

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito.....19.03.99.

Assinatura:.....*Mauro Gaudraro*.....

## **Desenvolvimento e Avaliação de Algoritmos Numéricos Paralelos**

**Omar Andrés Carmona Cortés**

**Orientadora: Profa. Dra. Regina Helena Carlucci  
Santana**

Dissertação apresentada ao Instituto de Ciências  
Matemáticas e de Computação – USP, como parte dos  
requisitos para obtenção do título de Mestre em Ciências -  
Área: Ciências de Computação e Matemática  
Computacional.

São Carlos, março de 1999.

**A meu irmão e a minha mãe em  
memória.**

## Agradecimentos

A Profa. Dra. Regina Helena Carlucci Santana pela amizade, orientação e pela oportunidade de aprender tantos assuntos novos.

Ao Prof. Dr. Marcos Santana que me aceitou na pós-graduação.

À minha família, pelo carinho, apoio, paciência e por agüentar a saudade de estar tão longe.

A meu pai por acreditar em mim, a meu irmão por agüentar a saudade, a minha madrasta pela ajuda e a Iraney Colau por me apoiar.

A Porfa. Dra. Neide M. B. Franco que me ajudou muito com a parte matemática deste trabalho.

Aos amigos do LASD-PC (Aleteia, Sarita, Adriana, Paulo, Luciano, Renato, Marcio, Renata, Roberta, Andrezza, Jorge, Daniel, Kalinka, Célia, Flavio, Rita e alguém que eu esqueci) pela ajuda e pelos momentos de alegria.

Aos meus amigos especiais Cristina Endo, Marcos Alvarez e Paulo Horst pelos excelentes momentos de alegria e inclusive de tristeza compartilhados.

A meu excelente amigo Osvaldo Saavedra Mendez que teve um papel fundamental para que eu estivesse aqui.

A família Casale (Cidinha, Marcos e Elaine) que me recebeu como filho em especial a Elaine Cristina.

A Karla Conká, Cynthia M. Lima e Arturo Alejandro pela força, carinho e amor dados, a maior parte a distância, mas fundamentais para que eu conseguisse.

A meu eterno amigo Claudio Barbosa pela força e apoio.

A minha amiga Nilda que muito me ajudou.

A meu amigo Renato Haber pelo companheirismo e apoio.

Aos meus instrutores e amigo Sr. Marcos Turci e Sr. Bum Jun Kim por me mostrar o caminho da raça e do espírito forte.

Ao CISC, LCCA e Fapema sem os quais esse trabalho não seria possível.

A Rogério T. Kondo e a Sra. Maria de Lurdes do CISC pelo ótimo atendimento. Ao Francisco do LCCA pela paciência.

A todas as pessoas que contribuíram diretamente ou indiretamente para a realização deste trabalho.

# Índice

CAPÍTULO 1 .....	13
1. INTRODUÇÃO .....	13
CAPÍTULO 2 .....	16
2. COMPUTAÇÃO PARALELA .....	16
2.1 INTRODUÇÃO .....	16
2.2 CONCEITOS BÁSICOS .....	17
2.2.1 Concorrência e Paralelismo .....	17
2.2.2 Granulação .....	18
2.2.3 Speedup e Eficiência .....	18
2.2.4 Abordagem de programação .....	19
2.4 ARQUITETURAS PARALELAS .....	20
2.4.1 Classificação de Flynn .....	21
2.4.2 Classificação da Duncan .....	23
2.5 COMPUTAÇÃO VETORIAL .....	25
2.6 PROGRAMAÇÃO CONCORRENTE .....	29
2.5.1 PVM (Parallel Virtual Machine – Máquina Paralela Virtual) .....	30
2.5.2 MPI (Message Passing Interface – Interface de Passagem de Mensagens) .....	34
2.6 CONSIDERAÇÕES FINAIS .....	37
CAPÍTULO 3 .....	40
3. BENCHMARKS .....	40
3.1 INTRODUÇÃO .....	40
3.2 CONCEITOS BÁSICOS .....	42
3.2.1 Classificação dos benchmarks .....	43
3.2.2 Execução de um benchmark .....	45
3.3 PADRONIZAÇÃO DOS BENCHMARKS PARALELOS .....	45
3.3.1 Metodologia .....	46
3.3.1.1 Símbologia .....	47
3.3.1.2 Medidas de tempo .....	47
3.3.1.3 Contagem de operações com ponto flutuante .....	48
3.3.1.4 Benchmarks para aplicações científicas .....	49
3.4 SPEEDUP E SUAS RESTRIÇÕES .....	50
3.5 EXEMPLOS DE BENCHMARKS PARALELOS .....	52
3.5.1 Genesis .....	52
3.5.2 NAS Parallel Benchmarks .....	54
3.5.3 ParkBench (Parallel Kernel Benchmark) .....	55
3.6 OUTROS EXEMPLOS DE BENCHMARKS [WAY95] .....	55
3.7 CONSIDERAÇÕES FINAIS .....	56
CAPÍTULO 4 .....	58
4. ALGORITMOS NUMÉRICOS .....	58
4.1 INTRODUÇÃO .....	58
4.2 ERROS EM ALGORITMOS NUMÉRICOS .....	62
4.3 Exemplos de Algoritmos Numéricos .....	63
4.3.1. Resolução de Sistemas Lineares por métodos iterativos .....	63
4.3.2 Integração Numérica .....	66
4.3.3 Multiplicação de matrizes .....	69
4.4 CONSIDERAÇÕES FINAIS .....	70

CAPÍTULO 5.....	71
<b>5. DESCRIÇÃO DO AMBIENTE .....</b>	<b>71</b>
5.1 HARDWARE UTILIZADO .....	71
5.1.1 <i>O IBM SP2</i> .....	71
5.1.2 <i>Sistema Cray de Computadores Vetoriais</i> .....	73
5.1.3 <i>Sistema Distribuído do LASD-PC</i> .....	74
5.2 SOFTWARE UTILIZADO .....	75
5.2.1 <i>PVM, PVMe, MPI para IBM SP2</i> .....	75
5.2.2 <i>O Depurador xpdbx</i> .....	81
5.2.3 <i>Sistemas Cray de Compilação e Execução de Programas</i> .....	83
5.2.4 <i>PVM e MPI no LASD-PC</i> .....	85
5.2.5 <i>Visualization Tool (VT)</i> .....	86
5.2.6 <i>Totalview</i> .....	90
5.2.7 <i>Proofview</i> .....	91
5.3 CONSIDERAÇÕES FINAIS.....	93
CAPÍTULO 6.....	94
<b>6. APLICAÇÕES .....</b>	<b>96</b>
6.1 MÉTODO DE JACOBI.....	96
6.2 MÉTODO DOS TRAPÉZIOS COMPOSTO.....	99
6.3 MULTIPLICAÇÃO DE MATRIZES .....	101
6.4 PARALELIZAÇÃO VETORIAL .....	103
6.5 COMPILAÇÃO E EXECUÇÃO DO <i>NAS PARALLEL BENCHMARK</i> , <i>BENCHMARK</i> .....	103
6.6 CONSIDERAÇÕES FINAIS .....	106
CAPÍTULO 7.....	106
6.6 CONSIDERAÇÕES FINAIS .....	107
CAPÍTULO 7.....	109
<b>7. RESULTADOS.....</b>	<b>109</b>
7.1 BALANCEAMENTO DE CARGA E UTILIZAÇÃO DO PROCESSADOR.....	110
7.2 EXECUÇÃO DOS ALGORITMOS .....	113
7.2.1 <i>Resultados do programa de integração numérica</i> .....	113
7.2.2 <i>Resultado do algoritmo de Jacobi</i> .....	126
7.2.3 <i>Resultado do programa de multiplicação de matrizes</i> .....	139
7.3 COMPARAÇÃO ENTRE AS APLICAÇÕES DESENVOLVIDAS E O <i>NAS</i> .....	155
7.4 CONSIDERAÇÕES FINAIS .....	156
CAPÍTULO 8.....	158
<b>8. CONCLUSÕES .....</b>	<b>158</b>
8.1 ANÁLISE DA REVISÃO BIBLIOGRÁFICA .....	158
8.2 COMENTÁRIOS SOBRE OS AMBIENTES UTILIZADOS .....	160
8.3 DISCUSSÃO SOBRE OS RESULTADOS OBTIDOS .....	161
8.4 DIFICULDADES ENCONTRADAS .....	163
8.5 SUGESTÕES PARA TRABALHOS FUTUROS.....	164
<b>REFERÊNCIAS BIBLIOGRÁFICAS.....</b>	<b>165</b>

# Lista de Figuras

Figura 2.1 - Exemplos de programação <i>N-Dimensional</i> .....	20
Figura 2.2 - Arquitetura SISD.....	21
Figura 2.3 - Arquitetura SIMD .....	21
Figura 2.4 - Arquitetura MIMD .....	22
Figura 2.5 -Tentativa de caracterização da arquitetura MISD. ....	22
Figura 2.6 - Classificação de Duncan.....	23
Figura 2.7 - Esquema simplificado de um computador vetorial.....	26
Figura 2.8 - Exemplos de termos utilizados na computação vetorial. ....	28
Figura 4.1 - Divisão da Matemática Computacional .....	59
Figura 4.2 – Exemplo de função $f(x)$ .....	68
Figura 4.3 - Regra do trapézio composta .....	68
Figura 5.1 : Exemplo de conexão do SP2 com diversos tipos de rede. ....	72
Figura 5.2 - <i>Switch</i> de alto desempenho. ....	73
Figura 5.3 - Estrutura básica de comunicação de um sistema Cray.....	74
Figura 5.4 - Rede LASD-PC para utilização do PVM e do MPI.....	74
Figura 5.5 - Exemplo de arquivo de comandos de tarefas. ....	76
Figura 5.6 - Ambiente gráfico do <i>loadlever</i> .....	79
Figura 5.7-Submetendo uma tarefa. ....	80
Figura 5.8 - Construindo um arquivo de comandos. ....	80
Figura 5.9 - Opções do menu <i>actions</i> . ....	81
Figura 5.10 - Tela do depurador <i>xpdbx</i> .....	82
Figura 5.11 – Exemplo de utilização da diretiva <i>#pragma</i> . ....	84
Figura 5.12 - Exemplo de um arquivo de submissão no Cray.....	84
Figura 5.13-Tela do seletor de janelas.....	87
Figura 5.14-Tela de replay.....	89
Figura 5.15 - Janelas de desempenho.....	89
Figura 5.16 – Exemplo de um sumário emitido pelo hpm. ....	91
Figura 5.17 - Tela do totalview (depurador) para sistemas Cray.....	92
Figura 5.18 – Opções mais importantes do <i>totalview</i> . ....	93
Figura 5.19 - Tela do <i>proofview</i> . ....	94
Figura 6.1-Estrutura da paralelização do algoritmo de Jacobi para 3 processadores.....	98
Figura 6.2 - Paralelização do algoritmo do trapézio composto. ....	100
Figura 6.3 - Paralelização do algoritmo de multiplicação de matrizes.....	102
Figura 6.4 – Alteração de código no programa de multiplicação de matrizes.....	103
Figura 7.1 - Utilização de processador por todas as aplicações. ....	110
Figura 7.2 - Balanceamento de carga onde só um processador está trabalhando. ....	111
Figura 7.3 – Balanceamento de carga para os algoritmos no SP2. ....	112
Figura 7.4 - Comunicação entre 3 processadores. ....	117
Figura 7.5 Comunicação entre 2 processadores .....	117
Figura 7.6 - Trecho final de comunicação no algoritmo de Jacobi. ....	128
Figura 7.7 - Execução do algoritmo de multiplicação de matrizes sem vetorização. ....	149
Figura 7.8 - Execução do algoritmo de multiplicação de matrizes com um nível agressivo de vetorização. ....	150

# Lista de Tabelas

Tabela 2.1 - Ordem das operações no modo escalar e no modo vetorial .....	26
Tabela 2.2 – Resumo dos <i>flags</i> utilizados no <i>pvm_spawn</i> .....	32
Tabela 2.3 – Funções de empacotamento de dados .....	33
Tabela 2.4 – Tipos de dados no MPI .....	37
Tabela 2.5 – Resumo das características das extensões paralelas de linguagens seriais .....	38
Tabela 3.1 - Nomenclatura estipulada pelo <i>Parkbench Comitee</i> .....	47
Tabela 2.2 - Contagem de operações em ponto flutuante .....	48
Tabela 3.3 - Exemplo de tempo e <i>speedup</i> .....	51
Tabela 3.4 - <i>Benchmarks</i> que formam o <i>ParkBench</i> .....	55
Tabela 4.1 - Resultados alcançados no processo de obtenção das iterações .....	66
Tabela 5.1 - Descrição das máquinas do LCCA .....	73
Tabela 5.2 – Máquinas utilizadas no LASD-PC .....	75
Tabela 5.3 - Compilação PVM e MPI no SP2 .....	75
Tabela 5.4 - Resumo das funções de cada botão no xpdbx .....	83
Tabela 5.5 - Diretivas utilizadas na compilação vetorial .....	83
Tabela 5.6 - Diretivas pragma e sua utilização .....	84
Tabela 5.7- Resumo dos comandos do console PVM .....	86
Tabela 5.8– Variáveis de sistema utilizadas pelo VT .....	87
Tabela 5.9 - <i>Flags</i> e eventos que podem ser capturados .....	88
Tabela 5.10 - Utilização de grupos no <i>hpm</i> .....	90
Tabela 6.1 – Classes e seus objetivos .....	105
Tabela 6.2 - <i>Benchmarks</i> executados com as respectivas classes .....	106
Tabela 6.3 - Resultados encontrados na execução do <i>NAS Parallel Benchmark</i> .....	106
Tabela 7.1 - Resultados encontrados no SP2 utilizando MPI com 3 processadores .....	114
Tabela 7.2 - Resultados encontrados no SP2 utilizando MPI com 2 processadores .....	114
Tabela 7.3 - Resultados encontrados no SP2 utilizando PVMe com 3 processadores .....	116
Tabela 7.4 - Resultados encontrados no SP2 utilizando PVMe com 2 processadores .....	116
Tabela 7.5 - Resultados encontrados no LASD-PC utilizando MPI com 3 processadores .....	119
Tabela 7.6 - Resultados encontrados no LASD-PC utilizando MPI com 2 processadores .....	119
Tabela 7.7 – Resultados encontrados no LASD-PC utilizando PVM com 3 processadores .....	121
Tabela 7.8 - Resultados encontrados no LASD-PC utilizando PVM com 2 processadores .....	121
Tabela 7.9 – Resultados encontrados no <i>Cray</i> utilizando 3 processadores .....	123
Tabela 7.10 - Resultados encontrados no <i>Cray</i> utilizando 2 processadores .....	123
Tabela 7.11 - Comparação entre arquiteturas com 3 processadores .....	125
Tabela 7.12 - Comparação entre arquiteturas com 2 processadores .....	126
Tabela 7.13 - Resultados encontrados no SP2 utilizando MPI com 3 processadores .....	127
Tabela 7.14 - Resultados encontrados no SP2 utilizando MPI com 2 processadores .....	127
Tabela 7.15 - Resultados encontrados no SP2 utilizando PVMe com 3 processadores .....	129
Tabela 7.16 - Resultados encontrados no SP2 utilizando PVMe com 2 processadores .....	130
Tabela 7.17 - Resultados encontrados no LASD-PC utilizando MPI com 3 processadores .....	131
Tabela 7.18 - Resultados encontrados no LASD-PC utilizando MPI com 2 processadores .....	131
Tabela 7.19 – Resultados encontrados no LASD-PC utilizando PVM com 3 processadores .....	133
Tabela 7.20 - Resultados encontrados no LASD-PC utilizando PVM com 2 processadores .....	134
Tabela 7.21 – Resultados encontrados no <i>Cray</i> utilizando 3 processadores .....	135
Tabela 7.22 - Resultados encontrados no <i>Cray</i> utilizando 2 processadores .....	135
Tabela 7.23 – Desempenho do <i>benchmark</i> para 3 processadores .....	137
Tabela 7.24 – Desempenho do <i>benchmark</i> para 3 processadores .....	138

Tabela 7.25 - Resultados encontrados no SP2 utilizando MPI com 3 processadores .....	139
Tabela 7.26 - Resultados encontrados no SP2 utilizando MPI com 2 processadores .....	140
Tabela 7.27 - Resultados encontrados no SP2 utilizando PVMe com 3 processadores.....	142
Tabela 7.28 - Resultados encontrados no SP2 utilizando PVMe com 2 processadores.....	142
Tabela 7.29 - Resultados encontrados no LASD-PC utilizando MPI com 3 processadores .....	143
Tabela 7.30 - Resultados encontrados no LASD-PC utilizando MPI com 2 processadores .....	144
Tabela 7.31 – Resultados encontrados no LASD-PC utilizando PVM com 3 processadores .....	145
Tabela 7.32 - Resultados encontrados no LASD-PC utilizando PVM com 2 processadores.....	145
Tabela 7.33 – Resultados encontrados no <i>Cray</i> utilizando 3 processadores .....	147
Tabela 7.34 - Resultados encontrados no <i>Cray</i> utilizando 2 processadores.....	147
Tabela 7.35 – Resumo do desempenho de <i>benchmark</i> para 3 processadores.....	151
Tabela 7.36 - Resumo do desempenho de <i>benchmark</i> para 2 processadores .....	153
Tabela 8.1 – Resumo dos resultados obtidos na execução dos algoritmos e dos <i>benchmarks</i> . ....	155

# **Lista de Gráficos**

Gráfico 7.1	- Tempo de execução do algoritmo do trapézio composto no SP2 utilizando MPI .....	115
Gráfico 7.2	- Speedup e eficiência alcançada por 2 e 3 processadores no SP2 .....	115
Gráfico 7.3	- Tempo de execução do algoritmo do trapézio composto no SP2 utilizando-se PVMe.....	118
Gráfico 7.4	- Speedup e eficiência alcançados por 2 e 3 processadores no SP2 utilizando PVMe.....	118
Gráfico 7.5	- Tempo de execução do algoritmo do trapézio composto no LASD-PC utilizando MPI .....	120
Gráfico 7.6	- Eficiência e speedup alcançados no LASD-PC utilizando MPI.....	120
Gráfico 7.7	- Tempo de execução do algoritmo do trapézio composto no LASD-PC utilizando PVM.....	122
Gráfico 7.8	- Eficiência e speedup alcançados no LASD-PC utilizando PVM .....	122
Gráfico 7.9	- Tempo de execução do algoritmo do trapézio composto no sistema Cray. ....	124
Gráfico 7.10	- Eficiência e speedup alcançados no sistema Cray.....	124
Gráfico 7.11	- Comparação entre arquiteturas com 3 processadores executando o algoritmo do trapézio. ....	125
Gráfico 7.12	- Comparação entre arquiteturas com 3 processadores executando o algoritmo do trapézio .....	126
Gráfico 7.13	- Tempo de execução do algoritmo de Jacobi no SP2 utilizando o MPI.....	128
Gráfico 7.14	- Speedup e eficiência alcançados por 2 e 3 processadores no SP utilizando MPI.....	129
Gráfico 7.15	- Tempo de execução do algoritmo de Jacobi no SP2 utilizando PVMe. ....	130
Gráfico 7.16	- Speedup e eficiência alcançados por 2 e 3 processadores no SP2 utilizando PVMe.....	131
Gráfico 7.17	- Tempo de execução do algoritmo de Jacobi no LASD-PC utilizando MPI.....	132
Gráfico 7.18	- Speedup e eficiência alcançados por 2 e 3 processadores no LASD-PC utilizando MPI .....	133
Gráfico 7.19	- Tempo de execução do algoritmo de Jacobi no LASD-PC utilizando PVM. ....	134
Gráfico 7.20	- Speedup e eficiência alcançados por 2 e 3 processadores no LASD-PC utilizando PVM.....	135
Gráfico 7.21	- Tempo de execução do algoritmo de Jacobi na sistema Cray. ....	136
Gráfico 7.22	- Speedup e eficiência alcançados no sistema Cray.....	137
Gráfico 7.23	- Comparação entre arquiteturas com 3 processadores executando o algoritmo de Jacobi .....	138
Gráfico 7.24	- Comparação entre arquiteturas com 2 processadores executando o algoritmo de Jacobi .....	139
Gráfico 7.25	- Tempo de execução do algoritmo de multiplicação de matrizes no SP 2 utilizando MPI. ....	140
Gráfico 7.26	- Speedup e eficiência alcançados por 2 e 3 processadores no SP2 utilizando MPI.....	141
Gráfico 7.27	- Tempo de execução do algoritmo de multiplicação de matrizes no SP2 utilizando PVM.....	142
Gráfico 7.28	- Speedup e eficiência alcançados por 2 e 3 processadores no SP2 utilizando PVMe.....	143
Gráfico 7.29	- Tempo de execução do algoritmo do trapézio composto no LASD-PC utilizando MPI. ....	144
Gráfico 7.30	- Speedup e eficiência alcançados por 2 e 3 processadores no LASD-PC utilizando MPI .....	145
Gráfico 7.31	- Tempo de execução do algoritmo de multiplicação de matrizes no LASD-PC utilizando PVMe .....	146
Gráfico 7.32	- Speedup e eficiência alcançados por 2 e 3 processadores no LASD-PC usando PVM .....	147
Gráfico 7.33	- Tempo de execução do algoritmo de multiplicação de matrizes no sistema Cray .....	148
Gráfico 7.34	- Speedup e eficiência encontrados no sistema Cray.....	151
Gráfico 7.35	- Comparação entre arquiteturas com 3 processadores executando o algoritmo de multiplicação de matrizes.....	152
Gráfico 7.36	- Comparação entre arquiteturas com 3 processadores executando o algoritmo de multiplicação de matrizes excluindo Cray.....	153
Gráfico 7.37	- Comparação entre arquiteturas com 2 processadores executando o algoritmo de multiplicação de matrizes .....	154
Gráfico 7.38	- Comparação entre arquiteturas com 2 processadores executando o algoritmo de multiplicação de matrizes excluindo Cray.....	154

## Resumo

O objetivo principal deste trabalho é o desenvolvimento e avaliação de algoritmos numéricos paralelos e sua execução em máquinas paralelas (máquinas multiprocessadas, máquinas vetoriais e máquinas paralelas virtuais). Os algoritmos desenvolvidos foram executados em diferentes condições tanto em termos de plataformas utilizadas como em termos de tamanho da aplicação considerada.

Os resultados obtidos na implementação dos algoritmos numéricos são analisados baseando-se em algumas métricas (tempo de execução e operações em ponto flutuante) comuns aos resultados apresentados nos principais *benchmarks* estudados. Através dos resultados obtidos, o desempenho das bibliotecas de passagem de mensagem MPI e PVM, o desempenho das arquiteturas consideradas e da implementação dos algoritmos numéricos são analisados.

# **Abstract**

The main objective of this dissertation is the development and evaluation of numerical parallel algorithms and their execution on parallel machines (multiprocessor machines, vectorial machines and parallel virtual environments).

The algorithms developed have been executed under different conditions both in terms of the hardware platform adopted and the problem size.

The results obtained with the numerical algorithms implementation are all analyzed according to some metrics (execution time and float-point operations) available in the main benchmarks studied.

The performance reached with the message passing libraries PVM and MPI together with the performance observed from the different architectures considered and the numerical algorithms implemented are all analyzed according to the result obtained in this work.

# Capítulo 1

## 1. *Introdução*

A grande evolução dos computadores nas últimas décadas, vem sendo acompanhada de perto pelo considerável aumento, tanto em número quanto em complexidade, das aplicações que utilizam estas máquinas. Desta forma, por mais que as arquiteturas tenham apresentado um processo evolutivo intenso, ainda é necessário buscar um melhor desempenho para a execução das aplicações que crescem tanto em volume como em complexidade muito mais rapidamente que o hardware disponível. Nesse contexto surge a computação paralela que tem como objetivo principal o aumento de desempenho na execução de uma aplicação. Nessa busca por melhor desempenho os sistemas paralelos apresentam-se como uma alternativa à máquina de *von Neumann*, sejam esses sistemas paralelos virtuais (sistemas distribuídos) ou em arquiteturas realmente paralelas (máquinas com múltiplos processadores).

Assim como em muitas outras áreas, não só da computação, a análise de desempenho vem se tornando imprescindível, principalmente quando o objetivo é atingir a máxima qualidade de um produto ou serviço, em particular, na área de computação. Devido ao grande crescimento dos sistemas paralelos, a necessidade de avaliação de desempenho tornou-se essencial, tanto para o software como para o hardware envolvido. Segundo Heidelberg e Lavemberg [Hei84] a

avaliação de desempenho se divide em três categorias: **Modelagem Analítica de Desempenho, Simulação e Medidas de Desempenho**. Sendo que, **Medidas de Desempenho** se divide em: **Benchmarks e Monitoração**.

A modelagem analítica de desempenho não envolve programação e sim análise matemática, probabilística e estatística. Este tipo de análise de desempenho nem sempre é viável, principalmente quando muitas variáveis estão envolvidas. A modelagem analítica pode ser utilizada com sucesso para verificar o desempenho de um dispositivo particular do computador, por exemplo, o sistema de disco, de E/S (Entrada/Saída), etc. A verificação do software através de modelagem analítica torna-se complicada por depender de um grande número de variáveis. Desta forma, esta abordagem tem utilização restrita.

A simulação utiliza programas que emulam o sistema em questão, executando as instruções que representam parcialmente ou totalmente um sistema real. Este tipo de análise é ideal para estimar o tempo de execução de programas, a influência de algumas características de uma arquitetura tais como utilização de memória e comunicação entre processadores entre outras.

As medidas de desempenho, podem ser efetuadas através de *benchmarks* ou de medidas efetuadas no programa a ser avaliado ou no hardware analisado. Os *Benchmarks* são programas que devem ser executados em sistemas reais para avaliar seu desempenho. O mesmo *benchmark* executado em diferentes sistemas, permite que os resultados e consequentemente os sistemas sejam comparados. Essas medidas podem ser a quantidade de soluções por segundo, número de operações em ponto flutuante por segundo e quantidade de comunicação entre outras.

O objetivo deste trabalho é verificar o desempenho de alguns algoritmos paralelos executando em diferentes plataformas. Esta avaliação é efetuada através de medições durante a execução dos algoritmos. Os valores obtidos na execução são utilizados para determinar-se diferentes métricas para avaliar o desempenho do algoritmo sendo executado em uma determinada plataforma.

Os resultados obtidos nestas medições serão comparados com resultados obtidos na execução de *benchmarks* disponíveis comercialmente para avaliação de arquiteturas paralelas.

Três plataformas básicas foram utilizadas para a avaliação dos algoritmos. Um computador multiprocessado - SP2, executando PVM e MPI. Um sistema distribuído baseado em

computadores tipo PC, rodando o sistema operacional *linux* e utilizando PVM e MPI. Uma máquina vetorial Cray com 3 processadores vetoriais.

Este trabalho está dividido em 8 capítulos. No Capítulo 2 são apresentadas as características básicas da computação paralela, tais como, conceitos, classificações e uma descrição, que permite criar programas paralelos de uma certa complexidade, das duas bibliotecas mais utilizadas em passagem de mensagens (PVM e MPI).

No Capítulo 3, são discutidas as principais características dos *benchmarks*, bem como os conceitos básicos, cuidados a serem seguidos, metodologia utilizada, formas de medir o desempenho, como devem ser construídos os *benchmarks* numéricos. Três *benchmarks* para arquiteturas paralelas (NAS, Genesis e *Parkbench*) e uma breve descrição de outros tipos de *benchmarks* são discutidos neste capítulo.

Os algoritmos numéricos que serão considerados neste trabalho são apresentados no Capítulo 4.

Os Capítulo 5 e 6 são responsáveis por descrever todo o ambiente utilizado (software e hardware) e como os algoritmos apresentados no Capítulo 4 foram paralelizados e implementados.

O Capítulo 7 mostra os resultados obtidos durante a execução dos algoritmos numéricos nas plataformas descritas no Capítulo 5.

Finalmente, o Capítulo 8 apresenta as conclusões, as dificuldades encontradas e as sugestões para trabalhos futuros.

# Capítulo 2

## 2. Computação Paralela

### 2.1 Introdução

Com o advento do VLSI (*Very Large System Integration*) em circuitos integrados, as regras para construção de computadores mudaram totalmente. Um resultado indiscutível foi a remoção da barreira criada pelo paradigma de *von Neumann* que impunha as CPUs um procedimento serial. Com essa barreira quebrada, os projetistas de computadores podem agora voltar-se a solucionar problemas utilizando arquiteturas paralelas, problemas estes que antes exigiriam longos períodos de computação se utilizada uma arquitetura serial.

Considerando os problemas a serem solucionados, origina-se a seguinte questão: qual a relação custo-benefício de se utilizar um sistema paralelo vetorial, como por exemplo uma máquina *Cray*, versus utilizar um sistema de passagem de mensagem num sistema distribuído ou em uma máquina multiprocessada? A resposta mais intuitiva seria a de utilizar-se a arquitetura de mais baixo custo, mas se for feita uma análise mais profunda, observa-se que a resposta é bem mais complicada do que se imagina. Outras questões devem entrar na análise, tais como: tempo exigido para conclusão de uma tarefa, periodicidade de resolução, se vários tipos de problemas

devem ser solucionados pela máquina, qual a sincronização exigida pelo problema, qual a linguagem ou biblioteca de comunicação mais adequada para implementar o algoritmo, e assim por diante. Um ponto que nunca deve ser esquecido, é se a instituição ou empresa tem recursos suficientes para implantar e manter a estrutura necessária funcionando [Vel94].

Neste capítulo serão discutidos alguns tópicos que devem ser considerados na utilização da computação paralela. A próxima seção introduz alguns conceitos básicos da área e conceitos referentes a computação paralela.

## 2.2 Conceitos Básicos

Devido ao surgimento relativamente recente da computação paralela, não existe uma terminologia unificada como a encontrada em algumas outras áreas da ciência da computação. Desta forma apresentam-se alguns conceitos que serão observados através deste trabalho.

### 2.2.1 Concorrência e Paralelismo

Estes dois conceitos são constantemente confundidos pelos usuários menos experientes por estarem intimamente ligados. O termo “paralelismo” é designado a múltiplos processadores executando processos ao mesmo tempo. Entende-se por processo como um programa qualquer em execução em um determinado instante de tempo.

O termo “concorrência” indica que mais de um processo está sendo executado, ou seja, vários processos já iniciaram e ainda não terminaram. O pseudo-paralelismo ocorre quando vários programas (processos) são executados em uma máquina que possui somente um processador. Neste caso, em um determinado instante de tempo somente um processo pode estar sendo executado. Cada um desses processos recebe uma porção de tempo para sua execução, esse determinado tempo pode ser denominado de *time-slice* ou *quantum*. Quando seu *quantum* termina, caso ele não tenha concluído sua execução, seu estado é salvo para uma posterior execução e um outro processo é executado. Desta forma o usuário tem a impressão de que todos os programas estão sendo executados ao mesmo tempo.

## 2.2.2 Granulação

A granulação representa o nível de paralelismo apresentado por uma aplicação, sendo que de maneira simplificada a granulação pode ser dividida em: grossa, média e fina. O conceito de granulação é largamente discutido em [Hwa84] [Nav89] [Kir91] [Alm94].

Pode-se dizer que quando os processadores tem grande carga de trabalho, dividida em poucos processos, e pouca comunicação entre eles a granulação é grossa. Geralmente, este tipo de granulação aplica-se a arquiteturas com poucos processadores.

A granulação fina por sua vez lida com paralelismo em nível de instruções ou operações e geralmente é utilizada com uma grande quantidade de processadores.

E por fim, pode-se dizer que a granulação média está situada entre as duas granulações anteriores. Geralmente é difícil distinguir onde aplicações de granulação média se situam dentre essas três classificações.

## 2.2.3 Speedup e Eficiência

O principal motivo da utilização da computação paralela é o aumento da velocidade de processamento o que consequentemente pode levar ao aumento da eficiência na resolução de um determinado problema.

*Speedup (Sp)* é uma medida utilizada para calcular o quanto um programa paralelo utilizando  $p$  processadores é mais rápido do que sua execução seqüencial. Onde,  $T_{seq}$  é o tempo de execução seqüencial e  $T_{par}$  representa o tempo de execução em paralelo [Hwa93][Alm94] como na equação a seguir:

$$Sp = T_{seq} / T_{par}$$

Pela fórmula pode-se observar que  $Sp = p$  representa a situação ideal em termos de ganho. Mas, essa situação dificilmente é alcançada pois outros fatores influenciam nos cálculos, tais como: sobrecarga na comunicação, balanceamento de carga inadequado, algoritmos com um nível pequeno de paralelismo e granulação inadequada. Em casos raros, pode ocorrer uma anomalia,

onde  $Sp > p$ , esse caso é chamado de *anomalia de speedup* ou *speedup super linear* [Fos95]. Isto normalmente ocorre devido a um grande aumento de memória, quando vários processadores são considerados.

Para quantificar a utilização do processador é usada uma métrica denominada eficiência ( $Ep$ ) e é calculada através do speedup ( $Sp$ ) e do número de processadores ( $p$ ):

$$Ep = Sp / p$$

A máxima eficiência do processador é dada por  $Ep = 1$ , que significa que sua capacidade foi totalmente aproveitada (100%). Os valores encontrados devem variar entre 0 (zero) e 1 (um) e dificilmente atinge o valor 1 (um). Quando ocorre a anomalia de *speedup* também ocorre uma exceção onde os valores de  $Ep$  serão maiores do que um.

## 2.2.4 Abordagem de programação

Existem duas abordagens de programação conhecidas como SPMD (*Simple Program Multiple Data* – Programa Único Múltiplos Dados) e MPMD (*Multiple Program Multiple Data* – Vários Programas Múltiplos Dados). O primeiro tipo utiliza um único programa, ou seja, o mesmo código executa em diferentes processadores utilizando diferentes dados. O segundo tipo utiliza diferentes programas usando diferentes dados.

Além das abordagens de programação, existem alguns paradigmas que podem ser classificados em: *Processor Farm* também conhecido como Mestre-Escravo, *Pipeline*, e N-dimensional.

No paradigma Mestre-Escravo é designado um processador, chamado de mestre, para distribuir o trabalho entre os demais processadores escravos. É apropriado para aplicações com pouca comunicação e no final de cada tarefa os escravos enviam os resultados do seu processamento ao mestre. Esta abordagem é de fácil implementação, uma vez que não depende totalmente da arquitetura. [Meg97] A programação SPMD torna-se mais simples devido a capacidade das linguagens de iniciar vários processos iguais de uma só vez.

No modo *Pipeline* uma tarefa é dividida em várias tarefas menores com um certo grau de dependência entre elas, ou seja, a segunda tarefa depende do resultado da primeira, a terceira

depende do resultado da segunda e assim por diante sendo que as mesmas devem ser uma diferente da outra.

O paralelismo *N-Dimensional* caracteriza-se pela forma de como os processos trocam informações entre si. Por exemplo, o paralelismo de uma dimensão comunica-se com os processos vizinhos que seguem o mesmo caminho, já o de duas dimensões permite duas linhas de comunicação entre os dados, o de três dimensões possuem três linhas de comunicação formando um cubo e assim por diante [Des87]. A Figura 2.1 mostra um exemplo dos três tipos descritos.

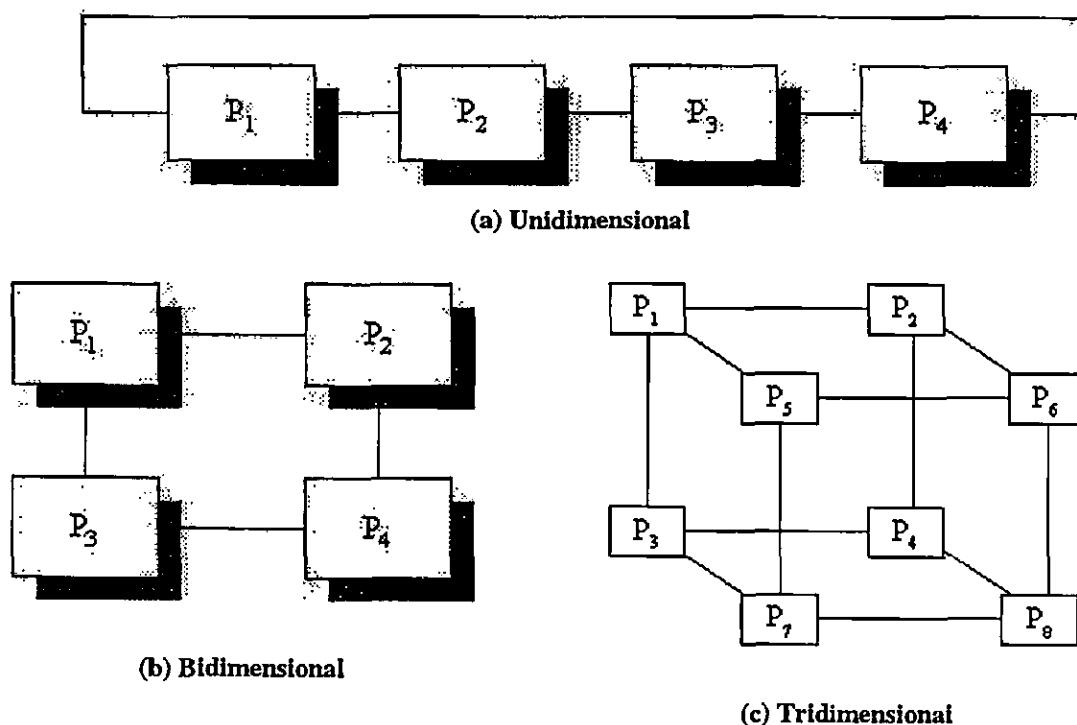


Figura 2.1 - Exemplos de programação *N-Dimensional*.

## 2.4 Arquiteturas Paralelas

Um grande número de arquiteturas paralelas vem sendo propostas ao longo dos últimos anos . Para organizar a apresentação destas arquiteturas, são utilizadas algumas classificações, a de Flynn e de Duncan foram adotadas por abrangerem a grande maioria das arquiteturas disponíveis.

### 2.4.1 Classificação de Flynn

A primeira tentativa de classificação das arquiteturas paralelas foi a classificação de Flynn que a dividia em quatro categorias [Hwa93]:

#### SISD - Única Instrução e Único Dado.

É o computador de *von Neumann*, ou seja, os computadores puramente seqüenciais. Executa as instruções seqüencialmente a cada operação atua sobre um único elemento de dados. Os computadores seqüenciais estão enquadrados nesta categoria, esta arquitetura é apresentada na Figura 2.2.

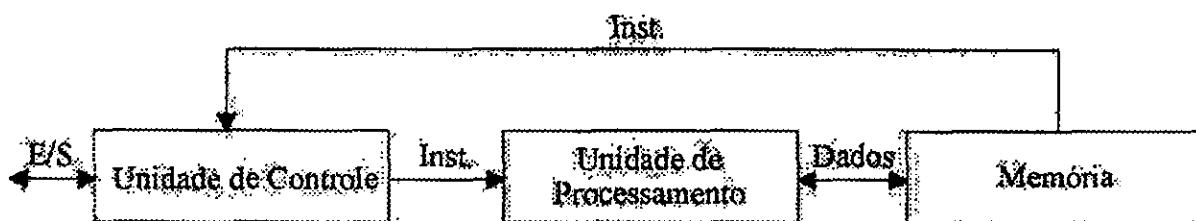


Figura 2.2 - Arquitetura SISD

#### SIMD - Única Instruções e Múltiplos Dados

Este tipo de arquitetura é síncrona e determinística, conveniente para paralelismo em nível de instrução/operação. Nesta categoria estão os computadores vetoriais como o IBM 9000 e o *Cray YMP* [Hwa94]. A figura 2.3 mostra a estrutura deste tipo de arquitetura.

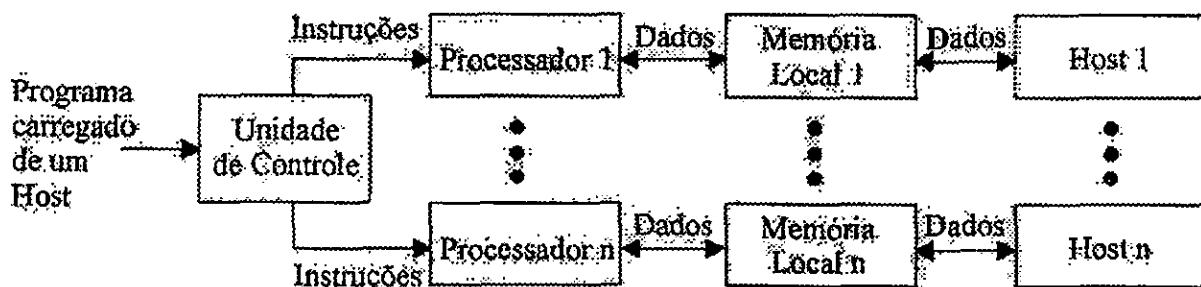


Figura 2.3 - Arquitetura SIMD

#### MIMD - Múltiplas Instruções e Múltiplos Dados

Este tipo de arquitetura é assíncrona e conveniente para paralelismo de blocos, "loops" e subrotinas [Hwa93]. O código pode ser igual ou diferente para todos os processadores (SPMD ou

MPMD). Seus principais representantes são: IBM ES/9000-900 e o IBM-SP2. A Figura 2.4 mostra a estrutura de uma arquitetura MIMD.

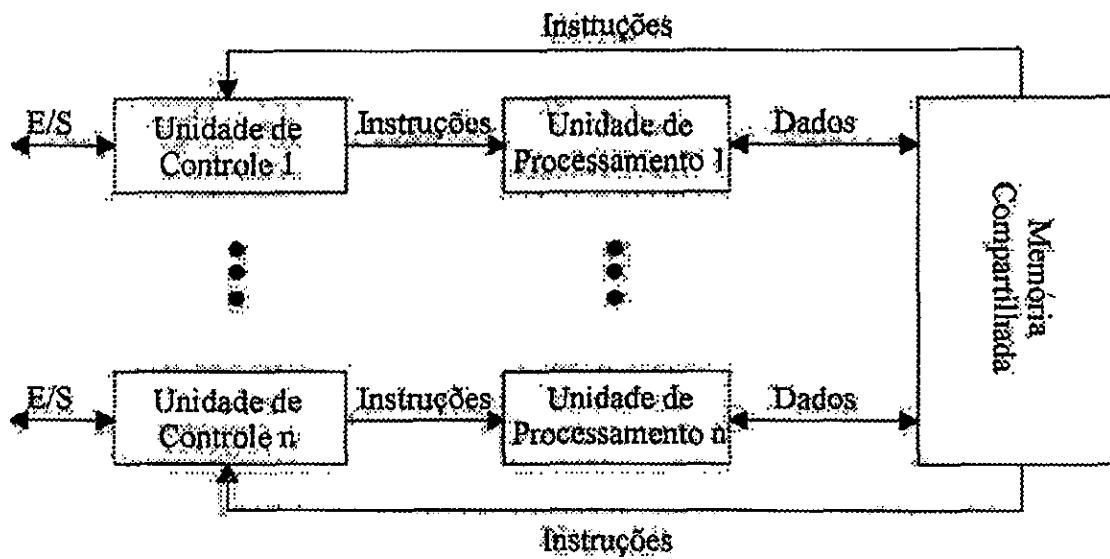


Figura 2.4 - Arquitetura MIMD

### MISD - Múltiplas Instruções e Dados Simples

Apesar de nesta categoria não existirem exemplos práticos a Figura 2.5 tenta demonstrar como seria esse sistema.

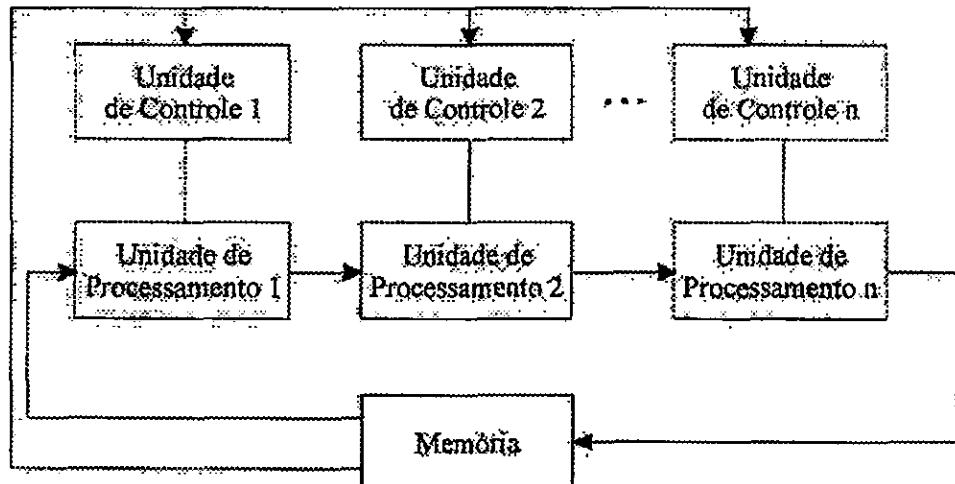


Figura 2.5 -Tentativa de caracterização da arquitetura MISD.

Analizando-se a classificação de Flynn observa-se a existência de apenas dois tipos de arquiteturas paralelas: SIMD e MIMD, o que não é suficiente para abranger a grande quantidade de arquiteturas existentes. A classificação de Duncan, apresentada resumidamente na próxima seção, expande estes tipos de arquiteturas, ela é amplamente discutida em [Dun90].

## 2.4.2 Classificação da Duncan

Com a crescente evolução das arquiteturas paralelas a classificação de Flynn tornou-se obsoleta e pouco genérica. A seguir na Figura 2.6 apresenta-se a classificação de Duncan que aproveitou os conceitos de Flynn e generalizou sua classificação [Dun90]. Basicamente elas são divididas em dois tipos (síncronas e assíncronas) e a partir destes as demais arquiteturas são detalhadas

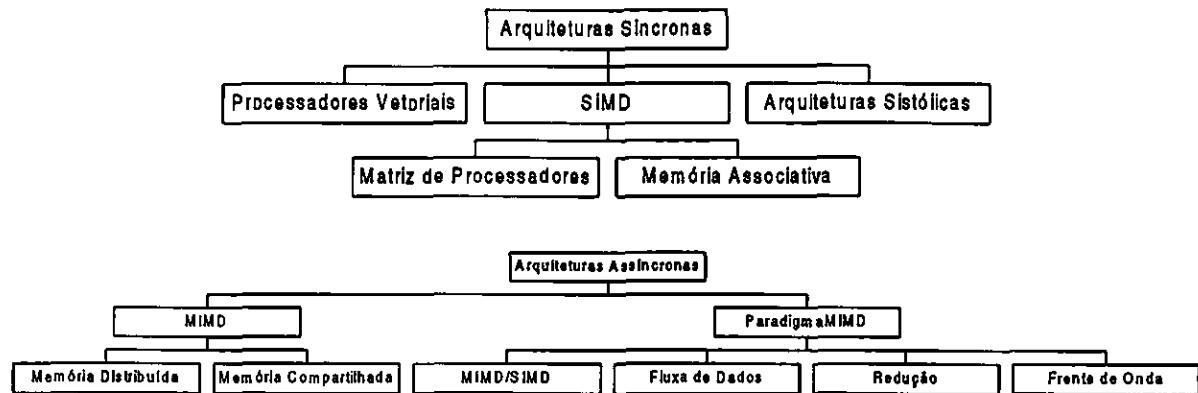


Figura 2.6 - Classificação de Duncan

As Arquiteturas Síncronas são baseadas em relógios globais, uma unidade central de controle e quando cabível, em unidades vetoriais de controle. Podem ser subdivididas em:

- Processadores vetoriais – possuem um hardware especializado em operações vetoriais e *pipelines* múltiplos os quais podem trabalhar com operações aritméticas e lógicas em vetores e escalares.
- SIMD – utiliza uma unidade central de controle para diversos processadores, onde a mesma efetua um *broadcast* de uma instrução para todas as unidades de processamento. As conexões permitem que os resultados dos cálculos sejam trocados entre processadores para que estes sirvam como operadores em novos cálculos. As máquinas SIMD podem ser divididas em duas categorias:
  - Matriz de processadores – utilizada em larga escala em aplicações científicas, processamento de imagens e modelagem de energia nuclear. Os processadores são conectados em forma de matrizes.

- Memória associativa – é uma arquitetura basicamente utilizada em bancos de dados orientados a objetos, permitindo o acesso simultâneo a memória através de padrões.
- Sistólica – surgiu aproximadamente em 1980 para solucionar problemas de propósito específico que exigem uma grande quantidade de Entrada e Saída (E/S), usa basicamente processadores *pipelined* e possui um relógio global.

Seguindo a Figura 2.6 encontram-se as arquiteturas assíncronas que são caracterizadas por possuírem relógios e unidades de controles próprios para cada processador e são classificadas como a seguir:

- Arquiteturas MIMD utilizam vários processadores, ativados por unidades de controle próprias, que executam instruções independentes. Apesar dessa independência os processadores podem ser sincronizados através de passagens de mensagens. Quanto a organização da memória, as arquiteturas MIMD podem ser divididas em dois tipos:

- Memória Distribuída – são processadores com memória independente, ou seja, cada processador tem seu espaço de endereçamento e podem comunicar-se por troca de mensagens. A conexão entre processadores pode dar-se através de diferentes formas, tais como: anel, árvore, hypercubo, malha, árvores mapeadas, etc. A desvantagem desta arquitetura é que uma mensagem pode ser perdida, no entanto, para solucionar esse problema as bibliotecas de passagem de mensagem geralmente tem funções que detectam esse tipo de inconveniente.
- Memória Compartilhada – na memória compartilhada, todos os processadores compartilham o mesmo espaço de memória. Neste caso, não existem os problemas da memória distribuída (perda de mensagens, eliminando a necessidade de novas trocas), mas outros problemas podem surgir tais como: a sincronização de acesso e coerência de cache. Os processadores e a memória podem ser conectados de diferentes formas, tais como: barramento, *crossbar* e *omega*. Neste caso a conexão entre memória e processador é a região mais crítica e deve ser cuidadosamente implementada com um meio veloz de comunicação.

