

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

STÉFANO DRIMON KURZ MOR

**Emprego da Técnica de *Workstealing*:
Estudo de Caso com o Problema da Mochila
e MPI**

Projeto de Diplomação

Prof. Dr. Nicolas Maillard
Orientador

Porto Alegre, Junho de 2007

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. José Carlos Ferraz Hennemann

Vice-reitor: Prof. Pedro Cezar Dutra Fonseca

Pró-Reitor de Graduação: Prof. Carlos Alexandre Netto

Diretor do Instituto de Informática: Prof. Flávio Rech Wagner

Coordenador do CIC: Prof. Raul Fernando Weber

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“Ouve e esquecerás, vê e recordarás, faz e saberás.”
— CONFÚCIO

AGRADECIMENTOS

- Agradeço a Deus, por tudo nessa vida.
- Agradeço aos meus pais e meu irmão, que me fizeram chegar até aqui.
- Agradeço ao meu orientador, Nicolas Maillard, por sempre estar disposto a me atender e por sempre respeitar minha opinião, sendo *impecável* durante todo este processo.
- Agradeço à minha "mentora" Márcia Cera, por tudo que me ensinou, pelo apoio e pelas correções.
- Agradeço à Jamile Moroszczuk, pelas discussões tão produtivas e o suporte inabalável.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS	8
LISTA DE FIGURAS	9
LISTA DE TABELAS	11
LISTA DE ALGORITMOS	12
RESUMO	13
ABSTRACT	14
1 INTRODUÇÃO	15
1.1 Atualidades	15
1.2 Descrição do Problema	15
1.3 Motivação	16
1.4 Trabalhos Anteriores	17
1.5 Ferramentas Empregadas	18
1.6 Metodologia e Abordagem de Estudo	18
1.7 Áreas de Interesse	19
2 CONTEXTO CIENTÍFICO	20
2.1 Programação Paralela	20
2.1.1 <i>Hardware</i>	20
2.1.2 Modelos Computacionais e Comunicação	24
2.1.3 Desempenho	27
2.2 Introdução ao MPI	28
2.2.1 Motivação	29
2.2.2 História	29
2.2.3 Definição e Características	30
3 MPI	32
3.1 As Seis Primitivas Básicas	32
3.1.1 Inicialização e Finalização	32
3.1.2 Identificação de Processos e Comunicadores	33
3.1.3 Primeiro Exemplo	33
3.1.4 Troca de Mensagens	36
3.1.5 Segundo Exemplo	38
3.1.6 Programação Distribuída × Programação Paralela	38

3.2	Funcionalidades Adicionais	40
3.2.1	Comunicação Coletiva	40
3.2.2	Terceiro Exemplo	43
3.2.3	Temporização	45
3.2.4	Envio e Recebimento Não-bloqueantes	45
3.2.5	Quarto Exemplo	47
3.3	Multiplicação Matriz $\times$ Vetor	48
3.3.1	Multiplicação Matriz $\times$ Matriz	48
4	PROBLEMA DA MOCHILA	56
4.1	Definição Intuitiva	56
4.2	Definição Formal	56
4.3	Outras Definições	57
4.3.1	Problema da Mochila <i>Limitado</i>	57
4.3.2	Problema da Mochila <i>0-1</i>	57
4.4	Complexidade	58
4.5	Solução	58
4.5.1	Algoritmo	58
4.5.2	MPI	63
5	WORKSTEALING	69
5.1	Difusão de Máximo Local	69
5.1.1	Algoritmo	69
5.1.2	Implementação	70
5.2	Workstealing	74
5.2.1	Algoritmo	74
5.2.2	Considerações	76
5.2.3	Implementação	77
6	AVALIAÇÃO DE DESEMPENHO	84
6.1	Implementação	84
6.2	Elaboração dos Casos de Teste	85
6.3	Velocidade de Execução	87
6.3.1	Análise do Tempo de Execução	87
6.3.2	<i>Speedup</i>	88
6.3.3	Eficiência	89
6.4	Consumo de Memória	90
6.5	Balanceamento de Carga	92
6.6	Impacto de Utilização do Workstealing	96
6.7	Deadlock	96
6.7.1	Difusão de Máximo Local	97
6.7.2	Workstealing	98
7	CONCLUSÕES	100
7.1	Problema da Mochila $\times$ Paralelização	100
7.2	Implementação	100
7.3	Desempenho	101
7.4	WS Clássico $\times$ Implementação	102
7.5	Escalabilidade	102
7.6	Trabalhos Futuros	102

APÊNDICE A	ANÁLISE DOS CASOS DE TESTE	104
A.1	Tempo de Execução	104
A.2	Consumo de Memória	107
A.3	Balanceamento de Carga	107
REFERÊNCIAS		112

LISTA DE ABREVIATURAS E SIGLAS

AMD	<i>Advanced Micro Devices.</i>
DML	Difusão de Máximo Local.
E/S	Entrada/Saída.
GiB	<i>Gigabyte</i> (binário).
GPPD	Grupo de Processamento Paralelo e Distribuído.
IP	<i>Internet Protocol.</i>
LAM	<i>Local Area Multicomputer.</i>
MIMD	<i>Multiple-Instruction, Multiple-Data.</i>
MISD	<i>Multiple-Instruction, Single-Data.</i>
MPI	<i>Message-Passing Interface.</i>
PVM	<i>Parallel Virtual Machine</i>
RAM	<i>Random Access Memory.</i>
SIMD	<i>Single-Instruction, Multiple-Data.</i>
SISD	<i>Single-Instruction, Single-Data.</i>
TCP	<i>Transmission Control Protocol.</i>
UCP	Unidade Central de Processamento.
UFRGS	Universidade Federal do Rio Grande do Sul.
WS	<i>Workstealing.</i>

LISTA DE FIGURAS

Figura 2.1:	Modelo clássico de multiprocessadores.	23
Figura 2.2:	Modelo clássico de multicomputadores.	23
Figura 3.1:	Protótipos de <code>MPI_Init</code> e <code>MPI_Finalize</code>	32
Figura 3.2:	Protótipos de <code>MPI_Comm_size()</code> e <code>MPI_Comm_rank()</code>	33
Figura 3.3:	Arquivos de inclusão MPI.	34
Figura 3.4:	Variáveis do programa <i>Hello World MPI</i>	34
Figura 3.5:	Programa “ <i>Hello World</i> ” sem troca de mensagens.	35
Figura 3.6:	Possibilidade de saída - primeiro exemplo.	36
Figura 3.7:	Outra possibilidade de saída - primeiro exemplo.	36
Figura 3.8:	Protótipos de <code>MPI_Send()</code> e <code>MPI_Recv()</code>	37
Figura 3.9:	Programa “ <i>Hello World</i> ” com troca de mensagens.	39
Figura 3.10:	Saída - segundo exemplo.	39
Figura 3.11:	Diagrama do algoritmo de passagem de <i>token</i>	40
Figura 3.12:	Impressão sequencial de <i>rank</i> com algoritmo de <i>token</i> em anel.	41
Figura 3.13:	Resultado para a impressão sequencial de identificadores.	41
Figura 3.14:	Implementação direta de <i>broadcast</i>	42
Figura 3.15:	Protótipo de <code>MPI_Broadcast()</code>	42
Figura 3.16:	Protótipo de <code>MPI_Reduce()</code>	43
Figura 3.17:	Programa para calcular uma aproximação de π	44
Figura 3.18:	Protótipos de <code>MPI_Wtime()</code> e <code>MPI_Wtick()</code>	45
Figura 3.19:	Programa para calcular uma aproximação de π , com medição de tempo.	46
Figura 3.20:	Protótipos de <code>MPI_Isend()</code> e <code>MPI_Irecv()</code>	47
Figura 3.21:	Protótipos de <code>MPI_Wait()</code> e <code>MPI_Test()</code>	47
Figura 3.22:	Portótipos de <code>MPI_Testany()</code> e <code>MPI_Waitany()</code>	47
Figura 3.23:	Protótipo de <code>MPI_Cancel()</code>	48
Figura 3.24:	Recebimento de mensagem que pode ter exatas 2 <i>tags</i>	48
Figura 3.25:	Código para o programa de multiplicação matriz $\times$ vetor (início).	49
Figura 3.26:	Código para o programa de multiplicação matriz $\times$ vetor (mestre).	50
Figura 3.27:	Código para o programa de multiplicação matriz $\times$ vetor (escravo).	51
Figura 3.28:	Desempenho: multiplicação matriz $\times$ vetor – MPI-1.2	51
Figura 3.29:	Código para o programa de multiplicação matriz $\times$ matriz (início).	53
Figura 3.30:	Código para o programa de multiplicação matriz $\times$ matriz (mestre).	54
Figura 3.31:	Código para o programa de multiplicação matriz $\times$ matriz (escravo).	55
Figura 4.1:	Formato de instância do problema da mochila.	58
Figura 4.2:	Algoritmo <i>Branch & Bound</i> para o Problema da Mochila, em <i>Python</i>	64

Figura 4.3:	Variáveis da implementação sequencial que resolve o Problema da Mochila (em C).	64
Figura 4.4:	Algoritmo <i>Branch & Bound</i> para o Problema da Mochila, em C . . .	65
Figura 4.5:	Algoritmo C com primitivas básicas para o Problema da Mochila . .	66
Figura 4.6:	Função que realiza a distribuição cíclica da entrada.	67
Figura 4.7:	Algoritmo C/MPI paralelizado.	68
Figura 5.1:	Implementação de <i>broadcast assíncrono</i>	70
Figura 5.2:	Implementação do gerenciador de difusão.	71
Figura 5.3:	Implementação de <code>mpi_process_nonblocking()</code>	72
Figura 5.4:	Implementação do mecanismo de barreira.	74
Figura 5.5:	Algoritmo C/MPI paralelizado e com <i>Difusão de Máximo Local</i> . . .	75
Figura 5.6:	Esquema de funcionamento – Fila de Distribuição Dupla	77
Figura 5.7:	Implementação da fila de distribuição dupla.	78
Figura 5.8:	Algoritmo C/MPI, versão com fila de distribuição dupla	79
Figura 5.9:	Implementação do gerenciador de <i>Workstealing</i>	80
Figura 5.10:	Implementação da tabela de processos.	81
Figura 5.11:	Solicitação de tarefas - <i>Workstealing</i>	82
Figura 5.12:	Implementação MPI com <i>Workstealing</i>	83
Figura 6.1:	Formato do arquivo de saída.	84
Figura 6.2:	Interface da estrutura de <i>Benchmark</i>	85
Figura 6.3:	Gráfico tempo de execução (número de elementos vs. tempo em segundos).	88
Figura 6.4:	Gráfico tempo de execução (número de processos vs. tempo em segundos).	89
Figura 6.5:	Gráfico Speedup.	90
Figura 6.6:	Gráfico Eficiência.	91
Figura 6.7:	Gráfico Consumo de Memória (número de elementos vs. memória em <i>bytes</i>).	92
Figura 6.8:	Gráfico Consumo de Memória (número de processos vs. memória em <i>bytes</i>).	93
Figura 6.9:	Gráfico mensagens enviadas/recebidas (num. de elementos vs. num. de mensagens).	94
Figura 6.10:	Gráfico mensagens enviadas/recebidas (num. de processos vs. num. de mensagens).	95
Figura 6.11:	Impacto do WS: tempo de execução.	97
Figura 6.12:	Impacto do WS: consumo de memória	98

LISTA DE TABELAS

Tabela 2.1:	Notação do cálculo do <i>Speedup</i>	27
Tabela 3.1:	As seis funções básicas do MPI.	32
Tabela 3.2:	Tempos da multiplicação matriz $\times$ matriz paralela.	52
Tabela 4.1:	Significado das variáveis na função de distribuição cíclica da entrada.	67
Tabela 6.1:	Significado dos arquivos de saída do programa.	85

LISTA DE ALGORITMOS

1	<i>Branch</i> (um galho)	59
2	<i>Branch</i> (toda a árvore).	60
3	<i>Bound</i>	61
4	Solução <i>Branch & Bound</i> para o Problema da Mochila	62

RESUMO

O cerne da discussão aqui introduzida é o estudo do emprego da técnica de roubo de tarefas (*Workstealing*) em programas paralelos, tomando-se como base a paralelização de um algoritmo do tipo *Branch & Bound* que resolve o *Problema da Mochila*. Como principal ferramenta, utiliza-se uma implementação da especificação MPI (*Message-Passing Interface*).

A monografia apresenta uma descrição detalhada da especificação MPI, com ênfase nos recursos empregados na confecção do algoritmo. São introduzidas definições de *Workstealing* e do *Problema da Mochila*, sendo apresentadas e comentadas as principais partes do código do programa MPI que combina ambas. Além disso, são apresentadas medições de desempenho, cujo objetivo é avaliar o impacto da solução apresentada sobre diferentes aspectos (*e.g.*, tempo de execução, consumo de memória, balanceamento de carga, complexidade algorítmica, *etc.*) da execução de programas.

Ao final, são apresentadas conclusões gerais sobre o assunto e indicados trabalhos futuros a serem realizados.

Palavras-chave: MPI, Problema da Mochila, Workstealing.

Use of The Workstealing Technique in Knapsack Problem's Parallelization Using MPI

ABSTRACT

The introduced discussion's kernel is the study of the use of the task stealing (Workstealing) technique on programs' parallelization process, taking as base the parallelization of a Branch & Bound algorithm that solves the Knapsack Problem. As main tool, it utilizes a implementation of the MPI (Message-Passing Interface) specification.

The monography presents a detailed description of the MPI specification, giving emphasis at the resources used on algorithm's confection. Definitions of the Workstealing concept and Knapsack Problem are introduced, being presented and commented the MPI program's main parts of code, that combines both. Moreover, performance measurements are presented, whose objective is to evaluate the presented solution's impact on different programs' execution aspects (*e.g.*, execution time, memory consumption, load balancing, algorithmic complexity, etc.).

Finally, general conclusions are presented and future to-be-done work is pointed.

Keywords: MPI, Knapsack Problem, Workstealing.

1 INTRODUÇÃO

Programação paralela e arquiteturas paralelas de computadores representam um passo natural na próxima geração da escala evolutiva da *Computação*. De fato,

[...] computadores paralelos evoluíram de bugigangas experimentais em laboratórios para tornarem-se as ferramentas cotidianas dos Cientistas da Computação, que precisam do que há de melhor em recursos computacionais, no intuito de resolver seus problemas. (GROPP; LUSK; SKJELLUM, 1999, p.1)

1.1 Atualidades

Mesmo os processadores destinados ao uso pessoal começam a ganhar feições multiprocessadas; num estágio de transição de *dual-core* (dois núcleos) para *quad-core* (quatro núcleos), fabricantes como *Intel* e *AMD* trazem para o cotidiano conceitos que antes faziam parte de empresas e universidades, apenas (CREEGER, 2005).

Recentemente, a *Intel* fez demonstração de um *chip* com oitenta núcleos e memória RAM embutida, o *TeraFLOP*, capaz de processar um *teraflop*¹ por segundo. Embora não seja retrocompatível com qualquer arquitetura de microprocessador da empresa, protótipos como esse evidenciam os rumos que o mercado de microprocessadores (tanto empresarial quanto doméstico) tenderá a adotar (MATTSON; HENRY, 1998).

1.2 Descrição do Problema

MPI é, desde 1996, o padrão *de fato* para a implementação do modelo de comunicação por troca de mensagens em Computação de Alto Desempenho. MPI oferece um modelo de replicação de processos (execução do mesmo código) em todas as UCPs que participam da computação, bem como primitivas de troca de mensagens entre estes processos (GROPP; LUSK; SKJELLUM, 1999).

Um problema recorrente em aplicações paralelas é o balanceamento de carga de trabalho entre os processadores; para um melhor aproveitamento do paralelismo, é conveniente que todos os processos tenham aproximadamente a mesma quantidade de tarefas a realizar. Qualquer desbalanceamento implica aumento do tempo de ociosidade, o que acaba por afetar o desempenho. (PACHECO, 1997).

Uma técnica que visa obter o balanceamento de carga é o *Workstealing*. Esta abordagem consiste em fazer com que um processo que tenha esgotado sua carga computacional

¹Um *trilhão* de operações em ponto flutuante.

“roube” tarefas de outros processos que ainda têm trabalho pendente, equilibrando a distribuição de tarefas (BLUMOFÉ; LEISERSON, 1994).

A utilização do *Workstealing*, no entanto, introduz questões relevantes para a computação. *e.g.*,

Impacto no desempenho. WS tenta diminuir o tempo de execução ao trocar tarefas entre os processos. Mas esse ganho pode não compensar o tempo perdido ao fazer as operações de sincronização (trocas de mensagens) e o tempo agregado ao gerenciamento destas operações.

Escalabilidade. Quanto mais escalável o problema, mais difícil a sua implementação através de *Workstealing*, visto que existem questões clássicas como ocorrência de *deadlocks*, gerenciamento eficiente de memória (o consumo tende a ser elevado) e adequação ao *hardware* utilizado.

Implementação Certos aspectos do algoritmo podem ser implementados de várias maneiras. Tais maneiras implicam, muitas vezes, em custos e dependências de certas estruturas pré-definidas, que tornam o processo menos transparente. Em especial, há a questão da portabilidade da solução para aplicações genéricas, que necessita ser investigada, em decorrência da forte dependência em relação ao problema resolvido com o qual se pode construir o algoritmo.

Algumas destas questões ainda tem caráter em aberto; é difícil mensurar um caso médio e eficiente, sendo necessário um estudo mais aprofundado.

1.3 Motivação

Estimar os possíveis benefícios da do emprego de *Workstealing* tem impacto forte sobre questões relevantes da área. Traçar e resolver questões inerentes a este problema introduz melhoramentos em áreas como:

Escalonadores de Processos. Implementações de escalonadores de processos MPI (*e.g.*, LAM) não possuem um sistema de balanceamento de cargas eficiente² e que conte com recursos de roubo de processos³ (CERA et al., 2006), o que poderia ofertar benefícios à computação realizada.

Bibliotecas Paralelas. Uma biblioteca genérica, eficiente e transparente ao máximo, que realize *Workstealing* sobre tipos genéricos de dados pode garantir um ganho de desempenho em aplicações paralelas de uma maneira ampla.

Otimização de Recursos. O balanceamento de cargas diminui ociosidade, otimizando o uso de recursos.

Conforme já mencionado, o algoritmo utilizado para a aplicação da técnica e realização das medições foi um algoritmo do tipo *Branch & Bound* que resolve o Problema da Mochila. Tal problema foi escolhido por uma série de motivos, a citar:

²Carga, neste caso, é a alocação do número de processos por processador.

³Outro tópico fundamental neste processo é a *migração de processos*, uma maneira eficiente e efetiva de transferir os processos, uma vez que um algoritmo de *Workstealing* determine que é vantajoso fazê-lo.

- popularidade;
- escalabilidade; e
- relevância.

Nem todos os problemas possuem uma estrutura adequada ao uso de *Workstealing* (BLUMOFÉ; LEISERSON, 1994). O Problema da Mochila, por sua estrutura combinatorial (KELLER; PFERSCHY; PISINGER, 2005), parece, *a priori*, apto a se beneficiar do uso da técnica. Parte do estudo, portanto, consiste em determinar o quão o Problema da Mochila é adequado para a aplicação de WS. Além disso, o Problema da Mochila é especialmente interessante quando escrito em formas de maximização de lucros, como:

- decidir onde perfurar poços de petróleo;
- problemas de otimização de transporte de cargas;
- problemas de otimização de rotas aéreas.

Vale ressaltar que estes problemas são inerentemente intratáveis do ponto de vista computacional, visto que o Problema da Mochila pertence à classe dos problemas *NP-Completo*⁴ (TOSCANI; VELOSO, 2002).

Esta monografia, portanto, preocupa-se em

1. Analisar e apresentar os principais recursos para se construir um programa paralelo usando a especificação MPI;
2. Propor uma estratégia de paralelização do Problema da Mochila e fazê-lo através de programação MPI;
3. Introduzir e dissertar sobre a técnica de *Workstealing*, apresentado vantagens e desvantagens, integrando esta característica ao programa construído; e
4. Fazer medições de diversas características na execução desta solução (*e.g.*, tempo de execução, memória, número de trocas de mensagens, *etc.*).

1.4 Trabalhos Anteriores

Parte dos assuntos abordados dão continuidade aos trabalhos do GPPD que tratam do escalonamento dinâmico de processos. Em especial, os trabalhos de Guilherme Pezzi sobre a implementação de *Workstealing* em algoritmos de divisão e conquista usando MPI-2 (PEZZI et al., 2006) e os trabalhos de Márcia Cera sobre melhoras no escalonador MPI para criação dinâmica de processos (CERA et al., 2006) são tomadas como referência básica. Estende-se o primeiro, tentando estabelecer uma especificação de algoritmo de *Workstealing* genérico (em contraposição ao caráter específico do programa apresentado) e se objetiva, com os dados medidos, fornecer resultados que ampliem o leque de opções na melhora do escalonador MPI que é proposta pelo segundo.

Resultados parciais deste trabalho foram publicados na Escola Regional de Alto Desempenho 2007 (ESCOLA REGIONAL DE ALTO DESEMPENHO, 2007) (no Fórum de Iniciação Científica), em conjunto com uma abordagem introdutória sobre impacto da criação dinâmica de processos com MPI-2, ausente nesta monografia.

⁴Indica que a melhor solução só pode ser encontrada por meio da análise de todo o espaço combinatorial, ou seja, possui complexidade exponencial quando executado em máquinas determinísticas.

1.5 Ferramentas Empregadas

Os exemplos e demonstrações cujos códigos são apresentados foram executados no *cluster labtec* do GPPD da UFRGS. Ele executa sobre a distribuição LAM MPI 7.2⁵, uma implementação de ambos os padrões MPI-1.2 e MPI-2.

Os códigos de programas expostos estão, predominantemente, em linguagem *C*. Em alguns casos, quando o algoritmo seqüencial ganha foco, opta-se por mostrar sua implementação na linguagem *Python*, que fornece uma maneira mais próxima da descrição matemática do problema. Os *scripts* apresentados são escritos em *Bash Script*⁶. Estes programas, por sua natureza e pelas características dos *clusters*, são executados em ambiente *Linux*.

1.6 Metodologia e Abordagem de Estudo

A cada capítulo apresentado procurou-se evidenciar o máximo de informações possíveis. No entanto, dependendo do interesse do leitor, pode-se dar enfoque a algum aspecto específico da monografia. Por exemplo, se houver maior interesse na parte de implementação, os códigos do programa implementado estão disponíveis e comentados ao longo da monografia. Se por outro lado, o interesse for na abordagem formal do problema e nos resultados alcançados, não há necessidade marcante de se observar todos os trechos de programa apresentados.

Procura-se tornar os capítulos o mais independentes possível, para se ajustar aos conhecimentos prévios que o leitor possui. Se já houver conhecimento da especificação MPI ou do Problema da Mochila, por exemplo, pode-se dar um maior foco aos outros capítulos. Não se aconselha, no entanto, a omissão da leitura de algum capítulo; considerações importantes sobre a implementação da solução do problema são feitas ao longo de todos eles.

Os capítulos abordados, e seus respectivos temas, são:

Contexto Científico. Agrupa conceitos básicos e considerações relevantes encontradas ao longo de todo o processo de montagem e seleção da bibliografia. Divide-se em dois grandes tópicos (seções): *Programação Paralela* e *Introdução ao MPI*.

MPI. Aborda os comandos e primitivas básicas de MPI. Demonstra, através de exemplos, as funções básicas e a construção de programas paralelos elementares no modelo de troca de mensagens.

Problema da Mochila. Descreve detalhadamente o problema da Mochila, enumerando sua definição e características. Apresenta uma solução seqüencial para este e também uma solução paralela, empregando MPI.

Workstealing. Introduz o conceito de *Workstealing* e discute seu emprego. Aborda, também, as modificações necessárias à resolução paralela do Problema da Mochila para que se obtenha um algoritmo que aplique *Workstealing*.

Avaliação de Desempenho. Expõe os principais resultados obtidos na confecção da solução do problema. Vários aspectos são mensurados e, ao final, apresenta conclusões sobre estes resultados e o emprego da técnica proposta.

⁵Mais sobre a distribuição pode ser encontrado em <http://www.lam-mpi.org/>.

⁶Linguagem de *script* para o *bash* (*bourne-again shell*).

Conclusões. Enumera as principais conclusões inferidas ao longo do processo de confecção do trabalho. Relaciona conceitos e estabelece hipóteses sobre os resultados apresentados. Ao término, traça um perfil dos trabalhos futuros a serem realizados.

1.7 Áreas de Interesse

Ao longo do desenvolvimento da monografia, são permeadas várias áreas da Ciência da Computação e Matemática em geral; em especial, cada capítulo tem um foco distinto, sendo interessante enumerar os conceitos básicos abordados ao longo de cada um:

Contexto Científico: Arquiteturas Paralelas, Programação Paralela, Programação Distribuída, Análise de Desempenho.

MPI: Programação Concorrente, Algoritmos e Programação, Estruturas de Dados, Classificação e Pesquisa de Dados, Técnicas de Construção de Programas, Algoritmos Paralelos, Redes de Computadores.

Problema da Mochila: Programação Paralela e Distribuída, Complexidade de Algoritmos, Teoria da Computação, Pesquisa Operacional.

Workstealing: Algoritmos Paralelos, Sistemas Operacionais, Arquiteturas Paralelas, Técnicas de Otimização.

Avaliação de Desempenho: Probabilidade e Estatística, Complexidade de Algoritmos Paralelos, Análise de Desempenho.

Conclusões: Teoria da Computação, Complexidade de Algoritmos, Arquiteturas Paralelas, Sistemas Operacionais, Programação Distribuída.

2 CONTEXTO CIENTÍFICO

Este capítulo descreve o atual estado da arte de tópicos fundamentais no desenvolvimento da monografia, sobretudo no que tange *Programação Paralela* e MPI. Dessa maneira, o que se apresenta é a síntese e apontamento das direções da área, verificados na bibliografia consultada; serve, ao mesmo tempo, de referência e de base para o restante do conteúdo desenvolvido ao longo do estudo.

A abordagem adotada é mais conceitual. O leitor que deseje uma abordagem mais prática (sobretudo a presença de códigos) e focada em aplicações deve consultar os capítulos posteriores.

2.1 Programação Paralela

Embora o enfoque do estudo seja a programação e codificação de algoritmos paralelos, existe uma plataforma de *hardware* que desempenha papel essencial para o processamento em alto desempenho e, sob certa perspectiva, torna-se o foco da área; a programação em si passa a ser um artifício para se valer dos recursos disponíveis no *hardware* (GROPP; LUSK; SKJELLUM, 1999).

Considerando esta importância, torna-se conveniente apresentar alguns conceitos da parte física, a fim de contextualizar o leitor sobre o *modelo* de programação paralela e o porquê deste modelo depender fortemente do *hardware* sobre o qual opera.

2.1.1 Hardware

A classificação original de computadores paralelos é conhecida como *Taxonomia de Flynn*. Michael Flynn classificou as arquiteturas paralelas quanto ao número de fluxos de dados e número de fluxo de instruções. A máquina de *Von Neumann*, precursora das arquiteturas modernas, por exemplo, possui um fluxo de instrução e um fluxo de dados. Desta maneira, é classificada como “*single-instruction, single-data*” (SISD). No extremo oposto, temos o modelo “*multiple-instruction, multiple-data*” (MIMD), máquinas de vários processadores, executando instruções em paralelo sobre diferentes dados (FLYNN, 1972).

2.1.1.1 SISD

Máquinas nesta classificação remetem ao modelo clássico da máquina de *Von Neumann*; não há paralelismo e tudo é sequencial. Neste modelo, há uma memória, representada por um grande bloco, e um processador. Entre processador e memória são movimentados dados e instruções, através de um barramento.

O “gargalo de *Von Neumann*” é justamente o acesso à memória. Por mais rápidos que

sejam os processadores, a latência de acesso à memória não decresce na mesma medida em que a velocidade dos primeiros aumenta. Há, portanto, um decréscimo de desempenho considerável. Como consequência, poucas máquinas, atualmente, seguem *estritamente* o modelo de *Von Neumann*. Utilizam-se, na maioria das vezes, memórias hierárquicas, com modelos baseados em memória *cache*¹, que aproveitam-se do princípio da localidade² para aumentar o desempenho (PATTERSON; HENNESSY, 2005).

É possível estender a arquitetura SISD através de *pipelining* e o processamento vetorial. *Pipelining* é uma técnica bastante conhecida e melhora o aspecto do desempenho ao quebrar instruções de máquina em *microinstruções*; dessa maneira, é possível começar o processamento de uma instrução posterior sem que o ciclo da instrução corrente esteja finalizado. Ser vetorial significa, em última instância, adicionar novas instruções ao conjunto de instruções da máquina para que ela forneça o processamento do mesmo comando para vários dados em paralelo.(NAVAUX; ROSE, 2003)

É importante salientar que há certa discordância sobre a natureza dos processadores vetoriais. Existem classificações em que, por exemplo, estes processadores capazes de operar sobre vetores são vistos como máquinas SIMD (TANENBAUM, 1995). Esta classificação acaba variando entre autores; alguns afirmam que máquinas vetoriais são MISD, outros que máquinas MISD sequer existem (são, portanto, variações de SIMD) e, ainda, há aqueles que nem classificam as máquinas vetoriais como paralelas. Esta falta de uniformidade é natural; fluxos de instruções são abstrações em nível de *hardware* e, portanto, são relativos (PACHECO, 1997).

O principal ponto fraco destas técnicas é que processadores vetoriais e processadores *pipeline*, em geral, não “escalam” bem, isto é, não são facilmente modificáveis, do ponto de vista do *hardware*, para processar desafios maiores (NAVAUX; ROSE, 2003).

2.1.1.2 SIMD

Há distinção entre um sistema SIMD puro e processadores vetoriais. A definição canônica de um processador SIMD é ter uma UCP mestra e várias subordinadas; a cada ciclo, a UCP mestra faz o *broadcast* de uma instrução para as UCPs subordinadas operarem sobre sua pequena porção de memória. Assim, as UCPs subordinadas ou executam essa instrução ou ficam ociosas. Diante deste enquadramento, máquinas vetoriais são vistas como monoprocessadas e este processador é que tem extensões multiprocessadas. Conforme mencionado anteriormente, no entanto, essa definição é relativa (NAVAUX; ROSE, 2003).

A desvantagem de sistemas SIMD reside nos códigos com muitos *branches* condicionais ou que dependam muito de estruturas condicionais. É muito provável que vários processos fiquem ociosos por vários períodos de tempo. Seu principal trunfo é a fácil programação (se o problema abordado possui uma estrutura regular). O custo de comunicação é alto, mas igual ao de máquinas MIMD de memória distribuída (revisadas adiante). Por fim, elas possuem excelente escalabilidade (TANENBAUM, 2003).

2.1.1.3 MISD

De acordo com a *Taxonomia de Flynn*, não existem máquinas que satisfaçam esta classificação (PATTERSON; HENNESSY, 2003). Alguns autores (ROOSTA, 1999) clas-

¹ Memórias de tamanho e velocidade de acesso intermediário entre memória principal e os registradores (PATTERSON; HENNESSY, 2005).

² Tal princípio enumera que dados utilizados em lugares próximos na execução do programa tendem a estar dispostos assim também na memória (PATTERSON; HENNESSY, 2005).

sificam máquinas vetoriais como máquinas MISD, mas a bibliografia não é uniforme sobre o assunto.

2.1.1.4 MIMD

A diferença fundamental entre máquinas MIMD e SIMD é que os processadores MIMD são autônomos, não precisam todos executar o mesmo código (todos os processadores são UCPs individuais, com seus respectivos componentes). Ao contrário de máquinas SIMD, modelos MIMD são assíncronos (sem relógio global) e devem ser programados para se sincronizarem, se esta for a intenção. O mundo MIMD é dividido em dois blocos: os de memória compartilhada (multiprocessadores) e os de memória distribuída (multicomputadores) (TANENBAUM, 1995).

Multiprocessadores são um conjunto de processadores e módulos de memória ligados por uma rede. Desta maneira, podem se classificar como (NAVAUX; ROSE, 2003):

Arquitetura baseada em barramento. É a mais simples forma de conexão. Acesso à memória através de barramento. Não se comporta bem para um grande número de processadores, visto que o barramento fica sobrecarregado. Justamente por isso, não é uma solução escalável em grande porte. Para contornar este problema, usualmente os processadores têm grande quantidade de memória *cache*.

Arquitetura baseada em switches. Usa switches para gerenciar o acesso aos módulos da memória. Um exemplo disso é uma arquitetura do tipo “*cross-bar switch*”. Esta arquitetura se caracteriza por um engranzamento (*mesh*) retangular, com *switches* nas intersecções e terminais nas bordas esquerda e superior. Processadores e módulos de memória podem ser conectados aos terminais. Os *switches* podem permitir que um sinal se propague na vertical e horizontal simultaneamente ou redirecionar o sinal de um eixo para o outro. Então, por exemplo, se tivermos processadores à esquerda e memória acima, qualquer processador pode acessar qualquer módulo de memória ao mesmo tempo que algum outro processador acessa algum outro módulo. Assim, não se sofre com o problema de saturação encontrado em barramentos. Infelizmente, este tipo de arquitetura tende a ser muito caro. Uma matriz $m \times n$ requeria $m \times n$ *switches*. Logo, a maioria destes sistemas são pequenos.

Multiprocessadores, independentemente da disposição física dos módulos de memória, possuem um *sistema de memória compartilhada*, ou seja, todos os processadores têm uma visão lógica de um único espaço de endereçamento pertencente a uma memória global (PATTERSON; HENNESSY, 2003). Um esquema de um multiprocessador clássico é apresentado na Figura 2.1.

No sistema de *Multicomputadores*, cada processador tem sua memória privativa, ou seja, é um *sistema de memória distribuída*. Isto significa, basicamente, que cada processador tem um espaço de endereçamento próprio e diferente dos demais (TANENBAUM, 1995). Se observarmos tais sistemas como grafos, veremos dois tipos de grafos: aqueles cujos vértices correspondem ao par processador/memória (nodos) e aqueles em que alguns vértices são nodos e outros *switches*. Redes do primeiro tipo são “redes estáticas” e do segundo tipo, “redes dinâmicas” (PATTERSON; HENNESSY, 2005). Uma ilustração de um sistema multicomputador pode ser visto na Figura 2.2

Dos quatro modelos propostos pela *Taxonomia de Flynn*, o que se torna relevante para o estudo são as máquinas MIMD; *clusters* e *grids* são exemplos clássicos de máquinas MIMD e, atualmente, são as estruturas funcionais que oferecem um maior poder de processamento em razão de sua escalabilidade (DONGARRA et al., 2003).

