Avaliação das Rotinas de Comunicação Ponto-a-Ponto do MPI

Márcio Augusto de Souza

Orientadora: Profa. Dra. Regina Helena Carlucci Santana

Dissertação apresentada ao Instituto de Ciências Matemáticas de São Carlos da Universidade de São Paulo - ICMSC/USP, como parte dos requisitos para a obtenção do título de Mestre em Ciências - Área: Ciências de Computação e Matemática Computacional.

USP - São Carlos
Dezembro/1996

Este trabalho é dedicado a José Augusto e Marly,

meus pais, pois sem a confiança e o apoio a mim dedicados,

nada disso seria possível.

Agradecimentos

À Profa. Dra. Regina Helena C. Santana, pela orientação, amizade e paciência.

Ao Prof. Dr. Marcos José Santana, pela amizade e pela boa vontade demonstrada quando minha ficha de inscrição se extraviou.

Aos amigos Paulo Sérgio e Flávio, pelas contribuições inestimáveis feitas a este trabalho.

À Luiza, pelo carinho, apoio e colaboração na análise estatística.

Ao Prof. Dr. Onofre Trindade Júnior, por liberar as fontes do LINUX, me possibilitando economizar um belo trabalho.

A todos os amigos que eu fiz durante mestrado, principalmente ao pessoal do grupo de SD. Sem a alegria e companheirismo que me foram dedicados, meu caminho seria muito mais difícil.

À CNPq, pelo apoio financeiro.

E sobretudo, aos meus pais, que tornaram possíveis todas as minhas conquistas.

Resumo

O MPI é uma tentativa de padronização para ambientes de programação via troca de mensagens, tendo como objetivo portabilidade e eficiência em qualquer plataforma.

O requisito de alta portabilidade sem perda de eficiência torna o MPI um padrão extenso. Rotinas de comunicação ponto-a-ponto, por exemplo, são estruturadas de várias maneiras, apresentando diferentes desempenhos.

O objetivo deste trabalho é estudar o desempenho das rotinas de comunicação ponto-a-ponto do MPI, em uma rede de computadores LINUX, a fim de avaliar objetivamente a relação custo/benefício de cada uma.

A avaliação é feita através da execução de *benchmarks* e um exemplo de aplicação, executados em três implementações de domínio público do MPI (MPICH, LAM e UNIFY), permitindo a comparação dessas implementações. Resultados obtidos com a utilização do PVM também são incluídos e comparados aos do MPI, visto que o PVM é uma plataforma de programação via troca de mensagens muito difundida na comunidade computacional.

A apresentação clara e concisa dos aspectos fundamentais das diferentes formas de comunicação disponíveis em diferentes implementações do MPI e a avaliação de desempenho efetuada, que é capaz de orientar o usuário final na escolha de uma implementação do MPI, bem como da forma de comunicação mais adequada à sua aplicação, são contribuições importantes dessa dissertação.

Abstract

The MPI is an attempt of standardization for message-passing programming environments, aiming high portability and efficiency in any platform.

The requirement of high portability without loss of efficiency makes the MPI an extensive standard. Point-to-point communication routines, for instance, are structured in many ways, issuing different performance.

This work aims to study the performance of MPI point-to-point communications routines, providing objective results about the cost/benefit relation of each routine.

This evaluation is performed by means of executing benchmarks and an application example, executed on three MPI public domain implementations (MPICH, LAM e UNIFY), also allowing the comparison of the implementations. Results obtained from PVM are included and compared to those from MPI, once PVM is widely used by the computational community.

The clear and concise presentation of fundamental issues for the different MPI communication modes available on different MPI implementations, together with the performance evaluation developed, that is able to orient the final user in the choice of a given MPI implementation as well as the communication mode suitable to its application, are important contributions of this dissertation.

Conteúdo

Capítulo 1. Introdução 1

Capítulo 2. Computação Paralela 5

2.1. Introdução 5
2.2. Conceitos Básicos 6
2.2.1. Tipos de Paralelismo *6*
2.2.2. Granularidade *9*
2.2.3. Speedup e Eficiência *9*
2.3. Classificação de Arquiteturas Paralelas 10
2.3.1. Classificação de Flynn *11*
2.3.2. Classificação de Duncan *14*
2.4. Programação Concorrente 19
2.4.1. Projeto de um Algoritmo Paralelo *20*
2.4.2. Ativação de Processos Paralelos *21*
2.4.3. Comunicação e Sincronismo *24*
2.4.4. Suporte Para Programação Paralela *28*
2.5. Considerações Finais 30

Capítulo 3. Sistemas Distribuídos 32

3.1. Introdução 32
3.2. Definição de um Sistema Distribuído 33
3.3. Vantagens/Desvantagens de Sistemas Distribuídos 35
3.4. Características Básicas de Sistemas Distribuídos 36
3.5. Comunicação em Sistemas Distribuídos 43
3.5.1. Comunicação Cliente-Servidor *43*
3.5.2. Comunicação de Grupo *45*
3.6. Considerações Finais 45

Capítulo 4. Programação via Troca de Mensagens 47

4.1. Introdução 47
4.2. Ambientes de Programação via Troca de Mensagens 48
4.3. Biblioteca de Troca de Mensagens 50
4.3.1. O Que é Uma Mensagem? *50*
4.3.2. Rotinas de Comunicação Ponto-a-Ponto *52*
4.3.3. Rotinas coletivas *53*
4.3.4. Rotina Probe() *55*
4.3.5. Rotinas Informativas *55*
4.4. MPI - *Message Passing Interface* 55
4.4.1. A História do MPI *57*

4.4.2. Especificação do MPI ... *57*
4.4.3. Comunicação no MPI ... *59*
4.4.4. Programa Exemplo ... *61*
4.4.5. Comentários Finais Sobre o MPI ... *62*
4.5. Considerações Finais ... 63

Capítulo 5. Implementações de Plataformas de Portabilidade ... 65

5.1. Introdução ... 65
5.2. O Ambiente Computacional ... 66
5.2.1. O Sistema LINUX ... *67*
5.2.2. Plataformas de Portabilidade ... *69*
5.3. MPICH ... 69
5.3.1. Arquitetura do MPICH ... *70*
5.3.2. MPICH e LINUX ... *72*
5.3.3. Características Gerais do MPICH ... *73*
5.3.4. Ferramentas Adicionais ... *74*
5.4. LAM ... 74
5.4.1. Arquitetura do LAM ... *75*
5.4.2. Daemons e Depuração de Programas ... *76*
5.4.3. Características Gerais do LAM ... *76*
5.4.4. Ferramentas Adicionais ... *77*
5.5. PVM ... 77
5.5.1. Arquitetura do PVM ... *79*
5.5.2. Comunicação Entre Processos no PVM ... *79*
5.6. UNIFY ... 80
5.6.1. Arquitetura do UNIFY ... *80*
5.6.2. Características Gerais do UNIFY ... *80*
5.7. Comparando as Implementações MPI ... 81
5.8. PVM ou MPI? ... 82
5.9. Considerações Finais ... 84

Capítulo 6. Comunicação Entre Processos no MPI ... 85

6.1. Introdução ... 85
6.2. Comunicação Ponto-a-Ponto ... 86
6.2.1. Comunicação Bloqueante X Não Bloqueante ... *87*
6.2.2. Modos de Comunicação ... *88*
6.2.3. Requisições Persistentes ... *93*
6.2.4. Comunicação em Dois Sentidos ... *93*
6.3. Resumo das Rotinas de Comunicação do MPI ... 94
6.4. Considerações Finais ... 96

Capítulo 7. Modelos de Avaliação e Métricas de Desempenho ... 97

7.1. Introdução ... 97

7.2. Modelo Computacional de Avaliação 98
7.3. Métricas de Desempenho 100
7.3.1. Tamanho das Mensagens *102*
7.4. *Benchmarks* Considerados 103
7.5. Algoritmo Paralelo de Jacobi 105
7.6. Considerações Finais 108

Capítulo 8. Avaliação de Desempenho 109

8.1. Introdução 109
8.2. Requisitos Gerais da Avaliação 110
8.2.1. Configuração de Hardware *110*
8.2.2. Configuração das Plataformas de Portabilidade *111*
8.2.3. Configuração dos Benchmarks *111*
8.2.4. Apresentação Gráfica *112*
8.2.5. Análise Estatística *112*
8.3. Comunicação Ponto-a-Ponto 113
8.3.1. Rotinas Bloqueantes *113*
8.3.2. Rotinas Não Bloqueantes *120*
8.3.3. Rotinas de Comunicação nos Dois Sentidos *122*
8.3.4. Requisições Persistentes *123*
8.4. Implementações de Plataformas de Portabilidade 124
8.5. Algoritmo de Jacobi 127
8.6. Considerações Finais 129

Capítulo 9. Conclusões 133

9.1. Considerações Iniciais 133
9.2. Conclusões 133
9.3. Contribuições Deste Trabalho 134
9.4. Dificuldades Encontradas 135
9.5. Trabalhos Futuros 135

Referências Bibliográficas 137

Apêndice A. Análise Estatística 143

Índice de Figuras

Figura 2.1 - Paralelismo Lógico. 7
Figura 2.2 - Paralelismo Físico. 7
Figura 2.3 - *Pipeline*. (a) Organização dos Estágios (b) Gráfico de Execução. 8
Figura 2.4 - Modelo Computacional SISD. 11
Figura 2.5 - Modelo Computacional SIMD. 12
Figura 2.6 - Modelo Computacional MISD. 12
Figura 2.7 - Modelo Computacional MIMD. 13
Figura 2.8 - Classificação de Duncan. 14
Figura 2.9 - Arquiteturas MIMD. (a) Mem. Centralizada (b) Mem. Distribuída. 16
Figura 2.10 - Exemplo da Utilização de Corotinas 21
Figura 2.11 - Exemplo da Utilização de *Fork/Join*. 22
Figura 2.12 - Exemplo da Utilização de *Cobegin/Coend*. 23
Figura 2.13 - Exemplo da Utilização de *Doall*. 23
Figura 2.14 - *Send/Receive*. (a) Síncrono (b) Assíncrono. 26
Figura 2.15 - Mecanismos de Comunicação. (a) Ponto-a-Ponto (b) *Rendezvouz*. 27
Figura 2.16 - RPC. 28
Figura 3.1 - Modelo Simples de um Sistema Distribuído. 34
Figura 3.2 - O Mecanismo RPC. 44
Figura 4.1 - Transferência de uma Mensagem. 51
Figura 4.2 - Variações Sobre a Rotina B*roadcast()*. 61
Figura 5.1 - Estrutura do MPICH Sobre o LINUX. 71
Figura 5.2 - Estrutura do MPICH Sobre o Dispositivo ch_p4. 72
Figura 5.3 - Estrutura de *Software* do LAM. 75
Figura 6.1 - Comunicação Ponto-a-Ponto a) Bloqueante b) Não Bloqueante. 87
Figura 6.2 - Semânticas de Comunicação a) *Bufferizada* b) Síncrona. 89
Figura 6.3 - Estrutura de Comunicação Insegura. 90
Figura 6.4 - Estrutura de Comunicação Segura. 91
Figura 7.1 - Modelo de Avaliação. 98
Figura 7.2 - a) Modelo de Comunicação *Ping* b) Algoritmo Simples. 103
Figura 7.3 - a) Modelo de Comunicação *Ping-Pong* b) Algoritmo Simples. 104
Figura 7.4 - Problema de Distribuição de Temperatura. 106
Figura 7.5 - Particionamento de uma Matriz em Blocos. 106
Figura 7.6 - Algoritmo Simplificado de Jacobi. 107

Indíce de Tabelas

Tabela 6.1 - Rotinas de Comunicação no MPI 95
Tabela 6.2 - Rotinas Auxiliares Para Comunicação Não Bloqueante 95
Tabela 8.1 - Comunicação Bloqueante X Não Bloqueante 127
Tabela 8.2 - Diferenças Entre os Modos de Comunicação 128
Tabela 8.3 - Diferenças Entre as Implementações 128
Tabela 8.4 - Tabela de Comparação Entre as Implementações MPI 131

Índice de Gráficos

Gráfico 8.1 - Latências da Biblioteca e Ponto-a-Ponto - Mensagens pequenas ... 114
Gráfico 8.2 - Latências da Biblioteca e Ponto-a-Ponto - Mensagens Grandes..... 115
Gráfico 8.3 - Latência da Biblioteca no LAM ... 116
Gráfico 8.4 - *Throughput* no LAM ... 116
Gráfico 8.5 - *Throughput* para Mensagens Pequenas no MPICH ... 117
Gráfico 8.6 - *Throughput* para Mensagens Grandes no MPICH ... 118
Gráfico 8.7 - *Throughput* para Mensagens Pequenas no LAM ... 118
Gráfico 8.8 - *Throughput* para Mensagens Grandes no LAM ... 119
Gráfico 8.9 - *Throughput* Duplo Para Mensagens Pequenas no LAM ... 122
Gráfico 8.10 - *Throughput* Duplo Para Mensagens Grandes no LAM ... 123
Gráfico 8.11 - Latência da Biblioteca para Mensagens Pequenas no MPICH 124
Gráfico 8.12 - Latência de Biblioteca - Mensagens Pequenas ... 125
Gráfico 8.13 - *Throughput* - Mensagens Pequenas ... 125
Gráfico 8.14 - Latência da Biblioteca - Mensagens Grandes ... 126
Gráfico 8.15 - *Throughput* - Mensagens Grandes ... 126

Capítulo 1

Introdução

A computação, nos últimos anos, tem sofrido profundas transformações, resultando numa grande evolução em um tempo relativamente pequeno. Processadores tornam-se mais rápidos, a capacidade da memória aumenta, redes transmitem maior volume de dados mais rapidamente, e todo esse progresso é acompanhado por uma gradual diminuição de custo, tornando a tecnologia computacional acessível e aumentando o leque de aplicações que podem ser computacionalmente resolvidas de maneira eficiente [TUR93].

Uma área de pesquisa que apresentou grande avanço junto ao progresso tecnológico foi a computação paralela. Apesar da idéia de processar informações paralelamente ser antiga, só nos últimos anos começou realmente a se desenvolver, sendo que atualmente esta área é alvo de intenso estudo dentro da comunidade acadêmica [ALM94] [COR96] [KIR91] [NAV89]. A expansão da computação paralela está atingindo a comunidade industrial e comercial, que enxerga muitas vantagens na ótima relação custo/desempenho que esta tem a oferecer [LEN95].

Um problema relacionado à computação paralela é o alto custo de aquisição e manutenção de arquiteturas paralelas. Além disso, a compra de uma arquitetura geralmente implica na dependência do comprador ao fabricante. Uma tendência atual, neste contexto, é a utilização de sistemas distribuídos como plataformas de execução paralela, a fim de que se forneça menor custo de implantação e maior flexibilidade no processo computacional paralelo [BEG94] [BLE94] [COR96] [ZAL91].

A computação paralela e a computação distribuída surgiram por motivos diferentes. A necessidade de se compartilhar recursos motivou o uso de sistemas distribuídos, enquanto a busca por maior desempenho no processo computacional motivou a utilização do processamento paralelo. Atualmente, as duas áreas têm convergido, de maneira que a combinação entre os dois enfoques computacionais oferece benefícios para as duas áreas, sendo que a computação paralela utilizando sistemas distribuídos oferece flexibilidade e uma maneira eficiente de se explorar *hardware* interligado disponível [ZAL91].

Um paradigma muito utilizado para a implementação de programas paralelos em ambientes distribuídos fracamente acoplados, como arquiteturas paralelas de memória distribuída e sistemas distribuídos, trata-se da programação via **troca de mensagens**. [MCB94] [KIT95] [QUE94].

Dentre os ambientes de programação via troca de mensagens, destacam-se as plataformas de portabilidade, (destinados a possibilitar o transporte de programas paralelos entre plataformas computacionais distintas) das quais dois representantes merecem destaque no cenário computacional atual: o MPI [DON95] [SNI96] [MAC96] [MCB94] e o PVM [BEG94] [SUN94] [PVM96].

O PVM destaca-se por ser considerado por muitos autores um padrão de fato para plataformas de portabilidade, enquanto o MPI é uma tentativa de padronização de direito, levada a cabo por diversas organizações mundiais. [MCB94].

O objetivo principal do MPI é garantir eficiência em qualquer plataforma paralela (arquiteturas paralelas ou sistemas distribuídos), e por isso, mesmo as funções básicas de uma biblioteca de troca de mensagens, que possibilitam enviar e receber uma mensagem, possuem diversas variações. O grande problema para o usuário é entender todas essas variações, suas vantagens e as implicações de desempenho que estas possuam.

Dentro deste contexto, este trabalho objetiva fazer um estudo a nível de estrutura e desempenho das rotinas de comunicação ponto-a-ponto do MPI sobre sistemas distribuídos baseados em uma rede de computadores pessoais (PC's) sobre o controle do sistema operacional LINUX (versão do *kernel* 1.3.20).

Esta análise é feita através da utilização de *benchmarks* e de um exemplo de aplicação paralela, possibilitando estudar o comportamento do MPI em determinadas situações.

Os testes foram realizados em três implementações do MPI de domínio público que executam sobre a plataforma LINUX:

Implementação	**Organizações**	**Referências**
MPICH 1.0.12	*Argonne National Laboratory* *Mississipi-State University*	[GRO96a] [MPI96]
LAM 6.0	*Ohio SuperComputer Center*	[BUR95] [LAM96]
UNIFY 0.9.2	*Mississipi-State University*	[CHE94] [VAU94]

Tal procedimento possibilita uma análise comparativa entre as três implementações, a fim de se determinar, por exemplo, até que ponto uma especificação centrada na eficiência pode garanti-la em qualquer implementação.

Por fim, será analisado o comportamento das três implementações face a uma aplicação paralela real, de maneira a comparar-se os resultados obtidos com situações reais de paralelismo.

Em alguns casos, foram incluídos resultados obtidos com o PVM, versão 3.3.10, a fim de possibilitar uma base onde possa ser avaliado o desempenho das implementações MPI em comparação com uma plataforma de portabilidade extremamente difundida.

Entre os pontos principais que motivaram a realização deste trabalho, destacam-se:

- As duas principais plataformas de portabilidade na atualidade, são o PVM e o MPI. O PVM está sendo explorado por algumas dissertações de mestrado dentro deste grupo de pesquisa, de maneira que é interessante estudar também o MPI, afim de que se possua um ponto de vista mais abrangente sobre plataformas de portabilidade;
- A aceitação da computação paralela está intimamente ligada a possibilidade de portabilidade direta de programas entre sistemas heterogêneos. Neste contexto, segundo alguns autores, o MPI pode tornar-se um padrão de grande importância no futuro da computação, tanto a nível acadêmico como comercial [WAL95]. Sob esse ponto de vista, é importante estudar o comportamento de algumas de suas implementações;
- A plataforma de *hardware* utilizada (rede de PC's) é muito difundida, sendo interessante um estudo do comportamento de programas paralelos nestas plataformas. Além disso, todos os *softwares* utilizados neste trabalho são de domínio público, de maneira que uma grande parcela da comunidade computacional tem estas ferramentas ao seu alcance.
- Os primeiros contatos dos autores com o MPI, antes da definição deste trabalho, mostraram que a extensa quantidade de rotinas de comunicação dificultava a sua compreensão. Assim, seria útil a existência de uma dissertação onde o usuário pudesse partir de conceitos teóricos simplificados e resultados práticos, afim de poder escolher as melhores opções para a escrita de um algoritmo paralelo.

Segundo Blech [BLE94], o futuro da computação paralela (e portanto, também o futuro da ciência computacional) está intimamente ligado à utilização de plataformas MIMD com memória distribuída, em virtude de sua flexibilidade e facilidade de ampliação. Essas plataformas serão formadas por sistemas

distribuídos interligados via tecnologias de interconexão confiáveis e rápidas. Neste contexto se insere a importância da programação via troca de mensagens (que é utilizada para a implementação de algoritmos paralelos em memória distribuída) e também das plataformas de portabilidade (principalmente o MPI), visto a importância da possibilidade de migração de programas paralelos entre diferentes plataformas.

Esta dissertação se divide em nove capítulos. Nos capítulos 2, 3 e 4 é apresentada uma revisão bibliográfica discutindo os conceitos computacionais básicos utilizados neste trabalho. Esta revisão está dividida respectivamente em: computação paralela, sistemas distribuídos e programação via troca de mensagens.

No capítulo 5 são apresentadas em detalhes as quatro implementações de plataformas de portabilidade utilizadas. São apresentadas comparações (a nível de estrutura) entre as implementações MPI e entre o MPI e o PVM.

No capítulo 6 são discutidas todas as formas de comunicação ponto-a-ponto definidas no MPI, discutindo suas vantagens e características básicas.

A seguir, no capítulo 7, são apresentados os métodos de avaliação e as métricas de desempenho utilizadas para a análise de desempenho das rotinas de comunicação ponto-a-ponto.

No capítulo 8, são apresentados os resultados obtidos pela aplicação dos métodos de avaliação definidos e uma análise destes resultados.

Finalmente, no capítulo 9, esta dissertação é concluída, apresentando as principais contribuições deste trabalho e as sugestões de trabalhos futuros.

No apêndice A é apresentado o método de análise estatística utilizado para avaliar a confiabilidade dos resultados obtidos nesse trabalho, e também as tabelas construídas a partir destes resultados.

Capítulo 2

Computação Paralela

Este capítulo apresenta uma visão geral da computação paralela através da apresentação de vários conceitos a nível de *software* e *hardware*, e divide-se basicamente nos seguintes tópicos: introdução e motivação, conceitos básicos, classificação de arquiteturas paralelas e programação concorrente.

2.1. Introdução

Hwang [HWA84] define processamento paralelo como:

"Forma eficiente do processamento de informações com ênfase na exploração de eventos concorrentes no processo computacional"

Processamento paralelo implica na divisão de uma determinada aplicação de maneira que esta possa ser executada por vários elementos de processamento, que por sua vez deverão cooperar entre si (comunicação e sincronismo), buscando eficiência através da quebra do paradigma de execução seqüencial do fluxo de instruções ditado pela filosofia de Von Neumann.

A idéia de se utilizar paralelismo no processo computacional é quase tão antiga quanto os computadores eletrônicos. O próprio Von Neumann, em seus artigos, sugeria formas paralelas de se resolver equações diferenciais. O passo inicial do processamento paralelo, porém, é considerado o surgimento do computador ILLIAC IV, construído na Universidade de *Illinois* e composto por 64 processadores, em 1971 [AMO88] [NAV89].

Vários fatores explicam a necessidade do processamento paralelo. O principal deles é a busca por maior desempenho [ALM94] [COR96] [ZAL91]. As diversas áreas nas quais a computação se aplica, sejam científicas, industriais ou militares, requerem cada vez mais poder computacional, em virtude dos complexos algoritmos que são utilizados e do tamanho do conjunto de dados a ser processado [KIR91].

Além da busca de maior desempenho, outros fatores motivaram (e motivam) o desenvolvimento da computação paralela [ALM94] [AMO88] [COR96]. Entre eles:

- O desenvolvimento tecnológico (principalmente o surgimento da tecnologia VLSI de projeto de microprocessadores), permitiu a construção de microprocessadores de alto desempenho, que agrupados, possibilitavam um ganho significativo de poder computacional. Além disso, tais configurações possibilitam uma melhor relação preço/desempenho quando comparadas aos caros supercomputadores [ZAL91];
- Restrições físicas, como a velocidade finita da luz, tornam difícil o aumento de velocidade em um único processador;
- Vários processadores fornecem uma configuração modular, o que permite o agrupamento destes processadores em módulos de acordo com a natureza da aplicação. Além disso, tem-se um meio de extensão do sistema através da inclusão de novos módulos;
- Tolerância a falhas (por exemplo, através da redundância de *hardware*).

Além disso, várias aplicações são inerentemente paralelas, e perde-se desempenho pela necessidade de torná-las seqüenciais. O chamado "gargalo de Von Neumann", segundo Almasi [ALM94], tem diminuído a produtividade do programador, daí a necessidade de novas maneiras de organização do processamento computacional.

Contudo, substituir uma filosofia computacional já firmemente estabelecida pelas várias décadas de existência da computação, como é a filosofia de Von Neumann, é algo que representa um obstáculo de dimensões muito grandes, e que de certa maneira dificulta a difusão da computação paralela [ALM94].

2.2. Conceitos Básicos

Esta seção apresenta alguns conceitos básicos relacionados a computação paralela. São discutidos os seguinte tópicos: tipos de paralelismo, granularidade, *speedup* e eficiência.

2.2.1. Tipos de Paralelismo

Processar concorrentemente processos computacionais implica que, em um determinado instante, dois ou mais processos foram iniciados e ainda não

terminados. Dentro desta definição, pode-se abstrair dois tipos de paralelismo, denominados paralelismo físico e lógico [KIR91].

Paralelismo Lógico: Neste caso, tem-se apenas um elemento de processamento, e os processos são executados intercaladamente, de maneira que apenas um está ativo a cada instante. Este tipo de paralelismo é exemplificado na figura 2.1, que apresenta um gráfico que demonstra a execução de três processos (e1, e2 e e3) em relação ao tempo.

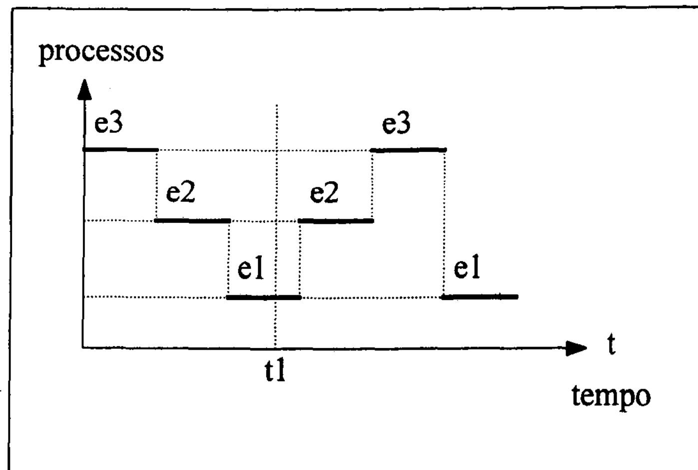

Figura 2.1 - Paralelismo Lógico.

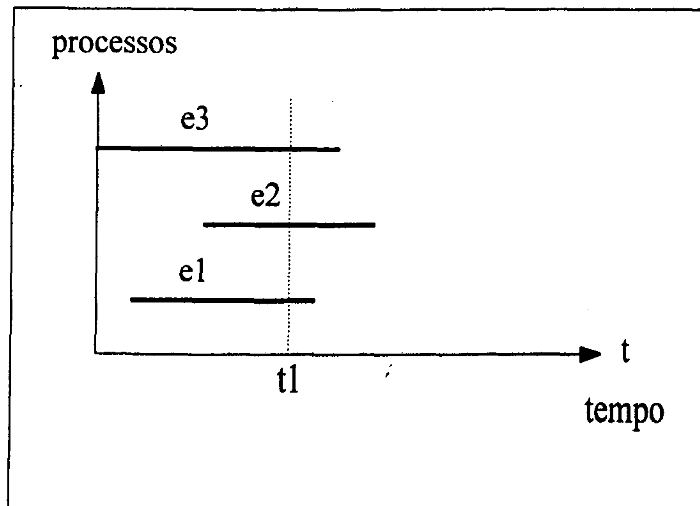

Figura 2.2 - Paralelismo Físico.

Paralelismo Físico: Neste caso, tem-se um elemento de processamento para cada processo sendo executado, sendo que todos eles podem estar ativos simultaneamente, como pode ser visto na figura 2.2.

Se existem *n* elementos de processamento, e *m* processos sendo executados concorrentemente, onde $n<m$, forma-se uma situação de **paralelismo misto**, onde encontra-se, ao mesmo tempo, paralelismo físico e lógico.

O paralelismo físico pode ser dividido em três tipos: **espacial**, **temporal** (***pipeline***) ou **combinado**.

O **paralelismo espacial** se relaciona à execução simultânea dos processos, e este pode ser síncrono, quando os elementos de processamento são ligados a uma única unidade de controle e portanto executam a mesma instrução sincronamente, ou assíncronos, quando os elementos de processamento são independentes entre si.

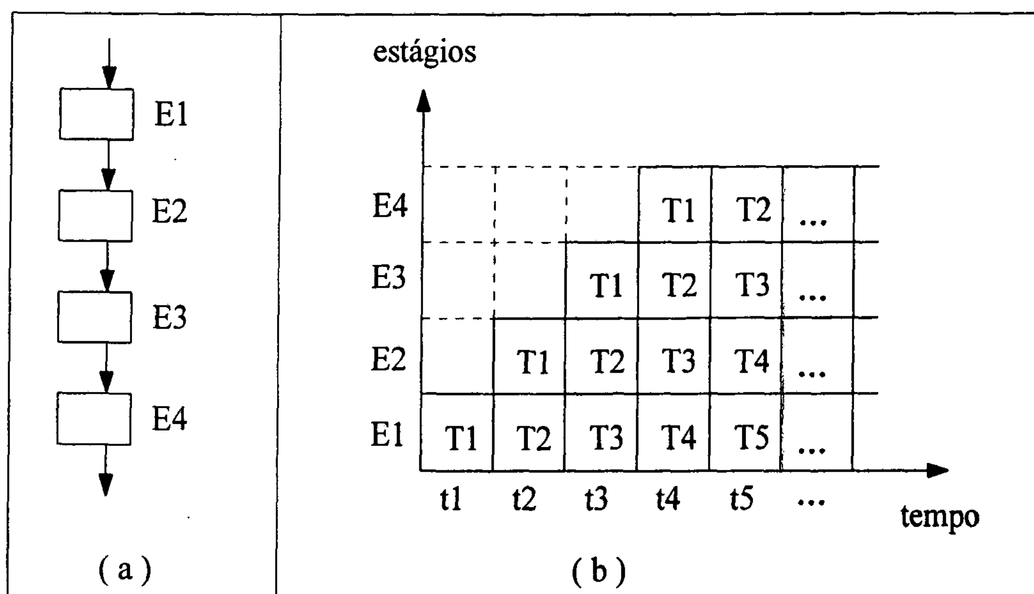

Figura 2.3 - *Pipeline*. (a) Organização dos Estágios (b) Gráfico de Execução.

O **paralelismo temporal**, ou ***pipeline***, implica na execução de eventos sobrepostos no tempo. Uma determinada tarefa é subdividida em uma seqüência de sub-tarefas, sendo cada uma delas executada por um estágio especializado de *hardware* e *software* que opera paralelamente com os outros estágios do *pipeline* [HWA84].

Considere, por exemplo, uma determinada tarefa dividida em 4 estágios, chamados E1, E2, E3 e E4. Os estágios são organizados seqüencialmente, de maneira que a partir do estágio inicial E1, cada um deles é alimentado pelo estágio

imediatamente anterior (figura 2.3 (a)). A cada instante de tempo, uma nova tarefa é iniciada, denominadas T1, T2, T3, ... , Tn. A figura 2.3 (b) apresenta um gráfico relacionando os vários estágios de execução *pipeline* em relação ao tempo transcorrido. Após 4 unidades de tempo, o fluxo de tarefas completas (que completaram os 4 estágios) torna-se contínuo, sendo uma tarefa terminada a cada unidade de tempo. Daí resulta o paralelismo de eventos sobrepostos.

O **paralelismo combinado** eqüivale a vários estágios *pipeline* sendo executados em paralelo.

Paralelismo X Concorrência: O termo paralelismo, na literatura, é normalmente utilizado para designar paralelismo físico. É importante então ressaltar a diferença entre paralelismo e concorrência. Concorrência implica em mais de um processo iniciado e não ainda terminado. Não necessariamente existe mais de um elemento de processamento, enquanto paralelismo é um caso especial de concorrência, onde existem processos sendo executados ao mesmo tempo.

2.2.2. Granularidade

Granularidade, ou nível de paralelismo, define o tamanho das unidades de trabalho submetidas aos processadores. Esta é uma definição muito importante na computação paralela, visto que está intimamente ligada ao tipo de plataforma (o porte e a quantidade de processadores) à qual se aplica o paralelismo [KIR91].

Diversas definições de granularidade podem ser encontradas na literatura [ALM94] [COR96] [HWA84] [KIR91] [NAV89]. Mas, de maneira simples, a granularidade pode ser dividida em três níveis: fina, média e grossa.

Granularidade grossa relaciona o paralelismo a nível de processos e programas, e geralmente se aplica a plataformas com poucos processadores grandes e complexos [KIR91] [NAV89].

Granularidade fina, por outro lado, relaciona paralelismo a nível de instruções ou operações e implica em grande número de processadores pequenos e simples. A granularidade média situa-se em um patamar entre as duas anteriores, implicando em procedimentos sendo executados em paralelo [KIR91] [NAV89].

2.2.3. *Speedup* e Eficiência

Uma característica fundamental da computação paralela é o aumento de velocidade de processamento através da utilização do paralelismo. Neste contexto,

duas medidas muito importantes para a verificação da qualidade de algoritmos paralelos são *speedup* e eficiência. Novamente, várias definições existem na literatura [QUI87] [KIR91].

Uma definição largamente aceita para *speedup* é: aumento de velocidade observado quando se executa um determinado processo em *p* processadores em relação a execução deste processo em 1 processador. Então, tem-se:

speedup = T_1 / T_p , onde T_1 = tempo de execução em 1 processador

T_p = tempo de execução em p processadores

Idealmente, o ganho de *speedup* deveria tender a *p*, que seria o seu valor ideal. Porém, três fatores podem ser citados que influenciam nesta relação, gerando uma sobrecarga que diminui o valor de *speedup* ideal. Estes são: sobrecarga da comunicação entre os processadores, partes do código executável estritamente seqüenciais (que não podem ser paralelizadas) e o nível de paralelismo utilizado (em virtude do uso de granularidade inadequada à arquitetura) [ALM94] [QUI87].

Outra medida importante é a eficiência, que trata da relação entre o *speedup* e o número de processadores.

eficiência = *speedup* / p

No caso ideal (*speedup* = p), a eficiência seria máxima e teria valor 1 (100%).

2.3. Classificação de Arquiteturas Paralelas

Existem muitas maneiras de organizar computadores paralelos. Para que se possa visualizar melhor todo o conjunto de possíveis opções de arquiteturas paralelas, é interessante classificá-las. Segundo Ben-Dyke [BEN93], uma classificação ideal deve ser:

- **Hierárquica**: iniciando em um nível mais abstrato, a classificação deve ser refinada em subníveis à medida que se diferencie de maneira mais detalhada cada arquitetura;
- **Universal**: um computador único, deve ter uma classificação única;
- **Extensível**: futuras máquinas que surjam, devem ser incluídas sem que sejam necessárias modificações na classificação;
- **Concisa**: os nomes que representam cada uma das classes devem ser pequenos para que a classificação seja de uso prático;

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

Muito já foi desenvolvido em termos de *hardware* paralelo, e várias classificações foram propostas [ALM94] [BEN93] [CORN96] [DUN90] [HWA84]. A mais conhecida pela comunidade computacional é a classificação de Flynn [FLY72].

2.3.1. Classificação de Flynn

Segundo Flynn, o processo computacional deve ser visto como uma relação entre fluxos de instruções e fluxos de dados. Um fluxo de instruções equivale a uma seqüência de instruções executadas (em um processador) sobre um fluxo de dados aos quais estas instruções estão relacionadas [ALM94] [COR96] [DUN90] [FLY72] [HWA84] [NAV89].

Baseando-se nas possíveis unicidade e multiplicidade de fluxos de dados e instruções, divide-se as arquiteturas de computadores em 4 classes. Para cada classe, é apresentada uma esquematização genérica.

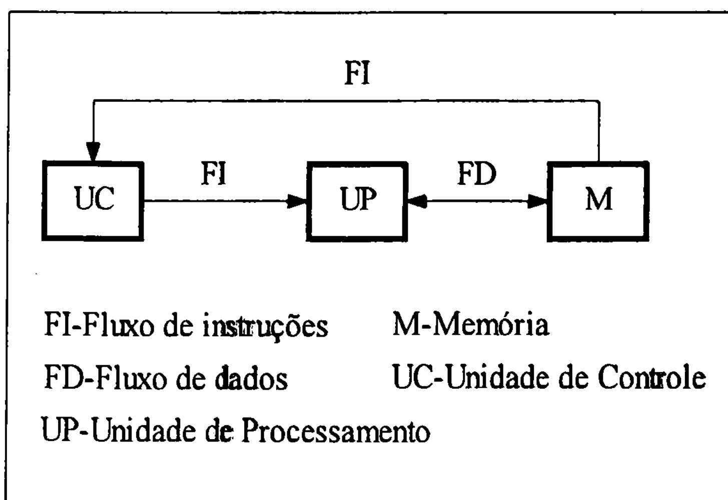

Figura 2.4 - Modelo Computacional SISD.

<u>SISD</u> - *Single Instruction Stream / Single Data Stream* (Fluxo único de instruções / Fluxo único de dados): Corresponde ao tradicional modelo Von Neumann. Um processador executa seqüencialmente um conjunto de instruções sobre um conjunto de dados (figura 2.4).

SIMD - *Single Instruction Stream / Multiple Data Stream* (Fluxo único de instruções / Fluxo múltiplo de dados): Envolve múltiplos processadores executando simultaneamente a mesma instrução em diversos conjuntos de dados (figura 2.5).

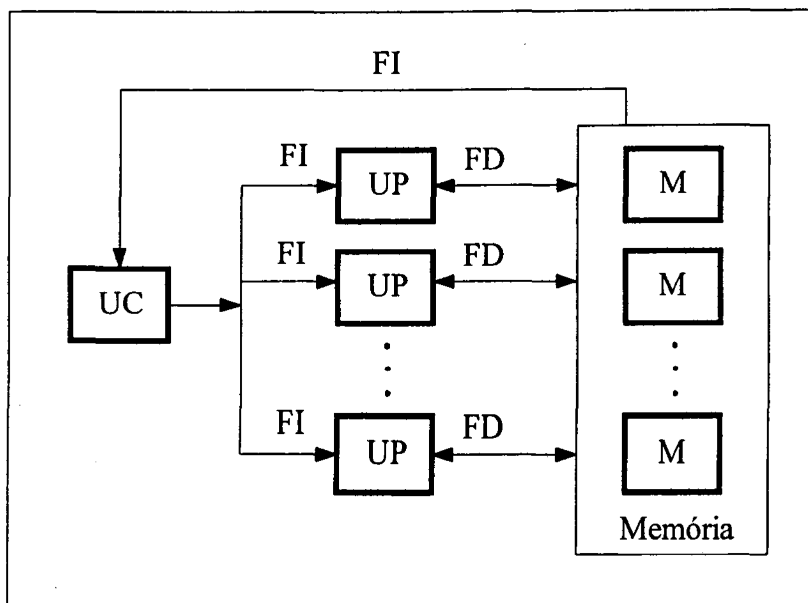

Figura 2.5 - Modelo Computacional SIMD.

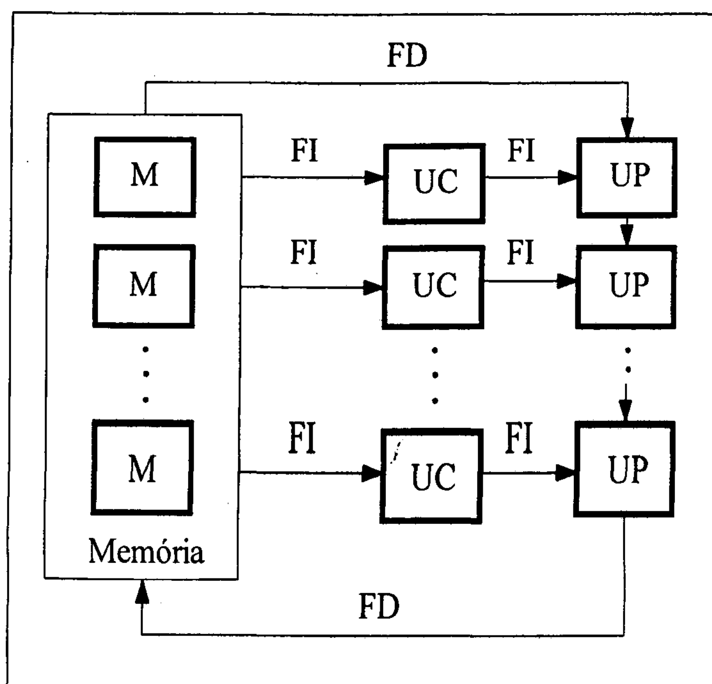

Figura 2.6 - Modelo Computacional MISD.

MISD - *Multiple Instruction Stream / Single Data Stream* (Fluxo múltiplo de instruções / Fluxo único de dados): Envolve múltiplos processadores executando diferentes instruções em um único conjunto de dados. Geralmente, nenhuma arquitetura é classificada como MISD, isto é, não existem representantes desta categoria. Alguns autores consideram arquiteturas *pipeline* como exemplo deste tipo de organização (figura 2.6).

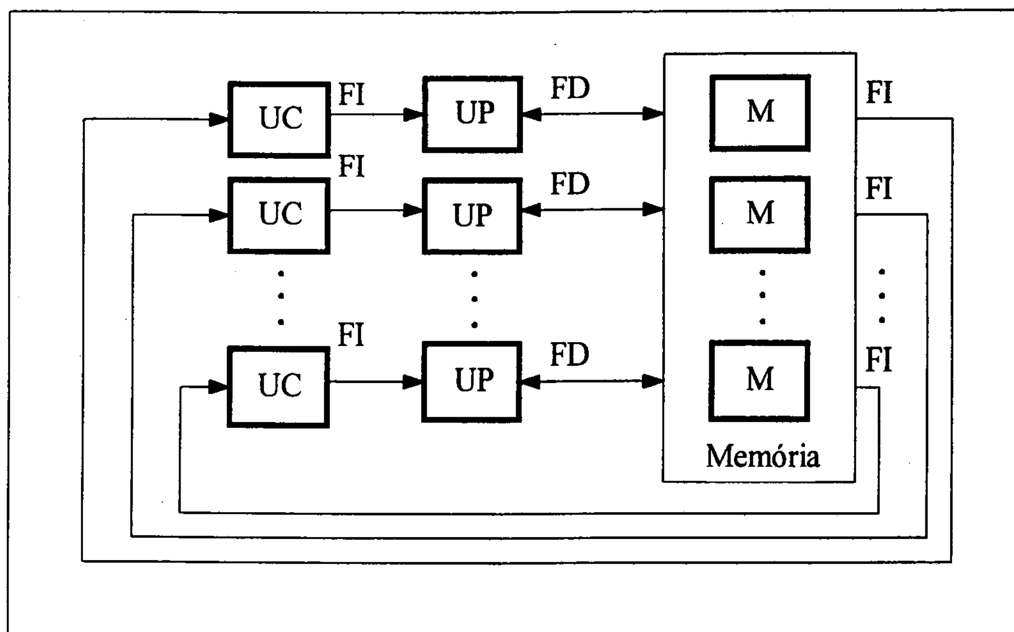

Figura 2.7 - Modelo Computacional MIMD.

MIMD - *Multiple Instruction Stream / Multiple Data Stream* (Fluxo múltiplo de instruções / Fluxo múltiplo de dados): Envolve múltiplos processadores executando diferentes instruções em diferentes conjunto de dados, de maneira independente (figura 2.7).

A classificação de Flynn não é abrangente o suficiente para incluir alguns computadores modernos (por exemplo, processadores vetoriais e máquinas de fluxo de dados), falhando também, no que concerne a extensibilidade da classificação. Outro inconveniente desta classificação é a falta de hierarquia. A classificação MIMD, por exemplo, engloba quase todas as arquiteturas paralelas sem apresentar sub-níveis. No entanto, apesar de antiga (proposta em 1972), a classificação de Flynn é bastante concisa e a mais utilizada.

A fim de acrescentar novas arquiteturas paralelas surgidas, sem descartar a classificação de Flynn (visto que esta é muito difundida), Duncan [DUN90] propõe

uma classificação mais completa, e que permite apresentar uma visão geral dos estilos de organização para computadores paralelos da atualidade.

2.3.2. Classificação de Duncan

Duncan, em sua classificação, exclui arquiteturas que apresentem apenas mecanismos de paralelismo de baixo nível, que já se tornaram lugar comum nos computadores modernos. Exemplos desses mecanismos são: *pipeline* dos estágios de execução de uma instrução e unidades funcionais múltiplas em uma única CPU.

A classificação de Duncan é apresentada na figura 2.8, e a seguir cada um dos seus componentes são brevemente discutidos.