**Paradigma MIMD** – Cada uma das arquiteturas é baseada nos princípios de operações assíncronas e manipulação concorrente de múltiplas instruções da arquitetura MIMD. Alguns tipos de MIMD não convencional são:

- Híbrida - MIMD/SIMD – nesta arquitetura alguns processadores MIMD são designados como mestres que controlam outros processadores escravos que se utilizam da arquitetura SIMD, isso dá a aparência de uma árvore a arquitetura.
- Fluxo de Dados – neste paradigma diz-se que as instruções podem ser executadas tão logo seus operadores estejam disponíveis. Isto implica numa dependência de dados que permite uma concorrência de tarefas, rotinas e instruções. [Dun90]
- Redução – neste paradigma uma instrução é executada somente quando uma segunda instrução que depende da primeira já está pronta para execução apenas dependendo dos resultados da primeira. Este paradigma é ideal para expressões grandes e complexas.
- Frente de Onda – esta arquitetura é uma mistura da arquitetura sistólica com o paradigma Fluxo de Dados.

Na próxima seção serão apresentados alguns tópicos concernentes a programação de arquiteturas paralelas de modo geral.

## 2.5 Computação Vetorial

Um computador vetorial é uma máquina que tem uma grande capacidade para trabalhar com vetores, ou seja, o processamento de elementos de um vetor é feito em grupos. A diferença entre uma entidade escalar e uma vetorial, é que a escalar é uma entidade de 16, 32, ou 64 bits que pode ser um número em ponto flutuante, um número inteiro ou um valor lógico (verdadeiro ou falso). Essa entidade escalar é processada através de registradores e *pipelines* escalares. Uma entidade vetorial é uma coleção de entidades escalares, por exemplo: uma coleção de entidade de 64 bits para números em ponto flutuante ou uma coleção de entidades de 128 bits para números complexos. As entidades vetoriais são processadas através de registradores e *pipelines* vetoriais. A Figura 2.7, mostra como trabalha uma máquina vetorial.

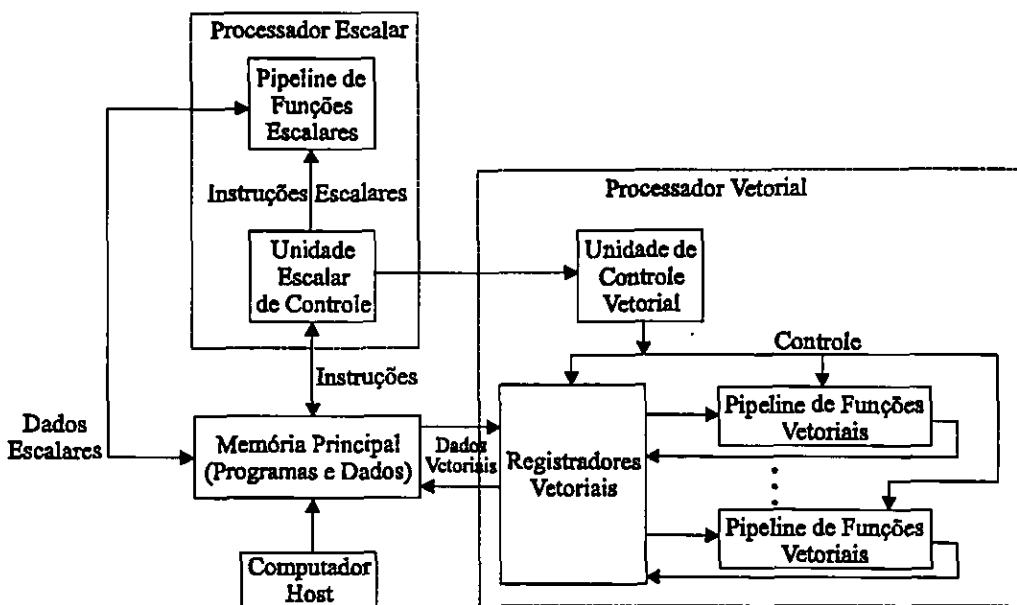


Figura 2.7 - Esquema simplificado de um computador vetorial.

Através de um *host* (computador remoto) compila-se um programa, em seguida e necessário submete-lo a uma fila de execução através de um *script* (arquivo de comandos) quando houverem processadores disponíveis o programa é enviado para a memória principal. Se a instrução a ser executada for escalar, ela será enviada ao processador escalar, se a instrução for vetorial ela será direcionada ao processador vetorial.

O pequeno código a seguir e a Tabela 2.1 mostram a diferença entre como um processador escalar e um processador vetorial tratam os dados.

```
for (i = 1 ; i < 4 ; i++) {
    l[i] = j[i] + k[i];
    n[i] = l[i] + m[i];
}
```

Tabela 2.1 - Ordem das operações no modo escalar e no modo vetorial.

Modo escalar	Modo vetorial
$l[1] = j[1] + k[1]$	$l[1] = j[1] + k[1]$
$n[1] = l[1] + m[1]$	$l[2] = j[2] + k[2]$
$l[2] = j[2] + k[2]$	$l[3] = j[3] + k[3]$
$n[2] = l[2] + m[2]$	$n[1] = l[1] + m[1]$
$l[3] = j[3] + k[3]$	$n[2] = l[2] + m[2]$
$n[3] = l[3] + m[3]$	$n[3] = l[3] + m[3]$

Note que no processador escalar os dados são tratados item por item. No caso vetorial cada vetor é tratado de uma vez só. Note também que se a máquina tiver mais de um processador e no código a variável  $n$  não dependesse de  $l$  todas as instruções seriam tratadas de uma só vez.

### Condições que impedem a vetorização

A seguir são relacionadas as principais circunstâncias que impedem a vetorização:

- Dependência de dados – se existir uma dependência de dados, ou seja, se o valor que está sendo gerado depende do valor anterior ou posterior como em  $a[i] = b[i-1] + j$  e  $a[i] = b[i+1] + j$ , respectivamente;
- Se existirem chamadas a funções do usuário dentro do laço;
- Comandos de entrada e saída;
- Dependência de dados como em  $a[i] = b[i+1] + c$  ou  $a[i] = b[i-1] + c$ ;
- Comandos de quebra de laço, tais como: *break* e *exit*;
- Teste de condições: *if*, *switch*;
- Manipulação de bits;
- Diretivas de compilação.

Em programas científicos as condições acima são pouco encontradas, com exceção das chamadas as funções de usuários e da dependência de dados. Apesar disso, outras regras devem ser observadas. Nos sistemas *Cray*, na existência de laços aninhados, somente o mais interno é vetorizado. Isso é válido também para os comandos *while* e *do...while*. A seguir é dada uma breve descrição do jargão utilizado na computação vetorial com alguns exemplos na Figura 2.8.

- Laço invariável – é uma variável constante que é referenciada no corpo do laço mas não é alterada por ele.
- Laço variável induzida – é uma variável que é incrementada ou decrementada por uma expressão invariável.

- Vetor candidato – é um elemento referenciado por um índice que pode variar no corpo do laço.
- Vetor candidato temporário – é uma variável escalar que é referenciada em cada passo do laço.
- Expressão vetorial - é uma expressão lógica ou aritmética que consiste na combinação de qualquer dos itens acima
- Laço vetorial - é um laço que contém apenas expressões vetoriais.

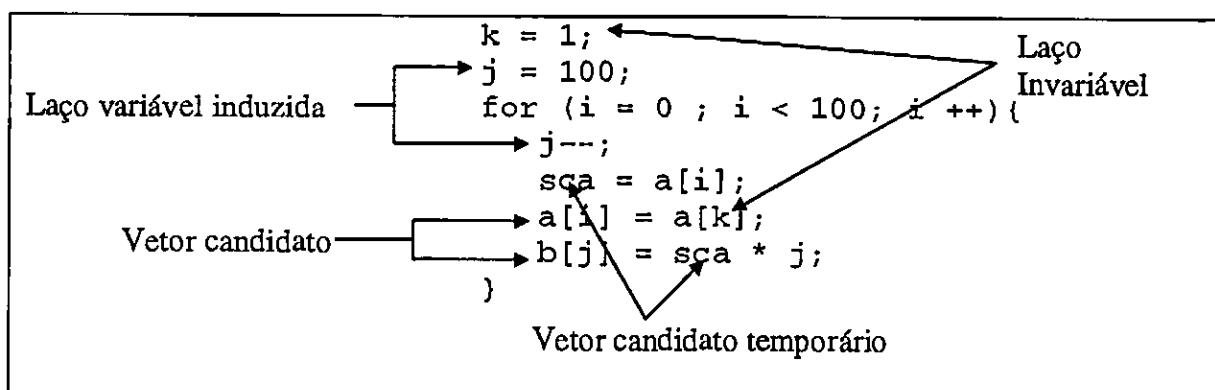


Figura 2.8 - Exemplos de termos utilizados na computação vetorial.

Dadas as definições utilizadas na vetorização, a seguir serão mostrados os diferentes tipos de vetorização.

**Vetorização total** - utiliza somente expressões vetoriais dentro do laço. O código gerado é extremamente eficiente.

```

for (i = 0 ; i < 101 ; i++){
    c[i] = a[i] + b[i] / 2.0;
    m3[i] = m1[i] - m2[i];
    trig[i] = cos(c[i]);
}

```

**Vetorização especial** - é um laço que contém variáveis escalares e vetoriais, onde há uma transformação de elementos vetoriais para escalares.

```
for (i = 0; i < 101; i++) sum += a[i];
```

**Pesquisa de laços** - são laços que contém comandos que podem transferir o controle para fora do mesmo. Este tipo de código não é vetorizável.

```
for (i = 0 ; i < n ; i++) {
    if (a[i] < b[i]) break;
    c[i] = d[i];
}
```

## 2.6 Programação Concorrente

Atualmente a Programação Concorrente não tem uma terminologia padronizada, desta forma considera-se que a programação concorrente é dividida em duas categorias: programação concorrente com memória compartilhada e com passagem de mensagens (comandos *Send* e *Receive*).

Quando a memória compartilhada é usada, ou melhor dizendo, quando são usadas variáveis compartilhadas, dois tipos de sincronização são necessários: exclusão mútua e condição de sincronização (também chamado de condição de concorrência). Para solucionar esses problemas existem diversas técnicas, tais como: *busy-waiting*, semáforos e monitores [Mis89].

Usando a passagem de mensagens, tem-se os comandos *Send* e *Receive* que enviam e recebem mensagens respectivamente. Dependendo da organização destes comandos pode-se ter comunicação ponto-a-ponto, *rendezvous* ou RPC [Mis89].

A programação concorrente vem sendo de fundamental importância para todos os tipos de programação, incluindo sistemas de gerenciamento de dados, computação científica, sistemas de tempo real e sistemas de controle [Geh89], sendo largamente utilizada para a computação numérica.

A vinte anos atrás os programas eram desenvolvidos para utilizar a máxima capacidade de uma determinada arquitetura. Atualmente, além desta abordagem, pode-se desenvolver programas para explorar ao máximo o paralelismo existente no problema a ser solucionado. Para programas paralelos é necessário a utilização de uma ferramenta que possibilite representar os processos paralelos, a comunicação e o sincronismo entre eles [Tan95]. Os ambientes de passagem de mensagens são largamente utilizados porque suprem com relativa facilidade essas necessidades.

Uma grande vantagem dos ambientes de troca de mensagens é que são compostas por bibliotecas que podem ser implementadas em diversas arquiteturas, tornando os programas portáveis entre essas arquiteturas. Exemplos de ambiente de passagem de mensagem são: MPI e PVM.

Nos dias atuais tem-se tornado viável a utilização da computação paralela, principalmente devido ao surgimento de bibliotecas que permitem simular uma máquina virtual paralela em ambientes distribuídos. Já não é mais necessário investir milhares de dólares em máquinas, basta ter uma pequena rede de estações de trabalho (até mesmo de PC's) para desfrutar das vantagens da programação paralela.

Na programação paralela, um dos primeiros objetivos é desenvolver programas tão independente quanto possível das arquiteturas preocupando-se apenas com a eficácia e com a eficiência. Alguns problemas, exigem uma preocupação com a arquitetura devido à sua natureza principalmente onde questões como *desempenho* são críticas (sistemas de tempo real). Como em alguns algoritmos numéricos a convergência de uma solução em máquinas monoprocessadas é muito lenta, vê-se claramente a aplicação da computação paralela nesses casos.

A seguir são apresentados os ambientes de passagem de mensagem mais conhecidos atualmente e que serão utilizados no desenvolvimento deste trabalho.

### **2.5.1 PVM (*Parallel Virtual Machine* – Máquina Paralela Virtual)**

O PVM é uma biblioteca que oferece ferramentas de software necessárias para programar um sistema computacional paralelo de propósitos gerais [Beg94]. Inicialmente foi escrito em Fortran, mas, já possui versões em C e C++. Seu projeto teve início em 1989 no *Oak Ridge National Laboratory* – ORNL nos EUA, atualmente envolvendo diversos institutos de pesquisa. Sua primeira versão (PVM 1.0) foi implementada por Vaidy Sunderman e Al Geist sendo que atualmente o PVM encontra-se na versão 3.0 (1993).

#### **Utilizando o PVM**

O PVM é formado por duas partes básicas. A primeira é um *daemon* que deve residir em cada máquina, ou em cada processador, no caso de um máquina multiprocessada, para formar o ambiente paralelo virtual. A segunda parte é composta pela biblioteca de interface PVM, que

contém todas as primitivas necessárias para a montagem das tarefas que compõe a aplicação. As funções ou primitivas existentes permitem realizar as seguintes operações:

- Iniciar, sincronizar e terminar processos;
- Empacotar, enviar, receber e desempacotar mensagens;
- Alterar o número de cada nó, criar, manipular e apagar grupos de processos.

A menor unidade em uma aplicação PVM chama-se tarefa e os programas em PVM podem utilizar tanto a abordagem de programação SPMD ou MPMD. Segundo [Gei94], as principais características do PVM são:

- *Fila configurável de máquinas* - As tarefas de uma aplicação são executadas por um conjunto de máquinas que podem ser selecionadas pelo usuário.
- *Acesso transparente ao hardware* - Os programas PVM podem ter acesso ao hardware como se ele fosse um conjunto de processadores dentro de uma mesma máquina, ou, podem explorar ao máximo as capacidades de um determinado equipamento realmente multiprocessado.
- *Tolerância a falhas* - Se uma das máquinas do sistema paralelo virtual falhar, ela é imediatamente retirada da fila pelo sistema. Mas, cabe ao usuário realocar o processo em outra máquina e executar novamente a aplicação.
- *Suporte a máquinas heterogêneas* - O PVM permite a passagem de mensagem entre máquinas de diferentes arquiteturas com diferentes tipos de representação de dados.

Antes de rodar qualquer programa PVM é necessário assegurar-se de que a máquina paralela virtual está configurada corretamente. No caso do SP2, isso não é necessário porque a alocação de tarefas é feita automaticamente pelo sistema.

### Principais Primitivas

O primeiro comando PVM que geralmente é utilizado é o *pvm\_mytid()*, que devolve um número que identifica a tarefa em execução. Todas as tarefas são iniciadas através de uma tarefa mestre, a função que se encarrega de iniciar os processos é a *pvm\_spawn* e tem o seguinte formato:

```
int pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)
```

Sendo que os parâmetros têm os seguintes significados:

- *char \*task* – nome da tarefa a ser executada;
- *char \*\*argv* – ponteiro para argumentos de tarefas, geralmente utilizado com o valor NULL.
- *int flag* – opções de distribuição seguindo a Tabela 2.2.
- *char \*where* – nome da máquina onde será executada a tarefa, *null* no caso do SP2.
- *int ntask* -representa a quantidade de escravos que devem ser iniciados;
- *int \*tids* – vetor que conterá a identificação das tarefas iniciadas.

Tabela 2.2 – Resumo dos *flags* utilizados no *pvm\_spawn*.

Valor	Opção	Significado
0	PvmTaskDefault	O PVM escolhe onde iniciar os processos.
1	PvmTaskHost	O usuário escolhe onde os processos serão iniciados.
2	PvmTaskArch	Os processos serão iniciados através de um arquivo.
4	PvmTaskDebug	Inicia os processos num depurador.
8	PvmTaskTrace	Cria um arquivo de rastreamento.
16	PvmMppFront	Inicia as tarefas a partir de uma máquina <i>front-end</i> .
32	PvmHostCompl	Dados complementares sobre onde será iniciada uma tarefa.

Para terminar uma tarefa no PVM utiliza-se a função *pvm\_kill()*. Outros comandos importantes são:

- *pvm\_parent* – retorna a identificação de quem criou a tarefa.
- *pvm\_tidtohost* – retorna um vetor com a identificação de todas as tarefas que estão sendo executadas numa máquina.
- *pvm\_addhost* – permite adicionar máquinas a Máquina Paralela Virtual.

Os comandos mais utilizados no PVM são os responsáveis por enviar e receber mensagens. Para enviar uma mensagem é necessário seguir alguns passos. O primeiro é inicializar o *buffer* de envio de mensagens através do comando *pvm\_initsend(codificação)*. Onde *codificação*, refere-se a como os dados serão codificados para o envio, sendo de três maneiras diferentes:

- *PvmDataDefault* – utiliza o formato padrão do PVM que é o XDR.

- *PvmDataRaw* – não utiliza nenhuma forma de codificação e é utilizado onde a máquina paralela virtual é homogênea, ou seja, todas as máquinas são do mesmo tipo.
- *PvmDatainPlace* – simplesmente mostra onde os dados a serem empacotados estão, lá só estarão contidas as informações de tamanho e ponteiros, quando o procedimento de envio se iniciar os dados serão buscados diretamente na memória, isto é, não é utilizado um *buffer* para armazenar temporariamente os dados que se desejam enviar.

Depois de inicializado o *buffer* de envio é necessário empacotar os dados que se desejam enviar. Para cada tipo de dados existe uma função no PVM, conforme mostra a Tabela 2.3, mas todas elas apresentam os mesmos argumentos que são: *pvm\_pktipo(dado, qitens, stride)*

- *dado* – endereço onde se encontra o dado a empacotar;
- *qitens* – quantidade de dados que se deseja enviar;
- *stride* – tempo de acesso a memória, geralmente é utilizado o valor 1.

Tabela 2.3 – Funções de empacotamento de dados.

Função	Tipo de dado empacotado em C
<i>pvm_pkint()</i>	<i>int</i>
<i>pvm_pkshort()</i>	<i>unsigned</i>
<i>pvm_pklong()</i>	<i>long int</i>
<i>pvm_pkfloat()</i>	<i>float</i>
<i>pvm_pkdouble()</i>	<i>double</i>
<i>pvm_pkcplx()</i>	<i>complex</i>
<i>pvm_pkdcplx()</i>	<i>double complex</i>
<i>pvm_pkbyte()</i>	<i>byte</i>

Umas das restrições do PVM é que ele não pode empacotar dados do tipo estrutura. Cada dado deve ser empacotado individualmente em *buffers* auxiliares e a estrutura deve ser remontada na tarefa que recebeu a mensagem. Quando todo esse processo estiver completo o PVM já está pronto para enviar os dados. Nota-se que todos os dados empacotados são transmitidos de uma só vez. O comando que envia uma mensagem a uma determinada tarefa é o *pvm\_send( )* e apresenta o seguinte formato:

*int info = pvm\_send (tid, tag)*

onde *info* é um inteiro que identifica um erro caso ele ocorra, *tid* é a identificação da tarefa que irá receber os dados e *tag* é uma etiqueta que identifica a mensagem enviada.

O processo de recebimento é feito de modo inverso ao processo de envio, só que não é necessário inicializar um *buffer* de recebimento. O comando que recebe uma mensagem é:

$$\text{int } bufid = pvm\_recv(tid, tag)$$

onde *bufid* é um inteiro que identifica um erro caso ele ocorra, *tid* é o identificador da tarefa que enviou e *tag* é uma etiqueta que identifica a mensagem. Depois de recebida a mensagem o primeiro passo a ser seguido é desempacotá-la. A sintaxe do comando de desempacotamento é semelhante a de empacotamento, e é definida como:

$$\text{int } info = pvm\_upktipo(buffer, qitem, stride)$$

onde *info* recebe o código de retorno de erro, *buffer* é o endereço de memória que receberá o dado desempacotado, *qitem* é a quantidade de itens a desempacotar e *stride* é o tempo de acesso a memória.

Com a teoria apresentada nesta seção é possível desenvolver programas paralelos com um certo nível de complexidade . A próxima seção descreve outra das mais importantes bibliotecas de passagem de mensagem.

## 2.5.2 MPI (*Message Passing Interface* – Interface de Passagem de Mensagens)

De maneira geral um algoritmo seqüencial pode ser portado para qualquer arquitetura desde que está tenha o compilador adequado e o algoritmo não utilize funções especiais de hardware de uma determinada arquitetura. Atualmente uma das maiores exigências dos usuários é o alto grau portabilidade que um programa/algoritmo deve possuir.

O mesmo deve ocorrer com programas que utilizam o sistema de passagem de mensagens. O MPI tenta suprir essa necessidade estendendo linguagens como o FORTRAN, C e C++, para que estas tenham a capacidade de executar programas paralelos.

A idéia do MPI surgiu por volta de 1990, quando 60 pessoas de diferentes organizações (usuários, vendedores e construtores de arquiteturas e programas paralelos da Europa e do Estados Unidos) formaram o que ficou conhecido como “Fórum MPI”. Os debates ocorriam entre pessoas experientes, o que incluía especialistas em PVM, PARMAKS e EPCC, entre outros

[Mac94]. Os documentos gerados desses debates e discussões deu origem a uma nova proposta a qual originou a primeira versão do MPI. Na verdade, o processo de desenvolvimento do MPI iniciou-se em abril de 1992, sendo que em novembro do mesmo ano a primeira versão foi apresentada [Wal94].

A questão da portabilidade em sistemas de passagens de mensagens é recente. No entanto, diversas plataformas de portabilidade foram propostas (PVM, PARMACS, etc). O MPI foi a primeira tentativa de se construir um sistema padrão, sendo que seus objetivos e seu escopo podem ser resumidos em: [Mac94]

- Prover alta portabilidade de código – atendendo a reivindicação dos usuários por portabilidade;
- Permitir a implementação eficiente em um grande número de arquiteturas – ou seja, permitir que um programa seja desenvolvido numa arquitetura de *cluster* (rede de estações de trabalho ou PC's) e depois portada para uma máquina paralela especial de alto desempenho;
- Suporte a arquiteturas paralelas heterogêneas – permitir que programas sejam executados em redes, independentemente das arquiteturas que se encontram nela;
- Facilidade de uso – permitir um aprendizado rápido da linguagem, sem envolver altos custos.

Apesar das facilidades que o MPI apresenta, alguns recursos foram deixados de lado, tais como:

- Iniciar processos dentro de outros processos;
- Iniciar processos durante a execução do programa;
- Funções de depuração;
- Entrada e Saída paralela.

Novas versões do MPI têm surgido e essas facilidades, inicialmente ausentes, vêm sendo introduzidas.

## Passagem de mensagem utilizando o MPI

Para utilizar o sistema de passagem de mensagem em MPI, a primeira rotina que deve ser chamada num programa é *MPI\_Init* (*int \*argc, char \*\*argv*), que definirá quantos processos serão criados. No caso do SP2 não são necessários argumentos pois a quantidade de processos é definida no arquivo de comandos de tarefas que será detalhado no Capítulo 5. Essa rotina também define o que se chama de *communicator*, quando não especificado o padrão é chamado de *MPI\_COMM\_WORLD*. Os processos só podem se comunicar se compartilharem um mesmo *communicator*, ou seja, a programação obrigatoriamente tem que se basear em grupos.

Para terminar um programa MPI são definidas duas funções básicas: a *MPI\_Finalize()*, que geralmente é utilizada no final do programa, caso hajam processos ainda em execução esta função não surte efeito. E a *MPI\_Abort(communicator, errcode)*, que termina abruptamente um programa caso ocorra o erro definido em *errcode*.

Como o MPI trabalha com a abordagem SPMD de programação, todos os processadores executam o mesmo código, dessa forma, para ter as informações sobre cada processo existem as rotinas *MPI\_Comm\_size* e *MPI\_Comm\_rank*. A primeira retorna a quantidade de processos que compartilham um mesmo *communicator*. A segunda retorna para cada processo a sua identificação que é um número inteiro.

As principais rotinas de envio e recepção de mensagens são:

*MPI\_Send(buf, count, datatype, dest, tag, comm)*

onde:

*buf* – é o endereço onde se encontra o dado a ser enviado;

*count* – é a quantidade de dados a serem enviados;

*datatype* – é o tipo de dado a ser enviado. A Tabela 2.5 mostra mais detalhes sobre como são definidos os tipos em MPI.

*dest* – é um número que identifica o processo destino.

*tag* – é como uma etiqueta que identifica a mensagem que está sendo enviada;

*comm* – é o *communicator* que os processos compartilham;

***MPI\_Recv(buf, count, datatype, source, tag, comm, status)***

onde:

*buf* – é o endereço onde o dado recebido deve ser armazenado;

*count* - é a quantidade de dados que será armazenada;

*datatype* – é o tipo de dado a ser armazenado, que também segue a Tabela 2.4;

*source* – é o número que identifica quem está mandando a mensagem;

*tag* – é a etiqueta que identifica a informação que foi enviada, ela deve ser a mesma especificada no comando de envio.

*comm* – é o *communicator* que os processos compartilham;

*status* – armazena o código de erro se houver algum na recepção.

**Tabela 2.4 – Tipos de dados no MPI.**

Tipo do dado	Representação em C
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	Int
MPI_LONG	long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned
MPI_UNSIGNED_LONG	unsigned long
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_LONG_DOUBLE	long double
MPI_BYTE	8 dígitos binários
MPI_PACKED	structure, vetores e matrizes

## 2.6 Considerações finais

Com a constante queda dos preços, os computadores vêm se tornado cada vez mais acessíveis, seja a empresas ou a pessoas físicas. Atualmente a construção de computadores com múltiplos

processadores já não é um desejo impossível. Com as novas ferramentas disponíveis no mercado, montar uma pequena rede de baixo custo que consiga simular uma máquina paralela virtual torna-se acessível a um número considerável de pessoas, ou seja, a computação paralela tornou-se economicamente viável.

Muitas áreas da computação vem necessitando cada vez mais de poder computacional, sendo que a matemática computacional também está incluída. A computação paralela veio exatamente tentar suprir essa necessidade. Principalmente no que diz respeito aos problemas que possuem um paralelismo natural.

Atualmente existem três tipos de ferramentas que dão suporte a programação paralela: ambientes de paralelização automática (computadores vetoriais), extensões paralelas para linguagens seqüenciais (PVM e MPI) e linguagens concorrentes (Ada) [Alm94].

Neste trabalho serão utilizadas as extensões paralelas para linguagens seqüenciais MPI e PVM pois elas são caracterizadas pela simplicidade, eficiência e são independentes de máquina. As características principais dessas linguagens são resumidas na Tabela 2.5.

Tabela 2.5 – Resumo das características das extensões paralelas de linguagens séries.

Característica	PVM	MPI
Manipulação de Mensagens	<ul style="list-style-type: none"><li>- Empacota dados;</li><li>- Não permite a passagem de mensagens formadas por estruturas e vetores não contíguos.</li></ul>	<ul style="list-style-type: none"><li>- Não empacota dados;</li><li>- Permite a passagem de estruturas complexas.</li></ul>
Portabilidade	<ul style="list-style-type: none"><li>- Portável sem nenhuma modificação de código dependendo da arquitetura.</li></ul>	<ul style="list-style-type: none"><li>- Portável sem nenhuma modificação de código dependendo da arquitetura.</li></ul>
Máquina Virtual	<ul style="list-style-type: none"><li>- Permite uma rede heterogênea;</li><li>- Interrompe processos em tempo de execução;</li><li>- Manipulação individual de processos ou em grupos realizada manualmente.</li></ul>	<ul style="list-style-type: none"><li>- Não possui o conceito de máquinas virtual;</li><li>- Não interrompe processos em tempo de execução;</li><li>- Possui um bom nível de abstração em relação a topologias trabalhando apenas com grupos.</li></ul>
Tolerância a Falhas	<ul style="list-style-type: none"><li>- Esquema básico de notificação de falhas;</li><li>- Não interrompe a aplicação. É possível que outras máquinas ainda recebam mensagens.</li></ul>	<ul style="list-style-type: none"><li>- As falhas devem ser controladas no código do programa.</li></ul>

Baseado na Tabela 2.5 pode-se concluir que o MPI é mais adequado para sistemas homogêneos que necessitam de muita comunicação e o desempenho é fator decisivo. Já o PVM é perfeito para sistemas heterogêneos ou que possuem uma granulação grossa com muito pouca comunicação.

O próximo capítulo apresenta um resumo dos conceitos básicos e das principais características desejadas em *benchmarks*.



Capítulo  
3

### **3. Benchmarks**

#### **3.1 Introdução**

Analizar o desempenho de máquinas paralelas é uma tarefa relativamente complexa devido ao não determinismo dessas máquinas, ou seja, elas nunca executam uma mesma tarefa da mesma forma em instantes consecutivos de tempo. Dentro desse contexto surgiram os *benchmarks* como uma poderosa ferramenta. Ainda não existe um *benchmark* que avalie completamente um sistema paralelo, seja ele multiprocessado ou um sistema distribuído. No entanto, podem dar uma boa idéia das capacidades do sistema em questão. Pela pouca sedimentação desse tipo de ferramenta, uma forma comum e errônea de medir-se desempenho é através da freqüência em que trabalha o processador, há de notar-se que *Megahertz* (Mhz) não é sinônimo de desempenho. Acatar essa afirmação é como medir a velocidade de um carro por sua RPM (Rotações por Minutos) [Int98]. A RPM simplesmente mede o quão rápido é uma engrenagem mecânica. Muitas pessoas comparam processadores apenas pela sua velocidade de *clock*. Existem outras questões que também influenciam diretamente, por exemplo, em programas numéricos de

cálculos intensivos, o que mais influenciará é como a unidade em ponto flutuante foi implementada e quão rápidos são seus registradores. Em termos de programação, deve-se considerar a forma de como o algoritmo matemático foi implementado e as ferramentas utilizadas.

Outra forma errônea de se medir desempenho é através dos MIPS (Milhões de Instruções por Segundo) os quais estão sujeitos a eficiência do compilador, ao nível de otimização utilizado e ao tipo de programa gerado. Segundo [Wei91], compiladores para o sistema operacional VAX produzem um código trinta porcento mais rápido que os compiladores para *Unix* da Berkeley. Com relação a quantidade de instruções executadas, um programa que tenha somente operações NOOP (que não fazem absolutamente nada) atinge o pico máximo de instruções por segundo, o que leva a conclusão que está-se sujeito a quantos ciclos de *clock* são necessários para executar uma instrução.

A questão de qual tipo de processador está sendo utilizado também influencia diretamente na análise. No caso de processadores RISC (*Reduce Instruction Set Code*) são executadas apenas instruções simples que não exigem muito trabalho do processador o que permite um alto grau de *pipeline*. Se comparado a um processador CISC (*Complex Instructions Set Code*) de mesma velocidade com certeza o RISC terá vantagem, mas não significa que será o mais rápido, isso irá depender diretamente do tipo de programa em questão.

Uma das abordagens corretas que começam a ser utilizadas para avaliação de desempenho é através de *benchmarks* os quais podem ser divididos por camadas [Emi96]. Nessa abordagem o *benchmark* é executado em diversos níveis de complexidade. No primeiro nível, chamado de baixo nível, são medidas as propriedades da máquina, tais como *bandwidth* e tempo para começar a executar. As informações obtidas nesse nível podem ser passadas para um nível mais alto conhecido como nível de *kernel* (núcleo). Neste nível são executadas pequenas porções de código que geralmente são utilizadas em simulações científicas, tais como resoluções de: integrais, equações diferenciais, multiplicação de matrizes, sistemas lineares, etc. O último nível é chamado de nível de aplicação ou aplicações compactas onde são realizadas complexas simulações que se utilizam do código em nível de núcleo, dentre essas simulações estão: a: cromo-dinâmica, a termodinâmica, a meteorologia, a mecânica de fluídos, etc.

A análise de desempenho também depende muito das considerações que são feitas sobre o sistema analisado tais como: número de processadores, velocidade dos processadores utilizados, nível de otimização do compilador e biblioteca de comunicação.

Um dos grandes problemas nesta área é a falta de ferramentas de suporte a *benchmarks* as quais são necessárias para avaliar com precisão o potencial de paralelismo. Dentre essas ferramentas pode-se citar os monitores de desempenho de hardware que são encontrados com mais facilidade para computadores vetoriais, e quando encontrados para máquinas com memória distribuída geralmente tem que ser executados com privilégios de administrador inviabilizando sua utilização por usuários comuns.

Na próxima seção são apresentados alguns conceitos básicos sobre *benchmarks* e como estes podem ser utilizados para medir o desempenho de uma máquina paralela.

## 3.2 Conceitos básicos

*Benchmarks* envolvem a execução de programas em ambientes reais [Don87], sendo uma das abordagens mais utilizadas para a avaliação de desempenho de sistemas. A execução de programas reais pode avaliar todos os aspectos necessários tais como a arquitetura do sistema, a eficiência do compilador, sobrecarga do sistema operacional, sobrecarga de comunicação, etc. A verificação do desempenho de um único componente do sistema não permite uma avaliação precisa de *desempenho* global, uma vez que não se conhece o impacto que a interação entre esses componentes irá causar no sistema.

Segundo a *Intel Corporation* [Int98], um *benchmark* é um programa o qual mede o desempenho de um computador, de uma parte do mesmo ou de um outro programa os quais executam sempre a mesma tarefa de tempos em tempos. Assim, *benchmarks* são usados para medir o desempenho de um sistema ou sub-sistema em uma tarefa bem definida ou conjunto de tarefas. [Wai95]

Cada *benchmark* testa um tipo diferente de trabalho. Alguns testam quão rápido um computador pode gerar um documento. Outros, o quão rápido um programa pode desenhar uma figura na tela. Alguns *benchmarks* podem testar tudo isso ao mesmo tempo. Na realidade um *benchmark* reflete a forma de como se está utilizando um programa, algoritmo ou computador.

A seguir apresenta-se como os *benchmarks* podem ser divididos e classificados.

### 3.2.1 Classificação dos *benchmarks*

Existem dois níveis de *benchmarks*, de componentes e sistemas. Os de componentes testam partes individuais de um computador, tais como, processador, memória, placas de vídeo, entre outros. Dessa forma pode-se avaliar o impacto de um componente no desempenho geral de um sistema. Dois representantes desse tipo de *benchmarks* são o SPECint95 e o CPUmark36. O primeiro é um dos *benchmarks* mais conhecidos para ambiente *Unix*. Seus testes refletem o desempenho de um processador e da arquitetura de memória de um sistema em computação intensiva em aplicações de 32 bits. O CPUmark36 é um *benchmark* para ambiente Windows 95/NT e foi desenvolvido para testar aplicações de 32 bits [Int98].

Os *benchmarks* de sistemas testam todos os componentes juntos. Eles são utilizados para responder questões tais como: O sistema *X* é mais rápido que o sistema *Y*? Quanto vou ganhar em desempenho se uma aplicação for executada num sistema com processador com *Z* Mhz? Dessa forma, se dois sistemas que tenham somente um componente diferente um do outro, pode-se usar um *benchmark* de sistema para avaliar o impacto no desempenho do sistema devido a esse componente. Um exemplo de *benchmark* de sistema é o SYSmark 4.0 [Int98] para Windows NT que pode ser executado em qualquer plataforma que aceite o sistema NT.

Além da classificação por níveis, os *benchmarks* podem ser subdivididos quanto a sua origem em: de aplicações e sintéticos. Os de aplicações medem o desempenho de programas reais, ou seja, o *benchmark* tenta modelar o comportamento do usuário em um determinado programa utilizando “scripts”, isto é, todas as ações (mudança de janelas, abertura de arquivos, etc) do usuários são gravadas em um arquivo e depois o *benchmark* as repete como se fosse o próprio usuário. A desvantagem de se utilizar um *benchmark* de aplicações é que ele pode levar muito tempo para ser concluído e é muito suscetível ao modo de como o usuário utiliza os recursos do programa em análise. Além disso, esse tipo de *benchmark* é mais apropriado para aplicações comerciais e não científicas. O SYSmark 4.0 e o SPECint95 são representantes dos *benchmarks* de aplicações.

Os *benchmarks* sintéticos são programas criados especialmente para analisar o desempenho das aplicações através de estatísticas, ou seja, ele mostra a porcentagem de memória utilizada, a quantidade de disco utilizada em *swap* e mudanças de contexto, velocidade de comunicação entre processadores e a porcentagem de uso do processador entre outras. De maneira geral essas

estatísticas são mostradas em forma de gráfico. O CPUmark36 encaixa-se dentro dos *benchmarks* sintéticos, dentro desse tipo também encaixa-se o NORTON SI32 [Int98] para Windows 95 que compara sub-sistemas (CPU, memória e cache) executando aplicações de 32 bits.

Os *benchmarks* ainda podem ser divididos nas seguintes categorias: [Wai95].

*Benchmarks proprietários* – é um tipo mais antigo, onde os desenvolvedores de programas e máquinas criavam seus próprios *benchmarks* e os utilizavam em produtos concorrentes. Claro que este beneficiava somente a empresa que o projetou. Atualmente esse tipo de *benchmark* não é mais considerado.

*Benchmarks de subsistemas padronizados* – este tipo mede alguns aspectos particulares de um subsistema. Ele provê informações sobre o desempenho alcançado em diferentes áreas do produto testado. Essas informações irão determinar a aplicabilidade e a importância do subsistema em seu ambiente.

*Benchmarks de aplicações padronizadas* – este *benchmark* constitui um tipo que simula aplicações ou executa uma série de testes em programas que são muito difundidos no comércio e na indústria. Novamente as informações provenientes desses testes ou simulações vão determinar a aplicabilidade do programa em seu ambiente.

*Benchmarks de aplicações personalizadas* – neste tipo as aplicações funcionam como seu próprio *benchmark*, ou seja, desenvolve-se a aplicação e o *benchmark* dentro da mesma para testar seu desempenho. Isso aumenta o custo de desenvolvimento, mas, por outro lado pode ajudar a criar um produto com maior qualidade e depois submete-lo a *benchmarks* padronizados para validar os testes já realizados.

Os tipos de código utilizados também podem caracterizar uma divisão dos *benchmarks*, podendo ser classificados em: sintéticos, núcleo, algoritmos e aplicações. Os Sintéticos são códigos pequenos e representam situações reais. Os de núcleo são segmentos utilizados em programas que representam situações reais tais como simulações. Algoritmos representam modos de programação bem definida como a computação numérica. E os de aplicações são programas que resolvem problemas científicos bem definidos. [Ber91]

Finalmente, pode-se dividir os *benchmarks* atuais em: *benchmarks* para arquiteturas seqüenciais e *benchmarks* para arquiteturas paralelas. Os *benchmarks* para arquiteturas seqüenciais visam analisar o desempenho de uma máquina com um único processador. Esses *benchmarks* já foram largamente pesquisados e existem alguns representantes que são amplamente utilizados, dentre eles cabe citar: *Linpack* (que atualmente também tem versões para máquinas vetoriais), *drystone*, *whetstone*, *Sysmark*, etc. Os *benchmarks* para arquiteturas paralelas são relativamente recentes representando o foco de recentes pesquisas. Alguns exemplos de *benchmarks* são apresentados na seção 3.6.

### 3.2.2 Execução de um *benchmark*

Para executar um *benchmark* algumas questões devem estar bem definidas, tais como:

- Em termos de máquina deve-se saber onde o *benchmark* irá ser executado. Isso inclui, saber qual o processador da máquina, velocidade, cache, memória, disco rígido, periféricos e número de CPUs. É ainda necessário conhecer seu estado, isto é, mono-usuário ou multi-usuário.
- Em termos de software deve-se saber qual sistema operacional será utilizado, qual a versão do *benchmark* que está sendo executado, quais os compiladores e/ou interpretadores usados na compilação do *benchmark* ou do programa que será avaliado.

A utilização de *benchmarks* para computação paralela é um tópico relativamente recente e não tão sedimentado quanto os *benchmarks* seqüenciais. Os *benchmarks* paralelos devem considerar, adicionalmente aos *benchmarks* seqüenciais, problemas de comunicação, sincronismo, ativação de processos, etc. Desta forma, surgiu a necessidade de uma padronização tanto na criação, como na execução e na representação dos dados obtidos. A próxima seção descreve toda a metodologia que deve ser utilizada.

## 3.3 Padronização dos *benchmarks* paralelos

Para estudar a padronização de *benchmarks* paralelos foi criada uma comissão chamada *Parkbench (PARallel Kernels and Benchmarks) Committee* que tem os seguintes objetivos [Hoc96]:

- Estabelecer um conjunto de *benchmarks* que são aceitos tanto pelos usuários como pelos fabricantes.
- Direcionar as atividades dos *benchmarks* paralelos para impedir a duplicação de esforços e impedir a proliferação desnecessária de *benchmarks*.
- Padronizar a metodologia utilizada em *benchmarks* paralelos e seus resultados através de um banco de dados de *benchmarks* e seus resultados.
- Tornar os resultados e os *benchmarks* de domínio público.

O relatório do *Parkbench Comitee* é composto de cinco capítulos. O primeiro diz respeito a metodologia e padronização do jargão utilizado. O segundo aos *benchmarks* de baixo nível que são responsáveis por medir o desempenho de componentes de hardware, como comunicação, latência, velocidade de processadores, etc. O terceiro capítulo diz respeito aos *benchmarks* de núcleo (“*kernel benchmarks*”) que descrevem um conjunto de subrotinas científicas, isso inclui, manipulação de matrizes, transformada de Fourier e solução de sistemas de equações diferenciais parciais. O quarto capítulo compõem-se dos códigos compactos dos *benchmarks* apresentados e algumas questões das áreas abrangidas pelo *benchmark* tais como, modelagem climática, sismologia, química computacional e física quântica. E no último capítulo algumas questões relevantes a linguagem HPF (*High Performance Fortran*), como por exemplo, usá-la como interface em diversos computadores paralelos e alguns blocos de código para testar compiladores de HPF.

Na próxima seção são apresentadas algumas das conclusões do *Parkbench Comitee*, que consideram a metodologia e padronização dos *benchmarks* paralelos.

### 3.3.1 Metodologia

Nesta seção é apresentado o padrão do *Parkbench Comitee* para apresentação de relatórios, tais como, a simbologia e as métricas que expressam os resultados de um determinado *benchmark* dando ênfase a computação científica.

### 3.3.1.1 Símbologia

Como já foi mencionado, o *Parkbench Comitee* é um associação criada para padronizar a aplicação de *benchmarks* e estabeleceu uma nomenclatura padrão para sua utilização. Ela será utilizada através de todo o trabalho e está descrita na Tabela 3.1.

Tabela 3.1 - Nomenclatura estipulada pelo *Parkbench Comitee*

Termo	Significado
flop	operação em ponto flutuante.
inst	uma instrução qualquer.
intop	Operações inteiros.
vecop	Operações vetoriais.
send	operação <i>send</i> (enviar dado para outro processador/processo).
iter	iteração de um "loop" (laço).
mref	Referência a memória (leitura/escrita).
barr	operação de barreira (sincronização entre processos).
b	dígito binário.
b	Byte (8 bits).
sol	solução ou uma única execução de um <i>benchmark</i> .
w	palavra binária (2 Bytes).
tstep	tempo para solucionar um problema.

### 3.3.1.2 Medidas de tempo

A primeira avaliação que um *benchmark* deve fazer é a avaliação de tempo. O *Parkbench Comitee* estipulou dois *benchmarks* de baixo nível para isso. São eles:

TICK1 – é definido como o intervalo entre sucessivos *ticks* de relógio. Esse *benchmark* chama uma subrotina de tempo dentro de um laço que é executado muitas vezes. O relógio do *UNIX*, por exemplo, tem um *tick* típico de  $10\mu s$ , e um computador *Cray* tem um *tick* médio de  $2\eta s$ . [Hoc96] Isso significa que o *Cray* possuiu um relógio bem mais preciso.

TICK2 – verifica se o valor retornado pelo TICK1 é correto.

O tempo num *benchmark* pode ser entendido como uma função  $T(N,p)$ , onde,  $N$  é um vetor de tamanho variável que descreve o tamanho do problema e  $p$  é o número de processadores usados na resolução do problema.

Essa métrica é utilizada para medir o desempenho de algoritmos que realizam a mesma função. Por exemplo, comparar o algoritmo do trapézio composto e o algoritmo de Simpson para solução

de integrais numéricas. De posse dessa medida é possível encontrar o desempenho temporal que é representado por  $R_T$ , definido como o inverso do tempo de execução, como mostra a equação abaixo:

$$R_T(N,p) = T^I(N,p)$$

A unidade desta medida é soluções por segundo (sol/s), ou melhor dizendo, *timesteps/s* (tstep/s). Nota-se que com esta equação é possível encontrar o melhor algoritmo para solucionar um determinado problema.

### 3.3.1.3 Contagem de operações com ponto flutuante

Um primeiro passo na avaliação de um programa científico é determinar o número de operações em ponto flutuante que são realizadas. Isso pode ser contado através da análise do algoritmo ou do seu código fonte. Em MacMahon sugere a Tabela 3.2 para contagem de operações em ponto flutuante:[Mah88]

Tabela 2.2 - Contagem de operações em ponto flutuante.

Operações	Quantidade de flop
Adição, subtração, multiplicação	1 flop
Divisão e raiz quadrada	4 flop
Exponencial, seno, cosseno e outras operações trigonométricas	8 flop
Operação if	1 flop

A avaliação inicial, através do código ou algoritmo, é chamada de nominal e representada por  $F_B(N)$ . Por exemplo, seguindo a Tabela 3.2, a execução das instruções:

```
If (a == b) {
    a = a + 1;
    b = b -1;
}
```

tem um  $F_B(N) = 3$ . Outra notação também pode ser encontrada em arquiteturas paralelas a  $F_h(N,p)$ , onde  $p$  é o numero de processadores e  $N$  é o tamanho do problema. Essa notação representa a quantidade real de operações em ponto flutuante executadas por uma máquina paralela. Este deve ser bem maior que a quantidade nominal devido ao credito dado as operações redundantes. Este valor só é encontrado através da utilização de monitores de performance.

Para analisar o desempenho de um *benchmark* num determinado computador e depois compará-lo com outros, é necessário solucionar algum tipo de problema matemático e medir os *flop* encontrados [Hoc96]. Usando a contagem de operações em ponto flutuante obtém-se a seguinte equação:

$$R_B(N,p) = F_B(N) / T(N,p)$$

onde, a quantidade de operações em ponto flutuante é divida pelo tempo de execução em  $p$  processadores. A unidade encontrada será de Mflop/s (megaflops por segundo) e também pode ser representada como Mflop<sup>-1</sup>. Deve notar-se, que  $F_B(N)$  é seqüencial, mas é utilizado porque a faixa de execução de operações em ponto flutuante é mantida. O objetivo desta métrica é comparar diferentes implementações e algoritmos em diferentes máquinas e arquiteturas para solucionar o mesmo problema, sendo que o que será levado em consideração é o tempo de execução.

A utilização de  $F_B(N)$  (que é seqüencial), pode ser utilizado devido as arquiteturas com memória distribuída executarem uma quantidade diferente de operações em ponto flutuante em cada processador, mantendo a faixa dos resultados. O objetivo deste *benchmark* é comparar diferentes implementações e algoritmos em diferentes máquinas e arquiteturas para solucionar o mesmo problema.

### 3.3.1.4 Benchmarks para aplicações científicas

Alguns *benchmarks* tais como o Linpack e o Whetstone que se utilizam de grande quantidade de operações em ponto flutuante são mais apropriados para máquinas seqüenciais [Gus86]. Com o aumento crescente de máquinas paralelas esses *benchmarks* já não conseguem representar muito bem o desempenho alcançado pelas aplicações científicas nesse tipo de máquina. Um dos principais requisitos que um *benchmark* deve considerar é que ele possa rodar em uma grande variedade de arquiteturas. O *NAS Parallel Benchmark* tenta suprir essa necessidade pois pode rodar tanto em máquinas vetoriais vetorial, como em paralelo e *pipeline*.

Todo *benchmark* científico deve ser acompanhado das seguintes informações:

- Formato numérico utilizado;
- Sistema Operacional e sua versão;

- Linguagem utilizada;
- Tamanho da memória, cache, discos e outras informações de armazenamento;
- Número de processadores, velocidade de comunicação e velocidade dos processadores;
- Forma de medir o desempenho.

Estes requisitos são a base para a reprodução do experimento. Quando um *benchmark* é executado deve-se assumir que nenhum outro usuário está usando a máquina para outros *benchmarks* ou mesmo para outras aplicações[Gus86].

Quando um *benchmark* científico é implementado algumas regras devem ser seguidas. Algumas delas são especificadas pela IEEE, a qual, propõem que:

- Todos os números em ponto flutuante devem ser de 64 bits;
- Os números complexos devem ser formado por 128 bits, isto é, dois números em ponto flutuante;
- Para medir o tempo de execução, o tempo de inicialização de variáveis, vetores e matrizes deve ser desconsiderado sendo desaconselhado a tomada de tempos intermediários, ou seja, o primeiro instante de tempo medido deve iniciar-se junto com os cálculos e o último instante de tempo deve ser tomado quando os cálculos terminarem;
- Finalmente devem ser computados os Mflop (lê-se “megaflops”), onde a quantidade total de operações com ponto flutuante deve ser dividida pelo tempo de execução dos cálculos.

Na próxima seção serão apresentadas as principais restrições e erros cometidos na utilização de uma das métricas mais conhecidas para a análise de desempenho o *speedup*.

### 3.4 Speedup e suas restrições

Dentro da computação paralela o *speedup* é uma medida muito popular e utilizada para avaliar o desempenho de programas paralelos. Sendo que uma das coisas esquecidas por esses avaliadores é a velocidade dos processadores. Isso quer dizer que analisar o *speedup* sem analisar a velocidade dos processadores leva a conclusões irreais [Hoc96]. Por isso o *Parkbench Comitee*

tem lutado contra a publicação de *speedup* em relatórios de publicação de resultados de *benchmarks*.

Como explicado no Capítulo 2, *speedup* relaciona o  $T_s$  que é o tempo de execução em um processador e  $T_P$  que é o tempo de execução em vários processadores, assim, entram as seguintes questões: [Hoc96]

- Se o algoritmo continua o mesmo, qual é a probabilidade de haver sobrecarga desnecessária na computação paralela ?
- Deve ser usado o melhor algoritmo serial para solucionar o problema ?
- Se é usado outro algoritmo serial, qual a possibilidade de comparação entre esses algoritmos ?

