

UNIVERSIDADE DO OESTE DE SANTA CATARINA  
CENTRO TECNOLÓGICO  
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**COMPARAÇÃO ENTRE BIBLIOTECAS PARA PROCESSAMENTO  
PARALELO COM BASE EM UM CLUSTER BEOWULF**

MONOGRAFIA SUBMETIDA À  
UNIVERSIDADE DO OESTE DE SANTA CATARINA PARA OBTENÇÃO DO  
GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO

**LUCIANO CARLOS FROSI**

CHAPECÓ (SC), DEZEMBRO DE 2000

**COMPARAÇÃO ENTRE BIBLIOTECAS PARA PROCESSAMENTO  
PARALELO COM BASE EM UM CLUSTER BEOWULF**

**LUCIANO CARLOS FROSI**

ESTA MONOGRAFIA FOI JULGADA PARA OBTENÇÃO DO TÍTULO DE  
**BACHAREL EM CIÊNCIA DA COMPUTAÇÃO**, ÀREA DE PROCESSAMENTO  
PARALELO E APROVADA PELO CURSO DE CIÊNCIA DA COMPUTAÇÃO

ORIENTADOR: Prof. Saulo Popov Zambiasi

COORDENADORA DO CURSO: Prof<sup>ª</sup>. Rosicler Fellipe

**BANCA EXAMINADORA**

PRESIDENTE: Prof. Saulo Popov Zambiasi

Prof. Almir Jr. Antunes

Prof. Jorge Di Domenico

# Sumário

<i>Sumário</i> .....	<i>ii</i>
<i>Lista de Figuras</i> .....	<i>v</i>
<i>Lista de Tabelas</i> .....	<i>vi</i>
<i>Lista de Gráficos</i> .....	<i>vii</i>
<b>RESUMO</b> .....	<b>viii</b>
<b>ABSTRACT</b> .....	<b>ix</b>
<b>1. INTRODUÇÃO</b> .....	<b>1</b>
<b>2. FUNDAMENTOS DE COMPUTAÇÃO PARALELA</b> .....	<b>5</b>
<b>2.1. Arquiteturas de Máquinas Paralelas</b> .....	<b>6</b>
2.1.1. Arquiteturas MIMD de Memória Compartilhada.....	9
2.1.2. Arquiteturas MIMD de Memória Distribuída.....	10
<b>2.2. Medidas de Desempenho</b> .....	<b>11</b>
2.2.1. Speedup.....	12
2.2.2. Eficiência.....	12
<b>2.3. Modelos de Programação Paralela</b> .....	<b>12</b>
2.3.1. Programação Baseada em Variáveis Compartilhadas.....	12
2.3.2. Programação Baseada em Passagem de Mensagem (Message Passing).....	13
2.3.3. Programação Baseada no Paralelismo dos Dados.....	15
<b>2.4. Concepção de Programas Paralelos</b> .....	<b>15</b>
<b>3. CLUSTERS BEOWULF</b> .....	<b>18</b>
<b>3.1. O Programa NASA HPCC</b> .....	<b>18</b>
<b>3.2. PoPC: Uma Pilha de PCs</b> .....	<b>19</b>
<b>3.3. Arquiteturas das Estações de Trabalho Beowulf</b> .....	<b>21</b>
3.3.1. Protótipos Usados nos Experimentos.....	21
<b>4. PVM</b> .....	<b>23</b>
<b>4.1. Componentes do PVM</b> .....	<b>23</b>
<b>4.2. Configuração do Ambiente</b> .....	<b>24</b>
<b>4.3. Criação Dinâmica de Processos</b> .....	<b>25</b>
<b>4.4. Comunicação</b> .....	<b>27</b>

4.5. Grupos Dinâmicos de Processos .....	30
4.6. Performance .....	32
4.7. Aplicação Exemplo .....	33
5. MPI .....	36
5.1. MPICH - Uma Implementação do Padrão MPI .....	37
5.2. Uma Breve Análise sobre MPI .....	38
5.3. Comunicação Ponto-a-Ponto .....	40
5.3.1. Modos de Comunicação.....	41
5.4. Comunicadores .....	42
5.5. Tipos Derivados com MPI_Type_struct() .....	45
5.6. Aplicação exemplo .....	48
6. PRINCIPAIS DIFERENÇAS ENTRE PVM E MPI.....	51
6.1. Portabilidade & Interoperabilidade.....	51
6.2. Máquina Paralela Virtual .....	52
6.2.1. Controle de Processos .....	52
6.2.2. Controle de Recursos .....	53
6.3. Tolerância à Falhas .....	54
6.4. Contexto de Comunicação .....	55
6.5. Tipos Definidos Pelo Usuário.....	56
7. TRABALHO IMPLEMENTADO .....	57
7.1. Implantação do Cluster Beowulf.....	57
7.2. Aplicação Implementada.....	58
7.2.1. Inicialização da Aplicação .....	59
7.2.2. Distribuição de Matrizes e Controle dos Processos Filhos .....	60
7.2.3. Processos Filhos.....	61
7.2.4. Aplicação Monoprocessada .....	62
8. COMPARAÇÃO ENTRE BIBLIOTECAS.....	63
8.1. Obtenção de Resultados .....	63
8.2. Apresentação dos Resultados Obtidos .....	64
8.3. Pontos Observados .....	68
8.3.1. PVM.....	68
8.3.2. MPI .....	69

<b>9. CONCLUSÃO.....</b>	<b>70</b>
<b>10. REFERÊNCIAS BIBLIOGRÁFICAS .....</b>	<b>72</b>

## Lista de Figuras

Figura 1.1 - Esquema das tendências de arquiteturas de processamento.....	02
Figura 2.1 - Computador SISD.....	07
Figura 2.2 - Computador SIMD.....	07
Figura 2.3 - Computador MIMD.....	08
Figura 2.4 - Modelo de arquitetura com memória compartilhada.....	09
Figura 2.5 - Modelo de arquitetura com memória distribuída.....	10
Figura 2.6 - Exemplo de uma operação com paralelismo de dados.....	15
Figura 3.1 - Um cluster Beowulf.....	20
Figura 5.1 - Comportamento não esperado do programa, causando deadlock no sistema .....	44
Figura 5.2 - Uso de comunicadores, evitando interceptação errônea de mensagens.....	43
Figura 6.1 - Mensagens enviadas com um contexto de comunicação.....	55

## Lista de Tabelas

Tabela 5.1 - Classificação e nomes de funções para envio e recebimento de mensagens .....	41
Tabela 8.1 - Média obtida sobre valores de três diferentes execuções .....	63
Tabela 8.2 - Speedup obtido com os algoritmos paralelos com base no cluster apresentado....	68

## Lista de Gráficos

Gráfico 8.1 - Computação paralela utilizando matrizes quadradas de ordem 400 X 400 elementos (em ponto flutuante).....	65
Gráfico 8.2 - Computação paralela utilizando matrizes quadradas de ordem 250 X 250 elementos (em ponto flutuante).....	66
Gráfico 8.3 - Computação paralela utilizando 100 matrizes quadradas de ordem variável.....	67

## RESUMO

Este projeto tem como objetivo o estudo e a implantação de um Supercomputador Beowulf na UNOESC - CHAPECÓ, bem como a implementação de aplicações paralelas sobre tal cluster, especificamente utilizando as bibliotecas de processamento paralelo PVM e MPI, a fim de posterior comparativo entre ambas, no que diz respeito à performance e peculiaridades de implementação.

Beowulf é um tipo de computador de alto desempenho com processamento maciçamente paralelo construído principalmente com componentes comuns de hardware e rodando um sistema operacional que atenda as especificações de “free software”, como Linux ou FreeBSD. Consiste, então, em um cluster de PCs interconectados através de uma rede de alta velocidade com a finalidade de processar aplicações de modo paralelo e com desempenho superior, que atenda ao projeto original do Centro de Excelência em Dados Espaciais e Informáticas (CESDIS), subdivisão da NASA [BEO 00A].

O princípio de funcionamento consiste em uma tarefa ser dividida em partes independentes, distribuídas entre os vários computadores que fazem parte do cluster, onde as informações são processadas. Para fazer um programa para esse tipo de processamento usa-se uma biblioteca específica para passagem de mensagens entre os nós do cluster. Os dois grandes padrões de bibliotecas de passagem de mensagens são a Parallel Virtual Machine (PVM) e a Message Passing Interface (MPI). PVM foi o padrão de facto por muito tempo, até que MPI surgiu, sendo esta última ainda uma incógnita para muitos especialistas na área, pois possui uma filosofia diferente da primeira em muitos aspectos, os quais serão abordados no decorrer do trabalho.

O presente projeto, tem assim, como objetivo, a implementação de uma mesma aplicação em três diferentes formas: uma convencional monoprocessada e as duas outras em ambas as bibliotecas, de modo que estas últimas sejam executadas em paralelo com base no cluster Beowulf, a fim de posterior coleta de informações a respeito dos resultados e comparação entre os mesmos. Espera-se, dessa forma, que este sirva como uma referência para trabalhos acadêmicos, científicos ou comerciais na área em questão.

## ABSTRACT

This project has as objective the study and the implantation of a Supercomputer Beowulf in UNOESC - CHAPECÓ, as well the implementation of parallel applications on such cluster, specifically using the libraries of parallel processing PVM and MPI, in order to subsequent comparative between both, in what concerns the performance and implementation peculiarities.

Beowulf is a type of computer of high performance with a massively parallel processing built mainly with components common of hardware and executing an Operating System that assists the specification of “free software” like Linux or FreeBSD. It consists, then, in a cluster of PCs interconnected though of high-speed network with the purpose of processing applications in a parallel way and with superior performance, that assists to the original project of the Center of Excellency in Space Data and Information Sciences (CESDIS), subdivision of NASA [BEO 00A].

Basically, the operation consists of a task to be divided in independent parts, distributed among the several computers that compose the cluster, where the information are processed. To do a program for this type of processing, make use of specific library for message passing among of knots of the cluster. The two mains standards of libraries of message passing are Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). PVM was the facto standard for a long time, until that MPI appeared, being still this last un-know for many specialists in the area, because it possesses a philosophy different from the first in many aspects, which will be approached in elapse of the work.

The present project, has like this, as objective, the implementation of the same application in three different forms: a conventional monoprocess and the other two in both libraries, so that these last ones are executed in parallel with base in the cluster Beowulf, in order to subsequent collect of information regarding the results and comparison among the same ones. Expect, in that way, that this can be useful as a reference for academic, scientific or commerce works in the area in subject.

## 1. INTRODUÇÃO

Apesar da constante evolução dos processadores, que vem dobrando a sua capacidade a cada 18 meses (Lei de Moore), continua havendo a necessidade de grandes capacidades de processamento por parte de muitas organizações, sejam elas comerciais ou científicas. Além disso, limites físicos de construção de processadores vem impondo fortes restrições à essa evolução, que começa a diminuir cada vez mais seu ritmo. Em contrapartida, há muitos estudos que tentam provar que um computador permanece ocioso a maior parte do tempo de sua vida útil. Tempo esse, que poderia ser ocupado por pequenos processos rodando em background, os quais seriam parte de uma grande aplicação paralela, tirando proveito dos ciclos “perdidos” de máquinas ociosas. É com base nesses preceitos que vem surgindo sistemas que ao longo dos últimos anos tentam responder a esses requisitos.

Esses sistemas são comumente classificados como sistemas paralelos e sistemas distribuídos, ambos compostos por vários processadores que operam concorrentemente, cooperando na execução de uma determinada tarefa. Nas chamadas arquiteturas paralelas o objetivo principal é o aumento da capacidade de processamento, utilizando o potencial oferecido por um grande número de processadores. A comunicação dos processadores é realizada através de redes especiais de conexão ou por meio de uma memória compartilhada, implicando em estruturas fisicamente concentradas. Por outro lado, nas arquiteturas distribuídas os atrativos principais são a flexibilidade, obtida pela integração de computadores de diversos tipos em um mesmo sistema, e a segurança, obtida com serviços de alta disponibilidade e tolerância à falhas, por exemplo [CUL 99].

O desafio é dominar essas tecnologias e usá-las na construção de sistemas que atendam às necessidades atuais. A figura 1.1 classifica as tendências que vem se destacando no campo da computação paralela, segundo a sua arquitetura interna:



**Figura 1.1** Esquema das tendências de arquiteturas de processamento.

Uma abordagem mais detalhada sobre tais arquiteturas será apresentada mais adiante. Neste momento, basta mencionar que dentre estas soluções, a tecnologia de cluster é relativamente nova, e vem despontando como uma tendência, graças a suas vantagens visíveis quanto a custos e escalabilidade.

Desta nova tendência destaca-se uma arquitetura projetada pelo CESDIS (Center of Excellence in Space Data and Information Sciences), subdivisão da NASA. Este projeto, apelidado de Beowulf em homenagem ao herói da literatura medieval inglesa, estuda as vantagens de utilizar PCs interligados construídos a partir de componentes do mercado doméstico e utilizando software gratuito, sendo essa a arquitetura abordada por este trabalho.

Basicamente, qualquer grande necessidade computacional em que seja possível o código ser paralelizado é uma candidata a utilizar uma “arquitetura Beowulf”. Segurança, por exemplo, é uma das áreas que podem ser beneficiadas com processamento paralelo. Uma das razões para tal fato reside na grande capacidade de processamento necessária para a quebra de cifras por “brute force attack” ou por outros métodos. De mesmo modo, um supercomputador Beowulf pode também ser utilizado para pesquisas intensivas a grandes bases de dados para computação numérica, principalmente no campo da Inteligência Computacional, como resolução de redes neurais e algoritmos genéticos. Finalmente, a utilização de um cluster justifica-se também na abordagem da computação gráfica. Neste campo, é bastante comum o tempo de processamento ser uma limitação à evolução da qualidade dos projetos.

Para que uma aplicação possa ser “dividida” em várias tasks<sup>1</sup> e estas serem processadas de forma simultânea no cluster, normalmente é usado um tipo de biblioteca específica para passagem de mensagens. Embora nada impeça que o programador use sockets diretamente, não há motivo para aumentar a complexidade do programa em função de detalhes de baixo nível. O uso de uma biblioteca permite que o programador só precise se preocupar com instruções como “envie uma mensagem” e não com detalhes mais específicos de como a mensagem será enviada entre as tasks **[REV 00]**.

Há dois grandes padrões de bibliotecas de processamento paralelo, que são o PVM (Parallel Virtual Machine) e o MPI (Message Passing Interface). PVM vem sendo o padrão de facto já há algum tempo, mas a MPI vem despontando como uma nova tendência, com paradigmas de comunicação que são fundamentalmente diferentes das atuais em determinados pontos.

Tanto no envio de uma mensagem como no seu recebimento, vários fatores estão envolvidos e devem ser considerados, e ambas bibliotecas possuem características peculiares e visões diferentes em alguns quesitos, como por exemplo, controle de processos, interoperabilidade entre as máquinas que compõem o cluster, agregação de dados, entre outros.

Assim, o presente trabalho pretende responder questões relacionadas às características oferecidas por PVM e características oferecidas por MPI, mostrando quais situações uma biblioteca pode ser favorável sobre outra. Programadores podem, então, avaliar as necessidades de suas aplicações e decidirem de acordo sobre qual sistema seguir. O resultado é a criação de um supercomputador Beowulf na UNOESC, o qual poderá beneficiar não somente o curso de Ciência da Computação, mas demais cursos que necessitem realizar experimentos que exijam alto poder de processamento, dando ao programador uma referência comparativa entre as bibliotecas de processamento paralelo necessárias para tal feito.

---

<sup>1</sup> Task é o termo mais comum em processamento paralelo para se referenciar a processos criados pelo usuário ou por uma aplicação. Durante este trabalho, os termos task, tarefa, subtarefa e processo serão usados indistintamente.

Para atingir tais objetivos, este trabalho traz, no Capítulo 2, uma introdução às arquiteturas paralelas existentes, bem como formas e modelos de programação paralela, sobre as quais se faz necessário um conhecimento prévio, para que se possa discernir sobre tais tecnologias. Também é abordada a concepção de aplicações utilizando essa forma de programação.

No Capítulo 3, é realizada uma abordagem a respeito de clusters Beowulf, desde a criação e evolução do projeto, até sua finalidade e filosofia de funcionamento. Discussões a respeito dos critérios para que um cluster possa ser chamado de Beowulf, também se fazem presentes neste capítulo.

As bibliotecas de programação paralela PVM e MPI, são apresentadas, respectivamente, nos Capítulos 4 e 5, onde são abordadas as formas de programação com estas bibliotecas, bem como as características principais de cada uma.

Depois de apresentadas as bibliotecas, o trabalho realiza, no Capítulo 6, uma comparação discursiva entre as mesmas, onde são discriminadas suas principais diferenças e o que uma pode oferecer de vantagem em relação à outra.

O Capítulo 7 apresenta a aplicação implementada pelo acadêmico, discutindo a lógica dos algoritmos e como essa aplicação foi paralelizada. Também neste capítulo é apresentado o cluster Beowulf implantado por este acadêmico.

Finalmente, no Capítulo 8, são apresentados os resultados obtidos com as diferentes formas de implementação da mesma aplicação. Aqui, os resultados são comparados e debatidos, apontando-se em que situações uma pode desempenhar uma performance superior às outras.

As conclusões formuladas pelo acadêmico, com base nas informações obtidas com o presente trabalho, compõem o Capítulo 9.

## 2. FUNDAMENTOS DE COMPUTAÇÃO PARALELA

As aplicações informáticas, tanto as científicas como as comerciais, tendem a tornar-se maiores, quer no volume de dados manipulados, quer na complexidade do processamento que lhes está associado. Esta demanda crescente de processamento tem motivado a evolução dos computadores, viabilizando implementações de aplicações que envolvem um intenso processamento e grandes volumes de dados.

Segundo [MID 96], na busca pelo alto desempenho, tem-se as alternativas:

1. **Aumentar o desempenho dos monoprocessadores:** este aumento pode ser obtido através de:
  - a) Aumento da velocidade de clock. Esta alternativa envolve o desenvolvimento de tecnologia de confecção e circuitos integrados, trazendo entretanto, problemas, tais como o aumento de temperatura;
  - b) melhoria na arquitetura. Este objetivo motivou o surgimento dos processadores RISC, vetoriais e super-escalares;
  - c) melhoria no acesso à memória, por exemplo, através da exploração da hierarquia de memória, utilizando registradores, memória cache e memória principal.
2. **Construir computadores especializados em determinados problemas:** esta é uma abordagem pouco flexível, já que o tempo necessário e custos dispendidos para tal abordagem a torna uma escolha pouco atraente para a maioria das instituições, sejam elas comerciais ou científicas.
3. **Usar um computador paralelo:** dividindo o cálculo a ser efetuado em várias partes e entregando cada parte a um processador diferente e específico, diminui a necessidade de processadores com desempenho e preço tão elevados.