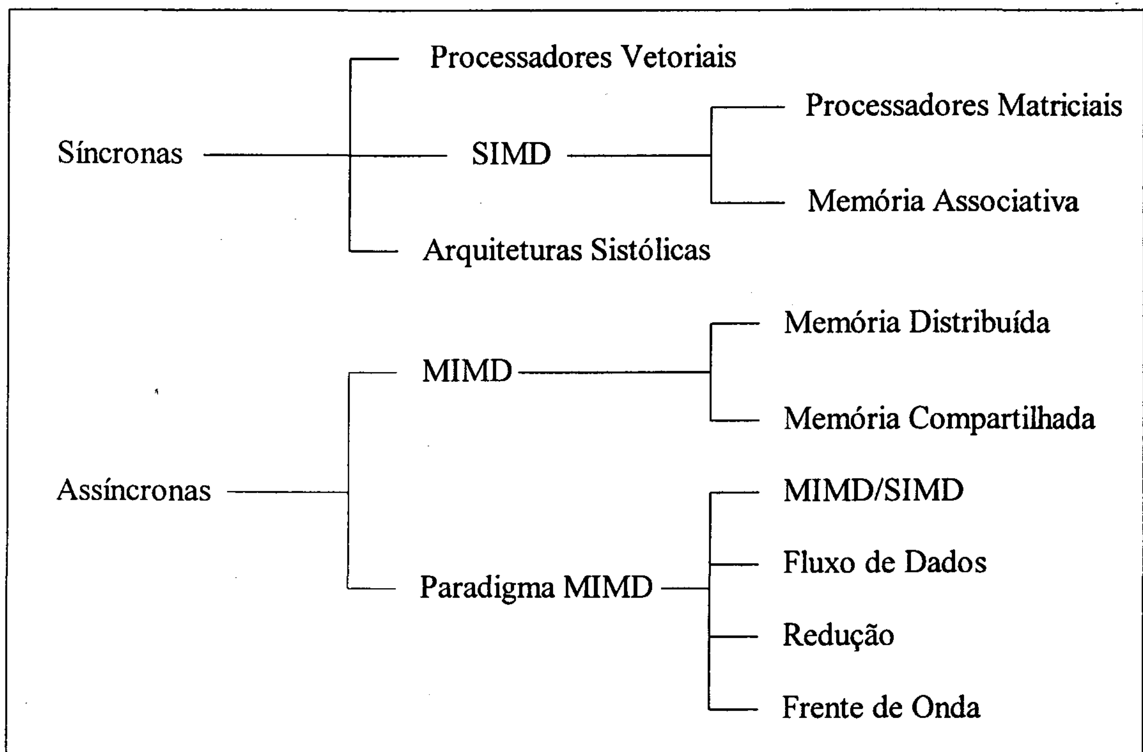

Figura 2.8 - Classificação de Duncan.

Arquiteturas Síncronas: Arquiteturas paralelas síncronas coordenam suas operações concorrentes sincronamente em todos os processadores, através de relógios globais, unidades de controle únicas ou controladores de unidades vetoriais [DUN90]. Tais arquiteturas apresentam pouca flexibilidade para a expressão de algoritmos paralelos [BLE94] [COR96].

Processadores Vetoriais: Processadores vetoriais caracterizam-se pela existência de um *hardware* específico para a execução de operações em vetores. Essas operações são implementadas através de instruções vetoriais. Os processadores vetoriais *pipeline* são caracterizados por possuírem múltiplas unidades funcionais organizadas de maneira *pipeline*, onde são implementadas as instruções vetoriais. Essas arquiteturas foram criadas com o intuito de prover mecanismos eficientes para o suporte de cálculos pesados em matrizes ou vetores.

Esquematicamente, a organização básica de um processador vetorial apresenta: um processador escalar (para a execução do código não vetorizável do programa), uma unidade de processamento vetorial, e um processador de instruções (que define quais instruções serão executadas em qual dos processadores) [KIR91].

Arquiteturas SIMD: Arquiteturas SIMD apresentam múltiplos processadores, sobre a supervisão de uma unidade central de controle, que executam a mesma instrução sincronamente em conjuntos de dados distintos. Podem ser organizadas de duas maneiras [HWA84]:

- **Processadores Matriciais**: Projetados especialmente para fornecer estruturas para computação sobre matrizes de dados, são empregados para fins específicos. Fornecem acesso a memória via endereço, o que os diferencia do modelo associativo, discutido a seguir.
- **Memória Associativa**: Relacionam arquiteturas SIMD cujo acesso à memória é feita de acordo com o seu conteúdo, em contraste ao método de acesso usual, via endereço. O esquema associativo visa permitir o acesso paralelo a memória, de acordo com certo padrão de dados.

Arquiteturas Sistólicas: Propostas no início da década de 80, por H.T. Kung na Universidade de Carnegie Mellon, estas arquiteturas têm como principal objetivo fornecer uma estrutura eficiente para a solução de problemas que necessitem de computação intensiva junto a grande quantidade de operações de E/S. Estas arquiteturas se caracterizam pela presença de vários processadores, organizados de maneira *pipeline*, que formam uma cadeia na qual apenas os processadores localizados nos limites desta estrutura possuem comunicação com a memória.

Desta maneira, o conjunto de dados percorre toda cadeia de processadores, de maneira rítmica e sincronizada por um relógio global, não havendo armazenamento temporário em memória na comunicação entre os processadores.

__Arquiteturas Assíncronas__: Estas arquiteturas caracterizam-se pelo controle descentralizado de *hardware*, de maneira que os processadores são independentes entre si. Esta classe é formada basicamente pelas arquiteturas MIMD, sejam convencionais ou não [DUN90].

Arquiteturas MIMD: Relacionam arquiteturas compostas por vários processadores independentes, onde se executam diferentes fluxos de instruções em dados locais a esses processadores.

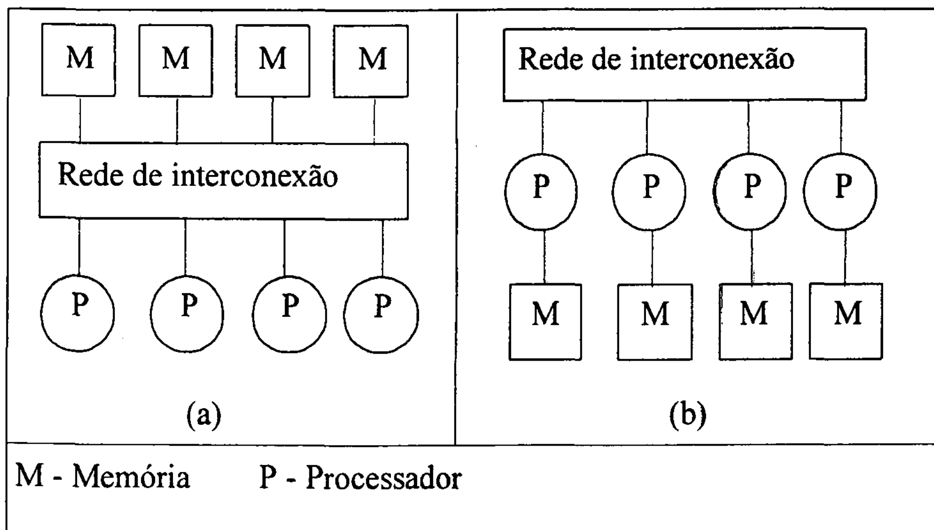

Figura 2.9 - Arquiteturas MIMD. (a) Mem. Centralizada (b) Mem. Distribuída.

Este tipo de organização pressupõe algoritmos de granularidade mais grossa e com pouca comunicação entre processos (em virtude da sobrecarga de comunicação ser maior), enquanto arquiteturas síncronas utilizam algoritmos de granularidade mais fina. Além disso, arquiteturas MIMD oferecem grande flexibilidade para a construção de algoritmos paralelos [BLE94] [COR96]. Apesar de independentes, os processos executando nos diversos processadores devem cooperar entre si, tornando necessário a comunicação e o sincronismo entre estes processos. A implementação dos métodos de comunicação e sincronismo depende da organização de memória, que pode ser centralizada ou distribuída (figura 2.9).

- **Memória Centralizada**: Todos os processadores são ligados a uma memória global e única, e através do compartilhamento de posições desta memória ocorre a comunicação entre processos. Este tipo de organização também é conhecida como **multiprocessador**.

- **Memória Distribuída**: Cada processador possui memória local. Em virtude de não haver compartilhamento de memória, os processos comunicam-se via **troca de mensagens**, que se trata da transferência explicita de dados entre os processadores. Este tipo de organização é também conhecida como **multicomputador**.

A distinção entre as duas organizações de memória citadas deve ser feita com dois referenciais em mente: *hardware* e *software* [ALM94] [TAN95] [BLE94].

Do ponto de vista de programação, a diferença básica reside nas primitivas de comunicação entre processos. A comunicação em memória centralizada é baseada no compartilhamento de posições de memória, e para que os dados se mantenham consistentes, é necessário um método de controle de acesso a essas variáveis compartilhadas, como por exemplo: semáforos, monitores, etc. Estes mecanismos de programação já são bem conhecidos pelo programador em geral.

No caso de memória distribuída, a comunicação é feita através de troca de mensagens, o que gera algumas complicações, entre elas, o controle de fluxo, controle sobre mensagens perdidas, *bufferização* e bloqueio de rotinas. Então, do ponto de vista do programador, a melhor escolha é a memória compartilhada. É interessante ressaltar que pode-se usar o paradigma de troca de mensagens em memória compartilhada, o que aumenta o leque de opções para o programador.

Em relação ao *hardware*, esta situação se inverte. Arquiteturas de memória distribuída são mais fáceis de se construir e são naturalmente ampliáveis [MCB94] [BLE94]. Portanto, do ponto de vista do projetista, a melhor escolha é uma arquitetura de memória distribuída.

O caso ideal é a existência de uma arquitetura que seja fácil de projetar e programar. Com esse objetivo em mente, foi criada a idéia das arquiteturas de memória centralizada distribuída (ou memória compartilhada virtual). Nesse caso, constrói-se uma plataforma de memória distribuída e, via mecanismos implementados em *hardware* e *software*, emula-se um ambiente de memória centralizada [BLE94] [COR96] [MCB94] [TAN95].

Paradigma MIMD: Esta classe engloba as arquiteturas assíncronas que, apesar de apresentarem a característica de multiplicidade de fluxo de dados e instruções das arquiteturas MIMD, são organizadas segundo conceitos tão fundamentais a seu projeto quanto suas características MIMD. Estas características próprias de cada arquitetura, dificultam a sua classificação como puramente MIMD. Por isso, tais arquiteturas se denominam paradigmas arquiteturais MIMD.

- **Arquiteturas MIMD/SIMD (híbridas)**: Também conhecidas por MSIMD, tais arquiteturas caracterizam-se por apresentarem controle SIMD para determinadas partes de uma arquitetura MIMD;
- **Arquiteturas a Fluxo de Dados** ***(dataflow)***: Máquinas convencionais Von Neumann são denominadas computadores baseados em fluxo de controle, uma vez que instruções são executadas de acordo com a seqüência ditada pelo contador de programas [HWA84]. Para explorar o paralelismo máximo em um programa, foi proposto mudar o controle da execução das instruções de acordo com a dependência dos dados aos quais se aplicam estas instruções, gerando computadores baseados no fluxo de dados. Basicamente, uma instrução é executada assim que todos os seus operandos estão disponíveis. Assim, controla-se de maneira não centralizada a execução de um programa, com os dados fluindo de instrução a instrução, de maneira que se consiga paralelismo em alta escala;
- **Arquiteturas de Redução**: Também conhecidas como arquiteturas dirigidas a demanda, estas arquiteturas são baseadas no conceito de redução, que implica que partes do código fonte original sejam reduzidas aos seus resultados em tempo de execução [KIR91]. As instruções são ativadas para serem executadas quando os seus resultados são necessários como operandos por outra instrução já ativada para execução;
- **Arquiteturas de Frente de Onda**: Construídas com os mesmos objetivos de balanceamento entre computação e E/S das arquiteturas sistólicas, estas arquiteturas de propósito específico caracterizam-se por apresentarem uma estrutura sistólica de processadores, combinada ao paradigma assíncrono de execução baseada no fluxo de dados.

Conforme visto nesta seção, a classificação de Duncan engloba a maioria dos tipos de arquiteturas existentes, tendo atingido os itens abrangência e extensibilidade das características ideais de uma classificação citadas por Ben-Dyke [BEN93], nos quais a classificação de Flynn falha. Além disso, uma característica importante da classificação proposta por Duncan é a presença dos termos MIMD e SIMD, propostos por Flynn e largamente aceitos.

Dentre as arquiteturas em geral, o modelo MIMD tem se destacado (principalmente o modelo de memória distribuída), devido à sua flexibilidade e por representar uma boa opção para o desenvolvimento de algoritmos paralelos de granularidades média e grossa [BLE94] [COR96] [ZAL91]. Assim, nas próximas

seções serão discutidos alguns tópicos sobre a programação neste tipo de arquitetura.

2.4. Programação Concorrente

Um programa seqüencial é composto de um conjunto de instruções que são executadas seqüencialmente, sendo que a execução destas instruções é denominada um **processo**. Um programa concorrente especifica dois ou mais programas seqüenciais que podem ser executados concorrentemente como **processos paralelos** [AND83].

A programação concorrente existe para que se forneçam ferramentas para a construção de programas paralelos, de maneira que se consiga melhor desempenho e melhor utilização do *hardware* paralelo disponível. A computação paralela apresenta muitas vantagens em relação à computação seqüencial, como foi exposto na seção 2.1, e todas essas vantagens podem ser citadas como pontos de incentivo para o uso da programação concorrente.

Um programa seqüencial é constituído basicamente de um conjunto de construções já bem dominadas pelo programador em geral, como por exemplo, atribuições, comandos de decisão (if... then... else), laços (for... do), entre outras. Um programa concorrente, além dessas primitivas básicas, necessita de novas construções que o permitam tratar aspectos decorrentes da execução paralela dos vários processos.

Segundo Almasi [ALM94], para a execução de programas paralelos, deve haver meios de:

- Definir um conjunto de tarefas a serem executadas paralelamente;
- Ativar e encerrar a execução dessas tarefas;
- Coordenar e especificar a interação entre essas tarefas.

A fase de definição da organização das tarefas paralelas é de extrema importância, pois o ganho de desempenho adquirido da paralelização depende fortemente da melhor configuração das tarefas a serem executadas concorrentemente. Alguns aspectos que devem ser considerados no desenvolvimento de algoritmos paralelos são discutidos na seção 2.4.1.

Definido o algoritmo, é necessário um conjunto de ferramentas para que o programador possa representar a concorrência, definindo quais partes do código serão executadas seqüencialmente e quais serão paralelas. As construções para

definir, ativar e encerrar a execução de tarefas concorrentes são discutidas na seção 2.4.2.

Além disso, processos cooperando para a resolução de determinado problema devem comunicar-se e sincronizar-se, afim de que haja interação entre eles. A maneira como se implementa a comunicação entre processos depende da arquitetura aonde se executa a aplicação: em caso de memória centralizada, utilizam-se variáveis compartilhadas; em caso de memória distribuída, é utilizada troca de mensagens. As formas de comunicação/sincronização entre processos são discutidos na seção 2.4.3.

2.4.1. Projeto de um Algoritmo Paralelo

Há, pelo menos, três maneiras de se construir um algoritmo paralelo [QUI87]. São elas:

- Detectar e explorar algum paralelismo inerente a um algoritmo seqüencial existente;
- Criar um algoritmo paralelo novo;
- Adaptar outro algoritmo paralelo que resolva problema similar.

Três aspectos importantes devem ser considerados quando se projeta um algoritmo paralelo. Primeiro, o processo de escolha da abordagem a ser seguida, entre as três citadas acima, deve ser feita de maneira cuidadosa, para que se consiga a melhor eficiência, levando em conta o tempo de escrita do algoritmo e o desempenho obtido.

Além disso, deve-se pesar o custo da comunicação entre processos em relação ao tempo de execução efetiva, visto que operações de comunicação geram uma sobrecarga que pode degradar o desempenho, tornando o algoritmo menos eficiente que o seqüencial.

Por último, deve-se considerar a arquitetura na qual se executará o algoritmo, visto que a sua eficiência pode variar de maneira drástica de acordo com o tipo de arquitetura.

2.4.2. Ativação de Processos Paralelos

Várias construções para a ativação e término de processos concorrentes são discutidas na literatura, apresentando características e finalidades distintas [ALM94] [AND83] [KIR89] [QUI87] [SNO92]. Modelos representativos dessas construções são apresentados a seguir.

__Corotinas__: corotinas são subrotinas que possuem um modo de transferência de controle não hierárquico. Uma subrotina comum, ativada através de uma chamada *call subrotina*, ao executar um comando *return*, retorna o controle ao módulo de programa que a ativou e termina a sua execução. Além disso, toda vez que esta é ativada, será executada desde o seu início.

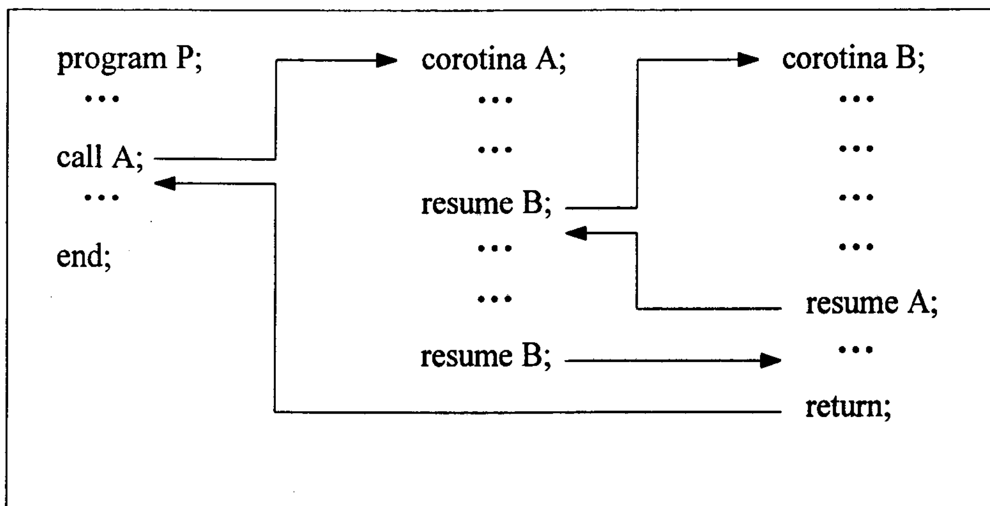

Figura 2.10 - Exemplo da Utilização de Corotinas.

Corotinas transferem controle entre si de maneira livre, através do comando *resume corotina*. E sempre que são ativadas, executam a partir do ponto onde foi executado a ultima chamada a *resume* (figura 2.10). Cada corotina pode ser vista como implementando um processo, e elas são executadas intercaladamente. Sempre existe apenas uma corotina ativa em cada instante, o que implica na adequação desta estrutura para a organização de programas concorrentes que compartilhem uma única CPU.

Fork/Join: O comando *fork* implica que um determinado conjunto de instruções (processo filho) deve iniciar a sua execução em paralelo com o processo que o executa (processo pai). O comando *join* é utilizado para a sincronização do

processo pai com os filhos gerados. Um exemplo de sintaxe para os comandos *fork/join* é:

Fork *end*: Executa a partir do endereço *end*, concorrentemente ao processo que executa a chamada *fork*;

Join *num, end1, end2*: Decrementa a variável *num*, que contém o número de processos que devem sincronizar-se. Se *num*=0, execute a partir de *end1*. Caso contrário, execute a partir de *end2* (geralmente é um comando *quit*, que termina a execução).

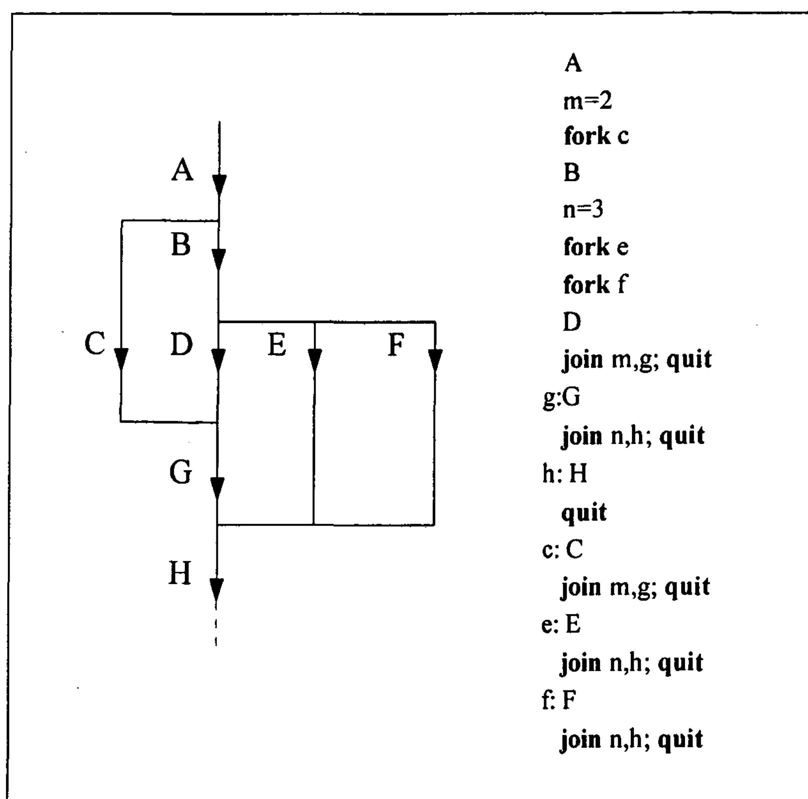

Figura 2.11 - Exemplo da Utilização de *Fork/Join*.

A utilização de *fork/join* é um meio poderoso e flexível de se especificar o processamento concorrente. Porém, programas escritos utilizando tais comandos devem ser escritos de maneira disciplinada, visto que a organização do código fonte obtido é desestruturada. Afim de proporcionar maior estruturação, a custa de perda de flexibilidade, foram propostos alguns modelos que são apresentados a seguir, como *cobegin/coend* e *doall*.

Cobegin/Coend: Também chamados de *parbegin/parend*, estes comandos oferecem uma maneira estruturada de ativação de um conjunto de instruções que devem ser executadas concorrentemente (figura 2.12). A execução concorrente das declarações S1, S2, ..., Sn pode ser ativada através da estrutura:

Cobegin S1 // S2 // S3 // ... // Sn ***Coend***

O processo pai será bloqueado até que S1, S2, ..., Sn estejam terminadas.

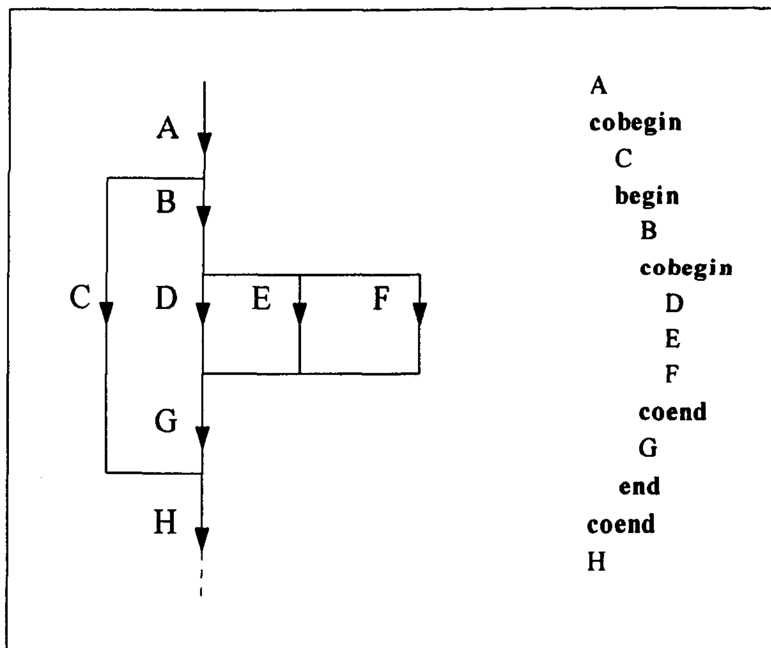

Figura 2.12 - Exemplo da Utilização de *Cobegin/Coend.*

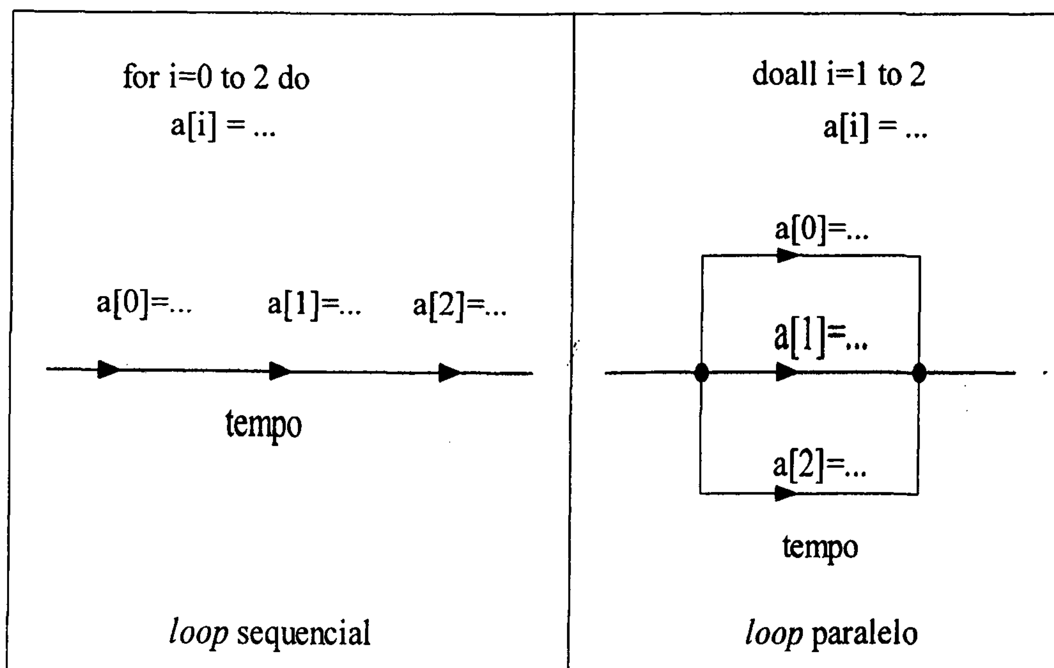

Figura 2.13 - Exemplo da Utilização de *Doall.*

Doall: Este comando pode ser visto como um comando *cobegin/coend* onde as instruções executadas em paralelo são as diversas instâncias de um bloco de comandos dentro de um comando de *loop* (figura 2.13). Alguns comandos de função semelhante são: *forall, pardo* e *doacross*.

A escolha do método de ativação de processos concorrentes deve ser feita de acordo com os objetivos do programador. Corotinas são utilizadas para ativação de processos concorrentes, *fork/join* e *cobegin/coend* para a ativação de processos paralelos e *doall* para a ativação paralela de instâncias de *loops*. Em relação ao contraste flexibilidade/estruturação, *fork/join* oferece um mecanismo flexível porém desestruturado, enquanto *cobegin/coend* e *doall* apresentam maior estruturação, o que diminui a flexibilidade.

2.4.3. Comunicação e Sincronismo

Comunicação é necessária para que processos interagindo na resolução de determinada aplicação troquem informações. E quando há comunicação, devem existir operações de sincronização, para fornecer controle de acesso e controle de seqüência.

Controle de seqüência (também chamado de sincronização condicional ou sincronização de atividades) é utilizado para que se determine uma ordem na qual os processos (ou partes deles) devem ser executados. Controle de acesso é necessário quando há competição entre processos para a manipulação de algum recurso. Deve-se garantir que acessos concorrentes a estes recursos sejam controlados, para que se mantenha a consistência [ALM94] [AND83] [KIR89] [QUI87] [SNO92].

Comunicação e Sincronismo em Memória Centralizada: Comunicação e sincronismo (controle de acesso e seqüência) em memória centralizada são implementados através da utilização de variáveis compartilhadas entre os diversos processos concorrentes.

Controle de acesso é geralmente implementado através de exclusão mútua. São definidas seções críticas, isto é, conjuntos de instruções que devem ser executadas de maneira mutuamente exclusiva, de maneira que processos não sofram interferência durante a execução destas seções.

Para a implementação de exclusão mútua e controle de seqüência utilizando-se de variáveis compartilhadas, vários métodos podem ser utilizados,

entre eles: *busy-waiting*, semáforos e monitores [ALM94] [AND83] [KIR89] [QUI87] [SNO92].

Busy-waiting: Utilizando-se, por exemplo, uma variável compartilhada cujo valor pode ser manipulado (modificado e testado / *test-and-set*) através de uma primitiva indivisível, cria-se um modo simples para a sincronização de processos concorrentes. Um processo querendo entrar em uma região crítica deve executar esta primitiva continuamente até conseguir permissão de entrada. São gastos ciclos de CPU enquanto se está testando a variável, caracterizando este método de sincronização como um exemplo de *busy-waiting*, ou espera ocupada.

Busy-waiting possui várias desvantagens, como por exemplo, o gasto supérfluo de CPU. Além disso, programas utilizando tais primitivas são difíceis de entender, depurar e provar que estão corretos, em virtude de serem implementados a baixo nível.

Semáforos: Um semáforo é uma variável compartilhada inteira e não negativa sobre a qual estão definidas duas operações atômicas (indivisíveis): **p** e **v** (também chamados de **down** e **up**, respectivamente). Dado um semáforo *s*, implementa-se **p** e **v** como:

p(s): Se ($s = 0$)
então bloqueia-se o processo
senão $s = s - 1$

v(s): s = s + 1

Semáforos oferecem um meio de sincronização de nível mais elevado do que *busy-waiting*, além de evitar o desperdício de CPU. Um exemplo de sua utilização é a implementação de exclusão mútua. Seja o semáforo s, iniciado com o valor 1. Então, a seqüência

p(s); *região crítica*; **v(s)**;

garante que apenas um único processo esteja executando esta região crítica em um determinado instante.

Monitores: Uma desvantagem de semáforos é o fato de serem pouco estruturados, o que pode levar a erros. Monitores oferecem uma maneira estruturada para a implementação de exclusão mútua.

Um monitor consiste de variáveis representando o estado de algum recurso compartilhado e procedimentos que implementam operações sobre esses recursos.

Essas variáveis podem ser acessadas somente pelos procedimentos internos a cada monitor, e a execução desses procedimentos é feita de maneira mutuamente exclusiva.

Comunicação e Sincronismo em Memória Distribuída: Comunicação e sincronismo em arquiteturas de memória distribuída devem ser implementados através de troca de mensagens entre processos [ALM94] [AND83] [KIR89] [QUI87] [SNO92]. Uma operação de comunicação via mensagens, de maneira genérica, é realizada pela utilização das primitivas *send/receive* (envia/recebe), cujas sintaxes, por exemplo, podem ser:

Send mensagem ***to*** processo_destino

Receive mensagem ***from*** processo_fonte

Uma transferência de mensagem pode ser realizada de duas maneiras. Uma operação síncrona (figura 2.14 (a)) implica que o processo que transmite a mensagem (transmissor) é bloqueado até que receba uma confirmação de recebimento da mensagem pelo processo receptor. Caso contrário, tem-se uma operação assíncrona (figura 2.14 (b)), isto é, o processo transmissor envia a mensagem (que deve ser armazenada em um *buffer*) e continua a sua execução.

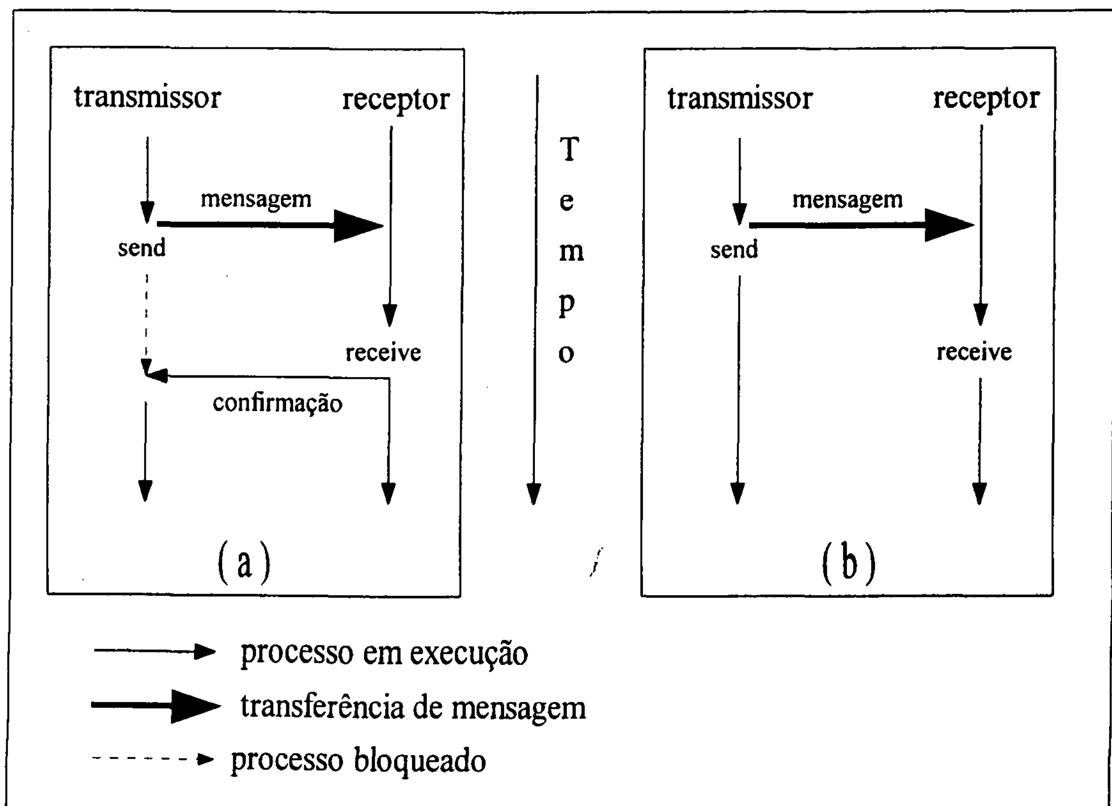

Figura 2.14 - *Send/Receive.* (a) Síncrono (b) Assíncrono.

As primitivas *send/receive* podem ser organizadas para efetuar trocas de mensagens entre dois processos, gerando três mecanismos básicos: comunicação ponto-a-ponto, *rendezvous* e RPC.

Comunicação Ponto-a-Ponto: caracteriza-se pelo uso de uma operação *send/receive* síncrona, de maneira que os processos se sincronizem (figura 2.15 (a)). Tem-se então, comunicação unidirecional.

Rendezvouz: estrutura de comunicação bidirecional que permite que um processo, por exemplo, comande a execução de um trecho de programa em outro processo. Isso é conseguido através do uso de dois conjuntos de operações *send/receive* bloqueantes (figura 2.15 (b)).

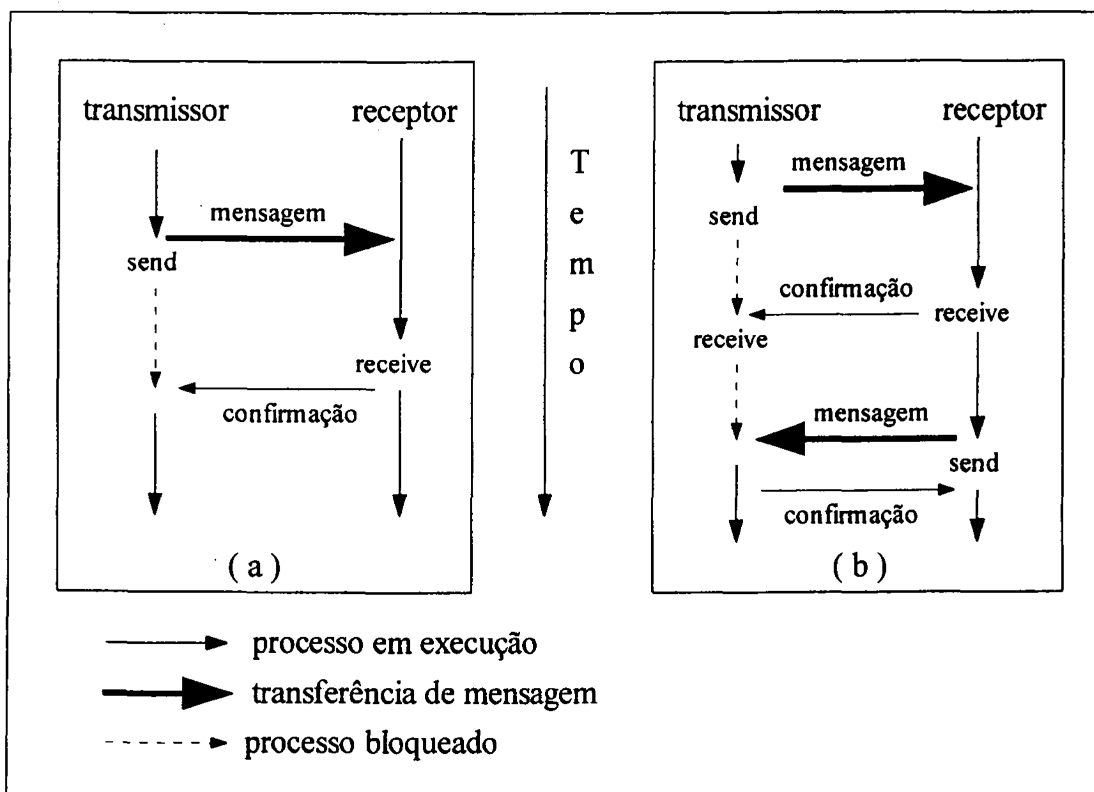

Figura 2.15 - Mecanismos de Comunicação. (a) Ponto-a-Ponto (b) *Rendezvouz.*

RPC: RPC (**R**emote **P**rocedure **C**all - Chamada Remota de Procedimento) caracteriza-se pela execução de um procedimento não local a um determinado processo, utilizando um sintaxe semelhante a chamada de procedimentos locais, sendo que o processo requisitador do serviço é bloqueado até que se obtenha os resultados desejados. O procedimento remoto é iniciado quando é recebida uma

requisição de execução. Este mecanismo é apresentado com maiores detalhes no capítulo 2 (Sistemas Distribuídos). (figura 2.16).

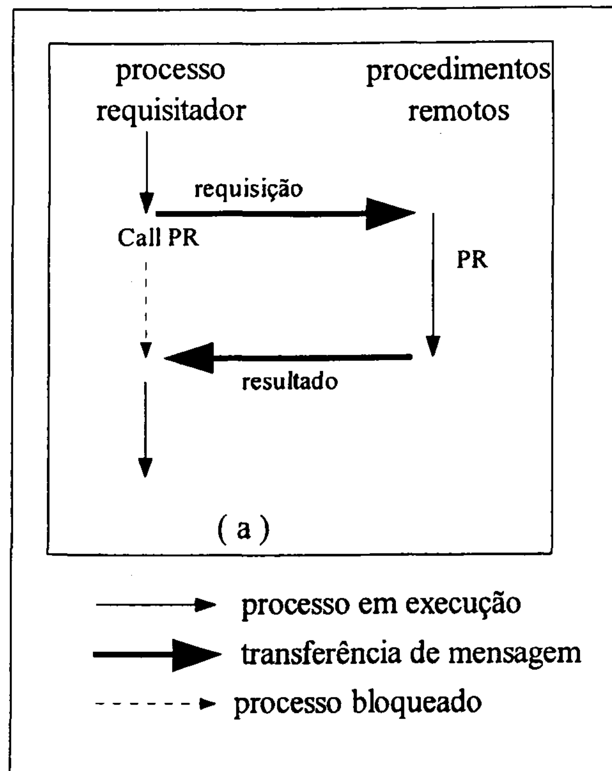

Figura 2.16 - RPC.

Os mecanismos de comunicação e sincronismo em memória compartilhada diferenciam-se pelo contraste entre estruturação e flexibilidade. Enquanto *busy-waiting* oferece flexibilidade e pouca estruturação, monitores são implementados a um nível mais alto, sendo mecanismos mais estruturados e menos flexíveis. Semáforos são mecanismos que apresentam um nível de estruturação e flexibilidade intermediário. Quanto aos mecanismos de memória distribuída, o que os diferencia é simplesmente a forma em que os comandos *send* e *receive* são utilizados.

2.4.4. Suporte Para Programação Paralela

Conhecidos os mecanismos de ativação de processos concorrentes e de comunicação e sincronismo citados nas seções anteriores, uma decisão extremamente importante é a escolha do tipo de ferramenta (que implemente esses

mecanismos) que será usada para a construção de um programa fonte que represente o algoritmo paralelo definido.

Vários aspectos devem ser levados em consideração nesta escolha. Entre eles, o tipo de aplicação, o tipo de usuário que utilizará a ferramenta e a arquitetura que executará os códigos gerados. Baseado no tipo de aplicação e na arquitetura, define-se o a granularidade desejada, que deve ser considerada também no processo de escolha. Segundo Almasi [ALM94], deve-se pesar dois fatores principais: o tempo de trabalho do programador (e aqui inclui-se o tempo de aprendizado da ferramenta), e o desempenho obtido.

Almasi [ALM94], relaciona três tipos de ferramentas para construção de programas paralelos: ambientes de paralelização automática, extensões paralelas para linguagens seriais e linguagens concorrentes.

Compiladores paralelizadores, que caracterizam ambientes de paralelização automática, são responsáveis por gerarem automaticamente versões paralelas de programas não paralelos. Tais compiladores exigem o mínimo de trabalho do usuário, mas o desempenho obtido geralmente é modesto. Além disso, nem sempre esses compiladores são disponíveis, principalmente para sistemas de memória distribuída [BLE94].

Extensões paralelas são bibliotecas que contêm um conjunto de instruções que complementam linguagens seriais já existentes. Estes ambientes requerem algum trabalho do programador, mas ainda evitam a necessidade de aprendizagem de uma nova linguagem ou a completa reescrita do código fonte (no caso onde se paraleliza um programa já implementado). Geralmente, o desempenho obtido é superior ao obtido pelos compiladores paralelizadores acima citados. Existem extensões para arquiteturas de memória distribuída (MPI, PVM, Multilisp) e para memória compartilhada (Pascal Concorrente). Dentro das extensões paralelas, é interessante citar os ambientes de passagem de mensagens, que são ambientes de programação paralela portáteis para memória distribuída, que permitem o transporte de programas paralelos entre diferentes arquiteturas (e sistemas distribuídos) de maneira transparente. Estes ambientes são discutidos no capítulo 4.

O terceiro tipo de ferramenta relaciona as linguagens criadas especialmente para processamento concorrente, o que implica em tempo de aprendizagem de uma linguagem totalmente nova e reescrita total do código fonte. Essas ferramentas tendem a fornecer melhores desempenhos, de maneira que se compense a sobrecarga sobre o programador. Outra vantagem destas linguagens é que geralmente elas possibilitam a construção de códigos bem estruturados, tornando fácil a identificação dos processos que estão executando em paralelo e a comunicação entre eles. Exemplos dessas linguagens são: *Occam* e *Ada*.

De maneira sucinta, a escolha de uma ferramenta para programação paralela, deve ser feita de acordo com os objetivos do programador. Compiladores paralelizadores oferecem desempenho ruim, porém com sobrecarga nula sobre o programador. Por outro lado, linguagens concorrentes oferecem melhor desempenho, mas oferecem uma sobrecarga considerável sobre o programador. Num patamar intermediário de desempenho e sobrecarga sobre o programador, situam-se as extensões paralelas.

2.5. Considerações Finais

A filosofia seqüencial de Von Neumann representou um paradigma computacional eficiente e suficiente para o panorama tecnológico das primeira décadas de existência da computação, onde ambos memória e processador eram recursos caros e preciosos. Na época atual, porém, essas restrições tecnológicas já não apresentam o mesmo peso, e a manufatura de computadores com vários processadores já não é uma opção economicamente inviável [KIR91].

As diversas áreas na qual a computação se aplica demandam cada vez mais poder computacional, e vários autores acreditam que esse poder computacional só pode ser conseguido através do processamento paralelo. Além disso, o aumento de velocidade em processadores seqüenciais tende a alcançar um valor máximo. Segundo Lenatti [LEN95], o processamento paralelo está rapidamente se tornando uma realidade dentro da comunidade empresarial, sendo a sua ótima relação preço/desempenho o seu fator de impulso comercial. Porém, substituir uma filosofia computacional de décadas de existência, como é a de Von Neumann, não é uma tarefa fácil ou rápida [ALM94] [AMO88] [KIR91] [NAV89].

A nível de *hardware*, muito já se desenvolveu em computação paralela, e muita experiência já foi adquirida. Porém, a nível de *software* paralelo, ainda existem muitas lacunas a serem preenchidas, e muito ainda deve ser pesquisado até que se encontrem soluções ótimas que impliquem na utilização dessas arquiteturas paralelas com grande desempenho [NAV89]. Segundo Zaluska [ZAL91], o custo de sistemas computacionais paralelos tende a ser dominado pelo custo de *software*, e características fundamentais que todo *software* paralelo deve oferecer são: ser facilmente transportado entre plataformas de *hardware* diferentes, ser facilmente ampliado para acomodar grandes problemas sem dificuldade e ser de uso fácil.