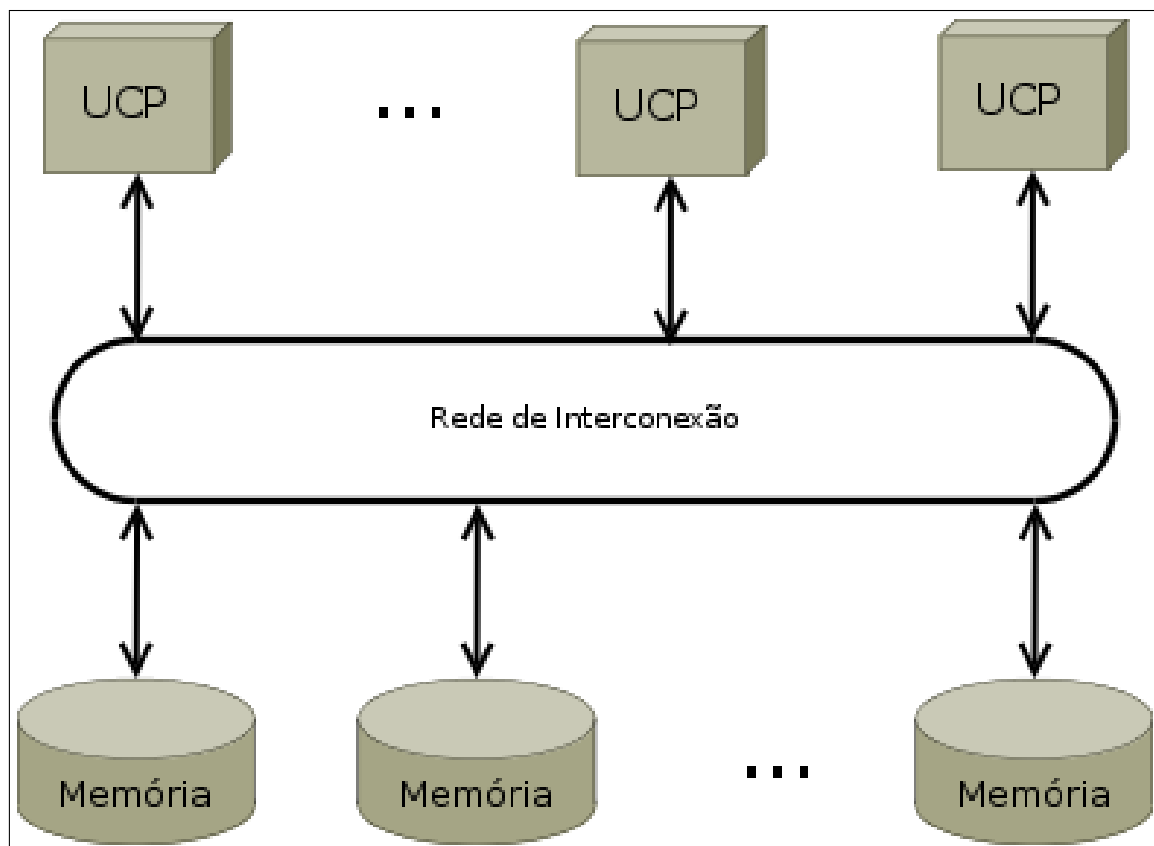


Figura 2.1: Modelo clássico de multiprocessadores.

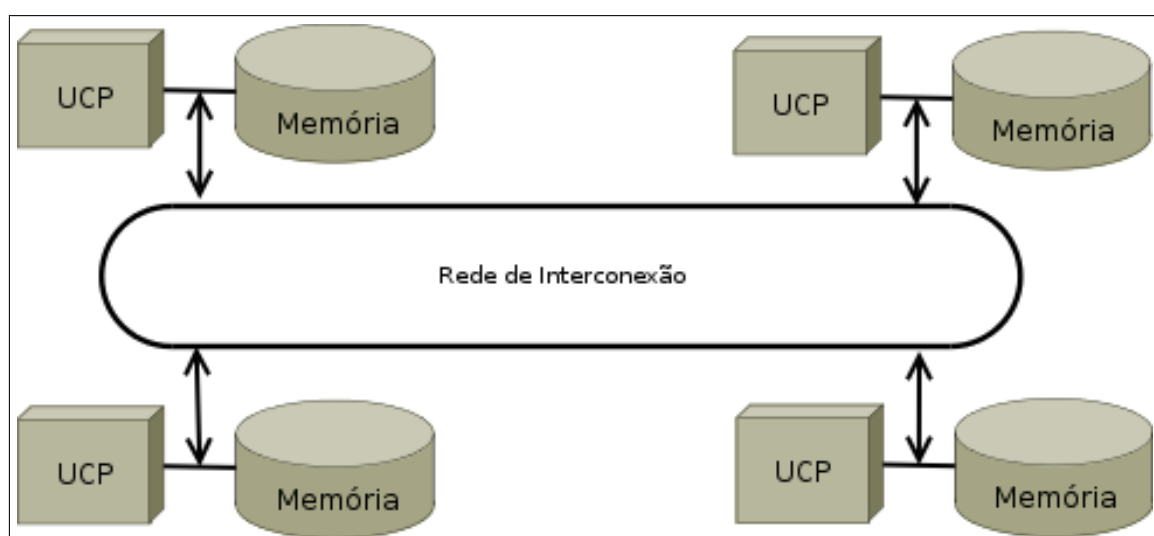


Figura 2.2: Modelo clássico de multicomputadores.

Um *cluster* é um computador paralelo, com um ponto de acesso centralizado. A visão que o usuário tem ao programar é de um grande computador com vários processadores paralelos. *Grids*, por outro lado, são computadores naturalmente heterogêneos, com pontos de acesso distribuídos. Além disso, a disposição e tipo dos nodos, dependendo da arquitetura da máquina, pode sofrer alterações a qualquer momento. O usuário enxerga o *grid* como um agregado de computadores que aproveitam os tempos ociosos uns dos outros para melhorar o desempenho. (NAVAUX; ROSE, 2003).

2.1.2 Modelos Computacionais e Comunicação

Além de ter definido um *hardware* (máquinas MIMD, no caso), é necessário definir um *modelo* de computação paralela que rode sobre este *hardware*. Modelos computacionais

[...] são uma visão conceitual dos tipos de operações disponíveis ao programa. Estes *não* incluem sintaxe específica de uma linguagem de programação ou biblioteca em particular, e são (quase) independentes do nível de *hardware* que os suporta (GROPP; LUSK; THAKUR, 1999, p. 3).

Alguns exemplos de tópicos que influem sobre a classificação são o compartilhamento (ou não) de memória, a quantidade de comunicação que está no *hardware* ou *software*, a unidade de execução básica e *etc.*

Alguns modelos-exemplo são (GROPP; LUSK; THAKUR, 1999):

Paralelismo de Dados. Introduzido com os processadores vetoriais que, à época, ofereciam paralelismo apenas a nível de dados; o programa era, sob todos os outros aspectos, seqüencial, com instruções que operassem em paralelo. Com o passar do tempo, o paralelismo de dados passou a ser enquadrado mais como uma *técnica de programação* do que uma arquitetura propriamente dita. O modelo, no entanto, continua o mesmo: todo o paralelismo continua advindo dos dados. Muito desse trabalho, hoje em dia, é feito pelo compilador.

Exemplo de implementação: HPF (*High Performance FORTRAN*) (KOELBEL; LOVEMAN; SCHREIBER, 1993).

Memória Compartilhada. No modelo oposto ao *Paralelismo de Dados*, o paralelismo não é extraído da independência implícita de certa parte do código seqüencial. Ao invés disso, o paralelismo é explicitado diretamente pelo programador, sendo chamado de *Paralelismo de Controle*³.

Memória compartilhada é uma técnica de comunicação no *Paralelismo de Controle*. Através dela todos os processadores acessam todo o espaço de endereçamento da memória e, portanto, compartilham variáveis. É um modelo extremamente eficiente, pois não envolve custos de comunicação, já que ela é toda feita através da memória principal. Este modelo, no entanto, traz dois inconvenientes:

1. Repassa as questões de sincronização para o programador; a consistência das variáveis deve ser garantida no nível de programação.

³Muito embora (GROPP; LUSK; THAKUR, 1999) omita qualquer referência, um modelo misto é perfeitamente factível; nada impede a existência de um modelo que possui, simultaneamente, paralelismo implícito e explícito.

2. Para mais do que algumas dezenas de processadores o acesso à memória fica comprometido, pois existe disputa pelo barramento e por células de armazenamento. Logo, o desempenho também acaba sofrendo um comprometimento.

Exemplo de implementação: OpenMP (*Open Multi-Processing*) (OPENMP C AND C++ APPLICATION PROGRAM INTERFACE, 2002)

Troca de Mensagens. É um modelo conceitualmente oposto ao modelo de Memória Compartilhada. Ao invés de uma memória global, acessada e compartilhada por todos, existe apenas uma memória local para cada processador, cujo único postulante a acessar é o próprio. Qualquer troca de informações é feita pelo envio e recebimento de mensagens, através de primitivas especiais. É um modelo mais seguro em termos de sincronização, mas também é mais caro, pois trocas de mensagens são típicas operações de E/S. Baseia-se no uso de primitivas do tipo `send()` e `receive()` sincronizadas entre processos. É, portanto, um modelo mais complexo do ponto de vista do programador.

Exemplo de implementação: PVM (*Parallel Virtual Machine*) (GEIST et al., 1994) e MPI (GROPP; LUSK; SKJELLUM, 1999).

Operações em Memória Remota. É um misto dos modelos de *Memória Compartilhada* e *Troca de Mensagens*. Existe uma única memória, acessada por todos (mas não diretamente), e cada processador também tem uma memória local disponível. A memória global é acessada via primitivas (tipicamente `put()` e `get()`) para escrita e leitura. Esta medida implica, diretamente, que os processos não precisam mais sincronizar-se para trocar informações; uma primitiva `send()` não mais necessita de um correspondente `receive()`. Esta estrutura remete ao modelo clássico de *Distributed Shared Memory* (NAVAUX; ROSE, 2003).

Este modelo consegue simplificar a complexidade da programação com troca de mensagens e soma algum desempenho de um modelo com memória compartilhada, uma vez que o acesso à memória passa a ser mais eficiente sem a sincronização de primitivas de E/S. Além disso, tal modelo inclui um novo recurso chamado de “Mensagem Ativa”, que consiste na execução de subrotinas de um processo no espaço de endereçamento de outro. Tal operação geralmente é usada para fins de cópia de memória remota, emulando `send()` e `receive()` de maneira unilateral.

Exemplo de implementação: a especificação MPI-2 (GROPP; LUSK; THAKUR, 1999) fornece recursos para programação com operações em memória remota.

Threads. Os primeiros modelos de *Memória Compartilhada* usavam acessos a espaços de endereçamento locais. A obtenção de memória compartilhada era feita através de uma primitiva especial (parecido com `malloc()` da linguagem C). Hoje em dia, no entanto, se assume um modelo onde toda a memória é distribuída. Isto dá espaço para a aplicação do modelo em sistemas *multithread*.

Tendo-se um processo como a definição intuitiva de um programa em execução (TOSCANI; OLIVEIRA; SILVA CARÍSSIMI, 2002), uma *thread* é um fluxo de execução de processo⁴ (TOSCANI; OLIVEIRA; SILVA CARÍSSIMI, 2003). Qualquer comunicação entre *threads* é feita através da memória principal, compartilhada. Variáveis locais, no entanto, ainda permanecem privativas para cada *thread*.

⁴Na verdade, pode-se, ainda, definir um fluxo de execução como registradores, *Program Counter*, *Stack Pointer* e alguns outros dados de contexto fortemente dependentes da arquitetura.

Como o chaveamento de uma *thread* para outra não requer ação explícita de programador, tal modelo aproxima-se de um modelo de Programação Paralela mesmo que, muitas vezes, seja executado em máquinas monoprocessadas.

Exemplo de implementação: Pthreads (*Posix Threads*) (BUTENHOF, 1997).

Dentre os modelos de programação paralela, atualmente, o modelo mais utilizado na programação de *clusters* é o de comunicação por *troca de mensagens*. Tal modelo também é implementado pela MPI, como será visto adiante. Alguns fatores que contribuem para esta estatística são (GROPP; LUSK; THAKUR, 1999):

Universalidade. A maioria das máquinas paralelas atuais são agregados de processadores conectados por uma rede de interconexão. Troca de mensagens ajusta-se bem a este tipo lógica computacional; adapta-se, por tanto, com facilidade ao *hardware* atual.

Expressividade. Algoritmos paralelos tendem a ser melhor expressos em termos de troca de mensagens do que leituras/escritas em memória global, por permitirem certo controle sobre a sincronização dos eventos.

Facilidade de depuração. Depurar programas paralelos permanece sendo um desafio considerável na área. Embora *depuradores* sejam mais facilmente construídos no modelo de *memória distribuída*, o processo de depuração, em si, é melhor realizado no modelo de *troca de mensagens*, visto que a maioria dos erros ocorre por sobrescrita inesperada na memória.

Performance. No sistema de *memória compartilhada* existe apenas uma memória global, com uma hierarquia de *cache* e memória para *todos* os processos. Embora o acesso a esta memória seja mais rápido que a sincronização por `send()`/`receive()`, não é possível aproveitar todo o potencial do sistema de hierarquias, causando sério prejuízo ao desempenho. O modelo de comunicação por *troca de mensagens* associa trocas de dados e consultas à memória a cada processo; é uma abordagem mais eficiente para o aproveitamento de princípio da localidade⁵, fundamental para o melhor desempenho da *cache* do sistema.

2.1.2.1 Arquiteturas Multicore

Processadores de vários núcleos encontram-se numa fase de transição entre arquiteturas de 2 e arquiteturas de 4 núcleos. Atualmente a comunicação entre os núcleos é feita via modelo de Memória Compartilhada (memória principal), variando o comportamento da memória *cache* (compartilhada ou privativa) de acordo com o modelo de processador (CREEGER, 2005).

No entanto, com o aumento considerável do número de núcleos, tal modelo se torna problemático, visto que há disputa intensa por barramentos. O modelo de comunicação que será adotado no futuro ainda é uma questão em aberto, mas há certas tendências de que se adote um modelo próximo à troca de mensagens; o *Intel TeraFLOP*, máquina com 80 núcleos da *Intel*, adota uma abordagem NOC (*Network on Chip*) para fazer comunicação entre os seus processadores, com emprego de roteadores e *switches*, onde dados são

⁵Baseia-se no fato de que dados próximos tendem a ser usados em trechos próximos do código. Trata-se de uma das bases da utilização de uma hierarquia de memória.

transferidos como numa rede de computadores clássica (TCP/IP) (MATTSON; HENRY, 1998).

Arquiteturas *Multicore* proporcionam abordagens mistas de alguns modelos. O uso destes *chips* em *clusters* tende a seguir um destes modelos mistos. Internamente, em cada *chip multicore*, usa-se o modelo de *threads* entre os processos do Sistema Operacional (e um modelo de memória compartilhada entre os núcleos), o que garante uma comunicação extremamente rápida. Cada nó, entretanto, comunica-se por um modelo de *Troca de Mensagens*, através de uma rede específica.

2.1.3 Desempenho

Até agora todo embasamento apresentado para o uso de programação paralela foi conceitual; organização do *hardware* e comunicação de processos foram o tema com maior ênfase. Existe, no entanto, um embasamento teórico importante para o emprego de Programação Paralela. É essencial que exista um modelo de referência para a medição e formalização dos conceitos de *desempenho* e *eficiência*.

Um conceito fundamental na área de programação paralela é o conceito de *speedup*. *Speedup* é o ganho de desempenho proporcionado pela adoção de um algoritmo paralelo (DONGARRA et al., 2003); Uma abordagem para avaliação do *speedup* é a razão entre o número de operações realizadas por uma execução seqüencial e o número de operações realizada por uma UCP na execução em paralelo (JAJA, 1992). Define-se a notação utilizada na Tabela 2.1

Notação	Significado
P	Problema computacional.
A	Algoritmo paralelo utilizado.
n	Tamanho da entrada.
p	Número de processadores (UCPs)
$T^*(n)$	Número de operações do <i>melhor</i> algoritmo <i>seqüencial</i> para uma entrada de tamanho n .
$T_p(n)$	Número de operações para cada UCP do algoritmo A para p processadores e uma entrada de tamanho n .
$S_p(n)$	<i>Speedup</i> obtido com p processadores e uma entrada de tamanho n .

Tabela 2.1: Notação do cálculo do *Speedup*.

A partir da notação empregada, define-se *speedup* como:

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \quad (2.1)$$

É importante ressaltar que a base da fórmula é o *melhor* algoritmo *seqüencial* para o problema ($T^*(n)$) e não o algoritmo paralelo executado com 1 processador ($T_1(n)$); embora seus tempos de execução sejam bem próximos, existem alguns testes feitos pelo algoritmo paralelo que independem do número de processadores que participarão da execução (e.g., teste que determina o número de processadores ativos) e que adicionam elementos ao cálculo da complexidade sem relação com o problema em si (JAJA, 1992). Outro ponto que se faz destacar é que deve ser utilizado o *melhor algoritmo* seqüencial para este

mesmo cálculo. Dessa maneira, tal algoritmo não necessita seguir o *modus-operandi* do algoritmo paralelo; ambos podem resolver o problema de maneira distinta.

Embora adequada, tal abordagem (número de operações executadas) é complexa de ser precisada e medida, visto que é difícil definir precisamente o conceito de operação. Logo, adota-se uma abordagem mais simples para o *speedup*, onde este é definido como a razão entre tempo de processamento em um processador e o tempo de processamento em uma configuração paralela (DONGARRA et al., 2003). Tem-se, então, a equação:

$$Speedup(n) = \frac{T(1)}{T(n)} \quad (2.2)$$

A medida que n (número de processadores) cresce, $T(n)$ (tempo de execução para n processadores) decresce e o *speedup* aumenta.

É interessante, também, apresentar o conceito de *eficiência*. A *eficiência* é a medida de utilização efetiva de todos os p processadores em uma execução paralela; representa, conceitualmente, o quanto o desempenho se beneficia do aumento do número de processadores. Um índice de eficiência máximo ($E_p(n) = 1$) é obtido quando a execução de A com p processadores é p vezes mais rápida do que a execução de A com 1 processador (JAJA, 1992).

Pode-se expressar *eficiência* pela fórmula

$$E(p) = \frac{T(1)}{p \times T(p)} \quad (2.3)$$

O ideal seria que o tempo de A se dividisse com o número de processadores; a execução com quatro 4 processadores, por exemplo, seria 4 vezes mais rápida que a execução com 1 processador. Sob esta óptica, torna-se óbvio que, ao se multiplicar o tempo de execução de A com 4 processadores pelo número total de processadores obtém-se, novamente, o tempo de execução com 1 processador.

Isto, no entanto, se configura somente numa situação ideal. Na maior parte dos casos, $0 < E_p(n) < 1$. Ocorre que existem custos de comunicação e sincronização que consomem tempo significativo durante a execução do algoritmo.

Por fim, é importante ressaltar que a *eficiência* é calculada com a base do algoritmo paralelo com 1 processador e não com o algoritmo sequencial; conforme explicitado na parte sobre *speedup*, o algoritmo paralelo não necessita ser uma mera versão portada do sequencial e, dessa maneira, não discorre sobre a *escalabilidade* do algoritmo ao variar-se o número de processadores. Esta abordagem está mais de acordo com o cálculo do *speedup* simplificado apresentado anteriormente.

Com a base teórica sobre arquiteturas de computadores e análise de desempenho já estabelecida, falta, ainda, apresentar o ferramental empregado em nível de *software* para a programação paralela, o MPI. A próxima seção trata disso.

2.2 Introdução ao MPI

MPI não é uma biblioteca, uma linguagem ou um jeito novo de programar. MPI é, na verdade, a especificação de uma *interface* para a implementação do modelo de comunicação por troca de mensagens (PACHECO, 1997). Esta seção foca a história, a filosofia de implementação e as motivações que levaram à especificação. Um aprofundamento técnico, com detalhes de implementação e primitivas básicas, pode ser visto no Capítulo 3.

2.2.1 Motivação

O padrão de comunicação por *troca de mensagens* demorou a se consolidar. A principal razão para essa situação ter ocorrido foi a falta de um *padrão de fato* a ser adotado pelos usuários de máquinas paralelas. Desta maneira, cada vendedor implementava bibliotecas para troca de mensagens funcionais no *seu* sistema; emergia, assim, a falta de portabilidade de tais sistemas, ao mesmo tempo em que, por se tratar de uma área nova, a Computação Paralela não havia sido suficientemente experimentada para que se abstraíssem os conceitos mais úteis na implementação de algoritmos (GROPP; LUSK; THAKUR, 1999).

Surgiram muitas tentativas dentro da comunidade científica para a implementação de um padrão (*e.g.*, PVM). Tal diversidade foi proveitosa para o desenvolvimento do estado da arte do tópico, mas acabou por gerar acirrado conflito e concorrência; cada novo padrão vinha a fragmentar uma parcela dos usuários o que resultava em pessoas especialistas em um ou outro destes padrões. Um grande entrave na utilização generalizada destas bibliotecas é que elas não se comunicavam, ou seja, não havia portabilidade de um sistema para o outro (DONGARRA et al., 2003).

Havia, então, uma dualidade na área, consistindo de adotar um sistema extremamente portátil (em geral, “meta-bibliotecas” sobre as bibliotecas existentes), mas escasso em funcionalidades, ou um sistema robusto, com muitas funções, mas de uso muito específico. *Sockets*, por exemplo, consistem no uso mais generalizado que se poderia implementar troca de mensagens em Programação Paralela. No entanto, as mais diversas implementações de *sockets* não oferecem qualquer facilidade em termos de algoritmos paralelos. (GROPP; LUSK; THAKUR, 1999).

2.2.2 História

A diversidade de soluções para a implementação de um modelo de comunicação por *troca de mensagens*, que introduziu as questões acima, atingiu um ponto largo de saturação; um padrão que aliasse portabilidade, performance e recursos passou ser um requisito demandado pela comunidade de usuários e fornecedores.

Na conferência *Supercomputing '92* (em Novembro), formou-se um comitê para a criação de um padrão a ser adotado na implementação do supracitado modelo. Uma enumeração dos principais objetivos deste comitê seria (GROPP; LUSK; THAKUR, 1999):

- definir um padrão portátil para a troca de mensagens, que não seria uma padrão oficial, mas que potencialmente atrairia implementadores e usuários;
- operar de uma maneira completamente aberta, permitindo a todos juntarem-se às discussões; e
- ser terminado em um ano.

Dos contrastes entre os três objetivos, foi completado, em Maio de 1994, a referência para o padrão MPI. O esforço para a criação do padrão foi vívido e ativo, fruto da participação ampla de uma comunidade focada. Adicionalmente, cumpriu sua meta ao atrair tanto implementadores quanto usuários, em grande parte por ser aberto a qualquer opinião; ambos os pontos de vista puderam ser discutidos. Assim, após sucessivas reuniões do fórum no período entre 1993 e 1995, surgiu o MPI (GROPP; LUSK; THAKUR, 1999).

Em 1995-1997 o fórum tornou a deliberar sobre extensões à norma MPI original; foram incluídos advenços para E/S paralela, gerenciamento dinâmico de processos, operações em memória remota e vários outros recursos. Como resultado deste esforço, o MPI foi estendido para sua segunda versão, o MPI - 2.

2.2.3 Definição e Características

MPI é uma especificação de uma biblioteca para a troca de mensagens em *C*, *C++* e *FORTRAN*, que especifica como devem ser (sintática e semanticamente) as estruturas para estes fins nestas três linguagens (*e.g.*, Funções, Classes, Subrotinas, *etc.*). MPI *não* é uma implementação específica; qualquer interessado pode implementar o padrão da maneira que preferir. Um programa MPI corretamente escrito deve rodar em todas as implementações MPI (GROPP; LUSK; THAKUR, 1999).

Além das estruturas de `send()` e `receive()` básicos, MPI provê, também, estruturas para a comunicação coletiva, ou seja, computações que envolvem todos ou um grande número de processos. Tal comunicação pode envolver o transporte de dados (*e.g.*, primitivas do tipo `send_broadcast()`) ou computações coletivas (*e.g.*, mínimos, máximos, soma, *etc.*) (PACHECO, 1997).

MPI oferece suporte para vários tipos de primitivas `send()` e `receive()`. Pode-se usá-las em modo bloqueante, esperando que a operação de comunicação se realize para o prosseguimento da computação, ou não-bloqueante, caso contrário. Além disso, existem outros modos, frutos da combinação dos anteriores, como (GROPP; LUSK; THAKUR, 1999)

Modo Síncrono. Primitivas do tipo `send()` só desbloqueiam quando o respectivo `receive()` houver ocorrido;

Modo de Prontidão (*Ready*). Para `send()`, provê ao programador maneiras de avisar o sistema do uso de uma primitiva `receive()` já postada no processo destino, para que a camada de baixo use um protocolo mais rápido, se disponível;

Modo *Buffered*. Operações de `send()` têm o controle de *buffer* feito pelo usuário; e

Modo Padrão. É a configuração normal das primitivas `send()` e `receive()` do MPI. O `receive()` é bloqueante e o `send()` é bloqueante apenas enquanto o *buffer* não for liberado.

Outra característica interessante do MPI são as topologias virtuais, ou seja, modelos de disposição da comunicação entre processos (*e.g.*, árvore, grafo, plano cartesiano, *etc.*) independentes da real implementação (física) dos processadores sobre os quais estes rodam. Com elas, provém-se um método em alto-nível de manipular-se grupos de processos e, ao mesmo tempo, abstrair sua localização real (DONGARRA et al., 2003).

Uma característica marcante, insistentemente trabalhada no desenvolvimento da especificação MPI, é o fato de que o suporte à múltiplos recursos é intrínseco à norma. Destes, se destacam (GROPP; LUSK; SKJELLUM, 1999):

- O suporte à depuração nativo, através de ferramentas que permitem colocar “ganchos” ao longo do código e, posteriormente, interceptar chamadas MPI através desses mecanismos;

- O suporte à implementação de bibliotecas paralelas personalizadas, que usam o MPI como base. Devido à noção de comunicadores, é possível construir bibliotecas independentes de código de usuário, com atributos e controladores de erros personalizados.

Uma das capacidades-chave especificadas no MPI é sua característica de executar em *redes heterogêneas*, pois tem uma definição de tipos portátil e compatível com diferentes arquiteturas. Desta maneira, diferentes tipos e estruturas de dados são convertidos de acordo com a implementação do MPI; os implementadores têm total liberdade em decidir como as conversões são realizadas, de acordo com a especificação.

Outra característica (que dá suporte a redes heterogêneas) é a abstração entre processadores e processos no sistema. A especificação MPI fala de processos e não de processadores; vai depender da implementação o número máximo de processos que podem executar sobre um único processador. Além disso, também fica a cargo dos implementadores definir as fronteiras, para a respectiva biblioteca, entre nós (no caso *clusters*) e processadores, pois um nó pode conter n processadores e é necessário estabelecer esta relação com os processos MPI. Normalmente, um processo MPI corresponde a um processo de Sistema Operacional. No entanto, isto não é necessariamente verdade; existem implementações sobre o UNIX em que vários processos MPI são mapeados sobre um processo UNIX (GROPP; LUSK; THAKUR, 1999).

Unindo uma definição de tipos suficientemente portátil e uma política flexível de mapeamento processador-processo, MPI consegue dar uma visão de máquina virtual ao usuário; pode-se enxergar apenas processos e mensagens, de acordo com a topologia desejada (por meio das topologias virtuais) sem a preocupação do *hardware* sobre o qual este sistema opera. Tal flexibilidade fornece ao MPI uma grande robustez no quesito portabilidade, conforme almejado na sua feitura (PACHECO, 1997).

3 MPI

Este capítulo versa sobre aspectos mais técnicos e aplicados da especificação MPI (versão 1.2). Serão apresentadas e descritas as primitivas básicas e exemplos de código (comentados) serão utilizados para exemplificar seu uso.

3.1 As Seis Primitivas Básicas

MPI é uma especificação abrangente, com muitos recursos. No entanto, para qualquer algoritmo paralelo que se queira utilizar, MPI pode ser tão simples quanto o uso de 6 funções básicas. Com estas funções, é possível implementar quase qualquer programa MPI; a maioria das outras subrotinas são simplificações e combinações destas primitivas de acordo com estruturas comuns em Programação Paralela.

A Tabela 3.1 mostra as seis primitivas principais usadas no MPI.

Primitiva	Função
MPI_Init	Inicializa o MPI
MPI_Finalize	Encerra o MPI.
MPI_Comm_size	Retorna o número de processos rodando.
MPI_Comm_rank	Retorna que processo eu sou.
MPI_Send	Manda uma mensagem.
MPI_Recv	Recebe uma mensagem.

Tabela 3.1: As seis funções básicas do MPI.

3.1.1 Inicialização e Finalização

MPI possui duas primitivas destinada a isso, cujo protótipo é apresentado na Figura 3.1.

```
int MPI_Init(int* argc char** argv[])
int MPI_Finalize()
```

Figura 3.1: Protótipos de MPI_Init e MPI_Finalize.

Todo o código MPI (ou seja, que se utilize de funções da biblioteca de implementação do MPI) utilizado *deve* estar contido entre estas duas primitivas; elas se encarregam de inicializar e instanciar estruturas de dados em um nível de abstração transparente ao usuário e também desalocam estas estruturas ao sair da área delimitada. Uma exceção a

esta regra é a função `MPI_Wtime`, utilizada para medição de tempos, que será visitada mais adiante.

3.1.2 Identificação de Processos e Comunicadores

É muito útil agrupar-se os processos de um modo conveniente para a resolução do problema (*e.g.*, processos produtores e consumidores de recursos). MPI oferece duas estruturas para esta manipulação: grupos e comunicadores.

Um grupo é simplesmente um conjunto de processos, sem qualquer propriedade sobre sua disposição. Pode ser usado como referência para a inclusão ou exclusão de processos em um canal e enxerga somente os processos de seu escopo. Grupos possuem um tipo específico em MPI, o `MPI_Group`.

A peça fundamental para a execução de um programa MPI, no entanto, são os *comunicadores*. Pode-se entender um comunicador como um agregado que guarda informações de um grupo de processos e da disposição (*e.g.*, grafo, árvore, plano cartesiano) pela qual estes processos relacionam-se. Desta maneira, esta estrutura permite referenciar um dado processo através de um identificador de processo, que, *a priori*, deixa transparecer sua topologia. Este é o mecanismo básico para se referenciar um processo; a fonte e o destino de qualquer mensagem é identificado univocamente por um identificador e o comunicador que fornece aquela referência.

A princípio, durante a execução de um programa MPI, todos os processos pertencem ao comunicador `MPI_COMM_WORLD`, uma constante da especificação que fornece um identificador de processo único e seqüencial (um inteiro) para todos os processos que foram disparados na inicialização do programa. O usuário pode, quando desejar, criar novos comunicadores, com todos ou com somente alguns processos, na topologia em que preferir. Também é possível extrair o grupo de processos de qualquer comunicador, inclusive o `MPI_COMM_WORLD`.

Existem duas funções de suma importância para o MPI que fazem uso de comunicadores. Elas são apresentadas na Figura 3.2

```
int MPI_Comm_size(MPI_Comm comm, int* size)
int MPI_Comm_rank(MPI_Comm comm, int* rank)
```

Figura 3.2: Protótipos de `MPI_Comm_size()` e `MPI_Comm_rank()`.

A função `MPI_Comm_size` retorna o tamanho do comunicador (número de processos contidos) `comm` na variável `size`. Ao ser usado em conjunto com `MPI_COMM_WORLD` como argumento, retorna o número total de processos participantes da execução.

`MPI_Comm_rank` retorna o identificador do processo dentro do comunicador passado como parâmetro. Se utilizado com `MPI_COMM_WORLD`, retorna o identificador do processo dentre todos os processos que estão sendo executados. Neste caso, seu *rank* é um número que varia, para p processos, de 0 até $p - 1$.

3.1.3 Primeiro Exemplo

A seguir é apresentado o código de um programa simples, que exhibe uma mensagem na tela no clássico estilo “*Hello World*”. O código *não* inclui troca de mensagens, a fim de refletir o que foi apresentado até aqui e apresentar o uso das primitivas `MPI_Init`, `MPI_Finalize`, `MPI_Comm_size` e `MPI_Comm_rank`.

Para executar o programa é necessário utilizar os arquivos de inclusão vistos na Figura 3.3.

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

Figura 3.3: Arquivos de inclusão MPI.

O arquivo `mpi.h` contém toda a especificação e os protótipos para a utilização das bibliotecas MPI. O arquivo `stdio.h`, conforme conhecido, inclui funções para a manipulação de E/S num terminal de execução.

As variáveis utilizadas no código são vistas na Figura 3.4

```
int rank; /* Rank of process. */
int p;    /* Number of processes. */
```

Figura 3.4: Variáveis do programa *Hello World MPI*.

A variável `rank` armazenará o identificador do processo. A variável `p`, por sua vez, armazena o número de processos. O que o programa faz é, para cada processo, imprimir seu identificador. Adicionalmente, faz distinção entre dois processos: o processo raiz e o processo de maior *rank*. Esta divisão não é necessária no momento; mais adiante, nos exemplos posteriores, ela revelará utilidade fundamental. O código do programa pode ser visto na Figura 3.5.

Para compilar o programa, é necessário fazer a ligação com a biblioteca MPI, nomeada de maneira específica em cada distribuição. Para facilitar o trabalho de pesquisa, existe um *script*, incluído na maioria das distribuições MPI, que faz essas ligações de maneira transparente ao usuário, chamado `mpicc`. Para compilar o código apresentado, assumindo que o nome do arquivo seja `hello_world.c` e querendo um nome de `HelloWorld` para o executável, basta invocar

```
mpicc -o HelloWorld hello_world.c
```

e, se nenhum erro ocorrer, o executável `HelloWorld` será criado.

No entanto, não basta apenas compilar o programa de maneira acertada; é preciso executá-lo apropriadamente. Os processadores e arquiteturas atuais não “entendem” a alocação de processos MPI; não há qualquer suporte no nível de *hardware* que distribua os processos MPI e os mapeie sobre processos do Sistema Operacional. Logo, é necessário uma camada de *software* (em geral, um processo de Sistema Operacional no modo *daemon*¹) a qual se encarregará de realizar este mapeamento (*e.g.*, o processo `lamd` na distribuição LAM-MPI) e, também, de se comunicar com outras máquinas, com o mesmo serviço, especificadas para participarem da computação.

Após o processo *daemon* ser executado e estiver rodando (tal processo e a maneira de fazê-lo é específico de cada distribuição) é necessário utilizar um comando que diga a este processo qual executável será disparado para a criação dos processos MPI. Para facilidade

¹Para todos os efeitos, dizer que um processo está em modo *daemon* é o mesmo que dizer que sua execução é feita em segundo plano pelo Sistema Operacional

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int
5  main(int argc, char *argv[])
6  {
7      int process_rank;    /* Rank of the process.          */
8      int p;               /* Number of processes.      */
9
10     MPI_Init(&argc, &argv);
11     MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
12     MPI_Comm_size(MPI_COMM_WORLD, &p);
13
14     if ( process_rank == 0 ) {
15         printf("Greetings from process %d, the root!\n",
16               process_rank);
17     }
18     else if ( process_rank == p-1 ) {
19         printf("Greetings from process %d, the greatest rank!\n",
20               process_rank);
21     }
22     else {
23         printf("Greetings from process %d!\n", process_rank);
24     }
25     MPI_Finalize();
26     return 0;
27 }
```

Figura 3.5: Programa “*Hello World*” sem troca de mensagens.

do usuário, as distribuições conhecidas também utilizam-se de um *script* padrão para fazê-lo, o `mpiexec`². A utilização do `mpiexec`, com o exemplo mostrado, é

```
mpiexec -n 4 HelloWorld
```

através do parâmetro `-n`, informa-se a quantidade de processos MPI que serão disparados. Neste caso, são 4 processos.

MPI implementa um padrão de réplicas; todos os processos MPI rodam sobre exatamente o mesmo código e, por consequência, todos os processadores também. Se é necessário que algum processo específico tome ações diferentes dos demais, pode-se testar o *rank* do processo, conforme demonstrado no exemplo. Em geral, a diferenciação mais recorrente é o processo zero ser o processo mestre em um algoritmo do tipo mestre-escravo (DONGARRA et al., 2003).

A execução desse programa deve transcorrer sem complicações, produzindo o resultado esperado. Um exemplo de resultado de uma execução é visto na Figura 3.6. Outro resultado de execução possível é mostrado na Figura 3.7.