A solução de usar um computador paralelo para suportar aplicações que movimentam grandes volumes de dados e/ou efetuam grande número de operações sobre eles é a mais promissora das três soluções apontadas, pois não depende de

melhorias oferecidas por fornecedores em particular, nem de tempo e custos altíssimos para alcançar uma performance aceitável, além de boa flexibilidade e escalabilidade.

Segundo [FOS 95], citado em [MED 98], um *computador paralelo* é um conjunto de processadores que têm a capacidade de trabalhar de forma cooperativa para resolver um problema. Mas esta definição é suficientemente vaga para incluir também os *sistemas distribuídos* e para levantar a habitual questão do que é que distingue um sistema paralelo de um sistema distribuído. A fronteira é difícil de traçar, mas tem a ver com os objetivos que cada um deles pretende atingir. Num sistema paralelo, a ênfase é posta no desempenho da execução, enquanto num sistema distribuído o objetivo é suportar um conjunto de serviços de forma transparente à sua localização, preocupando-se com questões como a tolerância a falhas, a proteção e autenticação de clientes e servidores. Esta maior funcionalidade dos modelos de programação dos sistemas distribuídos penaliza-os, normalmente, em termos de desempenho.

A implementação de algoritmos sobre um computador paralelo, requisita recursos de programação paralela que permitam expressar o paralelismo e incluir mecanismos para sincronização e comunicação. A programação paralela é a programação concorrente orientada para computadores paralelos, incorporando os seus requisitos de sincronização e comunicação, visando a utilização adequada dos recursos de processamento para otimizar o seu desempenho [MID 96].

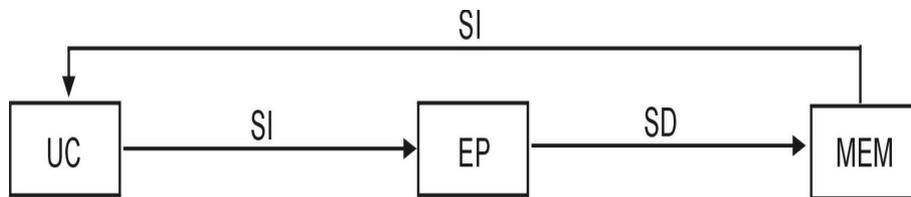
## 2.1. Arquiteturas de Máquinas Paralelas

Embora o processamento paralelo venha sendo usado há muitos anos em muitos sistemas, há alguns conceitos que não são familiares à maioria dos usuários. Assim, antes de prosseguir com o trabalho, é importante o entendimento e familiarização de alguns conceitos e termos usados.

Em [FLY 72], Flynn caracterizou diversos modelos de arquiteturas de computadores, segundo o número de fluxos de dados e de instruções existentes em cada instante, produzindo assim as seguintes quatro classes de computadores:

- **Uma Seqüência de Instruções, Uma Seqüência de Dados (SISD).**  
Corresponde aos computadores seqüenciais convencionais nos quais só existe

uma única unidade de controle que decodifica seqüencialmente instruções que operam sobre um conjunto de dados (Figura 2.1).

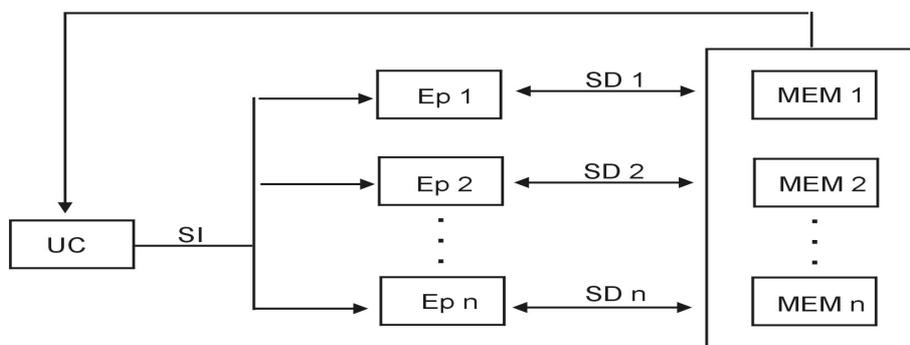


UC: Unidade de Controle;  
 EP: Elemento Processador;  
 MEM: Memória;  
 SI: Seqüência de Instruções;  
 SD: Seqüência de Dados.

**Figura 2.1.** Computador SISD

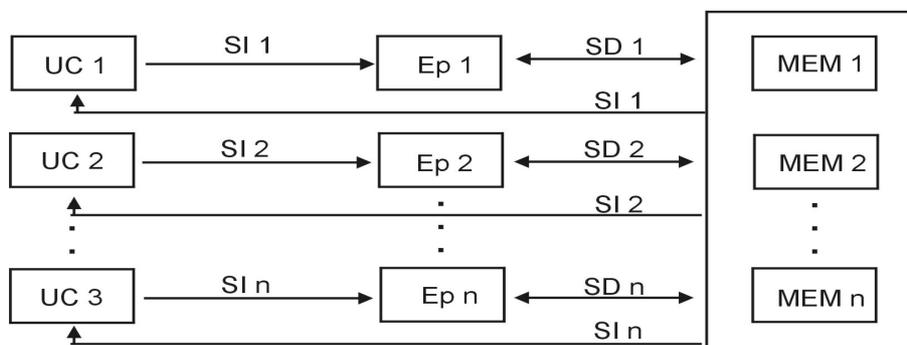
□ **Uma Seqüência de Instruções, Múltiplas Seqüências de Dados (SIMD).**

Refere-se ao modelo de execução paralela na qual vários elementos processadores são conectados por uma única unidade de controle (Figura 2.2). Esta decodifica seqüencialmente instruções e as transmite para todos os elementos processadores, os quais executam essa mesma instrução ao mesmo tempo, mas cada um sobre seus próprios dados. Este modelo naturalmente fita o conceito de efetuar a mesma operação em todos os elementos de um vetor de uma só vez, ou seja, é freqüentemente associado com manipulação de vetores e matrizes.



**Figura 2.2.** Computador SIMD

- **Múltiplas Seqüência de Instruções, Uma Seqüência de Dados (MISD).** É o contrário da arquitetura SIMD. Não existem computadores enquadrados nesta categoria.
- **Múltiplas Seqüências de Instruções, Múltiplas Seqüências de Dados (MIMD).** Refere-se ao modelo de execução paralela na qual cada processador está essencialmente agindo independentemente, havendo portanto múltiplos fluxos de instruções e de dados. Este tipo de máquina tem um campo de aplicação mais vasto do que as máquinas SIMD, podendo, se necessário, emular o modo de funcionamento destas (Figura 2.3).



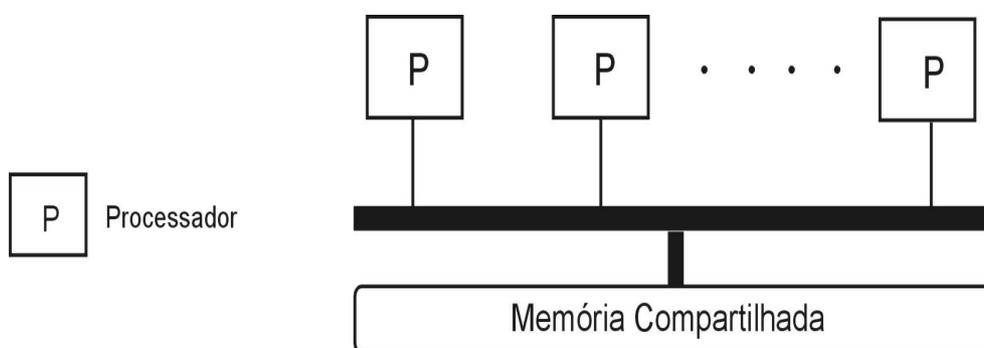
**Figura 2.3.** Computador MIMD

Embora a classificação de Flynn seja conveniente numa primeira aproximação, é bastante imprecisa para classificar as variedades de multiprocessadores existentes [AMO 88]. A categoria MIMD engloba uma grande diversidade de máquinas e tem havido várias tentativas de as classificar. Uma das formas mais usadas é uma classificação que usa como critério a forma como a memória central está fisicamente organizada e o tipo de acesso que cada processador tem à totalidade da memória, classificando, assim, esta arquitetura em computadores MIMD de memória compartilhada e computadores MIMD de memória distribuída. Apresentam-se, a seguir, as duas classes de arquiteturas MIMD:

### 2.1.1. Arquiteturas MIMD de Memória Compartilhada

Nesta classe incluem-se todas as máquinas com múltiplas unidades processadoras que compartilham um espaço de endereços de memória comum (máquinas multiprocessadas), como ilustrado na figura 2.4.

Além de ser pouco escalável e de difícil manutenção, esse tipo de arquitetura conduz a situações em que um processador, ao acessar a memória comum, pode entrar em conflito com outros processadores. A maior ou menor gravidade destes conflitos depende, no que diz respeito ao hardware, da sofisticação do dispositivo usado para interligar os processadores e as unidades de memória. Também o uso de “caches” associados a cada processador permite diminuir o número de conflitos. É necessário também que o Sistema Operacional que esteja rodando nessas máquinas suporte este tipo de arquitetura e implemente mecanismos mais sofisticados de escalonamento de processos entre os processadores. Faz-se necessário também o uso de semáforos ou monitores, garantindo, dessa forma, a exclusão mútua entre processos.



**Figura 2.4.** Modelo de arquitetura com memória compartilhada

As máquinas presentes nesta classe geralmente são compostas por processadores de uso comum e é habitual distinguir duas categorias:

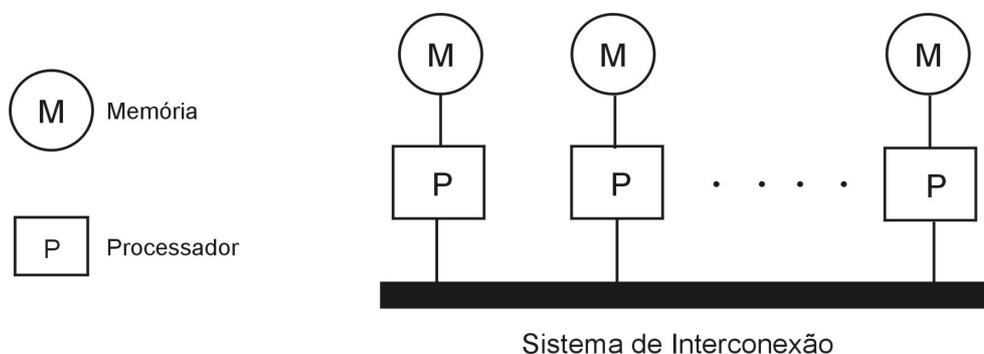
- **UMA (Uniform Memory Access):** são máquinas em que é garantido um igual tempo de acesso à memória. Normalmente à esta categoria está associado hardware de interligação pouco sofisticado e que impõe um limite ao número

máximo de processadores (inferior a algumas dezenas) que este tipo de arquitetura suporta. Se esse número é excedido, a quantidade de conflitos no acesso à memória elimina potenciais ganhos devido à adição de mais processadores.

- **NUMA** (Non-Uniform Memory Access): Nestas arquiteturas, o uso de múltiplos níveis de cache e hardware de interligação muito sofisticado, permite reduzir de tal forma a contenção no acesso à memória, que é possível construir máquinas com centenas de processadores. No entanto, hardware deste tipo tende a ser extremamente caro e complexo.

### 2.1.2. Arquiteturas MIMD de Memória Distribuída

Nesta classe incluem-se as máquinas formadas por várias unidades processadoras, cada uma com a sua memória privada (chamando-se **nó** a uma unidade constituída por um processador, memória local e dispositivos de entrada/saída), sendo geralmente chamados de *multicomputadores*. Neste caso não existe qualquer tipo de memória comum e a comunicação entre os diferentes nós do sistema é feita através de dispositivos físicos de entrada/saída. Com os progressos realizados nas formas de encaminhamento de dados entre nós, é possível construir máquinas deste tipo com milhares de unidades processadoras. Um exemplo desse tipo de arquitetura é ilustrado na figura 2.5.



**Figura 2.5.** Modelo de arquitetura com memória distribuída.

Nestas máquinas, os vários nós trocam informações enviando seqüências de bytes através da sua rede de interligação. O tempo gasto para enviar N bytes pode ser estimado pela fórmula:

$$\text{tempo total} = \text{latência} + (N / \text{largura de banda})$$

em que a latência é o tempo necessário para iniciar uma comunicação e a largura de banda é a taxa máxima de transmissão de informação numa ligação direta entre dois nós. Por mais eficaz que seja a interligação, o tempo para transferir N bytes entre dois nós será sempre muito maior (entre 10 e 1000000 de vezes) do que o tempo que demora para transferir os mesmos N bytes entre o processador e sua memória principal [MED 98].

Um multicomputador geralmente possui seus nós em placas separadas ligadas por barramentos/chaveadores contidas em um único gabinete, como os sistemas Intel Paragon, Meiko-CS2 e CRAY T3D. Esses sistemas são comumente referenciados como sistemas MPP (Massively Parallel Processors). Uma segunda forma, bastante interessante sob vários aspectos, é a implementação de multicomputadores como redes locais de estações de trabalho ou PC's, comumente chamados de *cluster*. Esta última é a forma mais barata e comum de implementar um multicomputador e é a arquitetura proposta e estudada por este projeto.

## 2.2. Medidas de Desempenho

Medidas de desempenho, que permitam a análise do ganho obtido com o aumento do total de processadores utilizados, são necessários. Duas medidas usuais são: o "speedup" (ganho de desempenho) e a eficiência.

### 2.2.1. Speedup

O speedup ( $S$ ) obtido por um algoritmo paralelo executando sobre  $p$  processadores é a razão entre o tempo levado por aquele computador executando o algoritmo serial mais rápido ( $T_s$ ) e o tempo levado pelo mesmo computador executando a versão paralela deste algoritmo, usando  $p$  processadores ( $T_p$ ). Ou seja, esta medida é usada para verificar o ganho de desempenho de uma aplicação paralela sobre sua versão serial.

$$S = \frac{T_s}{T_p}$$

### 2.2.2. Eficiência

A eficiência ( $E$ ) é a razão entre o speedup obtido com a execução com  $p$  processadores e  $p$ . Esta medida mostra o quanto o paralelismo foi explorado no algoritmo. Geralmente esta medida é usada quando se quer comparar o desempenho da mesma aplicação paralela com diferentes configurações de número de processadores, ou quando se quer comparar diferentes algoritmos para paralelizar a mesma aplicação.

$$E = \frac{S}{p}$$

## 2.3. Modelos de Programação Paralela

### 2.3.1. Programação Baseada em Variáveis Compartilhadas

Esse modelo de programação é implementado em sistemas MIMD com memória compartilhada (ou sistemas multiprocessados) e baseia-se no fato de que os

processos acessam regiões de memória comuns à seus espaços de endereçamento, cooperando entre si utilizando dados compartilhados.

A programação paralela utilizando variáveis compartilhadas permite implementações com menor complexidade em relação aos modelos com passagem de mensagem – modelo que será visto posteriormente. Entretanto, as leituras e escritas dessas variáveis devem ser feitas considerando algumas restrições. Uma leitura e escrita ou múltiplas escritas não podem ser executadas simultaneamente. Isso exige a utilização de uma seção crítica envolvendo o acesso a variáveis compartilhadas. Semáforos ou monitores são mecanismos que implementam a exclusão mútua, que garantem que uma seqüência de comandos seja executada exclusivamente por um processo.

### **2.3.2. Programação Baseada em Passagem de Mensagem (Message Passing)**

Bibliotecas de passagem de mensagem permitem implementar programas que executem em sistemas de memória distribuída (multicomputadores). Estas bibliotecas provêm rotinas para iniciar e configurar o ambiente de execução bem como enviar e receber pacotes de dados entre os processadores do sistema. Atualmente, os dois mais populares sistemas de alto nível para passagem de mensagem em aplicações científicas e de engenharia são o PVM (Parallel Virtual Machine) da OAK Ridge National Laboratory e o MPI (Message Passing Interface) definido pelo MPI Forum.

Para que uma aplicação possa ser executada por um computador paralelo com memória distribuída é necessário trocar trechos de seu código ou dados entre os processadores. Algumas bibliotecas de programação utilizam o modelo SPMD (single Program, Multiple Data), enquanto que outras permitem que trechos heterogêneos de código sejam distribuídos pelos processadores. O modelo SPMD é um modelo de programação onde várias instâncias do mesmo código são geradas, mas cada instância é uma task independente atuando sobre um trecho de código distinto das demais. Trechos heterogêneos de código são programas diferentes entre si, mas que se conhecem e trocam informações através de mensagens.

Nesses chamados multicomputadores, os processadores encontram-se fisicamente distribuídos, e somente podem acessar a sua própria memória local. Cada

par composto de um processador e sua respectiva memória local constituem um **nó**. Todos os nós de um multicomputador são interligados por algum tipo de barramento, ou sistema de interconexão, através do qual pode-se enviar e receber mensagens. Assim, o modelo mais elementar de programação de multicomputadores é aquele no qual os processos que compõem um programa paralelo são distribuídos pelos nós da rede, e trocam *mensagens* entre si. Esta modalidade é comumente chamada de programação distribuída.

Uma troca de mensagem envolve pelo menos dois processos, o *transmissor* que envia a mensagem, e o *receptor*, que a recebe. Geralmente isto é feito através de primitivas do tipo SEND e RECEIVE, em duas principais modalidades: a comunicação síncrona e a comunicação assíncrona.

