UNIVERSIDADE DE BRASÍLIA
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

MODELO HÍBRIDO DE PROGRAMAÇÃO PARALELA PARA UMA APLICAÇÃO DE ELASTICIDADE LINEAR BASEADA NO MÉTODO DOS ELEMENTOS FINITOS

LEONARDO NUNES DA SILVA

Dissertação apresentada como requisito parcial à
obtenção do grau de Mestre em Informática.

Orientador: Prof. Dr. Gerson Henrique Pfitscher.

BRASÍLIA, DEZEMBRO DE 2006

UNIVERSIDADE DE BRASÍLIA
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

MODELO HÍBRIDO DE PROGRAMAÇÃO PARALELA PARA UMA APLICAÇÃO DE ELASTICIDADE LINEAR BASEADA NO MÉTODO DOS ELEMENTOS FINITOS

LEONARDO NUNES DA SILVA

Dissertação aprovada como requisito parcial para a obtenção do grau de Mestre em Informática, pela banca examinadora composta por:

Orientador: Prof. Dr. Gerson Henrique Pfitscher
CIC/UnB

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo
CIC/UnB

Prof.ª Dr.ª Dianne M. Vianna
ENM/UnB

Brasília 07 de Dezembro de 2006

CIP – Catalogação Internacional na Publicação

Leonardo Nunes da Silva

Modelo Híbrido de Programação Paralela para uma Aplicação de Elasticidade Linear Baseada no Método dos Elementos Finitos/Leonardo Nunes da Silva UnB, 2006.

98p. : il; 29,5 cm

Dissertação de Mestrado – Universidade de Brasília, Brasília, 2006

1. Sistemas Distribuídos
2.Programação paralela
3. Método dos Elementos Finitos
4.Memória Compartilhada
5. Troca de Mensagens

AGRADECIMENTOS

À minha família.
Ao prof. Dr. Gerson Pfitscher pela orientação, amizade e dedicação.
À Flávia Romano Villa Verde pelo apoio e pela ajuda com a Engenharia Mecânica e com a Aplicação de Método dos Elementos Finitos.

RESUMO

Na área de processamento paralelo existem dois paradigmas principais de programação: Memória Compartilhada e Troca de Mensagens. Cada um deles é adequado a uma arquitetra de *hardware* específica. No entanto, existem arquiteturas de multiprocessadores para as quais o mapeamento para um desses paradigmas não é tão simples. *Clusters* de SMP, por exemplo, são construídos com máquinas de memória compartilhada, conectadas através de uma rede de interconexão.

Aplicações para *clusters* de SMP podem ser programadas para utilizar troca de mensagens entre todos os processadores. Mas existe a possibilidade de um melhor desempenho se utilizado um modelo híbrido de comunicação com troca de informações por memória compartilhada dentro do nó SMP e troca de informações por mensagens entre os nós.

Nesse trabalho foi desenvolvido e avaliado um modelo híbrido de programação para uma aplicação na área de engenharia mecânica baseada no método dos elementos finitos. O objetivo desse trabalho é avaliar esse modelo e comparar seu desempenho com uma versão pura, por troca de mensagens, da aplicação.

ABSTRACT

In the area of parallel processing there are two major programming paradigms: Shared Memory and Message Passing. Each of them fits into a specific physical model, but there are multiprocessor architectures whose mapping to one of these paradigms is not so simple. SMP clusters, for example, are built by connecting some shared memory machines through an interconnection network.

Applications on SMP clusters can be programmed to use message passing among all processors. However, it's possible to achieve better performance using a hybrid model with shared memory communication inside SMP nodes and message passing communication between them.

In this work, a hybrid model was used to develop an engineering application based on the Finite Element Method in order to evaluate this model and to compare its performance with a pure message passing version of the same application.

SUMÁRIO

Capítulo 1 Introdução 1
1.1 Motivação 2
1.2 Objetivos 3
1.3 Estrutura do trabalho 3
Capítulo 2 Computação de Alto Desempenho 4
2.1 – Arquiteturas Paralelas 6
2.1.1 Classificação 6
2.1.2 Modelos de comunicação. 9
2.2 *Clusters* 9
2.2.1 Atributos de *Clusters* 10
2.3 Desempenho em Processamento Paralelo 11
2.3.1 Tempo de Execução 11
2.3.2 Speedup 13
2.3.3 Eficiência 13
2.3.4 Escalabilidade 13
2.3.5 Lei de Amdahl 14
2.4 Tendências da Computação de Alto Desempenho 16
2.4.1 Lei de Moore 17
2.4.2 Explosão dos Sistemas Baseados em *Cluster* 17
2.4.3 Crescimento da Intel no cenário dos processadores 19
2.4.4 Projeções 21
Capítulo 3 Programação em Memória Compartilhada no Padrão OpenMP 23
3.1 Objetivos 24
3.2 Modelo 26
3.3 Construções OPENMP 26
3.4 Especificação de Paralelismo 27
3.5 Dados Privados e Compartilhados 29
3.6 Paralelismo Não Relacionado a *Loop* 31
3.7 Pragmas de Sincronização 31
3.8 Rotinas de Biblioteca 33
3.9 Vantagens e Desvantagens do Padrão OpenMP 34
Capítulo 4 Programação por Troca de Mensagens no Padrão MPI 36
4.1 O Padrão MPI 37
4.2 Plataformas 38
4.3 Aplicação MPI 38
4.4 Grupos, Contextos e Comunicadores 39
4.4.1 Grupos de Tarefas 39
4.4.2 Contextos de Comunicação 40
4.4.3 Comunicadores 40
4.5 Topologias de Aplicação 41
4.6 Comunicação Ponto a Ponto 41
4.7 Sincronização 43
4.8 Comunicação Coletiva 44
4.8.1 Rotinas de movimentação de dados 44

4.8.2 Rotinas de Computação Global 46
Capítulo 5 Modelos Híbridos de Programação Paralela 48
5.1 Arquiteturas SMP simples e em *Clusters* 48
5.2 Programação Híbrida 50
5.3 Benefícios da Programação Híbrida. 52
5.3.1 Aplicações com Problemas de balanceamento de carga 52
5.3.2 Aplicações de Grão Fino 52
5.3.3 Dados Replicados 53
5.3.4 Aplicações MPI restritas 53
5.3.5 Balanceamento de Poder Computacional 53
5.4 Modelos de Programação 53
5.4.1 Tamanho de Grão da Paralelização 54
5.5 Sobreposição de chamadas de comunicação 56
5.6 Outras Fontes de *Overhead* do OpenMP 56
5.7 Desempenho dos Modelos de Programação Paralela 57
Capítulo 6 Aplicação Híbrida Para um Código de Elementos Finitos Aplicado à Elasticidade Linear 58
6.1 Elasticidade Linear 59
6.2 O Método dos Elementos Finitos 61
6.2.1 Interpretação 61
6.3 Método dos Gradientes Conjugados 63
6.4 Código 65
6.4.1 Aplicação Pura MPI 65
6.4.2 Solução Híbrida 67
Capítulo 7 Resultados Experimentais 70
7.1 Configurações de *Hardware* 70
7.2 Metodologia 70
7.3 Tempos de Execução: Versão Híbrida x Versão Pura MPI 73
7.4 Tempos de Execução Por Etapas 73
7.4.1 Tempos de Comunicação 75
7.5 Speedup Relativo: Versão Híbrida x Versão Pura MPI 75
7.6 Desempenho: Versão Híbrida x Versão Pura MPI 76
7.7 Análise dos Resultados 76
Capítulo 8 Conclusão 79
Referências 81

LISTA DE TABELAS

TABELA 7.1: Especificação do *Cluster* 70
TABELA 7.2: Malhas utilizadas 71
TABELA 7.3: Valores de tempo total de execução (em segundos) 73
TABELA 7.4: Tempo total de Execução Decomposto para Malha 1 (em segundos) 74
TABELA 7.5: Tempo total de Execução Decomposto para Malha 2 (em segundos) 74
TABELA 7.6: Tempo total de Execução Decomposto para Malha 3 (em segundos) 74
TABELA 7.7: Tempo total de Execução Decomposto para Malha 4 (em segundos) 74
TABELA 7.8: Valores de *speedup* relativo. 75
TABELA 7.9: Valores de desempenho. 76

LISTA DE FIGURAS

Figura 2.1: Estrutura básica de um multiprocessador....8
de memória compartilhada centralizada....8
Figura 2.3: Arquitetura de um *Cluster* de Computadores....10
Figura 2.3: Lei de Amdahl para uma aplicação da qual 95% do código pode executar em paralelo....15
Figura 2.5: Desempenho dos computadores mais rápidos nas últimas seis décadas [55]....18
Figura 2.6: Principais arquiteturas no Top 500 na última década [55]....19
Figura 2.7: Processadores no TOP 500 na última década [55]....20
Figura 2.8: Projeções para o Top500 na próxima década [55]....21
Figura 3.1: Modelo de Execução do OpenMP....26
Figura 3.2: Formato de *pragmas* do OpenMP....27
Figura 3.3: Formato da diretiva *parallel*....27
Figura 3.4: Exemplo de um trecho de código utilizando a diretiva #*pragma omp* for....28
Figura 3.5: Paralelização incorreta de loop....28
Figura 3.6: Forma abreviada da diretiva *for*....29
Figura 3.7: Dependências entre iterações de *loop*....29
Figura 3.8: Redução no OpenMP....30
Figura 3.9: QuickSort utilizando Sections no OpenMP....31
Figura 3.10: Remoção de barreira implícita utilizando a cláusula *nowait*....32
Figura 3.11: Uso da diretiva #pragma omp single....32
Figura 4.1: Aplicação MPI....39
Figura 4.2: Topologia em Grafo para uma aplicação MPI....41
Figura 4.3: Sincronização de barreira; (a) para todas as tarefas do comunicador; b) para apenas um subconjunto das tarefas no comunicador....43
Figura 4.4: *Broadcast* um para todos....45
Figura 4.5: (a) Operação de scather; (b) Operação de Gather....45
Figura 4.6: Operação de *Reduce*....46
Figura 4.7: Operação de *AllReduce*....47
Figura 4.8: Operação de *scan* prefixada....47
Figura 5.1: Representação Memória Compartilhada (SMP)....49
Figura 5.2: Representação SMP *clusterizado*....49
Figura 5.3 Representação de uma abordagem hierárquica para um problema de um *array* bidimensional....51
Figura 5.4: Fluxo de execução com quatro processos MPI e duas *threads* OpenMP....51
Figura 5.5: Fluxograma de paralelização de Grão Fino....55
Figura 6.1: Teste de Tensão....59
Figura 6.2: Gráfico de tensão–deformação de materiais dúcteis [59]....60
Figura 6.3: Geometrias típicas de elementos finitos em uma, duas e três dimensões [15]....62
Figura 6.4: Formas quadráticas para tipos diferentes de matrizes; (a) positiva definida; (b) negativa definida; (c) positiva indefinida; (d) indefinida [1]....64
Figura 6.5: Busca de mínimo no método dos gradientes conjugados [1]....64
Figura 6.6: Fluxo da Aplicação....66
Figura 6.7: Fluxo da Aplicação Modificado (Híbrida)....68
Figura 6.8: Decomposição de dados; (a) versão pura; (b) versão híbrida....69
Figura 7.1 - Geometria Selecionada [59]....71
Figura 7.2: Malha dois com 933 nós e 4014 elementos [59]....72
Figura 7.3: Deformação obtida [59]....72

Figura 7.4 – Gráfico de comparação do tempo total de execução para o programa híbrido e para o programa puro MPI. 73
Figura 7.5 – Tempo de Execução Decomposto em Tempo de Computação e Tempo de Comunicação. 74
Figura 7.6 - Gráfico de comparação do tempo de comunicação para o programa híbrido e para o programa puro MPI. 75
Figura 7.7 – Gráfico de *speedup* relativo para programa híbrido e puro MPI. 76
Figura 7.8 – Gráfico de desempenho para o programa híbrido e puro MPI. 77

LISTA DE SÍMBOLOS

A: Matriz quadrada, simétrica e positiva-definida

$\mathbf{A_S}$: Matriz quadrada que armazena os valores de rigidez dos graus de liberdade compartilhados da matriz **A**

$\mathbf{A_p}$: Matriz quadrada que armazena os valores de rigidez dos graus de liberdade internos da matriz **A**

b: Vetor de carregamentos aplicados

$\mathbf{b_p}$: Vetor de carregamentos aplicados aos graus de liberdade internos ou privados

$\mathbf{b_s}$: Vetor de carregamentos aplicados aos graus de liberdade compartilhados

B: Matriz da derivada da função de forma

$\mathbf{B_p}$: Matriz que armazena os valores de rigidez que relacional os graus de liberdade compartilhados e privados da **A**

$\mathbf{B_p}^{\mathbf{T}}$: Matriz transposta de $\mathbf{B_p}$

d: Direção de busca do método dos gradientes conjugados

E: Módulo de elasticidade

e: Vetor erro do método dos gradientes conjugados

$f(x)$: Funcional de forma quadrática

$f'(x)$: Gradiente do funcional de forma quadrática

K: Matriz de rigidez

n: Vetor unitário normal a uma superfície

t: Vetor tensão

T: Tensor tensão de Cauchy

u: Campo de deslocamento

u, v, w: Componentes do vetor deslocamento **U**

x: Vetor solução do sistema de equações lineares

$\mathbf{x_p}$: Vetor solução privado do sistema de equações lineares

$\mathbf{x_s}$: Vetor solução compartilhado do sistema de equações lineares

x: Vetor que representa as coordenadas do ponto genérico

x, *y* e *z*: Eixo do sistema de coordenas globais

ε: Campo de deformações

ε: Deformação

σ: Vetor tensão normal

CAPÍTULO 1

INTRODUÇÃO

Na área de processamento paralelo, vários processadores são utilizados simultaneamente para executar uma única aplicação em menos tempo. Do ponto de vista do programador, esse tipo de aplicação pode ser projetado através de dois modelos principais: memória compartilhada e troca de mensagens.

O modelo de programação em memória compartilhada é direcionado para arquiteturas nas quais múltiplos processadores compartilham um único espaço de memória. A comunicação entre os processadores nesse modelo é realizada lendo-se e escrevendo-se dados nesse espaço de memória. A noção do que é privado e compartilhado se torna importante nesse caso. Dados compartilhados são visíveis para todos os processadores participando da execução paralela enquanto dados privados são locais para cada processador e não podem ser acessados por outros. A comunicação entre os processadores ocorre através da leitura e escrita nesses dados compartilhados.

Um outro modelo de processamento paralelo é direcionado para arquiteturas de troca de mensagens. Nesse modelo, processadores não compartilham memória. Ao invés disso, eles enviam e recebem mensagens através da rede de interconexão. Todos os dados são privados e a única forma de um processador obter uma informação que não está na sua memória local é requisitando-a ao processador que a possui.

Para que um programa seja executado em algum desses modelos é necessário algum tipo de construção de linguagem de programação. Esse tipo de construção controla o compartilhamento de dados, a sincronização e assim por diante. Cada um dos dois modelos de programação paralela oferece tipos diferentes de construções para alcançar esse fim.

No modelo de troca de mensagens, as construções geralmente se baseiam em bibliotecas de funções. Essas bibliotecas incluem funções para enviar e receber mensagens, execução de sincronização, comunicação coletiva, etc. Nesse modelo o programador precisa explicitamente particionar os dados, realizar a comunicação e a sincronização. O MPI (*Message Passing Interface*) [35] é o principal padrão desse tipo de biblioteca utilizado.

No padrão OpenMP [38] para programação em memória compartilhada, o programador adiciona diretivas de compilação ao código fonte. Essas diretivas não afetam a semântica do programa, apenas indicam como trabalho e dados devem ser compartilhados entre os processadores [39]. O código fonte é então compilado por um compilador (com

suporte ao padrão) que gera código para criar *threads* que executam em paralelo nos diversos processadores do sistema.

Cada um desses padrões é adequado a seus respectivos modelos de arquitetura de *hardware*, mas existem arquiteturas de multiprocessadores para as quais o mapeamento para um deles não é tão simples. Um exemplo importante são os *clusters* de SMPs (*Symmetric Multi-Processors*). *Clusters* de SMPs são construídos a partir de diversos nós SMP (um tipo de máquina de memória compartilhada) conectados através de uma rede de interconexão.

De fato, existe uma tendência em computação de alto desempenho de construir computadores paralelos acrescentando gradativamente nós SMP interconectados por redes de interconexão simples. Cada um desses nós consiste de um determinado número de processadores e uma grande quantidade de memória compartilhada [6].

Clusters de SMPs podem ser programados para utilizar troca de mensagens entre todos os processadores em todos os nós envolvidos no sistema. No entanto, existe a possibilidade de um melhor desempenho ser for utilizado um modelo hierárquico de programação que seja mapeado diretamente para o modelo físico que combina memória distribuída e compartilhada [61]. Nesse contexto, um modelo de programação híbrido é definido como o modelo que utiliza *multithread* em memória compartilhada dentro do nó SMP e troca de mensagens entre os nós SMP.

Teoricamente, programas híbridos deveriam oferecer um desempenho melhor que programas puros de troca de mensagens por três razões: 1) A troca de mensagens dentro do nó é substituída por um acesso de memória compartilhada mais rápido; 2) Há um volume menor de comunicação nos meios de transmissão de dados já que as mensagens intra-nó não são necessárias. 3) Há uma menor quantidade de processos envolvidos na comunicação o que deve levar a uma melhor escalabilidade [6].

1.1 Motivação

Esse trabalho é um estudo sobre um modelo híbrido para aplicações de programação paralela. O Departamento de Ciência de Computação da Universidade de Brasília possui como recurso, à sua disposição, um *cluster* de nós SMP. Esse recurso foi utilizado em diversas pesquisas anteriores para desenvolvimento de aplicações científicas paralelas, mas nunca ficou claro se a arquitetura dessa máquina era utilizada da forma mais efetiva possível. Decidiu-se então investigar se, com um estilo de programação que mapeasse diretamente para a arquitetura SMP de *hardware*, os resultados seriam melhores.

Na pesquisa realizada por [59] foi feito um estudo comparativo (em modelos puros de troca de mensagens) do desempenho de códigos paralelos baseados no Método dos Elementos Finitos (MEF) aplicado à elasticidade linear para problemas estruturais que utilizam o método dos gradientes conjugados para solução de sistemas de equações. Esse problema se apresenta como uma aplicação adequada para essa pesquisa por se tratar de um problema real no qual grandes benefícios podem ser obtidos pela paralelização do código. De fato, problemas modelados pelo Método dos Elementos Finitos apresentam elevados custos computacionais em termos de tempos de execução e uso de memória, principalmente devido à grande quantidade de dados a serem processados durante a solução do sistema de equações.

1.2 Objetivos

O objetivo da pesquisa apresentada nessa dissertação é desenvolver e avaliar o uso de um modelo híbrido de programação para uma aplicação real de engenharia baseada no método dos elementos finitos. Além disso, investigar quão efetivo é o seu uso através da análise do ganho de desempenho obtido e da comparação com o desempenho do modelo puro MPI.

1.3 Estrutura do trabalho

Na introdução, é apresentada uma visão geral do problema a ser analisado, além da motivação e dos objetivos desse trabalho. O restante desse documento é organizado da seguinte maneira:

O capítulo 2 apresenta uma visão geral da área computação de alto desempenho e nesse capítulo são introduzidos vários conceitos utilizados na dissertação. Segue-se o capítulo 3 sobre programação em memória compartilhada no padrão OpenMP. O capítulo 4 aborda programação em memória distribuída e descreve em detalhes o padrão MPI para programação com troca de mensagens. O capítulo 5 apresenta uma visão geral dos modelos híbridos de programação paralela, além de seus problemas, vantagens e desvantagens. O capítulo 6 descreve a aplicação e em seguida no capítulo 7 são apresentados os resultados experimentais obtidos. Finalmente são apresentas conclusões e sugestões de trabalhos futuros no Capítulo 8.