```
Greetings from process 0, the root!
Greetings from process 1!
Greetings from process 2!
Greetings from process 3, the greatest rank!
```

Figura 3.6: Possibilidade de saída - primeiro exemplo.

```
Greetings from process 1!
Greetings from process 2!
Greetings from process 3, the greatest rank!
Greetings from process 0, the root!
```

Figura 3.7: Outra possibilidade de saída - primeiro exemplo.

A impressão da mensagem em console possui 1 ponteiro para a saída padrão, fornecido pela linguagem C e o Sistema Operacional. Logo, ocorre uma condição de corrida (TOSCANI; OLIVEIRA; SILVA CARÍSSIMI, 2002) entre os processos MPI, e a ordem de impressão da mensagem pode variar entre uma execução e outra. Na verdade, o risco da condição de corrida que se criou é ainda maior; devido à característica não-atômica das primitivas de E/S em C é possível que as mensagens fossem escritas de forma embaralhada. Isto, no entanto, não acontece, pois estas mensagens são curtas e em pequena quantidade, tendo uma execução rápida. MPI oferece mecanismos para o controle de condições de corridas, como *barreiras* (TOSCANI; OLIVEIRA; SILVA CARÍSSIMI, 2002).

3.1.4 Troca de Mensagens

Para a troca de mensagens MPI se usa `MPI_Send` e `MPI_Recv`. Seus protótipos são apresentados na Figura 3.8.

A literatura propõe uma discussão sobre a necessidade específica de cada parâmetro (GROPP; LUSK; SKJELLUM, 1999), que foge deste escopo. Seu significado é

²Em referências mais antigas é utilizado o *script* `mpirun` (PACHECO, 1997). Embora a maioria das distribuições ainda suportem esta sintaxe, ela está, agora, obsoleta (GROPP; LUSK; SKJELLUM, 1999).

```

int MPI_Send(void* buf, int count, MPI_Datatype dt, int dest,
             int tag, MPI_Comm comm )

int MPI_Recv(void* buf, int count, MPI_Datatype dt, int source,
             int tag, MPI_Comm comm, MPI_Status* status )

```

Figura 3.8: Protótipos de MPI_Send() e MPI_Recv().

void* buf : é o *buffer* de recepção/envio. É o endereço de memória que aponta para o início da área de dados que será enviada (MPI_Send) ou recebida (MPI_Recv).

int count : é o número de variáveis de um certo tipo que será enviada/recebida. Pode ser usada para indicar que se deseja receber, por exemplo, dois inteiros, ou enviar três números em ponto flutuante, com a condição de que estes estejam contíguos na memória (*e.g.*, um vetor).

MPI_Datatype dt : é o que define o tipo das variáveis enviadas (*e.g.*, MPI_INT para inteiros). Presente por razões de portabilidade; nas arquiteturas de processadores, o número de *bytes* (ou sua ordem de leitura) que representa uma variável de um determinado tipo pode diferir. Desta maneira, o mapeamento é feito através de uma camada inferior, pertencente ao MPI, e é transparente ao usuário, que o enxerga por meio dos tipos da biblioteca. Por essa mesma razão não é recomendado enviar fluxos de *bytes* baseado nos tamanhos-padrão de cada tipo; para isso, MPI oferece o tipo MPI_BYTE.

int dest : é o identificador de processo, referente ao comunicador também passado por parâmetro, que aponta a qual processo a mensagem será enviada no MPI_Send.

int source : é o identificador de processo, referente ao comunicador, também passado por parâmetro, que aponta de qual processo a mensagem será recebida no MPI_Recv. Para se receber uma mensagem de qualquer processo, usa-se a constante MPI_ANY_SOURCE.

int tag : é uma marca identificadora para uma mensagem; serve para facilitar o controle do fluxo de mensagens entre dois ou mais processos. Pode, por exemplo, ser usada para indicar a seqüência das mensagens a serem recebidas, se esta for importante; por mais que a rede, eventualmente, entregue as mensagens fora de ordem, se houver uma seqüência de MPI_Recv que receba mensagens com *tags* seqüenciais (mandas do mesmo modo pelo MPI_Send), então a entrega respeitará essa seqüência. A constante MPI_ANY_TAG pode ser usada para se receber uma mensagem independentemente do valor da sua *tag*.

MPI_Comm comm : é o comunicador que dá semântica aos campos *dest* e *source*; guarda a informação do grupo de processos e da topologia a qual se está fazendo referência.

MPI_Status* status : é a variável que receberá uma estrutura de dados que contém informações diversas sobre a recepção da mensagem, incluindo quem a enviou e qual sua *tag*. É especialmente útil quando é necessário usar as constantes MPI_ANY_SOURCE ou MPI_ANY_TAG; embora possa-se receber a mensagem de qualquer fonte, com

qualquer marcação, é possível identificar quem enviou a mensagem (e em que contexto) para, por exemplo, enviar uma resposta. Abordagem muito usada em algoritmos do tipo *mestre-escravo* (DONGARRA et al., 2003).

A primitiva `MPI_Recv` é bloqueante e `MPI_Send` é bloqueante até que o *buffer* de envio possa ser reaproveitado. Existem outros modos de envio/recepção, mencionados anteriormente (e.g, *send()* *bloqueante*), que possuem uma correspondente primitiva em MPI.

Existe um detalhe importante para a recepção e envio corretos de mensagem, que é a questão do batimento dos parâmetros. Entre um `MPI_Send` e um `MPI_Recv` existem certos argumentos que devem estar em concordância e outros que são completamente independentes:

count, datatype : não necessariamente devem ser o mesmo; pode-se reinterpretar os dados recebidos através das mudanças destes parâmetros.

tag, comm : devem ser o mesmo em ambos ou, no `MPI_Recv`, pode ser `MPI_ANY_TAG`.

dest, source : devem concordar (`dest` de um como o `source` do outro e vice-versa) ou, no `MPI_Recv`, pode ser `MPI_ANY_SOURCE`.

buf : é completamente independente em ambas as primitivas.

3.1.5 Segundo Exemplo

Considerando o código de exemplo apresentado anteriormente, pode-se, agora, introduzir a troca de mensagens no corpo do algoritmo. O princípio, desta vez, é fazer com que todos os processos enviem uma mensagem ao processo raiz e este as imprima na tela, à medida que as for recebendo. O código resultante de aplicação de `MPI_Send` e `MPI_Recv` pode ser visto na Figura 3.9 (PACHECO, 1998). A execução deste programa deve ser semelhante à execução do programa anterior, conforme a Figura 3.10, com a diferença do processo de maior *rank* não estar identificado e o processo raiz (0) não imprimir nada, pois sua única função é receber mensagens dos outros processos e imprimi-las.

Ainda existe uma condição de corrida, presente no recebimento das mensagens. No entanto, não há mais o risco da impressão embaralhada: o processo raiz (0) imprime as mensagens seqüencialmente, na ordem em que forem chegando.

3.1.6 Programação Distribuída × Programação Paralela

A esta altura já existe ferramental suficiente para a apresentação de um exemplo importante para ilustrar as diferenças entre programação paralela e programação distribuída. A programação distribuída ocorre quando o processamento não é realizado em apenas uma unidade fundamental (UCP); a execução ocorre em vários processadores. Já a programação paralela, pressupõe que, além de existir uma distribuição da computação, estas partes são executadas *ao mesmo tempo*.

É difícil encontrar um exemplo de computação paralela *pura*. De fato, a maioria dos programas ditos paralelos apresentam trechos que, embora distribuídos, são seqüenciais. O exemplo a seguir mostra um programa MPI que ilustra a abordagem inversa, onde não há processamento paralelo.

```

1  #include <stdio.h>
2  #include <mpi.h>
3  #include <string.h>
4
5  int
6  main(int argc, char *argv[])
7  {
8      int process_rank;      /* Rank of process.          */
9      int p;                 /* Number of processes.    */
10     int source;             /* Rank of sender.          */
11     int dest;               /* Rank of receiver.        */
12     int tag = 50;           /* Tag for messages.        */
13     char message[100];      /* Storage for the message.  */
14     MPI_Status status;      /* Return status from receive. */
15
16     MPI_Init(&argc, &argv);
17     MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
18     MPI_Comm_size(MPI_COMM_WORLD, &p);
19
20     if ( process_rank != 0 ) {
21         sprintf(message, "Greetings from process %d!", process_rank);
22         dest = 0;
23         MPI_Send(message, strlen(message) + 1, MPI_CHAR, dest, tag,
24                 MPI_COMM_WORLD );
25     } else {
26         for (source = 1; source < p; source++) {
27             MPI_Recv(message, 100, MPI_CHAR, source, tag,
28                     MPI_COMM_WORLD, &status );
29             printf("%s\n", message);
30         }
31     }
32     MPI_Finalize();
33     return 0;

```

Figura 3.9: Programa “*Hello World*” com troca de mensagens.

```

Greetings from process 1!
Greetings from process 2!
Greetings from process 3!

```

Figura 3.10: Saída - segundo exemplo.

3.1.6.1 Impressão Sequencial

O exemplo apresentado aqui faz com que processos imprimam seu *rank* de maneira garantidamente sequencial, em ordem crescente. Para tanto, utiliza-se um algoritmo do tipo passagem de *token*, onde um processo só imprime seu *rank* ao receber uma mensagem do processo de identificador imediatamente anterior. A Figura 3.11 contém um diagrama do algoritmo, enquanto a Figura 3.12 contém o código C.

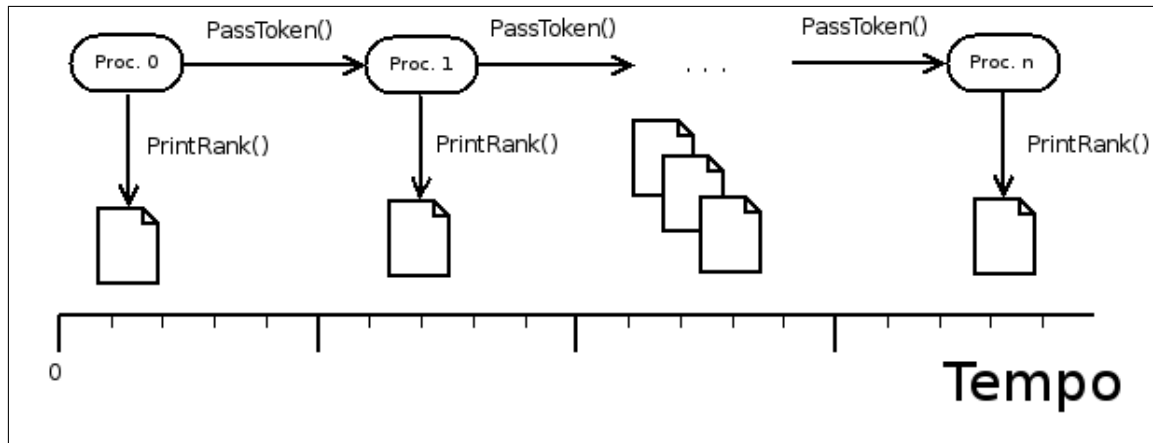


Figura 3.11: Diagrama do algoritmo de passagem de *token*.

Neste caso, *não há* programação paralela; toda a computação é distribuída, porém sequencial. O resultado obtido, para 10 processos, é visto na Figura 3.13.

3.2 Funcionalidades Adicionais

Esta seção enumera algumas funcionalidades adicionais que o MPI oferece como ferramenta ao programador paralelo. O objetivo não é exaurir a lista destas utilidades; são apresentados os principais recursos que contribuirão, nos capítulos posteriores, para a implementação do algoritmo que resolve o Problema da Mochila em paralelo.

3.2.1 Comunicação Coletiva

Além de operações `MPI_Send` e `MPI_Recv`, a especificação MPI oferece, também, funções utilitárias, que implementam estruturas de comunicação para modos recorrentes de transmitir mensagens entre processos.

Uma das principais comunicações coletivas é o *broadcast*, ou seja, mandar uma mensagem para todos os outros processos; de fato, tal tipo de envio é quase tão freqüentemente usado quanto `MPI_Send` (GROPP; LUSK; SKJELLUM, 1999).

A primeira solução que ocorre para fazer comunicação *broadcast* entre os processos seria a utilização de múltiplas chamadas `MPI_Send`, uma para cada processo, como na estrutura vista na Figura 3.14, onde todos os processos, menos o que enviou a mensagem, recebem os dados..

Existem, no entanto, dois motivos principais pelos quais esta abordagem não é a melhor:

Ineficiência. A rede de comunicação fica sobrecarregada de mensagens com o aumento do número de processos participantes. Além disso, não há uma lógica para distribuir as mensagens em uma hierarquia que aproveite topologias mais eficientes para a

```

1  #include <mpi.h>
2  #include <stdio.h>
3
4  #define TOKEN 0
5
6  int
7  main(int argc, char *argv[])
8  {
9      int my_rank;
10     int num_procs;
11     int sender = -1;
12     MPI_Status status;
13     int first_proc;
14     int last_proc;
15
16     MPI_Init(&argc, &argv);
17     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
18     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
19     first_proc = 0;
20     last_proc = num_procs - 1;
21
22     if ( my_rank != first_proc )
23         MPI_Recv(&sender, 1, MPI_INT, MPI_ANY_SOURCE, TOKEN,
24                 MPI_COMM_WORLD, &status );
25
26     printf("Rank %d", my_rank);
27     if ( sender >= 0 )
28         printf(" send by %d\n", sender);
29     else
30         printf("\n");
31     fflush(stdout);
32
33     if (my_rank != last_proc)
34         MPI_Send(&my_rank, 1, MPI_INT, my_rank + 1, TOKEN,
35                 MPI_COMM_WORLD);
36     MPI_Finalize();
37 }

```

Figura 3.12: Impressão sequencial de *rank* com algoritmo de *token* em anel.

```

Rank 0
Rank 1 send by 0
Rank 2 send by 1
Rank 3 send by 2
Rank 4 send by 3
Rank 5 send by 4
Rank 6 send by 5
Rank 7 send by 6
Rank 8 send by 7
Rank 9 send by 8

```

Figura 3.13: Resultado para a impressão sequencial de identificadores.

```

for ( i = 0; i < num_procs; ++ i ) {
    if ( i != my_rank )
        MPI_Send(message,
                  size_message,
                  MPI_INT,
                  i,
                  tag,
                  MPI_COMM_WORLD);
}

```

Figura 3.14: Implementação direta de *broadcast*.

distribuição de mensagens (*e.g.*, distribuir as mensagens numa topologia de *árvore*, onde cada processo é um nó da árvore, faz com que o tempo de distribuição seja logarítmico³, enquanto o tempo sequencial é linear⁴) (PACHECO, 1998).

Encapsulamento. Fazer uma distribuição mais eficiente promove muito trabalho ao programador (que não faz parte do algoritmo). Um detalhe importante a ser salientado é o controle feito para que o processo que enviou *não* receba a própria mensagem, fato que poderia causar *deadlocks* (TOSCANI; OLIVEIRA; SILVA CARÍSSIMI, 2002) inesperados.

Em MPI, há uma primitiva específica para se implementar *broadcast*, a `MPI_Bcast`, cujo protótipo é apresentado na Figura 3.15.

```

int MPI_Bcast(void* buf, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm )

```

Figura 3.15: Protótipo de `MPI_Broadcast()`.

O significado dos parâmetros é o mesmo que em `MPI_Send`, com a diferença de que não há um destino, mas sim a identificação do processo emissor das mensagens (*root*). Existem dois motivos para a presença de tal informação:

- Conforme mencionado anteriormente, é necessário que o processo que emite as mensagens não as envie para si mesmo; e
- Ao contrário do que se pensa inicialmente, não se recebe uma mensagem em *broadcast* através de um `MPI_Recv`, mas sim através de outro `MPI_Bcast`. Logo, é importante para o processo saber se é ele mesmo quem está mandando a mensagem ou a recebendo.

O último item citado acima causa estranheza, a princípio, mas justifica-se pelo uso eficiente da topologia; devido às otimizações topológicas que a primitiva pode fazer para o envio eficiente das mensagens, a recepção lógica das mesmas difere da recepção física. Em outras palavras, um processo pode estar recebendo a mensagem de um retransmissor,

³Mais precisamente, tenha uma complexidade média de, aproximadamente, $\log_2(\frac{p}{2})$ ciclos, onde p é o número de processos e $\frac{p}{2}$ é o número de processos que apenas recebem mensagens, sem enviá-las (nodos-folha da árvore).

⁴Complexidade de $p - 1$ envio de mensagens (ciclos), onde p é o número de processos.

ao invés da fonte. É importante, portanto, deixar a resolução física para um nível mais baixo e transparente, ao invés de se adotar a abordagem padrão do `MPI_Recv`.

Um programa, ao utilizar um algoritmo do tipo *mestre-escravo*, elege um processo como mestre. Este processo tem como função reunir as computações realizadas pelos outros processos, efetuar uma operação sobre elas e devolvê-la ao usuário. Esta também é uma operação muito freqüente em Programação Paralela.

Para simplificar este processo, MPI oferece maneiras de se unificar parcialmente o trabalho do processo raiz. É possível, através de uma primitiva, unificar o processo de receber uma mensagem de todos os processos, agrupá-las e realizar uma operação. Tal primitiva é o `MPI_Reduce`, cujo protótipo é visto na Figura 3.16.

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, int root, MPI_Comm comm )
```

Figura 3.16: Protótipo de `MPI_Reduce()`.

Da mesma maneira que `MPI_Bcast`, `MPI_Reduce` também deve ser invocada pelos processos que fizeram as computações parciais (onde terá um papel semelhante à `MPI_Send`) e pelo processo raiz (como um `MPI_Recv`). Os parâmetros da função (inéditos) são

void* sendbuf, void* recvbuf : para o processo raiz, `recvbuf` será o *buffer* de recepção da mensagem. Para os outros processos, `sendbuf` será o *buffer* de envio. Isto sugere que apenas uma variável do tipo *buffer* seria necessária, mas isso é incorreto; é possível que o próprio processo raiz calcule alguma coisa e tenha que fundir seus dados ao de todo o grupo, tendo de utilizar os dois *buffers*;

MPI_Op op : é uma constante da biblioteca que expressa qual operação deve ser feita sobre os dados (*e.g.*, `MPI_SUM`, para somar todos os dados e `MPI_MAX` para, dentre todos os dados recebidos, verificar qual é o maior) dentre as operações padrão da especificação ou de operações construídas pelo usuário, cujo suporte é oferecido pelo MPI, embora não seja detalhado aqui⁵

`MPI_Reduce` pode ser encarada como possuidora de uma lógica inversa à lógica de envio/recepção do `MPI_Bcast`; aqui, o processo raiz espera que cheguem mensagens de todos os processos, ao invés de enviá-las.

3.2.2 Terceiro Exemplo

O exemplo a seguir mostra a utilização das primitivas `MPI_Bcast` e `MPI_Reduce` através de um programa que usa o método dos retângulos (CLÁUDIO; MARINS, 1994) para calcular um valor aproximado da constante π . O código foi retirado de (GROPP; LUSK; THAKUR, 1999) e foi acrescido de algumas correções e otimizações⁶. O algoritmo MPI pode ser visto na Figura 3.17

⁵Em ambos os casos, com uma operação padrão ou personalizada, é importante que esta seja associativa, ou seja, a ordem de realização da operação sobre os subconjuntos de um conjunto de dados não influa sobre o resultado final.

⁶A destacar, a inclusão dos cabeçalhos para E/S em C e a transformação do valor de π com 25 (vinte e cinco) dígitos em uma constante.

```

1  #include <mpi.h>
2  #include <math.h>
3  #include <stdio.h>
4
5  #define PI25DT 3.141592653589793238462643
6
7  int
8  main(int argc, char *argv[]) {
9      int n;                /* Number of intervals.          */
10     int myid;              /* ID of current process.        */
11     int numprocs;          /* Number of processes           */
12     int i;                 /* "for" index.                  */
13     double mypi;           /* Part of the sum of each process */
14     double pi;             /* The final PI (sum) of all processes */
15     double h;              /* Height of each sub-rectangle.  */
16     double sum;            /* Rectangle's sum.              */
17     double x;              /* Middle value of the rectangle's base. */
18
19     MPI_Init(&argc, &argv);
20     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
21     MPI_Comm_rank(MPI_COMM_WORLD, &myId);
22     while (1) {
23         if ( myId == 0 ) {
24             printf("Enter the number of intervals: (0 quits) ");
25             scanf("%d", &n);
26         }
27         MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
28         if ( n == 0 ) {
29             break;
30         } else {
31             h = 1.0 / (double) n;
32             sum = 0.0;
33             for ( i = myid + 1; i <= n; i += numprocs ) {
34                 x = h * ((double)i - 0.5 );
35                 sum += (4.0 / (1.0 + x * x));
36             }
37             mypi = h * sum;
38             MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
39                       MPI_COMM_WORLD );
40             if ( myid == 0 )
41                 printf("PI is approximately %.16f, Error is %.16f\n",
42                         pi, fabs(pi - PI25DT) );
43         }
44     }
45     MPI_Finalize();
46     return 1;
47 }

```

Figura 3.17: Programa para calcular uma aproximação de π .

Este método baseia-se no cálculo da integral da função $4/(1+x^2)$; ela divide o eixo x em tantos intervalos quanto o usuário requisitar. Após, envia esta informação via `MPI_Bcast`, (altura, no eixo y , equivale ao valor de x constante naquele intervalo) para que os processos façam o cálculo da área do respectivo retângulo e devolvem ao processo raiz, que redistribui os intervalos pendentes. Após isso, todas as áreas são obtidas e somadas, através de `MPI_Reduce`. Para calcular o erro, é feita a comparação com um valor π suficientemente grande e previamente definido.

3.2.3 Temporização

Parte fundamental da programação paralela consiste em mensurar o ganho de desempenho obtido com o emprego de uma solução multiprocessada; até agora, foram apresentadas soluções para a paralelização básica de algoritmos. Agora, é apresentado um mecanismo simples para a medição do tempo de execução dos programas. Tal abordagem é útil para a confecção de comparativos e o estabelecimento de *benchmarks*, que adquirem grande importância em capítulos posteriores.

Visando suprir estas necessidades, a especificação MPI oferece duas primitivas para a medição de tempo, cujos protótipos são apresentados na Figura 3.18.

```
double MPI_Wtime()
double MPI_Wtick()
```

Figura 3.18: Protótipos de `MPI_Wtime()` e `MPI_Wtick()`.

`MPI_Wtime` retorna o tempo transcorrido (em segundos) desde a última chamada dela mesma ou, caso isto não tenha ocorrido, uma data padrão da implementação. De qualquer modo, duas chamadas consecutivas revelam o tempo decorrido entre estas. Para fins de uniformização da contagem de tempo, existe `MPI_Wtick`, que retorna o tempo (em segundos) entre dois pulsos de *clock* consecutivos. Ambas as primitivas podem ser usadas fora de `MPI_Init(...)` [...] `MPI_Finalize(...)`, pois muitas vezes é necessário medir a execução de um programa sequencial com o mesmo ferramental.

Para exemplificar seu uso, apresenta-se uma versão com medição de tempo do código apresentado na Figura 3.17. A Figura 3.19 contém o novo código.

3.2.4 Envio e Recebimento Não-bloqueantes

Conforme explanado anteriormente, existem vários modos no que se refere à espera de uma operação de *send* ou *receive* em MPI. Um que desperta interesse especial para fins da implementação que será desenvolvida é o envio e recebimento *não-bloqueantes*.

MPI possui `MPI_Isend` para envio e `MPI_Irecv` para recebimento não-bloqueante. Seus protótipos são visualizados na Figura 3.20.

`MPI_Isend` é idêntico à sua versão bloqueante, mas a execução do programa prossegue mesmo que o buffer não tenha sido esvaziado. `MPI_Irecv` também é muito semelhante ao já apresentado, mas, ao invés de um ponteiro para a estrutura `MPI_Status`, há um ponteiro para uma outra estrutura, `MPI_Request`. Esta estrutura é um argumento que guarda uma *requisição* após a chamada de um `MPI_Irecv` e pode ser testado a qualquer momento para verificar se a mensagem já chegou.

Existem duas primitivas usadas para testar se uma dada mensagem chegou, listadas na Figura 3.21.

```

1  #include <mpi.h>
2  #include <math.h>
3  #include <stdio.h>
4
5  #define PI25DT 3.141592653589793238462643
6
7  int
8  main(int argc, char *argv[])
9  {
10     (...)
11
12     /* Time in seconds application started. */
13     double start_time;
14     /* Time in seconds application ended. */
15     double end_time;
16
17     MPI_Init(&argc, &argv);
18     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
19     MPI_Comm_rank(MPI_COMM_WORLD, &myId);
20     while ( 1 ) {
21         if ( myid == 0 ) {
22             printf("Enter the number of intervals: (0 quits) ");
23             scanf("%d", &n);
24         }
25         /* Timing count begins. */
26         start_time = MPI_Wtime();
27         MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
28         if ( n == 0 ) {
29             break;
30         } else {
31
32             (...)
33
34             /* Timing count ends. */
35             end_time = MPI_Wtime();
36             if ( myid == 0 ) {
37                 printf("PI is approximately %.16f, Error is %.16f\n",
38                     pi,
39                     fabs(pi - PI25DT) );
40                 /* Timing and "clock tick" count. */
41                 printf("Time is %.16f seconds or %.16f clock ticks.\n",
42                     end_time - start_time,
43                     (end_time - start_time)/MPI_Wtick() );
44             }
45         }
46     }
47     MPI_Finalize();
48     return 1;
49 }

```

Figura 3.19: Programa para calcular uma aproximação de π , com medição de tempo.

```
int MPI_Isend(void* buf, int count, MPI_Datatype dt, int dest,
             int tag, MPI_Comm comm )

int MPI_Irecv(void* buf, int count, MPI_Datatype dt, int source,
             int tag, MPI_Comm comm, MPI_Request* request )
```

Figura 3.20: Protótipos de MPI_Isend() e MPI_Irecv().

```
int MPI_Wait (MPI_Request *request, MPI_Status *status)
int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)
```

Figura 3.21: Protótipos de MPI_Wait() e MPI_Test().

MPI_Wait bloqueia até que a mensagem chegue, sendo útil para adiantar alguma parte não-dependente da chegada da mensagem e, então, esperar. Quando a mensagem chegar será como um MPI_Recv, entregando os dados nas variáveis indicadas e seu *status* em *status*.

MPI_Test testa se a mensagem chegou, indicando tal fato na variável *flag*; em seguida, testando *flag* (e.g., `if ( flag == 0 )`), pode-se continuar de acordo com as preferências do programador.

A estas primitivas tem versões que atuam sobre vetores de ponteiros do tipo `MPI_Request`; estas funções têm o protótipo conforme visto na Figura 3.22.

```
int MPI_Testany(int count, MPI_Request* array_of_requests, int* index,
               MPI_Status* status)
int MPI_Waitany(int count, MPI_Request* array_of_requests, int* index,
               MPI_Status* status)
```

Figura 3.22: Protótipos de MPI_Testany() e MPI_Waitany().

index contém a posição do vetor que guarda o *request* da solicitação que foi atendida. Uma aplicação disto será vista no próximo exemplo a ser mostrado.

Visto que as estruturas MPI tem um *buffer*, pode ser útil cancelar um MPI_Irecv, que fica aguardando as mensagens; faz-se isso através de MPI_Cancel, descrito na Figura 3.23, que cancela uma requisição de aguardo de mensagem apontada por *request*.

3.2.5 Quarto Exemplo

Este exemplo ilustra o uso de primitivas não-bloqueantes. MPI fornece maneiras de se receber uma mensagem com 1 determinado indicador (uma *tag*) ou de qualquer indicador (através da constante `MPI_ANY_TAG`). Não há, entretanto, um recurso para se receber mensagens de um conjunto de *tags*. O código mostrado usa primitivas não-bloqueantes para receber uma mensagem que pode ter exatamente 2 tipos de *tag* e pode ser visto na Figura 3.24. O exemplo mostrado é claramente extensível para tantas *tags* quanto se queira.

No exemplo apresentado, algum processamento é feito enquanto se espera a mensagem, geralmente para evitar uma condição de *deadlock*. No entanto, ao se remover este processamento, tem-se o equivalente a um MPI_Recv, bloqueante, que aceita duas *tags* no cabeçalho da mensagem.

```
int MPI_Cancel(MPI_Request *request)
```

Figura 3.23: Protótipo de MPI_Cancel().

```
1 MPI_Irecv(&msg, 1, MPI_INT, target_proc, TAG_ONE, MPI_COMM_WORLD,
2   &drequest[0]);
3 MPI_Irecv(&msg, 1, MPI_INT, target_proc, TAG_TWO, MPI_COMM_WORLD,
4   &drequest[1]);
5 flag = 0;
6 index = -1;
7 do {
8   MPI_Testany(2, drequest, &index, &flag, &status);
9   /* Some proc. */
10 } while ( flag == 0 );
11 MPI_Cancel(&(drequest[2-index]));
```

Figura 3.24: Recebimento de mensagem que pode ter exatas 2 tags.

3.3 Multiplicação Matriz $\times$ Vetor

Um estudo de caso a ser abordado é o algoritmo que realiza a multiplicação de uma matriz por um vetor em paralelo. Lembrando, esta implementação é sobre MPI-1.2 e é apresentada em (GROPP; LUSK; SKJELLUM, 1999), na linguagem *FORTRAN*. Para fins desta monografia tal algoritmo foi convertido para a linguagem *C*.

Seja A uma matriz quadrada de ordem n e x um vetor de comprimento n tal que $A \times x = b$, sendo b o vetor resultante da multiplicação, de tamanho n . O algoritmo é do tipo mestre-escravo⁷ e decompõe A em linhas, que distribui ciclicamente a medida que os escravos requisitam trabalho. Cada escravo, tendo recebido do mestre x (por *broadcast*), faz a multiplicação do elemento de b correspondente.

Cada processo escravo, ao acabar a computação, requisita mais trabalho para o mestre, que pode enviá-lo ou dizer ao escravo para terminar, pois não há mais tarefas. Ao final, o processo mestre recolhe cada resultado e monta o vetor-reposta. O código está todo contido em um arquivo mas, para melhor entendimento, subdividiu-se em três partes: início (Figura 3.25), mestre (Figura 3.26) e escravo (Figura 3.27).

Esta computação, para diferentes tamanhos de matriz e número de processos, teve seu tempo de execução medido. As execuções são feitas para configurações utilizando de 2 a 10 nós. Para cada configuração, executou-se o mesmo programa com matrizes quadradas de ordem 100 até 1000, incrementados de 100. O mesmo procedimento foi repetido 5 vezes e, ao final, tomou-se a média aritmética do tempo de execução. A Figura 3.28 apresenta os resultados obtidos. Estes resultados foram publicados, em conjunto com medições sobre MPI-2 (com criação dinâmica dos processos) na Escola Regional de Alto Desempenho 2007 (ESCOLA REGIONAL DE ALTO DESEMPENHO, 2007).

3.3.1 Multiplicação Matriz $\times$ Matriz

É interessante reparar que, com poucas extensões, é possível aproveitar o código apresentado e alterá-lo para produzir um programa que compute, paralelamente, a multiplicação de duas matrizes quadradas ($A \times B = C$, com A , B e C de tamanho n).

O código estendido estava, originalmente, em *FORTRAN* (GROPP; LUSK; SKJEL-

⁷Um processo distribui tarefas a outros processos e colhe resultados (PACHECO, 1997).

```

1  #include <mpi.h>
2  #include <stdio.h>
3  #include <unistd.h> /* GNU GetOpt library. */
4
5  #define MAX_ROWS 1001
6  #define MAX_COLS 1001
7
8  int
9  main(int argc, char *argv[])
10 {
11     int rows;
12     int cols;
13     double matrix[MAX_ROWS][MAX_COLS]; /* Matrix to be multiplied. */
14     double vectorA[MAX_COLS];
15     double vectorB[MAX_ROWS];
16     double buffer[MAX_COLS];
17     double answer;
18     int myid;          /* ID of current process. */
19     int master;        /* Master process ID. */
20     int numprocs;      /* Number of processes. */
21     int i, j;          /* Index for looping. */
22     int numsent;
23     int sender;
24     int anstype;
25     int row;
26     MPI_Status status;
27
28     /* Benchmark variables. */
29     double start_time;
30
31     (... GET PARAM ...)
32
33     MPI_Init(&argc, &argv);
34     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
35     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

```

Figura 3.25: Código para o programa de multiplicação matriz $\times$ vetor (início).

```

1  master = 0;      /* Process 0 is the master process. */
2
3  if ( myid == master ) {
4      /* Master's part: */
5      /* Initializing vector and matrix. */
6      for (i = 0; i < cols; i = i + 1) {
7          vectorA[i] = 1;
8          for (j = 0; j < rows; j = j + 1)
9              matrix[j][i] = j;
10     }
11
12     start_time = MPI_Wtime();
13
14     /* Sends vectorA to each slave process. */
15     MPI_Bcast(vectorA, cols, MPI_DOUBLE, master, MPI_COMM_WORLD);
16
17     /* Sends a row for each slave process; tag with row number. */
18     numsent = 0;
19     for ( i = 0; i < numprocs-1; i = i + 1 ) {
20         for (j = 0; j < cols; j = j + 1)
21             buffer[j] = matrix[i][j];
22         MPI_Send(buffer, cols, MPI_DOUBLE, i + 1, i + 1,
23                 MPI_COMM_WORLD);
24         numsent += 1;
25     }
26     for ( i = 0; i < rows; i = i + 1 ) {
27         MPI_Recv(&answer, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
28                 MPI_ANY_TAG, MPI_COMM_WORLD, &status);
29         sender = status.MPI_SOURCE;
30         anstype = status.MPI_TAG;
31         vectorB[ansype-1] = answer;
32         if ( numsent < rows ) {
33             for ( j = 0; j < cols; j = j + 1 ) {
34                 buffer[j] = matrix[numsent][j];
35             }
36             MPI_Send(buffer, cols, MPI_DOUBLE, sender, numsent+1,
37                     MPI_COMM_WORLD);
38             numsent += 1;
39         }
40         else
41             MPI_Send(MPI_BOTTOM, 0, MPI_DOUBLE, sender, 0,
42                     MPI_COMM_WORLD);
43     }
44
45     printf("%d %d %f \n", rows, numprocs - 1,
46           MPI_Wtime() - start_time);
47     /* Prints solution */ (...)
48 }

```

Figura 3.26: Código para o programa de multiplicação matriz $\times$ vetor (mestre).

```

1  } else {
2      /* Slave's part: */
3      MPI_Bcast(vectorA, cols, MPI_DOUBLE, master, MPI_COMM_WORLD);
4      if ( myid > rows ) {
5          MPI_Finalize();
6          return 0;
7      }
8      while (1) {
9          MPI_Recv(buffer, cols, MPI_DOUBLE, master, MPI_ANY_TAG,
10                 MPI_COMM_WORLD, &status);
11          if ( status.MPI_TAG == 0 ) {
12              MPI_Finalize();
13              return 0;
14          } else {
15              row = status.MPI_TAG;
16              answer = 0.0;
17              for (i = 0; i < cols; i = i + 1)
18                  answer += buffer[i] * vectorA[i];
19              MPI_Send(&answer, 1, MPI_DOUBLE, master, row,
20                     MPI_COMM_WORLD);
21          }
22      }
23      MPI_Finalize();
24      return 0;
25  }

```

Figura 3.27: Código para o programa de multiplicação matriz $\times$ vetor (escravo).

