruções são, portanto, responsáveis pela troca de contexto que implementam o paralelismo virtual dos processos de um programa SuperPascal. Um processo, *a priori*, é executado do início ao fim, contudo, ao encontrar instruções bloqueantes de envio e recepção de mensagens (send2 e receive2, respectivamente) é necessário efetuar uma troca de contexto. Em cada uma das instruções que executam o procedimento *select*, ele tem uma função específica :

- √ endparallel2 : seleciona o processo do topo da pilha para ser executado que representa o último processo empilhado pelo procedimento *activate*;
- √ endprocess2 : seleciona o processo do topo da pilha para ser executado, caso a pilha não esteja vazia;
- √ forall2 : seleciona o processo do topo da pilha para ser executado que representa o último processo empilhado pelo procedimento *activate*;
- √ endall2 : seleciona os próximos processos do topo da pilha para serem executados;

- √ send2 : seleciona o processo do topo da pilha para ser executado quando não existe nenhum processo pronto para receber a mensagem;
- √ receive2 : seleciona o processo do topo da pilha para ser executado quando não existe nenhum processo pronto para enviar a mensagem.

As instruções *open*, *send* e *receive* em um programa fonte na linguagem SuperPascal, produzem, respectivamente, as instruções open2, send2 e receive2 da máquina SuperPascal. Essas instruções implementam o mecanismo de criação dinâmica de canais de comunicação e de envio e recepção de mensagens síncronas, respectivamente. São produzidas também duas instruções auxiliares na comunicação : checkio2 e endio2.

A estrutura de dados utilizada para implementar os canais de comunicação é composta por um vetor de números inteiros, chamado open, onde cada posição identifica o endereço do registro de ativação do processo que vai enviar ou receber mensagens e, de um **índice** que identifica o canal pela posição no vetor open.

A instrução open2 é responsável pela criação de um canal de comunicação que consiste em alocar uma posição do vetor open. Antes de executar instruções send2 ou receive2 é executada a instrução checkio2 que verifica se o canal a ser utilizado na comunicação é válido. Depois de executar send2 ou receive2, a instrução endio2 é executada para retirar da pilha o índice do canal.

Quando a instrução send2 está sendo executada sem que exista um processo receptor para a mensagem que está sendo enviada, o endereço do seu registro de ativação, onde está armazenada a mensagem, é colocado no vetor open na posição relativa ao índice do canal que está sendo utilizado. Quando a instrução receive2 está sendo executada sem que exista um processo emissor para a mensagem que está sendo recebida, o endereço do seu registro de ativação, onde a mensagem será armazenada, é colocado no vetor open na posição relativa ao índice do canal que está sendo utilizado. Dessa forma, as trocas de mensagens são simuladas através de cópias na memória.

## 5.5. Implementação do interpretador da linguagem SuperPascal no multicomputador Nó //

Duas alternativas foram consideradas para implementar o interpretador da máquina SuperPascal no multicomputador Nó //.

A primeira alternativa (figura 27), considerando o Nó // como uma **máquina Unix**, ou seja, como uma máquina de propósito geral que executa simultaneamente aplicações diversas, possui as seguintes características :

√ o interpretador é carregado inicialmente em apenas um nó de trabalho, a partir da máquina hospedeira;

√ o interpretador carrega o programa SuperPascal compilado a ser executado para esse nó, armazenando-o na memória da máquina SuperPascal local;

√ durante a execução do programa SuperPascal, processos são criados e finalizados através das chamadas de sistema *fork* e *exit* do sistema operacional *Crux*. Assim, cada novo processo SuperPascal gera um novo processo *Crux*, contendo uma cópia do interpretador e do programa SuperPascal.

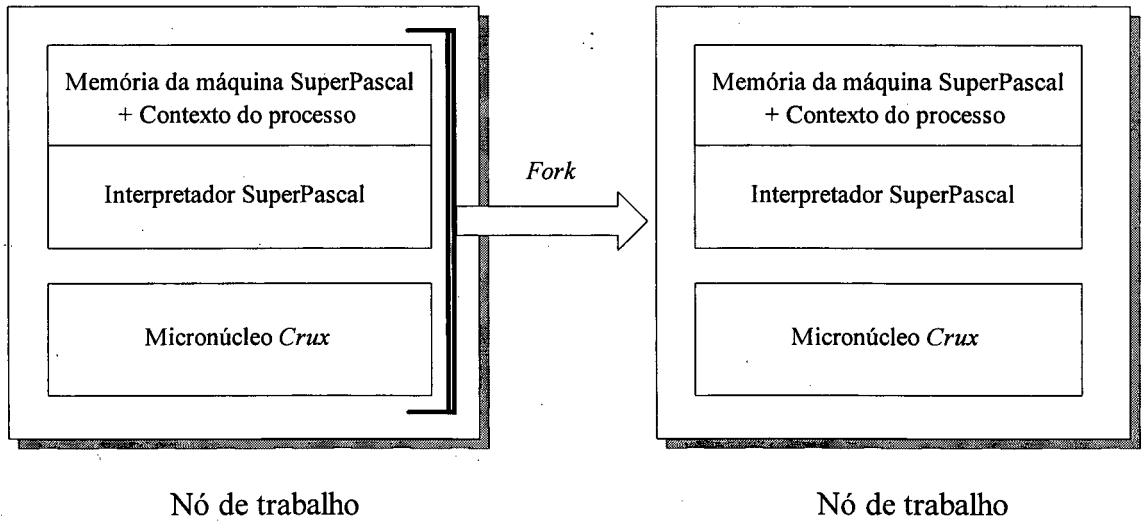


Figura 27 : Criação de processo na máquina Unix.