CAPÍTULO 2

COMPUTAÇÃO DE ALTO DESEMPENHO

O tempo de execução, *T,* de um programa depende do número de instruções a serem executadas, do número médio de ciclos de *clock* consumidos por instrução e do tempo de ciclo de *clock* [8]:

$$T = \textit{número de instruções} \times \textit{ciclos por instrução} \times \textit{tempo de clock} \qquad (2.1)$$

A redução do tempo de um ciclo de *clock* é uma questão relacionada à engenharia e pode ser alcançada através do uso de materiais mais avançados e da construção de circuitos menores e mais eficientes. Os outros dois fatores são melhorados através de estratégias de paralelismo, que replicam componentes básicos do sistema.

Paralelismo existe em uma grande variedade de máquinas e se apresenta de várias formas que podem ser classificadas em três níveis distintos [8]: paralelismo de *jobs*, paralelismo de aplicação e paralelismo de instruções.

Paralelismo de jobs

É o nível mais alto de paralelismo e é de interesse maior para administradores de sistema do que de usuários. Nesse tipo de paralelismo, o mais importante é que um laboratório ou centro de comunicação execute uma maior quantidade de *jobs* possíveis em um período de tempo específico. Isso pode ser alcançado, adquirindo-se mais máquinas de forma que uma maior quantidade de *jobs* seja executada ao mesmo tempo, apesar de para o usuário um *job* particular não executar mais rápido. Nesse caso há uma diferenciação entre *throughput* (número de *jobs* em um período de tempo) e latência (tempo para executar uma aplicação).

Paralelismo de aplicação

Ocorre quando um programa simples é dividido em partes que são executadas ao mesmo tempo em múltiplos processadores ou múltiplas unidades funcionais. Paralelismo em nível de programa geralmente se manifesta de duas formas: sobre seções independentes de um mesmo programa; ou sobre iterações individuais de um laço onde não há dependência entre de dados.

Paralelismo de Instruções

É invisível para os usuários e está no nível de organização de computadores. *Pipelines* são a forma mais comum de alcançar esse tipo de paralelismo [40] [8]. Nesse caso, instruções podem ser sobrepostas ou uma determinada instrução pode ser decomposta em suboperações e essas suboperações serem sobrepostas. Em geral, programadores não precisam se preocupar com esse nível de paralelismo já que compiladores são capazes de organizar programas para explorá-lo.

Um conceito relacionado ao nível de paralelismo é o tamanho de grão das tarefas paralelas. Em um sistema de grão grosso as tarefas que executam em paralelo representam trechos grandes da aplicação. Em sistemas paralelos de grão fino, por outro lado, as tarefas representam trechos bem pequenos da aplicação constituídos de algumas poucas instruções.

No projeto de computadores de alto desempenho, deve-se tomar uma decisão entre um pequeno número de processadores poderosos ou um grande número de processadores simples para alcançar o desempenho necessário. A vantagem de um pequeno número de processadores poderosos é a simplicidade de interconexão e a possibilidade de utilizar organizações de memória que facilitem a programação. Por outro lado esses processadores são extremamente caros.

Utilizar um grande número de processadores simples oferece uma grande economia nesse sentido, em detrimento de uma maior complexidade nas estratégias de interconexão e de estratégias de organização de memória que dificultam a programação. Com o rápido desenvolvimento dos microprocessadores, máquinas constituídas de algumas centenas de processadores alcançam o mesmo desempenho das máquinas de processadores especializadas mais poderosas e mais caras [55] [8].

2.1 – Arquiteturas Paralelas

2.1.1 Classificação

A principal terminologia para classificação de sistemas distribuídos foi proposta por Flyn [16] e apesar de ser rudimentar, é muito útil para a classificação de computadores de alto desempenho. Essa classificação se baseia na forma como fluxos de instruções e fluxos de dados são manipulados e inclui quatro classes de computadores: [55] [40]:

SISD (Fluxo de instruções único, fluxo de dados único)

Esses são os sistemas convencionais monoprocessados que possuem uma CPU e que acomodam um fluxo de instruções que é executado de forma serial.

SIMD (Fluxo de instruções único, vários fluxos de dados)

A mesma instrução é executada por diferentes fluxos de dados em diferentes processadores. Cada processador tem sua própria memória de dados, mas existe uma única memória de instruções e uma única unidade de controle que busca e despacha as instruções. As arquiteturas vetoriais são a classe mais ampla de processadores desse tipo.

MISD (Vários fluxos de instruções, fluxo de dados único)

Só existe um único fluxo de dados operando por sucessivas unidades funcionais. Nenhum multiprocessador comercial desse tipo foi construído até hoje, mas pode ser elaborado no futuro. Alguns processadores de fluxo de uso especial se aproximam de uma forma limitada dessa categoria.

MIMD (Vários fluxos de instruções, vários fluxos de dados)

Essas máquinas executam vários fluxos de instruções diferentes em paralelo em dados diferentes. A diferença em relação às máquinas SISD reside no fato que as instruções e os dados são relacionados porque representam partes diferentes da mesma tarefa a ser executada.

Dessa forma, sistemas MIMD podem executar diversas subtarefas em paralelo para diminuir o tempo de solução da tarefa principal.

Existe uma grande variedade de sistemas MIMD e especialmente nessa classe, a classificação de Flyn se mostra inadequada, pois inclui arquiteturas totalmente distintas. A categoria das máquinas MIMD pode ser subdividida em dois outros grupos: Sistemas de Memória Compartilhada e Sistemas de Memória Distribuída.

Sistemas de Memória Compartilhada

No caso de multiprocessadores com número pequeno de nós, é possível que os processadores compartilhem fisicamente uma única memória centralizada e que processadores e memória sejam interconectados por um barramento. Com caches grandes, o barramento e a memória única podem satisfazer às demandas de memória de um número pequeno de processadores. Substituindo-se o barramento único por vários barramentos ou até mesmo por um *switch*, um projeto de memória compartilhada pode ter sua escala aumentada até algumas dezenas de processadores. Embora esse aumento de escala seja tecnicamente concebível, essa organização se torna menos atraente à medida que aumenta o número de processadores devido à contenção no acesso à memória. Pelo fato de existir uma única memória principal que tem um relacionamento simétrico com todos os processadores e um tempo de acesso uniforme a partir de qualquer processador, esses multiprocessadores freqüentemente são chamados de multiprocessadores simétricos (de memória compartilhada) (SMP- *Symetric Multiprocessors*). A figura 2.1 mostra uma representação de um sistema de memória compartilhada.

Sistemas de Memória Distribuída

O segundo grupo consiste em multiprocessadores com memória fisicamente distribuída. Para dar suporte a quantidades maiores de processadores, a memória deve ser distribuída entre os processadores em vez de centralizada para atender à demanda de largura de banda sem incorrer em uma latência de acesso à memória longa demais. A distribuição de memória entre os nós tem duas vantagens importantes. Primeiro, é uma forma econômica de aumentar a escala da largura de banda de memória se a maior parte dos acessos se destina à memória local no nó. Em segundo lugar, ela reduz a latência para acesso à memória local.

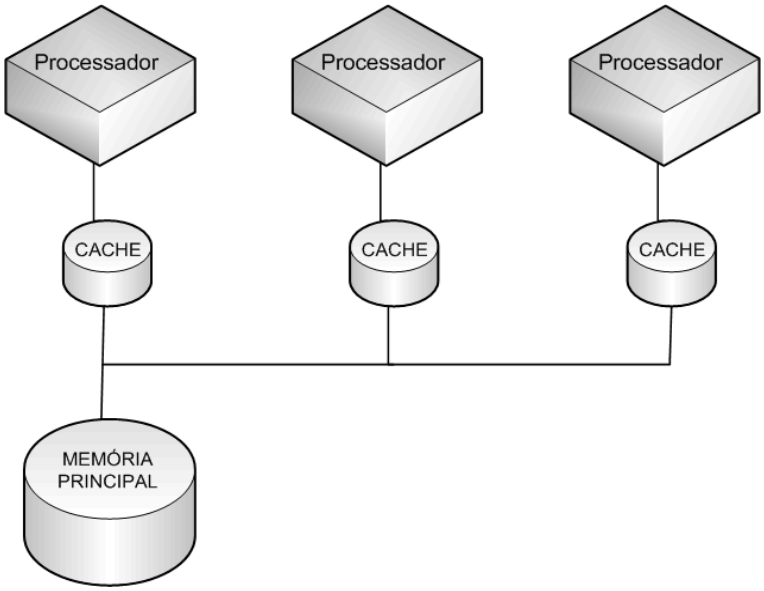

Figura 2.1: Estrutura básica de um multiprocessador
de memória compartilhada centralizada

A principal desvantagem é que a comunicação de dados entre os processadores se torna mais complexa e tem latência mais alta porque os processadores não compartilham mais uma única memória centralizada. A figura 2.2 mostra uma representação de um multiprocessador de memória distribuída [40] [26].

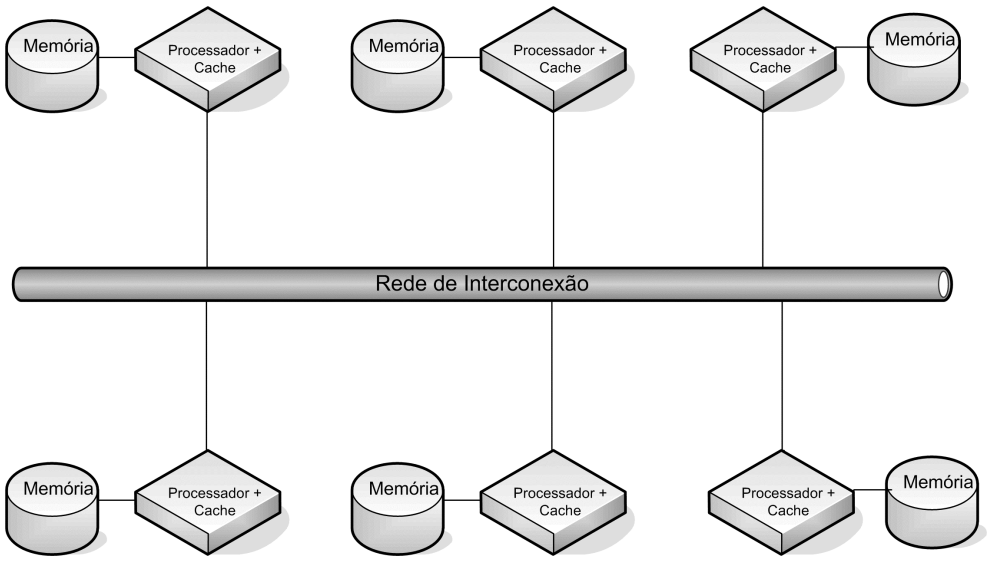

Figura 2.2: Arquitetura básica de um multiprcessador de memória distribuída

2.1.2 Modelos de comunicação.

Para os multiprocessadores de memória fisicamente compartilhada, a comunicação entre processadores é realizada simplesmente escrevendo-se e lendo-se informações do espaço de endereçamento comum. É uma forma de comunicação bem eficiente, porém bastante restrita quanto ao número de processadores participantes, devido a problemas de acesso à memória [26].

Normalmente, para qualquer multiprocessador com uma quantidade muito grande de nós deve-se utilizar módulos de memória distribuídos fisicamente com os processadores. Nesse caso, existem duas abordagens alternativas para troca de informações entre os processadores.

Na primeira abordagem, a comunicação ocorre por meio de um espaço de endereçamento virtual compartilhado. Isto é, as memórias fisicamente separadas podem ser endereçadas com um único espaço de endereços compartilhado logicamente. Cada referência à memória pode então ser realizada por qualquer processador e para qualquer posição no espaço de endereçamento global, supondo apenas que ele tenha os direitos de acesso corretos. Esses multiprocessadores são chamados de arquiteturas de memória compartilhada distribuída (DSM – *distributed shared memory*) [53]. O termo memória compartilhada nesse caso não significa que existe um compartilhamento físico de memória e sim um compartilhamento lógico.

Na segunda abordagem, o espaço de endereços é constituído de vários módulos de memória disjuntos que não podem ser endereçados por um processador remoto. Em tais máquinas, o mesmo endereço físico em dois processadores diferentes se refere a duas informações diferentes. Cada módulo processador-memória é em essência um computador separado e por essa razão esses computadores paralelos são chamados de multicomputadores. Um multicomputador pode consistir até mesmo de computadores completos, totalmente independentes e conectados apenas por uma rede local. Esse tipo de multicomputador recebe hoje em dia a denominação popular de *cluster*. A comunicação de dados em multiprocessadores é realizada através da troca explícita de mensagens. [40].

2.2 *Clusters*

Um *cluster* é uma coleção de computadores completos (*nós*) que são conectados fisicamente por uma rede de alta performance ou uma rede local. Tipicamente, cada nó é uma

estação de trabalho, um computador pessoal ou uma máquina SMP. O mais importante é que todos os nós do *cluster* devem ser capazes de trabalhar juntos como um recurso computacional único e integrado. Além disso, cada nó pode trabalhar como uma máquina individual. O objetivo dos *clusters* é fornecer serviços de alta disponibilidade de alto desempenho. A arquitetura conceitual de um *cluster* é apresentada na figura 2.3 [26].

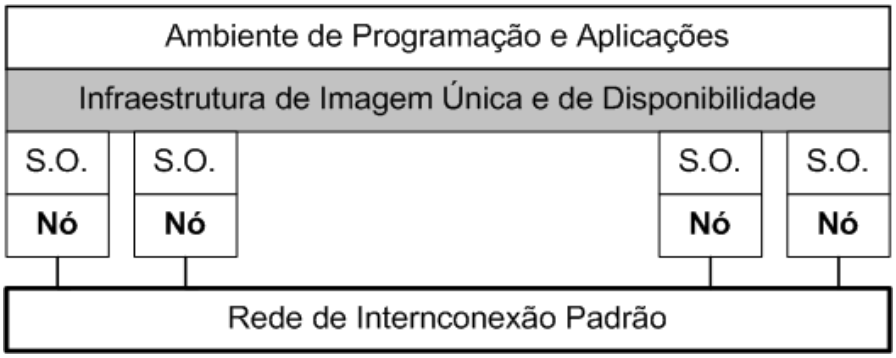

Figura 2.3: Arquitetura de um *Cluster* de Computadores

2.2.1 Atributos de *Clusters*

Nós de um Cluster

Cada nó é um computador completo. Isso implica que cada nó possui seu próprio processador, cache, memória, disco e dispositivos de E/S. Além disso, existe um sistema operacional completo e independente em cada nó. Um nó pode possuir mais de um processador, mas apenas uma cópia do sistema operacional.

Imagem Única

Um *cluster* é um recurso computacional único. Isso o diferencia de um sistema distribuído típico cujos nós são usados como recursos individuais. Um *cluster* realiza o conceito de recurso único através de uma das diversas técnicas de SSI (*single system image* - imagem única do sistema).

Conexão Entre Nós

Os *nós* de um *cluster* geralmente são conectados através de uma rede padrão como Ethernet, FDDI, Fiber-Channel ou ATM.

Alta Disponibilidade

Clusters representam uma forma de aumentar a disponibilidade de um sistema, ou seja a porcentagem de tempo que o sistema fica disponível para o usuário.

Aumento de Desempenho

Um *cluster* deve oferecer alto desempenho para várias finalidades. Uma finalidade é utilizá-lo como um superservidor. Se cada nó de um *cluster* pode servir *n* clientes, então o *cluster* como um todo pode servir *mn* clientes simultaneamente. Uma outra finalidade é utilizar o *cluster* para minimizar o tempo de execução de uma aplicação, distribuindo o trabalho em tarefas paralelas.

2.3 Desempenho em Processamento Paralelo

2.3.1 Tempo de Execução

O tempo de execução é a medida de desempenho mais confiável e que melhor traduz o que se busca em termos de velocidade de processamento, tanto do *hardware* quanto do *software*. Nesse sentido uma definição de desempenho pode ser:

$$desempenho = \frac{1}{tempo\ de\ execução} \tag{2.2}$$

O tempo de execução pode ser visto de duas maneiras: tempo decorrido desde o início até o final da execução do programa ou o tempo de CPU, isto é, o tempo que efetivamente foi utilizado pela CPU para executar o programa, excluindo-se o tempo consumido pelo próprio sistema operacional durante a execução do programa. O normal é a utilização do tempo decorrido ou do programa como um todo, ou parte dele. Pois o tempo de CPU também não contabiliza o tempo de comunicação, que é adicionado para se utilizar as máquinas em paralelo.

O tempo total de execução pode ser decomposto em tempo de computação, tempo de comunicação e tempo de espera [10].

Tempo de Computação

O tempo de computação de um algoritmo é o tempo consumido realizando algum trabalho. O tempo de computação geralmente vai depender de alguma forma, do tamanho do

problema. Se o algoritmo paralelo replica computação, então o tempo de computação também dependerá do número de tarefas ou processadores. Em sistemas heterogêneos, o tempo de computação também dependerá de em qual processador as operações foram realizadas. Além disso, também dependerá das características do processador e de seu sistema de memória.

Tempo de Comunicação

O tempo de comunicação de um algoritmo é o tempo que suas tarefas gastam enviando e recebendo mensagens. Existem dois tipos básicos de comunicação: interprocessos e intraprocessos.

Na comunicação interprocessos, as duas tarefas comunicantes estão localizadas em processos diferentes. Na comunicação intraprocessos, duas tarefas comunicantes estão localizadas no mesmo processador. O custo de envio de uma mensagem entre duas tarefas localizadas em processadores diferentes pode ser representado por dois parâmetros: o tempo para início da comunicação e o tempo de transferência por palavra (tipicamente 4 bytes) que é determinado pela largura de banda física do link de comunicação.

Tempo de Espera

Um processador pode estar em espera devida à falta de computação ou falta de dados. No primeiro caso, o tempo espera pode ser reduzido usando técnicas de balanceamento de carga. No segundo caso o processador está em espera enquanto computação e comunicação necessárias precisam ser realizadas.

Ambos, tempo de computação e tempo de comunicação são especificados explicitamente em um algoritmo paralelo. Assim, é mais fácil determinar suas contribuições para o tempo de execução. Tempo de espera pode ser mais difícil para determinar, já que muitas vezes depende da ordem na qual as operações são apresentadas.

O tempo de espera pode às vezes ser evitado estruturando o programa de forma que processadores realizem outras computações ou comunicações enquanto esperam por dados remotos. Essa técnica é chamada de sobreposição de computação e comunicação já que computação local pode ser realizada concorrentemente com comunicação remota e computação.

Essa sobreposição pode ser alcançada de duas formas. A abordagem mais simples é criar múltiplas tarefas em cada processador. Quando uma tarefa está bloqueada esperando por dados remotos, a execução pode ser passada para outra tarefa para a qual dados já estão disponíveis [17].

2.3.2 Speedup

Tratando-se de uma aplicação paralela, onde o objetivo é obter um aumento na velocidade de processamento pela subdivisão do problema em tarefas que podem ser executadas concorrentemente, é interessante verificar o desempenho do sistema através do speedup S_n, que é a razão entre o tempo gasto na execução seqüencial ou em uma única máquina, T_1 e o tempo de execução em paralelo, isto é, em mais de uma máquina, T_n. O *speedup* evidencia o ganho de tempo obtido na execução paralela para um dado número de tarefas concorrentes [17]:

$$s_n = \frac{T_1}{T_n} \tag{2.3}$$

2.3.3 Eficiência

A eficiência, E_n, é a razão entre o valor do *speedup* (S_n) e o número de tarefas n, utilizadas na execução paralela Assim, a eficiência mostra se os ganhos obtidos com a adição de máquinas estão sendo relevantes de forma a determinar a quantidade ótima de máquinas necessárias para a execução paralela de um dado tipo de problema e volume de dados [17]:

$$E_n = \frac{S_n}{n} \tag{2.4}$$

2.3.4 Escalabilidade

Um aspecto importante da análise de performance é o estudo de como o desempenho do algoritmo varia com relação a parâmetros como tamanho do problema, número de processadores, etc. Para algoritmos paralelos em particular, é de interesse o comportamento com aumento no número de processadores.

