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

Alexssandro Cardoso Antunes

IMPLEMENTAÇÃO DAS ROTINAS BÁSICAS DE COMUNICAÇÃO PONTO-A-PONTO DA INTERFACE DE PASSAGEM DE MENSAGENS NO MULTICOMPUTADOR ACRUX

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

Prof. Dr. José Mazzucco Júnior.

Florianópolis, Julho de 2003.

IMPLEMENTAÇÃO DAS ROTINAS BÁSICAS DE COMUNICAÇÃO PONTO-A-PONTO DA INTERFACE DE PASSAGEM DE MENSAGENS NO MULTICOMPUTADOR ACRUX

Alexssandro Cardoso Antunes

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____

Fernando Álvaro Ostuni Gauthier, Dr.
Coordenador do CPGCC

Banca Examinadora

_____

José Mazzucco Júnior, Dr.
Orientador

_____

Luis Fernando Friedrichi, Dr.
Membro

_____

Mário Antônio Ribeiro Dantas, Dr.
Membro

"A coisa mais nobre que podemos experimentar é o mistério. Ele é a emoção fundamental, paralela ao berço da verdadeira ciência."
(Albert Einsten).

"A dúvida é um dos nomes da inteligência."
Jorge Luís Borges (Escritor Argentino).

Aos professores Thadeu Botteri Corso e José Mazzucco Júnior, pelas orientações em todas as fases deste trabalho.

Ao Corpo Docente da Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina, que contribuíram para o meu aperfeiçoamento pessoal e profissional.

... E especialmente:

Aos meus pais, pilares fundamentais na formação de meu caráter.
A minha noiva, pelo apoio e tempo que lhe foi roubado por culpa desta dissertação.

... em tempo, á Deus, origem e final de tudo.

Sumário

Lista de Figuras

Resumo

O presente trabalho de pesquisa está inserido no âmbito do projeto Crux, desenvolvido no ambiente do LaCPaD – CPGCC da UFSC, cujo objetivo é a construção de um cluster de computadores constituídos por um conjunto de estações de trabalho comuns e independentes, interligados por uma rede de interconexão dinâmica. Este Multicomputador, oferece um ambiente para a execução de programas paralelos organizados como redes de processos comunicantes.

A proposta aqui apresentada envolve computação paralela sobre sistemas distribuídos, através de um mecanismo de troca de mensagens. O objetivo principal deste trabalho é identificar as estruturas de comunicação e sincronização da biblioteca de passagem de mensagens - MPI, visando aspectos de portabilidade das rotinas básicas de comunicação ponto-a-ponto (*bloqueante síncrona*) da implementação MPICH para o Multicomputador ACrux.

Desenvolvido para o sistema operacional Linux, estes mecanismos de comunicação permitem o envio e/ou recebimento de uma mensagem MPI entre nós de trabalho neste cluster, mantendo a transparência de acesso do modelo tradicional para o usuário final.

Palavras-chave: Cluster de Computadores, Interface de Passagem de Mensagens, Comunicação bloqueante síncrona.

Abstract

The present study is inserted in the scope of the project Crux that was developed in the LaCPaD's Environment – UFSC's CPGCC, whose goal is the construction of a cluster of computers constituted by a set of standard and independent workstations, an interconnection network dynamic interlinks them. This Multicomputer offers an environment for the execution of organized parallels programs as communications processes network.

The purpose presented here involves parallel computation about distributed systems through messages change mechanism. The main goal of this study is to identify the communication and synchronization structures of the library on the way of messages – MPI. In order to aim portable aspects of the basic routines of point by point communication (synchronizing blocking) of the MPICH implementation for Crux Multicomputer.

Which was developed for Linux operating system, these communication mechanisms allow sending or receiving MPI message between hosts works in this cluster, keeping the access transparency of the traditional model for the end user.

Keywords: Cluster of Computers, Message Passing Interface, Synchronizing blocking communication.

1 Introdução

A computação paralela de alto desempenho é atualmente uma necessidade fundamental em muitas aplicações que envolvem processamento de algoritmos complexos e/ou grande volume de dados. O princípio básico de um cluster é conectar estações comuns de trabalho de forma que elas se comportem como um único computador, para que possa atingir desempenho compatível com os dos grandes sistemas paralelos especialistas, com custos e complexidade inferiores, tanto em sua construção (hardware) como na programação (software).

Neste capítulo, são apresentadas as motivações e os objetivos que levaram a elaboração de um ambiente para a execução de programas paralelos organizados como redes de processos comunicantes. Este ambiente de programação paralela, consolidou-se através do uso otimizado da popular biblioteca de passagem de mensagens MPICH do padrão MPI, que utiliza em uma camada inferior o sistema operacional distribuído ACrux [BUD, 2002], ambos objetos de projetos correntes no ambiente do Laboratório de Computação Paralela e Distribuída (LaCPaD) do Curso de Pós-Graduação em Ciência da Computação (CPGCC) da Universidade Federal de Santa Catarina (UFSC).

A arquitetura do Multicomputador ACrux, foi baseado na arquitetura original do Multicomputador Crux [COR, 1999], que é um mecanismo de comunicação genérico com conexão dinâmica de canais físicos por demanda do mapeamento de canais lógicos, visando o paralelismo real. Este Multicomputador difere das soluções

baseadas em clusters existentes atualmente, trazendo algumas inovações vantajosas para o processamento paralelo e distribuído, tais como:

- A conexão dinâmica de canais físicos diretos, que são executadas entre nós de trabalho (rede de trabalho), onde a troca de mensagens só ocorre de acordo com a demanda das tarefas que estão sendo executadas.
- Comunicação através de troca de mensagens realizadas por operadores próprios.
- Administração das conexões diretas entre nós de trabalho por um servidor de comunicações (rede de controle).

O objetivo principal deste trabalho é a identificação das estruturas de comunicação e sincronização da biblioteca de passagem de mensagens MPI, visando a portabilidade das rotinas básicas de comunicação ponto-a-ponto para este cluster, utilizando estas primitivas na camada da rede de trabalho.

Este texto é constituído de sete capítulos, os quais têm o seu conteúdo básico listados a seguir:

- O capítulo 2 contém alguns fundamentos de computação paralela e distribuída julgados necessários como revisão de literatura, bem como um estudo da arquitetura Crux.
- O capítulo 3 ilustra a realização da computação paralela sobre sistemas distribuídos através de ambientes de passagem de mensagens, introduzindo alguns exemplos destas interfaces.

- O capítulo 4 se dedica a expor o conhecimento adquirido na utilização de clusters de computadores, a descrição de seus componentes, suas áreas de aplicação, e apresenta alguns projetos de *clusters*.
- O capítulo 5 apresenta a biblioteca de passagem de mensagens escolhida para o desenvolvimento deste trabalho, abordando as rotinas básicas de comunicação ponto-a-ponto da implementação MPICH.
- O capítulo 6 descreve o processo em si, isto é, os detalhes de implementação do multicomputador ACrux, enfocando os aspectos de instalação e/ou configuração deste cluster, bem como a alteração das chamadas de sistema para uma operação *bloqueante síncrona*, apresentado o resultado final da portabilidade.
- O capítulo 7 apresenta as conclusões, sugestões e perspectivas futuras. Como complemento, as tendências e a proposta de possíveis extensões deste trabalho, e novas direções a serem tomadas a partir dos resultados obtidos.

Para complementar as informações contidas na dissertação, ao final da mesma se apresentam os anexos, onde se encontram todos os scripts e os códigos fontes necessários para execução deste ambiente proposto.

2 Fundamentos de Computação Paralela e Distribuída

2.1 Considerações Iniciais

A literatura apresenta diversas definições para computação paralela (ou processamento paralelo), dentre as quais cabe citar a sugerida por Almasi [ALM, 1994]: "computação paralela constitui-se de uma coleção de elementos de processamento que se comunicam e cooperam entre si e com isso resolvem um problema de maneira mais rápida", e a apresentada por Quinn [QUI, 1987]: "computação paralela é o processamento de informações que enfatiza a manipulação concorrente dos dados, que pertencem a um ou mais processos que objetivam resolver um único problema".

As arquiteturas seqüenciais de von Neuman (tradicionalmente aceitas) conseguiram ao longo dos últimos anos um grande avanço tecnológico, sendo largamente aprimoradas no decorrer deste tempo, mas estas arquiteturas ainda demonstram deficiências quando utilizadas por aplicações que necessitam de maior poder computacional, algumas destas áreas de aplicações serão descritas no capítulo 4 (seção 4.4).

Dentre os principais motivos para o surgimento da computação paralela, destaca-se a necessidade de aumentar o poder de processamento em uma única máquina. Segundo [SOU, 1996], a computação paralela apresenta diversas vantagens em relação à seqüencial, tais como:

- alto desempenho para programas mais lentos;
- soluções mais naturais para programas intrinsecamente paralelos;
- maior tolerância a falhas;
- modularidade.

Apesar das vantagens apresentadas anteriormente, existem alguns fatores negativos na utilização da computação paralela que devem ser considerados:

- maior dificuldade na programação;
- necessidade de balanceamento de cargas para melhor distribuição dos processos nos vários processadores;
- intensa sobrecarga no sistema (por exemplo, na comunicação entre processos), que impede que se tenha um *speedup* ideal;
- necessidade de sincronismo e comunicação entre processos, o que gera certa complexidade.

2.2 Arquiteturas de Máquinas Paralelas

Uma arquitetura paralela, conforme Duncan [DUN, 1990], fornece uma estrutura explícita e de alto nível para o desenvolvimento de soluções utilizando o processamento paralelo, através da existência de múltiplos processadores, estes simples ou complexos, que cooperam para resolver problemas através de execução concorrente. As máquinas paralelas são formadas por nós processadores interligados fisicamente através de uma rede de interconexão, cujo objetivo é proporcionar grande poder computacional através da cooperação de processadores na execução de eventos

concorrentes, oferecendo confiabilidade e uma boa relação custo/desempenho, suprindo a necessidade de um processamento intensivo.

Existem muitas maneiras de se organizar computadores paralelos. Para que se possa visualizar melhor o conjunto de possíveis opções de arquiteturas paralelas, é interessante classificá-las. Segundo Ben-Dyke [BEN, 1993], uma classificação ideal deve ser:

Hierárquica: iniciando em um nível mais abstrato, a classificação deve ser refinada em subníveis à medida que se diferencie de maneira mais detalhada cada arquitetura;

Universal: um computador único, deve ter uma classificação única;

Extensível: futuras máquinas que surjam, devem ser incluídas sem que sejam necessárias modificações na classificação;

Concisa: os nomes que representam cada uma das classes devem ser pequenos para que a classificação seja de uso prático;

Abrangente: a classificação deve incluir todos os tipos de arquiteturas existentes.

Na construção de máquinas paralelas, o projeto pode diferir em diversos pontos, devido a diversidade de modelos propostos e dos critérios adotados. Contudo, essas possibilidades de construção (mecanismos de acesso a memória, forma de controle das unidades de processamento e maneiras de conectar os componentes), levaram Corso [COR, 1999] a sugerir uma possível classificação geral para a arquitetura dos computadores (figura 2.1).

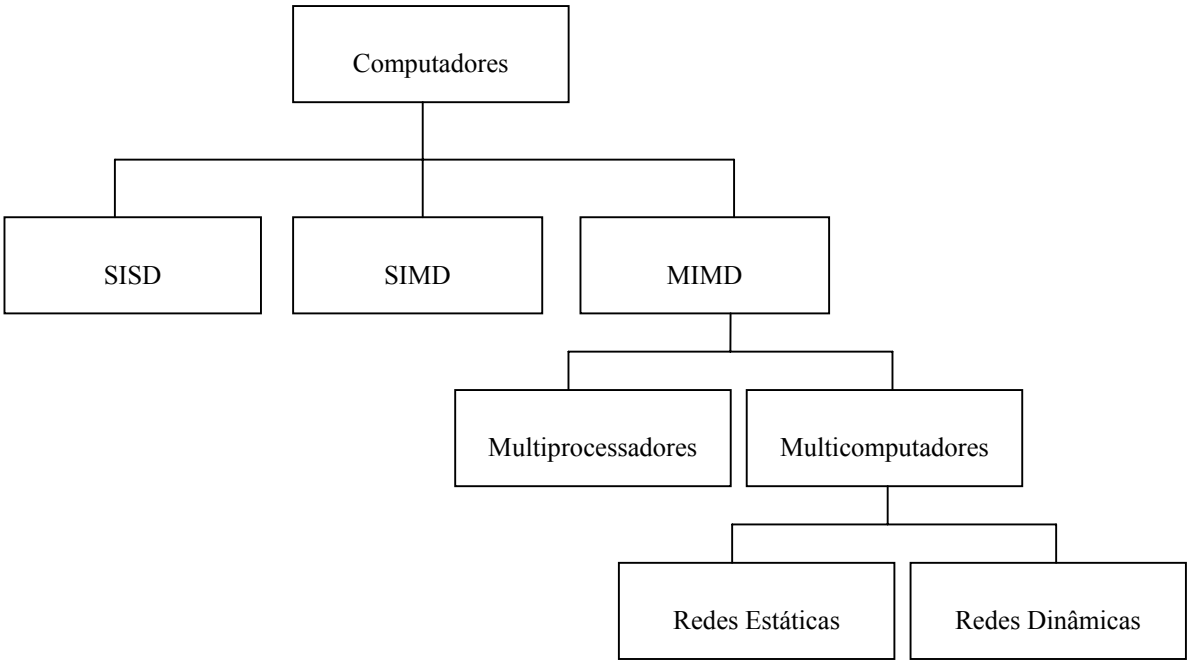

Figura 2.1 - Classificação geral para as arquiteturas de computadores.

Muito já foi desenvolvido em termos de *hardware* paralelo, e várias classificações foram propostas [ALM, 1994] [BEN, 1993] [DUN, 1990]. A mais utilizada e aceita pela comunidade computacional é a classificação de Flynn [FLY, 1972].

2.2.1 Classificação de Flynn

A classificação proposta por Flynn [FLY, 1972], embora muito antiga, é amplamente adotada e baseia-se no fluxo de instruções e no fluxo de dados (*o fluxo de instruções eqüivale a uma seqüência de instruções executadas em um processador*

sobre um fluxo de dados aos quais estas instruções estão relacionadas) para classificar um computador em quatro categorias:

- SISD (*Single Instruction Stream/Single Data Stream*) - Fluxo único de instruções/Fluxo único de dados: corresponde ao tradicional modelo von Neumann (*máquinas possuem uma unidade de processamento e uma unidade de controle*). Um processador executa seqüencialmente um conjunto de instruções sobre um conjunto de dados. Mais difundido em computadores comerciais e/ou convencionais.
- SIMD (*Single Instruction Stream/Multiple Data Stream*) - Fluxo único de instruções/Fluxo múltiplo de dados: envolve múltiplos processadores (escravos) sob o controle de uma única unidade de controle (mestre), executando simultaneamente a mesma instrução em diversos conjuntos de dados. Arquiteturas SIMD são utilizadas, por exemplo, para manipulação de matrizes e processamento de imagens, ou seja, aplicações que geralmente repetem o mesmo cálculo para diferentes conjuntos de dados.
- MISD (*Multiple Instruction Stream/Single Data Stream*) - Fluxo múltiplo de instruções/Fluxo único de dados: envolve múltiplos processadores executando diferentes instruções em um único conjunto de dados. Geralmente, nenhuma arquitetura é classificada como MISD, isto é, de difícil aplicação prática. Alguns autores consideram arquiteturas *pipeline* como exemplo deste tipo de organização.

- MIMD (*Multiple Instruction Stream/Multiple Data Stream*) - Fluxo múltiplo de instruções/Fluxo múltiplo de dados: envolve múltiplos processadores executando diferentes instruções em diferentes conjuntos de dados, de maneira independente. Este modelo abrange a maioria dos computadores paralelos e os sistemas distribuídos, ou seja, mais adequado para as aplicações de uso geral.

2.2.2 Classificação baseada no acesso a memória

A categoria MIMD engloba uma grande diversidade de máquinas paralelas, no qual, uma das formas mais usadas de classificá-las é o critério baseado no acesso a memória, isto é, a forma como a memória central esta fisicamente organizada e o tipo de acesso que cada processador tem a totalidade da memória, classificando esta arquitetura em computadores MIMD de memória compartilhada e memória distribuída [ALV, 2002].

Na classe de arquitetura MIMD de memória compartilhada incluem-se as máquinas com múltiplos processadores que compartilham um espaço de endereços de memória comum (máquinas multiprocessadas). Os *multiprocessadores* caracterizam-se pela existência de uma memória global e única, a qual é utilizada por todos os processadores (hardware fortemente acoplados), os processos comunicam-se via memória compartilhada.

Contudo, na outra classe de arquitetura MIMD de memória distribuída incluem-se as máquinas formadas por várias unidades processadoras (nós), cada uma

com a sua própria memória privativa (hardware fracamente acoplados), também conhecidos como *multicomputadores*. Em virtude de não haver compartilhamento de memória, os processos comunicam-se via troca de mensagens. No ambiente físico desse trabalho, os multicomputadores destacam-se como unidades alvo.

2.2.3 Multicomputadores

Um multicomputador é formado por vários computadores (nós autônomos), unidades constituídas por um processador, memória local, dispositivos de entrada/saída (não necessariamente) e de suporte físico para a comunicação com outros nós, ligados por uma rede de interconexão, conforme a figura abaixo [COR, 1999].

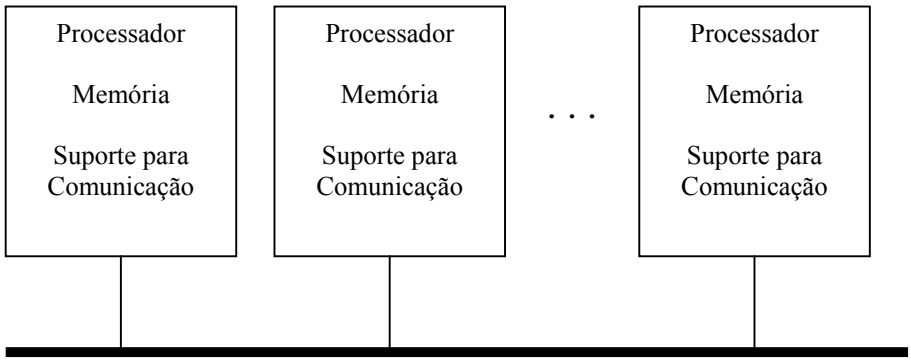

Figura 2.2 - Multicomputador.

Neste tipo de arquitetura, cada computador executa seu próprio programa, e o mesmo acessa a memória local diretamente e pode enviar e receber mensagem sobre a rede de comunicação. As mensagens são usadas para a comunicação entre um processador e outro ou para ler e escrever na memória remota. O custo para acessar a memória local, geralmente é menor que para acessar a memória remota. Todavia, o

custo de enviar uma mensagem é independente da localização do nó e de outros tráfegos na rede de comunicação, mas depende do tamanho da mensagem.

Neste ponto, cabe ressaltar um fator que pode prejudicar o desempenho desses sistemas (redes físicas dos multicomputadores e dos mecanismos de comunicação), o tempo de latência de rede, conseqüência da duração da troca de mensagens.

Entre algumas características dos multicomputadores, que servem para diferenciá-los de outras máquinas paralelas, tornando a sua utilização mais difundida, destacam-se:

- Grande número de nós homogêneos;
- Proximidade física dos nós;
- Comunicação entre os processos através da troca de mensagens;
- Alto grau de paralelismo real.

