Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Pós-Graduação em Ciência da Computação

MDX-cc: Ambiente de Programação Paralela Aplicado a *Cluster* de *Clusters*

Cassiano Ricardo Hess

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: César A. F. De Rose

Porto Alegre, março de 2003

Aos meus pais, meus irmãos e minha esposa Samantha

Dados Internacionais de Catalogação na Publicação (CIP)

H586m Hess, Cassiano Ricardo
MDX-cc: ambiente de programação paralela aplicado a *cluster* de *clusters* / Cassiano Ricardo Hess. – Porto Alegre, 2002.
132 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.

1. Arquitetura de Computador. 2. Programação Paralela. 3. Cluster - Informática. 4. Análise de Desempenho de Computadores. I. Título.

CDD 004.22

Ficha Catalográfica elaborada pelo
Setor de Processamento Técnico da BC-PUCRS

Av. Ipiranga, 6681 - Prédio 16
CEP 90619-900 - Porto Alegre - RS - Brasil
http://www.pucrs.br/biblioteca

Fone: 0 (xx) 51 3320-3544
Fax: 0 (xx) 51 3320-3548
e-mail: bceadm@pucrs.br

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Reconhecido pelo Parecer No. 930/98.C.N.E. Homologação Publicada no D.O.U. de 30/12/98.

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "***MDX-cc: Ambiente de programação Paralela Aplicado a Cluster de Clusters***", apresentada por **Cassiano Ricardo Hess**, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Área de Processamento Paralelo e Distribuído, aprovada em 28/03/2003 pela Comissão Examinadora:

Prof. Dr. César Augusto Fonticielha De Rose - Orientador — PPGCC/PUCRS

Prof. Dr. Avelino Francisco Zorzo – PPGCC/PUCRS

Prof. Dr. Fernando Luís Dotti– PPGCC/PUCRS

Prof. Dr. Gerson Geraldo Homrich Cavalheiro – UNISINOS

Homologada em 02/03/05, conforme Ata No. 004 pela Comissão Coordenadora.

Prof. Dr. ***Paulo Henrique Lemelle Fernandes***
Coordenador.

AV. IPIRANGA, 6681 - PRÉDIO 16/SALA 106
90619-900 - PORTO ALEGRE -RS -BRASIL
TELEFONE: 55 51 33203611 FAX 55 51 3320.3621
http: //www.inf.pucrs.br e-mail: ppgcc@inf.pucrs.br

Agradecimentos

Inicialmente, agradeço a minha esposa Samantha, que eu amo demais, que teve paciência e me deu a força necessária nesses dois últimos anos. Te amo sempre;

Aos meus pais, que são os exemplos que eu levo sempre comigo, e meus irmãos Estevão e Bia. Todos, mesmo longe, sempre estiveram comigo quando eu precisei;

Aos meus cães Mike, o mais velho, e Zeca, o caçula, que estão sempre dispostos a brincar e buscar a bolinha;

À PUCRS pela oportunidade e pela infra-estrutura a mim oferecidos;

À HP, pelo suporte financeiro que possibilitou este trabalho;

Aos meus orientadores, Prof. Celso M. da Costa, pela oportunidade e pelo acompanhamento no primeiro ano, e Prof. César A. F. De Rose, pela força e pela compreensão, pelos convites para o futebol e pelos momentos de descontração, que melhoram o ambiente de trabalho;

Aos colegas de mestrado, em especial Leonardo Sewald, Luciano Cassol e Eduardo Rödel, pela amizade e atenção nos momentos complicados;

A todos os colegas do CPAD, pela atenção e ajuda;

Aos meus amigos de graduação, Flávio, Leonardo, Sandro, Bento, Mille, etc., que sempre estão prontos para ajudar em qualquer circunstância.

Ao glorioso Grêmio FBPA, que possibilitou tantas flautas (deveria agradecer também ao Inter, eu acho) e continua sempre dando muitas alegrias;

A todos, muito obrigado.

Resumo

Em razão do surgimento de redes de comunicação de alta velocidade, tais como *Myrinet* e SCI, a construção de arquiteturas baseadas em máquinas comuns (PCs e estações de trabalho) conectadas por esse tipo de rede - o que se denomina agregado (ou *cluster*) - tornou-se viável. Tais arquiteturas vêm se consolidando como plataformas alternativas para a execução de aplicações paralelas complexas, principalmente devido à relação custo/benefício que oferecem. Esse avanço das tecnologias de redes possibilita também a agregação de *clusters*, formando uma estrutura de *cluster* de *clusters*, como uma única máquina paralela.

Um dos principais problemas no uso de *cluster* de *clusters* é o *software* utilizado para o desenvolvimento de aplicações paralelas, visto que cada agregado envolvido na estrutura possui certas características que precisam ser tratadas pela linguagem ou ambiente de programação, visando o alcance de alto desempenho.

Esta dissertação tem como objetivo apresentar uma ferramenta de programação paralela por troca de mensagens que executa sobre uma estrutura de *cluster* de *clusters*: o MDX-cc. A ferramenta foi concebida tendo como base o sistema MDX [PRE98][HES01] e uma primeira versão foi implementada oferecendo suporte à comunicação em agregados com redes SCI, *Myrinet* e *Fast-Ethernet*.

O principal objetivo do MDX-cc é oferecer recursos de comunicação e sincronização de processos que rodam em agregados interligados. Por sua arquitetura modular e abstração do uso de protocolos de comunicação dedicados a cada tecnologia de rede, o MDX-cc oferece uma interface de programação simples, com um conjunto reduzido de primitivas, e provê transparência total na comunicação entre processos que executam em nós de *clusters* com tecnologias de rede distintas.

Palavras-chave: programação paralela, *cluster* de *clusters*, avaliação de desempenho, troca de mensagens.

Abstract

Due to the appearance of fast communication networks, such as Myrinet and SCI, it becames possible the construction of new architectures based on commodity, off-the-shelf machines (PCs and workstations) connected by this kind of network - which are been called clusters. Such architectures are becoming an alternative execution platform for complex parallel applications, mainly due to the cost/benefit relation they present. The diversity of fast networks leads to the interconnection of clusters, building an architecture called cluster of clusters.

One of the main problems in the use of clusters of clusters is the programming software used for parallel application development, since this kind of architecture has some characteristics that must be addressed by the programming language or environment in order to provide high performance.

This work describes the development of a message passing parallel programming environment for cluster of clusters, the MDX-cc. This environment was projected based on the MDX system [PRE98][HES01] and a first version has been implemented supporting communication over Fast-Ethernet, SCI and Myrinet networks.

The main goal of MDX-cc is to provide communication and synchronization of processes that run on interconnected clusters. Thanks to its modular architecture and the use of specific communication protocols dedicated to each network interface, MDX-cc provides a simple programming interface, with a small set of primitives, and also provides a transparent communication among processes running on different network-based clusters.

Keywords: parallel programming, cluster of clusters, performance testing, message passing.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviaturas

SISD - *Single Instruction Single Data*

MISD - *Multiple Instruction Single Data*

SIMD - *Single Instruction Multiple Data*

MIMD - *Multiple Instruction Multiple Data*

UMA – *Uniform Memory Access*

NUMA – *Non-Uniform Memory Access*

NORMA – *Non-Remote Memory Access*

PVP – *Parallel Vector Processors*

SMP – *Symetrical Multiprocessors*

MPP – *Massively Parallel Processors*

DSM – *Direct Shared Memory*

NOW – *Network of Workstations*

COW – *Cluster of Workstations*

LAN – *Local Area Network*

SO – *Sistema Operacional*

SCI – *Scalable Coherent Interface*

RCD – *Remote Communication Daemon*

RPC – *Remote Procedure Call*

API – *Application Program Interface*

DMA – *Direct Memory Access*

TCP – *Transmission Control Protocol*

UDP – *User Datagram Protocol*

IP – *Internet Protocol*

MPI – *Message Passing Interface*

PVM – *Parallel Virtual Machine*

Capítulo 1

Introdução

O Processamento de Alto Desempenho é considerado uma ferramenta fundamental para as áreas da ciência e tecnologia. Sua importância estratégica é demonstrada pela quantidade de iniciativas em pesquisa e desenvolvimento nestas áreas, financiadas por governos de todo o mundo. Encontra-se, principalmente nas áreas da ciência como física, meteorologia e química, maior demanda por alto desempenho. O Processamento de Alto Desempenho, por sua vez, depende fundamentalmente de técnicas de processamento paralelo, capazes de prover o desempenho necessário para estas aplicações.

Em função da popularização das redes *Fast-Ethernet* [IEE95a], e do surgimento de redes de comunicação de dados extremamente rápidas, tais como *Myrinet* [BOD95] e SCI [IEE92], plataformas de *hardware* compostas por computadores comuns (PCs e estações de trabalho) conectados por esse tipo de rede começaram a ser utilizadas como alternativa de arquitetura de máquina paralela, surgindo daí a computação baseada em agregados (*cluster computing*) [HWA97] [BUY99].

De uma maneira geral, o principal objetivo da computação baseada em agregados (*clusters*) é o emprego de *hardware* barato e de ambientes de programação dedicados para permitir a execução de aplicações paralelas e distribuídas, oferecendo alto desempenho com um custo bastante reduzido.

Com os avanços das tecnologias de redes, há ainda a possibilidade da interligação de agregados, formando uma estrutura de *cluster* de *clusters*, que cooperam na resolução de um problema. A interligação de *clusters*, por meio de uma rede local ou via internet, visa principalmente alcançar o máximo desempenho na execução de aplicações paralelas. Uma vez que o número de máquinas aumenta, o poder computacional alcançado também é maior. Desta forma, aplicações complexas, que necessitam de um grande potencial de recursos, podem ser distribuídas nos agregados envolvidos na estrutura, de forma a conseguir uma melhora de desempenho.

Atualmente, o principal fator limitante no uso de *cluster* de *clusters* como plataforma de alto desempenho é o *software* utilizado para o desenvolvimento de aplicações. Os ambientes mais utilizados até então são úteis à utilização de um único agregado, com determinadas características (rede, SO, etc.). Os agregados, na sua maioria, são homogêneos, o que facilita a sua utilização. Surge a necessidade de ambientes que suportem a execução de programas paralelos em agregados interligados, estrutura que pode ser vista como uma única máquina paralela heterogênea.

Dentro desse contexto, este trabalho vem apresentar um ambiente de programação paralela aplicado a *cluster* de *clusters*, o MDX-cc. Esse ambiente tem como objetivo facilitar a programação e a execução de aplicações paralelas em agregados interligados, gerenciando a comunicação entre processos, independentemente do *cluster* onde os mesmos se encontram executando. Este trabalho não se preocupa com problemas avançados resultantes da heterogeneidade de um *cluster* de *clusters*, como balanceamento de carga e utilização de mais de um modelo de programação paralela (troca de mensagens e memória compartilhada) em uma mesma aplicação. Apenas se preocupa com a implementação de uma ferramenta que ofereça infra-estrutura para a sua utilização.

Este documento está dividido da seguinte maneira. O segundo capítulo apresenta aspectos gerais de programação paralela, como arquiteturas de máquinas paralelas utilizadas atualmente, modelos de programação paralela e bibliotecas desenvolvidas. O Capítulo 3 apresenta a estrutura de *cluster* de *clusters*, à qual é aplicado o ambiente MDX-cc. No Capítulo 4 são apresentadas ferramentas semelhantes àquela proposta nesse trabalho. O Capítulo 5 apresenta o ambiente MDX-cc, com detalhes de projeto e de implementação, e o Capítulo 6 mostra detalhes do protocolo de comunicação implementado no ambiente. Os testes de desempenho realizados com o MDX-cc são demonstrados no Capítulo 7 e, por fim, o

Capítulo 8 traz as conclusões sobre o trabalho, as principais dificuldades encontradas e sugestões de trabalhos futuros relacionados a este.

Capítulo 2

Programação Paralela

Este capítulo apresenta diferentes aspectos ligados à programação paralela, desde aspectos de *hardware*, como arquiteturas de máquinas usadas atualmente, até aspectos ligados diretamente à programação, como formas de paralelização de programas e modelos de programação paralela.

2.1. Arquiteturas de Máquinas Paralelas

Máquinas paralelas vêm se tornando mais populares em função da demanda sempre crescente por poder computacional. Infelizmente, os sistemas que oferecem a capacidade de processamento para satisfazer esta demanda, muitas vezes, ainda têm custo elevado, são difíceis de programar, ou ambos.

Como a programação de aplicações paralelas ainda exige o conhecimento de características específicas da máquina para a obtenção de desempenho, conhecimentos sólidos sobre arquiteturas de máquinas paralelas auxiliam no desenvolvimento de programas paralelos, desde sua modelagem até a fase de depuração e otimização.

2.1.1. Classificação de Arquiteturas Paralelas

Diversas taxonomias para arquiteturas paralelas foram propostas. Nessa seção serão abordadas a classificação genérica de Flynn e a classificação segundo o compartilhamento de memória.

2.1.1.1. Classificação de Flynn

Para uma classificação inicial de máquinas paralelas, pode ser usada a classificação genérica de Flynn. Baseando-se no fato de uma máquina executar uma seqüência de instruções sobre uma seqüência de dados, diferencia-se o fluxo de instruções (*instruction stream*) e o fluxo de dados (*data stream*). Dependendo da multiplicidade desses fluxos, e da combinação das possibilidades, são propostas quatro classes de máquinas: SISD, MISD, SIMD e MIMD [FLY72].

Em uma máquina da classe MIMD (*Multiple Instruction Multiple Data*), cada unidade de controle C recebe um fluxo de instruções próprio e o repassa para sua unidade de processamento P, para que seja executado sobre um fluxo de dados próprio (Figura 2.1). Dessa forma, cada processador executa o seu próprio programa sobre seus próprios dados de forma assíncrona. Sendo assim, o princípio MIMD é bastante genérico, pois qualquer grupo de máquinas, se analisado como uma unidade, pode ser considerado da classe MIMD. Para que o processamento nas diferentes posições de memória possa ocorrer em paralelo, a unidade M não pode ser implementada como um único módulo de memória. Nessa classe, enquadram-se servidores com múltiplos processadores (*dual*, *quad*), redes de estações e máquinas como CM-5 [HWA93], nCUBE [CUL99], Intel Paragon [CUL99] e Cray T3D [CUL99].

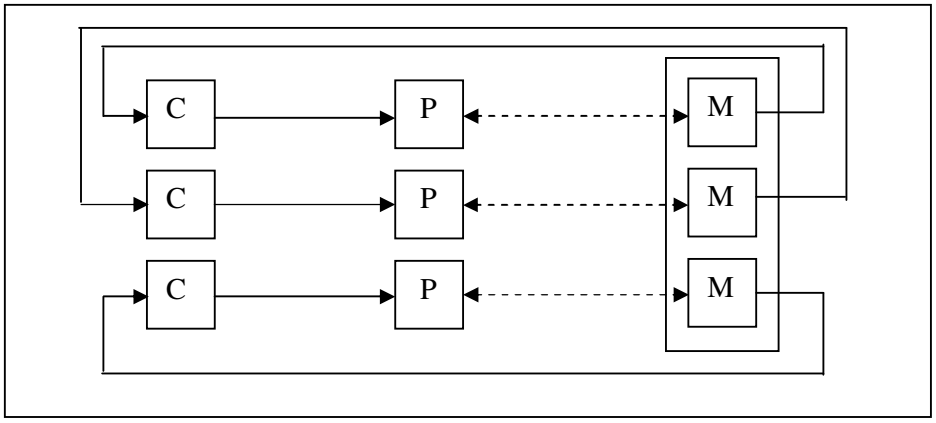

Figura 2.1: Diagrama da classe MIMD.

A maioria das máquinas paralelas atuais, incluindo os agregados abordados neste trabalho, se enquadram na classe MIMD [DER02].

2.1.1.2. Classificação segundo o compartilhamento de memória [DER02]

Um outro critério para a classificação de máquinas paralelas é o compartilhamento de memória. Quando se fala em memória compartilhada (*shared memory*), existe um único espaço de endereçamento que será usado de forma implícita para comunicação entre processadores, com operações *load* e *store*. Quando a memória não é compartilhada, existem múltiplos espaços de endereçamento privados (*multiple private address space*), um para cada processador. Isso implica na comunicação explícita através de troca de mensagens com operações *send* e *receive*.

Memória distribuída (*distributed memory*), por sua vez, refere-se a localização física da memória. Se a memória é implementada com vários módulos, e cada módulo é colocado próximo a um processador, então a memória é considerada distribuída. Outra alternativa é o uso de memória centralizada (*centralized memory*), ou seja, a memória encontra-se à mesma distância de todos os processadores, independentemente de ter sido implementada com um ou vários módulos [PAT94]. De acordo com o uso ou não de uma memória compartilhada, pode-se diferenciar máquinas paralelas da seguinte forma:

- Multiprocessadores: todos os processadores acessam, através de uma rede de interconexão, uma memória compartilhada. Existe apenas um espaço de endereçamento e os processos comunicam através da memória compartilhada. Em relação ao tipo de acesso à memória, multiprocessadores podem ser classificados como [HWA93]:

    - Acesso uniforme à memória (*uniform memory access*, UMA): memória centralizada e localizada à mesma distância de todos os processadores.
    - Acesso não-uniforme à memória (*non-uniform memory access*, NUMA): memória distribuída, implementada com módulos associados a diferentes processadores (distâncias diferentes).

- Multicomputadores: cada processador P possui uma memória local M, à qual só ele tem acesso. A comunicação entre processos é feita através de troca de mensagens pela

ede de interconexão. Em relação ao tipo de acesso à memória do sistema, multicomputadores podem ser classificados como:

- Sem acesso à memória remota (*non-remote memory access*, NORMA): cada processador só consegue endereçar sua memória local.

2.1.2. Tendências na Construção de Máquinas Paralelas [DER02]

Nesta seção serão apresentados os principais modelos de máquinas paralelas, que constituem atualmente as principais tendências para construção desses sistemas.

2.1.2.1. Multiprocessadores Simétricos

Multiprocessadores simétricos (SMP) são sistemas constituídos de processadores comerciais, também denominados "de prateleira", conectados a uma memória compartilhada (caracterizando essas máquinas como UMA) através de um barramento (Figura 2.2). A comunicação entre processos se dá por memória compartilhada.

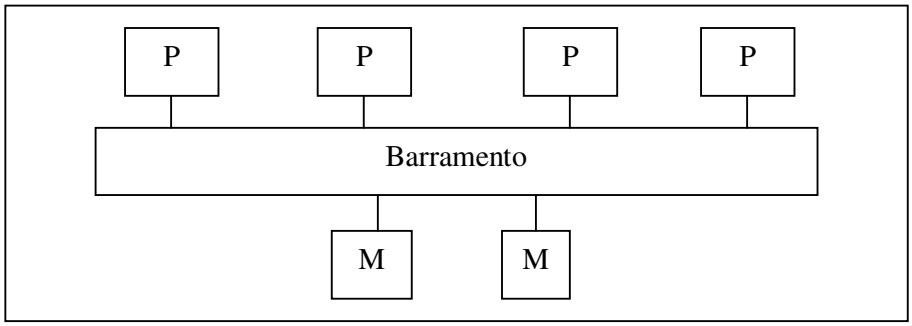

Figura 2.2: Arquitetura de um SMP.

A escalabilidade de uma máquina SMP é comprometida pelo fato da ligação com a memória ser através de um barramento. Como o barramento só permite uma transação por vez, o desempenho da máquina cai a medida que o canal é disputado por um maior número de processadores.

São exemplos de SMP o IBM R50, SGI Power Challenge e o HP/Convex Exemplar X-Class.

2.1.2.2. Máquinas com Memória Compartilhada Distribuída

Máquinas com memória compartilhada distribuída (DSM) são sistemas em que, apesar da memória encontrar-se fisicamente distribuída, todos os processadores podem endereçar todas as memórias. Essa implementação pode ser feita em *hardware*, *software* ou, ainda, a combinação dos dois. A distribuição da memória pode ser resultado da escolha entre uma arquitetura multiprocessada com memória distribuída (máquina NUMA) ou uma arquitetura de multicomputador com memória distribuída (máquina NORMA).

São exemplos de DSM o Stanford DASH, o Cray T3D e estações de trabalho rodando TreadMarks.

2.1.2.3. Redes de Estações de Trabalho

Redes de estações de trabalho (NOW) são sistemas constituídos por várias estações de trabalho interligadas por uma rede tradicional, como *Ethernet* [SOA95] e ATM [HAN98]. Na prática, uma rede local pode ser vista como uma máquina paralela em que vários processadores, com suas memórias locais, são interligadas por uma rede, constituindo uma máquina paralela NORMA de baixo custo.

Uma vez interligadas por redes como *Ethernet* e ATM, uma máquina NOW possui baixo desempenho na troca de mensagens, pois essas redes utilizam protocolos pesados na comunicação (TCP/IP), resultando em uma latência[1] muito grande.

2.1.2.4. Máquinas Agregadas

Com as novas tecnologias de redes locais de alta velocidade, o uso de redes de estações de trabalho como máquinas paralelas está se tornando cada vez mais atrativo. Neste contexto, tem se investido em uma nova classe de máquinas paralelas, que são as máquinas agregadas (COW – *cluster of workstations*).

Máquinas agregadas, como as NOW, são constituídas por várias estações de trabalho interligadas. A diferença é que foram projetadas especialmente para a execução de aplicações paralelas [DER02]. Desta forma, as estações podem ser otimizadas (tanto em *hardware* como em *software*) para este fim. Na maioria dos casos, as máquinas que servem de nó de uma COW não possuem monitor, teclado e mouse, e seu sistema operacional presta apenas serviços que serão úteis ao processamento paralelo de sistemas.

---

[1] Latência: tempo necessário para o envio de uma mensagem de tamanho zero [CUL99].

Encontra-se, atualmente, duas tendências na construção de máquinas agregadas [DER02]:

- Agregados interligados por redes padrão: esta tendência é impulsionada pelos grandes fabricantes como IBM e HP, que estão interessados na construção de máquinas paralelas poderosas agregando milhares de estações de trabalho de baixo custo (*low end*). Para a interligação de tantas máquinas, investir em uma rede especial aumenta muito o custo. Assim, utiliza-se para este fim uma rede padrão como a *Fast-Ethernet*. Para melhorar a latência da rede são usados grandes chaveadores (*switches*), em contraste aos *hubs*, que funcionam como grandes barramentos, solução mais comum em redes locais por causa do menor custo. O enfoque reside na obtenção de desempenho com muitos nós de pequeno poder computacional e, preferencialmente, com aplicações que não tenham muita necessidade de comunicação. Estas máquinas são claramente NORMA, já que não é possível o acesso a memória de nós remotos e o paradigma de comunicação utilizado é a troca de mensagens.
- Agregados interligados por redes de baixa latência: esta tendência é impulsionada por pequenas empresas que fabricam placas de interconexão especificamente para máquinas agregadas. Estas placas implementam protocolos de rede de baixa latência otimizados para as características de comunicação de aplicações paralelas. Como o custo destas placas é mais alto do que placas de rede padrão, é muito oneroso construir máquinas com muitos nós (centenas de nós). Para compensar o menor número de nós, são usados nós mais poderosos, muitas vezes servidores de médio porte com vários processadores (SMP). A máquina resultante fica mais equilibrada na relação poder de processamento do nó e desempenho da rede, obtendo um bom desempenho, mesmo com aplicações que necessitem de muita comunicação. A maioria destas máquinas é NORMA já que, em princípio, não é possível o acesso a memória de nós remotos. Porém, algumas das placas implementam um espaço de endereçamento global entre os nós permitindo o acesso de memórias remotas. Neste caso a máquina resultante é NUMA e são suportados tanto a troca de mensagens como o acesso a áreas de memória compartilhada.

A utilização de máquinas agregadas traz algumas vantagens como:

- Ótima relação custo-benefício: as máquinas agregadas, por serem construídas com componentes "de prateleira", que, por sua vez, são produzidos em grande escala, têm custo reduzido. Quanto ao desempenho, este é aproximado ao de um sistema MPP [MAI98] que, por exemplo, pode custar até duas vezes mais, além de possuir um dos menores custos por MFlops/s (milhões de instruções de ponto flutuante por segundo) em comparação às outras arquiteturas paralelas.
- Baixo custo de manutenção: a arquitetura possui um baixo custo de manutenção, pois os "componentes de prateleira" utilizados são facilmente encontrados no mercado, podendo ser feita a substituição de peças, igualmente, com um baixo custo. Devido a esta ótima relação custo-benefício, fabricantes de MPPs estão se utilizando de alguns conceitos da construção de agregados para diminuir o custo de seus produtos. Sendo assim, as diferenças entre estas máquinas diminuíram bastante ao ponto de ficar difícil sua classificação.
- Escalabilidade: com a utilização destas máquinas, tem-se a capacidade de expansão mantendo as mesmas características com configurações de baixo custo e que podem crescer na medida das necessidades, com a inclusão de nós adicionais. A arquitetura possui, também, um alto grau de configurabilidade, por se tratar de uma arquitetura aberta. A definição de seus componentes (nós e rede de interconexão) é generalizada, possibilitando diversas configurações.

Com relação à interconexão entre os nós deste tipo de máquina, existem vários padrões, que serão apresentados a seguir.

- *Fast-Ethernet* [TAN97] [STE99]: *Fast-Ethernet* é uma extensão da *Ethernet*, utilizada na maioria das LANs (*Local Area Networks*). Possui uma latência bem menor na comunicação e um aumento significativo na largura de banda. Utiliza as mesmas tecnologias do padrão *Ethernet*, permitindo uma implementação e interoperabilidade mais fáceis. A *switch Fast-Ethernet* possui uma vazão na ordem de 100Mbits/s e, pelo fato de ser uma placa convencional, a implementação das camadas de rede deve ser feita em *software*, o que compromete de forma significativa a latência. Nas outras tecnologias isto não se verifica, pois a interconexão das camadas é implementada em *hardware*. Além de ser utilizada em *clusters* com um grande número de nós, *Fast-*

Ethernet também é utilizada em agregados interligados por rede de baixa latência, como rede secundária para avaliação e monitoração da máquina.

- *ParaStation* [MAI98]: consiste em uma emulação de *sockets* UNIX e de ambientes amplamente utilizados para programação paralela, como PVM [GEI96]. Isto permite portar uma grande quantidade de aplicações paralelas e cliente/servidor para a *ParaStation*. Algumas implementações iniciais atingem uma latência em torno de 2 microssegundos e uma largura de banda de 1,5 Gbits/s por canal de comunicação. Uma rede *ParaStation* utiliza uma topologia baseada em uma malha de duas dimensões, mas para sistemas pequenos uma topologia em anel é suficiente. Neste padrão, a rede é dedicada a aplicações paralelas, não substituindo LANs comuns, de modo que os protocolos das LANs comuns podem ser eliminados. Isto permite utilizar propriedades mais especializadas na rede, como protocolos ponto-a-ponto e controle da rede em nível de usuário, sem interação com o SO. O protocolo *ParaStation* implementa múltiplos canais lógicos de comunicação em uma ligação física. Em contraste com outras redes de alta velocidade, não há custo adicional para componentes de *switch* central. *ParaStation* é utilizada na construção de agregados interligados por rede de baixa latência.
- *Myrinet* [SEI95]: *Myrinet* é uma tecnologia de chaveamento e comunicação de pacotes de alto desempenho (latência de cerca de 5 microssegundos) de custo relativamente baixo, que está sendo amplamente utilizada para interconectar máquinas baseadas em agregados. É um padrão que utiliza uma tecnologia baseada em comunicação através de pacotes. Desenvolvida pela *Myricom* [MYR00], pode ser utilizada tanto em máquinas agregadas de SANs (*Small Area Networks*) como em máquinas agregadas de LANs. Em comum com as LANs, os nós de uma máquina baseada em agregados que utilizam uma rede *Myrinet*, enviam e recebem os dados na forma de pacotes. Qualquer nó pode enviar um pacote para qualquer outro nó, porém, em contraste com as LANs comuns, uma rede *Myrinet* possui altas taxas de transferência. Uma ligação *Myrinet* é composta por um par de canais *full-duplex* que permite uma taxa de transferência em torno de 2 Gbit/s cada um. As características que tornam a *Myrinet* uma rede de alto desempenho incluem o desenvolvimento de canais robustos de comunicação com controle de fluxo, pacotes e controle de erro, baixa latência, interfaces que podem mapear a rede, rotas selecionadas, tradução de endereços da rede para estas rotas, bem como manipulação do tráfego de pacotes e *software* que permite comunicação direta

entre os processos em nível de usuário e rede. Uma rede *Myrinet* utiliza normalmente topologias regulares, tipicamente malhas de duas dimensões, embora permita a utilização de uma topologia arbitrária, uma vez que um cabo *Myrinet* pode conectar *hosts* entre si, ligar cada placa a um *switch* ou, ainda, dois *switches* entre si. Ao contrário de uma LAN típica, em que o tráfego de pacotes compartilha um mesmo canal físico, uma rede *Myrinet* com uma malha bidimensional pode ser considerada escalável, pois a capacidade dos agregados cresce com o número de nós devido ao fato de que muitos pacotes podem trafegar de forma concorrente por diferentes caminhos da rede. Uma rede *Myrinet* tem *switches* de múltiplas portas, que podem ser conectados por ligações para outros *switches* e para outros *hosts* em topologias variadas. *Myrinet* é utilizada na construção de agregados interligados por rede de baixa latência.

- SCI (Scalable Coherent Interface) [BUT98] [BUT99] [STE99] [HEL98]: o padrão SCI foi originalmente designado para proporcionar a interconexão entre sistemas de memória compartilhada com coerência de *cache* [STE99]. Especifica um *hardware* inovador e protocolo para conexão de até 64k nos nós, em uma rede de alta velocidade com características de comunicação de alto desempenho. A característica de coerência de *cache* é implementada com uma lista de módulos SCI. Cada nó é acoplado a um módulo. Quando um processador atualiza seu *cache*, o estado deste é propagado por todos os outros módulos, compartilhando a mesma *cache*. Esta lista para *caches* coerentes distribuídos é escalável devido a um sistema robusto poder ser criado pela inserção de mais módulos SCI na lista. Proporciona uma latência de menos de 1 microssegundo e alta largura de banda (de 250 Mbits/s a mais de 8 Gbits/s) em uma conexão ponto-a-ponto, o que também evita a limitação física dos barramentos, não havendo, deste modo, maiores dificuldades com relação à escalabilidade. O SCI define serviços de barramento, oferecendo soluções distribuídas para a sua realização. Esses serviços proporcionam um espaço de endereçamento físico de 64 bits entre os nós, o que permite transações de escrita, leitura e a criação de áreas de memória compartilhada entre estes. Dos 64 bits, 16 bits são utilizados para endereçar os 64K nós possíveis (2^{16} = 64K) e os 48 bits restantes são utilizados para o endereçamento em cada nó. A placa SCI permite construir máquinas com características NUMA, uma vez que permite acessos à memória remota, ou seja, um nó pode acessar a memória local de outro nó, realizados pelo *hardware*. No entanto, estes acessos são mais lentos

que os acessos locais, o que caracteriza acessos não uniformes à memória. A construção básica de blocos SCI é feita por pequenos anéis. Sistemas maiores podem ser obtidos através da criação de anéis de anéis, interconectados via *switches* SCI. Desta forma, além de permitir a troca de mensagens utilizando um *hardware* especial, o SCI ainda possui a capacidade de implementar, via *hardware*, uma memória compartilhada distribuída (DSM), através de operações de escrita e leitura em regiões de memória mapeadas em memórias remotas. Isto resulta em uma baixa latência, num ambiente baseado em agregados.

Atualmente, a construção de máquinas agregadas tornou-se uma tendência, já existindo algumas com centenas de nós, como, por exemplo, a máquina *iCluster*, com 225 nós e pico de 165.9 GFlops, instalada em 2001 no laboratório ID-IMAG/INRIA, em Rhone-Alpes, Grenoble, França[TOP01].

2.2. Modelos de Programação Paralela

Na programação paralela, existe a necessidade de comunicação entre os processos, pois os mesmos cooperam na resolução de um problema. A comunicação entre processos pode ser efetuada através de memória compartilhada ou por troca de mensagens.

Modelos de programação baseados em memória compartilhada permitem implementações com menor complexidade em relação aos modelos com troca de mensagens. Entretanto, leituras e escritas a um mesmo dado compartilhado não podem ser feitas simultaneamente, exigindo a utilização de uma seção crítica envolvendo o acesso a memória compartilhada.

2.2.1. Programação Baseada em Memória Compartilhada

O modelo que utiliza memória compartilhada distribuída apresenta uma menor complexidade para o programador, se comparado ao modelo por troca de mensagens, devido à semelhança com um programa seqüencial. O programador não tem nenhuma preocupação com a comunicação e com o gerenciamento do particionamento do conjunto de dados, podendo focar toda sua atenção única e exclusivamente no desenvolvimento dos algoritmos.