Uma abordagem para quantificar escalabilidade é determinar como o tempo de execução e a eficiência variam como o aumento do número de processadores, para um tamanho de problema fixo. Essa análise do problema fixo possibilita a resposta à questões como: o limite de velocidade para resolver um determinado problema em um computador específico; O maior número de processadores que se deve utilizar para manter uma eficiência

É importante considerar tanto eficiência quanto tempo quando a escalabilidade está sendo avaliada. Enquanto a eficiência vai geralmente diminuir monotonamente com o número de processadores, o tempo de execução pode aumentar se o modelo de desempenho inclui um

termo proporcional a uma potencia positiva do número de processadores. Em alguns casos, não será produtivo mais do que algum número máximo de processadores para um tamanho de problema em particular ou escolha de parâmetros de máquina [17].

2.3.5 Lei de Amdahl

Gene Amdahl, formulou o que hoje é chamado de a Lei de Amdahl para caracterizar a maneira como uma aplicação poderia utilizar de forma eficiente processadores paralelos escaláveis.

Praticamente todo programa paralelo mescla partes que são seriais e partes paralelas. Uma análise de engenharia é um bom exemplo. A parte de incialização provavelmente será completamente serial: os dados de entrada são lidos, a matriz é preenchida e os dados são particionados. A fase de solução do problema, por outro lado, poderá se beneficiar da presença de múltiplos processadores e ser altamente paralela.

Com a lei da Amdahl pode-se verificar quanto de um programa pode executar em paralelo e quanto desse mesmo programa precisa executar usando apenas um processador. Uma vez que a razão entre a porção paralela e a seqüencial é estabelecida, ela impõe um limite superior para o *speedup* possível da aplicação. Seja uma aplicação que execute em 100 minutos onde essa aplicação pode executar em paralelo por 95 dos 100 minutos. Nesse caso, mesmo se fossem utilizados tantos processadores que a porção paralela do código executasse em um piscar de olhos, o tempo total de execução seria ainda de 5 minutos (mais um piscar de olhos) devido à parte serial do código. Assim, mesmo adquirindo um número infinito de processadores, nós melhoramos o desempenho da aplicação apenas por um fator de 20. A figura 2.3 mostra um gráfico do *speedup* para o número de processadores e pode-se perceber que não há benefício em adicionar mais processadores a partir do momento que a parte serial do código se torna o fator dominante no tempo de execução.

Quando a lei de Amdahl começou a ser discutida parecia que máquinas paralelas em grande escala eram pouco vantajosas. Pesquisas em códigos existentes colocam a porcentagem do código que pode-se executar em paralelo para aplicações típicas entre 60% e 95%. Uma análise superficial concluiria que muito mais do que cerca de 8 processadores não seria muito vantajoso [46].

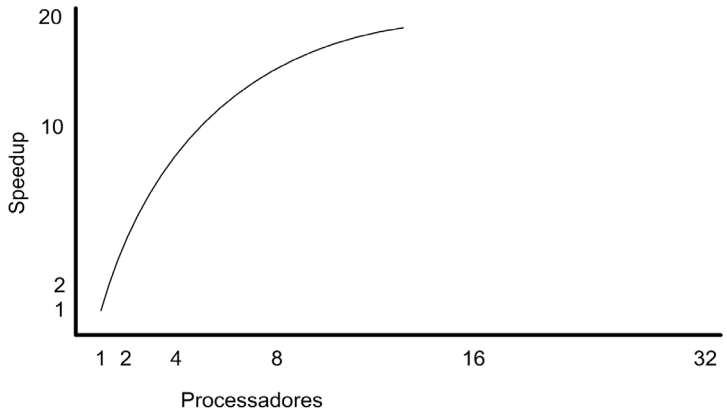

Figura 2.3: Lei de Amdahl para uma aplicação da qual 95% do código pode executar em paralelo

A Lei de Amdahl é correta, mas existem algumas suposições erradas, feitas quando ela foi usada para concluir que paralelismo maciço não era vantajoso [46]:

- Pelo fato de não haver processadores paralelos disponíveis no momento da concepção da maioria das aplicações da pesquisa citada, os programadores não se preocuparam em tentar tornar seus códigos mais adequados para serem paralelizados. À medida que esses sistemas se tornam muito utilizados, muito esforço é realizado para repensar essas aplicações de modo a maximizar as operações que podem ser realizadas em paralelo.

- Geralmente quando o tamanho do problema é duplicado, a parte serial leva o dobro do tempo para executar uma tarefa, enquanto a parte paralela leva de quatro a oito vezes mais tempo. A disparidade no aumento relativo de tempo significa que a aplicação gasta mais tempo na parte paralela da aplicação.À medida que as capacidades de memória e de processador aumentam, pesquisadores começaram a resolver problemas maiores. Assim, tornando-se um problema maior, um problema com porção paralela de 95% se torna 99% paralelo e tornando-se ainda maior se torna 99.9% paralelo e assim por diante.

Devido ao fato do *speedup* potencial ser tão influenciado pelo tamanho do problema, surgiram algumas novas leis que foram criadas para capturar esse efeito. Essas leis são chamadas Lei de Gustafson [21] e lei de Ni [37]. A lei de Gustafson analisa como o aumento do tamanho de um problema afeta a escalabilidade. Já a lei de Ni analisa a relação entre o

crescimento do tamanho do problema e a habilidade de executá-lo em paralelo. Ocorreram também avanços em pesquisas de ferramentas de análise de fluxo de dados para detectar e extrair o paralelismo de códigos onde essa tarefa realizada apenas pela análise manual do código era muito difícil [46].

2.4 Tendências da Computação de Alto Desempenho

O progresso na construção e no uso de processadores paralelos efetivos e eficientes é lento. Essa taxa de progresso foi limitada por sérios problemas de software, bem como por um longo processo de evolução da arquitetura de multiprocessadores para aumentar a facilidade de uso e melhorar a eficiência. A grande variedade de abordagens arquitetônicas e o sucesso limitado, além da curta direção de muitas arquiteturas, representam as principais dificuldades no que diz respeito ao software. Entretanto, o progresso realizado apresenta alguns motivos para otimismo quanto ao futuro do processamento paralelo e dos multiprocessadores.

Em primeiro lugar, o uso de processamento paralelo em alguns domínios começa a ser compreendido. Dentre eles talvez o principal seja o da computação científica e da engenharia. Esse domínio de aplicações tem uma ânsia quase ilimitada por maior capacidade de computação e nele existem muitas aplicações que têm uma grande porção de paralelismo natural. Outra área de aplicações importante e muito maior é a dos sistemas de bancos de dados e processamento de transações em larga escala. Esse domínio de aplicações também tem muito paralelismo natural disponível através do processamento paralelo de solicitações independentes, mas sua necessidade de computação em larga escala em oposição ao simples acesso a sistemas de armazenamento são menos compreendidas.

Além disso, existe hoje uma grande crença de que o modo mais efetivo de construir um computador que ofereça maior desempenho do que pode ser alcançado com um microprocessador de um único chip, é construir um *cluster* que amplie as vantagens significativas de preço-desempenho dos microprocessadores produzidos em massa.

Um terceiro motivo é que os multiprocessadores são altamente eficientes para cargas de trabalho multiprogramadas, que são com freqüência o uso dominante de mainframes e servidores de grande porte, como também de servidores de arquivo e servidores Web. Essas aplicações constituem efetivamente um tipo restrito de carga de trabalho paralela. No futuro, essas cargas de trabalho poderão representar um grande mercado para multiprocessadores de alto desempenho. Quando uma carga de trabalho quiser compartilhar recursos como o

armazenamento de arquivos ou puder compartilhar de forma eficiente um recurso como uma memória extensa, um multiprocessador poderá ser uma alternativa muito eficiente. Os multiprocessadores se mostram muito eficazes para certas cargas de trabalho comerciais intensivas e aplicações de pesquisa na Web em larga escala. No caso de aplicações comerciais que não exigem alto desempenho de comunicação, que tenham pequena necessidade de memória ou demanda limitada por computação, é provável que os *clusters* sejam mais econômicos que os multiprocessadores. Atualmente o espaço comercial é uma mistura de *clusters* de PCs básicos, SMPs e *clusters* de SMPs, com diferentes estilos arquitetônicos.

E finalmente, o multiprocessamento no chip cresceu em importância por duas razões. Primeiro, no mercado embutido no qual o paralelismo natural existe com freqüência, tais abordagens representam uma alternativa óbvia para processadores mais rápidos e possivelmente menos eficientes no uso do silício. Em segundo lugar, a diminuição dos rendimentos nos projetos de microprocessadores de ponta incentiva os projetistas a buscar o multiprocessamento no chip como uma solução potencialmente mais econômica. [40].

2.4.1 Lei de Moore

O mercado de computação de alto desempenho sempre se caracterizou pela rápida mudança de fornecedores, arquiteturas e tecnologias. Apesar de todas essas mudanças, a evolução do desempenho em larga escala parece ser um processo contínuo.

A lei de Moore [34] (“O número de transistores em um chip dobra a cada dois anos”) geralmente é citada nesse contexto.

A figura 2.5 mostra o gráfico dos computadores que alcançaram o pico de desempenho em suas respectivas épocas nas últimas seis décadas. Na média houve um aumento de desempenho de duas magnitudes a cada década. Nessa figura fica claro que a lei de Moore foi verdadeira praticamente em todo o desenvolvimento da computação moderna [52] [55].

2.4.2 Explosão dos Sistemas Baseados em *Cluster*

No final da década de 90, os *clusters* eram comuns no ambiente acadêmico, mas principalmente como objeto de pesquisa e não como uma plataforma para aplicações reais. A maioria desses *clusters* era pouco poderosa e, como resultado, a edição de Novembro de 1999 do TOP500 listava apenas sete sistemas baseados em *cluster*.

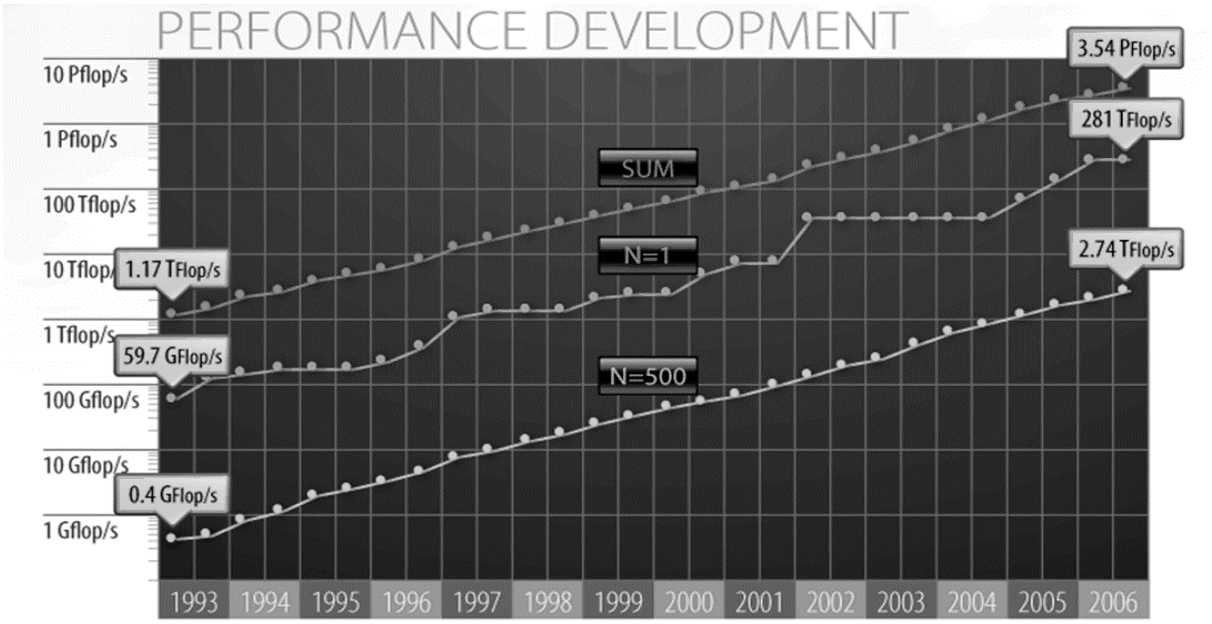

Figura 2.5: Desempenho dos computadores mais rápidos nas últimas seis décadas [55]

Isso mudou a partir do momento que os *clusters* chamaram a atenção de desenvolvedores de aplicações comerciais e industriais e que aplicações com requisitos de comunicação menos restritivos tornaram o custo benefício dos *clusters* baseados em componentes de prateleira interessante. Em pouco tempo, a maioria dos fornecedores no mercado de computação de alto desempenho comercializava esse tipo de sistema. A figura 2.6 mostra que em Novembro de 2006 os *clusters* eram a arquitetura dominante no Top500, com 361 sistemas [52] [55].

Há, no entanto, ainda uma grande diferença entre a utilização principal de *clusters* e o uso das outras arquiteturas mais integradas. Os grandes supercomputadores são usados principalmente para *turnaround computing* onde o poder de processamento máximo é aplicado para um único problema. O objetivo é resolver um grande problema que não pode ser resolvido em um tempo razoável seqüencialmente ou resolver um problema simples em um intervalo de tempo menor. Esse tipo de computação permite a solução de problemas com restrições de tempo real. O objetivo principal nesse caso é o tempo para solução do problema. Os *clusters* por outro lado, geralmente rodam diversos *jobs* simultaneamente e o objetivo principal é obter o máximo desempenho por custo [52].

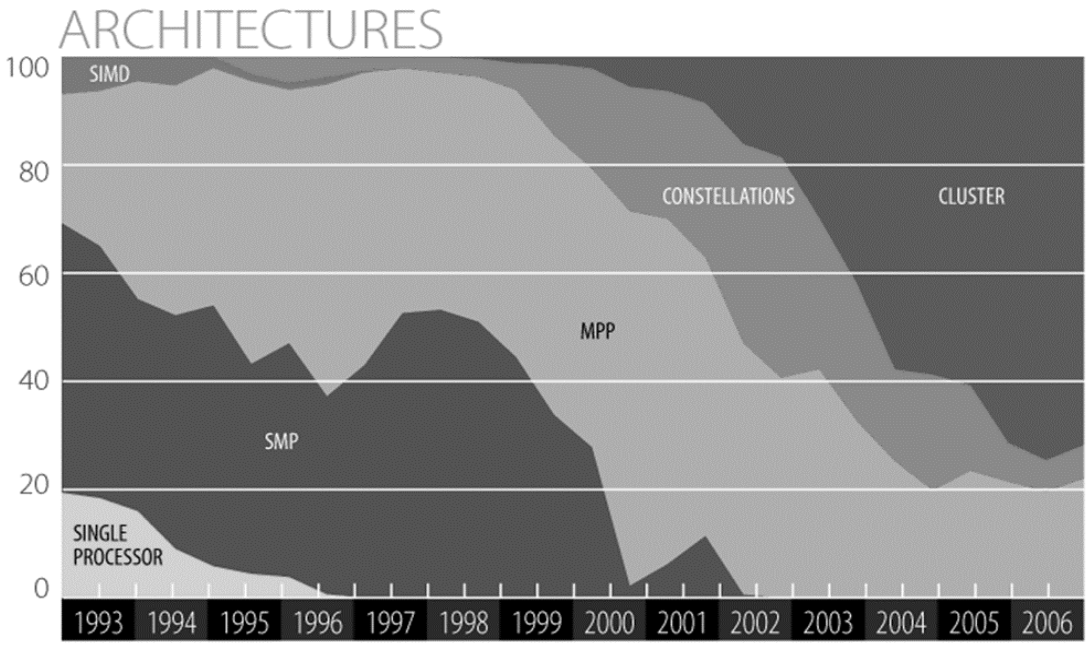

Figura 2.6: Principais arquiteturas no Top 500 na última década [55]

2.4.3 Crescimento da Intel no cenário dos processadores

A comunidade da computação de alto desempenho já utilizava componentes de prateleira maciçamente nos anos 90. MPPs e *Constellations* (*Clusters* de SMPs) tipicamente usavam processadores padrões de estações de trabalho apesar de usarem redes de interconexão customizadas. Havia, porém uma grande exceção, praticamente ninguém usava processadores Intel. Baixo desempenho e a limitação do projeto de 32 bits eram as principais razões para isso. Isso mudou com o surgimento do Pentium III e especialmente em 2001 com o surgimento do Pentium 4, que trazia grandes avanços no desempenho de memória graças ao novo modelo de barramento e suportava pontos flutuantes de 64 bits. Na figura 2.7, o número de processadores no Top500 com processadores Intel passou de apenas seis em Novembro de 2000 para trezentos e dezoito em Novembro de 2004.

O interesse em arquiteturas inovadoras sempre foi grande na comunidade de computação de alto desempenho. Isso não chega a ser surpreendente já que essa área nasceu como e sustenta seu crescimento nas inovações tecnológicas. Uma das preocupações atuais diz respeito à necessidade crescente de espaço e poder computacional nos *clusters* modernos.

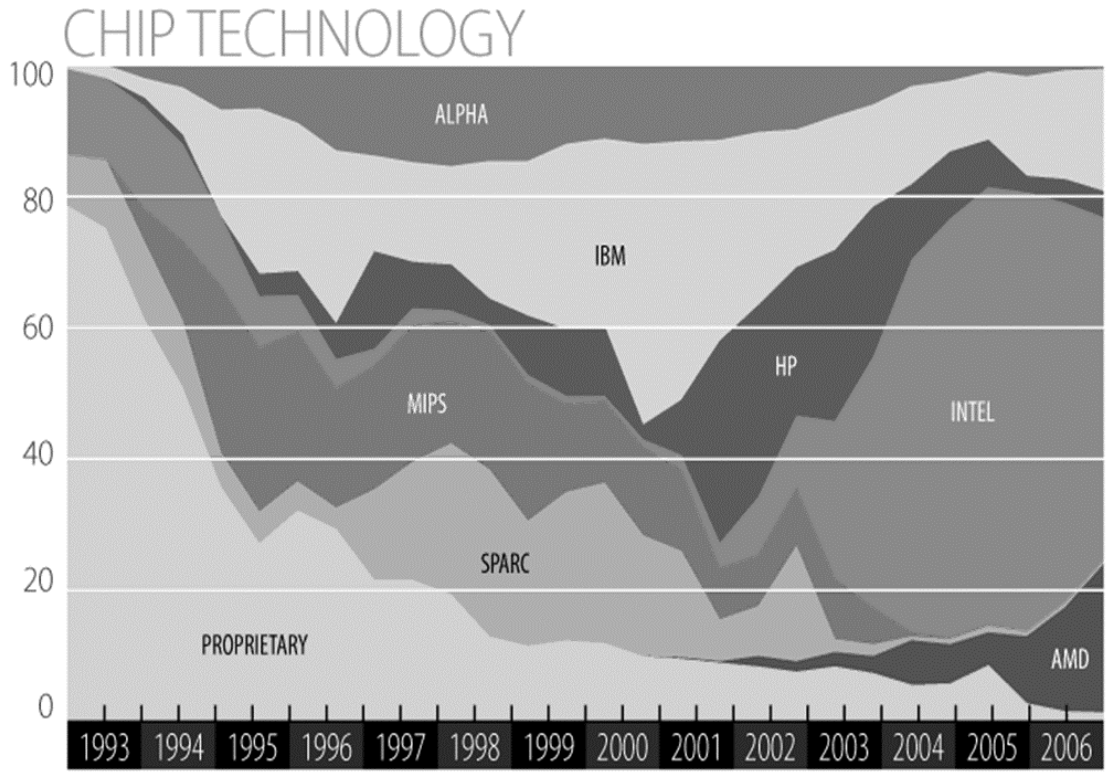

Figura 2.7: Processadores no TOP 500 na última década [55].