Dentre as plataformas de execução de programas paralelos, destaca-se o modelo MIMD com memória distribuída, em função de sua grande flexibilidade e facilidade de ampliação [BLE94] [COR96] [MCB94] [ZAL91]. Dentro do modelo MIMD, uma tendência atual é a utilização de sistemas distribuídos, interligados por redes de comunicação rápidas, como plataformas de programação paralela

[BLE94]. Sistemas distribuídos representam uma solução de custo acessível, em comparação às arquiteturas paralelas de alto custo, o que de certa forma expande o uso da computação paralela para uma comunidade maior de usuários [BEG94] [BLE94] [COR96] [ZAL91].

Uma revisão sobre sistemas distribuídos, juntamente com uma discussão sobre a sua utilização para implementação de algoritmos paralelos, é apresentada no próximo capítulo.

Capítulo 3
Sistemas Distribuídos

Este capítulo apresenta conceitos básicos relacionados a sistemas distribuídos, e é dividido nos seguintes tópicos: definição, vantagens e desvantagens, características básicas e comunicação em sistemas distribuídos.

3.1. Introdução

A computação, em geral, tem baseado o seu desenvolvimento numa relação muito forte com o progresso tecnológico. Isto é, a medida que a tecnologia computacional se moderniza, novos conceitos e novas maneiras de se processar informações são criados, sempre buscando o aperfeiçoamento do processo computacional.

A indústria computacional avançou como nenhuma outra. Há pouco mais de uma década, os sistemas computacionais eram em sua maioria **sistemas centralizados** (sistema de único processador). Isto é, grande *manfraimes*, computadores de grande porte e preço, eram compartilhados entre diversos usuários. A maioria das organizações possuíam poucos computadores, e por falta de meios de interconexão estes operavam separadamente [TAN95].

Dois avanços tecnológicos, envolvendo processadores e formas de interconexão de computadores, começaram a mudar este panorama. Poderosos microprocessadores foram desenvolvidos, alguns com poder computacional comparável a *manfraimes*, a custo mais acessível [TAN95]. Além disso, microprocessadores dobraram seu desempenho a cada 18 meses na última década, e continuam a evoluir em uma taxa muito maior que supercomputadores [TUR93].

A tecnologia de interconexão de computadores evoluiu de forma similar. Difundiu-se o uso de redes de computadores, à medida que estas se tornavam mais confiáveis e permitiam maiores taxas de transmissão.

Esta combinação de fatores levou ao desenvolvimento de uma nova forma de organização dos sistemas computacionais, baseada na interconexão de vários computadores, não necessariamente homogêneos, através de uma rede de computadores. Estes são geralmente chamados **sistemas distribuídos**, em contraste aos sistemas centralizados anteriormente citados [TAN95].

Coulouris [COU94] cita ainda um terceiro fator de estímulo para a difusão de sistemas distribuídos: o desenvolvimento do sistema UNIX, que serviu como base para grande parte dos primeiros sistemas distribuídos desenvolvidos.

3.2. Definição de um Sistema Distribuído

Uma definição precisa e completa do que é um sistema distribuído é difícil de apresentar, visto que cada autor na literatura disponível apresenta definições próprias e com diversos pontos conflitantes.

Tanenbaum [TAN95] define um sistema distribuído de maneira simples:

"Um sistema distribuído é uma coleção de computadores independentes que aparecem aos usuários do sistema como um único computador."

Coulouris [COU94], apresenta uma definição um pouco mais extensa:

"Um sistema distribuído consiste de uma coleção de computadores autônomos ligados por uma rede de computadores e equipados com *software* de sistema distribuído. Usuários de um sistema distribuído bem projetado devem perceber um sistema computacional único e integrado mesmo que ele possa estar implementado por muitos computadores em diferentes localizações."

Uma característica muito importante de um sistema distribuído, como pode ser visto nas definições apresentadas, é o fato deste apresentar uma **imagem de sistema único**. Isto é, apesar de existirem vários computadores ligados trabalhando em conjunto, é interessante que esta distribuição fique transparente ao usuário, de maneira que este enxergue todo o ambiente como um sistema único. Porém, existe uma discordância entre alguns autores sobre o limite onde um sistema deixa de ser distribuído à medida que os usuários percebam a existência de múltiplos recursos interligados. Tanenbaum [TAN95], por exemplo, considera a existência de **sistemas operacionais de rede**, que são formados pela interconexão de *workstations* em rede sob o controle do sistema operacional UNIX ou similares. Colouris [COU94], por outro lado, considera este tipo de organização como um sistema distribuído. Nesse trabalho, será aceita a definição de Couloris.

Tanenbaum [TAN95] descreve um conjunto de cinco características, que um projetista deve ter em mente durante a implementação de um sistema distribuído. Estas características são: transparência, flexibilidade (o sistema deve ser projetado de maneira que seja fácil a modificação de quaisquer um de seus componentes), confiabilidade (incluindo disponibilidade, segurança e tolerância a

falhas), desempenho e escalabilidade. Essas características são melhor descritas na seção 3.4.

Sistemas distribuídos podem ser implementados em plataformas com grande variação de tamanhos. Desde uma única rede local isolada até a *Internet* (rede mundial de computadores que já agrega milhões de computadores ligados em redes locais ou metropolitanas), podem ser exemplos de plataformas de implementação de sistemas distribuídos [COU94].

As aplicações de sistemas distribuídos também apresentam uma grande variação. Pode-se citar desde a provisão de computação de propósito geral para grupo de usuários até aplicações comerciais (como automação bancária, por exemplo) e aplicações multimídia [COU94].

Além disso, sistemas distribuídos oferecem uma plataforma de memória distribuída para execução de programas paralelos, substituindo de maneira eficiente arquiteturas paralelas, que apresentam alto custo de aquisição e manutenção. Porém, deve-se ressaltar que tais plataformas são adequados a programação paralela de alta granularidade e com poucas operações de comunicação entre processos, visto que apresentam maior sobrecarga de comunicação. A utilização de sistemas distribuídos como plataformas de programação paralela é apresentada com detalhes na seção 3.4, em virtude da sua relevância neste trabalho.

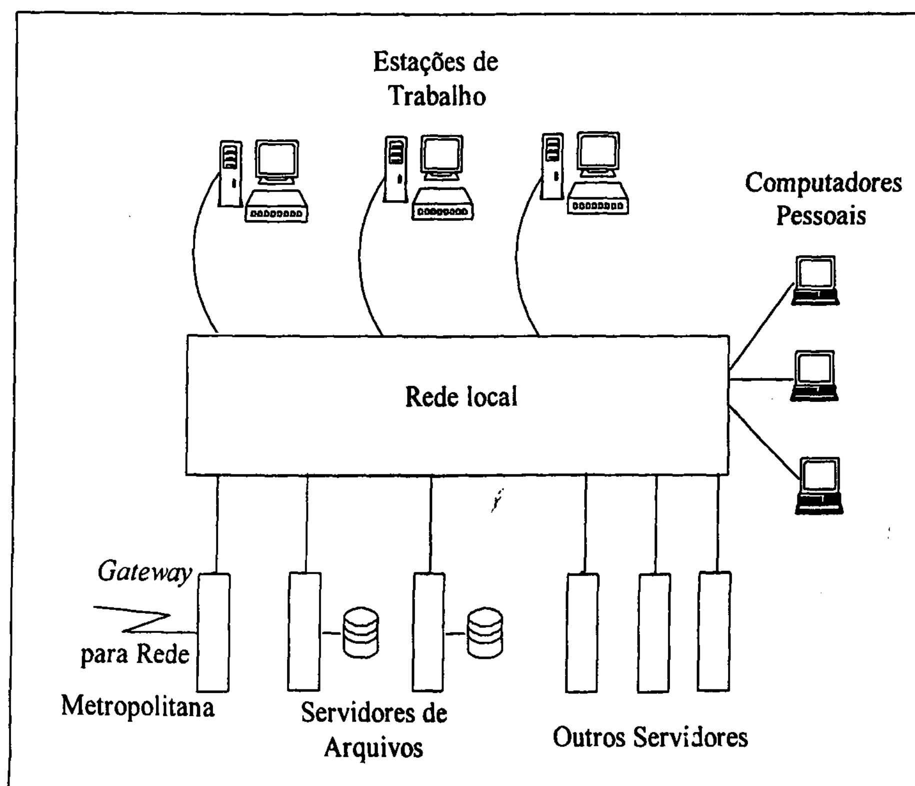

Figura 3.1 - Modelo Simples de um Sistema Distribuído.

Um exemplo de um sistema distribuído simples pode ser visto na figura 3.1 [COU94]. O sistema apresentado é constituído de estações de trabalho e computadores pessoais, ligados via uma rede local a vários servidores (que são computadores utilizados para fornecer serviços dentro do sistema). Além disso, o sistema é ligado a uma rede metropolitana via um *gateway*, que é um computador que possui a função de coordenar a comunicação entre redes com diferentes protocolos. A figura esquematiza a arquitetura do sistema, sendo que deve haver o chamado *software distribuído* controlando toda a atividade deste sistema e procurando esconder do usuário os detalhes de interconexão dos componentes.

O uso de sistemas distribuídos tem-se difundido nos últimos anos, em substituição aos tradicionais sistemas centralizados. A próxima seção apresenta uma comparação entre esses dois tipos de organização de sistemas computacionais, através da apresentação de vantagens e desvantagens de sistemas distribuídos em relação a sistemas centralizados.

3.3. Vantagens/Desvantagens de Sistemas Distribuídos

Segundo Tanenbaum [TAN95], um sistema distribuído oferece as seguintes vantagens:

- Microprocessadores interligados oferecem uma melhor relação preço/desempenho em comparação a *manframes*;
- Vários microprocessadores interligados podem alcançar um poder computacional que *manframes*, até por limitações físicas, não conseguiriam alcançar;
- Algumas aplicações são inerentemente distribuídas;
- O sistema é mais confiável. A quebra de uma máquina não implica que o sistema inteiro deixe de funcionar;
- Poder computacional pode ser acrescentado ao sistema de maneira gradual, como por exemplo, pelo acréscimo de novas CPU's.

Porém, apesar de apresentar várias características favoráveis, sistemas distribuídos possuem também alguns pontos fracos. Por exemplo [TAN95]:

- *Software* para sistemas distribuídos ainda é um ponto relativamente frágil. Apesar da grande evolução apresentada neste aspecto, existe ainda bastante controvérsia entre os autores da área, o que deve implicar em alguma dificuldade na consolidação de conceitos relacionados à maneira ideal de se organizar *software* distribuído;
- Um sistema distribuído geralmente se baseia em uma rede de computadores, que são meios que podem saturar e danificar dados, entre outros problemas. Estes aspectos devem ser convenientemente tratados pelo sistema, o que implica em custo de execução elevado. Além disso, redes de computadores apresentam algumas restrições em relação a segurança de dados privados.

Mesmo apresentando pontos desfavoráveis, acredita-se que as vantagens oferecidas pelos sistemas computacionais distribuídos superam e compensam quaisquer possíveis desvantagens [TAN95].

3.4. Características Básicas de Sistemas Distribuídos

Coulouris [COU94] apresenta um conjunto de seis características, que não são conseqüências automáticas da distribuição, mas que todo sistema distribuído bem projetado deve oferecer. Essas características, que são discutidas a seguir, demonstram a potencialidade e utilidade que esses sistemas têm a oferecer.

Compartilhamento de Recursos: Característica de vital importância, presente tanto em sistemas distribuídos como em centralizados, e que pode ser citada como uma vantagem destes tipos de sistemas em relação ao uso de computadores isolados [TAN95]. Entende-se, nesse caso, recurso como uma entidade computacional em geral, que pode representar *software*, *hardware* ou dados. Dependendo do recurso a ser compartilhado entre diversos usuários, alguns benefícios podem ser citados:

- Periféricos em geral (impressoras, por exemplo) podem ser compartilhados por conveniência ou com o intuito de se reduzir custos;
- Bancos de dados podem ser compartilhados, o que é fundamental em diversas aplicações, como por exemplo, automação bancária;

- *Software* básico (como editores, compiladores, etc.) pode ser compartilhado, de maneira que todos os usuários tenham acesso a versões mais recentes. Além disso, pesquisadores trabalhando em conjunto podem enxergar o projeto como um todo, visto que cada membro do grupo tem acesso ao trabalho dos companheiros.

Um problema gerado pelo compartilhamento de recursos relaciona-se à segurança do sistema. A partir do momento que se pode compartilhar dados, cria-se uma maneira para que informações sigilosas sejam alcançadas sem a devida permissão.

Um determinado recurso em um sistema distribuído geralmente é armazenado em um computador único (servidor), de tal maneira que devam ser usadas primitivas de comunicação para que estes recursos possam ser acessados remotamente (Formas de comunicação entre processos são discutidas na seção 3.5). Além disso, deve haver um processo que exerça controle sobre o recurso, a partir do qual este possa ser manuseado. Este processo é designado **administrador de recurso**.

Baseado no conceito de compartilhamento de recursos, a literatura apresenta dois modelos para a organização de sistemas distribuídos, que são: modelo cliente-servidor e modelo baseado em objetos.

Modelo cliente-servidor: Trata-se do modelo mais conhecido e utilizado. De maneira simples, um sistema distribuído é constituído de uma série de processos servidores (administradores de recursos) para um determinado conjunto de recursos e uma série de processos clientes acessando esses recursos. Quando necessitam de algum recurso, clientes enviam requisições aos servidores, que atendem estas requisições e enviam uma resposta. Além disso, servidores podem ser clientes de outros servidores, de maneira que um mesmo processo pode exercer as duas funções ao mesmo tempo.

Este modelo oferece uma abordagem eficiente, de propósito geral, que permite o compartilhamento de recursos e que pode ser implementado em uma grande variedade de plataformas de *software* e *hardware*.

Deve ser ressaltado que recursos geralmente não são fornecidos por um único servidor centralizado, por motivos de confiabilidade (Se este único computador quebra, o recurso não será mais disponível). Nesse caso, define-se um **serviço** como uma entidade computacional que representa um determinado recurso que pode estar sendo oferecido por mais de um servidor.

Modelo baseado em objetos: Este modelo segue a abordagem de orientação a objetos, que é um paradigma de programação onde entidades dentro de um programa são consideradas objetos. Neste modelo, é definida uma interface que especifica as operações que podem ser aplicadas sobre cada objeto. Dessa maneira, recursos compartilhados dentro deste modelo são considerados objetos, e o conjunto de procedimentos e variáveis de estado que caracterizam o acesso a estes recursos são denominados **administradores do objeto** (entidades computacionais similares aos administradores de recursos).

Uma característica importante do modelo baseado em objetos é o fato de que todos os objetos são identificados de maneira uniforme, sendo que estes podem migrar dentro do sistema distribuído sem a necessidade de mudar seus identificadores. Dessa maneira, é fornecida uma maneira simples e eficiente de se compartilhar recursos.

É claro que toda a simplicidade que este modelo oferece implica em uma grande dificuldade para a sua implementação, principalmente se ocorre migração dos objetos dentro do sistema, visto que deve-se manter a uniformidade de identificação dos objetos.

Abertura: O conceito de um sistema computacional aberto é ligado a extensibilidade deste sistema através do acréscimo de novos componentes a ele. Um sistema é aberto quando ele permite a inclusão de módulos de *hardware* ou *software* sem que haja prejuízo dos módulos já existentes. Para tornar esta característica possível, é necessário que todas as interfaces que o sistema oferece, tanto para aplicações como para periféricos, sejam completamente definidas e disponíveis.

Quando se estende este conceito a sistemas distribuídos, é necessário também que se ofereçam primitivas de comunicação entre processos bem definidas, de maneira que módulos acrescentados ao sistema possam oferecer seus serviços a quaisquer computadores incluídos na rede. Sistemas projetados de maneira a permitir o acréscimo de serviços, que ofereçam recursos a serem compartilhados, sem prejuízo de serviços já existentes são denominados **sistemas distribuídos abertos**.

Escalabilidade: Um sistema distribuído não possui um valor fixo que limite o número de computadores que o forma. Pode variar entre alguns poucos computadores até várias centenas deles. Um sistema distribuído não deve necessitar de mudanças à medida que se acrescente novos componentes ou se

substitua componentes já existentes por versões mais modernas. Um sistema que oferece esta característica é denominado **ampliável**.

Um sistema pode ser ampliado através da inclusão de novos computadores, de redes de maior desempenho, entre outros, respondendo a alguma necessidade gerada por determinados fatores. Por exemplo, a inclusão de novos usuários em um sistema pode causar um gargalo em determinado servidor de arquivos. Nesse caso, um novo servidor de arquivos deve ser anexado ao sistema. Um sistema ampliável deve permitir que essa operação seja feita sem a necessidade de mudanças no sistema, e sem que o usuário perceba.

Tolerância a Falhas: Sistemas computacionais podem falhar, seja em *hardware* ou *software*, ocasionando resultados errados. Um sistema tolerante a falhas deve detectar a falha e desfazer os efeitos causados por ela. Duas abordagens são geralmente usadas:

- **Redundância de *hardware***: não devem haver componentes únicos em um sistema. Por exemplo, se um computador que executa uma tarefa importante em um sistema distribuído quebra, deve haver algum outro que o substitua, de maneira que o sistema não pare;
- **Recuperação de *software***: *software* em geral deve ser projetado cuidadosamente, de maneira que erros de execução possam ser recuperados. Por exemplo, se dados globais ao sistema são manuseados por um processo que eventualmente falhe, quaisquer atualizações feitas nesses dados devem ser desfeitas.

Falhas em sistemas centralizados geralmente ocasionam a não operabilidade do sistema como um todo, enquanto que sistemas distribuídos, em virtude do grande potencial apresentado para tolerância a falhas, mantêm-se trabalhando (possivelmente, com perda de desempenho). Neste caso, diz-se que sistemas distribuídos possuem alto grau de **disponibilidade**, isto é, mantêm-se disponível aos usuários a maior parte do tempo.

Transparência: A distribuição de componentes é uma característica natural em um sistema distribuído, sendo que grande parte das vantagens que este apresenta são derivadas desta propriedade. Um sistema distribuído deve, idealmente, esconder esta característica de maneira que o usuário não perceba esta distribuição, apresentando uma **imagem de sistema único** [TAN95]. O usuário deve ter a

impressão de que está trabalhando, em um único computador e com todos os recursos dedicados a ele. Quanto menor a percepção do usuário da distribuição dos elementos, maior a transparência de um sistema distribuído.

Tanenbaum [TAN95] apresenta alguns tipos de transparência, que um sistema distribuído bem projetado deve procurar oferecer. É usado o termo **objeto de informação** para denotar entidades computacionais às quais a transparência se aplica [COU94].

- **Transparência de localização**: objetos de informação são acessados de maneira análoga (sejam remotos ou locais), sem conhecimento de suas localizações;
- **Transparência de replicação**: objetos de informação são replicados para aumentar a performance e a confiabilidade, sem o conhecimento do usuário;
- **Transparência de migração**: objetos de informação migram dentro do sistema sem a percepção do usuário;
- **Transparência de concorrência**: processos acessam objetos de informação compartilhados, sem interferência entre eles;
- **Transparência de paralelismo**: eventos ocorrem em paralelo, sem o conhecimento do usuário.

Concorrência: Sistemas distribuídos são formados por vários computadores, o que implica na existência de vários elementos de processamento, e torna estes ambientes um berço natural para processamento paralelo.

Exemplos de paralelismo em um sistema distribuído são:

- Vários usuários simultaneamente executando suas aplicações no sistema;
- Vários processos servidores executando concorrentemente atendendo a requisições de diferentes clientes.

Além disso, programas paralelos, antes restritos a arquiteturas paralelas, podem ser executados em um sistema distribuído. Esta característica de um sistema distribuído é discutida em detalhes a seguir, visto a sua importância neste trabalho.

Programação Paralela em Sistemas Distribuídos: Segundo Beguelin [BEG94], dois fatores foram especialmente importantes para o desenvolvimento da computação paralela: os processadores maciçamente paralelos (MPP) e a difusão da computação distribuída.

Sistemas distribuídos podem ser utilizados como plataformas de memória distribuída para a execução de programas paralelos, formando as chamadas **máquinas paralelas virtuais**. Com a existência de redes de computadores de alta performance ligando várias estações de trabalho de propósito geral, obtém-se um poder computacional que pode exceder o poder de um computador paralelo de alto desempenho.

A vantagem principal que a computação distribuída pode oferecer neste contexto é o custo. Arquiteturas paralelas geralmente são muito caras e de uso específico. Através da utilização de um sistema distribuído, pode-se ter uma plataforma de programação com um custo normalmente mais baixo, que pode ser usado tanto para este fim como para compartilhamento de recursos e informações ou para outros objetivos de um sistema distribuído [BEG94].

A utilização de redes de computadores para computação paralela, em contraste a arquiteturas paralelas, apresenta problemas e vantagens. A principal característica que pode ser citada quando se utiliza uma rede de computadores é a heterogeneidade de componentes. Em arquiteturas maciçamente paralelas (MPP's), por exemplo, todos processadores são exatamente iguais em capacidade, recursos, *software* e velocidade de comunicação, o que não necessariamente acontece em uma rede de computadores. Os componentes desta rede podem ser de diferentes fabricantes ou ter diferentes compiladores. Deve-se lidar com vários tipos de heterogeneidade derivados da interconexão de computadores, entre elas: arquiteturas, formato de dados, potência computacional, carga de trabalho em cada máquina e carga de trabalho na(s) rede(s).

Uma rede de computadores pode incluir vários tipos de arquiteturas, como por exemplo: PC's 486/Pentium, estações de trabalho de alta performance, multiprocessadores, multicomputadores, entre outros. Cada arquitetura pode possuir características diferentes para o desenvolvimento de programas. Esse aspecto pode ser visto como uma vantagem deste tipo de sistema, uma vez que fornece alta flexibilidade, ou como uma desvantagem. Computadores diferentes podem possuir formatos de dados diferentes, o que torna a comunicação entre os vários componentes de um sistema distribuído de difícil execução.

Mesmo para o caso de formatos de dados iguais, pode haver diferença de velocidades entre os diversos computadores interligados. Os computadores mais rápidos não devem ficar inativos esperando dados de computadores mais lentos. Outro fator que influi no aspecto velocidade, é o balanceamento de carga de

trabalho nas diferentes máquinas. Visto que podem haver vários usuários utilizando o sistema ao mesmo tempo, o poder computacional efetivo pode variar drasticamente no tempo, mesmo em computadores de mesma velocidade, dificultando ainda mais o balanceamento. Além da carga de trabalho em cada máquina, deve-se considerar a carga de trabalho na rede, visto que o tempo de tráfego de uma mensagem na rede depende da carga imposta pelos outros usuários.

Além disso, a sobrecarga de comunicação em uma rede de computadores é mais significativa que em uma arquitetura paralela, de maneira que deve-se pesar cuidadosamente a relação entre execução efetiva e operações de comunicação. A utilização de tecnologias de interconexão de computadores mais rápidas e confiáveis tende a minimizar este problema [BEG94].

Apesar dos problemas gerados pela heterogeneidade de componentes, a computação distribuída oferece várias vantagens para a programação paralela, entre elas:

- O uso de *hardware* já existente evita a necessidade de um alto investimento;
- O desempenho pode ser melhorado através da distribuição de tarefas para as arquiteturas mais apropriadas para executá-las;
- Pode-se explorar a natureza heterogênea de determinadas aplicações;
- Os componentes da máquina virtual podem evoluir gradativamente através da utilização de tecnologia de ponta;
- A produtividade do programador é maior devido a utilização de ambientes (editores, compiladores, *debuggers*) familiares;
- Computação distribuída facilita o trabalho cooperativo.

Todos esses fatores procuram oferecer menores tempos de desenvolvimento e depuração de programas, custos reduzidos, e melhor desempenho para determinadas aplicações. Estas são as vantagens que todos os ambientes de programação que utilizam sistemas distribuídos devem oferecer.

Sistemas distribuídos são capazes de oferecer muitos benefícios ao usuário, os quais são conseqüências diretas de suas características básicas, e que representam ponto de incentivo para a difusão da utilização da computação distribuída.

Um ponto de fundamental importância, que está relacionado de alguma maneira com todas as características básicas de um sistema distribuído e que é um

fator crítico para o seu desempenho, trata-se dos métodos de comunicação entre processos. Como foi exposto, um sistema distribuído caracteriza-se pela distribuição de seus componentes, o que leva à necessidade de se implementar primitivas de comunicação eficientes que permitam cooperação eficiente destes componentes. Este aspecto torna-se de vital importância quando se programa paralelamente em sistemas distribuídos, visto que a eficiência de um programa paralelo se baseia em primitivas de comunicação entre processos eficientes.

A comunicação entre processos em sistemas distribuídos é apresentada em detalhes na próxima seção.

3.5. Comunicação em Sistemas Distribuídos

Sistemas distribuídos, em virtude de não possuírem espaços de endereçamento de memória globais, utilizam primitivas de comunicação entre processos via troca de mensagens [COU94] [TAN95].

A comunicação via troca de mensagens ocorre através do uso de primitivas *send/receive* (seção 2.4.3.). Dois tipos de estruturas baseadas nestas primitivas são utilizadas para a implementação de operações de comunicação em sistemas distribuídos: modelo cliente-servidor e comunicação de grupo.

3.5.1. Comunicação Cliente-Servidor

O modelo de comunicação cliente-servidor é orientada para a provisão de serviços, e se baseia no modelo cliente-servidor de sistemas distribuídos [COU94] [TAN95]. Uma operação de comunicação consiste de:

1. Transmissão de uma requisição de serviço de um processo cliente para um processo servidor;
2. Execução da requisição pelo servidor;
3. Transmissão de uma resposta para o cliente.

Uma maneira estruturada de realizar esta operação é através do uso de RPC (Remote Procedure Call), ou chamada de procedimento remoto, que é uma abstração de nível mais elevado da comunicação cliente-servidor. Este mecanismo caracteriza-se por esconder os detalhes de comunicação entre processos, através do uso do paradigma de chamada de procedimentos, que é um recurso já conhecido e dominado pelo programador comum. De maneira básica, um cliente acessa um

serviço remoto através da execução de um procedimento, que é chamado de maneira análoga a um procedimento local, mas que é executado em uma máquina remota. Todo o mecanismo de transmissão da mensagens é transparente para o usuário. Informações podem ser transportadas como parâmetros do procedimento e respostas podem ser transmitidas como resultados destes procedimentos [COU94] [TAN95].

Toda a operação de comunicação entre cliente e servidor é controlada por procedimentos chamados ***stubs***, que supervisionam todos os detalhes da comunicação, como por exemplo, o empacotamento dos parâmetros e respostas e o envio de mensagens para os respectivos servidores. Cada procedimento remoto possui um *stub* correspondente. O esquema RPC é apresentado na figura 3.2.

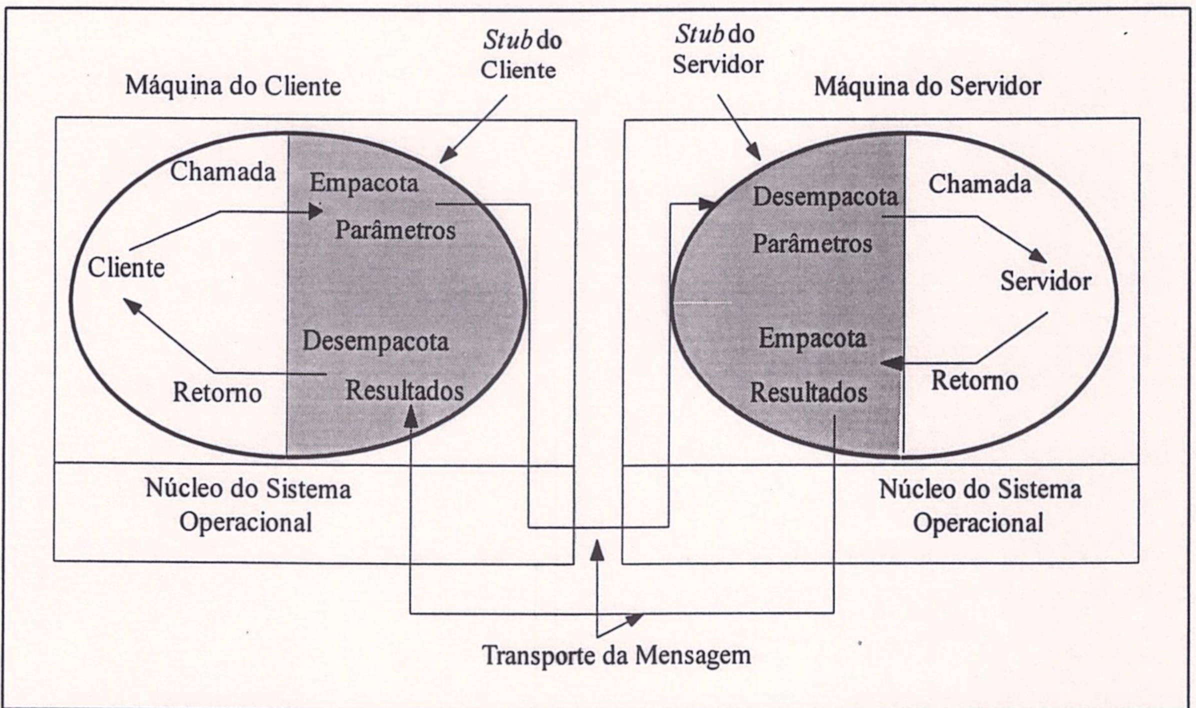

Figura 3.2 - O Mecanismo RPC.

Embora o RPC apresente uma idéia básica simples que oferece vantagens em função de sua similaridade com chamadas de procedimentos locais, vários problemas decorrentes do seu uso são bastante discutidos na literatura [COU94] [SOU95] [STE90] [TAN95] [WIL87]. Exemplos desses problemas são: cliente e servidor executam em máquinas diferentes, com espaços de endereçamento distintos, o que pode causar complicações; ambos o cliente e o servidor podem falhar, e cada possível falha causa diferentes problemas. A maioria desses

problemas já apresentam soluções satisfatórias, de maneira que, o mecanismo RPC é muito empregado na implementação da grande maioria dos sistemas distribuídos.

3.5.2. Comunicação de Grupo

O modelo de comunicação cliente-servidor caracteriza-se pela participação de dois processos. Determinadas circunstâncias necessitam da comunicação entre vários processos, não somente dois. Um grupo é uma coleção de processos que agem em conjunto para a realização de alguma operação. Na comunicação de grupo, uma mensagem remetida para este grupo é recebida por todos os seus processos componentes [COU94] [TAN95].

Exemplos do uso de comunicação de grupo são:

- **Localização de um recurso**: um cliente remete uma mensagem procurando um determinado recurso para um grupo de servidores, e apenas aquele que o possuir responde a mensagem;
- **Tolerância a falhas**: um cliente envia uma requisição para vários servidores, a fim de que se garanta que servidores quebrados não inviabilizem a operação;
- **Atualização múltipla**: um evento como "são 18:00 horas" pode ser enviado para diversos servidores interessados.

3.6. Considerações Finais

Muito já se pesquisou sobre sistemas distribuídos, existindo um grande número de sistemas implementados, tendo-se adquirido uma larga experiência na área. Exemplos de sistemas já implementados são: Amoeba [COU94] [TAN95], Mach [COU94] [TAN95], Chorus [COU94] [TAN95], TRICE [SAN89a] [SAN89b], etc.. Porém, ainda existem muitas contradições entre os diversos autores da área. Muito necessita ser estudado e discutido até que se chegue a um consenso.

Apesar das contradições, algumas características de sistemas distribuídos são consideradas qualquer que seja o autor, como por exemplo: transparência, flexibilidade e desempenho. Além disso, a maioria dos autores concordam sobre a importância deste tipo de organização para sistemas computacionais no futuro da computação (tanto a nível acadêmico como comercial) [COU94] [ENG95] [TAN95]. Vários motivos explicam essa tendência: as vantagens da computação

distribuída sobre a computação centralizada, a adequação destes sistemas à programação paralela, a grande difusão da *Internet* (que é um berço de grande potencial para aplicações distribuídas de grande escala), etc.

A evolução da computação distribuída está intimamente ligada à evolução da tecnologia de interconexão de computadores, visto que a eficiência de sistemas distribuídos baseia-se no fornecimento de altas velocidades de comunicação. Novas tecnologias de interconexão (FDDI, SONET e ATM, por exemplo) ligando microprocessadores cada vez mais poderosos, tendem a fornecer plataformas extremamente eficientes para a utilização da computação distribuída [BEG94].

Nos últimos anos, a computação distribuída tem apresentado, em muitos aspectos, uma relação de convergência com a computação paralela [ZAL91]. Apesar das duas áreas terem surgido por razões diferentes, apresentam características (escalabilidade, tolerância a falhas e concorrência) e problemas (balanceamento de carga) comuns. E também, sistemas distribuídos têm sido utilizado eficientemente como plataformas de programação paralela, formando as chamadas máquinas paralelas virtuais.

Dentro deste contexto, inclui-se a importância da programação via troca de mensagens e das plataformas de portabilidade, que se destinam a fornecer maneiras de se utilizar sistemas distribuídos como ambientes de programação paralela, e assim fornecer formas flexíveis e de menor custo para a construção e execução de algoritmos paralelos. A programação via troca de mensagens e as plataformas de portabilidade são discutidas no próximo capítulo.

Capítulo 4

Programação via Troca de Mensagens

Este capítulo apresenta o paradigma de programação via troca de mensagens em detalhes, através da discussão dos seguintes tópicos: conceitos básicos, ambientes de programação, biblioteca de troca de mensagens genérica e discussão do MPI.

4.1. Introdução

Dentro da computação paralela, tem-se destacado o modelo computacional MIMD, em virtude de sua flexibilidade e potencialidade para a execução de programas paralelos de média e alta granularidade. Entre as plataformas MIMD, destacam-se as plataformas paralelas de memória distribuída, que podem ser computadores paralelos ou máquinas paralelas virtuais [BLE94] [MCB94] [ZAL91].

Como já foi exposto, é essencial que exista cooperação (comunicação e sincronismo) entre processadores trabalhando em conjunto na execução de uma aplicação. Quando se possui memória compartilhada, pode-se conseguir tal cooperação através de espaços de memória compartilhados entre os diversos processos paralelos. Porém, em caso de memória distribuída, quando cada processador possui seu próprio dispositivo de memória local, devem ser definidas primitivas explícitas que possibilitem que os processos se comuniquem e se sincronizem [ALM94] [AND83] [KIR89] [QUI87] [SNO92].

Para esse fim, é definido um conjunto de primitivas que permitem que os processos troquem mensagens entre si, requisitando explicitamente dados de outros processadores. Estas primitivas de comunicação entre processos, caracterizam o paradigma de **troca de mensagens**.

Angela Quealy [QUE94] apresenta a seguinte definição:

"**Troca de mensagens** é um método para a comunicação de processos quando não há compartilhamento de memória. Ela é necessária porque:

- Memórias são locais aos processadores;
- Não há compartilhamento de variáveis;
- Única maneira de se conseguir dados de outras memórias."

O paradigma de troca de mensagens tem-se tornado extremamente popular em tempos recentes. Podem-se relacionar alguns fatores que justificam a sua grande aceitação [DON95]:

- Programas utilizando troca de mensagens podem ser executados em uma grande variedade de plataformas, como arquiteturas paralelas de memória distribuída e sistemas distribuídos. Apesar deste tipo de paradigma não se adaptar naturalmente a arquiteturas de memória centralizada, não há um fator que inviabilize a sua execução neste tipo de arquitetura;
- Adequa-se naturalmente a arquiteturas ampliáveis. Segundo McBryan [MCB94], a capacidade de aumentar o poder computacional de maneira razoavelmente proporcional ao aumento de componentes de um sistema, trata-se de um fator chave para o desenvolvimento da computação paralela, afim de que seja possível a conquista dos grandes desafios computacionais do final do século *(great challenge)*. Arquiteturas de memória distribuída apresentam melhores características de escalabilidade [BLE94], estas que são as arquiteturas em que melhor se caracteriza o paradigma de troca de mensagens;
- Não deve se tornar obsoleto, nas próximas décadas, por redes mais rápidas ou arquiteturas que combinem memória compartilhada e memória distribuída. A algum nível, sempre será necessário utilizar-se de alguma maneira troca de mensagens.

4.2. Ambientes de Programação via Troca de Mensagens

Um programa que utiliza troca de mensagens pode ser definido, de maneira simples, como um conjunto de programas seqüenciais, distribuídos em vários processadores, que se comunicam através de um conjunto limitado e bem definido de instruções. Estas instruções formam o **ambiente de troca de mensagens** e são disponíveis através de uma **biblioteca de troca de mensagens** [MCB94].

Um ambiente de programação via troca de mensagens é formado pelos seguintes componentes:

- Uma linguagem seqüencial (como C e Fortran): a qual será utilizada na implementação dos programas seqüenciais nos processadores;
- A biblioteca de troca de mensagens: a qual fornece as ferramentas necessárias para a ativação e cooperação entre os processos paralelos.

Não necessariamente os programas nos diversos processadores devem ser distintos. Pode-se utilizar o paradigma **SPMD** (Single Program - Multiple Data). Este paradigma implica que seja distribuído pelos processadores o mesmo código fonte, e cada processador deve executá-lo de maneira independente, o que implica na execução de diferentes partes deste programa em cada um dos processadores. Quando se distribui códigos fonte distintos para os processadores, utiliza-se o paradigma **MPMD** (Multiple Program - Multiple Data) [COR96].

Visto que os diferentes programas em um ambiente MPMD podem ser unidos em um só, excetuando-se um eventual gasto maior de memória em cada processador, praticamente não há perda de performance na utilização do paradigma SPMD. Ganha-se, nesse caso, na simplicidade de gerenciamento de um sistema deste tipo [MCB94].

Os primeiros ambientes de programação via troca de mensagens surgiram junto às arquiteturas paralelas de memória distribuída. À medida que fabricantes lançavam novos modelos no mercado, surgiam seus respectivos ambientes de programação, geralmente incompatíveis entre si por serem ligados às arquiteturas que os geraram. Exemplos que podem ser citados desses sistemas são: CROS da *Caltech*, NX1 da *Intel*, NX2 da *Intel*, PSE da *nCUBE*, EUI da *IBM*, CS da *Meiko*, CMMD da *Thinking Machines* [MCB94].

Esta especificidade arquitetura/ambiente tornava a portabilidade de programas entre sistemas diferentes de difícil, senão impossível, execução. Para a solução deste tipo de problema, surgiram as chamadas **plataformas de portabilidade** [MCB94]. Nesse caso, um ambiente de programação é definido e implementado em várias plataformas paralelas (por plataforma paralela, entenda-se arquiteturas paralelas ou sistemas distribuídos), possibilitando que programas possam ser portados de maneira fácil e direta. Essas plataformas de portabilidade também participaram ativamente do processo de migração da programação paralela para sistemas distribuídos em virtude da adequação natural destes tipos de plataformas para ambientes heterogêneos, formados pela ligação de diferentes arquiteturas em um ambiente único. Podem ser citados como exemplos de plataformas de portabilidade [KIT95] [MCB94]: Express [FLO94], Linda [CAR94], p4 [BUT94], PARMACS [CAL94], ZipCode [SMI94] e principalmente o PVM [BEG94] [SUN94]. Este último, trata-se da plataforma que alcançou maior aceitação e atualmente, pode ser descrito como um padrão "de fato" de

programação neste tipo de sistema. O PVM é descrito com maiores detalhes no capítulo 5, visto que foi utilizado como ferramenta neste trabalho.

Porém, o grande número de plataformas de portabilidade existentes gera um problema semelhante ao de ambientes específicos a arquiteturas em relação à real portabilidade de programas. Além disso, grande parte destas plataformas suportam apenas um subconjunto das características de determinadas arquiteturas, ocasionando o não uso de características importantes de algumas delas [MCB94].

Baseado nesses aspectos, iniciou-se um processo de padronização para plataformas de portabilidade, que agregou vários representantes de várias organizações, principalmente européias e americanas. Este padrão, que não é apoiado por nenhuma organização oficial, foi nomeado **MPI** (**Message Passing** Interface) [DON95] [DOS96] [MCB94] [WAL94].

O MPI baseia-se nas melhores características de todas as plataformas de portabilidade, levando-se em consideração as características gerais das plataformas paralelas, tentando explorar as vantagens de cada uma delas. Este padrão será discutido em detalhes na seção 4.4, em virtude da sua importância neste trabalho.

Na próxima seção, é apresentado um conjunto básico de rotinas necessárias para a implementação de uma biblioteca de troca de mensagens genérica.

4.3. Biblioteca de Troca de Mensagens

Angela Quealy [QUE94] descreve um conjunto de rotinas básicas necessárias para a formação de uma biblioteca de troca de mensagens. É importante ressaltar que a descrição apresentada é feita de maneira genérica, e o conjunto de rotinas e respectivas sintaxes podem variar de acordo com o ambiente de programação.

4.3.1. O Que é Uma Mensagem?

Antes de se discutir cada uma das rotinas, é importante definir precisamente o que é uma mensagem, e as informações que são necessárias para o controle de uma operação de troca de mensagens.

Qualquer tipo de informação que deve ser transferida entre processos que não compartilhem memória, deve ser transportada explicitamente via mensagens. Pode-se dizer que o processo necessita de um dado que está fora do seu espaço de endereçamento, ou seja, não está armazenado em sua memória local. Portanto, uma

troca de mensagens pode ser entendida, de maneira geral, como um acesso a memória não local. Por exemplo, se um processo executando no processador *Pn*, necessita de um dado do processo executando no processador *P0*, então este dado será transferido explicitamente da memória local a *P0* e será copiado na memória local a *Pn*. Este modelo genérico é mostrado na figura 4.1 [QUE94].

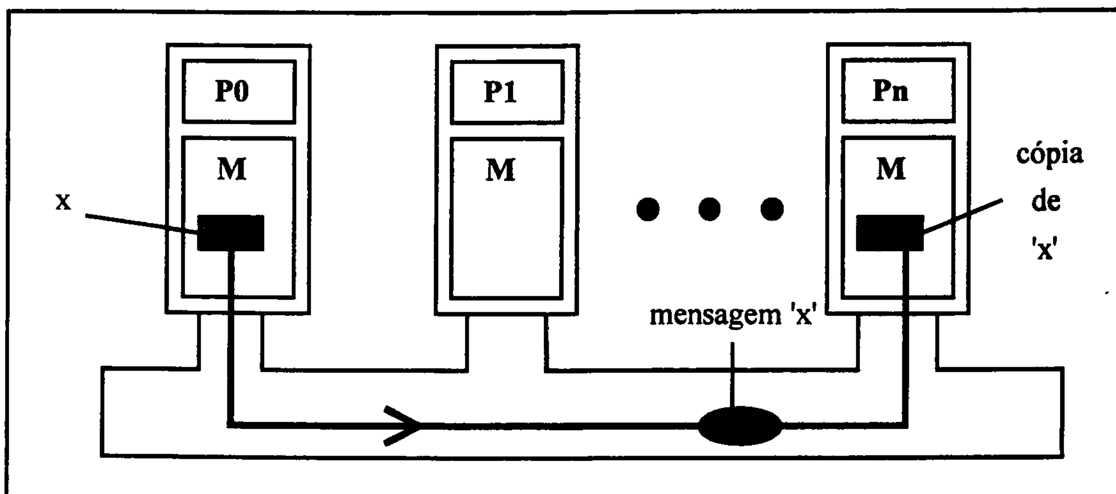

Figura 4.1 - Transferência de uma Mensagem.

Para uma operação de troca de mensagem entre processos, há um conjunto de informações que devem ser consideradas, de maneira que tal operação esteja completamente definida [MAC96]. Estas informações são:

- Qual processo está enviando a mensagem (transmissor);
- Qual os dados que formam esta mensagem;
- Qual o tipo, ou tipos, dos dados;
- Qual o tamanho da mensagem;
- Qual processo vai receber a mensagem (receptor);
- Aonde os dados serão armazenados no processo receptor;
- Qual a quantidade de dados suportada pelo receptor, de maneira que não se envie dados que o receptor não esteja preparado para receber.

Estas informações devem ser supervisionadas pelo sistema que está gerenciando a transferência, e algumas delas devem ser anexadas à mensagem. Uma informação que também deve fazer parte de uma mensagem, trata-se de um identificador, a partir do qual a mensagem possa ser selecionada pelo processo

receptor. Outras informações, como o identificador do processo fonte, podem ser utilizadas para a seleção de mensagens.

Uma mensagem pode ser transmitida ou recebida de diferentes "maneiras", variando na forma de comunicação (síncrona ou não) e também em relação ao número de processos envolvidos na operação. Nas próximas seções serão discutidas as rotinas de comunicação ponto-a-ponto (seção 4.3.2) e coletivas (seção 4.3.3).