Três abordagens têm sido utilizadas na implementação de sistemas com memória compartilhada:

- Implementação por *hardware*: estendem técnicas tradicionais de *caching* para arquiteturas escaláveis;
- Implementação de bibliotecas e pelo SO: o compartilhamento e a coerência de memória são obtidos através de mecanismos de gerenciamento de memória virtual;
- Implementação pelo compilador e bibliotecas: acessos compartilhados são automaticamente convertidos em primitivas de coerência e sincronização.

Os sistemas com memória compartilhada, como o IVY [LI86] e TreadMarks [KEL94], provêem uma abstração de memória compartilhada em redes de estações de trabalho. A facilidade que esses sistemas de memória compartilhada proporcionam para o programador tem um custo que é, normalmente, um desempenho inferior aos dos sistemas com troca de mensagens. Outro problema enfrentado por este modelo é a baixa escalabilidade, pois fica difícil conseguir um bom desempenho à medida que aumenta o número de processadores do sistema [PRE98].

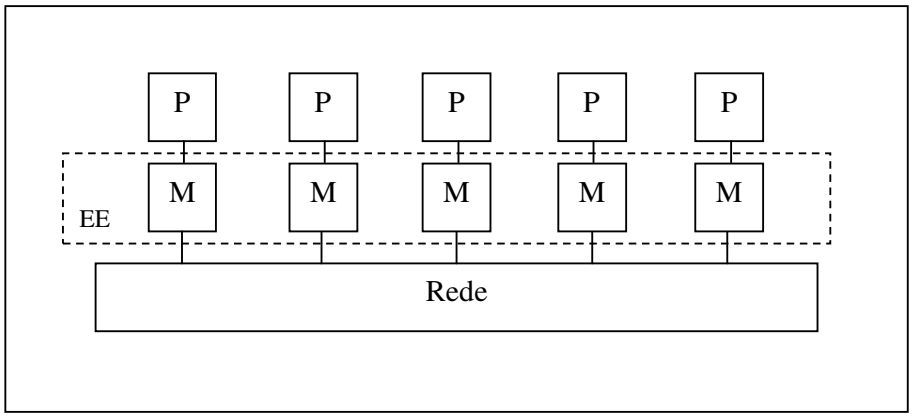

Figura 2.3: Sistema DSM com espaço de endereçamento único (EE).

A Figura 2.3 mostra um sistema de memória compartilhada distribuída, onde cada processador acessa uma memória compartilhada que pode ser local ou distante (NUMA).

2.2.2. Programação Baseada em Troca de Mensagens

No modelo de programação paralela baseado em troca de mensagens, os processadores encontram-se distribuídos fisicamente sobre uma rede e podem acessar somente sua memória

local (NORMA). Através da rede, os processos podem enviar e receber mensagens, de acordo com a Figura 2.4. que ilustra o envio de uma mensagem de um processo a outro.

A comunicação por troca de mensagens envolve, pelo menos, dois processos: o transmissor e o receptor, sendo que pode haver vários receptores e transmissores. A comunicação se dá pelas primitivas básicas *send* e *receive*, através das quais mensagens são enviadas e recebidas, respectivamente.

Atualmente, este modelo de programação é o mais usado para programação paralela sobre uma rede de estações de trabalho [PRE98]. Diversas bibliotecas foram desenvolvidas, como PVM [GEI96], MPI [MES94].

Com a troca de mensagens, a falta de uma memória compartilhada é completamente exposta para o programador, que precisa decidir e conhecer a localização dos processos e dos dados, bem como efetuar explicitamente a comunicação entre os mesmos, enviando e recebendo mensagens e sincronizando cada um dos processos. Isto torna o modelo de programação bastante complexo.

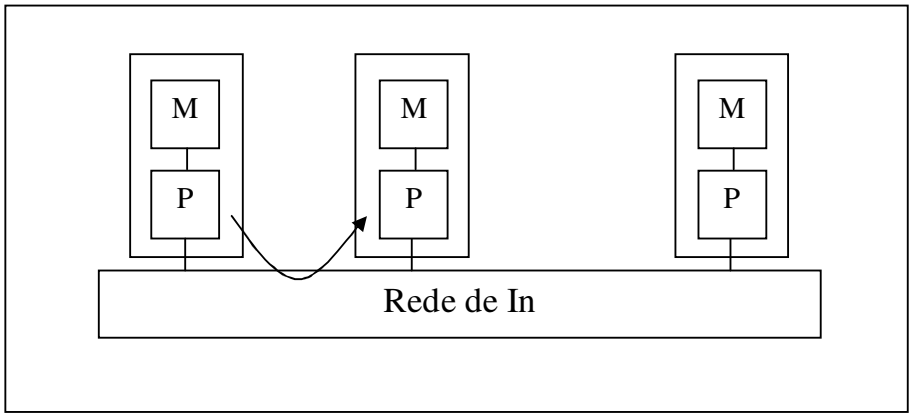

Figura 2.4: Sistema baseado em troca de mensagens.

2.3. Formas de Paralelização de Programas

Dada a diversidade de arquiteturas paralelas, foram propostas diversas linguagens, compiladores e bibliotecas visando gerenciar programas paralelos. O paralelismo em um programa pode ser explorado implicitamente através de compiladores paralelizantes, como o Suif [AMA95][HAL96], que paralelizam automaticamente programas escritos em C, ou então explicitamente, através de primitivas inseridas no programa, ou, ainda, através de uma linguagem de programação, como C Concorrente [GEH86][GEH92] ou Orca [BAL92].

A experiência tem mostrado que a programação paralela é significativamente mais difícil que a programação seqüencial, principalmente porque envolve a necessidade de sincronização entre tarefas e a análise de dependência de dados. A utilização de um sistema paralelizante minimiza essas dificuldades, e permite também o reaproveitamento de programas seqüenciais já implementados. Por outro lado, o paralelismo explícito permite que fontes não passíveis de detecção por um sistema paralelizante possam ser exploradas.

O paralelismo expresso pelo usuário pode ser especificado em um programa através de comandos específicos de uma linguagem de programação paralela, por chamadas a rotinas de uma biblioteca que gerencia os recursos de paralelismo da máquina ou pelo uso de diretivas do compilador.

Nos agregados, por causa da necessidade de troca de mensagens, as bibliotecas específicas têm sido uma forma muito utilizada de implementar programas paralelos. São exemplos de bibliotecas PVM – [GEI96], MPI [MES94], TreadMarks [KEL94] e MDX [PRE98].

2.4. Bibliotecas de Programação Paralela

Neste capítulo serão apresentadas duas bibliotecas de programação paralela por troca de mensagens, sendo uma para comunicação sobre redes *Myrinet* e outra para comunicação sobre redes SCI. Bibliotecas de memória compartilhada não serão apresentadas por não fazerem parte do escopo do trabalho.

2.4.1. GM

GM é uma API de comunicação para redes *Myrinet*. Tem como características o baixo *overhead* na troca de mensagens, baixa latência e alta vazão [GM01]. Adicionalmente, possui outras características:

- acesso direto à interface de rede, sem intervenção do SO
- suporta *clusters* de mais de 10000 nós
- permite a troca de mensagens de até 2^{31}-1 bytes
- as mensagens devem estar localizadas em áreas de memória DMA
- faz o mapeamento automático de redes *Myrinet*

A comunicação em GM é feita através de estruturas chamadas “portas”. Este modelo de comunicação determina que não seja necessária a conexão entre clientes que se comunicam. Basta que um cliente monte a mensagem e a envie, através de uma porta, para qualquer outra porta na rede. O cliente destino então consome a mensagem da porta à qual foi enviada a mesma (Figura 2.5).

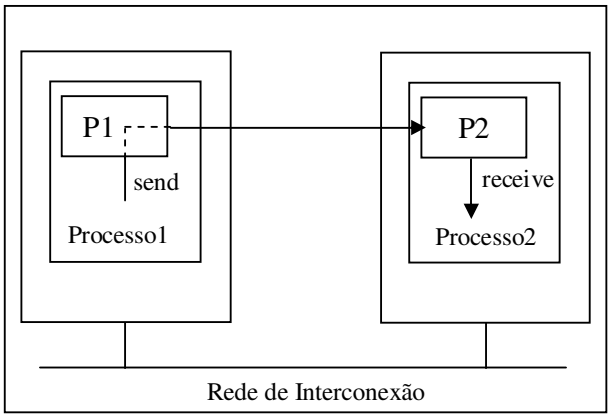

Figura 2.5: Comunicação entre processos GM.

O envio da mensagem é regulado por *tokens*, de maneira semelhante ao FM [PAK97]. O emissor só pode enviar mensagens para uma porta quando possuir um *token* para aquela porta. Durante o envio de uma mensagem, o emissor libera o *token* e o recebe de volta somente quando a mensagem é corretamente entregue. Analogamente, o recebimento de mensagens também é regulado por *tokens*; porém, é possível ao receptor especificar diversas primitivas de recebimento, de acordo com o tamanho e a prioridade das mensagens que deseja receber. Essa característica fornece bastante flexibilidade, pois somente serão recebidas as mensagens que satisfizerem o tamanho e a prioridade especificadas em cada *receive( )*. E importante salientar que a prevenção de *deadlocks* é tarefa do programador: deve haver sempre um *token* disponível para cada tamanho e prioridade de mensagem que se deseja receber em uma determinada porta.

A ordenação é garantida somente para mensagens de mesma prioridade, vindas do mesmo emissor e enviadas ao mesmo receptor. Não existe dependência entre mensagens de prioridades diferentes, o que permite que mensagens com prioridades mais baixas possam ser processadas antes de mensagens com prioridades mais altas.

As principais primitivas da API GM são:

gm_init( )

Inicia o ambiente GM. Deve ser a primeira função chamada em uma aplicação GM.

gm_open(struct gm_port **port, int device, int port, char *port_name)

Abre a porta de comunicação número *port* da placa de rede identificada por *device*. As informações da porta são retornadas em **port*. *port_name* é uma *string* que serve para identificar a porta. Serve unicamente para depuração de programas.

gm_host_name_to_node_id(struct gm_port *port, char *host_name)

Retorna o identificador GM associado ao nó *host_name*.

gm_dma_malloc(struct gm_port *port, unsigned int length)

Aloca *length* bytes de memória DMA.

gm_bcopy(void *from, void *to, unsigned long len)

Copia *len* bytes de *from* para *to*.

gm_send_with_callback(struct gm_port *port, void *msg, unsigned int size, unsigned int len, unsigned int priority, unsigned int target_node_id, unsigned int target_port_id, gm_send_completion_callback_t callback, void *context)

Envia *msg*, de tamanho *len*, com prioridade *priority*, para a porta *target_port_id* do nó *target_node_id*. Assim que o processo de envio é completado, a função *callback(port, context, status)* é chamada, com *status* indicando que a operação de envio foi executada com sucesso.

gm_provide_receive_buffer(struct gm_port *port, void *msg, unsigned int size, unsigned int priority)

Permite ao GM receber uma mensagem de tamanho *size*, com prioridade *priority*, na variável *msg*.

gm_recv_event gm_receive(struct gm_port *port)

Retorna um evento de recebimento de mensagem em *port*. Caso nenhum evento esteja pendente, GM_NO_RECV_EVENT é retornado. Caso haja uma mensagem a ser recebida, esta é armazenada no *buffer* associado a *port* em *gm_provide_receive_buffer(...)*.

gm_dma_free(struct gm_port *port, void *ptr)

Libera memória apontada por *ptr*, alocada por *gm_dma_malloc(...)*.

gm_close(struct gm_port *port)

Fecha porta de comunicação.

gm_finalize( )

Finaliza o ambiente GM. Deve ser a última função chamada em uma aplicação GM.

2.4.2. MPI

MPI consiste em uma interface padrão para troca de mensagens (comunicação direta) definida por um comitê que reuniu várias instituições de ensino, pesquisa e fabricantes de máquinas paralelas.

MPI é um sistema complexo, pois sua completa especificação abrange 129 funções. Muitas destas possuem numerosos parâmetros e variáveis. Entretanto, a maioria das aplicações pode ser programada com apenas seis destas funções [FOS95].

Um programa MPI, compreende um ou mais processos que comunicam através de funções da biblioteca MPI para enviar e receber mensagens. Em MPI, um número fixo de processos é criado durante a fase de inicialização. Os processos podem executar diferentes programas. Por esta razão, o modelo de programação MPI é algumas vezes referenciado como um *Multiple Program Multiple Data* (MPMD), diferentemente do modelo *Single Program Multiple Data* (SPMD) em que cada processador executa um trecho de código de um mesmo programa.

A comunicação entre processos pode ocorrer de duas formas: comunicação ponto-a-ponto (envolve apenas o emissor e o receptor da mensagem) ou comunicação coletiva (envolve um grupo de processos).

Como citado anteriormente, MPI contém um número muito grande de funções. Contudo, pode-se resolver um grande número de problemas usando-se somente seis funções

básicas. Além das seis funções básicas, será apresentada outra, usada para sincronização dos processos:

MPI_Init (int *argc, char ***argv)

É a primeira chamada que deve ser efetuada em um aplicação. Dispara a inicialização do ambiente MPI. Os parâmetros *argc* e *argv* são requeridos somente na linguagem C, e utiliza os argumentos do programa *main*.

MPI_Finalize ( )

Essa função é usada para terminar uma aplicação.

MPI_Comm_size (MPI_Comm comm, int *size)

Esta função retorna o número total de processos no grupo do comunicador. Comunicador é um conjunto de processos que podem trocar mensagens entre si. Quando o comunicador usado for o comunicador global pré-definido MPI_COMM_WORLD, então esta função indica o número total de processos envolvidos no programa. *comm* identifica o comunicador, *size* é o número de processos no grupo do comunicador.

MPI_Comm_rank (MPI_Comm comm, int *rank)

Esta função retorna o identificador, isto é, o número de criação do processo atual no comunicador. Cada comunicador contém um grupo de processos, que são identificados por um *rank* que começa em 0. *comm* é o comunicador, *rank* é o identificador do processo atual no grupo do comunicador.

MPI_Send (void *buf, int count, MPI_DataType datatype, int dest, int tag, MPI_Comm comm)

Função usada para enviar uma mensagem. *buf* é o endereço inicial do *send buffer*, *count* é o número de elementos do *send buffer, datatype* é o tipo dos dados do *send buffer*; *dest* é o identificador do processo destino. *tag* é a tag da mensagem. Este parâmetro possibilita que o usuário indique o que será feito com os dados recebidos/enviados, pois os mesmos poderão ser usados para diferentes aplicações. *comm* é o comunicador.

MPI_Recv (void *buf, int count, MPI_DataType datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Utilizada por um processo para receber uma mensagem. Se não existir uma mensagem, o processo fica bloqueado. *buf* é o endereço inicial do *receive buffer*, *count* é o número de elementos do *receive buffer*, *datatype* é o tipo dos dados do *receive buffer*; *source* é o identificador do processo origem. *tag* é a tag da mensagem, *comm* é o comunicador, e *status* é o *status* do objeto.

MPI_Barrier (MPI_Comm comm)

Esta função tem como objetivo bloquear o processo que a chama até que o número (quorum) de processos participantes desta barreira seja alcançado. Ou seja, os processos que a chamarem ficarão bloqueados até que o último processo execute esta operação. Quando isso ocorrer, todos os processos envolvidos na barreira serão desbloqueados e continuarão nas execuções. O parâmetro *comm* identifica o comunicador.

A aplicação apresentada pelo Algoritmo 2.1 tem como objetivo enviar uma mensagem de teste do processo com identificador 0 para o processo com identificador 1. Inicialmente, inicia-se a aplicação em MPI com a primitiva *MPI_Init*. Esta retorna se a operação ocorreu ou não com sucesso. Em seguida, recupera-se o número total de processos envolvidos no comunicador MPI_COMM_WORLD através da primitiva *MPI_Comm_size*. Com a primitiva *MPI_Comm_rank*, recupera-se o número do identificador do processo. Caso o identificador seja 0, é enviada uma mensagem, cujo conteúdo é "Mensagem de teste" e está armazenada em *buf*, através da primitiva *MPI_Send( )*. Caso o identificador do processo seja 1, este recebe uma mensagem e armazena na variável *buf*. Em seguida, imprime o seu identificador e a mensagem recebida. Por último, finaliza-se a aplicação em MPI e termina-se o programa.

2.4.3. YAMPI

O YAMPI (*Yet Another Message Passing Interface*) [NOV01] consiste em um *sub-set* de comandos do MPI, que é a API de troca de mensagens mais utilizada, para uma plataforma Linux com interface de interconexão *Dolphin* SCI. A proposta de API feita neste trabalho procura satisfazer os dois pontos principais deste cenário: uma API de troca de mensagens sobre um ambiente de memória compartilhada e com desempenho próximo ao uso de transações nativas.

Esta API foi concebida especificamente para o ambiente *cluster* SCI para ser utilizada por desenvolvedores de *middleware*. Ela leva em conta as vantagens e limitações da interface SCI, com opções restritas e, portanto, com menor complexidade.

O YAMPI utiliza uma variação do *Active Messages* [EIC92] quanto ao tratamento das mensagens curtas. Estas são endereçadas a regiões de memória do nó destino sem que o mesmo esteja em modo de recepção.

O uso de uma interface de troca de mensagens sobre SCI tem por objetivo oferecer a boa performance de comunicação encontrada no SCI, especialmente sua baixa latência de mensagem, sem adicionar um grande *overhead* de *software*.

A Figura 2.6 situa esta API dentro de um ambiente operacional que utiliza outras ferramentas de programação distribuída.

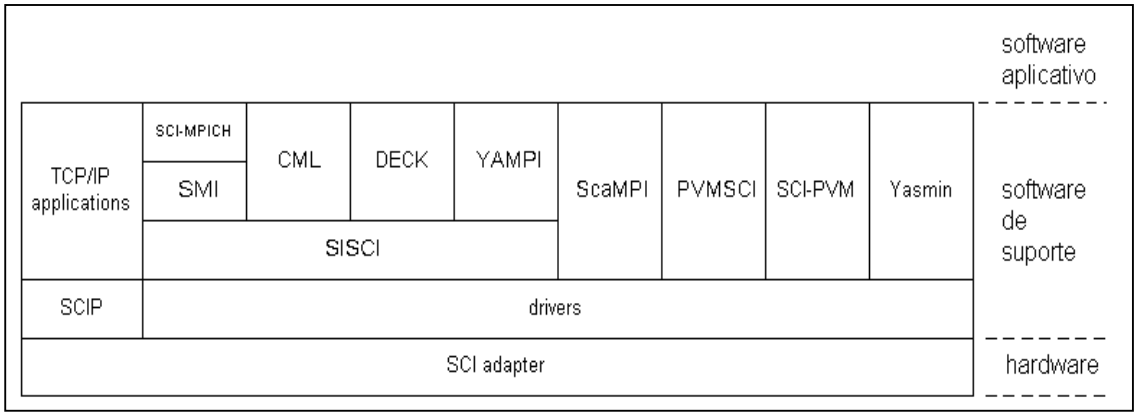

Figura 2.6: Interação YAMPI com ambiente operacional.

Os principais objetivos do projeto YAMPI são:

- Eficiência: este projeto é totalmente voltado para as capacidades da interface SCI, na tentativa de se obter valores de performance próximos a transferência de dados de memória obtidas de maneira nativa, que têm valores entre 3 e 5 μs. Deste modo as aplicações desenvolvidas poderão ser capazes de tirar proveito das larguras de banda extremamente altas oferecidas pela interconexão SCI.
- Similaridade com padrão estabelecido: a API desenvolvida deve possuir interface similar ao padrão determinado pelo MPI.

- Adequação ao modelo de *threads*: o resultado deste projeto deve naturalmente ser compatível com ambientes *multi-thread*. Este comportamento poder ser configurado em tempo de execução, uma vez que a adição de *locks* irá adicionar *overhead* na execução das operações.
- Portabilidade: todo o projeto é baseado na API SISCI, para que possa ser rapidamente portado para outros ambientes.

O YAMPI contempla seis funções básicas do MPI, sendo elas:

YAMPI_Init (int *argc, char *argv[])

Este comando inicia um processo YAMPI e deve sempre ser a primeira rotina a ser chamada. *argc* e *argv* são os parâmetros passados pela aplicação.

YAMPI_Finalize ( )

Finaliza o processo YAMPI. Deve ser a última rotina a ser chamada em uma aplicação YAMPI.

YAMPI_Comm_Rank (int *rank)

Identifica um processo YAMPI dentro do grupo. O identificador é retornado em *rank*.

YAMPI_Comm_Size (int *size)

Identifica o número de processos YAMPI dentro do grupo. O número total de processos é retornado em *size*.

YAMPI_Send (void *buf, int dest)

Rotina básica para envio de mensagens no YAMPI, e que utiliza o modo de comunicação bloqueante (*blocking send*). Neste modo a finalização da chamada depende da recepção, pelo nó destino, da mensagem enviada. Neste caso, o dado deve ter sido enviado com sucesso ao processo *dest*, indicando que o *buffer buf* pode ser reutilizado.

YAMPI_Recv (void *buf, int count, int source)

Rotina básica para a recepção de mensagens no YAMPI, e que utiliza o modo de comunicação bloqueante. Neste modo a finalização da chamada depende do envio, para o nó

remetente, da sinalização de recebimento da mensagem. Neste caso o dado, proveniente do processo *source*, deve ter sido armazenado em *buf*, estando pronto para ser utilizado.

A aplicação apresentada no Algoritmo 2.2 tem como objetivo enviar uma mensagem de teste do processo com identificador 0 para o processo com identificador 1. Inicialmente, inicia-se a aplicação em YAMPI com a primitiva *YAMPI_Init*. Esta retorna se a operação ocorreu ou não com sucesso. Em seguida, recupera-se o número total de processos envolvidos no comunicador YAMPI_COMM_WORLD através da primitiva *YAMPI_Comm_size*. Com a primitiva *YAMPI_Comm_rank*, recupera-se o número do identificador do processo. Caso o identificador seja 0, é enviada uma mensagem, cujo conteúdo é "Mensagem de teste" e está armazenada em *buf*, através da primitiva *YAMPI_Send( )*. Caso o identificador do processo seja 1, este recebe uma mensagem e armazena na variável *buf*. Em seguida, imprime o seu identificador e a mensagem recebida. Por último, é finalizada a aplicação com a primitiva *YAMPI_Finalize( )*.

2.4.4. MDX

O sistema MDX implementa um ambiente de execução paralela em um *cluster* de estações de trabalho rodando sistema operacional Linux. A principal característica do MDX é que ele permite os dois modelos de programação paralela: troca de mensagens e memória compartilhada.

Este sistema é implementado sobre o sistema operacional nativo (Linux) de cada nó do sistema. Um núcleo de comunicação disponibiliza ao usuário transparência na execução dos programas paralelos, encaminhando as solicitações de serviços aos servidores especializados de forma transparente. Os servidores especializados do sistema são: Servidor de Nomes, Servidor de Memória, Servidor de Sincronização, Servidor de Comunicação e Servidor de Nomes de Portas. O sistema foi organizado em uma arquitetura *multithread*, onde *threads* especializadas são responsáveis pela execução dos serviços oferecidos pelo ambiente.

O MDX serviu de base para várias dissertações de mestrado [COP00][LIB01] e trabalhos de conclusão de curso [HES99][SAL99], sendo que, atualmente, o modelo baseado em troca de mensagens encontra-se mais desenvolvido. Quanto ao modelo baseado em memória compartilhada, estão em fase de depuração os mecanismos que implementam a sincronização, a criação local e distante de *threads* e o balanceamento de carga. Detalhes da implementação do sistema MDX são descritos em [HES01]. Uma implementação do MDX para redes ATM é descrita em [COP00].

2.4.4.1. Núcleo de Comunicação

O núcleo de comunicação tem a função de receber todas as mensagens de requisição de serviços de clientes, identificar o servidor a que se destina a mensagem, descobrir a localização do mesmo e enviar a requisição do serviço. O servidor então, processa a mensagem e a reenvia ao núcleo de comunicação que a entrega ao cliente.

As diferentes localizações dos servidores não são percebidas pelos clientes, pois estas sempre enviam suas requisições de serviços para o núcleo local. Sendo assim, este núcleo fica responsável pela tarefa de localizar o servidor e realizar o envio e a recepção de mensagens distantes. O núcleo é composto por duas camadas: protocolo e encaminhamento de mensagens.

A camada protocolo é responsável por implementar um modelo cliente/servidor de base. Essa camada oferece quatro funções básicas: clientes enviam requisições, servidores recebem requisições, servidores enviam respostas e clientes recebem as respostas. Esse protocolo é uma variante do protocolo usado no modelo cliente/servidor e permite a chamada de um procedimento em uma máquina por um processo em execução em outra – RPC (*Remote Procedure Call*).

A camada encaminhamento de mensagens é responsável por disponibilizar transparência na localização dos clientes e servidores. É composta por dois processos que fazem a recepção, o roteamento e o envio de mensagens Em caso de cliente local, a mensagem é repassada diretamente a ele, enquanto que no caso distante a mensagem é repassada para o núcleo do cliente em questão, que a encaminha ao servidor destino.

2.4.4.2. Servidores Especializados

O Sistema MDX é composto, além do Núcleo de Comunicação, por servidores especializados, responsáveis por prestar serviços aos processos clientes. Esses servidores recebem requisições dos clientes, processam essas requisições, e enviam as respostas aos processos que solicitaram os serviços.

Os servidores podem ser centralizados ou distribuídos. No caso de servidor distribuído, instâncias do servidor são disparadas em diferentes nós do ambiente e cooperam para oferecer os serviços aos processos clientes. Desta forma, pode se obter uma melhor performance na execução dos serviços. No caso de servidor centralizado, apenas uma instância do servidor é disparada em um nó do ambiente. Neste caso, pode-se obter atomicidade na realização dos serviços.

Quando um servidor é executado, este fica em um *loop* infinito onde recebe requisições dos clientes, executa o serviço e envia resposta ao cliente. O servidor é finalizado quando uma requisição de término é recebida.

O MDX oferece ao usuário as seguintes primitivas

MDX_Task_Create (char *task_name, int address)

Inicia uma *task* em um determinado nó do sistema. O parâmetro *task_name* indica o nome do programa a ser executado e *address* indica o nó onde a *task* será executada.

MDX_Barrier_Create (barrier_id barrier, char *label, int quorum)

Cria uma barreira de sincronização. O parâmetro *label* indica o rótulo da barreira no sistema. *barrier* retorna os dados da barreira e *quorum* especifica o número de processos que irão sincronizar através da estrutura criada.

MDX_Barrier (barrier_id barrier)

Bloqueia o processo na barreira *barrier* até que o número de processos bloqueados indicado na sua criação seja atingido. Assim que este número é atingido, todos os processos são desbloqueados e continuam executando.

MDX_Barrier_Destroy (barrier_id barrier)

Finaliza a barreira indicada por *barrier*. Todos os processos que estão bloqueados nessa barreira são desbloqueados.

MDX_Barrier_LookUp(barrier_id *barrier, char *label)

Retorna o identificador de uma barreira previamente criada com um rótulo indicado por *label*. O retorno desta primitiva indica se a operação obteve sucesso (a barreira foi encontrada) ou não (não existe barreira associada ao *label* especificado).

No sistema MDX, programas que usam troca de mensagens se comunicam de forma indireta, através de portas de comunicação. As portas são criadas pelos processos e são gerenciadas pelo sistema. Uma porta possui um identificador único e os processos enviam e recebem mensagens através deste mecanismo. As portas são criadas de forma explícita em

qualquer nó do sistema. Esta possibilidade de criação de portas locais ou remotas implica na necessidade de um controle dos custos de comunicação relacionados à localização da porta.

A recepção de mensagens em uma porta não é restrita ao processo que a criou nem a processos localizados no mesmo nó em que se encontra a porta. Desta forma, o custo de uma operação *receive* em uma porta localizada remotamente é maior, uma vez que o envio da requisição e o recebimento da mensagem são feitos entre diferentes nós do ambiente. As portas de comunicação armazenam as mensagens enviadas em uma fila (FIFO) de mensagens, de onde são consumidas pelos processos que executam *receive*.

As primitivas de comunicação do MDX são:

MDX_Port_Create(port_id *port, int address)

Cria uma porta em um nó do sistema. Assim que esta primitiva é executada, *port* armazena os dados da porta criada no nó *address*.

MDX_Port_Destroy (port_id port)

Primitiva usada para destruir uma porta de comunicação. É passado como parâmetro o identificador único da porta a ser destruída.

MDX_SendtoPort(port_id port, void *msg, int size)

Envia uma mensagem para uma porta de comunicação. O parâmetro *port* indica a porta a qual se destina a mensagem. O parâmetro *msg* armazena o conteúdo da mensagem e *size* representa o número de bytes a serem enviados.

MDX_RecvFromPort(port_id port, void *msg, int size)

Primitiva usada para receber uma mensagem de uma porta. O parâmetro *port* indica a porta da qual deve ser recebida a mensagem. O parâmetro *msg* armazenará o conteúdo recebido e *size* representa o tamanho da mensagem. Esta primitiva é bloqueante, ou seja, caso não haja mensagens na porta indicada, o processo fica bloqueado até que uma mensagem seja enviada para ela.

MDX_Port_Register(port_id port, char *label)

Associa um rótulo (*label*) a uma porta de comunicação do sistema. O retorno dessa primitiva indica o sucesso ou não da execução do serviço. O primeiro parâmetro, *port*, armazena a

estrutura da porta a ser registrada no sistema. O segundo parâmetro, *label,* guarda o rótulo a ser associado à porta.

MDX_Port_LookUp(port_id *port, char *label)

É usada para retornar os dados de uma porta registrada no sistema. O primeiro parâmetro retornará o identificador da porta. O segundo parâmetro é o rótulo pelo qual a porta será localizada.

No modelo de memória compartilhada, variáveis compartilhadas são declaradas globais ao programa e são gerenciadas pelo servidor de memória do sistema MDX. Usando este modelo de programação, o programador não especifica onde os processos são criados. Esta decisão é tomada pelo sistema, através de algoritmos de balanceamento de carga. A única primitiva de memória compartilhada é:

MDX_shared [int, float, char, double] var_name

Declara uma variável global que é compartilhada pelos diferentes processos do mesmo programa paralelo. O sistema suporta os tipos de dados básicos da linguagem C, assim como *arrays* multidimensionais.

2.4.4.3. Implementação

O MDX possui algumas implementações já analisadas e documentadas em [HES01]. Partindo da implementação original, otimizações foram feitas e novas versões foram produzidas. Neste capítulo será apresentada a implementação que obteve melhor desempenho [HES01], MDX-v1.

MDX–v1

Em [COP00] é descrita a implementação de uma versão otimizada do sistema MDX, chamada MDX-v1, que também roda em redes ATM. Esta implementação visa solucionar os problemas do MDX, analisados no mesmo trabalho. Os problemas apontados são:

- Núcleo de comunicação complexo, o que resulta em um maior número de chamadas ao núcleo do sistema operacional.

- A estrutura do núcleo, com *threads* que fazem o roteamento de mensagens entre processos comunicantes, torna o caminho da mensagem cliente-servidor-cliente muito longo (6 trocas de mensagens na versão original, para o pior caso).
- *Overhead* de controle na troca de mensagens, ou seja, pacotes de requisição e de resposta carregam muitos dados para controle.

Esta implementação tem como objetivo reduzir a complexidade do núcleo de comunicação, diminuindo ao máximo a comunicação entre clientes e servidores, e mantendo disponíveis as primitivas das versões anteriores. A seguir será detalhada a implementação desta nova arquitetura para o MDX.

O núcleo de comunicação foi reduzido a uma tabela de local de nomes (NLT), a qual armazena a localização dos servidores na rede de processadores e é compartilhada pelos processos clientes. Desta forma, processos clientes podem se comunicar diretamente com servidores, sem que a mensagem passe pelo núcleo. Para enviar uma solicitação de serviço a um servidor, o cliente consulta a NLT e obtém a localização do servidor e o *socket* através do qual o mesmo recebe as requisições. Caso a NLT não contenha o *socket* de comunicação com o servidor, este é criado pelo cliente e a conexão com o servidor é estabelecida. O *socket* a ser usado na comunicação com o servidor é adicionado à tabela. Através deste mecanismo, o cliente se comunica diretamente com o servidor, enviando requisições e recebendo respostas. A busca por informações na NLT e o estabelecimento da conexão com os servidores são transparentes para o programador, sendo realizados pelas primitivas MDX.