Uma análise errônea sobre os resultados obtidos no *speedup* seria: Supondo que existem dois algoritmos que solucionam o mesmo problema denominados por A e B respectivamente. A Tabela 3.3 mostra um exemplo de tempos de execução dos algoritmos seriais e paralelo e o *speedup* alcançado.

Tabela 3.3 - Exemplo de tempo e *speedup*.

Algoritmo	Tempo Serial (s)	Tempo em Paralelo(s)	speedup
A	10	5	2
B	7	4	1.75

O algoritmo A teve um tempo de execução serial de 10 segundos e de 5 segundo em paralelo obtendo um *speedup* de 2. O algoritmo B teve um tempo de execução serial de 7 segundos e de 4 segundos em paralelo obtendo um *speedup* de 1.75. Se for levado em consideração somente o *speedup* o algoritmo escolhido como o melhor seria o A, que é uma conclusão totalmente equivocada. A melhor escolha seria o algoritmo B, não pelo *speedup* alcançado, e sim porque ele soluciona o mesmo problema mais rápido.

Devido a esses problemas o *Parkbench Comitee* tomou as seguintes decisões:

- Nenhuma estatística ou resultados envolvendo *speedup* poderá ser mantida nos bancos de dados de *Parkbench Comitee*;

- Estatísticas baseadas em *speedup* nunca podem ser consideradas como figura de mérito quando sistemas diferentes estão sendo comparados.
- *Speedup* pode ser usado em estudos de *desempenho* quando estão sendo avaliadas as características de um único sistema. Sendo que a base de  $T_1$  deve ser bem especificada.
- O valor de  $T_1$  (tempo inicial) deve ser baseado numa implementação monoprocessada eficiente. Nenhum código de sincronização ou de passagem de mensagem pode estar presente.
- Como muitos problemas não se encaixam facilmente em algoritmos monoprocessados, ou seja possuem um paralelismo natural, é permitido que algumas estatísticas utilizando *speedup* possam ser utilizadas, desde que para um número pequeno de nós e que o número de processos não seja superior ao número de processadores.

A seguir são apresentados alguns exemplos de *benchmarks* que serão utilizados neste trabalho.

## 3.5 Exemplos de *benchmarks* paralelos

### 3.5.1 Genesis

Este *benchmark* foi criado para avaliar o desempenho de sistemas com memória distribuída. Seu foco principal é a avaliação de programas científicos e das engenharias [Gle94]. Foi desenvolvido em HPF e em Fortran 77. Em Fortran 77 trabalha com PARMACS e com PVM, sendo que uma versão que trabalha com MPI está em desenvolvimento. Ele é dividido em três categorias: *benchmarks* de baixo nível, *kernel benchmarks* (*benchmarks* de núcleo) e aplicações.

No *benchmark* de baixo nível são medidos os parâmetros básicos de um máquina, tais como tempo de comunicação, custo de sincronização e capacidade de vetorização. Este nível apresenta as seguintes funções [Gle94]:

- TICK1 – mede a resolução do relógio do sistema
- TICK2 – confirma a precisão encontrada pelo TICK1.
- RINF1 – mede a capacidade de vetorização.

- COMMS1 – *benchmark* de ping-pong, ou seja, troca de mensagens entre dois processadores.
- COMMS2 – *benchmark* onde dois processadores trocam mensagens de forma bidirecional.
- COMMS3 – *benchmark* que mede *bandwidth*.
- SYNCH1 - mede o custo de sincronização (barreiras).
- POLY1 – mede a saturação de entrada da memória cache
- POLY2 – mede a saturação de saída da memória cache, tanto POLY2 como POLY1 são versões monoprocessadas.
- POLY 3 – tem a mesma função de POLY1, com a diferença que os dados necessários estão armazenados em outros processadores.

O *benchmark* de núcleo, é o que apresenta o código mais complexos [Gle94] e representa um núcleo com computação intensiva de diferentes aplicações. Suas principais funções são:

- TRANS1 – calcula a transposta de uma matriz.
- IO1 – mede a taxa de transferência paralela onde cada processador escreve 12.5 Mb de dados e depois efetua sua leitura.
- FFT1 – FFT (*Fast Fourier Transformation* – Transformação Rápida de Fourrier) de uma dimensão.
- SOLVER – solução de um sistema linear pelo método do gradiente conjugado.
- PDE1 – resolve uma equação tridimensional de Poisson .
- PDE2 - resolve uma equação bidimensional de Poisson .

Nos *benchmarks* de aplicações são realizadas algumas simulações, a saber [Gle94]:

- QCD1 – simulação de *chromo* dinâmica utilizando o método de Monte Carlo.
- GR1 – simulação de relatividade.
- LPM1 – simulação de dispositivos eletrônicos.
- MD1 – simulação de dinâmica molecular.

### 3.5.2 NAS Parallel Benchmarks

O NAS (*Numerical Aeronautical Simulation*) foi criado pelo *NASA Ames Laboratory* que fica localizado na California. O NAS surgiu devido a necessidade de avaliar sistemas paralelos de alto desempenho que não fossem vetoriais [Bai94a], além da necessidade de verificar a portabilidade das máquinas, característica de fundamental importância. Este *benchmark* é construído com oito algoritmos (denominados por duas letras), onde cinco são do tipo *kernel* e três simulações:

- EP (*Embarrassingly Parallel*) – utilizando o algoritmo de Monte-Carlo, consegue medir o desempenho de uma máquina em ponto flutuante. Este núcleo não utiliza comunicação entre processos.
- MG (*Multigrid*) – algoritmo que soluciona equações de Poisson. Mede o desempenho de comunicação utilizando mensagens longas e curtas.
- CG (*Conjugate Gradient*) – calcula o gradiente conjugado. Faz cálculos em matrizes esparsas, simétricas e positivas. Utilizado para medir o desempenho de comunicação de longa distância.
- FT – soluciona equações diferenciais parciais. Faz um teste rigoroso com a comunicação de dados.
- IS (*Integer Sorting*) – é o único que não utiliza operações em ponto-flutuante. Testa a capacidade do sistema em trabalhar com números inteiros e ao mesmo tempo testa a comunicação utilizando os mesmos.

As simulações utilizam três métodos para solucionar equações. O primeiro é o LU (*lower-upper*) que fatora uma matriz regular-esparsa até sua solução utilizando blocos de 5x5. Este algoritmo apresenta um nível baixo de paralelismo se comparado aos outros dois [Bai94b].

O segundo SP (*scalar pentadiagonal*), faz a multiplicação de sistemas independentes não-diagonais dominantes e resolve equações escalares pentadiagonais. O terceiro método utilizado é o BT (*block tridiagonal*) que é muito semelhante ao SP sendo que a diferença está na quantidade de transmissão [Bai94b].

### 3.5.3 ParkBench (*Parallel Kernel Benchmark*)

Este *benchmark* consiste de 4 pacotes distintos, onde todos eles requerem que o PVM ou o MPI estejam adequadamente instalados e configurados. A Tabela 3.4 mostra um resumo dos pacotes que o compõem, alguns deles foram extraídos de outros *benchmarks* tais como o NAS e o Genesis.

Tabela 3.4 - *Benchmarks* que formam o *ParkBench*.

Baixo-Nível	Kernel	NPB2.1	Aplicações
Comms1	FT	BT	BT
Comms2	LU_solver	FT	LU
Comms3	MATMUL	LU	SP
Poly1	TRANS	MG	PSTSWM
Poly2	QR	SP	
Poly3	TRD		
Rinf1			
Synch1			
Tick1			
Tick2			

A primeira coluna mostra os *benchmarks* de baixo nível , sendo que todos eles foram extraídos do Genesis. A segunda coluna exibe os *benchmarks* de *kernel* (núcleo). Eles foram retirados do Genesis e do NAS versão 2.1 com exceção do MATMUL e do TRD. Na terceira coluna é mostrado quais algoritmos foram tirados no NAS versão 2.1 para formar o pacote. E finalmente a quarta coluna mostra quais são os *benchmarks* de aplicação, que com exceção do PSTSWM também foram retirados do NAS.

Todos os pacotes estão disponíveis tanto para PVM como para MPI.

## 3.6 Outros exemplos de *benchmarks* [Way95]

### Linpack

Este é um *benchmark* para aplicações que se utilizam de rotinas de álgebra linear, mais precisamente com rotinas de manipulação de matrizes. É encontrado tanto em Fortran como em C. Todo o tempo do *benchmark* é gasto em duas rotinas básicas, a *saxpy* para precisão simples, e a *daxpy* para precisão dupla, as quais dão a solução de um sistema linear pelo método da triangularização de Gauss com pivoteamento. Ambas trabalham com operações matriciais do

tipo  $y(i) = y(i) + a*x(i)$ . A versão padrão trabalha com matrizes de 100 x 100, mas existem versões que trabalham com matrizes de 300 x 300 e de 1.000 x 1.000. Sua principal desvantagem é que ele trabalha apenas com aplicações matriciais. Até algum tempo atrás esse *benchmark* era específico para máquinas seriais, atualmente sua aplicação se dá também em máquinas vetoriais.

### **Whetstone**

É um dos primeiros *benchmarks* sintéticos criados, foi desenvolvido para analisar o desempenho de programas numéricos que se utilizam intensamente de ponto flutuante. Foi desenvolvido pela *National Physical Lab* baseado totalmente em estatística e foi escrito inicialmente em Algol 60, atualmente possui versões em Fortran77/90 e em linguagem C.

### **Nhfsstone**

Este *benchmark* é utilizado para medir o desempenho de servidores de arquivos que utilizam o protocolo NFS. Este *benchmark* é complementado com o SPEC e com o LADDIS.

### **Drystone**

É um *benchmark* sintético que avalia o desempenho de sistemas de programação inteira. Também baseia-se em estatísticas. Foi originalmente desenvolvido em Ada e atualmente é mais encontrado em linguagem C.

## **3.7 Considerações finais**

Os *benchmarks* paralelos são relativamente recentes no mercado, sejam eles acadêmicos ou comerciais. As pesquisas nessa área vem crescendo lentamente, sendo que o alvo dos *benchmarks* paralelos é a computação paralela científica que vem exigindo cada vez mais poder computacional. Essa é a razão da maioria dos *benchmarks* estar escrito em Fortran com suas extensões paralelas. Apesar de que, atualmente versões em C vem sendo criadas devido a crescente otimização dos compiladores.

Um bom *benchmark* deve ser portável entre máquinas e escrito numa linguagem popular de alto nível como C ou Fortran. Deve utilizar bibliotecas conhecidas de passagem de mensagem como o PVM ou MPI que permitem uma mudança de plataforma praticamente sem nenhuma mudança

de código, ou utilizar a linguagem HPF que atualmente está bem sedimentada na área da computação científica.

Além das características descritas acima ele deve representar algum tipo de programação em especial tais como: sistemas operacionais, compiladores, programação numérica, programação comercial, etc.

Com relação a padronização, graças ao *Parkbench Comitee* não existe uma confusão generalizada de *benchmarks*. Este comitê criou padrões de medição e centralizou o controle sob suas mãos, validando resultados e armazenando-os em um banco de dados centralizado. A cada período de tempo, o *Parkbench Comitee* lança um relatório contendo as novas criações na área de *benchmarks* e os resultados obtidos nas avaliações de desempenho, isso inclui os resultados da utilização de *benchmarks* já conhecidos.



Capítulo  
4

## ***4. Algoritmos Numéricos***

### **4.1 Introdução**

Devido ao grande esforço de cálculo exigido pelos métodos numéricos, antes do advento dos computadores estes métodos sofriam grandes restrições. A partir da década de 40 quando começaram a surgir os computadores, muitos problemas passaram a ser resolvidos a contento [Cla94]. Nos últimos tempos, a capacidade de processamento dos computadores vem crescendo vertiginosamente melhorando a exatidão e aumentando a velocidade de processamento.

A matemática numérica, no ponto de vista da computação, faz parte de uma área maior denominada **Matemática Computacional**. Dessa forma, alguns fatores devem ser considerados. O primeiro é que algumas propriedades básicas da aritmética não valem no mundo digital, principalmente se forem considerados os números infinitos. Um computador só pode representar números finitos porque as palavras de memória também são finitas. A Matemática Computacional pode ser subdividida em várias sub-áreas conforme mostra a Figura 4.1 [Cla94].

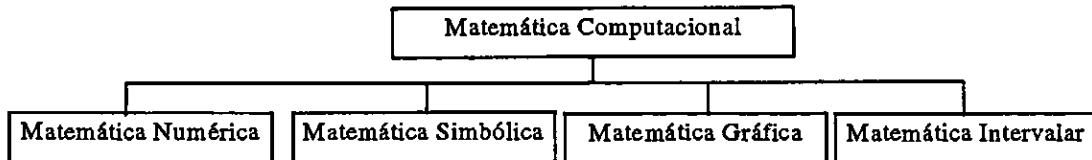


Figura 4.1 - Divisão da Matemática Computacional.

A Matemática Numérica procura encontrar soluções aproximadas de problemas através da representação do mesmo por um modelo matemático. A Simbólica trata das formas literais. A Matemática Gráfica estuda os problemas através de uma forma gráfica e tenta solucioná-los através da mesma técnica. E finalmente, a Matemática Intervalar trata os dados na forma de intervalos controlando os limites de erro dos processos da Matemática Numérica de forma 100% confiável.

O objetivo da Matemática Numérica é estudar processos numéricos, ou seja, algoritmos para a solução de problemas visando a máxima confiabilidade e economia. Como esses algoritmos são implementados em computadores, alguns fatores devem ser considerados. O primeiro deles é, como já foi mencionado, que os computadores só podem representar números finitos. Outro fator está relacionado com o tempo de execução. Neste caso encaixa-se a computação paralela (multiprocessada ou distribuída) que tem como principal objetivo o aumento de desempenho na execução de uma aplicação.

Para solucionar um problema através de um computador é necessário executar uma série de passos preestabelecidos de modo formal, onde esses passos dão origem ao que se chama de algoritmo, no caso da matemática computacional, algoritmo numérico. Um algoritmo numérico de boa qualidade deve apresentar as seguintes características [Cla94]:

- Inexistência de erros lógicos;
- Inexistência de erro operacional;
- Quantidade finita de cálculos;
- Existência de um critério de exatidão;
- Independência de máquina;
- Eficiência.

Cada uma destas características será comentada a seguir.

Inexistência de erros lógicos vale para qualquer algoritmo, seja ele numérico ou não. Sendo que nos algoritmos numéricos faz-se necessário uma previsão completa de todas as possibilidades, considerando as características das operações aritméticas que serão utilizadas. Por exemplo, qual seria o algoritmo correto para encontrar o valor de  $x$  na equação  $ax = b$  ?

Intuitivamente pode-se pensar que  $x = b/a$  seria a solução correta. Mas, se esta solução for analisada pelo ponto de vista dos algoritmos numéricos ela está incompleta. O algoritmo correto seria:

*Se  $a = 0$  então*

*Se  $b = 0$  então Imprima (“Não existe solução”)*

*senão Imprima (“Erro -Divisão por zero”)*

*Fim\_se*

*Senão*

*Se  $b = 0$  então Imprima (“Solução identidade ( $x=0$ )”)*

*Senão  $x \leftarrow b/a$*

*Fim\_se*

### Inexistência de Erros Operacionais

Corrigidos todos os erros lógicos parte-se para a análise de restrições, ou seja, verificar se o algoritmo não está violando as restrições da máquina, como por exemplo, se a precisão dos números não ultrapassa a capacidade em ponto flutuante da máquina. Esses erros são geralmente detectados em tempo de execução e caracterizam os erros operacionais. A representação matemática das restrições é dada por [Cla94]:

- Seja  $T = \mathbb{R}$  (conjunto dos reais) e  $x \in \mathbb{R}$
- $\forall x \in T, -x \in T$
- $t_1 = \text{limite\_inferior} \{x / x \in T \text{ e } x > n\}$
- $t_2 = \text{limite\_superior}\{x / x \in T \text{ e } x < z \text{ e } x > n\}$  onde  $z > t_1$

$t_1$  representa o limite inferior, ou seja, o menor valor de  $x$  que pode ser utilizado. E  $t_2$  representa o limite superior, isto é, o maior valor que  $x$  pode alcançar. Se existir algum valor de  $y$ , de forma

que,  $y < t_1$  ou  $y > t_2$  tem-se dois erros chamados de *underflow* e *overflow* respectivamente. Isso caracteriza um erro operacional.

### Quantidade finita de cálculos e existência de critérios de exatidão

Existem muitos métodos para resolução de problemas numéricos que são iterativos, isto é, repetitivos. Dessa forma, tem-se a necessidade de estabelecer um critério de parada, ou seja, uma **quantidade finita de cálculos** (repetições). Um dos principais motivos que leva a quantidade finita de cálculos é a capacidade finita de uma máquina representar números em ponto flutuante. Levando isso em consideração, todo resultado obtido em um computador deve estar dentro de um **critério de exatidão** fornecido antes de iniciado o cálculo. Deste modo o resultado pode ser escrito na forma de:

$$\text{Resultado} = \text{Valor aproximado} \pm \text{Limite de erro}$$

onde valor aproximado é o resultado encontrado pela máquina e o limite de erro é a diferença entre o que foi encontrado pela máquina e o valor real.

### Independência de máquina

Quando o algoritmo estiver pronto para ser implementado, este não pode ser dependente de uma máquina, a não ser que seu propósito seja muito específico. De maneira geral os algoritmos numéricos devem ter **independência de máquina** visando a máxima portabilidade.

### Eficiência do algoritmo

Na solução de qualquer problema, uma grandeza comum sempre aparece. Essa grandeza é conhecida como economizar os recursos envolvidos na solução, ou seja, aumentar ao máximo a relação custo/benefício. E isso só é conseguido com a **eficiência** do algoritmo. Tem-se que deixar bem claro a diferença entre eficiência e eficácia. A eficácia é a capacidade do algoritmo de produzir uma resposta correta. E a eficiência é a capacidade do algoritmo em obter uma resposta correta no menor tempo possível. Por exemplo, o algoritmo de Crammer para resolução de sistemas lineares é muito eficaz, mas é extremamente ineficiente para um certo número  $n$  de elementos, por exemplo, com  $n=10$  o sistema já se torna extremamente lento.

Analisando-se as características discutidas, observa-se que um ponto crucial no desenvolvimento e execução de um algoritmo numérico é o tratamento de erros. Desta forma, a próxima seção é “destinada” ao estudo de erros em algoritmos numéricos.

## 4.2 Erros em algoritmos numéricos

Dentro dos algoritmos numéricos existem três tipos básicos de erros: os *inerentes*, os de *discretização* e os de *arredondamento*. Os inerentes aparecem quando um modelo matemático é criado ou simplificado. Os valores de, por exemplo, tempo, temperatura, distância e intensidade luminosa são medidos através de aparelhos que também são limitados em sua representação numérica. Dessa forma, esses erros não podem ser evitados ou minimizados.

Os erros de discretização, também conhecidos como erros de aproximação ou de truncamento, são erros que ocorrem quando um processo infinito de cálculo é substituído por um processo finito, ou seja, por um processo discretizado que o torne computacionalmente viável. Por exemplo, qual seria a solução computacionalmente viável para a série:

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Nota-se que é uma série infinita e para torná-la computacionalmente viável tem-se que transformá-la numa série discretizada, isto é, efetuar somente algumas parcelas do somatório. Com isso, cria-se um erro causado pelo abandono das parcelas que não foram somadas.

Os erros de arredondamento surgem quando se trabalha com máquinas digitais representando números reais [Cla94]. De maneira geral o arredondamento é feito com números em ponto flutuante para o mais próximo do número real. Neste caso entram mais dois tipos de erro: o *erro absoluto* e o *erro relativo*. O erro absoluto é caracterizado pela diferença entre o número real e o valor arredondado.

$$E_A = |x^* - x| ; \text{ onde } x^* \text{ é o número real e } x \text{ é o número arredondado}$$

Seguindo a mesma representação, tem-se o erro relativo que é caracterizado:

$$E_r = \frac{|x^* - x|}{|x|} \quad \text{ou} \quad E_r = \frac{|x^* - x|}{|x^*|}$$

Tanto o erro relativo como o erro absoluto são utilizados nos algoritmos numéricos como critério de parada.

A seguir são apresentados alguns exemplos de algoritmos numéricos. Para cada algoritmo é desenvolvido o método de solução considerando o teste de parada, baseado em alguma forma de erro.

## 4.3 Exemplos de Algoritmos Numéricos

A maioria dos algoritmos numéricos utiliza métodos iterativos para resolver o problema considerado. Os métodos iterativos estão divididos em quatro partes:

- Estimativa inicial: onde apresenta-se uma ou mais aproximações da solução desejada;
- Atualização: considera-se uma fórmula que atualize a solução aproximada;
- Critério de parada: é uma regra que permite a parada do processo iterativo quando uma solução é encontrada ou quando a solução converge;
- Estimador de exatidão: é associado ao critério de parada e provê uma estimativa do erro cometido em relação a solução real.

### 4.3.1. Resolução de Sistemas Lineares por métodos iterativos

Nesta seção mostra-se o método iterativo de Jacobi para resolução de sistemas lineares por ser naturalmente paralelo, ou seja, não há dependência entre as equações utilizadas no sistema.

A classe dos métodos iterativos de Gauss-Seidel e Jacobi, resolvem o sistema convertendo um vetor  $x^{(k)}$  em outro,  $x^{(k+1)}$ , que depende de  $x^{(k)}$  ( $x$  no passo  $k$ ). A idéia central dos métodos iterativos para solução de sistemas lineares pode ser colocada na seguinte forma[Rug96]:

Seja o sistema linear  $Ax = b$ , onde:

$A$  : matriz dos coeficientes,  $n \times n$ ;

$x$  : vetor de variáveis,  $n \times 1$ ;

$b$  : vetor dos termos independentes,  $n \times 1$ .

Inicia-se o passo em  $x^{(0)}$ , onde  $x^{(0)}$  é o vetor de aproximação inicial.

De um modo geral, as próximas iterações ( $k + 1$ ) são dependentes da iteração no passo  $k$ .

### Teste de Parada

O processo de iteração repete-se até que o vetor  $x^{(k+1)}$  esteja próximo do vetor  $x^{(k)}$ , dado um determinado erro ou precisão ( $\varepsilon$ ) de modo que  $M_R^{(k)} < \varepsilon$ , onde:

$$M_R^{(k)} = \frac{\|x^{(k+1)} - x^{(k)}\|_{\infty}}{\|x^{(k+1)}\|_{\infty}} \quad \text{ou} \quad M_R^{(k)} = \frac{\|x^{(k+1)} - x^{(k)}\|_{\infty}}{\|x^{(k)}\|_{\infty}}$$

sendo que  $\|x\|_{\infty} = \max_{1 \leq i \leq n} |x_i|$ . A seguir mostra-se o método iterativo de Jacobi para solucionar sistemas de equações lineares [Rug96].

### Método iterativo de Jacobi

Considere-se o seguinte sistema:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ a_{31}x_1 + a_{32}x_2 + \dots + a_{3n}x_n = b_3 \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases}$$

Isolando o termo  $x_i$ , onde,  $i = 1, 2, \dots, n$ , em cada equação tem-se:

$$\begin{cases} x_1 = \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \dots - a_{1n}x_n) \\ x_2 = \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \dots - a_{2n}x_n) \\ \vdots \\ x_n = \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \dots - a_{n,n-1}x_{n-1}) \end{cases}$$

O método de Jacobi consiste em, dado uma aproximação inicial  $x^{(0)}$ , encontrar  $x^{(1)}, \dots, x^{(k)}$ , através de um relação recursiva, ou seja, usar o valor de  $x^{(k)}$  para encontrar a próxima aproximação  $x^{(k+1)}$  através das equações:

$$\begin{cases} x_1^{(k+1)} = \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \dots - a_{1n}x_n^{(k)}) \\ x_2^{(k+1)} = \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \dots - a_{2n}x_n^{(k)}) \\ \vdots \\ x_n^{(k+1)} = \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(k)} - a_{n2}x_2^{(k)} - \dots - a_{n,n-1}x_{n-1}^{(k)}) \end{cases}$$

A aproximação inicial  $x^{(0)}$ , que geralmente é usada nos métodos iterativos é:

$$x^{(0)} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad (\text{I})$$

No método de Jacobi, essa aproximação conduz a segunda iteração como sendo:

$$x^{(1)} = \begin{pmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} \\ \vdots \\ \frac{b_n}{a_{nn}} \end{pmatrix} \quad (\text{II})$$

neste caso pode-se utilizar a iteração  $x^{(1)}$  diretamente como sendo a aproximação inicial  $x^{(0)}$ . Computacionalmente, devido a velocidade de cálculo, pode-se adotar como aproximação inicial o vetor de zeros (I)

**Exemplo: Resolver o seguinte sistema com  $\varepsilon=0.05$ :**

$$\begin{cases} 10x_1 + 2x_2 + x_3 = 7 \\ x_1 + 5x_2 + x_3 = -8 \\ 2x_1 + 3x_2 + 10x_3 = 6 \end{cases}$$

Pela equação dada acima e considerando-se a opção 2 para escolha da primeira iteração, tem-se

$$\text{que } \mathbf{x}^{(0)} = \begin{pmatrix} 0.7 \\ -1.6 \\ 0.6 \end{pmatrix}.$$

O processo para a obtenção das iterações é:

$$\begin{cases} x_1^{(k+1)} = -0.2x_2^{(k)} - 0.1x_3^{(k)} + 0.7 = 0.96 \\ x_2^{(k+1)} = -0.2x_1^{(k)} - 0.2x_3^{(k)} - 1.6 = -1.86 \text{ para } k=0,1,\dots, \text{ até } M_R^{(k)} < \varepsilon. \\ x_3^{(k+1)} = -0.2x_1^{(k)} - 0.3x_2^{(k)} + 0.6 = 0.94 \end{cases}$$

A Tabela 4.1, mostra os valores utilizados para  $k$  e os respectivos resultados para  $x_i$  quando utilizado o processo de obtenção das iterações, a última linha apresenta o erro relativo correspondente.

Tabela 4.1 - Resultados alcançados no processo de obtenção das iterações.

$k$	0	1	2	3
$x_1$	0,7	0,96	0,978	0,9994
$x_2$	-1,6	-1,86	-1,98	-1,9888
$x_3$	0,6	0,94	0,699	0,9984
Erro ( $M_R^{(k)}$ )	-	0,1828	0,0606	0,0163

Nota-se que na terceira iteração obteve-se um  $M_R^{(k)} < \varepsilon$  o que ocasiona a parada do processo.

Este método possui um paralelismo natural, ou seja, não existem dependências entre as equações. Dessa forma o paralelismo pode ser feito mandando-se um conjunto de equações para cada processo/processador.

### 4.3.2 Integração Numérica

Existem quatro abordagens para se solucionar uma integral. Pelos métodos analíticos, mecânicos, gráficos e por métodos numéricos. Olhando pelo ponto de vista analítico existem inúmeras regras a serem seguidas, mas nem sempre essas regras são fáceis de se processar. Por exemplo, calcule:

$$f(x) = \int x^2 dx$$

Uma resposta simples e rápida pode ser dada, ou seja,  $f(x) = x^3/3$ . Mas a solução de:

$$f(x) = \int \frac{1}{x} dx$$

já não é tão trivial assim, sua solução seria  $f(x) = \ln x$ . Alguns softwares como o *Reduce* e o *Formac* utilizam-se de técnicas de inteligência artificial para produzirem tais primitivas, mas elas exigem muita memória da máquina. [Cla94]

O método mecânico utiliza instrumentos para determinar uma área delimitada por uma curva qualquer, que podem ser planos simples. O método gráfico é baseado em segmentos de reta e utiliza basicamente o plano cartesiano. Por não ser de interesse neste trabalho estes métodos não serão detalhados.

No método numérico será calculado uma integral definida de modo que será utilizado uma combinação linear de valores de uma função  $f$  em certos pontos de  $x_i$ , onde  $a \leq x_i \leq b$ , chamados de nós e certos valores  $\omega_i$  que constituem os pesos, isto é:[Cla94]

$$\int_a^b f(x) dx \cong \omega_1 f(x_1) + \omega_2 f(x_2) + \dots + \omega_n f(x_n) + \omega_{n+1} f(x_{n+1})$$

### Regra dos trapézios

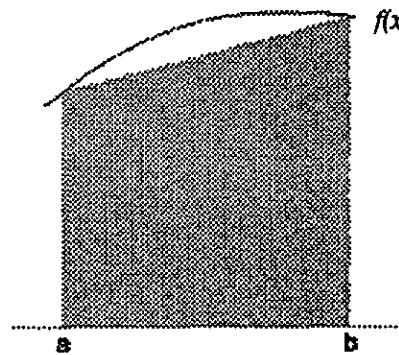
Suponha que  $f$  seja uma função cujo gráfico é mostrado na Figura 4.2. Se  $f$ , for aproximada, por um polinômio de 1º grau ( $f^*$ ), chega-se a seguinte fórmula:

$$f(x) \cong f^*(x) = f(a) \frac{(x-a)}{(b-a)} + f(b) \frac{(x-b)}{a-b} \quad (\text{III})$$

Integrando ( III ) de  $a$  até  $b$  obtém-se:

$$\begin{aligned} \int_a^b f(x) dx &\cong \int_a^b f^*(x) dx = \frac{f(a)}{b-a} \int_a^b (x-a) dx + \frac{f(b)}{a-b} \int_a^b (x-b) dx \\ \int_a^b f(x) dx &\cong \frac{b-a}{2} [f(a) + f(b)] \end{aligned}$$

Esta equação é conhecida como regra do trapézio pois sua expressão final irá conduzir à área do trapézio, como mostra a área em cinza na Figura 4.2.

Figura 4.2 – Exemplo de função  $f(x)$ .

A área entre o trapézio e a curva é o erro cometido.

Se o intervalo  $[a,b]$  for subdividido em  $n$  subintervalos e em cada um deles for feita uma aproximação da função  $f$  utilizando a regra do trapézio tem-se o que se chama de regra dos Trapézios Composta, conforme mostra a Figura 4.3. Note que a área não coberta pelo trapézio agora é bem menor, isso significa uma maior precisão nos cálculos.

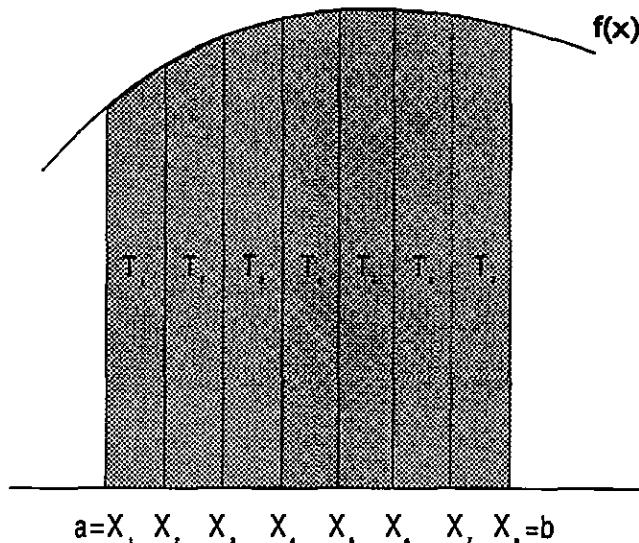


Figura 4.3 - Regra do trapézio composta

Para obter-se a regra do trapézio composta, basta a aplicação da regra do trapézio em cada subintervalo  $[X_j, X_{j+1}]$ ,  $j = 1, \dots, n$ . Com isso tem-se a seguinte equação:

$$T(h) = \frac{h}{2} [f(x_1) + 2(f(x_2) + f(x_3) + \dots + f(x_n)) + f(x_{n+1})]$$

onde  $f(x_i)$  é a função calculada no ponto inicial de cada subintervalo e  $h = \frac{b-a}{n}$ .

**Exemplo: Encontre a solução de:**

$$\int_0^1 e^{-x^2} dx$$

utilizando  $n=4$ , onde  $h = (b - a)/n$ .

$h = (1 - 0) / 4 = 0.25$ , dessa forma  $x_1 = 0.0, x_2 = 0.25, x_3 = 0.5, x_4 = 0.75, x_5 = 1.0$

$$T(0.25) = \frac{0.25}{2} [f(x_1) + 2f(x_2) + 2f(x_3) + 2f(x_4) + f(x_5)]$$

$$T(0.25) = 0.125[1 + 2*0.939413063 + 2*0.778800783 + 2*0.569788 + 0.367879441] = 0.742984098$$

Para realizar a paralelização deste algoritmo basta dividir o intervalo pelo número de processos ou processadores que se quer sendo que um  $n$  grande pode ser usado para aumentar a precisão e a carga de processamento em cada processo/processador, este procedimento é descrito com detalhes no Capítulo 6.

### 4.3.3 Multiplicação de matrizes

Apesar desse algoritmo não ser numérico, ele está sendo apresentado neste capítulo por exigir um grande esforço de processamento apresentando uma complexidade  $n^3$ , ou seja, o crescimento de custo computacional é relativamente grande e rápido.

O algoritmo é simples onde a matriz resultante da multiplicação é obtida através do pequeno algoritmo a seguir:

```

Definir tamanho das matrizes quadradas -> TAM;
Inteiros i,j,k;
Reais A[TAM][TAM],B[TAM][TAM],C[TAM][TAM];
Criar matriz A e B;
Inicializar matriz C;
Para k=0 ate k< TAM faca
    Para i=0 ate i < TAM faca
        Para j=0 ate j < TAM faca
            C[k][i] <- C[k][i] + (A[k][j] * B[j][i]);
        Fim-para-j
    Fim-para-i
Fim-para-k

```

Como nos demais algoritmos sua paralelização é também mais detalhada no Capítulo 6.

## 4.4 Considerações finais

A busca pelo alto desempenho é o que leva a grande utilização do processamento paralelo [Alm94]. As diversas áreas na qual a computação paralela pode ser aplicada , seja na computação científica, na indústria ou em aplicações militares requerem cada vez mais poder computacional em virtude dos algoritmos cada vez mais complexos.

No projeto de algoritmos paralelos vários fatores, que não são considerados nos algoritmos seqüenciais, devem ser considerados, tais como: arquitetura onde o algoritmo será implementado, se o mesmo possui muita comunicação, se o algoritmo possui dependência em seus cálculos e o sincronismo imposto pelo problema.

Na matemática computacional existem muitos algoritmos que são intrinsecamente paralelos e podem ser implementados tanto seguindo a abordagem MPMD como SPMD, explorando granulação grossa e atingindo valores de *speedup* e eficiência atrativos. Os algoritmos descritos neste capítulo podem ser desenvolvidos tanto em computadores multiprocessados como em sistemas paralelos virtuais.

O próximo capítulo descreve todo o ambiente utilizado no desenvolvimento dos algoritmos numéricos aqui descritos.



Capítulo  
5

## ***5. Descrição do Ambiente***

Este capítulo descreve os ambientes utilizados no desenvolvimento deste trabalho começando pelo hardware e em seguida detalhando o software que auxiliou o desenvolvimento e a execução das aplicações.

### **5.1 Hardware Utilizado**

#### **5.1.1 O IBM SP2**

IBM SP2 (*IBM Scalable PowerParallel System*), é uma máquina paralela de memória distribuída, adequada tanto para o processamento de programas seqüências como paralelos que demandem grande potência computacional e/ou grande quantidade de memória. O sistema é composto de nós (processadores com memória e disco próprios) e interligados por um *switch* de alto desempenho que proporciona alta velocidade de comunicação entre os processadores e também entre outras máquinas SP2. Seu sistema flexível de interligação também permite que ele seja conectado a outros tipos de tecnologia de redes tais como: *Ethernet*, FDDI, ATM, etc. A Figura 5.1 mostra um sistema SP2 onde os nós estão interligados a dois servidores de arquivos,

estes servidores estão interligados por uma rede *token-ring*, que por sua vez está interligada ao SP2 através de uma rede *Ethernet* e quatro dos processadores do SP2 estão ligados a três servidores por uma rede FDDI.

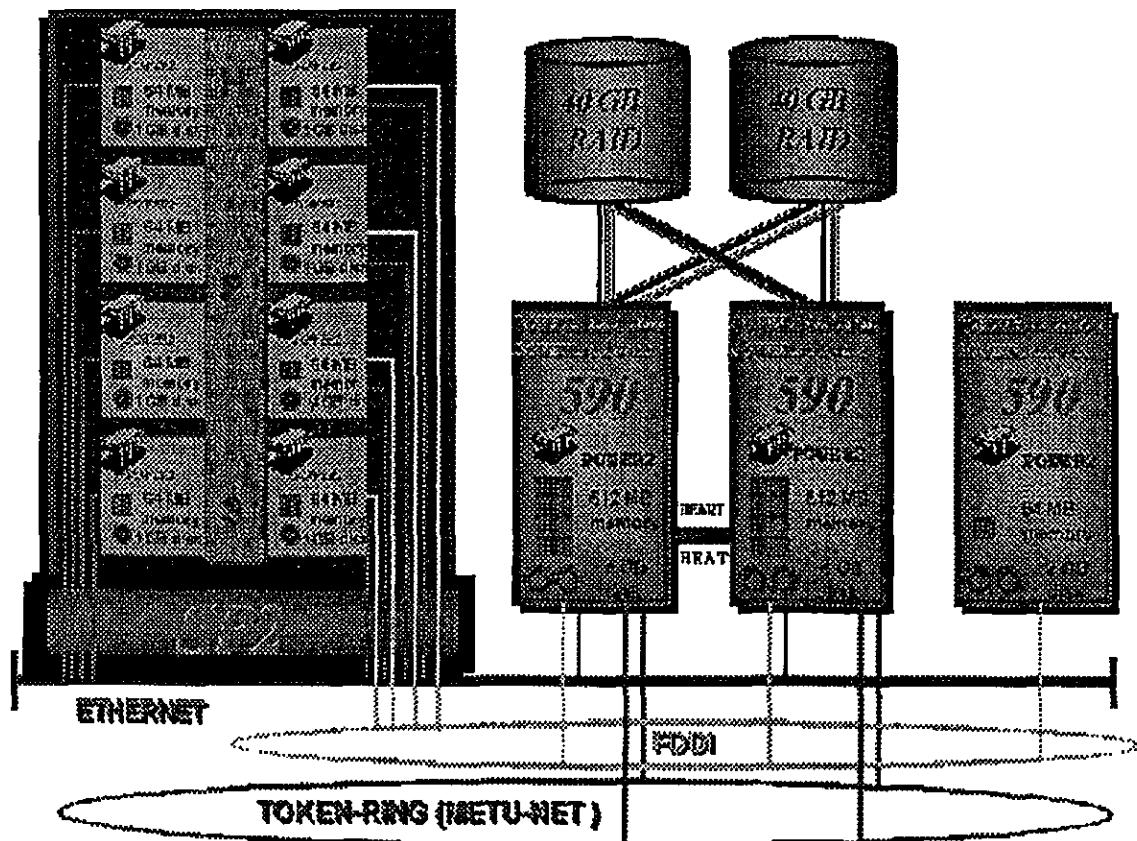
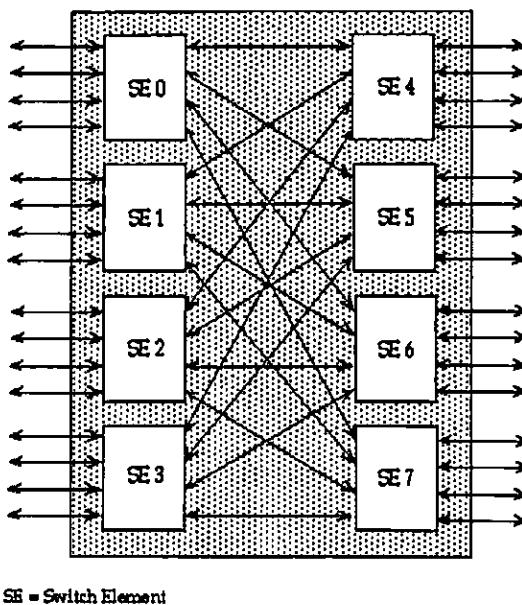


Figura 5.1 - Exemplo de conexão do SP2 com diversos tipos de rede.

A característica mais marcante dos sistemas SP2 é o *switch* de alto desempenho. Ele consiste de dois elementos básicos: uma placa principal e um adaptador de rede. Estes elementos permitem a utilização de passagem de mensagens entre todos os nós do sistema ao mesmo tempo. O SP2 possui ainda um sistema de segurança que impede a perda das mensagens. Como possui diversos caminhos disponíveis para roteamento (como pode ser visto na Figura 5.2) caso um deles apresente defeito ou esteja sendo utilizado, o adaptador se encarrega de encontrar um novo caminho para a mensagem. [Ibm98]

Dois tipos de nós estão disponíveis para o SP2: o do tipo *thin* e do tipo *wide*. Os nós do tipo *thin* apresentam um barramento de 64 bits e o segundo um barramento de 128 bits. A máquina é basicamente vista como uma espécie de estante, chamada de *rack*, onde são colocados os nós. Em cada gaveta é possível colocar dois nós do tipo *thin* ou um nó do tipo *wide*.

Figura 5.2 - *Switch* de alto desempenho.

O IBM SP2 do CISC está equipado com três processadores sendo dois do tipo *thin* e um do tipo *wide* com as seguintes configurações: o do tipo *wide* possui processador POWER2 de 66,7 Mhz, 256 Mbytes de memória RAM, 256 Kbytes de cache, barramento de 128 bits e 9 Gbytes de disco rígido; os do tipo *thin* possuem processador POWER2 de 66,7 Mhz, 128 Mbytes de memória RAM, 128 Kbytes de cache, barramento de 64 bits e 9 Gbytes de disco. Os três nós perfazem um total de 27 Gbytes de disco para aplicações, sistema operacional, etc. [Cis98]

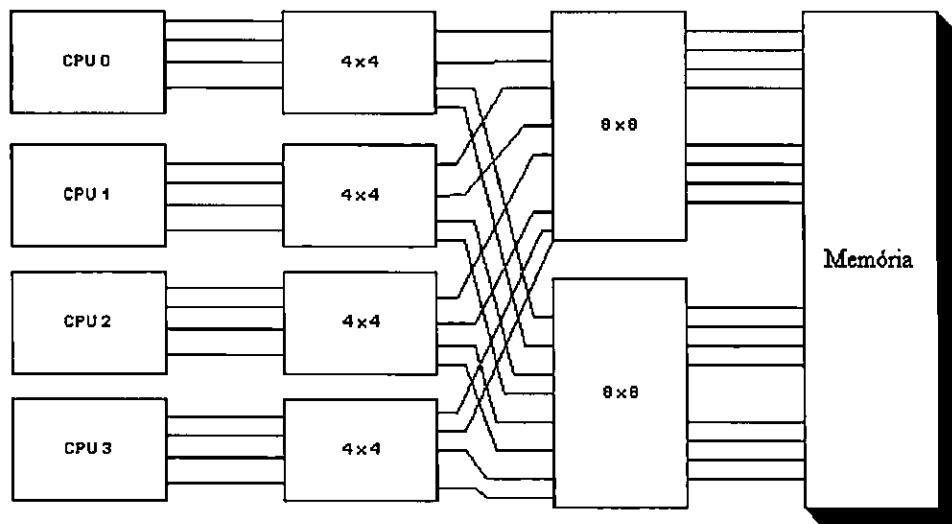
### 5.1.2 Sistema *Cray* de Computadores Vetoriais

O sistema *Cray* de computadores foi criado pela *Silicon Graphics* sendo um dos principais representantes na computação vetorial. O LCCA possui duas máquinas deste tipo sendo que suas características são apresentadas na Tabela 5.1.

Tabela 5.1 - Descrição das máquinas do LCCA.

Nome da máquina	Modelo	Nome	Configuração
boto.lcca.usp.br	J90	dolphin.lcca.usp.br	12 processadores, 1024 MB de memória RAM, 55 GB de disco rígido, pico máximo de operações em ponto flutuante 2.4 Gflop.
dolphin.lcca.usp.br	EL98	boto.lcca.usp.br	6 processadores, 512 MB de memória RAM, 13 GB de disco rígido, pico máximo de operações em ponto flutuante de 2.4 Gflop.

A Figura 5.3 mostra a estrutura interna básica de comunicação entre os processadores [Hwa93].

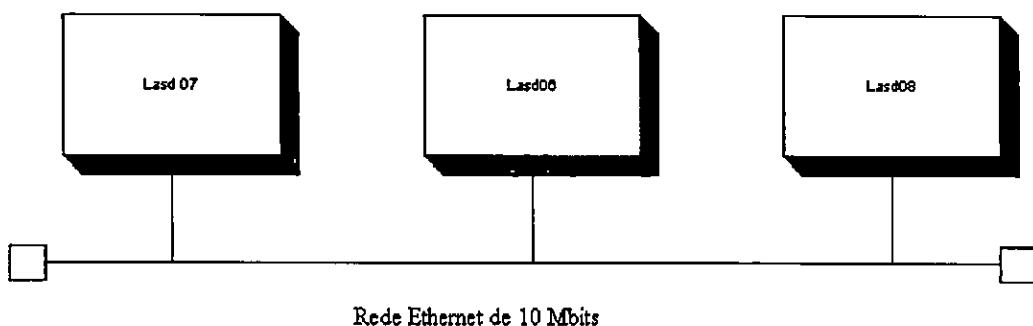


**Figura 5.3 - Estrutura básica de comunicação de um sistema Cray.**

A comunicação entre processadores e memória é feita por uma rede *crossbar* de múltiplos estágios. Como a memória é compartilhada os processadores não precisam estar conectados um com outro. O primeiro estágio da rede é formada por blocos de 4 entradas por 4 saídas e o segundo estágio da rede é formada por blocos de 8 entradas por oito saídas.

### 5.1.3 Sistema Distribuído do LASD-PC<sup>1</sup>

Os computadores utilizados no LASD-PC compõem-se de três máquinas compatíveis com o IBM-PC. As três possuem o sistema operacional *Linux Slackware*, seus respectivos nomes e configurações são apresentados na Tabela 5.2, sendo que todas elas possuem o PVM e o MPI instalados. Todas as máquinas estão interligadas a uma rede *ethernet* de 10Mbits como mostra a Figura 5.4.



**Figura 5.4 - Rede LASD-PC para utilização do PVM e do MPI.**

---

<sup>1</sup> LASD-PC representa a abreviação de Laboratório de Sistemas Distribuídos e Programação Concorrente de Instituto de Ciências Matemática e de Computação da Universidade de São Paulo no campus de São Carlos.

Tabela 5.2 – Máquinas utilizadas no LASD-PC.

Nome	Configurações
Lasd06	LASD 6 - Pentium 100 MHz, 32 Mbytes, 512 de cache L2
Lasd07	LASD 7 - Pentium 200 MMX, 32 MBytes, 512 de cache L2
Lasd08	LASD 8 - Pentium 166 MHz, 32 Mbytes, 512 de cache L2

## 5.2 Software Utilizado

### 5.2.1 PVM, PVMe, MPI para IBM SP2

No IBM SP2 do CISC estão disponíveis quatro tipos de bibliotecas de passagem de mensagens o PVM3, o PVMe, o MPI e o MPL. O PVM3 não é utilizado por funcionar através de uma camada do MPI, oferecendo desta forma, um desempenho bem pobre. O PVM3 e o PVMe utilizam a mesma biblioteca de funções diferindo apenas pelo seu *deamon* e pelo desempenho já que o PVMe utiliza-se do *switch* de alto desempenho e o PVM3 não, por esse motivo ele não foi utilizado neste trabalho. O MPL foi descartado por ser uma biblioteca proprietária da IBM para o SP2 impedindo a portabilidade. A seguir é descrito como os programas são executados neste tipo de arquitetura.

#### Submetendo tarefas (*jobs*) em máquinas SP2 – *LoadLever*

Antes de mais nada é necessário compilar o programa. A Tabela 5.3 apresenta os comandos necessários tanto para o MPI quanto para o PVMe.

Tabela 5.3 - Compilação PVM e MPI no SP2.

Biblioteca	Comando para compilação
MPI	mpCC –o <nome_do_executável> <nome_do_arquivo.cpp>
PVMe – arquivo mestre	xlc -O3 -lpvm3 -lmpci -L/usr/lpp/ssp/css/libus -bl:/usr/lib/pvm3e.exp -o <executável> <arquivo_mestre.cpp>
PVMe – arquivo escravo	xlc -O3 -lpvm3 -lmpci -L/usr/lpp/ssp/css/libus -bl:/usr/lib/pvm3e.exp -o <executável> <arquivo_escravo.cpp>

Como o MPI no SP2 só trabalha com programas com abordagem SPMD sua compilação torna-se mais simples.

Para o PVMe destaca-se a diretiva de compilação *-L/usr/lpp/ssp/css/libus* que indica que o *switch* de alto desempenho não será compartilhado com outros usuários, como será visto mais adiante nos arquivos de comando de tarefas, durante a execução da aplicação. Para testar o

funcionamento da aplicação pode-se usar a diretiva *-L/usr/lpp/ssp/css/libip* que permite o compartilhamento do *switch*.

Depois do processo de compilação, para executar uma tarefa nesse tipo de máquina é necessário submeter os programas a uma fila através de um sistema de gerenciamento de tarefas chamado *LoadLever*. Seu trabalho consiste, basicamente, em alocar todos os recursos necessários a execução de um *job*. As tarefas devem ser submetidas através de um arquivo de comandos chamado de arquivo de comando de tarefas onde serão especificados todos os parâmetros necessários a sua execução. Este arquivo é composto por duas partes: [Ibm95c]

- *Palavras chaves* – passam informações ao *LoadLever* e são precedidas pelos caracteres #@.
- *Comentários* – são simples comentários e são precedidos pelo símbolo #.

A Figura 5.5 mostra um exemplo de arquivo de comandos de tarefas e a descrição de cada um dos comandos.

```
# Programa executável (ambiente POE-Parallel Operation Environment)
# @ executable = /bin/poe

# Seu programa paralelo
# @ arguments = intpar

# Arquivo de saída padrão
# @ output = intpar.out

# Arquivo de erros
# @ error = intpar.err

# Tipo do job
# @ job_type = parallel

# Necessidades: hps_user, modo USUARIO do switch (exclusividade)
#           hps_ip, modo IP do switch
# #@ requirements = (Adapter == "hps_user")
# @ requirements = (Adapter == "hps_ip")

# Número mínimo de processos participantes
# @ min_processors = 3

# Número máximo de processos participantes
# @ max_processors = 3

# Insira a fila para submissão
# @ class = p24h

# Submete na fila
# @ queue
```

Figura 5.5 - Exemplo de arquivo de comandos de tarefas.