A segunda alternativa (figura 28), considerando o Nó // como uma **máquina SuperPascal paralela**, ou seja, como uma máquina de propósito especial que executa apenas processos SuperPascal, possui as seguintes características :

√ quando a máquina SuperPascal é instalada, uma cópia do interpretador é carregada inicialmente em cada nó de trabalho, a partir da máquina hospedeira. Todos os nós de trabalho passam a conter de forma permanente o núcleo do sistema operacional acrescido do ambiente de execução da linguagem SuperPascal;

√ o interpretador de um dos nós carrega o programa SuperPascal compilado a ser executado para esse nó, armazenando-o na memória da máquina SuperPascal local;

√ quando processos são criados na máquina SuperPascal, apenas a memória da máquina SuperPascal e o contexto do processo a ser executado são enviados para um nó livre qualquer. Os nós livres permanecem esperando por um processo SuperPascal a ser executado, recebem-no, executam-no e enviam resposta de término ao processo pai;

√ novos serviços de comunicação introduzidos na camada de comunicação de alto nível do sistema operacional *Crux* devem ser utilizados para implementar a criação e o término de processos através de mensagens;

√ quando a máquina SuperPascal é finalizada, são removidos os interpretadores de todos os nós de trabalho.

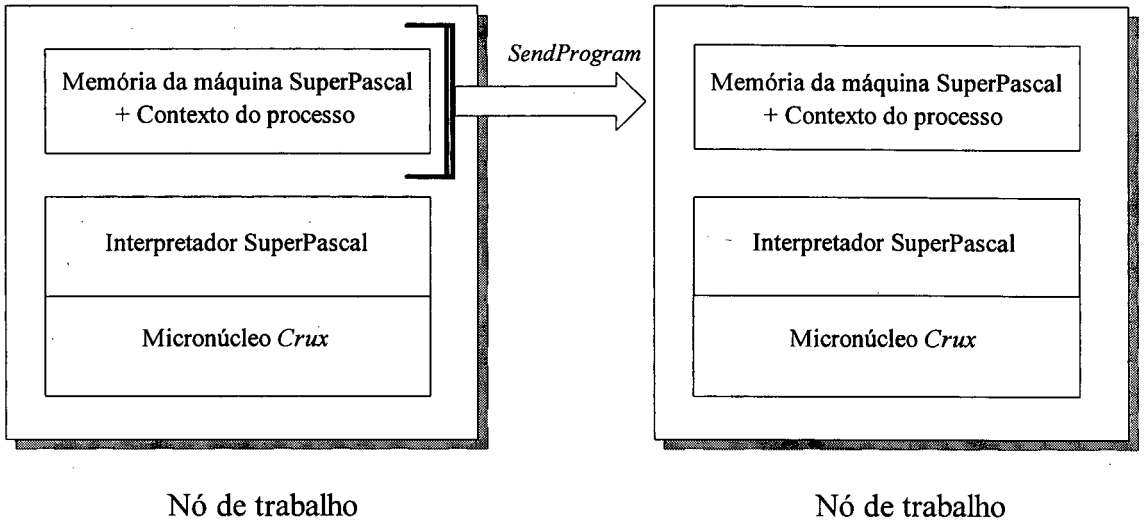


Figura 28 : Criação de processo na máquina SuperPascal paralela.

As duas alternativas foram implementadas para permitir a futura comparação de desempenho, quando a máquina real estiver disponível. Antes disso, algumas características dessas alternativas podem ser confrontadas.

Características da máquina Unix :

√ a máquina é de propósito geral;

√ é feita uma cópia de toda a memória do nó de trabalho para criar um novo processo SuperPascal;

√ o ambiente de execução da linguagem SuperPascal é instalado por demanda;

√ o tempo de carga inicial do ambiente de execução é menor;

√ o tempo total de execução de programas SuperPascal, a princípio, é maior;

√ a criação e término de processos é realizada através de chamadas do sistema operacional *Crux*.

Características da máquina SuperPascal paralela :

√ a máquina é de propósito especial;

√ é feita uma cópia apenas do código do programa armazenado na memória da máquina SuperPascal e do contexto do processo a ser executado para criar um novo processo SuperPascal;

√ o ambiente de execução da linguagem SuperPascal é instalado em todos os nós de forma permanente;

√ o tempo de carga inicial do ambiente de execução é maior;

√ o tempo total de execução de programas SuperPascal, a princípio, é menor;

√ a criação e término de processos é realizada exclusivamente através de trocas de mensagens entre os diversos nós.

## 5.6. SuperPascal na Máquina Unix

O interpretador da máquina Unix para o Nó // permite a execução simultânea de programas SuperPascal entre outras aplicações nesse multicomputador. Todas as características da linguagem SuperPascal são preservadas nessa implementação, sendo que o compilador não foi alterado.

As seguintes tarefas foram executadas para a implementação do interpretador da Máquina Unix:

√ tradução do interpretador, originalmente escrito em Pascal, para a linguagem C cujo compilador gera código compatível com o multicomputador Nó // e o sistema operacional *Crux*;

√ definição e implementação dos mecanismos de distribuição dos processos gerados pelas instruções *parallel* e *forall* pelos nós do multicomputador usando os mecanismos de criação de processos já disponíveis no sistema *Crux*;

√ definição e implementação da comunicação entre processos, fazendo uso de mecanismos já disponíveis no sistema *Crux* e introduzindo novos mecanismos específicos para a linguagem SuperPascal.

### 5.6.1. Tradução do interpretador para a linguagem C

O interpretador foi originalmente escrito em linguagem Pascal. Para gerar código compatível com o multicomputador Nó // e com o sistema operacional *Crux*, o interpretador SuperPascal (*common.p* e *interpret.p*) foi traduzido para a linguagem C.