No desenvolvimento do *BlueGene/L*, a IBM atacou esse problema projetando um sistema muito eficiente tanto do ponto de vista computacional como do ponto de vista de O *BlueGene/L*, não utiliza os processadores mais poderosos disponíveis no mercado e sim processadores projetados especialmente para sistemas embarcados. Além de uma redução grande na memória principal total disponível, uma característica marcante nesse sistema é ele ser extremamente denso. Para alcançar o desempenho esperado um número enorme desses processadores (mais de 128.000) foi combinado, usando mecanismos de interconexão especializados. Havia muitas dúvidas quanto à capacidade desse sistema em alcançar o desempenho prometido e quanto à sua viabilidade como um sistema de uso geral. Os primeiros resultados da versão beta foram encorajadores e uma versão quatro vezes menor figurou na primeira posição da edição de Novembro de 2004 da lista dos top500.

Ao contrário do progresso no desenvolvimento de *hardware*, houve pouco, talvez uma regressão, na facilidade de programação de sistemas escaláveis. Algumas tentativas iniciadas na década de 90 na direção de software mais acessível foram completamente abandonadas.

O movimento para o modelo de memória distribuída forçou mudanças no paradigma de programação. O alto custo de comunicação e sincronização da interação processador-

processador exige novos algoritmos que minimizem essas operações. O uso de sistemas de memória distribuída provocou o crescimento de novos modelos de programação, principalmente o paradigma de troca de mensagens. No entanto, os progressos na área de *debuggers* e ferramentas de desempenho é lento e a maioria dos usuários considera as ferramentas de programação para supercomputadores paralelos extremamente inadequadas [52] [55].

2.4.4 Projeções

Baseando-se nos dados atuais da lista dos Top500 que cobre os últimos 20 anos e assumindo que o desenvolvimento atual de desempenho se mantenha nos próximos anos, o desempenho observado atualmente pode ser extrapolado para um provável cenário futuro. Essa projeção foi feita na figura 2.8 utilizando regressão linear sobre a escala logarítmica dos níveis de desempenho do Top500.

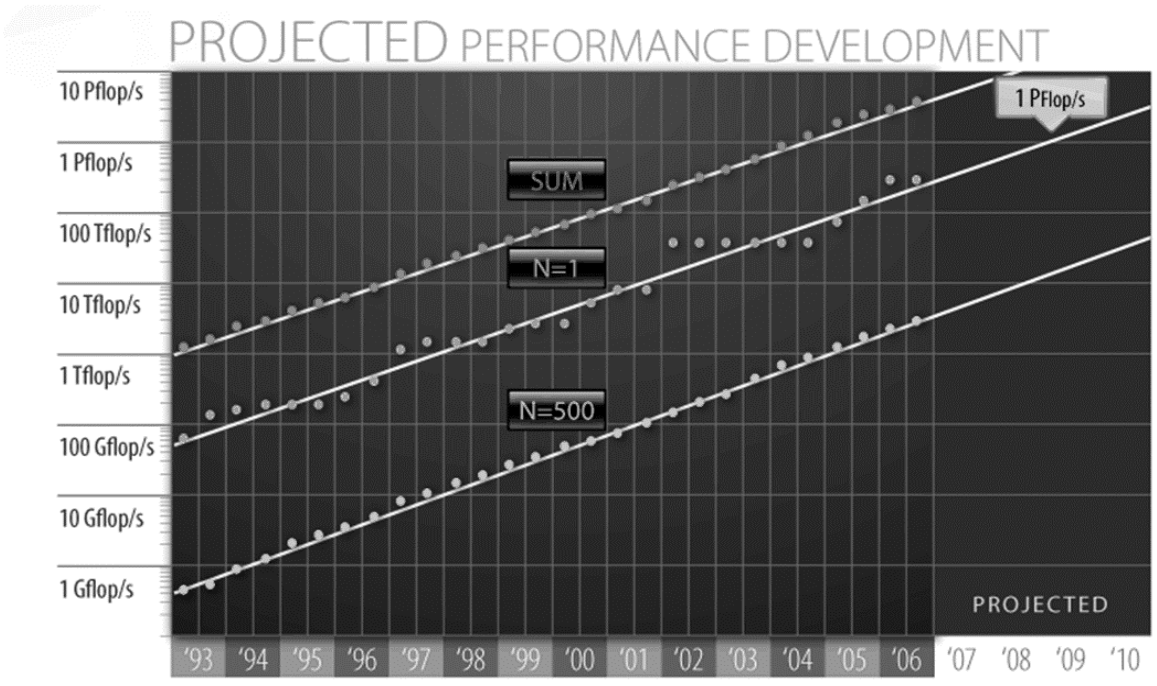

Figura 2.8: Projeções para o Top500 na próxima década [55].

Olhando quatro anos a frente espera-se que o primeiro sistema *PetaFlop* apareça na lista por volta de 2009. Olhando ainda mais longe, pode-se especular que baseado na duplicação de desempenho todo ano, o primeiro sistema excedendo 100 *Petaflops*/s deve estar

disponível por volta de 2015. No entanto, devido à rápida mudança nas tecnologias usadas em computação de alto desempenho não há como imaginar ao certo como será a arquitetura desses sistemas e como alcançarão esses níveis. O fim da lei de Moore como nós a conhecemos já foi previsto muitas vezes e talvez um dia aconteça. Novas tecnologias como a computação quântica, possivelmente permitirão no futuro aumentar capacidades computacionais muito além daquelas previstas nessas projeções. No entanto, apesar da área de alto desempenho ter mudado radicalmente diversas vezes desde o surgimento do Cray1 quarenta anos atrás, não há perspectiva para uma mudança nesse ciclo de constantes redefinições [52].

CAPÍTULO 3

PROGRAMAÇÃO EM MEMÓRIA COMPARTILHADA NO PADRÃO OPENMP

Em um sistema de memória compartilhada, cada processador tem acesso direto à memória dos demais, isto é, pode recuperar ou gravar informações em qualquer endereço que faz parte da memória compartilhada. O programador pode ainda declarar certas partes da memória como exclusivas a um processador o que constitui um simples, mas poderoso modelo para expressar e gerenciar o paralelismo em uma aplicação.

Apesar de sua simplicidade e da escalabilidade conseguida com avanços tecnológicos recentes, muitos desenvolvedores de aplicações paralelas resistem ao modelo e uma das razões diz respeito à portabilidade. Ao longo do tempo cada fornecedor de *hardware* para o modelo criou suas próprias extensões de C ou Fortran para programação paralela em memória compartilhada de forma que uma aplicação a ser portada de uma plataforma para outra precisasse ser reescrita. Esse capítulo apresenta uma alternativa portável para memória compartilhada, o OpenMP.

Pelo lado da programação com troca de mensagens, o MPI praticamente padronizou o modelo. O MPI é portável, disponível para diversas plataformas e é aceito como um padrão para esse tipo de aplicações. No entanto, troca de mensagens é uma técnica de programação difícil. Ela exige que os dados do programa sejam divididos explicitamente e então toda a aplicação deve ser paralelizada para trabalhar com os dados particionados. Não há uma forma incremental de paralelizar uma aplicação. Além disso, arquiteturas de multiprocessadores modernas têm gradualmente fornecido suporte a técnicas para memória compartilhada (*hardware* para coerência de cachê, por exemplo), o que torna a troca de mensagens uma opção às vezes desnecessária e muito restritiva.

Existe ainda o padrão *Pthreads* [54]. Esse é um padrão algumas vezes utilizado para memória compartilhada em sistemas de baixo nível. No entanto não está focado no campo de computação de alto desempenho. Há muito pouco suporte a *pthreads* em Fortran além de não ser uma estratégia escalável. Até mesmo para aplicações escritas em C, o modelo com *pthreads* é de nível baixo demais; muito mais do que o necessário para a maioria das aplicações científicas. Além disso, é um modelo voltado mais para paralelismo de instruções e não paralelismo de dados além de possuir portabilidade limitada.

Um outro ponto é que desenvolvedores de software para aplicações científicas e laboratórios governamentais possuem um grande volume de código que precisa ser paralelizado de forma portável. Os desenvolvedores precisam paralelizar tais códigos sem a necessidade de reescrevê-los completamente, mas isso não é possível com a maioria dos padrões de programação paralela. O OpenMP possibilita isso. O OpenMP é o modelo ideal para programadores que precisam paralelizar rapidamente aplicações científicas já existentes, mas é flexível o suficiente para atender a um conjunto muito maior de finalidades. O OpenMP fornece um caminho incremental para a conversão de software existente para execução em paralelo. Também fornece escalabilidade e performance para reescrever uma aplicação completamente ou para desenvolver uma nova aplicação [11].

3.1 Objetivos

Basicamente, OpenMP é um conjunto de diretivas de compilação e uma biblioteca de rotinas que estendem o Fortran e C/C++ para expressar paralelismo em termos de memória compartilhada. Ele não especifica a linguagem base e pode ser implementado em qualquer compilador. Vários fornecedores possuem produtos que suportam OpenMP, incluindo compiladores, ferramentas de desenvolvimento e ferramentas de análise de desempenho. O "*OpenMP Review Board*" inclui membros como Digita, HP, Intel, IBM. Todas essas companhias estão ativamente desenvolvendo compiladores e ferramentas para o OpenMP. [11]

Um compilador com suporte a OpenMP, transforma o código original em uma versão paralela para memória compartilhada, se guiando pelas anotações em forma de diretivas. O OpenMP define assim, um processo de paralelização automático, porém guiado pelo usuário. Isto significa que o compilador não precisa realizar uma vasta análise de código, já que se baseia apenas em informações fornecidas pelo programador. Isto dá ao usuário total controle sobre o que deve ser paralelizado e como, ao mesmo tempo que diminui sensivelmente a complexidade do compilador [3]

O OpenMP foi projetado para ser um padrão flexível e de fácil implementação em diferentes plataformas. O padrão tem quatros partes distintas [11].

Estrutura de controle:

O OpenMP possui um conjunto minimalista de estruturas de controle. A experiência indica que apenas algumas poucas são necessárias para escrever a maioria das aplicações paralelas. Dessa forma, o OpenMP inclui estruturas de controles apenas para aquelas situações onde o compilador pode oferecer funcionalidade e desempenho superiores ao que poderia ser desenvolvido pelo próprio programador.

Ambiente de dados:

Associado a cada tarefa existe um único ambiente de dados fornecendo um contexto para execução. A tarefa inicial possui um ambiente de dados que existe por tanto tempo quanto dure o programa. Ela constrói novos ambientes de dados apenas para aquelas tarefas criadas durante a execução do programa. Os objetos que fazem parte de um ambiente de dados podem ter um dos três atributos básicos: *shared*, *private*, *reduction*.

Sincronização:

Há dois tipos de sincronização: explícita ou implícita. A sincronização implícita ocorre no começo e no fim de blocos *parallel* e blocos de diretivas de controle. O usuário especifica sincronizações explícitas para gerenciar ordem ou dependência de dados. Sincronização é um tipo de comunicação entre processos e como tal pode afetar significativamente o desempenho do programa. Assim, em geral, quando a sincronização é minimizada obtêm-se melhor desempenho. Por essa razão, o OpenMP fornece um rico conjunto de funcionalidades de sincronização para que os programadores possam ajustar adequadamente a sincronização em suas aplicações.

Biblioteca de Rotinas:

O OpenMP fornece ainda uma biblioteca de rotinas e um conjunto de variáveis de ambiente. A biblioteca de rotinas inclui rotinas de *query* e *lock*. Além de permitir à uma aplicação especificar o modo como ela deve ser executada. As variáveis de

ambiente por sua vez, ajudam os programadores a criar além de aplicações portáveis, ambientes de execução também portáveis.

3.2 Modelo

O paralelismo no OpenMP é baseado em *threads* através do padrão *fork-join*. Todos os programas OpenMp iniciam com uma única *thread,* a *thread* mestre. A *thread* mestre executa seqüencialmente até que a primeira região paralela é encontrada. Ela então realiza um *fork* criando um conjunto de *threads* e assim as instruções originalmente delimitadas no código pela diretiva , são executadas em paralelo nas diversas *threads* do conjunto [28]. Quando as *threads* filhas completam sua execução das instruções da região paralela elas sincronizam e terminam, restando por fim apenas a *thread* mestre novamente (figura 3.1).

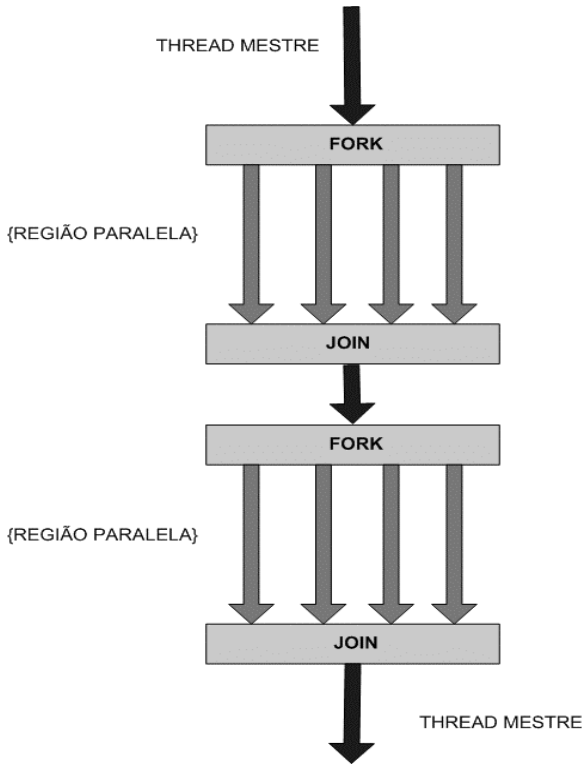

Figura 3.1: Modelo de Execução do OpenMP

3.3 Construções OPENMP

O OpenMP é de fácil utilização e consiste basicamente de dois tipos de construções: *pragmas* e rotinas de biblioteca. Os *pragmas* do OpenMP instruem o compilador para

paralelizar seções de código. Todos os *pragmas* do OpenMP começam com *#pragma omp*. Como toda diretiva *pragma*, essas também são ignoradas pelo compilador se ele não suporta a funcionalidade, nesse caso OpenMP [5].

A finalidade principal das rotinas do OpenMP é de atribuir e recuperar informações sobre o ambiente. Há também rotinas para alguns tipos de sincronização. Para utilizar as rotinas de biblioteca do OpenMP, o programa deve incluir o arquivo de *header* 'omp.h'. Se a aplicação utilizar apenas *pragmas*, o arquivo de *header* pode ser ignorado.

Os *pragmas* do OpenMP têm a seguinte forma (Figura 3.2):

```
#pragma omp <diretiva> [clausula[[,]clausula]...]
```

Figura 3.2: Formato de *pragmas* do OpenMP

As diretivas incluem o seguinte: *parallel*, for, *parallel* for, *section*, *sections*, *single*, *master*, *critical*, *flush*, *ordered*, e *atomic*. Essas diretivas especificam compartilhamento de trabalho entre *threads* ou instruções de sincronização. As cláusulas são opcionais e alteram o comportamento das diretivas. Cada diretiva possui um conjunto diferente de cláusulas disponível e cinco diretivas (*master*, *critical*, *flush*, *ordered* e *atomic*) nunca aceitam cláusulas [18] [5].

3.4 Especificação de Paralelismo

Apesar de existirem muitas diretivas, é possível escrever aplicações relevantes utilizando apenas algumas delas. A diretiva mais comum e importante é a diretiva *parallel*. Essa diretiva cria uma região paralela para o bloco estruturado que segue a diretiva. A figura 3.3 apresenta o formato da diretiva *parallel*.

```
#pragma omp <diretiva> [clausula[[,]clausula]...]
{
   //bloco estruturado
}
```

Figura 3.3: Formato da diretiva *parallel*

Essa diretiva diz ao compilador que o bloco estruturado de código deve ser executado em paralelo em múltiplas *threads*. Cada *thread* irá executar o mesmo fluxo de instruções, no entanto não necessariamente o mesmo conjunto de instruções, devido a possíveis instruções de controle de fluxo como *if-else*.

Segue-se, na figura 3.4, um exemplo que calcula a média de dois valores em um vetor e armazena o resultado em outro vetor. Aqui é introduzida uma nova diretiva OpenMP: *#pragma omp* for. Essa é uma diretiva de compartilhamento de trabalho que instrui o compilador para dividir as iterações do *loop* que se segue entre as *threads* do time.

```
#pragma omp parallel
{
      #pragma omp for
      for(int i = 1; i < size; ++i)
              x[i] = (y[i-1] + y[i+1])/2;
}
```

Figura 3.4: Exemplo de um trecho de código utilizando a diretiva *#pragma omp* for

Nesse caso, se size tem o valor 100 e o *loop* é executado em uma máquina com 4 processadores, as iterações do *loop* são alocadas de forma que o processador 1 fica responsável pelas iterações de 1 a 25, o processador 2 pelas iterações de 26 a 50, o processador 3 de 51 a 75 e o processador 4 pelas iterações de 76 a 99. Nesse caso é utilizada uma política estática de escalonamento, mas outras políticas podem ainda ser selecionadas.

Se o trecho de código não utilizasse o *pragma for*, então cada *thread* utilizaria o *loop* completo e realizariam trabalho redundante. Um erro desse tipo é apresentado na figura 3.5.

```
#pragma omp parallel // provavelmente essa não era a intenção
{
      for(int i = 1; i < size; ++i)
              x[i] = (y[i-1] + y[i+1])/2
      }
```

Figura 3.5: Paralelização incorreta de loop

A construção *for* possui uma forma abreviada para o *pragma* (Figura 3.6):

```
#pragma omp parallel for
for(int i = 1; i < size; ++i)
          x[i] = (y[i-1] + y[i+1])/2;
```

Figura 3.6: Forma abreviada da diretiva *for*.

Em todos os *loops* precedentes não existia dependência entre as iterações de *loop*. No exemplo que se segue na figura 3.7, existem duas dependências de *loop* diferentes:

```
for(int i = 1; i <= n; ++i)     // Loop (1)
    a[i] = a[i-1] + b[i];

for(int i = 0; i < n; ++i)      // Loop (2)
          x[i] = x[i+1] + b[i];
```

Figura 3.7: Dependências entre iterações de *loop*

A paralelização do *loop* 1 é problemática porque, para executar a iteração i do *loop*, é necessário o resultado da iteração i-1 e portanto há uma dependência entre a iteração i e a iteração i-1. A paralelização do *loop* 2 é problemática também, mas por uma razão diferente. Nesse caso é possível calcular o valor de x[i] antes do valor de x[i-1], mas fazendo isso não é possível calcular o valor de x[i+1]. Há uma dependência da iteração i-1 para a i.

Quando for realizada paralelização de *loops*, o programador deve se certificar de que não existem dependências entre as iterações. Quando não há dependências, o compilador pode executar o *loop* em qualquer ordem, inclusive em paralelo. Esse é um requisito importante porque o compilador não é capaz de fazer esse tipo de verificação [18].

3.5 Dados Privados e Compartilhados

Ao escrever programas paralelos, é muito importante entender quais dados são privados e quais dados são compartilhados para uma execução correta e para um desempenho adequado. No OpenMP essa distinção é bem clara. Se variáveis são compartilhadas por todas as *threads* no time, uma mudança em uma dessas variáveis realizada por uma *thread* é vista por todas as outras. Por outro lado, as variáveis privadas, possuem cópias exclusivas para cada uma das *threads*, de forma que mudanças realizadas por uma delas não são visíveis para as outras.