Além das rotinas para comunicação, uma biblioteca de troca de mensagens deve ainda fornecer rotinas que retornem informações específicas, as quais são discutidas nas seções 4.3.4 e 4.3.5.

4.3.2. Rotinas de Comunicação Ponto-a-Ponto

Estas rotinas são responsáveis pela ação básica de uma biblioteca de troca de mensagens, que se trata da transferência de uma mensagem. Cada transferência ponto-a-ponto envolve exatamente dois processos, um que transmite a mensagem (denominado transmissor) e um que a recebe (denominado receptor). Estas operações são feitas pela utilização de duas rotinas: ***send( )*** que é responsável pela transmissão e ***receive( )*** que é responsável pela recepção de uma mensagem.

Rotina *Send( )*: De maneira genérica, esta rotina é definida por:

send(msg_id, msg, tamanho, destino)

onde:

- *msg_id*: identificador da mensagem;
- *msg*: dado(s) a ser(em) transmitido(s);
- *tamanho*: tamanho dos dados a serem transmitidos;
- *destino*: processo receptor.

Geralmente, quando ocorre uma operação de transferência de mensagens, os dados que compõe a mensagem são copiados em um *buffer*, de onde serão transmitidos quando possível. Após a cópia para o *buffer*, as posições de memória (variáveis) onde os dados transmitidos estavam armazenados podem ser reutilizadas sem problemas.

A rotina *send( )* apresenta diversas variações de semântica, discutidas na literatura [COU94] [QUI87] [QUE94] e com interpretações distintas entre os

vários autores. Este trabalho segue as definições elaboradas pelo Fórum MPI [SNI96], que são:

- ***Send( )* síncrono**: A rotina só se completa com o recebimento de uma confirmação de que a mensagem foi recebida. Caso contrário, têm-se uma rotina ***Send( )*** assíncrona;
- ***Send( )* bloqueante**: O processo transmissor é bloqueado até que se copie a mensagem para o *buffer* de transmissão;
- ***Send( )* não bloqueante**: O processo transmissor não espera a cópia da mensagem para o *buffer*. Então, antes de se reutilizar a variável que está sendo enviada, deve-se testar se a mensagem já foi copiada para o *buffer*.

Rotina *Receive( )*: De maneira genérica, esta rotina é definida por:

receive(msg_id, msg, tamanho, fonte)

onde:

- *msg_id*: identificador da mensagem;
- *msg*: posição de memória onde os dados recebidos são armazenados;
- *tamanho*: tamanho máximo dos dados a serem recebidos;
- *fonte*: processo transmissor.

Novamente, segundo as definições seguidas pelo Fórum MPI [SNI96], pode-se ter:

- ***Receive( )* bloqueante:** A partir do momento que esta rotina é ativada, só se completará quando a mensagem pretendida for recebida;
- ***Receive( )* não bloqueante**: Similar a rotina *Send( )* não bloqueante, a mensagem é recebida sem o bloqueio da rotina, de maneira que deve-se testar o término da operação antes de sua utilização.

4.3.3. Rotinas coletivas

As rotinas coletivas caracterizam-se pela participação de dois ou mais processos em cada operação de comunicação. Estas funções podem ser construídas a partir de rotinas de comunicação ponto-a-ponto.

Existem três operações coletivas mais comuns, que são: ***broadcast( )***, rotinas aritméticas globais e rotinas de sincronização.

Rotina *Broadcast( )*: Uma mensagem é distribuída, a partir de um processo raiz, para todos os processos participando de uma determinada aplicação. A sintaxe genérica deste comando é:

broadcast (msg_id, msg, tamanho, raiz)

onde:

- *msg_id*: identificador da mensagem;
- *msg*: dados a serem transmitidos;
- *tamanho*: tamanho dos dados a serem transmitidos;
- *raiz*: processo raiz.

Rotinas Aritméticas Globais: Também conhecidas como operações *reduce()*, relacionam rotinas que computam um determinado valor baseado em valores enviados por um grupo de processos. Geralmente, um processo raiz é responsável por receber os operandos de uma determinada operação de cada um dos processos participantes e calcular o resultado desejado. Feito isso, o resultado final é feito disponível para todos os processos participantes. Operações que podem ser feitas globalmente são, por exemplo: soma e produto global, máximo e mínimo, entre outras. De maneira genérica, a sintaxe desta rotina é:

reduce(x, n_elementos, trabalho)

onde

- *x*: operando;
- *n_elementos*: número de operandos;
- *trabalho*: tipo de operação (por exemplo, soma) a ser executada.

Rotinas de Sincronização: Rotinas responsáveis por uma sincronização global de todos os processos. Uma rotina comumente utilizada para esse fim é a rotina *barrier( )*. Todos os processos que devem se sincronizar, executam a rotina *barrier( )*, de maneira que eles se bloqueiam até que o último processo do grupo a tenha executado. Então, todos os processos estão sincronizados e são desbloqueados.

4.3.4. Rotina *Probe( )*

Esta rotina é responsável por verificar se uma determinada mensagem já foi recebida. Um modelo genérico para esta rotina é:

integer probe(msg_id)

onde *msg_id* trata-se do identificador da mensagem procurada. Esta rotina retorna um valor inteiro, que pode ser: 0 em caso de mensagem não recebida e 1 caso contrário.

4.3.5. Rotinas Informativas

Rotinas responsáveis pelo retorno de informações a respeito da última mensagem recebida ou *"probed"*. Estas rotinas possibilitam que o processo receptor analise as mensagens sem que se necessite recebê-las. Exemplos são:

integer info_pid( )	Retorna o identificador do processo transmissor
integer info_tam( )	Retorna o tamanho da mensagem
integer info_id( )	Retorna o identificador da mensagem

Nesta seção foi apresentado um conjunto básico de rotinas genéricas para uma biblioteca de troca de mensagens. O MPI, que é apresentado na próxima seção, apresenta um modelo, que pretende ser um padrão, para a estruturação dessas rotinas.

4.4. MPI - *Message Passing Interface*

O MPI é uma tentativa de padronização, independente de plataforma paralela, para ambientes de programação via troca de mensagens. Como apresentado na seção 4.2., o MPI surgiu da necessidade de se resolver alguns problemas relacionados às plataformas de portabilidade, como por exemplo [MCB94]:

- o grande número de plataformas existentes ocasiona restrições em relação à real portabilidade de programas;
- o mau aproveitamento de características de algumas arquiteturas paralelas.

David Walker [WAL95] relaciona, de maneira mais geral, uma série de motivos que explicam a necessidade de um padrão para este tipo de sistema:

- **Portabilidade e facilidade de uso**: À medida que aumente a utilização do MPI, será possível portar transparentemente aplicações entre um grande número de plataformas paralelas;
- **Fornecer uma especificação precisa**: Fabricantes de *hardware* podem implementar eficientemente em suas máquinas um conjunto bem definido de rotinas;
- **Crescimento da indústria de *software* paralelo**: A existência de um padrão torna a criação de *software* paralelo (ferramentas, bibliotecas, aplicativos, etc.) por empresas uma opção comercialmente viável;
- **Incentivar uma maior utilização de arquiteturas paralelas**: O crescimento da indústria de *software* paralelo implica em maior difusão do uso de computadores paralelos.

O processo de padronização envolveu cerca de 80 pessoas, provenientes de 40 organizações, principalmente americanas e européias. A maioria dos principais fabricantes de computadores paralelos participaram do desenvolvimento do MPI, além de universidades e laboratórios ligados ao governo. Os principais objetivos que guiaram o processo de padronização foram [DON95] [DOS96]:

- Lançar uma versão inicial em um tempo predefinido, de maneira que não se perdesse o controle sobre o padrão;
- Prover portabilidade real;
- Prover implementação eficiente em plataformas paralelas distintas;
- Possuir uma aparência compatível com a atualidade, visto que uma interface totalmente nova dificultaria a sua aceitação.

Procurou-se aproveitar as melhores características de cada uma das plataformas de portabilidade, de maneira que todo o esforço e estudo que já havia

sido despendido neste tipo de plataforma fosse aproveitado, levando-se também em consideração a necessidade de eficiência em todas as plataformas paralelas. O problema da eficiência em qualquer plataforma gera complicações uma vez que determinadas operações são feitas com diferente eficiência em diferentes plataformas paralelas e por diferentes protocolos. Por isso, algumas rotinas no MPI são implementadas de várias formas (de maneira que se aproveite as qualidades intrínsecas de cada plataforma paralela), o que gera uma certa complexidade. Porém tal problema é inevitável quando almeja-se um padrão que englobe diferentes tipos de arquiteturas computacionais. Um exemplo desse problema é discutido na seção 4.4.3, quando são apresentadas as operações coletivas [DON95], e também no capítulo 6, onde são discutidas as formas de comunicação ponto-a-ponto.

4.4.1. A História do MPI

O processo de desenvolvimento do MPI iniciou em abril de 1992, sendo que, em novembro do mesmo ano, uma primeira versão foi apresentada (MPI1). Esta era uma versão incompleta, e tinha como principal função a de "manter a bola rolando", isto é, mostrar que o esforço era válido [DOS96] [WAL94].

A partir do MPI1, decidiu-se colocar o processo de padronização em bases mais formais, de maneira mais organizada. Foram então criados comitês, um para cada componente do MPI, e também foram criadas *mailing-lists* para cada um desses comitês. O grupo de desenvolvimento passaria a se encontrar de 6 em 6 semanas, e o conjunto dessas reuniões e *mailing-lists* se definiu como o **Fórum MPI**.

Em novembro de 1993 foi apresentada a especificação do padrão MPI versão 1.0, sendo esta publicada em maio de 1994. Em junho de 1995, foi publicada a versão 1.1, apresentando correções de erros e melhor esclarecimento de determinadas propriedades do MPI.

4.4.2. Especificação do MPI

Levando-se em consideração o limite de tempo imposto à formulação do padrão, definiu-se um conjunto básico de rotinas relacionadas à comunicação ponto-a-ponto e coletiva, deixando para uma próxima versão uma maior abrangência de operações.

O MPI define um conjunto de 129 rotinas, que oferecem os seguintes serviços [DON95] [MAC96] [SNI96]:

- Comunicação ponto-a-ponto;
- Comunicação coletiva;
- Suporte para grupos de processos;
- Suporte para contextos de comunicação;
- Suporte para topologia de processos.

Analisando cada um dos tópicos anteriores, têm-se [DON95] [MAC96] [PAC95] [SNI96] [WAL95]:

Comunicação Ponto-a-Ponto e Coletiva: O MPI implementa todos os tipos de comunicação citados nas seções 4.3.2. e 4.3.3., com algumas variantes. Estes dois tópicos serão abordados na próxima seção, e em grandes detalhes no capítulo 6.

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

Suporte para Contextos de Comunicação: Contextos podem ser definidos como escopos que relacionam um determinado grupo de processos. Estes tipos de instâncias são implementadas com o intuito de garantir que não hajam mensagens que sejam recebidas ambiguamente por grupos de processos não relacionados. Então, um grupo de processos ligados por um contexto não consegue comunicar-se com um grupo que esteja definido em outro contexto. Este tipo de estrutura não é visível nem controlável pelo usuário, e o seu gerenciamento fica a cargo do sistema.

Para a criação de contextos, o MPI se utiliza do conceito de *communicator*. Um *communicator* é um objeto manuseado pelo programador e relaciona um grupo (ou grupos) de processos com um determinado contexto. Se existem, por exemplo, aplicações paralelas distintas executando em um mesmo ambiente, para cada uma delas será criado um *communicator*. Isso criará contextos distintos que

relacionarão os grupos de processos de cada aplicação e evitará que estes interfiram entre si.

Suporte para Topologias: O MPI fornece primitivas que permitem ao programador definir a estrutura topológica com a qual os processos de um determinado grupo se relacionarão. Como exemplo de uma topologia, pode citar-se uma malha, onde cada ponto de intersecção na malha corresponde a um processo.

4.4.3. Comunicação no MPI

As rotinas de comunicação (ponto-a-ponto e coletiva) são o núcleo básico do MPI [DON95] [MAC96] [PAC95] [SNI96] [WAL95]. Antes de apresentar estas rotinas, é importante entender como o MPI organiza uma mensagem.

Uma mensagem no MPI é definida como um vetor de elementos de um determinado tipo. Ao se enviar uma mensagem, deve-se indicar o endereço do primeiro elemento deste vetor e o número de elementos que o formam. Estes elementos devem ser do mesmo tipo.

Deve-se indicar na mensagem o tipo dos elementos sendo enviados. Essa característica é essencial quando a comunicação é realizada entre sistemas heterogêneos, tornando necessária a conversão dos dados.

O MPI permite que se crie tipos definidos pelo usuário, tornando possível enviar mensagens compostas por elementos de tipos distintos, como por exemplo, estruturas (structs) [MAC96].

Para as operações de comunicação, o MPI define:

Rotina *Send( )*: A maneira eficiente de implementação de uma rotina *send( )*, depende de certa maneira do protocolo e da plataforma paralela sobre o qual o MPI está executando. Para garantir a eficiência em qualquer plataforma, o MPI define vários modos de comunicação, que definem diferentes semânticas para a rotina. Os modos disponíveis, que apresentam versões bloqueantes e não bloqueantes, são:

- **Síncrono**: o transmissor, ao enviar uma mensagem, espera uma confirmação de recepção da mensagem;
- ***bufferizado***: mensagens são transmitidas via *buffers* explicitamente criados pelo programador;

- **padrão**: o modo mais eficiente de comunicação. Pode ser síncrono ou *bufferizado*;
- ***ready***: necessita que o *receive( )* correspondente deve ter sido obrigatoriamente iniciado.

A definição de bloqueio de rotinas, utilizada pelo Fórum MPI, caracteriza-se pela sobreposição ou não de operações de comunicação em relação a execução do programa.

Outras variantes para rotinas de comunicação ponto-a-ponto também são definidas, como rotinas de comunicação em dois sentidos e requisições persistentes. Como este é um aspecto intimamente relacionado a este trabalho, modos de comunicação, bloqueio de rotinas, comunicação em dois sentidos e requisições persistentes são discutidos em detalhes no capítulo 6.

Rotina R*eceive( )*: Uma mensagem é selecionada para recebimento pelo *rank* do processo que a enviou e pelo seu identificador (*tag*), dentro de um determinado contexto. Esses dois valores podem ser *wild-cards*, isto é, pode-se utilizar rotinas que recebam de qualquer processo mensagens de qualquer *tag*. A rotina *receive( )* não possui desdobramento em modos, podendo ser bloqueante ou não.

Rotinas Coletivas: O MPI define todos os tipos de rotinas coletivas definidas na seção 4.4.4., que são rotinas *broadcast( )*, aritméticas globais e de sincronização. Também, outras variantes de rotinas coletivas foram definidas: rotinas *gather( )*, *scather( )*, e uma combinação das duas anteriores. Uma representação genérica destas funções é apresentada na figura 4.2. [DON95] [MAC96] [SNI96].

Esta grande variedade de funções coletivas se explicam pelo fato de se garantir operações coletivas eficientes em todas as plataformas paralelas. Todas as operações coletivas no MPI são bloqueantes e executam no modo padrão.

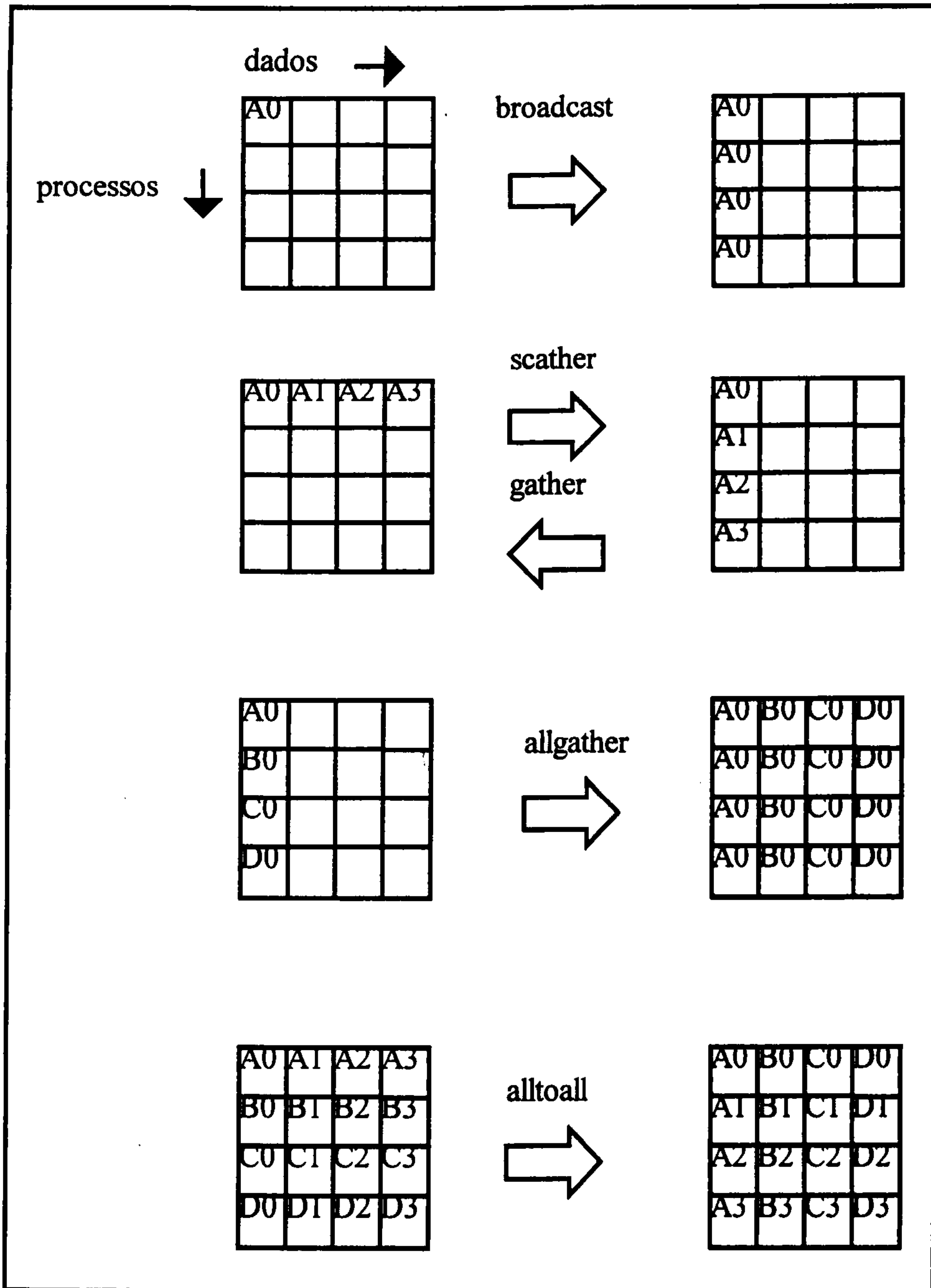

Figura 4.2 - Variações Sobre a Rotina B*roadcast( )*.

4.4.4. Programa Exemplo

Para que se tenha uma idéia da programação MPI, é apresentado nesta seção um programa exemplo escrito em C (o MPI possui *bindings* para C e Fortran), que implementa a transferência de uma mensagem entre dois processos.

Este programa executa de maneira SPMD em dois processadores. São gerados dois processos identificados por *ranks* 0 e 1, sendo que o processo de *rank* = 0 envia uma mensagem para o processo de *rank* = 1, que recebe esta mensagem.

```
#include <mpi.h>                /* Biblioteca MPI */

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

{

char msg[20];                   /* Mensagem a ser enviada */
int myrank;                     /* Rank de um processo */
int tag = 99;                   /* Identificador da mensagem (tag) */
MPI_Status status;              /* Variável status, utilizada pela rotina receive( ) */

MPI_Init(&argc, &argv);         /* Inicia o MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);        /* Define o rank do processo */

/* O processo com rank 0 envia a mensagem, o de rank 1 a recebe */
if (myrank == 0) {
        strcpy( msg, "Hello world");
        MPI_Send(msg, strlen(msg) + 1, MPI_CHAR, 1, tag,MPI_COMM_WORLD);
        }

        else {
                MPI_Recv( msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
                printf(" %s \n", msg);
                }
MPI_Finalize( );            /* Finaliza o MPI */
}
```

Este programa utiliza a rotina *send( )* padrão (*MPI_Send)*, bloqueante. Este é um exemplo extremamente simples; porém, geralmente só em casos isolados necessita-se da utilização das rotinas mais avançadas do MPI. Um dos objetivos do MPI é ser poderoso sem abrir mão da simplicidade, o que nem sempre é possível, levando-se em conta aplicações mais complexas.

4.4.5. Comentários Finais Sobre o MPI

Uma vez que um dos objetivos definidos na criação do MPI foi o lançamento de uma primeira versão em um tempo curto, optou-se por implementar uma versão inicial resumida. Essa versão ainda não define vários tópicos, como

gerenciamento de processos (por exemplo, criação, término, e mapeamento), E/S paralelos, entre outros. Porém, este padrão básico está sendo estendido, no projeto que se nomeia MPI2 [SKJ94]. As reuniões para o aumento do padrão já se iniciaram, e já existe um projeto, em estado avançado, para implementação de E/S paralelos, denominado MPI-IO.

Várias implementações do MPI, tanto comerciais como de domínio público, já estão disponíveis, e várias outras devem ser lançadas. Empresas que já lançaram versões comerciais são: *IBM*, *Meiko*, *Intel*, *Cray Research*, *Ncube*, *HP*, *Silicon Graphics*, *NEC* e *Telmat* [DON95] [LAM96]. O *Ohio SuperComputer Center* relaciona nove implementações de domínio público onde se destacam os sistemas MPICH, CHIMP e LAM [LAM96]. As implementações utilizadas neste trabalho serão discutidas com detalhes no próximo capítulo.

De maneira geral, os primeiros resultados da utilização do MPI foram satisfatórios, apesar de existirem alguns pontos ainda passíveis de uma possível melhora em versões posteriores [SKJ95].

4.5. Considerações Finais

Nas discussões entre as vantagens e desvantagens entre memória centralizada e memória distribuída, geralmente é citado que a programação em memória centralizada é mais fácil [ALM94] [TAN95]. Essa afirmação é totalmente verdadeira quando se programa em baixo nível. Para programação em alto nível, várias implementações tem sido propostas para fornecer mecanismos construídos sobre o paradigma de troca de mensagens, com o intuito de facilitar o trabalho do programador. Alguns exemplos destas implementações são: RPC [TAN95] [COU94] [STE90] [SOU95] [WIL87] e as plataformas de portabilidade [MCB94] [KIT95].

O paradigma de troca de mensagens, inserido neste contexto, oferece uma maneira simples, ampliável e principalmente, adaptável em qualquer plataforma, incluindo arquiteturas paralelas de memória centralizada. Além disso, alguns autores destacam a importância do modelo computacional MIMD com memória distribuída, indicando-o como tendência dentro da computação paralela [BLE94] [ZAL91]. Levando-se em consideração tais colocações, pode-se concluir que o paradigma de programação via troca de mensagens relaciona-se aos avanços computacionais do final do século, que prevê um grande desenvolvimento da computação paralela.

As plataformas de portabilidade têm-se mostrado abordagens eficientes para a implementação de programas paralelos utilizando troca de mensagens, sendo a

possibilidade de transportar programas diretamente entre ambientes heterogêneos, incluindo arquiteturas paralelas e sistemas distribuídos, um grande atrativo.

Dentre as plataformas de portabilidade destaca-se o PVM, por sua generalidade e aplicabilidade em uma grande gama de aplicações. Assim, é razoável considerá-lo como a plataforma mais utilizada entre todas [MCB94], sendo inclusive considerado por muitos autores como um padrão de fato [SUN94]. O PVM é apresentado com maiores detalhes no próximo capítulo.

Existe ainda, porém, a necessidade de uma plataforma de portabilidade que possibilite eficiência e segurança em qualquer aplicação executando sobre qualquer plataforma paralela e que seja um padrão "oficial", aceito largamente entre a comunidade computacional como tal. E que também ofereça determinadas ferramentas deixadas de fora pelo PVM em função da sua objetivada simplicidade. Dentro desta perspectiva, se insere a importância do MPI.

Sendo aceito como padrão, o MPI pode gerar um estímulo para o desenvolvimento da indústria de *software* paralelo. Porém, a história recente da computação no mundo ensina que processos de padronização são perigosos e muitas vezes ineficazes.

Além disso, o fato do MPI ter sido projetado para fornecer eficiência, não implica que necessariamente uma implementação o será. O Fórum MPI define apenas indicações de possíveis maneiras eficientes de implementação de determinadas partes do padrão. Por isso, a escolha de utilização ou não do MPI deve ser embasada também na análise da implementação a ser utilizada. No próximo capítulo, as implementações MPI utilizadas neste trabalho são apresentadas em detalhes.

Capítulo 5

Implementações de Plataformas de Portabilidade

Nesse capítulo são apresentadas as plataformas de portabilidade utilizadas nesse trabalho (MPICH, LAM, UNIFY e PVM), além de justificativas para a escolha dessas plataformas e do ambiente computacional escolhido para o seu desenvolvimento. Além disso, as diferentes implementações do MPI são comparadas e também é apresentada uma comparação entre o MPI e o PVM.

5.1. Introdução

Dentre as plataformas de portabilidade, duas devem ser destacadas por sua importância e relevância no cenário computacional atual. São elas o PVM [BEG94] [PVM96], por ser a mais difundida e utilizada [MCB94] e o MPI, por ser uma tentativa de padronização levada a cabo por diversas organizações acadêmicas e comerciais americanas e européias [DON95] [SNI96].

Estas duas plataformas de portabilidade apresentam em comum a definição de bibliotecas de rotinas para comunicação entre processos. Por outro lado, enquanto o PVM é uma implementação completa, o MPI é apenas uma especificação sintática e semântica de uma biblioteca de troca de mensagens. Dessa maneira, conclui-se que uma análise mais completa do MPI deve ser feita levando em consideração a implementação a ser utilizada. Segundo o censo mantido pelo *Ohio SuperComputer Center*, atualmente existem pelo menos quinze implementações do MPI, comerciais ou de domínio público [LAM96].

O MPI permite a sua utilização em uma grande gama de plataformas paralelas, e um estudo completo de sua utilização em diferentes ambientes está fora do escopo deste trabalho. Nosso objetivo é explorar a utilização do MPI em sistemas distribuídos (nesse caso, uma rede de computadores executando o sistema operacional LINUX [WEL95]), que é uma plataforma eficiente, flexível e de custo relativamente baixo para a utilização da computação paralela. A escolha do ambiente computacional LINUX para o desenvolvimento deste trabalho é descrita em detalhes na seção 5.2.

Uma maneira de avaliar o MPI de maneira mais geral, é estender este estudo para várias implementações do MPI, possibilitando uma análise mais abrangente e um estudo comparativo entre as várias implementações no aspecto desempenho da comunicação entre processos. Por isso, foram utilizadas três implementações de domínio público do MPI, que são: MPICH [GRO96a] [MPI96], LAM [BUR95] [LAM96] e UNIFY [CHE94], que foram escolhidas por possuírem suporte para o LINUX (ver seção 5.2). Nas seções 5.3, 5.4 e 5.6 são discutidas as estruturas de cada uma dessas três implementações.

Compreender a estrutura de cada implementação, além de proporcionar maior leque de informações para estudar as diferenças de desempenho entre elas, permite também uma comparação relacionando alguns outros aspectos que são importantes quando da escolha de uma implementação MPI. Dillon [DIL95] cita três características essenciais de comparação entre plataformas de portabilidade, que são: portabilidade, facilidade de utilização e desempenho.

Baseado nessas premissas, os tópicos de comparação discutidos neste capítulo são: portabilidade, facilidade de instalação, compilação e execução de programas, documentação e depuração de programas paralelos (depuração é definida, neste trabalho, como a habilidade de descobrir erros relacionados a sincronização entre processos e mensagens). Estes tópicos são discutidos para todas as implementações MPI (seções 5.3.3, 5.4.3 e 5.6.2) e relacionados comparativamente na seção 5.7. É importante ressaltar que as características dependentes de ambiente computacional ,como por exemplo, instalação, foram observadas apenas no sistema operacional LINUX, e não devem ser estendidas as todas as plataformas paralelas para as quais os sistemas citados são portáteis.

Este trabalho apresenta ainda resultados comparativos entre o PVM e o MPI (ver seção 5.2), tanto a nível de estrutura geral (seção 5.8) como a nível de desempenho. Assim, o PVM é discutido na seção 5.5, antes da apresentação do UNIFY (seção 5.6), que executa sobre ele.

5.2. O Ambiente Computacional

Para a execução deste trabalho, haviam duas alternativas possíveis de escolha para o ambiente computacionai: uma rede de *Suns SparcStations*, sob o controle do sistema operacional *SunOS*, e uma rede de computadores pessoais (PC's), executando um sistema operacional *UNIX-like*.

A primeira opção tinha como vantagem o fato de todas as principais implementações de domínio público possuírem suporte para o *SunOS* e já terem sido muito testadas nesse ambiente. Porém, a rede *Sun* está localizada em um

laboratório de uso de todo o instituto, sendo sua configuração controlada por técnicos contratados pela universidade. Encontrava-se então em período de reformulação, não sendo possível por isso prever possíveis mudanças de configuração durante o decorrer do trabalho. Além disso, todos alunos, professores e funcionários a utilizam, sendo difícil de lidar com o problema de cargas computacionais variáveis.

Para a execução dos *benchmarks* propostos nesse trabalho era necessário a utilização de um sistema computacional que se pudesse controlar a configuração e utilização. Nesse caso, a segunda opção, rede de PC's, está localizada em um laboratório do próprio grupo de pesquisa onde este trabalho está sendo desenvolvido, por isso, estando o controle e configuração a cargo do próprio grupo.

Além disso, a utilização de uma rede de PC's, pela sua grande difusão na comunidade computacional, poderia fornecer resultados interessantes, e não explorados na literatura pesquisada. Por isso, a escolha do ambiente computacional estendeu os objetivos deste trabalho, isto é, permitiu fornecer informações sobre a utilização do MPI em redes de computadores pessoais. Uma desvantagem desta escolha foi a necessidade de partir do zero, isto é, instalar e configurar todo o sistema.

Uma vez definido o ambiente, era necessário definir qual sistema operacional *UNIX-like* utilizar. Dentre as opções conhecidas, podia-se escolher entre: os sistemas operacionais baseados no BSD (FreeBSD, NetBSD, 386BSD) e o LINUX, todos de domínio público.

O LINUX [WEL95] foi escolhido devido as seguintes razões: popularidade (é um sistema operacional *UNIX-like* largamente utilizado na comunidade acadêmica), versatilidade e facilidade de uso. Além disso, o LINUX possui maior número de implementações que executam sobre ele.

5.2.1. O Sistema LINUX

O LINUX é um sistema operacional *UNIX-like* para computadores pessoais distribuído gratuitamente sobre os termos da Licença Pública Geral GNU. Sua implementação iniciou-se com um projeto desenvolvido por Linus Tovards, e tem-se desenvolvido continuamente [WEL95].

O LINUX se diferencia dos seus correlatos (família BSD) por possuir uma filosofia de projeto totalmente aberta, isto é, vários "gurus" UNIX de muitas localidades mundiais trabalham e contribuem para o seu crescimento e aperfeiçoamento. A nível de *kernel* (o núcleo do sistema operacional), também

existe este trabalho conjunto, supervisionado pelo seu criador, Linus Tovards, que organiza e controla a distribuição de cada nova versão do *kernel*.

Muitas contribuições são sugeridas para o desenvolvimento do *kernel* LINUX, e por isso, novas versões são distribuídas separadas por pequenos intervalos de tempo. Os números que representam cada uma das versões oferece ao usuário uma maneira de entender a evolução do sistema.

Tovards divide os *kernels* LINUX em dois tipos: estáveis e de desenvolvimento. Como o próprio nome sugere, *kernels* de desenvolvimento são versões acrescidas de novos módulos e correções de erros, e devem ser utilizados com cuidado. Essa fase de desenvolvimento dura até que uma determinada versão do *kernel* se estabilize (apresente poucos problemas), quando então é lançado uma versão estável.

Cada *kernel* possui um número de versão no formato *1.x.y*, onde: *x* sendo número par indica versão estável, e *y* indica pequenas variações e mudanças ocorridas no *kernel* e varia de 0 em diante, até que mude o valor de *x*. Recentemente, esse sistema de numeração foi modificado pela retirada do primeiro número.

No início deste trabalho, a última versão estável de *kernel* disponível possuía número 1.2.13, e por isso foi escolhida como versão a ser utilizada. Porém, problemas de incompatibilidade entre o sistema e as placas de rede, obrigou a mudança para um *kernel* de desenvolvimento onde este erro estivesse corrigido (versão 1.3.7).

Todas as implementações do MPI foram instaladas na versão 1.3.7. Porém, a implementação LAM apresentou problemas de execução, o que forçou a utilização da versão 1.3.20. Esta, então, é a versão do *kernel* LINUX utilizada neste trabalho. É importante ressaltar que, apesar de ser uma versão de desenvolvimento, na prática apresentou comportamento estável e extremamente satisfatório.

Além do problema com as placas de rede, erros de compatibilidade com um dos discos rígidos exigiu mudanças no código fonte do *kernel* para sua exata execução em um dos computadores componentes do laboratório.

Os erros citados acima foram encontrados mediante pesquisa detalhada em literatura especializada (HOWTO's, FAQ's, entre outros) e contato com a equipe de desenvolvimento do LINUX. Assim, apesar de ser relativamente simples a instalação do LINUX (para o usuário que não tem "preguiça" de ler manuais), na prática, o usuário deve ter cuidado e paciência para a correta instalação e configuração deste sistema operacional.

5.2.2. Plataformas de Portabilidade

A escolha das plataformas de portabilidade a serem consideradas neste trabalho foi efetuada através da pesquisa em material onde se relacionavam as diversas implementações do MPI [DOS96] [LAM96] Foram selecionadas 3 opções possíveis de utilização no LINUX: duas implementações de domínio publico (LAM e MPICH), e uma implementação de dupla interface (UNIFY), que se aproveitava da alta portabilidade do PVM.

O MPICH [GRO96a] [MPI96] e o LAM [BURNS95] [LAM96] foram escolhas naturais, visto que, além de implementarem a versão 1.1 completa do MPI, são implementações bastante utilizadas e de uso real (isto é, não são versões incompletas ou intermediárias). A escolha do UNIFY [CHE94] não é imediata, visto que é uma implementação incompleta e experimental e que aparentemente, não foi levado a cabo pela sua equipe de desenvolvimento. No entanto, é uma implementação que por suas características pode interessar para uma determinado tipo de usuário interessado na migração gradual entre o PVM e o MPI [VAU94]. Apesar de não ter sido testada sobre ambiente LINUX (segundo a documentação disponível), apresentou comportamento estável e executou corretamente nos primeiros testes. As implementações do MPI são discutidas em maiores detalhes nas seguintes seções: MPICH (seção 5.3), LAM (seção 5.4) e UNIFY (seção 5.6).

Um fator de enriquecimento do trabalho foi a inclusão do PVM [BEG94] [PVM96] como fator de comparação (onde possível) em termos de desempenho e interface. O PVM é considerado por alguns autores como padrão de fato e sua importância dentro da computação paralela distribuída é indiscutível [SUN94]. Assim, é muito interessante apresentar alguns resultados comparativos entre o PVM e o MPI.

É importante ressaltar que os resultados obtidos neste trabalho devem ser analisados dentro de um contexto específico. E este contexto é definido pelo uso da computação paralela em sistemas distribuídos, utilizando redes de PC's e o sistema operacional LINUX. Além disso, deve-se considerar as versões das plataformas de portabilidade utilizadas, que são: MPICH 1.0.12, LAM 6.0, UNIFY 0.9.2 e PVM 3.3.10.

5.3. MPICH

O estudo do MPICH apresentado nesta seção foi baseado nas seguintes referências: [GRO96a] [GRO96b] [GRO96c] [MEY94] [MPI96].

A implementação MPICH é o resultado de um trabalho conjunto entre as organizações americanas *Argonne National Laboratory* e *Mississippi State University*. Seu desenvolvimento iniciou-se junto aos trabalhos de definição do padrão MPI, providenciando uma experimentação prática das decisões tomadas pelo Fórum MPI e também uma primeira amostra para os usuários em geral da interface MPI à medida que esta se desenvolvia [GRO96a].

O principal objetivo que guiou o desenvolvimento do MPICH foi fornecer uma implementação portátil para uma grande variedade de sistemas computacionais paralelos que suportassem o modelo de computação via troca de mensagens (arquiteturas MIMD com memória distribuída e compartilhada e redes de computadores - plataformas paralelas), procurando afetar o mínimo possível a eficiência. Por ser altamente portátil, foi batizado de MPI *Chameleon* ou MPICH.

O MPICH implementa todas as rotinas do padrão MPI versão 1.1 e foi desenvolvido a partir da combinação de três implementações de plataformas de portabilidade já existentes e estáveis, o que permitiu que fosse implementado rapidamente, embora grande parte do código fonte necessitasse ser reestruturado. Estas plataformas são: p4 [BUT94], *Chameleon* [SMI93] e *Zipcode* [SMI94]. Durante os anos seguintes ao desenvolvimento de sua primeira versão (dezembro de 1992 - MPI1), o MPICH vem sendo estendido e aperfeiçoado com o intuito de fornecer melhor portabilidade e desempenho, e entre as suas principais modificações, podem ser citadas: o desenvolvimento do ADI (*Abstract Device Interface*), discutido na seção 5.3.1, e a elaboração do MPE (*Multi-Processing Environment*), apresentado na seção 5.3.4.

5.3.1. Arquitetura do MPICH

A fim de facilitar o trabalho de implementação e tornar mais simples a operação de transporte do código fonte entre diversas plataformas paralelas (é importante lembrar que um dos objetivos centrais do MPICH é ser uma implementação altamente portátil), a estrutura do MPICH se divide em dois níveis. No primeiro nível concentra-se todo o código fonte reaproveitável (independente de *hardware*), que pode ser transportado diretamente, e onde todas as funções do MPI estão implementadas. No segundo nível, situa-se todo o código dependente de plataforma, e estes dois níveis comunicam-se através de uma camada denominada ADI (*Abstract Device Interface* - Interface de Dispositivo Abstrata) [GRO96a] [MEY94]. Desta maneira, todas as funções do MPI são escritas utilizando-se rotinas e macros oferecidos pela ADI. É importante ressaltar que a ADI é projetada para permitir que qualquer biblioteca de passagens de mensagens possa ser implementada sobre ela [GRO96] [MEY94].

Denomina-se **dispositivo** (*device*), à organização de *software* que se encontra no segundo nível de estruturação do MPICH, e através de uma especificação precisa para a ADI, possibilita-se que os fabricantes de *hardware*, por exemplo, forneçam implementações eficientes de dispositivos próprios. Pode-se também implementá-los através de rotinas de bibliotecas de passagem de mensagens nativas (da própria plataforma de *hardware*) ou plataformas de passagens de mensagens já existentes, como o p4 [BUT94] ou PVM [BEG94].

Determinadas plataformas paralelas podem possuir mais de um dispositivo possível. Uma rede de *workstations Sun*, por exemplo, permite que se execute o MPICH sobre dispositivos implementados a partir das plataformas de portabilidade p4 ou PVM.

Em virtude da facilidade de transporte do código do MPICH para novas plataformas de *hardware*, esta implementação vem se difundindo de maneira significativa e rápida, e já executa com eficiência em uma grande gama de máquinas paralelas e redes de computadores.

A figura 5.1 apresenta um modelo resumido da estrutura do MPICH (versão 1.0.12) quando instalado sobre uma rede LINUX, esta que foi a configuração utilizada neste trabalho.

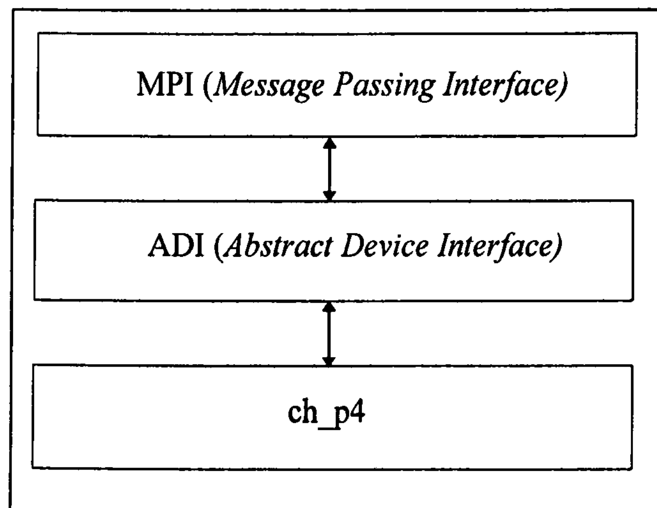

Figura 5.1 - Estrutura do MPICH Sobre o LINUX.

No caso da figura 5.1, o dispositivo utilizado é o ch_p4 (esta nomenclatura representa uma implementação mista baseada nas rotinas das plataformas de portabilidade *Chameleon* [SMI93] e p4 [BUT94]).

5.3.2. MPICH e LINUX

O dispositivo ch_p4, apresentado na seção anterior, pode ser utilizado em qualquer rede de computadores executando um sistema operacional *UNIX-like*, e para próximas versões do MPICH está sendo planejada uma implementação nativa e independente sobre o protocolo TCP/IP (protocolo de comunicação entre computadores UNIX [STE90]).

A estruturação básica de comunicação do MPICH utilizando o dispositivo ch_p4, possui organização semelhante à da implementação p4 [BUT94]. O p4 subdivide o conjunto de processos de uma aplicação em *clusters*, que são grupos de processos com códigos fontes idênticos executando no mesmo computador. Em caso de comunicação dentro do *cluster*, utiliza-se memória compartilhada (monitores implementam controle de acesso e seqüência[1]). Para comunicação entre processos em *clusters* diferentes, transmitem-se mensagens via *sockets* (implementação padrão UNIX de comunicação entre processos [STE90]). A figura 5.2 apresenta uma esquematização da estrutura do MPICH sobre o dispositivo ch_p4 [MEY94].

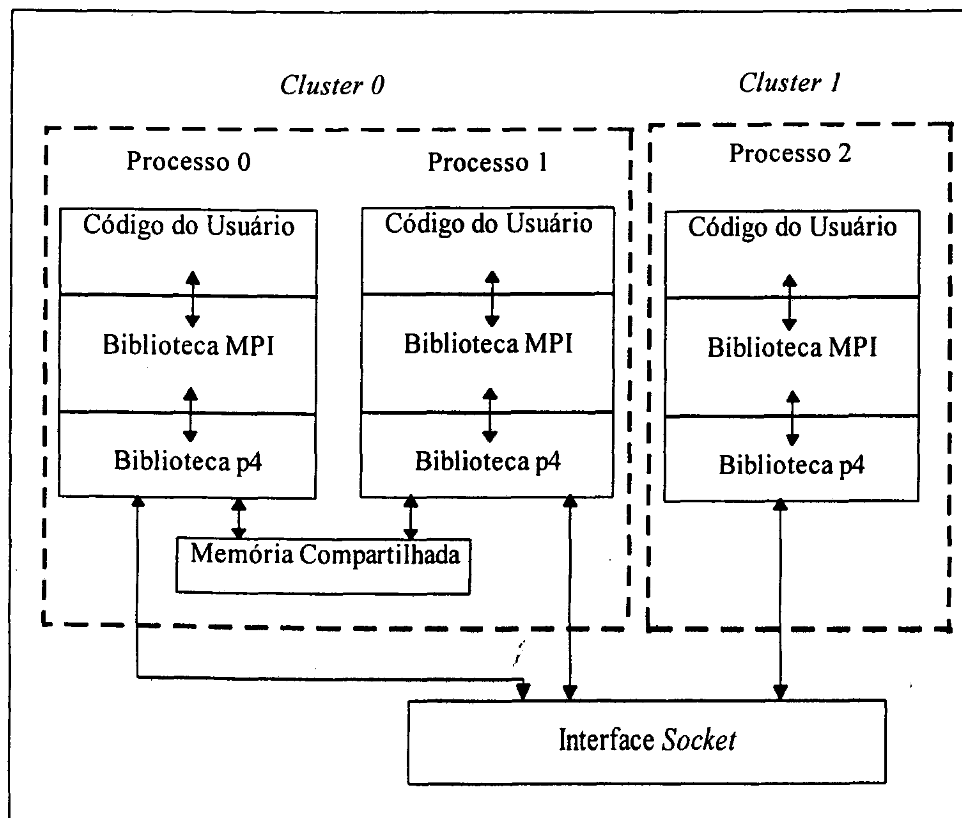

Figura 5.2 - Estrutura do MPICH Sobre o Dispositivo ch_p4.

[1] Ver Capítulo 2, seção 2.4.3 - **Comunicação e Sincronismo**.

5.3.3. Características Gerais do MPICH

Nesta seção são discutidos alguns aspectos importantes para a escolha de uma implementação MPI e que servem como base para a discussão das diferenças entre as implementações consideradas neste trabalho (seção 5.7).