A tradução foi feita utilizando inicialmente um programa utilitário do sistema Linux chamado *p2c*. Esse utilitário faz a tradução de programas escritos em Pascal para programas escritos em C. Obteve-se dessa forma os arquivos *common.h* e *interpret.c*, que puderam então ser

compilados e ligados às rotinas de biblioteca *Crux*, para gerar o interpretador SuperPascal executável pelo Nó //.

Atualmente, programas precisam respeitar grandes restrições de tamanho para serem executados pelo simulador do Nó // (essa limitação está sendo resolvida em outro trabalho de dissertação de mestrado em desenvolvimento nesse curso). Dessa forma, algumas restrições de tamanho foram introduzidas no interpretador original.

### 5.6.2. Definição e implementação dos mecanismos de distribuição de processos

Duas instruções geram processos na linguagem SuperPascal : *parallel* e *forall*. A instrução fonte *parallel* gera as seguintes instruções da máquina SuperPascal :

√ uma instrução **parallel2**;

√ um par de instruções **process2, endprocess2** para cada processo gerado;

√ uma instrução **endparallel2**.

Na implementação original, essas instruções executam as seguintes tarefas :

√ parallel2 : inicializa o contador do número de processos da instrução *parallel*;

√ process2 : incrementa o número de processos da instrução *parallel*, cria um registro de ativação para o processo que está sendo criado e empilha esse processo utilizando o procedimento *activate*;

√ endprocess2 : decrementa o número de processos da instrução *parallel*, libera o registro de ativação do processo que está terminando e, caso exista, seleciona um próximo processo pronto para executar da pilha utilizando o procedimento *select*;

√ endparallel2 : seleciona o processo do topo da pilha para ser executado, utilizando o procedimento *select*, que representa o último processo empilhado pelo procedimento *activate*.

A figura 29-a mostra o trecho da figura 25 que utiliza a instrução *parallel* no programa fonte e a figura 29-b mostra a seqüência de instruções da máquina SuperPascal geradas para esse trecho (o número indicado na frente das instruções da figura 29-b indica a ordem de execução dessas instruções pelo interpretador, considerando o programa completo).

```
...                 ...
parallel            4.  parallel2
  produtor |        5.  process2(4, 4, 10, 29)
                    14. globalcall2(-48)
                    22. endprocess2(15, 29)
  consumidor        6.  process2(5, 5, 10, 30)
                    8.  globalcall2(-39)
                    25. endprocess2(5, 31)
end                 7.  endparallel2(32)
...                 ...
```

(a) (b)

Figura 29 : Trechos de código para o texto do programa da figura 25.

A seqüência de execução de uma instrução *parallel* é mostrada na figura 29-b. Primeiramente são criados todos os registros de ativação dos processos pelo procedimento *activate* através das instruções process2 e, então, os processos são selecionados pelo procedimento *select* chamado pelas instruções endparallel2 e endprocess2.

Na implementação da instrução *parallel* no multicomputador Nó //, os processos são criados, finalizados e coordenados utilizando chamadas *fork*, *exit* e *wait* do sistema *Crux* e, considerando apenas um processo executando em cada nó, as tarefas realizadas pelas instruções da máquina SuperPascal são as seguintes :

√ parallel2 : armazena o contexto atual (registradores b, s, t) para retomar a execução quando os processos filhos terminarem e inicializa o contador do número de processos da instrução *parallel*;

√ process2 : realiza as mesmas tarefas da implementação original : incrementa o número de processos da instrução *parallel*, cria um registro de ativação para o processo que está sendo criado e empilha esse processo chamando o procedimento *activate*;

√ endprocess2 : finaliza o processo utilizando a chamada de sistema *exit* para liberar o nó onde ele está executando;

√ endparallel2 : inicialmente, armazena o endereço de retorno (endereço logo após o endparallel2). Então, seleciona o processo do topo da pilha e executa a chamada de sistema *fork*, para cada processo da instrução *parallel*. Depois, permanece esperando pelo término dos processos filhos executando chamadas de sistema *wait* para retomar sua própria execução.

A instrução *forall* gera as seguintes instruções da máquina SuperPascal :

√ uma instrução **forall2**;

√ gera instruções para a chamada do processo : **globalcall2, proccall2**;

√ uma instrução **endall2**.

Na implementação original, essas instruções executam as seguintes tarefas :

√ forall2 : determina o número de processos da instrução *forall*. Cria um registro de ativação para cada processo componente e empilha cada um deles chamando o procedimento *activate*. Seleciona o processo do topo da pilha para ser executado.

√ endall2 : decrementa o número de processos da instrução *forall*, libera o registro de ativação do processo que está terminando e, caso exista, seleciona o próximo processo pronto para executar da pilha chamando o procedimento *select*.

A figura 30-a mostra um trecho de programa fonte que utiliza a instrução *forall* e a figura 30-b mostra a seqüência de instruções da máquina SuperPascal geradas para esse trecho (o número indicado na frente das instruções da figura 30-b indica a ordem relativa de execução dessas instruções pelo interpretador, já que não está listado todo o programa fonte).

```
...
forall i := 1 to 9 do
  processo;
...
```

(a)

```
...
1. forall2(3, 5, 9, 22)
2. globalcall2(-22)
3. endall2(22)
...
```

(b)

Figura 30 : Trechos de código para exemplificar a instrução *forall*.

A seqüência de execução de uma instrução *forall* é mostrada na figura 30-b. Primeiramente são criados todos os registros de ativação dos processos pelo procedimento *activate* através da instrução forall2 e, então, os processos são selecionados pelo procedimento *select* chamado pelas instruções forall2 e endall2.