Em um multicomputador [JUN, 1999], a interação entre os seus nós (processadores autônomos com memórias privadas, de onde são obtidas suas instruções e dados) é alcançada através do envio de mensagens que trafegam pelo sistema por meio de uma rede de interconexão de alta velocidade. As características funcionais dessas redes, representam fatores determinantes no desempenho de um sistema multicomputador, podendo ser classificadas de acordo com a natureza das redes:

- **Redes estáticas**: rede cuja topologia não pode ser alterada. Apresenta uma topologia fixa, onde as ligações entre os seus nós processadores são

estabelecidas na construção do sistema e não podem ser reconfiguradas. Os canais são permanentes, podendo haver desperdício de banda passante, uma vez que muitas estações não vão enviar mensagens nos intervalos a ela destinados.

- **Redes dinâmicas**: rede cuja topologia pode ser alterada, através de programação de chaves. Não possuem uma topologia fixa, como as estáticas. As ligações entre os seus nós processadores são estabelecidas por meio de comutadores de conexões (circuitos de chaveamento eletrônico), cuja manipulação permite a atribuição de canais físicos temporários que podem ser criados de acordo com as características de comunicação de um determinado problema [JUN, 1999], ou seja, a alocação do canal é realizada por demanda de envio de mensagens.

As redes de interconexão dinâmicas fundamentais para o multicomputador alvo deste trabalho são o *barramento* e o *crossbar*.

Barramento

Este tipo de topologia (figura 2.3) é bastante semelhante ao conceito de arquitetura de barra em um sistema de computador, onde todas as estações (nós) se ligam ao mesmo meio de transmissão [SOA, 1997]. É um modelo bastante flexível, de conexão simples e de baixo custo, pois permite adição e subtração de nós, sem a necessidade de efetuar profundas alterações no sistema. Uma das principais desvantagens deste modelo, está relacionada com a extensibilidade, ou seja, este tipo de rede não funciona muito bem com um grande número de nós devido a sua baixa

capacidade de transmissão, e outra relacionada com o fator de tolerância a falhas e/ou faltas, isto é, uma interrupção da comunicação entre todos os nós, decorrente de uma eventual falha no *barramento* [JUN, 1999]. O desempenho do sistema pode aumentar significativamente, se as comunicações através do *barramento* não forem freqüentes.

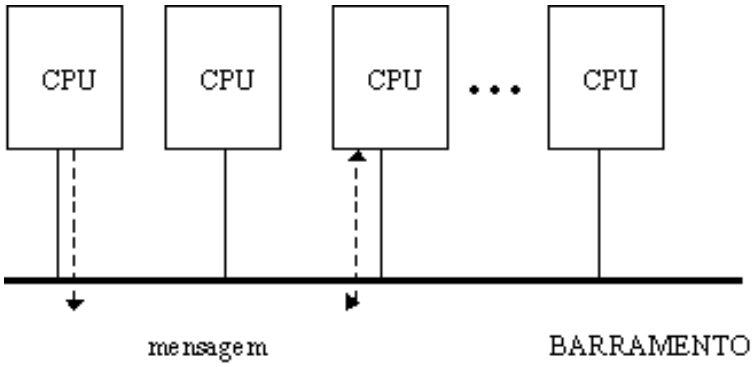

Figura 2.3 - Multicomputador baseado em barramento.

Crossbar

Um *crossbar* N x M possui N entradas dispostas horizontalmente e M saídas dispostas verticalmente, construindo uma rede do tipo grelha, na qual a interseção de duas linhas representa um comutador, que controla a conexão entre a entrada e a saída correspondentes. Na figura 2.4, é destacada uma conexão entre a entrada *E4* e a saída *S3*, resultante do fechamento do comutador correspondente. Nessa situação, o estabelecimento de uma nova conexão, envolvendo a entrada *E4* e a saída *S3*, não seria possível [JUN, 1999]. O *Crossbar* permite a conexão de quaisquer entradas N a quaisquer saídas M, onde cada linha de entrada ou saída pode participar de apenas uma conexão por vez. Este modelo é bastante poderoso, sem contenda (*conflitos*) quando comparada ao *barramento*, porém uma desvantagem que merece destaque está

relacionada com o seu crescimento, isto é, o seu custo cresce consideravelmente quando aumenta muito o seu tamanho, tornando-se uma rede muito cara.

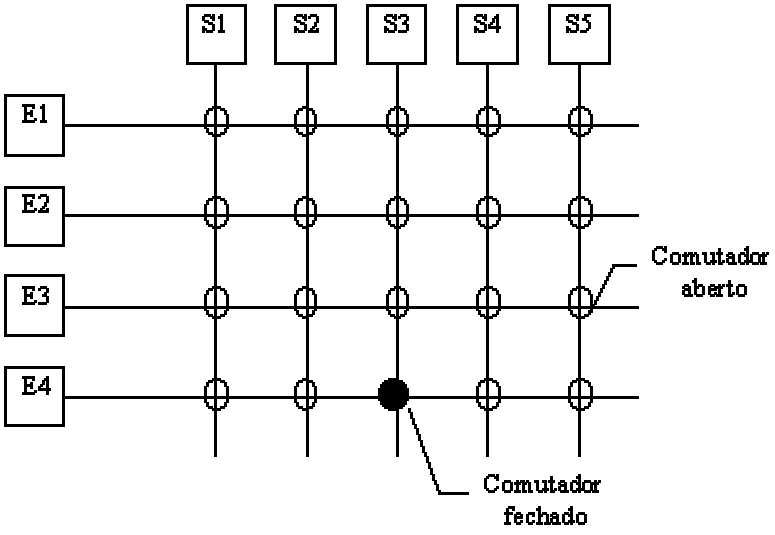

Figura 2.4 - Dispositivo Crossbar.

2.2.4 O Muticomputador Crux

O Multicomputador Crux (figura 2.5) é um mecanismo de comunicação genérico com conexão dinâmica de canais físicos por demanda do mapeamento de canais lógicos [COR, 1999], esta arquitetura resulta da reunião dos seguintes componentes:

- ***Nós de Trabalho***: computadores que possuem um processador, memória privativa e um *hardware* de suporte à comunicação, que executam os processos de programas paralelos;
- ***Rede de Trabalho***: um *crossba*r utilizado para transportar mensagens de tamanho arbitrário através de canais físicos diretos, que são criados pela rede de controle, entre pares de nós de trabalho;

- ***Nó de Controle***: utilizado para a configuração da rede de trabalho, é responsável pela conexão e desconexão dos canais físicos diretos entre os nós de trabalho, em resposta a solicitação dos nós de trabalho;
- ***Rede de Controle***: um *barramento* utilizado para transportar pequenas mensagens de controle entre os nós de trabalho e o nó de controle, para a gerência das conexões dos canais diretos.

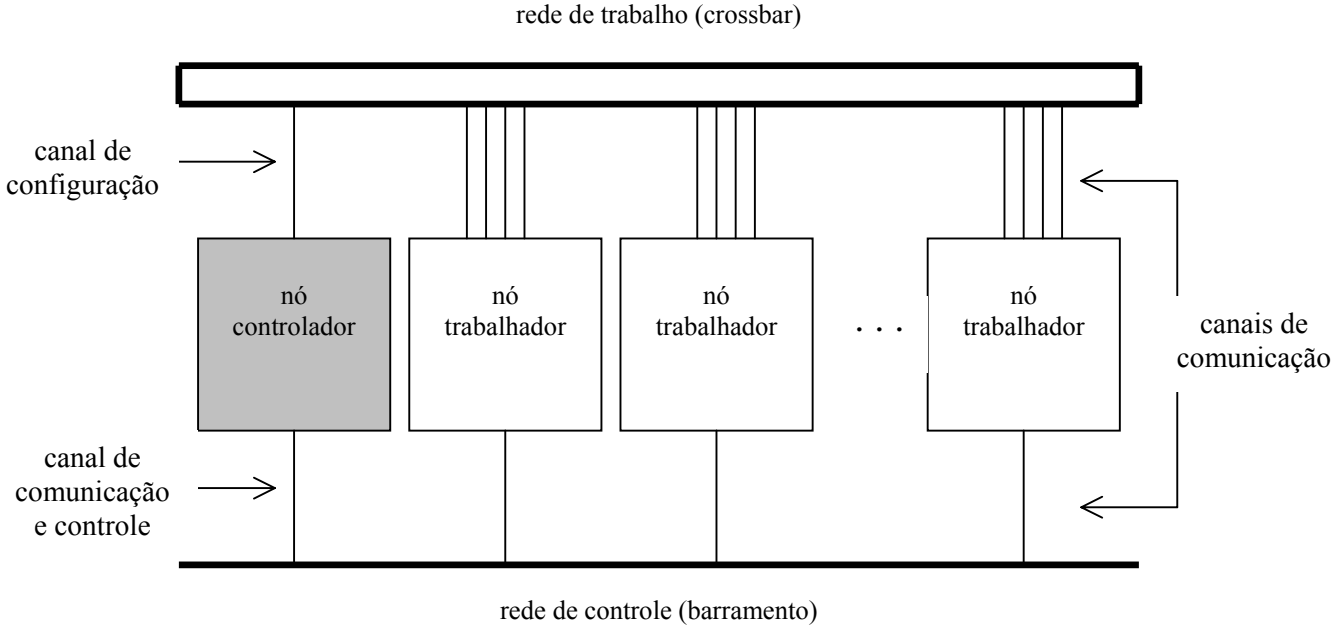

Figura 2.5 - Arquitetura do Multicomputador Crux.

Neste ambiente, intitulado Crux [COR, 1999], a rede de trabalho é usada para transportar mensagens de tamanho arbitrário através de canais físicos diretos entre dois nós trabalhadores e a rede de controle é usada exclusivamente para transportar mensagens de controle entre um nó trabalhador e o nó controlador. Esta arquitetura de multicomputador (figura 2.5), pode possuir um número infinito de nós, onde cada nó é conectado a rede de trabalho (*crossbar*) por quatro canais de comunicação e a rede de

controle (*barramento*) por um canal de comunicação, podendo haver quatro canais físicos diretos conectados e trocando mensagens simultaneamente.

A conexão dinâmica de canais físicos, só ocorre se os dois nós envolvidos na comunicação desejarem a conexão, ou seja, os mesmos devem estar dispostos a trocar mensagens. Antes do inicio da troca de mensagens, deve ser verificada a existência de um canal físico direto entre os respectivos nós trabalhadores, através da rede de trabalho. Caso exista um canal disponível, a transmissão é iniciada imediatamente, caso contrário, o nó trabalhador solicita um pedido de conexão através de uma mensagem enviada ao nó de controle pela rede de controle, que espera uma confirmação em forma de mensagem do nó de controle pela mesma rede de controle. O nó de controle cria os canais físicos diretos que serão utilizados pelos nós trabalhadores, envia mensagens de controle, e também recebe os pedidos de desconexão dos nós de trabalho.

Assim, cada nó trabalhador possui localmente informações sobre o estado das conexões de seus canais físicos [COR, 1999], todavia, o nó controlador, deve estar permanentemente disponível e acessível aos nós trabalhadores, possuindo ainda informações sobre a situação de todos os nós do multicomputador Crux.

2.3 Sistemas Computacionais Distribuídos

No início da década de 70, com o aparecimento das tecnologias de redes de computadores com maior desempenho e maior confiabilidade, foi possível o desenvolvimento dos sistemas computacionais distribuídos. Com o avanço tecnológico

das redes de comunicação e o crescente aumento da potência computacional dos computadores pessoais e das estações de trabalho, vários projetos foram desenvolvidos tornando os sistemas distribuídos mais eficazes e difundidos. Entre os exemplos desses projetos, destacam-se: Amoeba, Mach, Chrous e o TRICE [TAN, 1997] [COL, 1994] [MUL, 1993].

Um sistema computacional distribuído, conforme Tanenbaum [TAN, 1997] e Colouris [COL, 1994], é uma coleção de computadores autônomos, interligados por uma rede de comunicação e equipados com um sistema operacional distribuído, que permitem o compartilhamento transparente de recursos existentes no sistema, isto é, para o usuário final não é visível a existência de múltiplos recursos. Esta abordagem descreve o *sistema operacional distribuído* como o elemento responsável por manter as características necessárias utilizando como meio de comunicação a rede e o *software* como elemento determinante, o que caracteriza um sistema distribuído. A proposta de um *sistema operacional distribuído* é fazer com que este conjunto de máquinas interligadas pareça para seus usuários como se fosse uma única máquina, onde os mesmos (usuários) não tomam conhecimento de onde seus programas estão sendo processados e de onde seus arquivos estão armazenados. Esta proposta difere de um *sistema operacional de rede*, onde o usuário sabe em que máquina ele está, sendo necessário saber a localização de um recurso disponível para poder utilizá-lo, ou seja, os mesmos sabem da existência das várias máquinas da rede, podem abrir sessões em máquinas remotas e transferir dados de uma máquina remota para a máquina local.

Outra definição feita por Mullender [MUL, 1993], é o fato de que um sistema distribuído não deve ter pontos críticos de falha, característica que faz com um sistema distribuído leve vantagem em relação a um sistema centralizado.

Os sistemas distribuídos apresentam inúmeras vantagens, que os tornam mais difundidos: compartilhamento de recursos (componentes de software e hardware que estão disponíveis para a utilização do usuário), flexibilidade (crescimento, expansão da capacidade computacional do sistema, sistema aberto), confiabilidade (disponibilidade do sistema), performance (poder de computação total maior), escalabilidade (sistema se comporta da mesma forma independente do número de máquinas) e transparência (visão de um sistema único), sendo esta última uma característica fundamental e compartilhada pela maioria dos autores [TAN, 1997] [COL, 1994] [MUL, 1993]. Tolerância a falhas (ou faltas) e concorrência são também abordados por [COL, 1994] [MUL, 1993].

Por outro lado, os sistemas distribuídos apresentam também algumas desvantagens, tais como: carência de software para sistemas distribuídos, surgimento de "gargalos" representados pelas redes de comunicação (*podem saturar*) e falta de segurança (*fácil acesso*) apresentada pelo compartilhamento de dados e grande número de usuários.

2.4 Considerações Finais

As diversas áreas na qual a computação se aplica demandam cada vez mais poder computacional, esta necessidade contribuiu consideravelmente para o surgimento

da computação paralela. Como o avanço tecnológico e consequentemente o aumento do desempenho nas arquiteturas de von Neumann (*filosofia seqüencial*) é uma tarefa árdua e cara, imposta pela limitação tecnológica, a computação paralela compõe uma alternativa mais barata, elegante e aprimorada, principalmente para problemas essencialmente paralelos [SOU, 1996].

Classificações de arquiteturas surgiram em conseqüência do desenvolvimento de várias máquinas paralelas, sendo que a classificação proposta por Flynn, embora antiga, ainda é bastante utilizada e aceita pela comunidade computacional, nos quais são considerados os fluxos de instruções e de dados, criando-se quatro grupos: SISD, SIMD, MISD e MIMD [FLY, 1972]. A classificação de Duncan têm por objetivo resolver os problemas da classificação de Flynn, utilizando termos mais genéricos, classificando as arquiteturas em síncronas e assíncronas [DUN, 1990].

O desempenho dos computadores pessoais e das estações de trabalho vem aumentando significativamente nos últimos anos, sendo assim, quando interligados por uma rede de alta velocidade e alta confiabilidade, podem ser aplicados para solucionar uma variedade de aplicações que necessitam de alto poder computacional, ambiente este almejado para um usuário final do Multicomputador ACrux .

Os sistemas computacionais distribuídos tornaram-se muito populares e muitas linhas de pesquisa têm sido desenvolvidas com o objetivo de melhorar cada vez mais o compartilhamento de recursos com transparência, desempenho e confiabilidade.

Estas definições são suficientemente vagas para levantar a habitual questão do que é que distingue um sistema paralelo de um sistema distribuído. A fronteira é difícil de traçar, mas tem a ver com os objetivos que cada um deles pretende atingir. Num sistema paralelo, a ênfase é colocada no desempenho da execução, enquanto num sistema distribuído o objetivo é suportar um conjunto de serviços de forma transparente à sua localização, preocupando-se com questões como a tolerância a falhas, a proteção e autenticação de clientes e servidores. Esta maior funcionalidade dos modelos de programação dos sistemas distribuídos penaliza-os, normalmente, em termos de desempenho.

A utilização de sistemas computacionais distribuídos, com conceitos utilizados pela computação paralela, fez com que trabalhos fossem desenvolvidos para explorar o grande potencial da computação paralela utilizando os sistemas distribuídos. Baseando-se nesses princípios, ambientes de passagem de mensagem foram aperfeiçoados e/ou criados, que serão descritos no próximo capítulo.

3 Computação Paralela sobre Sistemas Distribuídos

3.1 Considerações Iniciais

Inicialmente, um dos motivos principais para o surgimento e posterior desenvolvimento dos sistemas distribuídos, foi a necessidade de se compartilhar recursos, normalmente de alto custo e separados fisicamente.

Entretanto, a computação paralela, teve como objetivo fundamental aumentar o desempenho observado na implementação de problemas específicos [ZAL, 1991].

Embora as duas áreas tenham surgido por razões diferentes, devido principalmente ao avanço tecnológico e às linhas de pesquisa ocorridos a partir da década de 80, atualmente percebe-se um forte inter-relacionamento entre as duas áreas, caracterizado por muitos aspectos em comum.

Consequentemente, devido às características em comum, vários trabalhos foram desenvolvidos com o objetivo de utilizar os sistemas distribuídos para computação paralela. Nesse sentido, a idéia básica é ter um grupo de computadores interligados por uma rede de comunicação, funcionando como os elementos de processamento (nós) de uma máquina paralela.

A realização da computação paralela sobre sistemas distribuídos é possível através de ambientes de passagem de mensagens (*ou interfaces de passagem de mensagens*). Esses ambientes têm sido aperfeiçoados nos últimos anos para serem utilizados por uma grande quantidade de equipamentos heterogêneos e/ou homogêneos, ganhando com isso popularidade e aceitação perante a comunidade computacional, visto que proporcionam um ambiente para o desenvolvimento de aplicações paralelas a um custo relativamente baixo em relação às máquinas paralelas [BEG, 1994].

3.2 Passagem de Mensagem

Quando os processadores trabalham em conjunto na execução de uma aplicação, é essencial que exista dois tipos de interação nessa cooperação: a comunicação e o sincronismo [JUN, 1999]. Contudo, se a memória for compartilhada, consegue-se essa cooperação através de espaços de memória compartilhados entre os diversos processos paralelos. Mas, se a memória for distribuída, ou seja, cada processador (nó) possui seu dispositivo de memória local, consegue-se essa cooperação através de **troca de mensagens**, sendo uma maneira pela qual os processos possam se comunicar e se sincronizar.

Esse paradigma (troca de mensagens) ilustrado na figura 3.1, define um conjunto de primitivas que permite a comunicação entre processos, nos quais pode-se destacar alguns fatores que justificam a sua grande aceitação, tais como:

- pode ser executado em uma grande variedade de plataformas;

- pode adequar-se naturalmente a arquiteturas ampliáveis, ou seja, a capacidade de aumentar o poder computacional proporcionalmente ao aumento de componentes do sistema;
- a tendência é não se tornar obsoleto, por redes mais rápidas ou arquiteturas que combinem memória compartilhada e memória distribuída. Em algum nível, sempre será necessário utilizar troca de mensagens de alguma maneira.