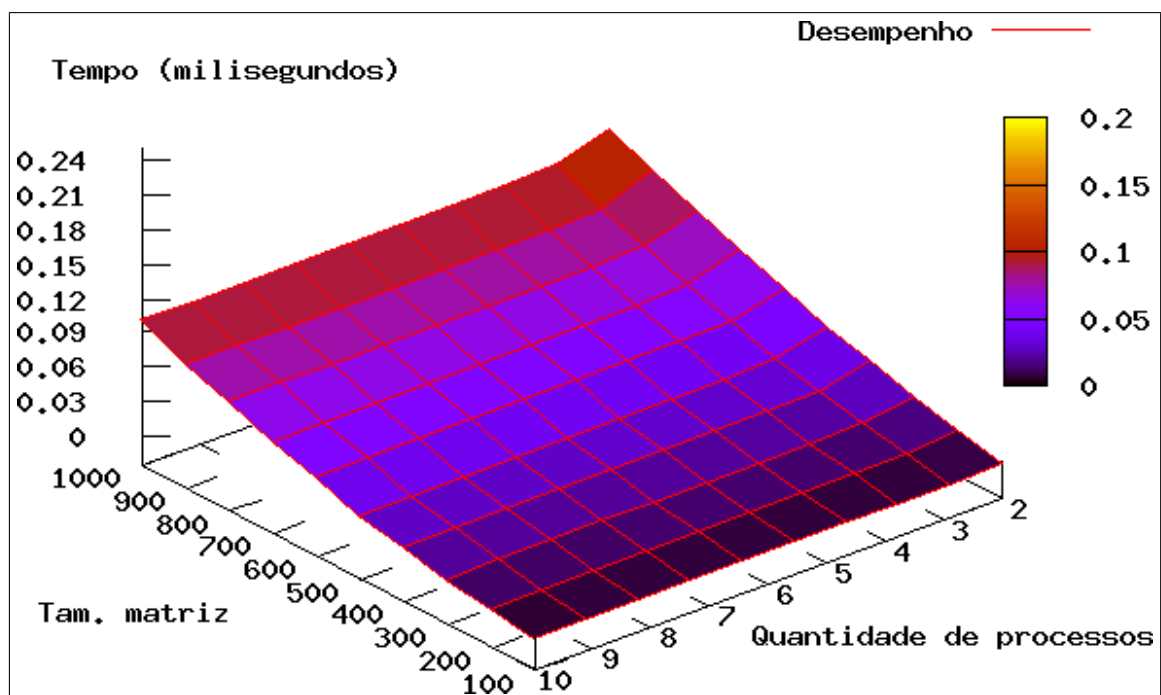


Figura 3.28: Desempenho: multiplicação matriz $\times$ vetor – MPI-1.2

LUM, 1999) e foi portado para C. O código, como o anterior, também é apresentado em três partes: início (Figura 3.29), mestre (Figura 3.30) e escravo (Figura 3.31)

As medições de tempo, tomadas variando-se apenas o número de processos para uma matriz 1000×1000 , foram realizadas 5 vezes e tomada a média aritmética; o número de processos listados desconsidera o processo mestre (*e.g.*, 0 significa o algoritmo seqüencial, rodado em separado da estrutura MPI), resultados apresentados na Tabela 3.2.

N. Proc.	Tempo (s)	N. Proc.	Tempo (s)
0	0.026577	8	0.084667
1	0.030405	9	0.110110
2	0.069374	10	0.093460
3	0.049342	11	0.103958
4	0.054366	12	0.160807
5	0.057348	13	0.141569
6	0.076734	14	0.394071
7	0.073961	15	0.258599

Tabela 3.2: Tempos da multiplicação matriz $\times$ matriz paralela.

É interessante notar que, contrariando a lógica, os tempos apresentados sobem com o aumento do número de processos. Pode-se justificar isso através de três argumentos,

1. Com o aumento do número de processos, o número de troca de mensagens (operações de E/S, custosas em termos de tempo) sobe;
2. O fato da multiplicação envolver duas matrizes ao invés de um vetor e uma matriz gera um aumento considerável no envio de mensagens (E/S); e
3. As matrizes não são grandes o suficiente para se beneficiar do aumento de processos e sobrepujar o custo de comunicação.

```

1  #include <mpi.h>
2  #include <stdio.h>
3  #include <unistd.h> /* GNU GetOpt library. */
4
5  #define MAX_A_ROWS 100
6  #define MAX_A_COLS 100
7  #define MAX_B_COLS 100
8
9  int
10 main(int argc, char *argv[])
11 {
12     /* Algorithm variables. */
13     int a_rows;
14     int b_rows;
15     int c_rows;
16     int a_cols;
17     int b_cols;
18     int c_cols;
19     double matrixA[MAX_A_ROWS][MAX_A_COLS];
20     double matrixB[MAX_A_COLS][MAX_B_COLS];
21     double matrixC[MAX_A_ROWS][MAX_B_COLS];
22     double buffer[MAX_A_COLS]; /* Buffer for temp. multiplication. */
23     double answer[MAX_B_COLS];
24     int myid; /* ID of current process. */
25     int master; /* Master process ID. */
26     int numprocs; /* Number of processes. */
27     int i; /* Index for looping. */
28     int j; /* Index for looping. */
29     int numsent; /* Number of sent processes. */
30     int sender; /* The sender identifier. */
31     int anstype; /* Where the answer vector should be. */
32     int row; /* Holds the row number. */
33     MPI_Status status;
34     double start_time;
35
36     (... GET PARAM ...)
37
38     /* Matrixes initialized sizes (indexes limits). */
39     master = 0; /* Process 0 is the master process. */
40     a_rows = MAX_A_ROWS;
41     a_cols = MAX_A_COLS;
42     b_rows = MAX_A_COLS;
43     b_cols = MAX_B_COLS;
44     c_rows = a_rows;
45     c_cols = b_cols;
46
47     /* Initializing MPI. */
48     MPI_Init(&argc, &argv);
49     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
50     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

```

Figura 3.29: Código para o programa de multiplicação matriz $\times$ matriz (início).

```

1  if ( myid == master ) {
2      /** Master's part: */
3
4      /* Initializing matrixes. */
5      (...)
6
7      start_time = MPI_Wtime();
8
9      /* Sends MatrixB to each slave process. */
10     for ( i = 0; i < b_rows; i = i + 1 )
11         MPI_Bcast(&(matrixB[i][0]), b_cols, MPI_DOUBLE, master,
12                 MPI_COMM_WORLD);
13
14     /* Sends a row for each slave process; tag with row number. */
15     numsent = 0;
16     for ( i = 0; i < min(numprocs-1, a_rows); i = i + 1 ) {
17         for ( j = 0; j < a_cols; j = j + 1 )
18             buffer[j] = matrixA[i][j];
19         MPI_Send(buffer, a_cols, MPI_DOUBLE, i + 1, i + 1,
20                 MPI_COMM_WORLD);
21         numsent = numsent + 1;
22     }
23
24     for ( i = 0; i < c_rows; i = i + 1 ) {
25         MPI_Recv(&answer, c_cols, MPI_DOUBLE, MPI_ANY_SOURCE,
26                 MPI_ANY_TAG, MPI_COMM_WORLD, &status);
27         sender = status.MPI_SOURCE;
28         anstype = status.MPI_TAG;
29         for ( j = 0; j < c_cols; j = j + 1 )
30             matrixC[anstype-1][j] = answer[j];
31         if ( numsent < a_rows ) {
32             for ( j = 0; j < a_cols; j = j + 1 )
33                 buffer[j] = matrixA[numsent][j];
34             MPI_Send(buffer, a_cols, MPI_DOUBLE, sender, numsent +
35                     1, MPI_COMM_WORLD);
36             numsent = numsent + 1;
37         } else {
38             MPI_Send(MPI_BOTTOM, 0, MPI_DOUBLE, sender, 0,
39                     MPI_COMM_WORLD);
40         }
41     }
42
43     /* Prints the data into the GNUPLOT format. */
44     printf("%d %f \n", numprocs - 1, MPI_Wtime() - start_time);

```

Figura 3.30: Código para o programa de multiplicação matriz $\times$ matriz (mestre).

```

1      } else {
2          /** Slave's part: */
3          for ( i = 0; i < b_rows; i = i + 1 )
4              MPI_Bcast(&(matrixB[i][0]), b_cols, MPI_DOUBLE, master,
5                      MPI_COMM_WORLD);
6          if ( myid > b_rows ) {
7              MPI_Finalize();
8              return 0;
9          }
10         while ( 1 ) {
11             MPI_Recv(buffer, a_cols, MPI_DOUBLE, master, MPI_ANY_TAG,
12                     MPI_COMM_WORLD, &status);
13             if ( status.MPI_TAG == 0 ) {
14                 MPI_Finalize();
15                 return 0;
16             } else {
17                 row = status.MPI_TAG;
18                 for ( i = 0; i < b_cols; i = i + 1 ) {
19                     answer[i] = 0.0;
20                     for ( j = 0; j < a_cols; j = j + 1 )
21                         answer[i] = answer[i]
22                             + buffer[j] * matrixB[j][i];
23                 }
24                 MPI_Send(&answer, b_cols, MPI_DOUBLE, master, row,
25                         MPI_COMM_WORLD);
26             }
27         }
28     }
29     MPI_Finalize();
30     return 0;
31 }

```

Figura 3.31: Código para o programa de multiplicação matriz $\times$ matriz (escravo).

4 PROBLEMA DA MOCHILA

O Problema da Mochila é um problema de otimização combinatória de especial importância na *Ciência da Computação*, pois pertence à classe de problemas *NP-Completo*s (GAREY; JOHNSON, 1979). É, também, um dos principais problemas da área de *Pesquisa Operacional*, visto que muitas situações onde se busca maximizar o lucro podem ser reduzidas a resolver este problema. Sua resolução demanda um grande esforço computacional, tornando-se especialmente interessante para a área de *Programação Paralela*. O seguinte capítulo apresenta o Problema da Mochila e mostra um algoritmo paralelo que o resolve, implementado em MPI.

4.1 Definição Intuitiva

O Problema da Mochila pode ser definido intuitivamente através de um cenário. Suponha-se um ladrão que almeja roubar uma fábrica de peças de informática. Tudo de que o ladrão dispõe para carregar o produto de seu furto é uma mochila escolar, de porte médio. Existem objetos das mais variadas dimensões, com os mais variados valores (*e.g.*, placas-mãe, memória de vídeo, memórias RAM, *etc.*). Caberá ao ladrão, portanto, escolher os itens a carregar na sua mochila de modo que o valor a ser roubado seja máximo e que ele possa carregar.

O Problema da Mochila, portanto, pode ser visto como o problema de preencher uma mochila de capacidade limitada com itens que possuem um valor associado de modo a não exceder esta capacidade e maximizar o valor resultante.

4.2 Definição Formal

A definição formal do Problema da Mochila é (KELLER; PFERSCHY; PISINGER, 2005):

Definição (Problema da Mochila *Ilimitado*). *Sejam n tipos de itens ($n \in \mathbb{N}$). Cada $x_i \in \mathbb{N}$ representa a quantidade de itens do tipo i , onde cada tipo tem associado um valor $v_i \in \mathbb{R}_+$ e um peso $p_i \in \mathbb{R}_+$ ($i \in \{1, \dots, n\}$). O máximo peso que a mochila pode carregar é $C \in \mathbb{R}_+$. O Problema da Mochila *Ilimitado* é, então,*

$$\text{maximizar} \quad \sum_{i=1}^n x_i \times v_i \quad | \quad \sum_{i=1}^n x_i \times p_i \leq C$$

Esta definição chama-se “Problema da Mochila *Ilimitado*” pois não há número de itens de cada tipo (o valor de x_i) máximo. Este é o problema cuja implementação futuramente apresentada resolve.

4.3 Outras Definições

Existem outros tipos de Problema da Mochila, com muitas variações nas restrições e condições de execução (*e.g.*, Problema da Mochila *Compartimentada*, Problema da Mochila *Multidimensional*, *etc.*). É interessante analisar, entretanto, duas variações do Problema da Mochila que podem ser obtidas adicionando, cada uma, uma pequena restrição no Problema da Mochila original. Estas variações são especialmente interessantes porque podem ser facilmente adicionadas ao programa que resolve tal problema, por meio da modificação de uma única função (*branch*).

4.3.1 Problema da Mochila *Limitado*

Retomando o exemplo do ladrão, este seria o caso do assaltante estar furtando uma residência, ao invés de uma loja de computadores. Na residência, o número de itens que pode ser levado do mesmo tipo tende a ser limitado¹; é improvável, por exemplo, que existam mais do que três ou quatro televisores. Assim cada item pode ser incluído no somatório um número limitado de vezes. Formalmente (KELLER; PFERSCHY; PISINGER, 2005),

Definição (Problema da Mochila *Limitado*). *Sejam n tipos de itens ($n \in \mathbb{N}$). Cada $x_i \in \{0, \dots, b_i\}$ representa a quantidade de itens do tipo i , onde cada tipo tem associado um número de itens disponíveis máximo $b_i \in \mathbb{N}$, um valor $v_i \in \mathbb{R}_+$ e um peso $p_i \in \mathbb{R}_+$ ($i \in \{1, \dots, n\}$). O máximo peso que a mochila pode carregar é $C \in \mathbb{R}_+$. O Problema da Mochila *Limitado* é, então,*

$$\begin{aligned} \text{maximizar} \quad & \sum_{i=1}^n x_i \times v_i \quad | \quad \sum_{i=1}^n x_i \times p_i \leq C \\ \text{onde} \quad & 0 \leq x_i \leq b_i \end{aligned}$$

4.3.2 Problema da Mochila *0-1*

No mesmo cenário dos anteriores, o ladrão, desta vez, atenta em roubar uma loja de raros vasos chineses. Tal loja tem apenas um tipo de cada vaso e, deste modo, a decisão do ladrão é entre levar o item ou não. Formalmente (KELLER; PFERSCHY; PISINGER, 2005),

Definição (Problema da Mochila *0-1*). *Sejam n tipos de itens ($n \in \mathbb{N}$). Cada $x_i \in \{0, 1\}$ representa a quantidade de itens do tipo i , um valor $v_i \in \mathbb{R}_+$ e um peso $p_i \in \mathbb{R}_+$ ($i \in \{1, \dots, n\}$). O máximo peso que a mochila pode carregar é $C \in \mathbb{R}_+$. O Problema da Mochila *0-1* é, então,*

$$\begin{aligned} \text{maximizar} \quad & \sum_{i=1}^n x_i \times v_i \quad | \quad \sum_{i=1}^n x_i \times p_i \leq C \\ \text{onde} \quad & x_i = 0 \quad \text{ou} \quad x_i = 1 \end{aligned}$$

¹ *A priori*, o número de itens da fábrica também é limitado. No entanto, pode-se considerar que o ladrão tem tempo disponível para produzir quantas peças de um mesmo tipo quiser.

4.4 Complexidade

O Problema da Mochila é um problema *NP-Completo*. Portanto, o PM é resolvível em tempo polinomial por um algoritmo *não-determinístico*. De fato, não há prova de que existe algoritmo que resolva o Problema da Mochila em tempo polinomial em uma máquina determinística (pertence à classe *NP*) e qualquer outro problema *NP* pode ser reduzido polinomialmente a um Problema da Mochila, sendo, assim, o problema *NP-Completo* (GAREY; JOHNSON, 1979). A complexidade da solução adotada será apresentada junto com o código do algoritmo.

Devido à sua complexidade exponencial, uma solução eficiente para o problema deve ser aquela que consegue reduzir, ao máximo, os termos da expressão exponencial. O fato do Problema da Mochila ser *NP-Completo* permite que os resultados obtidos nesta monografia possam ser generalizados para toda a classe de problemas *NP-Completo*s. Em outras palavras, é provável que o emprego de uma técnica de *Workstealing* produza impacto semelhante quando aplicada à soluções paralelas de outros algoritmos que resolvem problemas *NP-Completo*s.

4.5 Solução

O algoritmo adotado para a solução do problema é do tipo *Branch & Bound*. Algoritmos desta natureza concentram-se em gerar ramos da árvore de possibilidades de todas as soluções possíveis (*branch*) e cortam os ramos que não são promissores, ou seja, garantidamente não vão gerar uma solução ótima (*bound*). Conforme explicitado, a enumeração de todas as soluções é feita em tempo exponencial, sendo de suma importância a otimização do processo.

4.5.1 Algoritmo

Toma-se a Definição 4.2, “Problema da Mochila *Ilimitado*”. A solução apresentada (KELLER; PFERSCHY; PISINGER, 2005) considera que a entrada está disposta de maneira decrescente em relação aos valores relativos dos objetos. Como valor relativo, entende-se a razão entre o valor de um objeto do tipo i (v_i) e o peso do mesmo objeto (p_i). Ou seja, há a relação

$$\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \dots \geq \frac{v_n}{p_n}$$

Tal ordenamento, se desejado, pode ser feito com uma complexidade $O(n \times \log_2(n))$ (e.g., *Quicksort*). Para a implementação deste algoritmo construiu-se um gerador de casos em *Python*, que gera arquivos de instâncias do Problema da Mochila com o formato apresentado na Figura 4.1. Esta medida garante a ordem decrescente dos valores relativos, retirando este custo da implementação.

```
<número de elementos (n)> <capacidade da mochila (C)>
v1 v2 v3 v4 ... vn
p1 p2 p3 p4 ... pn
```

Figura 4.1: Formato de instância do problema da mochila.

A construção do algoritmo *Branch & Bound* que resolve o Problema da Mochila será abordado em separado, para um melhor entendimento, já que as partes de *branch* e de

bound possuem certa independência.

4.5.1.1 Branch

Esta parte consiste em gerar os galhos da árvore de soluções (cada galho corresponde a um conjunto $\{x_1, \dots, x_n\}$ candidato à solução ótima do problema), um a um. Para fazê-lo, usa-se uma técnica de geração baseada em um algoritmo guloso.

Conforme foi estabelecido, os itens da entrada estão ordenados de acordo com seu valor relativo. Logo, o primeiro item é o melhor em termos de custo/benefício. O algoritmo guloso atribui a este item (x_1) o maior valor possível tal que o peso deste não ultrapasse o limite da mochila, ou seja, tal que $x_1 \times p_1 \leq C$. Sobrando espaço na mochila ($C - x_1 \times p_1 > 0$), tenta-se preenchê-lo elevando-se ao máximo o valor de x_2 respeitando que $(x_1 \times p_1) + (x_2 \times p_2) \leq C$. Sobrando espaço, ainda, na mochila, parte-se para x_3 e assim sucessivamente, até x_n . Com isso, gera-se o primeiro galho da árvore de soluções possíveis. Exemplo no Algoritmo 1.

Algoritmo 1 Branch (um galho)

```

1:  $x_1 \leftarrow \lfloor C/p_1 \rfloor$ 
2: for  $j \leftarrow 2, n$  do
3:    $x_j \leftarrow \lfloor (C - \sum_{i=1}^j (p_i \times x_i)) / p_j \rfloor$ 
4: end for
```

Fica evidente, assim, que é possível distribuir os itens a caberem na mochila simplesmente alterando o valor inicial de j ; pode-se gerar um novo galho tanto da raiz (caso apresentado), quanto de qualquer nodo intermediário. Isso implica que, para gerar a árvore toda, basta utilizar o algoritmo acima $n - 1$ vezes, onde, a cada execução o valor inicial de j , inicializado com n , é decrescido de uma unidade.

O processo de gerar toda a árvore imediatamente remete a uma implementação recursiva. De fato, uma função recursiva é muito mais fácil de ser implementada e muito mais legível. No entanto, para fins desta monografia, torna-se especialmente interessante a abordagem iterativa, onde simula-se a pilha da recursividade. Os motivos para isso ficarão claros quando, ao implementar a solução baseada em MPI, as tarefas tiverem de ser divididas, inviabilizando o uso da recursividade oferecida pelo Sistema Operacional. Por hora, é apresentada a função iterativa. É importante notar que esta versão necessita que se inicialize o primeiro galho conforme descrito acima antes de gerar todos os outros, que usam este como base. Vê-se o resultado no Algoritmo 2.

4.5.1.2 Bound

Este procedimento faz com que galhos que comprovadamente não possam gerar o valor ótimo não sejam nem expandidos. Basicamente, faz-se isso através da comparação do máximo valor que ainda pode ser atingido com uma ramificação da árvore com o maior valor corrente. Uma vez que o ponteiro k aponta para determinado nodo da árvore, acaba particionando-a; todos os valores anteriores já foram estabelecidos, e os que estão à frente ainda precisam ser expandidos iterativamente. Desse modo, tem-se

$$\{x_1, x_2, \dots, x_k, \overrightarrow{x_{k+1}, \dots, x_n}\}$$

Toma-se V e P como $\sum_{i=1}^n (x_i \times v_i)$ e $\sum_{i=1}^n (x_i \times p_i)$, respectivamente. Além disso, sejam V' e P' o valor e o peso, respectivamente, da partição $\{x_1, \dots, x_k\}$. Dessa partição, o espaço restante na mochila $C - P'$ (denotado, doravante, P'') deve ser preenchido com

Algoritmo 2 *Branch* (toda a árvore).

```

1:  $x_1 \leftarrow \lfloor C/p_1 \rfloor$ 
2: for  $j \leftarrow 2, n$  do
3:    $x_j \leftarrow \lfloor (C - \sum_{i=1}^j (p_i \times x_i)) / p_j \rfloor$ 
4: end for
5:  $k \leftarrow n$ 
6: while true do
7:    $k \leftarrow n$ 
8:   while  $(k \geq 1) \wedge (x_k = 0)$  do
9:      $k \leftarrow k - 1$ 
10:  end while
11:  if then  $(k \geq 1)$ 
12:    exit
13:  end if
14:  if  $k = n$  then
15:     $x_k \leftarrow 0$ 
16:    continue
17:  end if
18:   $x_k \leftarrow x_k - 1$ 
19:  for  $j \leftarrow k + 1, n$  do
20:     $x_j \leftarrow \lfloor (C - \sum_{i=1}^j (p_i \times x_i)) / p_j \rfloor$ 
21:  end for
22:   $V \leftarrow \sum_{i=1}^n (v_i \times x_i)$ 
23:  if  $V > V_{best}$  then
24:     $x_{best} \leftarrow x$ 
25:     $V_{best} \leftarrow V$ 
26:  end if
27:   $k \leftarrow n$ 
28: end while

```

a distribuição $\{x_{k+1}, \dots, x_n\}$ que maximize $V - V'$. Note-se, também, que o maior valor relativo dos itens que restam é $\frac{v_{k+1}}{s_{k+1}}$, dado o ordenamento inicial dos valores relativos.

Para fazer a poda, basta supor o melhor caso possível para a partição não analisada e verificar se esta situação supera ou não a melhor configuração corrente (x_{best}), que tem o valor V_{best} . No caso, a melhor situação possível seria se todos os itens restantes possuíssem o valor relativo $\frac{v_{k+1}}{s_{k+1}}$. Basta, então, calcular a soma como se possuíssem este valor e comparar V_{best} com este; caso não haja superação do primeiro pelo segundo, o algoritmo prossegue sem abrir aquele ramo da árvore. Matematicamente, o Algoritmo 3 reflete estas considerações.

Algoritmo 3 Bound

```

1:  $V' = \sum_{i=1}^k (v_i \times x_i)$ 
2:  $P'' = P - \sum_{i=1}^k (p_i \times x_i)$ 
3: if  $V' + (v_{k+1}/p_{k+1} \times P'') \leq V_{best}$  then
4:   continue
5: else
6:   (branch)
7: end if
```

Para uma versão completa do algoritmo, basta adicionar o trecho de *bound* ao código já apresentado até aqui (MARTELLO; TOTH, 1990), o que pode ser visualizado no Algoritmo 4.

O código apresentado resolve o Problema da Mochila. No entanto, uma abordagem formal é por demais abstrata; é bom que se use uma abordagem mais prática para ilustrar o algoritmo. A Figura 4.2 mostra a mesmo algoritmo implementado na linguagem *Python*. Com ela, há fácil compreensão das transcrições feitas, ao mesmo tempo em que a análise fica mais limpa e clara. O módulo `modknapsack` contém apenas uma função (`readInstance(...)`) que lê a entrada mostrada na Figura 4.1.

A demonstração formal da complexidade desta solução foge ao escopo desta monografia. No entanto, emxerga-se que:

1. O cálculo do valor total (V) associado a um conjunto $\{x_1, \dots, x_n\}$ e a verificação da obtenção de um novo maior (linhas 28-30) é realizada em tempo polinomial $O(n)$;
2. A geração de um conjunto $\{x_1, \dots, x_n\}$ candidato à solução (linhas 24-26) é feito em tempo polinomial $O(n)$, mas depende da geração de todos os valores de k .
3. A geração de todos os conjuntos $\{x_1, \dots, x_n\}$ possíveis é exponencial; o número total de conjuntos que existem, pelo Princípio Fundamental da Contagem, é (considera-se $\#x_i$ como o número de valores que x_i pode assumir)

$$\#x_1 \times \#x_2 \times \#x_3 \times \dots \times \#x_n = \prod_{i=1}^n \#x_i$$

logo, como todos os conjuntos possíveis são gerados, este é o mínimo (exponencial) de operações a serem realizadas.

De fato, a complexidade exata do Problema da Mochila é difícil de ser estimada, mas podem-se traçar conjecturas em cima da complexidade pessimista e do tipo de Problema da Mochila analisado:

Algoritmo 4 Solução *Branch & Bound* para o Problema da Mochila

```

1: procedure KNAPSACKBRANCHBOUND( $C, n, \{v_1, \dots, v_n\}, \{p_1, \dots, p_n\}$ )
2:    $x_1 \leftarrow \lfloor C/p_1 \rfloor$ 
3:   for  $j \leftarrow 2, n$  do
4:      $x_j \leftarrow \lfloor (C - \sum_{i=1}^j (p_i \times x_i)) / p_j \rfloor$ 
5:   end for
6:    $V_{best} \leftarrow \sum_{i=1}^n (v_i \times x_i)$ 
7:    $x_{best} \leftarrow x$ 
8:   while true do ▷ O laço será quebrado internamente, via break.
9:      $k \leftarrow n$ 
10:    while  $(k \geq 1) \wedge (x_k = 0)$  do
11:       $k \leftarrow k - 1$ 
12:    end while
13:    if  $k = n$  then
14:       $x_k \leftarrow 0$ 
15:      continue
16:    end if
17:    if  $k \geq 1$  then
18:       $x_k \leftarrow x_k - 1$ 
19:       $V' = \sum_{i=1}^k (v_i \times x_i)$ 
20:       $P'' = P - \sum_{i=1}^k (p_i \times x_i)$ 
21:      if  $V' + (v_{k+1}/p_{k+1} \times P'') \leq V_{best}$  then
22:        continue
23:      else
24:        for  $j \leftarrow k + 1, n$  do
25:           $x_j \leftarrow \lfloor (C - \sum_{i=1}^j (p_i \times x_i)) / p_j \rfloor$ 
26:        end for
27:         $V \leftarrow \sum_{i=1}^n (v_i \times x_i)$ 
28:        if  $V > V_{best}$  then
29:           $x_{best} \leftarrow x$ 
30:           $V_{best} \leftarrow V$ 
31:        end if
32:         $k \leftarrow n$ 
33:      end if
34:    else
35:      break
36:    end if
37:  end while
38: end procedure

```

- O *Problema da Mochila 0-1* possui complexidade pessimista $O(2^n)$, pois cada x_i pode assumir dois valores, 0 e 1;
- O *Problema da Mochila Limitado* possui complexidade pessimista $O(\max(\{b_1, \dots, b_n\})^n)$, pois, no pior caso, o número máximo de sub-árvores geradas é igual à maior quantidade de itens disponível;
- O *Problema da Mochila Ilimitado* possui complexidade pessimista $O((C/\min(\{p_1, \dots, p_n\}))^n)$, pois, analogamente ao caso anterior, o número máximo de sub-árvores é determinado pelo maior x_i possível, que, não possuindo limites explícitos, é determinado pela quantidade máxima do item que possui menor peso $(C/\min(\{p_1, \dots, p_n\}))$.

tais considerações se mostram afinadas com a literatura (MARTELLO; TOTH, 1990).

Em geral, o Problema da Mochila é resolvido por Programação Dinâmica, técnica algorítmica que utiliza recursões e uma estrutura de dados auxiliar que armazena os resultados parciais destas recursões para evitar a repetição de trabalho. No entanto, ambas as soluções possuem a mesma complexidade (KELLER; PFERSCHY; PISINGER, 2005).

4.5.2 MPI

Uma vez que se tenha um algoritmo formal estabelecido e implementado em uma linguagem flexível, como *Python*, o próximo passo é paralelizar este algoritmo, através de uma implementação da especificação MPI. No entanto, há um passo intermediário, que é implementar o algoritmo sequencial em *C* e, aí então, paralelizá-lo. Primeiramente, apresenta-se a mera transcrição do programa em *Python* para *C*, na Figura 4.4. A declaração das variáveis é vista na Figura 4.3

A função `read_instance(...)` lê um arquivo (`instance.txt`) o qual contém uma instância do Problema da Mochila, conforme a Figura 4.1. A função `dotprod(...)` realiza o produto escalar entre dois vetores ou partes de vetores (indicados através dos parâmetros `int low` e `int high`). O produto escalar corresponde ao somatório do produto entre vetores (Algoritmo 4).

Para evidenciar um processo de paralelização mais claro, convém agrupar certas expressões em termos de operações fundamentais do algoritmo, como *branch e bound*, que pudessem ser transformadas em funções independentes, trazendo mais clareza ao código. Tal abordagem pode ser visualizada na Figura 4.5.

Para realizar a paralelização deste algoritmo usa-se uma técnica conhecida de programação paralela, a *Decomposição Cíclica*. Basicamente, esta técnica consiste em replicar a entrada entre todos os processos participantes (processo realizado automaticamente pelo MPI) e o processo selecionar, com base no seu identificador, que partes de entrada processará e que partes da entrada deixará que outros processos computem. O nome “cíclica” advém do fato desejado que a distribuição da entrada seja cíclica, ou seja, que posições sucessivas em um vetor de entrada sejam distribuídos dentre processos diferentes (PACHECO, 1997).

Existe outra abordagem para a Distribuição da entrada muito utilizada em programa do tipo mestre-escravo, chamada *Decomposição por Blocos*; ao invés de enviar elementos de um vetor sucessivos para processos diferentes, se enviam blocos de elementos sucessivos para processos diferentes. Tal abordagem visa economizar o custo da comunicação envolvido no primeiro caso, visto que o envio de uma mensagem grande é mais eficiente

```

1  from modknapsack import readInstance
2
3  knapsack_instance = readInstance(sys.argv[1])
4  C = knapsackInstance['size']
5  v = knapsackInstance['value']
6  p = knapsackInstance['weight']
7  n = knapsackInstance['itemnum']
8  x = [0 for i in range(n)]
9
10 x[0] = C // p[0]
11 for j in range(1, n) :
12     x[j] = (C - sum([p[i]*x[i] for i in range(0, j)])) // s[j]
13
14 V_best = sum([v[i]*x[i] for i in range(0, n)])
15 x_best = x[:]
16
17 while True :
18     k = n-1
19     while (k >= 0) and (x[k] == 0) :
20         k = k - 1
21     if k == n-1 :
22         x[k] = 0
23         continue
24     if k >= 0 :
25         x[k] = x[k] - 1
26         V_temp = sum([v[i]*x[i] for i in range(0, k+1)])
27         P_temp = C - sum([p[i]*x[i] for i in range(0, k+1)])
28         if (V_temp + ((v[k+1] * P_temp) / p[k+1])) <= V_best :
29             continue
30         else :
31             k = k + 1
32             for j in range(k, n) :
33                 x[j] = (C-sum([p[i]*x[i] for i in range(0, j)]))//p[j]
34             V = sum([v[i]*x[i] for i in range(0, n)])
35             if V > V_best :
36                 x_best = x[:]
37                 V_best = V
38     else :
39         break

```

Figura 4.2: Algoritmo *Branch & Bound* para o Problema da Mochila, em *Python*.

```

1  int i, j;
2  struct ks_instance_s ks_ins;
3  read_instance("instance.txt", &ks_ins);
4  int *x = (int *) calloc(ks_ins.item_num, sizeof(int));
5  int *x_best = (int *) calloc(ks_ins.item_num, sizeof(int));
6  int v_temp, b_temp;
7  int k;

```

Figura 4.3: Variáveis da implementação sequencial que resolve o Problema da Mochila (em C).

```

1  int
2  main(int argc, char* argv[])
3  {
4      (... VARS... )
5      x[0] = (int) (ks_ins.ks_capacity / ks_ins.item_weight[0]);
6      for ( j = 1; j < ks_ins.item_num; ++ j )
7          x[j] = (int) ((ks_ins.ks_capacity
8                        - dotprod(ks_ins.item_weight, x, 0, j - 1))
9                        / ks_ins.item_weight[j]) ;
10
11     int v = 0;
12     int v_best = dotprod(ks_ins.item_value, x, 0, ks_ins.item_num - 1);
13     memcpy(x_best, x, ks_ins.item_num * sizeof(int) / sizeof(char));
14
15     k = ks_ins.item_num - 1;
16     while ( 1 ) {
17         while ( (k >= 0) && (x[k] == 0) )
18             k = k - 1;
19         if ( k == ks_ins.item_num - 1 ) {
20             x[k] = 0;
21             continue;
22         }
23         if ( k >= 0 ) {
24             x[k] = x[k] - 1;
25             v_temp = dotprod(ks_ins.item_value, x, 0, k);
26             b_temp = ks_ins.ks_capacity
27                     - dotprod(ks_ins.item_weight, x, 0, k);
28             if ( (v_temp + ((ks_ins.item_value[k+1] * b_temp)
29                       / ks_ins.item_weight[k+1])) <= v_best ) {
30                 printf("x[%d] = %d BOUND!\n", k, x[k]);
31                 continue;
32             } else {
33                 for ( j = k + 1; j < ks_ins.item_num; ++ j )
34                     x[j] = (int) ((ks_ins.ks_capacity
35                                   - dotprod(ks_ins.item_weight, x, 0, j - 1))
36                                   / ks_ins.item_weight[j]) ;
37                 v = dotprod(ks_ins.item_value, x, 0, ks_ins.item_num-1);
38                 if ( v > v_best ) {
39                     memcpy(x_best, x, ks_ins.item_num
40                           * sizeof(int)
41                           / sizeof(char));
42                     v_best = v;
43                 }
44                 k = ks_ins.item_num - 1;
45             }
46         } else {
47             break;
48         }
49     }

```

Figura 4.4: Algoritmo *Branch & Bound* para o Problema da Mochila, em C

```

1  read_instance(filename, &ks_ins);
2  x = (int *) calloc(ks_ins.item_num, sizeof(int));
3  branch(ks_ins, x, 0);
4  v = 0;
5  ks_sol.x = (int *) myalloc(ks_ins.item_num, sizeof(int));
6  ks_sol.v = dotprod(ks_ins.item_value, x, 0, ks_ins.item_num - 1);
7  memcpy(ks_sol.x, x, ks_ins.item_num * sizeof(int) / sizeof(char));
8
9  v = dotprod(ks_ins.item_value, x, 0, ks_ins.item_num - 1 );
10 k = ks_ins.item_num - 1;
11 while ( 1 ) {
12     while ( (k > 0) && (x[k] == 0) )
13         k = k - 1;
14     if ( k == ks_ins.item_num - 1 ) {
15         x[k] = 0;
16         continue;
17     }
18     if ( k > 0 ) {
19         x[k] = x[k] - 1;
20         if ( bound(ks_ins, x, k, ks_sol.v) ) {
21             continue;
22         } else {
23             branch(ks_ins, x, k + 1);
24             v = dotprod(ks_ins.item_value, x, 0, ks_ins.item_num - 1 );
25             update_best_sol(ks_ins, &ks_sol, x, v);
26             k = ks_ins.item_num - 1;
27         }
28     }
29     else
30         break;
31 }
32 free(x);
33 free(ks_sol.x);
34 free_instance(&ks_ins);

```

Figura 4.5: Algoritmo C com primitivas básicas para o Problema da Mochila

que o envio de várias pequenas, pois minimiza trocas de mensagens (DONGARRA et al., 2003).