Na implementação da instrução *forall* no multicomputador Nó //, os processos são criados, finalizados e coordenados utilizando chamadas *fork*, *exit* e *wait* do sistema *Crux* e, considerando apenas um processo executando em cada nó, as tarefas realizadas pelas instruções da máquina SuperPascal são as seguintes :

- √ forall2 : inicialmente, realiza as mesmas tarefas da implementação original : cria os registros de ativação dos processos componentes e os empilha chamando o procedimento *activate*. Então, armazena o contexto atual (registradores b, s, t) e o endereço de retorno. Seleciona o processo do topo da pilha para ser executado utilizando o procedimento *select* e executa a chamada de sistema *fork* para cada processo da instrução *forall*. Depois, permanece esperando pelo término dos processos filhos executando chamadas de sistema *wait* para retomar sua própria execução;
- √ endall2 : finaliza o processo utilizando a chamada de sistema *exit* para liberar o nó onde ele está executando.

### 5.6.3. Definição e implementação da comunicação entre processos

Na linguagem SuperPascal, a comunicação é síncrona indireta. O mecanismo existente (chamadas *Send*, *Receive* e *ReceiveAny*) provê comunicação síncrona direta, não combinando com o mecanismo proposto pela linguagem SuperPascal. Para conservar as características originais da comunicação entre processos da linguagem SuperPascal, foi necessário a introdução de novas chamadas de sistema no sistema *Crux* para comunicação através de canais.

A estrutura de dados utilizada para implementar os canais de comunicação, por ser compartilhada, foi colocada no nó de controle. Ela é manipulada na camada das chamadas do núcleo do sistema *Crux*, responsável pela gerência dos nós e conexões.

A figura 31 apresenta a estrutura de dados utilizada para implementar os canais de comunicação.

```
#define MAXCHAN 128 /* 0..127 canais */

/* canal SuperPascal */
struct chansp_entry
{
  t_nid             nid;
  unsigned char status;
};

static struct chansp_entry chansp[MAXCHAN];
```

Figura 31 : Estrutura de dados dos canais de comunicação.

O campo nid armazena o identificador do nó que deseja realizar uma comunicação através do canal identificado pelo índice do vetor chansp. A operação a ser realizada (envio ou recepção) fica armazenada no campo status.

Foram realizadas as seguintes tarefas :

√ criação de três novas chamadas de sistema na camada das comunicações de alto nível do *Crux*:

⇒ *unsigned int Opensp()*

Abre um canal de comunicação e retorna o seu índice.

⇒ *unsigned int Sendsp(t_cid cid, void far * msg, t_typemsg typeno, unsigned int len)*

Envia a mensagem *msg*, de tipo *typeno* e tamanho *len*, através do canal de identificador *cid*. Caso não exista processo receptor bloqueado no canal *cid*, o processo emissor fica bloqueado.

⇒ *unsigned int Receivesp(t_cid cid, void far * msg, t_typemsg typeno, unsigned int len)*

Recebe a mensagem *msg*, de tipo *typeno* e tamanho máximo *len*, através do canal de identificador *cid*. Caso não exista processo emissor bloqueado no canal *cid*, o processo receptor fica bloqueado.

√ criação de três novas chamadas de sistema na camada das chamadas do núcleo do *Crux*:

⇒ *unsigned int Allocatesp()*

Aloca um canal livre e devolve o identificador do canal aberto.

⇒ *unsigned int Connectsp(t_cid cid, t_op operation)*

Conecta o processo que quer realizar *operation* com o canal *cid*. *Operation* identifica emissão ou recepção de mensagens.

⇒ *unsigned int Disconnectsp(t_cid cid)*

Desconecta o canal *cid* do processo.

√ alteração do código executado pelo nó de controle para atender os novos serviços : *Allocatesp*, *Connectsp* e *Disconnectsp*;

√ alteração das instruções open2, send2 e receive2 do interpretador :

⇒ open2 : a instrução open2 foi modificada para executar a chamada *Opensp* da camada das comunicações de alto nível do sistema *Crux*;

⇒ send2 : foi modificada para executar a chamada *Sendsp* da camada das comunicações da alto nível do sistema *Crux*;

⇒ receive2 : foi modificada para executar a chamada *Receivesp* da camada das comunicações da alto nível do sistema *Crux*;

Observação : as instruções checkio2 e endio2 não foram alteradas.

Para ocorrer uma comunicação entre dois processos filhos, o processo pai executa inicialmente a chamada de sistema *Opensp*, enviando ao nó de controle um pedido de alocação de um canal livre da tabela de canais e recebe o índice correspondente ao canal alocado. Esse índice é herdado pelos processos filhos. A figura 32 mostra um exemplo de seqüência de execução de uma comunicação. O processo emissor, localizado no Nó 3, utiliza a chamada de sistema *Sendsp* para enviar a mensagem *msg* de tipo *typeno* tamanho *length* através do canal *cid* (1). O processo mestre recebe esse pedido de envio de mensagem pelo barramento de serviço e o armazena na tabela de canais openc, no índice *cid* (no exemplo, igual a 1), o identificador do nó emissor e a operação desejada. O processo emissor é bloqueado. O processo receptor, localizado no Nó 5, utiliza a chamada de sistema *Receivesp* para receber a mensagem *msg* de tipo *typeno* tamanho máximo *length* através do canal *cid* (2). O processo mestre recebe esse pedido de recepção de mensagem pelo barramento de serviço, verifica que no índice *cid* da tabela de canais openc existe um processo querendo enviar mensagens, conecta os nós dos processos através do *crossbar* e envia o identificador do nó receptor para o processo emissor (3) e o identificador do nó emissor para o processo receptor (4), desbloqueando a execução do processo emissor. A partir desse instante, os processos estão conectados e se comunicam diretamente através do canal do *crossbar* (5). No término da comunicação, os nós são desconectados e o canal *cid* permanece aberto.