- **Comunicação síncrona:** nesta modalidade, o transmissor envia a mensagem para o receptor, e aguarda até que este último sinalize o recebimento da mesma. Se o receptor não estiver pronto para receber a mensagem, o transmissor é bloqueado temporariamente. Se o receptor quiser receber a mensagem antes que esta tenha sido enviada, ele também será bloqueado temporariamente. Quando ambos estiverem prontos, e só então, o transmissor envia a mensagem, e o receptor envia de volta um sinal confirmando seu recebimento. Imediatamente os dois são desbloqueados e podem seguir seu fluxo de execução normalmente.
- **Comunicação assíncrona:** esta modalidade permite que o transmissor envie a mensagem, e prossiga em seu fluxo normal de execução sem sofrer nenhum tipo de bloqueio. Caso o receptor ainda não esteja pronto para receber a mensagem, esta deve permanecer armazenada temporariamente em um buffer. Caso o receptor esteja pronto para receber uma mensagem que ainda não foi enviada, ele então deverá permanecer bloqueado temporariamente. Os sistemas que implementam esta modalidade geralmente oferecem primitivas adicionais que apenas verificam se alguma mensagem chegou ou não, sem bloquear o processo receptor. Apesar desta modalidade aparentar melhor desempenho porque não impõe tantas situações de bloqueio, existem custos adicionais para fazer o tratamento de buffers que são necessários.

### 2.3.3. Programação Baseada no Paralelismo dos Dados

O paradigma de programação paralela baseada no paralelismo dos dados representa uma forma de exploração de operações simultâneas sobre grandes conjuntos de dados. Este estilo de programação é adotado em máquinas contendo centenas a milhares de processadores. A concorrência então, não é especificada na forma de diferentes operações simultâneas, mas na aplicação da mesma operação sobre múltiplos elementos de uma estrutura de dados. Um exemplo de uma destas operações é “some 2 sobre todos os elementos da matriz X”, como ilustrado na figura 2.6. Este estilo de programação é adotado sobre máquinas paralelas do tipo SIMD, ou MIMD com memória distribuída (multicomputadores, clusters) programadas sob o estilo SPMD. Uma das primeiras máquinas a adotarem este paradigma de programação foi o Connection Machine Modelo CM-1, introduzido pela Thinking Machines Corporation em 1986.



**Figura 2.6:** Exemplo de uma operação com paralelismo de dados.

## 2.4. Concepção de Programas Paralelos

Dada uma descrição do problema a resolver, a tarefa de criar um programa paralelo envolve identificar as atividades que podem ser executadas em paralelo e decidir sobre a melhor maneira de distribuir essas atividades pelos vários processadores bem como os dados que são manipulados. É, por outro lado, necessário

permitir o acesso aos dados pelos vários processadores, bem como a comunicação e sincronização entre eles. Estas operações têm, na maior parte dos casos, de ser feitas explicitamente pelo programador.

Descreve-se, a seguir, as várias fases da criação de um programa paralelo, segundo uma metodologia que se baseia nas propostas de Foster em [FOS 95] e Culler em [CUL 97]:

- **Decomposição:** esta fase permite definir as várias atividades que podem ser desempenhadas em paralelo. Esta decomposição deve maximizar o número de atividades e permitir a sua execução paralela. Esta fase é muito dependente do problema a resolver.
- **Comunicação:** nesta fase especifica-se a transferência de dados entre as atividades. Esta fase depende do modelo de programação e da arquitetura. O principal objetivo é reduzir os custos associados à comunicação e sincronização entre atividades.
- **Aglomerção:** na fase de decomposição procurou-se decompor a aplicação no maior número de atividade possível. Em muitos casos, a correspondência direta entre uma atividade a ser executada e um processo não é suficientemente eficaz. Tal deve-se que a granularidade<sup>2</sup> do paralelismo torna-se demasiado fina, isto é, o tempo que uma atividade demora a executar é pequeno quando comparado com a latência da comunicação. Nesta fase procura-se juntar várias atividades num único processo, de forma a atingir uma adequada granularidade. Nota-se também que, aumentando muito a granularidade, o número de processos que podem ser executados num dado instante diminui e pode haver situações em que há processadores não utilizados.
- **Mapeamento:** os objetivos dessa fase são os de colocar os vários processos nos processadores de forma a:

---

<sup>2</sup> Granularidade, em computação paralela, diz respeito ao grau de processamento de uma determinada tarefa, ou seja, quanto maior a quantidade de processamento que uma tarefa executa, sem necessitar para isso desviar sua atenção ou comunicar-se com outra tarefa, diz-se que mais grossa é a sua granularidade.

- Minimizar as comunicações, isto é, os processos que se comunicam com frequência devem ser colocados no mesmo processador;
- Maximizar a concorrência, o que se consegue colocando os processos que sejam independentes uns dos outros em processadores diferentes;
- Equilibrar a carga entre os vários processadores.

Não há uma regra simples para a criação de algoritmos paralelos. Alguns autores, como os acima citados, já tentaram descrever seqüências de passos para que essa tarefa fosse efetuada, mas a verdade é que cada caso é diferente dos demais e depende de avaliações a respeito das necessidades, exigências, condições de trabalho (hardware e software) e, principalmente, da aplicação propriamente dita.

### **3. CLUSTERS BEOWULF**

Um dos mais notáveis avanços tecnológicos dos dias atuais, tem sido o crescimento da performance computacional dos PCs (Personal Computers). Nos últimos cinco anos, microprocessadores de estações workstations científicas (basicamente processadores RISC) tiveram um aumento de performance na ordem de 50% ao ano. Mas isso tem sido facilmente ultrapassado pelo aumento de performance da classe dos microprocessadores de PCs, os quais tem aumentado sua performance na ordem de duas vezes ao ano, nos últimos quatro anos.

A verdade é que o mercado de PCs é maior que o mercado de workstations, permitindo que o preço de um PC decresça, enquanto sua performance aumenta substancialmente, sobrepondo, em muitos casos, a performance de máquinas workstations dedicadas.

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

No início de 1994, o projeto Beowulf foi iniciado sobre o patrocínio da NASA HPCC (High Performance Computing and Communications) com o ESS Project (Earth and Space Sciences) para investigar o potencial de clusterizar PCs a fim de se efetuar tarefas computacionais importantes, sem um grande custo. Em Outubro de 1996, foi anunciado que o sistema Beowulf havia excedido a ordem dos Gigaflops à um custo total de \$50.000. Esta foi uma inovação em performance e preço que teve implicações importantes em uma larga gama de aplicações industriais e científicas.

#### **3.1. O Programa NASA HPCC**

O programa NASA HPCC nasceu em janeiro de 1992, com o objetivo de alcançar e avançar o estado de processamento massivamente paralelo (MPP) e aplica-lo ao maior número de problemas computacionais importantes da NASA, mais objetivamente nas aerociências computacionais (CAS) e no Projeto ESS (Earth and

Space Sciences). O Projeto ESS representa um domínio computacional que inclui manipulação direta de grande número de dados por cientistas. Como os estudos nesta área partem de simulações de fenômenos físicos, evolução das galáxias, convenções do plasma na corona solar, entre outros, cientistas precisam adquirir, examinar, explorar, manipular, visualizar, e às vezes, transformar grandes coleções de dados complexos. Na maioria das vezes, isso envolve atividades de computação numérica intensiva e movimentação de grandes volumes de dados. O Projeto Beowulf Parallel Workstation foi iniciado para atender estes requisitos.

### 3.2. PoPC: Uma Pilha de PCs

PoPC (Pile-of-PCs) é o termo usado hoje para descrever uma montagem solta de PCs, ou, mais precisamente, um cluster de PCs, aplicado na resolução de um ou mais problemas. Segundo [BEC 99], é similar a um cluster de workstations (COW), mas tem o diferencial de enfatizar:

- uso de componentes disponíveis no mercado de massa;
- processadores dedicados, ao invés de usar tempo ocioso das estações;
- uma rede de sistema privada.

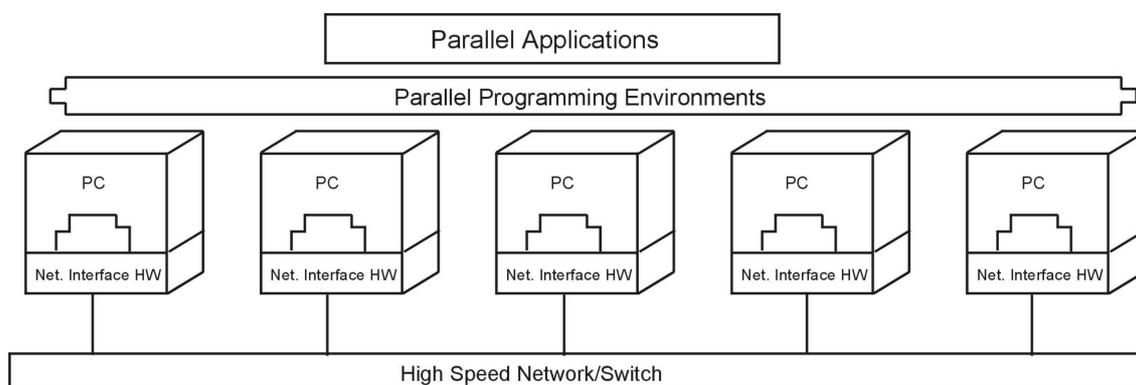
Além disso, para um cluster de PCs ser considerado um Beowulf (figura 3.1), precisa atender às seguintes características:

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

Como vantagens dessa filosofia de trabalho, os autores enumeram:

- nenhum fornecedor possui direitos sobre o produto. Sistemas podem ser construídos usando componentes de diversas origens, graças ao uso de interfaces padrão, tais como IDE, PCI, SCSI;
- pode-se tomar vantagem das rápidas evoluções tecnológicas, permitindo adquirir sistemas mais recentes, melhores, a menores preços, capazes de continuar rodando o mesmo software. Os primeiros sistemas construídos baseavam-se no processador 80486DX4-100, enquanto os mais recentes usam Pentium II e III;
- os sistemas podem ser montados e modificados ao longo do tempo, de acordo com as necessidades e recursos (inclusive financeiros) do usuário, sem depender de configurações disponíveis de um vendedor;
- Beowulf usa software disponível livremente, tão sofisticado, robusto e eficiente quanto o comercial.

Sendo assim, uma característica chave de um cluster Beowulf, é o uso do Sistema Operacional Linux, assim como de bibliotecas para troca de mensagens (PVM e MPI) de livre distribuição. Isto permitiu fazer alterações no Linux para dota-lo de novas características que facilitaram a implementação de aplicações paralelas.



**Figura 3.1.** Um cluster Beowulf

### 3.3. Arquiteturas das Estações de Trabalho Beowulf

Existem dois sistemas “de referência”: o primeiro montado no CESDIS (Center of Excellence in Space Data and Information Sciences) da NASA, em 1994, e seu sucessor, de 1996, também da NASA. Ambos foram testados com a mesma base de software e com os mesmos esquemas de interconexão, mas usam hardware diferente. Os próprios autores lembram que não há dois Beowulfs absolutamente iguais, em decorrência de cada um deles ter sido construído sob diferentes aspectos. Outras implementações feitas fora da NASA usam variações na topologia da rede de interconexão e nos microprocessadores, mas sempre usando componentes disponíveis no mercado, mantendo a filosofia “faça você mesmo o seu supercomputador”.

#### 3.3.1. Protótipos Usados nos Experimentos

##### - Beowulf Parallel Workstation (1994)

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

- processador Intel 80486DX4 (100 MHz);
- barramento local padrão VESA;
- 16MB de memória RAM;
- 16KB de memória cache primária (interna ao 486);
- 256Kb de memória cache secundária;
- um disco IDE de 512MB;
- duas interfaces Ethernet de 10Mbit/s.

Dois desses subsistemas processadores possuíam interfaces Ethernet extra para redes locais e acesso remoto. Dois outros nós possuem controladoras de vídeo de alta resolução, sendo um deles dotado também de mouse e teclado. A interconexão entre os subsistemas originalmente usava duas redes Ethernet, sendo uma de par trançado e outra de cabo coaxial fino. Outros arranjos foram experimentados mais tarde.

### **- Beowulf Demonstration System (1996)**

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

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

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

Beowulf é um projeto bem sucedido. A opção feita por seus criadores de usar hardware popular e software aberto tornou-o fácil de replicar e alterar. Prova disso é a grande quantidade de sistemas construídos à moda Beowulf. Mais do que um experimento, foi obtido um sistema de uso prático que continua sendo melhorado.

## 4. PVM

A biblioteca de programação paralela PVM (Parallel Virtual Machine) foi produzida pelo Heterogeneous Network Project, um esforço conjunto da Oak Ridge National Laboratory, University of Tennessee, Emory University e Carnegie Mellon University, em 1989, para facilitar o campo da computação paralela heterogênea. PVM foi um dos primeiros sistemas de software a possibilitar que programadores utilizem uma rede de computadores heterogêneos ou sistemas MPP (Massively Parallel Processors) para desenvolver aplicações paralelas sob o conceito de passagem de mensagens. Esta biblioteca ou API de programação tem sido muito difundida em ambientes computacionais científicos e recentemente tem ganhado muitos adeptos também no campo de aplicações comerciais, sendo referenciada como o “padrão de facto” em sua área.

O pacote PVM é relativamente pequeno (cerca de 4.5 MB de código fonte em C), roda sobre ambiente Unix e seus derivados e necessita ser instalado apenas uma vez em cada máquina para ser acessível à todos os usuários. Além disso, a instalação não requer privilégios especiais e pode ser efetuada por qualquer usuário.

Esse capítulo pretende apresentar as características da biblioteca PVM (versão 3.X) e possíveis avaliações a respeito destas.

### 4.1. Componentes do PVM

O sistema PVM é composto de duas partes. A primeira parte é um daemon, chamado *pvmd*, que reside em todas as máquinas que compõem o cluster, criando o que se refere como uma *máquina paralela virtual*. Quando um usuário deseja rodar uma aplicação PVM, ele executa o daemon *pvmd* em um dos computadores do cluster, o qual responsabiliza-se por chamar outros processos de daemons *pvmd* escravos em cada computador da máquina paralela virtual. Uma aplicação PVM pode então ser iniciada em um prompt Unix em qualquer console do cluster.

A segunda parte do sistema é uma biblioteca de rotinas PVM. Esta biblioteca contém rotinas chamáveis pelo usuário para passagem de mensagens, criação de processos, sincronização de tarefas e modificação da máquina virtual. As aplicações devem ser linkadas com esta biblioteca para poderem usufruir do ambiente paralelo criado pelo PVM.

## 4.2. Configuração do Ambiente

A principal idéia por trás do PVM é utilizar um conjunto de computadores heterogêneos interconectados, como um recurso virtualmente único. Cada computador existente na rede pode ser utilizado como sendo um nó da máquina paralela virtual. O papel de console da máquina paralela é assumido pelo próprio nó local onde o usuário está localizado fisicamente. O usuário pode alocar qualquer nó da rede localmente, ou a longa distância, desde que o mesmo esteja devidamente autorizado.

Um daemon mestre cria os diversos daemons escravos nas máquinas do cluster via ssh ou rsh, os quais devem estar devidamente configurados em todos os hosts antes do uso do PVM. Como rsh e ssh são assuntos bastante amplos e fogem do escopo deste projeto, não serão apresentados aqui.

A partir do console, o usuário pode criar sua própria configuração de máquinas. Uma configuração é composta de um conjunto de nós e pode ser determinada de duas maneiras:

- Após a criação da máquina paralela virtual, antes de disparar uma aplicação paralela. Isto é feito ao nível da linha de comando do console, através do comando ADD, seguido do nome da máquina;
- Durante o tempo de execução da aplicação paralela, pela função `pvm_addhosts()`, como no exemplo abaixo:

```
int num_hosts = 4;
char *h_name[] = {"labcom02", "labcom03", "labcom04", "labcom05"};
int h_status[] = {0, 0, 0, 0};
.....
if ((info = pvm_addhosts(h_name, num_hosts, h_status)) < 0) {
    printf("\nErro: impossível acrescentar hosts\n");
}
```

```

    pvm_exit();
}
-----
-----

```

Os argumentos passados à rotina `pvm_addhosts()` são, na ordem: um vetor contendo os nomes dos nós a serem alocados, a quantidade de nós, e, por último, um vetor de inteiros onde a função irá retornar os códigos de status de cada nó alocado, indicando sucesso ou fracasso. Somente uma única máquina virtual pode ser criada para cada usuário em um dado instante, e cada uma delas não tem relação com a máquina de outro usuário. Isto evita a possibilidade de uma aplicação causar problemas nas aplicações de outros usuários.

### 4.3. Criação Dinâmica de Processos

Aplicações utilizando PVM são constituídas de várias subtarefas, as quais são executadas em paralelo nas diversas máquinas do cluster. Essa subtarefas são referenciadas com muitos termos diferentes, como `tasks`, `processos`, `filhos`, `folhas`, `tarefas`, e ainda outros. Essa subtarefas podem ser criadas e terminadas (abortadas) a qualquer momento sobre qualquer máquina ou em uma máquina ou arquitetura específica desse mesmo cluster. Sendo assim, os processos podem ser alocados de diferentes maneiras, sendo as três principais:

- **Transparente:** modo no qual uma subtarefa é alocada automaticamente na máquina mais apropriada naquele momento. A decisão é tomada pelo `daemon pvmd` sobre qual máquina do cluster deve ser gerada a nova subtarefa. A maneira ou os critérios que PVM toma para tal decisão não foram descobertos por este projeto, mas baseado em dados visuais, conclui-se que os critérios fundamentais são a carga da máquina e o número de subtarefas já rodando nesta máquina no instante da criação do novo processo, ou seja, PVM procura "balancear" a carga entre as máquinas do cluster.
- **Dependente de arquitetura:** modo no qual o programador especifica ou identifica a arquitetura na qual o novo processo (subtarefa) deve ser gerado e executado. Caso exista mais de uma máquina no cluster que corresponda à

arquitetura especificada, então os critérios de escolha são os mesmos relatados no item anterior.

- **Específico à máquina:** neste modo, o processo deve ser gerado na máquina específica informada. Caso esta máquina falhe ou não esteja presente no momento da criação, um erro é gerado e a subtarefa não é criada.

O seguinte trecho de código cria *ntask* subtarefas de um executável chamado *task* e deixa o PVM escolher as máquinas onde essas tarefas serão criadas e executadas:

```
info = pvm_spawn(task, (char**)0, PvmTaskDefault, (char*)0, ntask, child);
if (info != ntask) {
    pvm_perror("Gerando filhos");
    pvm_exit();
    return(-1);
}
```

A variável *info* é um inteiro e deve possuir o número de filhos gerados. Caso esse número seja diferente do desejado – *ntask* – o programa é abortado. O último argumento passado é um vetor que conterá as task IDs dos filhos gerados - algo semelhante à identificação de processos em sistemas Unix (PID) - e deve conter no mínimo *ntask* posições. O mesmo código é apresentado a seguir, porém agora identificando em qual máquina um processo filho deve ser gerado:

```
info = pvm_spawn(task, (char**)0, PvmTaskHost, labcom02, 1, &tid);
if (info != 1) {
    pvm_perror("Gerando filhos");
    pvm_exit();
    return(-1);
}
```

A criação dinâmica de processos é uma das grandes vantagens do PVM em relação à especificação MPI, como será visto mais adiante.