O primeiro comando encontrado é o *executable* que deve receber o nome do programa que será executado, geralmente é um programa chamado POE (*Parallel Operation Environment* – Ambiente Paralelo de Operações) que é responsável por gerenciar o ambiente do SP2. Em alguns sistemas é permitido que o nome do programa a ser executado seja colocado diretamente como no caso do LCCA (Laboratório de Computação Científica Avançada), mas no CISC (Centro de Informática de São Carlos) só é permitido a execução de programas via POE. Quando esse sistema é utilizado o comando *arguments* é necessário para especificar o nome do programa que se deseja executar.

Como não existe interface com o usuário, faz-se necessário o uso de mais alguns comandos. O *output* define um arquivo com extensão “*.out*” que conterá as saídas do programa. O comando *error* define um arquivo com extensão “*.err*” que conterá todas as mensagens de erro. Ainda existe o comando *input* que deve conter as entradas necessárias para o programa, como se fosse um usuário digitando. Os arquivos de *input* devem ter a extensão *inp*.

*Job\_type* define se o programa a ser executado é serial ou paralelo. *Requirements* determina se o *switch* de comunicação será disponibilizado somente para o usuário que está executando (*hp\_user*) ou se será compartilhado (*hp\_ip*).

Os comandos *min\_processor* e *max\_processor* definem a quantidade mínima e máxima, respectivamente, de processadores que devem ser alocados para a execução do programa.

O comando *class* define o nome da fila em que o programa será colocado, isso varia de sistema para sistema, no LCCA existem vários tipos de fila, tais como: *short*, *long*, *extra* e *very long*. No CISC são definidas duas filas paralelas chamadas *p24h* e *p168h* e uma fila serial denominada *s168h*. E finalmente, o comando *queue* que envia o programa para execução.

Para programas PVMe, o comando de arquivo de tarefas é semelhante. A diferença é que este tipo de programa não é executado no ambiente POE e sim por um *deamon* próprio do PVMe. Como será visto mais adiante isso impede a depuração através do depurador gráfico.

Além do arquivo de comandos de tarefa, que deve ter a extensão “*.cmd*”, o *LoadLever* apresenta alguns comandos básicos para submissão e controle das tarefas na fila de execução. São eles:

*llsubmit* – submete um programa a fila de execução através do respectivo arquivo de comando (*arquivo.cmd*).

*llcancel* – cancela a execução de um programa.

*llq* – exibe o estado da fila de execução.

*llprior* – aumenta ou diminui a prioridade do processo na fila, entretanto este comando só afeta os processos do mesmo usuário.

*llstatus* – exibe o estado atual das máquinas que compõem o sistema.

*llclass* – exibe quais filas estão disponíveis para utilização.

Para submeter um processo deve-se digitar o comando *llsubmit <arquivo.cmd>*. Este exibirá uma mensagem dizendo que a tarefa foi submetida e um número que corresponde a identificação do *job* no sistema como mostrado a seguir.

*submit: the job “hades01.1432” has been submitted.*

Esse número deve ser utilizado no comando *llcancel* caso se deseje cancelar a execução, ou no comando *llprior* caso seja necessário aumentar a prioridade da tarefa.

Após a submissão pode-se utilizar o comando *llq*, para exibir qual o estado de todas as tarefas que foram submetidas ao sistema. Caso se queira verificar somente o estado dos *jobs* submetidos por um só usuário, usa-se o comando:

*llq -u <nome do usuário>*

Todo esse controle é feito a partir de duas máquinas, uma denominada *front-end* que é responsável por manter os arquivos dos usuários e executar os principais comandos e outra chamada de controle que é responsável por manipular o *status* das tarefas. Por exemplo, quando o comando *llcancel <número da tarefa>* é executado, a máquina *front-end* envia uma mensagem a máquina de controle que se responsabiliza por retirar o *job* da execução ou somente da fila caso este não esteja sendo executado.

Para facilitar a utilização do *Loadlever* existe uma interface gráfica para o programa conforme mostra a Figura 5.6.

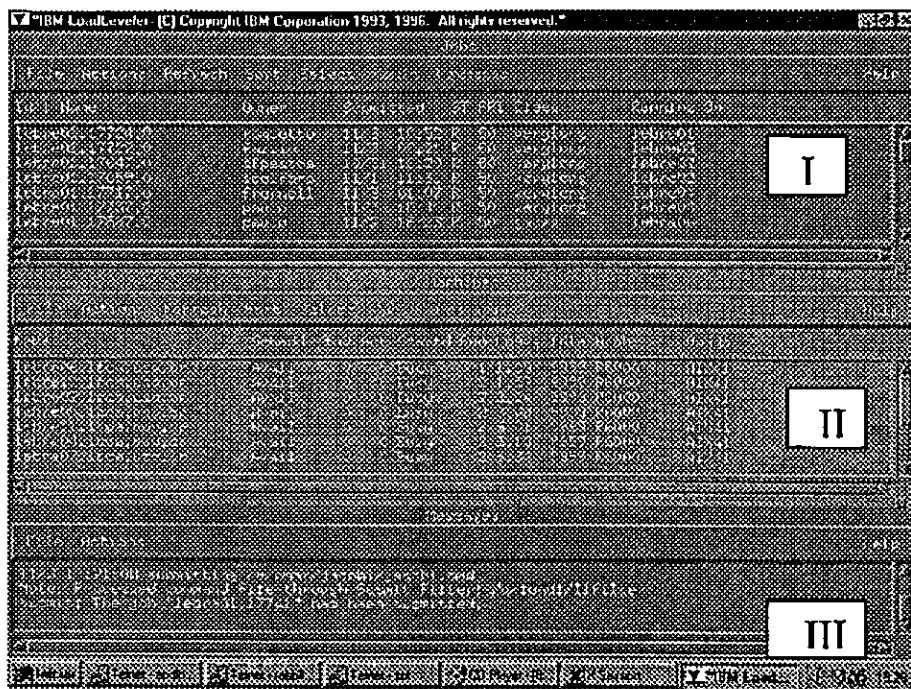
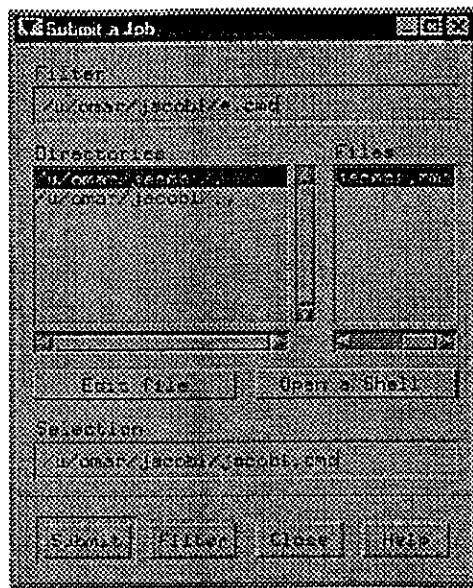


Figura 5.6 - Ambiente gráfico do *loadlever*.

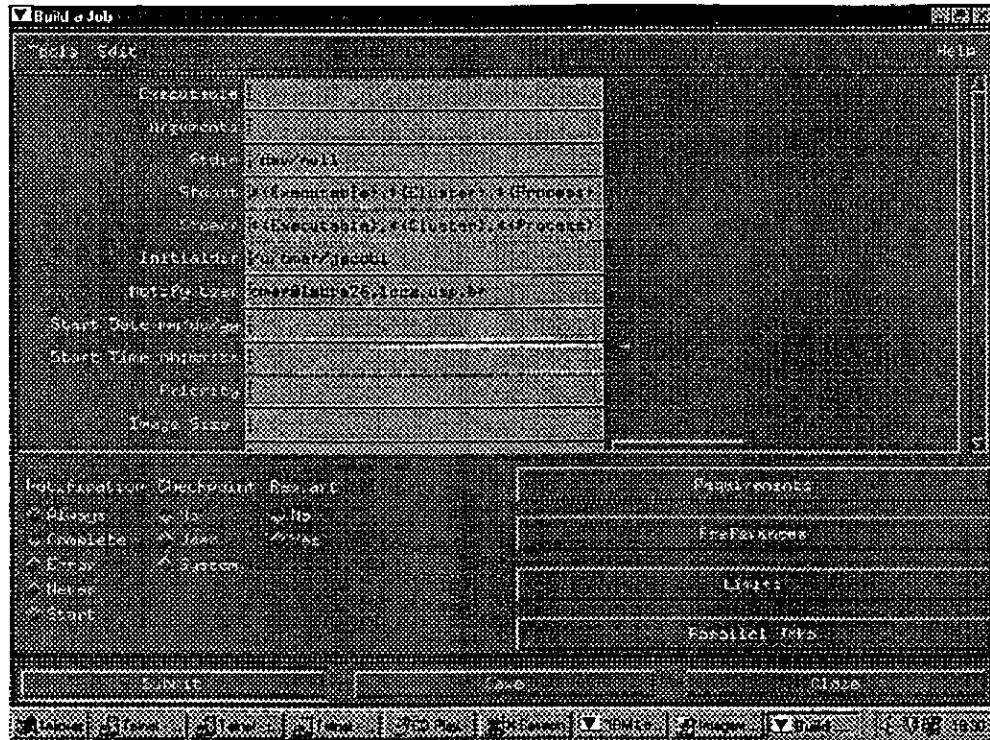
A tela é dividida em três partes. A primeira, apresentada em I da Figura 5.4, mostra todas as tarefas que estão na fila, esteja ou não em execução, correspondendo ao comando *llq*. A parte II, apresenta o estado de cada máquina e substitui o comando *llstatus*. A última parte, identificada por III, exibe as mensagens do sistema.

Para submeter uma tarefa, basta escolher o comando de menu *File* da Figura 5.4 e em seguida escolher o comando *submit-job* o qual vai originar a tela mostrada na Figura 5.7, bastando escolher o arquivo e dar um clique no botão *Submit*.

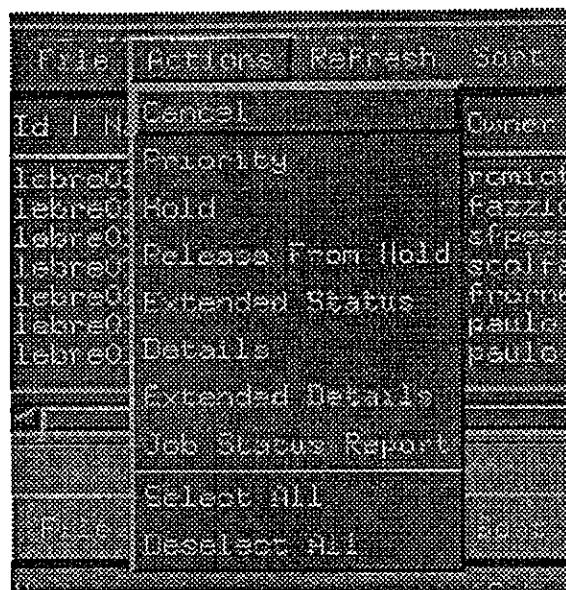


**Figura 5.7-Submetendo uma tarefa.**

Existe ainda outra facilidade que é a de criar de maneira mais simplificada e rápida o arquivo de comando para submissão de tarefas, estando na opção File do menu na sub-opção chamada de *build\_job* como mostra a Figura 5.8. Os demais comandos encontram-se no menu *actions* como pode ser visto na Figura 5.9.



**Figura 5.8 - Construindo um arquivo de comandos.**

Figura 5.9 - Opções do menu *actions*.

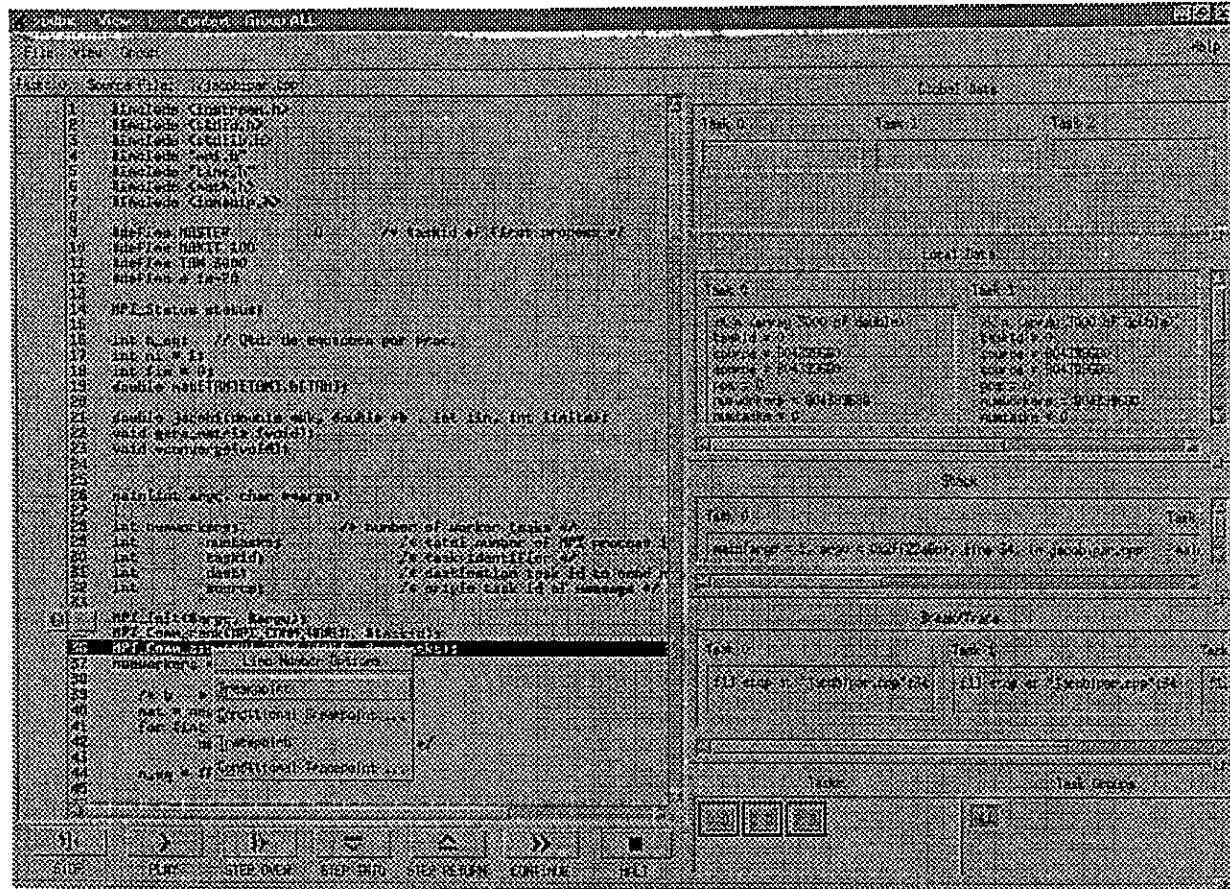
As duas primeiras sub-opções do menu *actions* na figura 5.9 são as mais importantes pois substituem os comandos *llcancel* e *llprior*, respectivamente.

### 5.2.2 O Depurador *xpdbx*

O *xpdbx* é um depurador gráfico específico para máquinas do tipo SP2 que utilizam o sistema POE, isto quer dizer que só é possível depurar programas feitos em plataformas que rodem sobre POE, por exemplo o MPI. Para iniciá-lo é necessário que se esteja conectado em um dos nós de um computador SP2 (não funciona se executado pela máquina *front-end*), compilar o programa desejado com a diretiva “*-g*” e executar o depurador com o comando:

```
xpdbx <nome_executável> -procs n
```

*procs n* indica o número de processadores que serão utilizados na depuração. A Figura 5.10 mostra o *xpdbx* em execução.

Figura 5.10 - Tela do depurador *xpdbx*.

A parte esquerda da tela exibe o código do programa e seus respectivos pontos de parada. Ao pressionar o botão esquerdo do mouse em uma das linhas exibe-se o menu *Line Number Options* cuja a principal opção é a de inserir um ponto de parada no programa (*Breakpoint*).

A parte direita da tela exibe informações sobre as variáveis, sobre a pilha e sobre grupos de processos. A pequena janela denominada de *Tasks* controla qual processo está sendo depurado e exibe seu respectivo código do lado esquerdo do vídeo. Os processos também podem ser agrupados e sua exibição é controlada pela janela *Task Group*.

Logo abaixo da região que exibe o código estão os botões de controle e suas respectivas funções que são resumidas na Tabela 5.4.

Tabela 5.4 - Resumo das funções de cada botão no xpdbx.

Botão	Nome	Função
	Stop	Pára a execução do programa e permite a visualização das variáveis.
	Play	Executa o programa até um ponto de parada definido. Se o ponto não for especificado executa o programa inteiro.
	Step Over	Executa o programa linha por linha.
	Step Into	Executa o programa até o próximo ponto de parada.
	Step Return	Volta o programa até o ponto de parada anterior.
	Continue	Continua a execução do programa.
	Halt	Suspende a execução, esvazia a pilha e zera as variáveis.

## 5.2.2 Sistemas Cray de Compilação e Execução de Programas

Os computadores do sistema *Cray* do LCCA não utilizam bibliotecas de passagem de mensagem, o responsável pelo aumento de desempenho é o compilador que se encarrega de vetorizar o código de acordo com a teoria apresentada no Capítulo 2. Neste contexto, cabe definir algumas diretivas de compilação que foram muito úteis no desenvolvimento deste trabalho e são apresentadas na Tabela 5.5.

Tabela 5.5 - Diretivas utilizadas na compilação vetorial.

Diretiva	Descrição
-O	Utilizada para definir o nível de otimização da compilação.
-g	Cria um executável que pode ser depurado.
-h	<p>Esta diretiva está diretamente relacionada a vetorização, algumas das suas funções são:</p> <ul style="list-style-type: none"> <li>-h report = flags: flags podem ser f,V,s. Quando f é utilizado, o compilador gera um arquivo com todo o processo de compilação. V especifica ao compilador para gerar o relatório somente com os trechos que foram ou não vetorizados. E finalmente s, diz ao compilador para verificar a otimização feita no código escalar.</li> <li>-h vectorN: N especifica o grau de vetorização, que pode ser 0,1,2 ou 3. Zero indica que o compilador não deve executar nenhum tipo de vetorização. Um define um baixo nível de vetorização. Dois determina um nível médio e três especifica um grau alto de vetorização.</li> </ul> <p>OBS: Caso seja necessário impedir a vetorização a diretiva -O não deve ser utilizada pois ela também indica automaticamente níveis de vetorização.</p>

A utilização do compilador é relativamente simples e segue a seguinte sintaxe:

*CC -o <nome do executável> <nome do programa.C> -diretivas*

Existem ainda as diretivas de compilação definidas como `#pragma`. Elas são utilizadas junto ao código de uma aplicação, devem estar sempre antes do laço a ser analisado e instruem o compilador a ignorar dependências de dados quando possível, evitar as pesquisas de laços, forçar a tentativa de vetorização, etc. A Tabela 5.6 [Cra98] resume as principais características da diretiva e a Figura 5.11 mostra um exemplo de sua utilização. Em seguida é apresentado como os programas são submetidos as máquinas do tipo *Cray*.

Tabela 5.6- Diretivas `pragma` e sua utilização.

Diretiva	Função
<code>#pragma _CRI ivdep</code>	Utilizado para ignorar as dependências de dados. Geralmente utilizado quando o laço está trabalhando com variáveis ponteiro ou com matrizes. No caso de matrizes o laço é substituído por uma biblioteca de funções do próprio <i>Cray</i> .
<code>#pragma _CRI novector</code>	Utilizado para ignorar a vetorização do laço a seguir.
<code>#pragma _CRI prefervector</code>	Dentro de laços aninhados esta diretiva especifica qual é o laço que deve ter preferência na vetorização. Deve ser utilizada antes do primeiro laço e a diretiva <code>#pragma _CRI ivdep</code> deve estar precedendo o laço preferencial.

```
#pragma _CRI prefervector
for (i = 0; i < n; i++) {
    #pragma _CRI ivdep
    for (j = 0; j < m; j++)
        a[i] += b[j][i];
```

Figura 5.11 – Exemplo de utilização da diretiva `#pragma`.

### Submetendo programas ao sistema *Cray*

Para executar tarefas num computador *Cray*, mais especificamente nas máquinas disponíveis no LCCA, é necessário utilizar um pequeno *script* como mostra a Figura 5.12.

```
#QSUB -IT 3600
#QSUB -lM 6mw
#QSUB -A omar
cd /hom2/omar/integral
/opt/ctl/bin/CC -o vectors vectors.C -O3 -h report=vf
./vectors
```

Figura 5.12 - Exemplo de um arquivo de submissão no *Cray*.

Na primeira linha é solicitado 3600 segundos de execução que equivale a uma hora de utilização, caso esse limite seja ultrapassado será gerado um arquivo informando o erro ocorrido. A segunda e a terceira linha, alocam 6 Megawords de memória para o usuário *omar*. A quarta linha vai até o diretório onde o programa se encontra. A quinta linha compila o programa gerando o relatório de vetorização. E finalmente a última linha submete o programa a execução. Esse *script* deve ser submetido com o comando *qsub <nome do arquivo>*. Os principais comandos de controle são:

*qsub <arquivo.sh>*: submete um *job* na fila de execução.

*qstat -a*: mostra a fila de execução e seu estado

*qdel -k <numero\_do\_processo>*: retira um *job* da sua execução

Como as máquinas do LCCA possuem 12 (*dolphin*) e 4 (boto) processadores cada uma, é necessário também configurar uma variável de ambiente chamada *NCPUs* e por se tratar de um sistema operacional *Solaris* a configuração é feita com o comando *setenv NCPUs N*, onde *N* é a quantidade de processadores que se deseja utilizar. Para o caso deste trabalho o valor de *N* pode ser 2 ou 3. A próxima seção trata de como o PVM e o MPI estão configurados no LASD-PC.

### 5.2.3 PVM e MPI no LASD-PC

No LASD-PC estão disponíveis o PVM 3.0 nas versões para *Linux* e *Windows 95* e o MPI (*MPICH*) para *Linux*. Neste trabalho só foram utilizadas as versões para *Linux*.

Com relação ao MPI basta compilar o programa com um arquivo *Makefile* que contém todos os caminhos e diretivas necessárias para a compilação. Depois basta executar a aplicação com o comando:

*mpirun -np 3 <arquivo\_executável>*

Para utilizar o PVM, também compila-se o programa utilizando um arquivo *Makefile*. A execução pode-se dar através da utilização do console do PVM. Para iniciar o console basta digitar-se *pvm*, o que fará surgir o *prompt* “*pvm>*”. A partir desse *prompt* os programas são executados. Para iniciar uma aplicação digita-se o seguinte comando:

*spawn -> /caminho/<arquivo\_executavel>*

Se o caminho não for especificado, o arquivo executável deve estar no caminho indicado pela variável PVM\_ARCH que é configurada quando o PVM é instalado. O símbolo de menos seguido pelo símbolo de maior indica ao console que a saída de todos os programas incluindo a saída dos escravos será direcionada para o vídeo.

Os programas podem também ser executados fora do console, sendo necessário neste caso que o *deamon* do PVM esteja rodando. Isso é feito inicializando o console e saindo do mesmo com o comando *quit*. Além disso, dentro do código é necessário uma pequena alteração, a adição do comando *pvm\_addhosts* que tem a seguinte sintaxe:

*info = pvm\_addhosts(hosts, qm, infos)*

*hosts* é uma variável ponteiro que contém o nome das máquinas que serão adicionadas a máquina paralela virtual, no caso deste trabalho todas as aplicações terão nessa variável algo do tipo:

*char \*hosts[]={"lasd06","lasd08"};*

A máquina lasd07 não precisa ser adicionada pois é nela que as tarefas mestres estão sendo executadas. A variável *qm* indica quantas máquinas serão adicionadas e *infos* é um endereço de memória que contém o *status* de cada máquina iniciada. Caso esta linha não seja adicionada, o programa retornara um erro chamado “*pvm\_send error*”. A Tabela 5.7 resume os principais comandos do console do PVM.

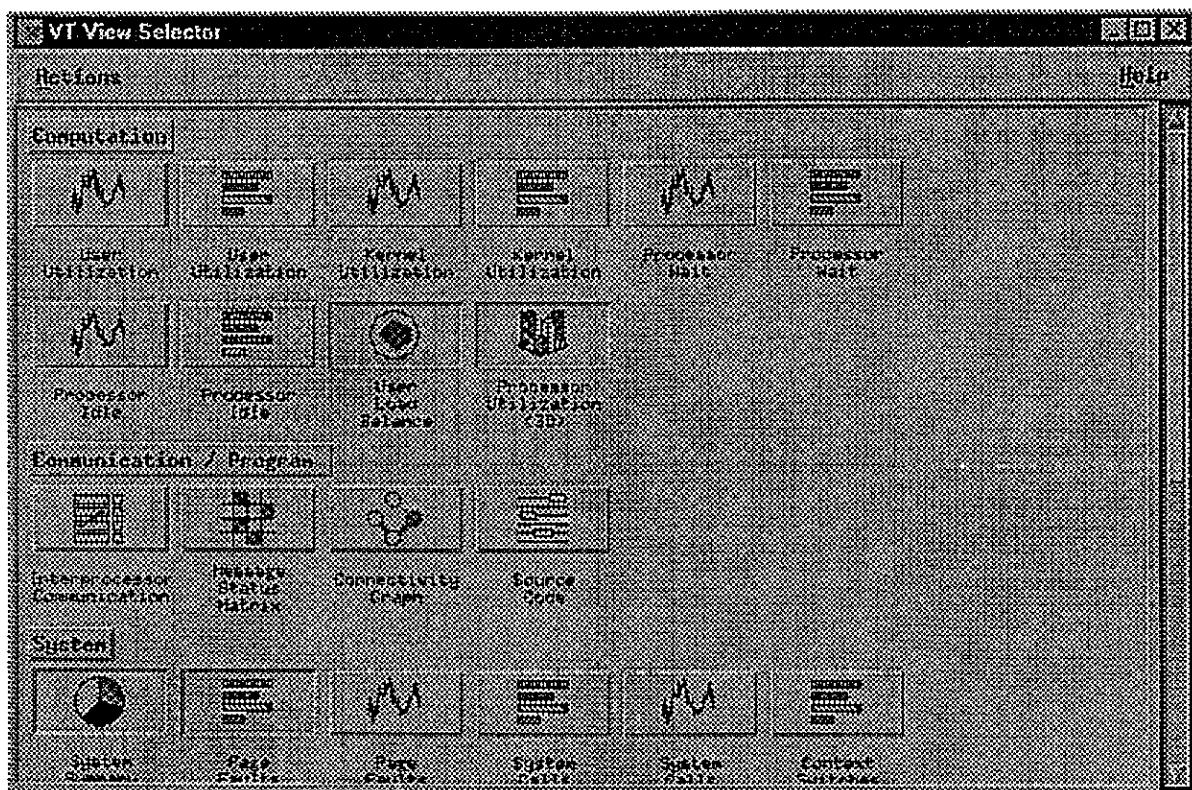
Tabela 5.7- Resumo dos comandos do console PVM.

Comando	Descrição
conf	Exibe quais <i>hosts</i> fazem parte da máquina paralela virtual.
add	Adiciona um <i>host</i> à máquina paralela virtual.
delete	Retira um <i>host</i> da máquina paralela virtual.
spawn	Inicia um programa na máquina em que o console está sendo executado.
quit	Sai do console mantendo o <i>deamon</i> rodando.
halt	Sai do console e retira o <i>deamon</i> do pvm de execução.

### 5.2.4 Visualization Tool (VT)

A *Visualization Tool* é uma ferramenta de apoio a análise de desempenho. Através dela é possível analisar aspectos importantes da programação concorrente tais como o balanceamento de carga, utilização do processador, utilização da máquina pelos usuários, tempo de espera do processador e comunicação entre processos. A Figura 5.13 mostra a tela inicial onde são

escolhidos os modos de exibição de desempenho. No andamento deste trabalho somente foram observados o balanceamento de carga e a utilização do processador.



**Figura 5.13-Tela do seletor de janelas.**

Esta ferramenta pode reproduzir a execução de um programa repetindo todos os eventos que ocorreram desde seu início até seu término. Para realizar esse *replay* são salvos todos os estados do sistema de tempo em tempo. Antes de sua execução são necessários alguns ajustes nas variáveis de sistema. Ao todo são quatro variáveis que são mostradas na Tabela 5.8 e para definí-las é utilizada sempre a mesma sintaxe mostrada abaixo:

*setenv VARIÁVEL valor ou export VARIÁVEL=valor*

**Tabela 5.8– Variáveis de sistema utilizadas pelo VT.**

Variável	Valores	Função
MP_TRACELEVEL	0,1,2,3,9	Especifica o nível de captura de eventos.
MP_SAMPLEFREQ	N	Configura o intervalo de tempo para captura.
MP_PROCS	N	Define a quantidade de processadores utilizados.
RM_POOL	N	Determina quantos processos são executados por processador

O comando *setenv* é utilizado em sistemas *Solaris* e o *export* no IBM-AIX.

A primeira variável a ser configurada é a MP\_TRACELEVEL que define quais tipos de capturas de eventos são possíveis. A Tabela 5.9 mostra os *flags* que podem ser usados e o que cada um deles permite capturar. Geralmente o mais utilizado é o *flag* com valor nove pois com ele tem-se uma gama maior de possibilidades na análise de desempenho.

Tabela 5.9 - *Flags* e eventos que podem ser capturados.

Flag	Passagem de Mensagem	Comunicação Coletiva	Estatísticas do Núcleo	Marcas em Programas
0				
1				X
2			X	X
3	X	X		X
9	X	X	X	X

Ainda seguindo a Tabela 5.8, a segunda variável que deve ser configurada é a MP\_SAMPLEFREQ que define a freqüência com que os eventos devem ser capturados, sendo que o intervalo mais utilizado é o de dez mili-segundos , ou seja, de dez em dez mili-segundos ocorre a captura de um estado do sistema.

A terceira variável que deve ser especificada é a MP\_PROCS que simplesmente determina quantos processadores serão utilizados na execução do programa.

E, finalmente, a variável RM\_POOL especifica quantos processos vão ser iniciados em cada processador da máquina.

Com tudo definido, basta executar o programa que vai gerar um arquivo com o mesmo nome adicionando-se um “.trc”. Por exemplo, se for executado um programa chamado *teste.executavel* será criado um arquivo chamado *teste.executavel.trc* que contém todos os eventos que ocorreram durante sua execução. Nota-se que a execução não pode se dar por filas de *LoadLever* porque isso impedirá a criação do arquivo “.trc”.

Após todo esse processo a tela da Figura 5.14 estará liberada para utilização, ou seja, para reproduzir a execução de um programa. Ela funciona de maneira análoga a um painel de videocassete sendo de fácil utilização.

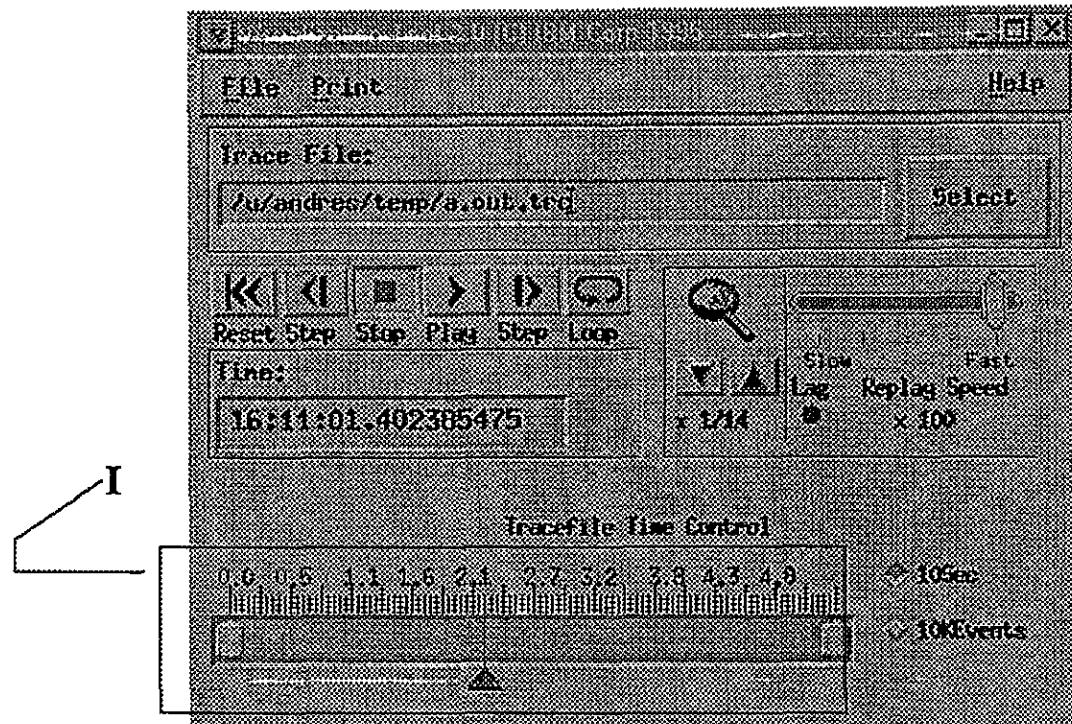


Figura 5.14-Tela de replay.

Destaca-se a parte I, Figura 5.14, onde através dela pode-se posicionar o cursor em forma de triângulo e visualizar o desempenho desejado naquele instante de tempo. A Figura 5.15 mostra as janelas que serão utilizadas neste trabalho, porcentagem de uso do processador (a), balanceamento de carga (b) e comunicação entre processadores (c) quando necessário.

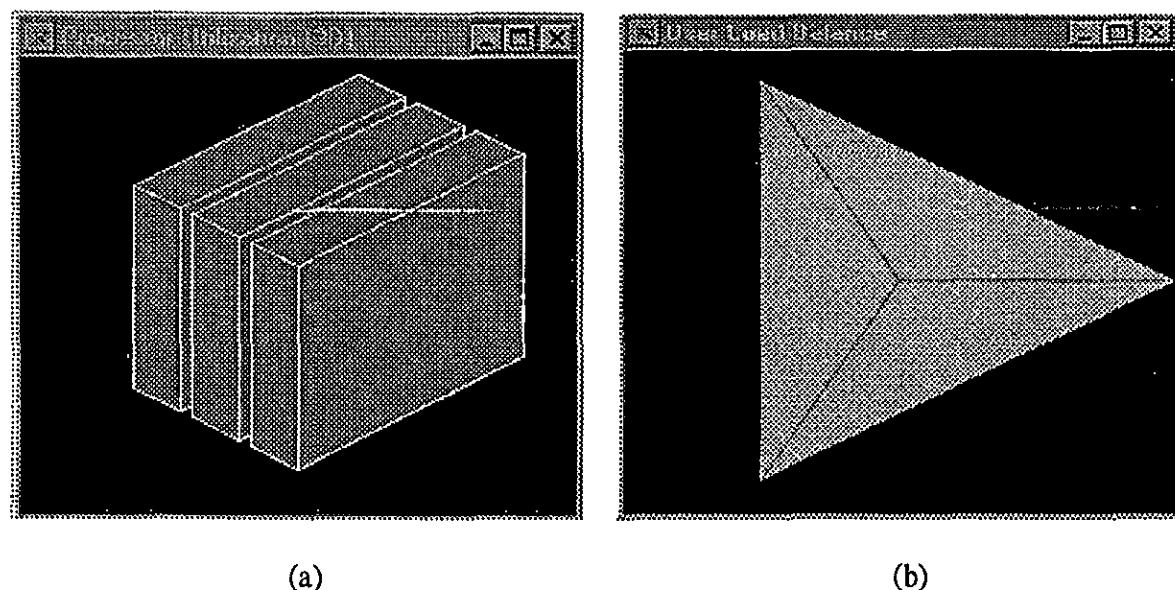
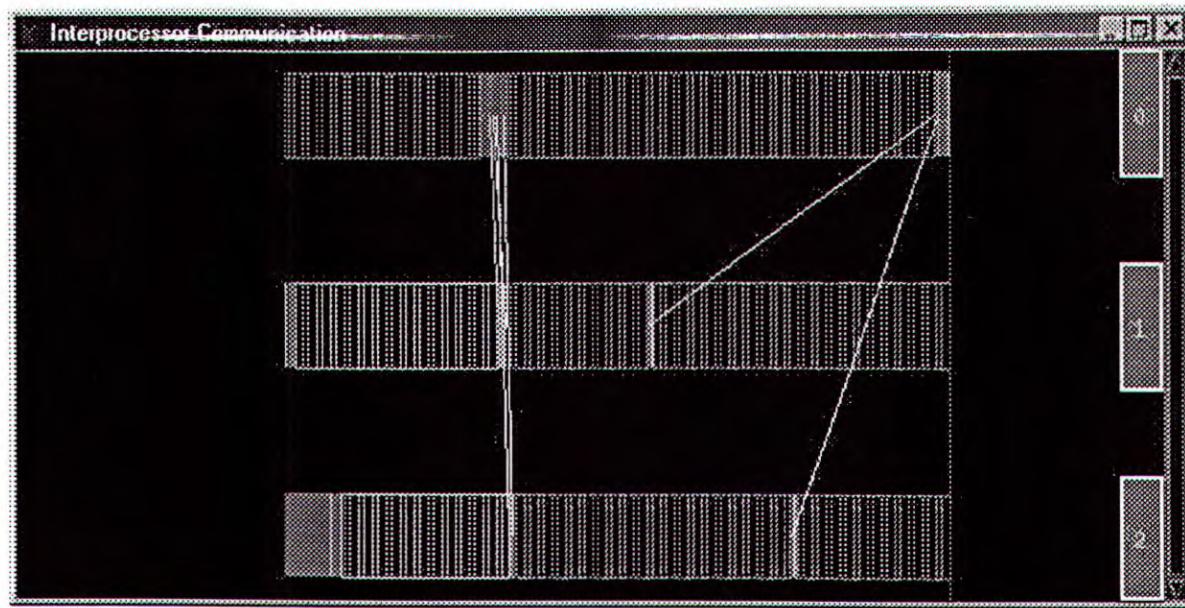


Figura 5.15 - Janelas de desempenho



(c)

**Figura 5.15 - Janelas de desempenho**

### 5.2.5 HPM – Hardware Performance Monitor

Para validar os dados encontrados em  $F_B(N)$ , ou seja, para ter uma idéia aproximada de quantas operações em ponto flutuante foram realizadas pela aplicação, foi utilizado um software chamado *hpm* para sistemas *Cray*. Seu funcionamento é extremamente simples e para ativa-lo basta usar a seguinte sintaxe:

```
hpm -gn <nome do executável>
```

Sendo que n representa o grupo de hardware ao qual o programa será analisado, a Tabela 5.10 mostra a utilização de cada grupo.

**Tabela 5.10 - Utilização de grupos no *hpm*.**

Grupo	Elementos
0	Sumário da execução, este grupo é considerado o padrão.
1	Mantém algumas condições pré-definidas
3	Atividade de memória.
4	Eventos vetoriais e sumário da execução.

Para efetuar uma medição das operações em ponto flutuante foi utilizado o grupo zero (os outros grupos não são permitidos para usuários sem permissão de administradores) sendo que o programa foi compilado sem nenhum grau de vetorização pois o valor procurado é o de  $F_B(N)$ .

qual pode ser aproximado. A Figura 5.16 apresenta um sumário emitido pelo *hpm* para um dos programas desenvolvidos.

Group 0: CPU seconds :	135.41	CP executing :	13540954818
Million inst/sec (MIPS) :	47.95	Instructions :	6492793799
Avg. clock periods/inst :	2.09		
% CP holding issue :	40.59	CP holding issue :	5496496936
Inst.buffer fetches/sec :	0.00M	Inst.buf. fetches:	1338
Floating adds/sec :	6.93M	F.P. adds :	938533815
Floating multiplies/sec :	4.79M	F.P. multiplies :	648891104
Floating reciprocal/sec :	0.00M	F.P. reciprocals :	2
I/O mem. references/sec :	1.37M	I/O references :	186054973
CPU mem. references/sec :	4.76M	CPU references :	645200027
Floating ops/CPU second :	11.72M		

Figura 5.16 – Exemplo de um sumário emitido pelo *hpm*.

## 5.2.6 Totalview

O *totalview* é um poderoso depurador para sistemas *Cray*. Para utilizá-lo basta compilar o programa com a diretiva `-g` e executá-lo com a instrução *totalview <nome\_do\_programa>*, a Figura 5.14 mostra a tela de execução do depurador.

A utilização dos botões encontrados logo acima do código do programa é semelhante a dos botões apresentados na Tabela 5.4. Este depurador é mais fácil de utilizar do que o *xpdbx*, pois a execução é vista numa única tela, sem a necessidade de criar grupos de processos e janelas para visualizar cada processo/processador. A opção mais importante é a *display variable* que é mostrada na Figura 5.18(a). Essa opção abre uma janela exibida na Figura 5.18(b) onde serão inseridas as variáveis que se deseja visualizar, estas devem ser inseridas uma por uma. E finalmente, a Figura 5.18(c) apresenta como os dados das variáveis desejada são exibidos. Não existe um limite para a quantidade de variáveis que se deseja depurar.

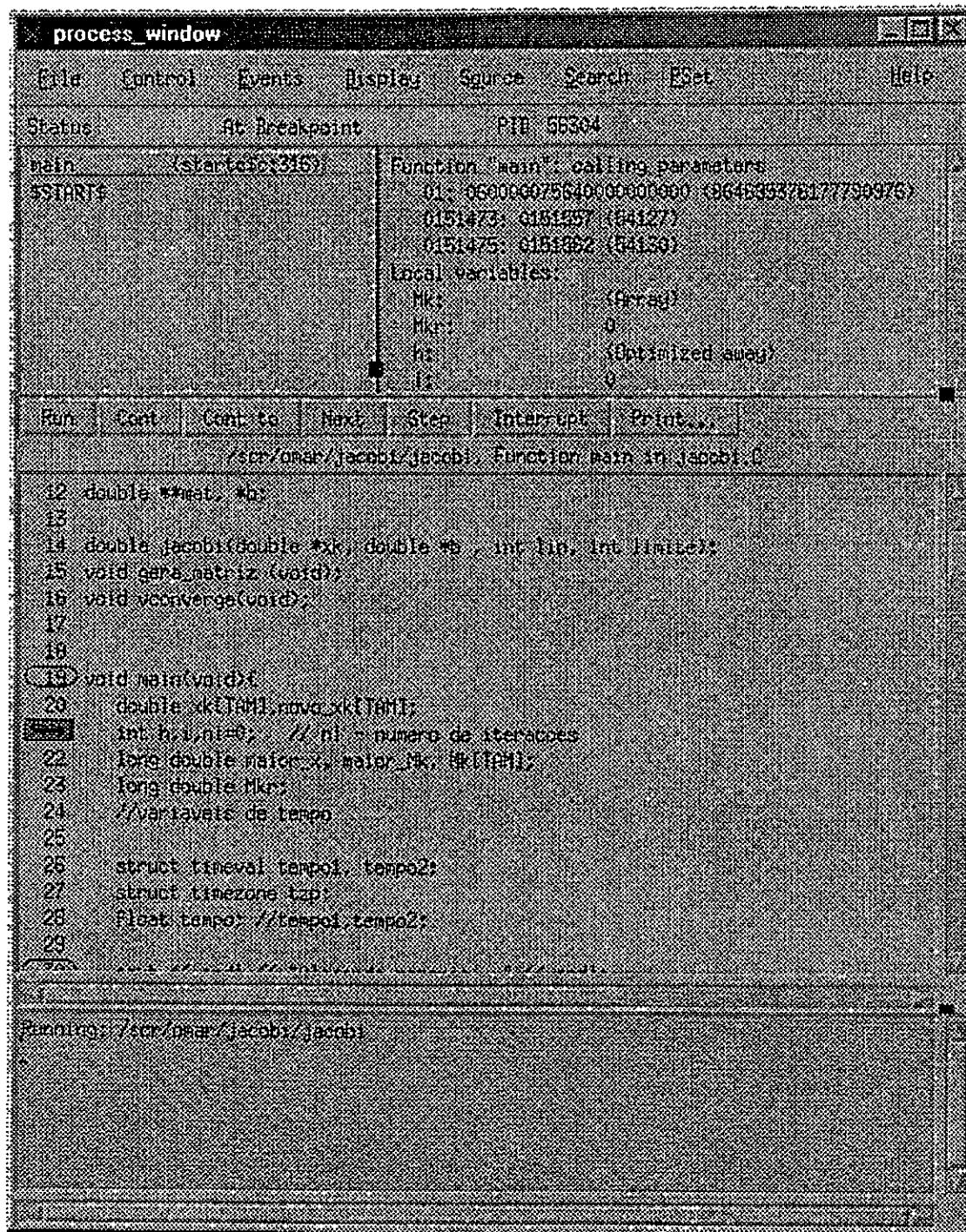
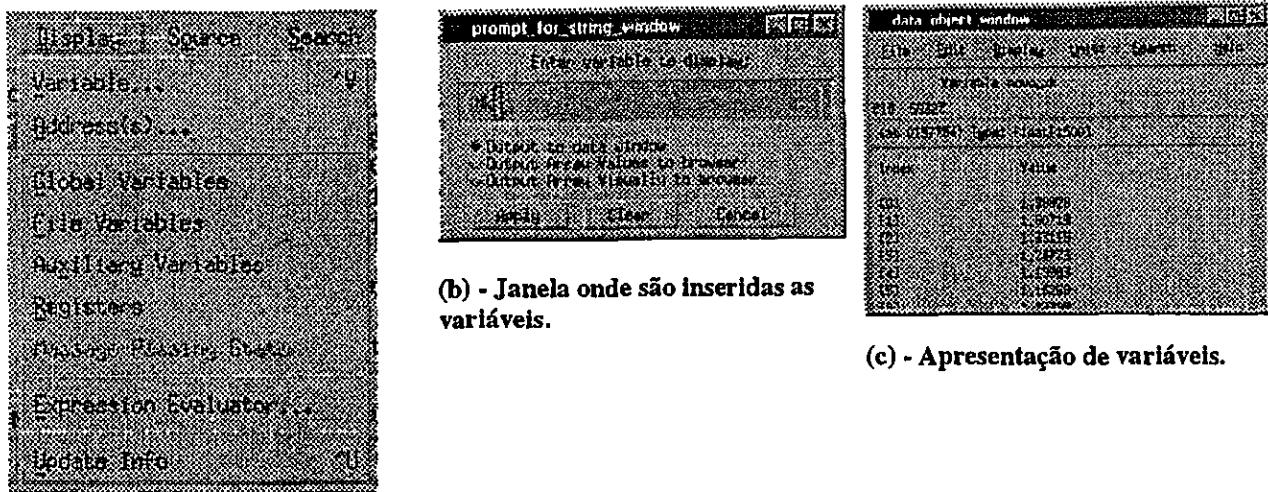


Figura 5.17 - Tela do totalview (depurador) para sistemas Cray.



**Figura 5.18 – Opções mais importantes do totalview.**

### 5.2.7 Proofview

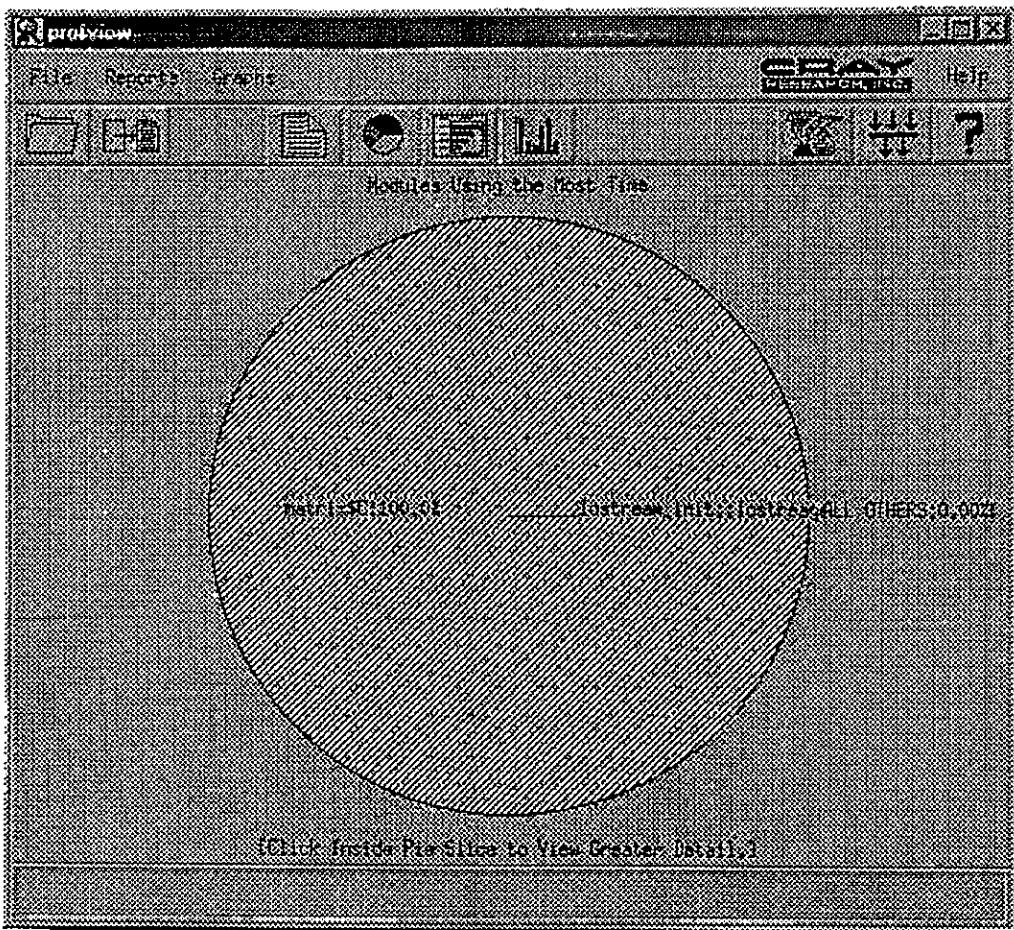
Este é um programa que de maneira geral pode fornecer uma visão aproximada da vetorização alcançada num programa quando os gráficos de ambos (seqüencial e vetorizado) são comparados. Para visualizar as informações desejadas é necessário executar os seguintes comandos:

Compilar o programa com: `CC -Gp -l prof -l sci matrix.C`, que indica que o `profview` será utilizado;

Em seguida é necessário executar o programa;

Executar o comando *prof* com a seguinte sintaxe: *prof -x a.out > prof.raw*, que irá gerar uma saída contendo as informações sobre a execução no arquivo *prof.raw*;

E finalmente executar o *profview* com o seguinte comando: *profview prof.raw*, que exibirá uma tela como mostrada na Figura 5.19.