A Figura 2.7 mostra a arquitetura do sistema MDX-v1, onde núcleo de comunicação e cliente compartilham a NLT, e a comunicação do cliente com o servidor é direta.

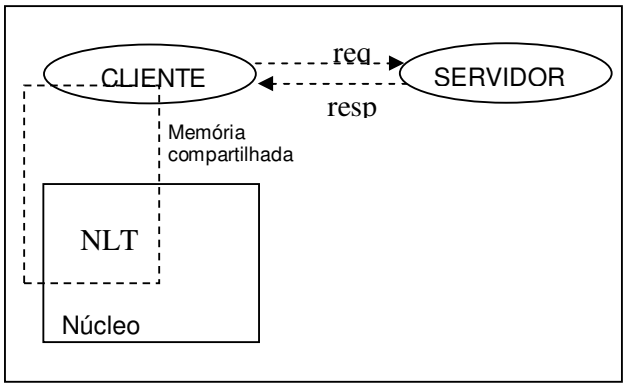

Figura 2.7: Arquitetura do MDX.

As funções do núcleo de comunicação são as seguintes:

- Criar a área de memória compartilhada onde será armazenada a NLT.
- Carregar os dados da NLT, tendo como fonte um arquivo de configuração do ambiente.

A NLT consiste em uma tabela residente na memória através da qual os processos podem localizar rapidamente o endereço do destino da mensagem. No esquema do MDX, o núcleo recebe todos os pedidos de clientes e executa uma consulta na NLT em busca da localização do servidor.

Como os processos do núcleo, dos servidores e dos clientes estão separados, há a necessidade da troca de informações contidas na NLT entre esses processos. Para isso existem algumas alternativas de IPC (*Interprocess Communication*): troca de mensagens, memória compartilhada, sinais, semáforos, dutos (*pipes*) e RPCs. A alternativa de memória compartilhada é que mais eficiência proporciona na implementação do compartilhamento da NLT, pois não há necessidade de cópia entre os processos nem de operações de entrada/saída.

No MDX-v1, a NLT (Tabela 2.1) reside na memória compartilhada, uma região de memória alocada pelo núcleo e mapeada pelos processos clientes e servidores. A implementação dessa área compartilhada é feita via chamadas de sistema: *shmget* (criação), *shmat* (mapeamento) e *shmctl* (controle da área).

Tabela 2.1: Estrutura da NLT.

Identificador do processo
Identificação do servidor MDX
Endereço do servidor (IP, atm_svc, atm_pvc)
Socket envio pedido
Identificador do Nó

Na NLT, o campo identificador do processo é um número retornado pela chamada de sistema *getpid( )*. O motivo dessa identificação deve-se ao fato de cada cliente ter um *socket* para envio e recebimento de mensagens. Se o cliente for enviar uma requisição a um servidor,

ele deve primeiramente consultar a NLT, na entrada correspondente ao seu número de processo.

Os campos de identificação e endereço do servidor especificam a localização do servidor no ambiente paralelo e seu endereço. O campo *socket* de envio permite que o cliente, quando transmite sua segunda requisição, envie diretamente para a conexão já estabelecida na primeira vez. Do lado do servidor, a resposta é encaminhada pelo mesmo *socket* onde foi feita a leitura. O campo de identificador do nó se aplica a servidores distribuídos, ou seja, replicados na rede de processadores.

Uma vez criada a área de memória compartilhada, é necessário carregar esta tabela com os dados gravados em um arquivo de configuração. Encerrado o preenchimento da NLT, o núcleo apenas aguarda até que o usuário o finalize. O usuário encerra o núcleo pressionando qualquer tecla.

No MDX-v1, quando um processo cliente deseja se comunicar com um servidor, ele deve estabelecer uma conexão com o mesmo. Cada servidor do MDX-v1 aguarda conexões em uma determinada porta.

Visando paralelismo no atendimento de requisições de diferentes clientes, assim que um cliente estabelece conexão com o servidor, é criada uma *thread* para atender exclusivamente a este cliente. A partir de então, o *socket* através do qual o cliente efetuou a conexão e a *thread* criada são dedicados exclusivamente à comunicação e ao atendimento de requisições daquele cliente. Desta forma, o cliente estabelece conexão com o servidor apenas uma vez. O *socket* desta conexão é armazenado no campo *socket* na NLT (ver Tabela 2.1).

A *thread* criada para atender a determinado cliente possui a mesma funcionalidade dos servidores da versão original do MDX. É um processo que fica em *loop* aguardando mensagens (*receive*) do cliente, tratando as requisições e devolvendo respostas.

A eliminação do núcleo de comunicação no processo de envio de requisições, também influencia no momento da localização do servidor. Para isso, os processos clientes dispõem do número da porta padrão em que cada servidor MDX-v1 aguarda requisições. De posse desta informação, basta localizar, na NLT, o endereço do servidor destino.

Esta forma de comunicação também diminui o *overhead* de controle nas requisições, pois não é necessário o cliente especificar na mensagem qual o servidor ao qual se destina a requisição, apenas o serviço e os dados relevantes para a realização do mesmo.

No MDX-v1, além das primitivas de troca de mensagens através de portas de comunicação, foi também implementada a comunicação direta entre processos. Neste novo

formato de comunicação, cada cliente, ao iniciar seu funcionamento, registra-se no servidor de comunicação com um identificador único. A partir desse registro, os processos podem enviar mensagens diretamente para outro, indicando o identificador do processo destino. As novas primitivas de comunicação são:

MDX_RegisterClient(int id)

Essa primitiva cria um *socket* para futura comunicação e envia uma requisição de registro de cliente (REG_CLIENT) ao servidor de comunicação. As informações do cliente (localização e *socket*) são registradas com o identificador passado em *id*.

MDX_LookupClient(int id)

Localiza um cliente no sistema para futura comunicação. Esta primitiva envia ao servidor de comunicação uma requisição de localização de cliente (LOOKUP_CLIENT), informando o identificador do mesmo, passado por parâmetro em *id*. O servidor devolve como resposta as informações (endereço e *socket*) do cliente procurado. Com essas informações, a primitiva adiciona à tabela local de nomes (NLT) o cliente localizado e retorna 1. Caso algum erro aconteça, o valor 0 é retornado.

MDX_SendTo(int id_dest, void *msg, int msg_length)

Envia uma mensagem de forma direta a um outro processo do sistema. As informações (enderece e *socket*) do cliente destino, passado por parâmetro em *id_dest*, são localizadas na NLT. O conteúdo da mensagem, passado em *msg*, com tamanho *msg_length*, é enviado.

MDX_RecvFrom(void *msg, int msg_length)

Esta primitiva efetua um *receive* no *socket* criado para comunicação em *MDX_RegisterClient( )*. Caso não haja mensagens no *socket*, o processo fica bloqueado aguardando a chegada da primeira mensagem. São passados como parâmetro a variável onde será armazenada a mensagem recebida, *msg*, e o número de bytes a serem recebidos, *msg_length*.

Após a apresentação das bibliotecas de programação, pode-se verificar que cada uma delas possui suas características próprias, na sintaxe das primitivas, ou ainda na forma de programação. Por exemplo, enquanto o MPI comunica de forma direta por chamadas

MPI_Send( ) e *MPI_Recv( )*, o GM utiliza portas de comunicação para troca de mensagens. Isto dificulta a utilização de mais de uma biblioteca de forma conjunta, como, por exemplo, para executar programas paralelos em uma estrutura de *cluster* de *clusters* onde um *cluster* utiliza rede *Myrinet* e biblioteca GM para programação, e outro utiliza rede SCI e biblioteca YAMPI.

Capítulo 3

Cluster de *Clusters*

Conforme visto anteriormente (seção 2.1.2.4), máquinas agregadas (COW – *cluster of workstations*) constituem a tendência mais atrativa na construção de máquinas paralelas, devido ao avanço nas tecnologias de redes locais (*Myrinet*, *Fast-Ethernet*, SCI) e de processadores, e ao baixo custo em relação a outras arquiteturas.

Esse avanço das tecnologias de processadores e de redes possibilita também a agregação de *clusters*, formando uma estrutura de *cluster* de *clusters*, como uma única máquina paralela. Esta interligação de agregados, por meio de uma rede local ou via internet, visa principalmente alcançar o máximo desempenho na execução de aplicações.

Uma possibilidade de ganho de desempenho está no aumento do número de nós disponíveis. Uma vez que este número aumenta, o poder computacional alcançado também é maior. Aplicações complexas, que necessitam de um grande potencial de recursos, são distribuídas nos nós dos agregados envolvidos na estrutura de forma a conseguir uma melhora de desempenho.

Outra possibilidade de ganho de desempenho está no mapeamento de processos. De acordo com as características de cada agregado, como, por exemplo, a velocidade da rede utilizada e o poder computacional dos nós, partes da aplicação podem executar de forma mais eficiente em um determinado *cluster*. Por exemplo, uma aplicação poderia ser mapeada da

seguinte maneira: processos que realizam um grande número de trocas de mensagens executam em nós de um agregado com rede mais rápida enquanto que, processos que realizam operações mais complexas e com menos comunicação rodam em um *cluster* cujos nós tenham um poder de processamento maior.

A Figura 3.1 mostra um exemplo de interligação de *clusters*. O agregado C1, com doze nós interligados por rede primária *Myrinet* está ligado aos *clusters* C2, com seis nós e rede primária SCI, e C3, com oito nós e rede primária *Fast-Ethernet*. Os nós dos agregados C1 e C2 possuem como rede secundária *Fast-Ethernet*, através da qual estão ligados aos nós dos outros agregados por um *switch Fast-Ethernet*.

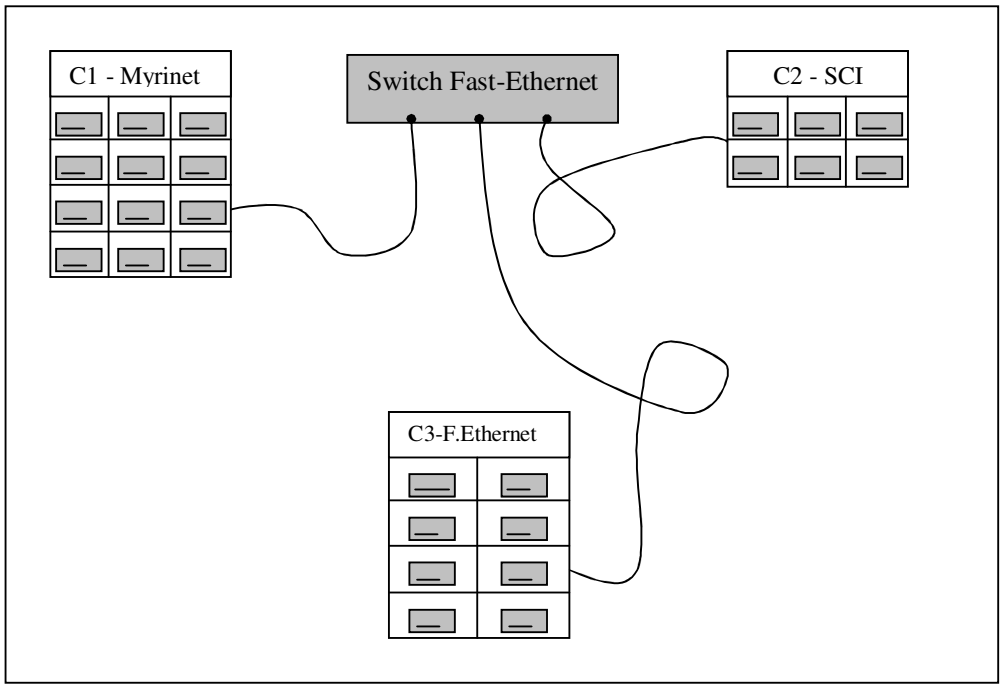

Figura 3.1: Exemplo de *cluster* de *clusters*.

Atualmente, pesquisas vêm sendo desenvolvidas em uma área chamada *Grid Computing*, que visa a interligação de um grande número de máquinas, que podem ser *clusters*, geograficamente distribuídas, de maneira a formar um metacomputador e tirar o máximo proveito do poder computacional alcançado [BUY99]. A utilização de *cluster* de *clusters* pode ser um passo importante nessa direção.

A Figura 3.2 mostra uma arquitetura de *cluster* de *clusters* formada por quatro agregados. O agregado C1 possui doze nós interligados por rede *Myrinet*, e está ligado por uma rede local a um agregado C2, que possui seis nós interligados por rede SCI. C2 está

conectado, via internet, aos agregados C3 e C4, sendo que o primeiro possui quatro nós interligados por tecnologia *Ethernet*, e o último é formado por cinco nós interconectados por rede *Myrinet*.

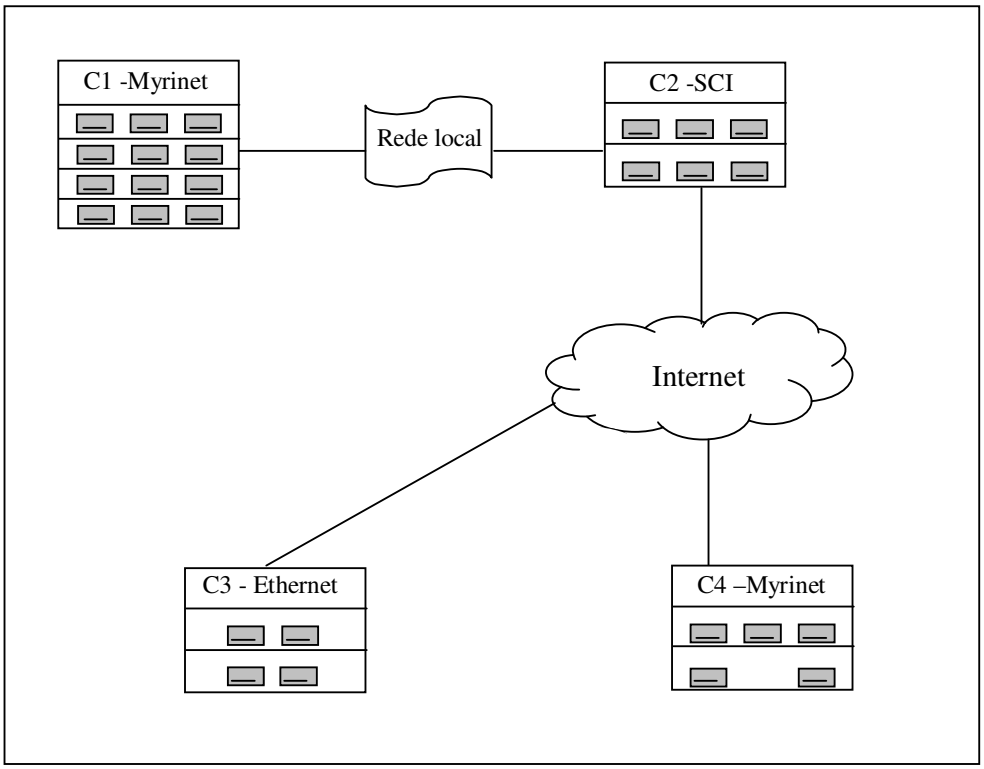

Figura 3.2: Exemplo de metacomputador.

Evidentemente, o uso de uma estrutura de *cluster* de *clusters* esbarra em alguns problemas. Um dos principais é a complexidade de programação, principalmente para que seja explorada a possibilidade de ganho de desempenho pelo mapeamento de processos. Deve haver uma análise das necessidades de cada processo, bem como das características de cada agregado. Por exemplo, pode ser interessante que processos que necessitam compartilhar grande quantidade de dados executem em um *cluster* com rede SCI, que fornece mecanismo de memória compartilhada em *hardware*. Também pode ser interessante que processos que consomem muita CPU rodem em um *cluster* com nós mais poderosos, mesmo que tecnologia de rede utilizada seja mais lenta. Esse mapeamento torna a programação bastante complicada, uma vez que pode envolver mais de um modelo de programação (troca de mensagens e memória compartilhada), além de ser necessário o desenvolvimento de um algoritmo eficiente

de mapeamento de processos. Por sua complexidade, a exploração da possibilidade de ganho de desempenho pelo mapeamento de processos não faz parte do escopo deste trabalho.

A exploração da possibilidade de ganho de desempenho pelo aumento do poder computacional esbarra em problemas mais simples, como a necessidade de uma ferramenta de programação que ofereça suporte à heterogeneidade de tecnologia de rede. Nesse caso, o modelo de programação pode ser único, mas a ferramenta deve fornecer mecanismos de comunicação entre processos que executam em *clusters* com tecnologias de rede distintas. Atualmente, existe um grande número de bibliotecas de comunicação destinadas à programação sobre agregados com uma determinada tecnologia de rede. Podem ser citadas como exemplos a biblioteca YAMPI (seção 2.4.3) para redes SCI, GM (seção 2.4.1) para redes *Myrinet*, e MPI (seção 2.4.2), que possui diferentes implementações para redes SCI, *Myrinet* e *Fast-Ethernet*. Surge a necessidade de uma ferramenta única, que ofereça suporte à comunicação por essas redes e, além disso, ofereça possibilidade de comunicação entre processos que executam em agregados diferentes, mesmo que esses utilizem tecnologias de rede distintas.

Capítulo 4

Sistemas Relacionados

Neste capítulo, serão apresentados sistemas de programação paralela que atacam os problemas decorrentes do uso de uma estrutura de *cluster* de *clusters* como uma máquina paralela.

4.1. Projeto *MultiCluster*

O projeto *MultiCluster*, desenvolvido pelo Grupo de Processamento Paralelo e Distribuído da Universidade Federal do Rio Grande do Sul (UFRGS), objetiva um ambiente de desenvolvimento de aplicações paralelas voltado à utilização de agregados de *clusters*.

A idéia principal do projeto é permitir a integração de agregados baseados em tecnologias de comunicação diferentes, visando permitir a combinação de modelos de programação distintos (troca de mensagens e memória compartilhada). A biblioteca DECK [BAR00] serve de base para o modelo *MultiCluster*. Para melhor entendimento do modelo, ele pode ser dividido em aspectos de *hardware* e de *software*.

4.1.1. Aspectos de Hardware

A escolha da tecnologia usada para interligar os nós dos *clusters*, bem como daquela utilizada para interligar *clusters*, depende muito das necessidades de cada aplicação [BAR00a]. Para o

uso do *MultiCluster*, independe a tecnologia que interligue os nós e os agregados (*Fast-Ethernet*, *Myrinet* ou SCI). Basta que um dos nós de cada *cluster* faça o papel de *gateway* para ligação com outro agregado.

4.1.2. Aspectos de Software

A camada de *software* do modelo *MultiCluster* segue algumas definições conceituais para viabilizar a integração de maquinas agregadas.

Nós Lógicos e Nós Físicos.

Um nó físico corresponde a uma máquina disponível em qualquer dos agregados envolvidos na estrutura. Nós lógicos correspondem ao conjunto de máquinas disponíveis do ponto de vista da aplicação. No caso de agregados que utilizam troca de mensagens um nó lógico corresponde a um nó físico. No caso de agregados que utilizam memória compartilhada, um nó lógico pode corresponder a mais de um nó físico.

Comunicação Intra-nós e Inter-nós

As aplicações trabalham apenas com a visão de nós lógicos. Desta forma, é relativamente fácil adaptar diferentes modelos de programação: dentro de um mesmo nó lógico a comunicação é feita por memória compartilhada; entre nós lógicos, a comunicação é feita por troca de mensagens. Do ponto de vista do usuário, o modelo de comunicação entre processos é único, e a camada de comunicação se encarrega de implementar um ou outro.

Heterogeneidade

Embora seja um problema menos freqüente, a heterogeneidade pode aumentar de acordo com as características dos *clusters* que serão interligados. Neste ponto, são consideradas as diferentes representações de dados e a necessidade de indicar ao destinatário da mensagem qual a arquitetura do processo remetente. Esse problema deve ser tratado implicitamente pela camada de comunicação.

Para diminuir qualquer queda de performance que possa haver em uma integração deste tipo, é dada ao usuário a possibilidade de definir a melhor localização para as *tasks* do programa, criando mecanismos próprios de comunicação para cada uma. Com essa facilidade, a comunicação entre processos em *clusters* diferentes pode ser balanceada e diminuída tanto quanto possível, diminuindo o tráfego na rede que interliga os agregados.

4.1.3. Biblioteca DECK

DECK (*Distributed Execution and Communication Kernel*) [BAR00] é uma biblioteca de que provê abstrações básicas para aplicações paralelas, como *threads* e caixas postais, além de serviços mais complexos, como serviço de nomes e comunicação coletiva.

DECK possui duas camadas. A camada de baixo é chamada de μDECK e é responsável pelas abstrações básicas: *threads*, semáforos, mensagens, caixas postais e memória compartilhada. A camada mais acima é a camada de serviços, serviços estes que podem ser escolhidos em tempo de compilação. Dois serviços devem ser analisados de forma mais completa: serviço de nomes e RCD (*Remote Communication Daemon*).

O servidor de nomes é um processo dedicado que executa no primeiro nó de cada *cluster*. Por exemplo, pode haver um servidor de nomes rodando no nó “verissimo" e outro no nó “scliar". Cada servidor de nomes é responsável por registrar caixas postais criadas no agregado onde estão rodando e é executado automaticamente no início da aplicação. A camada RCD é descrita mais detalhadamente na seção 4.1.6.

O DECK funciona, atualmente, sobre redes *Myrinet* e SCI. A parte de *multithread*, em ambas as implementações, utilizam chamadas POSIX *Threads* [IEE95].

4.1.4. DECK/*Myrinet*

Para implementação da comunicação sobre redes *Myrinet*, DECK utiliza a biblioteca BIP – *Basic Interface for Parallelism* [PRY98], principalmente por sua simplicidade e desempenho. BIP provê primitivas de comunicação ponto-a-ponto (*send* e *recv* assíncronos), com a obrigatoriedade da troca de mensagens grandes (termo usado por BIP) seguirem um modelo *rendez-vouz*, que indica que uma primitiva *recv* deve ser completada antes de cada operação *send* correspondente.

Para seguir esse modelo, DECK utiliza um protocolo de negociação entre processos onde o remetente envia mensagens de requisição pequenas antes de enviar uma mensagem grande. Essas requisições são tratadas, no lado do receptor da mensagem, por uma *thread* dedicada, a *rv-daemon*, criada no início do processo.

Quando um processo envia uma mensagem pequena, esta é colocada diretamente na caixa postal destino, pois BIP suporta o armazenamento de mensagens pequenas em *buffers* internos. No caso de uma mensagem grande, uma requisição é enviada ao *rv-daemon* do lado do receptor. Essa *thread* verifica se há espaço nos *buffers* para armazenar a mensagem e

responde. O remetente permanece enviando a requisição ao *rv-daemon* do receptor até receber uma resposta positiva, quando, enfim, envia a mensagem.

4.1.5. DECK/SCI

A implementação do DECK/SCI é feita sobre duas bibliotecas de programação para SCI: *Yasmin* [TAS99], que provê mecanismos para criação, mapeamento e sincronização de segmentos compartilhados, e *Sthreads* [REH99], que oferece chamadas POSIX *Threads* sobre o *Yasmin*.

São oferecidos, pela camada μDECK, serviços de criação, denominação, mapeamento e *locking* de segmentos compartilhados. Diferentemente de *Myrinet*, SCI oferece uma fácil implementação dos dois modelos de programação. Desta forma, DECK/SCI oferece caixas postais e troca de mensagens, além de mecanismos de memória compartilhada. É necessário lembrar que segmentos de memória só podem ser compartilhados por processos localizados em um mesmo nó lógico.

4.1.6. RCD

Para oferecer suporte ao modelo *MutliCluster*, o RCD (*Remote Communication Daemon*) foi implementado como um serviço do DECK para realizar a comunicação entre *clusters*. Como cada *cluster* possui um nó que faz o papel de *gateway*, o RCD é executado, neste nó, automaticamente no início da aplicação.

O RCD atua em duas ocasiões: quando a aplicação procura por um nó localizado em outro agregado e quando mensagens são enviadas a uma caixa postal remota (em outro *cluster*). Quando uma primitiva DECK falha na busca por uma caixa postal em um servidor de nomes local, ela contata o RCD, que envia uma mensagem *broadcast* aos outros RCDs no sistema e aguarda por uma resposta, retornando-a à primitiva que o chamou. No segundo caso, quando uma primitiva DECK necessita enviar mensagem a uma caixa postal remota, ela contata o RCD, que encaminha a mensagem ao RCD responsável pela comunicação remota no agregado onde a caixa postal destino está localizada.

4.2. IMPI

Desde a publicação do padrão MPI [MES94], um grande número de implementações MPI de alta qualidade vêm sendo disponibilizados. Dentre essas implementações, podem ser

destacadas LAM/MPI [BUR94], da *University of Notre Dame*, e MPICH [DOS93], do *Argonne National Laboratory*.

Uma vez que apenas a funcionalidade do MPI é especificada pelo padrão, cada implementação é otimizada para uma determinada arquitetura. Desta forma, não é possível que versões diferentes do MPI cooperem na execução de uma aplicação.

Para resolver este problema, foi formado o *Interoperable MPI Steering Committee* (IMPI). Este comitê elaborou uma proposta de padrão para a interoperabilidade entre diferentes implementações MPI

4.2.1. Visão Geral do IMPI

Um dos principais objetivos do IMPI é oferecer uma interface de programação idêntica a do MPI, mas utilizar múltiplas versões do MPI para executar uma única aplicação. Assim, uma aplicação que executa corretamente com MPI não necessita qualquer alteração para rodar com o IMPI.

O padrão IMPI é dividido em quatro partes: protocolos de *startup*/*shutdown*, protocolo de transferência de dados, algoritmos coletivos e metodologia de testes.

4.2.2. Terminologia

O padrão IMPI usa o termo "implementações MPI" para referir universos MPI que são utilizados para executar uma aplicação. Não significa necessariamente implementações diferentes do MPI, mas também múltiplas instâncias da mesma implementação. Por exemplo, o IMPI pode utilizar duas instâncias de uma versão LAM/MPI.

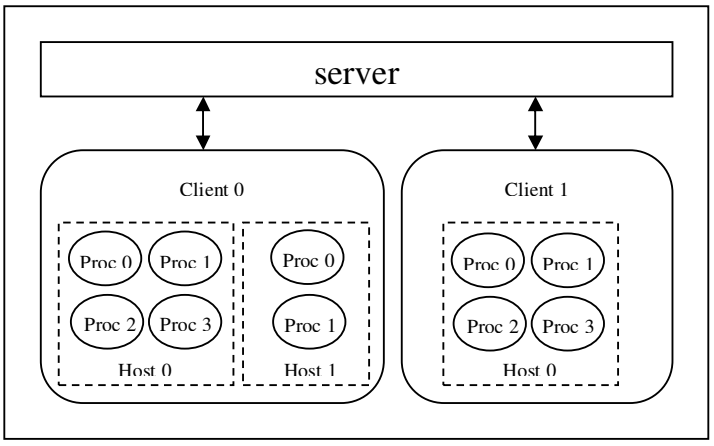

Figura 4.1: Entidades do IMPI.

Existem quatro entidades IMPI: *server*, *clients*, *hosts* and *procs*. Essas entidades são apresentadas pela Figura 4.1 e são descritas a seguir.

O *server* é o ponto de encontro entre implementações MPI. Há um *client* por implementação e esse *client* é o representante da implementação no *server*. Pode haver um máximo de 32 *clients* para executar uma aplicação IMPI. Cada *client* pode ter um ou mais *hosts*. Cada *host* representa um grupo de processos (*procs*).

O *server* é o ponto de encontro entre implementações MPI. Há um *client* por implementação e esse *client* é o representante da implementação no *server*. Pode haver um máximo de 32 *clients* para executar uma aplicação IMPI. Cada *client* pode ter um ou mais *hosts*. Cada *host* representa um grupo de processos (*procs*).

4.2.3. Protocolos de *Startup/Shutdown*

Dois passos são necessários para disparar uma aplicação IMPI. O primeiro passo é iniciar o *server*, que fica aguardando conexões dos *clients*. Um *client*, ao conectar o *server*, envia ao mesmo as suas informações, como o número de *hosts* que ele representa, o número de *procs* de cada *host*, tamanho máximo de pacote suportado, etc.. Então o *server* junta as informações de todos os *clients* e as envia a todos os *clients*. Desta forma, cada *client* obtém as informações referentes a todos os outros *clients* e pode passá-las a seus *hosts*.

Com base nessas informações, os *clients* decidem de forma será feita a comunicação entre eles. Por exemplo, cada *client* envolvido em uma comunicação compara o tamanho máximo de pacote suportado pelos dois e opta pelo menor valor. Este processo é chamado "negociação", embora as decisões sejam tomadas de forma independente e distribuída.

Após o *startup*, tem início o segundo passo. Os *hosts* devem criar conexões TCP entre eles, e essa conexão será usada para troca de mensagens entre implementações MPI. Após a criação dessas conexões, o *server* e os *clients* ficam simplesmente aguardando o fim da execução da aplicação.

O processo de *shutdown* ocorre da seguinte maneira. Cada *proc* envia uma mensagem ao seu *host* indicando a seu término. Assim que um host recebe essa indicação de todos os seus *procs*, ele transmite essa mensagem para seu *client*. De forma semelhante, quando o *client* recebe mensagens de término de todos os seus *hosts*, ele repassa a mensagem ao *server*. Finalmente, o *server* termina quando receber mensagem de término de todos os *clients*.

4.2.4. Protocolo de Transferência de Dados

Mensagens trocadas entre processos executando em uma mesma implementação MPI utilizam os mecanismos otimizados desenvolvidos para aquela implementação. O IMPI só interfere na comunicação entre processos que rodam em diferentes implementações.

Caso um processo deseja enviar uma mensagem a outro que executa em outra implementação, a mensagem deve ser passada do *host* local, que a repassa ao *host* remoto. O *host* remoto deve transmitir a mensagem ao processo destino.

4.2.5. Algoritmos Coletivos

Como as comunicações por TCP são lentas, se comparado à troca de mensagens em uma única implementação MPI (para redes *Myrinet*, por exemplo), os algoritmos coletivos, como MPI_BARRIER, por exemplo, foram desenvolvidos de forma a minimizar a comunicação entre diferentes implementações. A maioria desses algoritmos tem uma fase local e uma fase global. Por exemplo, a execução de uma barreira entre múltiplas implementações possui duas fases. Na primeira, todos os processos executando em uma mesma implementação sincronizam por sua barreira local. A segunda fase é a sincronização das barreiras locais por uma barreira global.

4.2.6. Metodologia de Testes

Está disponível uma ferramenta de teste do IMPI. Um *applet* Java, que pode ser encontrado em http://impi.nist.gov/IMPI/, oferece um simulador IMPI, capaz de emular um IMPI *server* e qualquer número de *hosts* e *procs*.

Através desse simulador, uma implementação do IMPI pode ser testada. Uma vez passando esses testes, uma implementação IMPI teoricamente está pronta para cooperar com qualquer outra implementação IMPI que também tenha passado pelos testes.

4.3. MPICH/Madeleine

O objetivo do MPICH/Madeleine é oferecer uma versão do MPICH [DOS93] com suporte eficiente e simultâneo a diferentes tecnologias de rede. Esta versão foi produzida sobre uma biblioteca de comunicação que já oferece suporte a múltiplas tecnologias de rede, o Madeleine [AUM00], que é o módulo de comunicação da ferramenta de programação PM².

A interface de programação Madeleine provê um pequeno conjunto de primitivas orientadas à troca de mensagens. Basicamente, oferece primitivas para envio e recebimento de mensagens, bem como primitivas que permitem que o usuário defina de que maneira os dados serão inseridos ou extraídos das mensagens.

O Madeleine gerencia o uso de diferentes protocolos de rede ao mesmo tempo, além de gerenciar o uso de múltiplos adaptadores de rede para cada um desses protocolos. Isso permite que a aplicação do usuário escolha dinamicamente um determinado protocolo, de acordo com suas necessidades.

Este controle é oferecido por dois objetos básicos. O objeto *channel* define um universo fechado de comunicação. Cada *channel* é associado a um protocolo de rede, um adaptador de rede correspondente e um conjunto de objetos *connection*. Cada *connection* representa uma conexão ponto-a-ponto entre dois processos.

A definição do tamanho do *buffer* de mensagens é dependente das tecnologias disponíveis para comunicação. Os pontos de *buffer* para rede TCP/*Fast-Ethernet*, SISCI/SCI e BIP/*Myrinet* são, respectivamente, 64 Kbytes, 8 Kbytes e 7 Kbytes. No caso do uso de mais de uma tecnologia de rede, o tamanho do *buffer* que armazena a mensagem deve ser igual ao maior ponto de escolha entre as tecnologias disponíveis. Por exemplo, no caso de um agregado que utiliza rede SCI e *Fast-Ethernet*/TCP, o *buffer* deve ser de 64 Kbytes. Caso a mensagem seja enviada por rede SCI, o *buffer* nunca será completado e vários valores *null* serão enviados, prejudicando o desempenho da comunicação.