Portabilidade: A versão 1.0.12 do MPICH pode ser configurada para executar em diversas plataformas paralelas. Entre elas: arquiteturas paralelas de memória distribuída e centralizada, redes de *workstations*, redes de computadores pessoais e ambientes heterogêneos (apesar de que, o suporte a heterogeneidade ainda apresenta problemas [GRO96a]).

Instalação: A configuração do sistema possui diversas opções, algumas direcionadas a usuários experimentados. Porém, o procedimento mínimo, aconselhado no manual do usuário [GRO96c], é simples e bem documentado. No LINUX, o MPICH é instalado sem problemas nas duas versões de *kernel* testadas (1.3.7 e 1.3.20).

Compilação e Execução de Programas: O MPICH oferece maneiras simplificadas e bem documentadas de compilar programas (excetuando-se quando há necessidade da utilização de opções mais avançadas). Porém, um dos *scripts* de compilação de programas, chamado ***mpicc*** e que tem como vantagem ser simples e compacto, não funciona no LINUX. Erro a ser corrigido na próxima versão, segundo a equipe de desenvolvimento do MPICH.

Documentação: Ponto favorável do MPICH, visto que esta é extensa e bem estruturada, incluindo manuais do usuário e de instalação, e também páginas de ajuda para utilização com o comando UNIX *man*.

Depuração de Programas: O MPICH fornece ferramentas para a visualização do comportamento de programas paralelos em ambiente gráfico (ver seção 5.3.4), o que facilita o trabalho de depuração.

5.3.4. Ferramentas Adicionais

O MPICH possui, além das rotinas básicas do MPI, uma biblioteca cujo objetivo é auxiliar o desenvolvimento e a depuração de programas paralelos, chamada MPE (*Multi-Processing Environment*) [KAR94].

Exemplos de rotinas do MPE são: rotinas gráficas básicas (como desenhar pontos e retas) e funções de *logging* (geram arquivos log, que ao serem interpretados, fornecem uma visualização gráfica da execução do programa paralelo). Os dois exemplos citados executam apenas em ambiente gráfico.

5.4. LAM

O estudo do LAM apresentado nesta seção foi baseado nas seguintes referências: [BUR89] [BURN90] [BUR94] [BUR95] [LAM96].

LAM significa *Local Area Multicomputer*, e seu nome se explica pela sua característica principal que é simular uma arquitetura paralela de memória distribuída (multicomputador) sobre uma rede local de computadores (LAN - *Local Area Network*). Cada um dos computadores que formam parte dessa máquina paralela virtual executa um *daemon* (processo que executa intermitentemente, em *background*, em um sistema UNIX [STE90]), que fornece um rígido controle sobre o funcionamento da máquina virtual e sobre a depuração de programas executados sobre ela (seção 5.4.2).

Burns afirma que "LAM possui uma longa história, na qual o episódio mais recente é o MPI" [BUR95]. O LAM originou-se do sistema *Trollius* [BUR90], desenvolvido por volta de 1985, e que tinha como objetivo ser um sistema operacional padronizado para arquiteturas paralelas que suportassem o paradigma de troca de mensagens. O LAM foi implementado através da extensão e modificação do código fonte do *Trollius* para que executasse sobre redes de computadores. Tanto o *Trollius* quanto o LAM foram desenvolvidos pelo *Ohio SuperComputer Center*.

O LAM apresenta uma estrutura de *software* modular onde, em um nível mais elevado, podem ser implementadas bibliotecas de troca de mensagens. A interface padrão do LAM é o MPI versão 1.1. A estrutura de *software* do LAM é discutido na próxima seção.

5.4.1. Arquitetura do LAM

Como foi exposto, o LAM caracteriza-se por formar uma máquina paralela virtual através da utilização de *daemons* locais. Cada *daemon* é composto de diversos processos "virtuais", tendo como ponto central de controle um *nano-kernel* [BUR95] [BUR94].

Esta organização em processos chamados "virtuais" explica-se pela necessidade de fornecer uma estrutura modular para o *daemon*. Cada processo é responsável por um serviço que pode ser ativado ou desativado de acordo com as necessidades do usuário. Exemplos de serviços oferecidos pelos processos dos *daemons* são: acesso a arquivos remotos e execução de processos remotos. Esta organização em módulos possibilita a extensão das capacidades do *daemon* pela inclusão de novos serviços, e facilita a depuração de problemas isolados, visto que, cada componente pode ser estudado separadamente. É importante ressaltar que esta organização é transparente para o usuário, que enxerga apenas um *daemon* convencional.

A nível de projeto de *software*, a estrutura completa do LAM é rigidamente dividida em camadas sem dependências entre si, como é mostrado na figura 5.3 [BUR95]. Na primeira camada situa-se o *nanokernel*, com controle apenas local sobre cada nó da rede, e responsável pela comunicação local. A segunda camada é o subsistema de comunicação (ou camada de rede), que oferece completa funcionalidade para comunicação remota entre processos. Utilizando as rotinas definidas pelo sub-sistema de comunicação, são implementadas bibliotecas de troca de mensagens e processos cliente/servidor necessários ao funcionamento da máquina paralela virtual. Na camada 3, encontram-se os aplicativos dos usuários. É importante não confundir a estrutura do *daemon* (aspectos de implementação do LAM) com a estrutura de *software* LAM (organização do projeto de *software* LAM).

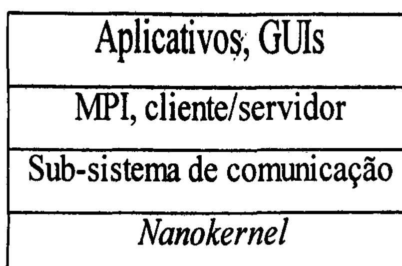

Figura 5.3 - Estrutura de *Software* do LAM.

5.4.2. *Daemons* e Depuração de Programas

A característica mais importante do LAM, segundo seus criadores, é o controle total que é oferecido sobre a máquina paralela virtual [BUR95]. A estrutura de *daemons* permite que programas dos usuários possam ser executados e encerrados em qualquer componente da máquina virtual. Mensagens podem ser visualizadas à medida que são transmitidas, fornecendo uma ferramenta importantíssima para a depuração de problemas de sincronização entre processos e mensagens [BUR95] [BUR94].

É claro que exige do usuário o conhecimento de todas os comandos de início e controle da máquina paralela virtual, mas os benefícios fornecidos por estes *daemons* com certeza sobrepõe suas desvantagens.

O controle imposto pelo *daemon* implica em certa sobrecarga, principalmente durante a transmissão de mensagens, visto que cada mensagem transmitida é monitorada. O LAM permite que se execute algoritmos paralelos sem o controle do *daemon*, transmitindo mensagens diretamente entre os processos. Porém, tal configuração só deve ser usada quando se tiver certeza do funcionamento correto do algoritmo.

5.4.3. Características Gerais do LAM

Portabilidade: Seu funcionamento restringe-se a redes de *workstations* ou computadores pessoais, executando versões específicas de seus sistemas operacionais. No LINUX, executa apenas na versão 1.3.20. Instala-se também na versão 1.3.7, mas não funciona corretamente.

Instalação: Este trata-se de um ponto negativo do LAM em ambiente LINUX. Apesar de ser facilmente configurado, a compilação do LAM não tem sucesso a não ser que se modifique o seu código fonte. Este problema não é documentado e é de difícil descoberta. Os autores do LAM foram notificados de tal problema, e responderam que este é um problema do LINUX.

Compilação e Execução de Programas: LAM fornece aplicativos padrão que compilam (***hcc***) e iniciam programas (***mpirun***) sem grandes complicações. Uma vantagem do LAM que está relacionada com a existência de *daemons*, é a

facilidade e rapidez no início remoto dos diversos processos a serem executados concorrentemente [MEG95].

Documentação: No LAM é dado maior ênfase na documentação consultada via o comando UNIX ***man*** (apenas para as rotinas próprias do LAM). A documentação escrita é boa, apesar de não ser completa.

Depuração de Programas: Ponto forte do LAM, como foi exposto na seção anterior.

5.4.4. Ferramentas Adicionais

Além das rotinas do MPI, o usuário deve se familiarizar com o ambiente LAM, que implementa diversos comandos para a iniciação, controle e término da máquina paralela virtual. Este conjunto de comandos são simples e úteis em diversas tarefas, como por exemplo, depuração de programas paralelos.

LAM também implementa várias rotinas que complementam e fornecem maior funcionalidade ao MPI. Não são rotinas padronizadas, e a utilização delas implica em perda de portabilidade. Essas rotinas oferecem suporte, por exemplo, a gerenciamento de processos e acesso remoto a arquivos.

O *Ohio SuperComputer Center* também distribui via domínio público um aplicativo com interface gráfica, denominado XMPI, que fornece ferramentas para auxiliar a escrita e depuração de programas paralelos. Infelizmente, o XMPI ainda não é portátil ao sistema operacional LINUX.

5.5. PVM

O PVM (*Parallel Virtual Machine* = Máquina Paralela Virtual), como seu próprio nome sugere, permite que um conjunto de plataformas paralelas heterogêneas conectadas em rede seja utilizado como uma arquitetura paralela de memória distribuída, formando uma máquina paralela virtual. Sua implementação é mantida conjuntamente pelas *Emory University*, *Oak Ridge National Laboratory* e *University of Tennesse* [BEG94] [SUN94].

Os principais objetivos que guiaram a equipe de desenvolvimento do PVM foram investigar características e fornecer soluções para computação paralela heterogênea. Partindo desta premissa, foi desenvolvido um conjunto integrado de

ferramentas de *software* e bibliotecas, que juntos, fornecem uma estrutura computacional sobre a qual podem ser implementadas e executadas aplicações paralelas.

Por possuir uma interface simples e completa, o PVM conseguiu grande aceitação na comunidade acadêmica, e alguns autores o consideram o padrão de fato de plataformas de portabilidade (Enquanto o MPI pretende ser um padrão de direito) [BEG94] [SUN94].

O PVM é composto por duas partes: um processo *daemon*, que executa em todos os computadores da máquina paralela virtual, e um conjunto de rotinas para a implementação de programas paralelos disponíveis em uma biblioteca de passagem de mensagens.

Para execução de tarefas paralelas utilizando o PVM, há a necessidade de que a máquina paralela virtual esteja devidamente iniciada e configurada. Para isso, o PVM dispõe de um aplicativo que oferece um *shell* a partir de onde pode-se incluir máquinas, definir opções de configuração, verificar o *status* da máquina paralela, entre outras funções. Este aplicativo é chamado de PVM *console*.

O PVM é altamente portátil, executando em diversas plataformas paralelas, inclusive computadores paralelos.

Basicamente, a biblioteca de rotinas PVM oferece a seguinte variedade de serviços [BEG94] [SUN94]:

- Comunicação e sincronização entre processos: incluem rotinas de comunicação ponto-a-ponto e coletiva. Os mecanismos de comunicação entre processos são discutidos na seção 5.5.2;
- Gerenciamento de tarefas: permitem iniciar, terminar e obter informações de tarefas dentro da máquina virtual. Um processo executando sobre o controle do PVM é denominado tarefa;
- Configuração dinâmica da máquina virtual: pode-se, por exemplo, adicionar ou retirar computadores da máquina virtual a partir de rotinas ativadas pelo programa do usuário a tempo de execução;
- Grupos dinâmicos de tarefas: se necessário, as tarefas podem ser organizadas em grupos.

O projeto do PVM, durante o seu desenvolvimento, passou por várias fases. A sua primeira versão foi apenas experimental e não foi disponibilizada para domínio público. Passou a domínio público a partir da segunda versão, sendo então em boa parte reescrito para oferecer melhor desempenho, gerando a versão atual,

chamada PVM 3. A arquitetura do PVM 3 é apresentada brevemente na próxima seção.

5.5.1. Arquitetura do PVM

Os mais importantes objetivos do PVM 3 são, segundo Beguelin [BEG94]: tolerância a falhas, escalabilidade, heterogeneidade e portabilidade. As decisões de projeto do PVM foram definidas tendo como prioridade fornecer estas quatro características.

O PVM possui mecanismos que garantem a resistência da máquina paralela virtual a falhas individuais dos computadores e da rede que os une. Porém, esta tolerância a falhas não se aplica ao nível das tarefas executadas pelo usuário (veja seção 5.5.2). O controle da máquina paralela virtual é feita de maneira descentralizada, sempre que possível, garantindo que a máquina possa aumentar de escala sem afetar o seu desempenho.

De maneira similar ao LAM, o PVM também permite que as tarefas comuniquem-se via *daemon* ou diretamente entre elas. A comunicação é implementada utilizando *sockets*.

5.5.2. Comunicação Entre Processos no PVM

Mensagens do PVM, tanto as enviadas por *daemons* ou diretamente, são tipadas, possuem tamanho arbitrário, e são identificadas por um inteiro (*tag*), definido pelo transmissor. O receptor aceita mensagens de acordo com o endereço do transmissor ou da *tag*.

O transmissor da mensagem não espera uma confirmação de recebimento vindo do receptor, mas sim que a mensagem tenha sido completamente enviada (quando as posições de memória de onde se originou podem ser reutilizadas com segurança). Assim, diz-se que rotinas de transmissão de mensagens no PVM são bloqueantes, mas não síncronas (utilizando a terminologia aceita pelo Fórum MPI).

Uma operação de transmissão de mensagens não é tolerante a falhas. Portanto, cabe ao usuário construir sua aplicação de maneira que exista recuperação caso uma mensagem seja perdida antes de alcançar seu destino.

O PVM é utilizado como berço para a implementação do UNIFY, versão do MPI que é apresentada a seguir.

5.6. UNIFY

O projeto UNIFY foi iniciado em uma tese de mestrado na *Mississipi State University*, em 1994, e é uma implementação MPI construída sobre o PVM. Tal característica possibilita ao UNIFY ter duas interfaces disponíveis, de maneira que se possam usar rotinas MPI e PVM no mesmo programa fonte [CHE94] [VAU94].

Esta dualidade de interfaces tem como objetivo proporcionar aos seus usuários uma ferramenta que permita uma migração gradual e fácil da programação PVM para MPI, diminuindo também a curva de aprendizado para os novos usuários do padrão MPI.

A versão do UNIFY utilizada neste trabalho implementa um subconjunto do MPI, e uma versão completa foi prometida pelos autores para o final de 1994. Porém, ou esta versão não é disponível pela *internet*, ou seus autores não terminaram a implementação como previsto.

5.6.1. Arquitetura do UNIFY

O UNIFY foi implementado através de pequenas mudanças no código fonte do PVM, para acrescentar funcionalidade necessária para a implementação das rotinas do MPI. Cada rotina do MPI, quando executada, ativa uma rotina similar do PVM, e segundo os autores do UNIFY, a sobrecarga gerada nesse caso é muito pequena [VAU94].

O UNIFY não possui muita documentação e é difícil afirmar até que ponto todas as ferramentas oferecidas pelo PVM podem ser utilizadas com sucesso sobre um programa UNIFY.

5.6.2. Características Gerais do UNIFY

Portabilidade: Como executa sobre o PVM, possui alta portabilidade.

Instalação: Problemas do LINUX (problemas no *script* **f2c**) e do PVM impedem que o UNIFY se instale sem modificações na sua configuração original. A versão 0.9.2 do UNIFY executa sobre o PVM 3.3.2, versão esta que não executa corretamente sobre o LINUX. Por isso, foram aplicadas as mudanças necessárias no código fonte do PVM 3.3.3. Este não é um procedimento explicado na manual do usuário [CHE94].

Compilação e Execução de Programas: Aspecto bastante restrito do UNIFY. Possibilita apenas execução SPMD ou MPMD com dois programas fonte diferentes (1 mestre e vários escravos).

Documentação: Suficiente, mas de pequeno volume.

Depuração de programas: Como exposto anteriormente, a documentação restrita dificulta saber até que ponto a funcionalidade do PVM pode ser estendida para o UNIFY.

5.7. Comparando as Implementações MPI

Além do desempenho obtido em cada implementação MPI utilizada neste trabalho, tópico a ser discutido no capítulo 8, é importante discutir outros aspectos relacionados a cada uma dessas implementações, a fim de fornecer uma análise mais geral sobre seus aspectos positivos e negativos. Foram escolhidos alguns tópicos que devem ser considerados durante a escolha da implementação a ser usada, e cada um deles foi resumidamente discutido nas seções relacionadas a cada implementação.

É importante lembrar que determinados tópicos devem ser analisados considerando os objetivos centrais de cada implementação, que são: MPICH objetiva ser altamente portátil sem perder eficiência; LAM tem como objetivo emular um multicomputador sobre uma rede de computadores heterogêneos; UNIFY tem como objetivo fornecer um sistema que facilite a migração de programas entre PVM e MPI.

O MPICH, por ser altamente portátil (aspecto positivo do UNIFY e do MPICH) apresenta maior complexidade (quando comparado a outras implementações MPI) para instalação do sistema e para a execução e compilação de programas. Apesar disso, opções complexas nestas tarefas geralmente só são utilizadas em casos especiais e por usuários mais experimentados, e também existe boa documentação disponível.

LAM reduz sua portabilidade apenas para redes de computadores. Mas considerando os seus objetivos, este não pode ser considerado um ponto negativo. Um possível problema, principalmente no sistema LINUX, trata-se da necessidade de execução do LAM em versões específicas do sistema operacional.

Em relação a instalação (que consiste de dois passos: configuração e compilação do sistema), a única plataforma que é instalada sem problemas é o MPICH. O UNIFY e o LAM compilam apenas com a intervenção do usuário. Além disso, enquanto o MPICH e o UNIFY instalaram-se nas duas versões de *kernel* LINUX testadas (e instalam-se em qualquer uma, segundo os seus manuais), LAM apenas funciona na versão 1.3.20.

Em relação a compilação e execução de programas, apenas o UNIFY possui restrições significativas, que podem ser considerados pontos negativos. Um ponto discutido por Meglicki [MEG95], é a sobrecarga no tempo de início de processos maior no MPICH, visto que *daemons* fornecem uma maneira eficiente e simples de iniciar processos (tem-se, no caso, apenas a sobrecarga para o início dos *daemons* para o estabelecimento da máquina paralela virtual). Este problema, na utilização prática dos sistemas, não foi considerado relevante.

A plataforma mais bem documentada é o MPICH, mas apenas o UNIFY possui na documentação um ponto negativo.

Vale a pena ressaltar as ferramentas de depuração do LAM, que pelo uso prático provaram ser de extrema utilidade. Este foi considerado um aspecto positivo bastante significativo da interface LAM.

A nível de ferramentas adicionais, as implementações apresentam diferenças consideráveis, e cabe ao usuário decidir, de acordo com suas aplicações, quais ferramentas podem lhe ser úteis.

Como foi exposto, todas as implementações possuem pontos positivos e negativos, e a escolha entre elas deve ser feita de acordo com a funcionalidade esperada pelo usuário. As conclusões desta seção devem ser consideradas em conjunto com as conclusões do capítulo 8, que apresenta uma comparação a nível de desempenho das rotinas de comunicação ponto-a-ponto de todas as implementações.

5.8. PVM ou MPI?

Uma discussão das diferenças entre as duas plataformas de portabilidade é plausível apenas a nível de funcionalidade oferecida pelas respectivas bibliotecas de passagens de mensagem, visto que o PVM é uma implementação completa (inclui especificação e implementação) e o MPI é apenas uma descrição sintática e semântica de uma biblioteca. Detalhes de implementação não são tratados a fundo pelo documento do padrão MPI.

O usuário, na escolha entre as duas plataforma de portabilidade, deve pesar também a qualidade da implementação MPI a ser utilizada, visto que podem haver

diferenças estruturais muito grandes entre as implementações (veja, por exemplo, as diferenças entre o MPICH e o LAM).

Fundamentalmente, o MPI e o PVM diferem na relação que apresentam entre complexidade/funcionalidade. O MPI possui uma especificação longa e relativamente complexa, para se tornar viável como padrão eficiente para arquiteturas computacionais tão diversas e para fornecer todas as características consideradas importantes para uma biblioteca de passagem de mensagens. O PVM, por outro lado, considera como um objetivo de projeto ser simples (e suficientemente completo), a fim de facilitar o trabalho do programador. Como exemplo, considere uma das rotinas básicas de comunicação ponto-a-ponto, a rotina *send( )*. No PVM, pode-se mandar uma mensagem de duas maneiras diferentes, enquanto que, no MPI, existem 14 (quatorze) variações.

É claro que o usuário tem um leque muito maior de opções no MPI, e pode escolher a opção que melhor funcione baseado na sua plataforma paralela e na sua aplicação. Mas, para fazer essa escolha o usuário deve conhecer o MPI em detalhes, e também deve escolher uma implementação eficiente na sua plataforma, o que pode não ser uma tarefa simples. Além disso, é importante lembrar que a maioria dos algoritmos paralelos podem ser descritos com um conjunto pequeno e simples de rotinas.

Ambas as plataformas de portabilidade têm pontos favoráveis, que podem influir na escolha entre elas. Por exemplo, o PVM define rotinas de início e término de processos (o MPI ainda não), e suporta explicitamente a heterogeneidade, enquanto o MPI foi projetado para ser utilizado em ambientes heterogêneos mas deixa a cargo da implementação possibilitar ou não esta característica.

Por outro lado, o MPI fornece estruturas que possibilitam maior confiabilidade e segurança a operações de comunicação entre processos (contextos, grupos e *communicators*) e rotinas que possibilitam a sobreposição entre comunicação e execução (rotinas não bloqueantes).

A escolha de uma plataforma de portabilidade deve ser feita levando em conta quais as necessidades da aplicação, qual plataforma paralela a ser usada, a implementação do MPI a ser utilizada, entre outros aspectos. Dessa maneira, deve-se ser cauteloso para afirmar que uma plataforma de portabilidade seja, no geral, melhor que outra.

5.9. Considerações Finais

Atualmente, diversas implementações de bibliotecas de passagem de mensagens estão disponíveis, fornecendo um leque variado de escolhas para o usuário. Mas, enquanto esta diversidade pode trazer vantagens, visto que pode-se optar pela biblioteca que melhor se comporte na aplicação do usuário, fica difícil a portabilidade de programas paralelos e, por conseguinte, a difusão comercial e empresarial da computação paralela é dificultada.

Neste capítulo foram apresentadas três das principais implementações de plataformas de portabilidade existentes, uma que atualmente pode ser chamada de padrão "de fato" (PVM), e duas que são implementações de uma plataforma que pretende ser um padrão de lei (MPICH e LAM). Todas têm seus pontos fortes e fraquezas, e é difícil prever no futuro quais se sobressairão.

Ambas as plataformas de portabilidade têm sido muito estudadas pela comunidade acadêmica mundial, possibilitando cada vez mais a melhor compreensão das diferenças e qualidades de cada uma delas. O MPI, por ser uma iniciativa ambiciosa e recente, ainda mostra-se como uma incógnita, e este trabalho, como outros correlatos, tem como objetivo estudar o seu comportamento em determinadas condições, permitindo maior compreensão de seu poder e utilidade.

Neste capítulo, foram estudadas as três implementações MPI utilizadas neste trabalho e também o PVM, possibilitando uma visão geral de cada uma das plataformas de portabilidade e também da relação entre elas. Compreender as implementações possibilita melhor segurança para concluir sobre o desempenho de suas rotinas de comunicação.

No próximo capítulo, são apresentadas em detalhes todas as variantes possíveis de estruturas para a comunicação entre dois processos no MPI, denominada comunicação ponto-a-ponto.

Capítulo 6

Comunicação Entre Processos no MPI

Neste capítulo são apresentadas em detalhes as rotinas de comunicação ponto-a-ponto no MPI, discutindo seus tipos, características, vantagens e desvantagens.

6.1. Introdução

Uma biblioteca de passagem de mensagens, em geral, possui um conjunto de rotinas que possibilitam a ativação, organização e comunicação de processos paralelos. O núcleo destas bibliotecas é composto pelas rotinas que fornecem funcionalidade e organização para a comunicação entre processos, levando-se a concluir que o desempenho oferecido por estas rotinas possui influência significativa no desempenho do sistema como um todo.

Esta afirmação é particularmente verdadeira quando se considera o MPI, cuja maioria de funções são relacionadas as operações de comunicação, sejam ponto-a-ponto ou coletivas [WAL94]. Por isso, avaliar o desempenho das rotinas de comunicação definidas pelo MPI fornece resultados extremamente significativos, que até certo ponto podem ser considerados quando da análise do padrão como um todo.

Este trabalho objetiva estudar o desempenho das rotinas de comunicação ponto-a-ponto. As rotinas coletivas não são consideradas neste estudo por dois motivos:

- a grande variedade de rotinas coletivas existentes no MPI justifica a escrita de uma dissertação própria (seção 9.5);
- muitas plataformas de portabilidade implementam suas rotinas coletivas sobre rotinas de comunicação ponto-a-ponto.

O MPI oferece várias maneiras de comunicação ponto-a-ponto e é importante estudar todas elas, a nível de funcionalidade oferecida e desempenho, a fim de possibilitar conclusões mais completas.

As rotinas de comunicação ponto-a-ponto são caracterizadas por 4 aspectos básicos, que são: bloqueio, modo de comunicação, persistência de requisições e

sentido da comunicação. Cada um desses aspectos é discutido separadamente (seções 6.2.1, 6.2.2, 6.2.3, 6.2.4), para na seção 6.3 serem discutidas as relações entre eles.

Neste capítulo, toda referência a comunicação entre processos deve ser entendida como comunicação ponto-a-ponto. Além disso, para evitar confusão de nomenclaturas, a partir deste capítulo é evitado a utilização dos termos rotinas *send( ) e receive( )* no sentido genérico, como apresentado nos capítulos anteriores, preferindo-se a utilização dos termos rotinas de transmissão e recepção.

6.2. Comunicação Ponto-a-Ponto

O MPI possui como um de seus objetivos centrais ser um padrão multi-plataforma sem abrir mão da eficiência. Isto quer dizer que, qualquer que seja a plataforma, sempre deve ser possível escrever um programa MPI eficiente. Além disso, o usuário deve possuir alternativas para que possa escolher a melhor organização para seu algoritmo paralelo, baseado em seus objetivos e idéias.

A necessidade de garantir eficiência e generalidade (características básicas necessárias a uma plataforma de portabilidade que pretende ser um padrão), levou o Fórum MPI a definir um padrão relativamente extenso, de maneira que determinadas funções do MPI possuem muitas variações. E dentro deste contexto inserem-se as operações de comunicação ponto-a-ponto.

O MPI fornece uma grande variedade de rotinas de comunicação ponto-a-ponto, e estas são caracterizadas basicamente por quatro aspectos básicos:

- quanto ao bloqueio
  - ⇒Comunicação bloqueante;
  - ⇒Comunicação não bloqueante;

- quanto ao modo de comunicação
  - ⇒padrão;
  - ⇒síncrono;
  - ⇒*bufferizado*;
  - ⇒*ready*;

- quanto a persistência
  - ⇒rotinas ativadas por requisições persistentes;
  - ⇒rotinas não persistentes;

- quanto ao sentido da comunicação
  - ⇒rotinas de comunicação em dois sentidos;
  - ⇒.rotinas isoladas.

Nas seções 6.2.1 até 6.2.4, cada um desses aspectos são discutidos. O estudo das diferentes variações de rotinas de comunicação ponto-a-ponto no MPI foi feito baseado nas seguintes referências: [BUR95] [DON95] [MAC96] [SNI96] [WAL94] [WAL95].

6.2.1. Comunicação Bloqueante X Não Bloqueante

Uma operação de transferência de uma mensagem é dita completa (terminada) quando a "variável de transmissão" (posições de memória de onde a mensagem é originada) pode ser reutilizada com segurança (para o transmissor), ou quando a mensagem foi totalmente transferida para a "variável de recepção" (para o receptor). Rotinas bloqueantes garantem que o processo transmissor ou receptor ficarão bloqueados até que a transmissão da mensagem seja completada. Caso contrário, tem-se uma rotina não bloqueante.

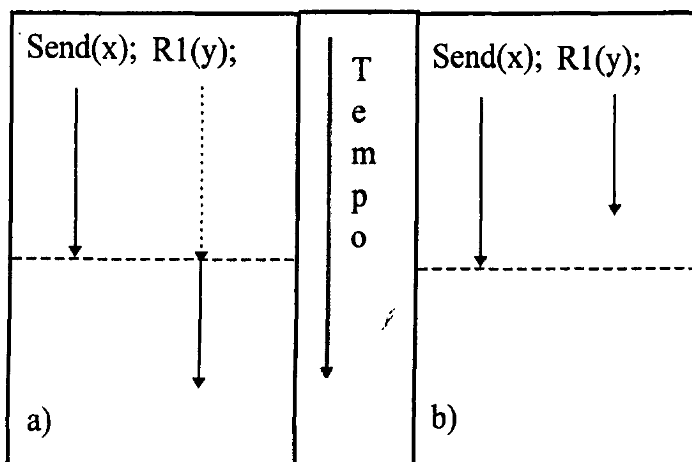

Figura 6.1 - Comunicação Ponto-a-Ponto a) Bloqueante b) Não Bloqueante.

O bloqueio de uma rotina de comunicação no MPI é associado ao momento de execução das rotinas que a seguem. Suponha, por exemplo, que após uma operação de transmissão de uma mensagem seja executada uma rotina qualquer, genericamente chamada R1(y). Se a rotina de transmissão for bloqueante, a rotina R1(y) só será executada quando a transmissão estiver completada (Figura 6.2. (a)). Se a rotina for não bloqueante, a rotina R1(y) será executada imediatamente após o inicio da comunicação, de maneira que ambas as rotinas serão executadas concorrentemente (Figura 6.1. (b)).

O objetivo de uma rotina não bloqueante é permitir a sobreposição entre a transmissão (ou a recepção) de uma mensagem com a execução das rotinas que a sucedem, de maneira que se possa continuar a execução do programa paralelo enquanto as operações de comunicação estejam em andamento. Dessa maneira pode-se conseguir maior desempenho em determinadas aplicações. Rotinas não bloqueantes podem ser especialmente utilizadas quando se possui *hardware* de comunicação especializado e também suporte a múltiplas linhas de controle (*threads*) [MAC96] [SNI96]..

Uma operação não bloqueante é dividida em duas partes: uma rotina que inicia a transmissão ou a recepção da mensagem e retorna imediatamente e uma segunda rotina que verifica se a operação de comunicação terminou ou não (ver seção 6.3). Para rotinas não bloqueantes, fica a cargo do programador o cuidado de não utilizar as variáveis participantes da comunicação (variáveis de transmissão e recepção) sem a prévia verificação de seu término.

Além de permitir a possibilidade de comunicação ocorrer em paralelo ao processamento, rotinas não bloqueantes também evitam a possibilidade de *deadlocks*, e podem ser usadas como substitutas as rotinas de transmissão utilizando o modo *bufferizado* (ver seção 6.2.2) [MAC96] [SNI96].

Um exemplo prático da utilização de rotinas não bloqueantes é apresentado na seção 7.5, onde é discutido o algoritmo de Jacobi.

6.2.2. Modos de Comunicação

Uma operação de transferência de uma mensagem, apesar de simples no seu conceito, possui implicações de extrema importância e que influenciam no seu desempenho e confiabilidade. Por exemplo, supondo que um processo inicie a transmissão de uma mensagem, quando do término desta operação, pode-se saber se o processo receptor iniciou ou não a recepção desta mensagem? Aspectos como estes são estudados a partir de duas características centrais de uma operação de

comunicação ponto-a-ponto: a semântica das primitivas de comunicação e os protocolo que a implementam [MAC96] [SNI96].

Se o usuário queira utilizar um protocolo de transmissão de mensagens que evite a cópia e a *bufferização* de mensagens, pode optar por uma semântica síncrona, onde o transmissor espera até que receba uma resposta do receptor. Por outro lado, para se bloquear processos pelo menor tempo possível, deve-se optar por uma semântica de *bufferização* de mensagens, de maneira que após copiadas para o *buffer* possam ser transferidas quando possível, liberando imediatamente o processo transmissor [SNI96].

A figura 6.2 apresenta uma esquematização dos dois tipos de semânticas. Nesse exemplo, dois processos trocam uma mensagem identificada por **x**, e considerando que o tempo flui na direção indicada, as setas abaixo das rotinas de comunicação indicam o momento onde a próxima rotinas podem modificar o valor de **x**, isto é, a operação de comunicação é considerada terminada (este exemplo apresenta rotinas de comunicação bloqueante).

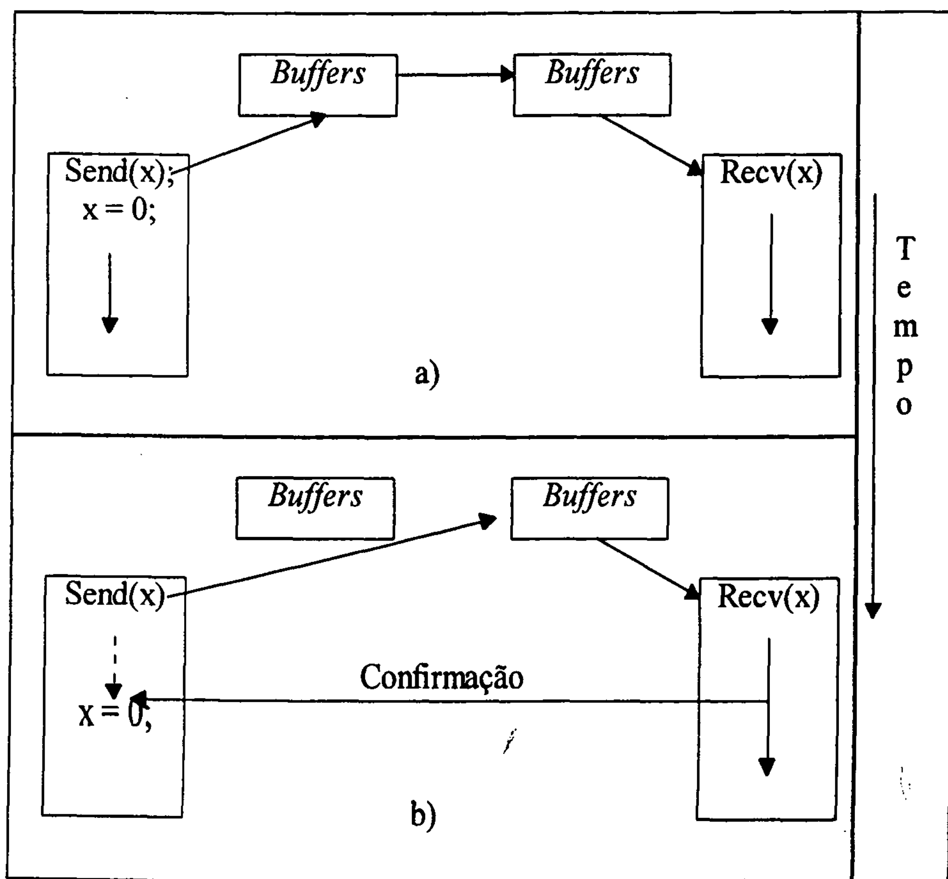

Figura 6.2 - Semânticas de Comunicação a) *Bufferizada* b) Síncrona.

Diferentes aplicações e arquiteturas podem oferecer diferentes desempenhos para uma determinada semântica. Por isso, o MPI permite que o usuário escolha a

semântica que melhor se encaixe na sua aplicação, através da definição de **modos de comunicação**.

O modo de comunicação define precisamente quando uma operação de comunicação pode ser considerada terminada, de maneira que as variáveis de transmissão e recepção possam ser utilizadas com segurança. Definido o momento exato de término de uma rotina de comunicação, pode-se então definir variantes bloqueantes e não bloqueantes para cada um desses modos.

Modos de comunicação são discutidos no escopo das rotinas de transmissão, visto que, a transferência de uma mensagem sempre é iniciada pelo seu transmissor, que por isso exerce um papel fundamental de controle da comunicação utilizada em um programa paralelo. Assim, os modos de comunicação são definidos para as rotinas de transmissão de mensagens e a avaliação apresentada neste trabalho se concentra sobre as rotinas de transmissão de mensagens.

Nas próximas seções, são apresentados todos os modos de comunicação, explicando suas características, benefícios e problemas.

Modo Padrão: No modo padrão, o término de uma operação de comunicação *pode ou não* significar que o receptor já foi ativado. Fica a cargo da implementação escolher entras as semânticas síncrona ou de *bufferização.* Geralmente, essa escolha é baseada na semântica que seja mais eficiente na arquitetura para a qual a implementação foi projetada [MAC96] [SNI96].

A utilização do modo padrão deve ser feita com certo cuidado, de maneira que o usuário deve seguir determinadas regras ao utilizá-lo. Suponha, por exemplo, dois processos (A e B) trocando mensagens (Figura 6.3).

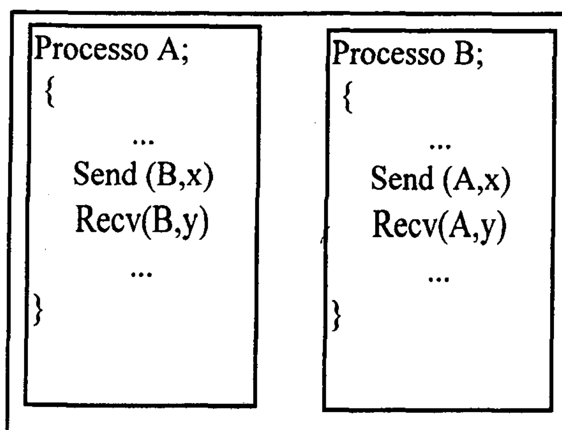

Figura 6.3 - Estrutura de Comunicação Insegura.

Este tipo de organização para operações de comunicação é considerada insegura, pois pode levar a situações de *deadlock*, onde ambos os processos estão bloqueados cada um esperando que o outro desbloqueie. Analisando este caso nas duas semânticas citadas acima, têm-se: implementando a rotina *send( )* utilizando a semântica síncrona, os processos p1 e p2 certamente estarão em *deadlock*. Já, utilizando a semântica de *bufferização*, não existe o bloqueio das rotinas, evitando o *deadlock*. Porém, se o espaço em *buffer* se esgota, o processo acaba bloqueado até que haja liberação deste espaço. Nesse caso, dependendo da situação, pode haver o *deadlock* [MAC96] [SNI96].

A escolha da semântica de comunicação para o modo padrão e o espaço em *buffer* disponível quando da escolha de uma semântica de *bufferização*, são todas decisões livres para cada implementação. Nesse caso, é fácil perceber que um algoritmo construído de maneira insegura pode executar de maneiras totalmente distintas em diferentes implementações. Por isso, a implementação de algoritmos portáteis requer que sejam criadas estruturas seguras de comunicação. No caso acima, uma maneira de construção segura da comunicação entre A e B é:

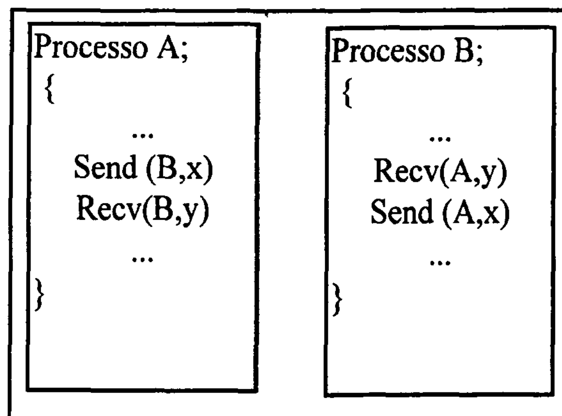

Figura 6.4 - Estrutura de Comunicação Segura.

O exemplo citado na figura 6.3 pode ser estendido para mais de dois processos comunicando-se entre si de maneira semelhante, possibilitando também o risco de *deadlocks*. Nesse caso, uma alternativa segura seria a utilização do padrão "vermelho-preto" (*red-black*) de comunicação, onde processos pares primeiro enviam e depois recebem, e processos ímpares fazem o contrário [MAC96].

Modo Síncrono: O modo síncrono implementa uma semântica síncrona para a transmissão de uma mensagem. Assim, o receptor deve enviar uma confirmação de

recebimento da mensagem, de maneira que o transmissor pode ter certeza de que a mensagem foi recebida.

O processo transmissor deve esperar até que o processo receptor alcance determinado ponto. Algoritmos que utilizam comunicação síncrona podem ser mais lentos que os não síncronos ou *bufferizados*. Porém, consegue-se maior confiabilidade (em casos, por exemplo, de perda de mensagens), e evita-se sobrecarga da rede por mensagens não recebidas. Além disso, possibilita que o comportamento de um programa paralelo seja melhor controlado, facilitando a sua depuração [MAC96] [SNI96].

É importante ressaltar que problemas de sincronização não desejada (como *deadlocks)* podem ocorrer, de maneira que o usuário deve ter cautela na organização de um algoritmo paralelo utilizando rotinas síncronas.

Modo ***Bufferizado***: A transmissão de uma mensagem utilizando *buffers* permite que esta se complete rapidamente, visto que, após ser copiada para o *buffer*, fica a cargo do sistema transmitir a mensagem quando possível. Nesse caso, não há a necessidade de que transmissor e receptor estejam sincronizados, possibilitando em determinados casos maior desempenho [MAC96] [SNI96].

Um possível problema é a sobrecarga gerada sobre a rede, onde receptores lentos podem criar um congestionamento de mensagens não entregues, gerando um possível esgotamento dos *buffers*. O modo *bufferizado* obriga que seja retornado uma mensagem de erro nesses casos, não gerando problemas de *deadlock*.

Uma restrição neste caso é que o usuário não deve considerar nenhum tipo de *buffer* pré-alocado pelo sistema, e deve explicitamente criar e controlar *buffers* suficientes para sua aplicação através de rotinas definidas pelo MPI.

Modo ***Ready***: Uma rotina de transmissão de mensagens no modo *ready* é terminada rapidamente, sem a utilização de *buffers* ou de confirmações do processo receptor, objetivando conseguir melhor desempenho em determinados ambientes computacionais (principalmente arquiteturas paralelas). O transmissor simplesmente envia a mensagem, de maneira que o receptor já deve estar preparado para recebe-la. Se o processo receptor não estiver pronto, o resultado desta operação será imprevisível [MAC96] [SNI96].

Percebe-se que a responsabilidade de garantir o seu correto funcionamento fica a cargo do usuário, que deve organizar seu programa de maneira que o fluxo dos vários processos paralelos garanta a sincronização de suas respectivas operações de comunicação.

O modo *ready* é um tipo de comunicação que deve ser utilizada em casos especiais e com extremo cuidado.

6.2.3. Requisições Persistentes

É comum que uma rotina de comunicação (recepção ou transmissão) se repita varias vezes em um mesmo programa paralelo, de maneira que todos os parâmetros da rotina (receptor ou transmissor, tipo, *tag*) mantenham-se os mesmos, variando apenas o conteúdo da mensagem.

Uma requisição persistente de comunicação tem o objetivo de evitar a sobrecarga gerada pelas consecutivas chamadas a uma mesma rotina. Todo tipo de procedimentos iniciais relacionados a uma operação de comunicação, que não dependem do conteúdo de uma mensagem, são então executados apenas quando a requisição persistente é criada. Uma vez criada, é apenas necessário ativar a requisição (através de rotinas definidas para esse fim - seção 6.3) para que a rotina de comunicação seja executada [MAC96] [SNI96].

Requisições persistentes iniciam apenas rotinas de comunicação não bloqueante, e são utilizadas para rotinas de transmissão (em todos os modos de comunicação) ou recepção de mensagens. Oferecem maiores benefícios para arquiteturas paralelas e processadores que possuam *hardware* especializado para comunicação.

6.2.4. Comunicação em Dois Sentidos

É relativamente comum em algoritmos paralelos a ocorrência do padrão de comunicação inseguro exemplificado no figura 6.3, onde dois ou mais processos trocam mensagens através da execução de operações de transmissão e recepção de mensagens em rotinas consecutivas (comunicação *ping-pong*).

Uma maneira de evitar o problema de *deadlocks* sem modificar a ordem das rotinas de comunicação (padrão "vermelho-preto", exposto na seção 6.2.2) é a utilização de rotinas de comunicação em dois sentidos, que são formadas pela aglutinação de operações de transmissão e recepção de mensagens consecutivas em uma única rotina. Dessa maneira, as rotinas são ativadas ao mesmo tempo e executam concorrentemente, permitindo ganho de desempenho em determinadas situações [MAC96] [SNI96].

Rotinas de comunicação em dois sentidos são bloqueantes, combinam-se com quaisquer outras rotinas de transmissão ou recepção do MPI, e apresentam duas variantes, que são: **MPI_Sendrecv** e **MPI_Sendrecv_replace**.