## 4.4. Comunicação

O modelo PVM de comunicação assume que qualquer tarefa pode enviar uma mensagem para qualquer outra tarefa, e que não há um limite no tamanho ou número de tais mensagens. Enquanto todos os hosts possuem limitações físicas de memória que restringem potenciais espaços de buffers, este modelo de comunicação não se restringe às limitações de uma máquina particular. Ou seja, PVM aloca espaços de buffer dinamicamente, mas o tamanho ou volume de mensagens que podem chegar à um host ao mesmo tempo é limitado pela quantidade de memória livre neste mesmo host.

O modelo PVM de comunicação provê 3 diferentes tipos de metodologias de envio e recebimento de mensagens: envio assíncrono com bloqueio, recebimento assíncrono com bloqueio e recebimento sem bloqueio. Um envio com bloqueio retorna logo que o buffer de envio estiver livre para reuso, ou seja, o programa pára sua execução a partir do início do envio de uma mensagem alocada no buffer de envio até o momento do término dessa operação, que é bastante rápida. Um envio assíncrono não depende da confirmação do receptor para poder enviar uma nova mensagem. Um recebimento sem bloqueio vasculha o buffer de recebimento, caso este esteja cheio, imediatamente retorna com os dados, caso contrário retorna com um flag que os dados não chegaram e continua sua execução. A função de recebimento com bloqueio retorna somente quando os dados chegarem ao buffer de recebimento de mensagens. A seguir, os protótipos de algumas funções de envio e recebimento de mensagens:

```
int bufid = pvm_rcv(int tid, int msgtag);   recebimento com bloqueio
int bufid = pvm_nrcv(int tid, int msgtag);  recebimento sem bloqueio
int info = pvm_send(int tid, int msgtag);   envio com bloqueio
```

PVM também garante que a ordem das mensagens é preservada entre duas tarefas. Se a task 1 envia uma mensagem A à task 2 e logo em seguida envia uma outra mensagem B à mesma task 2, a mensagem A chegará antes que a mensagem B à task 2. A fim de especificar qual mensagem o receptor deve esperar, o programador

pode especificar uma *tag* (etiqueta) que é enviada junto com a mensagem. Com isso, o processo receptor somente receberá mensagens que combinem com a *tag* desejada.

Para enviar uma mensagem, é necessário que se façam chamadas à três funções diferentes. Primeiro, o buffer de envio é inicializado com uma chamada à função `pvm_initsend()`. Segundo, a mensagem precisa ser “empacotada” dentro deste buffer usando qualquer combinação das funções `pvm_pk*`(`*`). Em terceiro lugar, a mensagem completa é enviada à outro processo com a função `pvm_send()`, ou outra função similar.

Existem muitas vantagens em se ter três passos separados para enviar uma mensagem. O método permite à um usuário “quebrar” uma mensagem em muitos pedaços e armazenar esses pedaços todos juntos, necessitando apenas uma operação de envio para comunicar-se com outro processo. Por exemplo, uma mensagem pode conter um vetor de números em ponto flutuante e um inteiro definindo o tamanho desse vetor. Ou uma simples mensagem pode conter uma estrutura inteira, incluindo vetores de inteiros, strings de caracteres e vetores de ponto flutuante. Isso é importante porque empacotar uma mensagem é relativamente rápido se comparado à transferência de dados sobre uma rede de comunicação, embora isso está começando a mudar com o advento das redes de alta velocidade. Combinando muitos pedaços de dados dentro da mesma mensagem, o usuário pode diminuir o número de envios em um algoritmo, eliminando o tempo desperdiçado com latência e overhead sobre a rede.

Outra vantagem deste método de comunicação baseado nestes três passos, consiste no fato de que o usuário necessita codificar e fragmentar os dados somente uma vez, ou seja, depois que uma mensagem é empacotada, ela pode ser enviada à muitos destinos, seja uma a uma, seja por broadcast, sem a necessidade de se empacotar esta mensagem novamente. O buffer de envio só é limpo quando é feita uma nova chamada à `pvm_initsend()`.

Embora existam todas estas vantagens neste modelo de comunicação, há casos que o usuário necessita enviar somente um dado simples à outra máquina do cluster. Para esses casos, a versão 3.3 ou superior do PVM implementa a função `pvm_psend()`, a qual combina inicialização do buffer, empacotamento da mensagem e envio desta no mesmo passo. Isso aumenta a performance, pois o programa não perde

tempo com chamadas à funções diversas. A função que complementa `pvm_psend()` é `pvm_precev()`, embora não seja necessário que sejam usadas em conjunto.

A seguir, trechos de código de um algoritmo implementado por este projeto que ilustram o uso do modelo de comunicação PVM baseado nos três passos descritos anteriormente:

```
// Envio ao Pai posição que ocupo na matriz resultante + elemento calculado
pvm_initsend(PvmDataRaw);
pvm_pkint(&who, 1, 1);
pvm_pkfloat(&cij, 1, 1);
pvm_send(pvm_parent(), FMSGTAG);

// Recebo filhos
flag = 0;
for (i=0; i<ntask; i++) {
    pvm_recev(-1, FMSGTAG);
    pvm_upkint(&who, 1, 1);
    pvm_upkfloat(&cij, 1, 1);
    c[who] = cij;
    if (c[who] != a[who])
        flag = 1;
}
```

O primeiro trecho de código é o processo filho que deseja enviar ao processo pai determinado elemento que foi calculado (`cij`). A primeira linha de código inicializa o buffer de envio com a opção `PvmDataRaw`, que diz ao PVM para não perder tempo codificando os dados com XDR<sup>3</sup>, já que o cluster implementado possui uma arquitetura homogênea. A segunda e terceira linha empacotam os dados que o processo deseja enviar, sendo um inteiro e um ponto flutuante. A última linha deste trecho de código descobre a task ID do pai por meio da chamada à `pvm_parent()`, e envia à este o conteúdo do buffer com uma tag `FMSGTAG`, que foi definida no início do programa como um número inteiro (2).

---

<sup>3</sup> O XDR (External Data Representation) é um padrão para codificação e decodificação de dados para o transporte entre diferentes arquiteturas (SUN, VAX, PC, CRAY, etc.), semelhante ao ASN1, do modelo OSI. Cria uma representação independente de máquina, sendo a conversão automática e transparente.

O segundo trecho de código é do processo pai que neste momento aguarda por mensagens provenientes de qualquer outro processo. A rotina é realizada *n* vezes, que é o número de processos filhos criados que enviarão mensagens. O recebimento é assíncrono com bloqueio e diz ao pai para aguardar até que uma mensagem com a tag FMSGTAG chegue de qualquer filho (-1). Assim que a mensagem chega, ela é desempacotada e armazenada na posição informada do vetor resultante (c[who]).

Um cluster consiste basicamente em processos trocando mensagens sobre uma rede de comunicação e PVM possui muitas funções de recebimento e envio de mensagens. Não é necessário que uma complemente outra, isto é, não é necessário que uma mensagem enviada com `pvm_psend()` seja recebida com `pvm_precv()`, por exemplo, ficando a cargo do programador a escolha da melhor solução sobre determinado problema.

#### 4.5. Grupos Dinâmicos de Processos

Uma biblioteca separada chamada *libgpvm3* precisa ser linkada com a aplicação do usuário para que os programas possam usar qualquer função de grupo provida pelo PVM. O daemon `pvmd` não executa tarefas de grupo. Estas são providas por um servidor de grupo, que é automaticamente iniciado quando a primeira função de grupo é invocada.

Mantendo a filosofia PVM, funções de grupo são projetadas para serem transparentes para o usuário, trazendo, com isso, algum custo relacionado à eficiência. Uma tarefa PVM pode unir-se ou deixar um grupo a qualquer momento, sem ter que informar qualquer outra tarefa deste mesmo grupo. Além do mais, tarefas pertencentes a um grupo podem comunicar-se com tarefas de outros grupos. No geral, qualquer tarefa PVM pode chamar qualquer uma das funções de grupo a qualquer momento, com exceção de `pvm_lvgroup()`, `pvm_barrier()` e `pvm_reduce()`, devido à natureza destas. Uma tarefa pode também pertencer a vários grupos.

A política dinâmica de grupos de processos em PVM é um dos aspectos mais relevantes no que diz respeito à sua comparação com a filosofia MPI. Segue adiante uma abordagem sucinta de algumas funções de grupo PVM:

```
int inum = pvm_joingroup(char *group);  
int info = pvm_lvgroup(char *group);
```

Estas rotinas permitem uma tarefa unir-se ou deixar um grupo existente. A primeira chamada à `pvm_joingroup()` cria um grupo de nome *group* e coloca a tarefa neste grupo, retornando o número de instância do processo neste grupo (*inum*). Números de instâncias vão de 0 ao número de membros menos 1. Se um processo deixa e entra novamente em um grupo, pode ganhar um número de instância diferente do qual possuía no momento de sua saída. É responsabilidade do usuário manter a seqüência sobre os números de instância se o algoritmo precisar disto. Se muitas tarefas deixam um grupo e não retornam ou novas tarefas não entram neste grupo, haverá então aberturas na seqüência de números instanciados.

```
int tid = pvm_gettid(char *group, int inum);  
int inum = pvm_getinst(char *group, int tid);  
int size = pvm_gsize(char *group);
```

A rotina `pvm_gettid()` retorna a task ID de um processo, dado seu grupo e seu número de instância. Isto é importante, pois permite que duas tarefas que não conhecem o task ID uma da outra e precisam comunicar-se entre si, entrem em um grupo e troquem esta informação, simplesmente com uma chamada a esta função. A rotina `pvm_getinst()` retorna o número de instância de um processo dado seu grupo e seu task ID. A rotina `pvm_gsize()` retorna o número de membros de um grupo especificado por *group*.

```
int info = pvm_barrier(char *group, int count);
```

A função `pvm_barrier()` bloqueia todos os processos até que *count* membros do grupo chamem `pvm_barrier()`. No geral, *count* é o número total de tarefas naquele grupo. Um *count* é requerido porque grupos dinâmicos de processos não conhecem quantos membros possuem em um dado instante. Um erro ou travamento da

aplicação ocorrerá se uma tarefa chamar `pvm_barrier()` com um *count* de 4 e outra tarefa do mesmo grupo chamar esta função com um *count* de 5, por exemplo.

```
int info = pvm_bcast(char *group, int msgtag);
```

Esta função nomeia a mensagem com um inteiro *msgtag* e envia o conteúdo do buffer de envio à todas as tarefas de um grupo via broadcast, menos para si mesma. Se uma tarefa entrar no grupo durante a operação de broadcast, não receberá a mensagem.

## 4.6. Performance

PVM usa sockets UDP e TCP para transferir dados sobre uma rede de comunicação. UDP é um protocolo sem conexão onde a entrega do pacote não é garantida. O protocolo TCP requer uma conexão entre processos e implementa sofisticados algoritmos de retransmissão para garantir que os dados sejam entregues.

Em operações normais é usado o protocolo UDP. Quando dados necessitam ser enviados, a tarefa emissora envia a mensagem desejada ao daemon PVM local. O daemon local transfere a mensagem ao daemon PVM remoto usando UDP, e este finalmente transfere a mensagem à tarefa remota quando esta fizer uma chamada à `pvm_recv()`.

Há um método de transferência menos escalável, mas em compensação mais rápido, nas versões do PVM acima da 3.x. A função PVM `pvm_setopt(PvmRoute,PvmRouteDirect)`, montará um link TCP direto entre todas as tarefas existentes no cluster naquele momento. Esse processo consome bastante tempo, mas todas as subseqüentes transferências de mensagens entre estas tarefas são entre 2 e 3 vezes mais rápidas que o processo normal. A desvantagem deste método é que cada link TCP consome um descritor de arquivo (fd). Assim, há a necessidade potencial de  $n^2$  descritores de arquivos, onde  $n$  é o número de tarefas na máquina paralela virtual. Desde que a chamada à `pvm_setopt()` precisa ser feita somente uma vez no topo de um programa PVM, é possível efetuar testes a fim de verificar se há melhora no desempenho.

Outra forma de aumentar a performance é fazer com que PVM não perca tempo codificando os dados. Como uma máquina virtual é geralmente composta de computadores com arquiteturas heterogêneas, PVM codifica os dados com XDR antes de enviá-los ao destino. Caso a arquitetura da máquina virtual seja homogênea, o usuário pode ignorar este procedimento simplesmente inicializando o buffer com a opção `PvmDataRaw`. A inicialização do buffer ficaria então: `pvm_initsend(PvmDataRaw)`.

Há muitas formas de se melhorar a performance com PVM, mas a verdade é que não existe uma “receita” ou método a ser seguido, pois tudo depende da arquitetura do cluster, da velocidade da rede, das configurações do sistema e de muitos outros fatores determinantes que devem ser avaliados no momento de se escolher uma metodologia para resolver determinado problema.

## 4.7. Aplicação Exemplo

Segue abaixo um exemplo muito simples e compacto de uma aplicação PVM. Este programa inicialmente não tem aplicabilidade, mas serve para ilustrar a comunicação entre processos e demais características apontadas até aqui. O programa “vetor\_pvm.c” quando executado, cria um processo filho que é outra instância dele mesmo, e ambos trocam dados entre si. Não é apresentado aqui funções de grupo, pois a aplicação cria somente duas tarefas na máquina virtual, visando a simplicidade do algoritmo para seu bom entendimento.

### VETOR\_PVM.C

```
#include <stdio.h>
#include <pvm/pvm3.h>
#define MSGTAG 11

int main(int argc, char **argv)
{
    int mytid, tam, i, pai, tid, info;
    float *vet1, *vet2;                // ponteiros para vetores

    pvm_setopt(PvmRoute, PvmRouteDirect);
    mytid = pvm_mytid();                // busca minha task ID
    pai = pvm_parent();                // Task ID do processo pai
```

```

if (pai == PvmNoParent) {
    tam = atoi(argv[1]); // Se não tem pai, eu sou o pai
    vet1 = (float*)malloc(sizeof(float)*tam); // tamanho do vetor
    vet2 = (float*)malloc(sizeof(float)*tam);
    for (i=0; i<tam; i++) { vet1[i]=i+i; vet2[i]=i*i; }

    // Cria processo filho
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0, 1,
&tid);

    // Envia os dois vetores e o tamanho ao processo filho
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&tam, 1, 1);
    pvm_pkfloat(vet1, tam, 1);
    pvm_pkfloat(vet2, tam, 1);
    pvm_send(tid, MSGTAG);

    // Aguarda resposta do filho
    pvm_recv(tid, MSGTAG);
    pvm_upkfloat(vet1, tam, 1);

    printf("\nVetor resultante:\n");
    for (i=0; i<tam; i++) printf("%.2f ", vet1[i]);
}
else { // Sou um filho
    // Recebo vetores do processo pai
    pvm_recv(pai, MSGTAG);
    pvm_upkint(&tam, 1, 1);
    vet1 = (float*)malloc(sizeof(float)*tam);
    vet2 = (float*)malloc(sizeof(float)*tam);
    pvm_upkfloat(vet1, tam, 1);
    pvm_upkfloat(vet2, tam, 1);

    for (i=0; i<tam; i++) vet1[i] *= vet2[i];
    // Envia vetor resultante ao pai
    pvm_initsend(PvmDataDefault);
    pvm_pkfloat(vet1, tam, 1);
    pvm_send(pai, MSGTAG);
}
pvm_exit();
return(0);
}

```

Esse algoritmo é implementado sob o conceito SPMD, pois, quando executado, cria duas instâncias de si mesmo. Quando o usuário executar esta aplicação, uma primeira instância deste código será criada, dando origem a um primeiro processo. Esse processo aceitará o teste condicional, pois saberá, através de sua task ID, que não foi criado a partir de outro processo, mas sim por uma chamada na linha de comando. Depois de alocar memória e inicializar dois vetores, este criará uma segunda instância de si mesmo, com a chamada a `pvm_spawn()`. Sendo assim, enquanto esse primeiro processo continua seu fluxo de execução, o outro processo paralelo inicia o seu. O primeiro processo empacota e envia ao segundo os dois vetores criados e aguarda uma

resposta, bloqueando sua execução. O segundo processo, que recebeu uma resposta negativa no teste condicional, aguarda e recebe uma mensagem do processo que o criou. Depois de recebida a mensagem com os dois vetores e seus tamanhos, efetua uma operação de multiplicação entre ambos e envia o resultado ao primeiro processo. Este então recebe o resultado e o imprime na tela.

## 5. MPI

MPI (Message Passing Interface) é uma especificação padrão para implementação de bibliotecas de passagem de mensagens sobre uma máquina MIMD de memória distribuída, e foi definida pelo MPI Fórum, o qual é basicamente constituído de vendedores de computadores paralelos, programadores e principalmente especialistas em aplicações paralelas. Sendo assim, muitas implementações de bibliotecas do padrão MPI tem sido desenvolvidas, sendo algumas proprietárias e outras de código livre, mas todas seguindo o mesmo padrão de funcionamento e uso.

O esforço de padronização MPI envolve cerca de 60 pessoas de 40 organizações, principalmente dos Estados Unidos e da Europa. A maioria dos vendedores de computadores paralelos e concorrentes estão envolvidos com o padrão MPI, através de pesquisas em Universidades e laboratórios governamentais e industriais. O processo de padronização começou com um seminário sobre o Centro de Pesquisas em Computação Paralela, realizado em 29 e 30 de Abril de 1992 em Williamsburg, Virginia. Nesse seminário, as características essenciais para uma padronização de uma interface de passagem de mensagem foram discutidas, sendo estabelecido também um grupo de trabalho para continuar o processo de padronização.

As funcionalidades que MPI designa são baseadas em práticas comuns de paralelismo e outras bibliotecas de passagem de mensagens similares, como Express, NX/2, Vertex, Parmacs e P4. A filosofia geral do padrão MPI é implementar e testar características novas que venham a surgir no mundo da computação paralela. Assim que uma determinada gama de características novas tenham sido testadas e aprovadas, o grupo MPI reúne-se novamente para considerar sua inclusão na especificação do padrão. Muitas das características do MPI tem sido investigadas e usadas por grupos de pesquisas por muitos anos, mas não em ambientes de produção ou comerciais. Sendo assim, existe um grande espaço de tempo entre novidades do padrão e sua implementação por vendedores e colaboradores. Entretanto, a incorporação destas características dentro do padrão MPI é justificada pelas expressivas inovações que elas trazem.