**Figura 5.19 - Tela do *proofview*.**

Nesta figura vê-se a execução do programa de multiplicação de matrizes sem vetorização, essa informação é dada no centro do gráfico com o nome de `matrix$C:100%`. Quando a vetorização é realizada o gráfico é dividido em mais pedaços como será visto na seção 7.3 do Capítulo 7.

### 5.3 Considerações Finais

Neste capítulo foi descrito todo o ambiente utilizado para o desenvolvimento e execução das aplicações numéricas. Nota-se ainda uma falta considerável de programas. Por exemplo, foi encontrado um monitor de desempenho para sistemas SP2 que só é executado no modo supervisor, caso ocorra um travamento no programa, todo o sistema seria parado e a máquina deveria ser reiniciada inviabilizando sua utilização. Por esse motivo não se tem exatamente a quantidade de operações em ponto flutuante dessa máquina. Além de que falta um depurador gráfico para o PVMe.

Com relação ao *hpm*, geram-se informações sobre toda a execução do programa o que não é interessante no caso de *benchmarking* pois só um trecho do programa deve ser levado em

consideração, além de que as operações de divisão não são registradas pelo monitor devido a sua pouca utilização. Esses fatos levaram o *hpm* a ser utilizado apenas para ter uma noção da quantidade de operações em ponto flutuante utilizadas o que muitas vezes pode se distanciar da quantidade encontrada pelos contadores dentro do código.

O VT apesar de prover uma grande quantidade de informações só foi utilizado para verificar se o balanceamento de carga foi feito adequadamente e se os processadores estão ociosos em algum instante de tempo.

O próximo capítulo apresenta, em detalhes, como os algoritmos mostrados no Capítulo 4 foram paralelizados e implementados em MPI, PVM e nas máquinas *Cray*.

Capítulo  
6

## 6. Aplicações

Neste capítulo apresenta-se como os algoritmos numéricos mostrados no Capítulo 4 foram paralelizados e implementados utilizando as bibliotecas de passagem de mensagem MPI e PVM. Foram utilizados dois e três processadores em todo desenvolvimento. A compilação e execução do *NAS Parallel Benchmark* também é discutida.

### 6.1 Método de Jacobi

Neste método o primeiro passo é criar as matrizes que serão utilizadas, a mesma geração é feita por todos os processadores utilizados. Isto é feito utilizando-se uma função encontrada no LINPACK *benchmark* que sofreu pequenas alterações para produzir apenas números positivos. Em seguida foi criada uma função denominada *vconverge( )* que transforma a matriz gerada numa matriz convergente que segue as regras descritas a seguir:

- Ser *diagonalmente dominante*, isto é, que não exista nenhum número nulo na sua diagonal principal. Esta regra é satisfeita logo na geração das matrizes que não permite a criação de números nulos.

- que a soma dos números em linhas ou colunas seja menor que o valor da sua respectiva diagonal principal como mostrado a seguir:

$$a_{nn} > \sum_{m=1}^n a_{nm}$$

Para satisfazer essa condição, a função `vconverge()` atribui ao elemento diagonal de cada linha o somatório de todos os elementos da linha somando seu resultado com o respectivo elemento da diagonal principal. Por exemplo, se a linha de índice 2 fosse {2, 4, 5, 7} após a execução da função ela corresponderia a {2, 18, 5, 7}. A seguir é dada a demonstração matemática dessa regra.

$$a_{22} > \sum_{m=1}^4 a_{2m}, m \neq 2$$

$$a_{22} > 2 + 5 + 7$$

$$a_{22} > 14$$

Depois da execução da função `vconverge()`

$$a_{22} = 18 \Rightarrow a_{22} > 14$$

Após a geração da matriz e de efetuada a convergência, é criado o vetor b. Nessa operação é feita uma soma de todos os elementos de cada linha e atribuído a respectiva posição do vetor b. Dessa forma as equações geradas sempre terão a solução  $x[1] = x[2] = x[3] = \dots = x[n] = 1$ . A aproximação inicial é dada por um vetor de zeros, e esta pode ser configurada facilmente alterando-se apenas uma linha de código.

A abordagem de programação utilizada foi a SPMD e segue a estrutura da Figura 6.1. A quantidade de equações tem que ser divisível por três e por dois processadores ao mesmo tempo para que possa ser efetuada uma comparação dos *speedups* encontrados.

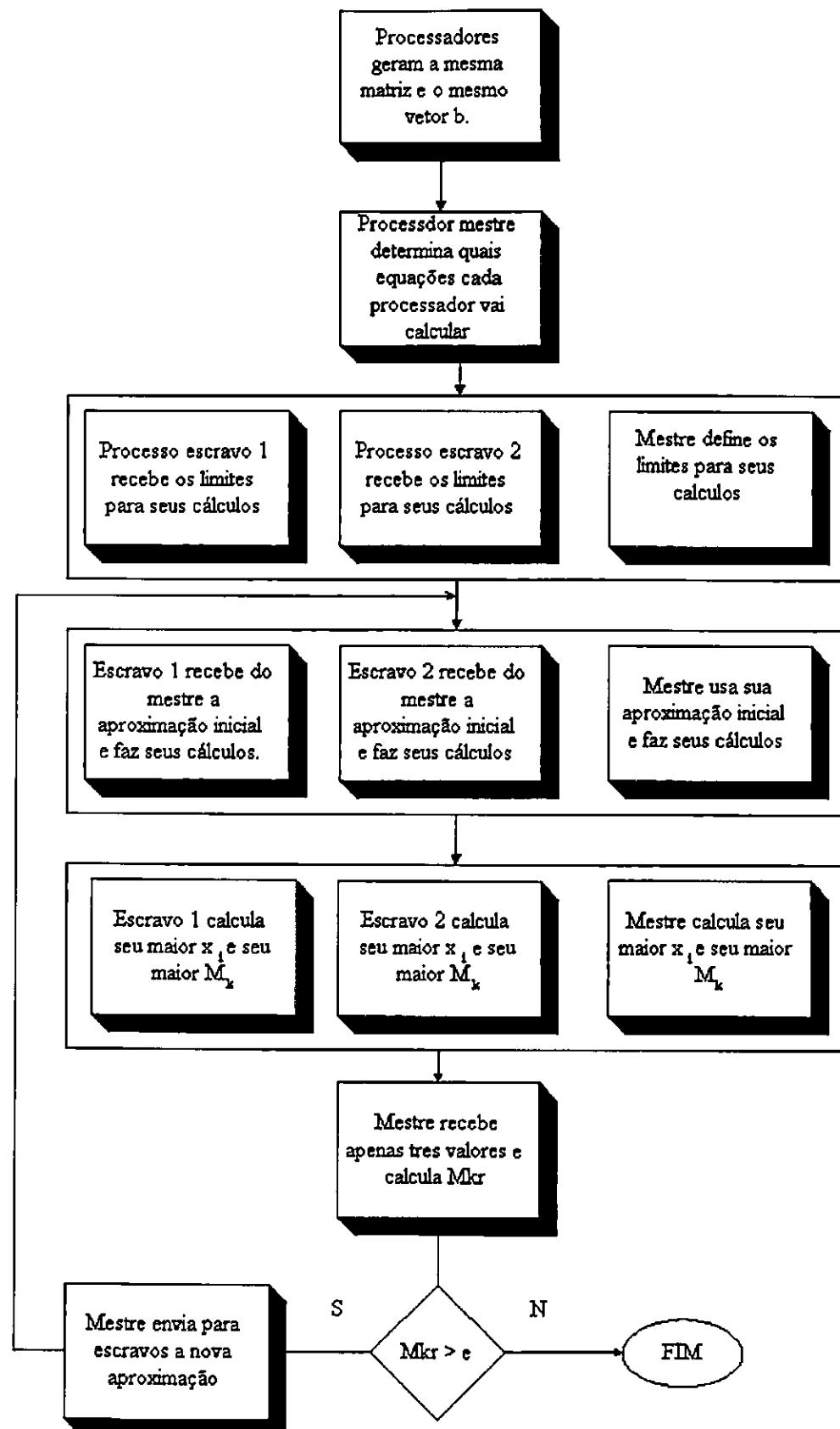


Figura 6.1-Estrutura da paralelização do algoritmo de Jacobi para 3 processadores..

## 6.2 Método dos trapézios composto

Este método soluciona uma integral definida pela aproximação de diversos trapézios. Este algoritmo é considerado pouco preciso quando são utilizadas poucas divisões, por esse motivo, as integrais foram divididas em um número grande e crescente de trapézios.

A paralelização foi realizada através de uma pequena propriedade das integrais, onde, dada uma função qualquer  $f(x)$  que pode ser integrada nos pontos de  $a$  até  $b$ , ela pode ser divida na soma de duas outras integrais definidas como mostrado a seguir:

$$\int_a^b f(x)dx = \int_a^{b/2} f(x)dx + \int_{b/2}^b f(x)dx$$

Por exemplo:

$$\int_0^1 \frac{1}{x} dx = \int_0^{0.5} \frac{1}{x} dx + \int_{0.5}^1 \frac{1}{x} dx$$

Dessa mesma forma essa integral pode ser dividida no somatório de  $n$  outras integrais definidas, onde cada uma delas pode ser calculada por um processo ou processador, como no SP2 do CISC está-se limitado em um processo por processador essa integral foi inicialmente dividida em três e posteriormente somente em duas. A Figura 6.2 mostra um diagrama de paralelização para 3 processadores. Pode-se considerar possível a divisão em um processo tornando o cálculo como de uma integral tradicional, mas os comandos de passagem de mensagem irão interferir no tempo de execução, desta forma nesse caso é aconselhável utilizar o programa seqüencial.

De forma análoga ao método de Jacobi a quantidade de trapézios dentro da área desejada também deve ser divisível por 2 e por 3 ao mesmo tempo. Por exemplo, uma integral que no modo seqüencial foi dividida em 9 milhões de trapézios em três processadores cada um foi responsável por calcular uma integral com 3 milhões de divisões e em 2 processadores cada um usou 4,5 milhões de trapézios.

A região para cálculo, ou seja, a diferença entre  $a$  e  $b$ , também deve apresentar essa propriedade de ser divisível por 2 e por 3 ao mesmo tempo. O intervalo utilizado na execução foi de  $a = 0$  e  $b = 1,2$ . As integrais ficaram divididas da seguinte forma:

$$\int_0^{1,2} f(x)dx = \int_0^{0,6} f(x)dx + \int_{0,6}^{1,2} f(x)dx : \text{para 2 processadores};$$

$$\int_0^{1,2} f(x)dx = \int_0^{0,4} f(x)dx \int_{0,4}^{0,8} f(x)dx \int_{0,8}^{1,2} f(x)dx : \text{para 3 processadores}.$$

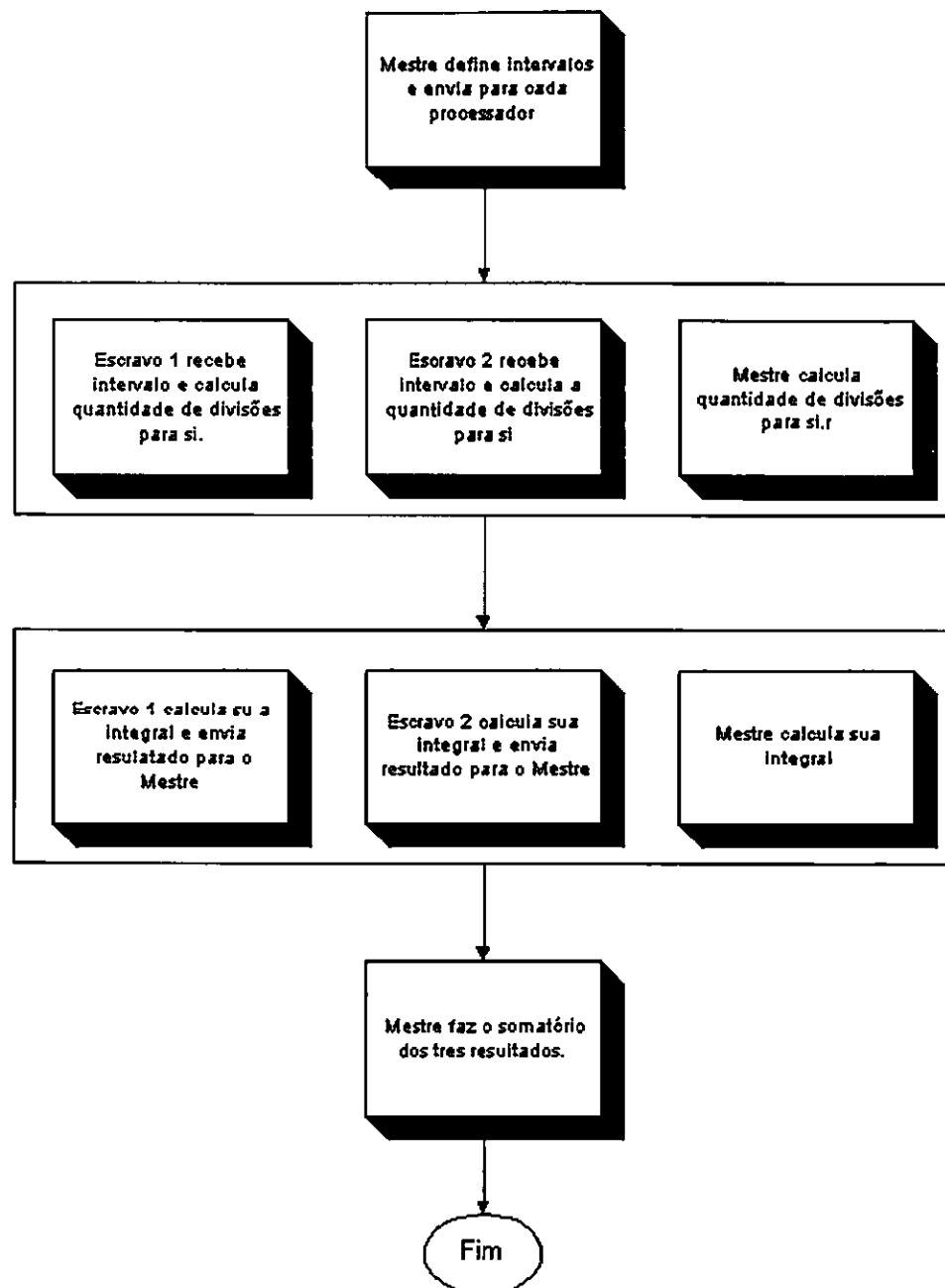


Figura 6.2 - Paralelização do algoritmo do trapézio composto.

## 6.3 Multiplicação de matrizes

O terceiro e último algoritmo desenvolvido foi o de multiplicação paralela de matrizes. Sua escolha deve-se a grande quantidade de operações em ponto flutuante exigida para a solução utilizando o algoritmo tradicional de complexidade  $n^3$ .

A paralelização foi feita da seguinte maneira:

A geração das matrizes A e B é feita de forma semelhante ao do método de Jacobi, criando-se duas matrizes quadradas. O processo mestre gera a matriz A, e ao mesmo tempo os processos escravos geram a mesma matriz B. Ao terminarem o processo, o mestre manda a sua matriz gerada A para os processos escravos, e um dos processos escravos envia a matriz B para o mestre. De posse dos dados o mestre envia aos escravos a posição da linha onde cada processo iniciará a sua parte da multiplicação. Todos os processadores executam a multiplicação incluindo o mestre. No final do processamento os escravos mandam os respectivos pedaços calculados para o mestre. A Figura 6.3 mostra um diagrama da paralelização.

Por exemplo, se as matrizes geradas tiverem o tamanho de 90x90 ocorrerá o seguinte processamento:

- Mestre gera uma matriz A90x90 e escravos geram a matriz B90x90 ao mesmo tempo;
- Mestre envia matriz A para escravos;
- Mestre e escravos calculam a quantidade de linhas que cada um irá trabalhar;
- Escravo 1 envia a matriz B para o mestre;
- Mestre envia a posição zero para escravo 1;
- Mestre envia a posição 30 para escravo 2;
- Escravo 1 realiza a multiplicação da linha 0 até a linha 29;
- Escravo 2 realiza a multiplicação da linha 30 até a linha 59;
- Mestre realiza a multiplicação da linha 60 até a linha 89;
- Escravo 1 envia da linha 0 a linha 29 da matriz C;

- Escravo 2 envia da linha 30 a linha 59 da matriz C;
- Mestre recebe as respectivas linhas e as junta a seu resultado.

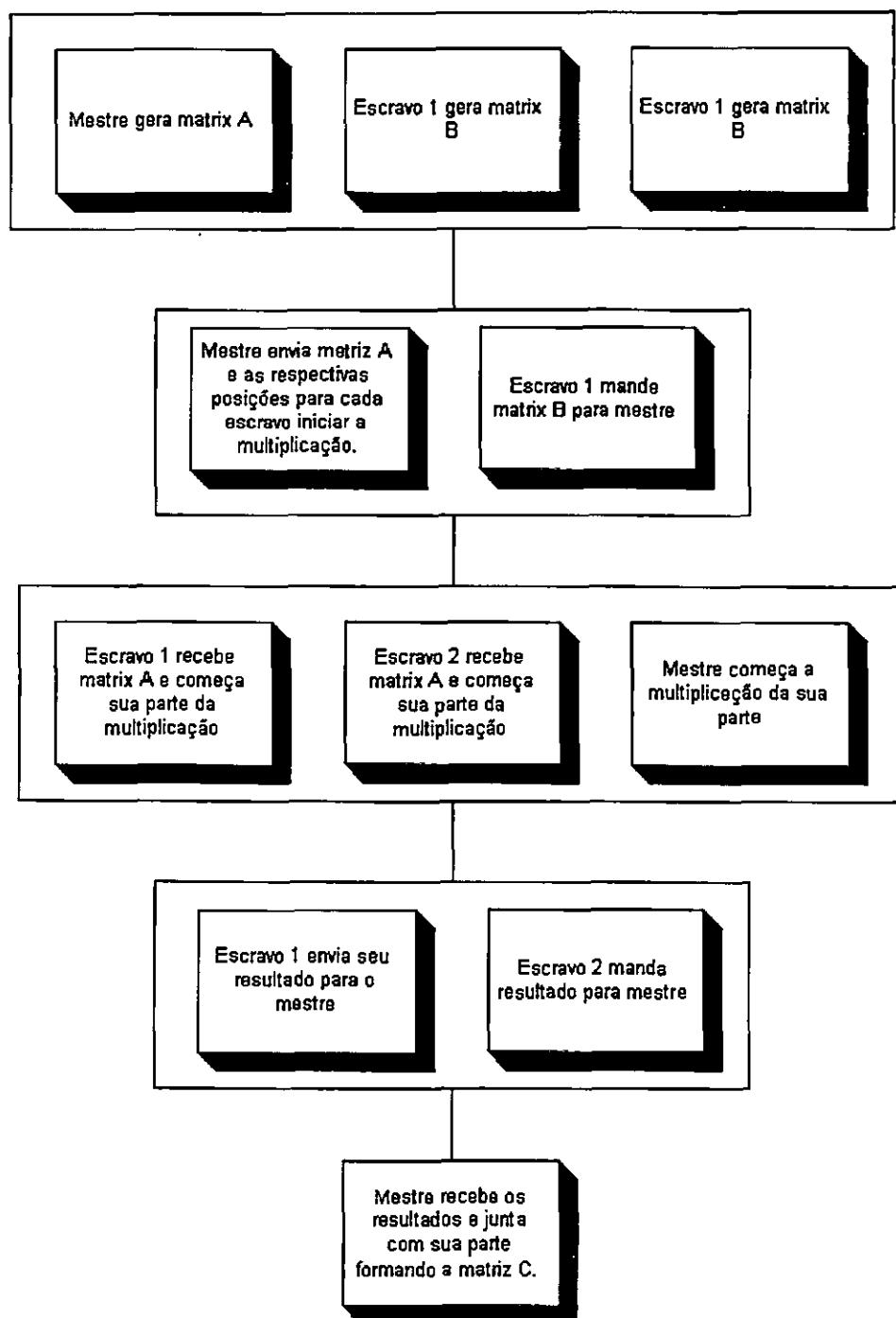


Figura 6.3 - Paralelização do algoritmo de multiplicação de matrizes.

## 6.4 Paralelização vetorial

Para todas as aplicações desenvolvidas no sistema *Cray* foram utilizadas as versões seriais implementadas de modo a se atingir um bom grau de vetorização. Cabe uma observação ao programa de multiplicação de matrizes que teve no seu código uma diretiva *#pragma* adicionada para forçar a vetorização como pode ser visto no pequeno trecho de código da Figura 6.4.

```
#pragma _CRI prefervector
for(k=0; k<TAM; k++) {
    for(i=0; i<TAM; i++) {
        #pragma _CRI ivdep
        for(j=0; j<TAM; j++) {
            C[k][i] += (A[k][j] * B[j][i]);
        }
    }
}
```

Figura 6.4 – Alteração de código no programa de multiplicação de matrizes.

Na próxima seção é apresentada uma descrição mais detalhada sobre a execução do *NAS Parallel Benchmark*.

## 6.5 Compilação e execução do *NAS Parallel Benchmark*

O processo de compilação do *NAS* no *SP2* é relativamente simples, basta seguir o roteiro que acompanha o *benchmark* no diretório */NPB2.3/Doc*. O primeiro passo é criar um arquivo de nome *make.def* onde são definidos os parâmetros de compilação. Essa criação é feita geralmente através da cópia de um arquivo chamado *make.def.template* que se encontra no diretório */config* do *NAS*.

Caso a máquina a ser utilizada seja muito específica, como por exemplo, uma *Fujitsu*, existem arquivos no diretório */config/NAS.samples* já prontos para serem copiados.

Em seguida é necessário editar o arquivo *make.def*. No caso deste trabalho só foi necessário a configuração de algumas variáveis as quais definem quais compiladores e link editores serão utilizados como mostrado a seguir:

MPIF77 = mp xl f

FLINK = mp xl f

MPICC = mp CC

CLINK = mp CC

Existem também outras variáveis que podem ser configuradas como as mostradas a seguir:

FFLAGS – define opções de compilação tais como: o parâmetro “O” que otimiza o código; ou o parâmetro “g” que diz ao compilador para gerar um código que pode depurado através de ferramentas auxiliares.

CLINKFLAGS e FLINKFLAGS – especifica os *flags* utilizados pelo link editor para C e Fortran respectivamente.

Para compilar o *benchmark* desejado usa-se o comando:

*make <benchmark> NCLASS=<classe> NPROCS=<numero de processadores>*

onde *benchmark* especifica qual deles será compilado, por exemplo, *ft* corresponde ao *benchmark* que utiliza *Fast Fourier Transformation* para solucionar equações diferenciais (o significado das abreviações estão na seção 3.5.2 do Capítulo 3), o parâmetro NCLASS especifica qual classe de *benchmark* será utilizado (A, B, C, S ou W), a Tabela 6.1 mostra a equivalência de cada uma das classes. O parâmetro NPROCS define o número de processadores que serão utilizado na execução do *benchmark*. O processo de compilação gera um arquivo executável com o seguinte nome:

*<nome do benchmark>.classe.num\_de\_processadores*

dessa forma, o comando *make sp CLASS=A NPROCS=1*, gera um arquivo executável de *nome sp.A.I.*

**Tabela 6.1 – Classes e seus objetivos**

Classe	Objetivo
S	Exemplificar os resultados do <i>benchmark</i> .
W	Para execução em <i>Workstations</i> .
A	Para pequenos sistemas paralelos.
B	Para máquinas paralelas de médio porte.
C	Para máquinas paralelas de grande porte (acima 128 processadores)

Existem algumas restrições para o processo de compilação. O valor de NPROCS para os *benchmarks* de núcleo tem que ser obrigatoriamente uma potência de dois , ou seja, a quantidade de processadores utilizados deve ser 1, 2, 4, 8 e assim por diante. No caso das simulações a quantidade de processadores tem que ser o quadrado de um número, isto é, 1, 4, 9, 16, etc.

Caso seja necessário gerar vários arquivos executáveis de um só vez, é possível a utilização de um arquivo chamado *suite.def*, que também encontra-se no diretório */config*, contendo as especificações desejadas. As linhas a seguir mostram um exemplo de arquivo *suite.def*.

```
sp    A    4
sp    B    9
ft    W    2
mg    C    8
mg    A    16
```

Este arquivo deverá ser compilado com o comando *make suite.def* que gerará os seguintes arquivos executáveis:

```
sp.A.4
sp.B.9
ft.W.2
mg.C.8
mg.A.16
```

Todos os arquivos gerados pelo utilitário *make* são armazenados no diretório */bin*, a não ser, que no *Makefile* seja definido um outro diretório.

### Coleta de dados

Existem duas formas possíveis de se executar os *benchmarks* do *NAS Parallel Benchmark* dentro do SP2. A primeira é a forma tradicional de execução de programas paralelos, onde um código fonte é compilado e depois submetido a fila de execução através do comando *llsubmit* do *LoadLever*.

A segunda forma de executar um *benchmark* é estando conectado diretamente em um dos nós do SP2, nesse caso é utilizado o seguinte comando:

*poe <nome do programa> -procs <número de processadores>*

Em qualquer uma das formas é necessário assegurar-se que nenhum dos nós está sendo utilizado por outro usuário para não afetar o desempenho na execução dos *benchmarks*. Devido a grande quantidade de recursos que os mesmos exigem em determinadas classes, alguns deles não puderam ser executados por falta de memória. A Tabela 6.2 a seguir mostra os *benchmarks* e as classes que puderam ser executadas.

**Tabela 6.2 - Benchmarks executados com as respectivas classes**

Benchmark	Classes
CG	S,W,A
EP	S,W,A,B
FT	S,W
MG	S,W

O IS não foi incluído nesta tabela pois o mesmo não trabalha com nenhum algoritmo numérico e sim com um algoritmo de ordenação de vetores. A tabela 6.2 mostra os resultados obtidos na execução dos mesmos.

**Tabela 6.3 - Resultados encontrados na execução do NAS Parallel Benchmark**

Bench.	Classe	NP	Erro	Tamanho	Nº de iter.	Tempo de Execução(s)	Mflop/s
MG	S	2	$0.4861 \times 10^{-16}$	32x32x32	4	0,15	50,18
MG	W	2	$0.1624 \times 10^{-18}$	64x64x64	40	7,9	76,22
EP	S	2	-	33554432	0	25,10	1,34
EP	W	2	-	67108864	0	50,18	1,34
EP	A	2	-	536870912	0	447,86	1,20
EP	B	2	-	2147483648	0	1612,32	1,33
FT	S	2	-	64x64x64	6	2,72	65,13
FT	W	2	-	128x128x32	6	5,98	62,30
CG	S	2	$0.8888 \times 10^{-14}$	1400	15	1,47	45,40
CG	W	2	0.0	7000	15	7,64	55,02
CG	A	2	$0.8989 \times 10^{-12}$	14000	15	23,77	62,96

A primeira coluna da Tabela 6.3 indica qual das diversas opções de *benchmark* oferecidas pelo NAS foi utilizado. A segunda refere-se as classes de execução conforme apresentado na Tabela

6.1. Como o NAS só funciona com 2º processadores e o SP2 do CISC contém 3 processadores, a coluna 3 indica que todos os *benchmarks* foram executado em 2 processadores. A quarta coluna indica o erro utilizado na respectiva classe quando o *benchmark* utiliza um algoritmo iterativo. A coluna *Tamanho* indica o da tamanho do problema, ou seja, da estrutura utilizada no cálculo (matrizes e vetores). Quando o algoritmo é numérico a quinta coluna indica a quantidade de iterações realizadas. As duas últimas colunas indicam o tempo de execução em segundos de cada *benchmark* seguido do desempenho do *benchmark* ( $R_B(N,p)$ ).

No *benchmark* MG apresentado na primeira e segunda linhas, o tamanho do problema praticamente dobra, o numero de iteração é 10 vezes maior e o tempo de execução que teoricamente deveria também ser multiplicado por 10 devido a quantidade de iterações é multiplicado praticamente por 52. Isso ocorre pelo fato do *benchmark* executar muita comunicação e testar a capacidade de enviar grandes e pequenas mensagens. A quantidade Mflop/s aumenta devido ao aumento do número de iterações.

O *benchmark* EP, não realiza comunicação durante o processo de cálculo, isso induz que o tempo de execução está diretamente relacionado ao tamanho do problema, sendo que o desempenho de *benchmark* deve continuar constante.

Em todos os *benchmarks* a quantidade de Mflop/s é alterada devido ao contador de operações em ponto flutuante estar diretamente ligado ao tamanho do problema. O erro mostrado na quarta coluna para o *benchmark* CG não é o erro desejado para que ocorra a finalização das iterações e sim o erro encontrado após a execução das 15 iterações. Esse número é constante para todo os tamanhos de problema.

## 6.6 Considerações finais

Dentre os problemas encontrados, destaca-se encontrar as condições de corrida que aconteciam durante a execução. Para ajudar nessa tarefa foi utilizado o depurador gráfico *xpdbx* apresentado no Capítulo 5.

A Tabela 6.1 de classes é muito vaga e não define adequadamente o tamanho dos problemas, ou seja, esse tamanho deveria ser definido com a quantidade mínima de memória RAM e cache para a execução de todos os programas.

Quanto aos demais *benchmarks* eles continham muito pouca informação para sua compilação e não tinham arquivos *make.def* específicos para o SP2 nem gerais como foi o caso do *NAS Parallel Benchmark*.

O próximo capítulo apresenta em detalhes todos os resultados encontrados na execução dos programas desenvolvidos em todas as plataformas já descritas.

A graphic element consisting of a black square containing the word "Capítulo" in white serif font at the top and the number "7" in a large, bold, white sans-serif font below it.

## 7. Resultados

Este capítulo descreve os resultados encontrados nos três ambientes utilizados: SP2, *Cray* e Sistema Distribuído (LASD-PC). No SP2, os programas seqüenciais são executados num nó chamado *cronus* que tem as mesmas características do nó *wide* (*hades03*). No sistema *Cray*, a versão seqüencial foi executada sem vetorização. E no Sistema Distribuído a execução serial deu-se na máquina mais rápida da rede a *lasd07* (*Pentium* 200 Mhz). Todos os aplicativos foram rodados várias vezes e os tempos de execução considerados foram os mais altos obtidos.

A primeira seção deste capítulo apresenta o balanceamento de carga e a porcentagem de utilização de CPU para cada aplicação, observa-se que esta característica vale apenas para programas que rodam sobre o ambiente POE, como por exemplo o MPI. Apesar disso, o balanceamento de carga e a utilização de CPU podem ser estendidas para os programas em PVM já que o algoritmo e a forma como são enviadas as mensagens são as mesmas. Estas informações foram obtidas através da execução da ferramenta *Visualization Toolkit*.

A segunda seção apresenta os resultados obtidos na execução dos algoritmos. O desempenho do *benchmark* ( $R_B(N,p)$ ) só será utilizado no final de cada subseção para comparar as arquiteturas. O desempenho temporal ( $R_T(N,p)$ ) é utilizado para fazer uma comparação entre algoritmos do mesmo tipo, ou seja, comparar algoritmos que resolvem o mesmo problema de forma diferente. Pelo fato de que neste trabalho não foram implementados algoritmos diferentes para o mesmo problema, o desempenho temporal só é utilizado de forma ilustrativa para demonstrar a teoria apresentada no Capítulo 3.

Nos resultados da arquitetura *Cray*, a máquina *boto* (configuração apresentada no Capítulo 5) não é considerada por ter um desempenho muito pobre. Os programa seqüenciais demoram muito para terminar sua execução e a maioria das vezes extrapolam o tempo permitido de execução. Mesmo nos algoritmos que foram vetorizados o tempo de execução as vezes é ultrapassado.

## 7.1 Balanceamento de Carga e Utilização do Processador

Em todas as aplicações desenvolvidas para o SP2 utilizando a biblioteca de passagem de mensagem MPI houve 100% de utilização de processador, ou seja, nenhum processador ficou ocioso tempo suficiente capaz de ser detectado pelo VT. A Figura 7.1 mostra o gráfico de utilização para todas as aplicações.

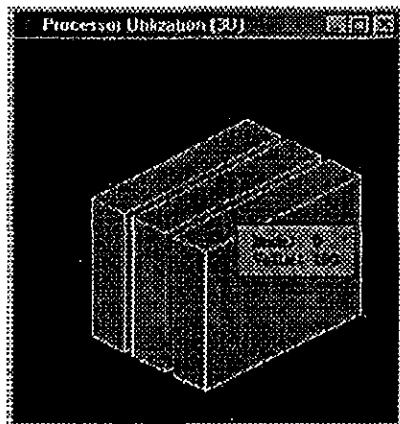


Figura 7.1 - Utilização de processador por todas as aplicações.

Em seguida foi testado o balanceamento de carga para cada aplicação. A Figura 7.2 (a) mostra um instante em que o balanceamento de carga tende para um só processador. O triângulo mais escuro indica que o *Nó 0* está trabalhando e os demais esperando. Este instante é muito rápido e

foi de difícil captura já que ele ocorre quando outros processadores estão esperando por uma mensagem.

A Figura 7.2 (b) mostra um quadro de informações sobre o balanceamento de carga que será exibido juntamente com a Figura 7.2 (a). Neste quadro, o primeiro dado é o instante de tempo em que os dados foram obtidos. Esse valor não é relevante pois é baseado no intervalo configurado na variável MP\_SAMPLEFREQ apresentada no Capítulo 5. O *Node*, indica qual nó está sendo avaliado (neste exemplo o nó 2). *Instantaneous* indica a porcentagem de execução do programa quando a informação foi obtida, o valor 100 indica que as informações foram obtidas no término de execução. E *Average*, indica a média de balanceamento de carga, neste caso, a média 99 indica que o processador teve uma carga média de 99%.

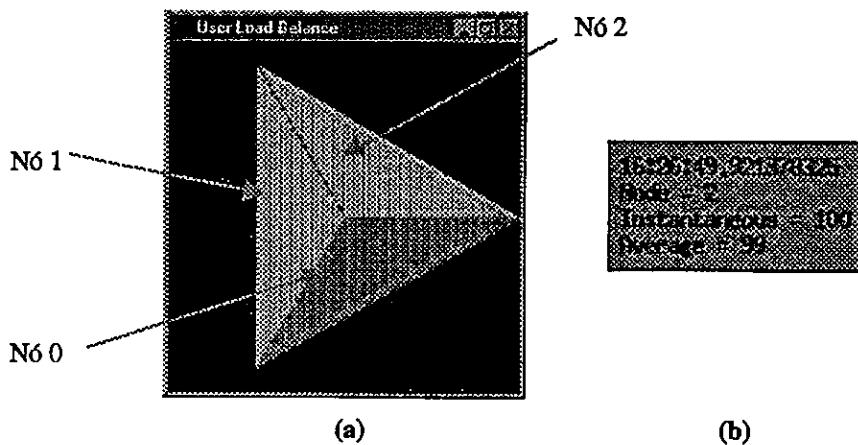
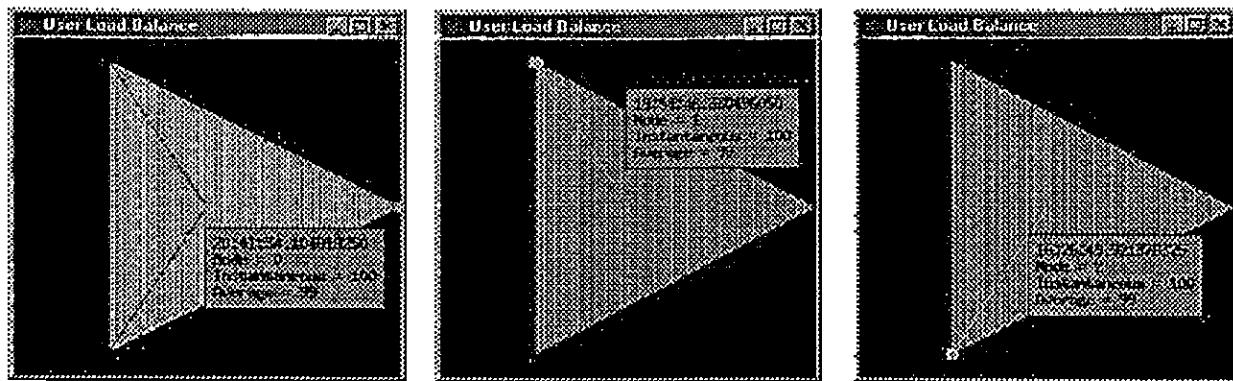
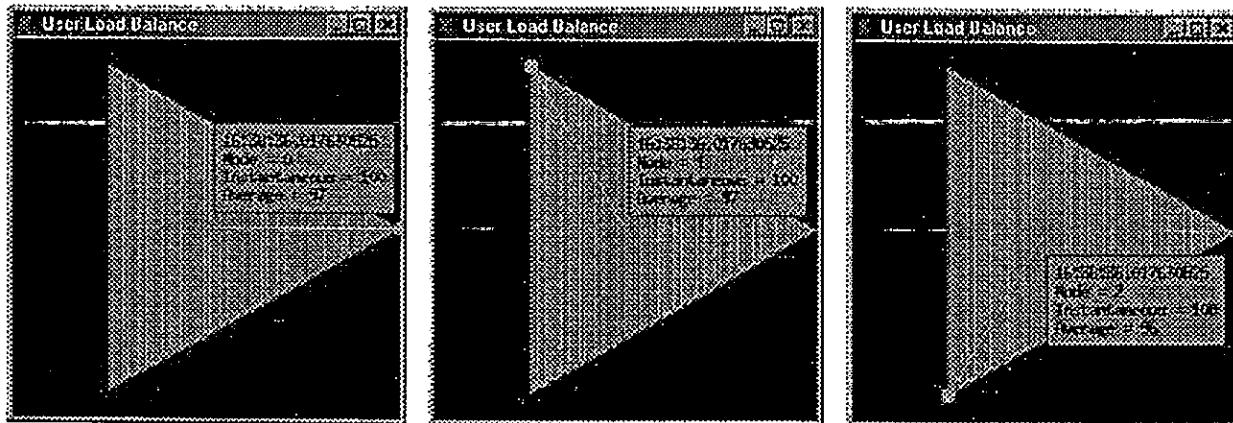


Figura 7.2 - Balanceamento de carga onde só um processador está trabalhando.

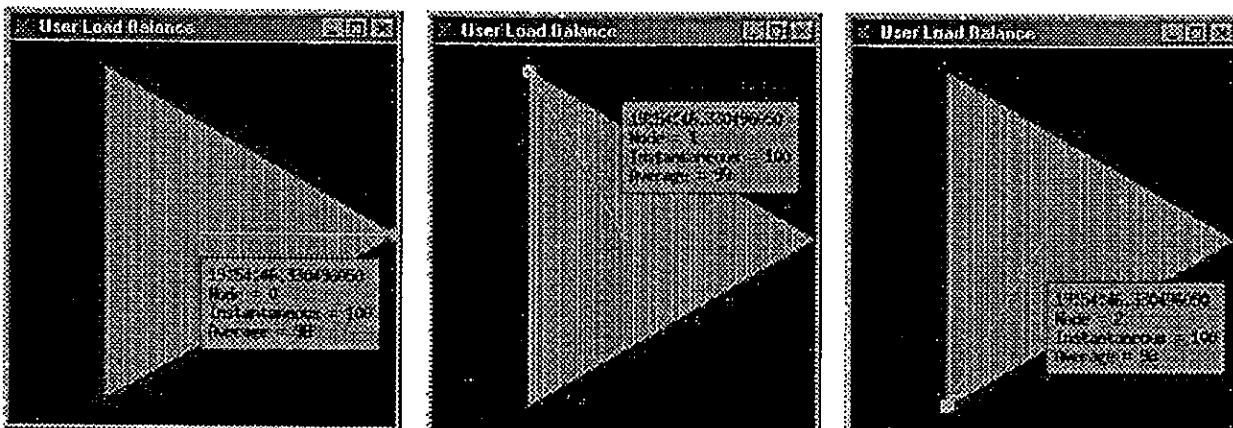
A Figura 7.3 mostra o balanceamento de carga para cada aplicação e para cada nó utilizado, a Figura 7.3 (a) apresenta o balanceamento para o algoritmo do trapezio, a (b) para o algoritmo de Jacobi e a (c) para o algoritmo de multiplicação de matrizes. Um pequeno retângulo como o mostrado na Figura 7.2 (b), é mostrado em cada figura. Essa média é feita automaticamente pelo VT no final da execução do programa.



(a) Balanceamento de carga para o algoritmo do trapézio.



(b) Balanceamento de carga para o algoritmo de Jacobi.



(c) Balanceamento de carga para o algoritmo de multiplicação de matriz.

**Figura 7.3 – Balanceamento de carga para os algoritmos no SP2.**

Dos três algoritmos, o que apresentou melhor média foi o algoritmo de multiplicação de matrizes, seguido do algoritmo do trapézio. Isso ocorreu porque ambos só fazem duas comunicações durante todo o processo, uma no inicio do programa para especificar os limites e passar as matrizes para os respectivos processadores quando necessário, e uma comunicação no final da execução para juntar o resultado.

## 7.2 Execução dos algoritmos

Os algoritmos foram implementados conforme o apresentado nos Capítulos 4 e 6. Na arquitetura Cray foi utilizado um nível agressivo de vetorização. Para realizar todos os cálculos foram executados os seguintes passos:

- Primeiro é executado a versão serial do algoritmo guardando o tempo de execução;
- Em seguida é executado o algoritmo paralelo armazenando o tempo de execução;
- Calcula-se o *speedup*;
- Calcula-se a eficiência;
- Executa-se uma versão do programa serial com contadores para encontrar o número de operações em ponto flutuante ( $F_B(N)$ );
- Calcula-se o desempenho temporal ( $R_T(N,p)$ ) seguindo a teoria apresentada no Capítulo 3, neste capítulo o  $R_T(N,p)$  também será referenciado como simplesmente  $R_T(N)$ ;
- Calcula-se o desempenho do *benchmark* ( $R_B(N,p)$ ) também seguindo o explicado no Capítulo 3.

### 7.2.1 Resultados do programa de integração numérica

As Tabela 7.1 e 7.2 mostram os resultados encontrados no SP2 utilizando o MPI. As Tabelas 7.3 e 7.4 mostram os resultados obtidos utilizando o PVMe, também no SP2. As duas primeiras linhas expressam o tempo de execução em segundos para 3, 2 e 1 processador respectivamente. A terceira e quarta linhas mostram o resultado do cálculo do *speedup* e da eficiência . A quinta linha ilustra o desempenho temporal que é calculado a título de demonstração pois não foram implementados vários algoritmos para solucionar o mesmo problema. A penúltima linha mostra o resultado obtido com o programa contador de operações em ponto flutuante. E a última linha de todas as tabelas, geralmente só é comentada no final de cada subseção onde as arquiteturas são comparadas. O cabeçalho das tabelas indicam a quantidade total de divisões da integral em milhões. Por exemplo, o valor 9 indica que a integral foi dividida em 9 milhões de pedaços.

Tabela 7.1 - Resultados encontrados no SP2 utilizando MPI com 3 processadores.

	9	18	36	72	144	288
3 proc. (s)	5,20	10,38	20,82	41,55	83,00	175,80
Serial (s)	14,55	29,17	58,29	116,59	233,16	473,93
Speedup	2,7981	2,8102	2,7997	2,806	2,8092	2,6958
Eficiência	0,9327	0,9367	0,9332	0,935	0,9364	0,8986
$R_T(N,p)^*$ - sols/s	0,19	0,1	0,05	0,02	0,01	0,0057
$F_B(N)^{**}$ - Mflop	198	396	792	1584	3168	6336
$R_B(N,p)^{***}$ - Mflop/s	38,08	38,15	38,04	38,12	38,17	36,04

\*  $R_T(N,p)$  – Desempenho temporal

\*\*  $F_B(N)$  – Quantidade de operações em ponto flutuante

\*\*\*  $R_B(N,p)$  – Desempenho do *benchmark*.

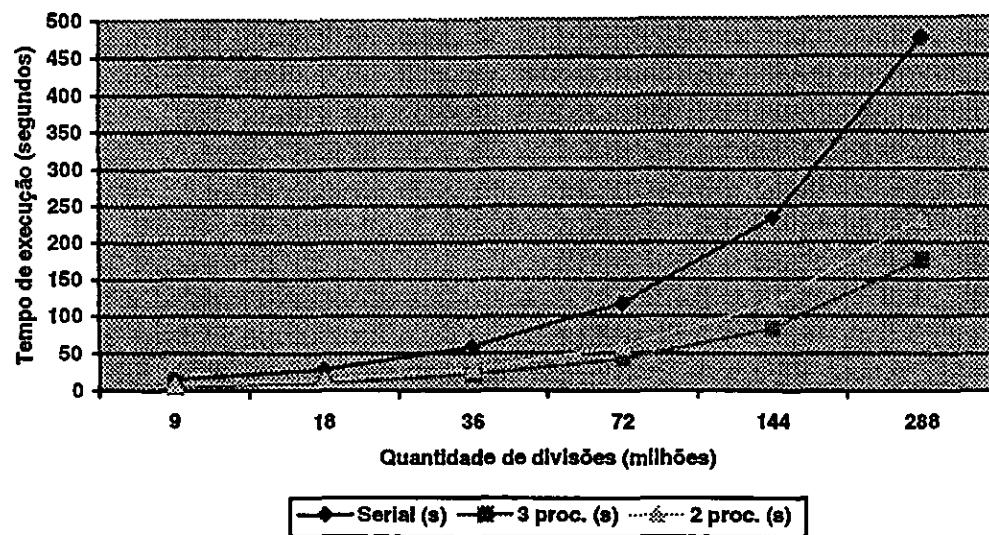
Tabela 7.2 - Resultados encontrados no SP2 utilizando MPI com 2 processadores.

	9	18	36	72	144	288
2 proc.(s)	7,27	14,52	29,12	58,16	116,29	232,67
Serial (s)	14,55	29,17	58,29	116,59	233,16	473,93
Speedup	2,0014	2,009	2,0017	2,0046	2,005	2,0369
Eficiência	1,0007	1,0045	1,0085	1,0023	1,0025	1,018
$R_T(N)$ – sols/s	0,14	0,07	0,03	0,02	0,01	0,0043
$F_B(N)$ – Mflop	198	396	792	1584	3168	6336
$R_B(N,p)$ – Mflop/s	27,24	27,27	27,2	27,24	27,24	27,23

O resultado da execução das três versões do algoritmo pode ser melhor visto no Gráfico 7.1, onde a execução em 3 processadores obteve o menor tempo de execução. Como a granulação desta aplicação é grossa e a quantidade de processadores é pequena a Lei de *Ahmdal*<sup>2</sup> não se aplica. Houve uma pequena anomalia de *speedup* na versão com 2 processadores devido a grande quantidade de memória disponível nos processadores do SP2 (256 Mbytes para o nó *wide* e 128 Mbytes para o nó *thin*), consequentemente houve uma anomalia na eficiência. Nota-se que apesar da ocorrer a anomalia com 2 processadores o desempenho de *benchmark* ( $R_B(N)$ ), ou seja, a quantidade de operações em ponto flutuante por segundo com dois processadores foi menor. A eficiência alcançada ficou por volta de 93% bem semelhante ao balanceamento de carga mostrado na Figura 7.3 (a). Essa diferença entre a eficiência e o balanceamento de carga deu-se pelo fato de que o balanceamento de carga é feito no programa inteiro e a eficiência é baseada somente no tempo de cálculo da integral.

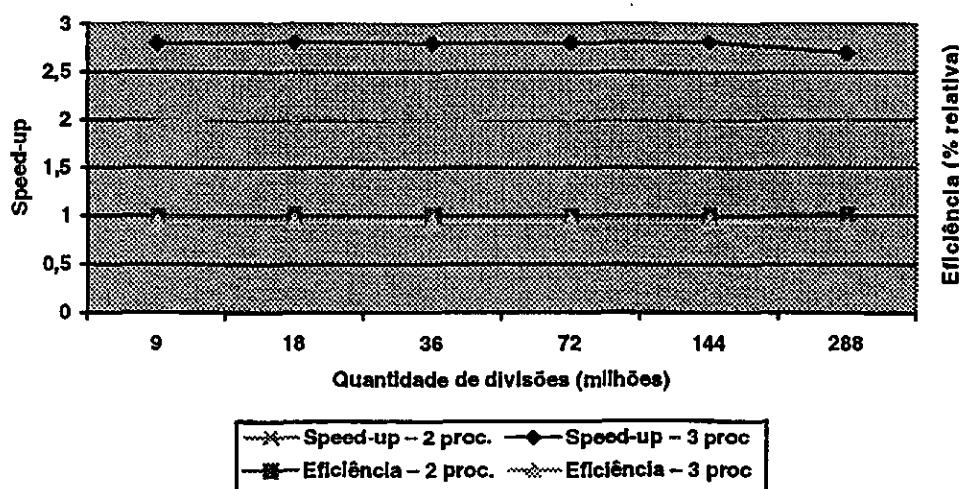
<sup>2</sup> A lei de Ahmdal diz que a partir de um número  $n$  de processadores ( $n$  representa a quantidade de processadores que torna a aplicação a mais rápida possível) a velocidade de execução diminui a medida que novos processadores são adicionados.