As duas rotinas diferenciam-se pela utilização de variáveis de transmissão (onde a mensagem se origina) e recepção (onde a mensagem será copiada) coincidentes ou não. **MPI_Sendrecv_replace** utiliza as mesmas posições de memória para transmitir e receber a mensagem enquanto **MPI_Sendrecv** não. No primeiro caso, há economia de memória mas existe a necessidade das mensagens enviadas e recebidas terem tipos iguais. Ambas as duas rotinas, apesar de serem bloqueantes, garantem a não ocorrência de *deadlock.*

Deve-se ressaltar que a nomenclatura "comunicação em dois sentidos" não é definida pelo MPI, e foi utilizada neste trabalho para facilitar a organização do texto.

6.3. Resumo das Rotinas de Comunicação do MPI

Nas seções anteriores, foram apresentadas as possíveis maneiras de transmissão e recepção de uma mensagem no MPI. A melhor maneira de compreender as relações entre elas, porém, é estudá-las comparativamente, a fim de perceber as diferenças e semelhanças entre elas.

Na tabela 6.1 são apresentadas todas as rotinas de transmissão e recepção de mensagens no MPI, relacionadas em conjuntos definidos pelas suas características de bloqueio e semântica (definida pelo modo de comunicação). Para todas as rotinas são apresentadas as respectivas sintaxes adotadas pelo MPI.

Existem 12 maneiras de transmitir uma mensagem, divididas em quatro modos de comunicação: padrão, síncrono, *bufferizado* e *ready.* Cada um dos modos apresenta versões bloqueantes ou não bloqueantes, onde, rotinas não bloqueantes podem ser definidas por requisições persistentes ou não

A recepção de uma mensagem pode ser feita de 3 maneiras, sempre no modo de comunicação padrão. Uma versão é bloqueante, e duas versões são não bloqueantes (uma delas iniciada via uma requisição persistente).

Rotinas bloqueantes podem também ser organizadas em operações de comunicação em duas sentidos, onde são combinadas rotinas de transmissão e recepção de mensagens.

De maneira similar, as rotinas de suporte as rotinas não bloqueantes e requisição persistentes são apresentadas na tabela 6.2. Estas são separadas de acordo com o número de rotinas as quais elas se relacionam.

Rotinas não bloqueantes (ativadas por requisições persistentes ou não) apenas iniciam a transmissão ou recepção de uma mensagem, e seu término é verificado pela utilização de 4 variações de rotinas de teste (testa se a comunicação terminou) e 4 de bloqueio (bloqueia-se até o final de comunicação).

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

Tabela 6.1 - Rotinas de Comunicação no MPI

	Rotina Específica	**Qualquer rotina**	**Algumas rotinas**	**Todas rotinas**
Bloqueia até o final da comunicação	MPI_Wait	MPI_Waitany	MPI_Waitsome	MPI_Waitall
Testa o final da comunicação	MPI_Test	MPI_Testany	MPI_Testsome	MPI_Testall
Inicia Requisição Persistente	MPI_Start	X	X	MPI_Startall

Tabela 6.2 - Rotinas Auxiliares Para Comunicação Não Bloqueante

Suponha que sejam iniciadas 4 rotinas de comunicação não bloqueante, identificadas pelos letras **x**, **y**, **z** e **w**. Pode-se verificar o término de uma específica (x, por exemplo), de qualquer uma (**x**, **y**, **z** ou **w**), de algumas explicitamente definidas (**x** e **y**, por exemplo) e de todas (**x**, **y**, **z** e **w**). Além disso, requisições persistentes são ativadas por 2 rotinas, onde uma inicia apenas uma rotina específica e a outra inicia todas as rotinas definidas.

6.4. Considerações Finais

Como foi exposto neste capítulo, o usuário tem uma grande gama de opções quando deseja estruturar as operações de comunicação entre processos quando da utilização do MPI. Esta grande variedade de opções possibilita que o usuário escolha o tipo de rotina de comunicação que melhor se adapta a sua aplicação, e deste ponto de vista, pode-se considerar esta uma vantagem da utilização do MPI.

Porém, estudar e entender cada uma das opções, e mais, verificar o quanto esta pode oferecer de desempenho na plataforma paralela utilizada, oferece um obstáculo que pode não compensar o esforço despendido. Por isso, é mais natural que a primeira opção do usuário geralmente são as rotinas de comunicação básicas, geralmente bloqueantes e implementadas no modo padrão. Pode-se dizer que o usuário se sentirá mais seguro em utilizar funções que lhe parecem mais naturais e descomplicadas.

Dentro deste contexto, este capítulo apresentou um estudo a nível de estrutura e desempenho das rotinas de comunicação ponto-a-ponto do MPI para plataforma LINUX, de maneira que o usuário possua um conjunto de informações as quais possam facilitar a sua compreensão do padrão e também a escolha das melhores opções para a sua aplicação.

Enquanto a estrutura de cada uma das rotinas de comunicação foi apresentada em detalhes nesse capítulo, para estudar o seu desempenho é necessário definir métricas e modelos que permitam avaliar e comparar estas rotinas. No próximo capítulo, é apresentada a metodologia de estudo e avaliação de desempenho utilizada neste trabalho.

Os resultados da avaliação de desempenho serão combinados com as considerações apresentadas nesse capítulo, possibilitando uma discussão completa das vantagens e desvantagens de cada rotina de comunicação ponto-a-ponto que será apresentada na seção 8.6.

Capítulo 7

Modelos de Avaliação e Métricas de Desempenho

Neste capítulo são discutidos os métodos de avaliação de desempenho utilizados nesse trabalho. É apresentado o modelo de avaliação utilizado, as métricas de desempenho consideradas, os *benchmarks* implementados e o algoritmo de Jacobi.

7.1. Introdução

Avaliar o desempenho de uma rotina de comunicação ponto-a-ponto significa avaliar o quanto as sobrecargas de *software* (geradas pela biblioteca de comunicação) e de rede (gerada pelo protocolo de comunicação de rede) influenciam sobre a transferência de uma mensagem [NUP94]. Estes são dois dos principais fatores de diferenciação de desempenho para rotinas de comunicação executadas de diferentes maneiras em diferentes implementações de bibliotecas de passagem de mensagens.

Através da execução de experimentos práticos chamados *benchmarks*, pode-se coletar informações que possibilitem medir a eficiência de uma determinada operação de comunicação. Essas informações, que possibilitam avaliar os diferentes efeitos das sobrecargas de *software* e rede na transferência de uma mensagem, são utilizadas como parâmetros para a análise de desempenho de uma operação de comunicação e são definidas como métricas de desempenho. As métricas utilizadas neste trabalho são apresentadas na seção 7.3, e os *benchmarks*, na seção 7.4 [DIL95] [NUP94].

A execução dos *benchmarks* é feita a nível de aplicação (não são inseridas rotinas nos códigos fontes das implementações) e é organizada segundo requisitos definidos por um modelo de avaliação, que é apresentado na seção 7.2.

Além das métricas de desempenho, resultados obtidos da execução de um algoritmo paralelo real, chamado algoritmo de Jacobi [QUI94] [SNI96], foram incluídos nesse trabalho, pelos seguintes motivos:

- Determinadas rotinas de comunicação do MPI possuem características próprias que dificultam a sua análise através das métricas de desempenho, como por exemplo, as rotinas não bloqueantes;

- É interessante demonstrar o quanto os valores das métricas de desempenho obtidas se refletem quando da utilização de uma aplicação real.

O algoritmo de Jacobi é apresentado na seção 7.5.

Todos os *benchmarks* e métricas de desempenho utilizadas nesse trabalho foram definidas através do estudo de trabalhos correlatos [DIL95] [DOU93] [DUN95] [GRO95] [NEV96] [NUP94].

7.2. Modelo Computacional de Avaliação

Para a execução dos *benchmarks* e do algoritmo de Jacobi, foi considerado o modelo de avaliação apresentado na figura 7.1 [DIL95] [NUP94].

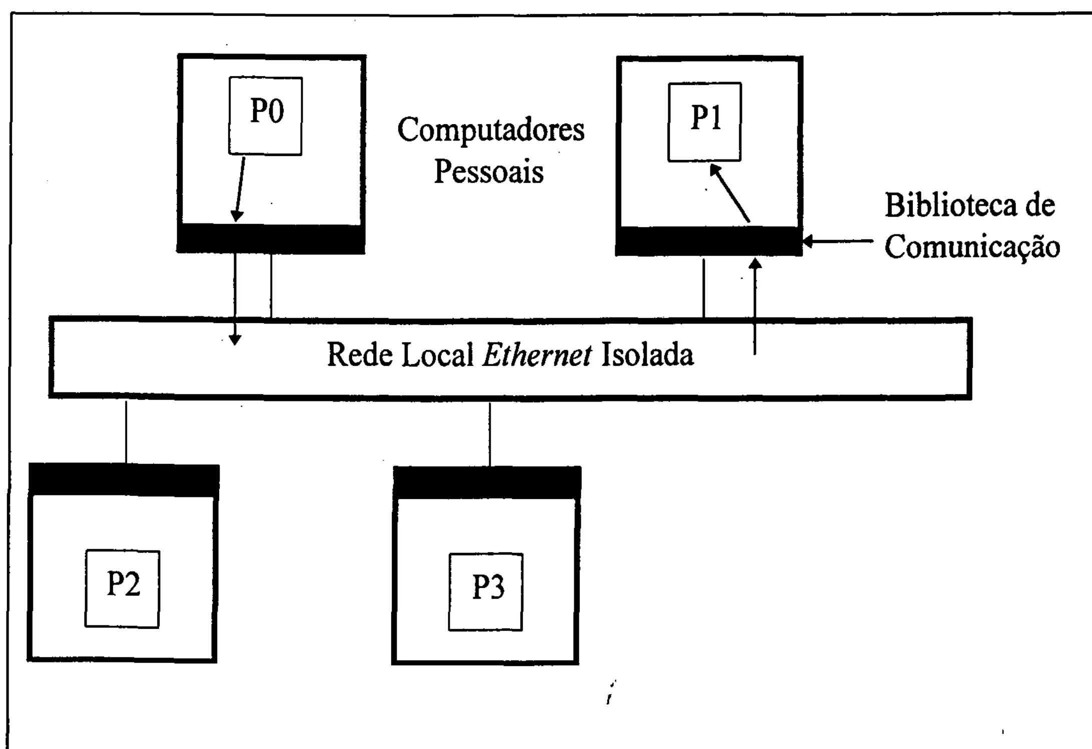

Figura 7.1 - Modelo de Avaliação.

Os *benchmarks* e o algoritmo de Jacobi são executados em uma rede local *Ethernet* isolada (utilizando dois e quatro computadores, respectivamente), onde cada computador executa apenas um processo (para evitar a competição por recursos). O LINUX é um sistema multi-tarefa, e o processo já deve compartilhar a CPU com diversos processos *daemons* comuns a um sistema UNIX. O modelo considera computadores homogêneos, de maneira que todas e quaisquer operações para o tratamento de heterogeneidade

(como a codificação de uma mensagem em um formato padrão, como o XDR [STE90]), não são consideradas.

Os *benchmarks* foram executados sobre condições idênticas. Foram executados de madrugada (para evitar uma carga de trabalho nos computadores muito variável e para não atrapalhar outros usuários) e a disposição e localização de seus processos constituintes foi mantida idêntica. Dessa maneira, toda e qualquer possível influência ocasionada por determinadas configurações computacionais (como potência do processador, quantidade de memória, entre outros) são refletidas de maneira idêntica em todos os resultados obtidos. Essas afirmações também devem são consideradas para a execução do algoritmo de Jacobi.

Os *benchmarks* calculam o tempo decorrido em determinadas fases da transmissão de uma mensagem, a fim de fornecer relações entre quantidades de dados transmitidos e tempo gasto para transmiti-los (a partir destes dados, podem ser calculadas as métricas de desempenho).

A transmissão de uma mensagem pode ser dividida em três fases distintas, cada uma definindo um período de tempo de latência (demora) em virtude de suas respectivas sobrecargas [DIL95] :

- **Latência de transmissão** (t_{send}): Antes de ser transmitida, a mensagem precisa ser processada, a fim de acrescentar cabeçalhos, calcular *checksums*, entre outras tarefas. Esta latência é definida pela sobrecarga imposta pela biblioteca de comunicação;
- **Latência da rede** (t_{net}): Preparada a mensagem, ela está sujeita a sobrecarga imposta durante a transposição da rede de comunicação, e aí deve ser incluído a sobrecarga imposta, por exemplo, pelo protocolo de comunicação de rede;
- **Latência de recepção** (t_{recv}): Após ser recebida, a mensagem é processada pelo receptor (Por exemplo: verificação de *checksums*, retirada de cabeçalhos, etc.). Esta latência também existe em virtude da sobrecarga da biblioteca.

Concentrando o estudo sobre estas três variáveis, pode-se avaliar a sobrecarga imposta à uma rotina de comunicação ponto-a-ponto. Estes três valores dependem, em diferentes graus, do tamanho da mensagem a ser transmitida, e por isso devem ser analisadas em conjunto com o tamanho dos dados transmitidos. Utilizando-se sintaxe semelhante à definida por Dillon [DIL95], pode-se definir o tempo de transferência de uma mensagem entre dois processos A e B, como:

$$t_{A \rightarrow B}(n) = t_{send}(n) + t_{net}(n) + t_{recv}(n) \quad (7.1)$$

onde *n* é o tamanho da mensagem transmitida (em *bytes*) e t_{send}, t_{net}, e t_{recv} são os tempos relacionados a cada uma das latências citadas acima.

Baseado na equação 7.1, são definidas as métricas de desempenho, que são apresentadas na próxima seção.

7.3. Métricas de Desempenho

As métricas de desempenho foram definidas de acordo com as características de uma rotina de comunicação ponto-a-ponto (partindo da análise da equação 7.1) e também de acordo com a organização dos *benchmarks* [DIL95] [NEV96] [NUP94].

Latência da Biblioteca de Comunicação (t_{send}): Representa a sobrecarga imposta pela biblioteca, isto é, o tempo gasto para o processamento efetivado pelas rotinas internas da biblioteca. Nesse trabalho, esta latência foi definida em termos práticos como o tempo que uma rotina de transmissão gasta até devolver o controle para o usuário, independente da semântica utilizada para esta transmissão. Pelas características do MPI, o retorno do controle para o usuário pode implicar no recebimento completo da mensagem pelo receptor (rotinas síncronas) ou não (rotinas *bufferizadas*). Então, a latência da biblioteca, como definido neste trabalho, pode incluir também as tempos de latência de rede.

A latência da biblioteca é calculada sobre as rotinas de transmissão de mensagens, pois essa operação é responsável pelo início da transferência da mensagem e sobre ela estão definidos os modos de comunicação. Além disso, a estrutura dos *benchmarks* dificultam o cálculo do tempo de recepção, pois não há como precisar o momento que esta rotina se iniciará.

A latência da biblioteca é definida, utilizando a nomenclatura e variáveis apresentadas na equação 7.1, como:

$$t_{send}(n) = t_{pack}(n) + t_{trans}(n) + t_{buffer}(n) + t_{start} \quad (7.2)$$

Na equação 7.2., $t_{pack}(n)$ representa o tempo gasto no tratamento de heterogeneidade (por exemplo, codificando a mensagem para o formato XDR [STE90]), que nesse trabalho será considerado nulo. $t_{trans}(n)$ implica no tempo de transmissão da mensagem (aqui também pode ser incluído o tempo de espera por confirmações do receptor). Dependendo do modo de comunicação e da

implementação do MPI utilizada, deve-se considerar também o tempo de cópia da mensagem para o *buffer* ($t_{buffer}(n)$). O tempo t_{start} representa a sobrecarga constante e independente do tamanho da mensagem, e que pode ser calculada através da transmissão de uma mensagem de tamanho nulo. Cada um desses tempos influi significativamente no tempo de transmissão de uma mensagem. Como a ênfase desse trabalho está na análise das rotinas de comunicação a nível de aplicação, está fora de escopo a apresentação de um estudo para cada um desses tempos citados.

Latência de Comunicação Ponto-a-Ponto ($t_{A \to B}$): Esta latência representa o tempo total de transmissão de uma mensagem, incluindo o tempo de recepção. Além da sobrecarga da biblioteca de comunicação, a sobrecarga imposta pelo protocolo de comunicação de rede também influencia nos tempos de transmissão ponto-a-ponto. Esta latência é representada exatamente pela equação 7.1.

Um problema para o seu cálculo é o fato de não haver sincronização de *clocks* entre os computadores de uma rede LINUX. Como os processos transmissor e receptor executam em máquinas diferentes, não há como calcular o tempo de transmissão considerando apenas uma operação de comunicação. Nesse caso, deve-se calcular o tempo de *round-trip* ($t_{A \leftrightarrow B}$), ou seja, o tempo de transmissão da mensagem do transmissor (por exemplo, processo A) para o receptor (processo B) e o tempo da mesma operação executada em sentido inverso. O tempo obtido é dividido por dois. Colocando em termos mais formais, tem-se:

$$t_{A \leftrightarrow B}(n) = t_{send_A}(n) + t_{net_{A \to B}}(n) + t_{recv_B}(n) + t_{send_B}(n) + t_{net_{B \to A}}(n) + t_{recv_A}(n) \quad (7.3)$$

Posto que os computadores onde os processos A e B executam são homogêneos, e assumindo que $t_{net_{A \to B}}(n) = t_{net_{B \to A}}(n)$, tem-se que:

$$t_{A \to B}(n) = \frac{t_{A \leftrightarrow B}(n)}{2} \quad (7.4)$$

Throughput: Também conhecido por *bandwidth*, esta grandeza é extensamente utilizada entre fabricantes de *hardware* por sua simplicidade, e representa a taxa na qual rotinas de comunicação podem transferir mensagens em um intervalo de tempo definido (geralmente, este valor é medido em Mbits/s).

Para mensagens de tamanho n *bytes*, o *throughput* (ρ) obtido pode ser calculado a partir da latência ponto-a-ponto pela seguinte fórmula:

$$\rho(n) = \frac{n}{t_{A \to B}(n)} \quad (\text{bytes/s}) \tag{7.5}$$

A fim de fornecer resultados em Mbits/s, modifica-se a equação 7.5 para

$$\rho(n) = \frac{8 * n}{t_{A \to B}(n) * 10^6} \quad (\text{Mbits/s}) \tag{7.6}$$

Nesse caso, representa a quantidade de *Megabits* que podem ser transferidos em 1 segundo utilizando mensagens de tamanho *n bytes*.

O *throughput* alcançado a nível de aplicação é limitado pelo *throughput* de pico do meio de interconexão dos computadores. No entanto, seu valor prático é menor, em virtude das sobrecargas de *software* e de congestionamentos na rede [NUP94].

7.3.1. Tamanho das Mensagens

Um aspecto muito importante a ser considerado quando da análise de uma métrica de desempenho, é o tamanho das mensagens utilizadas. Mensagens de tamanhos diferentes podem apresentar diferenças significativas de desempenho.

Por exemplo, toda mensagem, ao ser transmitida, carrega consigo um determinado número de *bytes* adicionais que são necessários para o controle interno das rotinas da biblioteca de troca de mensagens. E dentre esses *bytes*, podem ser incluídos o envelope da mensagem (estrutura de identificação da mensagem, que contém, por exemplo, o endereço do transmissor, *tag*, etc.), caracteres de controle, entre outros. A mais baixo nível, inclui-se ainda a sobrecarga imposta pelo protocolo de comunicação. Essa sobrecarga de *bytes* transmitidos junto a mensagem impõe um tempo de transmissão adicional significativo, dependendo do tamanho da mensagem.

Este tempo adicional pode ser calculado através da transmissão de uma mensagem vazia, e é fácil perceber que seu efeito será significativo para mensagens pequenas e não para mensagens grandes (nesse caso, a quantidade adicional de *bytes* de controle acaba sendo desprezível em relação a mensagem completa).

Colocando em outros termos, o tempo de transmissão de mensagens pequenas geralmente é dominado pela sobrecarga inicial da biblioteca de comunicação (t_{start} na equação 7.2) e pelas chamadas sobrecargas de *software*, enquanto que, para mensagens grandes, essa influência é diminuída e tem-se maior influência do tempo de transmissão da mensagem (t_{trans} na equação 7.2). Por isso, transmitir uma mensagem grande é mais eficiente que transmitir uma pequena [NUP94].

Assim, é importante separar as mensagens em grupos de mensagens grandes e pequenas. Exemplos da diferenças de desempenhos entre diferentes tamanhos de mensagens são discutidos na seção 8.3.1.

7.4. *Benchmarks* Considerados

O cálculo das três métricas de desempenho apresentadas acima é efetivado através da execução de dois *benchmarks*, que calculam o tempo de transmissão de uma mensagem de duas maneiras diferentes [DIL95] [NEV96] [NUP94].

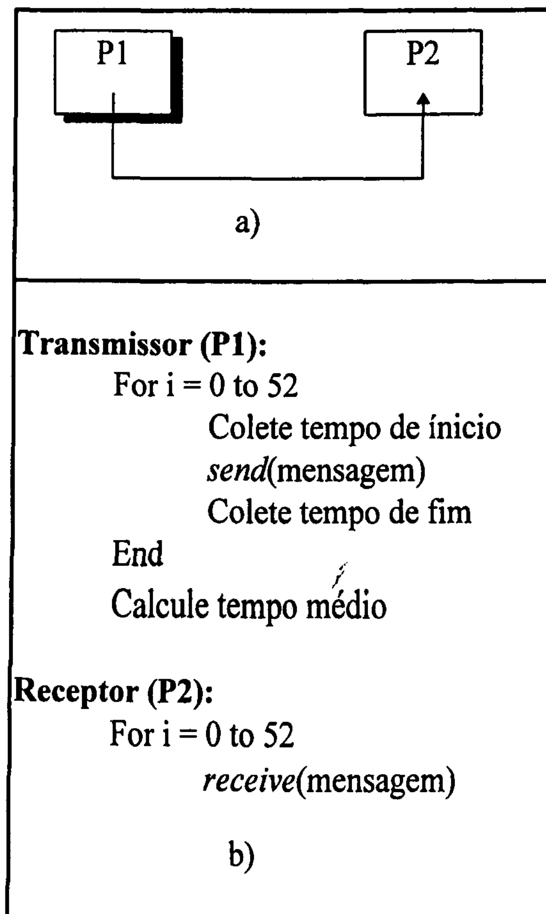

Figura 7.2 - a) Modelo de Comunicação *Ping* b) Algoritmo Simples.

Para oferecer medidas de tempo mais precisas, cada coleta de tempo é feita diversas vezes sendo considerado como valor final a média dos tempos coletados. Apesar de considerar apenas os tempos médios para a avaliação apresentada neste trabalho, todos os valores de tempos obtidos são armazenados a fim de possibilitar que sejam analisados estatisticamente.

Benchmark Ping: Este *benchmark* implementa simplesmente uma troca de mensagem entre dois processos (P1 e P2), de maneira que se colete o tempo gasto apenas no transmissor. O modelo de comunicação é apresentado na figura 7.2., juntamente como um algoritmo que o implemente.

Através da execução deste *benchmark*, pode-se obter o tempo de latência da biblioteca.

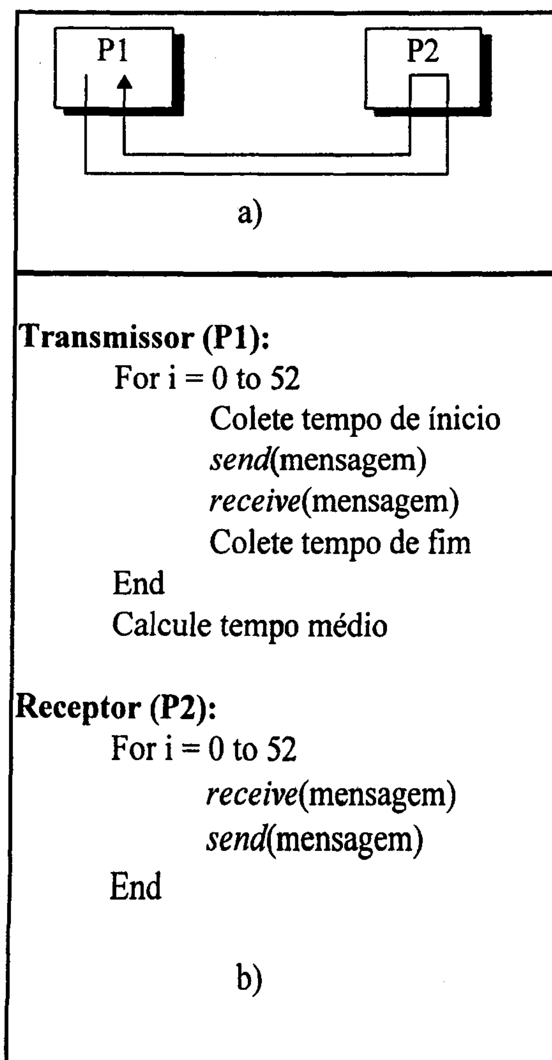

Figura 7.3 - a) Modelo de Comunicação *Ping-Pong* b) Algoritmo Simples.

Benchmark Ping-Pong: Sua função é calcular o tempo de *round-trip* (ver seção 7.3), possibilitando assim medir a latência de comunicação ponto-a-ponto

através da aplicação da equação 7.5, e também o *throughput* (equação 7.6). Na figura 7.3. é apresentado uma esquematização do modelo de sua comunicação e de seu algoritmo.

7.5. Algoritmo Paralelo de Jacobi

Equações diferenciais parciais possuem um papel primordial na física matemática, pois a partir delas são formulados modelos matemáticos para fenômenos físicos(ver figura 7.4). Pode-se obter o resultado de uma equação diferencial parcial através de sua discretização (conseguida através da aplicação de determinados métodos matemáticos), que resulta em um sistema linear ao qual podem ser aplicados métodos numéricos de solução [ORT85] [VIC81].

O **método de Jacobi** é um método de solução iterativo (uma tradução livre do original *iterative solver*) para o sistema linear (que é representado por uma matriz Jacobiana) resultante da discretização da equação diferencial parcial. Um método iterativo implica na aplicação sucessiva de determinados passos até que a solução convirja para o resultado esperado.

No método de Jacobi, cada ponto $x_{i,j}$ (excetuando os pontos localizados nas fronteiras da matriz, que mantêm-se constantes por não possuírem todos os vizinhos) é substituído a cada iteração pela seguinte equação [ORT85] [QUI94] [VIC81]:

$$x_{i,j} = \frac{x_{i-1,j} + x_{i,j-1} + x_{i+1,j} + x_{i,j+1}}{4} \qquad (7.7)$$

Um exemplo da aplicação do método de Jacobi é na solução do problema de distribuição de temperatura em estado estável em duas dimensões. Neste problema, uma estrutura quadrada e fina de aço é cercado em três de seus lados por vapor comprimido (a 100° C) e no quarto lado é banhado por gelo (a 0° C). Deve-se então determinar a distribuição de temperatura em 100 pontos igualmente distribuídos pela sua superfície, conforme mostra a figura 7.4. [QUI94].

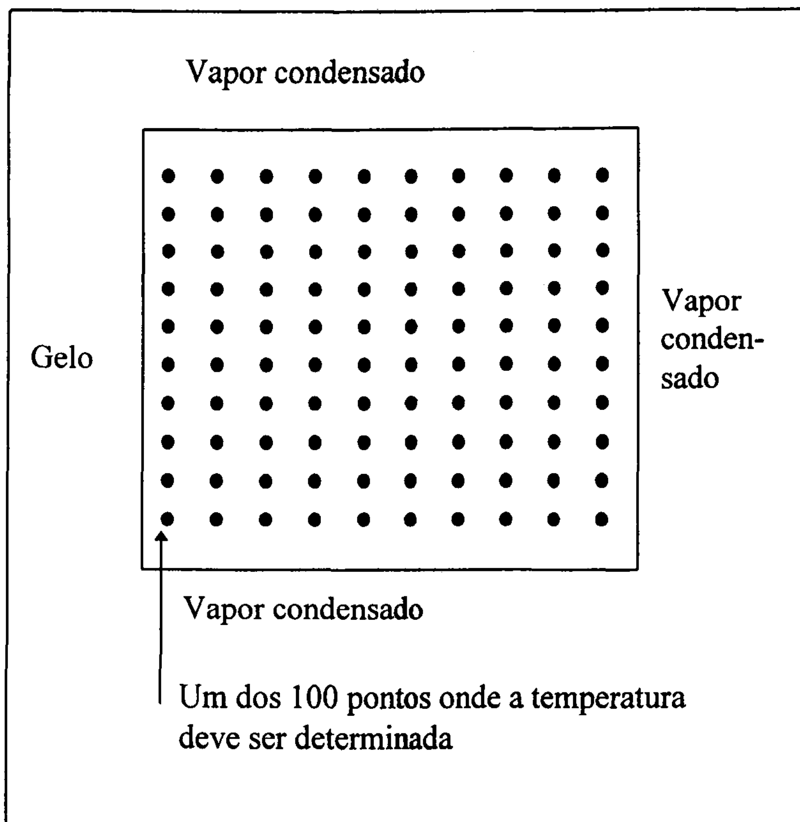

Figura 7.4 - Problema de Distribuição de Temperatura.

Uma organização paralela possível para o **algoritmo de Jacobi** (que é o algoritmo paralelo que implementa o método de Jacobi) é a divisão da matriz Jacobiana em várias sub-matrizes (chamadas blocos) de tamanho semelhante, sendo cada uma delas distribuída para um processador. Essa distribuição pode ser feita dividindo a matriz Jacobiana em blocos de uma dimensão (1 D) ou duas dimensões (2 D), como mostra a figura 7.5. O algoritmo implementado nesse trabalho, para manter a simplicidade, obedece a divisão em colunas (2 D) [QUI94] [SNI96].

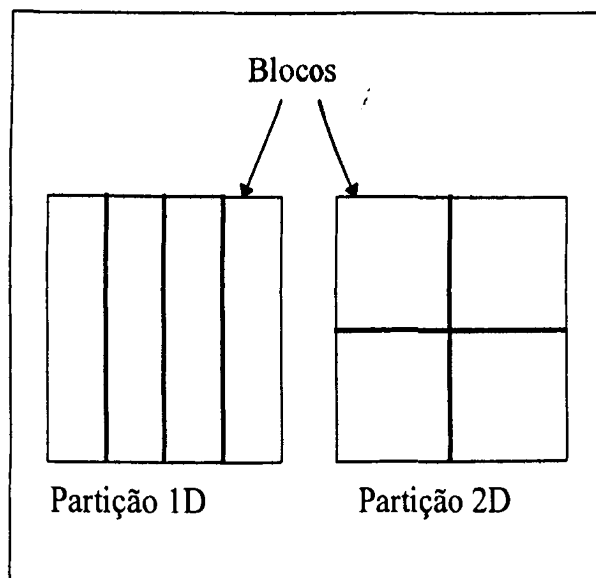

Figura 7.5 - Particionamento de uma Matriz em Blocos.

Para a execução do algoritmo de Jacobi é necessário que haja comunicação nas fronteiras de cada bloco, pois os elementos localizados nas posições de fronteira necessitam de elementos vizinhos que estão sendo calculados em outros processadores.

Suponha, por exemplo, que se divida a matriz Jacobiana em **n** blocos, distribuídos respectivamente para os processos 0,1,2,..., **n**-1. Então a cada iteração, um processo **x** $(0 < x < n-1)$ necessita enviar seus elementos de fronteira para os processos **x-1** e **x+1** e também receber desses mesmos processos os seus elementos de fronteira. Então, a cada iteração, cada processo executa duas rotinas *send()* e duas rotinas *receive()*. A exceção nesse caso são os processos que possuam a primeira e última coluna da matriz Jacobiana (processos 0 e **n**-1).

Um algoritmo SPMD (que executa em todos os processadores) simplificado é apresentado na figura 7.6.

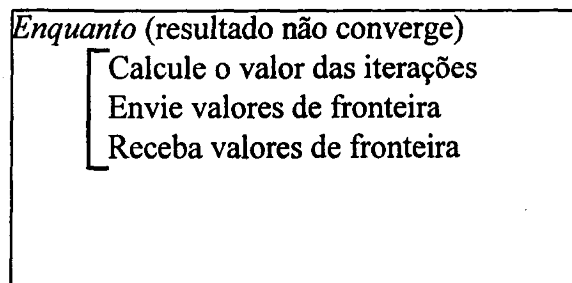

Figura 7.6 - Algoritmo Simplificado de Jacobi.

Se todos os processos enviam mensagens no mesmo momento dentro do algoritmo (após o cálculo da iteração), existe o risco de *deadlock*. Maneiras possíveis de se evitar *dealock*, nesse caso, são:

- Construção de um algoritmo seguro onde, por exemplo, processos de *rank* par primeiro enviam e depois recebem, e processos de *rank* ímpar fazem o contrário (vermelho/preto);
- Utilização de rotinas no modo de comunicação *bufferizado*;
- Utilização de rotinas não bloqueantes;
- Utilização de rotinas de comunicação em duas vias.

Rotinas não bloqueantes além de evitarem o risco de *deadlock*, possibilitam também um possível ganho de desempenho, pois, este algoritmo requer grande quantidade de comunicação. (A cada iteração, mensagens são enviadas por todos os processos).

7.6. Considerações Finais

Neste capítulo, foram discutidos os *benchmarks* a serem utilizados para a avaliação de desempenho a ser apresentada no próximo capítulo. Esses *benchmarks* fornecem informações que possibilitam estudar valores de desempenho específicos, ou seja, que calculam as sobrecargas e latências de uma operação de comunicação.

Embora o desempenho de uma operação de comunicação seja essencial a uma aplicação paralela, outros fatores também influem para modificar a eficiência desta aplicação. Por isso, é interessante investigar a relação dos resultados dos *benchmarks* com uma aplicação paralela real. Nesse trabalho, foi escolhido o algoritmo de Jacobi.

É importante ressaltar, porém, que o algoritmo de Jacobi é apenas um exemplo de aplicação, e as sobrecargas de comunicação podem possuir diferentes efeitos de acordo com a organização de uma aplicação.

Capítulo 8

Avaliação de Desempenho

Este capítulo apresenta os resultados obtidos da aplicação dos *benchmarks* para as rotinas de comunicação ponto-a-ponto e para as implementações e também da execução do algoritmo de Jacobi, e as conclusões obtidas desses resultados.

8.1. Introdução

O Fórum MPI, em abril de 1994, publicou a especificação do padrão MPI, onde estão explicadas, em termos de sintaxe e semântica, todas as rotinas constituintes deste padrão para plataformas de portabilidade. Dessa especificação, constam os objetivos de cada uma dessas rotinas, além de explicações sobre a sua utilização em programas paralelos [DOS96].

A necessidade de ser um padrão multi-plataforma, sem perder a eficiência, levou o MPI a ter um grande número de rotinas, sendo a maior parte delas variações de uma determinada função. Como exemplo, podem ser citadas as rotinas de comunicação ponto-a-ponto discutidas no capítulo 6 [SNI96] [DON95].

Porém, nem todos ambientes computacionais (*hardware* e *sofware*) oferecem o suporte necessário para que essas rotinas ofereçam as vantagens planejadas. Além disso, nem todas as implementações MPI são necessariamente eficientes, uma vez que a especificação do MPI não define detalhes de implementação de suas rotinas.

Este capítulo apresenta os resultados da avaliação das rotinas ponto-a-ponto do MPI, a fim de apontar seus pontos positivos e o custo de sua utilização. Essa avaliação é feita através da análise das métricas de desempenho definidas no capítulo anterior e suas conclusões devem ser restritas ao ambiente computacional definido neste trabalho (rede LINUX). Isto é, as conclusões obtidas aqui podem não ser verdadeiras em outros ambientes correlatos, como, por exemplo, uma rede de *workstations* SUN.

Na primeira parte deste capítulo (seção 8.2) são apresentados alguns requisitos mais gerais relacionados a parte prática da avaliação, a fim de explicitar informações a respeito de configurações e estruturação de *benchmarks*, entre outras.

Em seguida (seção 8.3), é apresentada a avaliação de desempenho das rotinas de comunicação ponto-a-ponto em todas as suas variantes, discutindo cada uma delas a nível de custo/benefício.

Na seção 8.4, são discutidos os resultados obtidos para as implementações MPI e para o PVM e na seção 8.5 são avaliados os resultados do algoritmo de Jacobi.

Finalmente, na seção 8.6. são resumidas as principais conclusões obtidas neste capítulo.

8.2. Requisitos Gerais da Avaliação

Nesta seção, são apresentados alguns tópicos específicos relacionados a características importantes de configuração do *hardware* e do *software* utilizado neste trabalho, além de tópicos relacionados a representação gráfica e análise estatística dos resultados obtidos.

8.2.1. Configuração de *Hardware*

A rede de computadores na qual este trabalho foi desenvolvido possui a seguinte configuração de *hardware*:

- 2 pc's 486 DX2 / 66 *MegaHertz*;
- 1 pc *Pentium* / 100 *MegaHertz*;
- 1 pc 486 DX4 / 100 *MegaHertz*.

Os *benchmarks* (*ping* e *ping-pong*) foram executados sobre os dois PC's 486 DX2, de maneira que se mantivesse a homogeneidade dos computadores envolvidos. Os dois computadores restantes foram incluídos para a execução do algoritmo de Jacobi.

É importante ressaltar que, quando procura-se analisar uma métrica de desempenho em termos quantitativos e não comparativos, deve-se levar em conta o poder computacional dos computadores nos quais foram executados os *benchmarks*, principalmente quando procura-se compará-los com resultados obtidos em ambientes computacionais semelhantes. Por exemplo, a nível de *throughput*, a latência de biblioteca tem um peso considerável que depende da rapidez com que o processador processa os dados a serem transmitidos. Nesse caso, computadores mais rápidos podem oferecer resultados mais expressivos e melhor desempenho.

8.2.2. Configuração das Plataformas de Portabilidade

As implementações de plataformas de portabilidade utilizadas neste trabalho foram compiladas e instaladas utilizando opções padrão descritas nos respectivos manuais de instalação. Determinadas implementações, como o MPICH, fornecem diversas opções de compilação para usuários experientes que podem até fornecer desempenho diferenciado. A intenção deste trabalho é avaliar as implementações em condições reais de utilização pelo usuário comum e, por isso, foram evitadas opções de configuração avançadas [BEG94] [BUR95] [CHE94] [GRO96b] [GRO96b].

8.2.3. Configuração dos *Benchmarks*

A coleta de tempos em um sistema LINUX apresenta diversos problemas pelo fato dele ser um sistema operacional multi-tarefa. O tempo coletado é o chamado *wall time* (conseguido pela utilização da rotina ***gettimeofday()***), que representa o tempo decorrido total e não o tempo decorrido durante a execução efetiva do processo na CPU (é importante notar que vários processos *daemons* concorrem pela CPU). Ignorar esses processos concorrentes que influem nos tempos finais não é viável, uma vez que se pretende obter resultados que sejam o mais próximo possível da utilização real do sistema [NEV96].

Além disso, deve ser destacado a pouca granularidade de *clock* de um sistema LINUX. Valores muito pequenos de tempo fatalmente acabam sendo influenciados pela próprio mecanismo de acesso ao tempo do sistema. Uma maneira de diminuir o efeito deste problema é através da obtenção dos tempos várias vezes, considerando como resultado final o tempo médio. Esse procedimento foi utilizado nesse trabalho [DIL95].

Quanto ao tamanho das mensagens, são consideradas pequenas as mensagens cujos tamanhos concentram-se na faixa entre 0 e 1.000 *bytes* e são consideradas mensagens grandes as de tamanho entre 20.000 e 100.000 *bytes*. Estes valores foram escolhidos baseando-se em trabalhos correlatos da área [DIL95] [DOU93] [NEV96] [NUP94] e foram considerados satisfatórios quando analisados após a execução dos experimentos. Foram também consideradas mensagens de tamanhos maiores, entre 200 *kilobytes* e 1 *Megabyte*. As conclusões obtidas para estas mensagens seguiram o padrão obedecido pelas mensagens grandes.

Outro requisito importante é o tipo de dado a ser transmitido nas mensagens enviadas pelos *benchmarks*. O tipo escolhido, presente em todas as implementações, foi o tipo ***byte*** (definido como MPI_Byte pelo Fórum MPI

[SNI96]), que representa um sequência de oito *bits* sem um significado preestabelecido. Este tipo é interpretado da mesma maneira pelo MPI e pelo PVM.

As mensagens são transmitidas sem a utilização de mecanismos especiais fornecidos pelas implementações, o que implica na não utilização de comunicação via *daemons*, de mecanismos de garantia de entrega de mensagens e de tratamento de heterogeneidade, entre outros.

As colocações desta seção se aplicam apenas em parte a execução do algoritmo de Jacobi, pois, nesse caso, o tamanho e o tipo das mensagens foi escolhido de acordo com as características deste algoritmo.

8.2.4. Apresentação Gráfica

Para a representação gráfica das métricas de desempenho, foi escolhida a utilização de gráficos de linha, pois permitem a visualização clara das diferenças entre os resultados obtidos.

Os gráficos de linha aqui utilizados são construídos da seguinte maneira: um conjunto de pontos (x,y) são conhecidos e uma linha é traçada por sobre esses pontos, construída através da ligação dos pontos vizinhos. Como os valores das métricas de desempenho se comportam regularmente (ver seção 8.3.1), pode-se afirmar que a reta gerada é uma aproximação muito boa da reta construída a partir do cálculo de todos os valores de x.

Para os gráficos das latências de biblioteca e ponto-a-ponto, o eixo x representa os valores de tamanhos das mensagem (representados em *bytes*) e o eixo y representa as respectivas latências (consideradas em segundos). Os gráficos de *throughput* possuem estrutura semelhante, excetuando no eixo y, onde é utilizada a unidade Mbits/s (Megabits por segundo).

É importante ressaltar que as diferenças de escala do eixo y e x quando da representação de mensagens grandes e pequenas são significativas. Por exemplo, para mensagens pequenas, os valores do eixo x são separados por 200 bytes, enquanto que, para mensagens grandes, os valores são separados por 20.000 bytes.

8.2.5. Análise Estatística

Todos os resultados obtidos por este trabalho foram analisados estatisticamente através do método de comparação de dois tratamentos, apresentado em Achcar [ACH95] e Soares [SOA91]. Uma revisão sobre a estrutura deste método e também as tabelas contendo os resultados estatísticos obtidos são apresentados no Apêndice A.

Toda e qualquer avaliação ou conclusão contida neste trabalho levou em conta o estudo estatístico dos dados sobre os quais elas se referiam.

8.3. Comunicação Ponto-a-Ponto

Essa seção apresenta a avaliação de desempenho das rotinas de comunicação ponto-a-ponto do MPI e é organizada da seguinte maneira:

- Rotinas bloqueantes (seção 8.3.1);
- Rotinas não bloqueantes (seção 8.3.2);
- Rotinas de comunicação nos dois sentidos (seção 8.3.3);
- Requisições persistentes (seção 8.3.4).

Para cada um dos grupos citados acima são definidos requisitos de análise e métricas de desempenho que forneçam resultados conclusivos. Além disso, quando possível, os dados são apresentados graficamente, a fim de apresentar visualmente as diferenças que merecerem destaque.

Todos os resultados mostrados nesta seção foram obtidos nas duas implementações completas do MPI: LAM e MPICH. Só serão destacados os resultados obtidos nas duas implementações quando estes forem diferentes, ou seja, possibilitarem conclusões diferentes.

O UNIFY não é utilizado nesta análise por ser uma implementação incompleta, que apenas define rotinas de transmissão e recepção de mensagens no modo padrão (bloqueantes ou não bloqueantes).

8.3.1. Rotinas Bloqueantes

Qual a relação entre os modos de comunicação para rotinas bloqueantes? Quais rotinas realmente oferecem ganho de desempenho? Esta seção tem como principal objetivo apresentar respostas objetivas para as duas perguntas acima, baseando-se em fatos comprovados pela prática e analisando até que ponto as implementações MPI conseguiram manter-se fiéis aos objetivos propostos pelo Fórum MPI [SNI96].

Após a análise de todas as métricas de desempenho aplicadas para as rotinas bloqueantes, foi escolhido utilizar os valores de latência de biblioteca e *throughput* para avaliar as diferenças entre os modos de comunicação. Não são considerados os valores de latência ponto-a-ponto por dois motivos:

- Os tempos de latência ponto-a-ponto apresentam um comportamento similar quando comparados em relação aos valores de latência da biblioteca, isto é, permitem conclusões semelhantes;
- O *throughput*, calculado através da aplicação da equação 7.6 aos valores de latência ponto-a-ponto, apresenta resultados mais interessantes a nível de análise gráfica. Em virtude da utilização de uma unidade (Mbits/s) mais precisa, as diferenças entre os diversos valores são melhor explicitadas .

Antes de apresentar os resultados da comparação entre os modos de comunicação, é interessante analisar separadamente o comportamento dos valores de latência de biblioteca e *throughput*, a fim de compreender melhor o significado de cada uma dessas métricas de desempenho.

Latência da Biblioteca: A relação de grandezas entre a latência da biblioteca e a latência ponto-a-ponto são distintas quando da transmissão de mensagens pequenas e grandes.