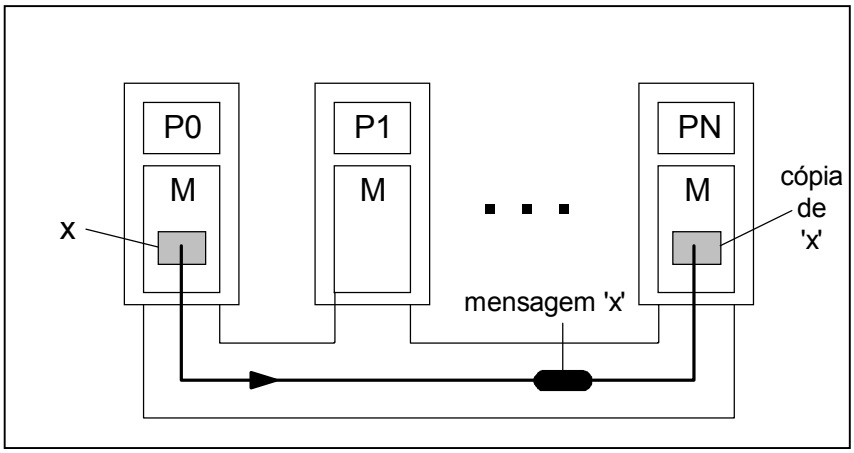

Figura 3.1 - Transferência de uma mensagem.

A figura acima, descreve um modelo genérico de transferência de uma mensagem, no qual, se um determinado processo executando no processador PN necessita de um dado do processo executando no processador P0, então esse dado será transferido explicitamente da memória local de P0 e será copiado na memória local de PN.

Tradicionalmente em uma operação de troca de mensagens, algumas informações devem ser consideradas: o processo transmissor da mensagem; os dados que por ventura formam a mensagem; o tipo dos dados; o tamanho da mensagem; o

processo receptor da mensagem; aonde os dados serão armazenados no processo receptor; um identificador (*para que a mensagem possa ser selecionada pelo processo receptor*) e qual a quantidade de dados suportada pelo receptor, de maneira que não se envie dados que o receptor não esteja preparado para receber (*controle de fluxo*).

Geralmente essas informações são supervisionadas pelo sistema que está gerenciando a transferência da mensagem correspondente, e algumas destas informações devem ser anexadas à mensagem.

3.3 Comunicação e Sincronismo em Memória Distribuída

A comunicação e o sincronismo em arquiteturas de memória distribuída, devem ser implementados através de troca de mensagens entre processos [ALM, 1994] [TAN, 1997] [QUI, 1987]. Uma operação de comunicação via mensagens, de maneira genérica, é realizada pela utilização das primitivas send/receive (*envia/recebe*), cujas sintaxes podem variar de acordo com o ambiente de programação:

Send mensagem to processo_destino

Receive mensagem from processo_fonte

Uma transferência de mensagem pode ser realizada através de duas maneiras: operação bloqueante síncrona ou operação não bloqueante assíncrona. Uma operação bloqueante síncrona (figura 3.2 (a)), implica que o processo que transmite a mensagem (*transmissor*) é bloqueado até que receba uma confirmação de recebimento da mensagem pelo processo receptor. Caso contrário, se tem uma operação não

bloqueante assíncrona (figura 3.2 (b)), isto é, o processo transmissor envia a mensagem (que deve ser armazenada em um *buffer*) e continua a sua execução.

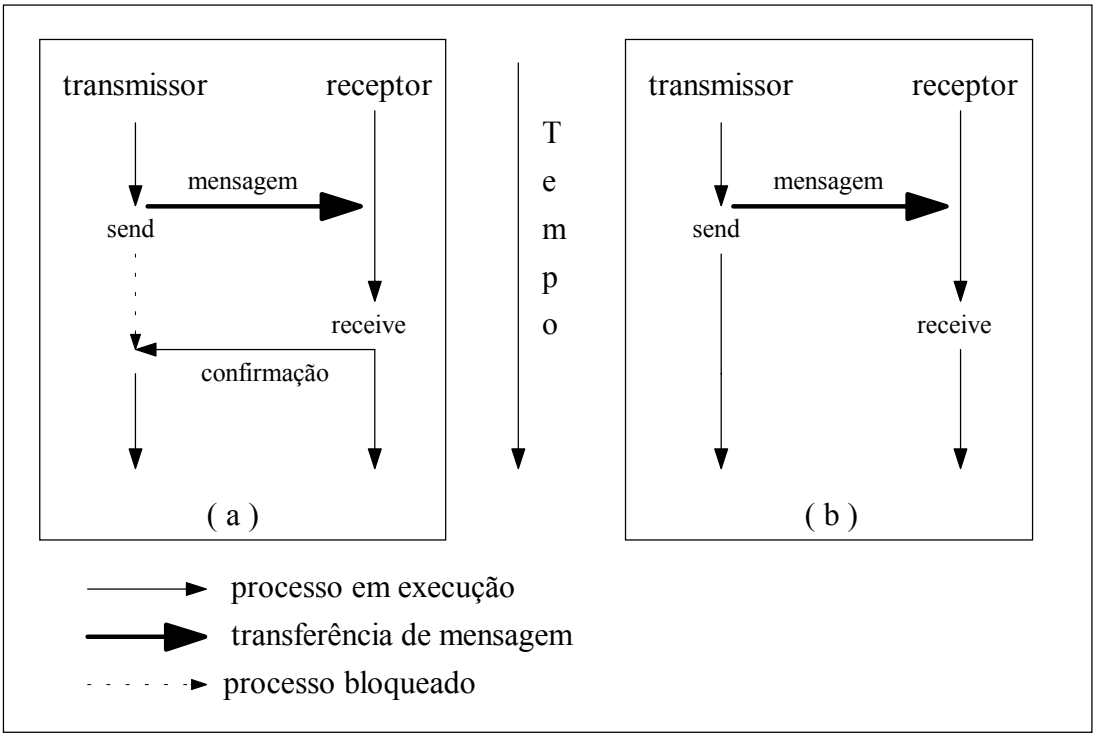

Figura 3.2 - Primitivas Send/Receive. (a) Bloqueante (b) Não-Bloqueante

3.4 Ambiente de Passagem de Mensagens

Um programa que utiliza troca de mensagens pode ser definido como um conjunto de programas seqüenciais, distribuídos em vários processadores (nós), que se comunicam através de um conjunto geralmente limitado e bem definido de instruções. Estas instruções formam o **ambiente de troca de mensagens** e são disponíveis através de uma **biblioteca de troca de mensagens**.

Um ambiente de passagem de mensagens (programação via troca de mensagens) consiste basicamente de uma linguagem seqüencial como C e/ou Fortran,

que será utilizada na implementação dos programas seqüenciais nos processadores, e de uma biblioteca de comunicação, atuando como uma extensão das linguagens seqüenciais, permitindo assim, a elaboração de aplicações paralelas [SOU, 1996].

Os programas nos diversos processadores não precisam ser necessariamente distintos. Pode-se utilizar o paradigma **SPMD** (*Single Program - Multiple Data*), isto é, o mesmo código fonte é distribuído pelos processadores, e cada processador executa de maneira independente, o que implica na execução de diferentes partes do programa em cada um dos processadores. Quando se distribui códigos fonte distintos para os processadores, utiliza-se o paradigma **MPMD** (*Multiple Program - Multiple Data*). Os diferentes programas em um ambiente MPMD podem ser unidos em um só, excetuando-se um eventual gasto maior de memória em cada processador, mas, praticamente não há perda de performance na utilização do paradigma SPMD. A vantagem deste paradigma, está na facilidade e simplicidade de gerenciamento de um sistema deste tipo.

3.5 Exemplos de Ambiente de Passagem de Mensagens

Inicialmente os ambientes de passagem de mensagens foram desenvolvidos para máquinas com processamento maciçamente paralelo (*Massively Parallel Processing* - MPP) onde, devido à ausência de um padrão, cada fabricante desenvolveu seu próprio ambiente, sem se preocupar com portabilidade, ou seja, enfatizavam aspectos diferentes (particularidades) para o seu sistema [SOU, 1996]. Pode-se citar como exemplos desses sistemas os seguintes projetos de ambiente de passagem de

mensagens: nCUBE PSE, IBM EUI, Meiko CS System e Thinking Machines CMMD [MCB, 1994].

Com o passar dos anos, muita experiência foi adquirida, e na atualidade os ambientes de passagem de mensagens foram remodelados e/ou desenvolvidos para plataformas portáteis, onde a idéia é definir um conjunto de passagem de mensagens independentes da máquina que está sendo utilizada e implementá-la em várias plataformas de hardware e sistemas operacionais. Pode-se citar como exemplos de ambientes de passagem de mensagens com plataformas portáteis para equipamentos heterogêneos, os seguintes grupos de pesquisa: P4, PARMACS, Express, PVM, MPI, entre outros [SUN, 1994] [MCB, 1994] [WALL, 1994].

As interfaces de passagem de mensagens com plataformas portáteis para sistemas heterogêneos P4, PARMACS, Express, Linda e PVM são descritas brevemente nas próximas seções, com exceção do MPI, que será melhor apresentado no Capítulo 5, visto a sua importância neste trabalho.

3.5.1 P4

O ambiente P4 foi desenvolvido pelo *Argonne National Laboratory*, sendo a primeira tentativa de desenvolvimento de uma plataforma portátil [SOU, 1996]. O seu projeto iniciou em 1984 (chamado na época de Monmacs), e a partir da sua terceira geração (em 1989), ele foi rescrito com o objetivo de produzir uma **biblioteca de macros** mais robusta para as necessidades recentes de passagem de mensagens entre máquinas heterogêneas com eficiência e simplicidade.

Uma das principais características do P4, é a possibilidade de ser utilizado por múltiplos modelos de computação paralela , por exemplo, os paradigmas SPMD e o MPMD (seção 3.4). Porém, este sistema é mais empregado em arquiteturas MIMD (capítulo 2, seção 2.2.1) com memória distribuída (*funções bloqueantes para enviar e receber mensagens*), com memória compartilhada (*modelo de monitores para coordenar o acesso aos dados compartilhados*) e em clusters (*coleção de máquinas*), onde os processos podem ser coordenados por *monitores* ou por *passagem de mensagens*, dependendo da organização da memória. Em clusters, há também funções para identificar o processo principal de cada *cluster*, determinar o número de *clusters* e obter os identificadores dos processos pertencentes a um *cluster*.

A flexibilidade no modo como são iniciados os processos é outra característica importante do P4, que dependendo a plataforma, são iniciados vários processos ou apenas um, o qual irá iniciar os outros. Esta flexibilidade se propaga através da utilização de um arquivo que especifica as características dos processos que estão em execução, tais como: as máquinas em que eles estão sendo executados, os arquivos executáveis e os grupos de clusters de processos que compartilham memória [SOU, 1996].

Devido à necessidade de desempenho, a tradução de mensagens entre máquinas com diferentes formatos de dados (*plataformas heterogêneas*), só é realizada quando necessária através do formato **XDR** (*padrão que define a representação para dados comuns, como strings, arrays e records, usado no SUNNFS*). Pode-se citar como

exemplos de equipamentos que executam o P4, as seguintes plataformas: Intel Touchstone, CM-5 e redes heterogêneas de estações de trabalho.

3.5.2 PARMACS

O ambiente PARMACS (PARallel MAcroS) começou a ser desenvolvido em 1987 no *Argonne National Laboratory*, herdando muitas características do sistema P4. Na sua primeira versão foi implementada um conjunto básico de instruções de passagem de mensagens utilizando a linguagem C, sendo que a versão 6.0 está sendo distribuída comercialmente. A principal diferença da versão 6.0 em relação às versões anteriores, foi a substituição da biblioteca de macros por uma **biblioteca de funções**, onde a chamada a subrotinas tornam o Parmacs mais próximo das linguagens de programação atuais [SOU, 1996].

O seu modelo de programação é baseado em memória local, isto é, os processos paralelos podem ter acesso somente ao seu espaço de endereçamento, não existindo neste caso variáveis globais compartilhadas [CAL, 1994].

O Parmacs inicia os processos paralelos através de um processo principal (*hosts*), onde sua tarefa é criar os processos (*nodes*) simultaneamente e distribuí-los através do hardware disponível. Após a criação dos processos (*nodes*), é atribuído um identificador a cada um (*node*), podendo ser finalizados pelo processo (*host*), o qual pode criar outros processos (*nodes*).

A comunicação entre processos no Parmacs pode ser síncrona ou assíncrona (seção 3.3) através de troca de mensagens. Assim como o P4, a tradução de mensagens entre processadores com diferentes representações de dados é realizada através do padrão **XDR**. Esta biblioteca de comunicação portátil, tem sido implementada na maioria das máquinas com arquitetura MIMD, em máquinas com MPP e em redes de estações de trabalho.

3.5.3 EXPRESS

O sistema Express é um produto comercializado pela empresa *Parasoft*, o qual foi desenvolvido para ser um ambiente de passagem de mensagem atuando sobre várias máquinas MIMD com memória distribuída [ALM, 1994], buscando obter o máximo de desempenho de cada plataforma utilizada.

Uma das principais características do Express (versão 3.0), é abordar problemas como o balanceamento dinâmico de carga e desempenho de I/O paralelos, fatores estes quase totalmente ignorados pela maioria dos ambientes de passagem de mensagens.

Com o seu avanço tecnológico, o produto Express passou a ser considerado como um conjunto de ferramentas e utilitários para desenvolvimento de software paralelo [FLO, 1994]. A sua interface para o usuário está sendo aprimorada, com o objetivo de esconder detalhes internos de implementações, o que levou ao desenvolvimento de mapeamentos e bibliotecas de comunicação para diferentes tipos de

topologias empregadas, visando obter o melhor desempenho de acordo com a topologia inserida.

3.5.4 Linda

O sistema Linda começou a ser desenvolvido em 1980 na *Yale University*, o qual foi desenvolvido para ser um ambiente independente da máquina para implementação de códigos paralelos. Em vez de empregar os modelos de memória compartilhada tradicional ou de passagem de mensagens, o seu ambiente está baseado em um conjunto de operações que implementam o conceito **Tuple Space (TS)**, isto é, a criação e a conseqüente coordenação dos múltiplos processos realiza-se de maneira diferente dos demais ambientes de passagem de mensagens, como Parmacs, Express e outros.

Uma das principais características do TS (*espaço de tuplas*) é o de ser um modelo de memória associativa compartilhada, descrito por uma linguagem de alto nível como C e/ou Fortran, onde os processos possam inserir e extrair tuplas de forma associativa. O TS fornece um nível de abstração onde é possível a construção lógica de uma memória compartilhada sobre memórias distribuídas, ou seja, memória compartilhada virtual.

As tarefas de gerenciamento de processos, sincronização e comunicação são realizadas através de operações que manipulam as tuplas [MCB, 1994]. Os objetos de dados são conhecidos como *tuples*, os quais podem ser de dois tipos: *Process Tuples* (PT) ou *Data Tuples* (DT). O *Process Tuples* (PT) realiza a troca de dados através da

criação, leitura e alteração de *Data Tuples* (DT). O TS é quem permite a comunicação entre os processos envolvidos, por exemplo, para um processo *A* transmitir dados para o processo *B*, *A* envia uma *tuple* de dados (DT) para o TS e *B* procura a *tuple* em TS.

A memória subjacente pode ser distribuída, mas isto fica oculto pelo próprio sistema Linda. Neste caso, o sistema atua como um conjunto de rotinas que oculta o hardware e permite realizar as operações de inserção e extração neste espaço de tuplas. Atualmente existem duas versões comerciais do ambiente Linda, uma para máquinas paralelas com memória compartilhada e distribuída, e outra para estações de trabalho em rede.

3.5.5 PVM (Parallel Virtual Machine)

O projeto PVM teve início no verão de 1989 no Oak Ridge National Laboratory - ORNL, com a construção do primeiro protótipo (versão 1.0) implementada por *Vaidy Sunderam* e *Al Geist* (ORNL), direcionada apenas para testes em laboratório. A partir da versão 2.0 (fevereiro de 1991), houve a participação de outras instituições, como a *Universidade de Tennessee* e a *Carnegie Mellon University*, difundida para uso geral, principalmente em aplicações científicas, que foi sucessivamente alterada (PVM 2.1 - 2.4). A versão 2.0 deu início a distribuição gratuita do PVM, o que favoreceu bastante a sua divulgação. Em Fevereiro de 1993 estava concluída a versão 3.0, sendo rescrita completamente, distribuída livremente e utilizada em vários projetos. A versão 3.3 é o resultado de várias mudanças feitas na versão 3.0 com o objetivo de retirar erros de programação e ajustar pequenos detalhes, como oferecer melhor interface para o

usuário e aumentar o desempenho de certas comunicações, como em multiprocessadores.

O PVM (*Parallel Virtual Machine*) é uma biblioteca de rotinas, utilizada para efetuar a comunicação entre processos paralelos, utilizando memória distribuída. É um conjunto integrado de bibliotecas e ferramentas de software, cuja finalidade é emular um sistema computacional concorrente heterogêneo, flexível e de propósito geral [BEG, 1994]. O projeto PVM nasceu com o objetivo de permitir que um grupo de computadores fossem conectados, permitindo diferentes arquiteturas de maneira a formar uma **máquina paralela virtual** [GEI, 1994].

O termo *máquina virtual* é utilizado para designar um computador lógico com memória distribuída e o termo *host* para designar um dos computadores que formam a máquina virtual. Uma aplicação no PVM pode ser paralelizada pelo método (*paradigma*) SPMD ou MPMD, como também um método híbrido (*uma mistura dos dois*). A figura abaixo mostra um exemplo do modelo computacional do PVM e uma visão arquitetural destacando a heterogeneidade do sistema.

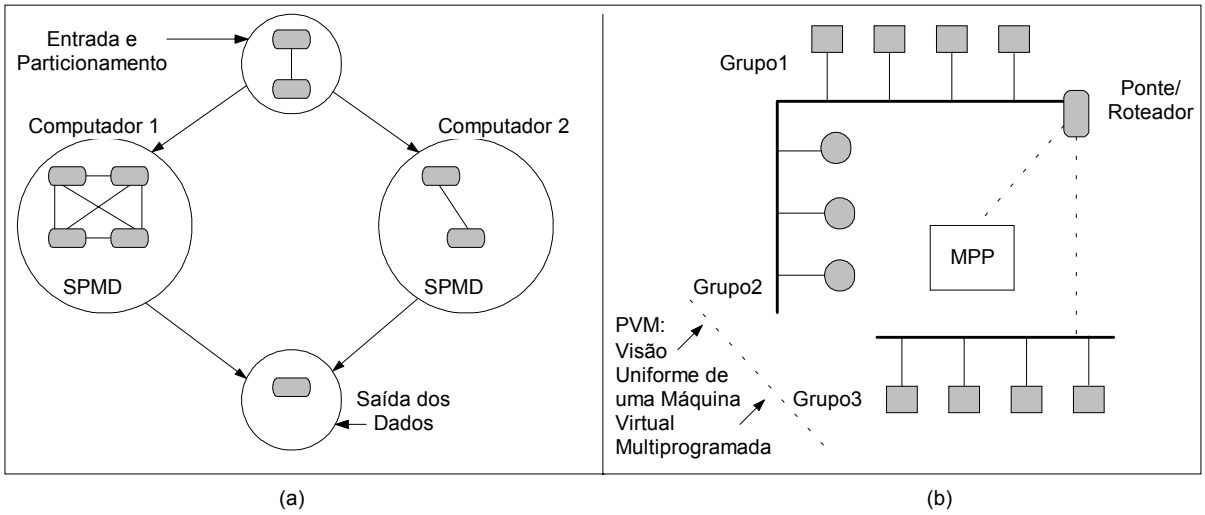

Figura 3.3 - Sistema PVM. (a) Modelo Computacional e (b) Visão Arquitetural

O ambiente PVM é composto de duas partes [GEI, 1994]. A primeira parte é um *daemon* chamado *pvmd*, que reside em todos os hosts que compõem o cluster, criando o que se refere como uma *máquina paralela virtual*. Quando um usuário deseja rodar uma aplicação PVM, ele executa o daemon *pvmd* em um dos hosts do cluster, geralmente a máquina mestre, o qual responsabiliza-se por chamar outros processos daemons *pvmd* escravos em cada host da máquina paralela virtual. Uma aplicação PVM pode então ser iniciada em um prompt Unix em qualquer console do cluster. A segunda parte do sistema é uma biblioteca de rotinas com a interface PVM (Libpvm). Esta biblioteca contém rotinas chamáveis pelo usuário para passagem de mensagens, criação de processos, sincronização de tarefas e modificação da máquina virtual. Este conjunto de primitivas atuam como um elo de ligação entre uma tarefa e a máquina virtual (o Pvmd e as outras tarefas). As aplicações devem ser linkadas com esta biblioteca para puderem usufruir do ambiente paralelo criado pelo PVM.