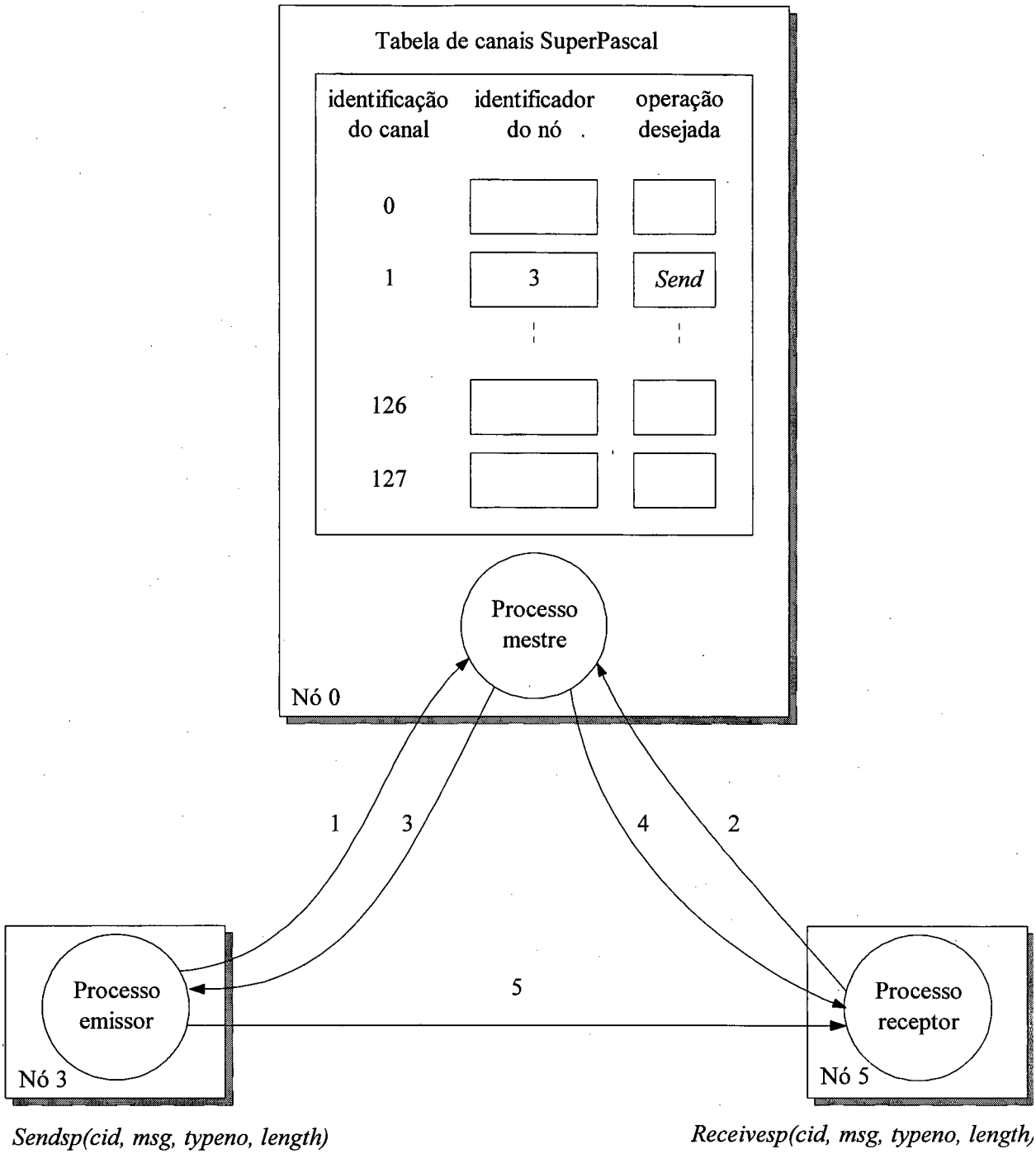


Figura 32 : Seqüência de execução de uma comunicação.

## 5.7. SuperPascal na Máquina SuperPascal Paralela

O interpretador da máquina SuperPascal paralela para o Nó // permite somente a execução de programas SuperPascal nesse multicomputador. Todas as características da linguagem SuperPascal são preservadas nessa implementação, sendo que o compilador não foi alterado.

A tradução do interpretador SuperPascal para a linguagem C e a comunicação entre processos SuperPascal implementados na alternativa anterior foram utilizados nessa implementação sem modificações.

As seguintes tarefas foram executadas na implementação da Máquina SuperPascal Paralela:

√ definição e implementação dos mecanismos de distribuição dos processos gerados pelas instruções *parallel* e *forall* pelos nós do multicomputador usando novos serviços de comunicação da camada de comunicação de alto nível do sistema *Crux* para a criação e término de processos;

√ procedimentos de carga e término do ambiente de execução permanente em todos os nós.

### 5.7.1. Definição e implementação dos mecanismos de distribuição de processos

Na implementação da instrução *parallel* e *forall* no multicomputador Nó //, para a criação e término de processos, foram introduzidos novos serviços de comunicação na camada de comunicação de alto nível do sistema *Crux*. O mecanismo existente (chamadas *Send*, *Receive* e *ReceiveAny*) provê comunicação síncrona direta e não é adequado para enviar processos para qualquer nó disponível.

A estrutura de dados utilizada para implementar o envio de processos SuperPascal foi colocada no nó de controle (figura 33). Ela é manipulada na camada das chamadas do núcleo do sistema *Crux*, responsável pela gerência dos nós e conexões.

```
#define NPROC  30

/*  filhos SuperPascal  */
struct father_entry
{
  t_nid             nid;
  unsigned char status;
};

static struct father_entry sons[NPROC];

static struct t_op operation[NPROC];
```

Figura 33 : Estrutura de dados para criar e finalizar processos SuperPascal.