Diferentemente de outras abordagens, que utilizam conexões TCP para conectar múltiplas implementações MPI, o MPICH/Madeleine permite às aplicações utilizar *clusters Myrinet*, SCI e *Ethernet* conectados por qualquer dessas tecnologias de rede. No entanto, há a necessidade de que todos os nós da estrutura estejam conectados diretamente. Atualmente, encontra-se em desenvolvimento um mecanismo de conexão que utiliza nós que fazem o papel de *gateway*, de maneira semelhante àquela implementada no projeto *Multicluster* (seção 4.1).

4.4. Comparação entre os Sistemas

Após apresentados os sistemas de programação, pode ser montada a seguinte tabela de comparação entre eles:

Tabela 4.1: Comparação entre sistemas.

Sistema	*Biblioteca de Programação*	*Tecnologias Suportadas*	*Conexão entre os clusters*	*Tecnologia de conexão entre os clusters*
IMPI	MPI	Fast-Ethernet Myrinet SCI	Todos os nós conectados	Fast-Ethernet/TCP
Multicluster	DECK	Fast-Ethernet Myrinet SCI	Um nó em cada cluster faz o papel de gateway	Fast-Ethernet
MPICH/Madeleine	MPI	Fast-Ethernet Myrinet SCI	Todos os nós conectados	Fast-Ethernet Myrinet SCI

Todos os sistemas de programação apresentados possuem características próprias, conforme pode ser visto na Tabela 4.1. Além disso, cada um deles apresenta seus problemas. O Projeto *Multicluster* teve seu desenvolvimento interrompido, e ainda nenhum protótipo foi disponibilizado. O padrão IMPI foi implementado apenas uma vez como um protótipo para validar o padrão. Este protótipo não contempla todas as definições do padrão. Da mesma forma, se encontra disponível apenas um protótipo do MPICH/Madeleine, no qual estão sendo feitas alterações no sentido de melhorar sua performance.

Capítulo 5

Ambiente MDX-cc

Este trabalho propõe e implementa um ambiente de programação paralela para rodar sobre uma arquitetura de *cluster* de *clusters*, oferecendo os recursos necessários para o aproveitamento de uma estrutura desta natureza. Este capítulo descreve esse ambiente, em termos de recursos e funcionalidades, e apresenta detalhes de implementação do mesmo.

5.1. Motivação

O projeto aqui descrito foi motivado por alguns aspectos importantes. O principal deles é a necessidade de um melhor aproveitamento da infra-estrutura oferecida pela universidade. Hoje, a PUCRS conta com um importante centro de pesquisa em processamento de alto desempenho, o CPAD. Esse centro dispõe de quatro agregados, sendo que dois deles utilizam rede *Fast-Ethernet*, um usa rede SCI e outro rede *Myrinet*. O CPAD tem interesse em executar aplicações paralelas utilizando todo o poder computacional disponível. Para isso, é interessante a utilização, por uma mesma aplicação, de todos os agregados interligados, formando uma estrutura heterogênea de *cluster* de *clusters.* A utilização dessa estrutura requer um ambiente que possibilite a programação e a execução de programas paralelos em *cluster* de *clusters*.

O Grupo de Processamento Paralelo e Distribuído do curso de Pós-Graduação em Ciência da Computação da PUCRS vem desenvolvendo, há alguns anos, uma ferramenta de programação paralela, o Sistema MDX [HES99][COP00], que executa sobre máquinas agregadas. A idéia deste trabalho é unir a experiência no desenvolvimento do MDX com a experiência na programação sobre *clusters*, adquirida no CPAD, para implementar um ambiente de programação paralela para rodar sobre *cluster* de *clusters*: o MDX-cc. O MDX-cc é baseado no Sistema MDX, em sua implementação MDX-v1 [COP00] (seção 2.4.4). No entanto, oferece apenas o paradigma de programação por troca de mensagens (o MDX oferece também o paradigma de memória compartilhada). A programação por memória compartilhada foi deixada de lado neste primeiro momento devido ao tempo disponível para este trabalho e ao grande número de modificações que se fizeram necessárias na estrutura do ambiente original para adaptar o seu uso a uma estrutura de *cluster* de *clusters* (seção 5.3).

O MDX-cc executa processos de um mesmo programa paralelo, no modelo SPMD (Figura 5.1), em vários *clusters*, ligados um ao outro por rede *Fast-Ethernet*. Esses *clusters* devem, internamente, ter seus nós conectados por rede *Fast-Ethernet*, SCI ou *Myrinet*. A idéia principal é oferecer ao usuário total transparência na comunicação entre os processos. A troca de mensagens deve ocorrer independentemente da tecnologia de rede disponível no nó onde cada um está localizado, de forma que o usuário simplesmente faça chamadas do tipo *send* e *receive*, não se preocupando com qual tecnologia será usada para comunicação.

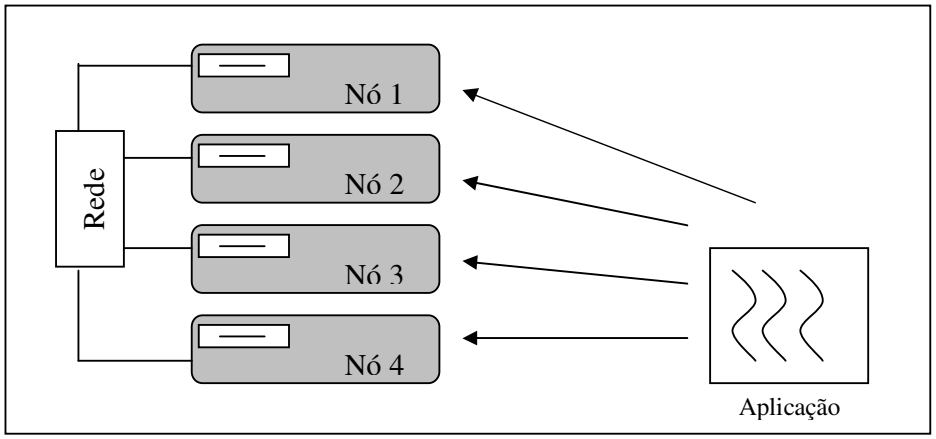

Figura 5.1: Modelo SPMD do MDX-cc.

5.2. Definições do MDX-cc

O MDX-cc provê algumas funcionalidades básicas para executar aplicações paralelas sobre agregados heterogêneos interligados. Essas funcionalidades são:

Detecção das tecnologias de rede disponíveis em cada nó

Tendo em vista que processos podem rodar em nós de agregados que usam, internamente, redes distintas, cada processo MDX-cc identifica, de forma automática, as tecnologias de rede disponíveis no nó onde está executando. Com base nessa informação, entre outras, o MDX-cc vai decidir por qual rede a comunicação com outro processo será realizada.

Protocolo de Comunicação

O MDX-cc fornece os mecanismos de troca de mensagens entre processos. A tecnologia de rede utilizada na comunicação deve ser transparente ao usuário, de forma que ele apenas faça chamadas do tipo *send* e *receive*, ficando a cargo do protocolo de comunicação decidir qual a melhor rede a ser utilizada.

O programador pode expressar, nas chamadas *send* e *receive*, uma eventual preferência na tecnologia a ser usada para a comunicação. Essa preferência é levada em consideração na hora da tomada de decisão sobre qual rede será utilizada. Outras informações consideradas são as tecnologias de rede disponíveis nos nós onde rodam os processos envolvidos na comunicação e o tamanho da mensagem.

Sincronização por Barreiras

Técnicas de sincronização (semáforos, monitores, barreiras entre outros) são geralmente utilizadas na programação paralela para máquinas com memória comum, sendo que os processos paralelos interagem através de memória compartilhada. Todavia, as máquinas sem memória comum necessitam de um mecanismo de sincronização determinado por um modelo de programa paralelo: processos comunicantes ou cliente/servidor.

O MDX-cc implementa um mecanismo de barreiras de sincronização. As barreiras são gerenciadas por um servidor especializado, o Servidor de Sincronização. Podem ser criadas e executadas barreiras a fim de sincronizar um grupo de processos.

Memória Compartilhada Distribuída

Seguindo o projeto inicial do Sistema MDX, que prevê também o paradigma de memória compartilhada, um módulo de memória compartilhada deve oferecer primitivas de criação, escrita, leitura e remoção de variáveis compartilhadas. Em *clusters* com rede SCI, os mecanismos de memória compartilhada do *hardware* SCI devem ser utilizados. Para agregados com as outras tecnologias de rede, deve haver uma abstração, por meio de um servidor de memória compartilhada distribuída. Este serviço não foi implementado nesta primeira versão do MDX-cc.

Balanceamento de Carga

Visando evitar a sobrecarga de processos em nós na execução de um programa paralelo, um serviço de balanceamento de carga deve avaliar o estado atual de carga de cada um dos nós no momento do disparo de cada processo. Desta forma, deve evitar que um determinado nó receba processos mais pesados que outros, tornando aquele nó um ponto de degradação de desempenho. Este serviço não foi implementado. O MDX-cc usa um modelo de *round-robin* simples no disparo dos processos nos nós envolvidos no sistema. Um arquivo de nomes de máquinas armazena os nós onde os processos devem ser disparados. No caso do arquivo apresentado na Figura 5.2, é disparado um processo em cada um dos seis nós, começando pelo nó "amazonia01", seguindo a ordem apresentada no arquivo, até o nó "tropical02". Caso haja mais processos do que nós disponíveis, o disparo recomeça no nó "amazonia01" e assim segue até que o número total de processos tenha sido disparado.

```
amazonia01.cpad.pucrs.br
amazonia02.cpad.pucrs.br
amazonia03.cpad.pucrs.br
amazonia04.cpad.pucrs.br
tropical01.cpad.pucrs.br
tropical02.cpad.pucrs.br
```

Figura 5.2: Arquivo de nomes de máquinas do MDX-cc.

Comunicação em Grupo

Um mecanismo interessante em programas paralelos é o de criação de grupos de processos. O MDX-cc deve prover mecanismos de envio de mensagens para grupos de processos (*Multicast*) e envio de mensagens a todos os processos da aplicação (*Broadcast*). Também

deve oferecer rotinas de recebimento genérico, ou seja, sem especificar a origem da mensagem. Este serviço não foi implementado. O MDX-cc oferece somente mecanismos de comunicação ponto-a-ponto entre processos, onde deve ser informado a origem (no caso de um *receive*) ou o destino (no caso de uma primitiva *send*) da mensagem.

5.3. Alterações MDX – MDX-cc

Para que o MDX execute sobre uma estrutura de *cluster* de *clusters*, alterações na estrutura do sistema se fizeram necessárias. Esta seção apresenta as principais modificações realizadas.

5.3.1. Eliminação do Núcleo de Comunicação

O MDX, em sua implementação MDX-v1, possui um Núcleo de Comunicação, que executa em todos os nós do ambiente, e é responsável única e exclusivamente por criar uma área de memória compartilhada, para onde é carregada a Tabela Local de Nomes (NLT). A Figura 5.3 ilustra a estrutura do MDX.

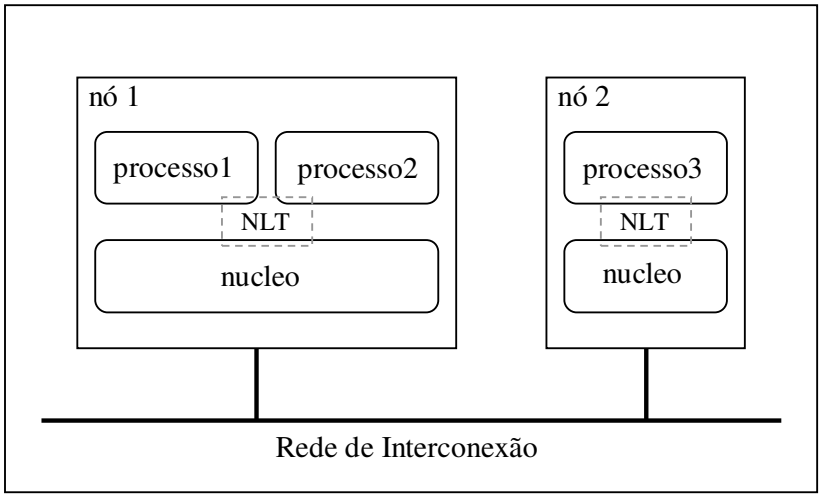

Figura 5.3: Estrutura do MDX.

A fim de dar uma maior autonomia aos processos para que gerenciem essa tabela, essa estrutura foi modificada. No MDX-cc, a figura do Núcleo de Comunicação deixa de existir, e cada processo gerencia a sua própria NLT. Desta forma, o MDX-cc fica mais enxuto, uma vez que um *daemon* do ambiente não fica mais executando em todos os nós do sistema. A Figura 5.4 ilustra a nova estrutura.

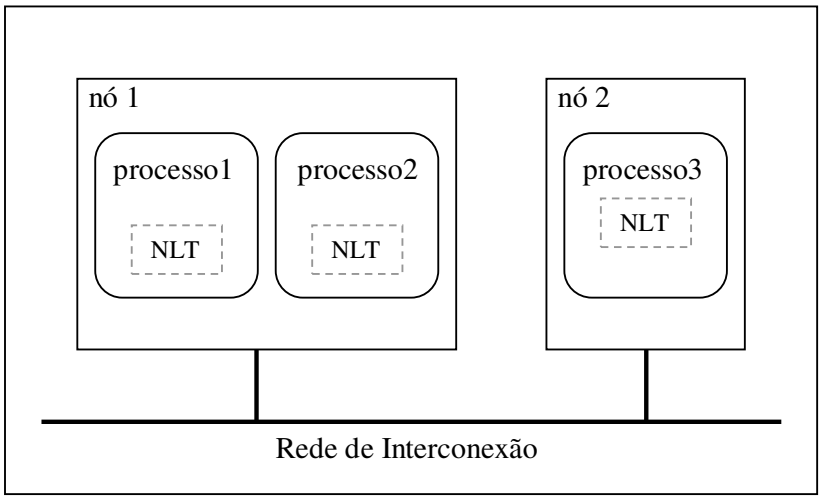

Figura 5.4: Nova estrutura no MDX-cc.

5.3.2. Infra-estrutura

O sistema MDX oferecia poucas ferramentas de infra-estrutura, como, por exemplo, aplicativos de configuração da ferramenta, programas que facilitem o início e o término do ambiente, *scripts* de disparo de aplicações e ferramenta de compilação de programas.

Para o MDX-cc, foram implementados utilitários que auxiliam na programação e na execução de aplicações. Para a configuração do ambiente, foi implementado o ***mdxconf*** (seção 5.4.1). Para a compilação de programas, a ferramenta desenvolvida é o ***mdxcc*** (seção 5.4.6), e, para início e término de aplicações, as ferramentas implementadas são o ***mdxrun*** (seção 5.4.6) e o ***mdxshutdown*** (seção 5.4.6).

5.3.3. Programação SPMD

No MDX-v1, tarefas eram criadas dinamicamente, com a primitiva *MDX_Task_Create( )*. Em uma estrutura de *cluster* de *clusters*, o número de tarefas criadas dinamicamente pode ser muito alto, dado o aumento dos recursos computacionais disponíveis.

A Figura 5.5 ilustra a criação de tarefas no MDX. No exemplo apresentado, o processo1, que executa no nó 1, cria o processo2, no nó 2, através de uma chamada à primitiva *MDX_Task_Create( )*.

Para facilitar a programação sobre *cluster* de *clusters*, o MDX-cc oferece suporte à programação SPMD. Agora, a aplicação é distribuída nos nós disponíveis, e cada processo, ao se registrar no Servidor de Comunicação, recebe, na variável *MDX_id* um identificador único, e, na variável *MDX_np* o número de processos disparados.

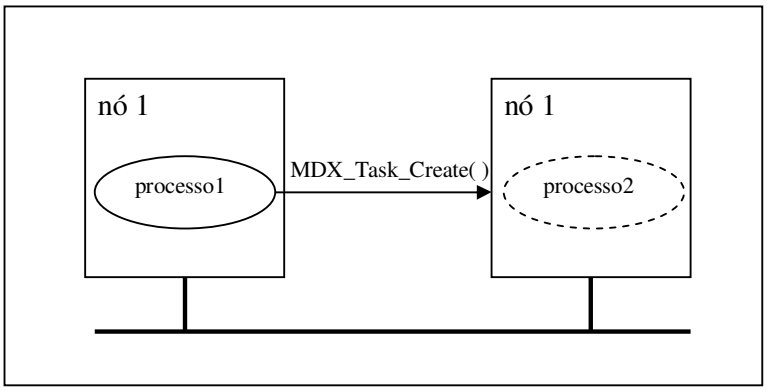

Figura 5.5: Criação de tarefas no MDX.

5.3.4. Suporte a outras Tecnologias de Rede

Até a sua versão MDX-v1, o MDX oferecia suporte às tecnologias *Fast-Ethernet*, por protocolo UDP, e ATM. As tecnologias mais usadas atualmente na construção de agregados são *Fast-Ethernet*, *Myrinet* e SCI [SQU00]. Nesse contexto, para o MDX-cc, foi deixada de fora a implementação com suporte a rede ATM e foi implementado versões com suporte às tecnologias *Fast-Ethernet* TCP, *Myrinet* e SCI. Com a implementação dessas versões, o MDX passa a oferecer suporte à comunicação por rede *Fast-Ethernet* (TCP e UDP), *Myrinet* e SCI.

5.3.5. Tabela Local de Nomes

No MDX, a estrutura de armazenamento de informações de servidores e de processos era única e muito simples, a NLT. Ela armazenava informações sobre a localização dos servidores, bem como as informações para comunicação com os processos da aplicação (identificador, endereço do nó onde executa e porta de comunicação). A Figura 5.6 ilustra essa estrutura.

Com a implementação do suporte a outras tecnologias de rede, há um aumento no número de informações necessárias para comunicação entre os processos. Isto torna impossível o armazenamento das informações dos processos na NLT. Por isso, essa estrutura foi modificada completamente e dividida em duas: NLT e *LocalClientTable*. A NLT foi simplificada e armazena informações somente sobre a localização dos servidores do ambiente (seção 5.4.1). A *LocalClientTable* é uma estrutura mais complexa e armazena informações sobre os processos com os quais há comunicação (seção 6.2.1). A nova estrutura é ilustrada pela Figura 5.7.

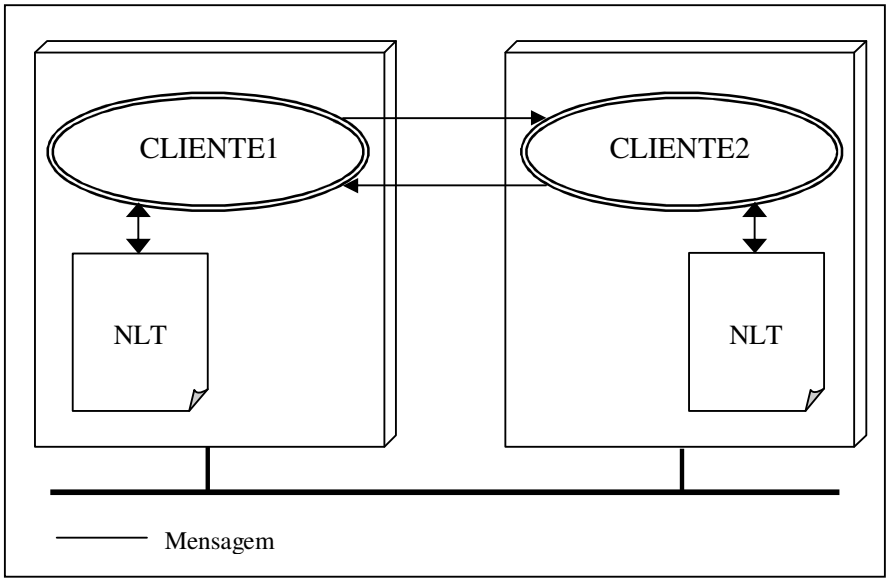

Figura 5.6: NLT no MDX.

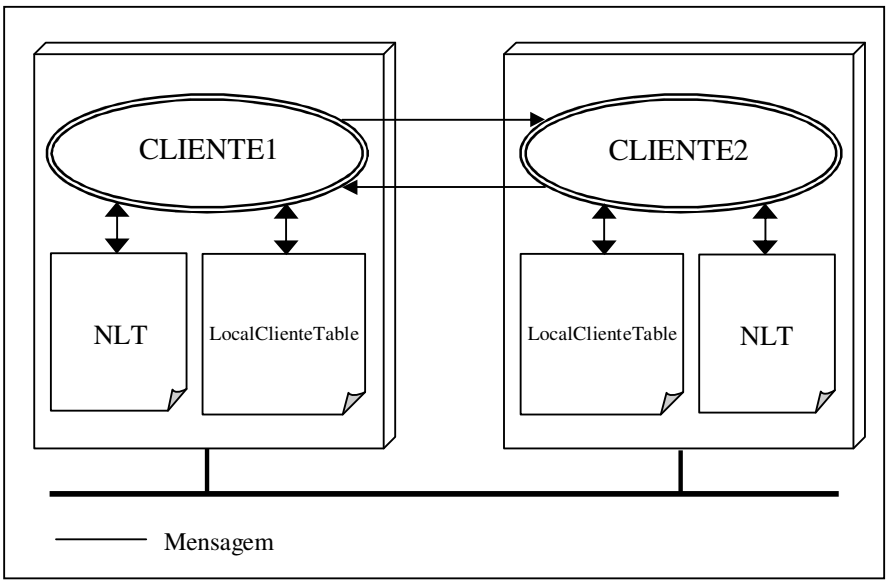

Figura 5.7: NLT no MDX-cc.

5.3.6. Simplificação ou Eliminação de Serviços

Um dos objetivos do MDX-cc é oferecer ao usuário uma interface de programação simples. A fim de atingir este objetivo, alguns mecanismos oferecidos pelo MDX foram modificados.

No MDX, o mecanismo de comunicação é o de portas, onde uma mensagem é enviada a uma determinada porta e consumida da mesma. Esse mecanismo apresentou um péssimo

desempenho [HES01], sendo por isso deixado de fora dos mecanismos de comunicação do MDX-cc. Para a comunicação direta, o número de primitivas foi reduzido a dois, e essas primitivas são apresentadas na seção 5.4.4.3.

O uso de barreiras de sincronização também foi simplificado. As primitivas de sincronização foram reduzidas a duas (seção 5.4.4.2) e são de mais fácil utilização.

Com essa simplificação e eliminação de serviços, alguns servidores especializados do MDX também ficaram de fora do MDX-cc.

Além dessas alterações, uma série de avaliações de desempenho e de otimizações foram realizadas no decorrer do trabalho. Essas otimizações e avaliações são apresentadas em [HES01].

5.4. Arquitetura do MDX-cc

A arquitetura do MDX-cc é baseada no modelo cliente-servidor, onde processos servidores atendem requisições de processos clientes. Esses processos clientes são partes da aplicação paralela que executa sobre o ambiente.

A estrutura interna do MDX-cc é dividida em duas camadas. A camada inferior comporta a parte dependente do *hardware* e provê os mecanismos de comunicação através das tecnologias de rede suportadas pelo ambiente. A camada superior é composta pelos servidores especializados, que atendem as requisições dos processos clientes. Essa camada apresenta os seguintes servidores: **Servidor de Comunicação**, responsável por gerenciar as informações sobre todos os processos da aplicação, e o **Servidor de Sincronização**, responsável por atender requisições de criação e de execução de barreiras de sincronização. A Figura 5.8 ilustra a estrutura interna do MDX-cc.

O MDX-cc oferece suporte a comunicação através de redes *Fast-Ethernet*, *Myrinet* e SCI. Para a implementação dos mecanismos de comunicação sobre essas tecnologias, foram utilizadas as bibliotecas *sockets* para rede *Fast-Ethernet*, GM [GM01] para redes *Myrinet*, e YAMPI [NOV01] para rede SCI.

A Figura 5.9 ilustra a estrutura externa do MDX-cc, onde o ambiente é implementado sobre as bibliotecas de comunicação citadas anteriormente. A biblioteca de *sockets* executa sobre o sistema operacional, enquanto as bibliotecas YAMPI e GM acessam diretamente as

interfaces de rede SCI e *Myrinet*, respectivamente. Sobre o MDX-cc executam as aplicações paralelas.

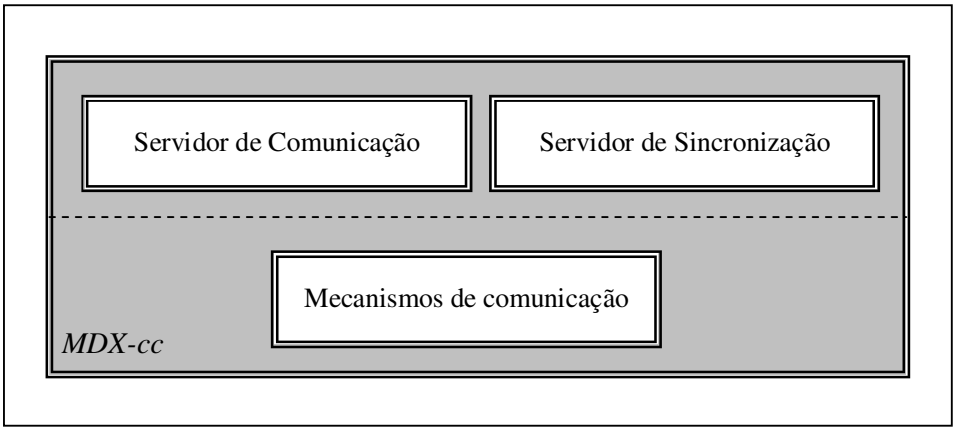

Figura 5.8: Estrutura interna do MDX-cc.

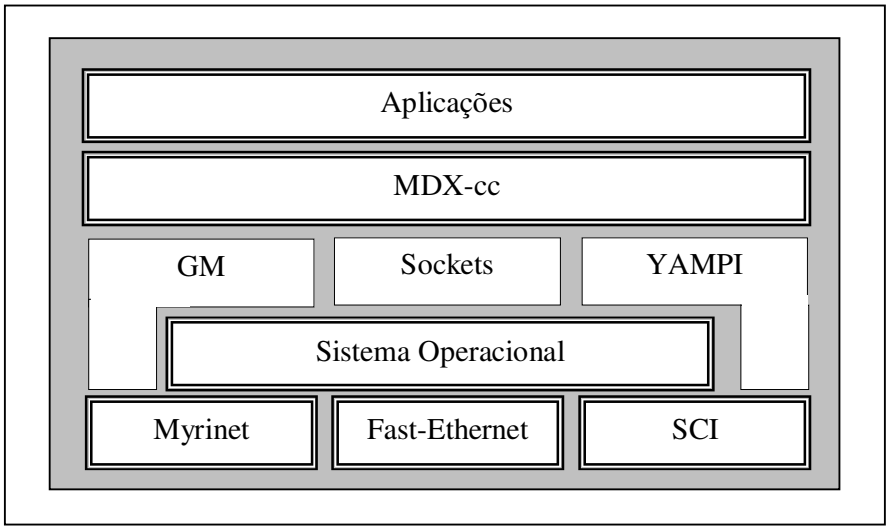

Figura 5.9: Estrutura externa do MDX-cc.

5.4.1. Tabela Local de Nomes (NLT)

O MDX-cc possui um arquivo de configuração (*nlt.conf*) que armazena a localização dos servidores do ambiente. Esse arquivo armazena apenas o identificador do servidor (sincronização ou comunicação) e o endereço IP do nó onde o mesmo deve executar (Tabela 5.1). Através da modificação desse arquivo, o programador pode escolher em quais nós do sistema serão executados os servidores de comunicação e sincronização do MDX-cc. Para

modificação desse arquivo, um aplicativo foi implementado, o ***mdxconf***. Esse aplicativo permite verificar a configuração atual do ambiente, bem como alterar essa configuração, de forma simples. A Figura 5.10 apresenta a tela inicial do ***mdxconf***, onde aparecem opções para configurar um novo ambiente, adicionar um servidor ao sistema ou listar a configuração atual do ambiente.

Tabela 5.1: Estrutura do arquivo *nlt.conf*.

Campo	***Informação***
serv_id	Identifica qual o servidor (comunicação ou sincronização)
serv_addr	Armazena o endereço IP do nó onde o servidor está executando

Cada processo de uma aplicação MDX-cc possui uma Tabela Local de Nomes, a NLT. Ao ser iniciado, cada processo carrega a sua NLT com os dados do arquivo de configuração *nlt.conf*. Essa tabela será usada na localização dos servidores no sistema, para o envio de requisições de serviços. Por exemplo, quando um processo cliente efetua uma chamada de criação de barreira de sincronização, a primitiva localiza o servidor de sincronização na NLT e envia uma requisição de criação de barreira.

A Tabela 5.2 apresenta os campos da NLT, bem como a informação armazenada por cada um deles. As informações armazenadas nos primeiros dois campos são carregadas do arquivo *nlt.conf* no início do processo. O campo *serv_socket* é preenchido no estabelecimento da conexão com o servidor, e armazena o *socket* através do qual o processo se comunica com o mesmo. A comunicação entre processos clientes e servidores é detalhada a seguir.

Tabela 5.2: Estrutura da NLT.

Campo	***Informação***
serv_id	Identifica qual o servidor (comunicação ou sincronização)
serv_addr	Armazena o endereço IP do nó onde o servidor está executando
serv_socket	Identificador do *socket* através do qual o processo se comunica com o servidor

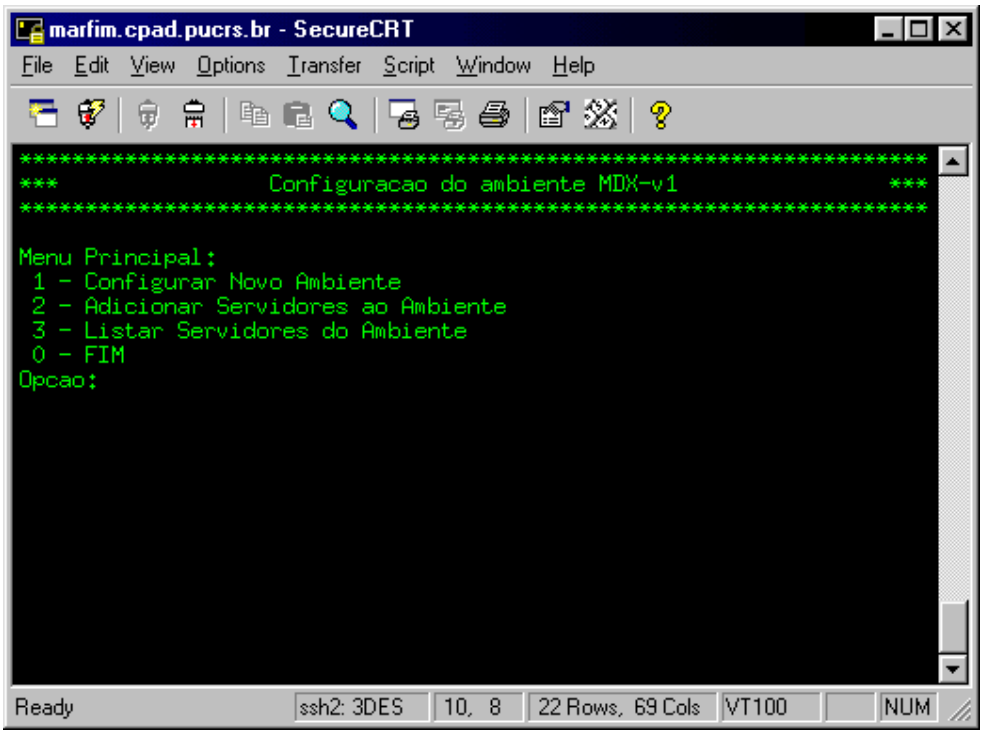

Figura 5.10: Tela inicial do *mdconf*.

5.4.2. Comunicação entre processos clientes e servidores

A comunicação entre processos clientes e servidores ocorre de forma implícita nas primitivas do MDX-cc. Quando uma primitiva (por exemplo, de execução de barreira de sincronização) necessita de algum serviço da camada de serviços do MDX-cc, ela localiza, na NLT, o servidor que atenderá a requisição, envia a solicitação ao mesmo e recebe a resposta.

A Figura 5.11 ilustra, de forma simplificada, o sistema de comunicação entre processos clientes e servidores. O processo cliente consulta a NLT, localiza o servidor, envia uma requisição e recebe uma resposta.