As tabelas de hosts e de tarefas contêm a situação atual da máquina virtual, indicando como está sendo feita a execução e a comunicação em cada elemento de processamento.

Atualmente, esse ambiente de passagem de mensagens está disponível para uma grande variedade de plataformas, permitindo que sejam escritas aplicações nas linguagens Fortran, C e C++, devido ao fato de a maioria das aplicações passíveis de paralelização estarem escritas nessas linguagens. A tabela 3.1 mostra alguns dos equipamentos que podem ser utilizados pela versão 3.3 do PVM.

Alliant FX/8
DEC Alpha
Sequent Balance
Bbn Butterfly TC2000
80386/486/Pentium com UNIX (Linux ou BSD)
Thinking Machines CM2 CM5
Convex C-series
C-90. Ymp, Cray-2, Cray S-MP
HP-9000 modelo 300, HP-9000 PA-RISC
Intel iPSC/860, Intel iPSC/2 386 *host*
Inter Paragon
DECstation 3100, 5100
IBM/RS6000, IBM RT
Silicon Graphics
Sun 3, Sun 4, SPARCstation, Sparc multiprocessor
DEV Micro VAX
Windows 32 bits

Tabela 3.1 - Alguns dos principais equipamentos que executam o PVM.

Entre alguns usuários PVM, pode-se destacar os seguintes: Ford, Boeing, Texaco, General Eletric, Siemens, Mobil Oil, Cray Research, Shell Oil, IBM, entre outros [BEG, 1994].

Devido a extensão e a complexidade desta popular biblioteca de passagem de mensagens, torna-se difícil descrever todos os tópicos relacionados à Máquina Paralela Virtual. Sendo assim, foram abordados nesta seção os aspectos julgados mais importantes.

3.6 Considerações Finais

A rápida evolução da computação e as transformações que isso tem proporcionado contribuíram para o surgimento prático da computação paralela. Os sistemas distribuídos, por sua vez, foram cada vez mais aperfeiçoados, destacando-se com mudanças significativas, como maior eficácia dos meios de comunicação e protocolos, e computadores com maior potência computacional.

Uma das principais vantagens da computação paralela é o *alto desempenho*, porém existem alguns fatores que podem influenciar de forma negativa, dificultado consideravelmente o trabalho, como alto custo de aquisição e manutenção, e a dependência ao fabricante (*soluções proprietárias*). Uma solução para este problema é a utilização de sistemas distribuídos (*inicialmente compartilhavam recursos de alto custo*) como plataformas de execução paralela, fornecendo menor custo de implantação e maior flexibilidade no processo computacional paralelo.

Inicialmente, os projetos de ambientes de passagem de mensagens não foram desenvolvidos especificamente com o intuito de utilizar os sistemas distribuídos para o desenvolvimento de aplicações paralelas. Devido ao avanço tecnológico sofrido, as aplicações puderam ganhar a portabilidade perdida e serem executadas em todos os

equipamentos para os quais o ambiente foi desenvolvido. Recentemente os ambientes de passagem de mensagens portáteis (*como o P4, PARMACS, Express, PVM e MPI*), também têm sido desenvolvidos para sistemas heterogêneos, onde dois ou mais tipos de computadores diferentes cooperam para resolver um problema.

A escolha de uma plataforma de portabilidade deve ser realizada levando em consideração alguns fatores como: quais as necessidades da aplicação, qual plataforma paralela a ser utilizada, entre outros aspectos. Dessa maneira, deve-se ter cautela para afirmar que uma plataforma de portabilidade seja, no geral, melhor que a outra, pois o balanceamento de carga e a perda de desempenho devido ao congestionamento no meio de comunicação são exemplos de problemas comuns às duas áreas e que são, ainda, amplamente pesquisados [ZAL, 1991].

A computação paralela que empregava quase que na totalidade arquiteturas SIMD, começou também a utilizar devido à sua versatilidade e alto desempenho máquinas MIMD com memória distribuída. Essas máquinas passaram a contar com processadores de propósito geral, tornando possível a sua visualização como um conjunto de computadores autônomos (*com Unidade de Controle, Unidade de Processamento e Memória*), interligados por uma rede de comunicação. As razões para o surgimento e a sua rápida aceitação perante a comunidade computacional, bem como outros aspectos envolvendo cluster de computadores serão ilustrados no próximo capítulo.

4 Cluster de Computadores

4.1 Considerações Iniciais

A computação paralela de alto desempenho é atualmente uma necessidade fundamental em muitas aplicações que envolvem processamento de algoritmos complexos e/ou grande volume de dados. No entanto, sistemas especialistas de processamento paralelo, com vários processadores interligados, são ainda caros e complexos, tanto em sua construção (hardware) como na programação (software).

O princípio básico de um cluster é conectar estações comuns de trabalho de forma que elas se comportem como um único computador que possa atingir desempenho compatível com os dos grandes sistemas paralelos especialistas com custos e complexidade inferiores. O paradigma mais importante para um ambiente de cluster é a configuração paralela que, de certa forma engloba um aumento na velocidade do processador e o uso de algoritmos mais otimizados [DAN, 2002].

Clusters são redes de estações coordenadas por um sistema operacional paralelo, realizando operações distribuídas de forma paralela, também conhecidos como PoPCs – Pile of PCs (uma pilha de PCs).

Pode-se dividir os clusters em duas categorias [ALV, 2002]: Alta Disponibilidade (HA - High Availability) e Alta Performance de Computação (HPC - High Performance Computing). O cluster HA tem a finalidade de manter um

determinado serviço de forma segura o maior tempo possível. O cluster de HPC é uma configuração designada a prover grande poder computacional, maior do que somente um único computador poderia oferecer em capacidade de processamento.

4.2 Vantagens em se utilizar Cluster de Computadores

Os clusters oferecem muitos benefícios sobre os outros tipos de ambientes computacionais de alto desempenho. Algumas vantagens dessa filosofia de trabalho, são:

- Alto desempenho. Possibilidade de se resolver problemas complexos através de processamento paralelo, o que diminui o tempo de resolução do problema.
- Redução de custo efetivo. Pois se obtém uma máquina de poder de processamento utilizando computadores de baixo custo, normalmente PCs, ligados através de uma rede local. Como os componentes são de fácil disponibilidade e interdependência de fornecedores de equipamentos, gera uma facilidade de menor custo.
- Mantém continuidade com a tecnologia. Com o uso de peças tradicionais de mercado de informática, basta atualizar os componentes e deixará mais rápido o "supercomputador", sem depender de soluções proprietárias. Os computadores que compõem os clusters podem ser heterogêneos [MEM, 2000], garantindo independência em relação aos fabricantes.
- Flexibilidade de configuração. Sistemas podem ser construídos usando componentes de diversas origens, graças ao uso de interfaces padrão, tais como IDE, PCI, SCSI.

- Escalabilidade. Quando existir a necessidade de aumento de processamento, pode-se anexar mais computadores escravos e velozes ao sistema, sem desligá-lo. Depois de tudo estabilizado, retira-se ou não as máquinas mais lentas.
- Alta disponibilidade. Cada computador escravo é uma unidade independente e, se algum deles falhar não afetam os demais ou a disponibilidade inteira do cluster. Tem-se uma pequena redução de performance até a máquina retornar, no momento em que a referida estiver fora do cluster, construindo assim um sistema de tolerância a falhas.
- Balanceamento automático de carga. A distribuição dos processos necessários para a resolução de um determinado problema, pode ser realizada de maneira automática, ou seja sem a intervenção do usuário final. Assim, o sistema tenta equilibrar o uso dos diversos computadores pertencentes ao cluster, para não sobrecarregar nenhuma das partes e ao mesmo tempo dar o máximo de desempenho para os usuários, nas suas respectivas tarefas.

4.3 Cuidados ao Projetar um Cluster

Apesar das vantagens apresentadas anteriormente, fatores como o tipo de aplicação alvo e as tecnologias existentes devem ser consideradas na hora de planejar um cluster. Alguns aspectos que podem dificultar o trabalho e devem ser avaliados são:

- Os equipamentos de rede não foram criados pensando em processamento paralelo.

- Existe software que não trata o cluster como uma máquina única.
- Algumas aplicações não podem ser paralelizadas, exigindo que as tarefas sejam executadas em série e de forma independente.

4.4 Áreas de Aplicação

A área de aplicação dos clusters é diversificada. Em qualquer local onde houver um problema computacional em que o processamento paralelo seja considerado uma vantagem, pode ser indicada a utilização de um cluster. É uma boa alternativa para a execução de aplicações paralelas e distribuídas [BAR, 1999].

As aplicações para clusters podem ser divididas em várias tarefas que podem ser executadas então concorrentemente por vários processadores. Alguns exemplos de áreas onde a utilização de clusters pode ser indicada são: aplicações matemáticas complexas, resolução de problemas de engenharia, processamento de imagens, balanceamento de carga de servidores Internet, quebra de código do DNA humano, aplicações meteorológicas, análises sísmicas de explorações de petróleo, simulação de aerodinâmica de motores e desenho de aeronaves , pesquisas biomédicas para modelagem molecular e como um ambiente para pesquisas dentro da área de computação paralela [ZOM, 1996], dentre várias outras aplicações.

4.5 Tecnologias de Redes para Clusters

Existem várias tecnologias de rede que podem ser adotadas na construção de um cluster. Entre as tecnologias de redes mais comuns, destacam-se as seguintes:

- ***Ethernet*** - O padrão Ethernet foi desenvolvido pela Xerox, DEC e Intel em meados de 1972, com uma largura de banda de 1 Mbit/s, sendo posteriormente padronizado a 10 Mbit/s pelo IEEE (Instituto de Engenheiros Eletricistas e Eletrônicos dos EUA), sob a normalização IEEE802.3. Utiliza como método de acesso ao meio físico o protocolo CSMA/CD (Carrier Sense Multiple Access with Collision Detection). É uma rede de transmissão de barramento, que permite descentralização a velocidades de 10 ou 100 Mbit/s [TAN, 1997].
- ***Fast Ethernet*** - A tecnologia Fast Ethernet (100 Base T), é uma versão de 100 Mbits/s da popular Ethernet (10 Base T). Foi oficialmente denominada de padrão IEEE 802.3u e é um padrão suplementar ao já existente, o IEEE 802.3, fornecendo uma largura de banda dez vezes mais rápido que o padrão Ethernet.
- ***Gigabit Ethernet*** - É uma proposta recente na forma da norma IEEE 802.3z, que surgiu como uma possível alternativa ao emprego ATM (Asyncronous Transfer Mode) em redes que demandam grande largura de banda para a transmissão de dados. Esta tecnologia provê capacidade de 1 Gbps, tanto em transmissão duplex quanto semi-duplex, mediante ao emprego do protocolo CSMA/CD [USP, 2002].
- ***ATM (Asyncronous Transfer Mode)*** - É um padrão internacional de rede produzido pelo ATM fórum. ATM é projetado dentro do conceito de células ou pequenos pacotes de tamanho fixo, baseada em *switche* que utiliza a idéia de circuitos virtuais para obter um serviço de conexão orientada [ALV, 2002]. Foi projetada para prover uma interface de software com baixo

overhead, para maior eficiência no gerenciamento de pequenos pacotes e em aplicações de tempo real [DIE, 1997].

- ***HIPPI (High Performance Parallel Interface)*** - É uma rede Gigabit produzida pelo High Performance Networking Fórum, que foi inicialmente planejada para interconectar computadores paralelos de alto desempenho, com redes de dispositivos de armazenamento fixos que seguem o padrão ANSI X3T11. Esta tecnologia funciona com velocidades de 800 Mbps ou 1.6 Gbps e foi desenvolvida para interligar mainframes ou supercomputadores em um cluster. Este padrão passou por várias versões e atualmente suporta o nome de Gigabit System Network (GSN) [ALV, 2002].
- ***FC (Fibre Channel) -*** Sucessor do HIPPI, possui como estrutura básica um crossbar [TAN, 1997], formando canais de dados com conexões que podem ser estabelecidas para um único pacote ou permanecer por um longo período.
- ***ARCNET (Attached Resource Computer Network) –*** É uma tecnologia para LAN (Local Area Network) desenvolvida pela Datapoint Corporation, utiliza o protocolo token-bus para gerir o acesso à rede dos diversos dispositivos ligados. Neste tipo de rede circulam constantemente pacotes vazios (*frames*) no barramento contento um "token" e a mensagem, cada pacote chega a todos os dispositivos da rede mas cada dispositivo só lê o pacote que contém o seu respectivo endereço. Este processo é eficaz com um grande volume de tráfego uma vez que todos os dispositivos têm a mesma oportunidade de usar a rede [ARC, 2002] .
- ***Myrinet*** - É uma rede local (LAN) para a construção de programas paralelos [SCH, 1999], desenvolvida pela empresa Myricon, visando formar uma

tecnologia de interconexão baseada em chaveamento e comunicação por pacotes [OVE, 2002].

Dentre as tecnologias citadas acima, a Myrinet foi desenvolvida para utilização no cluster NOW de Berkeley e em muitos outros clusters acadêmicos. Levando este fato em consideração, essa arquitetura é descrita com mais detalhes a seguir.

Essa tecnologia é voltada principalmente para interconexão de *clusters* em uma LAN, com baixo custo e alto desempenho. As características que distinguem a Myrinet das demais redes são [OVE, 2002]:

- Portas e interfaces full-duplex.
- Controle de fluxo, de erro, baixa latência e monitoramento contínuo dos *links.*
- Switches para aplicações de alta disponibilidade.
- Suporte a qualquer topologia.
- Estações possuem programa de controle para interagir diretamente com os processos no nível de usuário e da rede.
- Possui interfaces que podem mapear a rede e traduzir os endereços da rede para as rotas selecionadas.
- Serviço de tolerância a falhas devido ao mapeamento automático da configuração da rede, o que também facilita a tarefa de configuração.
- Suporta os Sistemas Operacionais Linux e Windows NT.

Um *link myrinet* (figura 4.1) é formado por um par de canais full-duplex que são responsáveis pela transmissão de pacotes de tamanho variável e pelo controle do fluxo de informações [MYR, 2002]. Assim como em redes Ethernet, podem ser transportados concorrentemente pacotes de vários tipos de protocolos, suportando diferentes interfaces de software, como o TCP/IP por exemplo. O roteamento dos pacotes ao longo da rede é feito através do seu cabeçalho [MYR, 2002].

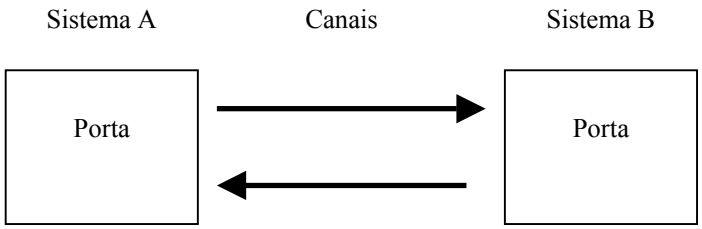

Figura 4.1 - Representação de um Link Myrinet.

Os itens necessários para a construção de uma rede Myrinet são as interfaces dos equipamentos de hardware, switches para redes de maior porte, cabos, adaptadores e um software de suporte a empresa Myricon ou algum outro software que suporte redes Myrinet [MYR, 2002]. O software desenvolvido pela Myricon é baseado na troca de mensagens e está disponível em várias plataformas, como por exemplo em várias distribuições do Linux.

O desempenho das soluções baseadas em Myrinet depende muito do software que é utilizado sobre a camada de rede [MYR, 2002]. Podem ser utilizados quaisquer tipos de software de interface, mas o ideal é a utilização de um software de

interface otimizado e projetado especialmente para Myrinet, o que permite alcançar resultados satisfatórios.

Esta tecnologia também é recomendada para a construção de backbones e aplicações que utilizam muita largura de banda como transferência de imagens de elevada resolução ou vídeo [ALV, 2002].

4.6 Clusters Beowulf

O modelo da arquitetura baseado em clusters é representado principalmente pelos Clusters Beowulf. Como definição, Beowulf é uma arquitetura de multicomputadores utilizados para computação paralela, que consiste em um conjunto de máquinas formado por um nó servidor e nós clientes, que podem ser conectados via rede local (Ethernet, Fast Ethernet, Gigabit Ethernet) que são relativamente de baixo custo, quando comparada a outras tecnologias como fibra ótica (Fiber Distributed Data Interface – FDDI) e ATM (Asynchronous Transfer Mode). A coordenação das tarefas é feita por transmissão de mensagens de um computador para o outro. Os métodos mais populares de bibliotecas de passagem de mensagens são:

- Interface de Passagem de Mensagens (MPI – Message Passing Interface);
- Máquina Paralela Virtual (PVM – Parallel Virtual Machine).

Uma abordagem mais detalhada da Interface de Passagem de Mensagens será apresentada no capítulo 5.

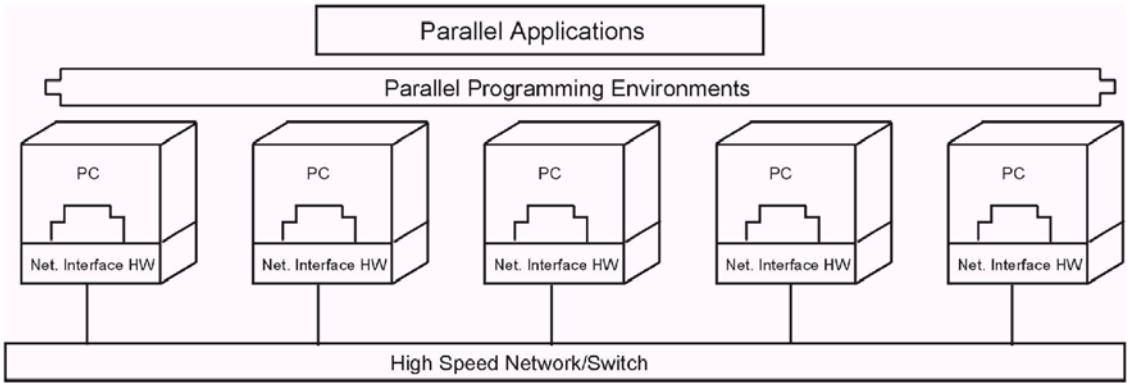

Figura 4.2 - Um cluster Beowulf.