Por padrão, todas as variáveis são compartilhadas exceto em três exceções. Em primeiro lugar, para *loops* parallel *for*, o índice é privado. Em segundo, variáveis declaradas

dentro do bloco *parallel* também são privadas. E ainda em terceiro lugar, quaisquer variáveis listadas em uma das cláusulas de controle de escopo: *private*, *firstprivate*, *lastprivate*, ou *reduction* são privadas. Cada uma das cláusulas de controle de escopo de variáveis recebe uma lista de variáveis, mas cada uma possui uma semântica diferente. A cláusula *private* diz que cada variável em sua lista deve possuir uma cópia privada em cada uma das *threads*. A cópia será inicializada com o valor padrão do tipo (por exemplo, 0 para o tipo inteiro).

As cláusulas *firstprivate* e *lasprivate* possuem a mesma semântica da cláusula *private*, exceto que para *fistprivate* o valor da variável imediatamente antes da região paralela é copiado para cada umas das cópias e que para *lastprivate* o valor da variável é copiado para a cópia principal na *thread* mestre na última iteração. A cláusula *reduction* possui uma semântica similar à da cláusula *private*, mas recebe além de uma variável, um operador. O conjunto de operadores é limitado (+, *, -, &, |, ^, &&, ||) e as variáveis de redução devem ser do tipo escalar (*float*, *int* ou *long*). Ao final do bloco de código, o operador de redução é aplicado às cópias privadas das variáveis e ao seu valor original.

No exemplo da figura 3.8, o valor de *sum* é implicitamente inicializado em cada *thread* com o valor 0. Assim que o *bloco #pragma omp* for é completado, as *threads* aplicam o operador + para todas as cópias privadas e para o valor original e o resultado é atribuído para a variável *sum*, original da *thread* mestre [18].

```
float sum = 10.0f;
MatrixClass myMatrix;
int j = myMatrix.RowStart();
int i;

#pragma omp parallel
{
   #pragma omp for firstprivate(j) lastprivate(i) reduction(+:   sum)
   for(i = 0; i < count; ++i)
   {
      int doubleI = 2 * i;
      for(; j < doubleI; ++j)
      {
         sum += myMatrix.GetElement(i, j);
      }
   }
}
```

Figura 3.8: Redução no OpenMP

3.6 Paralelismo Não Relacionado a *Loop*

O OpenMP é tipicamente utilizado para paralelismo de *loop*, mas também suporta paralelismo em nível de funções. Esse mecanismo é chamado de sessões OpenMP e é útil em várias situações. Segue-se na figura 3.9 um trecho de código de uma rotina de *QuickSort* que utiliza seções:

```
void QuickSort (int numList[],  int nLower, int nUpper)
{
   if (nLower < nUpper)
   {
      // create partitions
      int nSplit = Partition (numList, nLower, nUpper);

      #pragma omp parallel sections
      {
         #pragma omp section
         QuickSort (numList, nLower, nSplit - 1);

         #pragma omp section
         QuickSort (numList, nSplit + 1, nUpper);
      }
   }
}
```

Figura 3.9: QuickSort utilizando Sections no OpenMP

Nesse exemplo, o primeiro *#pragma* cria uma região paralela com sessões. Cada sessão é precedida por uma diretiva *#pragma omp section*. Cada sessão na região paralela é atribuída a uma única *thread* do time e todas elas podem executar concorrentemente [18].

3.7 Pragmas de Sincronização

Com várias *threads* executando concorrentemente, muitas vezes é necessário sincronizá-las. O OpenMP suporta vários tipos de sincronização. Um tipo de sincronização importante é a barreira implícita ao final de uma região *parallel*. Uma sincronização de barreira exige que todas as *threads* alcancem aquele ponto antes que qualquer uma possa continuar.

Há ainda uma barreira implícita ao final de cada bloco *#pragma omp for* e *#pragma omp single*. Para remover essa barreira implícita basta incluir a cláusula *nowait* (Figura 3.10):

```
#pragma omp parallel
{
   #pragma omp for nowait
   for(int i = 1; i < size; i++)
      x[i] = (y[i-1] + y[i+1])/2;
}
```

Figura 3.10: Remoção de barreira implícita utilizando a cláusula *nowait*.

Um outro tipo de sincronização é realizado com barreiras explícitas. Em algumas situações, é necessário incluir uma barreira em algum ponto da região paralela. Isso é incluído no código com a diretiva *#pragma omb barrier*

Além disso, em uma região paralela, às vezes pode ser necessário limitar o acesso a uma única *thread*, como por exemplo, se for necessário escrever em um arquivo no meio de uma região paralela. Em muitos desses casos, não importa qual *thread* executa o código contanto que apenas uma o faça. O OpenMP possui a diretiva *#pragma omp single* para essa finalidade. Há ainda uma diretiva muita parecida *#pragma omp master* que especifica que a *thread* única que utilizará o trecho de código será a *thread* mestre [18]. A figura 3.11 apresenta um exemplo de uso da diretiva *#pragma omp single*.

```
#pragma omp parallel
{
   if(omp_get_thread_num() > 3)
   {
      //Não pode ser acessado por todas as threads
      #pragma omp single
      x++;
   }
}
```

Figura 3.11: Uso da diretiva #pragma omp single

3.8 Rotinas de Biblioteca

Além das diretivas de compilação, O OpenMP possui também um conjunto de rotinas muito útil para o desenvolvimento de aplicação. Existem três classes de rotinas disponíveis: rotinas de ambiente, rotinas de sincronização e rotinas de tomada de tempo. Todas essas rotinas começam com *omp_* e são definidas no arquivo de *header omp.h.*

As rotinas de ambiente permitem ao programador recuperar e alterar vários aspectos do ambiente operacional onde o OpenMP está sendo executado. Funções que começam com *omp_set_* devem, de uma forma geral, ser executadas apenas fora da região paralela. Todas as outras podem ser executadas em trechos de códigos paralelos ou não paralelos.

Algumas das funcionalidades da biblioteca do OpenMP incluem, por exemplo, especificar ou recuperar o número de *threads* no time atual *(omp_set_num_threads, openmp_get_num_threads*) ou recuperar o número de processadores disponíveis.

Outra construção de sincronização utilizada é *lock*, que controla o acesso a seções críticas de código. Existem dois tipos de *locks* :*simple* e *nestable* e cada *lock* pode existir em um dos três estados: não-inicializado, bloqueado e desbloqueado.

Locks simples (*omp_lock_t*) não podem ser obtidos mais de uma vez, mesmo pela mesma *thread*. *Locks nestable* (*omp_nest_lock_t*) são idênticos aos *locks* simples, exceto quando uma *thread* tenta obter o *lock* que ela já possui, ela não será bloqueada. Além disso, *locks nestable* são contadores e mantêm registro de quantas operações de set foram realizadas sobre eles.

Existem rotinas de sincronização que atuam sobre esses *locks*. Para cada rotina existe uma variante simples em uma *nestable*. Existem cinco ações que podem ser realizadas em um lock: *initialize*(inicializa), *set* (adquire um *lock*), *unset* (libera um *lock*), *test* (verifica se o *lock* está livre) e *destroy* (destrói o lock).

Os programadores podem escolher tanto as rotinas de biblioteca como os *pragmas* para sincronização. A vantagem dos *pragmas* é que eles são extremamente estruturados. Isso torna mais fácil a compreensão dos programas já que olhando para o código é fácil ver o local de entrada e saída das regiões críticas.

Já as rotinas de biblioteca têm como grande vantagem a flexibilidade. É possível, por exemplo, passar um *lock* como parâmetro para uma outra função e dentro dessa função realizar uma operação de *set* ou *unset*. Isso não é possível com os *pragmas*. De uma forma geral, faz mais sentido usar os *pragmas* de sincronização a menos que seja necessário um grau maior de flexibilidade [18] [5].

3.9 Vantagens e Desvantagens do Padrão OpenMP

Uma primeira desvantagem do OpenMP é que com ele uma aplicação só pode ser executada em um espaço de endereçamento único, ou seja, não é possível executar uma aplicação totalmente OpenMP em um *cluster*. Além disso, OpenMP é construído em cima de um modelo nativo de *threads* e por isso adiciona *overhead* à aplicação. Um outro ponto importante é que, pelo fato da paralelização ser gerada pelo compilador, é muito fácil escrever trechos de código incorretos em relação a condições de corridas e *deadlocks* entre outros problemas.

Quanto às vantagens, talvez a principal seja a simplicidade e o pouco esforço de programação exigido, se comparado a outros modelos. Além disso, a paralelização de uma aplicação seqüencial já existente pode ser feita de forma incremental pela anotação do código com diretivas. Isso elimina a necessidade de reescrever o código. O código original nesse caso ainda é preservado já que se o programa for compilado sem a opção de suporte a OpenMP, todas as diretivas incluídas são ignoradas. Ainda como vantagens pode-se citar a portabilidade, já que a maioria dos fornecedores de *hardware* possui compiladores com suporte ao padrão e ainda o mapeamento natural do modelo OpenMP em arquiteturas SMP [56].

Saber quando usar o OpenMP é quase tão importante quanto saber como usá-lo. Os seguintes pontos são úteis nessa decisão [18]:

- Plataforma alvo é multiprocessador ou *multicore*: Nesse caso, se a aplicação estiver saturando um núcleo ou processador, transformá-la em uma aplicação *multithread* com o OpenMP irá melhorar o desempenho da aplicação.

- Aplicações Multi-Plataforma: O OpenMP é multiplataforma e uma API largamente utilizada e, como usa *pragmas*, a aplicação pode ser compilada mesmo se o compilador não suportar o padrão OpenMP

- Paralelização de *Loops*: O OpenMP é ideal para a paralelização de *loops*. Se a aplicação possui *loops* caros computacionalmente e que não possuem dependências entre as iterações, usar OpenMP é a escolha ideal.

- Otimização de última hora necessária. Pelo fato do OpenMP não exigir a reconstrução da aplicação, é uma ferramenta perfeita para realizar pequenas mudanças para melhorar o desempenho.

- OpenMP não é para todo problema *multithread*. Ele foi desenvolvido para ser usado em computação de alto desempenho e funciona melhor em estilos de programação que possuem código com muitos *loops* e *arrays* de dados compartilhados.

- A criação das *threads* para as regiões paralelas do OpenMP implica em algum *overhead*. Para que ocorra um ganho de desempenho, o *speedup* obtido pela região paralelizada deve se sobrepor ao *overhead* de inicialização das *threads*.

- As *pragmas* do OpenMP, apesar da facilidade de programação, não oferecem um bom *feedback* quando ocorrem erros. Se uma aplicação crítica precisa ser desenvolvida e esta precisa detectar falhas e se recuperar delas, então o OpenMP provavelmente não é uma boa escolha (pelo menos em sua versão atual). Uma das melhorias previstas para as próximas versões é um mecanismo de tratamento de erros mais consistente.

CAPÍTULO 4

PROGRAMAÇÃO POR TROCA DE MENSAGENS NO PADRÃO MPI

Troca de mensagens é a principal estratégia para comunicação de dados em multiprocessadores. Um sistema de troca de mensagens tipicamente combina uma memória local e um processador em cada nó, além de uma rede de interconexão.

Nesse modelo não há memória compartilhada global e para movimentar dados de uma memória local para outra é necessário trocar mensagens através da rede de interconexão. Isso é geralmente realizado através de pares de comandos *send/receive* que devem ser explicitamente chamados no código da aplicação. Isso elimina a necessidade de uma grande memória global, assim como os seus requisitos de sincronização.

Dois fatores importantes devem ser considerados no projeto de redes de interconexão para sistemas de troca de mensagens: a largura banda do *link* e a latência de rede. A largura de banda do link é definida como o número de *bits* que podem ser transmitidos por unidade de tempo (*bits/s*). Já a latência de rede é definida como o tempo para completar a transferência de uma mensagem.

Quando uma determinada aplicação é executada nesse modelo, o programa é dividido em tarefas concorrentes e cada uma pode ser executada em um processador diferente. Se o número de tarefas é maior que o número de processadores, então algumas delas terão que dividir um único processador em um esquema de compartilhamento de tempo. Tarefas executadas em um único processador utilizam canais internos para troca de mensagens enquanto tarefas em processadores diferentes usam canais externos.

Uma vantagem importante desse tipo de troca de dados é a eliminação da necessidade de construções de sincronização como *locks* e semáforos, o que resulta em um melhor desempenho. Além disso, um esquema de troca de mensagens possui uma melhor escalabilidade.

Uma mensagem é definida como uma unidade lógica para comunicação interprocessos. Cada mensagem é considerada como um conjunto de informações relacionadas enviadas como uma entidade.

Uma mensagem pode ser uma instrução, dados, sincronização ou um sinal de interrupção. Um sistema de troca de mensagens interage como o mundo exterior recebendo

mensagens de entrada e enviando mensagens de saída. É essencial que o mundo exterior perceba um comportamento consistente em um sistema desse tipo.

4.1 O Padrão MPI

O MPI [35] é um padrão de uma interface de troca de mensagens para programação paralela em memória distribuída. O MPI inclui rotinas de comunicação ponto a ponto, operações coletivas, assim como suporte a grupos de tarefas, contextos de comunicação e topologias de aplicação.

O principal objetivo do MPI é fornecer portabilidade entre diferentes plataformas. O mesmo código fonte de troca de mensagens pode ser executado em diversas arquiteturas de *hardware* e sistemas operacionais desde que a biblioteca MPI esteja disponível. Apesar de troca de mensagens ser geralmente associada a computadores paralelos com memória distribuída, o mesmo código pode também ser executado em computadores de memória compartilhada. Ele pode ainda ser executado em uma rede de estações de trabalho ou até mesmo em uma única máquina multiprocessada. O conhecimento de que existem implementações eficientes para o MPI para uma grande variedade de plataformas possibilita uma grande flexibilidade no desenvolvimento e na escolha do ambiente a ser utilizado.

Um outro tipo de compatibilidade possibilitada pelo MPI é a habilidade de executar aplicações de forma transparente em sistemas heterogêneos, isto é, coleções de processadores com arquiteturas de *hardware* distintas. Isso, graças a um modelo de máquinas virtuais que esconde as diferenças de arquitetura. O usuário não precisa se preocupar se o código está enviando mensagens entre processadores de uma arquitetura semelhante ou diferente.

Portabilidade é fundamental, mas um padrão não seria largamente utilizado se isso fosse obtido em detrimento do desempenho. Um ponto crucial é que o MPI foi cuidadosamente projetado para possibilitar o desenvolvimento de implementações eficientes. As decisões de projeto aparentemente foram corretas, já que muitas plataformas executam aplicações MPI com excelentes desempenhos.

Outro objetivo importante para processamento paralelo diz respeito à escalabilidade. O MPI possibilita uma boa escalabilidade através de várias características de seu projeto. Por exemplo, uma aplicação pode criar subgrupos de tarefas, que por sua vez, possibilitam operações de comunicação controladas e restritas a alguns delas, possivelmente envolvidas em um escopo específico.

Finalmente, MPI, define um comportamento bem conhecido e mínimo para implementações de troca de mensagens. Isso tira das costas do programador a preocupação com certas questões de nível mais baixo e o deixa livre para se concentrar na lógica do problema a ser resolvido. Por exemplo, o MPI garante que a transmissão de mensagens é confiável liberando o programador do trabalho de verificar se cada mensagem foi devidamente recebida [51].

4.2 Plataformas

O maior atrativo do paradigma de troca de mensagens é a sua portabilidade. Programas escritos dessa forma podem ser executados em multicomputadores de memória distribuída, multiprocessadores de memória compartilhada, redes de *worksations* e combinações de todos esses. O paradigma não se torna obsoleto em arquiteturas que combinam as visões de memória distribuída e compartilhada nem por aumentos expressivos na velocidade das tecnologias de comunicação. Assim é possível e eficiente a utilização desse padrão nas mais diversas variedades de máquinas, incluindo aquelas constituídas de uma coleção de outras máquinas, paralelas ou não, conectadas por uma rede de comunicação.

O padrão também é adequado para programas escritos em um estilo mais restrito de SPMD (*Single Program Multiple Data*), onde todos os processadores executam o mesmo fluxo de instruções. Apesar de não ser fornecido suporte explícito para *threads*, o projeto do MPI não prejudica o seu uso. O padrão possui ainda uma série de características que melhorariam o desempenho e a escalabilidade de *hardware*s de comunicação interprocessos especializados. Assim, espera-se que implementações da interface para essas máquinas sejam criadas em breve. Enquanto isso, as implementações do MPI para protocolos de comunicação interprocessos padrão Unix possibilitam portabilidade e comunicação eficiente para aplicações em *clusters* e redes de *workstations* [51].

4.3 Aplicação MPI

Uma aplicação MPI pode ser visualizada como uma coleção de tarefas concorrentes comunicantes. Um programa inclui código escrito pelo programador da aplicação que é ligado à biblioteca de funções do MPI. A cada tarefa é associado um *rank*. Esses *ranks* são utilizados pelas tarefas MPI para identificar umas às outras em cooperações que venham a ser realizadas. Tarefas MPI podem ser executadas no mesmo processador ou em processadores

diferentes de forma concorrente como ilustrado na figura 4.1. Enviar uma mensagem para uma tarefa na mesma ou em uma máquina diferente é transparente para a aplicação. O MPI automaticamente seleciona o mecanismo de comunicação disponível mais eficiente em uma máquina ou entre máquinas. O uso de *ranks* torna todas as operações de cooperação independentes da localização física dos participantes. No decorrer do texto tarefas MPI também serão chamadas de processos MPI.

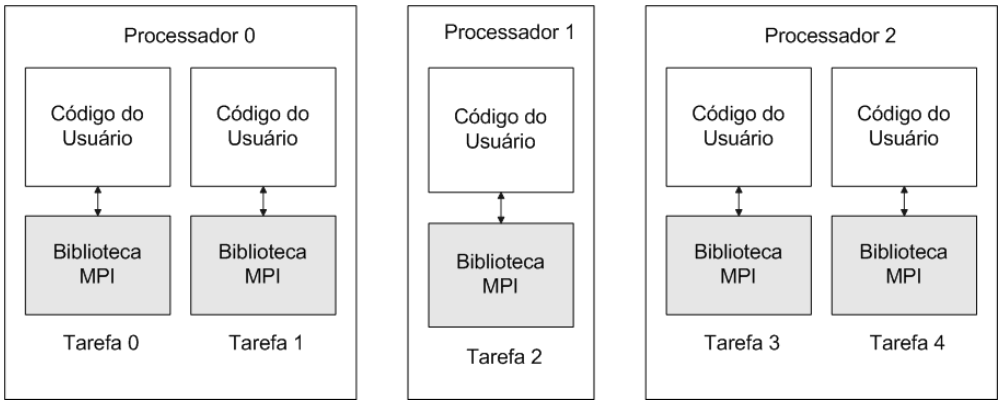

Figura 4.1: Aplicação MPI

4.4 Grupos, Contextos e Comunicadores

4.4.1 Grupos de Tarefas

Um grupo de tarefas é uma coleção ordenada na qual cada tarefa é identificada pelo seu *rank* dentro da ordem. Para um grupo de n tarefas os ranks variam de 0 até n-1.

Grupos de tarefas são usados com duas finalidades importantes. Em primeiro lugar, eles são utilizados para especificar quais tarefas estão envolvidos em uma operação coletiva de comunicação, como um broadcast, por exemplo. Em segundo lugar, eles são utilizados para introduzir paralelismo no código de forma que grupos diferentes resolvam partes diferentes da aplicação. Se isso é feito carregando-se códigos executáveis diferentes para cada grupo, então temos um paralelismo de tarefas MIMD. Por outro lado, se cada grupo executa um desvio condicional dentro do mesmo executável, temos um paralelismo de tarefas SIMD (paralelismo de controle). A especificação inicial do MPI adota um modelo estático de tarefas de forma que existe um número fixo delas do início até o fim da execução do programa [14].

Apesar do modelo de tarefas do MPI ser estático, os grupos são dinâmicos já que eles podem ser criados e destruídos durante a execução e cada tarefa pode pertencer a vários

grupos simultaneamente. No entanto, os membros de um grupo não podem ser alterados e, portanto, é necessário criar um novo grupo para alterar os membros de um já existente.