## 5.1. MPICH - Uma Implementação do Padrão MPI

Para a realização deste trabalho, foi usada uma implementação do padrão MPI-1.2<sup>4</sup>, chamada MPICH, do Argonne National Laboratory, a qual é disponibilizada livremente, sem restrições comerciais. Assim como o PVM, a biblioteca MPICH roda sobre ambiente Unix e seus derivados, mais precisamente, no contexto deste projeto, sobre Linux. Essa implementação foi iniciada em 1993 e seu último release é de 1999, sendo também considerada uma das únicas implementações existentes que combinam portabilidade, interoperabilidade e alta performance. Apesar desta implementação em particular seguir as especificações contidas no padrão MPI, ela possui muitas características adicionais, como suporte à interoperabilidade entre máquinas heterogêneas e suporte a máquinas SMP, mas essas características não serão levadas em conta neste trabalho, pois este pretende realizar uma avaliação entre o PVM o padrão MPI-1.2, desconsiderando particularidades de qualquer implementação MPI.

Apesar de possuir um tamanho relativamente grande se comparada com outras bibliotecas similares (cerca de 25 MB, em código C), a instalação e configuração dessa biblioteca é muito simples e não requer privilégios especiais de superusuário, podendo ser instalado por qualquer usuário comum que possua direito de login. Depois de configurada e compilada, deve-se realizar certas entradas simples em alguns poucos arquivos de configuração - como informar as máquinas que compõem o cluster - para que se possa utilizar o ambiente de processamento paralelo.

---

<sup>4</sup> MPI-1.2 é uma extensão para o padrão MPI-1 de 1992. Há uma terceira especificação chamada MPI-2, a qual adiciona várias extensões à versão 1.2, sendo uma das mais importantes o controle de processos dinâmicos. Devido à dificuldade de se encontrar uma implementação MPI-2 que seja distribuída livremente, este trabalho restringe-se ao padrão MPI-1.2. Portanto, qualquer menção ao padrão MPI, refere-se, na verdade, ao padrão MPI-1.2.

## 5.2. Uma Breve Análise sobre MPI

MPI pretende ser um padrão para interface de passagem de mensagens sobre uma máquina MIMD de memória distribuída. MPI não contém qualquer suporte a tolerância à falhas, assumindo comunicação segura. Adicionalmente, não é um ambiente paralelo completo, pois não consegue simular a biblioteca PVM na constituição de uma máquina paralela virtual na execução de uma aplicação. Ou seja, sua especificação não aborda ambientes heterogêneos. Isso porque geralmente cada vendedor de um grande sistema paralelo, possui sua própria implementação MPI, otimizada para sua arquitetura. Sendo assim, o padrão MPI não exige e não pode garantir que um vendedor crie sua implementação compatível com a de outro, em detrimento da performance sob seu sistema.

Em contrapartida, MPI oferece:

- Um conjunto completo de rotinas que suportam comunicação ponto-a-ponto entre pares de processos. Versões com bloqueio e sem bloqueio dessas rotinas são providas, as quais podem ser usadas de quatro diferentes modos, que serão abordados adiante;
- a abstração de um "comunicador" (grupo de comunicação), que permite implementações seguras de bibliotecas de software modulares;
- tipos de dados gerais e derivados;
- um rico conjunto de rotinas de comunicação coletiva, que executam comunicação coordenada através de um conjunto de processos.

Essas características são essenciais para um ambiente complexo de passagem de mensagens entre máquinas paralelas, sendo algumas dessas abordagens introduzidas pelo próprio padrão MPI, como comunicação ponto-a-ponto, por exemplo.

MPI não fornece uma maneira para criação dinâmica de processos, ou seja, há um número fixo de processos do início ao fim de uma aplicação paralela, sendo este número informado pelo usuário, via linha de comando, no momento da execução da aplicação MPI. Todos os processos são membros de ao menos um grupo de processos. Inicialmente todos os processos são membros do mesmo grupo, e um número de rotinas

são providas para prover que uma aplicação possa criar e destruir novos subgrupos. Com um grupo, cada processo possui um "rank" - comparado a task ID de uma tarefa PVM, ou a um PID de um processo Unix - que é usado para identificar um processo dentro de um grupo, variando de 0 à n-1, onde n é o número de processos deste grupo.

O trecho de código mostrado a seguir servirá para ilustrar o início de uma implementação típica de um programa MPI:

```
#include <stdio.h>
#include <time.h>
#include "mpi.h"

int main(int argc , char *argv[])
{
    int num_process, rank, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_process);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &namelen);

    .....
    .....

    MPI_Finalize()
    return(0);
}
```

Aqui, o ambiente MPI é iniciado pelo processo com a chamada à `MPI_Init()`, e a função `MPI_Comm_size()` recupera o número de processos criados pelo usuário, assim, cada processo sabe quantos outros processos existem para esta aplicação. A variável *rank* possuirá o id do processo dentro do grupo `MPI_COMM_WORLD`, e a variável *namelen* possuirá o nome da máquina na qual esse processo está sendo executado.

MPI foi projetado para ser uma interface de passagem de mensagens, em vez de um ambiente de programação paralela completo. Sendo assim, intencionalmente omite algumas características que seriam desejáveis:

- Mecanismos para criação e controle dinâmico de processos;
- operações de comunicação coletiva que possam envolver mais que um grupo de processos;

- tolerância à falhas;
- suporte a heterogeneidade.

### 5.3. Comunicação Ponto-a-Ponto

Uma das características mais importantes do padrão MPI é a gama de funções para comunicação ponto-a-ponto entre pares de processos. MPI provê um conjunto de funções de envio e recebimento de dados tipados, com uma tag (etiqueta) associada, semelhante à biblioteca PVM.

MPI provê rotinas com bloqueio e sem bloqueio para enviar e receber mensagens. Um envio com bloqueio não retorna enquanto o buffer de envio não puder ser alterado com segurança, sem corromper a mensagem a ser enviada. Um envio sem bloqueio retorna sem finalizar o envio da mensagem, e o programador deve ter o cuidado para não alterar o conteúdo do buffer até que esteja garantido que isto não corromperá a mensagem que está sendo enviada. A função de recebimento com bloqueio suspende a execução do processo, até que uma mensagem chegue e seja colocada no buffer de recebimento de mensagens. Um recebimento sem bloqueio recupera o conteúdo do buffer de recebimento de mensagens e retorna, podendo retornar sem que uma mensagem tenha chego e sido colocada no buffer, ficando a cargo do programador tomar os devidos cuidados para que não tente utilizar dados de uma mensagem que não tenha chego. Essas funções são mostradas na tabela 5.1. Aqui há a vantagem em relação ao PVM, de existir uma função p/ envio sem bloqueio, o que não existe na outra biblioteca.

No fragmento de código mostrado abaixo, dois processos são criados a partir do mesmo programa. O processo 0 envia ao processo 1 uma string de caracteres, usando `MPI_Send()`. Os três primeiros argumentos da função são referentes aos dados que estão sendo enviados: a mensagem propriamente dita, o tamanho da mensagem e o tipo de dado que esta mensagem contém. O quarto parâmetro especifica o processo destino (1) e o quinto parâmetro especifica a tag da mensagem. Finalmente, o último parâmetro é um comunicador, o qual especifica um domínio de comunicação, definindo quais processos podem comunicar-se entre si. Comunicadores serão abordados mais

adiante. `MPI_COMM_WORLD` é um comunicador padrão que define um domínio de comunicação para todos os processos que participam da computação:

```
char msg[20];
int myrank, tag = 99;
MPI_Status status;

.....

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    strcpy(msg, "Hello There");
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
}
```

**Tabela 5.1.** Classificação e nomes de funções para envio e recebimento de mensagens.

<b>ENVIO</b>	<b>Com Bloqueio</b>	<b>Sem Bloqueio</b>
<b>Standard</b>	<code>MPI_Send()</code>	<code>MPI_Isend()</code>
<b>Ready</b>	<code>MPI_Rsend()</code>	<code>MPI_Irsend()</code>
<b>Síncrono</b>	<code>MPI_Ssend()</code>	<code>MPI_Issend()</code>
<b>Buferizado</b>	<code>MPI_Bsend()</code>	<code>MPI_Ibsend()</code>
<b>RECEBIMENTO</b>	<b>Com Bloqueio</b>	<b>Sem Bloqueio</b>
<b>Standard</b>	<code>MPI_Recv()</code>	<code>MPI_Irecv()</code>

### 5.3.1. Modos de Comunicação

Em MPI, uma mensagem pode ser enviada de quatro modos diferentes:

- **Standard:** O fragmento de código mostrado acima usa o modo standard de comunicação. Nesse modo, uma operação de envio pode ser executada antes que uma operação de recebimento complementar seja iniciada. Sendo assim, MPI decidirá se uma mensagem necessita ou não ser armazenada em um buffer de envio.

- **Buf erizado:** Sua execução não depende de uma operação de recebimento complementar, podendo ser completada sem tal requisito. Nesse modo, uma mensagem é sempre armazenada em um buffer de envio alocado pela aplicação, antes de ser enviada ao destinatário. Isso pode melhorar como prejudicar a performance, já que depende da disposição do processo receptor no momento do envio. Os buffers devem ser alocados, desalocados e mantidos pelo programador.
- **Síncrono:** Esse modo de comunicação não necessita que um recebimento complementar tenha sido iniciado por outro processo. Entretanto, o envio não completará e aguardará até o momento que uma operação de recebimento tenha sido chamada pelo processo receptor. Assim, uma comunicação síncrona não somente indica que o buffer de envio pode ser reusado, como indica que o receptor alcançou tal ponto na execução.
- **Ready:** Modo que somente é iniciado se uma operação de recebimento pelo processo receptor já estiver aguardando. Caso contrário, a operação retorna um erro e o resultado é indefinido.

Apesar de todos esses modos de envio de mensagem, há somente um modo de recebimento, que pode ser usado como o complemento de todos estes. Sendo assim, MPI provê 8 funções para envio e 2 para recebimento de mensagens, como mostrado na tabela 5.1.

## 5.4. Comunicadores

Basicamente, um comunicador é uma coleção de processos que podem trocar mensagens entre si, seja entre um processo e outro, seja através de operações coletivas. Um comunicador permite que processos façam parte de:

- um grupo de comunicação;
- um contexto de comunicação.

Um grupo de comunicação é uma coleção ordenada de processos. Se um grupo consiste em  $p$  processos, cada processo no grupo é assinalado com um único *rank*

(ID do processo), o qual é um inteiro não negativo variando entre 0 e  $p-1$ . Grupos de processos permitem um maior controle organizacional das tarefas que compõem a aplicação paralela, pois permitem especificar, por exemplo, sincronização e operações coletivas sobre determinados processos.

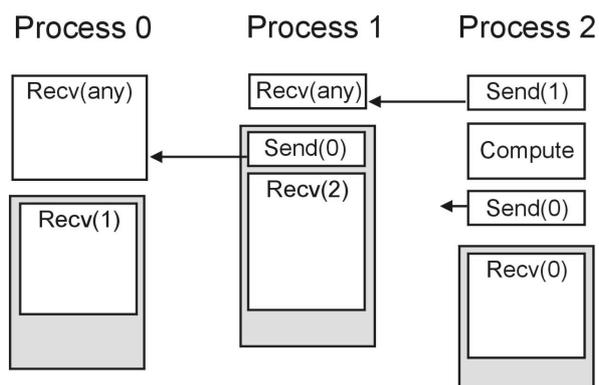
Um contexto de comunicação é um conceito muito importante em um ambiente paralelo, e pode ser pensado como uma tag (etiqueta) auxiliar, definida pelo sistema, que é anexada ao grupo. Assim, se dois processos possuem o mesmo contexto, eles podem comunicar-se entre si.

A questão é que, com o crescente aumento do uso de sistemas paralelos, é comum que uma aplicação de usuário use um pacote de software de terceiros, como uma biblioteca, agregada à sua aplicação. Dessa forma, torna-se aparente que somente uma tag de mensagem e o ID do emissor, não são suficientes para garantir que uma mensagem seja enviada e recebida corretamente entre tasks. Considere, por exemplo, o caso onde uma tarefa A em uma aplicação paralela chama uma rotina de uma biblioteca matemática, e essa rotina também usa o mesmo sistema de passagem de mensagens. Caso essa rotina matemática tenha sido escrita por terceiros, o desenvolvedor da aplicação não conhece quais tags de mensagens (ou máscaras) estão sendo usadas dentro dessa biblioteca. Sendo assim, não há meios de garantir que uma mensagem enviada pela tarefa A para uma tarefa B, não será inadvertidamente interceptada pela rotina matemática, causando falha tanto nesta biblioteca quanto na aplicação do usuário.

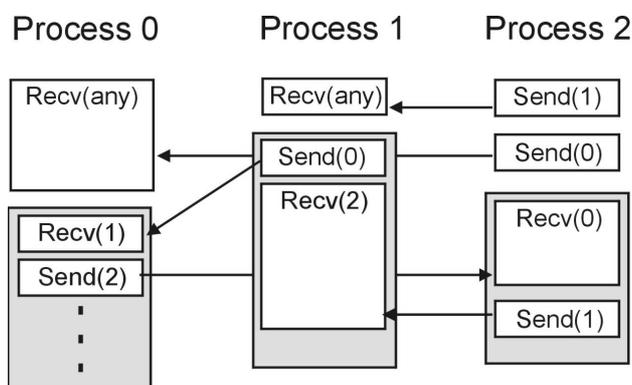
Assim, comunicadores são mecanismos úteis para evitar o não-determinismo em sistemas com passagem de mensagens. Um argumento comunicador é passado à todas as funções de passagem de mensagem em MPI, garantindo que uma mensagem comunique-se com outra, somente se o argumento comunicador passado às rotinas de envio e recebimento combinarem. Portanto, comunicadores provêm um critério adicional de seleção de mensagens, e conseqüentemente permitem a construção de rotinas que usem um espaço de tags sem ambigüidade.

Na figura 5.1, uma chamada à uma rotina de biblioteca conduz à um comportamento não intencional da aplicação, onde as áreas em cinza representam a chamada à rotina. No exemplo, a chamada à função de recebimento pelo processo 0, é satisfeita pela mensagem enviada pelo processo 1. Assim, a mensagem originária do processo 2 para o processo 0 nunca é recebida por este, causando um deadlock na

aplicação. Na figura 5.2, pode-se supor que comunicadores foram usados para evitar tal comportamento, pois, como pode-se notar, a segunda mensagem enviada pelo processo 2 é recebida pelo processo 0, e que a mensagem enviada pelo processo 0 é recebida pelo processo 2, ou seja, o comportamento da aplicação é exatamente o esperado pelo programador.



**Figura 5.1.** Comportamento não esperado do programa, causando deadlock no sistema



**Figura 5.2.** Uso de comunicadores, evitando interceptação errônea de mensagens.

Inicialmente, todos os processos criados por uma aplicação MPI, fazem parte do mesmo comunicador global `MPI_COMM_WORLD`. Caso um novo comunicador se faça necessário, ele será composto de um subconjunto de processos do comunicador original.

O trecho de código mostrado a seguir cria um comunicador que consiste em um subconjunto dos primeiros  $q$  processos do comunicador original `MPI_COMM_WORLD`, supondo que este consiste em  $p$  processos, onde  $q^2 = p$ . Sendo assim, esse novo comunicador consistirá em processos com ranks 0, 1, ...,  $q-1$ .

```
MPI_Group MPI_GROUP_WORLD;
MPI_Group primeiro_sub_grupo;
MPI_Comm primeiro_sub_comm;
int *process_ranks, proc;

// Cria uma lista de processos do novo comunicador
process_ranks = (int*)malloc(sizeof(int)*q);
for (proc=0; proc<q; proc++)
    process_ranks[proc] = proc;

// Recupera o grupo abaixo de MPI_COMM_WORLD
MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

// Cria o novo grupo
MPI_Group_incl(MPI_GROUP_WORLD, q, process_ranks, &primeiro_sub_grupo);

// Cria o novo comunicador
MPI_Comm_create(MPI_COMM_WORLD, primeiro_sub_grupo, &primeiro_sub_comm);
```

Primeiro é criado uma lista de processos, os quais constituirão o novo comunicador. Então é criado um grupo consistindo nestes processos. Isto requer dois comandos: primeiro recuperar o grupo associado com o comunicador `MPI_COMM_WORLD`, e então criar o grupo com a chamada à `MPI_Group_incl()`. Finalmente o comunicador atual é criado com `MPI_Comm_create()`. Agora, os processos em `primeiro_sub_comm` podem executar operações coletivas entre eles. Por exemplo, o processo 0 pode efetuar uma operação de broadcast a todos os outros processos de `primeiro_sub_comm`.

## 5.5. Tipos Derivados com `MPI_Type_struct()`

A comunicação entre máquinas de um cluster, a fim de que os processos troquem mensagens entre si, tem sempre um alto preço no que diz respeito à performance. Uma regra geral é que, quanto menor o número de mensagens, menor o overhead da rede, e conseqüentemente, melhor a performance da aplicação. No capítulo

referente ao PVM, foi visto que esta biblioteca provê a metodologia de “empacotar” muitos dados juntos em cada envio, diminuindo o número de envios e recebimentos necessários. A biblioteca MPI vai um pouco além e fornece uma alternativa para essa problemática: os tipos de dados definidos pelo usuário.

A fim de ilustração, considere que um processo necessita enviar três dados diferentes à todos os outros processos de seu grupo. Dois destes dados é um número em ponto flutuante e o outro é um inteiro (respectivamente: a, b, n).

Para resolver este problema, o programador poderia pensar em criar uma estrutura em C, com estes três elementos, e enviar esta estrutura à todos os processos com a função de operação coletiva `MPI_Bcast()`, que realiza um broadcast para todos os processos. A dificuldade aqui, é que `MPI_Bcast()`, assim como todas as funções de comunicação em MPI, requer um argumento que especifique o tipo de dado que está sendo enviado, e todos os tipos em MPI precisam ser declarados como `MPI_Datatype`, inclusive os tipos predefinidos, como por exemplo, `MPI_CHAR`, `MPI_FLOAT`, etc, são do tipo `MPI_Datatype`. Por exemplo, o código a seguir:

```
typedef struct {
    float a;
    float b;
    int n;
} INDATA_TYPE;

INDATA_TYPE indata;

MPI_Bcast(&indata, 1, INDATA_TYPE, 0, MPI_COMM_WORLD);
```

certamente ocasionará um erro, pois o tipo de dado `INDATA_TYPE`, não é um tipo `MPI_Datatype`, ou seja, nenhuma função em MPI conhece o tipo `INDATA_TYPE`.