O servidor tem a função de controlar o cluster, distribuir os arquivos e as tarefas para os nós clientes e servir de gateway para conexão externa. Existe a possibilidade de haver mais de um servidor em um cluster Beowulf, podendo ser dedicado à operações específicas como monitoração ou gateway, ou para se prestar somente como console.

Cada nó secundário é uma unidade independente que pode ser substituída, ou retirada do cluster, sem afetar a disponibilidade do cluster como um todo. Em muitos casos, os nós secundários de um cluster Beowulf devem ser o mais simples possível, ou seja, recomenda-se utilizar máquinas com o menor número de acessórios, dispensando unidades de disco flexível e rígido, monitor, teclado, mouse e outros periféricos. Todos os nós clientes são configurados a partir do servidor e só realizam o que lhes é determinado por ele, portanto simulando o comportamento de uma máquina única.

Em aplicações distribuídas, cada tarefa pode ser executada por apenas um nó secundário e o nó principal guarda a informação final, fazendo com que o cluster execute como um sistema de diferentes máquinas e não como uma única máquina virtual [KLA, 2002].

A configuração dos computadores que compõem o cluster variam em relação ao tipo e à quantidade de processadores utilizados, ao tamanho e velocidade da memória e da rede de interconexão. Essa configuração deve ser definida de acordo com o desempenho desejado para as aplicações alvo [PIT, 2002].

A denominação "Beowulf" é uma homenagem ao legendário herói do velho poema inglês, neste conto épico, Beowulf, um bravo cavaleiro inglês, salvou o reino dinamarquês de dois monstros do fundo do mar, que acabou virando nome genérico para todos os clusters que se assemelham à configuração original [BEO, 2002].

O potencial de expandir a força de processamento paralelo com o uso de PCs comuns, foi identificado como uma possibilidade importante pela NASA em suas aplicações de missão crítica, e atendia aos objetivos da empresa de “barato e rápido” [ALV, 2002].

No início de 1994, o projeto Beowulf foi iniciado sobre o patrocínio da NASA HPCC (High Performance Computing and Communications) com o ESS Project (Earth and Space Sciences) para investigar o potencial de clusterizar PCs a fim de se efetuar tarefas computacionais importantes, sem um grande custo. O valor total da

construção desta plataforma foi de 40.000,00 (quarenta mil dólares) ou dez pontos percentuais do preço de uma máquina comercial em 1994. Em outubro de 1996, foi anunciado que o sistema Beowulf havia excedido a ordem dos Gigaflops à um custo total de U$50.000,00 (cinquenta mil dólares). Esta foi uma inovação em performance e preço que teve implicações importantes em uma larga gama de aplicações industriais e científicas.

4.6.1 O Programa NASA HPCC

O programa NASA HPCC nasceu em janeiro de 1992, com o objetivo de alcançar e avançar o estado de processamento massivamente paralelo (MPP) e aplicá-lo ao maior número de problemas computacionais importantes da NASA, mais objetivamente nas aerociências computacionais (CAS) e no Projeto ESS (Earth and Space Sciences). O Projeto ESS representa um domínio computacional que inclui manipulação direta de grande número de dados por cientistas. Como os estudos nesta área partem de simulações de fenômenos físicos, evolução das galáxias, convenções do plasma na corona solar, entre outros, cientistas precisam adquirir, examinar, explorar, manipular, visualizar, e às vezes, transformar grandes coleções de dados complexos. Na maioria das vezes, isso envolve atividades de computação numérica intensiva e movimentação de grandes volumes de dados.

Estes dois sistemas “de referência”, o primeiro montado no CESDIS (Center of Excellence in Space Data and Information Sciences) da NASA, em 1994, e seu sucessor, de 1996, também da NASA, como módulo de demonstração. Ambos foram testados com a mesma base de software e com os mesmos esquemas de interconexão,

mas utilizavam hardware diferentes (computadores, acessórios e equipamentos para redes) [CES, 2002]. Os próprios pesquisadores Thomas Sterling e Donald J. Becker lembram que não há dois Beowulfs absolutamente iguais, em decorrência de cada um deles ter sido construído sob diferentes aspectos [ALV, 2002]. Outras implementações feitas fora da NASA usam variações na topologia da rede de interconexão e nos microprocessadores, mas sempre usando componentes disponíveis no mercado, mantendo a filosofia "faça você mesmo o seu supercomputador".

Protótipos Utilizados nos Experimentos

I - Protótipo Experimental (Wiglaf) - Beowulf Parallel Workstation (1994)

Este sistema é constituído de 16 nós, cada um possuindo:

- microprocessador Intel 80486DX4 (100 MHz);
- barramento local padrão VESA;
- 16MB de memória RAM;
- 16KB de memória cache primária (interna ao 486);
- 256Kb de memória cache secundária;
- um disco IDE de 540MB;
- duas interfaces de rede padrão Ethernet de 10Mbit/s.

Dois desses subsistemas processadores possuíam interfaces Ethernet extra para redes locais e acesso remoto. Dois outros nós possuem controladoras de vídeo de alta resolução, sendo um deles dotado também de mouse e teclado. A interconexão entre os subsistemas originalmente usava duas redes de tecnologia Ethernet, tendo uma como

meio físico o de par trançado e outra de cabo coaxial fino (10 baseT e 10 base 2 respectivamente). Outros arranjos foram experimentados mais tarde.

II - Sistema de Demonstração (Hrothgar) - Beowulf Demonstration System (1996)

Como uma evolução do sistema anterior, foi montado um novo protótipo, mantendo a mesma arquitetura geral, mas incorporando componentes novos. Este sistema é composto de 16 nós, cada um possuindo:

- microprocessador Intel Pentium (100 MHz);
- barramento local PCI;
- 32MB de memória RAM;
- um disco IDE de 1,2 GB;
- duas interfaces Fast Ethernet de 100Mbit/s.

A diferença mais importante entre os dois protótipos é o emprego, no segundo, da tecnologia Fast Ethernet para a rede de interconexão, que se chamou de “uma arquitetura de sistemas balanceada”, enquanto o protótipo experimental era considerado "não balanceado", podendo-se entender isso como um maior equilíbrio entre a capacidade de processamento e a capacidade de vazão da rede de interconexão.

Em novembro de 1999, outro cluster da mesma classe, o CPlant (Computational Plant) da Sandia National Laboratory, foi classificado em quadragésimo quarto lugar na lista dos quinhentos computadores mais rápidos do planeta, o Top500 [TOP, 2002].

4.7 Cluster de Estações de Trabalho

Um cluster de workstations (COW) é um muito parecido com o Beowulf, pois é constituído de estações de trabalho de alto desempenho, cada uma com monitor, teclado e mouse ligados por uma tecnologia de rede padrão Ethernet ou Fast Ethernet. Todas estas máquinas se comunicam entre si. A partir de um nó, pode-se acessar outro nó, podendo assim executar as tarefas remotamente. Em um COW pode-se ter acesso ao teclado (console) em qualquer computador (nó) da rede, o que não acontece ao Beowulf, que só se tem acesso ao teclado no computador principal, e a partir desse acessar os outros nós por acesso remoto (remote login).

Uma outra característica de um COW é que quando uma máquina não está participando para o processamento distribuído, todos os seus outros nós podem ser utilizados como postos de trabalho. No entanto, o seu funcionamento é idêntico ao do Beowulf, possuindo também um computador principal, escolhido no momento para ser o controlador de distribuição de tarefas para os nós escravos [ALV, 2002].

Um exemplo deste tipo de cluster (COW) é o projeto Mosix, um cluster OpenMOSIX do Dr. Moshe Bar, da Hebrew University de Jerusalém - Israel. O OpenMOSIX é um pacote de inserções (patches) no núcleo do sistema operacional Linux que habilita um balanceamento de carga de processos que estão em execução na sua rede local, que podem ser movimentados de forma transparente entre os computadores que compõem o cluster. Isso pode permitir que aplicações existentes funcionem num ambiente de cluster com poucas alterações.

Entretanto, o Mosix pode não ser usado por todas as aplicações, principalmente aquelas que fazem uso de muitos recursos de comunicação interprocessos ou requerem configuração de uma grande memória compartilhada, pois isto tornará muito lento [ALV, 2002].

A vantagem de usar Mosix em relação ao Beowulf é que o programa não precisa ser paralelizado ou customizado, isto é, a sutileza entre o processamento paralelo e o processamento distribuído com a visão de alta disponibilidade, migração de processos, transparência aos recursos, dentre outros [ALV, 2002].

4.8 PoPCs: Uma Pilha de PCs

PoPCs (Pile-of-PCs) é o termo usado atualmente para descrever uma montagem solta de PCs, ou, mais precisamente, um cluster de PCs, aplicado na resolução de um ou mais problemas. Segundo [BEC, 1997], é similar a um cluster de workstations (COW), mas tem o diferencial de enfatizar:

- uso de componentes comuns, disponíveis no mercado tradicional de informática;
- processadores dedicados na execução de tarefas, ao invés de usar tempo ocioso das estações;
- uma rede de sistema privada para os nós computacionais.

Além disso, para um cluster de PCs ser considerado um Beowulf (figura 4.2), precisa atender às seguintes características:

- nenhum componente feito sob encomenda;

- replicação fácil a partir de múltiplos vendedores;
- E/S escalável;
- uma base de software disponível livremente;
- uso de ferramentas de computação distribuída disponíveis livremente, com alterações mínimas;
- retorno à comunidade do projeto e melhorias.

4.9. Sistemas Operacionais para Clusters

Segundo [ALV, 2002]**,** o Linux encaixa-se com perfeição como sistema operacional para cluster de computadores, pois além de ser um sistema robusto, possui versões de uso livre e seu código fonte é aberto, permitindo ajustes necessários. Entre os sistemas operacionais mais adequados para clusters de computadores estão as versões do UNIX: Linux, FreeBSD ou NetBSD. O Windows NT também pode administrar um cluster, no entanto, a maioria dos softwares que são utilizados em clusters são executados com melhor desempenho em versões do UNIX. Existem sistemas operacionais puramente distribuídos como: Amoeba, Sprite, Plan 9 e Inferno.

Os sistemas operacionais de rede Windows NT e Windows 2000 Server tiraram vantagem da sua característica Multithreading para a construção do Beowulf-Like Windows NT.

4.10 Considerações Finais

O paradigma *cluster computing* pode ser compreendido como uma abordagem que visa um aumento no desempenho das aplicações, como uma maior velocidade na execução dos aplicativos e um maior número de dados a serem considerados na execução da própria aplicação [DAN, 2002].

Os primeiros sistemas construídos (*clusters*) baseavam-se em nós constituídos de processadores 80486DX4-100 e equipamentos ou acessórios de redes de baixo poder de transmissão, porém na atualidade utilizam-se de tecnologias avançadas, como processadores Pentium II e III e redes de alta velocidade, como ATM, Fast Ethernet e Gigabit Ethernet.

Clusters a moda Beowulf, propagam-se através de software disponível livremente, sofisticado, robusto e eficiente. Sendo assim, uma característica chave de um cluster Beowulf, é o uso do Sistema Operacional Linux, principalmente pelo fato de ser gratuito e de código fonte aberto, ou seja, qualquer usuário pode modificá-lo de acordo com seus critérios e necessidades para melhorá-lo, e se achar conveniente repassar para outros usuários essas "benfeitorias". Uma outra característica importante é o uso de bibliotecas para troca de mensagens PVM e MPI, no qual permite fazer alterações no Linux para dota-lo de novas características para facilitar a implementação de aplicações paralelas.

Beowulf é um projeto bem sucedido. A opção feita por seus criadores de usar hardware popular e software aberto tornou-o fácil de replicar e alterar, prova disso

é a grande quantidade de soluções construídas à moda Beowulf em diversas universidades, empresas americanas e européias. Mais do que um experimento, foi obtido um sistema de uso prático que continua sendo melhorado.

5 MPI - Interface de Passagem de Mensagens

5.1 Considerações Iniciais

O MPI (Message Passing Interface) é uma biblioteca de passagem de mensagens, o qual foi desenvolvido para ser um padrão entre os ambientes de memória distribuída, em passagem de mensagens e em computação paralela [GRO, 1994], ou seja, é uma tentativa de padronização para ambientes de programação via troca de mensagens.

Esta popular biblioteca de rotinas, surgiu da necessidade de se resolver alguns problemas relacionados às plataformas de portabilidade, como restrições em relação à real portabilidade de programas, devido ao grande número de plataformas e o mau aproveitamento de características de algumas arquiteturas paralelas [MCB, 1994].

No final da década de 80, com o desenvolvimento da computação paralela sobre sistemas distribuídos, vários fatores determinaram a necessidade de se desenvolver um padrão para esses tipos de ambientes de passagem de mensagens, dentre os quais pode-se citar:

Portabilidade e facilidade de uso: à medida que a utilização do MPI aumentar, será possível portar transparentemente aplicações entre um grande número de plataformas paralelas;

Fornecer uma especificação precisa: fabricantes de *hardware* podem implementar eficientemente em suas máquinas um conjunto bem definido de rotinas;

Crescimento da indústria de software paralelo: a existência de um padrão torna a criação de *software* paralelo por empresas uma opção comercialmente viável, o que também implica em maior difusão o uso de computadores paralelos.

5.2 Padronização do MPI

O processo de padronização teve início em abril de 1992 no Centro de Pesquisa em Computação Paralela em Williamsburg, na Virgínia (E.U.A), que promoveu um workshop intitulado "Padrões para Passagem de Mensagens em um Ambiente de Memória Distribuída", envolvendo na época aproximadamente cerca de 80 pessoas provenientes de 40 organizações, principalmente americanas e européias [MCB, 1994]. Deste encontro surgiu a necessidade de criar-se um grupo de trabalho para dar continuidade ao processo de padronização, no qual comunicavam-se através de correio eletrônico, sendo apresentada a primeira versão em novembro de 1992, isto é, o primeiro esboço da Interface de Passagem de Mensagens, o MPI1.

A partir do MPI1, que comprovou que o esforço de padronização era válido, decidiu-se organizar melhor todo o processo de padronização através da criação do **MPI Fórum**, o MPIF. Em novembro de 1993 foi apresentada a especificação do padrão MPI versão 1.0, sendo publicada a partir de maio de 1994, tornando-se domínio público [MPI, 2003]. Em junho de 1995, foi publicada a versão 1.1, apresentando correções de erros e melhor esclarecimento de determinadas propriedades do MPI. O MPI2, publicado em 1997, incluiu as seguintes inovações: gerenciamento de processos dinâmicos (*adicionar processos a uma computação MPI que esteja rodando e permitir que computações MPI paralelas conectem/desconectem*), acesso remoto a memória,

ferramentas para I/O paralelo, ligação para C++ e Fortran, interação com threads e interoperabilidade entre linguagens. Até o presente momento, algumas implementações do MPI incluem parte do padrão MPI-2 mas uma implementação completa deste padrão ainda não existe [PAC, 2003].

Dentre os principais objetivos que guiaram o processo de padronização, destacam-se:

- lançar uma versão inicial em um tempo predefinido, de maneira que não se perdesse o controle do padrão;
- prover portabilidade real;
- prover implementação eficiente em plataformas paralelas distintas;
- possuir uma aparência compatível com a atualidade, para possibilitar a sua fácil aceitação e difusão.

A maioria dos principais fabricantes de computadores paralelos participaram do desenvolvimento do MPI, além de universidades e laboratórios governamentais pertencentes à comunidade envolvida na computação paralela mundial [MCB, 1994].

Segundo o censo mantido pelo *Ohio SuperComputer Center*, existem pelo menos 15 implementações do MPI, comerciais ou de domínio público, os quais destacam-se as seguintes implementações:

- MPI-F: IBM Research.
- MPICH: ANL/MSU.
- UNIFY: Mississipi State University.

- CHIMP: Edinburgh Parallel Computing Center.
- LAM: Ohio SuperComputer Center.

5.3 Vantagens e Desvantagens do MPI

O MPI é um padrão de interface de passagem de mensagens para aplicações que utilizam computadores MIMD com memória distribuída [SOU, 1996]. Algumas vantagens da utilização do MPI quando comparado a outros ambientes são:

- alta performance;
- redução do tempo total de execução de programa (*wallclock*);
- portabilidade e funcionalidade transparente;
- flexibilidade (arquiteturas e redes de trabalho);
- software de domínio público;
- grande difusão e aceitação (comercialmente viável);
- facilidade de programação graças as bibliotecas para linguagens (Fortran e C);
- facilidade de instalação e configuração do ambiente;
- fácil definição e modificação da interface de passagem de mensagens (códigos fontes bem documentados e funções bem definidas).

O uso do MPI apresenta muitas vantagens conforme descrito anteriormente, porém também existem alguns contratempos, como o fato de ele não oferecer nenhum suporte para tolerância a falhas e assumir a existência de comunicações confiáveis. Outra desvantagem que pode-se considerar é o fato dele não ser um ambiente completo para programação concorrente, visto que o mesmo não implementa facilidades de

depuração (*debuger*) de programas concorrentes e canais virtuais para comunicação [MCB, 1994].

5.4 Composição do MPI

O MPI define um conjunto de aproximadamente 129 rotinas, que juntas oferecem os seguintes serviços:

- suporte para grupo de processos;
- suporte para contextos de comunicação;
- suporte para topologia de processos;
- comunicação ponto-a-ponto;
- comunicação coletiva.

Neste ponto, cabe ressaltar as definições de Grupos, Contextos e Comunicadores, pois são a base de funcionamento para as rotinas de comunicação ponto-a-ponto e coletiva, bem como para a aplicação de topologia de processos. Analisando cada um dos tópicos anteriores, tem-se:

Suporte para Grupos de Processos: o MPI relaciona os processos em grupos, e esses processos são identificados pela sua classificação dentro desse grupo. Por exemplo, suponha um grupo contendo n processos - estes processos serão identificados utilizando-se números inteiros entre 0 e n - 1. Essa classificação dentro do grupo é denominada ***rank***, ou seja, um processo no MPI é identificado por um ***grupo*** e por um ***rank*** dentro deste grupo. O MPI também apresenta primitivas de criação e destruição de grupos de processos.

Suporte para Contextos de Comunicação: os contextos de comunicação foram inicialmente propostos para permitir a comunicação de processos através de canais de fluxo de dados distintos. Estes escopos (*contextos de comunicação*) relacionam um determinado grupo de processos.

Os contextos de comunicação são implementados para assegurarem que uma mensagem enviada por um processo não seja incorretamente recebida por outro processo. Assim, um grupo de processos ligados por um contexto não consegue se comunicar com um grupo que esteja definido em outro contexto. Este tipo de estrutura não é visível nem controlável pelo usuário, e o seu gerenciamento fica sobre a responsabilidade do sistema MPI. Para criação de contextos, o MPI utiliza do conceito de ***communicator***, que é um objeto manuseado pelo programador e relaciona um grupo (ou grupos) de processos com um determinado contexto. Se existem, por exemplo, aplicações paralelas distintas executando em um mesmo ambiente, para cada uma delas será criado um ***communicator***. Isso criará contextos distintos que relacionarão os grupos de processos de cada aplicação e evitará que estes interfiram entre si.

Suporte para Topologias de Processos: o MPI fornece primitivas que permitem ao programador definir a estrutura topológica com a qual os processos de um determinado grupo se relacionarão. Como exemplo de uma topologia, pode-se citar uma malha, onde cada ponto de interseção na malha corresponde a um processo, ou topologia de grafos, que é mais genérica, onde os vértices correspondem aos membros do grupo e os arcos representam as ligações entre cada um dos membros. Em MPI, grupos podem ter ou não uma topologia associada. Para aqueles grupos que possuem topologia, cada membro tem associado, além do rank, uma localização específica dentro da topologia do grupo. O MPI mantém esta correspondência entre identificadores de tal forma que, a partir do

rank de um processo em um grupo, seja possível se determinar qual o lugar que este membro ocupa dentro da topologia do grupo e vice-versa.