Uma seqüência de passos é necessária para a comunicação entre os processos clientes e os servidores do MDX-cc:

1- O servidor é localizado na NLT, através do seu identificador (Tabela 5.3);
2- Localizado o servidor, é verificado se já existe uma conexão estabelecida com o mesmo, através do valor do campo *serv_socket* da NLT. Se o valor desse campo é diferente de –1, significa que uma conexão TCP já foi estabelecida com o servidor e a comunicação ocorre através do *socket* identificado pelo valor do campo. Caso o valor

do campo seja –1, é necessário estabelecer uma conexão TCP, através de uma chamada *connect*, com o servidor, que aguarda conexões em uma chamada *accept*. O identificador do *socket* resultante dessa chamada *connect* é armazenado no campo *serv_socket* da NLT (Tabela 5.2).

3- Uma vez identificada a existência de uma conexão com o servidor, basta que uma chamada de envio de mensagem (*send*) seja efetuada, e o conteúdo do pacote enviado seja a requisição de serviço e os dados necessários para a execução do mesmo. Para receber uma resposta do servidor, basta que seja efetuada uma chamada de recebimento de mensagem (*recv*).

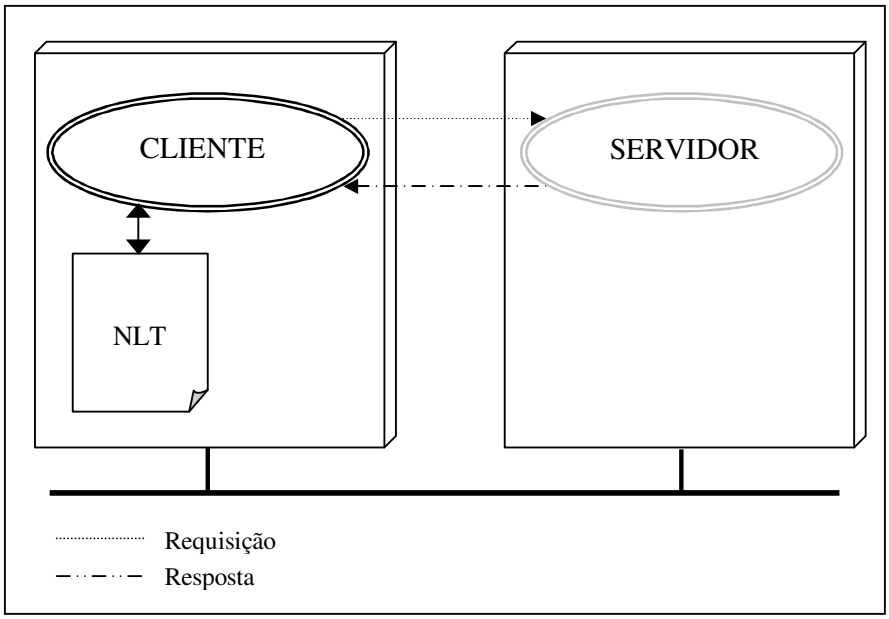

Figura 5.11: Comunicação entre clientes e servidores.

É importante ressaltar que a comunicação entre processos clientes e os servidores do MDX-cc sempre se dá através da rede *Fast-Ethernet*, de maneira a preservar a rede rápida (SCI ou *Myrinet*, quando utilizada) para a comunicação entre os processos da aplicação. A estrutura e o funcionamento dos servidores do MDX-cc são descritos de forma detalhada a seguir.

5.4.3. Servidores Especializados

No MDX-cc, cada servidor especializado foi implementado como um programa independente, que executa em algum dos nós do sistema, conforme configurado através do *mdxconf*. Quando

um processo cliente deseja se comunicar com um servidor, ele deve estabelecer uma conexão com o mesmo, através de uma chamada *connect*. Cada servidor do MDX-cc aguarda conexões, por uma chamada *accept*, em uma porta diferente, conforme mostra a Tabela 5.3.

Tabela 5.3: Identificadores e portas de comunicação dos servidores do MDX-cc.

Servidor	***Identificador do Servidor***	***Porta***
Servidor de Sincronização	SERVER_SYNC	3101
Servidor de Comunicação	SERVER_COMM	3105

Visando um maior paralelismo no atendimento de requisições de diferentes clientes, os servidores são programas *multithread*. No momento que um cliente estabelece conexão com o servidor, é criada uma nova *thread*. A partir de então, o *socket* através do qual o processo efetuou a conexão e a *thread* criada são dedicados exclusivamente à comunicação e ao atendimento de requisições daquele cliente. Assim, o cliente efetua a conexão com o servidor apenas uma vez, através de uma chamada *connect*, no endereço e porta de comunicação do servidor. O *socket* desta conexão é armazenado no campo *serv_socket* na NLT (Tabela 5.2) do processo cliente. O Algoritmo 5.1 mostra um esqueleto de código de um servidor.

A *thread* criada para atender a um cliente consiste em um processo que fica em *loop* aguardando mensagens do cliente, tratando as requisições e devolvendo respostas. O Algoritmo 5.2 mostra um esqueleto de código de uma *thread* que atende um determinado cliente.

Algoritmo 5.1: Esqueleto de código de um servidor.

```
main( )
{
      int socket;

       while(1)
       {
          //aguarda conexões de clientes
          socket = aguarda_conexao(PORTA);

          //cria a thread que implementa o servidor
          cria_thread_serv(socket);
       }
}
```

Algoritmo 5.2: Esqueleto de código de uma *thread* dedicada.

```
servidor( int socket )
{
       MDX_req *req;
       MDX_rep *rep;

       while(1)
       {
          //aguarda conexões de clientes
          recv(socket, req, 0);

          // trata a requisicao
          if (req.service_id == SERVICO_1) cria_thread_servico_1(&rep);
          else
          if (req.service_id == SERVICO_2) cria_thread_servico_2(&rep);
          else
          if (req.service_id == SHUTDOWN) break( );
          // envia resposta ao cliente
          send (socket, rep, 0);
       }
       close(socket);
}
```

5.4.3.1. Servidor de Sincronização

O Servidor de Sincronização é um servidor do MDX-cc que é disparado automaticamente no início da aplicação paralela e executa em apenas um nó do ambiente. A função do Servidor de Sincronização é o gerenciamento dos mecanismos de sincronização do ambiente.

A sincronização de processos é realizada através de barreiras. Estas são implementadas como uma lista encadeada simples, a *BarrierList*. Barreira é uma estrutura que contém quatro campos: *Identificador*, *Quorum*, *Count* e *ProcessList*, conforme mostra a Figura 5.12.

O *Identificador* de uma barreira é uma *string* que contém o nome da mesma. O campo *Quorum* expressa o número de processos que sincronizam através da barreira e o campo *Count* tem como objetivo a verificação do número de processos bloqueados, isto é, que já executaram a barreira. Cada vez que um processo executa a barreira este campo é incrementado. Quando seu valor atingir o *Quorum* todos os processos bloqueados são liberados. A *ProcessList* de uma barreira é uma lista que contém todos os processos bloqueados. A estrutura que armazena informações dos processos é composta por dois campos: *ProcId*, que armazena o identificador do processo, e *ProcSocket*, que armazena o identificador do *socket* de comunicação do servidor com o processo.

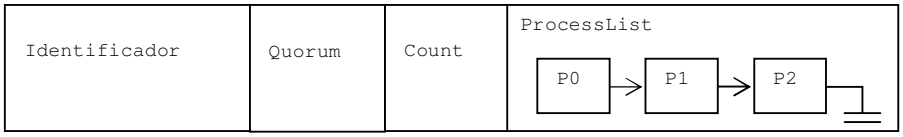

Figura 5.12: Barreira de sincronização.

Quando o servidor recebe uma mensagem, este identifica e trata o serviço solicitado. Quando um serviço de criação de barreira (*BarrierCreate*) é identificado, o Servidor de Sincronização dispara a função *BarrierCreate* para tratá-lo. Esta função cria uma nova barreira na sua lista de barreiras e atribui a mesma o *Identificador* e o *Quorum* recebidos. Em seguida, esta devolve uma mensagem contendo um indicativo de sucesso. Caso a criação não tenha ocorrido, esta devolve um indicativo de insucesso. Após o envio da mensagem de resposta, encerra-se a função.

Quando um serviço de execução de barreira (*Barrier*) é identificado, o Servidor de Sincronização dispara a execução da função *Barrier* para tratá-lo. Esta inclui o processo (*identificador* e *socket* de comunicação) na lista de processos (*ProcessList*) da barreira, e incrementa o campo *Count*. Em seguida, verifica-se se o processo foi incluído na *ProcessList* com sucesso. Se a inclusão não ocorreu, um indicativo de insucesso é enviando ao cliente. Caso contrário, verifica-se se o *Quorum* foi alcançado (*Count* = *Quorum*). Se isto ocorrer, envia-se um indicativo de sucesso para todos os processos contidos na *ProcessList*, desbloqueando os mesmos. Ao final desta operação, encerra-se a execução da *função*.

Existem duas barreiras criadas quando o servidor é disparado, que são *InitBarrier* e *FinalBarrier*. Essas duas barreiras servem para sincronizar o início e o fim de cada processo.

5.4.3.2. Servidor de Comunicação

O Servidor de Comunicação é um servidor do MDX-cc que é disparado automaticamente no início da aplicação paralela e executa em apenas um nó do ambiente. A principal função do Servidor de Comunicação é o gerenciamento das informações de todos os processos da aplicação paralela. Esse servidor também é responsável por atribuir um identificador único a cada processo.

O armazenamento das informações dos processos é feito em uma tabela, chamada de *ClientTable*. Essa tabela é implementada como um vetor de *np* posições, onde *np* é o número de processos da aplicação paralela. Cada posição desse vetor armazena as informações de um processo, cujo identificador é dado pela posição do vetor. Por exemplo, as informações do

processo 15 estão localizadas na posição 15 da *ClientTable*. A estrutura da *ClientTable* é ilustrada na Figura 5.13.

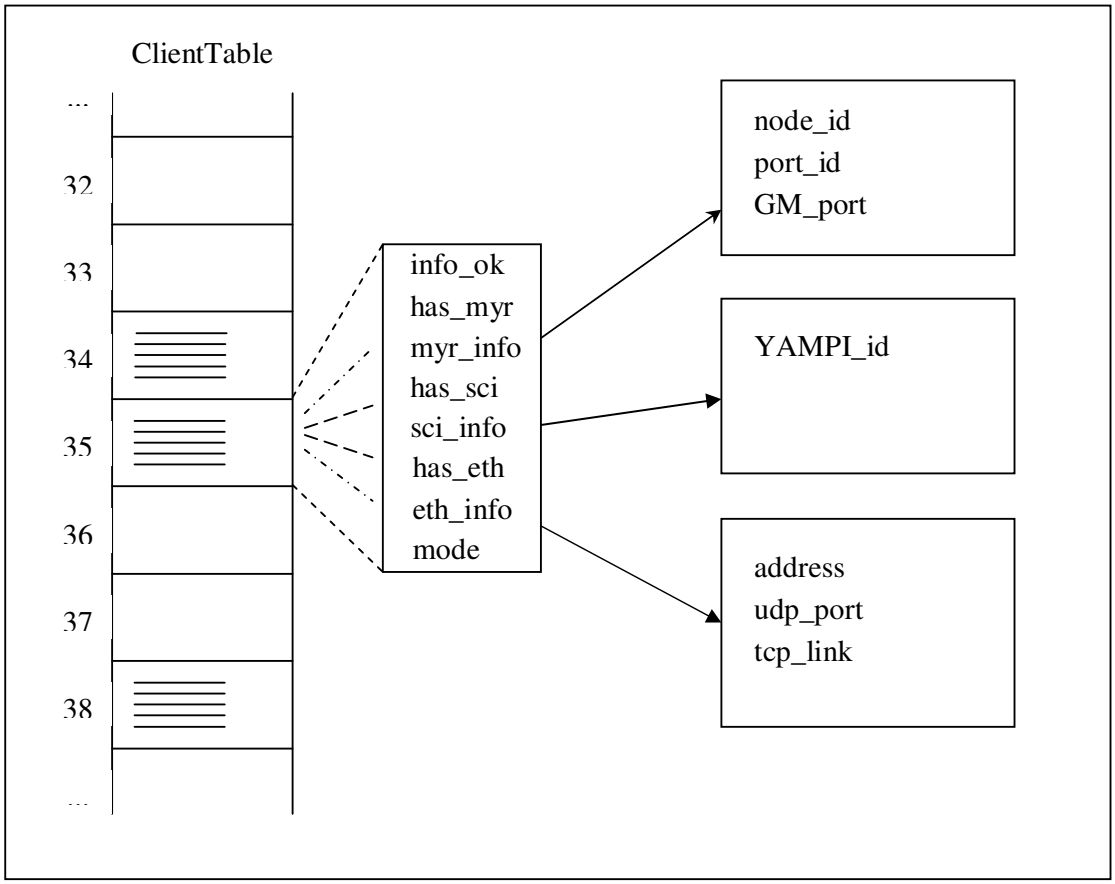

Figura 5.13: Estrutura da *ClientTable*.

Sobre um processo são armazenados os seguintes campos:

- info_ok: indica se as informações do processo estão preenchidas
- has_MYR: informa se o processo pode comunicar por rede *Myrinet*
- myr_info: ponteiro para uma estrutura que armazena informações para a comunicação com o processo por tecnologia *Myrinet*. Essa estrutura contém os seguintes campos
    - node_id: identificador do nó, atribuído pela biblioteca GM
    - port_id: porta de comunicação GM, criada pelo processo
    - GM_port: estrutura usada pela biblioteca GM para comunicar com outros processos
- has_SCI: indica se o processo pode comunicar através da tecnologia SCI

- sci_info: ponteiro para uma estrutura que armazena informações para comunicação com o processo através da tecnologia SCI. Essa estrutura possui os seguintes campos:
    - YAMPI_id: identificador do processo, usado para comunicação pela biblioteca YAMPI
- has_ETH: informa se o processo pode comunicar através de rede *Fast-Ethernet*
- eth_info: ponteiro para uma estrutura que armazena informações para comunicação com o processo através da tecnologia *Fast-Ethernet*. Essa estrutura possui os seguintes campos:
    - address: endereço IP do nó onde o processo está executando
    - udp_port: porta usada para comunicação com o processo através do protocolo UDP
    - tcp_link: identificador do *socket* usado para comunicação com o processo através do protocolo TCP.
- mode: indica qual a tecnologia usada na última comunicação com o processo

Quando o servidor recebe uma mensagem, este identifica e trata o serviço solicitado. O Servidor de Comunicação atende a dois serviços do MDX-cc: *RegisterClient* e *LookupClient*. O primeiro é o serviço de registro de informações no servidor. Cada processo MDX-cc registra suas informações no Servidor de Comunicação. Essas informações serão recuperadas por outros processos para futura comunicação. O segundo serviço é o de recuperação de informações de um processo.

Ao ser identificada uma requisição de *RegisterClient*, o servidor dispara a função *AddClient*, passando as informações do processo, recebidas junto com a requisição. Essa função gera um identificador *i*, único para o processo, e armazena as informações recebidas na posição *i* da *ClientTable*. Em seguida, o identificador gerado é enviado ao processo requisitante, bem como o número total de processos da aplicação. Caso o processo não tenha sido registrado na tabela de clientes, um indicativo de erro é enviado como resposta.

Quando uma requisição de *LookupClient* é identificada, o servidor dispara a função *LookupClient*, passando o identificador do processo, que foi enviado com a requisição Essa função localiza as informações do processo, que estão na posição *i* da *ClientTable*, e envia essas informações ao processo requisitante. Caso as informações do processo não estejam registradas na tabela, um indicativo de erro é enviado como resposta.

5.4.4. Interface de programação MDX-cc

O MDX-cc, em sua concepção, prevê um conjunto reduzido de primitivas. Isso proporcionou a implementação de uma interface de programação bastante simples. Esta seção apresenta as primitivas do MDX-cc, dividindo-as em Primitivas de Controle, Primitivas de Sincronização e Primitivas de Comunicação.

5.4.4.1. Primitivas de Controle

As primitivas de controle são usadas para inicio é término de programas MDX-cc. Essas primitivas são apresentadas pela Figura 5.14.

A primitiva *MDX_Init( )* deve ser chamada no início de um programa MDX-cc. Essa primitiva, inicialmente, chama a função *MDX_EnvDetect( )*, para detecção das tecnologias de rede disponíveis no nó onde o processo executa. Essa função detecta, através dos arquivos */proc/pci* e */proc/modules* do Linux, quais as interfaces de rede que estão instaladas no nó - *Fast-Ethernet* (obrigatório) e SCI e/ou *Myrinet*.

```
int MDX_Init();
int MDX_Finalize();
```

Figura 5.14: Primitivas de controle do ambiente.

Uma vez detectadas as interfaces de rede disponíveis, o processo pode iniciar os ambientes através dos quais ele poderá comunicar. Se a tecnologia SCI está disponível no nó, é chamada a função *YAMPI_Init( )*, que é a primeira função a ser chamada em uma aplicação YAMPI. Caso a tecnologia *Myrinet* tenha sido detectada, são chamadas duas funções: a função *gm_init( )*, que inicia um processo GM, e a função *gm_open( )*, que serve para abrir uma porta de comunicação GM, estrutura que será usada para enviar e receber mensagens via rede *Myrinet*. Para a comunicação por rede *Fast-Ethernet* são criados dois *sockets*, um para comunicação por protocolo TCP e outro para comunicação por protocolo UDP.

Depois de iniciados os ambientes através dos quais o processo poderá comunicar, as informações devem ser registradas no Servidor de Comunicação. Para isso, é chamada a função *MDX_RegisterClient( )*. Esta função envia uma requisição de serviço *RegisterClient* ao Servidor de Comunicação, enviando também as informações sobre o processo e sobre as tecnologias de rede detectadas. Como resposta, em caso de sucesso no registro das informações, o processo recebe o seu identificador e o número de processos disparados na

aplicação. O identificador é armazenado na variável *MDX_id*, e o número de processos na variável *MDX_np*.

Finalmente, o processo executa a barreira *InitBarrier*, que sincroniza o início de todos os processos da aplicação.

A outra primitiva de controle do MDX-cc é a *MDX_Finalize( )*. Essa primitiva executa a barreira *FinalBarrier*, a fim de sincronizar o término de todos os processos da aplicação. Em seguida, são finalizados os ambientes de programação iniciados em *MDX_Init( )*. Caso o ambiente YAMPI tenha sido iniciado, é chamada a função *YAMPI_Finalize( )*. Caso o ambiente GM tenha sido iniciado, é chamada a função *gm_finalize( )*. Finalmente, o processo é encerrado.

5.4.4.2. Primitivas de Sincronização

A sincronização de processos no MDX-cc é feita através do mecanismo de barreiras, gerenciado pelo Servidor se Sincronização. São oferecidas ao usuário duas primitivas de sincronização, ilustradas na Figura 5.15.

```
int MDX_BarrierCreate(char *Identificador, int Quorum);
int MDX_Barrier(char *Identificador);
```

Figura 5.15: Primitivas de sincronização do ambiente.

A primitiva *MDX_BarrierCreate( )* envia ao Servidor de Sincronização uma requisição de criação de barreira (*BarrierCreate*), enviando também o *Identificador* da barreira e o *Quorum*, que indica o número de processos que sincronizam através da barreira. O servidor envia como resposta um indicativo de sucesso ou de insucesso na operação.

A outra primitiva, *MDX_Barrier( )*, envia uma requisição de execução de barreira (*Barrier*) ao Servidor de Sincronização, enviando também o *Identificador* da barreira a ser executada. Em seguida, A primitiva fica bloqueada aguardando a resposta do servidor. Essa resposta indica que todos os processos já executaram a barreira e podem seguir sua execução.

5.4.4.3. Primitivas de Comunicação

A comunicação entre processos no MDX-cc é feita através de duas primitivas: *MDX_Send( )* e *MDX_Recv( )* (Figura 5.16).

```
int MDX_Send(int id_dest, char *msg, int tam, int mode)
int MDX_Recv(int id_from, char *msg, int tam, int mode)
```

Figura 5.16: Primitivas de comunicação do ambiente.

A primitiva *MDX_Send( )* envia a mensagem contida em *msg*, de tamanho *tam*, ao processo *id_dest*. O campo *mode* indica uma preferência do usuário sobre a tecnologia de rede empregada na comunicação (*Fast-Ethernet*, *Myrinet* ou SCI). As alternativas são:

- MDX_USE_TCP: dar preferência à utilização de rede *Fast-Ethernet*, com protocolo TCP.
- MDX_USE_UDP: dar preferência à utilização de rede *Fast-Ethernet*, com protocolo UDP (apenas para mensagens menores que 65507 bytes)
- MDX_USE_MYR: dar preferência à utilização de rede *Myrinet*.
- MDX_USE_SCI: dar preferência à utilização de rede SCI.
- MDX_USE_ANY: indica que o usuário não quer dar preferência a nenhuma das tecnologias.

A primitiva *MDX_Recv( )* recebe uma mensagem de tamanho *tam*, proveniente do processo *id_from*. A mensagem é armazenada em *msg*. O campo *mode* indica uma preferência do usuário sobre a tecnologia de rede empregada na comunicação. As alternativas são as mesmas da primitiva *MDX_Send( )*.

As primitivas de comunicação devem oferecer transparência ao usuário sobre qual a tecnologia de rede é utilizada na comunicação. Para tanto, deve haver uma análise das alternativas e uma decisão sobre a rede usada. Este protocolo de comunicação será detalhado no Capítulo 6.

5.4.5. Exemplos de Programas

Nesta seção serão apresentados alguns exemplos simples de programas MDX-cc, que mostram a utilização das primitivas apresentadas na seção anterior.

O programa apresentado pelo Algoritmo 5.3 é uma aplicação bem simples, onde uma mensagem de *hello* é enviada do processo zero ao processo com identificador 1. Inicialmente, é definida uma variável, *buffer*, de 30 caracteres, que armazenará a mensagem. A primeira

primitiva MDX-cc chamada é a *MDX_Init( )*, a fim de iniciar o processo e receber, em *MDX_id* e *MDX_np*, o seu identificador único e o número de processos da aplicação. Em seguida, é colocada em *buffer* o conteúdo da mensagem. O processo com identificador zero envia a mensagem (*buffer*), com tamanho 30, para o processo com identificador 1, através da primitiva *MDX_Send( )*.

O processo com identificador 1 chama a primitiva *MDX_Recv( )*, passando como remetente o processo zero. A mensagem é armazenada na variável *buffer*, com 30 caracteres. Recebida a mensagem, o processo 1 a imprime na tela.

Para finalizar a aplicação, é chamada a primitiva *MDX_Finalize( )*, que sincroniza os processos e termina a aplicação.

Algoritmo 5.3: Exemplo de *helloworld* no MDX-cc.

```
main(int argc, char **argv)
{
    char buffer[30];

    MDX_Init(); //inicio de um processo MDX-cc

    if(MDX_id == 0) // testa se é o mestre
    {
        strcpy(buffer, "hellow process 1");
        MDX_Send(1, buffer, 30, MDX_USE_TCP); //envia ao prox 1
    }
    else
    {
        MDX_Recv(0, buffer, 30, MDX_USE_TCP); //recebe do proc 0
        printf("%s\n", buffer);
    }

    MDX_Finalize();
}
```

Nessa aplicação, em todas as chamadas *MDX_Send( )* e *MDX_Recv( )* está indicada a preferência do usuário pela utilização de rede *Fast-Ethernet*, com protocolo TCP, na comunicação, através do parâmetro MDX_USE_TCP.

O programa apresentado pelo Algoritmo 5.4 é uma aplicação de *pipeline*, onde uma mensagem é gerada em um processo, e essa mensagem passa por todos os outros processos, até ser enviada novamente ao processo que a gerou. Inicialmente, é definida uma variável, *buffer*, de 1024 caracteres, que armazenará a mensagem. A primeira primitiva MDX-cc chamada é a *MDX_Init( )*, a fim de iniciar o processo e receber, em *MDX_id* e *MDX_np*, o seu identificador único e o número de processos da aplicação. Em seguida, é atribuído à variável

prox o identificador do processo ao qual a mensagem deverá ser passada. Uma vez que o processo tem seu identificador, pode ser iniciado o processo de envio da mensagem. O processo "mestre", com identificador zero, passa a mensagem (*buffer*), com tamanho 1024, para o processo com identificador 1, através da primitiva *MDX_Send( )*. Concluído o envio, o processo "mestre" aguarda, em uma chamada *MDX_Recv( )*, o retorno da mensagem, que será enviada pelo último processo, ou seja, com maior identificador (*MDX_np –1*).

Algoritmo 5.4: Exemplo de *pipeline* no MDX-cc.

```
main(int argc, char **argv)
{
    int it = 0, prox, ant, j;
    char buffer[1024];

    MDX_Init(); //inicio de um processo MDX-cc
    prox = (MDX_id+1)%MDX_np; //proximo processo a receber a mensagem

    if(MDX_id == 0) // testa se é o mestre
    {
      for(j = 0; j<10; j++)
      {
        MDX_Send(prox, buffer, 1024, MDX_USE_MYR); //envia ao prox
        MDX_Recv(MDX_np-1, buffer, 1024, MDX_USE_MYR);//recebe do ultimo
       }
    }
    else
    {
      for(j = 0; j<10; j++)
      {
        MDX_Recv(MDX_id-1, buffer, 1024, MDX_USE_MYR); //recebe do proc. anterior
        MDX_Send(prox, buffer, 1024, MDX_USE_MYR); //envia ao prox
       }
    }
    MDX_Finalize();
}
```

Os processos com identificador diferente de zero recebem a mensagem do processo anterior e a repassam ao próximo. Para isso, é chamada a primitiva *MDX_Recv( )*, indicando como remetente o processo anterior, ou *MDX_id-1*. A mensagem é armazenada na variável *buffer*, com 1024 caracteres. Recebida a mensagem, o processo deve repassá-la ao próximo. Isso é feito através de uma chamada *MDX_Send( )*, passando como identificador do processo destino a variável *prox*, o tamanho da mensagem, 1024, e variável *buffer*, que armazena a mensagem.

Para finalizar, é chamada a primitiva *MDX_Finalize( )*, que sincroniza os processos e termina a aplicação.

Nessa aplicação, em todas as chamadas *MDX_Send( )* e *MDX_Recv( )* está indicada a preferência do usuário pela utilização de rede *Myrinet* na comunicação, através do parâmetro MDX_USE_MYR.

O programa apresentado pelo Algoritmo 5.5 é uma aplicação de ordenação de vetor, onde um vetor é gerado em um processo, e esse vetor sofre uma reordenação em cada um dos outros processos, em sistema de *pipeline*, até ser enviado, totalmente ordenado, novamente ao processo que a gerou. O vetor gerado possui *np* posições, onde *np* é o número de processos da aplicação. A primeira primitiva MDX-cc chamada é a *MDX_Init( )*, a fim de iniciar o processo e receber, em *MDX_id* e *MDX_np*, o seu identificador único e o número de processos da aplicação. Em seguida, pode ser iniciado o processo de ordenação. O processo "mestre", com identificador zero, gera o vetor com números aleatórios, através da chamada *gera_vetor( )* e passa o vetor gerado para o processo com identificador 1, através da primitiva *MDX_Send( )*. Concluído o envio, o processo "mestre" aguarda, em uma chamada *MDX_Recv( )*, o retorno do vetor, que será enviada pelo último processo, ou seja, com maior identificador (*MDX_np –1*). Ao receber o vetor, este é mostrado na tela através da chamada *mostra_vetor( )*.

Os processos com identificador diferente de zero recebem o vetor do processo anterior, executam um processo de ordenação, e o repassam ao próximo. Para isso, é chamada a primitiva *MDX_Recv( )*, indicando como remetente o processo anterior, ou *MDX_id-1*. O vetor é armazenado na variável *buffer*. Recebido o vetor, o processo deve ordená-lo e passá-lo ao próximo processo. Isso é feito através de uma chamada à função *ordena_vetor( )*, seguida de uma chamada à primitiva *MDX_Send( )*, à qual é passado como identificador do processo destino a expressão *(MDX_id+1)%MDX_np*, o tamanho da mensagem, e a variável *buffer*, que armazena o vetor.

Para finalizar, é chamada a primitiva *MDX_Finalize( )*, que sincroniza os processos e termina a aplicação.

Nessa aplicação, em todas as chamadas *MDX_Send( )* e *MDX_Recv( )* está indicado que o usuário não tem preferência pela utilização de qualquer tecnologia de rede na comunicação. Essa opção é expressa no parâmetro MDX_USE_ANY.

Algoritmo 5.5: Exemplo de ordenação de vetor em paralelo no MDX-cc.

```
main(int argc, char **argv)
{
  int it = 0, aux, prox, ant, i = 0, *buffer;

  MDX_Init();
  prox = (MDX_id+1)%MDX_np;
  buffer = malloc(MDX_np*sizeof(int)); //vetor de np posicoes

  if(MDX_id == 0) // testa se é o mestre
  {
    gera_vetor(MDX_np, buffer);
    MDX_Send((MDX_id+1) % MDX_np, buffer, MDX_np * sizeof(int), MDX_USE_ANY);
    MDX_Recv(MDX_np-1, buffer, MDX_np * sizeof(int), MDX_USE_ANY);
    mostra_vetor(MDX_np, buffer);
  }
  else
  {
    MDX_Recv(MDX_id-1, buffer, MDX_np * sizeof(int), MDX_USE_ANY);
    ordena_vetor(MDX_np, buffer);
    MDX_Send((MDX_id+1) % MDX_np, buffer, MDX_np * sizeof(int), MDX_USE_ANY);
  }
  MDX_Finalize();
}
```

5.4.6. Compilação e Execução de Aplicações

Para compilar programas MDX-cc, deve ser chamado o aplicativo ***mdxcc*** (Figura 5.17). Esse aplicativo compila o código-fonte *nome_prog.c*, passando os parâmetros e incluindo as bibliotecas necessárias ao compilador *gcc* para geração do executável de nome *nome_prog*.

```
> mdxcc nome_prog
```

Figura 5.17: Aplicativo de compilação de aplicações paralelas no MDX-cc.

Para executar aplicações em uma máquina paralela, o MDX-cc oferece um aplicativo, o ***mdxrun*** (Figura 5.18). Esse aplicativo dispara, no modelo SPMD, *np* cópias do programa *nome_prog* nos nós disponíveis, que se encontram no arquivo de máquinas (Figura 5.2).

```
> mdxrun nome_prog np
```

Figura 5.18: Aplicativo de disparo de aplicações paralelas no MDX-cc.

O ***mdxrun*** dispara os servidores do ambiente conforme configuração feita pelo aplicativo ***mdxconf*** (seção 5.4.1). Em seguida, dispara as cópias da aplicação nos nós disponíveis, presentes no arquivo de máquinas.

Após o término da aplicação, os servidores ficam ativos no ambiente, para futuras execuções de programas. Para terminar a execução dos mesmos, deve ser chamado o aplicativo ***mdxshutdown*** (Figura 5.19). Esse aplicativo localiza os servidores e envia uma mensagem de encerramento aos mesmos. Os servidores, ao receberem a mensagem, terminam sua execução.

```
> mdxshutdown
```

Figura 5.19: Aplicativo de término dos servidores do ambiente MDX-cc.

5.5. Aspectos de *Hardware*

Para rodar o MDX-cc, alguns requisitos de *hardware* devem ser observados. É necessário no mínimo um agregado, e este agregado deve, no mínimo, ser interligado por rede *Fast-Ethernet*. Se o agregado utiliza como rede primária *Myrinet* ou SCI, ele deve utilizar rede secundária *Fast-Ethernet*. Esta foi escolhida como tecnologia comum a toda a estrutura por ser amplamente utilizada e de mais baixo custo em relação às demais tecnologias.

Uma estrutura de *cluster* de *clusters*, para o MDX-cc, consiste em um grande conjunto de nós interligados por rede *Fast-Ethernet*, onde existem subconjuntos de nós, que são os agregados, interligados por uma rede rápida.

A Figura 5.20 apresenta uma estrutura de *cluster* de *clusters* na visão do MDX-cc. Essa estrutura é composta por quatro agregados, sendo que dois deles utilizam rede primária *Myrinet*, um utiliza SCI e outro usa *Fast-Ethernet*. Todos os quatro agregados usam como rede secundária *Fast-Ethernet*, através da qual seus nós estão interligados com os nós dos outros *clusters*. O MDX-cc usa esses agregados interligados como uma única máquina com 42 nós interligados por rede *Fast-Ethernet*. Internamente aos *clusters*, a comunicação entre processos deve ser otimizada de maneira a utilizar a tecnologia empregada como rede primária do agregado.