Para solucionar então este problema, MPI permite que o programador construa novos tipos de dados MPI em tempo de execução. Tais tipos são chamados tipos de dados derivados. O fragmento de código abaixo mostra esta solução:

```
void Build_derived_type(INDATA_TYPE *indata, MPI_Datatype *message_type_ptr)
{
    int block_lengths[3];
    MPI_Aint displacements[3];
    MPI_Aint addresses[4];
```

```

MPI_Datatype typelist[3];

// Primeiro especifica os tipos
typelist[0] = MPI_FLOAT;
typelist[1] = MPI_FLOAT;
typelist[2] = MPI_INT;

// Especifica o número de elementos em cada tipo
block_lengths[0] = block_lengths[1] = block_lengths[2] = 1;

// Calcula os deslocamentos dos membros relativos a indata
MPI_Address(indata, &addresses[0]);
MPI_Address(&(indata->a), &addresses[1]);
MPI_Address(&(indata->b), &addresses[2]);
MPI_Address(&(indata->n), &addresses[3]);
displacements[0] = addresses[1] - addresses[0];
displacements[1] = addresses[2] - addresses[0];
displacements[2] = addresses[3] - addresses[0];

// Cria o tipo de dados derivado
MPI_Type_struct(3, block_lengths, displacements, typelist, message_type_ptr);

// Comita o novo tipo de dado, para que possa ser usado
MPI_Type_commit(message_type_ptr);
}

```

As primeiras três declarações especificam os tipos dos membros do tipo derivado, e a próxima especifica o número de elementos em cada tipo. As próximas quatro declarações calculam o endereço dos três membros relativos ao endereço do primeiro – ao qual é dado deslocamento 0. Com essas informações, são conhecidos os tipos, tamanhos e locações relativas dos membros de uma variável do tipo estrutura `INDATA_TYPE` em C. As duas últimas funções criam o novo tipo definido pelo usuário. Nota-se também que o tipo dos vetores *displacements* e *addresses* são `MPI_Aint`, e não `MPI_INT`. Este tipo é usado porque possibilita armazenar valores maiores do que um inteiro permitiria.

O novo tipo pode agora ser usado com qualquer função de comunicação. Para resolver o problema exposto acima, o programador agora só precisa se preocupar com as funções de comunicação. O código ficaria então:

```

void Get_data(INDATA_TYPE *indata, int my_rank)
{
    MPI_Datatype message_type;
    int root = 0;
    int count = 1;
    if (my_rank == 0) {
        printf("Enter a, b, n \n");
        scanf("%f %f %d", &(indata->a), &(indata->b), &(indata->n));
    }
}

```

```

}
// Chama a função criada pelo programador
Build_derived_type(indata, &message_type);
// Efetua broadcast
MPI_Bcast(indata, count, message_type, root, MPI_COMM_WORLD);
}

```

MPI também prove o conceito de “empacotar” e “desempacotar” um conjunto de dados, assim como a biblioteca PVM, a fim de diminuir o número de operações de envio e, conseqüentemente, diminuir o overhead na rede. Embora algumas vezes seja preferível usar funções de empacotamento, a filosofia MPI encoraja o uso de tipos de dados definidos pelo usuário, apesar destes serem mais complexos de se trabalhar. Isso porque o tempo gasto para criar um novo tipo de dado é somente necessário uma vez em uma aplicação, sendo usado depois por todos os processos desta, sem necessidade de novas declarações. Por outro lado, se for feito uso das funções de empacotamento, toda vez que se desejar enviar um conjunto de dados à outro processo, estas funções deverão ser chamadas, consumindo tempo de processamento. Assim, tipos de dados definidos pelo usuário provêm uma maneira eficaz de agrupar informações em uma única operação de envio, diminuindo em muito o tempo de processamento em clusters que primam pelo desempenho.

## 5.6. Aplicação exemplo

O fragmento de código a seguir é a mesma aplicação exemplo apresentada no capítulo sobre PVM, mas agora portada para MPI. Assim como o exemplo para PVM, este algoritmo é muito compacto e simples, mas serve para se ter uma noção das principais diferenças entre essas duas bibliotecas em nível de código. A simplicidade do algoritmo é proposital para se conseguir um alto nível de abstração nesta etapa do trabalho. O programa “vetor\_mpi.c” necessita que o usuário crie duas instâncias (processos) deste programa, os quais trocarão dados entre si. A necessidade do usuário em criar exatamente dois processos, é proveniente do fato de MPI não permitir que se criem processos dinamicamente. Assim, o usuário necessita possuir um conhecimento prévio da aplicação, para seu bom funcionamento. A linha de comando para tal finalidade, na implementação MPICH do padrão MPI, é:

```
./mpirun -np 2 ./vetor_mpi.c tam_vetor
```

onde o comando *mpirun* é responsável por iniciar o ambiente paralelo e criar o número de processos definidos pelo argumento *-np*, e *tam\_vetor* é o tamanho do vetor exigido pela aplicação em particular.

## VETOR\_MPI.C

```
#include <stdio.h>
#include "mpi.h"
#define MSGTAG 11
int main(int argc, char *argv[])
{
    int tam, i;
    float *vet1, *vet2;
    char machine_name[MPI_MAX_PROCESSOR_NAME];
    int nprocess, rank, namelen;
    MPI_Status status;

    MPI_Init(&argc, &argv); // Argumentos do programa
    MPI_Comm_size(MPI_COMM_WORLD, &nprocess); // Nº de processos criados
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // id do processo

    // Recupera o nome da máquina em que esse processo reside
    MPI_Get_processor_name(machine_name, &namelen);

    // Quantidade de processos gerados está correta?
    if (nprocess != 2) {
        printf("\n%s só aceita 2 processos\n", argv[0]);
        MPI_Finalize();
        return(-1);
    }

    if (rank == 0) { // Processo 0?
        tam = atoi(argv[1]); // Tamanho do vetor
        vet1 = (float*)malloc(sizeof(float)*tam);
        vet2 = (float*)malloc(sizeof(float)*tam);
        for (i=0; i<tam; i++) { vet1[i] = i+i; vet2[i] = i*i; }

        // Envia os dois vetores ao processo 1
        MPI_Send(vet1, tam, MPI_FLOAT, 1, MSGTAG, MPI_COMM_WORLD);
        MPI_Send(vet2, tam, MPI_FLOAT, 1, MSGTAG, MPI_COMM_WORLD);

        // Aguarda vetor resultante do processo 1
        MPI_Recv(vet1, tam, MPI_FLOAT, 1, MSGTAG, MPI_COMM_WORLD, &status);
        printf("\nVetor resultante:\n");
        for (i=0; i<tam; i++) printf("%.2f ", vet1[i]);
    }
    else { // Processo 1
        tam = atoi(argv[1]);
        vet1 = (float*)malloc(sizeof(float)*tam);
        vet2 = (float*)malloc(sizeof(float)*tam);
        //Recebe vetores do processo 0
        MPI_Recv(vet1, tam, MPI_FLOAT, 0, MSGTAG, MPI_COMM_WORLD, &status);
        MPI_Recv(vet2, tam, MPI_FLOAT, 0, MSGTAG, MPI_COMM_WORLD, &status);
    }
}
```

```
        for (i=0; i<tam; i++) vet1[i] *= vet2[i];
        // Envia vetor resultante ao processo 0
        MPI_Send(vet1, tam, MPI_FLOAT, 0, MSGTAG, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return(0);
}
```

Neste exemplo, Dois processos devem ser criados simultaneamente, via linha de comando. A linha *MPI\_Init(&argc, &argv)* passará quaisquer argumentos de programa para a biblioteca MPI. A linha de código posterior, colocará na variável *nprocess*, o nº de processos criados pelo usuário. A linha a seguir, dá a cada processo, seu ID de identificação, o qual é armazenado na variável *rank*. Pode-se observar que neste exemplo, há um teste condicional a mais, se comparado com o mesmo exemplo no capítulo a respeito do PVM. Esse teste condicional verifica se o nº de processos gerados pelo usuário, condiz com as necessidades do programa. O segundo teste condicional, separa o caminho seguido pelas duas instâncias criadas. O processo que possuir seu rank igual a 0, criará e instanciará dois vetores, os quais são enviados ao processo com rank 1, e bloqueará sua execução, aguardando uma resposta desse outro processo. O processo 1 recebe os vetores e efetua um cálculo de multiplicação sobre ambos. Após, envia uma mensagem com o vetor resultante ao processo 0, o qual recebe a mensagem e imprime o resultado.

## 6. PRINCIPAIS DIFERENÇAS ENTRE PVM E MPI

Neste ponto do trabalho, é feita uma comparação entre as diferenças fundamentais das bibliotecas PVM e MPI de processamento paralelo, apontando situações onde uma pode ser favorável à outra. Apesar de possuírem objetivos comuns, as duas bibliotecas diferem substancialmente em alguns aspectos, pois enquanto uma prima pela eficiência e facilidade de uso, outra detém seu foco à performance e desempenho.

### 6.1. Portabilidade & Interoperabilidade

MPI possui o conceito de *portabilidade*, que permite que um programa escrito para uma dada arquitetura possa ser copiado, compilado e executado sobre uma segunda arquitetura, simplesmente recompilando a biblioteca MPI para essa segunda arquitetura, sem alteração dos fontes da aplicação.

PVM também suporta este nível de portabilidade, mas expande a definição de portátil para incluir *interoperável*. Aplicações PVM podem, similarmente, serem copiadas para diferentes arquiteturas, compiladas e executadas sem modificação dos fontes. Entretanto, o executável PVM pode também comunicar-se sobre sistemas heterogêneos. Em outras palavras, uma aplicação MPI pode rodar, como um todo, em uma arquitetura homogênea, e ser portátil para várias outras arquiteturas. Mas uma aplicação PVM pode ser portada para outras arquiteturas e permitir que estas rodem cooperativamente entre qualquer conjunto de diferentes arquiteturas, ao mesmo tempo.

O padrão MPI não proíbe tal heterogeneidade, mas também não a obriga. Isso porque geralmente um distribuidor de uma multicomputador cria sua própria implementação MPI, para que rode especificamente sobre sua arquitetura. Além disso, dificilmente um distribuidor de um sistema multicomputador, seja em forma de cluster ou MPP, irá prover algum tipo de interoperabilidade entre sua implementação e de outro distribuidor, em detrimento da performance sobre seu sistema.

A solução PVM para este problema, é sacrificar um pouco de performance, em favor da flexibilidade para comunicar através dos limites arquiteturais. Quando a comunicação entre processos é feita sobre a mesma arquitetura, PVM simplesmente deixa os dados em sua forma nativa, assim como MPI. Quando a comunicação se dá entre hosts de arquiteturas diferentes, PVM usa funções de comunicação padrão da rede para codificar e decodificar os dados. O fato de que a biblioteca PVM necessita determinar o destino de cada mensagem, provoca um pequeno overhead na comunicação.

PVM e MPI também diferem na interoperabilidade entre linguagens de programação. Com PVM, um programa escrito em C pode enviar uma mensagem que é recebida por um programa escrito em Fortran, e vice-versa. Em contraste, MPI não permite que um programa C converse com outro em Fortran. Essa restrição ocorre porque C e Fortran suportam interfaces de linguagem fundamentalmente diferentes, causando dificuldades em definir uma interface padrão entre ambas. A decisão MPI foi não forçar as duas linguagens a interoperarem.

## **6.2. Máquina Paralela Virtual**

PVM foi construído em volta do conceito de uma *máquina paralela virtual*, a qual é uma coleção dinâmica de recursos computacionais, gerenciados como um simples computador paralelo. Em contraste, MPI dirigiu seu foco às funções de passagem de mensagens, melhorando-as e incrementando o leque de opções disponíveis, deixando o conceito de máquina paralela virtual fora de seu escopo.

### **6.2.1. Controle de Processos**

Controle de processos refere-se a habilidade de iniciar e parar processos dinamicamente, no meio da execução de uma aplicação paralela. É possível também encontrar quais tarefas estão rodando e aonde estão rodando. Um ou mais processos podem ser criados a qualquer instante por uma aplicação e cooperarem com os já

existentes. PVM contém todas essas capacidades, provendo um rico conjunto de funções para controle de recursos e processos. Em contraste, MPI não possui qualquer função para criação de processos em tempo de execução<sup>5</sup>, devendo todos os processos serem gerados no momento da inicialização da aplicação. Além disso, o padrão MPI não define um método para iniciar uma aplicação paralela, ficando essa decisão a cargo de cada implementação.

### 6.2.2. Controle de Recursos

Em termos de controle de recursos, PVM é inteiramente dinâmico. Hosts podem ser adicionados ou deletados a vontade, a partir de um console do cluster, ou até mesmo a partir da aplicação do usuário. Com isso, PVM consegue permitir aplicações a interagir e controlar seus ambientes computacionais provendo um poderoso paradigma para balanceamento de carga, migração de tarefas e tolerância à falhas.

Outro aspecto do dinamismo de uma máquina paralela virtual é referente à eficiência. Por exemplo, considere uma aplicação típica de usuário, a qual inicia e finaliza com uma computação basicamente serial, mas que contém algumas fases de computação paralela pesada. Um grande sistema MPP não precisa ser desperdiçado como parte da máquina virtual, no momento da execução das porções seriais da aplicação, e pode ser adicionado somente no momento em que é realmente necessário, sendo liberado posteriormente. Falta ao MPI tal dinamismo, que é, de fato, especificamente designado para ser estático, a fim de melhorar a performance. Há, claramente, uma preocupação do MPI Forum em ganhar sempre uma margem extra de performance, em detrimento da flexibilidade, eficiência e facilidade de uso, pois tais características, presentes no PVM, aumentam significativamente o overhead de uma aplicação.

---

<sup>5</sup> A especificação MPI-2 incluirá meios para criação dinâmica de processos

### 6.3. Tolerância à Falhas

Há aplicações paralelas críticas que exigem um ambiente tolerante à falhas, pois uma parada repentina na execução pode pôr a perder meses de trabalho, ou ainda ter conseqüências mais sérias.

PVM suporta um esquema básico de notificação de falhas. Sob o controle do usuário, PVM pode “notificar” as tasks em execução quando o status da máquina paralela virtual muda ou quando uma task falha. A notificação vem em forma de uma mensagem especial de evento, a qual contém informações a respeito do evento em particular. Neste cenário, se uma tarefa morre, a tarefa receptora receberá uma mensagem de notificação, no lugar de qualquer mensagem esperada. Essa notificação permite à tarefa uma oportunidade de responder à falha, sem entrar em deadlock ou falhar.

Similarmente, se um host específico é crítico para a aplicação, os processos desta aplicação podem emitir pedidos de notificação para este host. Caso esse servidor, por algum motivo, parar de responder, as tarefas da aplicação paralela podem receber a notificação e se reconfigurarem para alocar os recursos restantes.

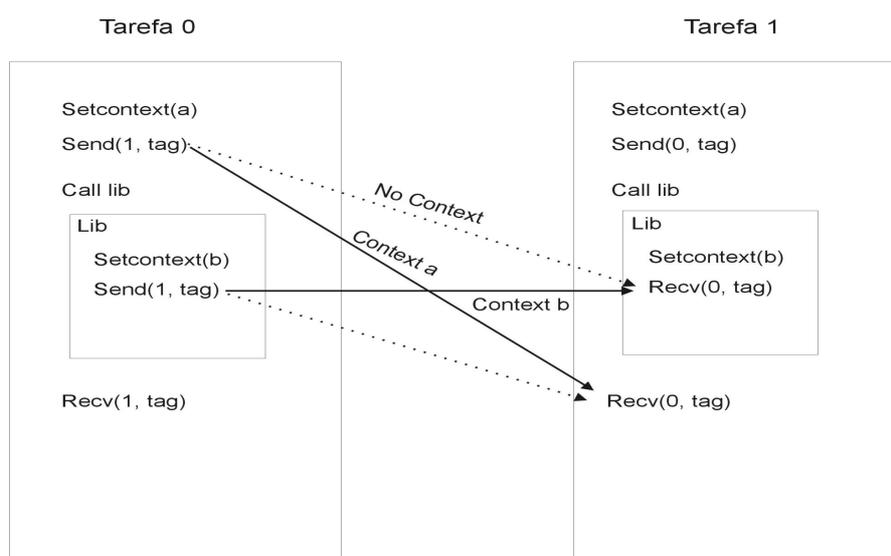
O padrão MPI não inclui qualquer mecanismo para tolerância à falhas, embora a nova especificação MPI-2 prometa suprir essa falta, criando um mecanismo similar ao PVM. O problema com MPI, em relação à tolerância à falhas, é que tanto as tarefas quanto os hosts que compõem o sistema paralelo, são considerados estáticos. Uma aplicação MPI precisa, no momento de sua inicialização, conhecer todas as tarefas e hosts que compõem o sistema. Se uma tarefa, ou recurso computacional falhar, a aplicação MPI inteira falhará. Embora isso seja realmente efetivo em termos de prevenir processos “pendurados” depois do término da aplicação, a falta dessa característica faz que muitas aplicações não possam ser desenvolvidas com esse sistema.

As razões para a natureza estática do MPI, são baseadas em performance e conveniência. Como todas as tarefas MPI estão sempre presentes, do início ao fim da aplicação, não é necessário que este sistema perca tempo descobrindo membros de grupos, novas tarefas, resolução de nomes para novos hosts, entre outros. Cada tarefa

sabe, já de início, sobre todas as outras tarefas, e todas as comunicações podem ser feitas sem a necessidade de um daemon especial para isso.

## 6.4. Contexto de Comunicação

Um dos mais importantes conceitos introduzidos pelo padrão MPI é o de um *comunicador*. Um comunicador pode ser pensado como a junção de um contexto de comunicação com um grupo de processos. Um contexto de comunicação permite um pacote de software – como uma biblioteca – escrito por terceiros, usar o sistema de passagem de mensagens de maneira segura, protegendo ou marcando suas mensagens para que elas não sejam recebidas incorretamente pelo programa do usuário. Isso porque uma tag de mensagem e o ID do emissor podem não ser o suficiente para assegurar uma distinção entre mensagens de uma biblioteca de terceiros e mensagens do programa do usuário, caso esse último use essa biblioteca. A figura 6.1 ilustra esse problema. Na figura, duas tarefas idênticas estão chamando uma rotina de biblioteca que também executa passagem de mensagens. A biblioteca e a aplicação do usuário escolheram ambas a mesma tag para marcar suas mensagens. Sem um contexto de comunicação, as mensagens seriam recebidas na ordem errada.