É nítido que, ao possibilitar o ajuste do tamanho do bloco enviado, na *Decomposição por Blocos*, pode-se encarar a *Decomposição Cíclica* como um caso especial desta, onde o tamanho do bloco é 1 (DONGARRA et al., 2003). Adicionalmente, para o Problema da Mochila, é interessante distribuir um ramo da árvore, contado da raiz, para cada processo; logo, o número de elementos que compõe a entrada para os processos do algoritmo paralelo que resolve o Problema da Mochila são os valores que x_1 (doravante referenciado como o código C , $x[0]$) pode assumir. A implementação da função de distribuição da entrada é apresentada na Figura 4.6. A função retorna o identificador do processo que deve processar aquele elemento da entrada e seu significado é apresentado na Tabela 4.1.

```
int
owner(int num_procs, int block_size, int index)
{
    int block = (int) index/block_size;
    int p = block % num_procs;
    return p;
}
```

Figura 4.6: Função que realiza a distribuição cíclica da entrada.

Variável	Significado
int num_procs	Número total de processos.
int block_size	Tamanho de bloco desejado.
int index	Posição da entrada a qual será distribuída.

Tabela 4.1: Significado das variáveis na função de distribuição cíclica da entrada.

Quando `block_size` é 1, há um simples *hash* da entrada, calculado através do módulo. Esta abordagem é a adotada, visto que o algoritmo *não* é do tipo mestre-escravo; todos os processos calculam o valor ótimo de seus ramos, sem precisar receber a entrada de alguém. Como não há envio (fora a distribuição inicial de dados que o MPI realiza de forma transparente) de dados a processar e todos os processos possuem, inicialmente, a entrada replicada, convém optar pela simples *Distribuição Cíclica*, visto os benefícios óbvios que esta traz em relação ao balanceamento de carga; blocos muito grandes agregam fragmentação externa e, possivelmente, os processos de *ranks* (identificadores) mais elevados computariam menos que processos de *ranks* mais baixos. A inserção desse mecanismo, bem como os preâmbulos MPI necessários à paralelização podem ser vistos na Figura 4.7, onde a estrutura `ks_sol` guarda a solução (a distribuição em `x[]` que apresenta o valor máximo, `ks_sol.x` e esse valor, `ks_sol.v`).

A implementação apresentada do algoritmo que resolve o Problema da Mochila, paralelizado, ainda está incompleta; cada processo encontra sua distribuição de valor máximo local, mas não há comunicação entre eles para determinar qual o melhor global, dentre todos os ramos analisados. Este processo será abordado no capítulo seguinte, quando será inserido num contexto mais complexo, ao se tratar de *Difusão de Máximo Local*, na parte que tange o *Workstealing*.

```

1  read_instance(filename, &ks_ins);
2  x = (int *) calloc(ks_ins.item_num, sizeof(int));
3  branch(ks_ins, x, 0);
4  v = 0;
5  ks_sol.x = (int *) calloc(ks_ins.item_num, sizeof(int));
6  ks_sol.v = dotprod(ks_ins.item_value, x, 0, ks_ins.item_num - 1);
7  memcpy(ks_sol.x, x, ks_ins.item_num * sizeof(int) / sizeof(char));
8
9  int num_procs; // Number of procs.
10 int my_rank;   // Current proc's rank.
11
12 MPI_Init(&argc, &argv);
13 MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
14 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16
17 v = dotprod(ks_ins.item_value, x, 0, ks_ins.item_num - 1);
18 k = ks_ins.item_num - 1;
19 while ( 1 ) {
20     while ( (k > 0) && (x[k] == 0) )
21         k = k - 1;
22     if ( k == ks_ins.item_num - 1 ) {
23         x[k] = 0;
24         continue;
25     }
26     if ( k > 0 ) {
27         x[k] = x[k] - 1;
28         if ( my_rank == owner(num_procs, 1, x[0]) ) {
29             if ( bound(ks_ins, x, k, ks_sol.v) ) {
30                 continue;
31             } else {
32                 branch(ks_ins, x, k + 1);
33                 v = dotprod(ks_ins.item_value, x, 0,
34                             ks_ins.item_num - 1);
35                 update_best_sol(ks_ins, &ks_sol, x, v);
36                 k = ks_ins.item_num - 1;
37             }
38         }
39     }
40     else
41         break;
42 }
43 free(x);
44 free(ks_sol.x);
45 free_instance(&ks_ins);
46 MPI_Finalize();

```

Figura 4.7: Algoritmo C/MPI paralelizado.

5 WORKSTEALING

Este capítulo se dedica a apresentar técnicas de otimização do algoritmo paralelo *Branch & Bound* que resolve o Problema da Mochila apresentado nos capítulos anteriores. O foco dessas técnicas será a técnica de *Workstealing* (BLUMOFFE; LEISERSON, 1994), que busca garantir um balanceamento de carga ideal e distribuição homogênea das tarefas entre os processos. Antes disso, no entanto, é necessário corrigir a deficiência apresentada no final do Capítulo 4, *Problema da Mochila*, oferecendo uma maneira de comparar os máximos locais e obter um máximo global. Essa correção não faz parte do algoritmo de WS em si. No entanto, como ambos são implementados com a mesma arquitetura, é interessante os colocar em seqüência.

5.1 Difusão de Máximo Local

A questão não é apenas a difusão dos máximos locais ao *final* da computação de todos os processos. É interessante que os processos, durante a execução das tarefas a eles atribuídas, tenham a informação de qual é o atual valor máximo global; com isso, é possível otimizar o uso do *bound* de forma a cortar um número maior de galhos que não apresentem potencial para obter uma solução ótima. Esta troca de informações, entretanto, deve permitir que os processos continuem a computar, enquanto esperam por uma notificação de que um novo melhor foi encontrado.

A maneira mais eficiente de esperar a notificação mencionada seria através do uso de *Threads*. Entretanto, as implementações MPI ainda não conseguem ter um comportamento estável quando utilizam tal recurso. Uma solução paliativa, no caso, é o emprego de comunicação não-bloqueante.

5.1.1 Algoritmo

O algoritmo empregado é bem simples, onde segue

1. Ao achar um novo máximo local, fazer um `broadcast` não-bloqueante a todos os processos;
2. Um processo, ao receber uma notificação de que existe um novo melhor local, com para este, recebido na notificação, com o que possui (`ks_sol.v`). Então, duas coisas podem ocorrer:
 - (a) Substituição do atual melhor, local, pelo melhor global. Neste caso uma indicação de que o atual melhor não foi encontrado pelo processo local é indicada zerando-se todos os elementos do vetor que fornece a melhor solução, `ks_sol.x`;

- (b) Não substituição do melhor local pelo global. Pode acontecer se os tempos relativos entre a descoberta e o envio do novo máximo, entre dois ou mais processos, for aproximado. Neste caso, a mensagem é simplesmente descartada.
- 3. Ao final de sua execução, um processo envia uma mensagem de fim de execução a todos os outros.
- 4. Ao receber a mensagem de fim, o processo subtrai uma unidade do número de processos ativos, que todos os processos guardam.

5.1.2 Implementação

A implementação desta difusão é de especial interesse, pois é construída com a mesma arquitetura que é empregada na implementação do *Workstealing*.

MPI não possui uma implementação de *broadcast* assíncrono, por questões de *deadlock*. Logo, deve-se implementar tal recurso e, ainda, garantir que o *deadlock* não ocorra. A prova formal de que ele não ocorre será apresentada no Capítulo 6, *Avaliação de Desempenho*. Apenas sua implementação isenta de travas é mostrada. O *broadcast* assíncrono, ao mandar uma mensagem, é implementado pela função vista na Figura 5.1, onde `MPI_Send(...)` é usado no lugar de `MPI_Isend` porque poupa gerenciamento de *buffer* e não depende de outros processos para se desbloquear. Apenas o recebimento desta mensagem deve ser não-bloqueante, para a sincronização.

```
int
mpi_async_broadcast(void *data, int num_item, MPI_Datatype datatype,
    int tag, MPI_Comm comm, int num_procs, int my_rank)
{
    int i;
    for ( i = 0; i < num_procs; ++ i )
        if ( i != my_rank )
            MPI_Send(data, num_item, datatype, i, tag, comm);
    return 1;
}
```

Figura 5.1: Implementação de *broadcast* assíncrono.

Para gerenciar o recebimento destes dados, é necessária uma lógica mais mais complexa. A opção escolhida foi montar um estrutura de dados gerenciadora, com todas as variáveis e funções (*interface*) implementadas de maneira modular, por dois motivos:

1. Dar suporte futuro a uma biblioteca feita sobre MPI para a *Difusão de Máximo Local* genérica; e
2. Melhor encapsulamento, controle e depuração, ao eliminar replicação de código.

A definição do gerenciador de difusão é dada na Figura 5.2. Os dois `MPI_Request` atendem, cada um, um tipo de mensagem, em modelo semelhante ao que foi abordado na Subseção 3.2.5. `v_global` guarda o atual melhor *notificado* e `num_active_procs` guarda quantos processos ainda estão ativos, para prevenir o *deadlock*, através da primitiva¹ `mpi_process_nonblocking(...)`, explanada a seguir.

¹Note-se que, até aqui, todas as funções implementadas que usem alguma primitiva MPI tem seu nome

```

/* Message Types. */
#define NEW_BEST_MSG          0
#define END_MSG              1

struct mpi_dif_manager_s {
    MPI_Request request_end_msg;
    MPI_Request request_new_best_msg;
    int v_global;
    int null; // For unimportant data.
    int num_active_procs;
    int whoinf; // Who gave this data.
};

void mpi_init_dif_manager(struct mpi_dif_manager_s *mpi_man,
                        int num_procs);
void mpi_free_dif_manager(struct mpi_dif_manager_s *mpi_man);
void mpi_process_nonblocking(struct mpi_dif_manager_s *mpi_man,
                            struct ks_solution_s *ks_sol,
                            struct ks_instance_s ks_ins);

int active_procs_remain(struct mpi_dif_manager_s mpi_man);
int mpi_nonblocking_broadcast(void *data,
                             int num_item,
                             MPI_Datatype datatype,
                             int tag,
                             MPI_Comm comm,
                             int num_procs,
                             int my_rank);

```

Figura 5.2: Implementação do gerenciador de difusão.

A função `mpi_process_nonblocking(...)` testa a recepção de uma mensagem de fim de computação ou de novo máximo local encontrado e deve ser invocada toda vez que se quiser tratar o recebimento de uma mensagem de difusão ao longo do código. Como o próprio nome já diz, ela é não bloqueante. Isso torna a verificação atômica e garante tempo de processamento reservado. Em geral é útil invocá-la a cada nova geração de um conjunto candidato a solução $x[]$, dando-lhe periodicidade. O tratamento dessa mensagem é feito automaticamente pela função, bem como a atualização do atual melhor valor (através do ponteiro para `ks_sol`), de maneira transparente ao programador, que não precisa adicionar linhas de código para tratar cada mensagem.

Esta transparência da `mpi_process_nonblocking(...)` tem pouco efeito sobre a lógica do algoritmo, simplificando a programação. No entanto, o tratamento destas mensagens exige certos detalhes que a função deve atender, para que não haja *deadlock*. A implementação desta função pode ser observada na Figura 5.3.

```
void
mpi_process_nonblocking(struct mpi_dif_manager_s *mpi_dif_man,
                        struct ks_solution_s *ks_sol,
                        struct ks_instance_s ks_ins )
{
    int flag;          // Flag for non-blocking comm.
    MPI_Status status; // Status for recv-like comm.

    MPI_Test(&(mpi_dif_man -> request_new_best_msg), &flag, &status);
    if ( flag != 0 ) {
        if ( (mpi_dif_man -> v_global) >= (ks_sol -> v) ) {
            ks_sol -> v = mpi_dif_man -> v_global;
            mpi_dif_man -> whoinf = status.MPI_SOURCE;
        }
        MPI_Irecv(&(mpi_dif_man -> v_global),
                  1,
                  MPI_INT,
                  MPI_ANY_SOURCE,
                  NEW_BEST_MSG,
                  MPI_COMM_WORLD,
                  &(mpi_dif_man -> request_new_best_msg));
    }
    MPI_Test(&(mpi_dif_man -> request_end_msg), &flag, &status);
    if ( flag != 0 ) {
        -- (mpi_dif_man -> num_active_procs);
        MPI_Irecv(&(mpi_dif_man -> null),
                  1,
                  MPI_INT,
                  MPI_ANY_SOURCE,
                  END_MSG,
                  MPI_COMM_WORLD,
                  &(mpi_dif_man -> request_end_msg));
    }
}
```

Figura 5.3: Implementação de `mpi_process_nonblocking()`.

precedido por `mpi_`. Faz-se isso para (a) diferenciar as funções implementadas na especificação MPI e as específicas do problema, (b) identificar, facilmente, quais funções fazem E/S de troca de mensagens em alto nível.

A função `mpi_init_dif_manager(...)` inicializa as *requests* e a função acima testa o recebimento de mensagens sobre elas, chamando-as novamente após o recebimento de alguma mensagem. Ao receber uma mensagem de fim, simplesmente subtrai o número de processos ativos de uma unidade.

Quando um processo acaba sua computação e envia sua mensagem de fim a todos os outros, é necessário que ele cancele seus *receives* não-bloqueantes. No entanto, esta abordagem simplista introduz alguns problemas, a saber:

1. Os processos não tem controle de quais outros processos acabam, apenas do número total dos processos ativos. Logo, os outros processos vão enviar mensagens normalmente que, sem ninguém para recebê-las, esgotarão o *buffer*;
2. Mesmo que um controle seja implementado, pode ser que o processo se encerre existindo mensagens destinadas a ele transitando na rede, lotando o *buffer*;
3. Se for optado por não cancelar os *receives*, então a LAM MPI travará quando chamar `MPI_Finalize(...)`, pois ocorrerá *deadlock* dentre seus processos (para que a LAM MPI encerre, é necessário que os *buffers* de mensagens estejam vazios, ou seja, que todos os *sends* tenham um *receive* correspondente).

A solução para evitar o *deadlock*, neste caso, é fazer com que todos os processos só cancelem as suas comunicações não-bloqueantes quando tiverem certeza de que nenhum dos outros processos pode mandar mensagens ou que mensagens estão em trânsito; em outras palavras, um processo só deve para de receber mensagens quando todos chegarem ao final.

A solução óbvia que surge é a implementação de um algoritmo de detecção de término distribuído (JAJA, 1992). No entanto, o cenário de aplicações de um algoritmo deste tipo é mais restrito, sem comunicação *broadcast*, e, por isso, sua implementação é mais complexa do que o necessário.

Algo que serve bem para sanar o problema apresentado é uma sincronização do tipo barreira² (TOSCANI; OLIVEIRA; SILVA CARÍSSIMI, 2003). De fato, MPI oferece `MPI_Barrier(...)` para esse propósito. No entanto, a solução MPI não é adequada; os processos não podem ficar parados enquanto os outros processos computam, sendo necessário que eles ainda possam receber notificações de novos melhores encontrados para que, ao final da computação, todos tenham a mesma resposta (global) que soluciona o problema.

É necessário implementar um mecanismo de barreira de mais alto nível, em que um processo, ao acabar sua computação, fique apenas recebendo notificações de novo máximo ou de fim de processo. Apenas quando uma mensagem de *fim* é recebida de todos os outros processos é que o processo se encerra. Tal mecanismo, no entanto, é de fácil implementação (ao final do código), conforme a Figura 5.4. A cada mensagem de fim que chega, o número de processos ativos é decrementado e, apenas quando todos informam isto é que todos estão livres para terminar a computação. Isto também garante que condições de corrida não ocorram; um ponto de barreira, ao final, garante que todos os processos de fato receberam a última mensagem de notificação, anterior à mensagem *end* e que todos possuem a mesma resposta.

A integração do código mostrado com a *Difusão de Máximo Local* é mostrado na Figura 5.5.

²Uma sincronização em que todos os fluxos de programa ficam parados ao atingirem certo ponto do código e só prosseguem quando todos estiverem neste mesmo ponto.

```

/* Prevents deadlock and the race condition. */
while ( active_procs_remain(mpi_dif_man) )
    mpi_process_nonblocking(&mpi_dif_man, &ks_sol, ks_ins);

```

Figura 5.4: Implementação do mecanismo de barreira.

5.2 Workstealing

A técnica de *Workstealing* (BLUMOFÉ; LEISERSON, 1994) é um conceito formalizado academicamente, mas que ainda não encontrou uma implementação *genérica* que se estabelecesse como padrão *de facto*. Muitos autores publicaram sua síntese sob outros nomes *e.g.*, árvore de busca paralela (PACHECO, 1997), mas a essência do algoritmo é mantida.

A técnica tem, por objetivo, obter um balanceamento de carga entre os processos. Isto significa que, ao garantir uma distribuição igualitária de tarefas, os processos ficarão menos ociosos e, portanto, será obtida uma exploração maior do paralelismo. Além disso, o método possui robustez suficiente para equilibrar cargas que, pela definição do problema, são naturalmente desbalanceadas (*e.g.*, árvore degenerada). Esta monografia se propõe a provar, na prática, se esta solução obtém o ganho de desempenho teórico a que se propõe.

5.2.1 Algoritmo

Os elementos necessários para a implementação são:

1. Cada processo possuir uma pilha de tarefas. Estas tarefas são, ao longo da execução, desempilhadas e processadas;
2. Mensagens específicas, definidas previamente, de solicitação de trabalho, de resposta com envio de trabalho e resposta vazia.

A sequência clássica de operações (BLUMOFÉ; LEISERSON, 1994) realizadas pelos processos consiste em

1. Caso a pilha de tarefas não esteja vazia, desempilhar as tarefas, sequencialmente, e realizá-las, uma a uma;
2. Caso a pilha estiver vazia, escolher aleatoriamente um processo e solicitar trabalho para este;
3. Se o processo fornecer tarefas, empilha-las e ir para 1. Caso contrário, ir para 4;
4. Caso o processo não forneça tarefas, repetir 2 até que se receba uma tarefa ou um *timeout* pré-definido seja atingido;
5. Parar.

No início do algoritmo tradicional, todos os processos começam com suas pilhas vazias, a exceção de um, cuja pilha contém todas as tarefas. Desta maneira, o balanceamento de carga ocorre naturalmente, enquanto o algoritmo se desenvolve.

```

1  (...INIT VARS...)
2
3  MPI_Init(&argc, &argv);
4  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
5  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
6
7  v = dotprod(ks_ins.item_value, x, 0, ks_ins.item_num - 1 );
8  k = ks_ins.item_num - 1;
9  while ( 1 ) {
10     while ( (k > 0) && (x[k] == 0) )
11         k = k - 1;
12     if ( k == ks_ins.item_num - 1 ) {
13         x[k] = 0;
14         continue;
15     }
16     if ( k > 0 ) {
17         x[k] = x[k] - 1;
18         if ( my_rank == owner(num_procs, 1, x[0]) ) {
19             if ( bound(ks_ins, x, k, ks_sol.v) ) {
20                 continue;
21             } else {
22                 branch(ks_ins, x, k + 1);
23                 v = dotprod(ks_ins.item_value, x, 0, ks_ins.item_num-1);
24                 if ( update_best_sol(ks_ins, &ks_sol, x, v) ) {
25                     /* The one that have the maximum is me! */
26                     mpi_dif_man.whoinf = my_rank;
27                     /* Broadcast the new local maximum. */
28                     mpi_async_broadcast(&(ks_sol.v), 1, MPI_INT,
29                                     NEW_BEST_MSG, MPI_COMM_WORLD, num_procs,
30                                     my_rank );
31                 }
32             }
33         }
34         else
35             break;
36     }
37     mpi_nonblocking_broadcast(&(ks_sol.v), 1, MPI_INT, END_MSG,
38                             MPI_COMM_WORLD, num_procs, my_rank);
39
40     /* Prevents deadlock and the race condition. */
41     while ( active_procs_remain(mpi_dif_man) )
42         mpi_process_nonblocking(&mpi_dif_man, &ks_sol, ks_ins);
43     /* Cancels the non-blocking comms. */
44     mpi_free_dif_manager(&mpi_dif_man);
45     free(x);
46     free(ks_sol.x);
47     free_instance(&ks_ins);
48     MPI_Finalize();

```

Figura 5.5: Algoritmo C/MPI paralelizado e com *Difusão de Máximo Local*.

5.2.2 Considerações

Para fins da aplicação neste trabalho, o algoritmo clássico sofreu algumas modificações. Tais modificações foram introduzidas visando obter maior desempenho, modificando certas premissas do algoritmo original. As alterações foram inseridas com especial cuidado para evitar a ocorrência de *deadlocks*, já que tocam na questão da sincronização do envio de mensagens.

Inicialmente, ao invés de um único processo possuir a pilha cheia, todas as pilhas são abastecidas com tarefas via *Distribuição Cíclica* (vide Subseção 4.5.2). Isso faz com que o fluxo de troca de mensagens grande que ocorre no início o *Workstealing* cesse, o que provoca:

1. Menor execução de primitivas de troca de mensagens (E/S);
2. Menor tráfego na rede;
3. Balanceamento de carga inicial (diminui a necessidade de troca de mensagens ao longo da execução);
4. Melhor aproveitamento do tipo de entrada do problema considerado (Problema da Mochila).

Todos os fatores acima parecem contribuir para um *aumento* direto do desempenho (a ser verificado no Capítulo 6, *Avaliação de Desempenho*). Além disso, visando melhorar a distribuição de processos, a estrutura de dados que armazena tarefas não é uma pilha, mas sim uma fila de distribuição dupla (*double sided queue*³). Esta organização permite que o a resolução das tarefas enxergue a estrutura de dados como uma pilha e que o roubo de tarefas enxergue a mesma situação como uma fila (Figura 5.6). Como, heurísticamente, as soluções que maximizam os elementos iniciais de $x[]$ oferecem maior potencialidade de serem a solução ótima, pode-se empilhá-las por último. Isto garante um bom índice de *bound* (pois serão desempilhadas primeiro) ao mesmo tempo em que, ao atender uma solicitação de roubo, o processo fornecerá tarefas menos promissoras, que são ideais para sofrerem um maior atraso, devido a latência de rede.

Outras alterações foram feitas para coibir a ocorrência de postergação indefinida. Ao imaginar um cenário onde a mesma tarefa é repassada circularmente entre todos os processos, sem nunca ser processada e sem nunca atingir o *timeout* fica evidente que, dentro do modelo clássico, a postergação indefinida é possível. Para contornar esta situação, adotam-se duas medidas:

1. Apenas *uma* tarefa é passada por vez a cada solicitação de tarefas atendidas; e
2. Uma tarefa recebida por meio de *workstealing* é imediatamente processada e não pode ser repassada.

O primeiro item tem implementação trivial. O segundo, no entanto, é mais complicado; a função que trata o recebimento de mensagens se comunica com os processos apenas via pilha. Para contornar esta limitação, opta-se por restringir o uso da primeira posição da pilha; qualquer tarefa alocada neste espaço nunca é enviada por *Workstealing*. Como, ao solicitar tarefas, um processo indica que sua pilha está vazia, qualquer tarefa recebida se alojará nesta posição e não correrá o risco de ser repassada.

³Esta estrutura de dados que possui característica de incluir dados somente através de um ponteiro para início da fila, mas de retirar dados através de qualquer uma de suas extremidades.

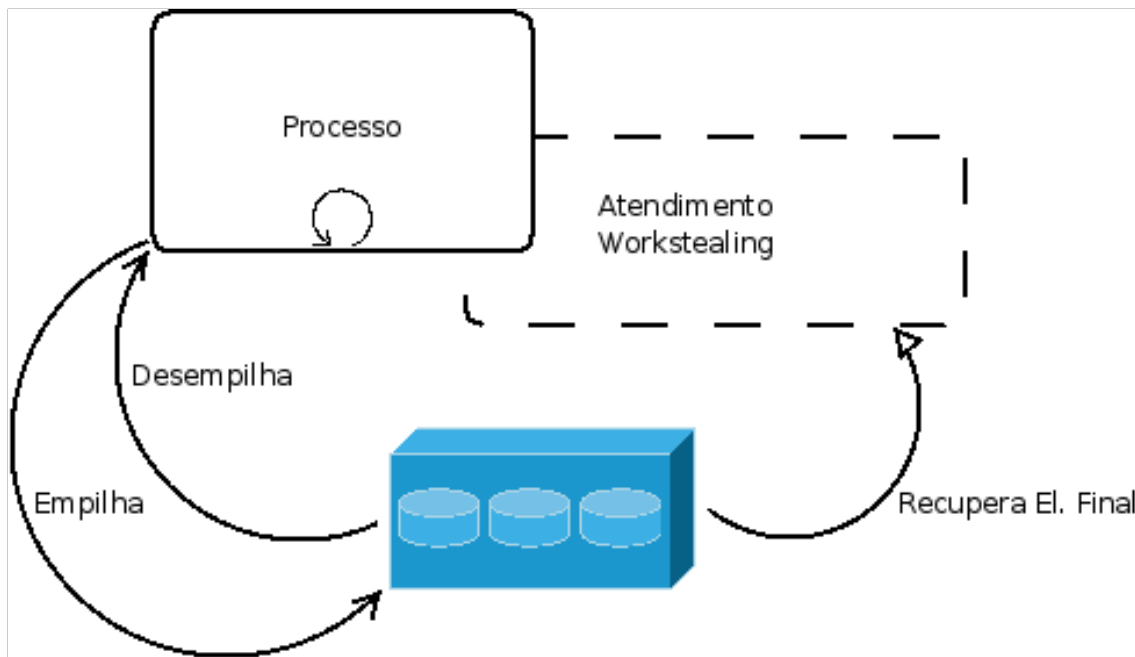


Figura 5.6: Esquema de funcionamento – Fila de Distribuição Dupla

Estas simples medidas claramente (a) evitam que uma tarefa seja repassada *ad infinitum* (eliminando o risco de postergação indefinida) e (b) uniformizam e simplificam o processo de implementação do *Workstealing*.

5.2.3 Implementação

Seguindo a ordem apresentada na subseção anterior, primeiramente deve-se modificar o código para que este processe o conceito de tarefas e consiga empilhar estas tarefas numa estrutura de fila de distribuição dupla. Inicialmente, o problema foi modelado definindo-se uma tarefa como um conjunto $\times []$ (galho) e a computação dessa tarefa como o cálculo do valor de sua utilidade e sua comparação com a melhor utilidade local. Essa abordagem, no entanto, revelou-se muito ruim, pois gera:

Mal aproveitamento de recursos computacionais. A complexidade do Problema da Mochila reside na geração de todos os conjuntos $\times []$ e não no cálculo de seu valor. Balancear a tarefa menos complexa é um grande fator de desperdício de recursos, pois a tarefa mais pesada fica desbalanceada;

Consumo intratável de memória. Em testes realizado no *cluster*, a execução desse problema com 100 elementos consumiu memória RAM na ordem de 2GB. Como a geração de conjuntos é uma operação exponencial e cada conjunto tem um grande número de elementos, o consumo de memória torna-se, também, exponencial, o que é intratável.

A abordagem alternativa (e que foi adotada), é considerar que uma tarefa é um número inteiro, que representa um dos valores possíveis de $\times [0]$. Com isso, cada tarefa passa a ser o cálculo de todas as sub-árvores cujo primeiro elemento possui o valor associado ao inteiro que representa a tarefa. Essa abordagem sana os defeitos da aproximação anterior ao problema.

Definindo uma pilha e a noção de tarefa, a estrutura de dados (com a respectiva interface) é implementada conforme a Figura 5.7.

```

struct task_dfq_s {
    struct task_dfq_node_s *start;
    struct task_dfq_node_s *end;
    int size;
};

struct task_dfq_node_s {
    int task;
    struct task_dfq_node_s *next;
    struct task_dfq_node_s *previous;
};

int init_dfq(struct task_dfq_s* dfq);
int pop(struct task_dfq_s* dfq, int task);
int push_front(struct task_dfq_s* dfq);
int push_back(struct task_dfq_s* dfq);
int is_empty(struct task_dfq_s dfq);
int get_size(struct task_dfq_s dfq);

```

Figura 5.7: Implementação da fila de distribuição dupla.

A inserção do conceito de pilha no código é simples (Figura 5.8), mas tem uma consideração importante. Como agora os valores de $x[0]$ não iniciam necessariamente no máximo (pois esse valor representa uma tarefa), o valor de $x[1]$ é que deve ser maximizado, visando uma execução correta do algoritmo guloso de geração de galhos.

O próximo passo é implementar a troca de mensagens em si. Para isso, usa-se uma construção análoga ao gerenciador de difusão apresentado anteriormente. Conforme já explicitado, um dos focos do trabalho realizado é oferecer as bases a um futuro suporte de uma biblioteca *Workstealing* genérica. Embora alguns fatores ainda dependam fortemente de elementos do Problema da Mochila, sua generalização é simples, mesmo que trabalhosa. Procurou-se, aqui, oferecer uma versão primitiva da abordagem de *callbacks*, presente em linguagens de paradigma de Orientação a Objetos, que permite uma generalização fortíssima a qualquer aplicação, bastando reescrever algumas funções que são chamadas internamente, em operações desta biblioteca.

Todas as considerações apresentadas sobre a construção de ambos os gerenciadores mantém a validade, gerando, inclusive, funções colocadas nos mesmos lugares. A estrutura de dados do gerenciador *Workstealing* é visitada na Figura 5.9. As funções da interface do gerenciador mais importantes são `mpi_ws_process_nonblocking(...)` (responsável por atender às requisições de roubo) e `mpi_workstealing(...)` (responsável por fazer as solicitações de roubo).

A função `mpi_ws_process_nonblocking(...)`, como o próprio nome sugere, é extremamente semelhante a `mpi_process_nonblocking(...)`, tendo modelos idênticos. É verificado se chegou mensagem de solicitação de tarefas (`WS_REQUEST_WORK_MSG`) e o tratamento dado consiste em, verificando se a pilha do atual processo tem tarefas, enviá-las ao processo solicitante (mensagem tipo `WS_TASK_MSG`) ou enviar uma mensagem de que não há tarefa disponível (`WS_EMPTY_MSG`). O cancelamento do *receive* é efetuado após a saída da barreira estabelecida pela *Difusão de Máximo Local*, aproveitando sua característica de ponto de retenção, para também evitar *deadlock*.

A essência do algoritmo, no entanto, reside na função `mpi_workstealing(...)`. Procura-se, ao máximo, dar transparência a esta função, que intermedia o acesso à pilha

```

1  MPI_Init(&argc, &argv);
2  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
3  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
4  mpi_init_dif_manager(&mpi_dif_man, num_procs);
5
6  // Builds a task and initializes the "stack".
7  init_dfq(&task_stack);
8  for ( i = 0; i <= x[0]; i ++ )
9      if ( my_rank == owner(num_procs, 1, i) )
10         pop(&task_stack, i);
11
12 // Points to end of the array, for disassembling the recursivity.
13 k = ks_ins.item_num - 1;
14
15 while ( ! is_empty(&task_stack) ) {
16     x[0] = push_front(&task_stack);
17     branch(ks_ins, x, 1);
18     v = dotprod(ks_ins.item_value, x, 0, ks_ins.item_num - 1 );
19     while ( 1 ) {
20         mpi_process_nonblocking(&mpi_dif_man, &ks_sol, ks_ins);
21         while ( (k > 0) && (x[k] == 0) )
22             k = k - 1;
23         if ( k == ks_ins.item_num - 1 ) {
24             x[k] = 0;
25             continue;
26         }
27         if ( k > 0 ) {
28             x[k] = x[k] - 1;
29             if ( bound(ks_ins, x, k, ks_sol.v) ) {
30                 continue;
31             } else {
32                 branch(ks_ins, x, k + 1);
33                 v = dotprod(ks_ins.item_value, 0, ks_ins.item_num-1 );
34                 if ( update_best_sol(ks_ins, &ks_sol, x, v) ) {
35                     mpi_dif_man.whoinf = my_rank;
36                     mpi_nonblocking_broadcast(&(ks_sol.v), 1, MPI_INT,
37                                             NEW_BEST_MSG, MPI_COMM_WORLD, num_procs,
38                                             my_rank );
39                 }
40                 k = ks_ins.item_num - 1;
41             }
42         } else
43             break;
44     }
45     mpi_async_broadcast(&(ks_sol.v), 1, MPI_INT, END_MSG, MPI_COMM_WORLD,
46                       num_procs, my_rank);
47     /* Prevents deadlock and the race condition. */
48     while ( active_procs_remain(mpi_dif_man) )
49         mpi_process_nonblocking(&mpi_dif_man, &ks_sol, ks_ins);
50     mpi_free_dif_manager(&mpi_dif_man);
51     (...)
52     MPI_Finalize();

```

Figura 5.8: Algoritmo C/MPI, versão com fila de distribuição dupla

```

/* Message Types */
#define WS_REQUEST_WORK_MSG 2
#define WS_TASK_MSG 3
#define WS_EMPTY_MSG 4

/* Request Types. */
#define WS_NUM_MSG_TYPE 2
#define WS_TASK_REQUEST 0
#define WS_EMPTY_REQUEST 1

struct mpi_ws_manager_s {
    MPI_Request request_ws_work_msg;
    struct ks_instance_s *ks_ins;
    struct task_dfq_s *stack;
    struct proc_table_s ws_proc_table;
    int my_rank;
    int num_procs;
    int null;
};

int mpi_init_workstealing(struct mpi_ws_manager_s *workst, struct
    task_dfq_s *stack, struct ks_instance_s *ks_ins, int my_rank, int
    num_procs );
int mpi_workstealing(struct mpi_ws_manager_s *workst);
int mpi_free_workstealing(struct mpi_ws_manager_s *workst);
int mpi_ws_process_nonblocking(struct mpi_ws_manager_s *workst);

```

Figura 5.9: Implementação do gerenciador de *Workstealing*.

de tarefas e, se esta se encontra vazia, faz a solicitação de tarefas. Encarando-se como uma máquina de estados, isso traz a seqüência:

1. Se existem tarefas na pilha, entrega a tarefa do topo da pilha ao processo;
2. Caso não exista tarefa na pilha, escolhe, dentre todos os processos, um, aleatoriamente;
3. Requisita-se tarefa ao processo escolhido;
4. Recebendo-se uma tarefa, entrega-a ao processo;
5. Recebendo uma negativa de tarefa, marca-se o processo em uma tabela como sem tarefas, escolhe-se outro processo e repetindo-se a tentativa;
6. Caso todos os processos existentes estejam marcados como sem tarefas, acaba-se a computação.