O índice do vetor sons identifica o processo filho onde o identificador do nó do processo pai fica armazenado no campo nid e a operação a ser realizada (término do processo pai ou do processo filho) fica armazenada no campo status. O vetor operation armazena a operação a ser realizada na criação de processos SuperPascal que pode ser envio ou recepção de programas.

Foram realizadas as seguintes tarefas :

√ criação de quatro novas chamadas de sistema na camada das comunicações de alto nível do sistema *Crux*:

⇒ *unsigned int SendProgram(void far *msg, unsigned int len)*

Envia a mensagem *msg*, de tamanho *len* para um nó livre qualquer. A mensagem transporta o programa SuperPascal e o contexto do processo a ser executado para um nó que esteja bloqueado na chamada *ReceiveProgram*. Caso não exista nenhum processo nesse estado, o processo emissor fica bloqueado.

O processo que realiza essa chamada de sistema é considerado **processo pai**.

⇒ *unsigned int ReceiveProgram(void far * msg, unsigned int len)*

Recebe a mensagem *msg*, de tamanho máximo *len* de um nó qualquer. A mensagem deve conter o programa SuperPascal e o contexto do processo a ser executado de um nó qualquer que esteja executando a chamada *SendProgram*. Caso não exista nenhum processo nesse estado, o processo receptor fica bloqueado.

O processo que realiza essa chamada de sistema é considerado **processo filho** e atualiza o vetor sons que armazena informações para coordenar os processos SuperPascal.

⇒ *unsigned int Waitsp()*

Realiza o término de um processo pai SuperPascal. Caso nenhum processo filho tenha executado a chamada *Exitsp*, o processo pai permanece bloqueado, caso contrário, um processo filho é desbloqueado.

⇒ *unsigned int Exitsp()*

Realiza o término de um processo filho SuperPascal. Caso o processo pai não tenha executado a chamada *Waitsp*, o processo filho permanece bloqueado, caso contrário, desbloqueia o processo pai.

√ criação de duas novas chamadas de sistema na camada das chamadas do núcleo do sistema *Crux*:

⇒ *unsigned int ConnectAnysp(t_nid far *nid, t_op operation)*

Conecta o processo que quer realizar *operation* com o nó *nid. Operation* identifica emissão ou recepção de programas.

⇒ *unsigned int Familysp(t_op operation)*

⇒ Executa a coordenação do processo SuperPascal. *Operation* identifica término do processo pai ou do processo filho.

⇒ alteração do código executado pelo nó de controle para atender os novos serviços : *ConnectAnysp* e *Familysp.*

Nessa implementação, as tarefas realizadas pelas instruções da máquina SuperPascal são as seguintes :

√ parallel2 : armazena o contexto atual (registradores b, s, t) para retomar a execução quando os processos filhos terminarem e inicializa o contador do número de processos da instrução *parallel*;

√ process2 : realiza as mesmas tarefas da implementação original : incrementa o número de processos da instrução *parallel*, cria um registro de ativação para o processo que está sendo criado e empilha esse processo chamando o procedimento *activate*;

√ endprocess2 : finaliza o processo utilizando a chamada de sistema *Exitsp* para liberar o nó onde ele está executando;

√ endparallel2 : inicialmente armazena o endereço de retorno (endereço logo após o endparallel2). Então, seleciona o processo do topo da pilha, armazena o contexto do processo nas posições iniciais da memória da máquina SuperPascal e executa a chamada de sistema *SendProgram*, para cada processo da instrução *parallel*. Depois, permanece esperando pelo término dos processos filhos executando chamadas de sistema *Waitsp* para retomar sua própria execução;

√ forall2 : inicialmente, realiza as mesmas tarefas da implementação original : cria os registros de ativação dos processos componentes e os empilha chamando o procedimento *activate*. Então, armazena o contexto atual (registradores b, s, t) e o endereço de retorno. Seleciona o processo do topo da pilha para ser executado utilizando o procedimento *select*, armazena o contexto do processo nas posições iniciais da memória da máquina SuperPascal e executa a chamada de sistema *SendProgram* para cada processo da instrução *forall*. Depois, permanece esperando pelo término dos processos filhos executando chamadas de sistema *Waitsp* para retomar sua própria execução;

√ endall2 : finaliza o processo utilizando a chamada de sistema *Exitsp* para liberar o nó onde ele está executando.

### 5.7.2. Procedimentos de início e término da Máquina SuperPascal Paralela

Existem dois tipos de interpretadores na máquina SuperPascal Paralela : o interpretador SuperPascal pai que carrega o programa SuperPascal compilado em sua memória local a partir da máquina hospedeira (*spm*) e os interpretadores SuperPascal filhos que permanecerão esperando processos SuperPascal a serem executados (*spms*).

O *spm* é responsável pela carga do primeiro *spms* que, utilizando chamadas de sistema *fork*, carrega os próximos processos *spms*. Dessa forma, existe o nó inicial, responsável pela carga do programa SuperPascal compilado e início de sua execução (*spm*) e nós livres esperando por processos SuperPascal (*spms*).

Para o término da máquina SuperPascal, um sinal é enviado a partir da máquina hospedeira para o *spm* que o envia para o primeiro *spms* e permanece esperando com a chamada *wait* do sistema *Crux*. Esse primeiro *spms*, envia um sinal de término para todos os demais nós e permanece esperando através da chamada de sistema *wait* do sistema *Crux*. Os demais nós terminam sua execução utilizando a chamada de sistema *exit* do sistema *Crux*. Após o término dos demais nós, o primeiro *spms* termina sua execução através da chamada de sistema *exit*. Finalmente, o processo *spm* executa a chamada de sistema *exit* para finalizar a máquina SuperPascal.