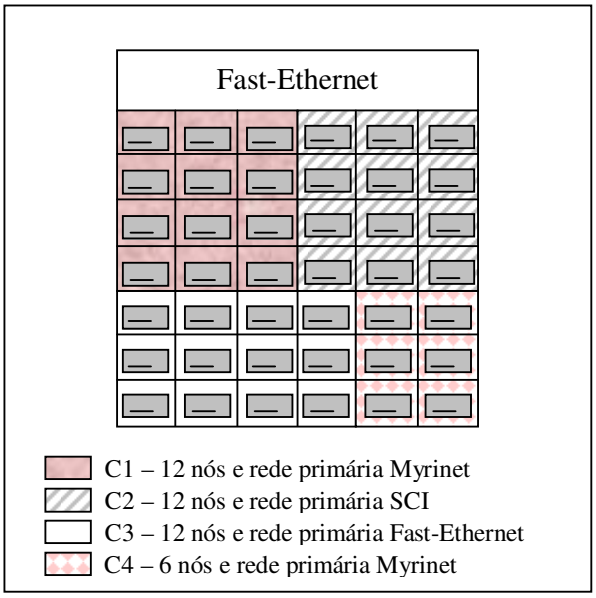

Figura 5.20: Visão do MDX-cc sobre a estrutura de *cluster* de *clusters*.

5.6. Aspectos de *Software*

O MDX-cc roda sobre sistema operacional Linux e necessita algumas bibliotecas para compilação e execução. Para *clusters* com rede *Myrinet*, o MDX-cc requer que seja instalada a biblioteca GM [GM01] em todos os nós. Para agregados baseados em rede SCI, a biblioteca SISCI [SCA00] deve ser instalada em todos os nós.

Capítulo 6

Comunicação entre Processos Clientes

Na comunicação entre processos clientes no MDX-cc, um problema se torna evidente: a forma como é tratada a comunicação entre processos que executam em nós de agregados que utilizam como rede primária tecnologias distintas. Um protocolo de comunicação de decida qual tecnologia deve ser usada na troca de mensagem entre esses clientes se faz necessária.

Na busca por um melhor desempenho na comunicação, a negociação entre processos que desejam se comunicar foi implementada de maneira a não envolver nenhuma troca de mensagens. Ambos os processos devem executar um mecanismo de análise que decida através de qual rede a mensagem deve ser enviada ou recebida. Para poder realizar essa análise, cada processo deve ter acesso a informações sobre aqueles com os quais deseja se comunicar. A busca dessas informações é detalhada a seguir.

6.1. Busca de Informações Sobre os Processos

No MDX-cc, a análise da comunicação terá como elemento base as informações sobre os processos envolvidos. Essas informações estão registradas no Servidor de Comunicação

(seção 5.4.3.2). Cada processo deve solicitar ao servidor os dados referentes aos clientes com os quais deseja se comunicar. Em três momentos distintos essas informações podem ser solicitadas. Essas três alternativas possuem vantagens e desvantagens e são detalhadas a seguir.

6.1.1. A cada Comunicação

Nessa implementação, as informações sobre o processo com o qual haverá comunicação (processo par) são solicitadas ao Servidor de Comunicação no momento da troca de mensagem, ou seja, em toda chamada *MDX_Send( )* ou *MDX_Recv( )*.

Esse modelo apresenta algumas vantagens. Uma delas é o fato de o processo sempre obter informações atualizadas sobre o par, pois busca as informações sempre que uma comunicação com o mesmo ocorre. Outra vantagem é que as informações sobre o processo par podem ser descartadas após a comunicação, não sendo necessária nenhuma estrutura de armazenamento.

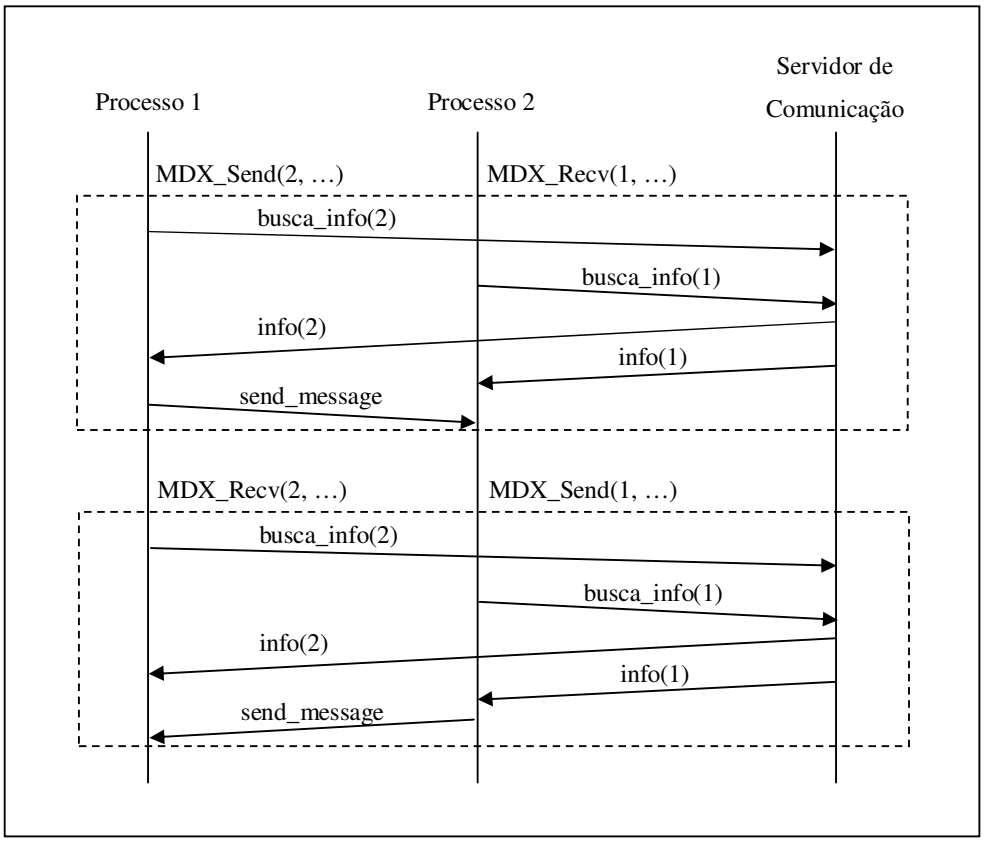

Figura 6.1: Busca de informações a cada comunicação.

A principal desvantagem desse modelo é que a comunicação entre um par de processos é prejudicada pela solicitação de informações ao Servidor de Comunicação. Antes de enviar uma mensagem, o processo deve solicitar ao servidor as informações do processo destino. O processo destino, para receber a mensagem corretamente, deve obter as informações do processo origem. Isso implica em uma troca de mensagem extra a cada chamada às primitivas de comunicação.

A Figura 6.1 apresenta o modelo descrito. Primeiro, o Processo 1 faz uma chamada *MDX_Send( )* para enviar uma mensagem ao Processo 2, que efetua uma chamada *MDX_Recv( )* para recebê-la. Em ambas as chamadas, o primeiro passo, antes de enviar ou receber a mensagem, é buscar as informações do processo par no Servidor de Comunicação (*busca_info( )*). O servidor envia resposta com as informações solicitadas e, enfim, a mensagem é enviada/recebida. Após essa comunicação, o Processo 2 faz uma chamada *MDX_Send( )* para enviar uma mensagem ao Processo 1, que efetua uma chamada *MDX_Recv( )* para recebê-la. Novamente, as informações do processo par são solicitadas ao servidor de comunicação (*busca_info( )*). De posse dessas informações, a comunicação é concluída com o envio da mensagem do Processo 2 ao Processo 1.

6.1.2. No Início do Processo

Nesse modelo, cada processo, ao ser iniciado, solicita ao Servidor de Comunicação as informações sobre todos os outros processos da aplicação. Essas informações são armazenadas e utilizadas para comunicação no decorrer da sua execução. Esse modelo é semelhante ao utilizado pelo IMPI (seção 4.2).

A vantagem desse modelo é que as primitivas de comunicação não são prejudicadas pela solicitação de informações, como no modelo anterior. Ao enviar ou receber uma mensagem, o processo busca as informações sobre o processo par em uma estrutura de armazenamento em sua memória local, por exemplo, uma tabela de processos.

Esse modelo apresenta algumas desvantagens. Uma delas é que cada processo armazena informações sobre todos os outros processos da aplicação. Desta forma, podem ser guardadas informações de processos com os quais não haverá comunicação. Outra desvantagem é o fato de todos os processos necessitarem de uma estrutura de armazenamento das informações. Uma terceira desvantagem pode ser destacada: como as informações são solicitadas ao servidor apenas no início de cada processo, podem ser armazenadas informações desatualizadas sobre alguns processos.

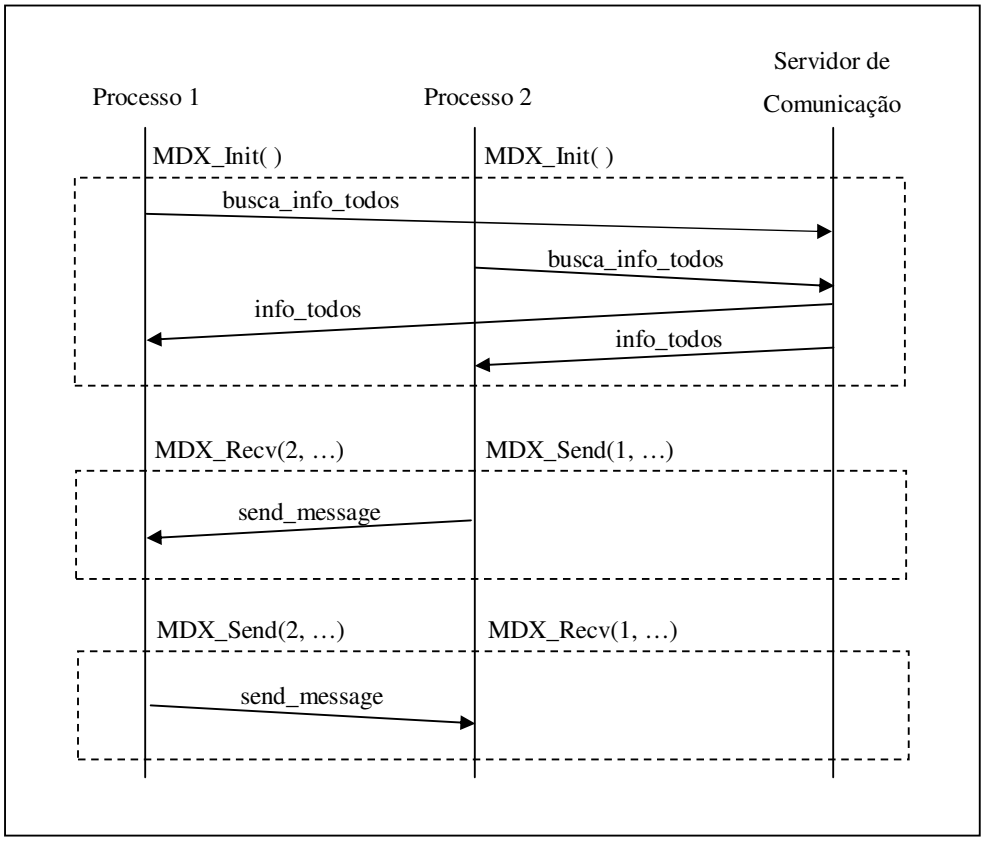

Figura 6.2: Busca de informações no início do processo.

A Figura 6.2 apresenta o modelo de busca no início do processo. Ao ser iniciado, em *MDX_Init( )*, cada processo busca as informações sobre todos os processos da aplicação, e o servidor envia resposta com as informações solicitadas. Essas são armazenadas pelos processos e, nas chamadas às primitivas de comunicação, são usadas para enviar ou receber uma mensagem. Na Figura 6.2, o Processo 2 faz uma chamada *MDX_Send( )* para enviar uma mensagem ao Processo 1, que efetua uma chamada *MDX_Recv( )* para recebê-la. Em seguida, o Processo 1 faz uma chamada *MDX_Send( )* para enviar uma mensagem ao Processo 2, que efetua uma chamada *MDX_Recv( )* para recebê-la. Em ambas as comunicações, não foi necessário solicitar ao Servidor de Comunicação as informações do processo par. Basta enviar ou receber a mensagem.

6.1.3. Na Primeira Comunicação

A terceira alternativa de implementação consiste na busca de informações sobre um determinado processo apenas na primeira comunicação com o mesmo. Por exemplo, se o

processo com identificador 1 deseja enviar uma mensagem ao processo com identificador 15, o primeiro solicita as informações do processo 15 ao Servidor de Comunicação. Essas informações são mantidas em uma estrutura de armazenamento na memória local. Daí em diante, sempre que o processo 1 desejar comunicar com o processo 15, são utilizadas as informações contidas na memória local, não sendo necessário busca-las novamente no Servidor de Comunicação.

Uma vantagem desse modelo é que são armazenadas informações apenas sobre os processos com os quais há comunicação. Outra característica é que apenas a primeira comunicação é afetada pela busca de informações no servidor. Essa característica pode ser considerada uma vantagem em relação à primeira alternativa, na qual todas as comunicações são prejudicadas, enquanto que se torna uma desvantagem em relação à segunda alternativa, onde nenhuma comunicação é afetada pela busca de informações, que ocorre apenas no início de cada processo.

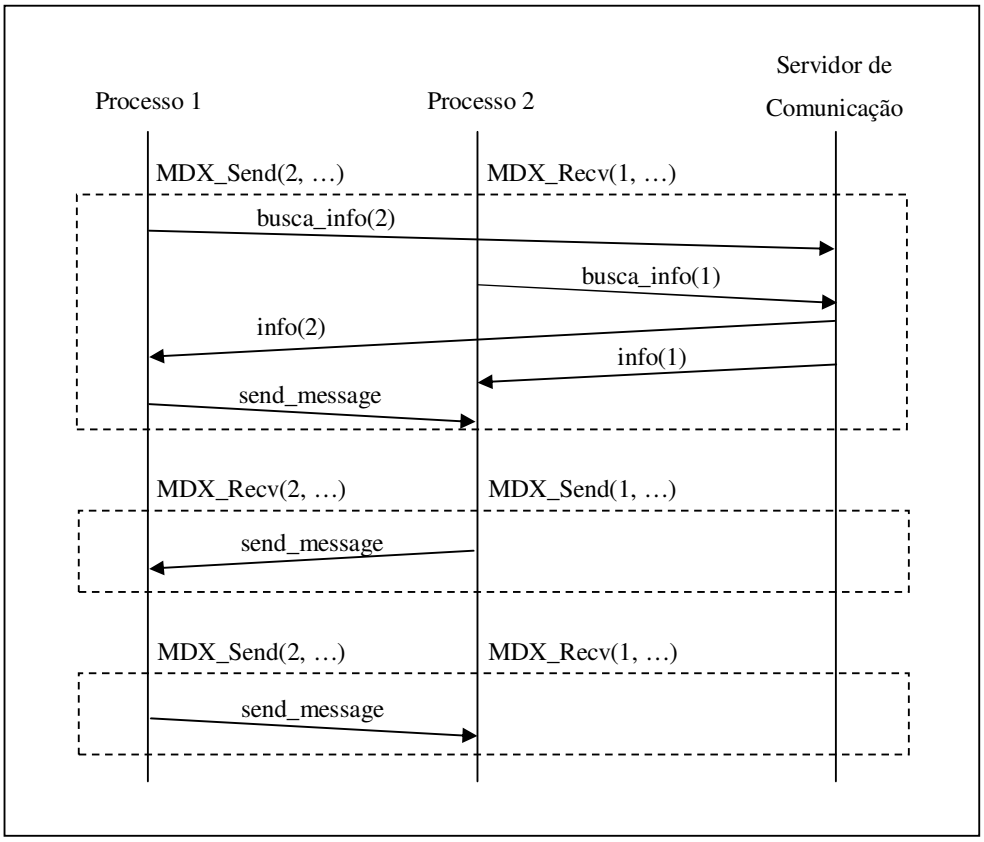

Figura 6.3: Busca de informações na primeira comunicação.

Podem ser citadas como desvantagens desse modelo o fato de ser necessária uma estrutura de armazenamento das informações e a possibilidade de um processo armazenar informações desatualizadas, pois as mesmas são solicitadas ao servidor apenas na primeira vez que se fazem necessárias.

A Figura 6.3 ilustra o modelo apresentado. Inicialmente, o Processo 1 faz uma chamada *MDX_Send( )* para enviar uma mensagem ao Processo 2, que efetua uma chamada *MDX_Recv( )* para recebê-la. Como é a primeira vez que há comunicação entre o Processo 1 e o Processo 2, cada um deles deve buscar as informações do processo par no Servidor de Comunicação (*busca_info( )*). O servidor envia resposta com as informações solicitadas e, enfim, a comunicação é concluída. As informações do processo par são armazenadas. Após essa primeira comunicação, o Processo 2 faz uma chamada *MDX_Send( )* para enviar uma mensagem ao Processo 1, que efetua uma chamada *MDX_Recv( )* para recebê-la. Ambos os processos já possuem as informações um sobre o outro, de forma que basta que a mensagem seja enviada do Processo 2 para o Processo 1. O mesmo ocorre na comunicação subsequente, onde uma mensagem é enviada do Processo 1 ao Processo 2.

6.2. Implementação no MDX-cc

Após uma análise das vantagens e desvantagens de cada uma das alternativas de implementação da busca de informações sobre os processos e das características do MDX-cc, decidiu-se pela implementação da busca de informações apenas na primeira comunicação (seção 6.1.3).

Nesse modelo, não há armazenamento de dados desnecessários, pois são recuperadas informações apenas sobre os clientes com os quais há comunicação. Além disso, no ambiente alvo, as informações sobre as tecnologias de rede não são alteradas no decorrer do processo, eliminando o problema da inconfiabilidade dos dados. O problema da perda de performance, apontado pelo modelo de busca de informações a cada comunicação, é diminuído, uma vez que apenas a primeira comunicação com cada um dos processos é afetada.

6.2.1. Tabela de Clientes

No MDX-cc, cada processo possui uma estrutura de armazenamento de informações de processos. Essa estrutura é idêntica à *ClientTable*, tabela usada pelo Servidor de Comunicação para gerenciar as informações dos processos, detalhada na seção 5.4.3.2, e é denominada

LocalClientTable. O preenchimento dessa tabela é feito pelas primitivas de comunicação *MDX_Send( )* e *MDX_Recv( )*.

O primeiro passo executado pelas primitivas de comunicação é localizar as informações do processo par (origem da mensagem, no caso de um *MDX_Recv( )*, e destino, no caso de um *MDX_Send( )*) na sua *LocalClientTable*. Caso as informações do processo par não se encontrem na tabela, é chamada a função *MDX_LookupClient( )*. Essa função envia uma requisição de busca de informações (*LookupClient*) ao Servidor de Comunicação, passando também o identificador do processo a ser localizado. O servidor envia resposta com as informações solicitadas e essas são armazenadas na posição *i* da *LocalClientTable*, onde *i* é o identificador do processo par. Nas próximas vezes que houver comunicação com o processo *i*, as informações do mesmo já se encontram na tabela local, não sendo necessário solicitá-las ao servidor.

De posse das informações sobre o processo com o qual haverá comunicação, as primitivas podem executar o processo de decisão sobre qual tecnologia de rede será utilizada. Esse processo é detalhado a seguir.

6.2.2. Comunicação entre Processos

No MDX-cc, a negociação entre processos não envolve troca de mensagens. Ambos os processos realizam uma análise e decidem através de qual rede ocorrerá a comunicação. Essa análise se baseia em três itens:

- informações sobre os processos envolvidos na comunicação
- confiabilidade e desempenho das tecnologias de rede
- preferência do usuário

Inicialmente, é feita a verificação de quais as tecnologias de rede estão disponíveis aos dois processos. Dentre essas, é dada a preferência à tecnologia mais rápida e confiável, na seguinte ordem: SCI, *Myrinet*, *Fast-Ethernet* TCP e *Fast-Ethernet* UDP. Este último somente é considerado no caso de o tamanho da mensagem ser menor que 65507 bytes. Em seguida, é adicionada ao processo de análise a preferência do usuário (parâmetro *mode* das primitivas de comunicação *MDX_Send( )* e *MDX_Recv( )*).

No caso de o usuário ter expressado alguma preferência de tecnologia a ser usada na comunicação (parâmetro *mode* com valor diferente de MDX_USE_ANY), é verificado se a

mesma se encontra entre as disponíveis aos dois processos. Em caso afirmativo, essa será a tecnologia usada na comunicação. Caso contrário, a tecnologia usada será aquela de melhor desempenho e maior confiabilidade disponível.

Processo 1: dispõe de rede SCI e Fast-Ethernet – chamada MDX_Send(2, msg, tam, MDX_USE_UDP)
Processo 2: dispõe de rede Myrinet e Fast-Ethernet – chamada MDX_Recv(1, buf, tam, MDX_USE_UDP)
tam = 20000

Análise	***Rede Selecionada***	
	Processo 1	***Processo 2***
Verifica as redes disponíveis a ambos os processos	Fast-Ethernet/TCP Fast-Ethernet/UDP	Fast-Ethernet/TCP Fast-Ethernet/UDP
Dá preferência a rede mais rápida e confiável	Fast-Ethernet/TCP	Fast-Ethernet/TCP
Verifica a preferência expressa pelo usuário	Fast-Ethernet/UDP	Fast-Ethernet/UDP
Rede Selecionada	***Fast-Ethernet/UDP***	***Fast-Ethernet/UDP***

Figura 6.4: Exemplo 1 de negociação entre processos.

O exemplo apresentado pela Figura 6.4 mostra a negociação entre dois processos, onde o Processo 1 dispõe de redes SCI e *Fast-Ethernet*, enquanto o Processo 2 dispõe de redes *Myrinet* e *Fast-Ethernet*. Ambos os processos expressam, nas chamadas *MDX_Send( )* e *MDX_Recv( )*, a preferência pela utilização de rede *Fast-Ethernet* e protocolo UDP. O primeiro critério, que verifica as tecnologias disponíveis a ambos os processos, deixa como possibilidades as tecnologias *Fast-Ethernet*/TCP e *Fast-Ethernet*/UDP. O segundo critério, que dá preferência à tecnologia mais rápida e confiável, eliminou a possibilidade de comunicação por *Fast-Ethernet/UDP*, selecionando *Fast-Ethernet*/TCP para comunicação. No entanto, o último critério de análise, que envolve a preferência do usuário, no caso, *Fast-Ethernet*/UDP, alterou essa decisão, uma vez que a tecnologia desejada pelo usuário se encontra entre aquelas selecionadas pelo primeiro critério de análise. A rede selecionada para comunicação é *Fast-Ethernet*/UDP.

No exemplo apresentado pela Figura 6.5, ambos os processos dispõe de redes SCI e *Fast-Ethernet*, e ambos expressam, nas chamadas *MDX_Send( )* e *MDX_Recv( )*, a preferência pela utilização de rede *Myrinet* na comunicação. O primeiro critério, que verifica as

tecnologias disponíveis a ambos os processos, deixa como possibilidades as tecnologias *Fast-Ethernet*/TCP, *Fast-Ethernet*/UDP e SCI. O segundo critério, que dá preferência à tecnologia mais rápida e confiável, selecionou a rede SCI para comunicação. O último critério de análise, que envolve a preferência do usuário, no caso, *Myrinet*, não altera essa decisão, uma vez que a tecnologia desejada pelo usuário não se encontra entre aquelas selecionadas pelo primeiro critério de análise. A rede selecionada para comunicação é SCI.

Processo 1: dispõe de rede SCI e Fast-Ethernet – chamada MDX_Send(2, msg, tam, MDX_USE_MYR)
Processo 2: dispõe de rede SCI e Fast-Ethernet – chamada MDX_Recv(1, buf, tam, MDX_USE_MYR)
tam = 128000

Análise	***Rede Selecionada***	
	Processo 1	***Processo 2***
Verifica as redes disponíveis a ambos os processos	Fast-Ethernet/TCP Fast-Ethernet/UDP SCI	Fast-Ethernet/TCP Fast-Ethernet/UDP SCI
Dá preferência a rede mais rápida e confiável	SCI	SCI
Verifica a preferência expressa pelo usuário	SCI	SCI
Rede Selecionada	***SCI***	***SCI***

Figura 6.5: Exemplo 2 de negociação entre processos.

Se a rede selecionada pelos critérios de análise é *Fast-Ethernet*/TCP, é verificada a existência de uma conexão TCP entre os processos envolvidos. Caso essa conexão não exista, a mesma deve ser estabelecida. Na primitiva *MDX_Recv( )*, é feita uma chamada à função *MDX_AcceptConn( )*, que aguarda uma conexão TCP do processo origem da mensagem. Na primitiva *MDX_Send( )* ocorre uma chamada à função *MDX_RequestConn( )*, que solicita uma conexão TCP no endereço e porta onde o processo receptor está aguardando. Este processo de estabelecimento de conexão TCP é realizado apenas uma vez, sendo que a conexão estabelecia será usada nas próximas comunicações entre os dois processos. Já existindo a conexão, a mensagem pode ser enviada ou recebida. A primitiva *MDX_Send( )* envia a mensagem via conexão TCP chamando a função *MDX_RSend( )*. Do outro lado, a

primitiva *MDX_Recv( )* recebe a mensagem por uma chamada à função *MDX_RRecv( )*, função de recebimento de mensagem via conexão TCP.

A Figura 6.6 mostra a comunicação entre dois processos através de tecnologia *Fast-Ethernet* com protocolo TCP. Na primeira comunicação, antes de enviar/receber a mensagem, ambos os processos devem solicitar as informações do processo par no Servidor de Comunicação e deve ser efetuada uma conexão TCP entre os dois processos. Em seguida, a mensagem é transmitida. Nas próximas comunicações, a busca de informações e o estabelecimento da conexão não mais são necessários. Basta que a mensagem seja enviada/recebida

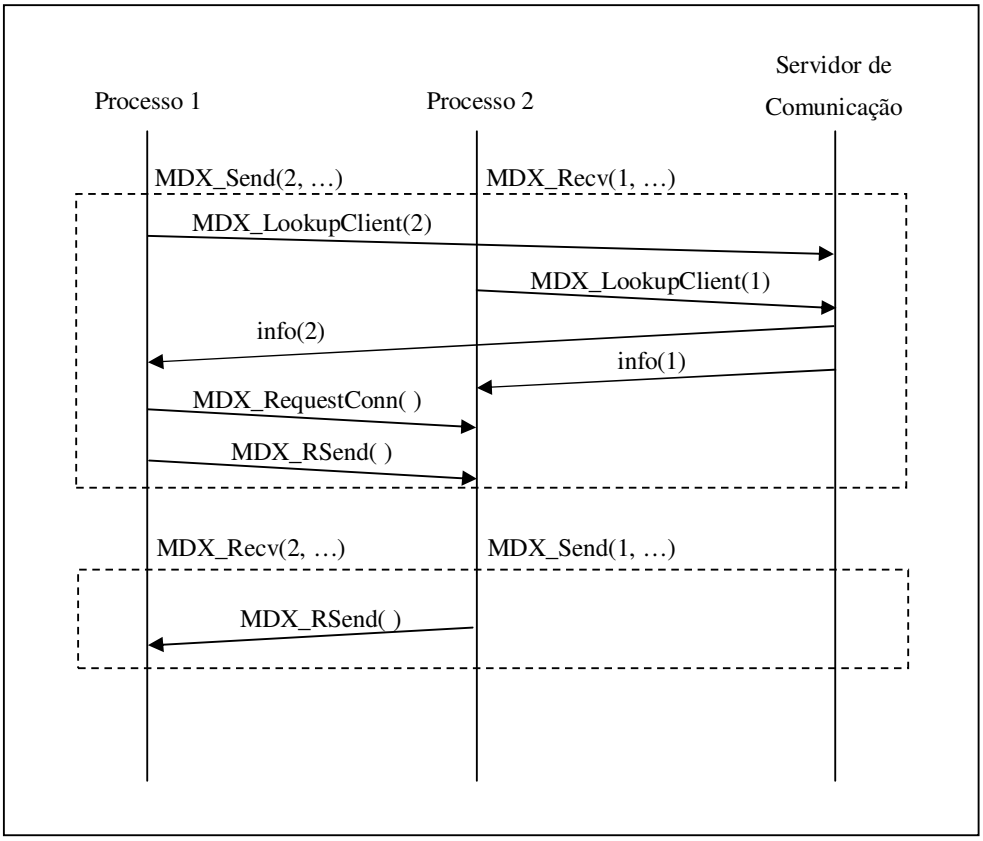

Figura 6.6: Comunicação entre processos via rede *Fast-Ethernet*/TCP.

Caso a tecnologia selecionada seja *Fast-Ethernet* com protocolo UDP, a mensagem é simplesmente transmitida entre os processos, não sendo necessária o estabelecimento de conexão. A primitiva *MDX_Send( )* envia a mensagem para o endereço e porta do processo par, através da função *MDX_SendTo( )*. Do lado do receptor, a primitiva *MDX_Recv( )* chama

a função *MDX_RecvFrom( )*, que recebe uma mensagem pela rede *Fast-Ethernet* e protocolo UDP.

A Figura 6.7 mostra a comunicação entre dois processos através de tecnologia *Fast-Ethernet* com protocolo UDP. Da mesma forma que na comunicação por TCP, na primeira comunicação, antes de enviar/receber a mensagem, ambos os processos devem solicitar as informações do processo par no Servidor de Comunicação. Em seguida, a mensagem é transmitida. Nas próximas comunicações, a busca de informações não é necessária. Basta enviar/receber a mensagem.

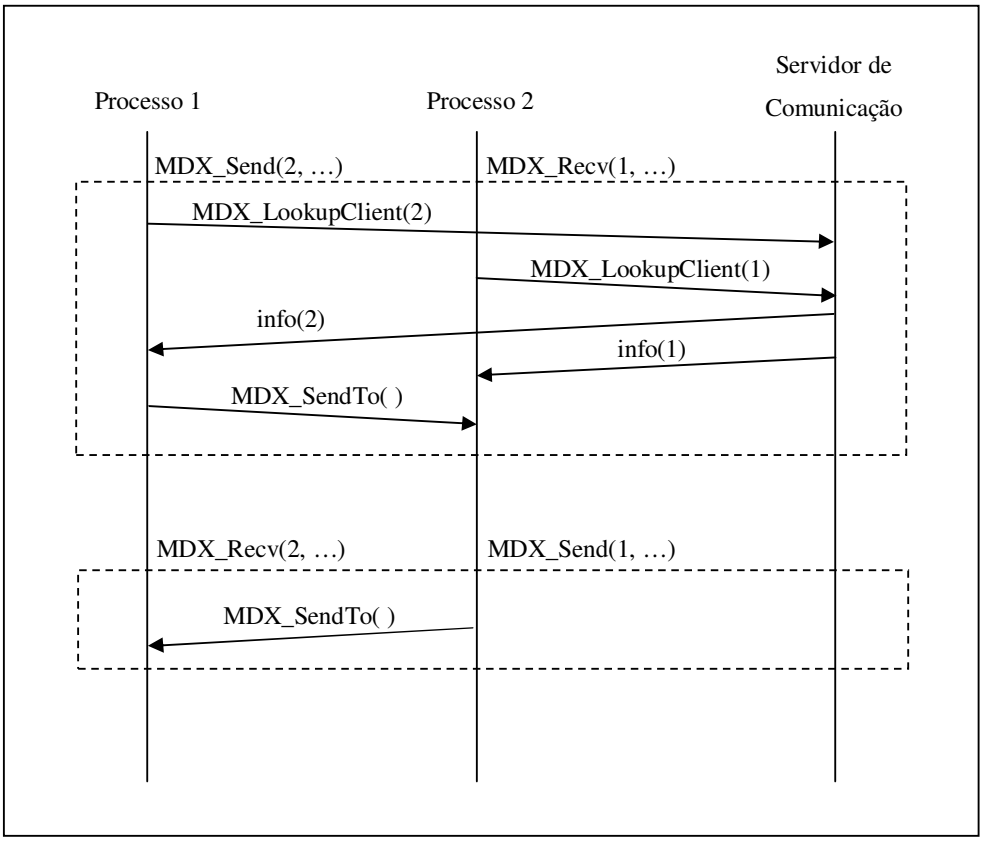

Figura 6.7: Comunicação entre processos via rede *Fast-Ethernet*/UDP.