**Figura 6.1.** Mensagens enviadas com um contexto de comunicação.

Até recentemente, PVM não possuía o conceito de contexto de comunicação, tendo essa característica agregada às suas funcionalidades a partir da versão 3.4 do produto. Mesmo assim, PVM não obriga o programador a usá-la. Caso o programador queira criar um contexto de comunicação, deverá chamar as funções necessárias para isso. MPI, no entanto, possui um contexto implícito em cada operação de comunicação, o que não demanda tempo de processamento com chamadas às novas funções.

## 6.5. Tipos Definidos Pelo Usuário

Quando um processo PVM necessita enviar um determinado conjunto de dados à outro processo, o programador dispõe do recurso de dividir esse conjunto de dados em unidades menores e “empacotar” esses dados, permitindo que esse processo realize uma única operação de envio, diminuindo a taxa de comunicação na rede.

MPI também possui funções de empacotamento, mas estende as alternativas para permitir que o programador crie outros tipos de dados, a partir dos tipos existentes. Embora essa operação seja um pouco complexa, isso é muito útil quando deseja-se enviar um conjunto grande e complexo de dados várias vezes através da rede do cluster. Isso porque a operação de empacotamento, apesar de mais simples, possui a desvantagem de necessitar que, a cada vez que um processo deseja efetuar um novo envio, uma nova operação de empacotamento com os mesmos dados deve ser efetuada, o que consome um tempo significativo no fim da aplicação.

O paradigma de tipos de dados definidos pelo usuário, permite que uma estrutura complexa seja definida no início de uma aplicação paralela como um novo tipo de dado, o qual pode ser usado posteriormente por qualquer processo a qualquer momento. Sendo assim, uma tarefa pode enviar esse novo tipo de dado à outra, economizando tempo de processamento sem a chamada às funções de empacotamento e baixando a taxa de transmissão da rede.

Sendo assim, PVM mantém sua filosofia de manter as coisas o mais simples possível ao usuário, enquanto a filosofia MPI continua sendo a performance acima de tudo.

## 7. TRABALHO IMPLEMENTADO

Nesta parte do trabalho, tem-se a apresentação dos algoritmos de processamento paralelo implementados, bem como as técnicas usadas para a realização dessa tarefa. Também se faz presente neste capítulo, a apresentação do cluster Beowulf implantado pelo acadêmico nas dependências do LABCOM, assim como suas características e configurações.

### 7.1. Implantação do Cluster Beowulf

Para implantação do cluster Beowulf, necessário para a conclusão deste projeto, foram utilizadas as instalações do LABCOM – Laboratório de Computação da UNOESC, campus de Chapecó – o qual se mostrava disponível e preenchia os requisitos necessários para tal feito. Compunha o cluster:

- 7 máquinas Pentium III, de 650 MHz e 128 MB de memória RAM cada. Embora não seja necessário, todas as máquinas possuem periféricos, como monitor e teclado;
- Rede de interconexão dedicada entre as máquinas, sendo esta rede de topologia Fast Ethernet, com uma taxa de transmissão de 100 Mbits/s;
- Sistema Operacional Linux, kernel 2.2.14, instalado em todas as máquinas;
- Biblioteca de processamento paralelo PVM, do Oak Ridge National Laboratory, versão 3.4;
- Biblioteca de processamento paralelo MPICH, do Argonne National Laboratory, concordante com a especificação MPI-1, de 1994.

É importante salientar que todos os componentes usados na construção do cluster respondem às características necessárias para que este seja classificado como um cluster Beowulf, pois é composto de componentes comuns de hardware e software distribuído livremente. Todas as implementações e testes foram realizados sobre essa

arquitetura. Também foi usada uma 8ª máquina, com configuração idêntica às demais, para realização de testes com a versão monoprocessada da aplicação.

Para execução do ambiente paralelo, um usuário com poderes limitados à essa função foi criado localmente em cada máquina, para que este pudesse efetuar login e executar comandos remotamente, via rsh (Remote Shell). Foi delegada a uma das máquinas, a responsabilidade de servir como console do cluster, sobre o qual eram iniciados os serviços necessários para configuração do ambiente e a partir de onde as aplicações eram iniciadas. O compartilhamento de um diretório público entre as máquinas do cluster, utilizando NFS (Network File System), também se fez necessário, assim como a configuração de um DNS (Domain Name Service), para que se pudesse resolver nomes de máquinas, requisito necessário para que o serviço de rsh funcione corretamente.

Outras exigências menos relevantes, bem como entradas em arquivos específicos, variáveis de ambiente e demais configurações particulares à cada uma das bibliotecas não se faz necessário descrever aqui.

## **7.2. Aplicação Implementada**

A aplicação implementada por este trabalho, tem como objetivos verificar e provar as diferenças de metodologia de trabalho e desempenho de ambas bibliotecas PVM e MPI, bem como comprovar a melhoria de performance sobre a mesma aplicação programada com base no modelo serial convencional.

Essa aplicação é basicamente fundamentada em cálculos numéricos, como computação matemática sobre matrizes. O objetivo é tentar simular as condições computacionais numéricas que um problema científico necessitaria resolver, já que estes geralmente são baseados em cálculos matemáticos, com seqüências muito longas de dados científicos. Obviamente, o algoritmo aqui apresentado não necessita ser tão ambicioso e se satisfaz com algumas operações sobre grandes matrizes, alcançando o objetivo proposto pelo acadêmico.

O funcionamento dessa aplicação paralela, consiste, basicamente, em um processo servidor (ou processo pai) criar, inicializar, distribuir, receber e gerenciar um

número  $X$  de matrizes de ordem  $N \times N$  (quadradas), sendo esses valores informados pelo usuário via linha de comando. Esse processo servidor não executa nenhum cálculo, deixando essa tarefa para os processos filhos gerados, os quais receberão as matrizes e efetuarão operações de multiplicação sobre estas. O número de processos criados pela aplicação é dependente do número de matrizes a serem calculadas. Além disso, o modelo de implementação utilizado foi o SPMD (Single Program, Multiple Data), no qual várias instâncias do mesmo código são geradas, mas cada uma atuando sobre dados diferentes.

### 7.2.1. Inicialização da Aplicação

Para iniciar a aplicação, o usuário deve informar o nome do executável e os parâmetros necessários, como o número de matrizes e a ordem destas, possuindo, para isso, um limite mínimo de 2 matrizes de ordem  $1 \times 1$ , e um limite máximo de 100 matrizes de ordem  $400 \times 400$  cada, o que totaliza 160.000 elementos em ponto flutuante para cada matriz. No caso do MPI, deve-se também informar o número de processos a serem criados, os quais permanecerão durante toda a vida da aplicação. Com PVM, os processos filhos serão criados e exterminados em tempo de execução.

Assim que o executável inicia, faz-se uma verificação de qual é o processo pai da aplicação, o qual terá a responsabilidade de gerenciar a computação dos demais processos. O processo servidor (ou processo pai), aloca espaço suficiente na memória para um vetor de tamanho  $ordem * ordem * numero\_matrizes$ , o qual conterá todas as matrizes geradas. Essas então são preenchidas com um valor constante em ponto flutuante e repassadas a uma rotina que as distribuirá pela rede. O motivo pelo qual as matrizes são instanciadas sempre com o mesmos valores, reside no fato de somente assim pode-se garantir uma comparação de desempenho precisa entre uma execução e outra.

O pseudo-código do início da aplicação fica então:

```
rotina_principal(argumentos_linha_comando)
{
    declaração_variáveis;
```

```

configuracao_ambiente_paralelo;
meuTaskId = recebe_Task_Id();
pai = Id_Processo_Pai();
se (pai == NaoTemPai) {
    // Eu sou o processo pai
    start_time = iniciaTempoProcessamento();
    recupera_argumentos_linha_comando;
    vetor_matrizes = alocaMemoria(numMat*ordem*ordem);
    inicializaMatrizes();
    distribui();
    end_time = fimTempoProcessamento();
    mostre(O algoritmo levou, end_time - start_time, segundos);
    liberaMemoria(vetor_matrizes)
}
senao {
    // Sou um filho
    -----
    -----
}
deixaAmbienteParalelo();
}

```

### 7.2.2. Distribuição de Matrizes e Controle dos Processos Filhos

A rotina *distribui()*, mostrada no pseudo-código acima, é responsável por gerar e controlar os processos filhos, distribuir as matrizes para estes processos e recebê-las depois de efetuadas todas as operações sobre elas.

A rotina é formada pelos seguintes passos:

1. Se o número de matrizes existentes for ímpar, multiplique a primeira matriz do vetor de matrizes pela última, guardando o resultado na primeira e excluindo a última matriz. Calcule o número de processos filhos como a metade do número de matrizes;
2. Decomponha o vetor de matrizes, juntando-as novamente, duas a duas, em vetores auxiliares;
3. Envie cada vetor auxiliar com duas matrizes, e a ordem destas, à exatamente  $num\_matrizes/2$  processos, onde *num\_matrizes* é o número de matrizes nesta iteração. Cada processo deverá receber apenas duas matrizes;
4. Aguarde resposta dos processos filhos, os quais retornarão com uma matriz resultante cada, que deverá ser armazenada novamente no vetor principal;

Esses 4 passos são executados em um laço, que finaliza quando somente duas matrizes forem enviadas à um único processo, o qual retornará com a matriz resultante.

### 7.2.3. Processos Filhos

No pseudo-código mostrado anteriormente, verifica-se que há um teste condicional que divide a aplicação em dois caminhos a serem tomados. O primeiro caminho será tomado apenas por um processo (o pai), o qual cuidará da distribuição de processos e dados pela rede do cluster. Todas as demais instâncias do código mostrado acima, serão consideradas “filhos” do primeiro processo. Assim, inevitavelmente, cairão no “senão” do trecho de código mostrado. A função de um processo filho é simplesmente receber duas matrizes do processo pai, e realizar a multiplicação destas, retornando a matriz resultante para o emissor. O pseudo-código de um processo filho ficaria então:

```

rotina principal(argumentos_linha_comando)
{
    -----
    -----
    -----
    senao {
        // Sou um filho
        faça {
            recebe();
        } enquanto_verdade;
        deixaAmbienteParalelo();
    }

rotina recebe()
{
    declaração variáveis;
    receba_do_processo_pai(ordem_das_matrizes);
    vetor = alocaMemoria(ordem*ordem*2);
    receba_do_processo_pai(vetor_com_duas_matrizes);
    matAux1[] = primeira_matriz;
    matAux2[] = segunda_matriz;
    matriz_resultado = matAux1[] * matAux2[];
    envia_para_processo_pai(matriz_resultado);
    -----
    -----
}

```

#### 7.2.4. Aplicação Monoprocessada

Um algoritmo programado sob a forma convencional, monoprocessada, também se fez necessário, a fim de posterior comparação entre resultados e metodologias. Esse algoritmo segue a mesma lógica das implementações paralelas, mas agora somente com um processo contendo todas as matrizes informadas pelo usuário, ficando esse processo resumido em realizar os cálculos sobre estas matrizes de forma puramente seqüencial. Todos os cuidados foram tomados para que a lógica fosse a mais eficiente possível em relação às implementações paralelas, isso para que se pudesse efetuar uma comparação correta entre ambas metodologias. Sendo assim, essa implementação resume-se basicamente nos seguintes passos:

1. Recupere os argumentos informados pelo usuário na linha de comando;
2. Aloque  $num\_matrizes * ordem * ordem$  de memória dinâmica para um vetor, o qual conterà as matrizes. Sendo  $num\_matrizes$  e  $ordem$  o número de matrizes e a ordem (quadrada) destas, respectivamente;
3. Multiplique todas as matrizes do vetor;
4. Imprima o resultado.

Esse tipo de algoritmo é muito comum e simples, portanto não se faz necessária uma abordagem mais detalhada sobre o mesmo.

## 8. COMPARAÇÃO ENTRE BIBLIOTECAS

Neste capítulo é realizada uma análise a respeito dos resultados obtidos com base na implementação realizada pelo acadêmico em ambas as bibliotecas, os quais são comparados principalmente no que diz respeito à performance, eficiência e facilidade de implementação.

### 8.1. Obtenção de Resultados

Para obtenção e posterior estudo dos resultados, foi empregada uma metodologia gradual de processamento. Essa metodologia consistia, inicialmente, na execução das aplicações sobre o cluster Beowulf com uma taxa mínima de cálculos a serem realizados, os quais eram incrementados gradativamente. Cada execução era realizada três vezes, e os dados obtidos eram então submetidos à uma média. Um exemplo é mostrado na tabela 8.1.

**Tabela 8.1.** Média obtida sobre valores de três diferentes execuções.

Nome do Executável	Número de Matrizes	Ordem de cada Matriz	Tempo1 (segundos)	Tempo2 (segundos)	Tempo3 (segundos)	Média (segundos)
NMTX	75	250 X 250	121	120	120	120,333333
NMTX	100	400 X 400	712	708	713	711
NMTX_PVM	75	250 X 250	42	42	41	41,666666
NMTX_PVM	100	400 X 400	250	255	264	265,333333
NMTX_MPI	75	250 X 250	48	48	52	49,333333
NMTX_MPI	100	400 X 400	206	204	206	205,333333

Faz-se necessário observar que, nos momentos em que os dados eram coletados, o cluster era isolado de qualquer outra rede, utilizando-se um hub separado da rede da Universidade. Além disso, os serviços executados nas máquinas desse cluster

eram reduzidos ao mínimo. Isso permitia que possíveis interferências externas, como requisições em broadcast provenientes de outras máquinas da rede, ou serviços executados repentinamente, como atualização da crontab do Linux, não mudassem os resultados das execuções.

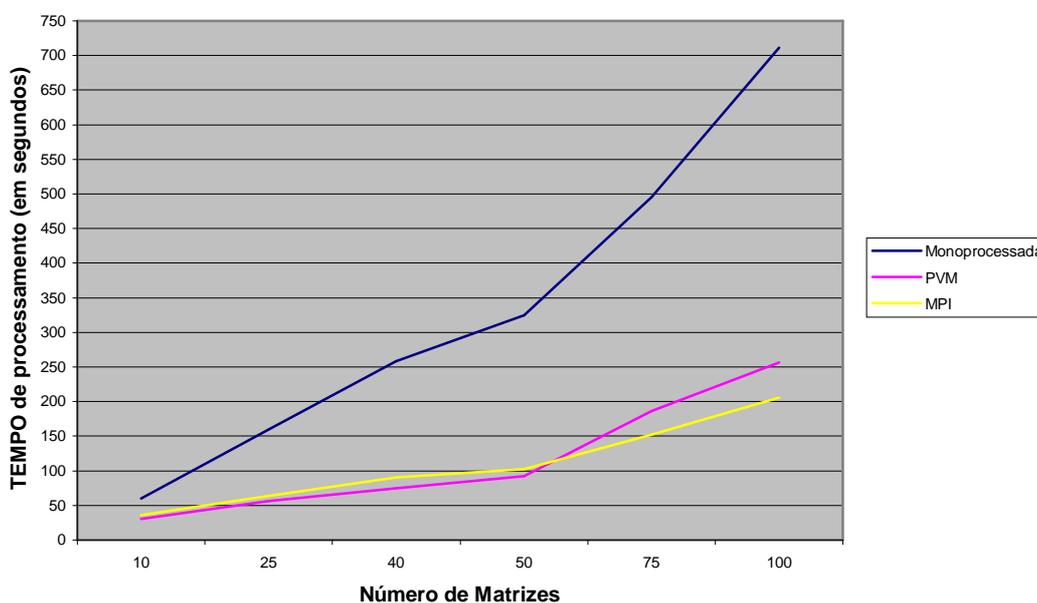
## 8.2. Apresentação dos Resultados Obtidos

Através dos resultados obtidos, prova-se que a paralelização de aplicações que demandam um alto poder de processamento, diminui drasticamente o tempo de execução computacional destas, principalmente se comparado com aplicações que rodem em computadores convencionais monoprocessados. Independente de qual das duas bibliotecas de processamento paralelo apresentadas neste trabalho o programador optar para desenvolver seu projeto, o ganho em performance se mostrará elevado.

Como a totalidade dos resultados obtidos é ampla demais para ser apresentada neste capítulo, somente serão abordadas as diferenças de tempo de execução mais relevantes. No gráfico 8.1, é mostrado os tempos de execução obtidos com matrizes quadradas de ordem 400 X 400 elementos em ponto flutuante cada, onde pode-se observar um substancial ganho de performance das execuções paralelas em relação à versão serial da aplicação. É importante reparar no comportamento das linhas que representam o uso das bibliotecas paralelas.

Vale ressaltar que os resultados foram obtidos com a configuração de cluster apresentada anteriormente, e que a possível agregação de um maior número de máquinas neste cluster, influirá em uma performance ainda mais elevada.

**Gáfico 8,1: Computação paralela utilizando matrizes quadradas de ordem 400 X 400 elementos (em ponto flutuante).**



Aqui, a diferença de tempo de processamento é de mais de 70%, entre o maior e o menor valor.

Nota-se também que a aplicação utilizando PVM, realiza um desempenho superior enquanto o número de matrizes é pequeno. Assim que esse número aumenta, aumentando também o throughput<sup>6</sup> sobre a rede, MPI passa a desempenhar uma performance superior, finalizando com uma diferença grande se comparada ao início da aplicação (mais de 20%).