Note-se que o *timeout* é suprimido; é garantido, pela abordagem da implementação (já apresentada), que não há risco de postergação indefinida. Adicionalmente, pode-se notar que, uma vez que um processo seja identificado como sem tarefas, não há risco de ignorar-se uma eventual realimentação da pilha; como já foi estatizado, o algoritmo é implementado sem que esta ocorra, para que uma tarefa não seja repassada *ad infinitum*. Na Figura 5.10, a estrutura de tabela de processos que foi empregada. O código da função `mpi_process_nonblocking(...)` pode ser visto na Figura 5.11. O código do programa com as partes relevantes, todo integrado com o que foi apresentado, pode ser visto na Figura 5.12.

```
struct proc_table_s {  
    int *vector;  
    int size;  
};  
int init_proc_table(struct proc_table_s *table, int num_procs);  
int free_proc_table(struct proc_table_s *table);  
int set_all_active(struct proc_table_s *table);  
int set_all_inactive(struct proc_table_s *table);  
int set_proc_active(struct proc_table_s *table, int proc_rank);  
int set_proc_inactive(struct proc_table_s *table, int proc_rank);  
int is_proc_active(struct proc_table_s *table, int proc_rank);  
int exist_active_proc(struct proc_table_s *table);
```

Figura 5.10: Implementação da tabela de processos.

```

1  int
2  mpi_workstealing(struct mpi_ws_manager_s *workst)
3  {
4      int i;
5      int msg;
6      int target_proc;
7      int task = 0;
8      int ws_flag;
9      MPI_Request ws_request[WS_NUM_MSG_TYPE];
10     MPI_Status ws_status;
11     int ws_index;
12
13     if ( ! is_empty(*(workst -> stack)) )
14         return 1;
15     set_all_active(&(workst -> ws_proc_table));
16     set_proc_inactive(&(workst -> ws_proc_table), workst -> my_rank);
17
18     while ( exist_active_proc(&(workst -> ws_proc_table)) ) {
19
20         do { target_proc = rand() % (workst -> num_procs); }
21         while ( ! is_proc_active(&(workst -> ws_proc_table),
22             target_proc) );
23
24         msg = WS_REQUEST_WORK_MSG;
25         MPI_Send(&msg, 1, MPI_INT, target_proc, WS_REQUEST_WORK_MSG,
26             MPI_COMM_WORLD);
27         MPI_Irecv(&task, 1, MPI_INT, target_proc, WS_TASK_MSG,
28             MPI_COMM_WORLD, &(ws_request[WS_TASK_REQUEST]) );
29         MPI_Irecv(&task, 1, MPI_INT, target_proc, WS_EMPTY_MSG,
30             MPI_COMM_WORLD, &(ws_request[WS_EMPTY_REQUEST]) );
31
32         ws_flag = 0;
33         do {
34             MPI_Testany(WS_NUM_MSG_TYPE, ws_request, &ws_index,
35                 &ws_flag, &ws_status );
36             mpi_ws_process_nonblocking(workst, benchmark);
37         } while ( ws_flag == 0 );
38         for (i = 0; i < WS_NUM_MSG_TYPE; i ++ )
39             if ( i != ws_index)
40                 MPI_Cancel(&(ws_request[i]));
41
42         switch ( ws_status.MPI_TAG ) {
43             case WS_TASK_MSG :
44                 pop(workst -> stack, task);
45                 return 1;
46
47             case WS_EMPTY_MSG :
48                 set_proc_inactive(&(workst -> ws_proc_table),
49                     ws_status.MPI_SOURCE);
50                 break;
51         }
52     }
53     return 0;
54 }

```

Figura 5.11: Solicitação de tarefas - *Workstealing*.

```

1  (...)
2  while ( mpi_workstealing(&workst) ) {
3      x[0] = push_front(&task_stack);
4      branch(ks_ins, x, 1);
5      v = dotprod(ks_ins.item_value, x, 0, ks_ins.item_num - 1 );
6      while ( 1 ) {
7          mpi_process_nonblocking(&mpi_dif_man, &ks_sol, ks_ins);
8          mpi_ws_process_nonblocking(&workst);
9          while ( (k > 0) && (x[k] == 0) )
10             k = k - 1;
11          if ( k == ks_ins.item_num - 1 ) {
12              x[k] = 0;
13              continue;
14          }
15          if ( k > 0 ) {
16              x[k] = x[k] - 1;
17              if ( bound(ks_ins, x, k, ks_sol.v) ) {
18                  continue;
19              } else {
20                  branch(ks_ins, x, k + 1);
21                  v = dotprod(ks_ins.item_value, x, 0,
22                          ks_ins.item_num - 1 );
23                  if ( update_best_sol(ks_ins, &ks_sol, x, v) ) {
24                      mpi_dif_man.whoinf = my_rank;
25                      mpi_nonblocking_broadcast(&(ks_sol.v), 1, MPI_INT,
26                                              NEW_BEST_MSG, MPI_COMM_WORLD, num_procs,
27                                              my_rank );
28                  }
29                  k = ks_ins.item_num - 1;
30              }
31          } else
32              break;
33      }
34      mpi_nonblocking_broadcast(&(ks_sol.v), 1, MPI_INT, END_MSG,
35                              MPI_COMM_WORLD, num_procs, my_rank);
36      /* Prevents deadlock and the race condition. */
37      while ( active_procs_remain(mpi_dif_man) ) {
38          mpi_process_nonblocking(&mpi_dif_man, &ks_sol, ks_ins);
39          mpi_ws_process_nonblocking(&workst, &benchmark);
40      }
41      mpi_free_dif_manager(&mpi_dif_man);
42      mpi_free_workstealing(&workst);
43      (...)

```

Figura 5.12: Implementação MPI com *Workstealing*.

6 AVALIAÇÃO DE DESEMPENHO

Este capítulo dedica-se a medir e avaliar o desempenho da aplicação implementada e discutida nos capítulos anteriores.

Os itens a serem avaliados foram:

1. velocidade de execução;
2. utilização de memória;
3. balanceamento de carga.

Todos os testes foram realizados no *cluster labtec*, do GPPD/UFRGS. Os experimentos foram realizados utilizando 10 UCPs *Pentium III 1.1 GHz* com 1 GiB de memória RAM. A rede de comunicação é *Gigabit Ethernet*.

6.1 Implementação

Para se efetuar as medições, foi criada uma estrutura gerenciadora de *benchmarks*. Certas funções desta estrutura são chamadas quando um evento ocorre e as medições vão sendo registradas. Ao final da execução, tudo é impresso em um arquivo que tem o formato visto na Figura 6.1 e cujo significado é descrito na Tabela 6.1.

```
EXTIME > xxxx
MUSAGE[0] > xxxx
STACK[0] > xxxx
WSSEND[0] > xxxx
WSRECV[0] > xxxx
.
.
.
MUSAGE[<num_proc-1>] > xxxx
STACK[<num_proc-1>] > xxxx
WSSEND[<num_proc-1>] > xxxx
WSRECV[<num_proc-1>] > xxxx
```

Figura 6.1: Formato do arquivo de saída.

Por condição de corrida, não se pode garantir que as medições serão impressas no arquivo na ordem desejada, visto que os tempos relativos dos processos variam. No entanto, o *script Python* construído para processar os dados do *benchmark* faz uma análise léxica do arquivo e obtém os dados corretos.

Item	Significado
EXTIME > xxxx	Programa executado em xxxx segundos
MUSAGE[n] > xxxx	Processo <i>n</i> consumiu xxxx bytes de memória.
STACK[n] > xxxx	Tamanho inicial da pilha de tarefas do processo <i>n</i> em bytes.
WSEND[n] > xxxx	Número de tarefas enviadas por <i>Workstealing</i> no processo <i>n</i> .
WSRECV[n] > xxxx	Número de tarefas recebidas por <i>Workstealing</i> no processo <i>n</i> .

Tabela 6.1: Significado dos arquivos de saída do programa.

A estrutura de *benchmark* deve ser passada como parâmetro em algumas funções, modificando seu protótipo; as modificações introduzidas, no entanto, são mínimas. Considerações mais relevantes sobre esta implementação serão feitas quando necessário. A interface da estrutura é apresentada na Figura 6.2. As funções do tipo *set_flag* informam se aquele parâmetro deve ser medido, enquanto as do tipo *get* têm função óbvia. `myalloc(...)` tem função específica, abordada mais adiante.

```

struct benchmark_s {
    int EXTIME_FLAG;
    int MEM_FLAG;
    int VERBOSE_FLAG;
    int LOAD_FLAG;
    double start_time;
    int mem_usage;
    int sstack;
    int nwssend;
    int nwsrecv;
};

void init_benchmark(struct benchmark_s *benchmark);
void set_extime_flag_on(struct benchmark_s *benchmark);
void set_mem_flag_on(struct benchmark_s *benchmark);
void set_verbose_flag_on(struct benchmark_s *benchmark);
void set_load_flag_on(struct benchmark_s *benchmark);
void *myalloc(int nvar, int svar, struct benchmark_s *benchmark);
int get_mem_usage(struct benchmark_s *benchmark);
void start_time_count(struct benchmark_s *benchmark);
double get_time_count(struct benchmark_s *benchmark);
int get_stack_size(struct benchmark_s *benchmark);
int get_wssend(struct benchmark_s *benchmark);
int get_wsrecv(struct benchmark_s *benchmark);

```

Figura 6.2: Interface da estrutura de *Benchmark*.

6.2 Elaboração dos Casos de Teste

Os casos de teste foram gerados automaticamente por um *script Python* escrito especialmente para isso. O *script* recebe como parâmetro o número de itens total e fornece, para cada item, um valor de utilidade e um peso. Tal programa produz estes valores de maneira

aleatória, o que introduz um problema: é difícil gerar casos bem balanceados¹ por causa do advento do *bound*; não importa o quão grande seja uma entrada, *bounds* realizados com sucesso no início da execução fazem o tempo de resolução ser muito pequeno².

É conveniente, para a análise de desempenho, que o tempo de resolução seja diretamente proporcional ao tamanho da entrada do algoritmo sempre e não apenas no pior caso. Baseado nisso, as seguintes medidas são tomadas para obter o cenário da avaliação:

- Desabilitar o *bound* no programa sendo testado;
- O gerador de casos de teste gera itens com o peso máximo equivalente a metade da capacidade da mochila para, heurísticamente, tentar usar mais itens distintos na solução encontrada;
- O gerador de casos de teste gera utilidades diversas de 1 a 1000 para os itens, para que, heurísticamente, o espaço do valor de utilidades se distribua entre os itens de maneira homogênea (sem muitas ocorrências de itens com o mesmo peso mas utilidades diferentes), visando reduzir o número de itens “mortos”, quem sempre são preteridos.

Por heurísticas se entendem medidas que produzem o efeito esperado intuitivamente ou em um grande número de execuções, mas que não funcionam sempre.

É importante salientar que desenhar este cenário não é tendencioso; o que se faz é uniformizar o número de operações realizadas com o tamanho da entrada para obter um comportamento pessimista e com isso *evidenciar* a efetividade (ou não) dos mecanismos implementados.

Para determinar um bom espaço de testes, foram feitos alguns testes preliminares, com número, valores e pesos de elementos ajustados empiricamente e 10 UCPs. Em 10000 e 15000 encontrou-se um limite superior para o número de itens, graças à complexidade exponencial do problema. O primeiro consumiu 7 horas de processamento do *cluster* e o segundo, 2 dias³. Com 2500 elementos, porém, o tempo de resolução foi de ~ 20 segundos para 10 processos e ~ 120 segundos para um processo. Como deve-se determinar uma média dos tempos de execução via múltiplas execuções (vide Seção 6.3), este tempo mostrou-se adequado ao número de testes realizados, elegendo-se este como o número de elementos máximo empregado no teste.

Vale destacar que, como o valor total deve ser maximizado, sem estar sujeito a restrições, os valores individuais de cada item não tem relevância sobre o número de conjuntos solução gerados. Ao contrário, o peso desses itens em relação ao peso da mochila determina o número de galhos da árvore de possibilidades. Logo, é interessante se fixar uma capacidade máxima da mochila de tal maneira que os pesos representem um espaço de variedade grande, gerando conjuntos-solução diversificados. Através de alguns testes com poucos elementos, chegou-se ao número fixo 5000 para a capacidade da mochila e um valor de peso máximo igual à metade disso, de modo a gerar, no máximo, 2500 valores de pesos diferentes possíveis, o que, no pior caso (para os 2500 elementos postulados

¹Um algoritmo que faça isso é, provavelmente, mais custoso computacionalmente do que resolver o Problema da Mochila! Isso decorre de que, para verificar se um caso possui uma árvore bem-equilibrada, é preciso executar n vezes um programa que resolva o Problema da Mochila.

²Isto *não* diminui a complexidade do problema; no pior caso o *bound* é inefetivo.

³Muito embora a política do cluster no que se refere a *walltime* não permita este período, resultados parciais foram guardados (estado das pilhas) e, após, a execução é retomada, carregando as pilhas armazenadas ao invés de refazer a distribuição cíclica.

anteriormente), geram todos os itens de pesos diferentes. Em todas as execuções realizadas foram usados de 1 a 10 UCPs no processamento do problema, oferecidos pelo *cluster labtec*.

O número de elementos iniciais escolhidos, empiricamente, foram 15, 50, 100, 1000, 2500, que apresentaram tempos mensuráveis de execução (a complexidade exponencial faz com que cada elemento, ao ser elevado, aumente bastante o tempo de processamento). Note-se que, na prática, a complexidade do algoritmo nunca atinge o valor pessimista apresentado ($O((C/\min\{p_1, \dots, p_n\})^n)$), pois muitos elementos, de valor relativo muito baixo jamais são considerados para entrarem na mochila. Elevar a capacidade da mochila para considerar estes elementos é redundante, pois o limite superior de elementos processados em tempo tratável seria apenas deslocado.

Para calcular-se os fatores medidos, toma-se a média aritmética das grandezas, visando uniformidade da medição. No entanto, a média só é confiável quando a análise do desvio padrão é feita; se os valores, a cada execução, forem muito distantes da média, ela não representará o tempo de execução do caso normal, pois pode ser o reflexo de uma deturpação muito grande em alguns dos dados. Para fins de execução deste programa, o seguinte algoritmo é adotado, visando determinar o número de execuções necessárias para a obtenção de uma média confiável:

1. Escolher uma constante N (e.g, 20), que representa o número inicial de execuções;
2. Elege-se um número ε (e.g, 0.01), que representa a margem de distância desejada em relação à média;
3. Roda-se N testes, calculando a média ($\bar{x}$) e o desvio padrão (s) destes;
4. Se $s < \varepsilon \times \bar{x}$, então acabou. Caso contrário faz-se $N \leftarrow N + k$ (e.g, $k = 10$) e volta-se a 3.

Após algumas execuções, discutidas no Apêndice A, montou-se os casos de teste com

- 30 execuções;
- conjuntos de 1000, 1500, 1800, 2000 e 2500 elementos; e
- número de UCPs utilizadas variando de 1 a 10.

6.3 Velocidade de Execução

Esta seção dedica-se a avaliar três aspectos da velocidade de execução do programa apresentado: a análise do tempo de execução, o *speedup* e a eficiência.

6.3.1 Análise do Tempo de Execução

A Figura 6.3 mostra os tempos de execução em contraste com o número de elementos processados, para execuções de 1 a 10 processos. É interessante observar que com 1800 elementos o comportamento sofre uma leve distorção, seja qual for o número de processos. Como foram procedidas várias execuções, é evidente que não se trata de uma anomalia isolada. Surge a hipótese de que o advento da aplicação do *Workstealing* tenha surtido tal efeito, visto que o roubo de tarefas não tem um comportamento constante, pois é modificado por uma condição de corrida entre a execução dos processos. Um fator mais

forte, no entanto, para esta distorção, seria o caráter aleatório do gerador de casos de teste; mesmo com o *bound* desabilitado, é possível que os valores relativos se distribuam de tal forma que a capacidade da mochila seja exaurida facilmente logo no início do algoritmo, gerando uma árvore degenerada, com um conjunto de possíveis soluções muito menor a ser gerado. Contudo, como, ao aumentar o número de processos, a anomalia se torna mais evidente, não se pode descartar a influência do *workstealing*, pois o aumento do número de processos leva a tamanhos iniciais de pilha menores, mais mensagens do protocolo de roubo de tarefas trocadas e, por consequência, maior impacto no desempenho.

O ganho de desempenho obtido ao aumentar-se o número de processos é progressivamente menor, graças ao custo de comunicação que cresce; esse valor tende a convergir para um valor ótimo a partir do qual o aumento do número de processos é inefetivo. Tal fato é evidenciado pela Figura 6.4.

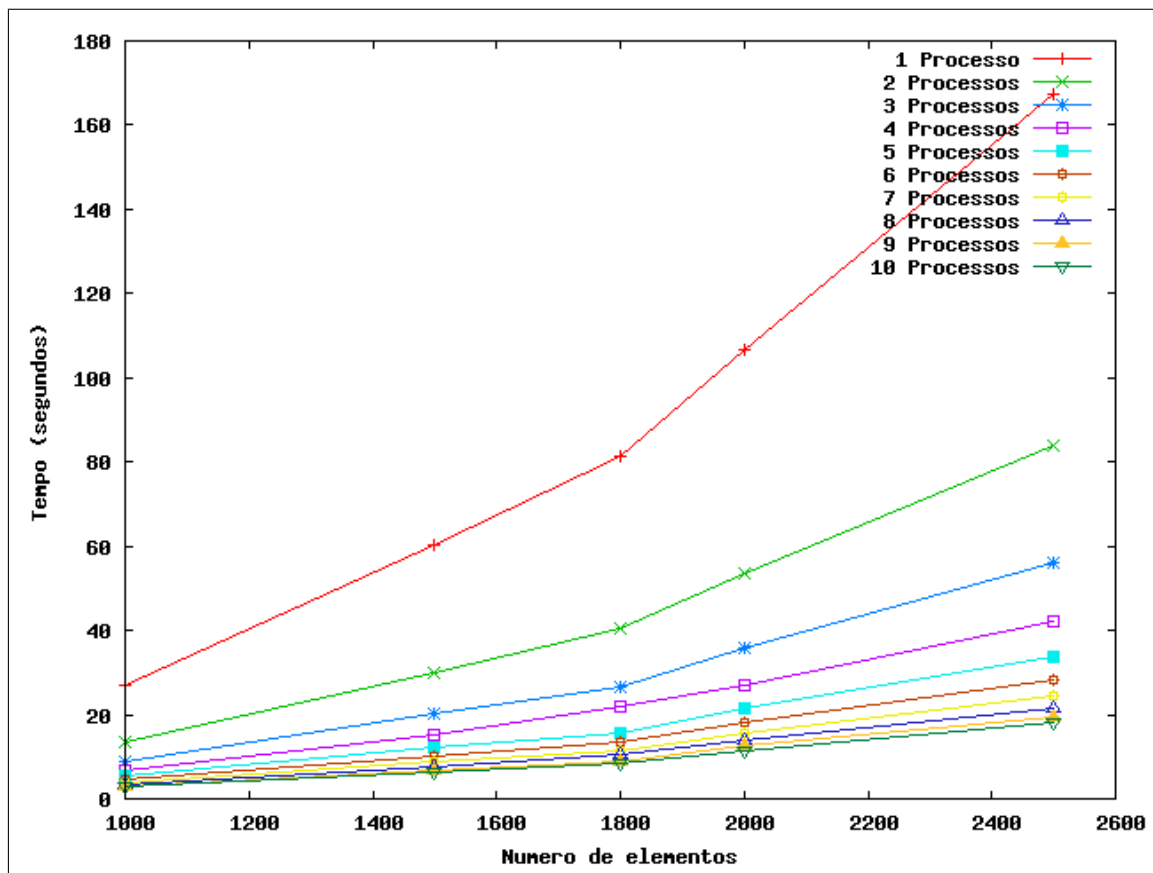


Figura 6.3: Gráfico tempo de execução (número de elementos vs. tempo em segundos).

6.3.2 Speedup

Conforme apresentado anteriormente, o *speedup*, ou seja, o ganho de desempenho relativo ao uso de vários processadores, pode ser calculado por

$$S_p = \frac{T(1)}{T(n)}$$

usando-se n processadores. Numa situação ideal, $S_p = n$.

Por usar-se a função `MPI_Wtime(...)` para o cálculo do tempo e pela necessidade de igualar os ambientes de execução (que é a noção de máquina virtual oferecida pelo

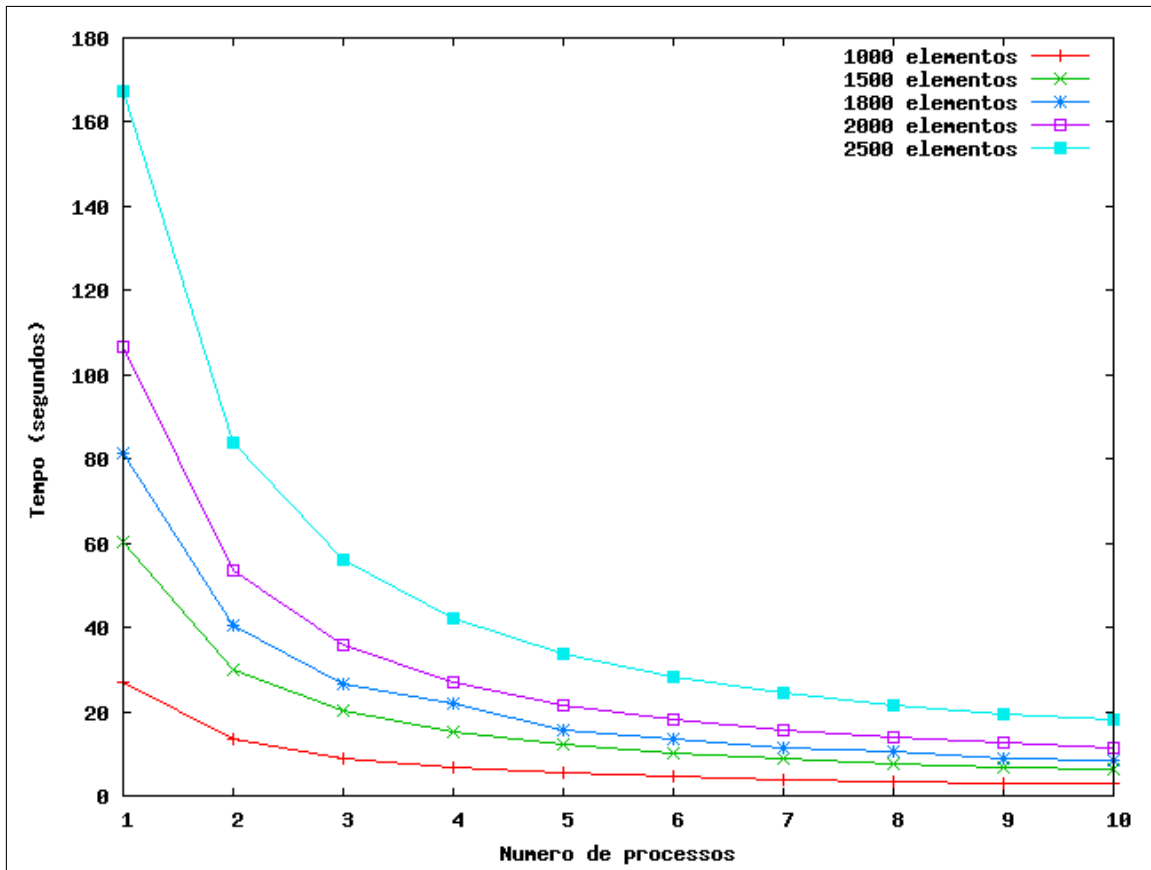


Figura 6.4: Gráfico tempo de execução (número de processos vs. tempo em segundos).

MPI), a versão sequencial é executada em ambiente MPI, mas não faz testes de verificação de *ranking* ou de comunicação. O gráfico correspondente pode ser visto na Figura 6.5.

A imagem que representa o *speedup* agrega pouca informação de desperdício de tempo, pois as retas são na maior parte do tempo constantes, paralelas e sobrepostas, exceto quando o número de processos fica perto de 10, onde começam a se diferenciar, sinal de que a comunicação (incluindo aí o *Workstealing*) começa a pesar (de fato, mais para frente é mostrado que, quanto maior o número de processos, mais tarefas tendem a ser roubadas). Essa constatação mostra que o comportamento do algoritmo proposto se aproxima do ideal.

Pode-se usar o gráfico de *speedup* como um *teste de regressão linear* e verificar que, comparados com a função $\frac{t}{p}$ (t sendo o tempo de execução e p o número de processos), as retas apresentadas aproximam-se da reta que o teste gera para a mesma função. Em outras palavras, o *speedup* evidencia que até que se atinja o valor ótimo no qual não se ganha velocidade, o gráfico do número de processos pelo tempo se comporta como uma parábola e, após, o tempo de execução tende a ser constante.

6.3.3 Eficiência

O conceito de eficiência é dado por:

$$E_p = \frac{T(1)}{T(p) \times p}$$

Para uma entrada de tamanho n e usando-se p processadores. Numa situação ideal,

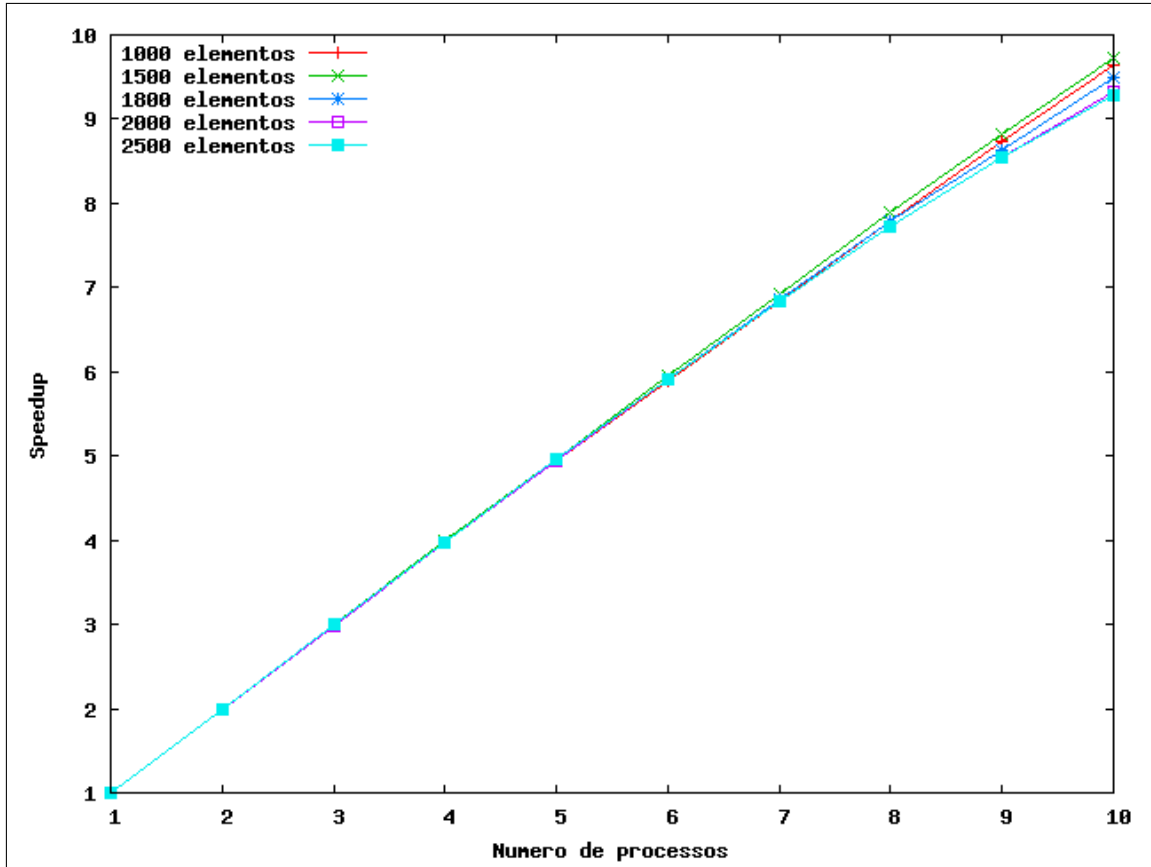


Figura 6.5: Gráfico Speedup.

$E_p = 1$. O gráfico correspondente pode ser observado na Figura 6.6.

A imagem da Eficiência, ao contrário do *speedup*, consegue evidenciar melhor as discrepâncias introduzidas pela comunicação (troca de mensagens). Até um número razoável de processos empregados (~ 5) o decaimento é o esperado, em virtude da comunicação normal da distribuição cíclica da entrada. No entanto, para um número de processos maiores, começa-se a observar distorções fora do padrão, provavelmente introduzidas pelo advento do *Workstealing*. Uma análise mais consistente será feita na Seção 6.5, Balanceamento de Carga.

6.4 Consumo de Memória

O consumo de memória representa parte importante da avaliação de desempenho, pois tem impacto direto na velocidade de execução e no esforço computacional realizado. Além disso, através dessa medição, pode-se estimar se a técnica de *Workstealing*, em troca de balanceamento de carga, não eleva o consumo de memória consideravelmente⁴.

Um programa corretamente escrito libera toda a memória que aloca. Este fator desagrega a capacidade de medição do uso de memória. Além disso, não adianta estabelecer um “*checkpoint*” no código, pois a memória é alocada e desalocada em posições diferentes da programação. Para solucionar este dilema, aborda-se o pior caso e avalia-se o

⁴Disso também depende a boa implementação do algoritmo; o mesmo problema pode gerar consumo linear (e.g., cada tarefa é um inteiro, como nesta implementação) ou exponencial (e.g., cada tarefa é um conjunto candidato à solução) da memória.

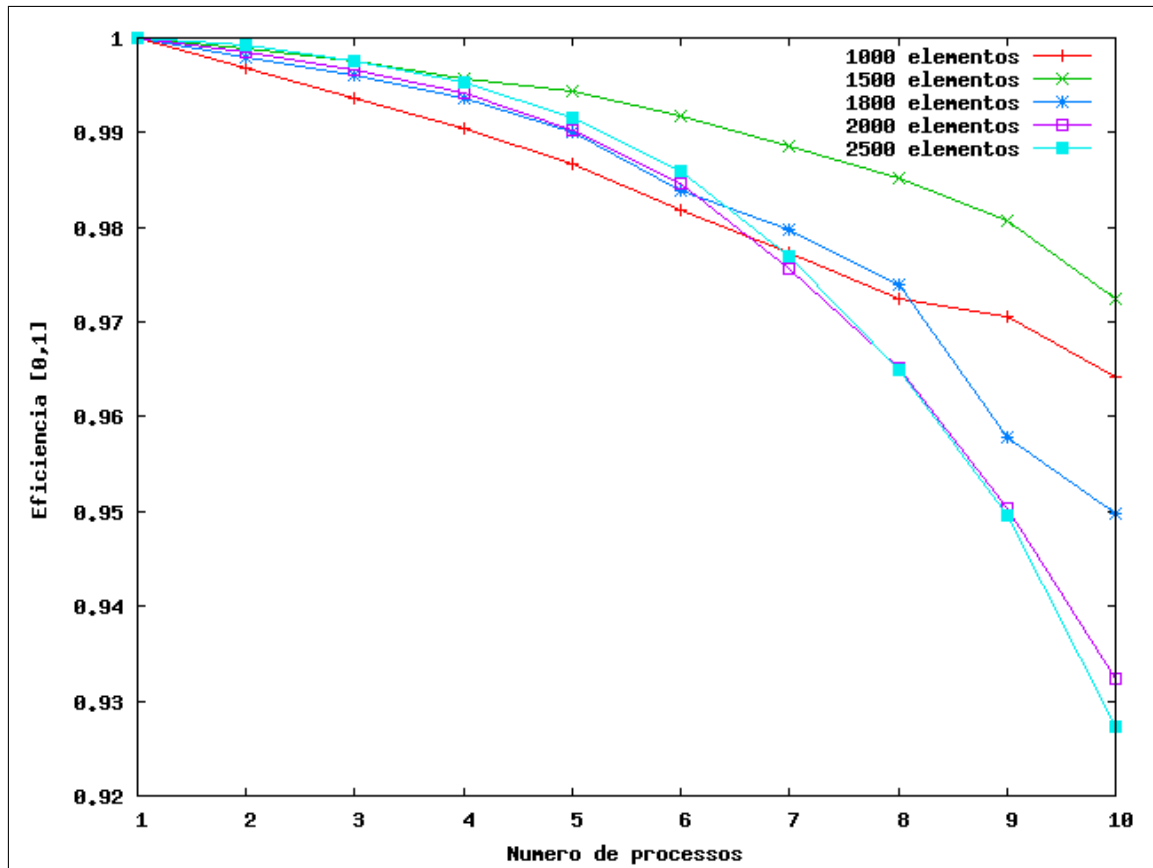


Figura 6.6: Gráfico Eficiência.

somatório de toda a memória alocada do programa.

Para se registrar a memória alocada, na implementação em *C*, construiu-se a primitiva `myalloc(...)`, que substitui `malloc(...)`, registra na estrutura de *benchmark* a quantidade de memória requerida e, somente após, aloca essa memória ao usuário. Ao final da execução, tem-se o somatório de todas as chamadas de alocação de memória realizadas pelo usuário.

Foi verificado, através do *framework* para a supervisão de execução de programas *Valgrind* (NETHERCOTE; SEWARD, 2003), que não existe memória não-liberada ao final da execução do programa e, portanto, não compromete a máquina em que está sendo executado.

É importante observar que a memória usada por um processo nem sempre é constante; como a função usada para fazer as medições é uma função do tipo somatório, há certa irregularidade na obtenção do consumo total de memória por culpa do *Workstealing*. Quando se rouba tarefas, mais memória é alocada em função das tarefas recebidas e isso desbalanceia o somatório. No entanto, dado o tamanho restrito das mensagens, essa diferença é pequena (mas, como já mencionado anteriormente, aumenta com o aumento do número de processos). Naturalmente para a execução com 1 processo o desvio padrão é 0, visto que não há *Workstealing* (a memória ocupada é constante para cada entrada.).

Ao final, a memória consumida por todos os processos é somada. Obtém-se o consumo de memória médio por processo dividindo-se esse somatório pelo número total de processos. Os gráficos que ilustram o consumo de memória (média por processo) são vistos nas Figuras 6.7 e 6.8.