A comunicação via rede *Myrinet* acontece através das portas de comunicação GM, criadas no início de cada processo (ver seção 5.4.4.1). A primitiva *MDX_Send( )* envia, através da sua porta, uma mensagem à porta criada pelo processo par. Isso é feito pela função *MDX_GM_Send( )*. Essa função efetua uma cópia da mensagem para uma área de memória DMA, utilizada pela biblioteca GM (seção 2.4.1). O processo receptor consome a mensagem da sua porta de comunicação através de uma chamada à função *MDX_GM_Recv( )*. Essa

função copia a mensagem, recebida em uma área de memória DMA, para a área de memória passada em *MDX_Recv( )*. Essas cópias de mensagem prejudicam o desempenho da comunicação.

A Figura 6.8 mostra a comunicação entre dois processos através de tecnologia *Myrinet*. Na primeira comunicação, antes de enviar/receber a mensagem, é realizada a busca de informações do processo par no Servidor de Comunicação. Em seguida, a mensagem é transmitida. Nas próximas comunicações, a busca de informações não mais é necessária. Basta enviar/receber a mensagem.

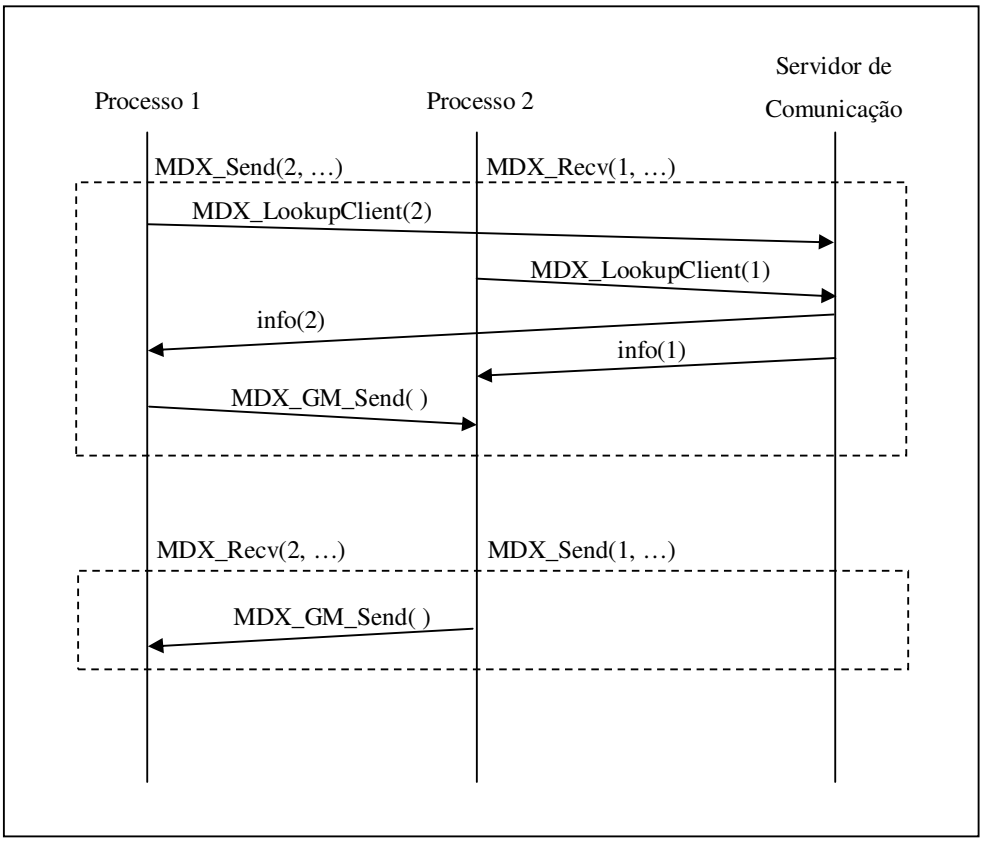

Figura 6.8: Comunicação entre processos via rede *Myrinet*.

Via rede SCI, a comunicação é feita através da biblioteca YAMPI. A primitiva *MDX_Send( )* efetua uma chamada à função *MDX_YAMPI_Send( )*, que envia uma mensagem de um processo YAMPI a outro. Do outro lado, a primitiva *MDX_Recv( )* recebe essa mensagem chamando a função *MDX_YAMPI_Recv( )*.

A Figura 6.9 mostra a comunicação entre dois processos através de tecnologia SCI. Na primeira comunicação, antes de enviar/receber a mensagem, é feita a busca de informações do

processo par no Servidor de Comunicação. Em seguida, a mensagem é transmitida. Nas próximas comunicações, a busca de informações não mais é necessária. Basta enviar/receber a mensagem.

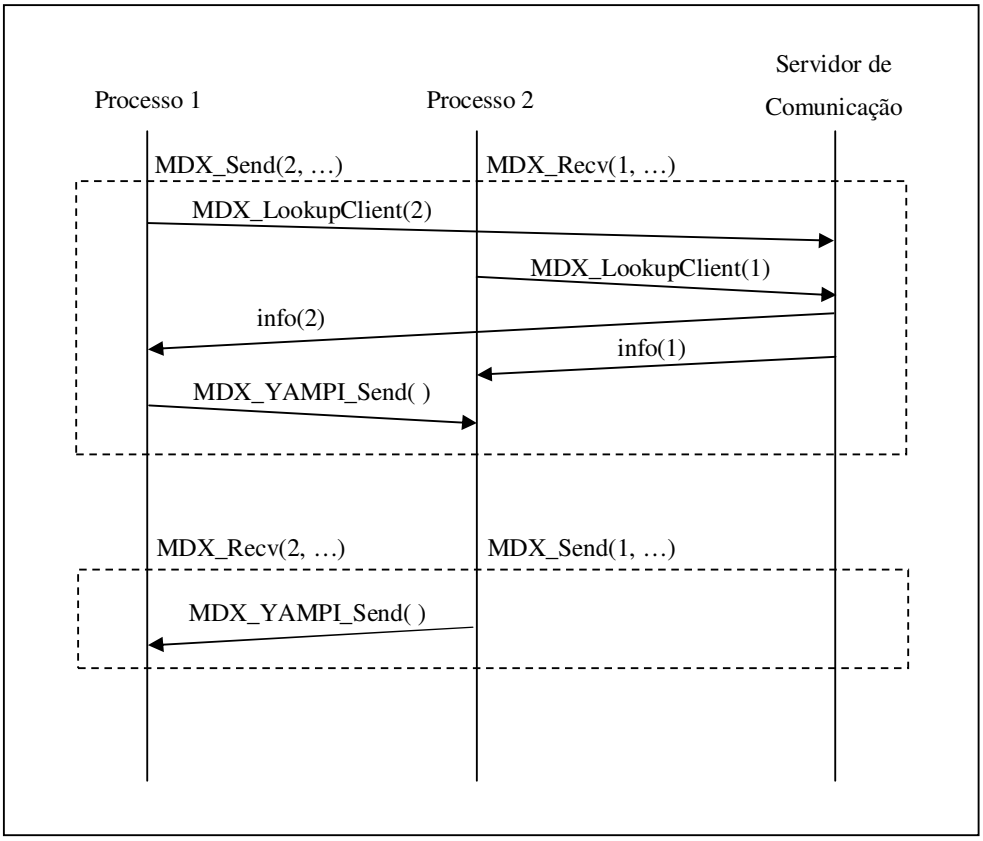

Figura 6.9: Comunicação entre processos via rede SCI.

Capítulo 7

Avaliação de Desempenho do MDX-cc

O desempenho de uma aplicação que é executada em uma plataforma paralela é um dos pontos principais a serem avaliados por uma equipe de desenvolvimento. Embora predizer a performance que será obtida por uma aplicação em um ambiente paralelo seja uma tarefa difícil [KRI92], uma boa avaliação de desempenho do sistema de comunicação sobre o qual a mesma está baseada é de grande ajuda. Este capítulo apresenta os testes de desempenho realizados com o MDX-cc.

7.1. Ambiente de Teste – CPAD

O ambiente de teste utilizado para avaliar a performance do MDX-cc é o CPAD (Centro de Processamento de Alto Desempenho). Este centro é o resultado de uma parceria entre a Pontifícia Universidade Católica do Rio Grande do Sul e a HP do Brasil. A HP possui atividades conjuntas com universidades e centros de pesquisa reconhecidos, objetivando a continuidade de um amplo processo de cooperação nacional através de parcerias.

O CPAD se propõe a ser um Centro de Pesquisa dedicado a investigar arquiteturas de *software* e *hardware* para processamento de alto desempenho. Para este fim o CPAD conta atualmente com quatro máquinas agregadas onde servidores HP são interligados através de redes de alta velocidade. Esses agregados são:

Amazonia

Agregado com 16 nós HP-E60 com dois processadores Pentium III 550 MHz, com 256 MB de RAM e HD de 20 GB (sem monitor e sem teclado). Esses nós utilizam rede primária *Myrinet* e rede secundária *Fast-Ethernet*, e rodam sistema operacional Linux.

Tropical

Agregado com 8 nós HP-E-PC com dois processadores Pentium III 1 GHz, com 256 MB de RAM e HD de 20 GB (sem monitor e sem teclado). Esses nós possuem rede *Fast-Ethernet* e rodam sistema operacional Linux.

Pantanal

Cluster com 4 nós HP-Vectra VEI8 com um processador Pentium III 550 MHz, com 256 MB de RAM e HD de 9 GB (sem monitor e sem teclado). Esses nós rodam com sistema operacional Linux, possuem rede primária SCI e rede secundária *Fast-Ethernet*.

Ombrofila

Agregado com 16 nós HP-E-PC com um processador Pentium III 1 GHz, com 256 MB de RAM e HD de 20 GB (sem monitor e sem teclado). Esses nós possuem rede *Fast-Ethernet* e rodam sistema operacional Linux.

A interligação dos nós desses agregados através de rede *Fast-Ethernet* forma a estrutura de *cluster* de *clusters* exigida pelo MDX-cc (seção 5.5). Sobre essa estrutura foram realizados os testes de desempenho do ambiente.

7.2. Medidas Básicas de Desempenho

As características e limitações de um sistema de troca de mensagens têm diferentes impactos sobre diferentes aplicações. Tendo isto em vista, é fundamental a comparação entre diferentes métodos para que se possa escolher o mais adequado para uma determinada aplicação.

A maioria das medidas de performance em sistemas de comunicação é dada em termos de dois parâmetros: latência e largura de banda (vazão), referenciados neste trabalho como *L* e *B* (*bandwidth*) respectivamente. O primeiro diz respeito à semântica de sincronismo de uma troca de mensagens e o segundo à semântica de transferência de dados.

7.2.1. Latência

O propósito da avaliação da latência é caracterizar a velocidade em que o sistema de comunicação consegue sincronizar, através de troca de mensagens, dois processos cooperativos. Quanto mais rápido o sistema é capaz de executar esta tarefa, melhor será a performance de aplicações altamente síncronas.

Pelo descrito acima é possível definir latência como o tempo necessário para se enviar uma mensagem da origem ao destino, ou seja, do instante em o processo origem inicia uma operação de envio até o instante em que o processo destino é notificado sobre o recebimento desta mensagem. Os processos "origem" e "destino" são aplicações executadas em diferentes nós de um agregado ou de uma estrutura de *cluster* de *clusters*.

A maneira mais comum de se computar a latência é pelo uso de uma avaliação do tipo *ping-pong*. Nesta avaliação são utilizados dois processos, chamados aqui de "*ping*" e "*pong*". O processo *ping* envia uma mensagem para o processo *pong* e espera a resposta, sendo que esta é apenas uma cópia da mensagem recebida pelo processo *pong*. O tempo gasto na execução desta transação é medida pelo processo *ping* e consiste no tempo gasto desde a transmissão da mensagem até a sua recepção.

$$L = \frac{\left(\sum_{n=1}^{500} \frac{t_n}{2} \right)}{500}$$

Figura 7.1: Cálculo de latência.

Para a avaliação do MDX-cc foram feitas medidas com mensagens de tamanhos diferentes, variando entre 1 byte e 1 Mbytes, repetidas 500 vezes cada . A latência *L* para tamanho é calculada como a média dos valores obtidos, dividida por dois, como mostra a Figura 7.1.

7.2.2. Largura de Banda

Esta medida caracteriza o quão rápida é a transferência de dados ocorre entre o transmissor e o receptor. Para esta medida também foram considerados dois processos, chamados também de "*ping*" e "*pong*".

Se o tempo *t* necessário para se transferir um bloco de dados com *S* bytes, medido a partir do instante que o processo *ping* inicia a operação de envio até o recebimento da mensagem por *pong*, então o valor da largura de banda *B*, será calculado segundo a equação mostrada pela Figura 7.2.

$$B = \frac{S}{t}$$

Figura 7.2: Cálculo de largura de banda.

Qualquer medida de largura de banda requer transferência de grandes quantidades de dados, para isto foram considerados três métodos para obter esta medida:

- *Single-message*: envio de uma mensagem muito longa. Esta técnica é a mais simples mas só é adequada quando o sistema troca de mensagens permite o uso de mensagens muito grandes.
- *Stream*: envio de uma única, e longa, mensagem. Esta técnica é possível em sistemas de comunicação que suportam transferência de dados em modo *stream*, como *sockets* TCP/IP [COM95].
- *Burst*: envio de uma rápida seqüência de mensagens de tamanho fixo. Esta técnica é utilizada quando o tamanho máximo mensagem permitido pelo sistema de troca de mensagens não é grande o suficiente. Esta técnica não é útil quando se necessita de uma avaliação do comportamento de um sistema de mensagens segundo o tamanho da

mensagem (*per-message*). Neste caso deve-se emular um mensagem grande pela fragmentação da mesma em uma longa seqüência de mensagens de tamanho fixo, que devem utilizadas para se reconstruir a mensagem original no receptor. Para se ter uma avaliação mais correta, os processos de fragmentação e reconstrução da mensagem devem ser incluídos no cálculo da largura de banda.

É interessante se fazer uma avaliação *per-message* de um sistema de mensagens pois desta maneira pode-se avaliar a eficiência do uso das características desta interface. Sendo assim, para este trabalho, adotou-se a técnica de *single-message* pois o tamanho de mensagens permitido no sistema é suficiente para realização dos testes. Para a medição da largura de banda, foi utilizada a mesma técnica *ping-pong* descrita na seção 7.2.1 para medir o intervalo entre o envio da mensagem e sua recepção (*round-trip*). O processo *ping* envia uma mensagem com *S* bytes de tamanho para o processo *pong*, que deve enviar os mesmos dados de volta. O tempo de *round-trip* (designado aqui como t_{RT}) necessário para se executar as duas operações é medido.

$$B_n = \frac{S}{\frac{t_{RT}}{2}}$$

Figura 7.3: Cálculo de largura de banda para uma mensagem.

O cálculo da largura de banda média para um tamanho de mensagem é dada como a média aritmética das larguras de bandas obtidas, como mostra a Figura 7.4

O processo descrito é repetido 500 vezes para cada tamanho de mensagem entre 1 byte e 1 Mbyte. O tempo necessário para se enviar uma mensagem de tamanho *S* é calculado como a metade de t_{RT}, e a vazão para cada repetição é dada pela equação mostrada na Figura 7.3.

$$B = \frac{\left(\sum_{n=1}^{500} B_n \right)}{500}$$

Figura 7.4: Cálculo de largura de banda média.

O Algoritmo 7.1 apresenta o código da aplicação de teste de latência e largura de banda:

Algoritmo 7.1: Programa de avaliação do MDX-cc.

```
#define SIZE 1024*1024*4
#define TIMES 20
typedef unsigned char byte;
struct timeval initial_time, final_time;

int main(int argc, char ** argv)
{
int i,j;
double lat = 0.0, double bw = 0.0;
byte buffer[SIZE];

MDX_Init();

if(MDX_id==0) //processo ping
  for(j = 1;j <= SIZE;j = ((j==0)?1:j*2))
       for(i = 0;i < TIMES;i++)
       {
         MDX_Recv(1, buffer, j, MDX_USE_ANY);
         MDX_Send(1, buffer, j, MDX_USE_ANY);
       }
else if(MDX_id==1)  //processo pong
  for(j = 1;j <= SIZE;j = ((j==0)?1:j*2))
   {
    for(i = 0;i < TIMES;i++)
    {
      gettimeofday(&initial_time,NULL);     //inicio da contagem de tempo
      MDX_Send(0, buffer, j, MDX_USE_ANY);
      MDX_Recv(0, buffer, j, MDX_USE_ANY);
      gettimeofday(&final_time,NULL);       //fim da contagem de tempo
      lat = lat + ((time2double(&final_time)-time2double(&initial_time))/2.0);
      bw += j/((time2double(&final_time) - time2double(&initial_time))/2.0);
    }
    printf("%-d\t\t%.4f usec\t\t%.4f MBytes/sec\n", j, (lat/(double)TIMES) *
           1000000.0, (bw/1048578)/TIMES); fflush(stdout);
    lat = 0.0;  bw = 0.0;
   }
  MDX_Finalize();
}
```

7.3. MDX-cc sobre *Fast-Ethernet*

O primeiro teste apresentado será o da comunicação sobre rede *Fast-Ethernet*. A avaliação foi feita com a aplicação *ping-pong* executando em dois nós do *cluster* Tropical. Nesta seção são mostrados os resultados obtidos com a utilização dos protocolos TCP e UDP. Am ambos os casos é feita uma comparação com os resultados obtidos por *sockets*.

6.2.3. TCP

Rodando sobre rede *Fast-Ethernet*, com a utilização do protocolo TCP, a aplicação de avaliação do MDX-cc obteve os seguintes resultados:

Tabela 7.1: Teste de desempenho do MDX-cc sobre *Fast-Ethernet* – protocolo TCP.

	MDX-cc		***Sockets TCP***	
Mensagem	**Latência**	**Vazão**	**Latência**	**Vazão**
1	117,4118	0,0122	76,41829	0,012488
4	78,5706	0,0486	77,35743	0,049332
32	83,8404	0,3641	82,66784	0,369301
128	102,6131	1,1900	101,0675	1,208254
1024	267,0973	3,6565	263,3053	3,709105
4096	578,0015	6,7528	577,9251	6,759766
32768	3022,2931	10,3398	3019,522	10,34933
131072	11385,2777	10,9791	11378,29	10,98583
1048576	89386,3135	11,1874	89374,25	11,18891

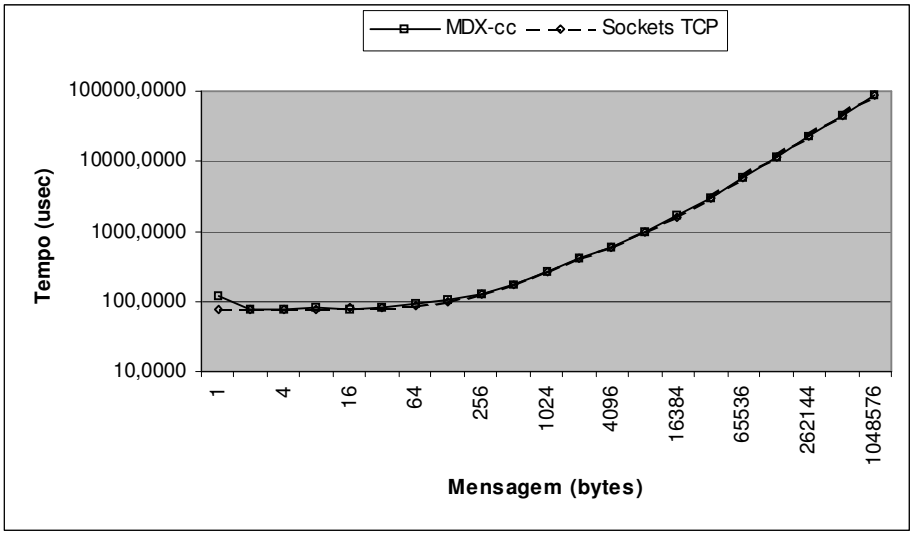

Figura 7.5: Gráfico de latência do MDX-cc sobre *Fast-Ethernet* – protocolo TCP.

Os resultados apresentados pelo teste de latência (Figura 7.5) indicam um desempenho bastante satisfatório, próximos ao desempenho obtido por *sockets* TCP puro. Isto indica que o MDX-cc não inclui um grande *overhead* na comunicação. A troca da primeira mensagem, de tamanho 1 byte, apresenta uma latência mais alta devido à necessidade da busca das

informações do processo par no Servidor de Comunicação (seção 6.2). As demais trocas de mensagem apresentam um melhor desempenho, pois cada processo já tem a sua disposição as informações do processo par, não sendo necessária a busca das mesmas no Servidor de Comunicação. A partir da segunda troca de mensagem, a latência da comunicação aumenta conforme há o aumento do tamanho da mensagem.

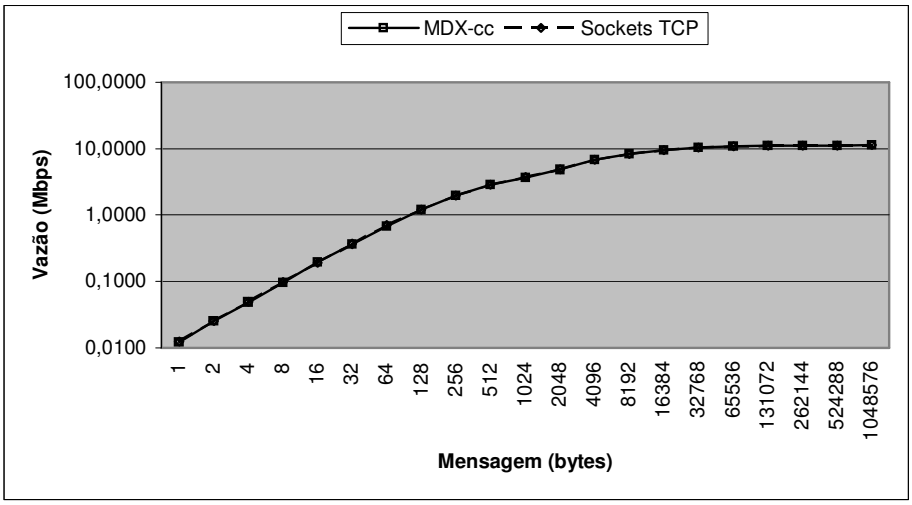

Figura 7.6: Gráfico de vazão do MDX-cc sobre *Fast-Ethernet* – protocolo TCP.

Os valores apresentados pelo teste de latência se refletem nos resultados do teste de vazão, apresentados pela Figura 7.6. O gráfico mostra uma proximidade entre os valores obtidos pelo MDX e por *sockets*. Na troca da primeira mensagem, de 1 byte, a vazão é bastante baixa. Para as outras trocas de mensagem, os valores vão ficando maiores à medida que o tamanho da mensagem também aumenta.

6.2.4. UDP

Rodando sobre rede *Fast-Ethernet*, com a utilização do protocolo UDP, o MDX-cc obteve os resultados mostrados na Tabela 7.2.

Os resultados apresentados pelo teste de latência do MDX-cc sobre *Fast-Ethernet* (Figura 7.7) com protocolo UDP indicam um desempenho razoável, mas um pouco distante do desempenho obtido pelo uso de *sockets* UDP. Essa diferença pode estar na forma como é montada a estrutura (*socket*) com os dados do *socket* destino. No MDX-cc, essa estrutura é montada em toda comunicação. Isto acontece na função *MDX_SendTo( )* (seção 6.2.2), de envio de mensagens por UDP. Já na aplicação utilizada para testar o uso de *sockets* puro, essa

estrutura é montada uma única vez, sendo utilizada a mesma para todas as trocas de mensagem. Os testes do protocolo UDP foram feitos apenas com mensagens menores que 65507 bytes, tamanho máximo suportado por esse protocolo no MDX-cc. Da mesma forma que no teste de latência apresentado anteriormente, a troca da primeira mensagem, de tamanho 1 byte, apresenta uma latência alta e as demais trocas de mensagem apresentam um melhor desempenho. Para mensagens maiores, a latência da comunicação aumenta conforme há o aumento do tamanho da mensagem.

Tabela 7.2: Teste de desempenho do MDX-cc sobre *Fast-Ethernet* – protocolo UDP.

	MDX-cc		***Sockets UDP***	
Mensagem	**Latência**	**Vazão**	**Latência**	**Vazão**
1	106,0367	0,0133	46,5450	0,0331
4	71,2671	0,0536	47,9970	0,0856
32	73,8500	0,4136	51,2450	0,5200
128	88,9140	1,3738	67,7340	1,5844
1024	256,3420	3,8100	216,5490	4,3560
4096	588,1519	6,6418	528,5460	6,9958
32768	3159,8795	9,8897	3035,9430	10,0568

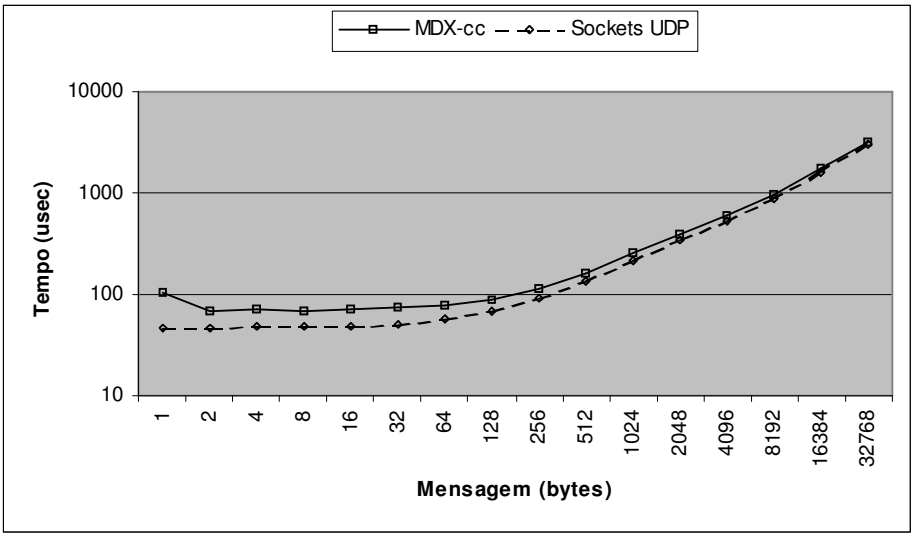

Figura 7.7: Gráfico de latência do MDX-cc sobre *Fast-Ethernet* – protocolo UDP.

Os resultados do teste de vazão, apresentados pela Figura 7.8, são conseqüência dos números do teste de latência. Os números obtidos são mais baixos em relação aos obtidos por *sockets*. Na troca da primeira mensagem, de 1 byte, a vazão é bastante baixa. Nas outras

trocas de mensagens, os valores vão ficando maiores à medida que o tamanho da mensagem também aumenta. Da mesma forma que no teste de latência, há uma leve vantagem do protocolo UDP para mensagens de até 4 Kbytes.

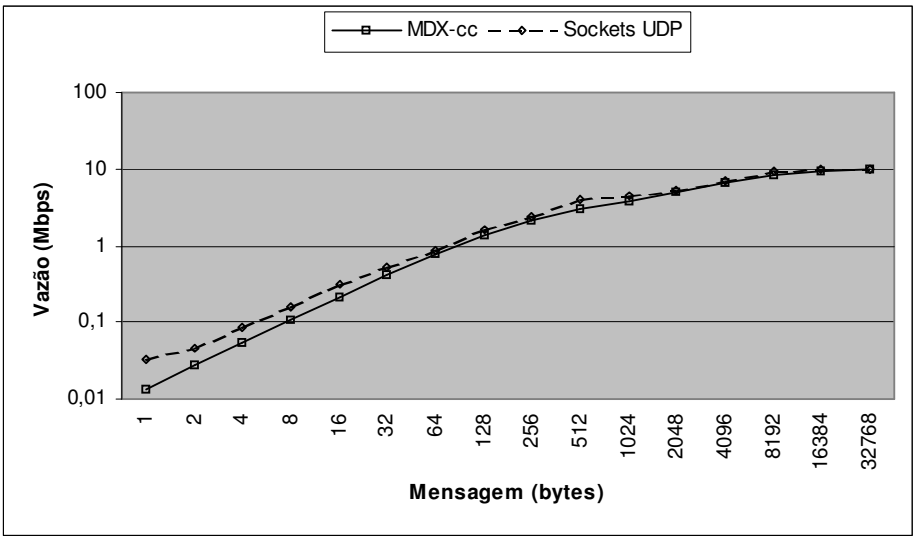

Figura 7.8: Gráfico de vazão do MDX-cc sobre *Fast-Ethernet* – protocolo UDP.

7.4. MDX-cc sobre *Myrinet*

A avaliação do MDX-cc rodando sobre rede *Myrinet* foi feita com a aplicação *ping-pong* rodando em dois nós do *cluster* Amazonia. Esta seção apresenta e comenta os resultados obtidos. Esses resultados são comparados com os obtidos pela ferramenta MPI.

A Tabela 7.3 mostra os valores obtidos pela aplicação de teste do MDX-cc sobre rede *Myrinet*:

Tabela 7.3: Teste de desempenho do MDX-cc sobre *Myrinet*.

	MDX-cc		***MPICH/GM***	
Mensagem	**Latência**	**Vazão**	**Latência**	**Vazão**
1	73,7420	0,0268	18,4425	0,0534
4	35,6264	0,1071	18,4249	0,2138
32	32,9956	0,9252	19,4826	1,6139
128	35,2291	3,4658	28,0269	4,4866
1024	63,4559	15,3956	56,7870	17,3416
4096	159,8974	24,4302	326,2133	27,6851
32768	754,3463	41,4287	484,0233	64,6329
131072	2872,3642	43,5207	2393,8515	78,9849
1048576	26545,2058	37,6756	17461,0610	81,6459

Os resultados apresentados pelo teste de latência do MDX-cc sobre *Myrinet* (Figura 7.9) indicam um desempenho apenas razoável, abaixo do esperado e bem abaixo do desempenho obtido por MPI. Esse resultado é reflexo das cópias de mensagem de e para áreas de memória DMA realizadas pelas funções *MDX_GM_Send( )* e *MDX_GM_Recv( )* (seção 6.2.2). Da mesma forma que nos testes apresentados anteriormente, a troca da primeira mensagem apresenta um valor elevado e as demais trocas de mensagem apresentam um melhor desempenho. Para mensagens maiores, a latência da comunicação aumenta conforme há o aumento do tamanho da mensagem.

Os resultados do teste de vazão, apresentados pela Figura 7.10, mostram o mesmo comportamento apresentado pelo teste de latência. Na troca da primeira mensagem, de 1 byte, a vazão é bastante baixa, enquanto que, para mensagens maiores, os valores vão aumentando à medida que o tamanho da mensagem também aumenta.

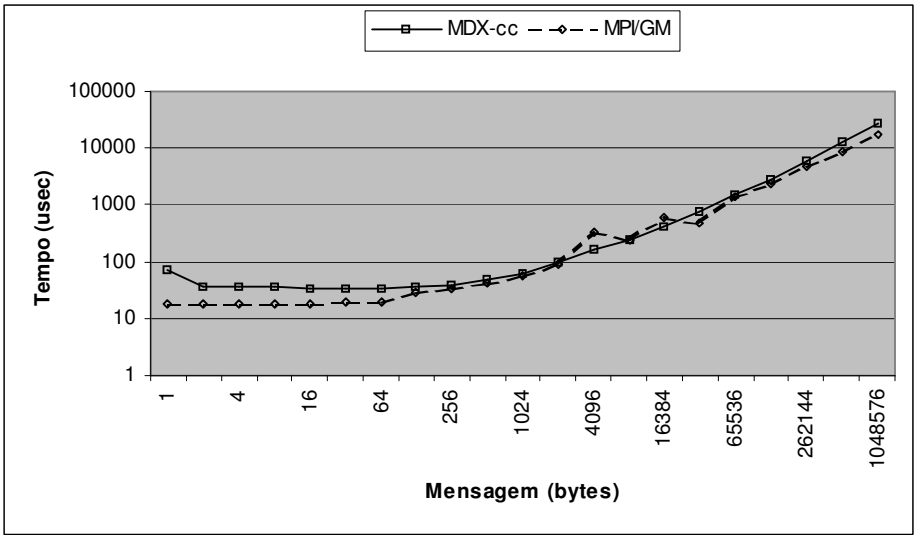

Figura 7.9: Gráfico de latência do MDX-cc sobre *Myrinet*.

Importante ressaltar a capacidade do *hardware Myrinet*. Apesar dos números apresentados pelo teste estarem abaixo do esperado, o desempenho ainda é muito superior àquele obtido pelo ambiente executando sobre rede *Fast-Ethernet*, tanto por protocolo UDP quanto por protocolo TCP.