**Gráfico 7.1 - Tempo de execução do algoritmo do trapézio composto no SP2 utilizando MPI.**



O Gráfico 7.2 apresenta o *speedup* e a eficiência alcançado nas versões para 2 e 3 processadores. Nota-se que apesar de haver uma nítida diferença de *speedup* a eficiência para 2 processadores é ligeiramente maior que para 3 processadores, isso não implica que deve-se utilizar 2 processadores no lugar de 3, pois o tempo de execução em 3 processadores é menor. Aqui o que deve ser considerado é o *speedup* tomando o cuidado de se observar o tempo de execução para que não ocorram os problemas do *speedup* apresentadas na seção 3.4 do Capítulo 3, que considera que nem sempre o algoritmo que tem o maior *speedup* é o mais rápido.

**Gráfico 7.2 - Speed-up e eficiência alcançada por 2 e 3 processadores no SP2.**



As Tabelas 7.3 e 7.4 apresentam os resultados encontrados no SP2 utilizando-se o PVMe. Como esperado, o comportamento foi semelhante ao da execução com MPI, já que ambos se utilizam do *switch* de alto desempenho para efetuar a comunicação entre processadores.

Tabela 7.3 - Resultados encontrados no SP2 utilizando PVMe com 3 processadores.

	9	18	36	72	144	288
3 proc.	5,076	9,9131	19,67	39,21	78,1961	156,246
Serial	14,55	29,17	58,29	116,59	233,16	473,93
Speedup	2,8664	2,9426	2,9634	2,9735	2,9817	3,0332
Eficiência	0,9555	0,9809	0,9878	0,9912	0,9939	1,0111
$R_T(N)$ – sols/s.	0,197	0,1009	0,05	0,03	0,0128	0,0064
$F_B(N)$ – Mflop	198	396	792	1584	3168	6336
$R_B(N,p)$ – Mflop/s	39,007	39,9471	40,26	40,4	40,5135	40,551

\*  $R_T(N)$  – Desempenho temporal

\*\*  $F_B(N)$  – Quantidade de operações em ponto flutuante

\*\*\*  $R_B(N,p)$  – Desempenho do *benchmark*.

Tabela 7.4 - Resultados encontrados no SP2 utilizando PVMe com 2 processadores.

	9	18	36	72	144	288
2 proc.	7,0865	14,038	27,64	55,0672	111,148	219,91
Serial	14,55	29,17	58,29	116,59	233,16	473,93
Speedup	2,0532	2,0779	2,1089	2,117231	2,0977	2,1551
Eficiência	1,0266	1,039	1,0545	1,058616	1,0489	1,0776
$R_T(N)$ – sols/s.	0,14	0,07	0,03	0,02	0,01	0,0043
$F_B(N)$ – Mflop	198	396	792	1584	3168	6336
$R_B(N,p)$ – Mflop/s	38,08	38,15	38,04	38,12	38,17	36,04

A medida que a quantidade de divisões aumenta, o PVMe tem uma nítida vantagem sobre a versão em MPI, principalmente na execução com 3 processadores. Isso acontece porque os dois processadores escravos no MPI, tem que esperar que o mestre esteja pronto para receber a mensagem final que faz o somatório dos cálculos, isso pode ser visualizado na Figura 7.4 obtida através da utilização do VT. As barras indicam a execução da aplicação. Os retângulos numerados do lado direito, indicam os respectivos nós, sendo que 0 (zero) representa o mestre e os demais representam os escravos. As linhas que interligam as barras representam a comunicação entre os processadores. A faixa do  $R_B(N,p)$  é mantida porque o problema tem uma complexidade  $n$ . Isso faz com que o crescimento do problema seja linear, ou seja, se o tamanho do problema dobra, a quantidade de operações em ponto flutuante e o tempo de execução também dobram. Conseqüentemente, a faixa de operações em ponto flutuante por segundo ( $R_B(N,p)$ ) se mantém quase que constante, já que  $R_B(N,p) = F_B(N) / T(N,p)$ , onde  $T(N,p)$  é o

tempo de execução em paralelo com  $p$  processadores e  $F_B(N)$  é a quantidade de operações em ponto flutuante.

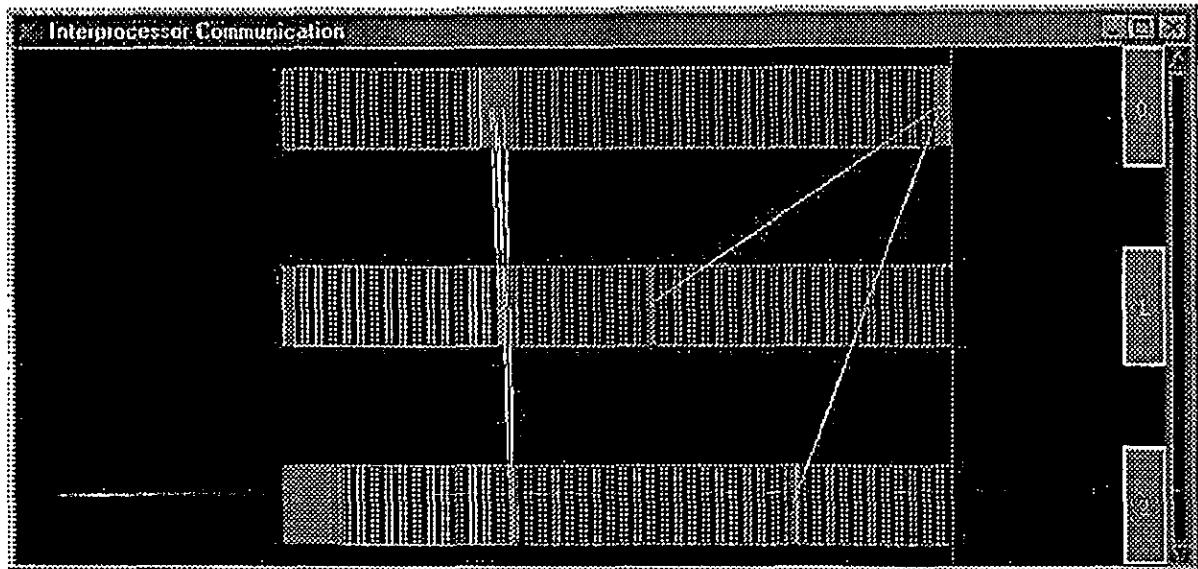


Figura 7.4 - Comunicação entre 3 processadores.

A Figura 7.5 mostra a execução do algoritmo do trapézio para 2 processadores. Nota-se que a espera pela comunicação com o mestre é nitidamente menor que a da Figura 7.4 levando a maior eficiência encontrada para 2 processadores.

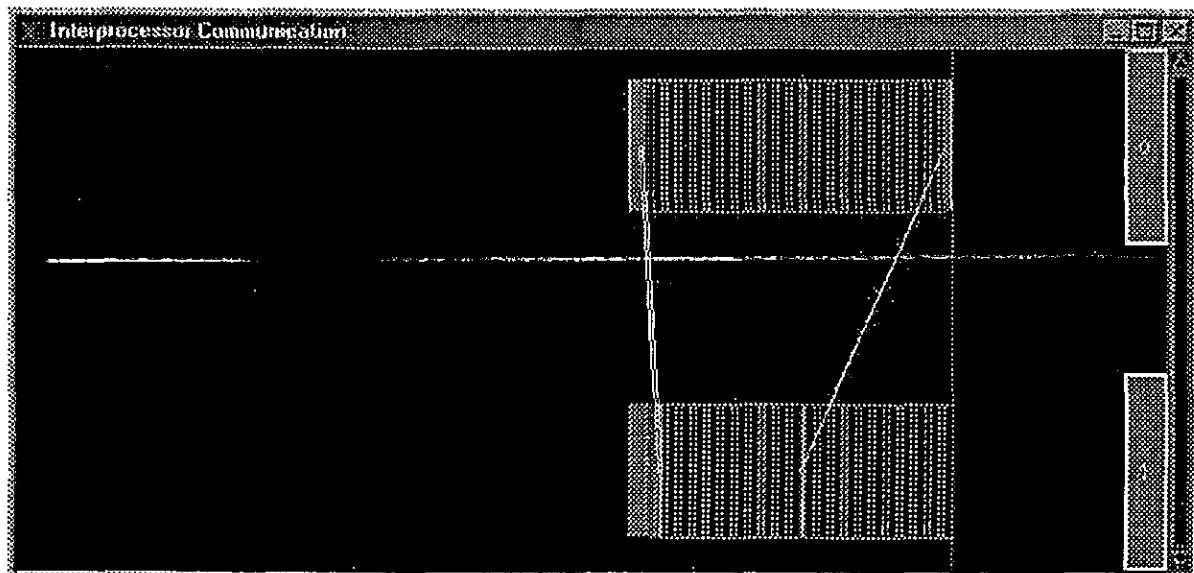
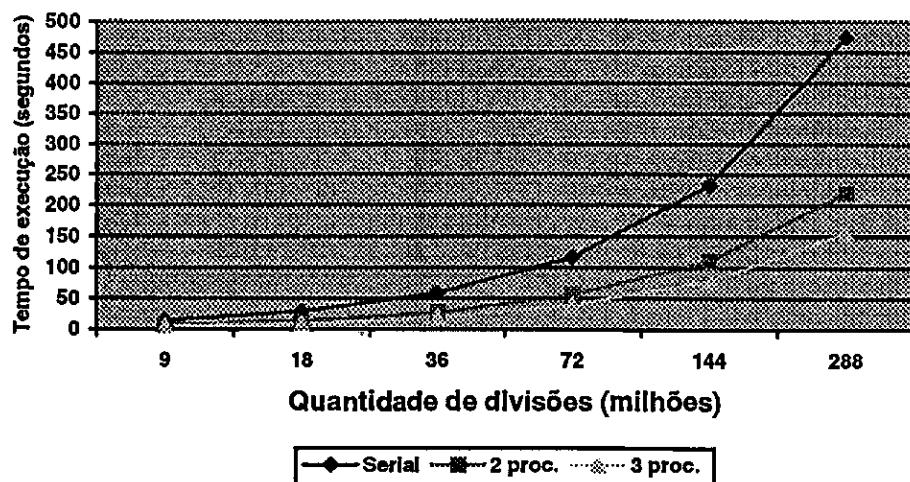


Figura 7.5 Comunicação entre 2 processadores

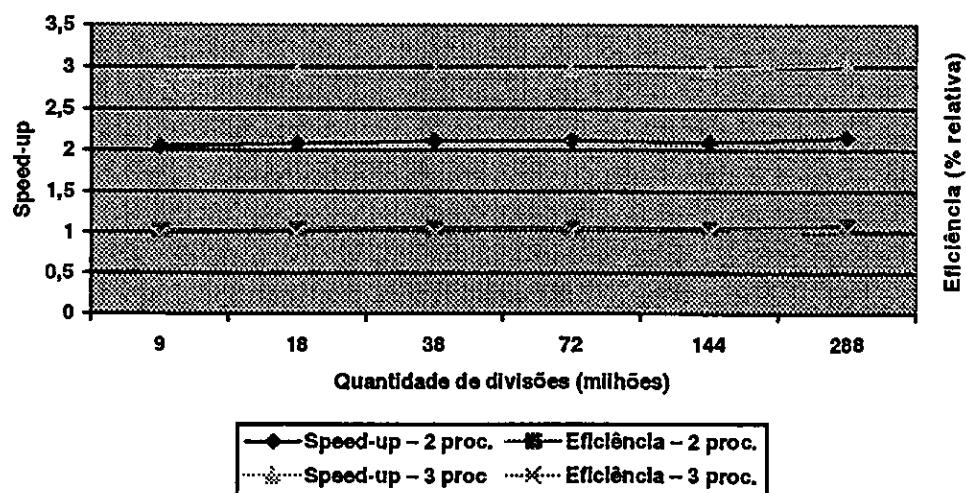
O Gráfico 7.3 mostra a execução utilizando PVMe. Se comparado ao Gráfico 7.1 a escala do gráfico não permite uma visualização clara, mas se os tempo de execução forem comparados entre as Tabelas 7.3 a 7.4 e as Tabelas 7.1 a 7.2 essa diferença será rapidamente notada.

**Gráfico 7.3 - Tempo de execução do algoritmo do trapézio composto no SP2 utilizando-se PVMe.**



A seguir o Gráfico 7.4 mostra o *speedup* e a eficiência alcançados. Como esperado, o comportamento foi semelhante ao do Gráfico 7.2 pois a faixa de *speedup* e de eficiência foi mantida devido a linearidade dos resultados.

**Gráfico 7.4 - Speed-up e eficiência alcançados por 2 e 3 processadores no SP2 utilizando PVMe**



As Tabelas 7.5 e 7.6 mostram os resultado encontrados na execução do algoritmo do trapézio no sistema distribuído do LASD-PC utilizando-se o MPI.

**Tabela 7.5 - Resultados encontrados no LASD-PC utilizando MPI com 3 processadores.**

	9	18	36	72	144	288
3 proc.	10,078	20,1103	40,1806	80,5113	160,59	321,148
Serial	16,29	35,51	65,01	130,22	260,10	520,16
Speedup	1,6164	1,7658	1,7658	1,6174	1,6197	1,6197
Eficiência	0,5388	0,5886	0,5886	0,5391	0,5399	0,5399
$R_T(N)$ – sols/s	0,07	0,0497	0,0249	0,0124	0,01	0,0031
$F_B(N)$ – Mflop	198	396	792	1584	3168	6336
$R_B(N,p)$ – Mflop/s	19,6468	19,6914	19,711	19,6743	19,73	19,729

\*  $R_T(N)$  – Desempenho temporal

\*\*  $F_B(N)$  – Quantidade de operações em ponto flutuante

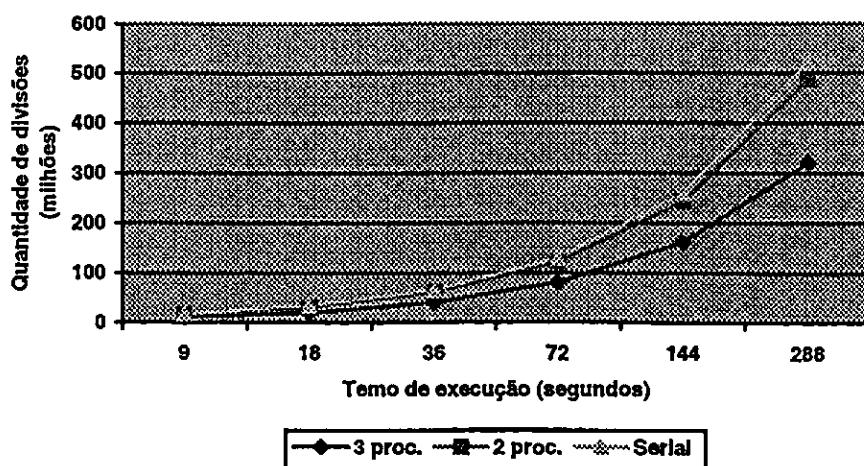
\*\*\*  $R_B(N,p)$  – Desempenho do *benchmark*.

**Tabela 7.6 - Resultados encontrados no LASD-PC utilizando MPI com 2 processadores.**

	9	18	36	72	144	288
2 proc.	15,20	30,37	60,711	121,41	242,771	485,519
Serial	16,29	35,51	65,01	130,22	260,10	520,16
Speedup	1,0717	1,1692	1,1692	1,0726	1,0714	1,0713
Eficiência	0,5359	0,5846	0,5846	0,5363	0,5357	0,5357
$R_T(N)$ – sols/s.	0,06	0,03	0,02	0,01	0,0038	0,0019
$F_B(N)$ – Mflop	198	396	792	1584	3168	6336
$R_B(N,p)$ – Mflop/s	13,03	13,04	13,045	13,05	13,049	13,05

Os tempos de execução apresentados nas Tabelas 7.5 e 7.6 foram coerente, dado novamente a linearidade do problema. Apesar da velocidade dos processadores ser maior no sistema distribuído se comparado aos processadores do SP2, o sistema distribuído leva desvantagem pela pouca eficiência da unidade de ponto flutuante das máquinas do tipo PC que realizam cálculos mais lentamente e também leva desvantagem pela quantidade de memória RAM e pela quantidade de memória cache. A métrica tempo de execução não deve ser utilizada para comparar as arquiteturas. O Gráfico 7.5 mostra melhor os tempos alcançados pelo sistema distribuído utilizando o MPI (*Mpich*).

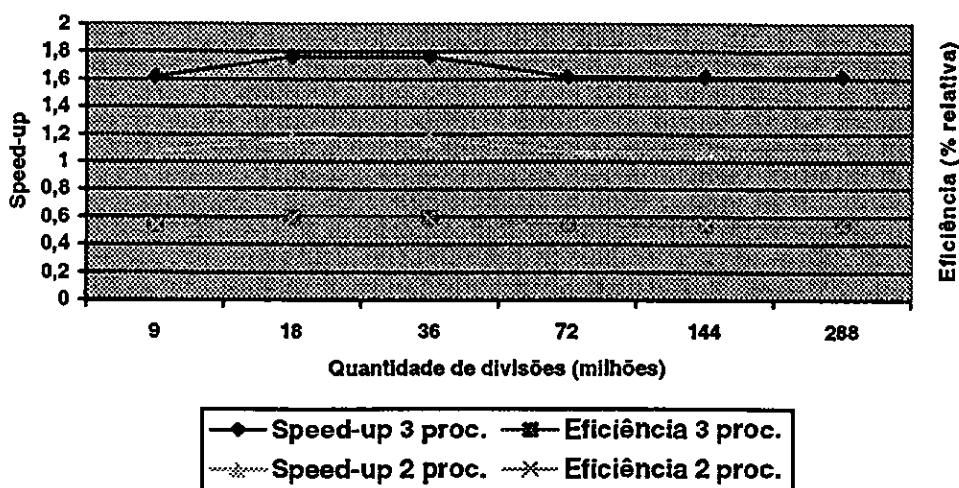
**Gráfico 7.5 - Tempo de execução do algoritmo do trapézio composto no LASD-PC utilizando MPI.**



O mestre que tem uma carga um pouco maior sempre foi executado no processador de 200 Mhz (Lasd07), como os demais processadores são mais lentos isso acarretou numa espera do mestre pelas mensagens finais. Para dois processadores essa afirmação torna-se mais clara, por exemplo, dividir 288 milhões de trapézios em dois processadores, o mestre vai calcular uma integral de 144 milhões de divisões bem mais rápido que o escravo e vai ter que esperar até este terminar para obter o resultado final.

O Gráfico 7.6 apresenta a eficiência e o *speedup* alcançados pelo algoritmo do trapézio utilizando MPI no LASD-PC.

**Grafico 7.6 - Eficiência e speed-up alcançados no LASD-PC utilizando MPI.**



Analisando-se o Gráfico 7.6 e as Tabelas 7.5 e 7.6, novamente pode-se cair na armadilha da eficiência e do *speedup*. O fato da eficiência com 2 processadores estar extremamente próxima da com 3 processadores e dos *speedups* serem não muito distantes um do outro, não seria aconselhável, a utilização de um sistema distribuído com 2 nós para resolver esse tipo de problema.

As Tabelas 7.7 e 7.8 a seguir mostram os resultados encontrados na execução do algoritmo do trapézio no LASD-PC utilizando o PVM.

**Tabela 7.7 – Resultados encontrados no LASD-PC utilizando PVM com 3 processadores.**

	9	18	36	72	144	288
3 proc.	10,89	18,685	37,1429	74,5757	149,60	297,337
Serial	16,29	35,51	65,01	130,22	260,10	520,16
Speedup	1,4959	1,9005	1,9005	1,7461	1,7386	1,7494
Eficiência	0,4986	0,6335	0,6335	0,582	0,5795	0,5831
$R_T(N)$ – sols/s	0,06	0,03	0,02	0,01	0,0038	0,0019
$F_B(N)$ – Mflop	198	396	792	1584	3168	6336
$R_B(N,p)$ – Mflop/s	18,18	21,193	21,3231	21,2402	21,18	21,309

**Tabela 7.8 - Resultados encontrados no LASD-PC utilizando PVM com 2 processadores.**

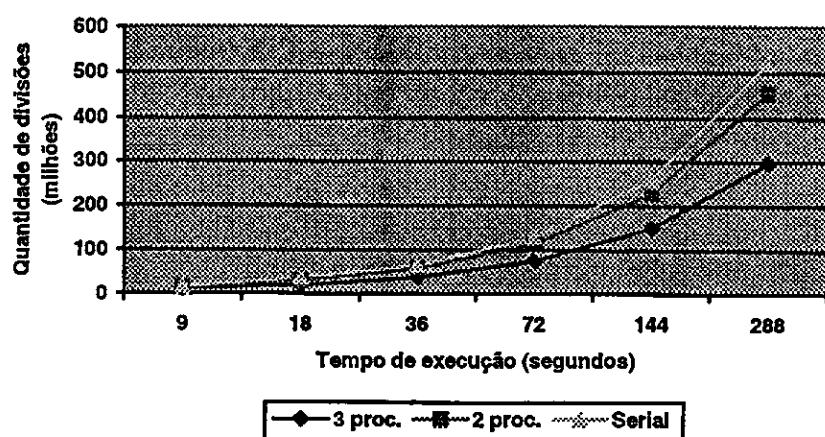
	9	18	36	72	144	288
2 proc.	14,2739	28,5584	57,1901	114,0264	228,0590	456,1955
Serial	16,29	35,51	65,01	130,22	260,10	520,16
Speedup	1,1412	1,2434	1,2434	1,142	1,1405	1,1402
Eficiência	0,5706	0,6217	0,6217	0,571	0,5703	0,5701
$R_T(N)$ – sols/s.	0,06	0,03	0,02	0,01	0,0038	0,0019
$F_B(N)$ – Mflop	198	396	792	1584	3168	6336
$R_B(N,p)$ – 2 proc. Mflop/s	13,8715	13,8663	13,8486	13,8915	13,8911	13,8888

No caso do LASD-PC usando PVM, a velocidade relativamente pobre de execução foi gerada novamente pela diferença de velocidade entre os processadores, essa diferença acontece quando o mestre (asd07 – 200 Mhz) termina a execução dos seus cálculos e fica esperando pelo resultado dos escravos (asd06 – 166 Mhz e asd08 – 133 Mhz), que com certeza ainda não terminaram. Somado a isso existe o problema de que a velocidade da rede é de apenas 10 Mbits/s causando uma maior sobrecarga na comunicação.

O Gráfico 7.7 mostra o tempo de execução para cada quantidade de processadores. A linha que representa a execução em dois processadores se aproxima muito da linha seqüencial. Esse

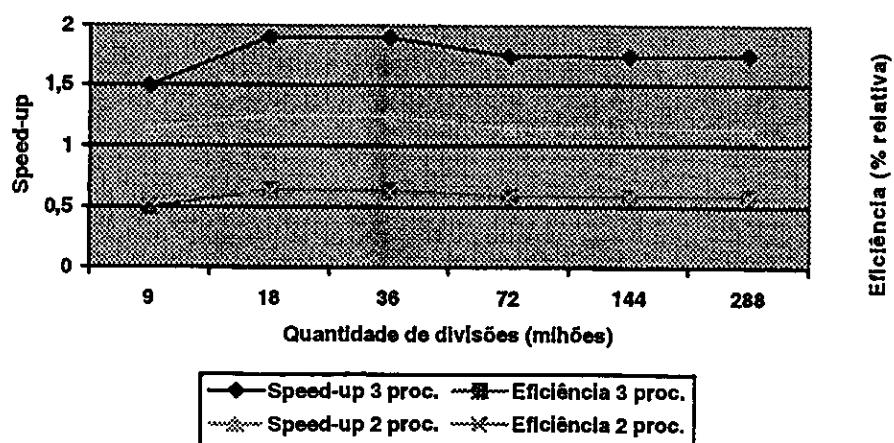
desempenho relativamente pobre deve-se novamente ao fato da velocidade dos processadores ser desigual e pela baixa velocidade de comunicação da rede. As três linhas quase se juntam no inicio devido a pouca quantidade de divisões efetuadas, quanto menor a quantidade de divisões há uma maior utilização da memória cache o que torna a execução muito mais rápida do que se a aplicação tivesse que usar a memória RAM. Aliado a isso, encontra-se a diferença de velocidade entre os processadores e a baixa velocidade de transmissão de dados da rede (10 Mbits/s).

**Gráfico 7.7 - Tempo de execução do algoritmo do trapézio composto no LASD-PC utilizando PVM.**



Usando-se três processadores a computação paralela começa a ficar interessante a partir de 36 milhões de divisões. Com dois processadores a utilização da computação paralela não é muito interessante pois seu ganho é muito pequeno, a utilização de uma única máquina com processador de 300 Mhz, por exemplo, tornaria a execução mais rápida e o custo mais baixo.

**Gráfico 7.8 - Eficiência e speed-up alcançados no LASD-PC utilizando PVM**



No gráfico 7.8 observa-se uma diferença maior entre o *speedup*/eficiência para 2 e 3 processadores. Neste caso elas podem ser adotadas como medida de desempenho, mas apenas para comparação do mesmo algoritmo na mesma arquitetura e ainda com muita cautela. Observa-se também o baixa *speedup* para 2 processadores que se encontra próximo ao valor 1. Isso indica que sua velocidade de processamento foi próxima a de um único processador.

Os resultados para a arquitetura *Cray* são mostrados nas Tabelas 7.9 e 7.10.

**Tabela 7.9 – Resultados encontrados no *Cray* utilizando 3 processadores.**

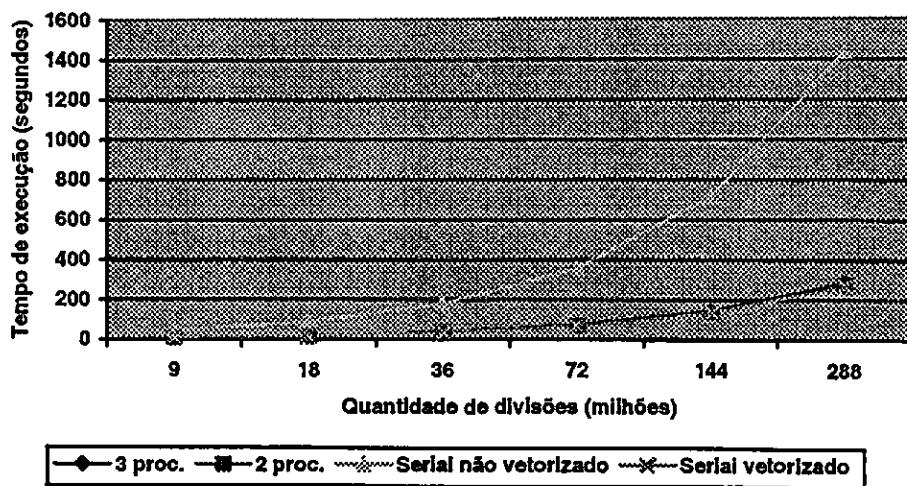
	9	18	36	72	144	288
3 proc.	8,81	17,34	34,78	70,94	141,44	284,04
Serial vetorizado	8,47	16,80	33,42	67,32	134,26	269,21
Serial não vetorizado	44,79	89,85	180,74	358,35	717,70	1439,95
Speedup	5,084	5,1817	5,1817	5,0515	5,0742	5,0695
Eficiência	1,695	1,7272	1,7272	1,6838	1,6914	1,6898
$R_T(N)$ – sols/s	0,1135	0,0577	0,0288	0,0141	0,0071	0,0035
$F_B(N)$ – Mflop	198	396	792	1584	3168	6336
$R_B(N,p)$ – Mflop/s	22,47	22,84	22,77	22,33	22,4	22,31

**Tabela 7.10 - Resultados encontrados no *Cray* utilizando 2 processadores.**

	9	18	36	72	144	288
2 proc.	8,50	17,15	34,18	68,44	137,55	274,37
Serial vetorizado	8,47	16,80	33,42	67,32	134,26	269,21
Serial não vetorizado	44,79	89,85	180,74	358,35	717,70	1439,95
Speedup	5,2694	5,2391	5,2391	5,236	5,2177	5,2482
Eficiência	2,6347	2,6196	2,6196	2,618	2,6089	2,6241
$R_T(N)$ – sols/s	0,1176	0,0583	0,0293	0,0146	0,0073	0,0036
$F_B(N)$ – Mflop	198	396	792	1584	3168	6336
$R_B(N,p)$ – Mflop/s	23,29	23,09	23,17	23,14	23,03	23,09

Nota-se um desempenho bem pobre para a execução seqüencial não vetorizada. A versão vetorizada teve uma leve vantagem sobre as demais devido ao fato de esta utilizar intensivamente de registradores vetoriais, *pipelines* vetoriais e memória cache local para comunicação e acesso aos dados. O mesmo não ocorre com diversos processadores já que estes tem que efetuar a comunicação via memória compartilhada inviabilizando o uso de cache local para comunicação e inserindo problemas de sincronização (acesso concorrente). O Gráfico 7.9, apresenta a execução das quatro versões.

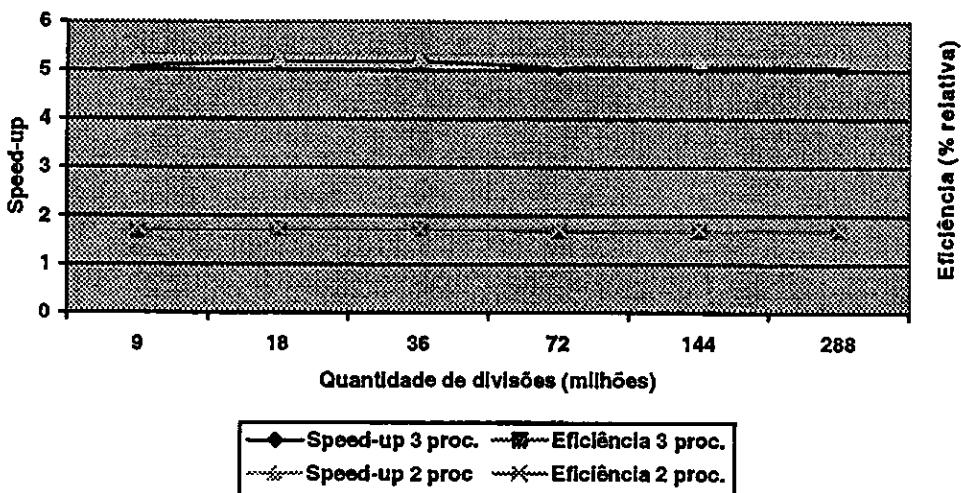
**Gráfico 7.9 - Tempo de execução do algoritmo do trapézio composto no sistema Cray.**



Os tempos de execução entre 3, 2, e 1 processador usando vetorização foi praticamente igual. A execução com 1 processador vetorial levou um pequena vantagem pelos motivos já mencionados como o acesso concorrente a memória.

O Gráfico 7.10 mostra o *speedup* e a eficiência alcançados no sistema *Cray*.

**Gráfico 7.10 - Eficiência e speed-up alcançados no sistema Cray.**



Novamente encontra-se uma anomalia de *speedup* e de eficiência, não só pelo fato da máquina possuir muita memória, mas sim pelo fato de que a vetorização proporciona uma alta velocidade de processamento e pelo fato de que a execução seqüencial foi extremamente pobre.

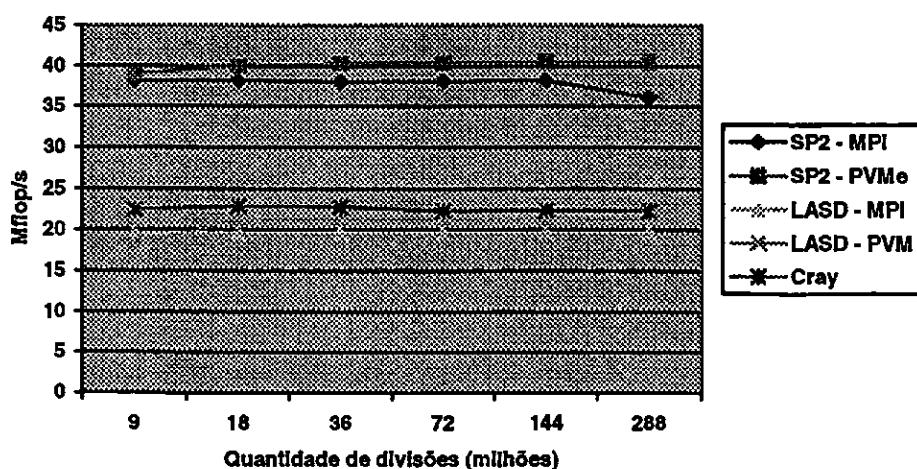
## Comparação de arquiteturas

A Tabela 7.11 seguida do Gráfico 7.11 apresentam um apanhado dos desempenhos *benchmark* encontrados nas três arquiteturas. A métrica utilizada para comparação será o Mflop/s, isto é, qual máquina consegue executar mais operações em ponto flutuante por segundo.

Tabela 7.11 - Comparação entre arquiteturas com 3 processadores

$R_B(N,p)$	9	18	36	72	144	288
SP2 - MPI	38,08	38,15	38,04	38,12	38,17	36,04
SP2 - PVMe	39,007	39,9471	40,26	40,4	40,5135	40,551
LASD - MPI	19,6468	19,6914	19,711	19,6743	19,73	19,729
LASD - PVM	18,18	21,193	21,3231	21,2402	21,18	21,309
Cray	22,47	22,84	22,77	22,33	22,4	22,31

Gráfico 7.11 - Comparação entre arquiteturas com 3 processadores executando o algoritmo do trapézio.



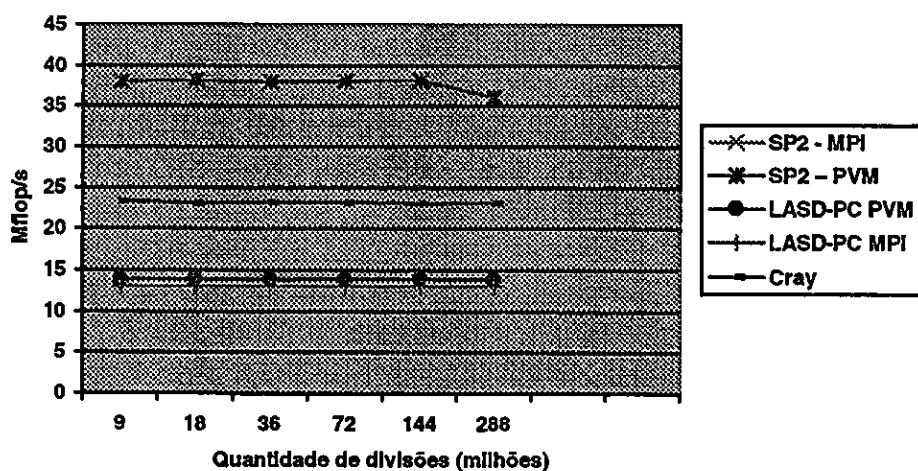
Como apresentado no Capítulo 3, devido a linearidade do problema, a métrica  $R_B(N,p)$  pode ser usada para comparar arquiteturas sem nenhuma restrição. Pelo Gráfico 7.11 nota-se que a arquitetura SP 2 utilizando-se PVMe foi superior as demais. Apesar da arquitetura Cray ter tido um desempenho superior a do sistema distribuído, em termos de custo, não valeria a pena adquiri-la para utilizá-la com esse algoritmo. Um sistema distribuído com mais memória (RAM e cache) seria mais rápido e extremamente mais barato.

A Tabela 7.12 e o Gráfico 7.12 apresentam os resultados obtidos no desempenho de *benchmark* para as três arquiteturas utilizando dois processadores.

**Tabela 7.12 - Comparação entre arquiteturas com 2 processadores**

	9	18	36	72	144	288
R <sub>B</sub> (N,p) SP2 – MPI	27,24	27,27	27,2	27,24	27,24	27,23
R <sub>B</sub> (N,p) SP2 – PVM <sub>e</sub>	38,08	38,15	38,04	38,12	38,17	36,04
R <sub>B</sub> (N,p) LASD-PC PVM	13,8715	13,8663	13,8486	13,8915	13,8911	13,8888
R <sub>B</sub> (N,p) LASD-PC MPI	13,03	13,04	13,045	13,05	13,049	13,05
R <sub>B</sub> (N,p) Cray	23,29	23,09	23,17	23,14	23,03	23,09

Comparando a Tabela 7.12 com os tempos de execução encontrados, nota-se a mesma ordem ganho, ou seja, SP2 utilizando PVMe executou a aplicação em menor tempo seguido do SP2 utilizando MPI e assim por diante.

**Gráfico 7.12 - Comparação entre arquiteturas com 3 processadores executando o algoritmo do trapézio.**

Novamente a arquitetura SP2 utilizando o PVMe teve um desempenho maior. O comportamento do PVM no LASD-PC também foi superior ao do MPI. Isso deve-se a eficiência do compilador ao gerar o código executável para as bibliotecas correspondentes.

## 7.2.2 Resultado do algoritmo de Jacobi

Esta seção apresenta os resultados obtidos executando o algoritmo de Jacobi. As linhas seguem o mesmo esquema das tabelas já apresentada. O cabeçalho das tabelas indicam o tamanho da matriz utilizada, por exemplo, 1200 indica que a matriz é de 1200 x 1200.

## Resultados

---

As Tabelas 7.13 e 7.14 mostram os resultados encontrados na arquitetura SP2 utilizando a biblioteca de passagem de mensagem MPI.

**Tabela 7.13 - Resultados encontrados no SP2 utilizando MPI com 3 processadores.**

	300	600	900	1200	1500	1800	2100
3 proc.	0,0805	0,1132	0,3399	0,6328	0,9547	1,3414	1,8676
Serial	0,0780	0,2688	0,6057	1,1595	1,6281	2,3461	3,1919
Speedup	0,9689	2,3746	1,782	1,8323	1,7054	1,749	1,7091
Eficiência	0,323	0,7915	0,594	0,6108	0,5685	0,583	0,5697
$R_T(N)$ – sols/s.	12,4224	8,8339	2,942	12,4224	8,8339	2,942	1,5803
$F_B(N)$ – Mflop	0,4527	1,8054	4,0581	7,2108	11,2635	16,2162	22,0689
$R_B(N,p)$ – Mflop/s	5,6236	15,9488	11,9391	11,3951	11,7979	12,089	11,8167

**Tabela 7.14 - Resultados encontrados no SP2 utilizando MPI com 2 processadores.**

	300	600	900	1200	1500	1800	2100
2 proc.	0,3839	0,4002	0,5442	0,8227	1,1968	1,6505	2,2873
Serial	0,0780	0,2688	0,6057	1,1595	1,6281	2,3461	3,1919
Speedup	0,2032	0,6717	1,113	1,4094	1,3604	1,4214	1,3955
Eficiência	0,1016	0,3359	0,5565	0,7047	0,6802	0,7107	0,6978
$R_T(N)$ – sols/s	2,6048	2,4988	1,8376	1,2155	0,8356	0,6059	0,4372
$F_B(N)$ – Mflop	0,4527	1,8054	4,0581	7,2108	11,2635	16,2162	22,0689
$R_B(N,p)$ – Mflop/s	1,1792	4,5112	7,457	8,7648	9,4113	9,825	9,6485

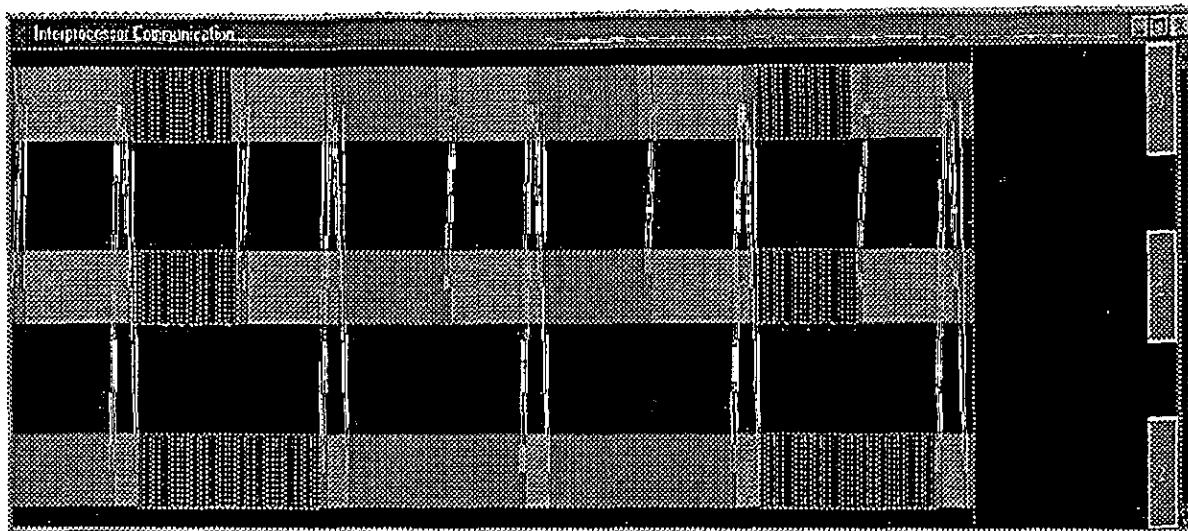
\*  $R_T(N)$  – Desempenho temporal

\*\*  $F_B(N)$  – Quantidade de operações em ponto flutuante

\*\*\*  $R_B(N,p)$  – Desempenho do *benchmark*.

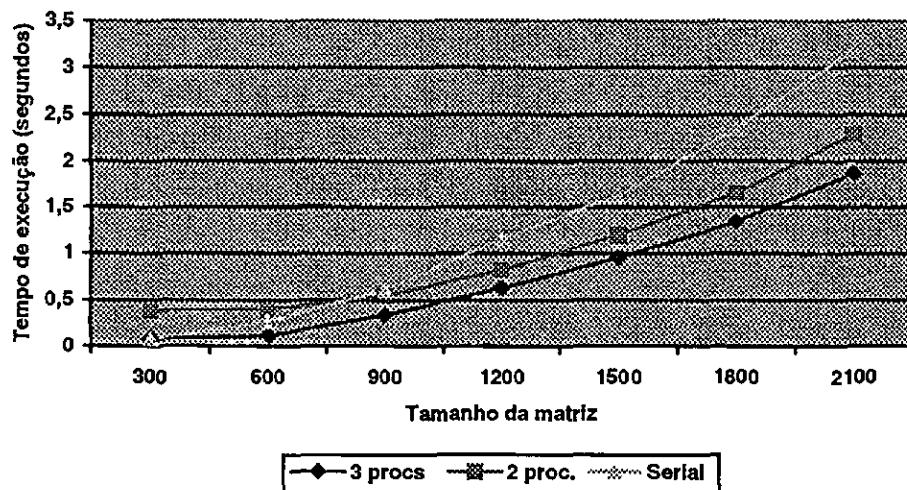
Como esperado, o tempo de execução em 3 processadores foi menor que o tempo para a execução em 2 processadores conforme mostra o Gráfico 7.13. Esta aplicação já não possui a mesma linearidade da aplicação anterior (algoritmo do trapézio). O algoritmo de Jacobi apresenta uma complexidade  $n^2$ , por esse motivo, a quantidade de operações em ponto flutuante cresce junto com o tamanho do problema e consequentemente o desempenho do benchmark ( $R_B(N,p)$ ) também é crescente.

Como esta é uma aplicação que executa um número maior de comunicações (Figura 7.6) em comparação com as outras (trapézio e multiplicação de matriz) o *speedup* e a eficiência são números que devem ser utilizados com cautela para analisar o desempenho. O Gráfico 7.14 apresenta o *speedup* e a eficiência alcançados pela aplicação. Observa-se que para esse exemplo, obteve-se *speedups* razoáveis para os casos em que as matrizes de ordem mais elevadas foram consideradas. Mesmo para matrizes grandes, a eficiência não foi tão atrativa como nos outros exemplos. A maior quantidade de comunicação explica esses resultados.



**Figura 7.6 - Trecho final de comunicação no algoritmo de Jacobi.**

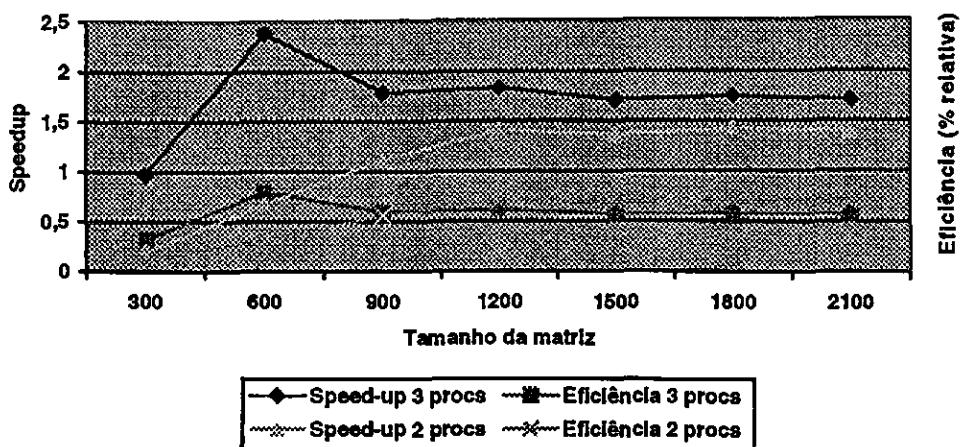
**Gráfico 7.13 - Tempo de execução do algoritmo de Jacobi no SP2 utilizando o MPI.**



O tempo de execução serial, em 2 processadores e 3 processadores é praticamente o mesmo até uma matriz 600 x 600. Nota-se que a diferença cresce com o aumento do tamanho da matriz. Houve um encontro entre a execução com dois processadores com a linha serial para o tamanho da matriz de 900x900, isso é esperado já que a execução serial começou mais rápida que a com 2 processadores.

O Gráfico 7.14 apresenta o *speedup* e a eficiência mostradas nas Tabelas 7.13 e 7.14.

**Grafico 7.14 - Speedup e eficiência alcançados por 2 e 3 processadores no SP utilizando MPI.**



Pode-se observar que o gráfico já não é tão linear como os anteriores (algoritmo do trapézio). Isso deve-se ao fato de que a aplicação necessita de mais comunicação gerando assim mais sobrecarga (*overhead*), juntamente com este fator está a complexidade do problema. Novamente, deve-se ter cuidado com a eficiência que com dois processadores foi maior do que com três, mas não necessariamente foi a que teve melhor tempo de execução. Os *speedups* não apresentaram anomalias e consequentemente as eficiências também não.

A seguir, as Tabelas 7.15 e 7.16 apresentam os resultados encontrados quando o algoritmo de Jacobi é executado usando o PVM no SP2.

**Tabela 7.15 - Resultados encontrados no SP2 utilizando PVM com 3 processadores.**

	300	600	900	1200	1500	1800	2100
3 proc.	0,0541	0,4400	0,783	0,8301	1,2056	1,8218	1,9355
Serial	0,0780	0,2688	0,6057	1,1595	1,6281	2,3461	3,1919
Speedup	1,4418	0,6109	0,7736	1,3968	1,3504	1,2878	1,6491
Eficiência	0,4806	0,2036	0,2579	0,4656	0,4501	0,4293	0,5497
$R_T(N)$ – sols/s	0,4527	1,8054	4,0581	1,2047	0,8295	0,5489	0,5167
$F_B(N)$ – Mflop	0,4527	1,8054	4,0581	7,2108	11,2635	16,2162	22,0689
$R_B(N,p)$ – Mflop/s	8,3678	4,1032	5,1828	8,6867	9,3427	8,9012	11,4022

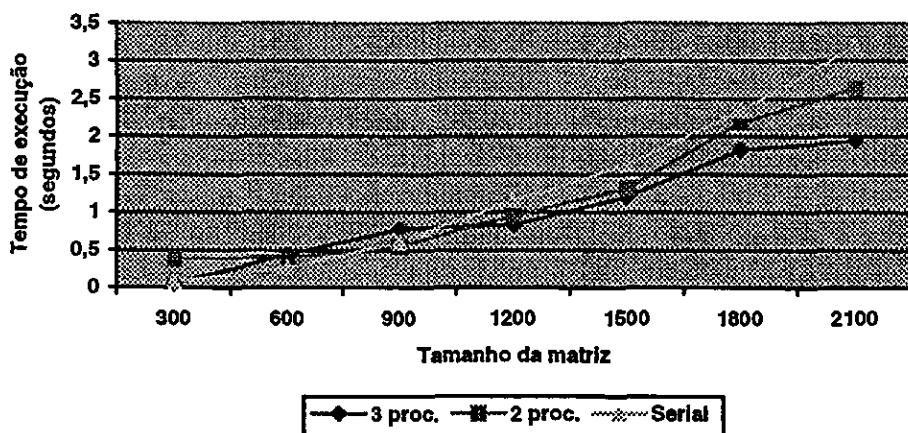
Tabela 7.16 - Resultados encontrados no SP2 utilizando PVMe com 2 processadores.

	300	600	900	1200	1500	1800	2100
2 proc.	0,3839	0,4002	0,5442	0,9486	1,3184	2,1782	2,6366
Serial	0,0780	0,2688	0,6057	1,1595	1,6281	2,3461	3,1919
Speedup	0,2032	0,6717	1,113	1,2223	1,2349	1,0771	1,2106
Eficiência	0,1016	0,3358	0,5565	0,6112	0,6175	0,5386	0,6053
$R_T(N)$ sols/s	0,4527	1,8054	4,0581	1,0542	0,7585	0,4591	0,3793
$F_B(N) - \text{Mflop}$	0,4527	1,8054	4,0581	7,2108	11,2635	16,2162	22,0689
$R_B(N,p) - \text{Mflop/s}$	1,1792	4,5112	7,457	7,6015	8,5433	7,4448	8,3702

O tempo de execução para dois processadores foi superior até o tamanho de 900x900, fato que se deve ao tamanho relativamente pequeno da matriz. Quando o porte aumenta para 1200 x 1200 o desempenho em 3 processadores é nitidamente melhor. O que fez com que o desempenho do *benchmark* aumentasse gradativamente foi a complexidade  $n^2$  do problema.