É evidente que quanto mais processos, mais se divide a entrada dentre as pilhas de

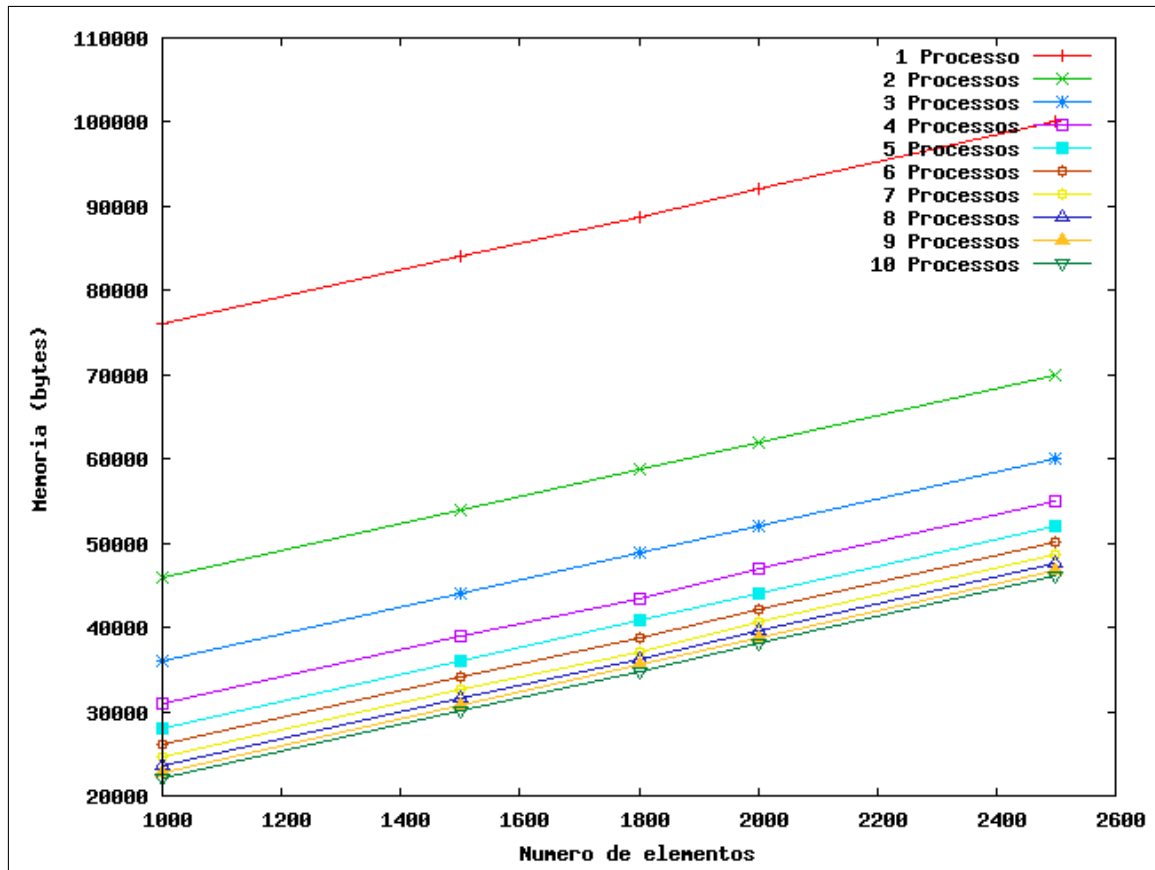


Figura 6.7: Gráfico Consumo de Memória (número de elementos vs. memória em *bytes*).

tarefas e o consumo de memória diminui. Pode-se notar, por observação, que o empilhamento das tarefas por *Workstealing* não tem contribuição significativa, para poucos processos. Isso, pela análise das estruturas de dados, é normal; como há uma distribuição cíclica da entrada (que contém centenas de elementos) no início, a troca de alguns inteiros tem influência pequena sobre o somatório final, a não ser que sejam muitas trocas, que ocorrem mais frequentemente com o aumento do número de processos. A Figura 6.7 mostra que o aumento de consumo de memória é linear; ao se adicionar um elemento, as estruturas de dados empregadas crescem proporcionalmente, resultando numa reta dentro da comparação gráfica. A Figura 6.8 mostra o mesmo cenário, pela óptica do número de processos utilizado em função da memória alocada.

6.5 Balanceamento de Carga

Para medir a efetividade do balanceamento de carga é importante, antes, relembrar que, diferentemente do WS original (BLUMOFE; LEISERSON, 1994), aqui se faz uma distribuição cíclica da entrada antes da execução do algoritmo, para otimizá-lo. Isso implica num impacto menor da adoção da técnica no algoritmo. Além disso, as pilhas entre os processos sempre serão carregadas com $\lfloor \frac{e}{p} \rfloor$ ou $\lceil \frac{e}{p} \rceil$ (onde p é o número de processos e e o número de elementos da entrada).

As medições são feitas quanto ao número de tarefas efetivamente recebidas e efetivamente enviadas⁵; requisições não são levadas em conta, pois pode ser que não sejam

⁵Este é um número total de mensagens, e não a média. Logo, não faz sentido falar em desvio-padrão,

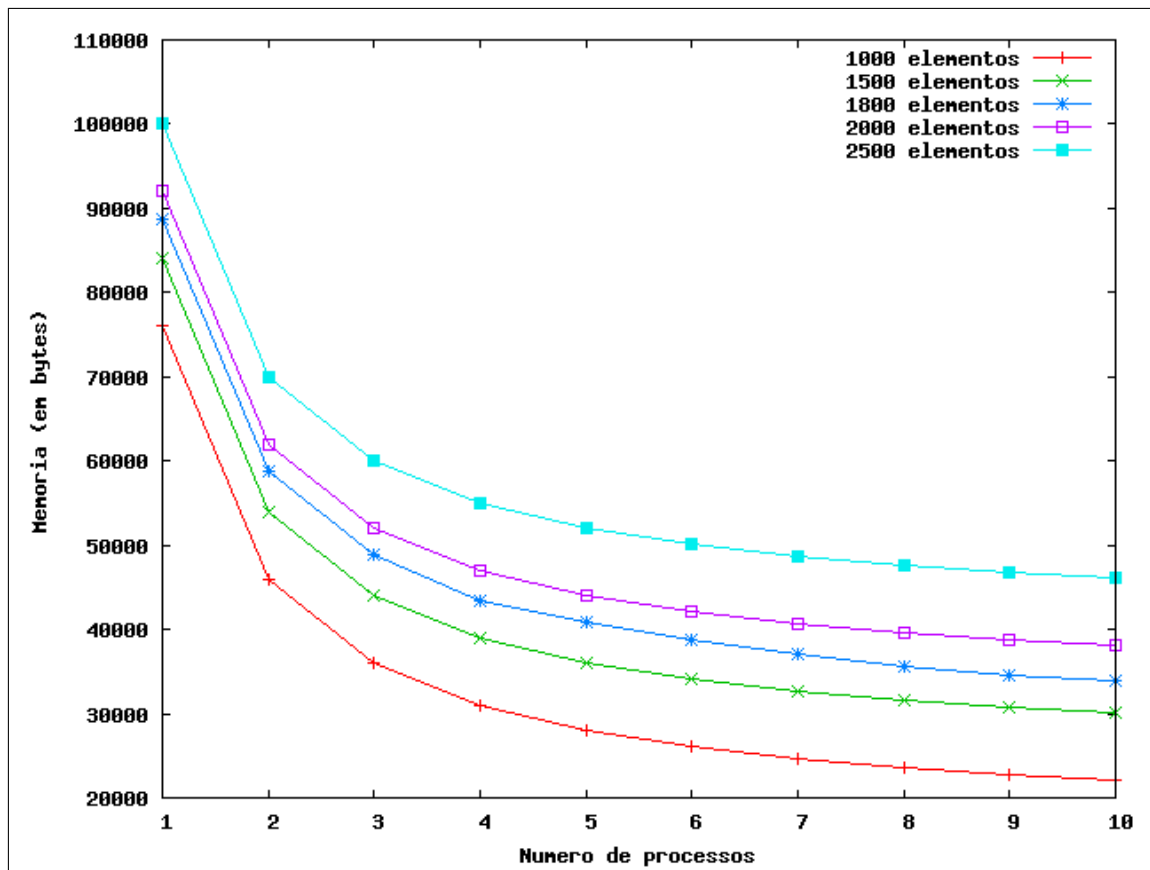


Figura 6.8: Gráfico Consumo de Memória (número de processos vs. memória em bytes).

atendidas. Devido às requisições não atendidas, o tempo de obtenção de uma tarefa pode variar pois, tanto o primeiro quanto o último processo tentado pode possuir as tarefas, sendo impossível medir o tempo associado à obtenção da tarefa, mesmo em uma rede ideal.

Ao invés de medir as requisições não atendidas, adota-se uma simplificação matemática para o problema: pode-se mensurar o caso médio do atendimento de uma solicitação (atendida com $\frac{p-1}{2}$ tentativas) ou o pior caso da mesma situação (atendimento em $p-1$ tentativas), de modo a adequar a análise ao grau de pessimismo que se quer empregar. Para fins desta monografia, trata-se apenas das tarefas recebidas e enviadas.

É interessante observar a heterogeneidade do comportamento, frente à variação do número de elementos e processos, conforme as Figuras 6.9 e 6.10. Como *sends* e *receives* são complementares, basta avaliar uma das operações e considerá-la em dobro caso queira se avaliar ambas.

Fica claro o não-determinismo implícito da ocorrência de *Workstealing* ao se comparar os gráficos apresentados. Embora não sirvam para se traçar uma função, é importante ressaltar que eles não tem utilidade nula; dadas alterações não esperadas nas medições de desempenho e/ou consumo de memória, é possível, usando os dados apresentados, ter evidências de que as alterações são provocadas pela ocorrência de atividades de roubo de tarefas, naquele contexto. Por exemplo, ao observar a Figura 6.6, Eficiência, observa-se que, para 1800 elementos, na transição do processamento com 8 para o processamento com 9 processadores, há uma queda abrupta de eficiência, que foge ao padrão teórico.

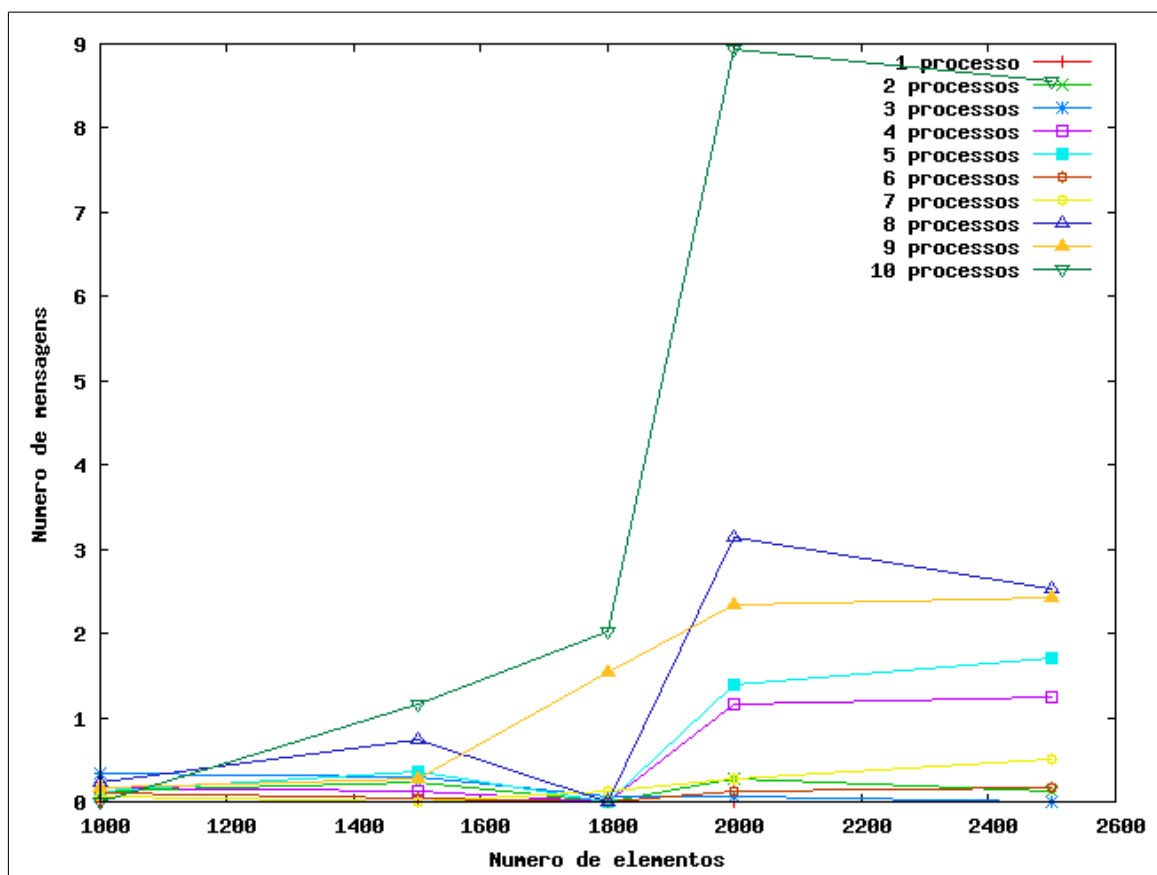


Figura 6.9: Gráfico mensagens enviadas/recebidas (num. de elementos vs. num. de mensagens).

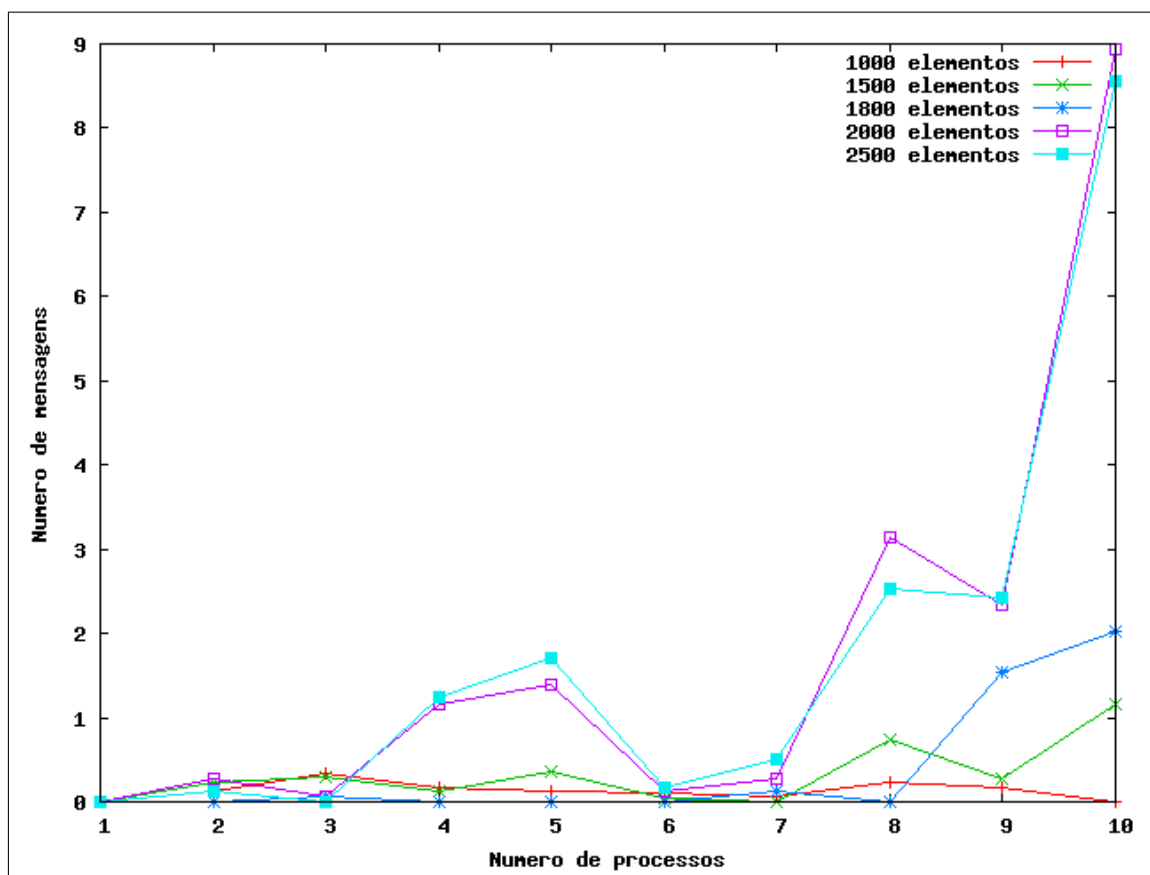


Figura 6.10: Gráfico mensagens enviadas/recebidas (num. de processos vs. num. de mensagens).

Ao verificar a Figura 6.10, Mensagens Enviadas/Recebidas vemos que, com 9 processos houve um acréscimo considerável de troca de mensagens por causa do *Workstealing* implementado, justificando o comportamento observado.

É importante salientar que as observações mostradas são válidas apenas para o *Workstealing* como foi implementado aqui. A técnica original (BLUMOFÉ; LEISERSON, 1994) difere em vários aspectos dessa versão e pode não responder da mesma maneira ao testes realizados.

6.6 Impacto de Utilização do *Workstealing*

Até agora, todas as medições apresentadas e análises se valiam do programa implementado com os recursos de WS habilitados e funcionando. No entanto, uma das propostas da monografia é responder se o *Workstealing* traz ganho real de desempenho ao buscar o balanceamento de carga em um ambiente paralelo. Além disso, caso este benefício seja percebido é necessário, também, determinar o quanto de memória pode ter sido sacrificado em prol do aumento de desempenho.

Conforme já mencionado, a ocorrência de roubo de tarefas aumenta com o aumento do número de processos. Além disso, um cenário com bastantes elementos também oferece uma maior chance de que aconteçam roubos. Desta maneira, os testes foram feitos com 2500 elementos e variando-se de 1 a 10 processos, o máximo de elementos e processos das medições anteriores. Além disso, foi calculada a média de execução que satisfaz um desvio padrão menor ou igual que 0.01, obtida com 30 execuções. O resultado das comparações pode ser observado nas Figuras 6.11 (tempo de execução) e 6.12 (consumo de memória).

Observando as Figuras, é fácil perceber que a introdução de *Workstealing* traz um grande ganho de desempenho para um número elevado de processos, visto que o tempo de execução passou de ~ 80 segundos para ~ 20 segundos com 10 UCPs, um aumento de velocidade de aproximadamente 25%, sem, no entanto, apresentar diferença significativa no consumo de memória.

Vale destacar que o ganho obtido vale para essa implementação do Problema da Mo-chila; não é garantido que a técnica traga o mesmo ganho quando aplicadas em outros problemas; fatores como o encapsulamento da noção de tarefas, latência da rede e natureza do problema podem afetar este resultado.

Devido ao emprego de *Workstealing*, os tempo de execução, com carga balanceada, são menos suscetíveis a atrasos de uma UCP, pois o trabalho desta, no meio tempo, é distribuído entre as outras, enquanto na abordagem convencional as UCPs que já acabaram o trabalho ficam ociosas.

6.7 *Deadlock*

Para validar todas as medições encontradas, é importante provar a corretude do programa, em especial a ausência de *deadlocks*. Lembrando, existem 3 condições básicas para a ocorrência de *deadlock* (TOSCANI; OLIVEIRA; SILVA CARÍSSIMI, 2003):

1. quando um processo que já possui um recurso pode requisitar outro recurso;
2. quando os recursos alocados a um processo não podem ser preemptados;
3. quando é possível a formação de um ciclo (cadeia fechada) no qual cada processo

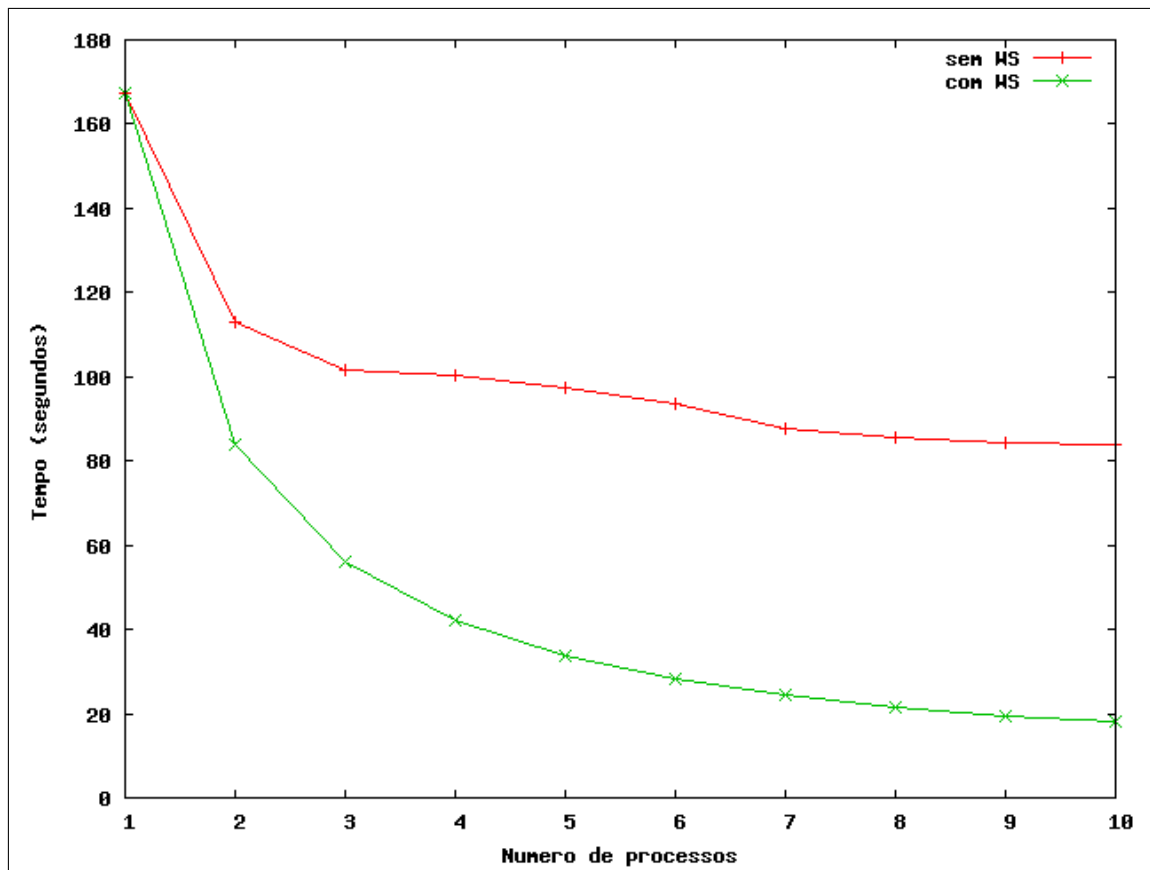


Figura 6.11: Impacto do WS: tempo de execução.

está bloqueado à espera de recursos que estão alocados para outros processos do mesmo ciclo.

Inexistindo uma destas condições, garante-se que não ocorrer *deadlock*. No programa que resolve o Problema da Mochila (versão final), existem duas partes passíveis da ocorrência da trava: a *Difusão de Máximo Local* e o *Workstealing*. A prova formal de que estes trechos não apresentam *deadlock* é apresentada a seguir.

Fica claro, também, a capacidade de adaptação que o algoritmo tem em ambientes heterogêneos; quanto maior a discrepância entre a eficiência dos processadores de uma máquina paralela, mais *Workstealing* tende a ser realizado, visto que os processos possuem velocidades de esvaziamento da pilha diferentes e os processos mais lentos tendem a doar tarefas a processos mais rápidos.

6.7.1 Difusão de Máximo Local

Neste algoritmo, como foi implementado, a única chance de um processo se bloquear é na barreira, ao final do programa; ao longo da execução não há dependência alguma do recebimento de mensagens para a computação; a barreira é o único ponto onde um processo fica bloqueado.

Pode-se entender o bloqueio do processo como uma entrada em um estado de espera do qual só é liberado (sendo p o número de processos) ao receber os $p - 1$ *tokens* informando o fim dos outros processos; neste caso, o recurso a ser consumido são *tokens* de final, ao qual cada processo possui, inicialmente, $p - 1$ *tokens* que distribui ao fim de sua computação e espera pelo recebimento de outros $p - 1$ *tokens*, oriundos um de cada processo. A

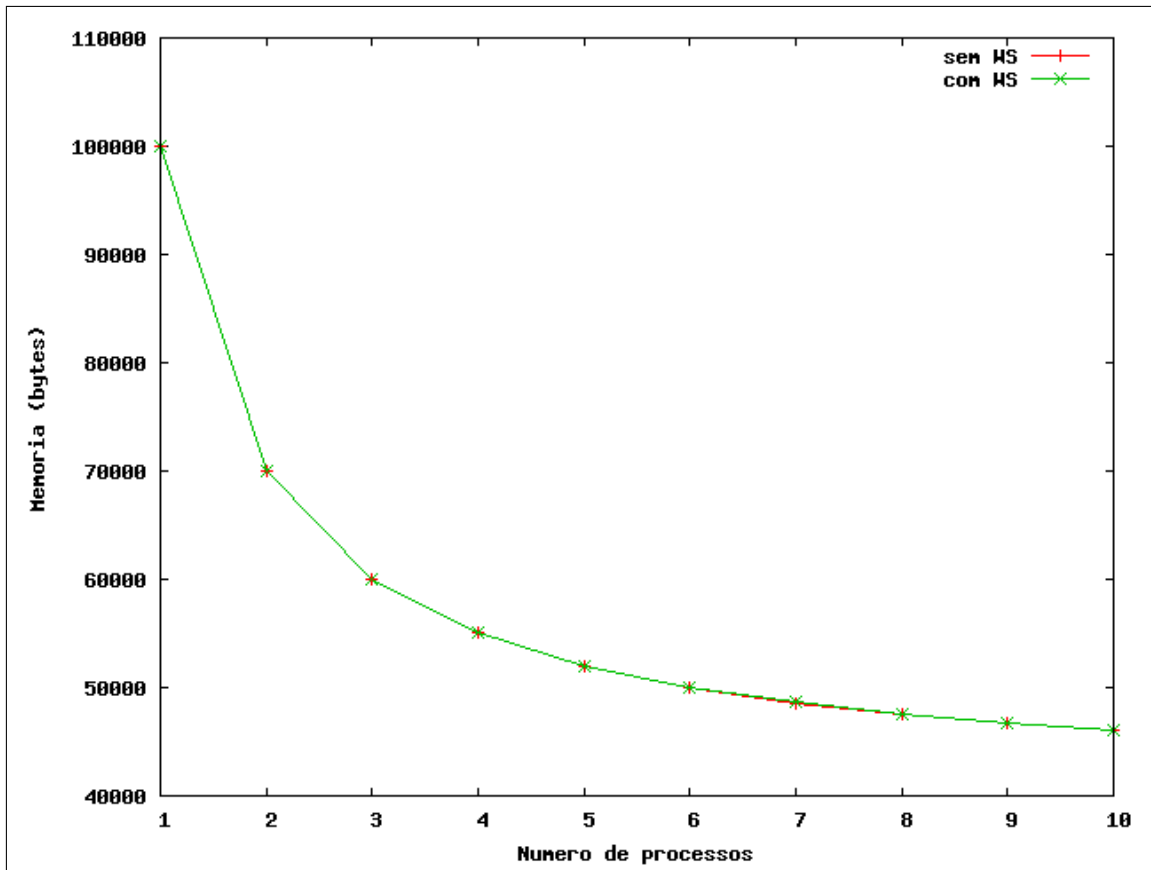


Figura 6.12: Impacto do WS: consumo de memória

prova da não-ocorrência *deadlock* baseia-se no fato de que (1) não procede e é feita por absurdo:

Teorema (Algoritmo DML $\times$ *Deadlock*). *O algoritmo de Difusão de Máximo Local implementado não entra em deadlock.*

Prova. (absurdo). Sejam um processo p_i e um número de processos n tal que $i \in \{1, \dots, n\}$. Supõe-se que (1) é válido:

(1) é válido	→
$\exists p_i \mid (p_i \text{ possui token não-enviado}) \wedge (p_i \text{ espera por tokens de outros processos})$	→
p_i possui token não-enviado	→
p_i não acabou a computação.	

no entanto,

(1) é válido	→
$\exists p_i \mid (p_i \text{ possui token não-enviado}) \wedge (p_i \text{ espera por tokens de outros processos})$	→
p_i espera por tokens de outros processos	→
p_i acabou a computação.	

o que é um absurdo; logo, (1) deve se falso. □

6.7.2 Workstealing

O algoritmo de *Workstealing* tem dois pontos críticos, que podem originar *deadlock*: a barreira ao final do programa e a solicitação de uma tarefa. Muito embora o programa

possa se bloquear no primeiro caso, a barreira não depende de recursos da parte de *Workstealing*, mas sim exclusivamente dos recursos do algoritmo DML; a prova de que esta parte não entra em trava mortal foi apresentada no Teorema “Algoritmo DML $\times$ *Deadlock*”.

Resta provar que, ao solicitar uma tarefa de outro processo, é impossível ocorrer um ciclo entre todos os processos, logo (3) não é válido. Nos casos apresentados a seguir, o recurso a ser consumido é uma tarefa como definida anteriormente, que pode ser empilhada (recurso disponível) ou executada (recurso consumido). É importante lembrar que algumas implicações são válidas pois partem dos códigos já apresentados do programa; em especial, é importante lembrar

1. um processo que está no aguardo de uma solicitação de *Workstealing* pode atender *sempre* requisições de tarefas de outros processos; e
2. há sincronização sobre a pilha de tarefas.

Teorema (Algoritmo WS $\times$ *Deadlock*). *O algoritmo de Workstealing implementado não entra em deadlock.*

Prova. (direta) Quer se provar que (3) é inválido pois um processo pode atender outro mesmo que esteja na esperando por tarefas. Seja um número de processos n e $a \leftarrow b$ significa que “ a está bloqueado, à espera de uma resposta de b ”.

$$\begin{aligned} \forall i \in \{1, \dots, n\} (p_i \leftarrow p_{i+1}) & \longrightarrow \\ (p_1 \leftarrow p_2) \wedge (p_2 \leftarrow p_3) \wedge \dots \wedge (p_{n-1} \leftarrow p_n) \wedge (p_n \leftarrow p_1) & \end{aligned}$$

no entanto, p_1 responde à solicitação de p_n , conforme postulado:

$$\begin{aligned} (p_1 \text{ responde a } p_n) & \longrightarrow \\ \forall i \in \{1, \dots, n\} (p_i \text{ é desbloqueado}) & \longrightarrow \\ (3) \text{ não ocorre.} & \end{aligned}$$

□

7 CONCLUSÕES

Este capítulo apresenta conclusões construídas sobre todo o processo definido ao longo desta monografia, baseado nos capítulos anteriores.

7.1 Problema da Mochila $\times$ Paralelização

Conforme referido no Capítulo 4, o Problema da Mochila é um problema *NP-Completo*, o que significa que possui complexidade exponencial quando processado por uma máquina não-determinística. O advento da paralelização consegue diminuir o tempo de execução do algoritmo. No entanto, tal advento *não* reduz a complexidade do problema; a curva de crescimento exponencial persiste para soluções paralelas do problema.

A paralelização, no entanto, mostra-se especialmente relevante na execução prática do problema. Relativamente ao número de elementos apresentados, o ganho no tempo de execução foi grande para o escopo experimentado.

O resultado mais importante obtido nesta área, no entanto, foi a constatação de que *é possível* paralelizar o Problema da Mochila e esta solução paralela se beneficia do emprego de WS, verificando a hipótese apresentada no Capítulo 1.

7.2 Implementação

Um dos pontos a serem verificados é a possibilidade de se construir uma biblioteca que implemente o *Workstealing* de maneira transparente ao usuário; por ser um algoritmo que depende fortemente das estruturas de dados sobre as quais atua, existe a possibilidade de que não se possa construir uma biblioteca robusta e portátil simultaneamente.

Verifica-se que a construção da biblioteca é viável e, ao longo da monografia, propõem-se estruturas de dados e interfaces para essas estruturas genéricas. Tais construções conseguiram mostrar, na implementação do código:

Encapsulamento. As funções a serem chamadas englobam detalhes de protocolos que não são interessantes aos usuários, permitindo que este se foque na resolução do problema a que se propõe.

Portabilidade. Como usa apenas bibliotecas-padrão da linguagem C e MPI, a solução é portátil para a maioria dos sistemas que ofereçam suporte a ambos os padrões.

No entanto, a implementação ainda não está em estágio satisfatório para a montagem de uma biblioteca autônoma; existem fortes dependências das estruturas em relação à paralelização específica do Problema da Mochila. Tais dependências impedem que a abordagem seja genérica o suficiente em relação aos tipos de problema utilizados.

A solução para o problema da dependência das estruturas de dados foi prototipada apenas, pois sua implementação foge ao escopo da monografia. Para atender ao problema, utiliza-se de estruturas de dados (*e.g.*, pilha de tarefas) e funções-chave (interfaces para as estruturas) que necessitam ser reescritas pelo usuário e fornecidas como parâmetros para certas subrotinas que atendem ao protocolo. Tal abordagem é conhecida como “utilização de *callbacks*” e é especialmente popular no contexto de linguagens orientadas a objetos. A biblioteca padrão de C (`stdlib.h`) usa a mesma abordagem para prover funções de ordenamento genéricas.

7.3 Desempenho

Uma das questões-chave do trabalho desenvolvido é o impacto do uso da técnica de *Workstealing* no desempenho de um programa paralelo. Objetiva-se verificar se a adoção do método impacta num ganho (pelo balanceamento de carga) ou num custo (pelos custos de comunicação) adicional ao programa paralelo sendo executado.

O capítulo 6 mostrou que há uma ganho perceptível de desempenho ao se utilizar a técnica de WS nesta implementação. Este não é um resultado absoluto, pois depende, também, da eficiência com que as estruturas de dados de pilhas de tarefas é gerenciada e das características das máquinas que executam o programa. Em outras palavras, esse ganho *não* pode ser generalizado para qualquer aplicação que, porventura, utilize uma biblioteca genérica baseada no algoritmo apresentado, muito embora um largo contingente de aplicações possa se beneficiar disto.

O trabalho é desenvolvido sobre máquinas homogêneas (*clusters* homogêneos). Essa fato é importante, como evidencia a seguinte linha de pensamento:

1. o *Workstealing* clássico tem um alto custo de comunicação no início, tendo desempenho inferior ao obtido ao se fazer uma simples decomposição cíclica da entrada;
2. o *Workstealing* sobre a decomposição cíclica (conforme apresentado aqui) é efetivo quando existe disparidade entre as velocidades de processamento dos nós, pois processos ociosos (mais rápidos) roubam carga dos processos cheios de tarefas (mais lentos), o que incrementa a velocidade global de resolução do problema.

O último item implica que quanto mais diferença houver entre as velocidades dos vários processadores, mais o efeito do *Workstealing* será proveitoso. Para um *cluster* homogêneo, a velocidade de processamento de pilhas do mesmo tamanho tende a ser a mesma, deixando um número pequeno de tempo ocioso e, ainda, tendo de arcar com o custo de comunicação de requisições não atendidas, fazendo do WS, apesar de efetivo, uma técnica que não desenvolve todo o seu potencial.

Grids e *clusters* heterogêneos, por outro lado, permitem que a técnica atinja seu ápice de eficiência (mesmo que isto implique, no caso de *grids*, um *perda* de desempenho não-desejável para o usuário); existe bastante diferença entre a capacidade de cada nó e isso provoca a ociosidade latente de aplicações paralelas, sanada adequadamente pelo *Workstealing* clássico.

Vale lembrar que implementação apresentada, não oferece um suporte consistente para a execução em *Grids*, conforme discutido na próxima seção.

7.4 WS Clássico × Implementação

Conforme referido, o algoritmo implementado difere do algoritmo clássico do WS, pois, na implementação apresentada, em especial, *há distribuição cíclica da entrada*.

Tal medida é implementada para fins de desempenho, visto que a execução em ambiente MPI (em arquiteturas do tipo *cluster*) possui uma replicação natural de processos e entradas que evita o grande número de solicitações iniciais feitas para o processo que possui todas as tarefas, no algoritmo clássico.