## 5.8 Ambiente de programação paralela no multicomputador Nó //

O ambiente de programação paralela para o Nó //, enriquecido pela implementação da linguagem SuperPascal, é composto por :

- √ **Ambiente de produção de programas** : Esse ambiente é representado por um editor de textos disponível nos sistemas Unix (*emacs*, por exemplo) e pelo compilador SuperPascal, chamado *scn*.
- √ **Ambiente de execução de programas** : Esse ambiente é representado pelo interpretador SuperPascal para a Máquina Unix, chamado *srn* e pelo sistema *Crux* alterado conforme 5.6.3 ou pelos interpretadores SuperPascal para a Máquina SuperPascal Paralela, chamados *spm* e *spms* e o sistema *Crux* alterado conforme 5.7.1.

# 6. Conclusões

Na implementação do ambiente de execução da linguagem SuperPascal para o multicomputador Nó //, o usuário não precisa conhecer a arquitetura da máquina destino para desenvolver seus programas de aplicação. O mecanismo de distribuição de processos e de canais de comunicação provido pelo ambiente de execução e implementado utilizando o sistema operacional *Crux* e o simulador do Nó //, estendidos por operadores específicos para a linguagem SuperPascal, provê o paralelismo ao usuário de forma automática.

A depuração é uma tarefa relevante quando se desenvolve programas paralelos. A dificuldade de depuração encontrada durante a implementação do ambiente, pela ausência de um depurador, foi compensada pela simplicidade do funcionamento do simulador do Nó // e do sistema operacional *Crux* ( [CAM95] e [MON95] ), utilizados como ambiente de implementação da linguagem SuperPascal.

O multicomputador Nó // mostrou-se adequado para implementar o ambiente de execução da linguagem SuperPascal porque a criação dinâmica de processos e de canais se adaptam corretamente à topologia dinâmica da máquina. A simplicidade e eficácia da implementação dessa linguagem em um multicomputador com rede de interconexão estática como, por exemplo, um hipercubo, ficariam comprometidas. Uma questão difícil para implementar linguagens como Joyce ou SuperPascal nessas máquinas seria, por exemplo, a localização da tabela de canais de comunicação que é uma estrutura compartilhada por todos os processos [AND89].

A facilidade de porte anunciada pelo autor do ambiente original, graças ao interpretador da linguagem, foi comprovada. Tanto assim que foi possível experimentar duas alternativas, que poderão ser comparadas mais tarde, quando a máquina real estiver disponível.

## 6.1. Contribuições

A implementação de uma linguagem paralela é um evento importante para o projeto Nó // já que um dos principais objetivos desse projeto é a obtenção de um ambiente de programação paralela em multicomputador. Dessa forma, o trabalho desenvolvido representa um avanço significativo no estado atual desse projeto.

Uma contribuição importante deste trabalho refere-se à definição dos mecanismos de distribuição de processos entre os nós e de comunicação entre eles a partir de um interpretador que executava originalmente como um único processo seqüencial em um sistema Unix.

A eficácia prevista desta implementação da linguagem SuperPascal no multicomputador Nó //, deriva, por exemplo, da localização da tabela de canais de comunicação no nó de controle (esse nó, além de ser responsável pela gerência de nós e conexões, passou a ser responsável também pela gerência dos canais de comunicação da linguagem SuperPascal).

Outra contribuição importante foi a extensão da camada das comunicações de alto nível do sistema operacional *Crux* pela definição e implementação de chamadas específicas para a linguagem SuperPascal.

## 6.2. Perspectivas Futuras

Por restrições de tamanho dos programas executados pelo simulador do Nó //, não foram implementadas todas as instruções do interpretador da máquina SuperPascal. A implementação das instruções faltantes, principalmente as que lidam com números reais, somente será viável na máquina real.

A modificação do compilador para gerar código nativo dos processadores da máquina real também pode ser um trabalho a ser considerado futuramente para aumentar o desempenho dos programas.

Outra tarefa importante a ser considerada é a avaliação do desempenho relativo das duas alternativas de implementação, o que somente será viável quando a máquina real estiver disponível.

# Referências


[AND75] Anderson, George A., Jensen, E. Douglas, *Computer Interconnection Structures : Taxonomy, Characteristics, and Examples*. ACM Computing Surveys, Vol. 7, No. 4, dezembro de 1975.

[AND89] Andersen, Birger, *Hypercube Experiments with Joyce*. Sigplan Notices, Vol. 24, No. 8, p. 13-22, agosto de 1989.

[AND91] Andrews, Gregory R., *Paradigms for process interaction in distributed programs*. In ACM Computing Surveys, Vol. 23, No. 1, p. 49-90, março de 1991.

[ATH88] Athas, William, Seitz, Charles L., *Multicomputers : Message-Passing Concurrent Computers*. 0018-9162/88/0800/0009 IEEE, agosto de 1988.

[ARI90] Ben-Ari, M., *Principles of Concurrent and Distributed Programming*. Prentice-Hall, 1990.

[AUS91] Austin, Paul, Murray, Kevin, Wellings, Andy, *The Design of an Operating System for a Scalable Parallel Computing Engine*. York (Inglaterra), Software-Practice and Experience, vol. 21(10), p. 989-1013, outubro de 1991.

[BAI88] Baillie, Clive F., *Comparing shared and distributed memory computers*. In Parallel Computing 8, p. 101-110, 1988.

[BAL89] Bal, H. E., Steiner, J. G., Tanenbaum, A. S., *Programming languages for distributed computing systems*. In ACM Computing Surveys, Vol. 21, p. 261-322, setembro de 1989.