Comunicação Ponto-a-Ponto: as rotinas de comunicação ponto-a-ponto são responsáveis pela ação básica de uma biblioteca de troca de mensagens, que se trata da transferência de uma mensagem. Cada transferência ponto-a-ponto envolve somente dois processos, um transmissor e um receptor.

Comunicação Coletiva: as rotinas de comunicação coletiva caracterizam-se pela participação de dois ou mais processos em cada operação de comunicação, voltadas para coordenar grupo de processos. Como exemplo, podem ser construídas a partir de rotinas de comunicação ponto-a-ponto. Uma função é denominada coletiva se todos os processo em um grupo necessitam executá-la para que ela tenha efeito. As comunicações coletivas no MPI têm por objetivo a transmissão de dados e/ou a sincronização de processos inseridos em um determinado contexto. As funções coletivas especificadas pelo padrão MPI são divididas em três grupos [SOU, 1997]:

- Sincronização.
- Envio de dados: Broadcast, Scatter/Gatter, All to All.
- Computação Coletiva: Min, Max, Add, Multiply, etc.

A noção de contextos e grupos são combinados em um simples objeto chamado comunicador [GRO, 1994]. Em uma comunicação coletiva ou em uma comunicação ponto-a-ponto entre membros de um mesmo grupo, somente um grupo precisa ser especificado, os processos (*transmissor e receptor*) são fornecidos pelos seus números de identificação (***rank***) dentro deste grupo. Em uma comunicação ponto-a-ponto entre processos de diferentes grupos, dois grupos devem ser especificados. Neste

caso, os processos (*transmissor e receptor*) devem fornecer suas identificações dentro dos respectivos grupos.

No ambiente lógico desse trabalho, as operações de comunicação ponto-a-ponto no MPI destacam-se como funções crucias para o desenvolvimento desta dissertação, que serão ilustradas na próxima seção.

5.5 Rotinas de Comunicação Ponto-a-Ponto do MPI

A necessidade de garantir eficiência e generalidade levou o Fórum MPI a definir um padrão relativamente extenso, de maneira que determinadas funções do MPI possuem muitas variações conforme (tabela 5.1). Dentro deste contexto, inserem-se as operações de comunicação ponto-a-ponto, que são caracterizadas basicamente por quatro aspectos básicos:

- Quanto ao bloqueio: pode ser bloqueante ou não bloqueante.
- Quanto ao modo de comunicação: pode ser *Ready, Padrão, Bufferizado* e *Síncrono*.
- Quanto a persistência: as rotinas podem ser ativadas por requisições persistentes (*iniciam apenas rotinas de comunicação não bloqueante, e são utilizadas para rotinas de transmissão em todos os modos de comunicação ou recepção de mensagens*) ou não persistentes.
- Quanto ao sentido da comunicação: as rotinas de comunicação podem ser em dois sentidos (variantes MPI_Sendrecv e MPI_Sendrecv_replace) ou isoladas.

A seguir, serão ilustrados os fundamentos julgados mais importantes da comunicação ponto-a-ponto.

5.5.1 Comunicação Bloqueante e Não Bloqueante

Uma função de envio bloqueante somente retorna quando o *buffer* utilizado para enviar a mensagem (*fornecido como parâmetro a função*) estiver livre para ser novamente utilizado pelo processo. Quando uma mensagem é enviada de forma não bloqueante, a função de envio pode retornar antes que o *buffer* possa ser alterado novamente. Deve-se garantir que qualquer alteração no conteúdo do *buffer* somente seja realizada quando da certeza de que não se irá mais afetar os dados da mensagem.

Na recepção bloqueante de mensagens, o processo destinatário é bloqueado até que a mensagem desejada esteja armazenada no *buffer* fornecido pelo processo. Um *receive* bloqueante não retorna enquanto a mensagem não for recebida e inserida no *buffer* [BEG, 1994], [GEI, 1994], [SUN, 1994]. A função de recepção não-bloqueante, no entanto, pode retornar antes que a mensagem tenha sido recebida. Neste caso, uma função de verificação terá de ser executada para indicar quando a mensagem estará, realmente, disponível ao receptor.

As rotinas de comunicação bloqueantes (capítulo 3, seção 3.3, figura 3.2 (a)) garantem que o processo transmissor ou receptor ficarão bloqueados até que a transmissão da mensagem seja completada, caso contrário, tem-se uma rotina não bloqueante.

5.5.2 Modos de Comunicação

O MPI estabelece quatro modos de comunicação que pode ser escolhido pelo usuário, estes modos indicam como o envio de uma mensagem deve ser realizado (*para cada um dos modos de comunicação estão associadas variantes bloqueantes e não bloqueantes para as rotinas de envio e recebimento de mensagem*).

No modo ***pronto (ready)*** a função de envio só ocorre com sucesso caso a recepção da mensagem já tenha sido iniciada pelo processo destinatário. É terminada rapidamente, sem a utilização de *buffers* ou de confirmações do processo receptor, objetivando conseguir melhor desempenho em determinados ambientes computacionais (*principalmente arquiteturas paralelas*).

No modo ***padrão (standard)*** as mensagens podem ser enviadas independentemente da função de recepção ter sido chamada ou não. Caso o envio da mensagem ocorra antes que a função de recepção tenha sido iniciada, a mensagem poderá ser armazenada internamente pelo sistema ou a função de envio poderá bloquear o processo remetente até que a mensagem seja recebida pelo processo destino.

O modo de envio ***bufferizado (buffered)*** é similar ao modo padrão, porém a mensagem deve ser obrigatoriamente armazenada pelo sistema caso a função de recepção não tenha sido iniciada no momento de envio da mensagem, isto é, após a mensagem ser copiada para o *buffer*, fica sobre a responsabilidade do sistema transmitir a mensagem quando possível. Neste caso, não há necessidade de que transmissor e receptor estejam sincronizados, possibilitando um maior desempenho no envio e

recebimento da mensagem, mas um possível problema que pode ocorrer é a sobrecarga gerada sobre a rede de trabalho, onde nós receptores lentos podem criar um congestionamento de mensagens não entregues, gerando um possível esgotamento dos *buffers*.

Por fim, no modo ***síncrono*** a função de envio somente retorna no momento em que a recepção da mensagem já tenha sido iniciada pelo processo destinatário [JUN, 1999]. O receptor deve enviar uma confirmação de recebimento da mensagem, de maneira que o transmissor possa ter certeza de que a mensagem foi recebida. Este modo penaliza o desempenho na troca de mensagens, porém consegue-se maior confiabilidade na entrega da mensagem e evita-se sobrecarga da rede por mensagens não recebidas. Além disso, possibilita que o comportamento de um programa paralelo seja melhor controlado, facilitando a sua depuração.

De acordo com o Fórum MPI [MPI, 2003], a tabela abaixo descreve um resumo das rotinas para comunicação ponto-a-ponto disponíveis no MPI.

		Padrão	**Síncrono**	***Bufferizado***	***Ready***
Bloqueantes		MPI_Send MPI_Recv	MPI_Ssend	MPI_Bsend	MPI_Rsend
	Comunicação em dois sentidos	MPI_Sendrecv MPI_Sendrecv_replace			
Não Bloqueantes		MPI_Isend MPI_Irecv	MPI_Issend	MPI_Ibsend	MPI_Irsend
	Requisições Persistentes	MPI_Send_init MPI_Recv_init	MPI_Ssend_init	MPI_Bsend_init	MPI_Rsend_init

Tabela 5.1 - Rotinas de Comunicação Ponto-a-Ponto do MPI.

5.6 Mensagem do MPI

Em uma operação genérica ***send*** e/ou ***receive***, uma mensagem consiste de um envelope indicando os processos de origem e destino, e de um corpo contendo o dado atual a ser enviado.

Esta Interface de Passagem de Mensagens utiliza três informações para caracterizar o corpo de uma mensagem em uma forma flexível [PAC, 2003]:

- Buffer - O endereço inicial da memória onde o dado de saída se encontra (*para uma operação de send*) ou onde o dado de entrada será armazenado (*operação receive*), isto é, a mensagem propriamente dita.

- Datatype - O tipo de dado a ser enviado. Em aplicações usuais, isto é, um tipo elementar como um inteiro ou um real, já em aplicações mais avançadas, este pode ser um tipo definido pelo usuário construído a partir de tipos básicos. Isto pode ser imaginado como uma estrutura em C, que pode conter dados em qualquer lugar, ou seja, não necessariamente em locais contíguos da memória. Esta habilidade de fazer uso de tipos definidos pelo usuário permite completa flexibilidade na definição do conteúdo de uma mensagem.
- Count - O número de itens (elementos) do tipo Datatype a ser enviado, ou seja, o tamanho da mensagem.

Por exemplo, (A,300,MPI_REAL) descreve um vetor A de 300 números reais, sem se preocupar com o tamanho ou formato de representação do número de ponto flutuante definida para uma determinada arquitetura. Uma implementação MPI para redes heterogêneas garante que os mesmos 300 números reais serão recebidos, mesmo se diferentes máquinas tenham uma representação distinta para ponto flutuante [GRO, 1994].

5.7 Tipo de dados MPI

Todas as mensagens MPI são tipadas, sendo que o tipo de conteúdo deve ser especificado tanto no envio quanto no recebimento. O usuário pode usar tanto os tipos de dado pré-definidos pelo padrão ou definir seus próprios tipos de acordo com suas necessidades.

5.7.1 Tipo de Dados Básico do MPI

Os tipos de dados básicos em MPI correspondem aos tipos básicos da linguagem C e Fortran. A tabela abaixo descreve as definições dos tipos de dados básicos do MPI em relação a linguagem C:

Definição Do MPI	Definição em C
MPI_CHAR	Signed char
MPI_INT	Signed int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_LONG_DOUBLE	long double
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_SHORT	Signed short int
MPI_BYTE	-
MPI_PACKED	-

Tabela 5.2 - Tipos de dados básicos no C.

5.7.2 Tipos Definidos pelo Usuário

Os mecanismos de comunicação MPI introduzidos anteriormente permite-nos enviar e receber uma seqüência de elementos idênticos que são contíguos na memória. Muitas vezes é desejável que se envie dados que não sejam homogêneos, como uma estrutura, ou que não seja contígua na memória, como uma parte de um array. Isto permite amortizar o *overhead* causado pelo envio e recebimento de muitos elementos sob um canal de comunicação. O MPI fornece dois mecanismos para realizar isto [SNI, 1996].

- O usuário pode definir tipos de dados derivados, que especificam arranjos (layouts) de dados mais gerais. Tipos de dados definidos por usuários podem ser usados em funções de comunicação MPI, em lugar dos tipos de dados pré-definidos.
- Um processo emissor pode explicitamente compactar dados não-contíguos em um buffer contíguo, e então enviá-lo. Um processo receptor pode explicitamente descompactar dados recebidos em um buffer contíguo e armazená-lo em locais não contíguos.

5.8 MPICH - Exemplo de uma Implementação do MPI

A implementação MPICH é o resultado de um trabalho conjunto entre as organizações americanas *Argone National Laboratory* e *Mississippi State University*, com contribuições de pesquisadores da IBM e estudantes voluntários, no qual é distribuída gratuitamente.

O desenvolvimento desta implementação, iniciou-se junto ao trabalho de definição do padrão MPI, sendo uma primeira amostra para os usuários em geral da interface MPI à medida que esta se desenvolvia [MSU, 2003].

O MPICH implementa a maioria das rotinas do padrão MPI versão 1.1 e foi desenvolvido a partir da combinação de implementações de plataformas de portabilidade já existentes e estáveis, o que permitiu que fosse implementado rapidamente, embora seu código fonte ainda continua em constante reestruturação. Dentre as principais plataformas e/ou ambientes de passagem de mensagens que

serviram de base para o desenvolvimento do MPICH, destacam-se: p4, *Chameleon* e *Zipcode*.

5.8.1 Arquitetura do MPICH

A estrutura do MPICH se divide em dois níveis. No primeiro concentra-se todo o código fonte reaproveitável (independente de *hardware*), que pode ser transportado diretamente, ou seja, é onde as funções do MPI são implementadas. No segundo nível situa-se o código dependente de plataforma e estes dois níveis comunicam-se através de uma camada denominada ADI (*Abstract Device Interface)*. Todas as funções do MPI são escritas utilizando-se rotinas e macros oferecidas pela ADI. É importante ressaltar que a ADI é projetada para permitir que qualquer biblioteca de passagem de mensagens possa ser implementada sobre ela.

Denomina-se dispositivo (***device***), à organização de *software* que se encontra no segundo nível do MPICH, é através de uma especificação precisa para a ADI que os fabricantes de *hardware*, por exemplo, forneçam implementações eficientes de dispositivos próprios. Pode-se também implementá-los a partir de rotinas de bibliotecas de passagem de mensagens nativas (da própria plataforma de *hardware*) ou plataformas de passagem de mensagens já existentes [MSU, 2003].

A figura abaixo apresenta um modelo resumido da estrutura do MPICH (versão 1.2.4) quando instalado sobre uma rede baseada no sistema operacional LINUX. O dispositivo utilizado é o ch_p4 (implementação mista baseada nas rotinas *Chameleon* e p4).

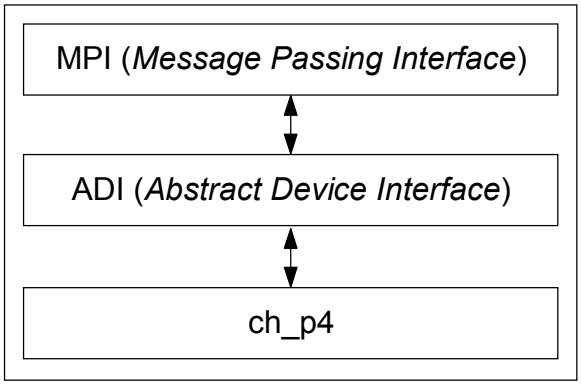

Figura 5.1 - Estrutura do MPI sobre o LINUX.

5.9 Considerações Finais

No final da década de 80, com o desenvolvimento da computação paralela sobre sistemas distribuídos, problemas relacionados a portabilidade, performance, funcionalidade e preço, determinaram a necessidade de se desenvolver um padrão.

Baseando-se nesses princípios, este padrão surgiu a partir da mobilização de entidades representando empresas e universidades que identificaram a importância de unificar seus esforços isolados, cada qual com seus bons resultados. Atualmente, o MPI (Message Passing Inteface) é um padrão entre os ambientes de passagens de mensagens, tornando-se uma referência de fato que continua em constante evolução.

O MPICH é uma implementação do padrão MPI distribuída livremente pela Internet [MSU, 2003], sendo a implementação escolhida para este trabalho em virtude da possibilidade encontrada de transporte do código fonte para novas plataformas de *hardware*, como o Multicomputador ACrux colocado nesta dissertação.

Outros motivos que levaram a escolha desta implementação, foram a sua grande popularidade, robustez, baixo custo, e o fato da mesma executar com eficiência em uma grande gama de máquinas paralelas (*arquiteturas paralelas de memória distribuída e centralizada*), redes de computadores (*redes de workstations e redes de computadores pessoais*) e ambientes heterogêneos [MPI, 2003].

Entre as rotinas de comunicação ponto-a-ponto no MPI a serem transportadas para o Multicomputador ACrux, destaca-se a Rotina ***Send( )*** síncrono e bloqueante, responsável pela transmissão de uma mensagem. Síncrono pelo fato de a rotina só se completar com o recebimento de uma confirmação de que a mensagem foi recebida, e bloqueante pelo fato de o processo ser bloqueado até que se copie a mensagem para o *buffer* de transmissão. Outra função colocada em questão é a Rotina ***Receive( )*** bloqueante, responsável pela recepção de uma mensagem. Bloqueante pelo fato de a rotina só se completar se a mensagem ativada for recebida.

A alteração destas funções nesta implementação MPICH, bem como outros detalhes de implementação das primitivas de comunicação da rede de trabalho e controle, serão melhores ilustrados no próximo capítulo.

6 Detalhes de Implementação do Multicomputador ACrux

6.1 Considerações Iniciais

Nos capítulos anteriores, fez-se uma revisão de conhecimentos necessários para a implementação de um ambiente de programação paralela - Multicomputador ACrux, proposto no presente trabalho de pesquisa. Neste capítulo, pretende-se abordar o processo em si, ou seja, as primitivas de comunicação da rede de trabalho, o qual optou-se por transportar para esta arquitetura as rotinas de comunicação ponto-a-ponto *MPI_Ssend* e *MPI_Recv* da implementação MPICH do padrão MPI [MPI, 2003], bem como outros detalhes de instalação e/ou configuração deste ambiente de programação paralela proposto.

Este cluster oferece um ambiente para a execução de programas paralelos organizados como redes de processos comunicantes, derivado do Multicomputador Crux [COR, 1999], sendo objeto de projetos correntes no ambiente do Laboratório de Computação Paralela e Distribuída (LaCPaD) do Curso de Pós-Graduação em Ciência da Computação (CPGCC) da Universidade Federal de Santa Catarina (UFSC).

6.2 O Multicomputador ACrux

A implementação do núcleo do sistema operacional distribuído ACrux [BUD, 2002], resultou um módulo que quando inserido no núcleo do Sistema Operacional Linux, independente da distribuição e da versão do kernel (2.2 ou 2.4), possibilita utilizar como rede de trabalho no âmbito do projeto Crux, um barramento

TCP/IP ou *FireWire* (*interface que permite a expansão de periféricos através de uma interconexão ponto-a-ponto*), substituindo o *Crossbar* da arquitetura original (capítulo 2, seção 2.2.4), na ausência de recursos financeiros.

Após ser instalado e adicionado no núcleo do sistema operacional carregável em memória, disponibiliza de forma transparente as camadas (*interface de acesso ao meio físico, interface da rede de controle e a interface da rede de trabalho*) que compõem o cluster ACrux (figura 6.1), permitindo acessá-las na forma de chamadas de sistema.

A interface de acesso ao meio físico escolhida para a propagação do presente trabalho de pesquisa foi o barramento TCP/IP, o qual foi elaborada para auxiliar na tarefa de implementação do núcleo do ACrux e do servidor de comunicações, isto é, um serviço de transporte orientado a conexão (*Transmission Control Protocol*) e um serviço de rede não orientado a conexão (*Internet Protocol*), através da utilização de sockets (*extremidade de um canal de comunicação entre processos, permitindo a transmissão bidirecional de dados*), uma vez que o desempenho almejado entre os nós de trabalho utilizando como rede de trabalho o barramento *FireWire* não se estabeleceu conforme a tecnologia propunha (IEEE 1394b).