Em MPI um grupo é um objeto referenciado através de um *handle*. O MPI possui rotinas para criar novos grupos especificando os *ranks* ou particionando um grupo existente através de uma chave. Outras rotinas consultam o *rank* de uma determinada tarefa dentro de um grupo específico, testam se uma tarefa faz parte de um determinado grupo, realizam sincronização de barreira em um grupo e consultam seu tamanho e os membros que a ele pertencem [35].

4.4.2 Contextos de Comunicação

Contextos de comunicação foram propostos inicialmente para permitir a criação de fluxos de mensagens distintos e bem delimitados entre tarefas, com cada fluxo pertencendo a um único contexto. Um uso importante dos contextos é garantir que mensagens enviadas em uma fase da aplicação não sejam incorretamente interceptadas em outra fase. Os contextos representam um critério adicional para seleção de mensagens e possibilitam a criação de espaços de *tags* de mensagens independentes.

O usuário realiza operações explícitas sobre contextos, já que não existe um tipo de dados visível para eles. No entanto, os contextos estão associados aos comunicadores de forma transparente ao usuário de maneira que mensagens enviadas através de um determinado comunicador só podem ser recebidas através do comunicador correto correspondente [35].

4.4.3 Comunicadores

O escopo de uma operação de comunicação é especificado pelo contexto de comunicação utilizado e o grupo ou grupos envolvidos. Em uma operação coletiva ou ponto a ponto entre os membros de um mesmo grupo, apenas este precisa ser especificado e as tarefas fonte e destino são identificadas por seus *ranks* dentro do grupo. Em uma comunicação ponto a ponto entre tarefas de grupos diferentes, os dois grupos precisam ser especificados e nesse caso tarefas fonte e destino são identificadas pelo seu *rank* em seu respectivo grupo. No MPI um objeto abstrato chamado Comunicador é usado para definir o escopo de uma operação de comunicação. Comunicadores usados em comunicação intra-grupos ou inter-grupos são chamados de intra ou inter-comunicadores respectivamente. Um intra-comunicador pode ser considerado como uma ligação entre um contexto e um grupo enquanto um inter-comunicador

liga um contexto e dois grupos, um com a tarefa fonte e outro com a tarefa destino. Os comunicadores são passados como parâmetro para todas as rotinas ponto a ponto ou coletivas para especificar o contexto e o grupo ou grupos envolvidos na operação [35].

4.5 Topologias de Aplicação

Em muitas aplicações, as tarefas são organizados em uma topologia particular como um grid bidimensional ou tridimensional, por exemplo. O MPI suporta diversas topologias, especificadas por grafos nos quais tarefas que se comunicam são ligadas por um arco. Por conveniência, o padrão já inclui suporte explícito para grids cartesianos n-dimensionais. No MPI um grupo possui uma topologia cartesiana, uma topologia de grafo ou não possui topologia. A figura 4.2 apresenta uma topologia em grafo para cinco tarefas.

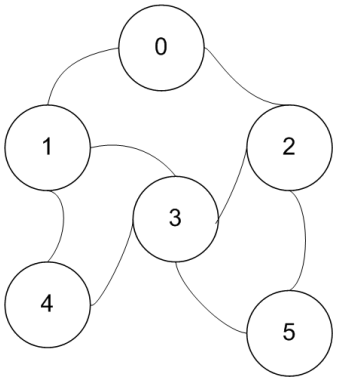

Figura 4.2: Topologia em Grafo para uma aplicação MPI

4.6 Comunicação Ponto a Ponto

O MPI utiliza um rico conjunto de mensagens de envio e recebimento de mensagens e a comunicação entre tarefas envolve os seguintes componentes:

1- Remetente, geralmente identificado por seu *rank*.

2- Receptor, geralmente identificado por seu *rank*.

3- Dados da mensagem

4- *Tag* da mensagem. Ajuda a distinguir múltiplas mensagens entre duas tarefas que devem ser tratadas de forma diferente ou em uma ordem pré-estabelecida.

5- O comunicador que fornece um contexto para a comunicação

O MPI, para comunicação ponto a ponto, seleciona mensagens explicitamente através de suas tarefas fonte, *tag* de mensagens e contextos de comunicação. A tarefa fonte e a *tag* podem ser ignoradas na seleção de mensagem, o contexto de comunicação não. As tarefas fonte e destino são especificadas através de um grupo e um *rank*. Para comunicações intra-grupos, o grupo e o contexto são ligados em um intra-comunicador enquanto para uma comunicação inter-grupos a fonte e o destino são ligados em um inter-comunicador. Assim, uma rotina de *send* recebe como parâmetros um comunicador, o *rank* da tarefa destino e o tipo de mensagem para especificar o contexto e o destino de uma mensagem. Uma rotina de *receive* recebe os mesmos três parâmetros para selecionar a mensagem que deve receber [14].

Comunicação ponto a ponto envolve a transmissão de uma mensagem entre um par de tarefas. O MPI possui uma grande variedade de rotinas de comunicação ponto a ponto, ao contrário de outras bilibotecas de troca de mensagens que em geral possuem apenas um método de comunicação desse tipo. Isso dá ao programador muito mais controle sobre como as mensagens serão tratadas.

De uma forma geral as rotinas de comunicação ponto a ponto podem ser classificadas quanto ao bloqueio (bloqueantes ou não bloqueantes) e quanto ao modo de comunicação (padrão, síncrono, bufferizado e *ready*). Quanto ao bloqueio, as rotinas bloqueantes garantem que a tarefa transmissora ou receptora ficará bloqueada até que a transmissão da mensagem seja completada. Caso contrário, tem-se uma rotina não bloqueante.

Quanto ao modo de comunicação existem quatro modos possíveis: modo síncrono, modo bufferizado, modo *ready* e modo padrão. No modo síncrono, o receptor deve enviar uma confirmação de recebimento de mensagem, de maneira que o transmissor possa ter certeza de que a mensagem foi recebida. No modo *bufferizado*, a transmissão de uma mensagem utilizando *buffers* permite que esta se complete rapidamente já que depois de copiada para o *buffer*, fica a cargo do sistema transmitir a mensagem quando possível. No modo *ready*, a operação é finalizada rapidamente sem a utilização de *buffers* ou de confirmações da tarefa receptora, objetivando um desempenho melhor para ambientes computacionais específicos. No modo padrão, o término de uma operação de comunicação pode ou não significar que o receptor foi ativado [35] [51].

4.7 Sincronização

Construções de sincronização são utilizadas para forçar uma determinada ordem de execução entre atividades de tarefas paralelas. Em alguns casos, em algum ponto da execução, é necessário que algumas tarefas paralelas realizem sincronização com outras tarefas. No MPI as operações de sincronização ocorrem devido a operações de troca de mensagens bloqueantes e através de operações de barreira.

Quando é utilizada, uma operação de *receive* bloqueante, força a tarefa que está recebendo a esperar até que a mensagem seja recebida. Se for utilizado o modo síncrono de comunicação, as duas tarefas devem se encontrar em um ponto de sincronização. Se tanto a rotina de envio quanto a de recebimento são bloqueantes, então a rotina de comunicação não se completará até que ambos, o remetente e o receptor se encontrem.

Tarefas em um grupo podem também ser sincronizadas em um ponto da execução através de uma barreira. Nenhuma tarefa pode continuar sua execução além de uma barreira, até que todas as outras tenham chegado até ela, como mostrado na figura 7.3(a). O grupo pode incluir todas as tarefas ou apenas um subconjunto das tarefas, dependendo do comunicador (figura 7.3 (b)). Na prática, quando uma tarefa chama a rotina de barreira ele se bloqueia e permanece assim até que todas as tarefas do subconjunto em questão tenham chamado a mesma rotina.

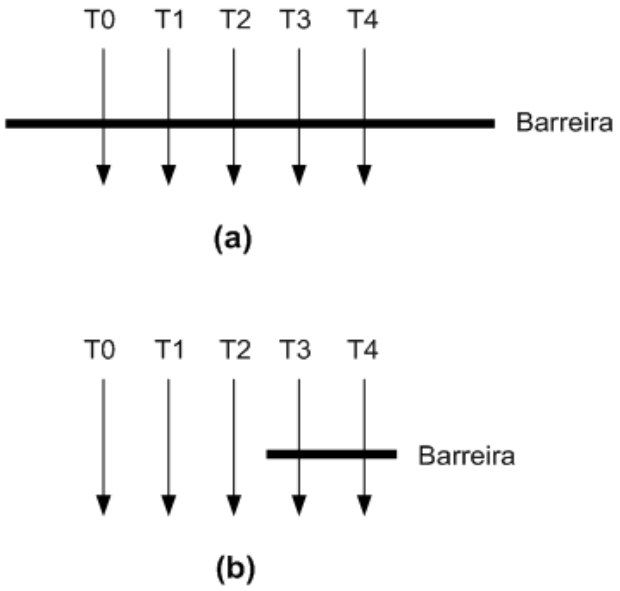

Figura 4.3: Sincronização de barreira; (a) para todas as tarefas do comunicador;
b) para apenas um subconjunto das tarefas no comunicador.

4.8 Comunicação Coletiva

Essas rotinas possibilitam comunicação coordenada entre um grupo de tarefas. O grupo de tarefas e o contexto são especificados pelo intra-comunicador passado como parâmetro para a rotina. As operações coletivas do MPI forma projetadas de tal forma que sua sintaxe e semântica são consistentes com as das rotinas ponto a ponto. As operações coletivas do MPI não possuem um argumento *tag* e assim devem ser chamadas por todos os membros do grupo. Assim que uma tarefa realiza seu papel na comunicação coletiva, ela pode continuar com outras instruções. Existem três tipos dessas operações: controle de tarefas, computação global, e movimento de dados. A função de barreira, discutida na seção anterior, pode ser classificada como uma operação coletiva de controle de tarefas. No restante dessa seção, serão apresentadas as rotinas de computação global e de movimentação de dados. O MPI não inclui operações de comunicação coletiva não bloqueantes [35].

4.8.1 Rotinas de movimentação de dados

Existem três tipos básicos de rotinas coletivas de movimentação de dados: *broadcast*, *scatter* e *gather*. Existem duas versões para cada uma delas: *Um para todos* e *todos para todos* [51].

A versão *um para todos* do broadcast transmite dados de uma tarefa para todas as outras no grupo. Já na versão *todos para todos* os dados são transmitidos de cada tarefa para todas as outras. Assim, cada uma das tarefas termina a operação com o mesmo *buffer* de saída que nada mais é do que a concatenação dos *buffers* de envio de cada tarefa na ordem dos *ranks*. A figura 4.4 apresenta uma operação de *broadcast um para todos*.

A versão um para todos da rotina *scatter* envia dados distintos de uma tarefa para todas as outras no grupo. Essa operação também é chamada de *comunicação um para todos personalizada.* No caso da rotina *scatter todos para todos* cada uma das tarefas envia dados para todas as outras no grupo sendo que esses dados diferem de uma tarefa destino para outra. É também conhecida como *comunicação todos para todos personalizada*. A figura 4.5 apresenta a uma operação de *scatter*.

Os padrões de comunicação na rotina *gather* são os mesmos da rotina *scatter*, exceto que a direção do fluxo de dados é reversa. Na versão *um para todos* do *gather*, uma tarefa recebe dados de todos as tarefas do grupo.

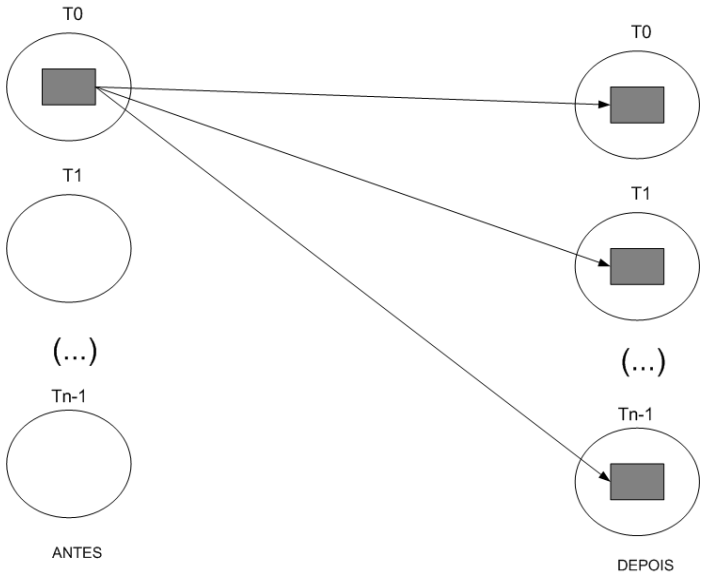

Figura 4.4: *Broadcast* um para todos

A tarefa raiz recebe a concatenação do buffer de entrada de todas as tarefas na ordem do *rank*. A versão *todos para todos* do *gather* é idêntica à versão *todos para todos* do *scatter* [14]. A figura 4.5(b) apresenta uma operação de *gather*.

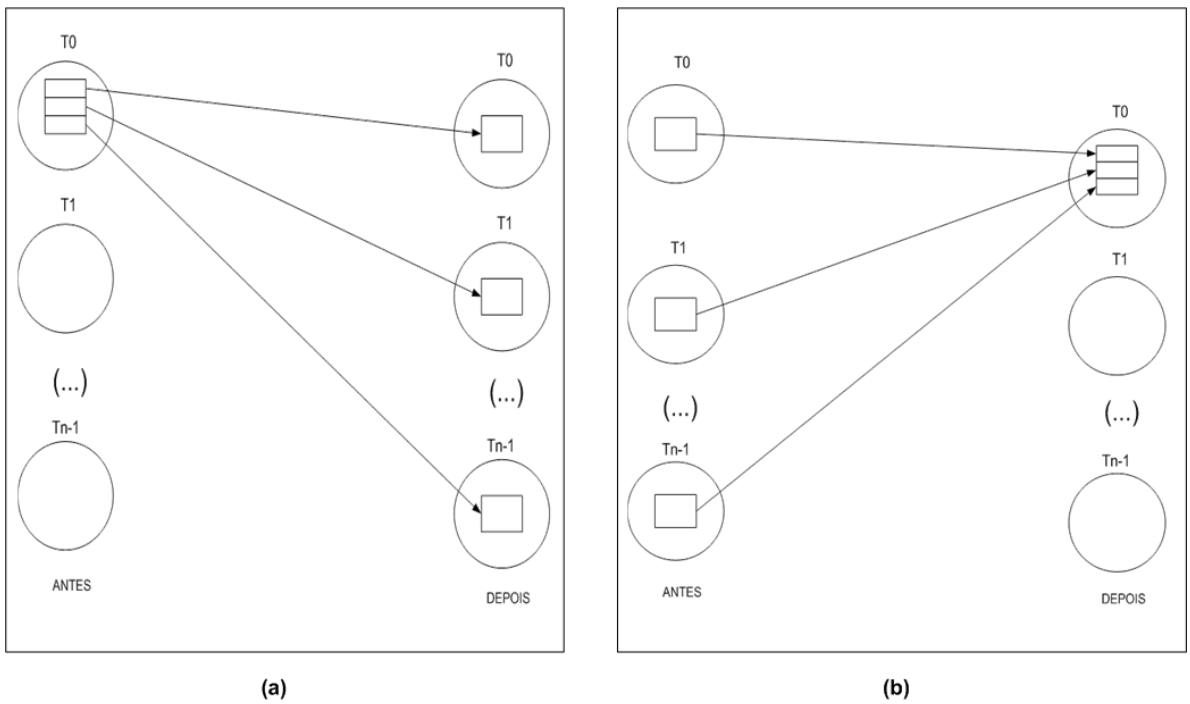

Figura 4.5: (a) Operação de *scather*; (b) Operação de *gather*

4.8.2 Rotinas de Computação Global

Existem duas rotinas básicas de computação global no MPI: *reduce* e *scan*. Ambas requerem a especificação de uma função para a computação. Existe uma versão na qual o usuário seleciona uma função de uma lista pré-definida e outra na qual o usuário fornece um ponteiro para função.

A operação *reduce* combina os elementos no buffer de entrada de cada tarefa no grupo usando a operação especificada e retorna o valor combinado no buffer de saída da tarefa cujo *rank* foi especificado como o *root* da operação. A figura 4.6 apresenta uma operação de reduce.

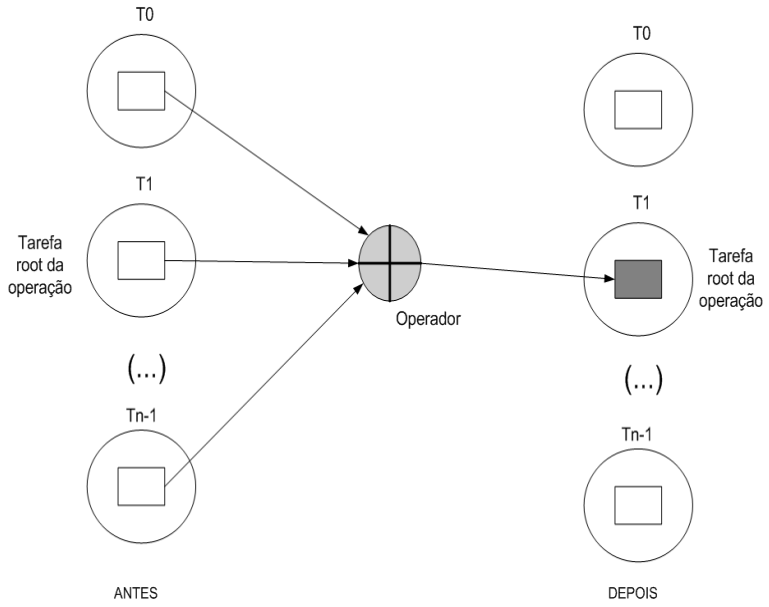

Figura 4.6: Operação de *reduce*.

Existe uma variação da rotina *reduce*, chamada *allReduce* que ao invés de retornar o valor da computação global apenas para a tarefa *root*, retorna para o *buffer* de todas as tarefas. A figura 4.7 apresenta uma operação de *allReduce*.

A Existem dois tipos de operações de *scan*: préfixada e posfixada. O resultado de uma operação de *scan* é diferente em cada tarefa, de acordo com o *rank* da tarefa. Por exemplo, sejam *T0*, *T* ... *Tn*-1 os membros de um grupo contendo os dados *d0*, *d1*... *dn*-1 respectivamente e um operador φ. O resultado em uma tarefa *Ti* será *d0* φ *d1* φ *d2* φ ... *di* para o *scan* prefixado. No *scan* pós-fixado, o resultado para a tarefa *Ti* será *di* φ *di*-1 φ *di*+1 φ ... φ *dn*-1. A figura 4.8 apresenta uma operação de *scan* préfixada.

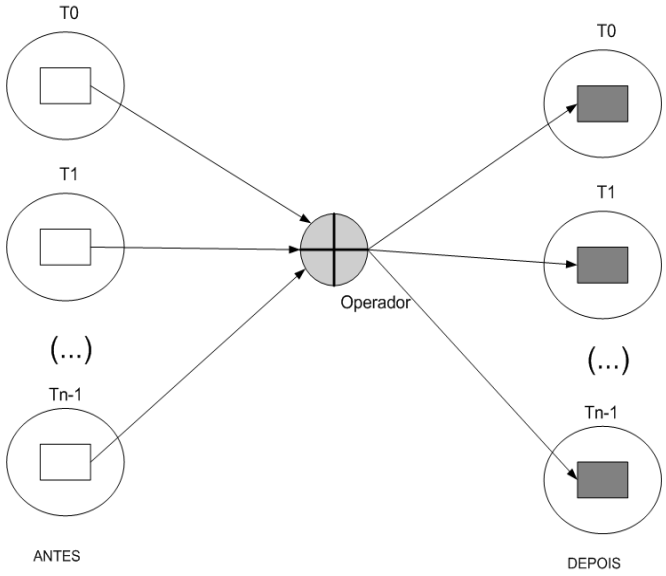

Figura 4.7: Operação de *allReduce*.