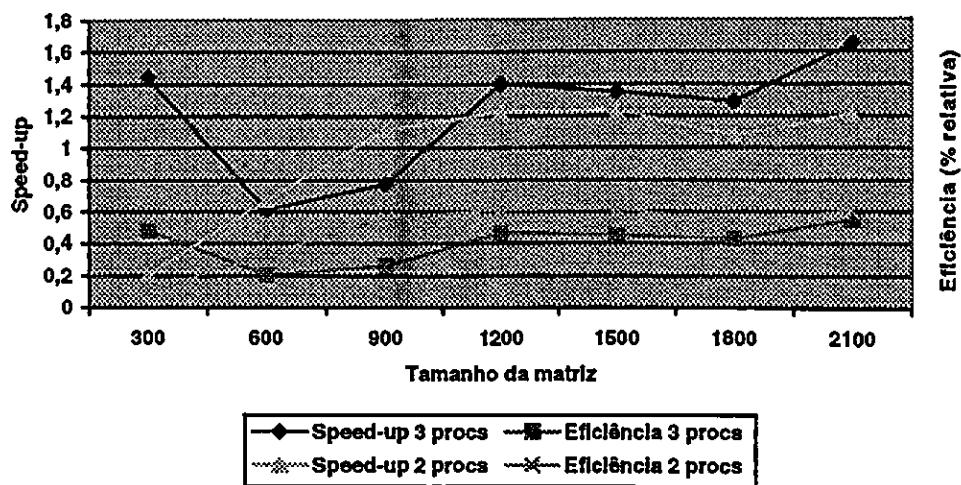
O Gráfico 7.15 mostra a variação na execução do algoritmo de Jacobi para 1, 2 e 3 processadores.

Gráfico 7.15 - Tempo de execução do algoritmo de Jacobi no SP2 utilizando PVMe.



Ainda no Gráfico 7.15 observa-se melhor a questão que envolve o tamanho das matrizes. Até 900 x 900 a sobrecarga na comunicação teve um fator decisivo no desempenho que nitidamente ficou melhor a partir da matriz de 1200 x 1200. O Gráfico 7.16 apresenta o *speedup* e a eficiência encontrados para 2 e 3 processadores conforme o tamanho do problema. Apesar do speedup ter sido melhor com 3 processadores ele está longe do ideal que seria o valor 3 ( $speedup = \text{número\_de\_processadores}$ ).

**Gráfico 7.16 - Speed-up e eficiência alcançados por 2 e 3 processadores no SP2 utilizando PVMe.**



As Tabelas 7.17 e 7.18 mostram os resultados obtidos utilizando-se o sistema distribuído (LASD-PC) com o MPI. O valor máximo para o tamanho da matriz foi de 1200 x 1200 sem alocação dinâmica, um teste com alocação dinâmica permitiu criar matrizes de até 2100 x 2100, mas o *ParkBench* não permite a utilização de alocação dinâmica pois impede a comparação futura dos resultados obtidos com algoritmos implementados em Fortran 77.

**Tabela 7.17 - Resultados encontrados no LASD-PC utilizando MPI com 3 processadores.**

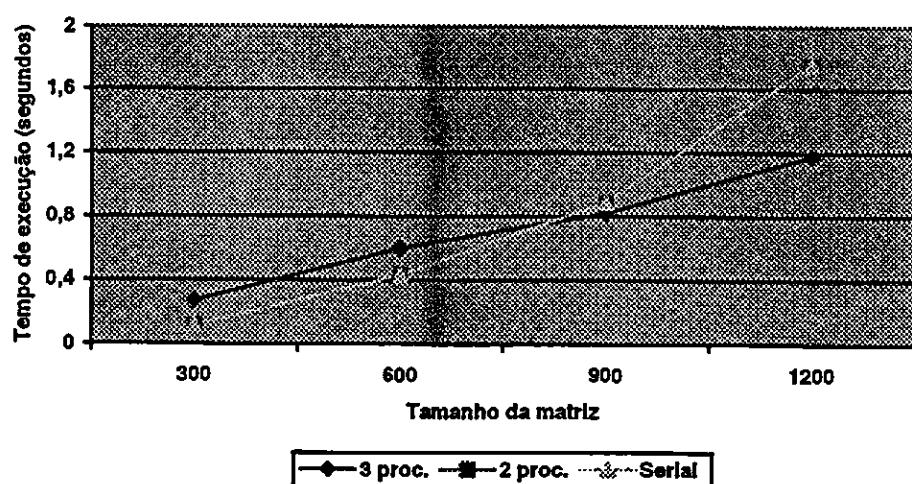
	300	600	900	1200
3 proc.	0,2688	0,5966	0,8177	1,1772
Serial	0,1159	0,4323	0,8874	1,7527
Speedup	0,4312	0,7246	1,0852	1,4889
Eficiência	0,2156	0,3623	0,5426	0,7445
$R_T(N)$ - sols/s	0,19	0,1	0,05	0,02
$F_B(N)$ - Mflop	0,4527	1,8054	4,0581	7,2108
$R_B(N,p)$ -Mflop/s	1,6842	3,0261	4,9628	6,1254

**Tabela 7.18 - Resultados encontrados no LASD-PC utilizando MPI com 2 processadores.**

	300	600	900	1200
2 proc.	0,3796	0,9556	1,0065	1,8197
Serial	0,1159	0,4323	0,8874	1,7527
Speedup	0,3053	0,4524	0,8817	0,9632
Eficiência	0,1527	0,2262	0,4409	0,4816
$R_T(N)$ - sols/s	0,14	0,07	0,03	0,02
$F_B(N)$ - Mflop	0,4527	1,8054	4,0581	7,2108
$R_B(N,p)$ -Mflop/s	1,1926	1,8893	4,0319	3,9626

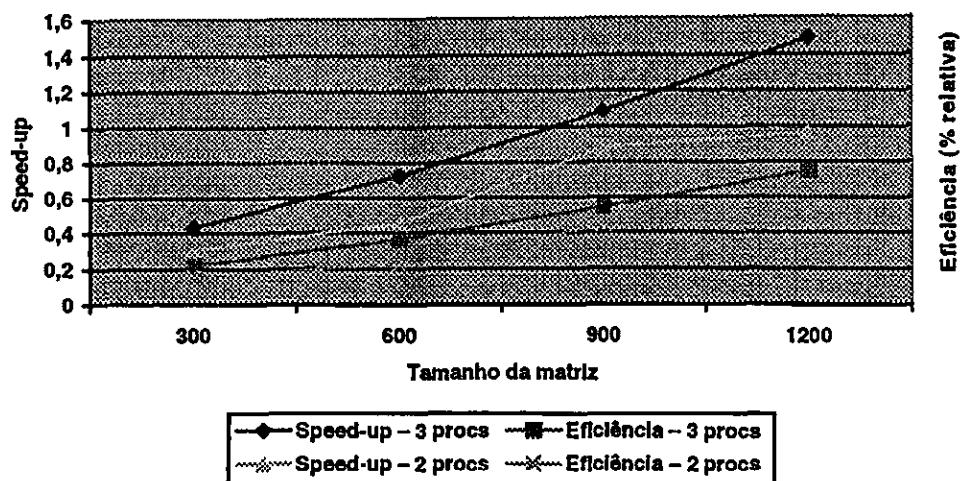
O desempenho do sistema distribuído foi muito baixo em relação a execução seqüencial. A única execução que teve um desempenho um pouco melhor, foi a execução em 3 processadores para uma matriz de 1200 x 1200. Apesar disso, o ganho foi tão pequeno que não valeria a pena investir em três máquinas para ter um resultado tão pouco eficiente. O baixo desempenho verificado pode ser justificado pela grande quantidade de comunicação do algoritmo em questão e da ineficiência da comunicação da rede *ethernet* de 10Mbits utilizada. O Gráfico 7.17 mostra mais claramente a velocidade de execução.

**Gráfico 7.17 - Tempo de execução do algoritmo de Jacobi no LASD-PC utilizando MPI.**



Quando o processamento com 2 processadores é comparado com o serial, o serial leva vantagem pois a execução do programa é muito rápida. Nota-se que para que haja um melhor desempenho o tamanho da matriz tem que ser igual ou superior a uma matriz de 900 x 900 em 3 processadores. Essa diferença é mais acentuada quando é utilizada uma matriz de 1200 x 1200, onde a distribuição de carga é mais notada. O Gráfico 7.18 mostra o *speedup* e a eficiência alcançados no sistema distribuído utilizando-se o MPI.

**Gráfico 7.18 - Speed-up e eficiência alcançados por 2 e 3 processadores no LASD-PC utilizando MPI.**



A eficiência para 2 processadores no Gráfico 7.18 é baixa considerando-se que ela deveria estar mais próxima de 1 para demonstrar um bom resultado. Com 3 processadores a eficiência só se torna atraente para uma matriz de tamanho de 1200 x 1200 já que está mais próxima de 1. O *speedup* para 2 processadores mostra que o desempenho é pior que do serial já que ele está abaixo de 1 e o ideal serial que ele estivesse próximo de 2. O *speedup* para 3 processadores também torna-se melhor utilizando-se uma matriz de 1200 x 1200, o que ainda é relativamente baixo, pois o ideal seria que ele estivesse próximo do valor 3.

A seguir as Tabelas 7.19 e 7.20 mostram os resultado obtidos utilizando-se o sistema distribuído com a biblioteca de passagem de mensagem PVM.

**Tabela 7.19 – Resultados encontrados no LASD-PC utilizando PVM com 3 processadores.**

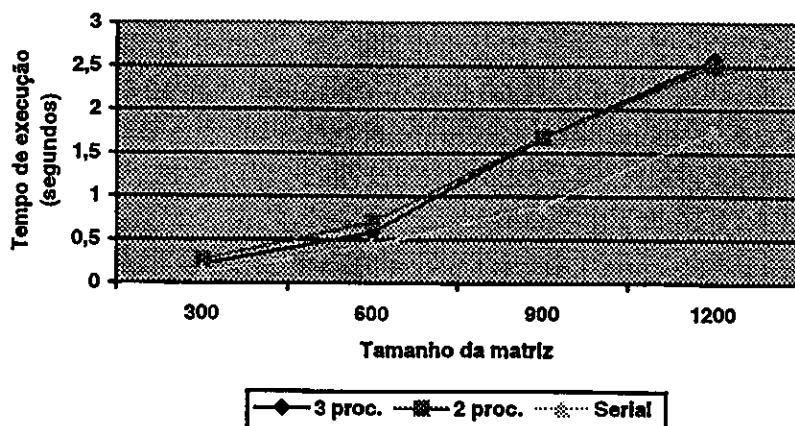
	300	600	900	1200
3 proc.	0,212	0,5839	1,6774	2,5729
Serial	0,1159	0,4323	0,8874	1,7527
Speedup	0,5467	0,7404	0,529	0,4906
Eficiência	0,2734	0,3702	0,265	0,2453
$R_T(N)$ - sols/s	0,14	0,07	0,03	0,02
$F_B(N)$ - Mflop	0,4527	1,8054	4,0581	7,2108
$R_B(N,p)$ - Mflop/s	2,1354	3,092	2,4193	2,8026

Tabela 7.20 - Resultados encontrados no LASD-PC utilizando PVM com 2 processadores.

	300	600	900	1200
2 proc.	0,258	0,6994	1,6797	2,4992
Serial	0,1159	0,4323	0,8874	1,7527
Speedup	0,5467	,6181	0,5283	0,7013
Eficiência	0,2734	0,3091	0,2642	0,3507
$R_T(N) - \text{sols/s}$	0,14	0,07	0,03	0,02
$F_B(N) - \text{Mflop}$	0,4527	1,8054	4,0581	7,2108
$R_B(N,p) - \text{Mflop/s}$	1,7547	2,5814	2,416	2,8852

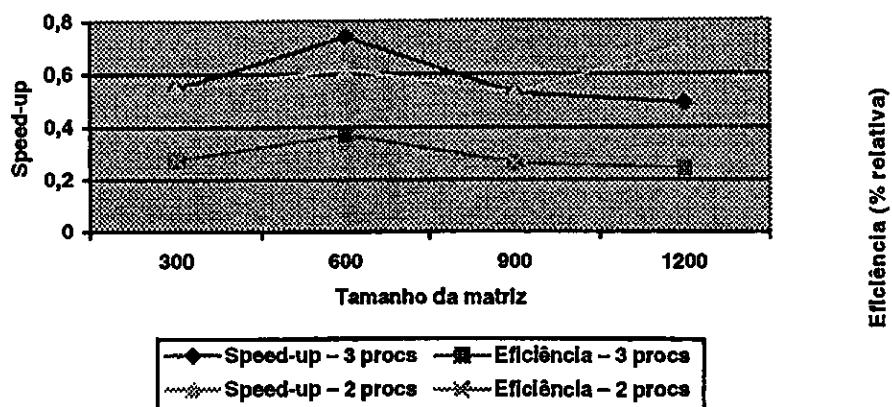
O tempo de execução em 2 e 3 processadores foi muito pobre fazendo com que a execução serial tive-se um melhor desempenho. Gráfico 7.17 mostra mais claramente os resultados encontrados no tempo de execução. A utilização do PVM mostrou-se neste exemplo, pior que a utilização do MPI na mesma plataforma de hardware, uma vez que em nenhum caso foi obtido um bom resultado.

Gráfico 7.19 - Tempo de execução do algoritmo de Jacobi no LASD-PC utilizando PVM.



A execução serial foi nitidamente melhor que a execução em paralelo. Se o tamanho da matriz pudesse ser aumentado com certeza num determinado ponto essa situação se inverteria. A quantidade maior de comunicação nesse algoritmo foi o responsável pelo baixo desempenho. Neste caso o MPI (*Mpich*) teve um melhor desempenho na comunicação devido a sua capacidade de trabalhar bem com a comunicação de matrizes, vetores e estruturas complexas. O Gráfico 7.20 apresenta o *speedup* e a eficiência alcançadas na execução.

**Gráfico 7.20 - Speed-up e eficiência alcançados por 2 e 3 processadores no LASD-PC utilizando PVM.**



Verificando-se o eixo-y nota-se que todas as linhas estão abaixo do valor 1, isso indica que o desempenho alcançado pela eficiência e pelo speedup foi inferior ao desempenho alcançado na execução serial.

As próximas tabelas (Tabela 7.21 e 7.22) apresentam os resultados encontrados na arquitetura *Cray*, mais precisamente na máquinas *dolphin* cujas características foram descritas no Capítulo 5.

**Tabela 7.21 – Resultados encontrados no Cray utilizando 3 processadores.**

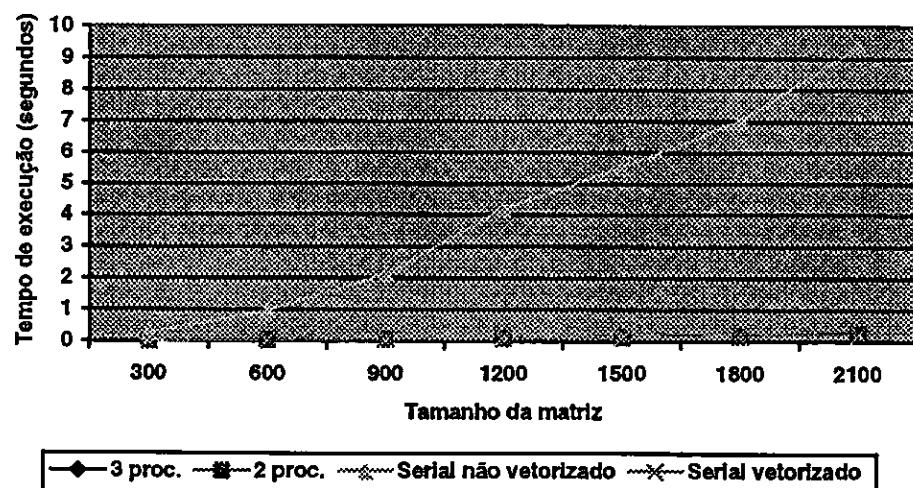
	300	600	900	1200	1500	1800	2100
3 proc.	0,0139	0,0325	0,0690	0,0948	0,1332	0,1687	0,2532
Serial vetorizado	0,0112	0,0310	0,0680	0,0930	0,1288	0,1650	0,2174
Serial ñ vetorizado	0,2648	0,9728	2,1307	4,2008	5,5196	7,0303	9,3717
Speedup	19,0504	29,9323	30,8797	44,3122	41,4384	41,6734	37,013
Eficiência	6,3501	9,9774	10,2932	14,7707	13,8128	13,8911	12,3377
R <sub>T</sub> (N) – sols/s	3,7764	1,028	0,4693	0,238	0,1812	0,1422	0,1067
F <sub>B</sub> (N) – Mflop	0,4527	1,8054	4,0581	7,2108	11,2635	16,2162	22,0689
R <sub>B</sub> (N,p) – Mflop/s	32,57	55,551	58,813	76,0633	84,5608	96,1245	87,16

**Tabela 7.22 - Resultados encontrados no Cray utilizando 2 processadores.**

	300	600	900	1200	1500	1800	2100
2 proc.	0,0138	0,0326	0,0678	0,0963	0,1386	0,1706	0,2649
Serial vetorizado	0,0112	0,0310	0,0680	0,0930	0,1288	0,1650	0,2174
Serial ñ vetorizado	0,2648	0,9728	2,1307	4,2008	5,5196	7,0303	9,3717
Speedup	19,1884	29,8405	31,4263	43,622	39,824	41,2093	35,3783
Eficiência	9,5942	14,9202	15,7131	21,811	19,912	20,6047	17,6892
R <sub>T</sub> (N) – sols/s.	3,7764	1,028	0,4693	0,238	0,1812	0,1422	0,1067
F <sub>B</sub> (N) – Mflop	0,4527	1,8054	4,0581	7,2108	11,2635	16,2162	22,0689
R <sub>B</sub> (N,p) – Mflop/s	32,8	55,38	59,854	74,8785	81,2662	95,0539	83,3103

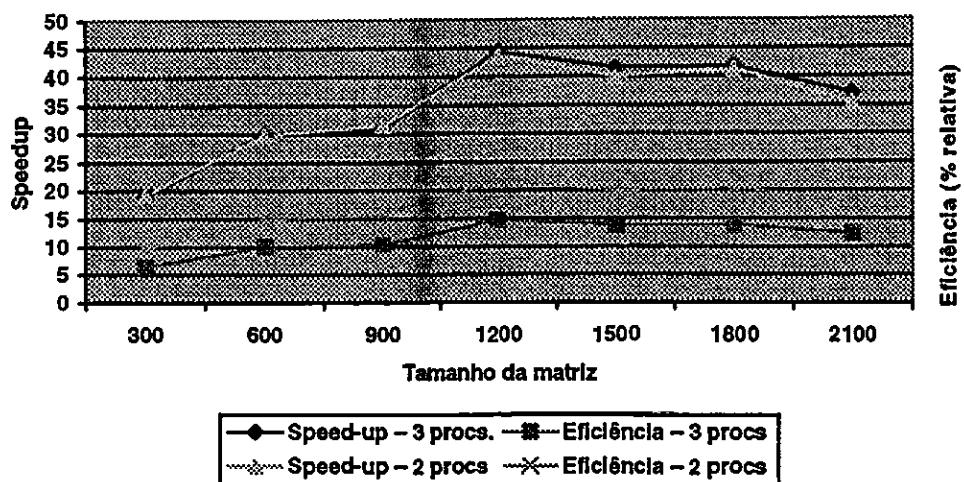
Novamente, nota-se um desempenho muito pobre no tempo de execução sem vetorização. Devido a implementação fazer uso exaustivo de vetores, a execução com vetorização teve um ganho excelente como pode ser visto no Gráfico 7.21. Da mesma forma que na aplicação anteriores a vetorização em um único processador teve um desempenho superior aos demais, relembrando que esse ganho deve-se a utilização de registradores vetoriais, *pipelines* vetoriais e cache local. A comunicação entre processadores envolve acesso a memória RAM criando problemas de sincronização e limitando a utilização dos registradores vetoriais e dos *pipelines* vetoriais a alguns dados.

**Gráfico 7.21 - Tempo de execução do algoritmo de Jacobi na sistema Cray.**



O Gráfico 7.20 mostra o *speedup* e a eficiência alcançada neste tipo de arquitetura. O baixo desempenho da execução sem vetorização novamente foi o responsável pela anomalia de *speedup* e pela anomalia da eficiência.

**Grafico 7.22 - Speedup e eficiência alcançados no sistema Cray.**



### Comparação entre arquiteturas

As Tabelas 7.23 e 7.24 resumem o desempenho do *benchmark* para o algoritmo de multiplicação de matrizes para 3 e 2 processadores respectivamente.

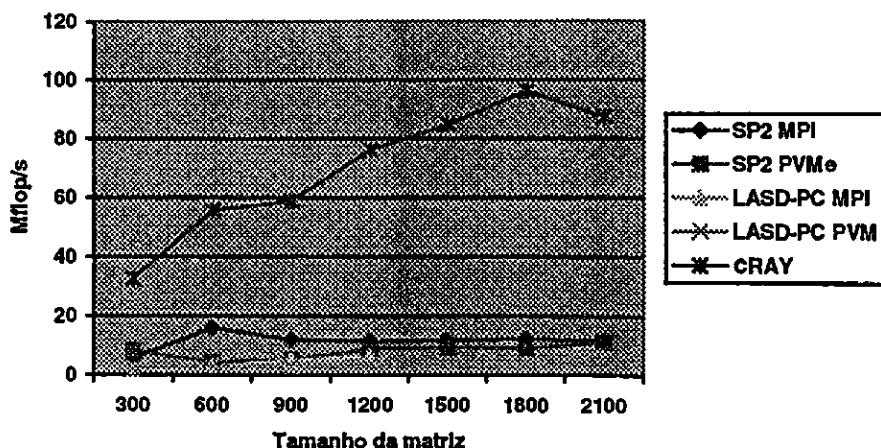
Observando-se o Gráfico 7.23, nota-se o comportamento não linear da aplicação em todas as arquiteturas.

**Tabela 7.23 – Desempenho do *benchmark* para 3 processadores.**

	300	600	900	1200	1500	1800	2100
SP2 MPI	5,6236	15,9488	11,9391	11,3951	11,7979	12,089	11,8167
SP2 PVMe	8,3678	4,1032	5,1828	8,6867	9,3427	8,9012	11,4022
LASD-PC MPI	1,6842	3,0261	4,9628	6,1254	-	-	-
LASD-PC PVMe	2,1354	3,092	2,4193	2,8026	-	-	-
Cray	32,57	55,551	58,813	76,0632	84,5608	96,1244	87,1599

Devido a aplicação trabalhar intensivamente com vetores e a mesma ter sido gerada com um nível agressivo de vetorização, a arquitetura *Cray* teve nítida vantagem sobre as outras como pode ser visto no Gráfico 7.23.

**Grafico 7.23 - Comparaçao entre arquiteturas com 3 processadores executando o algoritmo de Jacobi.**



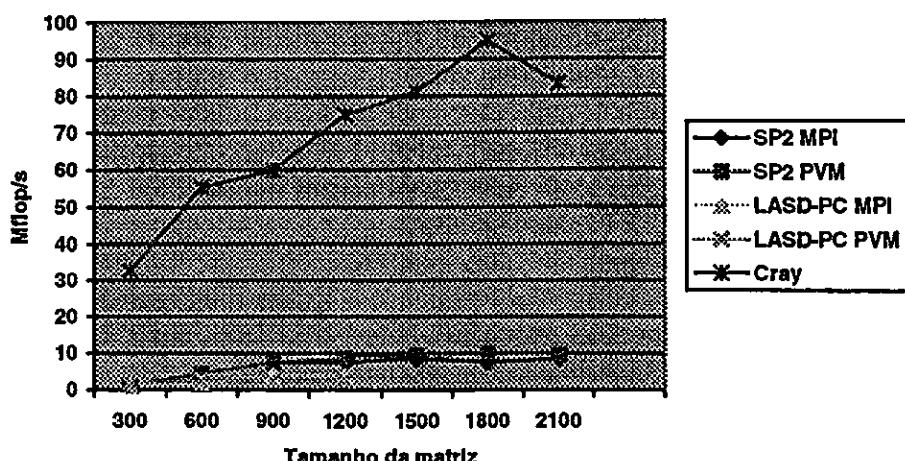
A Tabela 7.24 mostra um resumo do RB(N) para 2 processadores.

**Tabela 7.24 – Desempenho do *benchmark* para 3 processadores.**

	300	600	900	1200	1500	1800	2100
SP2 MPI	1,1792	4,5112	7,457	8,7648	9,4113	9,825	9,6485
SP2 PVMe	1,1792	4,5112	7,457	7,6015	8,5433	7,4448	8,3702
LASD-PC MPI	1,1926	1,8893	4,0319	3,9626	-	-	-
LASD-PC PVM	1,7547	2,5814	2,416	2,8852	-	-	-
Cray	32,8	55,38	59,8539	74,8785	81,2662	95,0539	83,3103

Através do Gráfico 7.24 nota-se que o comportamento foi semelhante a execução com 3 processadores. E como era de se esperar, a aplicação vetorial levou nítida vantagem sobre as demais arquiteturas.

**Gráfico 7.24 - Comparação entre arquiteturas com 2 processadores executando o algoritmo de Jacobi.**



### 7.2.3 Resultado do programa de multiplicação de matrizes

Neste seção são apresentados os resultados obtidos na execução do algoritmo tradicional de multiplicação de matrizes apresentado no Capítulo 6.

As tabelas 7.25 e 7.26 mostram os resultados encontrados na execução do programa no SP2 utilizando o MPI.

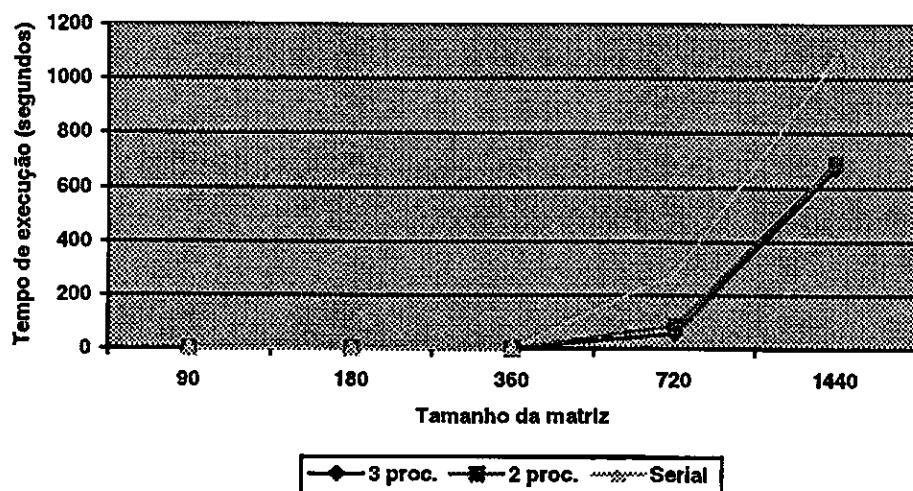
**Tabela 7.25 - Resultados encontrados no SP2 utilizando MPI com 3 processadores.**

	90	180	360	720	1440
3 proc.	0,0167	0,1068	0,6697	58,621	667,416
Serial	0,0431	0,3523	2,7995	278,723	1072,73
Speedup	2,5808	3,2987	4,1802	4,7547	1,6073
Eficiência	0,8603	1,0996	1,3934	1,5849	0,5358
$R_T(N)$ – sols/s.	1,5803	1,0474	0,7455	0,5354	0,4174
$F_B(N)$ – Mflop	2,4386	18,5014	143,9877	1135,818	9020,00
$R_B(N,p)$ Mflop/s	146,024	173,2341	215,0033	19,3756	13,5148

**Tabela 7.26 - Resultados encontrados no SP2 utilizando MPI com 2 processadores.**

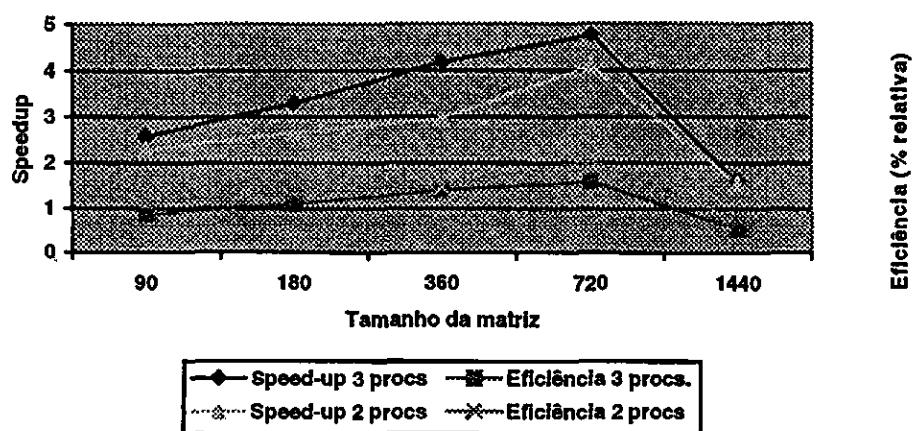
	90	180	360	720	1440
2 proc.	0,0188	0,1321	0,9427	67,7456	689,2
Serial	0,0431	0,3523	2,7995	278,723	1072,73
Speedup	2,2926	2,6669	2,9697	4,1143	1,5565
Eficiência	1,1463	1,3335	1,4849	2,0572	0,7783
$R_T(N)$ – sols/s	1,0542	1,5011	1,1226	0,8752	0,6945
$F_B(N)$ – Mflop	2,4386	18,5014	143,9877	1135,818	9020,00
$R_B(N,p)$ – Mflop/s	129,7128	140,056	152,7397	16,7659	13,0876

Os três primeiros valores de tempo de execução são baixos devido a comunicação com estruturas mais complexas através do MPI ser mais eficiente. Nota-se o crescimento do tempo de execução de acordo com a complexidade do problema que é de  $n^3$ . Logicamente, isso leva a uma queda de desempenho observada tanto no *speedup* quanto no desempenho do *benchmark* ( $R_B(N)$ ) a partir de uma matriz de 1440 x 1440. Essa queda deve-se a grande quantidade de trocas de páginas entre a memória RAM e a memória cache. Nota-se que para cada matriz de 1440 x 1440 é exigido uma memória de aproximadamente 16 Mbytes e o SP2 possui uma memória cache de 256 Kbytes para o nó *wide* e 128 Kbytes para os nós *thin*. Dessa forma, uma matriz de 720 x 720 acessa muito menos a memória RAM do que uma matriz de 1440 x 1440. Essa diferença de memória exigida pelas matrizes, aliada a complexidade do problema, causa uma queda brusca de desempenho na matriz de 1440 x 1440. O Gráfico 7.25 mostra mais claramente os tempos de execução.

**Gráfico 7.25 - Tempo de execução do algoritmo de multiplicação de matrizes no SP 2 utilizando MPI.**

Pelo tamanho das matrizes o tempo de execução é semelhante até um certo ponto, mais precisamente até uma matriz de  $360 \times 360$ . A partir da matriz de  $720 \times 720$  a distribuição de trabalho entre os processadores já é nitidamente notada dada a diferença dos tempos de execução entre a versão paralela e a versão serial.

**Gráfico 7.26 - Speedup e eficiência alcançados por 2 e 3 processadores no SP2 utilizando MPI.**



No Gráfico 7.26 é apresentado o *speedup* e a eficiência alcançados por 2 e 3 processadores. Para um problema de tamanho  $1440 \times 1440$ , há uma queda relativamente grande de *speedup* tanto para 2 como para 3 processadores, essa queda também deve-se a anomalia gerada nas matrizes de  $180 \times 180$  a  $720 \times 720$ . Essa anomalia é gerada pelo tamanho relativamente pequeno dessas matrizes ( $90 \times 90$ ,  $180 \times 180$  e  $720 \times 720$ ) o que leva a um maior uso da memória cache. Como já mencionado, cada matriz de  $720 \times 720$  ocupa aproximadamente 4 Mbytes de memória RAM e cada matriz de  $1440 \times 1440$  ocupa por volta de 16 Mbytes. A queda de desempenho não é maior porque todas as matrizes são armazenadas na memória. A grande quantidade de memória RAM disponível em cada nó do SP2 (128 Mbytes cada nó *thin* e 256 Mbytes o nó *wide*) permite que as três matrizes sejam armazenadas em cada um dos nós. Se a memória não fosse suficiente haveria acesso a disco e a queda seria ainda mais acentuada..

As Tabelas 7.27 e 7.28 apresentam os resultados encontrados na arquitetura SP2 utilizando-se a biblioteca de passagem de mensagem PVMe.

Tabela 7.27 - Resultados encontrados no SP2 utilizando PVM com 3 processadores.

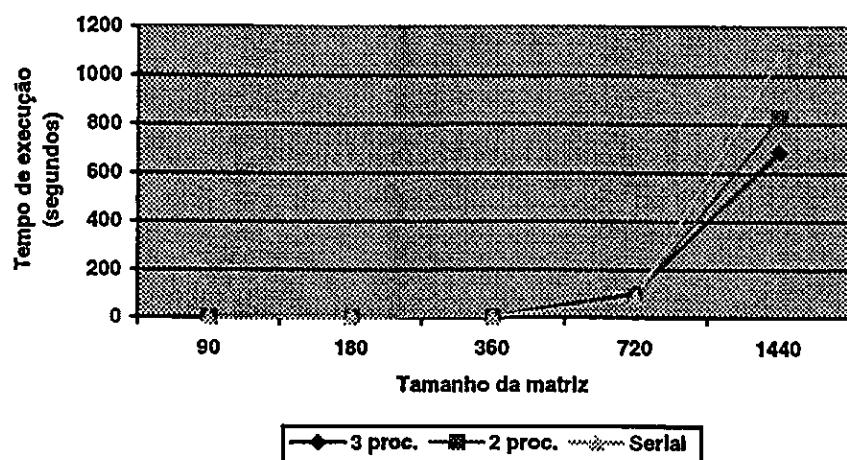
	90	180	360	720	1440
3 proc.	0,3442	0,4303	1,4285	102,144	684,692
Serial	0,0431	0,3523	2,7995	278,723	1072,73
Speedup	0,1252	0,8187	1,9597	2,728726	1,5667
Eficiência	0,0417	0,2729	0,6532	0,909575	0,5222
$R_T(N)$ sols/s	0,197	0,1009	0,05	0,03	0,0128
$F_B(N) \sim \text{Mflop}$	2,4386	18,5014	143,9877	1135,818	9020,00
$R_B(N,p) \sim \text{Mflop/s}$	7,0848	42,9965	100,7964	11,1198	13,1738

Tabela 7.28 - Resultados encontrados no SP2 utilizando PVM com 2 processadores.

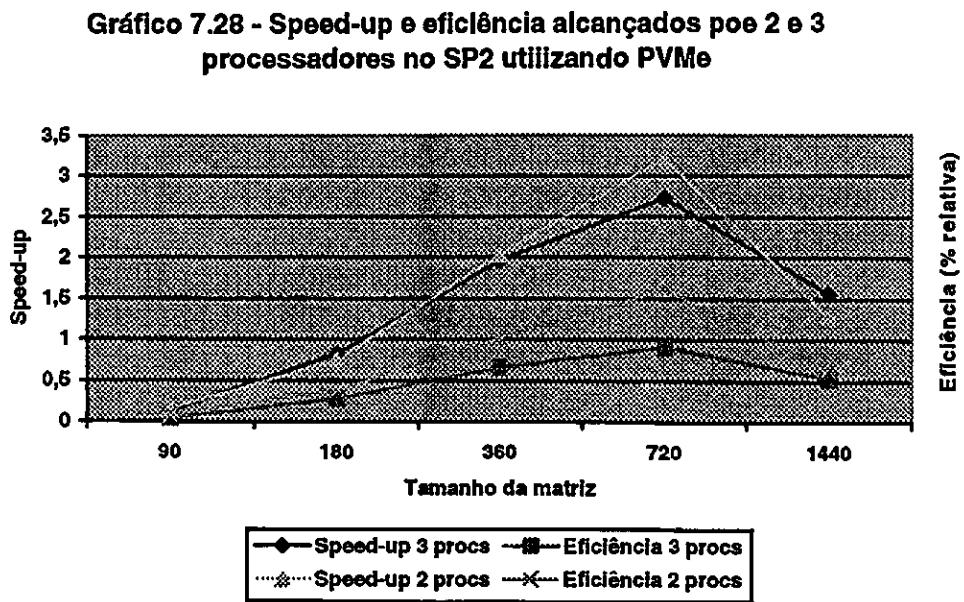
	90	180	360	720	1440
2 proc.	0,2829	0,4703	1,3750	86,6146	826,723
Serial	0,0431	0,3523	2,7995	278,723	1072,73
Speedup	0,1524	0,7491	2,036	3,218	1,2976
Eficiência	0,0762	0,3746	1,018	1,609	0,6488
$R_T(N) - \text{sols/s.}$	1,0542	1,5011	1,1226	0,8752	0,6945
$F_B(N) \sim \text{Mflop}$	2,4386	18,5014	143,9877	1135,818	9020,00
$R_B(N,p) \sim \text{Mflop/s}$	8,62	39,3396	104,7183	13,1135	10,91055

O comportamento utilizando o PVM no SP2 foi semelhante ao do MPI com uma pequena vantagem para o MPI. Isso deve-se ao fato de que o PVM não é otimizado para realizar a comunicação de matrizes e estruturas mais complexas. Através do Gráfico 7.27 observa-se que a distribuição de carga é notada a partir de uma matriz de 720 x 720 onde o tempo de execução em paralelo passa a ser melhor que o tempo de execução seqüencial. O tempo de execução foi consideravelmente melhor a partir de uma matriz de 1440 x 1440.

Gráfico 7.27 - Tempo de execução do algoritmo de multiplicação de matrizes no SP2 utilizando PVM



O Gráfico 7.28 mostra o speedup e a eficiência alcançados pela execução do algoritmo de multiplicação de matrizes no SP2 utilizando o PVMe.



Os Gráficos 7.28 e 7.26 tiveram comportamentos semelhantes pelos mesmos motivos, ou seja, a memória cache foi a grande responsável pelo desempenho. Observa-se que no  $R_B(N)$ , também existe uma queda acentuada a partir de uma matriz de 720 x 720. Isso demonstra a influência da memória cache na obtenção de desempenho.

As Tabelas 7.29 e 7.30 mostram a execução do algoritmo de multiplicação de matrizes no sistema distribuído utilizando o MPI. Polo fato de que só foi utilizada alocação estática de memória, no sistema distribuído só foi possível multiplicar matrizes de até 360 x 360. A alocação estática é utilizada pois o NAS não permite o uso de alocação dinâmica.

**Tabela 7.29 - Resultados encontrados no LASD-PC utilizando MPI com 3 processadores.**

	90	180	360
3 proc.	0,4136	5,2911	39,4338
Serial	0,097	1,0429	17,9138
Speedup	0,2345	0,1971	0,4543
Eficiência	0,0782	0,0657	0,1514
$R_T(N)$ – sols/s.	0,19	0,1	0,05
$F_B(N)$ – Mflop	2,4386	18,5014	143,9877
$R_B(N,p)$ Mflop/s	5,896	3,4967	3,6514

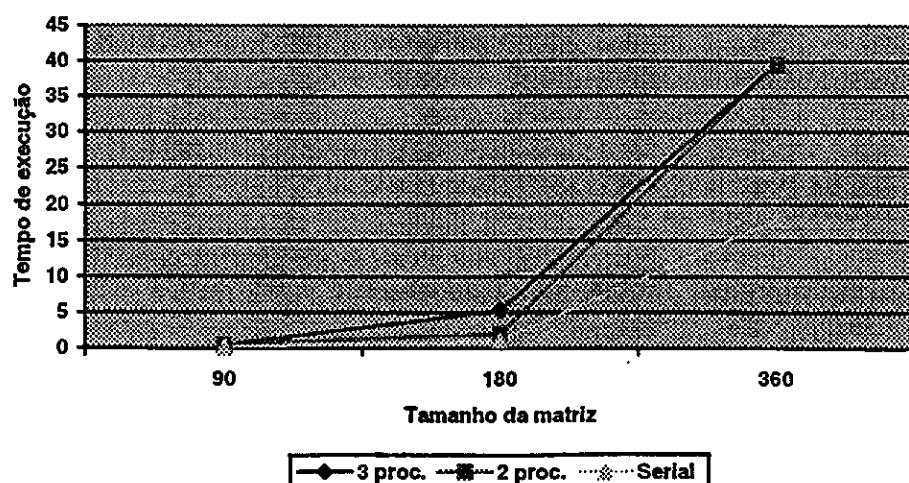
Tabela 7.30 - Resultados encontrados no LASD-PC utilizando MPI com 2 processadores.

	90	180	360
2 proc.	0,5753	2,0973	39,4338
Serial	0,097	1,0429	17,9138
Speedup	0,1686	0,5308	0,4542
Eficiência	0,0843	0,2654	0,2271
$R_T(N)$ - sols/s	0,14	0,07	0,03
$F_B(N)$ - Mflop	2,4386	18,5014	143,9877
$R_B(N,p)$ Mflop/s	4,2388	8,8215	3,6514

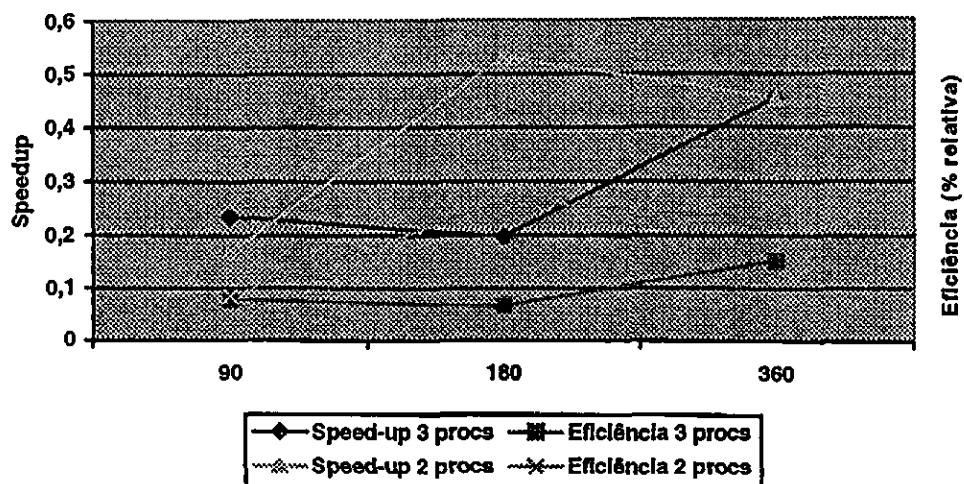
Para o sistema distribuído, este tipo de algoritmo executado sobre o MPI, tem um desempenho pobre se comparado com o PVM (Tabelas 7.31 e 7.32) além disso a execução seqüencial também foi mais rápida. Em relação ao PVM, seu baixo desempenho deve-se ao fato do MPI estar executando exatamente o mesmo código nos três processadores, ou seja, uma cópia de cada programa é enviada para cada processador e executada. O código é exatamente o mesmo até os programas reconhecerem quem é o mestre e quem são os escravos. No PVM isso não acontece, os programas escravos quando iniciados já sabem que são escravos e só possuem a parte do código necessária para a realizar a multiplicação.

Devido ao tamanho relativamente pequeno das matrizes, a diferença de velocidade dos processadores e a sobrecarga na comunicação (rede de velocidade baixa) a execução serial teve nítida vantagem sobre a paralela como mostrado pelo Gráfico 7.29.

Gráfico 7.29 - Tempo de execução do algoritmo do trapézio composto no LASD-PC utilizando MPI.



**Gráfico 7.30 - Speedup e eficiência alcançados por 2 e 3 processadores no LASD-PC utilizando MPI.**



O Gráfico 7.30 de *speedup* e eficiência mostra o desempenho pobre do algoritmo no sistema distribuído já que os valores para o *speedup* estão abaixo de zero indicando que a velocidade de processamento foi melhor em 1 processador (sequencial). E a eficiência não chegou a 0,3 que indica uma eficiência de apenas 30%.

A seguir, nas Tabelas 7.31 e 7.32 são mostrados os resultados conseguidos no sistema distribuído utilizando o PVM.

**Tabela 7.31 – Resultados encontrados no LASD-PC utilizando PVM com 3 processadores.**

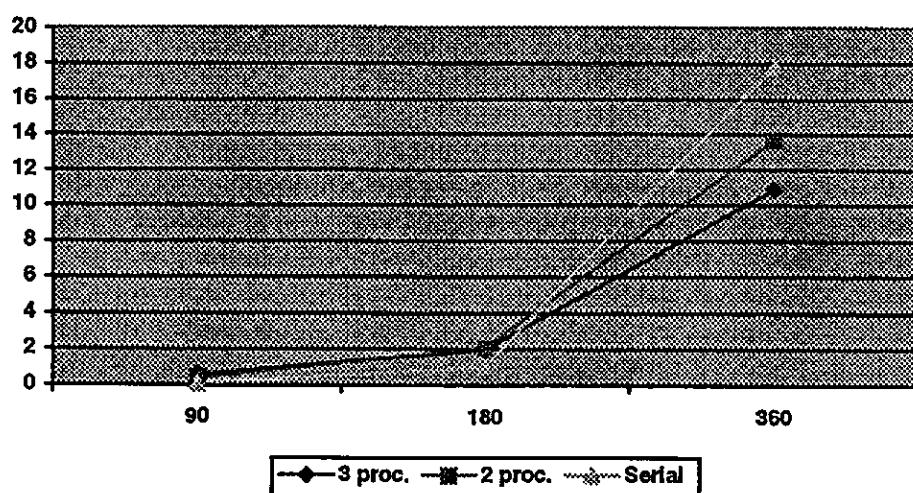
	90	180	360
3 proc.	0,5753	1,9646	10,8807
Serial	0,097	1,0429	17,9138
Speedup	0,1686	0,5308	1,6463
Eficiência	0,0843	0,2654	0,8231
$R_T(N)$ – sols/s.	0,19	0,1	0,05
$F_B(N)$ – Mflop	2,4386	18,5014	143,9877
$R_B(N,p)$ – Mflop/s	4,2388	9,4174	13,2333

**Tabela 7.32 - Resultados encontrados no LASD-PC utilizando PVM com 2 processadores.**

	90	180	360
2 proc.	0,4052	2,066	13,6628
Serial	0,097	1,0429	17,9138
Speedup	0,2393	0,5047	1,3111
Eficiência	0,119694	0,252396	0,6555
$R_T(N)$ – sols/s	0,14	0,07	0,03
$F_B(N)$ – Mflop	2,4386	18,5014	143,9877
$R_B(N,p)$ – Mflop/s	6,0183	8,9552	10,5387

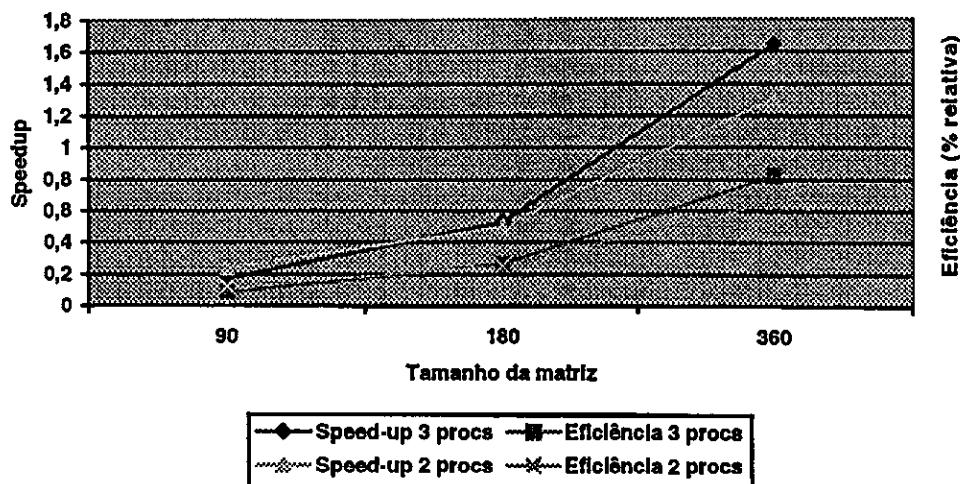
Observando-se o Gráfico 7.31 e analisando-se as Tabelas que apresentam os resultado (7.31 e 7.32), vê-se a nítida vantagem do PVM sobre o MPI neste tipo de aplicação. Isso ocorreu porque apesar da programação usar a abordagem SPMD, todos os processadores não executam exatamente o mesmo código (como o MPI), ou seja, os escravos quando iniciados já sabem que são escravos, e a parte semelhante dos programas só é encontrada no algoritmo de multiplicação de matrizes. Nota-se também, que a execução em paralelo é mais rápida a partir de uma matriz de 360 x 360 onde a distribuição de carga passou a ter mais influência que a velocidade de comunicação.

**Gráfico 7.31 - Tempo de execução do algoritmo de multiplicação de matrizes no LASD-PC utilizando PVMe.**



O Gráfico 7.32 mostra o *speedup* e a eficiência alcançados no sistema distribuído utilizando o PVM. Vê-se que o desempenho só se torna razoável a partir de uma matriz 360 x 360. Se uma matriz maior for considerada, com certeza esse desempenho iria aumentar.

**Gráfico 7.32 - Speed-up e eficiência alcançados por 2 e 3 processadores no LASD-PC usando PVM.**



A seguir são mostradas as Tabelas 7.33 e 7.34 que exibem os resultados encontrados na arquitetura *Cray*.