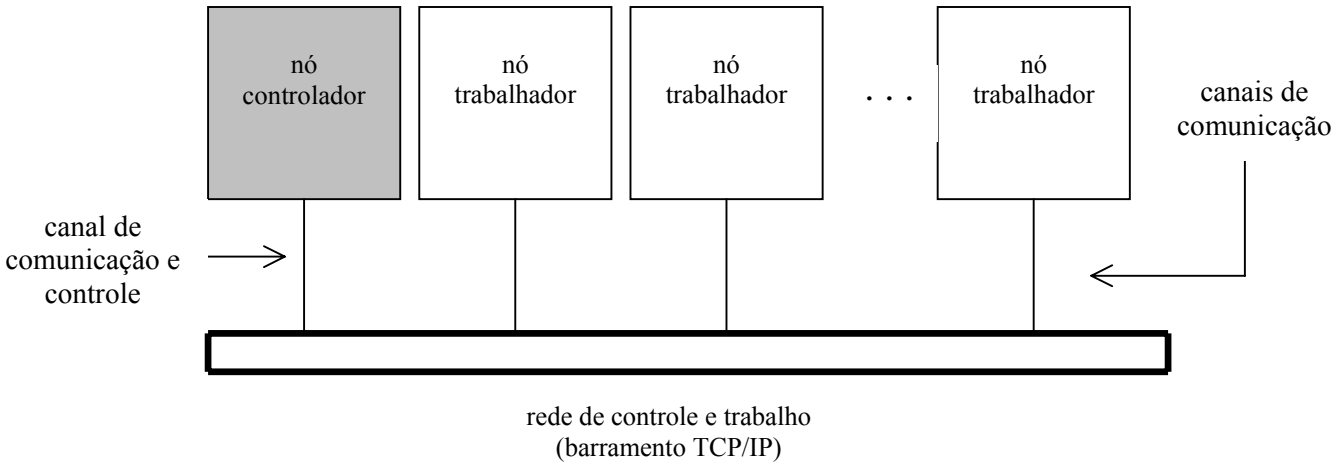

Figura 6.1 - Arquitetura do Multicomputador ACrux.

Com a utilização desta camada de acesso ao meio físico, permitiu-se desenvolver o Multicomputador ACrux sobre um barramento TCP/IP, incluindo todos os componentes da arquitetura original do cluster Crux [COR, 1999] (*nós de trabalho, rede de trabalho, nó de controle ou servidor de comunicações e rede de controle*). Esta rede em barra utilizando CSMA/CD (*Carrier Sense Multiple Access with Collision Detection*) como método de acesso, substituiu tanto a rede de controle quanto a rede de trabalho, sendo uma alternativa viável na ausência de uma rede de alta velocidade como ATM ou Gigabit Ethernet.

A troca de mensagens entre um nó de trabalho e o nó de controle para requisição de um serviço, é realizada com o auxílio das primitivas e/ou operadores ks_Send (*responde a uma requisição feita ao nó de controle por um nó de trabalho*), ks_ReceiveAny (*recebe requisições de quaisquer nós de trabalho*) e kc_SendReceive (*envia requisições ao nó de controle por um nó de trabalho, aguarda a resposta do nó de controle a um pedido realizado por um nó de trabalho*), que servem basicamente

para trocar um buffer que descreve o tipo de serviço requisitado entre um nó de trabalho e o nó de controle, formando assim a ***camada da rede de controle***.

A troca de mensagens entre nós de trabalho, são realizadas com o auxílio das primitivas e/ou operadores s_Send (*envia uma mensagem de um nó de trabalho a outro nó de trabalho*) e s_Receive (*recebe uma mensagem enviada de outro nó de trabalho*), formando assim a ***camada da rede de trabalho***.

Com o objetivo de facilitar a instalação e/ou configuração para o usuário final do Multicomputador ACrux, bem como trabalhos futuros que por ventura venham a utilizar esta arquitetura, criou-se dois *scripts* (Anexo B e C) para inserção e verificação do módulo no kernel, um especificamente para utilização no servidor de comunicações, executado no nó de controle (*responsável pela conexão e desconexão dos canais físicos diretos entre os nós de trabalho*), e outro executado nos nós de trabalhos (*executam os processos de programas paralelos*) do cluster ACrux.

6.3 A Implementação MPICH

O processo de instalação da implementação MPICH (versão 1.2.4) no nó controlador, por se tratar de uma biblioteca de passagem de mensagens de domínio público [MSU, 2003], propagou-se de maneira naturalmente, apenas houve a necessidade de configurá-la (Anexo F), desabilitando as bibliotecas de funções gráficas (MPE), bem como o sistema de arquivo paralelo (ROMIO) e os módulos que disponibilizam para o usuário final a possibilidade de escrever aplicações na linguagem Fortran, buscando uma maior otimização do ambiente como um todo.

O provimento do acesso a biblioteca de passagem de mensagens aos nós trabalhadores, foi estabelecido através da instalação e posterior configuração do serviço NFS (*Network File System*), ilustrado no (Anexo G). Este serviço foi desenvolvido pela Sun Microsystems e atualmente encontra-se disponível em várias plataformas, sendo o padrão para o compartilhamento de arquivos em um ambiente UNIX. No caso específico, permitiu o compartilhamento de arquivos do diretório de instalação da implementação MPICH no nó controlador, mantendo a transparência de acesso perante aos nós trabalhadores.

Com o ambiente aparentemente instalado, a atualização do arquivo de máquinas (Anexo D), é outro pré requisito indispensável para o funcionamento da biblioteca, usualmente encontra-se no diretório de instalação da própria implementação MPICH. Sua funcionalidade neste trabalho é ilustrar todos os nós envolvidos no processamento MPI, inclusive o servidor de comunicações - servidor NFS, servindo como base de consulta no momento da inicialização dos processos (*hosts*) envolvidos (*script mpirun*).

6.3.1 Execução Remota de Comandos

Em uma outra etapa da montagem deste ambiente de programação paralela, além das definições de rede (Anexo A) cruciais para o desenvolvimento, surgiu a necessidade de habilitar nas máquinas que compõem o cluster ACrux a permissão de execução de comandos remotos. Este recurso será utilizado principalmente pelo nó controlador, ou seja, para que o mesmo possa inicializar os nós trabalhadores e consequentemente distribuir as tarefas de acordo com os programas de usuário MPI.

Este recurso obteve-se através do programa rsh (*shell remoto*), que é uma parte (*pacote*) integrante da instalação de qualquer distribuição Linux padrão. Para tanto, foi atualizado o arquivo /etc/hosts.equiv (Anexo D) para conter o nome de todos os nós do cluster alvo e criado o arquivo /root/.rhosts (Anexo D) no diretório home do super usuário (*root*) para a execução de comandos de administração.

Embora os conteúdos dos arquivos sejam os mesmos, a função que cada um exerce perante ao sistema operacional é ligeiramente diferente. O arquivo de nome ***hosts.equiv*** torna as máquinas "equivalentes" permitindo que um usuário comum execute um comando remotamente em outra máquina, sendo um artifício que será utilizado para rodar por exemplo uma aplicação MPI. Por sua vez, o arquivo de nome ***.rhosts*** é utilizado apenas pelo usuário root para fins de administração, por exemplo, para desligar os nós trabalhadores que compõem o cluster ACrux (Anexo E), pois os únicos dispositivos de entrada e saída de que eles necessitam são suas interfaces de rede, dispensando o uso de teclados e monitores.

6.4 Aspectos de portabilidade e Implementação

Os operadores *s_Send* e *s_Receive* da camada da rede de trabalho, utilizam como interface ao núcleo do sistema operacional distribuído ACrux os operadores do núcleo *Connect e Disconnect*, responsáveis respectivamente pela conexão e desconexão de canais físicos diretos entre os nós de trabalho. Ao requisitar ao nó de controle o estabelecimento de um canal de comunicação perante a rede de trabalho, os nós de trabalho envolvidos na comunicação são identificados pelas variáveis *node* e *thisNode*. Após o estabelecimento do link de comunicação, pode-se afirmar que trata-se de uma

comunicação direta pelo fato da camada da rede de trabalho não criar conexões duradouras entre os processos envolvidos, mantendo um mecanismo de comunicação genérico com conexão dinâmica de canais físicos por demanda.

No caso específico das rotinas de comunicação ponto-a-ponto *MPI_Ssend* (figura 6.2) e *MPI_Recv* (figura 6.3) da implementação MPICH, cujo o principal objetivo é o envio ou recebimento de uma mensagem dado um determinado tipo e número de elementos a serem enviados ou recebidos, os processos envolvidos na comunicação são identificados na própria chamada de sistema da função correspondente, ou seja, o nó de trabalho origem pela variável *source* e o nó de trabalho destino pela variável *dest*.

```
//*********************************************************************
int MPI_Ssend( void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm )
{

//*********************************************************************

  int ret = 0;
  int node = dest+1;
  int thisNode = _thisProc;

  ret = s_Send ( thisNode, node, (void*)buf, count );

  return 0;

//*********************************************************************
```

Figura 6.2 - Rotina para comunicação ponto-a-ponto MPI_Ssend.

```
//*****************************************************************
int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Status *status )
{
   struct MPIR_COMMUNICATOR *comm_ptr;
   struct MPIR_DATATYPE     *dtype_ptr;
   static char myname[] = "MPI_RECV";
   int        mpi_errno = MPI_SUCCESS;

//*****************************************************************

  int ret = 0;
  int node = source+1;
  int thisNode = _thisProcid;

  ret = s_Receive ( thisNode, node, buf, &count );

  return 0;

//*****************************************************************
```

Figura 6.3 - Rotina para comunicação ponto-a-ponto MPI_Recv.

Outro aspecto de portabilidade das rotinas de comunicação ponto-a-ponto *MPI_Ssend* e *MPI_Recv* da implementação MPICH para o ambiente ACrux que merece destaque, é o inter-relacionamento entre os parâmetros trocados nas diferentes funções tratadoras de envio ou recebimento da mensagem na camada da rede de trabalho, ilustradas nas implementações acima pelo uso das variáveis *buf* e *count* que indicam respectivamente o endereço da aplicação *buffer* (dado de saída ou entrada) - *buffer* que contém a mensagem e o número de elementos a serem enviados ou recebidos - count *buffer* ou tamanho da mensagem. Estas variáveis são repassadas como parâmetros para as primitivas *MPI_Ssend* e *MPI_Recv* da implementação MPICH através de uma aplicação MPI, sendo utilizadas logo em seguida na sua forma original pelos operadores

da rede de trabalho *s_Send* e *s_Receive*, visando o estabelecimento de uma conexão ou desconexão através dos operadores do núcleo *Connect e Disconnect*.

Estes aspectos influenciaram positivamente na portabilidade das rotinas de comunicação ponto-a-ponto do MPI para o multicomputador ACrux, visto a semelhança aqui encontrada na utilização entre os parâmetros trocados nas diferentes funções tratadoras de envio ou recebimento de mensagens. O código fonte resultante das chamadas de sistema (*MPI_Ssend* e *MPI_Recv*) da implementação MPICH, visando o envio ou recebimento de uma mensagem entre nós de trabalhos, utilizando como base o sistema operacional distribuído ACrux, incluindo os *scripts* necessários para a instalação ou configuração deste ambiente de passagem de mensagens, estão presentes nos anexos deste trabalho.

6.5 Considerações Finais

Este capítulo reservou-se principalmente a comentar os detalhes de implementação do cluster ACrux. Vale ressaltar todo o estudo de identificação das estruturas de comunicação ponto-a-ponto da implementação MPICH do padrão MPI, a implementação (Anexo H e I) destes mecanismos de comunicação (*síncrono e bloqueante*) para o envio ou recebimento de uma mensagem MPI entre nós de trabalho neste ambiente, as primitivas da interface da rede de controle e da interface da rede de trabalho, alguns operadores julgados necessários do núcleo do sistema operacional distribuído ACrux, que foi tema de outra dissertação de mestrado [BUD, 2002].

Durante o desenvolvimento do trabalho, optou-se por utilizar como camada de acesso ao meio físico o barramento TCP/IP, devido a maioria das bibliotecas de passagem de mensagens estarem passíveis a este meio de transporte. Outros meios físicos recentes, como, por exemplo *SCSI*, *USB, Fibre Channel* e o próprio *FireWire*, bem como a rede de interconexão dinâmica *Crossbar* da arquitetura original, poderiam ser utilizados como rede de trabalho almejando um maior desempenho na troca de mensagens entre nós trabalhadores, uma vez que fossem adaptados e tratados em camadas inferiores no Multicomputador ACrux.

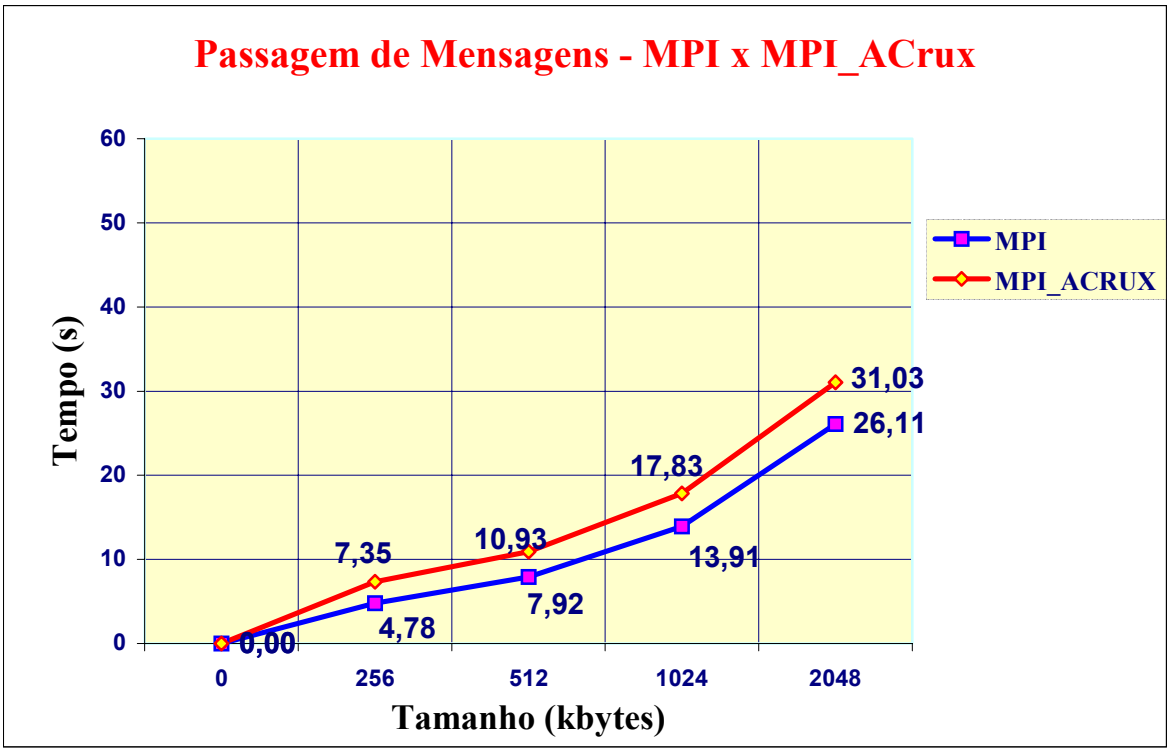

Figura 6.4 - Estudo Comparativo entre mensagens do tipo MPI x MPI_ACrux.

Finalmente, após a realização de testes no ambiente de programação paralela proposto, conforme apresenta a figura 6.4, faz-se necessário informar que os resultados referente ao tempo expressos em segundos, foram coletados através da chamada de sistema *time* padrão do sistema operacional Linux. Sendo assim, pode-se afirmar que o mesmo está preparado para receber, compilar e executar aplicações de usuários MPI (Anexo J) que envolvam as rotinas básicas para comunicação ponto-a-ponto (MPI_Ssend e MPI_Recv) com desempenho satisfatório, quando comparados com o modelo tradicional de envio e/ou recebimento de mensagens MPI (bloqueante síncrona). Evitou-se também, a alteração de parâmetros da chamada de sistema no seu formato usual, ou modificações nas aplicações MPI, no sentido de buscar o funcionamento das chamadas MPI_Ssend e/ou MPI_Recv no ACrux, mantendo a transparência de acesso ao usuário final.

7 Conclusão

A utilização de sistemas computacionais distribuídos, com conceitos utilizados pela computação paralela, fez com que trabalhos fossem desenvolvidos para explorar o potencial da computação paralela utilizando os sistemas distribuídos. Baseando-se nesses princípios, ambientes de passagem de mensagem foram criados e posteriormente aperfeiçoados para arquiteturas baseadas em *cluster de computadores*.

Esta dissertação, no âmbito do projeto Crux, se propunha a identificar as estruturas de comunicação e sincronização da biblioteca de passagem de mensagens da implementação MPICH do padrão MPI (*Message Passing Interface*), visando aspectos de portabilidade das rotinas básicas que permitem uma comunicação ponto-a-ponto de forma *bloqueante síncrona*. Os passos necessários para implementação deste ambiente, Multicomputador ACrux, foram uma revisão bibliográfica na área de computação paralela e distribuída, um estudo da arquitetura Crux e do módulo do sistema operacional distribuído ACrux, bem como as interfaces de passagem de mensagens recentemente disponíveis, todos eles descritos nos capítulos anteriores.

A partir dos conhecimentos adquiridos e dos trabalhos de implementação no cluster alvo, pode-se afirmar que este ambiente está preparado para receber, compilar e executar aplicações de usuários MPI que envolvam as rotinas básicas para comunicação ponto-a-ponto (MPI_Ssend e MPI_Recv), mantendo a transparência do modelo tradicional de forma a evitar a modificação da chamada de sistema do seu formato original.

7.1 Trabalhos Futuros

Após verificada a possibilidade do transporte de código fonte das rotinas básicas para este Multicomputador, buscando o aprimoramento deste trabalho, alguns estudos fazem-se necessários. Primeiramente, a verificação da possibilidade de implementação de um sistema de tolerância a faltas para o nó controlador, através da eleição de um novo servidor de comunicações entre os nós trabalhadores disponíveis. Outro trabalho proposto seria a implementação de um *console* para fins de administração no nó controlador, similar ao PVM que permite adicionar ou remover um *host* de forma dinâmica no processamento. Num terceiro momento, um estudo e implementação das rotinas de comunicação coletiva e/ou funções avançadas do MPI neste ambiente, visando um maior aproveitamento na camada da rede de trabalho. Outro mecanismo que sensivelmente aumentaria o desempenho na troca de mensagens entre nós trabalhadores, seria a substituição da camada de acesso ao meio físico, isto é, o barramento TCP/IP que utiliza o CSMA/CD como método de acesso por uma interface mais recente, como, por exemplo *SCSI*, *USB, Fibre Channel, FireWire*, uma vez que fossem adaptados e tratados em camadas inferiores do ACrux.

Estima-se que o projeto ACrux, devido a sua amplitude, proporcione de forma direta ou indireta o surgimento de outros trabalhos que aproveitem este mecanismo implementado, e continue em desenvolvimento paralelo ao avanço tecnológico do hardware e software. Dessa forma, interface de passagem de mensagens portáteis como PVM, podem sofrer adaptações para tirarem proveito desse mecanismo, uma vez ilustrado a viabilidade do MPI, que foi foco desta dissertação.

Em suma, a construção do ACrux, vem provar não só a viabilidade de sua arquitetura embasada no Multcomputador Crux, mas a possibilidade de adaptação de outras bibliotecas de passagem de mensagens, proporcionando consequentemente maximização dos benefícios.

Referências Bibliográficas

[ALM, 1994] ALMASI, G. S; GOTTLIEB A. **High Parallel Computing**. 2ª ed, The Benjamin Cummings Publishing Company, Inc., 1994.

[ALV, 2002] ALVES, Marcos J. P. **Construindo Supercomputadores com Linux**. Rio de Janeiro: Brasport, 2002.

[ARC, 2002] Disponível via www em: http://www.fe.up.pt/~goii2000/M3/redes4.htm. Consultado em dezembro de 2002.