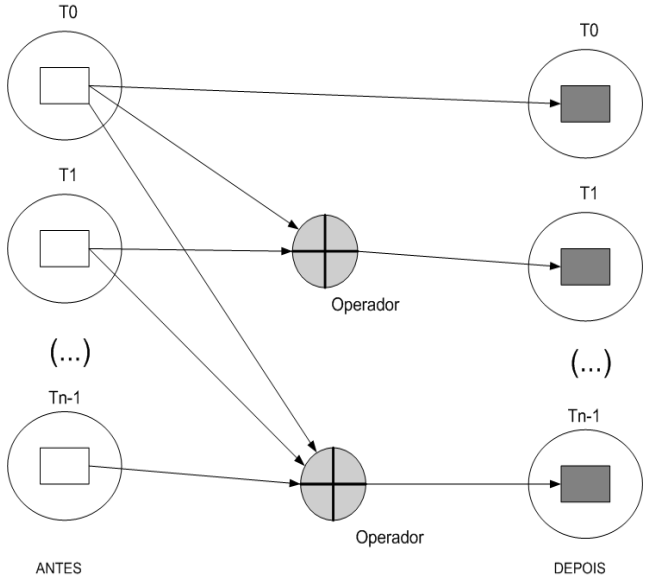

Figura 4.8: Operação de *scan* prefixada.

As operações pré-definidas suportadas pelas operações de *reduce* e *scan* são: máximo, mínimo, soma, produto, *AND* lógico, *AND bitwise*, *OR* lógico, *OR bitwise*, *XOR* lógico, *XOR bitwise* [14].

CAPÍTULO 5

MODELOS HÍBRIDOS DE PROGRAMAÇÃO PARALELA

Avanços tecnológicos recentes tornaram possível diversos processadores acessarem um único espaço de memória de forma eficiente e recolocaram as arquiteturas de memória compartilhada em foco na área de computação de alto desempenho. Cresce também nessa área a tendência de agrupar sistemas SMP em *clusters* em busca de um tempo de processamento ainda mais rápido [22] [55]. Essa tendência torna desejável que aplicações desenvolvidas para *clusters* de SMPs sejam portáveis e eficientes.

Aplicações baseadas em troca de mensagens, escritas com bibliotecas como o MPI, possuem naturalmente excelente portabilidade e podem ser utilizadas em *clusters* de SMPs sem maiores problemas. No entanto, apesar de ser claro que para a comunicação entre as máquinas SMP, a troca de mensagens é uma boa estratégia, o mesmo não se pode dizer para a comunicação realizada dentro das máquinas SMP.

Teoricamente um modelo de programação específico para memória compartilhada como o OpenMP seria uma estratégia mais eficiente para a comunicação dentro de uma máquina SMP. Portanto, uma combinação dos paradigmas de memória compartilhada e troca de mensagens na mesma aplicação tem potencial de oferecer um melhor desempenho do que uma abordagem MPI pura [49].

5.1 Arquiteturas SMP simples e em *Clusters*

Em arquiteturas de memória compartilhada (SMP – Multiprocessador simétrico), diversos processadores compartilham fisicamente um único espaço de endereçamento de memória. Isso pode limitar a escalabilidade do sistema porque o acesso à memória se torna um gargalo à medida que o número de processadores aumenta. A figura 5.1 mostra um sistema SMP.

Apesar do problema de contenção no acesso à memória, é mais simples programar no modelo de memória compartilhada do que no modelo com troca de mensagens já que o programador não precisa se preocupar com questões como partição e distribuição dos dados entre os processadores.

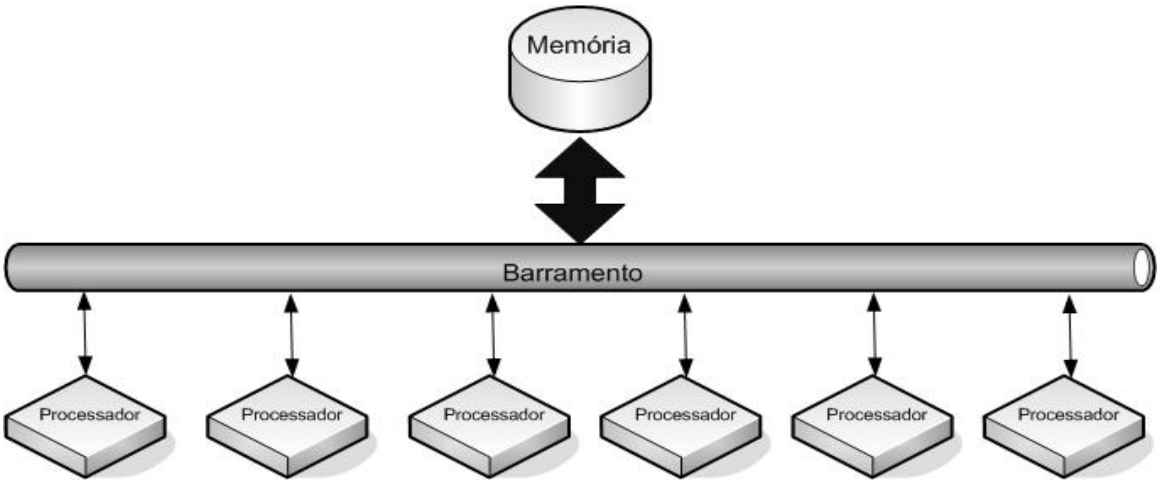

Figura 5.1: Representação Memória Compartilhada (SMP)

Um multiprocessador simétrico, no entanto, não atende aos requisitos de desempenho de diversas aplicações muito complexas. Unir vários sistemas SMP é uma forma de aumentar o número total de processadores disponíveis e conseqüentemente fornecer poder computacional suficiente para esse tipo de aplicação [61].

Clusters de SMPs podem ser descritos como uma união de memória compartilhada e memória distribuída em uma mesma arquitetura de *hardware*. Eles são constituídos de alguns *nós* SMP conectados através de uma rede, onde cada um deles contém dois ou mais processadores que compartilham memória fisicamente [4]. A figura 5.2 apresenta um *cluster* de SMPs.

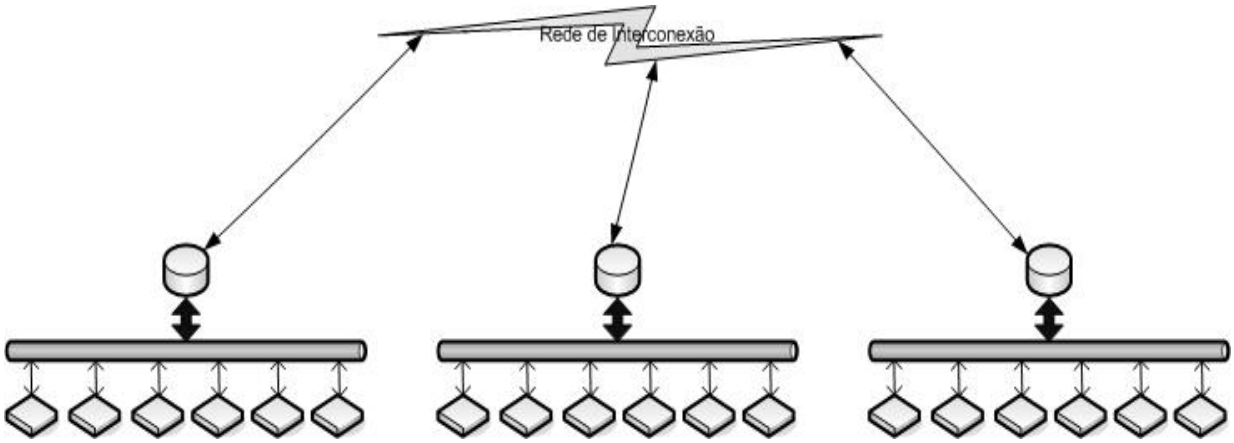

Figura 5.2: Representação SMP *clusterizado*

Alguns sistemas possuem suporte de *hardware* e de software para que um processador acesse a memória de um *nó* remoto diretamente. No entanto, na maioria dos casos, *clusters* de SMPs necessitam de trocas de mensagens explícitas para a comunicação entre os *nós* do *cluster* [49]. *Clusters* de SMPs, introduzem um nível adicional na hierarquia de memória constituído por um conjunto de sistemas de memória compartilhada. Infelizmente, esse nível adicional de memória torna o comportamento desses sistemas menos previsível e a sua programação mais difícil [61].

5.2 Programação Híbrida

A utilização de um modelo híbrido de programação se beneficia das vantagens dos dois modelos. Por exemplo, a programação híbrida nos permite o uso das políticas de particionamento explícito de dados, característico do paradigma de troca de mensagens junto com o paralelismo de grão fino, característico do paradigma de memória compartilhada [49].

Nesse trabalho utiliza-se a biblioteca MPI para a programação da parte de troca de mensagens e as diretivas de compilação e bibliotecas do padrão OpenMP para a programação da parte de memória compartilhada. Esses são os dois principais padrões de programação paralela e conseqüentemente, são os mais utilizados para estratégias híbridas de programação. Por isso, o restante do trabalho se concentra mais especificamente nesses dois modelos.

A maioria das aplicações híbridas apresenta um modelo hierárquico com a paralelização MPI em um nível superior e a paralelização OpenMP em um nível abaixo [12] [49]. A figura 5.3 mostra um *array* bidimensional que foi dividido entre quatro processos MPI. Cada um desses subarrays foi então subdividido entre *threads* OpenMP. Esse modelo hierárquico é um mapeamento muito próximo à arquitetura física de um *cluster* de SMPs.

Em um programa híbrido típico, o MPI é iniciado e finalizado de forma usual, com as rotinas *MPI_INIT* e *MPI_FINALIZE*. Uma região paralela do OpenMP ocorre entre essas chamadas, criando uma ou mais *threads* adicionais em cada processo. Se, o programa for executado utilizando quatro processos MPI e duas *threads* OpenMP então o fluxo de execução da figura 5.4 seria observado.

Para garantir a portabilidade da aplicação, o ideal é realizar as chamadas de comunicação do MPI fora de regiões paralelas do OpenMP. Em geral, as chamadas de comunicação são realizadas naturalmente fora dessas regiões, já que a paralelização OpenMP costuma ser transparente para o processo que participa de uma execução MPI. Entretanto, quando é inevitável realizá-las dentro da região paralelas é imprescindível usar alguma rotina

de sincronização que garanta que a comunicação seja realizada por apenas uma das *threads*. No OpenMP pode-se utilizar as rotinas de sincronização *CRITICAL*, *MASTER* ou *SINGLE* para esse propósito.

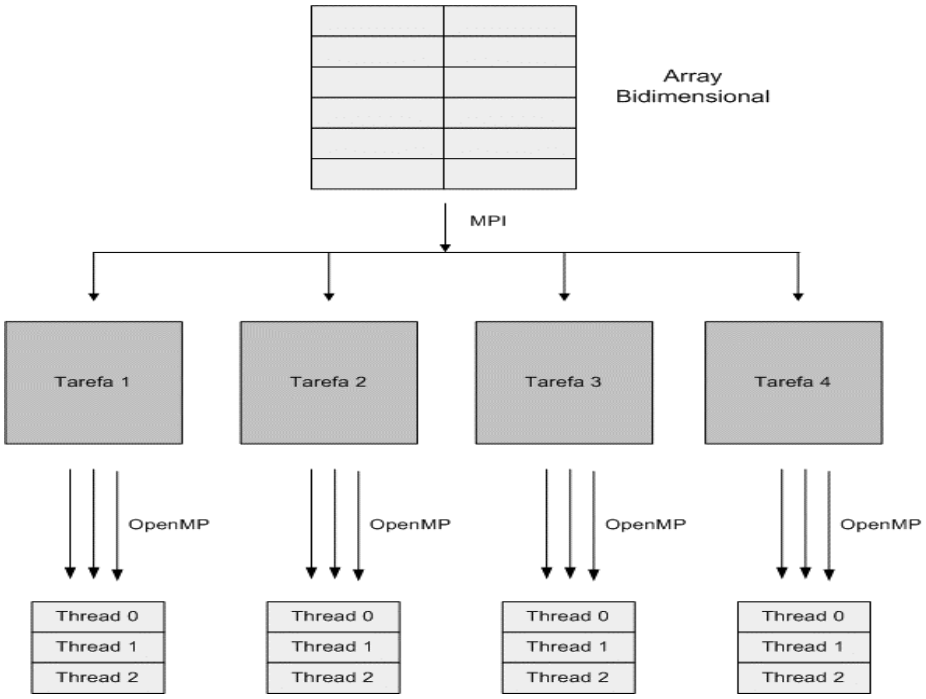

Figura 5.3 Representação de uma abordagem hierárquica para um problema de um *array* bidimensional.

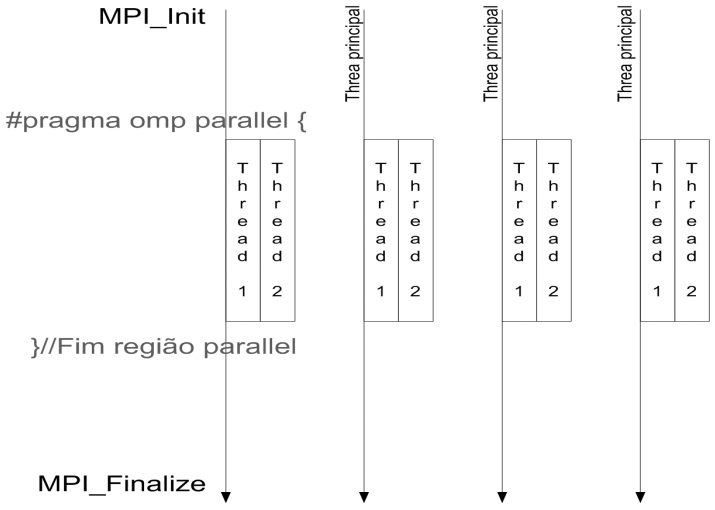

Figura 5.4: Fluxo de execução com quatro processos MPI e duas *threads* OpenMP

Ao escrever uma aplicação híbrida OpenMP/MPI é importante considerar também como cada paradigma pode paralelizar o problema e como combinar os dois para alcançar o melhor resultado. O problema do *grid* bidimensional envolve uma decomposição do *grid* em uma dimensão com MPI e em outra com OpenMP pois a conversão de problemas com estrutura hierárquica é simples de ser utilizada. Outros tipos de problema podem exigir soluções mais complexas [49].

Além de combinar explicitamente troca de mensagens e *multithreading* em um programa, outros estilos híbridos de programação incluem o uso de versões de bibliotecas de troca de mensagens que utilizam a memória compartilhada para comunicação dentro do SMP e ambientes de memória virtual compartilhada [6] [13].

5.3 Benefícios da Programação Híbrida.

5.3.1 Aplicações com Problemas de balanceamento de carga

Para executar em *cluster* de SMPs um sistema pode utilizar simplesmente MPI para comunicação dentro dos *nós* SMP. No entanto, um grande número de aplicações não escala bem com MPI, do qual grande parte é de aplicações que envolvem um balanceamento de carga complexo. Com um código híbrido, o MPI é utilizado apenas para comunicação entre *nós* e, uma vez que o OpenMP é menos suscetível a problemas de balanceamento, há uma grande possibilidade de um melhor desempenho.

5.3.2 Aplicações de Grão Fino

O OpenMP geralmente apresenta melhor desempenho em aplicações de grão fino, onde a quantidade de comunicação MPI pode ser superada pela quantidade de comunicação OpenMP. Quando uma aplicação exige boa escalabilidade, com uma paralelização de grão fino, um código híbrido pode ser mais eficiente. É claro que uma aplicação apenas OpenMP teria um desempenho melhor, mas para *clusters* de SMPs o MPI ainda é necessário para comunicação entre os *nós*. Reduzindo o número de processos MPI a escalabilidade, na maioria das situações, é melhorada.

5.3.3 Dados Replicados

Códigos que usam uma estratégia de dados replicados, geralmente sofrem de limitações de memória e de baixa escalabilidade devido a comunicações globais. Utilizando um modelo de programação híbrido em um *cluster* de SMPs, o problema pode ser limitado à memória do SMP e não à memória de dois processadores como é o caso da paralelização MPI pura. Essa é uma vantagem clara que permite o estudo de problemas de tamanhos maiores

5.3.4 Aplicações MPI restritas

Algumas aplicações MPI exigem um número específico de processos para serem executadas. Por exemplo, um código que limita o número de processos MPI para algumas combinações ou um número fixo ou que escalam apenas em potências de 2. Isso pode criar problemas de duas formas. Em primeiro lugar, o número de processos necessários pode não ser igual ao número de máquinas. Se for muito grande, o número de processos pode tornar a execução inviável e se for muito pequeno, torna a utilização dos recursos de *hardware* ineficiente com máquinas não utilizadas. Se um código híbrido é utilizado, a estratégia de decomposição natural do MPI pode ser usada, executando o número desejado de processos e *threads* OpenMP usadas para distribuir o trabalho entre processadores de forma a permitir uma utilização eficiente dos recursos.

5.3.5 Balanceamento de Poder Computacional

A técnica chamada de balanceamento de poder computacional dinamicamente ajusta o número de processos trabalhando em uma determinada computação. A aplicação é escrita em modo híbrido com diretivas OpenMP dentro do código MPI. Inicialmente o trabalho é distribuído entre processos MPI, mas quando a carga em um processador dobra, o código usa diretivas MPI para criar uma nova *thread* em outro processador. Assim, sempre que a carga de processamento de um processo MPI se torna excessiva, o trabalho pode ser redistribuído [49].

5.4 Modelos de Programação

A primeira escolha a ser feita no desenvolvimento de uma aplicação para um *cluster* de SMPs é entre um modelo unificado e um modelo híbrido. No modelo de programação

unificado, o programador usa uma única API para descrever tanto a comunicação dentro do *nó*, quanto a comunicação entre *nós* diferentes. Todos os sistemas de troca de mensagem ou DSM pertencem a essa categoria. Os modelos híbridos, por sua vez, misturam memória compartilhada dentro do multiprocessador e troca de mensagens entre *nós*. MPI + OpenMP, MPI+*pthreads* são dois exemplos de modelos híbridos [61] [4].

O desempenho de tais modelos depende pelo menos de três fatores: a) o compartilhamento de suporte de comunicação (memória do sistema e interface de rede) entre processadores, isto é, como a latência por processo e a largura de banda evoluem quando vários processadores utilizam a mesma interface de rede; b) o grau de paralelismo de memória compartilhada que pode ser alcançado com o modelo híbrido e c) o *speed-up* obtido na seção paralela [4] .

5.4.1 Tamanho de Grão da Paralelização

5.4.1.1 Paralelização de Grão Fino

A partir de um código MPI já existente, a abordagem mais simples consiste em paralelizar com o OpenMP os *loops* dentro do código MPI. Essa abordagem é chamada de paralelização de grão fino, em nível de *loop* ou ainda de paralelização incremental.

Várias abordagens diferentes podem ser utilizadas. A primeira possibilidade consiste em paralelizar os *loops* na parte procedural do MPI sem quaisquer otimizações. Apenas a corretude da versão paralela contra a versão procedural é verificada. Mas a abordagem incremental pode ser melhorada significativamente aplicando várias otimizações manuais (permutação de *loops*, troca de *loops*, uso de variáveis temporárias). Essas otimizações são necessárias por exemplo, para transformar um *loop* que não pode ser paralelizado em um que pode ser, ou então para melhorar a eficiência do *loop* paralelo, evitando falso compartilhamento e diminuindo o número de pontos de sincronização.