**Tabela 7.33 – Resultados encontrados no *Cray* utilizando 3 processadores.**

	90	180	360	720	1440
3 proc.	0,0027	0,0162	0,1236	0,9731	7,6809
Serial vetorizado	0,0035	0,0160	0,1229	0,9722	7,6719
Serial não vetorizado	14,55	29,17	58,29	116,59	233,16
Speedup	5388,8889	1800,6173	471,6019	119,813	30,3558
Eficiência	2694,4445	900,3087	235,801	59,907	15,1779
$R_T(N)$ – sols/s	0,19	0,1	0,05	0,02	0,01
$F_B(N)$ – Mflop	2,4386	18,5014	143,9877	1135,818	9020,00
$R_B(N,p)$ – Mflop/s	903,1852	1142,0617	1164,949	1164,949	1167,2161

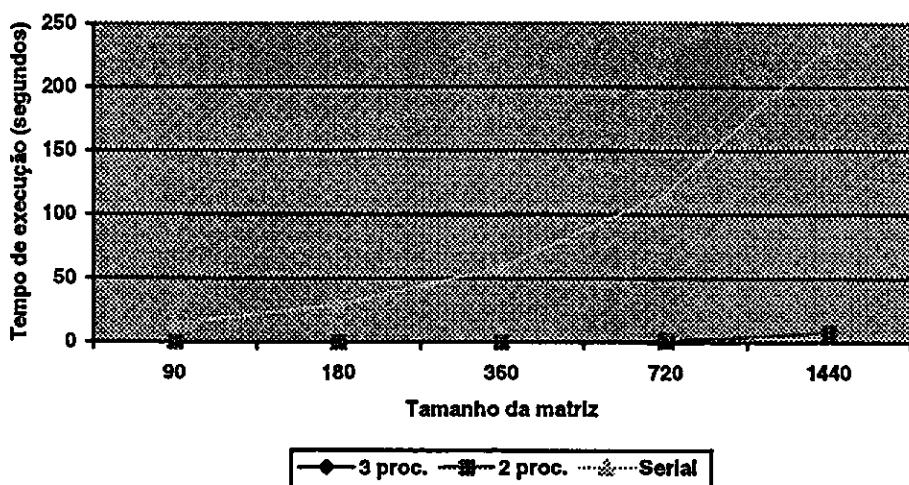
**Tabela 7.34 - Resultados encontrados no *Cray* utilizando 2 processadores.**

	90	180	360	720	1440
2 proc.	0,0029	0,0171	0,1222	0,9706	7,6809
Serial vetorizado	0,0035	0,0160	0,1229	0,9722	7,6719
Serial não vetorizado	14,55	29,17	58,29	116,59	233,16
Speedup	5017,2414	1705,848	477,0049	120,1216	30,3558
Eficiência	2508,6207	852,924	238,5025	60,0608	15,1779
$R_T(N)$ – sols/s	0,14	0,07	0,03	0,02	0,01
$F_B(N)$ – Mflop	2,4386	18,5014	143,9877	1135,818	9020,00
$R_B(N,p)$ – Mflop/s	840,8966	1081,9532	1178,2954	1178,2954	1170,2225

Devido à vetorização alcançou-se um caso raro de eficiência. Como a multiplicação tradicional de matrizes exige muitas operações vetoriais o desempenho desta arquitetura foi muito bom. A

execução com vetorização utilizando um único processador, novamente foi mais rápido que com 2 e 3 processadores. Esse resultado deve-se novamente ao fato de que com um único processador em uso, a utilização de registradores e *pipelines* vetoriais é mais eficiente e não exige comunicação entre os processadores. O Gráfico 7.27 mostra mais claramente o tempo de execução.

**Gráfico 7.33 - Tempo de execução do algoritmo de multiplicação de matrizes no sistema Cray.**



Para 2 e 3 processadores o tempo de execução não teve uma diferença que possa ser notada. A diferença extrema entre as versões vetorizadas e a seqüencial é explicada pelo fato que quando o compilador encontra uma multiplicação tradicional de matrizes, ele o substitui por uma biblioteca própria da arquitetura *Cray* que é totalmente vetorizável [Cra98a]. Utilizando o *profview* pode-se observar em termos estatísticos o quanto essa biblioteca influencia no tempo de execução. A Figura 7.7 mostra um gráfico que representa a execução da aplicação sem vetorização.

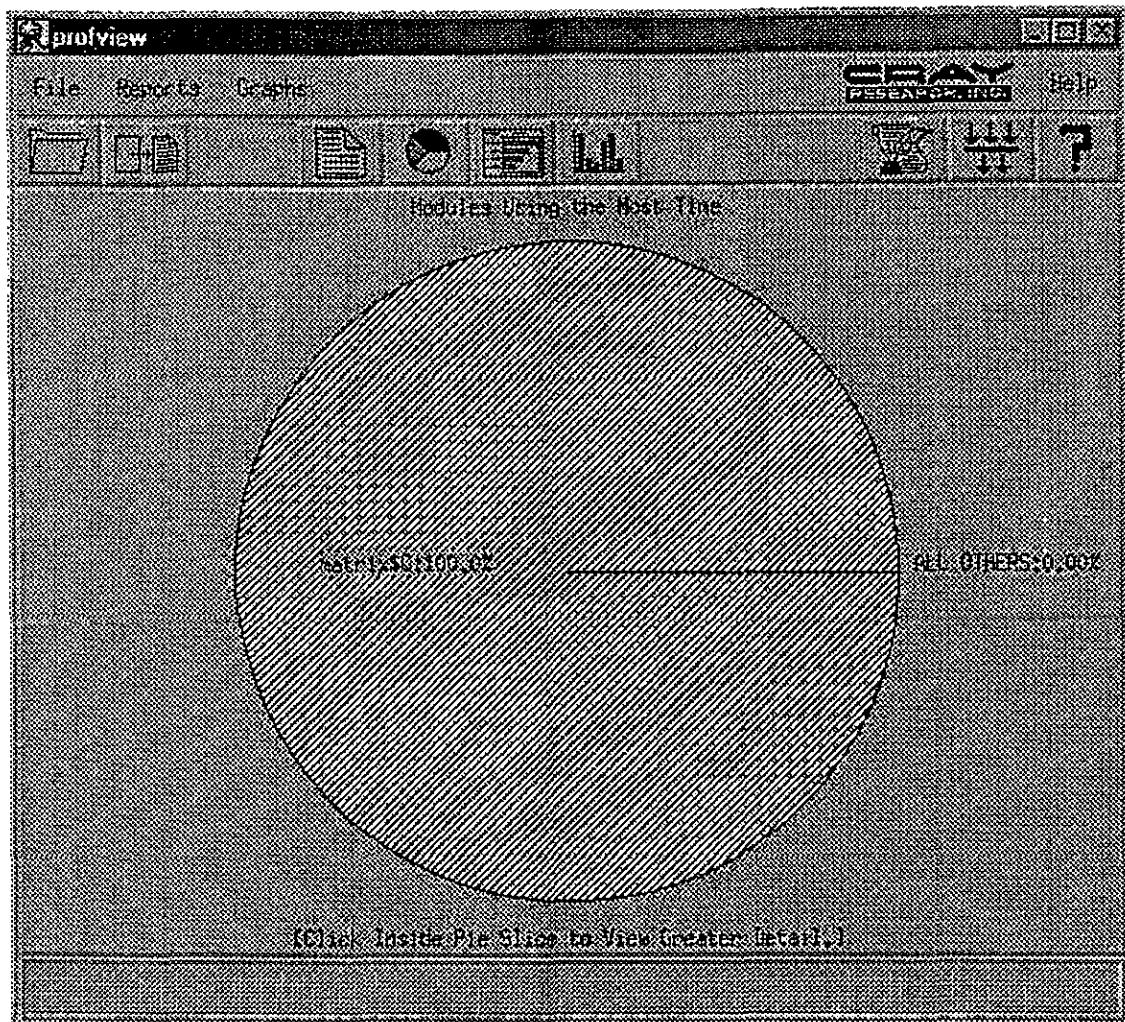


Figura 7.7 - Execução do algoritmo de multiplicação de matrizes sem vetorização.

No centro do gráfico de Pizza vê-se a porcentagem da aplicação que não foi vetorizada, ou seja, 100%. A Figura 7.8, mostra a porcentagem que a biblioteca inserida pelo compilador tem dentro da execução total do programa observa-se que 78,3% do tempo total é gasto pela biblioteca, e o restante que inclui inicialização de variáveis, matrizes e demais operações não vetorizadas.

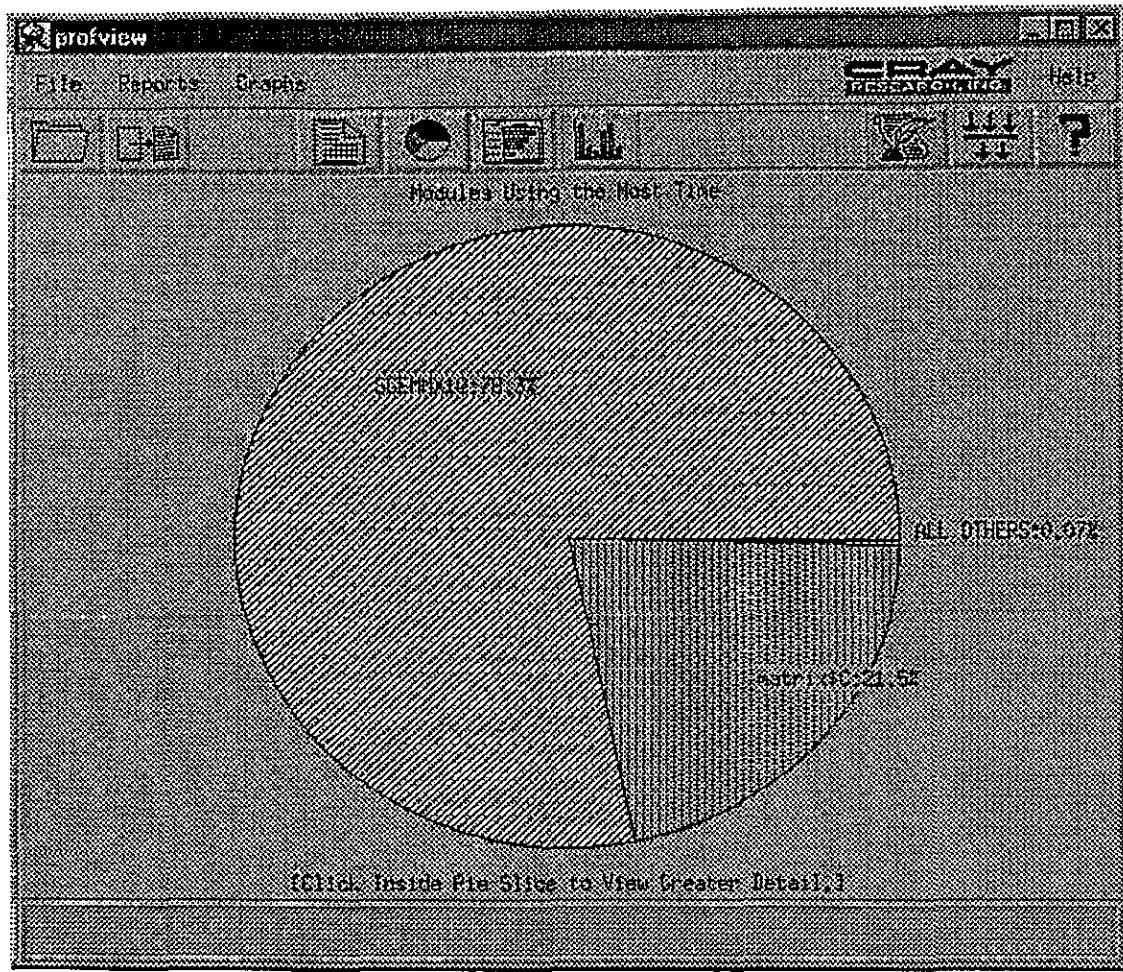
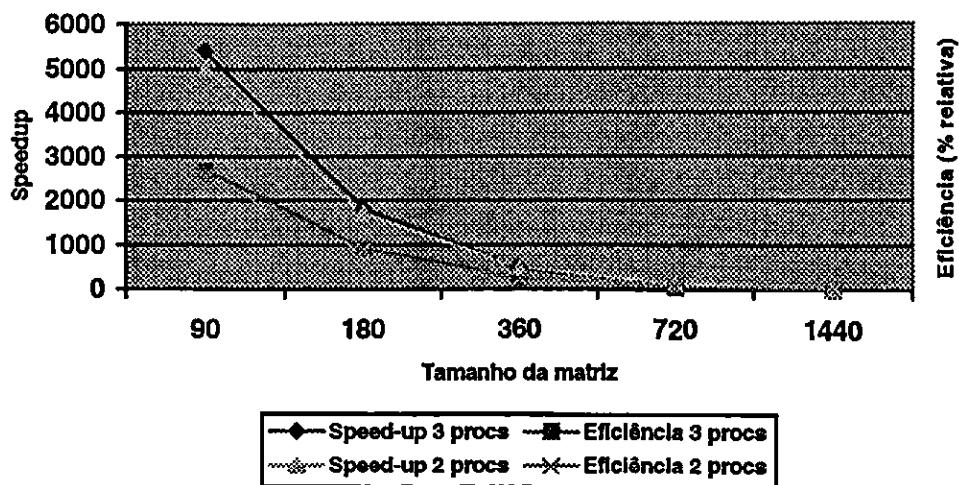


Figura 7.8 - Execução do algoritmo de multiplicação de matrizes com um nível agressivo de vetorização.

Nota-se que esses 21,5% da aplicação não vetorizada incluem trechos de código que não são contabilizados no tempo de execução do algoritmo. No algoritmo só é contabilizado o tempo necessário para solucionar o problema, neste caso o de multiplicação de matrizes. Dessa forma, o tempo de execução do algoritmo está quase que totalmente dentro dos 78,3% da substituição.

**Gráfico 7.34 - Speedup e eficiência econtrados no sistema Cray.**



O Gráfico 7.34 mostra o *speedup* e a eficiência alcançados pelo programa que é decrescente devido novamente a complexidade do problema ( $n^3$ ). Observa-se também uma anomalia rara de *speedup* e eficiência devido a vetorização e a substituição do trecho de multiplicação das matrizes pela biblioteca proprietária do *Cray*. Essa anomalia diminui a medida que a matriz aumenta de tamanho, e continua sendo uma anomalia até a matriz de tamanho 1440 x 1440. Apesar desses valores de *speedup* e eficiência muito altos, quem teve menor tempo de execução foi a versão vetorizada para 1 processador.

### Comparação entre arquiteturas

A Tabela 7.35 e o Gráfico 7.35 mostram um resumo dos desempenhos de *benchmark* encontrados.

**Tabela 7.35 – Resumo do desempenho de *benchmark* para 3 processadores.**

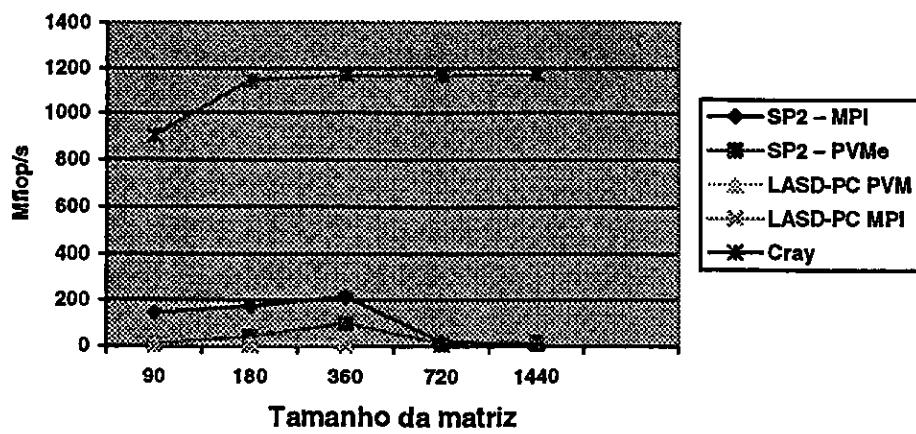
R <sub>B</sub> (N,p)	90	180	360	720	1440
SP2 – MPI	146,024	173,2341	215,0033	19,3756	13,5148
SP2 – PVMe	7,0848	42,9965	100,7964	11,1198	13,1738
LASD-PC PVM	4,2388	9,4174	13,2333	-	-
LASD-PC MPI	5,896	3,4967	3,6514	-	-
<i>Cray</i>	903,1852	1142,0617	1164,949	1164,949	1167,2161

No SP2 utilizando o MPI, os três primeiros valores crescentes são gerados pela grande velocidade de execução que é gerada pelo alto uso da memória cache, a medida que a matriz

cresce o número de operações em ponto flutuante começa a diminuir devido ao aumento no tempo de execução, lembrando que  $R_B(N,p) = F_B(N) / T(N,p)$ .

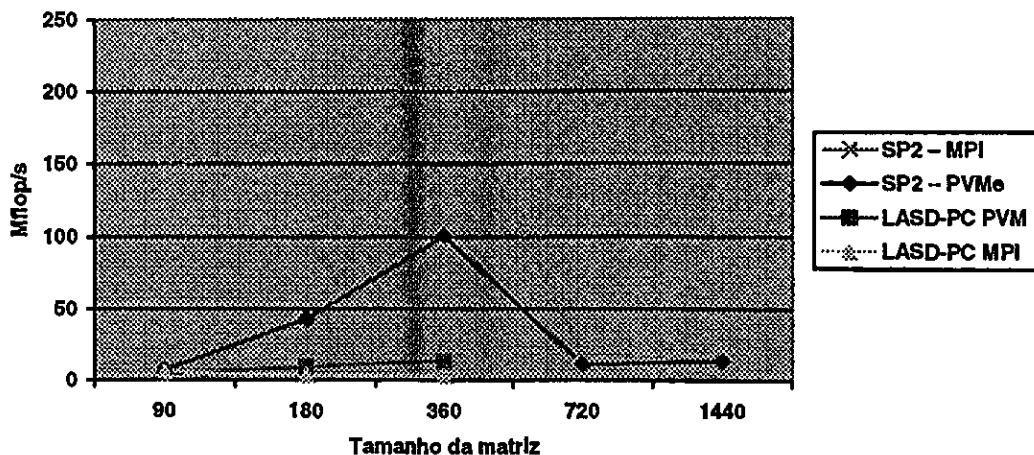
O desempenho do *benchmark* vai caindo gradativamente como pode ser visto mais claramente no Gráfico 7.28. Isso acontece devido a complexidade do problema  $n^3$ , ou seja, os custo das operações é diretamente relacionado com o tamanho da matriz.

**Gráfico 7.35 - Comparação entre arquiteturas com 3 processadores executando o algoritmo de multiplicação de matrizes.**



Devido a grande eficiência da vetorização e pela substituição por uma biblioteca própria da máquina, a arquitetura *Cray* levou uma vantagem mais que nítida sobre as demais. A substituição por essa biblioteca torna a comparação entre as arquiteturas injustas. Isso só seria possível se as demais arquiteturas também utilizassem bibliotecas proprietárias e mais eficientes. O Gráfico 7.36 mostra um melhor comparação entre as demais arquiteturas.

**Gráfico 7.36 - Comparação entre arquiteturas com 3 processadores executando o algoritmo de multiplicação de matrizes excluindo o Cray.**



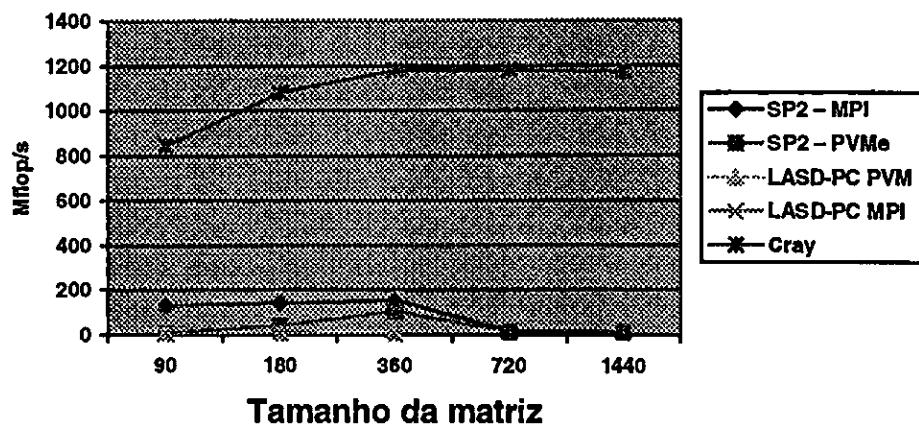
As matrizes de 90 x 90 até 360 x 360 se beneficiam nitidamente pela utilização do cache que efetua poucas trocas de páginas entre a cache e a memória. A partir da matriz de 720 x 720 essa troca de páginas aumenta drasticamente. A diferença inicial de Mflop entre o PVM e o MPI deve-se a diferença do tempo de execução. Essa diferença no tempo de execução deu-se por uma sobrecarga na execução dos escravos no PVM.

**Tabela 7.36 - Resumo do desempenho de *benchmark* para 2 processadores**

R <sub>B</sub> (N,p)	90	180	360	720	1440
SP2 - MPI	129,7128	140,056	152,7397	16,7659	13,0876
SP2 - PVM	8,62	39,3396	104,7183	13,1135	10,91055
LASD-PC PVM	6,0183	8,9552	10,5387	-	-
LASD-PC MPI	4,2388	8,8215	3,6514	-	-
Cray	840,8966	1081,9532	1178,2954	1178,2954	1170,2225

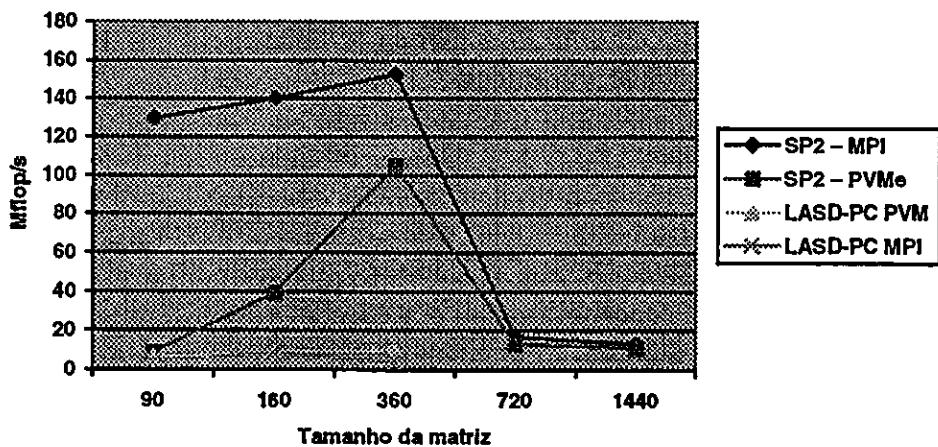
Para dois processadores o comportamento foi semelhante ao de três processadores como pode ser observado no Gráfico 7.37.

**Gráfico 7.37 - Comparação entre arquiteturas com 2 processadores executando o algoritmo de multiplicação de matrizes.**



O Gráfico 7.38 mostra o desempenho para as arquiteturas SP2 e sistema distribuído já que novamente incluir a arquitetura Cray na comparação seria extremamente injusto.

**Gráfico 7.38 - Comparação entre arquiteturas com 3 processadores executando o algoritmo de multiplicação de matrizes excluindo o Cray.**



Assim como para 3 processadores, o desempenho do *benchmark* para 2 processadores teve o mesmo comportamento. O baixo desempenho do sistema distribuído deve-se a diferença de velocidade de processamento entre as máquinas aliado a baixa velocidade de comunicação (10 Mbits/s).

## 7.3 Comparação entre as aplicações desenvolvidas e o NAS

As aplicações desenvolvidas seguem duas regras básicas do NAS:

- Não utilizar alocação dinâmica
- Os algoritmos iterativos desenvolvidos devem apresentar erro entre  $1 \times 10^{-10}$  e  $1 \times 10^{-25}$

A Tabela 8.1 mostra um resumo (de acordo com os tamanhos aproximados do problema executado) dos resultados obtidos na execução dos algoritmos e dos *benchmarks* utilizando 2 processadores sobre MPI. As três primeiras linhas representam os algoritmos numéricos desenvolvidos. As demais linhas representam os resultados encontrados na execução do *NAS Parallel Benchmark*.

**Tabela 8.1 – Resumo dos resultados obtidos na execução dos algoritmos e dos *benchmarks*.**

Benchmark	Tamanho do problema	Erro	Nº de iter.	Tempo de Execução(s)	Mflop/s
Integral	288.000.000	-	-	175,80	36,04
Jacobi	1500	$1 \times 10^{-20}$	9	1,1968	9,4113
Matrix	1440	-	-	689,2	13,0876
MG	64x64x64	$0.1624 \times 10^{-18}$	40	7,9	76,22
EP	536870912	-	-	447,86	1,20
FT	128x128x32	-	6	5,98	62,30
CG	1400	$0.8888 \times 10^{-14}$	15	1,47	62,96

A primeira coluna mostra qual algoritmo e qual *benchmark* ou aplicação foi utilizado. A próxima coluna exibe o maior valor utilizado para o tamanho do problema. A terceira coluna mostra o erro alcançado quando este é utilizado. A coluna quatro apresenta a quantidade de iterações executadas. A penúltima coluna mostra o tempo de execução e a última coluna apresenta a quantidade de operações em ponto flutuante executados por segundo.

Nota-se que os valores encontrados para os Mflop e para o tempo de execução são bem diferentes. Os tempos de execução são bem variados e extremamente dependentes do algoritmo. Não há como fazer uma previsão de qual será o tempo de execução.

Os erros encontrados também estão dentro do esperado que é de  $1 \times 10^{-15}$  a  $1 \times 10^{-20}$ , segundo o exigido na documentação do NAS. Esse parâmetro é essencial nas aplicações que exigem uma alta precisão como as aplicações de astronomia e astrofísica.

Para realizar uma avaliação utilizando-se o desempenho do *benchmark*, ou seja, a quantidade de operações em ponto flutuante por segundo é necessário saber exatamente que tipo de aplicação será executada numa determinada arquitetura. Por exemplo, se a aplicação a ser desenvolvida soluciona sistemas lineares como o CG (*Conjugate Gradiente*) do NAS ou como o algoritmo desenvolvido de Jacobi, pode-se utilizar o *benchmark* CG para prever o tempo de execução do algoritmo de Jacobi. Ao comparar as arquiteturas, o *benchmark* que tiver a maior quantidade de operações em ponto flutuante deve ser o escolhido.

A avaliação da arquitetura a ser utilizada as vezes não depende apenas do melhor desempenho. Outras variáveis estão incluídas na avaliação como custo, erro desejado numa aplicação entre outras. Cabe lembrar que quanto maior o desempenho geralmente o custo da máquina é maior o custo.

Outra observação pertinente é que o tamanho do cache influencia diretamente na quantidade de operações em ponto flutuante executadas pela máquina.

## 7.4 Considerações Finais

Neste capítulos foram apresentados os resultados obtidos com a execução de três algoritmos numéricos nas 5 plataformas consideradas neste trabalho. Algumas considerações sobre os resultados obtidos são discutidas a seguir.

Os algoritmos numéricos considerados permitem a obtenção de um bom balanceamento de carga e consequentemente uma boa utilização da CPU. Para a implementação do algoritmo do trapézio houve uma média de utilização de 97,33%. A implementação do algoritmo de Jacobi teve uma média de balanceamento de carga de 96,66%. E a implementação do algoritmo de multiplicação de matrizes teve a melhor porcentagem de balanceamento de carga que foi de 98% no sistema SP2.

Quanto aos *speedups* e eficiência alcançados uma grande discrepância pode ser observada nos diferentes exemplos e principalmente nas diferentes plataformas.

As execuções nos sistemas distribuídos apresentam na maioria dos casos, *speedups* abaixo de um. Estes resultados podem ser explicados pela utilização da rede *ethernet* – 10Mbit/s. A comunicação lenta torna a execução paralela inviável.

As execuções no SP2 apresentam na maioria dos casos bons resultados de *speedups* chegando próximo do caso ideal (*speedup* = número de processadores utilizados). Em alguns casos ocorreu a anomalia de *speedup* que gera a anomalia de eficiência devido a grande quantidade de memória. Isso permite o armazenamento completo das estruturas utilizadas na memória RAM. Quando essas estruturas eram pequenas podiam ser completamente armazenadas na memória cache. Aliado a essa característica, está também a grande velocidade de comunicação entre os processadores.

As execuções na máquina Cray apresentam ótimos resultados para as versões vetorizadas em aplicações que fazem uso intensivo de matrizes e vetores. As versões vetoriais monoprocessadas tiveram um melhor desempenho que as versões multiprocessadas devido a comunicação entre processadores que causa um maior acesso a memória RAM (gerando sincronização) e restringe a utilização de registradores e *pipelines* vetoriais a dados locais. Ao contrário da versão monoprocessada que usa intensamente os registradores e os *pipelines* vetoriais.

Assim, as aplicações que fazem uso intensivo de vetores ou matrizes tiveram um desempenho nitidamente melhor que as demais especialmente no sistema Cray. Em especial o algoritmo de multiplicação de matrizes apresentou speedups elevados, que nos sistemas Cray, uma vez que este se utilizou de bibliotecas próprias para realizar a multiplicação disponível somente no Cray.

As métricas de *speedup* e de eficiência devem ser utilizados com muita cautela, principalmente quando se está efetuando comparação entre arquiteturas. Dessa forma, para comparar as plataformas utilizadas foi usada uma outra métrica, isto é, de operações em ponto flutuante por segundo (desempenho de *benchmark*)

Comparado o desempenho de *benchmarks* das aplicações implementadas com os resultado obtidos pelo NAS (apresentado no Capítulo 6), os valores estavam dentro do esperado, não gerando nenhum valor absurdo nem fora do esperado.

A graphic element consisting of two black squares stacked vertically. The top square contains the word "Capítulo" in white serif font. The bottom square contains the number "8" in a large, bold, white sans-serif font.

Capítulo  
8

## **8. Conclusões**

Este capítulo apresenta uma discussão final sobre o trabalho desenvolvido, abordando um análise da revisão bibliográfica, comentários sobre os ambientes utilizados e as conclusões baseadas nos dados obtidos no trabalho. As principais dificuldades encontradas são discutidas e sugestões para trabalhos futuros são propostas.

### **8.1 Análise da revisão bibliográfica**

#### **Computação Paralela**

Os principais tópicos relacionados tanto às arquiteturas paralelas, quanto à programação dessas arquiteturas foram discutidos. Em termos de arquiteturas paralelas, vem sendo muito utilizado os sistemas distribuídos simulando uma arquitetura paralela virtual, também conhecida como *arquitetura de cluster*. Seu custo relativamente baixo tornou a computação paralela extremamente atrativa tanto para organizações públicas, como universidades, ou particulares como pequenas e médias empresas que antes não tinham acesso a esse tipo de sistemas.

Quanto a programação dessas arquiteturas paralelas, vem se tornando cada vez mais forte a utilização de programas MPMD e SPMD, fortemente apoiados pelas extensões paralelas de linguagens seqüenciais como o PVM e o MPI.

### **Benchmarks**

Neste tópico foi apresentada a utilização de *benchmarks* como ferramenta para avaliação de desempenho em sistemas computacionais, em geral, e em particular nos sistemas paralelos. Para organizar a grande quantidade de *benchmarks* existentes no mercado foi criado o *Parkbench Comitee*, esse comitê é responsável pela divulgação de novos *benchmarks* e pela divulgação dos resultados obtidos com a utilização dos já existentes. A área dos *benchmarks* paralelos é relativamente recente sendo que um dos mais conhecidos, e alvo de constantes estudos, é o *NAS Parallel Benchmark* desenvolvido pela *NASA Ames Research Center*. A maioria dos *benchmarks* conhecidos estão escritos em Fortran, os mais antigos encontram-se também em ALGOL, mas, pouco a pouco eles tem sido passados para a linguagem C. O principal motivo dos *benchmarks* estarem escritos em Fortran é que esta linguagem, por ser voltada para a matemática, trabalha com números de ponto flutuante mais rapidamente que as outras linguagens. Atualmente esse quadro vem sendo alterado drasticamente devido as otimizações de código feitas pelos compiladores C e C++ disponíveis atualmente [Vel97]. Na verdade, nos dias atuais essas três linguagens estão competindo de igual para igual na manipulação de números de ponto flutuante.

A área de *Benchmarks* paralelos ainda é relativamente nova, sua utilização não é simples e exige um bom conhecimento sobre:

- Configuração das bibliotecas de passagem de mensagem;
- Utilização do sistema operacional do tipo *Unix*;
- Linguagem de programação em que foi desenvolvido o *benchmark*;
- Noções de cálculo numérico quando utilizados os *benchmarks* de núcleo.

A documentação para alguns *benchmarks* paralelos como o *Genesis* e o *ParkBench* é muito pobre e geralmente muito difícil de ser encontrada.

O *NAS Parallel Benchmark*, é o *benchmark* mais bem documentado por isso não gera muitas dificuldades para ser instalado e executado.

Os resultados obtidos em *benchmarks* não dizem muita coisa sobre o sistema em teste, apenas dá um idéia do que se quer saber. Também não responde a questões que não são analisadas pelo seu nível, por exemplo, um *benchmark* de núcleo não pode dar muitas respostas sobre o sistema de comunicação pois para isso existem os *benchmarks* de baixo nível.

### Algoritmos Numéricos

Alguns exemplos de algoritmos numéricos e a possibilidade de paralelizá-los foram discutidos. Como em outras áreas, os algoritmos numéricos também têm se tornado cada vez mais complexos. Um exemplo disso é que atualmente o método de Gauss-Jacobi, que havia sido considerado obsoleto, pois convergia muito lentamente em sistemas lineares de grande porte, em relação a outros métodos diretos como o de Triangularização de Gauss, foi revivido pela computação paralela. Na área de Engenharia Elétrica, mais especificamente, na área de Sistemas de Potência, onde muitas vezes é necessário resolver sistemas lineares com matrizes de tamanho aproximado de 2000 x 2000, o método de Gauss-Jacobi e o de Jacobi são muito utilizados e muito se tem pesquisado devido a paralelização natural desses algoritmos.

A maioria dos algoritmos numéricos podem ser implementados tanto utilizando-se a abordagem de programação SPMD quanto MPMD. Sendo que a abordagem SPMD neste caso torna-se extremamente atraente por facilitar a programação e a manipulação dos dados.

## 8.2 Comentários sobre os ambientes utilizados

O aprendizado da utilização do ambiente SP2 não é trivial e exige um bom conhecimento sobre a operação do sistema operacional do tipo *Unix*. Para a biblioteca de passagem de mensagem MPI, existe um bom número de ferramentas gráficas que auxiliam no desenvolvimento das aplicações. Já para o PVM/PVMe, a quantidade de ferramentas disponíveis é muito pequena e quando encontrada é baseada em caractere.

A utilização dos sistemas *Cray* não é muito complexa. Assim como no SP2, para utilizar o sistema é necessário conhecer a operação de um sistema operacional do tipo *Unix*. Em comparação ao SP2 o sistema de submissão de tarefas do *Cray* é mais simples e a quantidade de comandos essenciais para manipular a fila de execução é bem menor.

A utilização do sistema distribuído é totalmente interativa, ou seja, não existem programas gráficos para auxiliar na execução dos programas paralelos. Dentre os três sistemas este é o que exige mais do usuário, já que o mesmo precisa estar acompanhando a execução da tarefa, seja para ver os resultados ou caso haja um travamento no sistema.

## 8.3 Discussão sobre os resultados obtidos

### Quanto aos ambientes utilizado

Um estudo sobre a paralelização do algoritmo que será utilizado com o intuito de obter-se um bom balanceamento de carga é essencial para bons resultados. Várias versões do programa devem ser criadas e testadas até a obtenção de um balanceamento adequado. O bom conhecimento sobre o ambiente que será utilizado e sobre o algoritmo a ser paralelizado ajuda na diminuição do tempo de desenvolvimento de um bom algoritmo.

O desempenho obtido em um algoritmo paralelo depende ainda da adequação da aplicação considerada ao ambiente utilizado.

Quando a aplicação não faz uso intenso de vetores, como no caso do algoritmo do trapézio composto, o desempenho nas máquinas *Cray* não é satisfatório, principalmente no que diz respeito a variável custo/benefício. Neste caso o SP2 leva nítida vantagem.

O desempenho dos programas seqüenciais nas máquinas *Cray* é extremamente baixo. Isso quase sempre leva a uma anomalia de *speedup* e de eficiência. Essas anomalias não caracterizam um real ganho de desempenho. Dessa forma, conclui-se que essa métrica não deve ser a única para tirar conclusões sobre o desempenho de um sistema, principalmente se for um sistema vetorial.

As máquinas *Cray* apresentam ainda grande vantagem para aplicações em que bibliotecas proprietárias podem substituir trechos de programas. Um exemplo deste caso é a multiplicações de matrizes.

O SP2, representa uma máquina multiprocessada de alto desempenho e com grandes vantagens para aplicações genéricas que não façam uso intensivo de vetores. Com um balanceamento de carga adequado apresenta-se como uma excelente alternativa. Um exemplo deste caso é a resolução de integrais pelo método do trapézio composto.

No sistema distribuído, a diferença de velocidade entre os processadores gera uma espera para a finalização da comunicação degradando o desempenho. Neste caso o balanceamento de carga deve ser revisto e alterado, ou seja, mais cálculos para o processador mais rápido e menos cálculos para os mais lentos. Um problema adicional observado no sistema distribuído é a comunicação muito lenta oferecida pela rede *Ethernet* a 10 Mbits/s.

### **Quanto as bibliotecas de passagem de mensagem**

Quanto ao desempenho das bibliotecas de passagem de mensagem, o PVM teve uma ligeira vantagem sobre o MPI, pelo fato de que o PVM foi projetado para rodar em sistemas distribuídos, isso aconteceu tanto no sistema distribuído como no SP2. O enfoque principal do MPI é a portabilidade, ou seja, o programa deve ser executado em qualquer máquina sem nenhuma modificação. Essa portabilidade do MPI foi notada pois os programas funcionaram tanto no SP2 como no sistema distribuído sem nenhuma alteração. Já o PVM exigiu a inserção de algumas rotinas para poder ser executado no sistema distribuído.

### **Quanto a utilização de benchmarks**

A substituição de trechos envolvendo laços por bibliotecas proprietárias para o sistema *Cray*, torna o desempenho de *benchmark* ( $R_B(N)$ ) inadequado para comparação entre essas arquiteturas, a menos que cada um delas substitua os mesmos trechos por bibliotecas proprietárias. O desempenho de *benchmark* demonstrou-se adequado para a comparação entre arquiteturas que utilizam o mesmo processo de paralelização.

Um dos grandes responsáveis pela obtenção de um bom desempenho em problemas de pequeno porte foi sem dúvida a memória cache.

A comparação entre sistemas de memória distribuída e de memória compartilhada deve ser feita com muito critério, a menos que as mesmas bibliotecas de passagem de mensagem sejam utilizadas. Como pode ser visto nos gráficos de comparação entre arquiteturas do Capítulo 7, as máquinas *Cray* quase sempre levaram vantagem devido a velocidade de comunicação entre os processadores.

O desempenho temporal ( $R_T(N)$ ) comportou-se de acordo com a complexidade do problema.

Pelo fato dos *benchmarks* paralelos serem muito recentes, o melhor teste de desempenho ainda continua sendo a aplicação que se quer utilizar, ou seja, se o objetivo é utilizar um programa de análise de contingência já construído, a execução do próprio programa em cada arquitetura observando-se o tempo de execução ainda é o melhor *benchmark*.

Caso não se tenha a aplicação ainda pronta, deve-se primeiro saber que tipos de algoritmos essa aplicação utilizará. Em seguida escolher os *benchmarks* que mais se aproximam desses algoritmos e verificar os respectivos desempenhos.

## 8.4 Dificuldades Encontradas

Inicialmente um dos maiores empecilhos para o andamento do projeto foi a configuração dos sistemas SP2 e *Cray*. No caso do SP2 do CISC a configuração e utilização foi mais fácil por encontrar-se localizado na própria região da pesquisa. O maior empecilho neste caso é a quantidade máxima de processos que podem ser gerados nesta máquina que são apenas três, ou seja, a quantidade máxima de processos que podem ser criados é igual ao número de processadores o que impede a implementação de algoritmos que gerem concorrência pelos processadores. A utilização do SP2 do LCCA tornou-se difícil devido ao grande número de usuários.

Quanto aos computadores da linha *Cray*, que se encontram no campus de São Paulo, o empecilho inicial foi a configuração dos sistemas que chegou a durar dias pois toda a comunicação com o suporte era feita através de *e-mail*.

Em relação a utilização de benchmarks, deve-se destacar a pouca documentação encontrada principalmente sobre instalação e utilização de alguns benchmarks. A utilização de *benchmarks* mostrou-se complicada exigindo muitos conhecimentos.

No desenvolvimento dos programas PVM, encontrar falhas na comunicação que provocavam o travamento da aplicação foi uma dificuldade encontrada, principalmente pela falta de ferramentas. Portar o programa em PVM do SP2 para o sistema distribuído causou dificuldades extras, pois no sistema distribuído o programa mestre utiliza a posição 0 (zero) do vetor de identificação de escravos para sua própria identificação, e no SP2 isso não ocorre.

Pelo fato de que a utilização no sistema distribuído é interativa, obrigando o usuário a acompanhar a execução das tarefas, os testes nas aplicações desenvolvidas consumiu uma boa quantidade de tempo. A falta de ferramentas para depuração causou dificuldade extras.

Notou-se também uma falta tremenda de monitores de desempenho para as arquiteturas com memória distribuída.

## **8.5 Sugestões para Trabalhos Futuros.**

- Desenvolver outros algoritmos numéricos para solucionar os mesmos problema abordados neste trabalho para validar a utilização da métrica desempenho temporal.
- Desenvolver uma ferramenta para monitorar o desempenho de máquinas com memória distribuída, contando todas as operações em ponto flutuante.
- Desenvolver uma ferramenta semelhante ao VT para um sistema distribuído com as funções de monitorar:
  - Comunicação entre processos;
  - Balanceamento de carga;
  - Utilização de CPU;
  - Acessos a disco;
  - Comunicação entre a memória cache e a memória RAM.
- Fazer a comparação de algoritmos numéricos implementados em PVM, MPI, HPF e PARMACS.
- Desenvolver o MPI para o sistema operacional Windows comparando-o com a versão do PVM para Windows desenvolvida no LASD-PC.

## *Referências Bibliográficas*

---

- [Alm94] Almasi, G. S.; Gottlieb A., “**Highly Parallel Computing**”, 2<sup>a</sup>. ed. The Benjamin Commings Publishing Company, Inc., 1994.
- [Bai94a] Bailey, D.; Barszcz, E.; Barton, J.; Browning, D.; Carter, R.; Dagum, L.; Fatoohi, R.; Fineberg, S. ; Frederickson, P. ; Lasinski, T. ; Schreiber, R.; Simon, H.; Venkatakrishnan, V.; S. Weeratunga. “**The NAS PARALLEL Benchmarks**”, <http://science.nas.nasa.gov/Pubs/TechReports/RNRreports/dbailey/RNR-94-007/html/npbspec.html>, NASA Ames Research Center, RNR Technical Report RNR-94-007, Março 1994.
- [Bai94b] Bailey, D.; Barszcz, E.; Barton, J.; Browning, D.; Carter, R.; Dagum, L.; Fatoohi, R.; Fineberg, S. ; Frederickson, P. ; Lasinski, T. ; Schreiber, R.; Simon, H.; Venkatakrishnan, V.; S. Weeratunga. “**Simulated CFD Application Benchmarks**”, [http://science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-94-001/section3\\_4.html](http://science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-94-001/section3_4.html), NASA Ames Research Center, RNR Technical Report RNR-94-007, Março 1994.
- [Beg94] Beguelin, A.; Geist, A.; Dongorra, J.; Jiang, W.; Manchek, R.; Sunderam., ‘**PVM: Parallel Virtual Machine. A User Guide and Tutorial for Networked Parallel Computing**’. The MIT Press, 1994.
- [Ber91] Berry, M.; Cybenko, G.; Larson, J., “**Scientific Benchmark Characterizations**”, Parallel Computing, vol. 17, pp. 1173-1194, 1991.
- [Ble94] Blech, R. A., “**An Overview of Parallel Processing**”, slides, Parallel Computing with PVM Workshop, Nasa Lewis research Center, 1994.
- [Cen97] Centurion, A. M., “**Análise de Desempenho de Algoritmos Paralelos Utilizando Plataformas de Portabilidade**”, Mini-dissertação, ICMSC/USP, novembro, 1997.
- [Cis98] Centro de Informática de São Carlos, “**IBM Scalable POWERParallel System**”, <http://www.cisc.sc.usp.br/servicos/sp2.html>, 1998
- [Cla94] Claudio, D. C. e Marins, J.M. “**Cálculo Numérico Computacional: Teoria e Prática**”, São Paulo: Atlas, 1994. 2 ed. 460p.

- [Cra98a] Cray Research Inc., “**Cray C/C++ Reference Manual**”, [http://www.Cray.com/cgi-bin/nph-dweb/dynaweb/t90\\_ieee\\_fp/004-2179-001/@Generic\\_\\_BookTextView/7207](http://www.Cray.com/cgi-bin/nph-dweb/dynaweb/t90_ieee_fp/004-2179-001/@Generic__BookTextView/7207).
- [Cra98b] Cray Research Inc., “**Optimizing Code on Cray PVP Systems**”, [http://www.Cray.com/cgi-bin/nph-dweb/dynaweb/t90\\_ieee\\_fp/004-2179-001/@Generic\\_\\_BookTextView/7207](http://www.Cray.com/cgi-bin/nph-dweb/dynaweb/t90_ieee_fp/004-2179-001/@Generic__BookTextView/7207).
- [Del95] Dell Computer Corporation, “**Server System Benchmarks**”, Advanced Tecnology Group, 1995.
- [Don87] J. Dongarra, J. Martin, and J. Worlton, “**Computer Benchmarking: Path and Pitfalls**”, IEEE Spectrum, pp. 38-45, Julho 1987.
- [Don96] Dongarra J., Mucci, P., Strohmaier, E., “**ParkBench 2.0: Release Notes and Run Rules**”, <http://netlib2.cs.utk.edu/parkbench/>, May, 1996.
- [Dun90] Duncan, R. “**A Survey of Parallel Computer Arquitectures**”, Computer IEEE, Fevereiro de 1990.
- [Emi96] Emilio G. H., “**A Methodology for Design of Parallel Benchmarks**”, Thesis submitted for the degree of Doctor Philosophy, University of Southampton, Dezembro 1996.
- [Fos95] Foster, Y., “**Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering**”, Addison-Wesley Publishing Company Inc., 1995.
- [Geh89] Gehani, N., e McGetrick A. D. “**Concurrent Programming**”, Cornwall: International Computer Science Series, 1989. 621p.
- [Gei94] Geist, A., Beguelin, A., Dongarra, J., Weicheng, J., Manchek, R., Sunderam, V., “**PVM 3 User’s Guide and Reference Manual**”, Oak Ridge National Laboratory, 1994.
- [Gle94] Glendinning, I., “**The GENESIS Distributed-Memory Benchmark Suite Release 3.0**”. High Performance Computing Centre. University of Southampton, 1994.
- [Gus86] Gustafson, J. L., Hawkinson, S., “**A Language\_Independent Set of Benchmarks for Parallel Processors**”, <http://www.scl.ameslab.gov/Publications/6-pack/6-pack.html>. Ames Laboratiry, Departament of Energy, ISU, Ames, Iowa, 1996.
- [Hay82] Haynes, L. S.; Lau, R. L., Siewiorek, D. P., “**A Survey of Highly Parallel Computing**”, Computer IEEE, Janeiro 1982.
- [Hei84] Heidelberg, P. e Lavemberg, S. “**Computer Performance Evaluation Methodology**”, IEEE Transaction Computer, vol C-33, nº 12, pp. 1195-1220, 1984.
- [Hoc96] Hockney, R. W., “**The Science of Computer Benchmarking**”, Philadelphia:

- society for Industrial and Applied Mathematics, 1996.
- [Hwa84] Hwang, K., Brigs, F. A., “Computer Architecture and Parallel Processing”, McGrawHill, 1984.
- [Hwa93] HWANG, Kai. “Advanced Computer Architecture: Parallelism, Scalability, Programmability”. Illinois: McGraw-Hill, 1993.
- [Ibm95a] IBM Corporation, “IBM Parallel Environment for AIX: Operation and Use Version 2.1.0”, IBM Departament 55JÁ, 1995
- [Ibm95b] IBM Corporation, “IBM PVMe for AIX User’s Guide and Soubrotine Reference Version 2, Release 1”, IBM Departament 55JÁ, 1995
- [Ibm95c] IBM Corporation, “LoadLever User’s Guide Release 2.1”, , <ftp://ftp.usp.br/pub/lcca>, 1995
- [Ibm98] IBM Corporation, “SP2 High Performance Switch Overview”, <http://aixport.sut.ac.jp/advsys/mhppc/ibmhsw/switch.html>, 1998.
- [Int98] INTEL Corporation, “Why Should You Care About Performance?”, <http://pentium.intel.com/procs/perf/intro/bintro.htm>, 1998.
- [Kir91] Kirner, C., “Arquiteturas de Sistemas Avançados de Computação”, Anais da Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho, pp. 307-353, 1991.
- [Mac94] MacDonald, N., Minty, E., Harding, T., Browa, S., “Writing Message-Passing Parallel Programs with MPI – Course Notes”. The University of Edinburgh, 1994.
- [Mah88] McMahon., F. H., “The livermore Fortran Kernels test of the Numerical Performance Range.”. Performance Evaluation of Supercomputers, pp. 143-186. Elvier Science B. V., North-Holland, Amsterdam, 1988.
- [Men97] Meng, X., “Parallel Computing CSCI 6356”, <http://bahia.cs.panam.edu/~meng/Course/CS6356/Notes/master/node20.html#SECTION005200000000000000000000>, Texas: Department of Computer Science University of Texas - Pan American, 1997.
- [Mis89] Misra, C. “Parallel Program Design: A Foundation”, Texas: Addison-Wesley Publishing Company, 1989. 516p.
- [Nav89] Navaux, P. O. A., “Introdução ao Processamento Paralelo”. RBC – Revista Brasileira de Computação, v. 5, nº 2, pp.31-43. Outubro, 1989.
- [Onl98] OnLine Knowldgment Home Page, “Types of *benchmarks*”, <http://www.systemhouse.mci.com/shltdemo/segments/lds00001/method/tech/mq306/mq307nd4.htm>, 1998.

- [Tan95] Tanenbaum, A. S., “**Distributed Operating Systems**”, Prentice Hall International Inc., 1995.
- [Vel94] Velde, E. F. Van De, “**Concurrent Scientific Computing**”, New York: Springer-Verlag, 1994. 328p.
- [Vel97] Veldhuizen, T., “**Scientific Computing: C++ Versus Fortran**”, Dr. Dobb’s Journal, nº. 271, pp. 34-38, Novembro, 1997.
- [Wai95] Wainner, S., “**Benchmarks FAQ version 6**”, <http://hpwww.epfl.ch/bench/bench.FAQ.html>, 1995.
- [Wal94] Walker, D. W., “**The Design of Standard Message Passing Interface for Distributed Memory Concurrent Computers**”, Parallel Computing, vol. 20, pp. 657-673, 1994.
- [Wei91] Weicker, R. P., “**Detailed Look at Some Popular Benchmarks**”, Parallel Computing, vol. 17, pp. 1153-1172, 1991.
- [Zan94] Zandt, J. V., “**Scientific Computing Paradigm**”, NASA Ames Research Center, RNR Technical Report RND-94-002, Janeiro, 1994 .