[BAR, 1999] BARRETO, Marcos E; NAVAUX, Philippe O.A. **Um Ambiente para Execução de Aplicações Paralelas em Agregados de Multiprocessadores.** Universidade Federal do Rio Grande do Sul. Porto Alegre. 1999.

[BEC, 1997] BECKER, Donald; MERKEY, Philip; RIDGE, Daniel; STERLING, Thomas. **Beowulf Harnessing the power of Paralellism in a Pile- of-PC´s** IEEE Aerospace, 1997.

[BEG, 1994] BEGUELIN, Adam; et al. **PVM: Parallel Virtual Machine**: A User´s Guide and Tutorial for Networked Parallel Computing. The MIT Press 1994.

[BEN, 1993] BEN-DYKE, A. D. **Architectural taxonomy**: A brief review. University of Birmingham, 1993.

[BEO, 2002] Disponível via www em: http://www.beowulf.org/beowulf/history.html. Consultado em novembro de 2002.

[BUD, 2002] BUDAG, Karlos H. **Implementação do Núcleo do Sistema Operacional Distribuído ACrux**. Dissertação de Mestrado. CPGCC. Universidade Federal de Santa Catarina. Florianópolis. 2002.

[CES, 2002] Disponível via www em: http://cesdis.gsfc.nasa.gov/beowulf/papers/HPDC96/hpdc96.html ou http://cesdis.gsfc.nasa.gov/beowulf/papers/HPDC96/hpdc96.ps. Consultado em setembro de 2002.

[CAL, 1994] CALKIN, R. **Portable programming with the PARMACS message-passing library**: Parallel Computing, v. 20, pp. 615-632, 1994.

[COL, 1994] COLOURIS, G; DOLLIMORE, J; KINDBERG, T. **Distributed System Concepts and Design**. 2ª ed, Addison-Wesley Publishing Company, 1994.

[COR, 1999] CORSO, Thadeu B. **Crux: Ambiente Multicomputador Configurado por Demanda**. Tese de Doutorado. CPGEE. Universidade Federal de Santa Catarina. Florianópolis. 1999.

[DAN, 2002] DANTAS, Mário Antônio Ribeiro. **Tecnologias de Redes de Comunicação e Computadores**. Axcel Books do Brasil, 2002.

[DIE, 1997] DIETZ, Hank. **LINUX Parallel Processing Using Clusters**. Purdue University School of Eletrical and Computer Engineering, 1997.

[DUN, 1990] DUNCAN, R. **A Survey of Parallel Computer Architectures, IEEE Computer**, pp.5-16, Fevereiro, 1990.

[FLY, 1972] FLYNN, M. J **Some Computer Organizations and Their Effectiveness IEEE Transactions on Computers**, v. C-21, pp.948-960, 1972.

[FLO, 1994] FLOWER, J; Kolawa A. **Express is not just a message passing system. Current and future directions in Express**: Parallel Computing, v. 20, pp. 597-614, 1994.

[GEI, 1994] GEIST, A; et al. **PVM3 User´s Guide and Reference Manual**. OAK National Laboratory. Setembro de 1994.

[GON, 1997] GONZAGA, A.M. **Máquina Paralela Virtual**. ICEB-UFOP. Novembro de 1997.

[GRO, 1994] GROPP, Willian; LUSK, Ewing; SKJELLUM, Anthony. **Using MPI**: Portable Parallel Programming with the Message-Passing Interface. Massachusetts Institute of Technology, 1994.

[JUN, 1999] JUNIOR, José Mazzuco. **Uma abordagem híbrida do problema da produção através dos algoritmos Simulated Annealing e Genético**. Tese de Doutorado. Programa de Pós-Graduação em Engenharia de Produção. Universidade Federal de Santa Catarina. Florianópolis. 1999. Disponível via www em: http://www.eps.ufsc.br/teses99/mazzuco/. Consultado em fevereiro de 2003.

[KLA,2002] Disponível via www em: http://www.inf.ufrgs.br/gpesquisa/propcar/disc/cmp134/trabs/T1/011/klat2. Consultado em setembro de 2002.

[MCB, 1994] MCBRYAN, O. A. **An overview of message passing environments**: Parallel Computing, v. 20, pp. 417-444, 1994.

[MEM, 2000] MENDES, A. L. de Andrade. **Migração de Aplicações para Clusters de PC's**. Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e Tecnológicas. Plano de Dissertação de Mestrado. Curitiba. Fevereiro de 2002.

[MYR, 2002] Disponível via www em: http://inf.ufrgs.br/gpesquisa/propcar/disc/cmp134/tras/T1/001/myrinet. Consultado em outubro de 2002.

[MPI, 2003] MPI Fórum. Disponível via www em: http://www.mpi-forum.org/docs/docs.html. Consultado em março de 2003.

[MSU, 2003] Implementação MPICH. Disponível via www em: http//www-unix.mcs.anl.gov/mpi/mpich/docs.html. Consultado em março de 2003.

[MUL, 1993] MULLENDER. **Distributed System**. 2ª ed, ACM PRESS Frontier Series Addison-Wesley Publishing Company, 1993.

[OVE, 2002] Disponível via www em: http://www.myri.com/myrenet/overview/index.html. Consultado em outubro de 2002.

[PAC, 2003] PACS Training Group. Introduction to MPI. University of Illinois.Disponível via www em: http://webct.ncsa.uiuc.edu:8900/webct/public/home.pl. Consultado em fevereiro de 2003.

[PIT, 2002] Disponível via www em: http://www.tiraduvidas.hpg.ig.com.br/Dicas/030801a.htm. Consultado em novembro de 2002.

[QUI, 1987] QUINN, M. J. **Designing Efficient Algorithms for Parallel Computers** McGraw Hill, 1987.

[SCH, 1999] SCHWEITZER, Marc Alexander; ZUMBUSCH, Gerhard; GRIEBEL, Michael. **Parnass2**: A Cluster of Dual-Processor PC's. Editor: W. Rehm, T. Ungerer, Chemnitzer Informatik Berichte, 1999.

[SOA, 1997] SOARES, Luiz Fernando G; LEMOS, G; COLCHER, S. **Redes de Computadores**: Das LANs, MANs e WANs às redes ATM. 2ª ed. Rio de Janeiro: Campus, 1997.

[SOU, 1996] SOUZA, Paulo S. Lopes de. **Máquina Paralela Virtual em Ambiente Windows**. Dissertação de Mestrado. Instituto de Ciências Matemáticas de São Carlos. Universidade de São Paulo. Maio de 1996.

[SOU, 1997] SOUZA, Paulo S. Lopes de. **MPI - Um Padrão para Ambientes de Passagem de Mensagem**. IFSC. Universidade de São Paulo. Novembro de 1997.

[SUN, 1994] SUNDERAM, V.S; et al. **The PVM Concorrent Computing System: evolution, experiences and trends, Parallel Computing**, v.20, 1994.

[SNI, 1996] SNIR, Marc; OTTO, Steve; LEDEMAN, Steven. Huss; DONGARRA, Jack. **MPI**: The Complete Reference. Ed. Mit Press. Cambridge, 1996.

[TAN, 1997] TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. Ed: Prentice Hall do Brasil Ltda. Rio de Janeiro. 1997.

[TOP, 2002] Disponível via www em: http://www.top500.org. Consultado em dezembro de 2002.

[USP, 2002] Disponível via www em: http://www.uspnet.usp.br/evolucao/node15.html. Consultado em setembro de 2002.

[WAL, 1994] WALKER, D. W. **The design of a standard message passing interface for distributed memory concurrent computers**: Parallel Computing, vol. 20, pp. 657-673, 1994.

[ZAL, 1991] ZALUSKA, E. J. **Research lines in distributed computing systems and concurrent computation**. Anais do Workshop em Programação Concorrente. Sistemas Distribuídos e Engenharia de Software, 1991.

[ZOM, 1996] ZOMAIA, Albert Y. H. **Parallel & Distributed Computing HandBook**. Series on Computer Engineering. Ed: McGraw-Hill. E.U.A 1996. Albert Y.H.

Anexos

Anexo A – Definições de Rede

Identificação	Nome	Endereço IP	Máscara da sub-rede de acesso
Nó controlador	host1	192.168.1.1	255.255.255.0
Nó trabalhador	host2	192.168.1.2	255.255.255.0
Nó trabalhador	host3	192.168.1.3	255.255.255.0
Nó trabalhador	host4	192.168.1.4	255.255.255.0
...	...	...	...

Conteúdo do arquivo /etc/hosts

```
127.0.0.1          localhost
192.168.1.1        host1
192.168.1.2        host2
192.168.1.3        host3
192.168.1.4        host4
#  ...              ...
```

Anexo B – Script Inicializa Nó Controlador

```
#inicializa o host controlador (rede de controle) do Multicomputador ACrux

if [ -z $1 ]; then

echo "Forneca o endereco ip do no controlador"

echo "Exemplo: ./no_controlador 127.0.0.1"

exit 1

fi

rmmod MainCrux

rm -r MainCrux.o

rm -r SrvNode

make MainCrux.o

insmod MainCrux.o addr=$1

echo

echo ">>> Primitivas instaladas com sucesso, No Controlador <<<"

make SrvNode

./SrvNode
```

Anexo C – Script Inicializa Nó Trabalhador

```
#inicializa um host trabalhador (rede de trabalho) do Multicomputador ACrux

if [ -z $1 ]; then

echo "Forneca o endereco ip do no controlador"

echo "Exemplo: ./no_trabalhador 127.0.0.1"

exit 1

fi

rmmod MainCrux

rm -r MainCrux.o

make MainCrux.o

insmod MainCrux.o addr=$1

echo

echo ">>> Primitivas instaladas com sucesso, No Trabalhador <<<"
```

Anexo D – Conteúdo dos arquivos hosts.equiv, .rhosts e machines.Linux

```
host1

host2

host3

host4

# ...
```

Anexo E – Script Desliga_Máquinas.sh

```
# Desliga os nós trabalhadores do Cluster ACrux
#
echo "Desligando os hosts..."
for i in 2 3 4
   do
        rsh host$i /sbin/shutdown -h now
   done
   sleep 5
echo "Pronto"
```

Anexo F – Configurador da Biblioteca de Passagem de Mensagens - Implementação MPICH

```
./configure --with-device=ch_p4 --with-arch=LINUX --without-mpe --without-romio

--disable-f77 --disable-f90 --with-f90nag --with-f95nag --disable-f90modules

-prefix=/usr/local/mpi/
```

Anexo G – Configuração do Network File System (NFS) - Nó Controlador e Nó trabalhador

```
#Nó controlador (servidor de arquivos)

#inicializa os daemons portmap e nfs

./etc/rc.d/init.d/portmap start

./etc/rc.d/init.d/nfs start

#exporta o conteúdo do arquivo /etc/exports, diretório (mpi) utilizado pelos clientes

exportfs -a



#Nó trabalhador (estação de trabalho)

#inicializa os daemons portmap e nfs

./etc/rc.d/init.d/portmap start

./etc/rc.d/init.d/nfs start

#cria diretório (mpi), ponto de montagem para acessar o diretório remoto

mkdir /mpi/

#monta diretório remoto

mount -t nfs 192.168.1.1 :/usr/local/mpi/ /mpi/
```

Conteúdo do arquivo /etc/exports

```
#exporta o diretório de instalação do mpi (nó controlador) com permissão de leitura e
#escrita para os nós trabalhadores

/usr/local/mpi/          (rw)
```

Anexo H – Chamada de sistema para comunicação ponto-a-ponto MPI_Ssend

Listagem do Arquivo ssend.c

```
/*
 *  $Id: ssend.c,v 1.10 2001/11/14 20:10:03 ashton Exp $
 *
 *  (C) 1993 by Argonne National Laboratory and Mississipi State University.
 *      See COPYRIGHT in top-level directory.
 */

#include "mpiimpl.h"
#ifdef HAVE_WEAK_SYMBOLS
#if defined(HAVE_PRAGMA_WEAK)
#pragma weak MPI_Ssend = PMPI_Ssend
#elif defined(HAVE_PRAGMA_HP_SEC_DEF)
#pragma _HP_SECONDARY_DEF PMPI_Ssend  MPI_Ssend
#elif defined(HAVE_PRAGMA_CRI_DUP)
#pragma _CRI duplicate MPI_Ssend as PMPI_Ssend
/* end of weak pragmas */
#endif

/* Include mapping from MPI->PMPI */
#define MPI_BUILD_PROFILING
#include "mpiprof.h"
/* Insert the prototypes for the PMPI routines */
#undef __MPI_BINDINGS
#include "binding.h"
#endif

/*@
   MPI_Ssend - Basic synchronous send

Input Parameters:
+ buf - initial address of send buffer (choice)
. count - number of elements in send buffer (nonnegative integer)
. datatype - datatype of each send buffer element (handle)
. dest - rank of destination (integer)
. tag - message tag (integer)
- comm - communicator (handle)
.N fortran

.N Errors
```

```
.N MPI_SUCCESS
.N MPI_ERR_COMM
.N MPI_ERR_COUNT
.N MPI_ERR_TYPE
.N MPI_ERR_TAG
.N MPI_ERR_RANK
@*/

//*********************************************************************

//Definicoes Primitivas Acrux
include "/Acrux/MainCrux.h"
define _print printf
int _serverAddr = 0;
int _thisProc = 1;

//*********************************************************************

int MPI_Ssend( void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm )
{

//*********************************************************************

  int ret = 0;
  int node = dest+1;
  int thisNode = _thisProc;

   ret = s_Send ( thisNode, node, (void*)buf, count );

   return 0;

//*********************************************************************

/*
   int        mpi_errno = MPI_SUCCESS;
   MPI_Request handle;
   MPI_Status  status;
   MPIR_ERROR_DECL;
   struct MPIR_COMMUNICATOR *comm_ptr;
   static char myname[] = "MPI_SSEND";

   if (dest != MPI_PROC_NULL)
   {
        comm_ptr = MPIR_GET_COMM_PTR(comm);
        MPIR_TEST_MPI_COMM(comm,comm_ptr,comm_ptr,"MPI_SSEND");
        MPIR_ERROR_PUSH(comm_ptr);
        MPIR_CALL_POP(MPI_Issend( buf, count, datatype, dest, tag, comm,
```

```
                    &handle ),comm_ptr,myname);

        MPIR_CALL_POP(MPI_Wait( &handle, &status ),comm_ptr,myname);
        MPIR_ERROR_POP(comm_ptr);
    }
    return mpi_errno;
*/
}
```

Anexo I – Chamada de sistema para comunicação ponto-a-ponto MPI_Recv

Listagem do Arquivo recv.c

```
/*
 *  $Id: recv.c,v 1.10 2001/11/14 20:10:01 ashton Exp $
 *
 *  (C) 1993 by Argonne National Laboratory and Mississipi State University.
 *      See COPYRIGHT in top-level directory.
 */

#include "mpiimpl.h"
#ifdef HAVE_WEAK_SYMBOLS
#if defined(HAVE_PRAGMA_WEAK)
#pragma weak MPI_Recv = PMPI_Recv
#elif defined(HAVE_PRAGMA_HP_SEC_DEF)
#pragma _HP_SECONDARY_DEF PMPI_Recv  MPI_Recv
#elif defined(HAVE_PRAGMA_CRI_DUP)
#pragma _CRI duplicate MPI_Recv as PMPI_Recv
/* end of weak pragmas */
#endif

/* Include mapping from MPI->PMPI */
#define MPI_BUILD_PROFILING
#include "mpiprof.h"
/* Insert the prototypes for the PMPI routines */
#undef __MPI_BINDINGS
#include "binding.h"
#endif

/*@
   MPI_Recv - Basic receive

Output Parameters:
+ buf - initial address of receive buffer (choice)
- status - status object (Status)

Input Parameters:
+ count - maximum number of elements in receive buffer (integer)
. datatype - datatype of each receive buffer element (handle)
. source - rank of source (integer)
. tag - message tag (integer)
- comm - communicator (handle)
```

```
Notes:
The 'count' argument indicates the maximum length of a message; the actual
number can be determined with 'MPI_Get_count'.

.N fortran
.N Errors
.N MPI_SUCCESS
.N MPI_ERR_COMM
.N MPI_ERR_TYPE
.N MPI_ERR_COUNT
.N MPI_ERR_TAG
.N MPI_ERR_RANK
@*/

//*****************************************************************

//Definicoes primitvas Acrux
include "/Acrux/MainCrux.h"
define _print printf
int _serverAddr = 0;
int _thisProcid = 2;

//*****************************************************************

int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Status *status )
{
   struct MPIR_COMMUNICATOR *comm_ptr;
   struct MPIR_DATATYPE     *dtype_ptr;
   static char myname[] = "MPI_RECV";
   int        mpi_errno = MPI_SUCCESS;

//*****************************************************************

  int ret = 0;
  int node = source+1;
  int thisNode = _thisProcid;

    ret = s_Receive ( thisNode, node, buf, &count );

    return 0;

//*****************************************************************

/*
   struct MPIR_COMMUNICATOR *comm_ptr;
   struct MPIR_DATATYPE     *dtype_ptr;
```

```
   static char myname[] = "MPI_RECV";
   int        mpi_errno = MPI_SUCCESS;
   if (source != MPI_PROC_NULL)
   {
         comm_ptr = MPIR_GET_COMM_PTR(comm);
         MPIR_TEST_MPI_COMM(comm,comm_ptr,comm_ptr,myname);

         dtype_ptr = MPIR_GET_DTYPE_PTR(datatype);
         MPIR_TEST_DTYPE(datatype,dtype_ptr,comm_ptr,myname);

#ifndef MPIR_NO_ERROR_CHECKING
         MPIR_TEST_COUNT(count);
         MPIR_TEST_RECV_TAG(tag);
         MPIR_TEST_RECV_RANK(comm_ptr,source);
         if (mpi_errno)
            return MPIR_ERROR(comm_ptr, mpi_errno, myname );
#endif

         MPID_RecvDatatype( comm_ptr, buf, count, dtype_ptr, source, tag,
                            comm_ptr->recv_context, status, &mpi_errno );
         MPIR_RETURN(comm_ptr, mpi_errno, myname );
   }
   else {
         MPID_ZERO_STATUS_COUNT(status);
         status->MPI_SOURCE     = MPI_PROC_NULL;
         status->MPI_TAG        = MPI_ANY_TAG;
         }
   return MPI_SUCCESS;
*/
}
```

Anexo J – Exemplo de uma aplicação MPI

```
# include "mpi.h"       /*Biblioteca MPI*/

main(argc, argv)
int argc;
char *argv[ ];

{
char msg[256];                          /*Mensagem a ser enviada*/
int myrank;                                     /*Rank de um processo*/
int tag = 99;                           /*Identificador da mensagem (tag)*/
MPI_Status status;              /*Variavel status,utilizada pela rotina receive( )*/

MPI_Init(&argc,&argv);                  /*Inicia o MPI*/
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);    /*Define    o    rank    do
processo*/
printf("%d\n" ,&MPI_Comm_rank);

/*O processo com rank 0 envia a mensagem, o de rank 1 a recebe*/
if (myrank == 0)   {
        strcpy(msg, "Enviando Mensagem Síncrona!");
        MPI_Ssend(msg, 256, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
        printf("%d\n" ,MPI_COMM_WORLD);
        }

        else   {
                MPI_Recv(msg,   256,   MPI_CHAR,   0,   tag,   MPI_COMM_WORLD,
&status);
                printf("%d\n" ,MPI_COMM_WORLD);
                printf( "%s \n", msg);
            }
// MPI_Finalize( );             /*Finaliza o MPI*/
}
```