[BRO83] Broomell, George, Heath, J. Robert, *Classification Categories and Historical Development of Circuit Switching Topologies*. Computing Surveys, Vol. 15, No. 2, junho de 1983.

[BUR88] Burns, Alan, *Programming in Occam 2*. Addison-Wesley, 1988.

[CSA90] Computer System Architects. *Occam2 Toolset - User manual*. 1990.

[CAM95] Campos, R. A., *Um sistema operacional fundamentado no modelo cliente-servidor e um simulador multiprogramado de multicomputador*. Dissertação de Mestrado, CPGCC-UFSC, Florianópolis, maio de 1995.

[COM84] Comer, D., *Operating System Design : The XINU Approach*. Prentice-Hall, 1984.

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

[COS93] Costa, Celso Maciel da, *Environnement D'Exécution Parallèle : Conception et Architecture*. These pour obtenir le titre de Docteur de l'Universite Joseph Fourier, Grenoble I, 1993.

[CRI88] Crichlow, J. M., *An Introduction to Distributed and Parallel Computing*. Prentice-Hall, 1988.

[DOW88] Dowsing, Roy D., *Introduction to Concurrency using Occam*. Van Nostrand Reinhold (International), 1988.

[FEN81] Feng, Tse-yun, *A Survey of Interconnection Networks*. The Ohio State University, IEEE, EHO217-0/84/0000/0005, 1981.

[FER75] Ferreira, Aurélio Buarque de Holanda, *Novo dicionário da língua portuguesa*. Editora Nova Fronteira, Rio de Janeiro, 1975.

[GEI94] Geist, Al, Beguelin, Adam, Dongarra, Jack Jiang, Weicheng, Manchek, Robert, and Sunderam, Vaidy, *PVM : Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[HAN85] Brinch Hansen, P., *Brinch Hansen on Pascal Compilers*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[HAN87a] Brinch Hansen, P., *Joyce - A Programming Language for Distributed Systems*. Software - Practice and Experience, vol. 17(1), p. 29-50, janeiro de 1987.

[HAN87b] Brinch Hansen, P., *A Joyce Implementation*. Software - Practice and Experience, vol. 17(4), p. 267-276, abril de 1987.

[HAN89a] Brinch Hansen, P., *The Joyce Language Report*. Software - Practice and Experience, vol. 19(6), p. 553-578, junho de 1989.

[HAN89b] Brinch Hansen, P., *A Multiprocessor Implementation of Joyce*. Software - Practice and Experience, vol. 19(6), p. 579-592, junho de 1989.

[HAN93a] Brinch Hansen, P., *Monitors and Concurrent Pascal : A Personal History*. School of Computer and Information Science, Syracuse University, Syracuse, NY, 1993.

[HAN93b] Brinch Hansen, P., *The SuperPascal software notes*. School of Computer and Information Science, Syracuse University, Syracuse, NY, 1993.

[HAN94a] Brinch Hansen, P., *SuperPascal - a publication language for parallel scientific computing*. Concurrency - Practice and Experience, vol. 6(5), p. 461-483, agosto de 1994.

[HAN94b] Brinch Hansen, P., *The Programming Language SuperPascal*. Software - Practice and Experience, vol. 24(5), p. 467-483, maio de 1994.

[HAN94c] Brinch Hansen, P., *Interference control in SuperPascal - a block-structured parallel language*. Computer Journal, vol. 37(5), p. 399-406, maio de 1994.

[HOA78] Hoare, C. A. R., *Communicating Sequential Processes*. Communications of the ACM, vol. 21, n. 8, p. 666-677, agosto de 1978.

[HWA87] Hwang, Kai, *Advanced Parallel Processing with Supercomputer Architectures*. Proceedings of the IEEE, Vol. 75, No. 10, outubro de 1987.

[HWA93] Hwang, Kai, *Advanced computer architecture : parallelism, scalability, programmability*. McGraw-Hill, 1993.

[INM84] INMOS Limited. *Occam Programming Manual*. London : Prentice-Hall, 1984.

[JUN89] Jung, Chul-Doo, Silbert, Ernest, *Indirect Naming in Distributed Programming Languagens*. Sigplan Notices, Vol. 24, No. 9, p. 126-132, setembro de 1989.

[LIM94] Crisóstomo, V. Lima, *Um mecanismo de comunicação e um método de ativação de servidores para um sistema operacional*. Dissertação de Mestrado, CPGCC-UFSC, Florianópolis, agosto de 1994.

[MIL93] Milaneze, Patrícia, *Canais compartilhados para comunicação entre processos usuários do Minix*. Trabalho de conclusão de curso, Curso de Ciências da Computação - UFSC, Florianópolis, agosto de 1993.

[MON95] Montez, C. B., *Um sistema operacional com micronúcleo distribuído e um simulador multiprogramado de multicomputador*. Dissertação de Mestrado, CPGCC-UFSC, Florianópolis, maio de 1995.

[PET85] Peterson, J. L., Silberschatz, A., *Operating Systems Concepts*. Addison-Wesley, 1985.

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

[SIE79] Siegel, Howard J., Mcmillen, Robert J., Mueller, Philip T. Jr. *A Survey of Interconnection Methods for Reconfigurable Parallel Processing Systems*. Purdue University, West Lafayette, Indiana, National Computer Conference, 1979.

[TAN92] Tanenbaum A. S., *Modern Operating Systems*. Prentice-Hall, 1992.

[VAR87] Varma A., Sawchuk, A. A., Jenkins, B. K., Raghavendra, C. S., *Optical Crossbar Networks*. Computer, 0018-9162/87/0600-0050, IEEE, junho de 1987.