Tal ganho, entretanto, desagrega flexibilidade ao modelo original. Embora otimize o desempenho num contexto MPI/*cluster*, tal implementação não é facilmente generalizada para outros tipo de arquiteturas (*e.g.*, *grids*), que tem um caráter mais dinâmico das máquinas e processos que participam da computação e, portanto, não há garantia da replicação de entrada que MPI proporciona. No contexto onde os processos só recebem informações da entrada via troca de mensagens, a implementação apresentada falha (pois se baseia fortemente na replicação da entrada), enquanto o WS clássico funciona sem problemas.

7.5 Escalabilidade

Para um número de elementos elevado, surgem duas questões referentes à escalabilidade da solução quando o problema cresce: a ocorrência de *deadlocks* e o consumo elevado de memória.

Demonstrou-se formalmente, no Capítulo 6, que o algoritmo implementado não entra em *deadlock*. Além disso, o controle de *deadlock* não impacta em redução de desempenho significativa, conforme apresentado. A inexistência de travas mortais comprovada genericamente faz com que o ambiente de execução ganhe heterogeneidade; é altamente provável que o algoritmo se comporte como o esperado independentemente do ambiente sobre o qual executa.

Já o consumo de memória não é inerente à implementação. Por mais que a solução desenvolvida consuma pouca memória e mantenha um patamar aceitável, uma biblioteca genérica não pode gozar da mesma propriedade; a implementação dos *callbacks* de pilha de tarefas, a cargo do usuário, é quem gerencia a quantidade de memória ocupada por cada pilha. É importante destacar que isso exige especial atenção na implementação de um programa *NP-Completo*; como deve haver a geração de todos os conjuntos-solução possíveis (que crescem exponencialmente), é altamente aconselhável que não se empilhem estes conjuntos, sob risco de uso exponencial da memória.

Além de ser escalável (comporta-se como esperado quando o problema aumenta), a solução apresentada também é bem-escalável, ou seja, continua eficiente quando se aumenta o tamanho do problema. Além disso, o algoritmo apresentado tende a ser mais eficiente para tamanhos de problema maiores, onde o percentual de tempo gasto com a comunicação é percentualmente menor.

7.6 Trabalhos Futuros

Alguns trabalhos futuros relacionados a esta monografia incluem

Construção da Biblioteca. O trabalho apresentado mostra um esqueleto e guias para a construção de uma biblioteca genérica e transparente de *Workstealing*. O próximo passo, natural, é a construção desta biblioteca.

Uso de MPI-2. A especificação MPI-2 oferece a criação dinâmica de processos, ou seja, novos processos podem ser criados e alocados a processadores a medida em que surge a necessidade. É interessante investigar até onde esta possibilidade contribui para a realização do WS, sobretudo no caso de *Grids*.

Medição em *Grids* e *Clusters* Heterogêneos. As medições foram realizadas em um *cluster* homogêneo. Seria interessante fornecer dados precisos sobre a performance do algoritmo em um *cluster* heterogêneo ou *grid*, visando comprovar a sua escalabilidade e enumerando as modificações necessárias que o código deve sofrer para suportar este contexto;

Migração de Processos. Seria interessante agregar o algoritmo de *Workstealing* a um escalonador de processos MPI. Para tanto, deve-se trabalhar na construção específica de um gerenciador de escalonamento que consiga abstrair processos como tarefas e, com isso, comutar tarefas através do uso de algoritmos de migração de processos.

APÊNDICE A ANÁLISE DOS CASOS DE TESTE

Este anexo dedica-se a discutir a obtenção dos valores apresentados no Capítulo 6, *Avaliação de Desempenho*, utilizados para a realização dos testes envolvendo tempo de execução, consumo de memória e troca de mensagens. Para todas as medições, $\varepsilon = 0.01$.

A.1 Tempo de Execução

A Figura A.2 ilustra os resultados de tempo de execução para todos os números de elementos (15, 50, 100, 2500), de 1 a 10 processos com . O formato apresentado é visto na Figura A.1 e seu significado na Tabela A.1.

[<num_elem>, <num_proc>] [EXT = xxx] [DPAD = xxx] <state>

Figura A.1: Formato da tabela de resultados do *benchmark*.

Item	Significado
<num_elem>	Número de elementos.
<num_proc>	Número de processadores.
EXT = xxx	Média dos tempos de execução para as N execuções.
DPAD = xxx	Desvio padrão.
<state>	OK, caso $s < \varepsilon \times \bar{x}$. FAIL, caso contrário.

Tabela A.1: Significado dos símbolos da tabela de resultados do *benchmark*.

A Figura A.2 mostra que, para menos do que 1000 elementos e tomando-se $\varepsilon = 0.01$, a média obtida não é satisfatória. Nos casos onde o número de processos é igual a 1, a média, obviamente, é satisfatória *sempre*, visto que ela é igual às suas parcelas e, por isso, o desvio padrão é o menor possível. Aparentemente há certa dificuldade em obter um valor médio satisfatório para um número reduzido de elementos. Para verificar isso, segue-se o algoritmo proposto no Capítulo 6 para obtenção da média, define-se o valor de $k = 20$ e se faz uma nova bateria de testes, com $N = 30$ (Figura)

A situação torna a não convergir para poucos elementos. Pode-se traçar duas possibilidades para tal ocorrência:

1. Para casos muito pequenos, o tempo de execução também é muito pequeno. Logo, o poder de representação da arquitetura do processador não consegue representar os números decimais que representam os desvios-padrão, muito baixos;

1	[15,1]	[EXT = 0.0107576]	[DPAD = 0.000339238100586]	FAIL
2	[15,2]	[EXT = 0.02639]	[DPAD = 0.0109487714887]	FAIL
3	[15,3]	[EXT = 0.0337298]	[DPAD = 0.00142401832073]	FAIL
4	[15,4]	[EXT = 0.0447095]	[DPAD = 0.00205701230213]	FAIL
5	[15,5]	[EXT = 0.044959]	[DPAD = 0.00293363771148]	FAIL
6	[15,6]	[EXT = 0.0528291]	[DPAD = 0.00245342918517]	FAIL
7	[15,7]	[EXT = 0.0609]	[DPAD = 0.00304944956782]	FAIL
8	[15,8]	[EXT = 0.063024]	[DPAD = 0.00272217258005]	FAIL
9	[15,9]	[EXT = 0.0700221]	[DPAD = 0.00449719044269]	FAIL
10	[15,10]	[EXT = 0.0717377]	[DPAD = 0.00858316771426]	FAIL
11	[50,1]	[EXT = 0.0142862]	[DPAD = 5.63674452927e-05]	OK
12	[50,2]	[EXT = 0.0198123]	[DPAD = 0.00259086798197]	FAIL
13	[50,3]	[EXT = 0.029569]	[DPAD = 0.019246385144]	FAIL
14	[50,4]	[EXT = 0.029055]	[DPAD = 0.00157378242178]	FAIL
15	[50,5]	[EXT = 0.0387701]	[DPAD = 0.0193391512077]	FAIL
16	[50,6]	[EXT = 0.0374721]	[DPAD = 0.00132459750281]	FAIL
17	[50,7]	[EXT = 0.0438698]	[DPAD = 0.00128360333091]	FAIL
18	[50,8]	[EXT = 0.0534928]	[DPAD = 0.0140158684497]	FAIL
19	[50,9]	[EXT = 0.053187]	[DPAD = 0.00149727448682]	FAIL
20	[50,10]	[EXT = 0.0631203]	[DPAD = 0.01259240406]	FAIL
21	[100,1]	[EXT = 0.0422086]	[DPAD = 0.00016504558293]	OK
22	[100,2]	[EXT = 0.0342021]	[DPAD = 0.00117051521894]	FAIL
23	[100,3]	[EXT = 0.0339237]	[DPAD = 0.000824536779046]	FAIL
24	[100,4]	[EXT = 0.0364187]	[DPAD = 0.00126314449336]	FAIL
25	[100,5]	[EXT = 0.0418637]	[DPAD = 0.00586649501738]	FAIL
26	[100,6]	[EXT = 0.0439955]	[DPAD = 0.00121264973875]	FAIL
27	[100,7]	[EXT = 0.0498867]	[DPAD = 0.00798741355787]	FAIL
28	[100,8]	[EXT = 0.0534728]	[DPAD = 0.00256856586877]	FAIL
29	[100,9]	[EXT = 0.0591513]	[DPAD = 0.0113798287826]	FAIL
30	[100,10]	[EXT = 0.0632516]	[DPAD = 0.00382172800358]	FAIL
31	[1000,1]	[EXT = 26.849526]	[DPAD = 0.0288982616629]	OK
32	[1000,2]	[EXT = 13.4566571]	[DPAD = 0.0143115247608]	OK
33	[1000,3]	[EXT = 8.9943672]	[DPAD = 0.00888024790306]	OK
34	[1000,4]	[EXT = 6.7585116]	[DPAD = 0.00891155488583]	OK
35	[1000,5]	[EXT = 5.4241147]	[DPAD = 0.00422864740978]	OK
36	[1000,6]	[EXT = 4.5401515]	[DPAD = 0.00968449113249]	OK
37	[1000,7]	[EXT = 3.9027259]	[DPAD = 0.00457775945818]	OK
38	[1000,8]	[EXT = 3.4260675]	[DPAD = 0.00494981492928]	OK
39	[1000,9]	[EXT = 3.0589852]	[DPAD = 0.00351585073676]	OK
40	[1000,10]	[EXT = 2.7664852]	[DPAD = 0.00631425626465]	OK
41	[2500,1]	[EXT = 167.2858947]	[DPAD = 0.0752806144414]	OK
42	[2500,2]	[EXT = 83.717032]	[DPAD = 0.0390441750138]	OK
43	[2500,3]	[EXT = 55.8893633]	[DPAD = 0.0191406907589]	OK
44	[2500,4]	[EXT = 42.01759]	[DPAD = 0.0171415084713]	OK
45	[2500,5]	[EXT = 33.757429]	[DPAD = 0.0236446289749]	OK
46	[2500,6]	[EXT = 28.2815779]	[DPAD = 0.0116703285491]	OK
47	[2500,7]	[EXT = 24.4468126]	[DPAD = 0.0212549473136]	OK
48	[2500,8]	[EXT = 21.6946972]	[DPAD = 0.0450827801559]	OK
49	[2500,9]	[EXT = 19.5886754]	[DPAD = 0.0490940156333]	OK
50	[2500,10]	[EXT = 18.0523394]	[DPAD = 0.0456056258905]	OK

Figura A.2: Medição dos tempos de execução (10 execuções).

1	[15,1]	[EXT = 0.01069306666667]	[DPAD = 0.000203823441739]	FAIL
2	[15,2]	[EXT = 0.02631583333333]	[DPAD = 0.0068566158642]	FAIL
3	[15,3]	[EXT = 0.03243016666667]	[DPAD = 0.00233781246237]	FAIL
4	[15,4]	[EXT = 0.0452368]	[DPAD = 0.00614348694315]	FAIL
5	[15,5]	[EXT = 0.04980123333333]	[DPAD = 0.025044165907]	FAIL
6	[15,6]	[EXT = 0.05191606666667]	[DPAD = 0.00242229953348]	FAIL
7	[15,7]	[EXT = 0.06050003333333]	[DPAD = 0.00228690821979]	FAIL
8	[15,8]	[EXT = 0.06563676666667]	[DPAD = 0.012495231404]	FAIL
9	[15,9]	[EXT = 0.072859]	[DPAD = 0.0213064093255]	FAIL
10	[15,10]	[EXT = 0.06895833333333]	[DPAD = 0.00629709258765]	FAIL
11	[50,1]	[EXT = 0.0142467]	[DPAD = 7.58270036523e-05]	OK
12	[50,2]	[EXT = 0.01944923333333]	[DPAD = 0.00217931858511]	FAIL
13	[50,3]	[EXT = 0.02584766666667]	[DPAD = 0.0111155109623]	FAIL
14	[50,4]	[EXT = 0.02918076666667]	[DPAD = 0.0025677076679]	FAIL
15	[50,5]	[EXT = 0.0359899]	[DPAD = 0.0125894234519]	FAIL
16	[50,6]	[EXT = 0.03926193333333]	[DPAD = 0.00945302219484]	FAIL
17	[50,7]	[EXT = 0.04411163333333]	[DPAD = 0.00212798322011]	FAIL
18	[50,8]	[EXT = 0.05204383333333]	[DPAD = 0.0111664808153]	FAIL
19	[50,9]	[EXT = 0.05560166666667]	[DPAD = 0.00802547821041]	FAIL
20	[50,10]	[EXT = 0.0610661]	[DPAD = 0.0123452958802]	FAIL
21	[100,1]	[EXT = 0.04216723333333]	[DPAD = 0.000132810646467]	OK
22	[100,2]	[EXT = 0.03433446666667]	[DPAD = 0.00133704134858]	FAIL
23	[100,3]	[EXT = 0.03410943333333]	[DPAD = 0.00102807274274]	FAIL
24	[100,4]	[EXT = 0.03673486666667]	[DPAD = 0.00215245599472]	FAIL
25	[100,5]	[EXT = 0.04100606666667]	[DPAD = 0.00399603013289]	FAIL
26	[100,6]	[EXT = 0.04415733333333]	[DPAD = 0.00144612567616]	FAIL
27	[100,7]	[EXT = 0.04790096666667]	[DPAD = 0.00479299221553]	FAIL
28	[100,8]	[EXT = 0.0547305]	[DPAD = 0.0100921236402]	FAIL
29	[100,9]	[EXT = 0.06171566666667]	[DPAD = 0.0141683640146]	FAIL
30	[100,10]	[EXT = 0.0650695]	[DPAD = 0.0140342567454]	FAIL
31	[1000,1]	[EXT = 26.852102]	[DPAD = 0.0288489445812]	OK
32	[1000,2]	[EXT = 13.4561770333]	[DPAD = 0.0146939932812]	OK
33	[1000,3]	[EXT = 8.99220156667]	[DPAD = 0.00827961494725]	OK
34	[1000,4]	[EXT = 6.76308713333]	[DPAD = 0.0134307544759]	OK
35	[1000,5]	[EXT = 5.42681156667]	[DPAD = 0.00856314744591]	OK
36	[1000,6]	[EXT = 4.53885793333]	[DPAD = 0.0117236657087]	OK
37	[1000,7]	[EXT = 3.9043424]	[DPAD = 0.00794709719687]	OK
38	[1000,8]	[EXT = 3.42688143333]	[DPAD = 0.00750067155899]	OK
39	[1000,9]	[EXT = 3.06061966667]	[DPAD = 0.00946464069412]	OK
40	[1000,10]	[EXT = 2.76708153333]	[DPAD = 0.0125237204887]	OK
41	[2500,1]	[EXT = 167.264477267]	[DPAD = 0.0972753907263]	OK
42	[2500,2]	[EXT = 83.7193909]	[DPAD = 0.0391541690631]	OK
43	[2500,3]	[EXT = 55.8895467667]	[DPAD = 0.0250703633823]	OK
44	[2500,4]	[EXT = 42.0201783333]	[DPAD = 0.0156450808144]	OK
45	[2500,5]	[EXT = 33.7555503]	[DPAD = 0.0266859214887]	OK
46	[2500,6]	[EXT = 28.2830196667]	[DPAD = 0.0211015637273]	OK
47	[2500,7]	[EXT = 24.4531655333]	[DPAD = 0.0331389888623]	OK
48	[2500,8]	[EXT = 21.6740607]	[DPAD = 0.0368826845733]	OK
49	[2500,9]	[EXT = 19.5902117]	[DPAD = 0.0419581238801]	OK
50	[2500,10]	[EXT = 18.0564178]	[DPAD = 0.0427533562805]	OK

Figura A.3: Medição dos tempos de execução (30 execuções).

2. O tempo de processamento é muito pequeno frente ao tempo de comunicação; o programa gasta mais tempo na troca de mensagens do que fazendo o processamento em si.

A primeira hipótese pode parecer razoável, mas basta uma olhada nos desvios-padrão apresentados para verificar que ela falha; o menor desvio padrão apresentado ($\sim 7.5827 \times 10^{-5}$) é verificado positivamente. A hipótese, portanto, é falha. A segunda hipótese parece ser a mais razoável; o alto tempo de latência que a troca de mensagens provoca supera o tempo de processamento. Como os tempos de troca de mensagens dependem de uma série de fatores quase aleatórios (carga da rede, ocupação dos *buffers*, etc.), os custos de comunicação também são determinados aleatoriamente, o que faz com que estes números nunca convirjam para uma média¹. Estes custos, de fato, são problemas recorrentes em programação paralela, abordada em vários eventos da área (ESCOLA REGIONAL DE ALTO DESEMPENHO, 2007).

Mediante isto, é útil, para fins de análise, usar mais elementos, afim de confirmar o postulado. Baseado nos dados apresentados, tendo certeza que os valores, para mais de 1000 elementos, convergem, é conveniente escolher mais elementos do que este valor, no caso, 1000, 1500, 1800, 2000, 2500. Para evitar a repetição desnecessária de testes, usam-se, novamente, 30 execuções. Estes elementos, de fato, confirmam a conclusão apresentada, como pode ser visto na Figura A.4.

A.2 Consumo de Memória

A Figura A.5 mostra o consumo de memória no mesmo formato em que é feita a avaliação de desempenho, substituindo EXT (tempo de execução em segundos) por MEM (memória consumida em *bytes*). As medições foram feitas, tal qual anteriormente, respeitando-se a máxima distancia da média dada pelo desvio padrão.

A.3 Balanceamento de Carga

O recebimento e envio de tarefas é um fator aleatório, pois é resultado de uma condição de corrida. Isto pode ser evidenciado pela Figura A.6, onde se mostra o resultado das 30 execuções para 2500 elementos e 10 processos, onde o referencial é o processo de número 10.

De fato, como mencionado anteriormente, o envio e recebimento de tarefas ocorre mais facilmente com o aumento do número de processos. A Figura A.7 mostra a execução do programa para 2500 elementos e 3 processos, tomando o processo 2 como referência.

Dessa maneira, fica difícil precisar corretamente uma função que expresse a taxa de envio ou recebimento de tarefas. Contornos para esta situação são descritos no Capítulo 6.

¹Esta conclusão deve ser válida, mesmo com o *Workstealing* (que implica maiores custos de comunicação) desabilitado, pois há o custo de comunicação MPI natural às aplicações. Mesmo que isso eventualmente permita que uma faixa do número de elementos menor venha a convergir, matematicamente sempre irá existir uma faixa de número de custo elevado de comunicação, que não convergem.

1	[1000,1]	[EXT = 26.8528707333]	[DPAD = 0.0333933707441]	OK
2	[1000,2]	[EXT = 13.4683186667]	[DPAD = 0.0129807776576]	OK
3	[1000,3]	[EXT = 9.00799513333]	[DPAD = 0.00749613369631]	OK
4	[1000,4]	[EXT = 6.77735542857]	[DPAD = 0.00612728575351]	OK
5	[1000,5]	[EXT = 5.44311833333]	[DPAD = 0.00552155552148]	OK
6	[1000,6]	[EXT = 4.55871196429]	[DPAD = 0.00644472073056]	OK
7	[1000,7]	[EXT = 3.92496076667]	[DPAD = 0.00841211594081]	OK
8	[1000,8]	[EXT = 3.45192932143]	[DPAD = 0.00463376159555]	OK
9	[1000,9]	[EXT = 3.07427937931]	[DPAD = 0.00462139043263]	OK
10	[1000,10]	[EXT = 2.7850921]	[DPAD = 0.0132990536696]	OK
11	[1500,1]	[EXT = 60.1102189]	[DPAD = 0.0721571054224]	OK
12	[1500,2]	[EXT = 30.0877856]	[DPAD = 0.0194270428265]	OK
13	[1500,3]	[EXT = 20.0838294333]	[DPAD = 0.0113542488599]	OK
14	[1500,4]	[EXT = 15.0915499524]	[DPAD = 0.0106060842986]	OK
15	[1500,5]	[EXT = 12.0909193846]	[DPAD = 0.00664547562375]	OK
16	[1500,6]	[EXT = 10.1025945714]	[DPAD = 0.00674633604601]	OK
17	[1500,7]	[EXT = 8.68623239286]	[DPAD = 0.00706197531535]	OK
18	[1500,8]	[EXT = 7.62627093333]	[DPAD = 0.0105303161567]	OK
19	[1500,9]	[EXT = 6.81066524138]	[DPAD = 0.0126470038737]	OK
20	[1500,10]	[EXT = 6.18087403448]	[DPAD = 0.0597001885098]	OK
21	[1800,1]	[EXT = 81.2249111333]	[DPAD = 0.0372123445571]	OK
22	[1800,2]	[EXT = 40.6557511]	[DPAD = 0.0159091125928]	OK
23	[1800,3]	[EXT = 26.4657286]	[DPAD = 0.0096779610483]	OK
24	[1800,4]	[EXT = 21.8746560714]	[DPAD = 0.0185950807239]	OK
25	[1800,5]	[EXT = 15.73247146667]	[DPAD = 0.0264851551593]	OK
26	[1800,6]	[EXT = 13.32193596667]	[DPAD = 0.0159201086801]	OK
27	[1800,7]	[EXT = 11.3027701]	[DPAD = 0.0102274948979]	OK
28	[1800,8]	[EXT = 10.5480813]	[DPAD = 0.00928848878609]	OK
29	[1800,9]	[EXT = 9.0141603]	[DPAD = 0.00977744383077]	OK
30	[1800,10]	[EXT = 8.5509957]	[DPAD = 0.0170690271447]	OK
31	[2000,1]	[EXT = 106.5561137]	[DPAD = 0.0926798866441]	OK
32	[2000,2]	[EXT = 53.3542754667]	[DPAD = 0.0320474529672]	OK
33	[2000,3]	[EXT = 35.6413687]	[DPAD = 0.0267787454417]	OK
34	[2000,4]	[EXT = 26.7958899231]	[DPAD = 0.0177569118023]	OK
35	[2000,5]	[EXT = 21.5221352]	[DPAD = 0.0149270678017]	OK
36	[2000,6]	[EXT = 18.0358812667]	[DPAD = 0.0144334347093]	OK
37	[2000,7]	[EXT = 15.6029128667]	[DPAD = 0.0213496455353]	OK
38	[2000,8]	[EXT = 13.7993478333]	[DPAD = 0.0228852701203]	OK
39	[2000,9]	[EXT = 12.4591445667]	[DPAD = 0.0304073834422]	OK
40	[2000,10]	[EXT = 11.4293376667]	[DPAD = 0.0288214095991]	OK
41	[2500,1]	[EXT = 167.294710867]	[DPAD = 0.104966505727]	OK
42	[2500,2]	[EXT = 83.7104580333]	[DPAD = 0.044471359152]	OK
43	[2500,3]	[EXT = 55.8979496667]	[DPAD = 0.0243657349261]	OK
44	[2500,4]	[EXT = 42.02193104]	[DPAD = 0.0173290449507]	OK
45	[2500,5]	[EXT = 33.7411595882]	[DPAD = 0.0180413635638]	OK
46	[2500,6]	[EXT = 28.2778684828]	[DPAD = 0.0136757405541]	OK
47	[2500,7]	[EXT = 24.4634930667]	[DPAD = 0.0312944140158]	OK
48	[2500,8]	[EXT = 21.6711581429]	[DPAD = 0.030941047535]	OK
49	[2500,9]	[EXT = 19.5753810345]	[DPAD = 0.0397428517781]	OK
50	[2500,10]	[EXT = 18.0408034333]	[DPAD = 0.0555629899421]	OK

Figura A.4: Medição dos tempos de execução (30 execuções) – versão final.

1	[1000,1]	[MEM = 76016.0]	[DPAD = 0.0]	OK
2	[1000,2]	[MEM = 46019.6]	[DPAD = 11.5716657588]	OK
3	[1000,3]	[MEM = 36019.6]	[DPAD = 6.41979697976]	OK
4	[1000,4]	[MEM = 31022.0]	[DPAD = 8.18745887065]	OK
5	[1000,5]	[MEM = 28024.0]	[DPAD = 5.75355961782]	OK
6	[1000,6]	[MEM = 26032.8]	[DPAD = 5.39731735105]	OK
7	[1000,7]	[MEM = 24608.4]	[DPAD = 2.190890206]	OK
8	[1000,8]	[MEM = 23543.2]	[DPAD = 4.38178045704]	OK
9	[1000,9]	[MEM = 22710.4]	[DPAD = 4.88205721529]	OK
10	[1000,10]	[MEM = 22045.2]	[DPAD = 6.81984932394]	OK
11	[1500,1]	[MEM = 84016.0]	[DPAD = 0.0]	OK
12	[1500,2]	[MEM = 54016.0]	[DPAD = 11.9308351588]	OK
13	[1500,3]	[MEM = 44018.4]	[DPAD = 5.81081036828]	OK
14	[1500,4]	[MEM = 39016.8]	[DPAD = 3.04449755451]	OK
15	[1500,5]	[MEM = 36024.0]	[DPAD = 7.27774121836]	OK
16	[1500,6]	[MEM = 34034.8]	[DPAD = 5.16219679054]	OK
17	[1500,7]	[MEM = 32614.8]	[DPAD = 7.51274779396]	OK
18	[1500,8]	[MEM = 31541.2]	[DPAD = 12.8717814849]	OK
19	[1500,9]	[MEM = 30713.6]	[DPAD = 6.08899516302]	OK
20	[1500,10]	[MEM = 30056.8]	[DPAD = 19.0650429331]	OK
21	[1800,1]	[MEM = 88816.0]	[DPAD = 0.0]	OK
22	[1800,2]	[MEM = 58811.2]	[DPAD = 5.39731735105]	OK
23	[1800,3]	[MEM = 48816.8]	[DPAD = 6.24996550452]	OK
24	[1800,4]	[MEM = 43321.6]	[DPAD = 6.08899517382]	OK
25	[1800,5]	[MEM = 40822.0]	[DPAD = 4.54858826147]	OK
26	[1800,6]	[MEM = 38831.2]	[DPAD = 5.39731736324]	OK
27	[1800,7]	[MEM = 36133.6]	[DPAD = 9.13349274993]	OK
28	[1800,8]	[MEM = 35598.0]	[DPAD = 7.77263102839]	OK
29	[1800,9]	[MEM = 34178.0]	[DPAD = 6.10257153259]	OK
30	[1800,10]	[MEM = 33857.2]	[DPAD = 19.3451410626]	OK
31	[2000,1]	[MEM = 92016.0]	[DPAD = 0.0]	OK
32	[2000,2]	[MEM = 62011.2]	[DPAD = 5.3973173998]	OK
33	[2000,3]	[MEM = 52022.4]	[DPAD = 8.1773446431]	OK
34	[2000,4]	[MEM = 47017.2]	[DPAD = 4.83093483736]	OK
35	[2000,5]	[MEM = 44038.4]	[DPAD = 21.5432203888]	OK
36	[2000,6]	[MEM = 42061.2]	[DPAD = 8.73518450756]	OK
37	[2000,7]	[MEM = 40653.6]	[DPAD = 14.5782076432]	OK
38	[2000,8]	[MEM = 39589.6]	[DPAD = 47.4171509799]	OK
39	[2000,9]	[MEM = 38761.2]	[DPAD = 18.5591319683]	OK
40	[2000,10]	[MEM = 38103.2]	[DPAD = 71.7881942083]	OK
41	[2500,1]	[MEM = 100016.0]	[DPAD = 0.0]	OK
42	[2500,2]	[MEM = 70010.8]	[DPAD = 5.16219676505]	OK
43	[2500,3]	[MEM = 60022.0]	[DPAD = 6.10257153259]	OK
44	[2500,4]	[MEM = 55019.6]	[DPAD = 9.532521489]	OK
45	[2500,5]	[MEM = 52030.8]	[DPAD = 17.6447706308]	OK
46	[2500,6]	[MEM = 50061.6]	[DPAD = 9.83168697937]	OK
47	[2500,7]	[MEM = 48651.6]	[DPAD = 18.5055443667]	OK
48	[2500,8]	[MEM = 47596.0]	[DPAD = 44.7151906611]	OK
49	[2500,9]	[MEM = 46764.0]	[DPAD = 13.4932933533]	OK
50	[2500,10]	[MEM = 46124.8]	[DPAD = 80.6484067858]	OK

Figura A.5: Consumo de memória (30 execuções).

NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	14
NUM.	ELEM.:	2500	PROC.:	10	SEND:	17	RECEIVED:	1
NUM.	ELEM.:	2500	PROC.:	10	SEND:	9	RECEIVED:	2
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	13
NUM.	ELEM.:	2500	PROC.:	10	SEND:	11	RECEIVED:	3
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	13
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	12
NUM.	ELEM.:	2500	PROC.:	10	SEND:	22	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	14
NUM.	ELEM.:	2500	PROC.:	10	SEND:	13	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	13
NUM.	ELEM.:	2500	PROC.:	10	SEND:	25	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	12
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	14
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	12
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	14
NUM.	ELEM.:	2500	PROC.:	10	SEND:	24	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	13
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	13
NUM.	ELEM.:	2500	PROC.:	10	SEND:	4	RECEIVED:	3
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	14
NUM.	ELEM.:	2500	PROC.:	10	SEND:	0	RECEIVED:	11
NUM.	ELEM.:	2500	PROC.:	10	SEND:	3	RECEIVED:	5
NUM.	ELEM.:	2500	PROC.:	10	SEND:	11	RECEIVED:	2
NUM.	ELEM.:	2500	PROC.:	10	SEND:	14	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	10	SEND:	24	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	10	SEND:	25	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	10	SEND:	15	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	10	SEND:	26	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	10	SEND:	14	RECEIVED:	0

Figura A.6: Tarefas enviadas e recebidas do processo 10 (2500 elementos, 10 processos).

NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	1
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	1
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	1
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	1
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	1
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	1	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	1	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	1	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	1
NUM.	ELEM.:	2500	PROC.:	2	SEND:	1	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	1
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0
NUM.	ELEM.:	2500	PROC.:	2	SEND:	0	RECEIVED:	0

Figura A.7: Tarefas enviadas e recebidas do processo 2 (2500 elementos, 3 processos).

REFERÊNCIAS

BLUMOFÉ, R. D.; LEISERSON, C. E. Scheduling Multithread Computations by Work Stealing. In: ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 35., 1994. **Proceedings...** [S.l.: s.n.], 1994.

BUTENHOF, D. R. **Programming with POSIX Threads**. [S.l.]: Addison-Wesley Professional, 1997.

CERA, M. C.; PEZZI, G. P.; MATHIAS, E. N.; MAILLARD, N.; NAVAUX, P. O. A. Improving the Dynamic Creation of Processes in MPI-2. In: EURO-PVM/MPI, 2006, 2006. **Proceedings...** [S.l.: s.n.], 2006. p.8.

CLÁUDIO, D. M.; MARINS, J. M. **Cálculo Numérico Computacional** : teoria e prática. 2.ed. [S.l.]: Atlas, 1994.

CREEGER, M. Multicore CPUs for the masses. **Queue**, New York, NY, USA, v.3, n.7, p.64–ff, 2005.

DONGARRA, J.; FOSTER, I.; FOX, G.; GROPP, W.; KENNEDY, K.; TORCZON, L.; WHITE, A. **Sourcebook of Parallel Computing**. 1.ed. São Francisco: Morgan Kaufmann Publishers, 2003.

ESCOLA REGIONAL DE ALTO DESEMPENHO, 2007. **Anais...** [S.l.: s.n.], 2007. n.7.

FLYNN, M. J. Some Computer Organization and Their Effectiveness. In: **IEEE Transactions on Computers**. [S.l.: s.n.], 1972. v.C-21, n.9.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability**: a guide to the theory of np-completeness. [S.l.]: W. H. Freeman, 1979. (Series of Books in the Mathematical Sciences).

GEIST, A.; BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAM, V. **PVM**: parallel virtual machine a users' guide and tutorial for networked parallel computing. 1.ed. [S.l.]: MIT Press, 1994. (The Scientific and Engineering Computation series).

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI - Portable Parallel Programming with Message-Passing Interface**. 2.ed. Massachusetts Institute of Technology - Cambridge, Massachusetts 02142: The MIT Press, 1999. (Scientific and Engineering Computation Series).

GROPP, W.; LUSK, E.; THAKUR, R. **Using MPI-2**: advanced features of the message-passing interface. Massachusetts Institute of Technology - Cambridge, Massachusetts 02142: The MIT Press, 1999. (Scientific and Engineering Computation Series).

JAJA, J. **An Introduction to Paralell Algorithms**. [S.l.]: Addison-Wesley, 1992.

KELLER, H.; PFERSCHY, U.; PISINGER, D. **Knapsack Problems**. [S.l.]: Springer, 2005. 546p.

KOELBEL, C. H.; LOVEMAN, D. B.; SCHREIBER, R. S. **The High Performance Fortran Handbook**. [S.l.]: The MIT Press, 1993. (Scientific and Engineering Computation).

MARTELLO, S.; TOTH, P. **Knapsack Problems**: algorithms and computer implementations. [S.l.]: John Wiley & Sons Inc, 1990. (Wiley Interscience Series in Discrete Mathematics and Optimization).

MATTSON, T. G.; HENRY, G. An Overview of the Intel TFLOPS Supercomputer. **Intel Technology Journal**, [S.l.], n.Q1, p.12, 1998.

NAVAUX, P. O. A.; ROSE, C. A. F. D. **Arquiteturas Paralelas**. 1.ed. [S.l.]: Editora Sagra Luzzato, 2003. n.15. (Série Livros Didáticos).

NETHERCOTE, N.; SEWARD, J. Valgrind: a program supervision framework. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.89, n.2, p.23, 2003.

OPENMP C and C++ Application Program Interface. 2.ed. [S.l.]: OpenMP Architecture Review Board, 2002.

PACHECO, P. S. **Paralell Programming With MPI**. 1.ed. [S.l.]: Morgan Kaufmann Publishers, 1997.

PACHECO, P. S. **A User's Guide to MPI**. 1998.

PATTERSON, D. A.; HENNESSY, J. L. **Arquitetura de Computadores**: uma abordagem quantitativa. 3.ed. [S.l.]: Editora Campus, 2003.

PATTERSON, D. A.; HENNESSY, J. L. **Organização e Projeto de Computadores**: a interface hardware software. 3.ed. [S.l.]: Editora Campus, 2005.

PEZZI, G. P.; CERA, M. C.; MATHIAS, E. N.; MAILLARD, N.; NAVAUX, P. O. A. Escalonamento Dinâmico de programas MPI-2 utilizando Divisão e Conquista. In: WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO 2006 - WSCAD, 2006, 2006. **Proceedings...** [S.l.: s.n.], 2006.

ROOSTA, S. H. **Parallel Processing and Parallel Algorithms**: theory and computation. 1.ed. [S.l.]: Springer, 1999.

TANENBAUM, A. S. **Distributed Operating Systems**. [S.l.]: Prentice-Hall, 1995.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2.ed. [S.l.]: Prentice-Hall, 2003.

TOSCANI, L. V.; VELOSO, P. **Complexidade de Algoritmos**. 2.ed. [S.l.]: Editora Sagra Luzzato, 2002. n.13. (Série Livros Didáticos).

TOSCANI, S. S.; OLIVEIRA, R. S. de; SILVA CARÍSSIMI, A. da. **Sistemas Operacionais**. 2.ed. [S.l.]: Editora Sagra Luzzato, 2002. n.11. (Série Livros Didáticos).

TOSCANI, S. S.; OLIVEIRA, R. S. de; SILVA CARÍSSIMI, A. da. **Sistemas Operacionais e Programação Concorrente**. 1.ed. [S.l.]: Editora Sagra Luzzato, 2003. n.14. (Série Livros Didáticos).