Os gráficos 8.1 e 8.2 ilustram claramente essa situação. O gráfico 8.1 apresenta as latência de biblioteca e ponto-a-ponto comparadas para as rotinas de transmissão padrão de mensagens pequenas, enquanto o gráfico 8.2 apresenta esta mesma relação para mensagens grandes. Ambas as métricas de desempenho foram calculadas no LAM e esta relação manteve-se constante em todas as bibliotecas testadas.

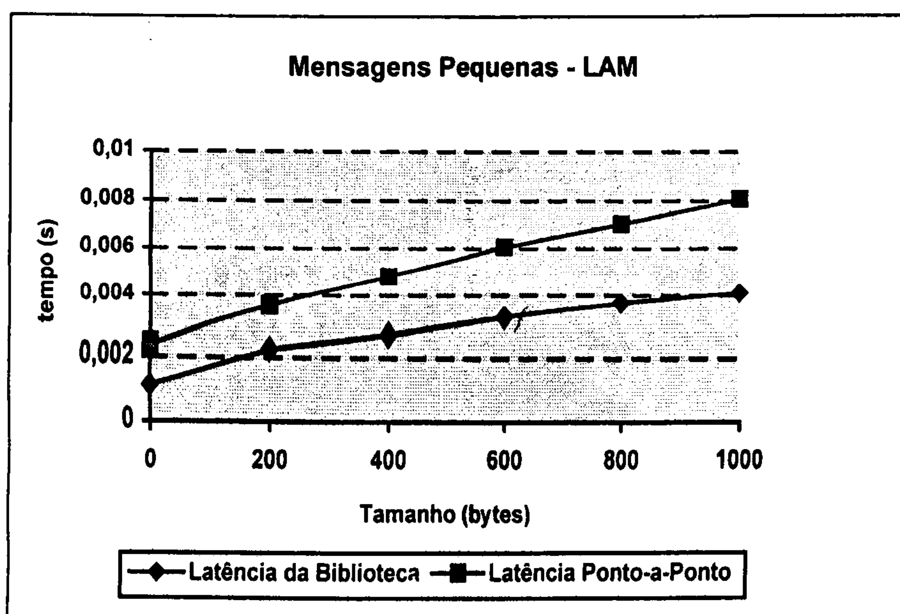

Gráfico 8.1 - Latências da Biblioteca e Ponto-a-Ponto - Mensagens pequenas

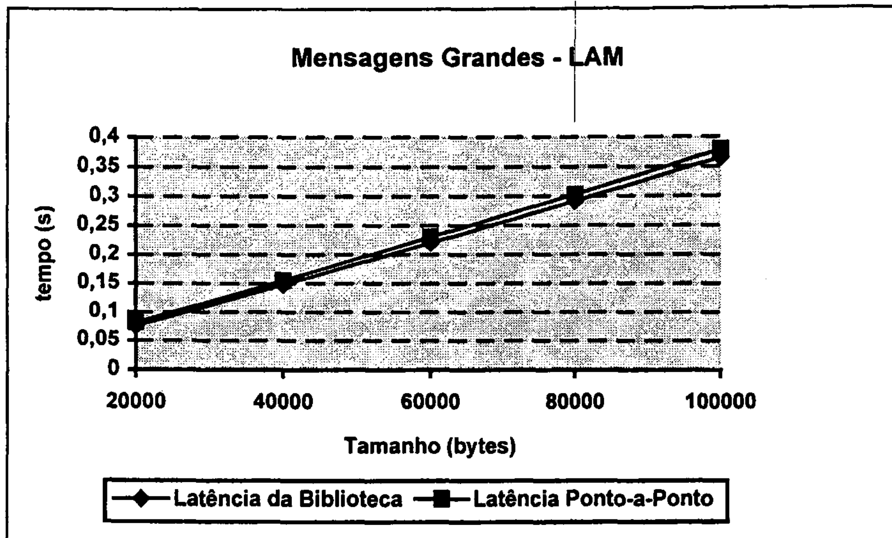

Gráfico 8.2 - Latências da Biblioteca e Ponto-a-Ponto - Mensagens Grandes

Os gráficos apresentam resultados para rotinas de transmissão padrão. Os outros modos de comunicação apresentam resultados similares, excetuando-se o modo síncrono, pois uma rotina de transmissão síncrona apenas retorna após o recebimento completo da mensagem, de maneira que os dois tempos serão parecidos para quaisquer tamanhos de mensagens.

Para mensagens pequenas, como mostrado na seção 7.3.1, a sobrecarga constante para todos os tamanhos de mensagens possui influência significativa, diminuindo a produtividade. Em contrapartida, essa influência diminui para mensagens grandes, e os tempos de latência ponto-a-ponto se aproximam do tempo de latência da biblioteca.

Neste ponto, é interessante acrescentar um comentário a respeito da teoria apresentada na seção 7.2, relacionada a equação 6.2, que representa uma transferência de mensagem em termos da soma de três latências (de transmissão, recepção e rede). Esses três valores, na prática, ocorrem em tempos relativamente sobrepostos. Isto significa que a mensagem é transmitida à medida em que é recebida. Apesar de sua formulação não ser errada (realmente existem esses três tempos), eles não devem ser simplesmente somados. Essa constatação não invalida os resultados subsequentes derivados desta equação, visto que o tempo de transferência de uma mensagem é sempre considerado como um tempo inteiro, não sendo desmembrado em seus valores constituintes.

A latência de biblioteca cresce de maneira proporcional à medida que se aumenta o número de *bytes* de uma mensagem. Quanto maior a mensagem, maior será o tempo de processamento interno das bibliotecas e dos protocolos que a preparam para a transmissão. Essa situação é mostrada no gráfico 8.3, que foi construído utilizando todos os tempos obtidos para mensagens de até

1.000.000 *bytes*. Houve a necessidade de se ajustar a escala do eixo *x*, o que gera uma grande concentração de pontos na parte inferior do gráfico. São apresentados os resultados colhidos no LAM, utilizando rotinas de comunicação padrão.

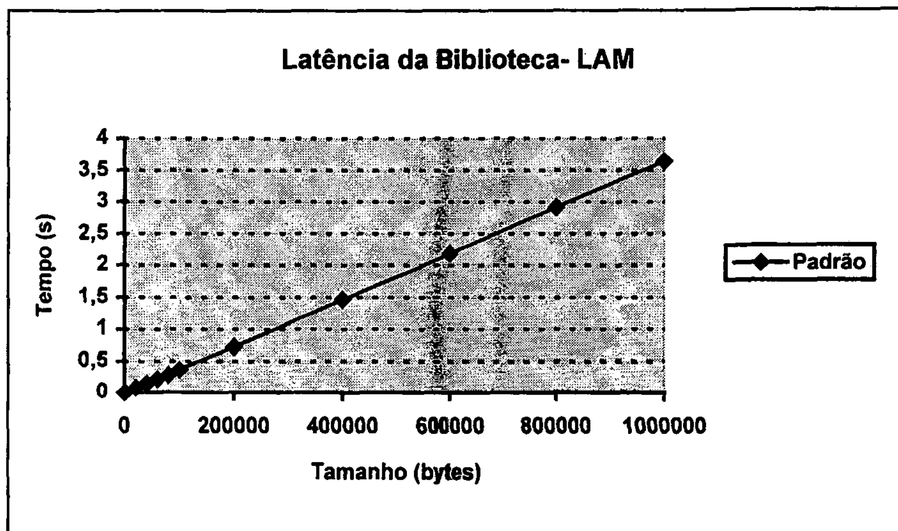

Gráfico 8.3 - Latência da Biblioteca no LAM

Throughput: O *throughput* oferecido por uma biblioteca de passagem de mensagem é determinado de acordo com o tamanho das mensagens que são transmitidas. Na teoria, quanto maior a mensagem maior o *throughput*, pois as sobrecargas constantes para todas as mensagens diminuem sua eficiência (seção 7.3.1).

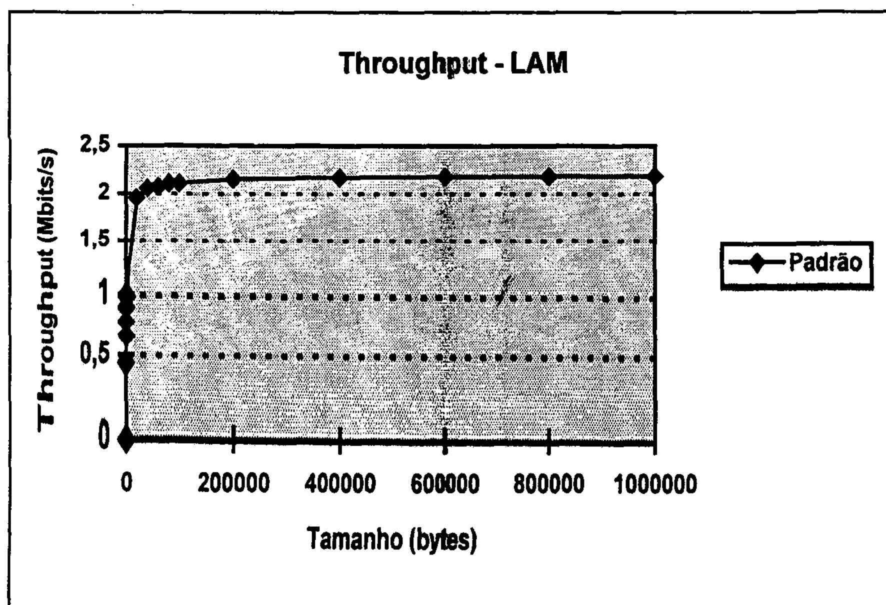

Gráfico 8.4 - *Throughput* no LAM

Os resultados obtidos na implementação LAM para rotinas de transmissão no modo padrão são apresentados no gráfico 8.4. Nele, pode-se

perceber que, até determinado limite, quanto maiores as mensagens transmitidas, melhor é o *throughput*. A partir de um certo tamanho de mensagem alcança-se o limite superior do *throughput*, valor a partir do qual as diferenças de tamanho de mensagens já não mais influem no seu desempenho.

É importante ressaltar que o valor de *throughput* depende da latência de biblioteca, de maneira que os computadores com maior poder de processamento oferecem melhores valores de *throughput*. Porém, as características do gráfico 8.4. deverão se manter constantes para qualquer ambiente correlato.

Comparação Entre os Modos de Comunicação: Nos gráficos 8.5 e 8.6 são apresentados os resultados de *throughput* obtidos para mensagens pequenas e grandes no MPICH. Esses mesmos resultados são apresentados para o LAM nos gráficos 8.7 e 8.8.

Analisando os resultados obtidos, conclui-se que as rotinas de modo padrão, como é determinado pelo Fórum MPI, apresentam os melhores desempenhos em ambas as implementações, tanto para mensagens pequenas como grandes. Em alguns pontos, não há praticamente diferença entre os modos de comunicação, mas, quando considerados os resultados como um todo, pode-se afirmar com certeza que o modo padrão é o mais eficiente.

Além disso, como é aconselhado pelo Fórum MPI, as rotinas padrão utilizam-se de uma semântica *bufferizada* para ambas as implementações (ver seção 6.2.2).

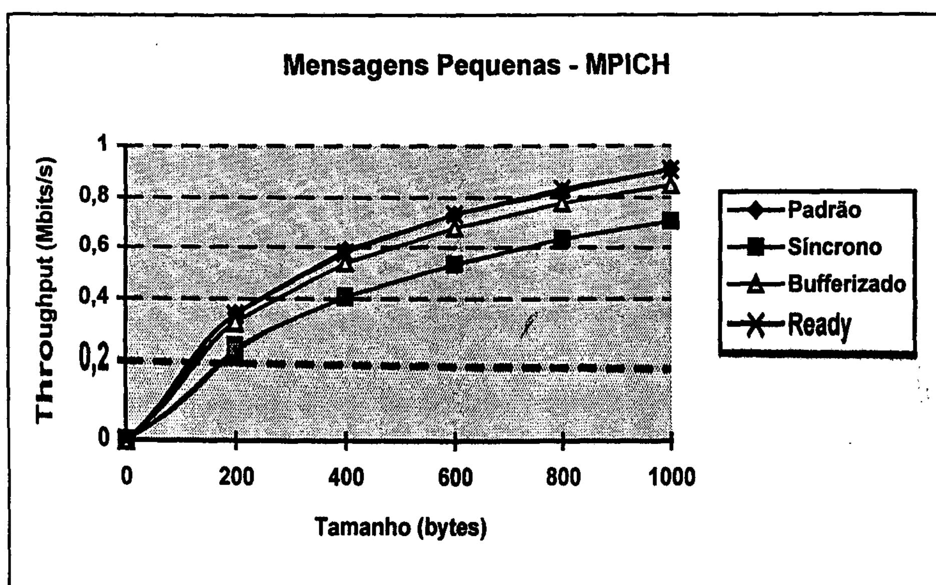

Gráfico 8.5 - *Throughput* para Mensagens Pequenas no MPICH

As Rotinas síncronas apresentam os piores resultados para mensagens pequenas, tanto a nível de latência de biblioteca quanto a nível de *throughput*. Para mensagens grandes, esta diferença diminui, pois a sobrecarga adicional de

software pouco influi para mensagens grandes. No LAM, a diferença em relação ao modo padrão diminui a ponto de em alguns tamanhos de mensagens não se constatar diferença estatística. Já no MPICH, a diferença continua perceptível, apesar de menor.

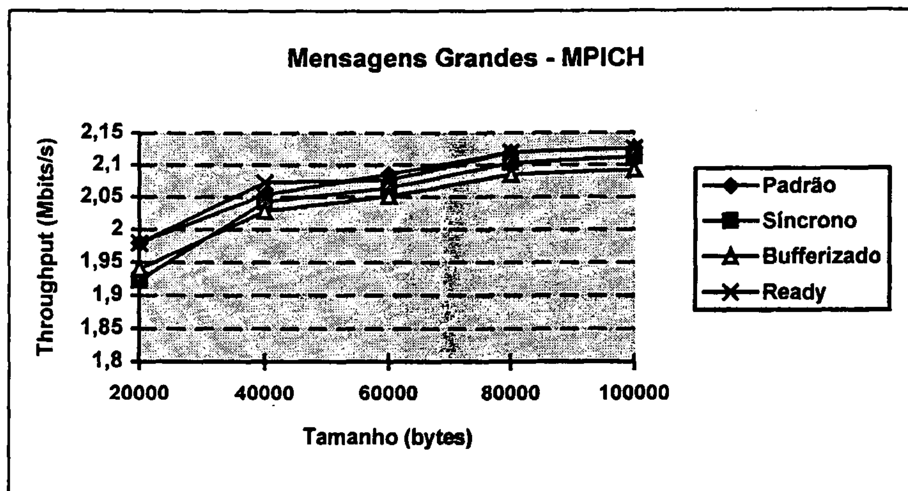

Gráfico 8.6 - *Throughput* para Mensagens Grandes no MPICH

O modo *bufferizado* apresentou os resultados mais curiosos. Para mensagens pequenas, no MPICH, as rotinas *bufferizadas* são apenas mais eficientes que as síncronas, enquanto que no LAM, mantêm-se no mesmo nível de desempenho das rotinas padrão. Para mensagens grandes, obteve-se o pior desempenho em ambas as implementações. Percebe-se que, a medida que as sobrecargas de *software* diminuem sua influência, a necessidade de cópias para *buffers* influenciam de maneira negativa o desempenho de uma operação de comunicação.

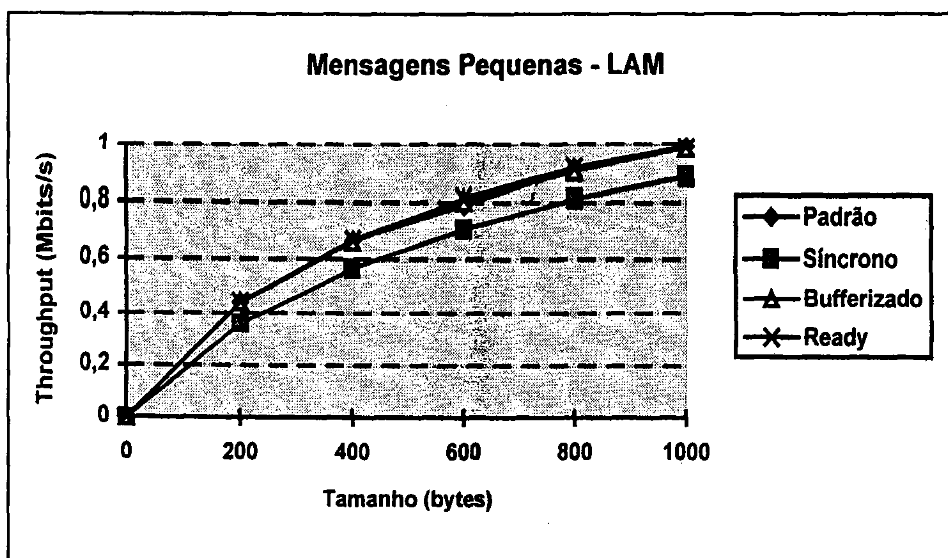

Gráfico 8.7 - *Throughput* para Mensagens Pequenas no LAM

O modo *ready*, apesar das pequenas diferenças apresentadas em relação aos tempos médios do modo padrão, não apresenta diferença estatística na grande maioria dos tempos colhidos em todas métricas de desempenho. Isso leva a conclusão de que o modo *ready* é, como sugerido pelo fórum MPI, implementado pelo modo padrão. As rotinas no modo *ready* foram criadas para aproveitar características intrínsecas de determinados computadores paralelos, mas, para sistemas distribuídos, sua influência demonstra ser nula.

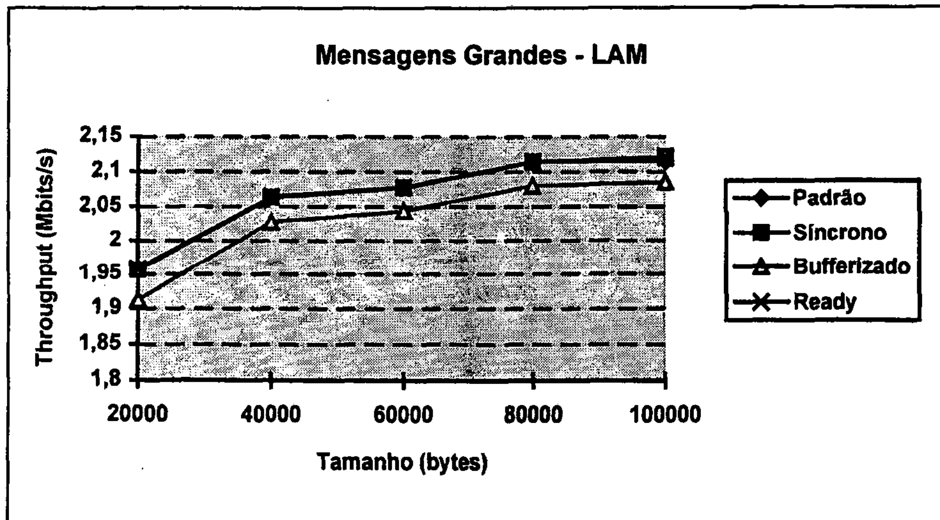

Gráfico 8.8 - *Throughput* para Mensagens Grandes no LAM

A nível de latência de biblioteca, os resultados obtidos possibilitaram conclusões semelhantes aos resultados de *throughput*, o que era esperado, já que os valores dessas duas métricas de desempenho são fortemente inter-relacionados. A única exceção a ser citada são os valores de latência de biblioteca para mensagens grandes no LAM.

Rotinas de transmissão de mensagens grandes no modo *bufferizado*, no LAM, apresentam os menores tempos de latência da biblioteca, embora forneçam os menores *throughputs*. Isso acontece em virtude da organização de uma rotina *bufferizada* no LAM, a qual segue as orientações do Fórum MPI [SNI96].

Uma mensagem, após ser colocada nos *buffers* criados pelo usuário, inicia uma rotina de transmissão não bloqueante e retorna o controle para o usuário imediatamente. Uma rotina não bloqueante possui uma latência de biblioteca pequena, perceptível principalmente quando são consideradas mensagens grandes. Porém, sua utilização não modifica os valores de *throughput*, já que o tempo de transmissão ponto-a-ponto não é modificado (veja seção 8.3.2).

Isso implica na possibilidade de cópia de mensagens para os *buffers* numa velocidade maior do que a biblioteca consegue enviar, causando o esgotamento dos *buffers*. Esse fato foi constatado na prática no LAM, que

constantemente apresentava erros de falta de espaço em *buffer*. É importante ressaltar, porém, que este comportamento está correto segundo o Fórum MPI, isto é, não deve haver bloqueio das rotinas de comunicação em caso de esgotamento de *buffers*.

O MPICH, por outro lado, não define rotinas não bloqueantes e, por isso, não apresenta diminuição da latência da biblioteca. Além disso, o MPICH utiliza um protocolo de comunicação definido pelos seus próprios criadores como *eager* (impetuoso), cuja característica principal é manter um ritmo forte de entrega de mensagens. Isto significa que uma mensagem tem que ser enviada o mais rápido possível a partir do momento em que estiver disponível. Isso acarreta uma instabilidade maior nos tempos coletados para o MPICH, quando comparado ao LAM e ao PVM. Essa característica é percebida no modo *bufferizado* também, visto que não houve problemas de esgotamento de *buffers* durante os testes práticos.

Resumindo as informações apresentadas nesta seção sobre o ponto de vista do tamanho das mensagens, pode-se afirmar que, para mensagens pequenas e grandes, o modo padrão apresenta o melhor desempenho. Rotinas síncronas, que apresentam uma sobrecarga adicional em virtude das confirmações de recepção das mensagens, apresentam os piores resultados para mensagens pequenas. Essa sobrecarga pouco influi para mensagens grandes, onde a necessidade de cópia das mensagens para *buffers* explicitamente criados pelos usuários oferece maior sobrecarga e, portanto, piores resultados para o modo *bufferizado*.

8.3.2. Rotinas Não Bloqueantes

A principal questão relacionada à utilização de rotinas não bloqueantes é se é possível conseguir melhor desempenho na sobreposição entre comunicação e execução em um sistema multi-tarefa como o LINUX, executando em computadores pessoais sem nenhum tipo especial de mecanismo de *hardware* de comunicação que possibilite tal comunicação ser realmente efetivada paralelamente a execução.

Segundo Snir [SNI96], as rotinas de comunicação não bloqueante não foram desenvolvidas com esse tipo de ambiente distribuído em mente. *E-mails* enviados pelos autores desse trabalho para alguns *newsgroups* relacionados ao assunto não obtiveram resposta satisfatória. Partiu-se então para a experimentação prática.

Para perceber a existência ou não de rotinas não bloqueantes, foram analisados os valores da latência de biblioteca para as rotinas não bloqueantes de transmissão e recepção de mensagens, em todos os modos. Uma rotina não bloqueante deve retornar assim que seja iniciada a comunicação, obtendo uma latência mínima e realizando o resto da comunicação em paralelo.

No MPICH, as rotinas de transmissão não bloqueantes apresentam valores de latência de biblioteca idênticos aos das rotinas bloqueantes. Portanto, não existe sobreposição de execução e comunicação. Porém, experimentos realizados mostram que, apesar das rotinas bloquearem-se até que a comunicação tenha terminado, não houve problemas com *deadlocks*. E evitar *deadlocks* também é um objetivo de uma rotina não bloqueante.

O MPI fornece outras rotinas que possibilitam evitar *deadlocks* (como rotinas *bufferizadas* e de comunicação em duas vias). Assim, pode-se concluir que a mais importante característica de uma rotina não bloqueante é a possibilidade de execução e comunicação em paralelo.

No LAM, a situação se inverte. Ambas as rotinas de transmissão e recepção retornam logo que se iniciam, de maneira que as latências de biblioteca são mínimas e praticamente iguais para qualquer tamanho de mensagem. Logo, as rotinas são realmente não bloqueantes. A diferença de latência é claramente percebida para mensagens grandes, já que as mensagens pequenas possuem latências de biblioteca pequenas e não muito diferentes das latências de rotinas não bloqueantes, onde ambas são dominadas pela sobrecarga constante para todas as mensagens.

É importante ressaltar que a latência da biblioteca, como foi definido neste trabalho, não inclui o tempo de transmissão da mensagem quando a operação de transmissão ou recepção está sendo executada em segundo plano, que também não deixa de fazer parte dessa latência. Porém, seu cálculo exigiria modificações diretas no código fonte do LAM, o que não era objetivo do trabalho.

Descoberto a existência de rotinas não bloqueantes no LAM, resta definir se esse tipo de estrutura de comunicação oferece ganho real de desempenho. Avaliar isso utilizando apenas a latência de biblioteca como tópico de análise é insuficiente, pois, pelas próprias características de uma rotina não bloqueante, o principal ganho de desempenho neste caso resume-se na sobreposição entre execução e comunicação. Para avaliar essa característica, nenhuma das métricas de desempenho definidas é suficiente.

Para isso, é necessário a execução de uma aplicação real, na qual este tipo de comunicação possibilite ganho de desempenho. Assim, através da avaliação dos resultados obtidos, pode-se concluir as vantagens da utilização de rotinas não bloqueantes. Para isso, foi escolhido o algoritmo de Jacobi (seção 7.5), pois ele necessita de grande quantidade de comunicação, possibilitando uma análise realista.

Estes resultados são apresentados na seção 8.5.

8.3.3. Rotinas de Comunicação nos Dois Sentidos

Como foi exposto no capítulo 6, em virtude da grande utilização do modelo de comunicação *ping-pong* (onde um processo transmite e recebe uma mensagem em operações consecutivas), o MPI fornece duas estruturas de comunicação alternativas que incluem as operações *send()* e *receive()* na mesma rotina, possibilitando **(MPI_Sendrecv_Replace)** ou não **(MPI_Sendrecv)** a sobreposição das variáveis de recepção e transmissão [SNI96] [MAC96].

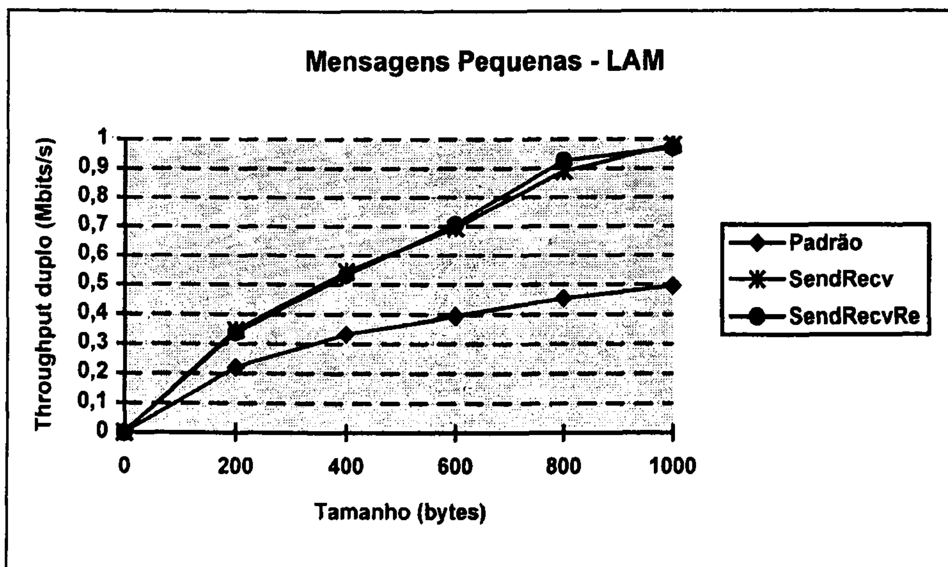

Gráfico 8.9 - *Throughput* Duplo Para Mensagens Pequenas no LAM

Além de tornar mais simples a escrita de algoritmos utilizando esta estrutura de comunicação, a utilização destas duas rotinas garante a não ocorrência de *deadlocks*. O objetivo desta seção é fornecer resultados que possibilitem concluir o quanto se perde ou ganha de desempenho pela utilização dessas duas rotinas.

Os gráficos 8.9 e 8.10 apresentam os valores de *throughput* duplo (por *throughput* duplo, entende-se a quantidade de dados transmitidos considerando-se o movimento de *round-trip*, ou seja, é considerado o tempo de transmissão de ida e volta da mensagem) para a estrutura de comunicação *ping-pong* utilizando as rotinas **MPI_Sendrecv** (SendRecv na legenda), **MPI_Sendrecv_replace** (SendRecvRe) e rotinas de transmissão e recepção de mensagens no modo padrão. A comparação é feita utilizando como ponto de referência as rotinas de comunicação no modo padrão pois estas são as mais eficientes entre todos os modos.

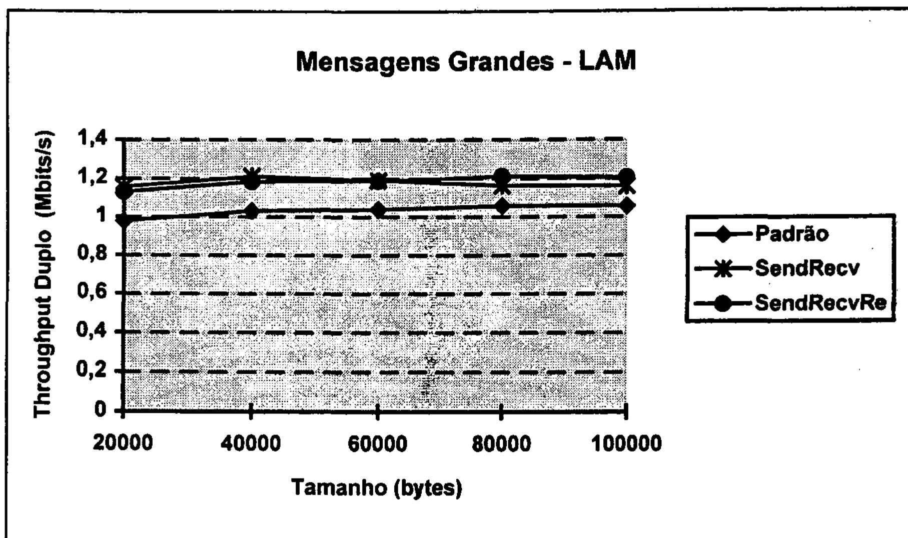

Gráfico 8.10 - *Throughput* Duplo Para Mensagens Grandes no LAM

Para mensagens pequenas, onde a sobrecarga inicial da biblioteca apresenta influência significativa, a execução simultânea das duas rotinas de comunicação possibilitou um aumento significativo de *throughput*, visto que a sobrecarga inicial pode ser executada concorrentemente nos dois computadores participantes da comunicação. Para mensagens grandes, como os valores de *throughput* simples (ver gráfico 8.4), o *throughput* duplo se estabiliza e as diferenças entres as duas rotinas de comunicação e as duas rotinas padrão diminuem, apesar de continuarem significativas (diminuem as sobrecargas de *software*).

Em relação a comparação entre as rotinas de comunicação em dois sentidos, para mensagens pequenas não existe diferença estatística entre elas e para mensagens grandes há uma pequena diferença, na prática pouco considerável.

8.3.4. Requisições Persistentes

Requisições persistentes permitem reduzir a sobrecarga inicial da transmissão de uma mensagem quando uma série de mensagens de mesmo formato são transmitidas. Assim, sua ação deve refletir na redução dos tempos de latência de biblioteca devido à diminuição da sobrecarga inicial de biblioteca [SNI96] [MAC96].

Segundo o Fórum MPI [SNI96], as requisições persistentes devem iniciar rotinas de comunicação não bloqueante, fato esse que é observado no LAM. Por isso, sua análise acaba recaindo nas mesmas restrições apresentadas para as rotinas não bloqueantes.

O MPICH não define rotinas que possibilitem a sobreposição de execução e comunicação e, por isso, é possível analisar os resultados através das métricas de desempenho definidas. É importante ressaltar que, apesar de todos os modos de comunicação possuírem variantes iniciadas via requisições persistentes, para manter esta análise simples foram considerados apenas exemplos utilizando o modo padrão de comunicação.

Nos resultados obtidos para a latência de biblioteca no MPICH, na maioria dos tamanhos de mensagens estudados, não foi constatada diferença estatística, principalmente para mensagens grandes. Para mensagens pequenas, até por se tratarem de rotinas diferentes (visto que as rotinas de início de comunicação relacionadas a requisições persistentes são mais simples e com menor número de parâmetros), existe uma diferença muito pequena a favor das requisições persistentes, que, na prática, pode ser considerada nula.

No gráfico 8.11, são mostradas as latências de biblioteca para as rotinas de transmissão no modo padrão iniciadas via requisições persistentes (persistente na legenda) e iniciadas normalmente. Para mensagens grandes, o comportamento do gráfico gerado é semelhante.

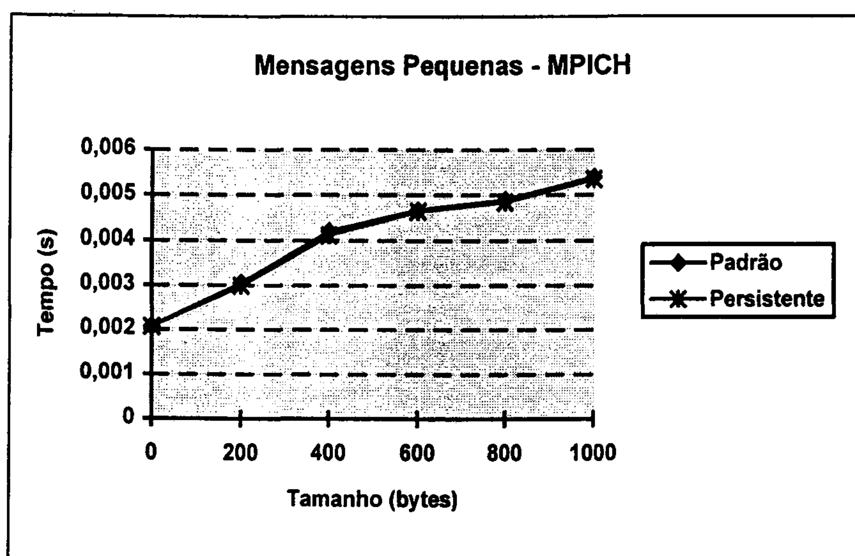

Gráfico 8.11 - Latência da Biblioteca para Mensagens Pequenas no MPICH

8.4. Implementações de Plataformas de Portabilidade

Nesta seção, são apresentados os resultados da comparação entre as plataformas de portabilidade. Não é possível afirmar com apenas essas informações de maneira absoluta que alguma plataforma seja melhor do que a outra. Vários fatores têm que ser considerados, como, por exemplo, a facilidade de uso e a portabilidade [DIL95].

É importante ressaltar que o PVM foi incluído nos testes a fim de fornecer uma base para que se possa avaliar o desempenho das implementações MPI tendo como modelo uma plataforma eficiente e amplamente utilizada. Não tem-se a pretensão de afirmar que o PVM é mais eficiente que o MPI, ou vice-versa. Mesmo porque foram estudadas apenas algumas das implementações MPI.

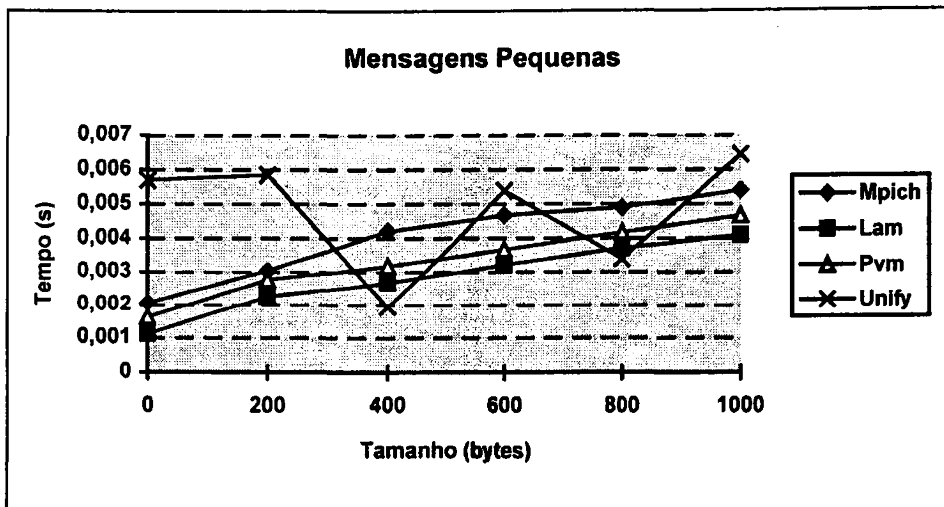

Gráfico 8.12 - Latência de Biblioteca - Mensagens Pequenas

Tanto no MPICH quanto no LAM, o modo de comunicação padrão de comunicação (bloqueante) é implementado através de *buffers* e sem sincronização. Isso se repete no PVM, cujo único modelo de transmissão também possui essas características. Além disso, o UNIFY apenas define o modo padrão de comunicação e, por ser implementado sobre o PVM, também mantém as mesmas características de *bufferização* e falta de sincronismo. Por isso, as comparações apresentadas nesta seção são relacionadas a rotinas de comunicação no modo padrão.

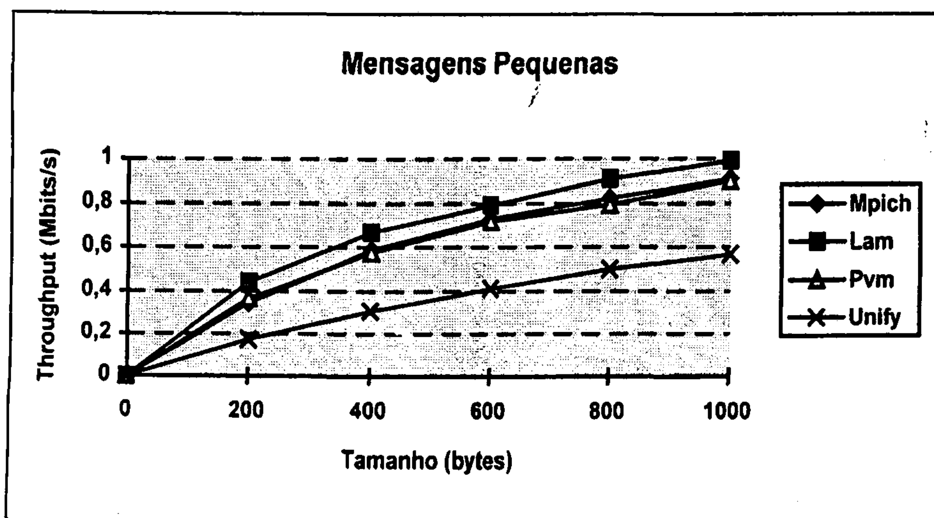

Gráfico 8.13 - *Throughput* - Mensagens Pequenas

De maneira similar à comparação entre os modos de comunicação, as implementações são comparadas ao nível de latência de biblioteca e *throughput* para mensagens pequenas e grandes.

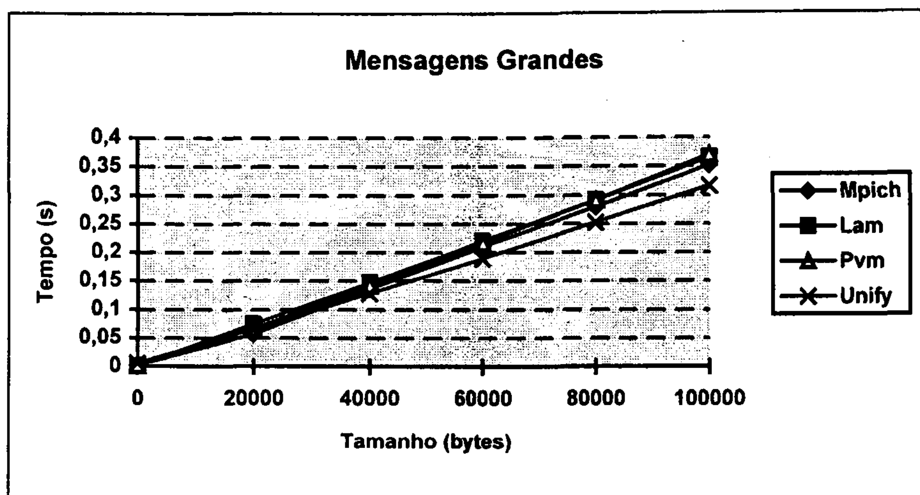

Gráfico 8.14 - Latência da Biblioteca - Mensagens Grandes

Para mensagens pequenas (gráficos 8.12 e 8.13), o melhor desempenho foi apresentado pelo LAM, tanto a nível de latência de biblioteca como a nível de *throughput*. O PVM apresenta menor latência de biblioteca que o MPICH, porém, quando analisados os valores de *throughput*, essa diferença desaparece e ambos fornecem desempenhos parecidos.

Já o UNIFY apresentou tempos pouco previsíveis e estatisticamente confusos para a latência de biblioteca, o que dificulta uma análise conclusiva. Já, para valores de *throughput* apresentou os piores resultados. Deve-se destacar que, para todos os resultados obtidos nesse trabalho, o UNIFY apresentou comportamento irregular, como é discutido na seção 8.6.

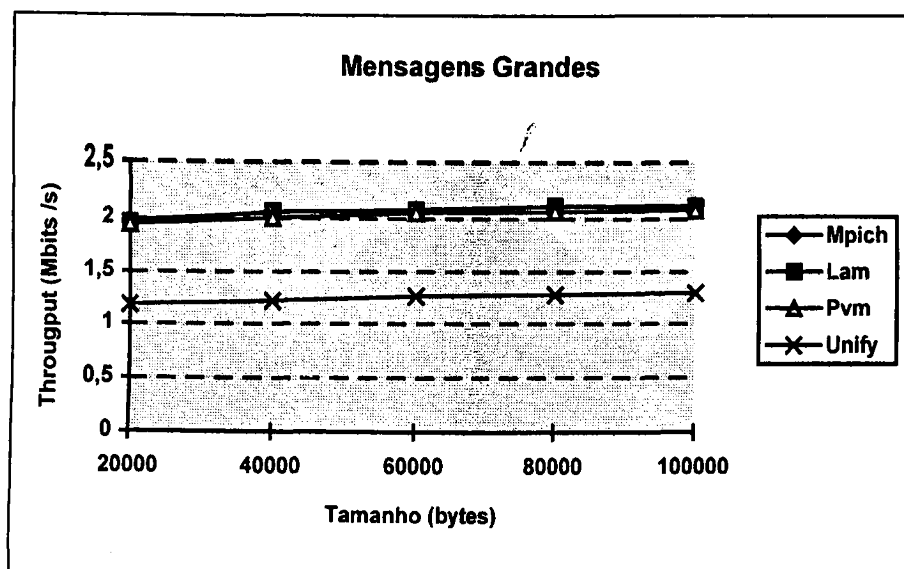

Gráfico 8.15 - *Throughput* - Mensagens Grandes

Para mensagens grandes (gráficos 8.14 e 8.15), as diferenças de desempenho entre as implementações diminuem consideravelmente, sendo que apenas o UNIFY apresenta valores destoantes. As diferenças entre as implementações diminuem a medida que as sobrecargas de *software* diminuem sua ação, pois é em mensagens pequenas que elas possuem maior influência.

8.5. Algoritmo de Jacobi

O algoritmo de Jacobi foi implementado em todas as plataformas de portabilidade utilizadas neste trabalho. Nas implementações MPI, ele foi executado para todos os modos de comunicação, excetuando o modo *ready,* que, segundo as conclusões obtidas, é implementado através de rotinas de transmissão padrão.

A matriz Jacobiana (ver seção 7.5) considerada possui dimensão 250×1000 e foi dividida em quatro blocos (distribuidos em quatro processos) de maneira que as mensagens transferidas entre os processos constituintes desta aplicação fossem de tamanho 1000 *bytes* (visto que são transmitidos vetores de 250 números reais, que no sistema LINUX são representados por 32 *bits*). Foi escolhido esse tamanho de mensagem a fim de possibilitar a comparação com os resultados obtidos pelos *benchmarks.*

Os resultados da execução do algoritmo de Jacobi utilizando rotinas de comunicação bloqueante e não bloqueante são mostrados na tabela 7.1. Tais testes foram executados no LAM, que é a única implementação que possibilita a execução de rotinas realmente não bloqueantes. Segundo os resultados obtidos, não houve diferença marcante dos tempos de execução. Esse já era um resultado esperado, pois não há mecanismos especiais de comunicação via *hardware* nos computadores PC's utilizados. Os processos de execução e de comunicação devem disputar a mesma CPU, de maneira que não se possibilite o ganho de desempenho pela sobreposição das duas operações. Essas conclusões são válidas apenas para a implementação LAM, mas é de se esperar que esse comportamento seja repetido em qualquer implementação que utilize a mesma plataforma computacional utilizada neste trabalho.

Vale a pena ressaltar que conclusões semelhantes foram obtidas através da utilização de requisições persistentes.