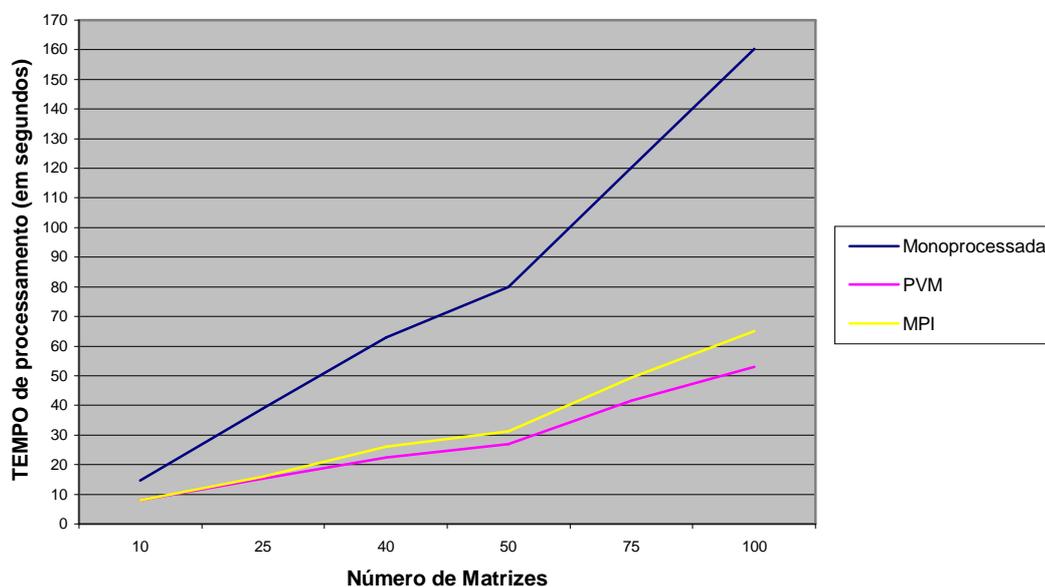
No gráfico 8.2, a granularidade<sup>7</sup> dos processos é mais fina do que no gráfico anterior, pois essa execução abrange matrizes de ordem inferior, agora de 250 X 250 elementos. O interessante aqui é que, com uma granularidade mais fina, PVM mostra um desempenho superior, não importando o número de matrizes.

---

<sup>6</sup> Taxa de transmissão sobre a rede

<sup>7</sup> O conceito de granularidade está intrinsecamente ligada à demanda por processamento. Quanto maior o uso de CPU por um processo em particular, dizemos que mais grossa é a sua granularidade. Inversamente, quanto menor o uso de CPU, mais fina é sua granularidade.

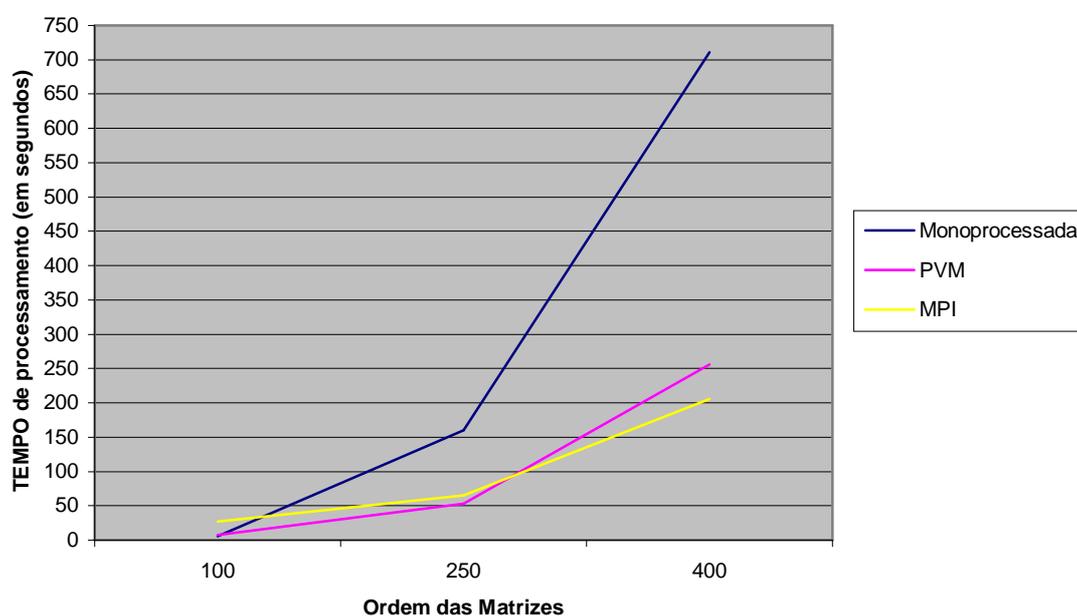
Gráfico 8,2: Computação paralela utilizando matrizes quadradas de ordem 250 X 250 elementos (em ponto flutuante).



A seguir, o gráfico 8.3 mostra um número fixo de matrizes quadradas sendo processadas (100 matrizes). Esse gráfico em particular talvez seja o mais interessante, pois mostra o aumento gradativo da ordem das matrizes e, conseqüentemente, o aumento da granularidade dos processos.

Como pode-se observar neste gráfico, quando a demanda por processamento é pequena (ordem 100 X 100), a aplicação monoprocessada é a mais eficaz, pois não necessita perder tempo com a “criação” do ambiente paralelo. Entende-se por criação, a necessidade que a aplicação paralela tem em logar em cada máquina do cluster, criar os processos nestas máquinas, efetuar o balanceamento de carga, entre várias outras tarefas que uma biblioteca de processamento paralelo necessita dispor antes de iniciar a execução propriamente dita. Mas essa vantagem é logo consumida, assim que a necessidade de processamento aumenta.

Gráfico 8,3: Computação paralela utilizando 100 matrizes quadradas de ordem variável.



Aqui, prova-se também os dois gráficos anteriores, pois nota-se que a aplicação utilizando PVM, possui uma performance superior enquanto a granularidade dos processos e o tamanho das mensagens são pequenas, sendo rapidamente ultrapassada por MPI, assim que a tamanho das mensagens aumenta e o uso de CPU se torna mais necessário. Esta mesma conclusão é obtida com base no *speedup*<sup>8</sup> alcançado por estes algoritmos, mostrado na tabela 8.2.

Nesta tabela pode-se observar que o speedup obtido por PVM diminui, conforme aumenta o tamanho das mensagens trocadas pelos processos da aplicação. MPI, ao contrário, aumenta seu ganho de desempenho conforme o paralelismo é mais exigido, em termos de necessidade de processamento sobre cada processador e sobre a rede.

---

<sup>8</sup> *Speedup* é o ganho de desempenho de um algoritmo paralelo sobre sua versão serial. A fórmula para obtenção do speedup é mostrado no Capítulo 2.

**Tabela 8.2.** Speedup obtido com os algoritmos paralelos com base no cluster apresentado.

EXECUTÁVEL	Nº MATRIZES	ORDEM	SPEEDUP
NTMX_PVM	50	250 x 250	2,962
NTMX_PVM	100	400 x 400	2,762
NTMX_MPI	50	250 x 250	2,553
NTMX_MPI	100	400 x 400	3,448

### 8.3. Pontos Observados

#### 8.3.1. PVM

A implementação utilizando esta biblioteca se mostrou significativamente mais fácil do que com MPI. As funções e metodologias de trabalho impostas por PVM, tentam livrar o programador de maiores detalhes, tentando sempre manter uma linguagem de mais alto nível.

Caso a aplicação à ser paralelizada possuir uma taxa de processamento não muito alta por processo (granularidade média à fina), aliado à mensagens de pequeno tamanho sobre a rede, esta biblioteca desempenha um papel superior em termos de performance.

Como PVM possibilita que o usuário “construa” manualmente seu ambiente paralelo antes do início da aplicação, inicializando os deamons nas máquinas do cluster, essa responsabilidade é tirada da aplicação. Isso faz com que o tempo de latência no início da aplicação seja pequeno, como mostrado nos gráficos.

### 8.3.2. MPI

A implementação com MPI se mostrou mais complexa, se comparada com PVM. Isso porque essa biblioteca deixa ao programador a alternativa de tratar detalhes de mais baixo nível, como tratamento de buffers, por exemplo. Isso pode ser uma vantagem se o programador desejar – e conseguir – otimizar a aplicação para determinadas condições de trabalho.

Devido à natureza estática de MPI, essa biblioteca não possui a alternativa de um pré-carregamento manual do ambiente paralelo, como PVM. Isso faz com que todas as necessidades sejam supridas no momento do início da aplicação paralela, ou seja, é só neste momento que MPI faz a verificação de quais máquinas estão presentes no cluster, efetua login em cada uma dessas máquinas e cria o número de processos indicados pelo usuário. Dessa forma, há um tempo de latência muito grande no início da aplicação, como pode ser observado no gráfico 8.3.

Sendo assim, a paralelização com MPI possui uma performance superior não só em casos onde a granularidade dos processos é mais grossa, como em casos onde o tamanho das mensagens é muito grande, possibilitando uma menor frequência de comunicação sobre a rede.

## 9. CONCLUSÃO

O crescente aumento de aplicações que demandam um alto poder de processamento, como computação numérica intensiva e movimentação de grandes volumes de dados, aliado a barreiras físicas e econômicas de evolução de novas arquiteturas de processamento, vem despontando uma necessidade de se achar novas formas de tratamento computacional para atender requisitos científicos e comerciais.

O uso de um cluster Beowulf, não só representa uma solução para casos onde a necessidade de processamento e performance é fundamental, como oferece uma alternativa de menor custo às dispendiosas máquinas paralelas convencionais.

A paralelização de uma aplicação não é uma tarefa trivial. Existem casos onde a paralelização simplesmente é impossível, como, por exemplo, em programas onde as partes do código são muito dependentes entre si. Nestes casos, uma abordagem de programação paralela pode trazer resultados desapontadores. A concepção de um algoritmo paralelo envolve várias etapas, desde a definição das atividades que podem ser desempenhadas em paralelo, até o equilíbrio da carga dos processos pelos processadores do cluster.

Embora existam várias APIs que possibilitam um ambiente de troca de mensagens entre processos de uma máquina MIMD de memória distribuída, as que mais tem se destacado são as bibliotecas de processamento paralelo PVM e MPI.

Até pouco tempo, PVM era o padrão de facto em computação distribuída. Mas as recentes publicações a respeito da especificação MPI, levam programadores a decidir se devem implementar suas aplicações com o padrão de facto, PVM, ou com o padrão de jure, MPI. Esta decisão é ainda mais difícil para pesquisadores que necessitam de alguma referência em língua portuguesa, já que bibliografia a respeito praticamente inexistente nesta língua.

Com base neste trabalho, pode-se observar que MPI possui a vantagem de possibilitar alta performance de comunicação, o que é uma vantagem quando se está desenvolvendo aplicações que fazem uso de um grande sistema paralelo, como sistemas MPP, ou clusters homogêneos com muitas máquinas agregadas. Adicionalmente, MPI possui um conjunto rico de funções de comunicação, que favorece aplicações que fazem

uso de algum modo especial de comunicação, que PVM não suporta. Um exemplo é a função de envio de mensagem sem bloqueio.

Mas para prover esta alta performance, alguns sacrifícios tem sido feitos pelo padrão MPI. Destes sacrifícios, três se destacam: a falta de interoperabilidade entre diferentes implementações MPI, a falta da habilidade em escrever aplicações tolerante à falhas, e, por fim, a falta de controle de processos dinâmicos. Características presentes na biblioteca PVM. Além disso, PVM possui a preocupação em manter uma interface de programação de alto nível ao desenvolvedor, enquanto MPI se dirige à programadores mais experientes.

Sendo assim, pode-se afirmar que a melhor escolha sobre qual biblioteca utilizar, depende muito da aplicação a ser paralelizada, assim como a maneira que esta aplicação irá ser paralelizada e, principalmente, sobre que tipo de máquina paralela ela vai ser executada.

Como a construção de um cluster Beowulf é baseada em torno de PCs comuns, a necessidade de suporte à heterogeneidade, provida por PVM, não se faz necessária. Outro ponto relevante, é que, como o objetivo de um cluster Beowulf é a performance, aspectos providos por PVM, como balanceamento de carga e tolerância à falhas ficam em segundo plano, apesar de também serem importantes.

Assim, com base nas experiências obtidas com a realização deste trabalho, conclui-se que, MPI apresenta-se como a melhor escolha quando a aplicação paralela possui processos com granularidade média ou grossa, e o tamanho das mensagens trocadas entre processos for grande. Ou seja, MPI é mais indicado em grandes aplicações paralelas, pois possibilita uma performance superior em relação a biblioteca PVM. Em contrapartida, se a necessidade de gerenciamento de recursos for uma necessidade ou os processos da aplicação possuírem uma granularidade fina, com mensagens pequenas, PVM se mostra como a melhor escolha.

## 10. REFERÊNCIAS BIBLIOGRÁFICAS

- [AMO 88] AMORIN, Cláudio Luis de; BARBOSA, Valmir Carneiro; FERNÁNDEZ, Edil Severiano Tavares. Uma introdução à Computação Paralela e Distribuída. Campinas: UNICAMP, IMECC, 1988. 258p.
- [BEC 95] BECKER, Donald; FRYXELL, Bruce; OLSON, Kevin; SAVARESE, Daniel; STERLING, Thomas. Communication Overhead for Space Science Applications on the Beowulf Parallel Workstation. High Performance and Distributed Computing, 1995
- [BEC 97] BECKER, Donald; MERKEY, Philip; RIDGE, Daniel; STERLING, Thomas. Beowulf: Harnessing the Power of Parellelism in a Pile-of\_PC's. IEEE Aerospace, 1997.
- [BEG 95] BEGUELIN, Adam; DONGARRA, Jack; GEIST, Al; MANCHEK, Robert; SUNDERAM, Vaidy. Recent Enhancements to PVM. International Journal for Supercomputer Applications, Vol. 9, No. 2, 1995.
- [BEO 00] BOWULF PROJECT. The Beowulf Project. Disponível por WWW em <http://www.beowulf.org> (12 Ago 2000).
- [BEO 00A] BOWULF MAILING LIST FAQ. The Beowulf Mailing List FAQ, Version 2. Disponível por WWW em <http://www.dnaco.net/~kragen/beowulf-faq.txt> (12 Ago. 2000).
- [BEO 00B] BOWULF HOWTO. The Beowulf Documentation Project. Disponível por WWW em [http://www.beowulf-underground.org/doc\\_project/HOWTO/english/Beowulf-HOWTO.ps](http://www.beowulf-underground.org/doc_project/HOWTO/english/Beowulf-HOWTO.ps) (13 Ago. 2000).
- [BEO 00C] BOWUFL HOWTO. Beowulf HOWTO. Disponível por WWW em <http://www.linuxdoc.org/HOWTO/Beowulf-HOWTO.html> (15 Ago. 2000).

- [BUY 00] BUYYA, Rajkumar; SILVA, Luís Moura. Parallel Programming Models and Paradigms. Disponível por WWW em <http://www-asc.di.fct.unl.pt/~pm/PP/v2chap1.ps.gz>. (01 Out. 2000).
- [CEN 00] CENAPADNE. Centro Nacional de Alto Desempenho do Nordeste. Encontrado por WWW em <http://www.cenapadne.br> (18 Ago 2000).
- [CUL 97] CULLER, David E. Parallel Computer Architecture. San Francisco: Morgan Kaufmann Publishers, 1997.
- [CUL 99] CULLER, David E.; SINGH, Jaswinder Pal; GUPTA, Anoop. Parallel Computer Architecture. San Francisco: Morgan Kaufmann Publishers, 1999. 1025p.
- [DON 93] DONGARRA, Jack; GEIST, G.; MANCHEK, Robert; SUNDERAM, V. Integrated PVM Framework Supports Heterogeneous Network Computing. Computers in Physics, Vol. 7, No. 2, p. 166-175, Abril 1993.
- [DON 94] DONGARRA, Jack; GEIST, G.; MANCHEK, R.; SUNDERAM, V. The PVM Concurrent Computing System: Evolution, Experiences and Trends. Parallel Computing, Vol. 20, No. 4, p. 531-547, Abril 1994.
- [DON 95] DONGARRA, Jack J.; OTTO, Steve W.; SNIR, Marc; WALKER, David. An Introduction to the MPI Standard. CS-95-274, Janeiro 1995.
- [DON 96] DONGARRA, Jack J.; WALKER, David W. MPI: A Standard Message Passing Interface. Supercomputer 96, Vol 12, No 1, p. 56-68, Janeiro 1996.
- [FLY 72] FLYNN, M. J. Some Computer Organizations and Their Effectiveness. IEEE Trans. Computers, 1972.
- [FOS 95] FOSTER, I. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley, 1995.
- [GEI 95] GEIST, G. A.; KOHL, J. A.; PAPADOPOULOS, P. M. New Features of PVM 3.4 and Beyond. Paris: Hermes Publishing, 1995.

- [GEI 96] GEIST, G. A.; KOHL, J. A.; PAPADOPOULOS, P. M. PVM and MPI: A Comparison of Features. *Calculateurs Paralleles* Vol. 8 No. 2, Março 1996.
- [GRO 98] GROPP, William; LUSK, Ewing. PVM and MPI are Completely Different. Argonne National Laboratory, Mathematics and Computer Science Division, 1998.
- [LIN 00] LINUX PARALLEL PROCESSING HOWTO. The Linux Parallel Processing HOWTO. Encontrado por WWW em <http://yara.ecn.purdue.edu/~pplinux/PPHOWTO/pphowto.html>. (18 Ago. 2000).
- [MED 98] MEDEIROS; Pedro de. Introdução ao Processamento Paralelo. Disponível por WWW em <http://www-asc.di.fct.unl.pt/~pm/PP/introPP.ps.gz> (02 Out. 2000).
- [MID 96] MIDORIKAWA, Edson Toshimi; SATO, Liria Matsumoto; Senger, Hermes. Introdução a Programação Paralela e Distribuída. Disponível por WWW em <http://www.lsi.usp.br/~liria/jai96/apost.ps> (1 Out. 2000)
- [MPI 00] MPI. MPI: The Complete Reference. Encontrado por WWW em <http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>. (21 Ago 00).
- [PAC 98] PACHECO, Peter S. A User's Guide to MPI. 1994. Disponível por FTP anônimo em <ftp://math.usfca.edu/pub/MPI/mpi.guide.ps.Z> (15 Out. 2000).
- [PVM 00] PVM. Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing. Encontrado por WWW em <http://www.netlib.org/pvm3/book/pvm-book.html>. (15 Ago 2000).
- [REV 00] REVISTA DO LINUX. Revista do Linux. Disponível por WWW em <http://www.revistadolinux.com.br/ed/002/beowulf.php3> (01 Ago 2000).

- [SPE 00] SPECTOR, David. Building Linux Clusters: Scaling Linux for Scientific and Enterprise Applications. Sebastopol: O'Reilly & Associates, 2000. 332p.
- [WAL 94] WALKER, David W. The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers. Parallel Computing, Vol. 20, No. 4, p. 657-673, Abril 1994.