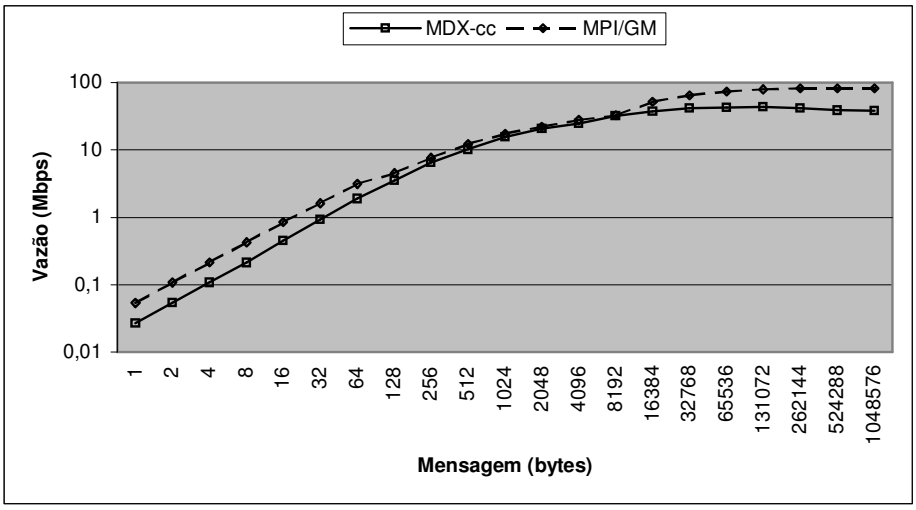

Figura 7.10: Gráfico de vazão do MDX-cc sobre *Myrinet*.

7.5. MDX-cc sobre SCI

Os testes do MDX-cc sobre rede SCI foram realizados em dois nós do *cluster* Pantanal. Os resultados obtidos na avaliação são apresentados e comentados a seguir. A Tabela 7.4 mostra os valores obtidos pela aplicação de teste do MDX-cc sobre rede *Myrinet*, juntamente com aqueles obtidos pelo MPI.

Tabela 7.4: Teste de desempenho do MDX-cc sobre SCI.

	MDX-cc		***ScaMPI***	
Mensagem	**Latência**	**Vazão**	**Latência**	**Vazão**
1	27,9881	0,0828	381,4320	0,1235
4	16,4410	0,2324	7,8940	0,4974
32	14,0457	2,1793	10,4270	2,9722
128	13,2181	9,5272	18,1830	6,8929
1024	25,9465	37,7122	36,6420	26,8196
4096	78,0773	50,1592	78,9650	49,5594
32768	564,4656	55,3882	522,6000	59,8330
131072	2232,7079	56,0035	1835,6360	68,1137
1048576	19486,2127	51,3314	12916,4750	77,4252

Os resultados apresentados pelo teste de latência do MDX-cc sobre SCI (Figura 7.11) indicam um desempenho dentro do esperado, próximo, mas ainda abaixo do obtido pela biblioteca MPI. Para mensagens com tamanho entre 128 bytes e 32 Kbytes o MDX-cc obteve até um desempenho melhor. Entretanto, para mensagens maiores, o MPI apresenta um melhor desempenho.

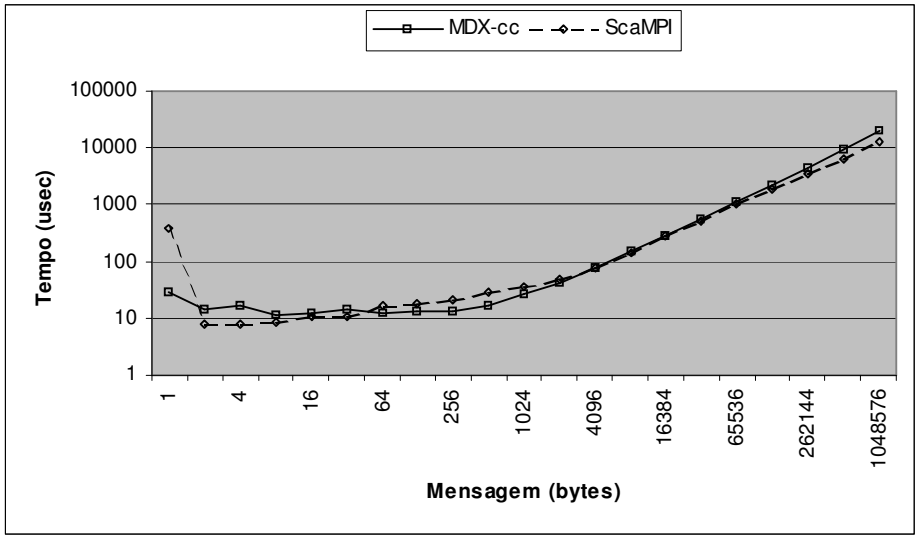

Figura 7.11: Gráfico de latência do MDX-cc sobre SCI.

A Figura 7.12 apresenta o gráfico do teste de vazão do MDX-cc sobre rede SCI. Esse gráfico é um reflexo do comportamento apresentado pelo teste de latência. O MDX-cc obtém uma maior vazão para mensagens entre 128 bytes e 32 Kbytes. Na troca da primeira mensagem a vazão é bastante baixa, enquanto que, para mensagens maiores, os valores vão aumentando à medida que o tamanho da mensagem também aumenta.

A comunicação através de tecnologia SCI foi a que apresentou o melhor desempenho no MDX-cc. Os gráficos apresentados pela Figura 7.11 e pela Figura 7.12 mostram uma menor latência e, consequentemente, maior vazão da comunicação sobre SCI em relação à comunicação sobre as tecnologias *Fast-Ethernet* TCP, *Fast-Ethernet* UDP e *Myrinet*.

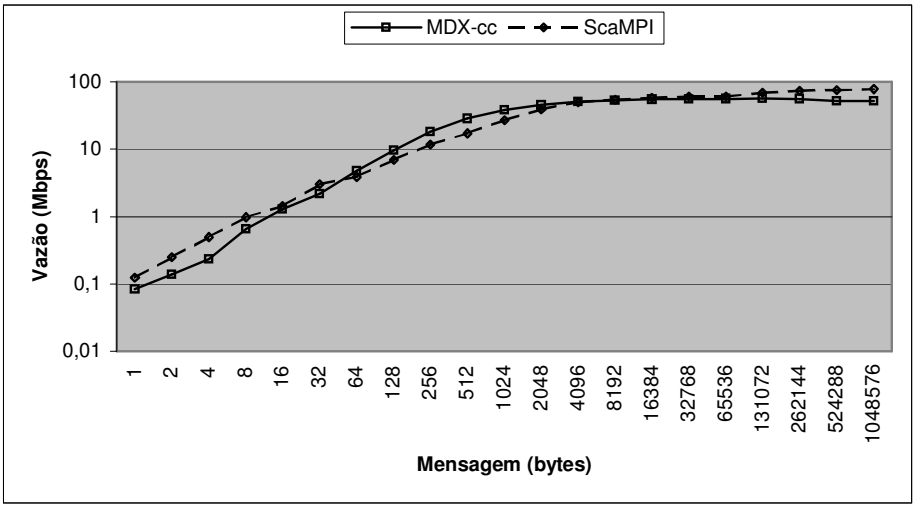

Figura 7.12: Gráfico de vazão do MDX-cc sobre SCI.

7.6. MDX-cc sobre *Cluster* de *Clusters*

Para avaliar o comportamento do MDX-cc sobre uma estrutura de *cluster* de *clusters*, com uma aplicação envolvendo todos os agregados disponíveis, foi implementado um protótipo baseado na aplicação descrita pelo Algoritmo 5.4. Esta aplicação passa uma mensagem entre os processos, formando um anel, onde a mensagem é enviada pelo processo mestre, com identificador 0, para o processo 1, este repassa a mensagem para o processo 2, que repassa ao processo 3, e assim sucessivamente, até que a mensagem chegue ao processo com maior identificador. Este processo devolve a mensagem ao processo mestre.

Esta aplicação não avalia o desempenho do MDX-cc, mas sim o funcionamento da ferramenta sobre uma estrutura heterogênea, avaliando o comportamento do protocolo de comunicação implementado.

Para a execução da aplicação, foram alocados quatro nós do *cluster* Amazonia (amazonia01, amazonia02, amazonia03 e amazonia04), quatro nós do agregado Tropical (tropical01, tropical02, tropical03 e tropical04), dois nós do *cluster* Ombrofila (ombrofila01 e ombrofila02) e dois nós do *cluster* Pantanal (pantanal01 e pantanal02). Não foram utilizados todos os nós de todas as máquinas a fim de facilitar a ilustração da execução. O arquivo de nomes de máquinas ficou configurado conforme mostra a Figura 7.13.

```
amazonia01.cpad.pucrs.br
amazonia02.cpad.pucrs.br
amazonia03.cpad.pucrs.br
amazonia04.cpad.pucrs.br
tropical01.cpad.pucrs.br
tropical02.cpad.pucrs.br
tropical03.cpad.pucrs.br
tropical04.cpad.pucrs.br
ombrofila01.cpad.pucrs.br
ombrofila02.cpad.pucrs.br
pantanal01.cpad.pucrs.br
pantanal02.cpad.pucrs.br
```

Figura 7.13: Arquivo de nomes de máquinas.

Disparada a aplicação, com 12 processos, estes ficaram distribuídos da seguinte maneira:

```
processo 0 - amazonia01.cpad.pucrs.br
processo 1 - amazonia02.cpad.pucrs.br
processo 2 - amazonia03.cpad.pucrs.br
processo 3 - tropical01.cpad.pucrs.br
processo 4 - tropical02.cpad.pucrs.br
processo 5- ombrofila01.cpad.pucrs.br
processo 6 - ombrofila02.cpad.pucrs.br
processo 7 - tropical03.cpad.pucrs.br
processo 8 - tropical04.cpad.pucrs.br
processo 9 - pantanal01.cpad.pucrs.br
processo 10 - pantanal02.cpad.pucrs.br
processo 11 - amazonia04.cpad.pucrs.br
```

Figura 7.14: Distribuição dos processos nos nós disponíveis.

A troca de mensagens entre cada par de processos ocorreu através da tecnologia selecionada pelo protocolo de análise apresentado na seção 6.2.2. Não foi passada preferência nenhuma nas primitivas de comunicação. Com base na rede utilizada em cada troca de mensagem, pode-se montar o gráfico apresentado pela Figura 7.15.

A comunicação entre os processos 0 e 1, e 1 e 2, se dá através de rede *Myrinet*, uma vez que ambos os nós dispõem dessa tecnologia. Os processos 2 e 3 se encontram em nós de agregados diferentes (Amazonia e Tropical), onde o agregado Amazonia dispõe de redes *Myrinet* e *Fast-Ethernet*, e o Tropical dispõe somente de rede *Fast-Ethernet*. Desta forma, esse processos se comunicam por *Fast-Ethernet*/TCP. O mesmo ocorre para a comunicação entre os processos 8 e 9. A comunicação entre os processos 3 e 4 se dá através da única tecnologia disponível, que é *Fast-Ethernet*, por protocolo TCP.

Assim como o *cluster* Tropical, o Ombrofila também utiliza somente rede *Fast-Ethernet*. Consequentemente, a comunicação entre os processos 4 e 5, 5 e 6, 6 e 7, e 7 e 8 ocorre através dessa tecnologia. Os processos 9 e 10 estão localizados em nós de *clusters* diferentes, Amazonia e Pantanal. O *cluster* Pantanal utiliza redes SCI e *Fast-Ethernet*, enquanto o agregado Amazonia dispõe de redes *Myrinet* e *Fast-Ethernet*. A comunicação entre os dois processos se dá através da tecnologia *Fast-Ethernet*/TCP, comum a ambos os agregados. O mesmo ocorre para a troca de mensagem entre os processos 11 e 0. Entre os processos 10 e 11, ambos localizados no *cluster* Pantanal, se dá através de rede SCI.

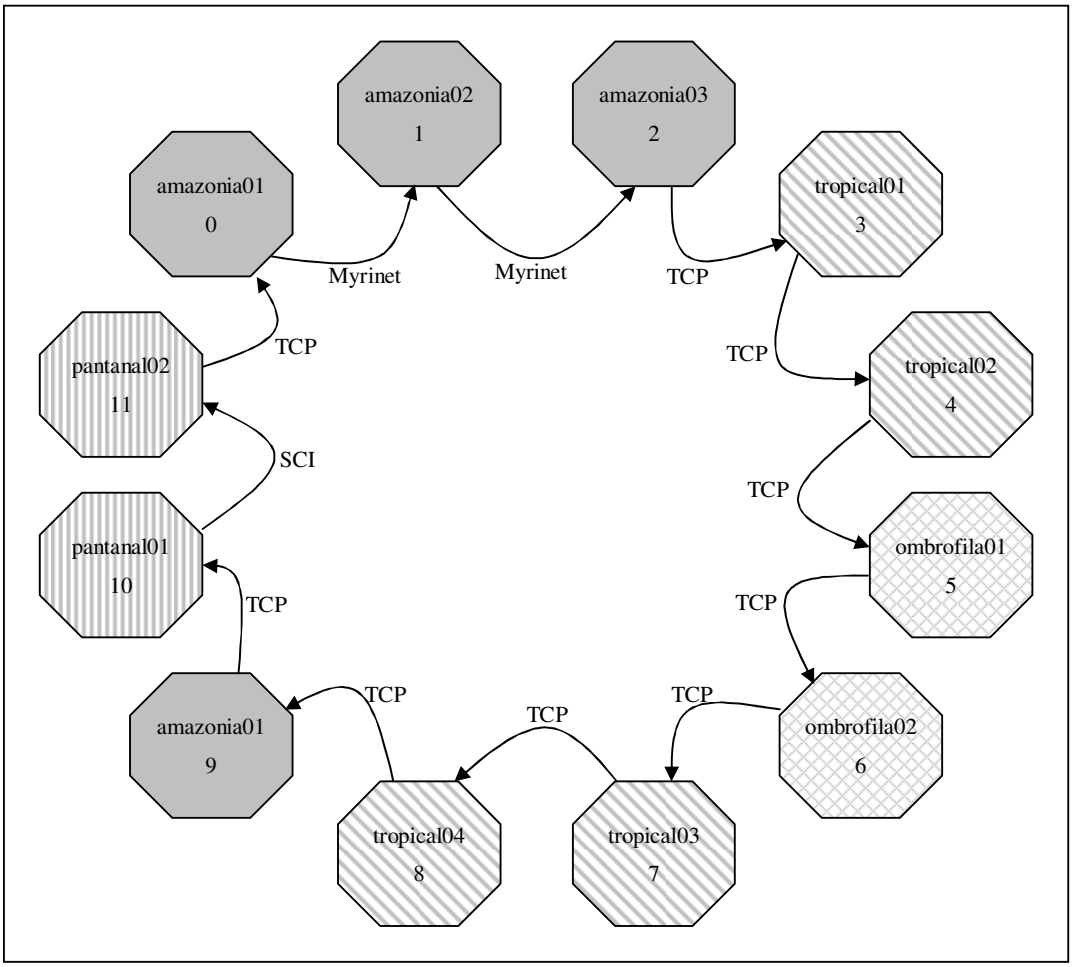

Figura 7.15: Tecnologia usada na comunicação entre processos.

A execução desta aplicação mostra que o MDX-cc facilita a utilização de uma estrutura de *cluster* de *clusters*, uma vez que gerencia a comunicação entre processos localizados em agregados diferentes. Além disso, torna mais fácil a programação, pois oferece

um protocolo de comunicação que decide através de qual tecnologia de rede acontece a troca de mensagens Desta forma, o usuário não se preocupa com a localização dos processos nos nós da estrutura.

Capítulo 8

Conclusão

Nos dias de hoje, é crescente a busca por alto desempenho na execução de aplicações, principalmente nas áreas da ciência e tecnologia. Nesse contexto, os *clusters* surgem como alternativa viável na construção de máquinas com maior poder computacional.

Em função do surgimento de redes de comunicação extremamente rápidas, surge a possibilidade da interligação de agregados, formando uma estrutura de *cluster* de *clusters*. A interligação de *clusters* visa melhorar o desempenho na execução de aplicações paralelas, através do aumento do número de máquinas interligadas.

Um dos principais problemas no uso de *cluster* de *clusters* é o *software* utilizado para o desenvolvimento de aplicações paralelas, visto que cada agregado envolvido na estrutura possui certas características que precisam ser tratadas pela linguagem ou ambiente de programação, visando o alcance de alto desempenho.

Esta dissertação apresentou um ambiente de programação paralela aplicado a *cluster* de *clusters*, o MDX-cc. Foram apresentados detalhes de projeto e de implementação da ferramenta, as primitivas oferecidas pelo ambiente e exemplos de programas. Também foram apresentados os testes de desempenho realizados com o MDX-cc, e os números obtidos nos testes foram interpretados.

8.1. Dificuldades Encontradas

Alguns problemas surgiram no decorrer do trabalho. A necessidade de modificações na estrutura do MDX original pode ser citada como o principal deles. Outros problemas que podem ser citados são os encontrados na documentação e na utilização da biblioteca GM, além da necessidade de correção de alguns *bugs* da biblioteca YAMPI.

A implementação da comunicação sobre rede *Myrinet* apresentou alguns obstáculos. A biblioteca GM, utilizada pelo MDX-cc, apresenta um modelo de programação complicado, com a utilização de *tokens* de acesso à rede e de eventos de recebimento de mensagens. A documentação disponível sobre a biblioteca é bastante incompleta, o que dificultou a implementação das funções de comunicação do MDX-cc via rede *Myrinet*.

A biblioteca YAMPI, pelo fato de estar em fase inicial de desenvolvimento, apresentou alguns problemas na troca de mensagens longas. Este fato atrasou um pouco o desenvolvimento das funções de comunicação via rede SCI. Após a correção desses *bugs*, a implementação do MDX-cc seguiu seu curso normal.

Apesar das dificuldades citadas, o trabalho obteve resultados interessantes. Esses resultados são apresentados a seguir.

8.2. Resultados Obtidos

Pode-se dizer que o trabalho atingiu seu objetivo, uma vez que o resultado do mesmo é uma primeira versão do MDX-cc já implementada e funcionando de forma estável. O principal objetivo da ferramenta proposta, que era de oferecer ao programador transparência total na comunicação entre processos, independentemente das tecnologias de rede disponíveis a cada um deles, foi alcançado. No MDX-cc, o usuário faz a troca de mensagens entre processos através de chamadas *MDX_Send( )* e *MDX_Recv( )*. Essas primitivas executam o protocolo que decide através de qual tecnologia a mensagem será enviada ou recebida. Desta forma, o usuário não precisa se preocupar com a localização dos processos de uma aplicação paralela e com as tecnologias de rede disponíveis a cada um deles.

Os testes de desempenho mostraram que o MDX-cc apresenta um desempenho satisfatório para comunicação por rede *Fast-Ethernet* (seção 7.3) e por rede SCI (seção 7.5). Na comunicação por redes *Myrinet*, o MDX-cc apresentou uma performance abaixo da esperada (seção 7.4). Esse fato se deve às cópias de mensagem realizadas pelas funções *MDX_GM_Send( )* e *MDX_GM_Recv( )*.

Em todos os testes de desempenho, a latência da primeira troca de mensagem é elevada. Esse fato era esperado e ocorre devido ao mecanismo de negociação entre os processos, que necessita buscar as informações do processo par no Servidor de Comunicação (Capítulo 6).

Finalmente, pode-se concluir que o MDX-cc é uma ferramenta interessante no auxílio à programação paralela sobre *cluster* de *clusters*. Esse trabalho representou apenas o primeiro passo no desenvolvimento do ambiente, com o projeto e a implementação de uma primeira versão do mesmo. Com base nos resultados dessa dissertação, possibilidades de trabalhos futuros aparecem e são apresentadas a seguir.

8.3. Trabalhos Futuros

Apesar de ter obtido um resultado satisfatório, esta dissertação pode sugerir alguns trabalhos a serem desenvolvidos no futuro. Os serviços propostos, mas não implementados nesse trabalho, podem ser citados como exemplo.

Pode ser sugerido o desenvolvimento de um serviço de memória compartilhada entre processos. Esse serviço pode utilizar, para processos localizados em um mesmo agregado com rede SCI, os mecanismos de memória compartilhada oferecidos pelo *hardware* SCI. Para processos executando em nós com outras tecnologias, um modelo de memória virtual distribuída pode ser implementado, como, por exemplo, o modelo Linda [HES99].

A fim de alcançar um maior desempenho para as aplicações MDX-cc, um mecanismo de balanceamento de carga pode ser estudado e desenvolvido. Esse mecanismo, além de evitar a sobrecarga de nós, pode levar em consideração o *hardware* de cada um dos agregados da estrutura, alocando processos com maior número de chamadas a primitivas de comunicação em agregados com rede mais rápida.

A fim de tornar a comunicação no MDX-cc mais completa, podem ser estudados mecanismos de comunicação em grupo. Podem ser desenvolvidas primitivas de criação de grupos de processos e de envio de mensagens a determinado grupo de processos (ou a todos os processos da aplicação). Além disso, uma primitiva de recebimento genérica, ou seja, de mensagens vindas de qualquer outro processo, também pode ser implementada.

Além desses trabalhos de implementação dos serviços já propostos nesse trabalho, podem ser sugeridos também trabalhos de otimização do MDX-cc. O estudo de outro mecanismo de negociação entre processos e a implementação de comunicação por rede

Myrinet sem cópias de mensagem não podem ser descartados e tornariam o MDX-cc mais completo e interessante.

Referências Bibliográficas

[ALM89] ALMASI, G. S.; GOTTLIEB, A. **Higly Parallel Computing**. Benjamin/Cummings Publ. Comp., Redwood City, Cal., 1989.

[AMA95] AMARASINGHE, S. P.; ANDERSON, J. M.; LAM, M. S.; TSENG, C. W. **The SUIF Compiler for Scalable Parallel Machines**. In the Seventh SIAM Conference on Parallel Processing for Scientific Computing. Proceedings. Feb. 1995.

[AUM00] AUMAGE, O.; BOUGÉ, L.; NAMYST, R. **A Portable and Adaptative Multi-Protocol Communication Library for Multithreaded Runtime Systems**. In Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00), vol. 1800 of Lecture Notes in Computer Science, pp. 1136-1143, Cancun, Mexico. Mai. 2000.

[BAL92] BAL, H. E. et al. **Orca: a language for parallel programming for distributed systems**. IEEE Trans. On Software Engineering. v. 18, n. 3, 1992.

[BAR00] BARRETO, M. **DECK: Um ambiente para programação paralela em agregados de multiprocessadores.** Dissertação de Mestrado, PPGC/UFRGS, Porto Alegre, 2000.

[BAR00a] BARRETO, M.; ÁVILA, R.; NAVAUX, P. **The Multicluster model to the integrated use of multiple workstation clusters.** In: WORKSHOP ON PERSONAL COMPUTER BASED NETWORKS OF WORKSTATIONS, 3., 2000, Cancun. Proceedings... Berlin: Springer-Verlag, 2000. pp.71-80. (Lecture Notes in Computer Science, v.1800).

[BOD95] BODEN, N. et al. **Myrinet: a gigabit-per-second local-area network**. IEEE Micro, Los Alamitos, v.15, n.1, p.29--36, 1995.

[BUR94] BURNS, G.; DAOUD, R.; VAIGL, J. **LAM: An open cluster environment for MPI**. In: Supercomputing Symposium '94. Proceedings… pp. 379-386, University of Toronto. 1994.

[BUT98] BUTENUTH, R.; HEISS, H. **Shared memory programming on pc-based sci clusters**. In: Emmsec 98 European Multimedia, Microprocessor Systems and Eletronic Commerce Conference and Exposition Bordeaux, 1998. Proceedings... 1998.

[BUT99] BUTENUTH, R.; HEISS, H. **Interfacing sci drivers to linux**. SCI Book, 1999.

[BUY99] BUYYA, R.. **High Performance Cluster Computing.** Prentice-Hall PTR, Upper Saddle River, NJ, 1999.

[CHR96] CHRISTALLER, M. **Athapascan-0: vers un support exécutif pour applications parallèles irréfurières efficacement portables**. PhD thesis. Université Joseph Fourier, Grenoble, Fr. Nov. 1996.

[COM95] COMER, D. E. **Internetworking with TCP/IP**, Volume 1, Prentice Hall, 1995.

[COP00] COPETTI, A. **MDX-v1: Implantação e Avaliação em redes ATM**. Dissertação de Mestrado. Mestrado em Ciência da Computação. PUCRS. Porto Alegre, 2000.

[CUL99] CULLER, D. E.; SINGH, J. P. **Parallel computer architecture: a hardware / software approach**. EUA: Morgan Kaufmann Publishers, Inc., 1999.

[DER02] DE ROSE, C. A. F.; PASIN, M. **Fundamentos de processamento de alto desempenho**. In: II Escola Regional de Alto Desempenho (ERAD'2001), 2002. Anais... 2002.

[DOS93] DOSS, N. E.; GROPP, W.; LUSK, E.; SKJELLUM, A. **An initial implementation of MPI**. Tech. Rep. MCS-P393-1193, Mathematics and Computer Science Division. Argonne National Laboratory, Argonne, IL 60439. 1993.

[EIC92] EICKEN, T.; CULLER, D; GOLDSTEIN, S.; SCHAUSER, K. **Active Messages: A Mechanism for Integrated Communication and Computation**. 19th Annual Int'l Symp. on Computer Architecture (ISCA'92). Proceedings… May 1992.

[FLY72] FLYNN, M. J. **Some Computer Organizations and their Effectiveness**. IEEE Transaction on Computers 21, v. 21, n. 9, pp. 948-960, 1972.

[FOS95] FOSTER, I. **Designing and Building Parallel Programs**. *In* Addison-Wesley Publishing Company, Inc. 1995.

[GEH86] GEHANI, N. H.; ROOME, W. D. **Concurrent C**. Software Practice and Experience, v. 16, n. 9, pp. 821-844. Sept. 1986.

[GEH92] GEHANI, N. H.; ROOME, W. D. **Implementing Concurrent C**. Software Practice and Experience, v. 22, n. 3, pp. 267-285. Mar. 1992.

[GEI96] GEIST, A.; BEGUELIM, A.; DONGARRA, J.; JEANG, W.; MANCHEK, R.; SUNDERAM, V. **PVM – Parallel Virtual Machine – a users guide and tutorial for networked parallel computing**. MIT Press, Cambridge, England. 1996.

[GM01] GM Home Page. The GM Message Passing System. Disponível em http://www.myri.com/scs/GM/doc/gm_toc.html. Capturado em 2001.

[HAL96] HALL, M. W. et al. **Maximizing Multiprocessor Performance with the SUIF Compiler**. IEEE Computer, Dec. 1996.

[HAN98] HAENDEL, M.; HUBER, M.; SHROEDER, S. **ATM Networks: Concepts Protocol Applications**. Addison-Wesley, 1998.

[HEL98] HELLWAGNER, H.; REINEFELD, H. **Sclable coherent interface: technology and applications**. In: SCI-EUROPE98, 1998. Proceedings... 1998.

[HES99] HESS, C. R.; SILVA, S. B.; KNÜPPE, G. **Concepção e Implementação de Memória Virtual Distribuída no Sistema MDX**. Trabalho de conclusão de graduação do curso de Informática, PUCRS, Porto Alegre, Dezembro, 1999.

[HES01] HESS, C. R. **Avaliação de Desempenho e Otimização do Sistema MDX**. Trabalho Individual I. Mestrado em Ciência da Computação. PUCRS, 2001.

[HOC88] HOCKNEY, R. W.; JESSHOPE, C. R. **Parallel Computers 2**. Adam Hilger, Bristol and Philadelfia, 1988.

[HWA93] HWANG, K. **Advanced Computer Architecture: Parallelism, Scalability, Programmability**. McGraw-Hill. 1993.

[HWA97] HWANG, K.; XU, Z.. **Scalable parallel computing: technology, architecture, programming**. New York, NY: McGraw-Hill, 1997.

[IEE92] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **IEEE standard for Scalable Coherent Interface (SCI)**. IEEE 1596-1992. New York, NY, 1992.

[IEE95] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **Information technology – portable operating system interface (POSIX), threads extension [C language]**. IEEE 1003.1c-1995. New York, NY, 1995.

[IEE95a] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **Local and metropolitan area networks-supplement---media access control (MAC) parameters, physical layer, medium attachment units and repeater for 100Mb/s operation, type 100BASE-T (clauses 21-30)**. IEEE 802.3u-1995. New York, NY, 1995.

[KEL94] KELEHER, P.; DWARKADAS, S.; COX, A.; ZWAENEPOEL, W. **Treadmarks: Distributed shared memory on standard workstation and operating systems**. 1994 Winter Usenix Conference, pages 115-131. Proceedings… January 1994.

[KRI92] KRISTIANSEN, B.; HULAAS, J. **Behavior of Scalable Coherent Interface in larger systems**. CAMAC 1992. Proceedings…

[LI86] LI, K. **Shared Virtual Memory on Loosely Coupled Multiprocessors**. PhD thesis, Yale University. Set. 1986.

[LIB01] LIBRELOTTO, G. R. **Um Compilador para a Linguagem RS Distribuída**. Dissertação de Mestrado UFRGS. 2001.

[MAI98] MAI, G. C.; DE ROSE, C. A. F.. **Arquiteturas paralelas versáteis e de baixo custo para a pesquisa e o ensino na área de processamento paralelo e**

distribuído. In: Conferencia Latino Americana de Informatica, 1998. Anais. . . 1998. p.757-768.

[MES94] Message Passing Interface Fórum. **MPI: A message passing interface standard, version 1.0**. Mai. 1994.

[MYR00] MYRICOM. Myricom homepage. Disponível em http://www.myri.com Capturado em 2001.

[NOV01] NOVAES, R. **YAMPI: uma biblioteca de troca de mensagens para agregados conectados por tecnologia SCI**. Dissertação de Mestrado. Mestrado em Ciência da Computação. PUCRS. Porto Alegre. 2001.

[PAK97] PAKIN, S.; KARAMCHETI, V.; CHIEN, A. **Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors**. IEEE Concurrency. Vol. 5. No 2. pp. 60-73. 1997

[PAT94] PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization & Design: The Hardware Software Interface**. Morgan Kaufmann Publishers, Inc. 1994.

[PRE98] PREUSS, E. **MDX: Um ambiente de programação paralela baseado em memória Virtual Distribuída**. Dissertação de Mestrado, Mestrado em Ciências da Computação. UFRGS, 1998.

[PRY98] PRYLLI, L.; TOURANCHEAU, B. **BIP: A new protocol designed for high-performance networking on Myrinet**. In José Rolin, editor, *Parallel and Distributing Processing*, number 1388 in Lecture Notes in Computer Science, pp.472-485, 1998.

[REH99] REHLING, E. **Multithreading for SCI clusters**. Eleventh Symposium on Computer Architecture and High Performance Computing, Natal-RN, 1999. Proceedings…

[SAL99] SALES, A. H.; CUNHA, J. N., BARRIONUEVO, R. **TCX – Ferramenta de programação paralela baseada em troca de mensagens**. Trabalho de conclusão de graduação do curso de Informática, PUCRS, Porto Alegre. Julho, 1999.

[SCA00] Scali homepage. Scalable Linux Systems – affordable supercomputing. Disponível em http://www.scali.com Capturado em 2001.

[SEI95] SEITZ, C. L. **Myrinet - a gigabit-per-second local-area-network**. IEEE Micro, v.15, n.1, feb 1995.

[SOA95] SOARES, L. F.; LEMOS, G.; COLCHER, S. **Redes de Computadores – das LANs, MANs e WANs às Redes ATM**. Editora Campus, 1995.

[SQU00] SQUYRES, J. M.; LUMSDAINE, A.; GEORGE, W. L.; HAGEDORN, J. G.; DEVANEY, J. E. **The Interoperable Message Passing Interface (IMPI) Extensions to LAM/MPI**. MPIDC 2000, MPI Developers and Users Conference. Cornell University, Ithaca, NY. Mar. 2000

[STE99] STERLING, T. L.; SALMON, J.; BECKER, D. J.; SAVARESE, D. F. **How to build a beowulf. a guide to the implementation an application of pc clusters**. EUA: Massachusetts Institute of Technology, 1999.

[TAN92] TANENBAUM, A. S. **Modern Operating Systems**. Englewood Clifs, NJ. Prentice-Hall, 1992.

[TAN97] TANENBAUM, A. S. **Redes de computadores**. Brasil: Editora Campus, 1997.

[TAS99] TASKIN, H. **Sinchronizationsoperationen für gemeinsamen Speicher in SCI-clustern**. Disponível em http://www.via.org. Capturado em 2001.

[TOP01] TOP500. Top500 supercomputer site. Disponível em http://www.top500.org Capturado em 2001.