	Bloqueante	**Não-bloqueante**
Padrão	39,560600 segundos	39,164146 segundos
Síncrono	40,606714 segundos	40,072026 segundos
Bufferizado	39,635800 segundos	39,213027 segundos

Tabela 8.1 - Comunicação Bloqueante X Não Bloqueante

A comunicação no algoritmo de Jacobi é organizada de maneira que sejam transmitidas mensagens de 1000 *bytes*. É interessante então comparar os resultados de latência ponto-a-ponto obtidos para mensagens deste mesmo tamanho com o tempo de execução do algoritmo, a fim de estimar o quanto os resultados obtidos pelos *benchmarks* se refletem na execução de um algoritmo paralelo real. Esses dois conjuntos de tempos foram medidos para todos os modos de comunicação no LAM e também para todas as implementações, sendo mostrados na tabela 8.2 e 8.3.

	Latência Ponto-a-Ponto	Algoritmo de Jacobi
Padrão	0.016143 s	39,560600 s
Síncrono	0,018101 s	40,606714 s
Bufferizado	0.016291 s	39,635800 s

Tabela 8.2 - Diferenças Entre os Modos de Comunicação

	Algoritmo de Jacobi	Latência Ponto-a-Ponto
MPICH	40,349073 s	0,017561 s
LAM	39,560600 s	0.016143 s
PVM	24,558111 s	0.017728 s
Sequencial	53,43 s	

Tabela 8.3 - Diferenças Entre as Implementações

Percebe-se na tabela 8.2. que os tempos obtidos mantiveram a mesma relação de "sequência de grandezas" (o modo de comunicação síncrono retorna os maiores tempos de execução enquanto que o padrão retorna os menores), embora não mantenham entre si a mesma proporção. Comportamento semelhante foi observado no MPICH. Os valores obtidos pela utilização de comunicação em duas vias ofereceram melhores resultados que todos os modos de comunicação (tempo = 39,251254 s). Esse resultado também era esperado, visto que o algoritmo de Jacobi utiliza o padrão de comunicação *ping-pong* (ver seção 6.2.4).

Segundo os resultados mostrados na tabela 8.3, o PVM apresentou os melhores resultados enquanto que, entre as implementações MPI, destacou-se o LAM. Considerando os valores obtidos para a latência ponto-a-ponto em todas as implementações, apenas o PVM apresentou resultados não esperados. O UNIFY não foi considerado nessa análise visto que não executou corretamente o algoritmo de Jacobi.

Antes de concluir, é importante ressaltar que o tempo de execução de um programa completo, e não apenas de suas rotinas de comunicação ponto-a-ponto, é influenciado por vários outros fatores que podem pesar no seus tempos de execução. Além disso, esses resultados são frutos de uma combinação de características da aplicação e de cada implementação. Então, não depende só da eficiência das implementações a diferença de tempos apresentadas. Isto é, a maneira como o algoritmo foi construído pode aproveitar melhor determinadas características de cada biblioteca.

Não é possível, através da execução de apenas uma aplicação, validar completamente os resultados obtidos pelos *benchmarks*. A maneira como foi estruturada a aplicação (número de blocos, tamanho das mensagens, número de iterações, etc.) pode influenciar nos resultados obtidos. A intenção desta comparação é apresentar alguns exemplos dos resultados práticos que possam oferecer uma idéia do quanto uma análise desse porte pode ser útil quando da avaliação de uma algoritmo.

Visto sobre esse ponto de vista, em relação às diferenças entre os modos de comunicação e entre as implementações do MPI, os resultados obtidos pelos *benchmarks* se espelharam na execução do algoritmo de Jacobi, e foram considerados satisfatórios e úteis.

Os resultados dos *benchmarks* apresentados nesse capítulo, sozinhos, não podem avaliar exatamente as diferenças de desempenho de algoritmos paralelos, mas, sim oferecer resultados importantes que devem ser considerados também durante a análise. O objetivo deles é fornecer informações a partir das quais possa-se entender as relações entre as rotinas de comunicação e entre as implementações, e a partir daí, escolher as melhores opções para sua aplicação.

8.6. Considerações Finais

Nesse capítulo, foram apresentados os resultados da avaliação de desempenho das rotinas de comunicação ponto-a-ponto e das implementações MPI. Tendo esses resultados em mãos, é interessante relacionar a eles as informações apresentadas nos capítulos anteriores, a fim de apresentar uma visão geral e resumida de algumas conclusões que foram obtidas nesse trabalho.

Rotinas de Comunicação Ponto-a-Ponto: Relacionando todas as formas de comunicação ponto-a-ponto apresentadas no capítulo 6, e discutindo-se as suas características relacionadas ao desempenho e vantagens obtidas de sua utilização, tem-se:

- **Bloqueante ou não bloqueante**: Em virtude da falta de mecanismos especiais de comunicação, uma rede LINUX não possibilita ganhos

significativos de desempenho a partir da sobreposição de comunicação e execução, esta que é a principal característica de uma comunicação não bloqueante. Algoritmos construídos com rotinas não bloqueantes são difíceis de organizar e depurar, e sempre pode-se utilizar rotinas de comunicação *bufferizadas* quando há riscos de *deadlock*. Por isso, o usuário deve preferir a utilização de rotinas bloqueantes, a não ser que a aplicação exija a utilização de rotinas não bloqueantes;

- **Modos de comunicação**: A primeira opção do usuário deve ser o modo padrão, visto que ele oferece o melhor desempenho e é mais robusto. É claro que devem ser construídos algoritmos seguros, como apresentado na seção 2.3.1. De acordo com as características da aplicação, pode-se utilizar o modo síncrono ou *bufferizado*. Rotinas *bufferizadas* são aconselháveis quando se quer escrever algoritmos portáteis, visto que o usuário pode controlar o tamanho dos *buffers* e garantir que só se esgotarão sobre condições patológicas. Já as rotinas síncronas oferecem determinismo na execução do algoritmo e confiabilidade. Os dois modos possuem restrições em relação ao desempenho que também devem ser consideradas;
- **Rotinas de comunicação em duas vias**: Vale a pena utilizá-las quando se escreve algoritmos que possuam a organização de comunicação que as caracteriza (o algoritmo de Jacobi, por exemplo), pois essas estruturas oferecem ganho de desempenho e garantia contra *deadlocks*, facilitando a escrita do algoritmo;
- Requisições persistentes: Possuem as desvantagens de rotinas não bloqueantes e pouco oferecem de ganho de desempenho em relação a utilização de rotinas padrão. Só devem ser utilizadas quando o usuário sabe exatamente o que explorar delas.

Implementações MPI: A partir dos resultados práticos obtidos nesse trabalho e também da comparação entre as implementações discutida no capítulo 5, tem-se:

- O LAM é fácil de usar, é bem documentado, fornece mecanismos de depuração de programas paralelos muito úteis e oferece o melhor desempenho entre as três implementações. Além disso, o LAM apresentou comportamento coerente com as recomendações do Fórum MPI. Seu maior problema são é na fase de instalação.
- O MPICH é um projeto ambicioso e bem sucedido até aqui [GRO96a]. É também fácil de usar, instala-se sem problemas e possui

ótima documentação. Porém, apresentou problemas de execução em algumas rotinas e apresentou menor desempenho quando comparado ao LAM.

- O UNIFY exigiu o maior trabalho entre todas as implementações para sua instalação. Além disso, apresentou comportamento instável e pouco confiável. É um projeto aparentemente abandonado por seus autores e, segundo seu guia de instalação, não tinha sido testado em ambiente LINUX. Por isso, não apresentou um comportamento satisfatório.

A partir dessas colocações, e considerando os tópicos de comparação discutidos no capítulo 5, a tabela 8.4 mostra as implementações MPI relacionadas e comparadas entre si. Os tópicos de comparação utilizados são: portabilidade, facilidade de instalação, interface (para compilação e execução de programas), documentação, ferramentas de depuração, fidelidade ao padrão MPI e desempenho. Para cada um dos tópicos, é assinalada uma nota de 0 a 4 para cada implementação, onde: 0 = não existe, 1 = ruim, 2 = regular, 3 = bom e 4 = ótimo. Essas notas representam o julgamento da qualidade das implementações em relação ao respectivo tópico. Determinadas considerações são definidas pelas impressões pessoais dos autores desse trabalho.

	LAM	MPICH	UNIFY
Portabilidade	4	4	4
Facilidade de instalação	1	3	1
Interface	4	3	2
Documentação	3	4	2
Depuração	4	2	1
Fidelidade	4	3	1
Desempenho	4	3	1
TOTAL	24	22	12

Tabela 8.4. Tabela de Comparação Entre as Implementações MPI

Segundo a tabela 8.1, conclui-se que o MPICH e o LAM são escolhas viáveis de utilização em ambiente LINUX. As duas implementações permitem resultados satisfatórios para o usuário. Porém, ao se considerar todos os fatores que foram analisados nesse trabalho, a implementação que forneceu os melhores resultados em ambiente LINUX é o LAM.

PVM e MPI: As implementações MPI mantiveram, nos resultados obtidos pelos *benchmarks*, desempenho comparável aos obtidos pelo PVM, o que demonstra que a organização definida pelo MPI possibilita, em ambiente LINUX, resultados satisfatórios (Excetuando o UNIFY, que apresentou comportamento irregular). Pode-se concluir que o MPI permite que suas implementações consigam desempenhos comparáveis a plataformas de portabilidade projetadas especialmente para sistemas distribuídos (como é o caso do PVM), e este pode ser citado como um ponto positivo do MPI, e em especial de suas implementações LAM e MPICH.

Capítulo 9
Conclusões

Nesse capítulo são apresentadas as principais conclusões obtidas durante o desenvolvimento desse trabalho.

9.1. Considerações Iniciais

Nesse trabalho, foi apresentado um estudo, tanto a nível de estrutura quanto a nível de desempenho, das rotinas de comunicação ponto-a-ponto do MPI. Seu objetivo principal foi discutir as relações custo/benefício entre elas, enfocando desempenho, facilidade de utilização, possibilidade de ocorrência de *deadlock*, etc.

Adicionalmente, foi efetuada uma avaliação e comparação entre três implementações do MPI, comparando também, sempre que possível, resultados obtidos pela utilização do PVM.

Relacionando alguns aspectos importantes a serem considerados a respeito desse trabalho, nas próximas seções são apresentadas as conclusões obtidas (seção 9.2), as contribuições oferecidas (seção 9.3) e as principais dificuldades encontradas (seção 9.4).

9.2. Conclusões

Os resultados obtidos por esse trabalho permitiram as seguintes conclusões:

- As rotinas de comunicação do MPI possibilitam diferentes desempenhos, e deve-se considerar esse aspecto quando da organização de uma algoritmo paralelo;
- Existem determinadas características das rotinas do MPI que não são bem aproveitadas em ambientes distribuídos, como por exemplo, rotinas não bloqueantes e rotinas *ready*. Esses eram resultados dos quais se suspeitavam e foram confirmados na prática;
- Não deve-se analisar o padrão MPI por si só quando se deseja escolher uma plataforma de portabilidade para a execução de algoritmos paralelos. É muito importante analisar o qualidade e abrangência de cada implementação. Por exemplo, características

como a quantidade de rotinas do MPI oferecidas e se são implementadas como o padrão define devem ser consideradas;

- As diferentes implementações MPI ofereceram diferentes desempenhos, de maneira que se conclui que não existe uma uniformidade de comportamento das implementações. Esse pode ser citado como um ponto negativo do MPI;
- Segundo os resultados obtidos nesse trabalho, a melhor implementação MPI em ambiente LINUX é o LAM. O MPICH também foi considerada uma implementação muito boa. O UNIFY, pelo comportamento instável, foi reprovado;
- Para a análise de uma aplicação paralela, vários fatores relacionados a organização dos algoritmos possuem influência. Porém, a eficiência e as características de suas rotinas de comunicação são fatores importantes e devem ser considerados quando da escrita de uma algoritmo paralelo utilizando o MPI em sistemas distribuídos.
- Segundo a comparação com os tempos obtidos pelo PVM, na execução dos *benchmarks*, a utilização do MPI sobre sistemas distribuídos foi considerada viável e satisfatória.

9.3. Contribuições Deste Trabalho

Entre as contribuições deste trabalho, podem ser destacadas:

- O usuário comum possui informações a partir das quais pode escolher entre as implementações MPI analisadas e organizar seus programas paralelos de acordo com seus objetivos;
- Apresenta resultados práticos da utilização do MPI em redes de computadores pessoais executando o sistema LINUX, que não foram explorados na bibliografia pesquisada;
- Apresenta de maneira concisa e organizada as rotinas de comunicação ponto-a-ponto do MPI, analisando-as a nível de estrutura e desempenho;
- Os métodos de análise de desempenho para rotinas de comunicação ponto-a-ponto através de *benchmarks* e métricas de desempenho foram estudados na bibliografia disponível e organizados de maneira concisa, de maneira que se forneça *feedback* para trabalhos que necessitem fazer avaliação similar;
- Fornece informações extremamente úteis que possibilitam que outros trabalhos possam utilizar o MPI como ferramenta, conhecendo melhor

as formas de comunicação e as implementações MPI. Além disso, todas as implementações foram testadas, o que possibilita segurança na sua utilização;

- Todos os *benchmarks* utilizados estão disponíveis e podem ser aproveitados para trabalhos similares. Podem ser utilizados, com algumas modificações, nos trabalhos futuros que são sugeridos na seção 9.5.

9.4. Dificuldades Encontradas

Várias dificuldades foram encontradas no decorrer desse trabalho, principalmente relacionadas ao fato de serem utilizados *softwares* não comerciais e em constante desenvolvimento (por isso, suscetíveis a erros). Por exemplo, como é observado pelos próprios autores dos *softwares* utilizados nesse trabalho, problemas na execução dos algoritmos podem ocorrer em virtude de erros nas próprias plataformas de portabilidade. A questão é descobrir aonde está o erro (na implementação ou no algoritmo). Nesse trabalho, ambas situações ocorreram.

De todas as dificuldades encontradas durante o desenvolvimento desse trabalho, podem ser destacadas:

- A instalação do sistema LINUX foi difícil (em virtude de incompatibilidades com o *hardware* disponível) e demorada. Além disso, foi necessário configurá-lo a fim de que executasse corretamente as implementações;
- Todas as implementações apresentaram problemas. O suporte ao sistema LINUX é recente em todas elas, e por isso, ainda instável;
- Foi necessária uma pesquisa em trabalhos correlatos da área a fim de definir quais os requisitos necessários para uma avaliação desse tipo. Todas as informações estudadas necessitaram ser filtradas e adaptadas para o problema considerado;
- Esse trabalho envolveu uma grande quantidade de números, que necessitavam ser organizados de maneira que se pudesse estudá-los. Isso implica na utilização de uma grande quantidade de tabelas e gráficos para serem avaliados e analisados estatisticamente.

9.5. Trabalhos Futuros

Para a continuidade e complementação dos resultados obtidos nesse trabalho, são sugeridos os seguintes trabalhos futuros:

- Estudar as rotinas coletivas do MPI, que também possuem grande variação de formas;
- Analisar resultados similares obtidos em ambientes computacionais correlatos, como uma rede de *workstations Sun*, verificando a semelhança entre as conclusões;
- Estudar o comportamento do MPI em ambiente *Windows95* (existe uma implementação MPI desenvolvida em Portugal baseada no MPICH [MPI96]), a fim de definir diferenças entre as implementações MPI e também entre os ambientes computacionais;
- Executar os *benchmarks* em computadores paralelos, a fim de avaliar as diferenças obtidas pela utilização do MPI em sistemas distribuídos e arquiteturas paralelas;
- Aprofundar o estudo das diferenças entre o MPI e o PVM, tanto a nível de estrutura quanto de desempenho oferecido. Verificar se as conclusões desse trabalho seriam semelhantes com a utilização de computadores paralelos.

Referências Bibliográficas

[ACH95] ACHCAR, J. A., Rodrigues, J., "Introdução à estatística para ciências e tecnologia", Universidade de São Paulo, São Carlos, 1995.

[ALM94] ALMASI, G. S., Gottlieb, A., *Highly Parallel Computing*, 2ª ed., The Benjamin Cummings, 1994.

[AMO88] AMORIM, C. L., *et al.*, *Uma Introdução a Computação Paralela e Distribuída*, VI Escola de Computação, 1988.

[AND83] ANDREWS, G. R., Schneider, F. B., "Concepts and notations for concurrent programming", *ACM Computing Survey*, vol.15, n.1, p.3-43, 1983.

[BEG94] BEGUELIN, A., *et al.*, *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, The MIT Press, 1994.

[BEN93] BEN-DYKE, A. D., "Architectural taxonomy, A brief review", University of Birmingham, 1993.

[BLE94] BLECH, R. A., "An overview of parallel processing", Slides, *Parallel Computing with PVM Workshop*, Nasa Lewis Research Center, http://www.lerc.nasa.gov/Other_Groups/IFMD/2620/tutorialPP.html, 1994.

[BUR89] BURNS, G., "A local area multicomputer", The Ohio State University, ftp://tbag.osc.edu/pub/lam, 1989.

[BUR90] BURNS, G., *et al*, "All about Trollius", The Ohio State University, ftp://tbag.osc.edu/pub/lam, 1990.

[BUR94] BURNS, G., *et al*, "LAM: An open cluster environment for MPI", Ohio Supercomputer Center, ftp://tbag.osc.edu/pub/lam, 1994.

[BUR95] BURNS, G., Daoud, R, "MPI Primer / Developing with LAM", Ohio Supercomputer Center, The Ohio State University, ftp://tbag.osc.edu/pub/lam, 1995.

[BUT94] BUTLER, R. M., Lusk, E. L., "Monitors, messages and clusters: The p4 parallel programming system", *Parallel Computing*, vol. 20, pp. 547-564, 1994.

[CAL94] CALKIN, R., *et al*, "Portable programming with the PARMACS message-passing library", *Parallel Computing*, vol. 20, pp. 615-632, 1994.

[CAR94] CARRIERO, N., *et al*, "The Linda alternative to message-passing systems", *Parallel Computing*, vol. 20, pp. 633-655, 1994.

[COR96] CORNELL THEORY CENTER, "Introduction to parallel processing", Virtual Workshops, 1996.

[CHE94] CHENG, F., *et al*, "The Unify system", Mississipi State University, ftp://ftp.erc.msstate.edu/unify/, 1994.

[COU94] COULOURIS, G., *et al*, *Distributed Systems Concepts and Design*, 2ª. ed., Addison-Wesley, 1994.

[DIL95] DILLON, E., *et al*, "Homogeneous and heterogeneous networks of workstations: Message passing overhead", *MPI Developers Conference*, University of Notre Dame, 1995.

[DON95] DONGARRA, J. J., *et al*, "An introduction to the MPI standard", University of Tennessee Technical Report CS-95-274, http://www.netlib.org/utk/papers/intro-mpi/intro-mpi.html,1995.

[DOS96] DOSS, N., "MPI Frequently Asked Questions", http://www.erc.msstate.edu/mpi/mpi-faq.html, 1996.

[DOU93] DOUGLAS, C. C., *et al*, "Parallel programming systems for workstations cluster", Relatório Técnico YALEU/DCS/TR-975, Departament of Computer Science, Yale University, 1993.

[DUN90] DUNCAN, R., "A survey of parallel computer architectures", *IEEE Computer*, pp. 5-16, 1990.

[DUN95] DUNIGAN, T., Dongarra, J. J., "Message-passing performance of various computers", Technical Report ut-cs-95-299, University of Tennessee, 1995.

[ENG95] ENGLER, N., "Riding the bleeding edge of distributed computing", *Open Computing*, vol. 12, nº 2, pp. 34-44, 1995.

[FLO94] FLOWER, J., Kolawa, A., "Express is not just a message passing system. Current and future directions in Express", *Parallel Computing*, vol. 20, pp. 597-614, 1994.

[FLY72] FLYNN, M. J., "Some computer organizations and their effectiveness", *IEEE Transactions on Computers*, vol. C-21, pp.948-960, 1972.

[GRO95] GROPP, W., Lusk, E., "Some early performance results with MPI on the IBM SP1", disponível no pacote MPICH, 1995.

[GRO96a] GROPP, W., *et al*, "A high-performance, portable implementation of the MPI Message Passing Interface standard", http://www.mcs.anl.gov/mpi/mpich.html, 1996.

[GRO96b] GROPP, W., Lusk, E., "User's guide for MPICH, a portable implementation of MPI", Argonne National Laboratory, Disponível no pacote MPICH, 1996.

[GRO96c] GROPP, W., Lusk, E., "Installation guide for MPICH, a portable implementation of MPI", Argonne National Laboratory, Disponível no pacote MPICH, 1996.

[HWA84] HWANG, K., Briggs, F. A., *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[KAR94] KARRELS, E., Lusk, E. "Performance analysis of MPI programs". *Environments and Tools for Parallel Scientific Computing*, pp. 195-200, SIAM, 1994.

[KIR89] KIRNER, C., "Sistemas operacionais para ambientes paralelos", *IX Congresso da SBC, Anais da VIII Jornada de Atualização em Informática*, 1989.

[KIR91] KIRNER, C., "Arquiteturas de sistemas avançados de computação", *Anais da Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho*, pp. 307-353, 1991.

[KIT95] KITAJINA, J.P.F.W., "Programação paralela utilizando mensagens", *XV Congresso da Sociedade Brasileira de Computação, XIV Jornada de Atualização em Informática*, 1995.

[LEN95] LENATTI, C., "Rethinking in parallel", *Open Computing*, vol. 12, pp. 57-58, 1995.

[LAM96] LAM Home Page, http://www.osc.edu/lam.html, 1996.

[MAC96] MACDONALD, N., *et al*, "Writing message-passing parallel programs with MPI", Course Notes, The University of Edinburgh, http://www.epcc.ed.ac.uk/epcc-tec/package.html, 1966.

[MCB94] MCBRYAN, O. A., "An overview of message passing environments", *Parallel Computing*, vol. 20, pp. 417-444, 1994.

[MEG95] MEGLICKI, Z., "The LAM companion to '*Using MPI ...*' ", The Australian National University, ftp://cisr.anu.edu.au/pub/papers /meglicki/mpi/tutorial/mpi/mpi.html, 1995.

[MEY94] MEYER, J., "Message-passing interface for Microsoft Windows 3.1", Msc. Thesis, Department of Computer Science, University of Nebraska, 1994.

[MPI96] MPICH Home Page, http://www.mcs.anl.gov/mpi/mpich.html, 1996.

[NAV89] NAVAUX, P. O. A., "Introdução ao processamento paralelo", *RBC- Revista Brasileira de Computação*, vol. 5, nº 2, pp. 31-43, 1989.

[NEV96] NEVIN, Nick., "The performance of LAM 6.0 and MPICH 1.0.12 on a workstation cluster", Ohio Supercomputer Technical Report, OSC-TR-1996-4, 1996.

[NUP94] NUPAIROJ, N., Ni, L. M., "Performance evaluation of some MPI implementations on workstations clusters", *Proceedings of the 1994 Scalable Parallel Libraries Conference*, SPLC94, pp. 98-105, 1994.

[ORT85] ORTEGA, J. M., "Solution of Differential Equations on Vector and Parallel Computers", SIAM, 1985.

[PAC95] PACHECO, P. S., "A user's guide to MPI", University of San Francisco, ftp://math.usfca.edu/pub/MPI/mpi.guide.ps, 1995.

[PVM96] PVM Home Page. http://www.epm.ornl.gov/pvm/ pvm_home.html. 1996.

[QUE94] QUEALY, A, "An introduction to Message Passing", slides, NASA Lewis Research Center, http://www.lerc.nasa.gov/Other_Groups /IFMD/2620/tutorialPP.html, 1994.

[QUI94] QUINN, M.J., *Parallel Computing: Theory and Practice*. McGraw Hill, 1994.

[QUI87] QUINN, M.J., *Designing Efficient Algorithms for Parallel Computers*, McGraw Hill, 1987.

[SAN89a] SANTANA, R. H. C., "Performance evaluation of LAN-based file-servers", *Ph.D. Dissertation*, University of Southampton, 1989.

[SAN89b] SANTANA, M. J., "An advanced filestore architecture for a multiple-LAN distributed computing system", *Ph.D. Dissertation*, University of Southampton, 1989.

[SKJ95] SKJELLUM, A., Lusk, E., "Early applications in the Message Passing Interface (MPI)", *International Journal of Supercomputing Applications and High Performance Computing*, vol. 9, nº 2 , 1995.

[SKJ94] SKJELLUM, A., *et al*, "Extending the Message Passing Interface (MPI)" *Proceedings of the 1994 Scalable Parallel Libraries Conference*, IEEE Computer Society Press, 1994.

[SMI93] SMITH, B., Gropp, W., "Chameleon parallel programming tools users manual", Technical Report ANL-93/23, Argonne National Laboratory, 1992.

[SMI94] SMITH, S. G., *et al*, "The design and evolution of Zipcode", *Parallel Computing*, vol. 20, pp. 565-596, 1994.

[SNI96] SNIR, M., *et* al, MPI*: The Complete Reference*, The MIT Press, 1996.

[SNO92] SNOW, C.R., *Concurrent Programming*, Cambridge University Press, 1992.

[SOA91] SOARES, J. F., *et al*, *Introdução à estatística*, Guanabara Koogan, 1991.

[SOU95] SOUZA, M. A., *et al.*, "Aplicações de RPC no ambiente SunOS", Notas Didáticas do ICMSC , USP - São Carlos, 1995.

[STE90] STEVENS, W. R., *UNIX Network Programming*, Prentice Hall,1990.

[SUN94] SUNDERAM, V. S., *et al*, "The PVM concurrent computing system: evolution, experiences and trends", *Parallel Computing*, vol. 20, pp. 531-545, 1994.

[TAN95] TANENBAUM, A. S., *Distributed Operating Systems* , Prentice Hall, 1995.

[TUR93] TURCOTTE, L.H., "A survey of software environments for exploiting networked computing resources", Engineering Research Center for Computational Field Simulation, Mississippi State University, ftp://bulldog.wes.army.mil/pub/report.ps, 1993.

[VAU94] VAUGHAN, P., *et al*, "Migrating from PVM to MPI, part I: The *Unify* system", Frontiers '94, 1994.

[VIC81] VICHNEVETSKY, R., *Computer Methods for Partial Differential Equations*, vol. 1, Prentice Hall, 1981.

[WAL94] WALKER, D. W., "The design of a standard message passing interface for distributed memory concurrent computers", *Parallel Computing*, vol. 20, pp. 657-673, 1994.

[WAL95] WALKER, D.W., "MPI: From fundamentals to applications", Oak Ridge National Laboratory, http://www.epm.ornl.gov/ ~walker /mpi /SLIDES/mpi-tutorial.html, 1995.

[WEL95] WELSH, M., *Linux Installation and Getting Started*, 1995.

[WIL87] WILBUR, S., Bacarisse, B., "Building distributed systems with remote procedure call", *Software Engineering Journal*, 1987.

[ZAL91] ZALUSKA, E. J., "Research lines in distributed computing systems and concurrent computation", *Anais do Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software*, pp. 132-155, 1991.

Apêndice A

Análise Estatística

A análise estatística utilizada nesse trabalho possibilita a comparação entre dois tratamentos, isto é, partindo de duas amostras de números, deve-se calcular se os respectivos valores médios obtidos sobre cada amostra são estatisticamente distintos [ACH95] [SOA91].

Para isso, são definidas hipóteses estatísticas, e a partir da aceitação ou rejeição dessas hipóteses, pode se concluir sobre a validade de uma determinada afirmação. Seja, no caso, duas amostras, calculadas a partir de um experimento X $(X_1, X_2, X_3, ..., X_n)$ e um experimento $Y(Y_1, Y_2, Y_3, ..., Y_n)$ e que possuem as respectivas médias populacionais μ_X e μ_Y. Quer se avaliar se as médias μ_X e μ_Y são estatisticamente diferentes, ou seja, se as diferenças obtidas dessas médias realmente traduzem a realidade demonstrada pelas amostras a partir das quais elas foram gerada.

Suponha que, nos experimento obtidos, tenha se verificado que $\mu_X > \mu_Y$. Então formula-se as seguintes hipóteses estatísticas:

H_1: $\mu_X > \mu_Y$

H_0: $\mu_X \leq \mu_Y$.

H_1 é a hipótese alternativa, na qual o resultado ocorrido é confirmado e H_0 é a hipótese nulidade, que afirma exatamente o contrário. Para provar qual hipótese é correta, tenta-se provar a hipótese nulidade. Se não é possível provar a hipótese nulidade, a hipótese alternativa é considerada verdadeira, e a diferença obtida por μ_X e μ_Y é considerada estatisticamente valida.

A estatística dos testes de hipótese acima é dada por

$$Z = \frac{\mu_X - \mu_Y}{\sqrt{\frac{S_X^2}{n_X} + \frac{S_Y^2}{n_Y}}} \qquad \text{(A.1)}$$

onde μ_X e μ_Y são as médias amostrais, S_X e S_Y são os desvios-padrão e n_X e n_Y são os tamanhos da amostras, valores esses relativos as experimentos X e Y respectivamente.

Considerando-se nível de significância de 0,99, tem-se:

Se $H_1: \mu_X > \mu_Y$, rejeitar $H_0: \mu_X \leq \mu_Y$ se $Z \geq 2,33$
Se $H_1: \mu_X < \mu_Y$, rejeitar $H_0: \mu_X \geq \mu_Y$ se $Z \leq 2,33$

Nas tabelas apresentadas nas próximas páginas, são relacionados e analisados estatisticamente os principais resultados obtidos nesse trabalho. Para cada dupla de valores médios comparados, são calculados os valores de Z. Se a diferença estatística é verificada, no campo relacionado ao valor de Z é colocado o palavra *aceita*, significando que a hipótese alternativa foi aceita e existe diferença estatística entre os valores medidos. Caso contrário, no campo Z é colocado o valor *rejeita*.

A.1. MPICH - Latência de Biblioteca

Benchmark: Ping

Modos de Comunicação	Dados Estatísticos	Tamanho da mensagem (bytes)										
		0	200	400	600	800	1000	20000	40000	60000	80000	100000
1 -Padrão	Média	0.002067	0.003046	0.004207	0.004671	0.004897	0.005395	0.056565	0.136158	0.210212	0.279128	0.351612
	Desvio Padrão	0.000351	0.000030	0.000103	0.000100	0.000034	0.000034	0.012077	0.004798	0.002323	0.003299	0.005188
2 - Síncrono	Média	0.007899	0.008644	0.009375	0.010519	0.011651	0.012919	0.084922	0.158225	0.233968	0.305846	0.380248
	Desvio Padrão	0.000052	0.000057	0.000050	0.000050	0.000072	0.000878	0. 000985	0.000203	0.000164	0.000417	0.000271
3 - *Bufferizado*	Média	0.002572	0.003609	0.004759	0.005249	0.005682	0.005991	0.057903	0.136763	0.208744	0.281565	0.352787
	Desvio Padrão	0.000283	0.000272	0.000033	0.000116	0.000104	0.000093	0.008405	0.007648	0.007066	0.003481	0.002715
4 - *Ready*	Média	0.002021	0.00304	0.004175	0.004665	0.004899	0.005391	0.057549	0.136477	0.210162	0.280287	0.352039
	Desvio Padrão	0.000242	0.000031	0.000033	0.000036	0.000113	0.000034	0.009046	0.004252	0.002171	0.002818	0.002998
	Z 1 / 2	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 1 / 3	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Rejeita	Rejeita	Rejeita	Aceita	Rejeita
	Z 1 / 4	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita
	Z 2 / 3	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 2 / 4	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 3 / 4	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita

Modos de Comunicação	Dados Estatísticos	Tamanho da mensagem (bytes)										
		0	200	400	600	800	1000	20000	40000	60000	80000	100000
1 -Padrão	Média	0.002067	0.003046	0.004207	0.004671	0.004897	0.005395	0.056565	0.136158	0.210212	0.279128	0.351612
	Desvio Padrão	0.000351	0.000030	0.000103	0.000100	0.000034	0.000034	0.012077	0.004798	0.002323	0.003299	0.005188
2 - Requisição Persistente	Média	0.002064	0.002988	0.004122	0.004638	0.004844	0.005348	0.057587	0.135800	0.207972	0.280105	0353367
	Desvio Padrão	0.000428	0.000030	0.000047	0.000136	0.000062	0.000036	0.008573	0.005084	0.005711	0.002726	0.025572
	Z 1 / 2	Aceita	Rejeita	Aceita	Rejeita	Aceita	Aceita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita

A.2. LAM - Latência de Biblioteca

Benchmark: Ping

Modos de Comunicação	Dados Estatísticos	Tamanho da mensagem (bytes)										
		0	200	400	600	800	1000	20000	40000	60000	80000	100000
1 -Padrão	Média	0.001121	0.002259	0.002672	0.003215	0.003707	0.004091	0.074753	0.147438	0.220609	0.291094	0.365122
	Desvio Padrão	0.000093	0.000221	0.000040	0.000030	0.000031	0.000106	0.006205	0.003137	0.006272	0.002326	0.001414
2 - Síncrono	Média	0.004466	0.005727	0.006893	0.008004	0.009089	0.010205	0.074736	0.147063	0.221071	0.290765	0.365175
	Desvio Padrão	0.000041	0.000039	0.000042	0.000039	0.000030	0.000175	0.003228	0.004503	0.004262	0.006923	0.001081
3 - *Bufferizado*	Média	0.001149	0.002312	0.002723	0.003278	0.003789	0.00426	0.023746	0.046806	0.070279	0.111735	0.107601
	Desvio Padrão	0.000140	0.000029	0.000099	0.000028	0.000032	0.000111	0.011392	0.020660	0.025976	0.067084	0.054437
4 - *Ready*	Média	0.001169	0.002257	0.002674	0.003199	0.003705	0.0041	0.074286	0.147929	0.220844	0.290987	0.365372
	Desvio Padrão	0.000132	0.000037	0.000041	0.000102	0.000029	0.000051	0.005500	0.003485	0.004166	0.005715	0.000277
	Z 1 / 2	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita
	Z 1 / 3	Rejeita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 1 / 4	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita
	Z 2 / 3	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 2 / 4	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita
	Z 3 / 4	Rejeita	Rejeita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita

A.3. MPICH - *Throughput*

Benchmark: Ping-Pong

Modos de Comunicação	Dados Estatísticos	Tamanho da mensagem (bytes)										
		0	200	400	600	800	1000	20000	40000	60000	80000	100000
1 -Padrão	Média	0.0	0.342319	0.590079	0.731372	0.823468	0.911109	1.979512	2.053559	2.085600	2.119422	2.125824
	Desvio Padrão	0.0	0.003739	0.003619	0.005293	0.041077	0.005798	0.004405	0.087394	0.002099	0.001249	0.000984
2 - Síncrono	Média	0.0	0.229242	0.406607	0.534134	0.633788	0.705529	1.923169	2.042470	2.063934	2.102845	2.112339
	Desvio Padrão	0.0	0.001732	0.005488	0.023826	0.006390	0.021555	0.004164	0.002089	0.001665	0.001538	0.001089
3 - *Bufferizado*	Média	0.0	0.311405	0.537634	0.677678	0.776369	0.848311	1.940052	2.027806	2.051036	2.085159	2.091763
	Desvio Padrão	0.0	0.000932	0.003394	0.003132	0.002467	0.039977	0.007226	0.040455	0.002343	0.001635	0.002991
4 - *Ready*	Média	0.0	0.340751	0.579342	0.732488	0.833007	0.907955	1.979352	2.072579	2.076358	2.119359	2.125347
	Desvio Padrão	0.0	0.015401	0.037779	0.003604	0.003571	0.003556	0.005340	0.002056	0.039659	0.001708	0.001377
	Z 1 / 2		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Rejeita	Aceita	Aceita	Aceita
	Z 1 / 3		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 1 / 4		Rejeita	Rejeita	Rejeita	Rejeita	Aceita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita
	Z 2 / 3		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 2 / 4		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 3 / 4		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita

A.4. LAM - *Throughput*

Benchmark: Ping-Pong

Modos de Comunicação	Dados Estatísticos	Tamanho da mensagem (bytes)										
		0	200	400	600	800	1000	20000	40000	60000	80000	100000
1 -Padrão	Média	0.0	0.442294	0.667570	0.791883	0.910060	0.991141	1.957689	2.064063	2.078416	2.114884	2.114813
	Desvio Padrão	0.0	0.008973	0.002165	0.062940	0.002409	0.001721	0.009183	0.001479	0.001554	0.000579	0.047532
2 - Síncrono	Média	0.0	0.357981	0.566622	0.707703	0.812853	0.883929	1.956217	2.063158	2.077135	2.113815	2.121202
	Desvio Padrão	0.0	0.001594	0.002261	0.013340	0.001995	0.036556	0.009695	0.001619	0.002034	0.001238	0.003337
3 - *Bufferizado*	Média	0.0	0.439983	0.663281	0.805774	0.901789	0.982137	1.912388	2.027452	2.043561	2.080340	2.085043
	Desvio Padrão	0.0	0.002322	0.001355	0.002249	0.007155	0.001987	0.040571	0.001353	0.001537	0.000806	0.015555
4 - *Ready*	Média	0.0	0.443889	0.668756	0.813904	0.912200	0.987837	1.959103	2.064462	2.078628	2.114832	2.119651
	Desvio Padrão	0.0	0.002796	0.003400	0.004542	0.005478	0.018039	0.004579	0.001096	0.001549	0.000609	0.013142
	Z 1 / 2		Aceita	Aceita	Aceita	Aceita	Aceita	Rejeita	Aceita	Aceita	Aceita	Rejeita
	Z 1 / 3		Rejeita	Aceita	Rejeita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 1 / 4		Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Rejeita
	Z 2 / 3		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 2 / 4		Aceita	Aceita	Aceita	Aceita	Aceita	Rejeita	Aceita	Aceita	Aceita	Rejeita
	Z 3 / 4		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita

A.5. MPICH - Latência da Biblioteca X Latência Ponto-a-Ponto

Benchmark: **Ping e Ping-Pong**

Latência de...	**Dados Estatísticos**	**Tamanho da mensagem (bytes)**										
		0	200	400	600	800	1000	20000	40000	60000	80000	100000
1 - Biblioteca	Média	0.002067	0.003046	0.004207	0.004671	0.004897	0.005395	0.056565	0.136158	0.210212	0.279128	0.351612
	Desvio Padrão	0.000351	0.000030	0.000103	0.000100	0.000034	0.000034	0.012077	0.004798	0.002323	0.003299	0.005188
2 - Ponto a Ponto	Média	0.003938	0.004674	0.005423	0.006563	0.007772	0.008780	0.080828	0.155827	0.230149	0.301969	0.376324
	Desvio Padrão	0.000027	0.000054	0.000033	0.000048	0.000580	0.000057	0.000181	0.009271	0.000232	0.000178	0.000174
	Z 1 / 2	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita

A.6. LAM - Latência da Biblioteca X Latência Ponto-a-Ponto

Benchmark: **Ping e Ping-Pong**

Latência de...	**Dados Estatísticos**	**Tamanho da mensagem (bytes)**										
		0	200	400	600	800	1000	20000	40000	60000	80000	100000
1 - Biblioteca	Média	0.001121	0.002259	0.002672	0.003215	0.003707	0.004091	0.074753	0.147438	0.220609	0.291094	0.365122
	Desvio Padrão	0.000093	0.000221	0.000040	0.000030	0.000031	0.000106	0.006205	0.003137	0.006272	0.002326	0.001414
2 - Ponto a Ponto	Média	0.002357	0.003617	0.004793	0.006061	0.007032	0.008071	0.081729	0.155034	0.230945	0.302617	0.378284
	Desvio Padrão	0.000021	0.000084	0.000016	0.000960	0.000019	0.000014	0.000392	0.000111	0.000173	0.000083	0.010023
	Z 1 / 2	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita

A.7. LAM - *Throughput* Duplo

Benchmark: Ping-Pong

Modos de Comunicação	**Dados Estatísticos**	**Tamanho da mensagem (bytes)**										
		0	200	400	600	800	1000	20000	40000	60000	80000	100000
1 -Padrão	Média	0.0	0.221147	0.333785	0.395941	0.455030	0.495570	0.978844	1.032031	1.039208	1.057442	1.057406
	Desvio Padrão	0.0	0.004486	0.001083	0.031470	0.001205	0.000860	0.004592	0.000740	0.000777	0.000290	0.023766
2 - *Sendrecv*	Média	0.0	0.3480536	0.544032	0.693842	0.889877	0.980512	1.156019	1.211474	1.188633	1.162961	1.159090
	Desvio Padrão	0.0	0.022161	0.097927	0.251029	0.013965	0.023188	0.035070	0.019663	0.009265	0.003947	0.070097
3 - *Sendrecv replace*	Média	0.0	0.339558	0.531384	0.707026	0.926462	0.969696	1.129385	1.183160	1.186365	1.211022	1.204246
	Desvio Padrão	0.0	0.079269	0.003736	0.228940	0.605535	0.142672	0.063334	0.010663	0.021591	0.039681	0.042870
	Z 1 / 2		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 1 / 3		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 2 / 3		Rejeita	Rejeita	Rejeita	Rejeita	Rejeita	Aceita	Aceita	Rejeita	Aceita	Aceita

A.8. MPICH, LAM, PVM e UNIFY - Latência de Biblioteca

Benchmark: Ping

Modos de Comunicação	Dados Estatísticos	Tamanho da mensagem (bytes)										
		0	200	400	600	800	1000	20000	40000	60000	80000	100000
1 - Mpich	Média	0.002067	0.003046	0.004207	0.004671	0.004897	0.005395	0.056565	0.136158	0.210212	0.279128	0.351612
	Desvio Padrão	0.000351	0.000030	0.000103	0.000100	0.000034	0.000034	0.012077	0.004798	0.002323	0.003299	0.005188
2 - LAM	Média	0.001121	0.002259	0.002672	0.003215	0.003707	0.004091	0.074753	0.147438	0.220609	0.291094	0.365122
	Desvio Padrão	0.000093	0.000221	0.000040	0.000030	0.000031	0.000106	0.006205	0.003137	0.006272	0.002326	0.001414
3 - PVM	Média	0.001622	0.002773	0.003177	0.003666	0.004175	0.004659	0.067053	0.141787	0.216729	0.291370	0.370232
	Desvio Padrão	0.000023	0.000028	0.000113	0.000030	0.000031	0.000106	0.004739	0.004265	0.006700	0.007273	0.004725
4 - Unify	Média	0.005701	0.005850	0.001932	0.005390	0.003391	0.006435	0.064885	0.128795	0.189041	0.252756	0.315927
	Desvio Padrão	0.000198	0.000234	0.000971	0.025451	0.008931	0.035021	0.005383	0.028166	0.001874	0.000530	0.000435
	Z 1 / 2	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 1 / 3	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 1 / 4	Aceita	Aceita	Aceita	Rejeita	Rejeita	Rejeita	Aceita	Rejeita	Aceita	Aceita	Aceita
	Z 2 / 3	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Rejeita	Aceita
	Z 2 / 4	Aceita	Aceita	Aceita	Rejeita	Rejeita	Rejeita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 3 / 4	Aceita	Aceita	Aceita	Rejeita	Rejeita	Rejeita	Rejeita	Aceita	Aceita	Aceita	Aceita

A.9. MPICH, LAM, PVM e UNIFY - *Throughput*

Benchmark: Ping-Pong

Modos de Comunicação	Dados Estatísticos	Tamanho da mensagem (bytes)										
		0	200	400	600	800	1000	20000	40000	60000	80000	100000
1 - Mpich	Média	0.0	0.342319	0.590079	0.731372	0.823468	0.911109	1.979512	2.053559	2.085600	2.119422	2.125824
	Desvio Padrão	0.0	0.003739	0.003619	0.005293	0.041077	0.005798	0.004405	0.087394	0.002099	0.001249	0.000984
2 - LAM	Média	0.0	0.442294	0.667570	0.791883	0.910060	0.991141	1.957689	2.064063	2.078416	2.114884	2.114813
	Desvio Padrão	0.0	0.008973	0.002165	0.062940	0.002409	0.001721	0.009183	0.001479	0.001554	0.000579	0.047532
3 - PVM	Média	0.0	0.365797	0.576213	0.718562	0.794217	0.902527	1.939264	2.004510	2.054332	2.067564	2.085388
	Desvio Padrão	0.0	0.002040	0.002426	0.003098	0.114412	0.001806	0.022516	0.006455	0.002368	0.001853	0.007136
4 - Unify	Média	0.0	0.174159	0.306073	0.412512	0.499395	0.566331	1.183607	1.220398	1.263369	1.280664	1.298346
	Desvio Padrão	0.0	0.003190	0.003024	0.002880	0.004090	0.032707	0.004175	0.055011	0.002391	0.043812	0.005068
	Z 1 / 2		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Rejeita	Aceita	Aceita	Rejeita
	Z 1 / 3		Aceita	Aceita	Aceita	Rejeita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 1 / 4		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 2 / 3		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 2 / 4		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita
	Z 3 / 4		Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita	Aceita