Uma outra questão é a escolha dos *loops* a serem paralelizados. Uma opção é paralelizar todos os loops. Essa claramente não é uma boa opção pois pode incluir *loops* que não contribuem de forma significativa para o tempo total de execução. Nesse caso a paralelização pode tornar essas porções de código mais lentas que a versão serial. A melhor alternativa consiste em selecionar através de uma análise detalhada os *loops* que influem significativamente para o tempo total de execução. A figura 5.5 mostra um fluxograma de um processo ideal para paralelização de grão fino [4] [13].

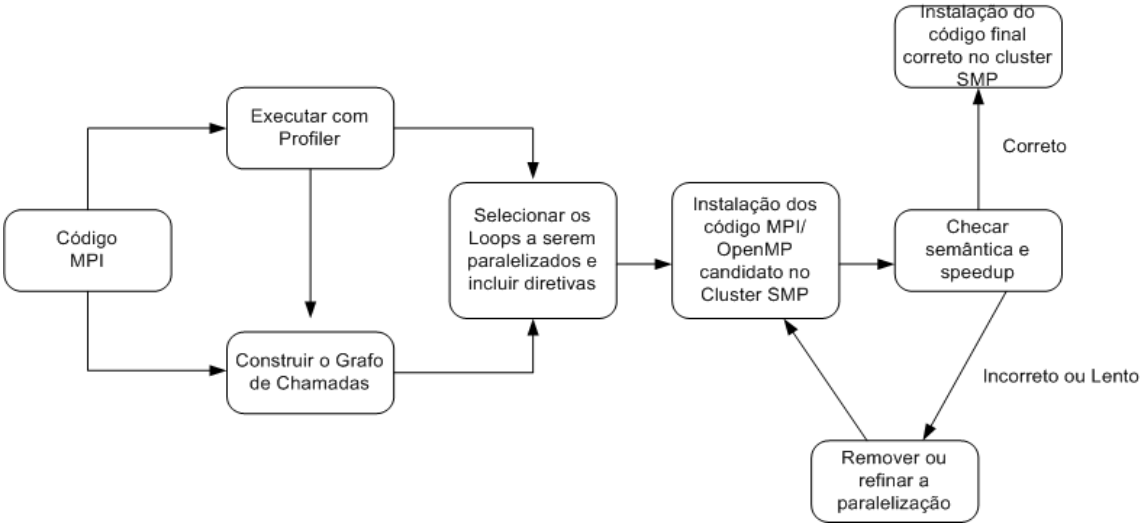

Figura 5.5: Fluxograma de paralelização de Grão Fino

5.4.1.2 Paralelização de Grão Grosso

Uma outra abordagem utilizada é a paralelização de grão grosso (ou SPMD). Nessa abordagem, o OpenMP ainda é utilizado para obter vantagem sobre a memória compartilhada dentro do *nó* SMP, mas um estilo de programação SPMD é utilizado. Nesse caso, o OpenMP é usado para criar N *threads* no início do programa principal, onde essas *threads* se comportam exatamente como um dos processos MPI original, mas se comunicam por memória compartilhada.

Da mesma forma que em um modelo SPMD de troca de mensagens, o programador deve tomar cuidado com algumas questões: distribuição de dados entre as *threads*, distribuição de trabalho e coordenação entre elas.

Já que a distribuição de dados é através da memória compartilhada, ela implica apenas a atribuição de diferentes regiões da mesma estrutura de dados para diferentes *threads* em execução. Geralmente, o programador calcula a região atribuída a cada *thread* com base em alguma função de *hash* sobre o número (identificador) da *thread*. A coordenação das *threads* envolve a gerência de seções críticas e barreiras, além do uso de ou a diretiva *MASTER* do OpenMP ou da chamada de biblioteca omp_get_*threads*_num() para instruções condicionais (exclusivas de uma ou grupo de *threads*). Existem poucas publicações sobre resultados em paralelização OpenMP+MPI de grão grosso [4] [13].

5.5 Sobreposição de chamadas de comunicação

Normalmente, quando as chamadas MPI são realizadas fora da região paralela do OpenMP a única *thread* que realiza comunicação é *thread* meste. Enquanto a comunicação está sendo realizada por uma *thread* todas as outras estão bloqueadas esperando a conclusão da operação, de forma que o tempo de CPU não é aproveitado da melhor maneira possível.

Existe a possibilidade de realizar a sobreposição de comunicação e computação através de *threads* concorrentes. Nesse caso, enquanto a comunicação é realizada pela *thread* mestre, outras *threads* não comunicantes, podem executar algum código da aplicação. Essa categoria exige que o código da aplicação seja separado em duas partes: código que pode ser sobreposto com comunicação e código que deve esperar até que as operações de comunicação sejam finalizadas.

A sobreposição de comunicação e computação é útil pra para obter o máximo aproveitamento do *hardware*. Porém, há algumas desvantagens. Em primeiro lugar, a maioria das aplicações não está preparada para distinguir entre instruções que podem ser realizadas antes do término das operações de envio e recebimento de dados e as que podem. Em segundo lugar, um modelo de programação de grão grosso é necessário, o que geralemente exige que a aplicação seja reescrita para que distribuição de trabalho entre as *threads* seja realizada manualmente baseada nos ranks de cada uma. E finalmente, é preciso implementar alguma funcionalidade de balanceamento de carga para compensar as diferentes cargas de comunicação e computação entre as *threads* [41] [57].

5.6 Outras Fontes de *Overhead* do OpenMP

A parelização OpenMP dentro de processos MPI pode causar *overhead* adicional. A criação das regiões paralelas e a sincronização ao seu final induzem algum trabalho adicional, principalmente se uma abordagem de grão fino é utilizada. Esse *overhead* pode ser reduzido com uma estratégia de grão grosso para a paralelização: a região paralela é inicializada apenas uma vez no início da aplicação e diretivas *OMP MASTER* e *BARRIER* são utilizadas para sincronização antes e depois da comunicação MPI. Se não for possível paralelizar todo o trabalho do MPI ou se a paralelização OpenMP não puder ser balanceada satisfatoriamente, então o *speedup* também será reduzido em virtude da Lei de Amdahl [41].

5.7 Desempenho dos Modelos de Programação Paralela

Existem muitos trabalhos que estudam o desempenho de modelos de programação paralela. Em [32], por exemplo, foi realizado um estudo de desempenho sobre arquitetura de troca de mensagens em *clusters* SMP. Em [47] foi realizada uma comparação entre os modelos de troca de mensagens e de memória compartilhada. Ambos [32] e [47] são focados em modelos tradicionais e não abordam modelos híbridos.

A respeito desses modelos existem alguns trabalhos como [4], onde foram realizados testes para 6 diferentes *benchmarks*. Nesse trabalho compara-se o desempenho de versões puras MPI dos programas com versões híbridas modificadas com diretivas OpenMP para diferentes tamanhos de dados e arquiteturas de *cluster*. Em [57] um modelo híbrido de programação com MPI e OpenMP foi aplicado para a solução de um sistema linear de equações em um *cluster* SMP. Dois trabalhos interessantes são [41] e [6] onde os autores discutem diversos aspectos relevantes no desenvolvimento de aplicações em um modelo híbrido de programação, além de realizarem uma grande variedade de testes de desempenho.

Particularmente em [36] foi desenvolvida uma aplicação híbrida para um código baseado no método dos elementos finitos para simulações climáticas e geológicas no computador *Earth Simulator*.

Alguns trabalhos como [19], [25] e [45] apresentam modelos híbridos baseados em ambientes de memória distribuída compartilhada (DSM) como uma alternativa para unificar os modelos de memória.

CAPÍTULO 6

APLICAÇÃO HÍBRIDA PARA UM CÓDIGO DE ELEMENTOS FINITOS APLICADO À ELASTICIDADE LINEAR

A aplicação escolhida para este estudo de arquiteturas híbridas de programação paralela trata-se da simulação de um problema de elasticidade linear baseada no método dos elementos finitos (MEF). Essa simulação utiliza o método dos gradientes conjugados para a solução de um sistema de equações.

Pelo método dos elementos finitos um modelo matemático descrito por equações diferenciais parciais em um domínio contínuo, é convertido em um modelo discreto de elementos finitos, com um número finito de graus de liberdade (GLs) [2] [24]. O trabalho realizado em [59] apresenta resultados de execuções de uma versão 3D do método dos gradientes conjugados (MGC) na solução das equações de equilíbrio de um problema de elasticidade tridimensional, utilizando programação paralela por troca de mensagens. Esse trabalho foi baseado no algoritmo desenvolvido em [29].

A aplicação, originalmente desenvolvida para o paradigma de troca de mensagens, foi transformada em uma aplicação híbrida para se beneficiar da comunicação via memória compartilhada dentro do nó SMP, com a comunicação entre os nós realizada por troca de mensagens.

Para essa transformação foi utilizada uma abordagem de grão fino com a paralelização dos *loops* responsáveis pela solução do sistema de equações através de *threads* OpenMP. Estas *threads* compartilham a mesma memória física. Isto possibilita que o número de tarefas MPI seja reduzido e conseqüentemente seja diminuído o volume de comunicação inter-processos. No caso um *cluster* com 8 máquinas SMP de dois processadores cada uma, utilizando-se todos os processadores em uma execução da aplicação, tem-se 16 tarefas MPI (duas por nó SMP) que trocam mensagens durante seu ciclo de vida. No programa modificado, pode-se iniciar a execução da aplicação com apenas 8 tarefas MPI (uma por nó SMP) e à medida que regiões paralelas são encontradas uma nova *thread* t2 é criada no processador ocioso do nó e executará algum trecho de código em paralelo com a *thread* t1 que corresponde ao programa principal da tarefa MPI. A comunicação entre t1 e t2 ocorre através da memória, compartilhada pelos processadores do mesmo nó SMP. A conseqüência prática é que a comunicação MPI entre duas tarefas é substituída por uma mais rápida na memória compartilhada.

6.1 Elasticidade Linear

Um material é elástico se ele deforma quando uma carga é aplicada a ele, mantém uma deformação constante enquanto a carga é mantida constante e retorna ao seu formato original, não deformado, quando a carga é removida.

As propriedades mecânicas de um material podem ser verificadas por um teste de tensão no qual uma barra ou cilindro de um material com comprimento L e seção A é fixada em uma de suas extremidades e sujeita a uma carga F (Figura 6.1). Enquanto a carga é gradualmente aumentada, o corpo de prova irá sofrer deformação até se romper em dois pedaços. Normalmente deseja-se compreender o comportamento desses materiais para diferentes tamanhos e formas de objetos, especialmente como a carga F aplicada se relaciona com a deformação ocorrida e que cargas o material suporta sem que ocorra uma fratura [44].

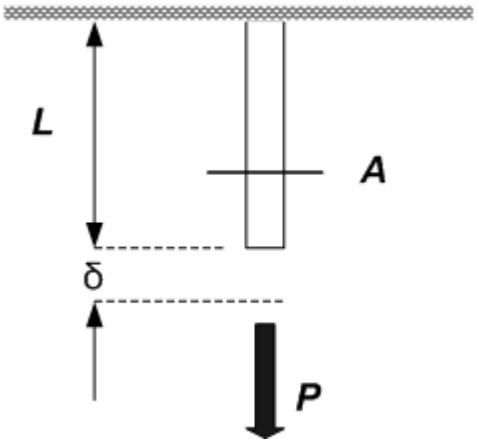

Figura 6.1: Teste de Tensão

Sólidos constituídos de materiais dúcteis, como aço e ferro, possuem uma relação entre tensão e deformação bastante complexa, envolvendo comportamentos mecânicos diversos. Tais sólidos, quando sujeitos a tensões suficientemente pequenas exibem comportamento elástico linear e apresentam valores de deformações e deslocamentos diretamente proporcionais às forças aplicadas. A figura 6.2 apresenta o gráfico dos regimes de deformação de materiais dúcteis. No gráfico pode-se ver que quando o sólido sai do regime elástico linear e entra no regime plástico a relação entre tensão e deformação deixa de ser constante. Além disso, no regime plástico o sólido não retorna à sua forma original quando

são retirados os esforços mecânicos [24]. Este trabalho está restrito a problemas no regime elástico linear.

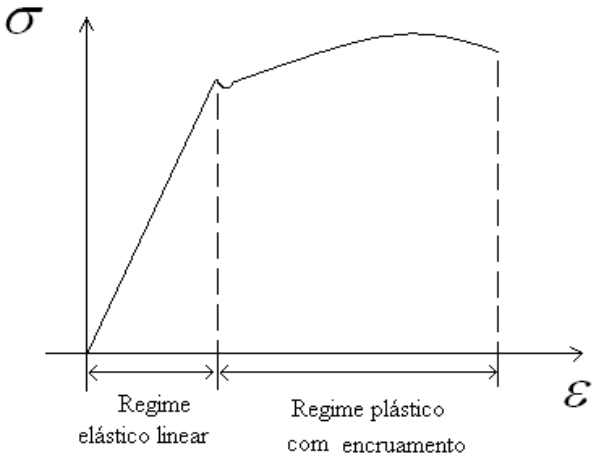

Figura 6.2: Gráfico de tensão–deformação de materiais dúcteis [59].

Para materiais elásticos lineares, a forma geral de uma equação constitutiva é:

$$\textit{Tensão} = \textit{constante} \times \textit{deformação}. \tag{6.1}$$

A constante de proporcionalidade é determinada experimentalmente e diz respeito às propriedades que constituem o material em questão.

Problemas de elasticidade linear podem ser resolvidos através do método dos gradientes conjugados aplicado a um sistema de equações gerado pelo método dos elementos finitos. O método dos elementos finitos, aplicado a um problema estático linear, gera um sistema de equações algébricas da forma:

$$K\,u = F, \tag{6.2}$$

onde K é uma matriz simétrica e positiva definida constante e representa a matriz de rigidez do material, F é um vetor de forças externas aplicadas ao sólido e u é um vetor de incógnitas e representa os deslocamentos resultantes da aplicação das forças externas. Os vetores u e F possuem dimensão igual ao número de graus de liberdade do sólido. K é uma matriz quadrada de dimensões também iguais ao número de graus de liberdade do sólido.

6.2 O Método dos Elementos Finitos

No campo da engenharia, a análise de problemas complexos requer a modelagem matemática do sistema físico e muitas vezes a solução analítica é complexa e geralmente impossível. Nesse caso é necessário recorrer ao uso de técnicas numéricas. O método dos elementos finitos é uma técnica numérica extremamente poderosa para a solução de problemas nas áreas de Estruturas, Transferência de Calor, Mecânica dos Fluidos, etc.

O procedimento geral do método é que cada estrutura ou corpo é dividido em elementos menores de dimensões finitas, chamados elementos finitos. O corpo original é então considerado como uma composição desses elementos. Esses elementos são conectados entre si através de junções chamadas *nós* ou pontos nodais para formar a estrutura completa. As propriedades desses elementos individuais são então formuladas e a partir dessas, as propriedades do corpo como um todo, são obtidas [9] [2].

O método dos elementos finitos é uma técnica numérica e as respostas obtidas com ele não são soluções exatas e sim soluções aproximadas. Entretanto, utilizando procedimentos apropriados e tendo à disposição recursos computacionais poderosos, é possível obter um alto grau de precisão [43].

6.2.1 Interpretação

As seguintes informações são associados a um elemento finito individual. Esses dados são usados por aplicações de elementos finitos para realizar os cálculos [15]:

Dimensionalidade:
Os elementos podem ter uma, duas ou três dimensões espaciais.

Pontos Nodais
Cada elemento possui um conjunto de pontos distintos chamados pontos nodais ou apenas *nós*. *Nós* servem a dois propósitos: definir a geometria do elemento e os graus de liberdade. Eles se localizam nos cantos ou nas terminações dos elementos.

Geometria
A geometria do elemento é definida pelo posicionamento dos pontos nodais. Na prática, a maioria dos elementos usados possui geometrias simples. Em uma dimensão,

elementos são geralmente linhas retas ou segmentos curvos. Em duas dimensões eles costumam ter formato triangular ou quadrilateral. Em três dimensões costumam ser tetraedros, pentaedros ou hexaedros. A figura 6.3 mostra algumas geometrias típicas em uma, duas e três dimensões.

Graus de Liberdade

Os graus de liberdade determinam a forma do elemento. Eles também funcionam como junções através das quais elementos adjacentes são conectados. Graus de liberdade são definidos como valores de variáveis primárias nos pontos nodais. No caso de aplicações na área de estruturas, essas variáveis primárias especificam, por exemplo, o deslocamento ocorrido em cada ponto nodal.

Forças Nodais

Há sempre um conjunto de forças nodais em uma relação de um para um com graus de liberdade. Em elementos mecânicos a correspondência é estabelecida em termos de energia.

Relações Constitutivas

Para um elemento mecânico, são as relações que especificam as propriedades do material. Em uma barra elástica linear, considerando-se a análise estrutural, é suficiente especificar o módulo elástico *E*.

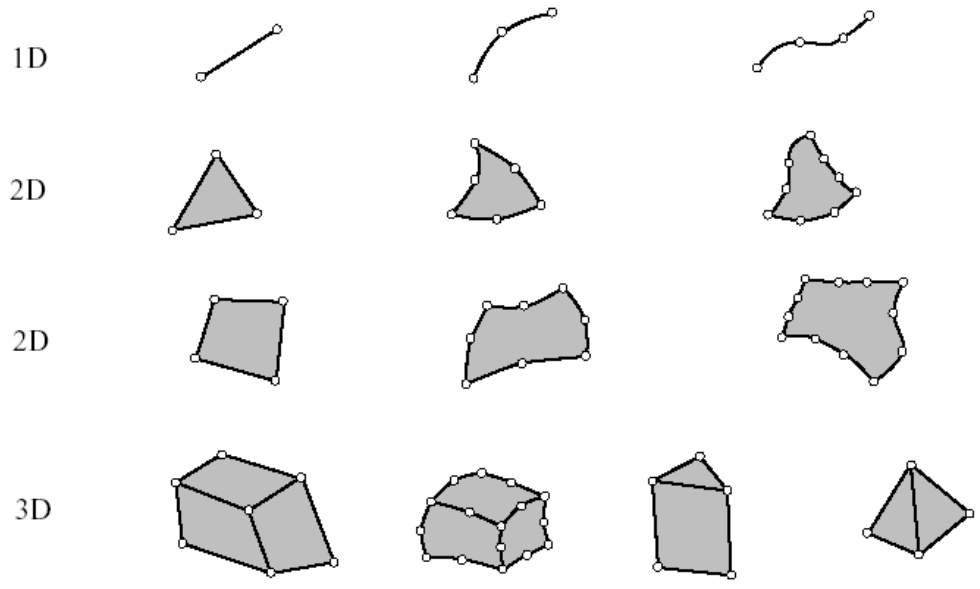

Figura 6.3: Geometrias típicas de elementos finitos em uma, duas e três dimensões [15].

6.3 Método dos Gradientes Conjugados

O Método dos Gradientes Conjugados (MGC) é um dos mais populares métodos iterativos para a resolução de grandes sistemas de equações lineares. O MGC é efetivo para sistemas da forma:

$$Ax = b \tag{6.3}$$

Onde x é um vetor desconhecido, b é um vetor conhecido e A é uma matriz simétrica, positiva definida e quadrada. Tais sistemas de equações aparecem em diversos problemas importantes da Matemática, da Física e da Engenharia.

Métodos iterativos como o MGC, são adequados para uso com matrizes esparsas. Se A é densa, a melhor alternativa é fatorá-la e resolvê-la por retrosubstituição. A equação 6.3 pode ser reescrita como o sistema em 6.4 [48].

$$\begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \tag{6.4}$$

Resolver as equações combinadas da Equação 6.4 é equivalente a encontrar o mínimo da forma quadrática correspondente ao sistema:

$$f(x) = (1/2)x^{\mathrm{T}}Ax - b^{\mathrm{T}}x + c. \tag{6.5}$$

Devido ao relacionamento entre a matriz A e a função escalar $f(x)$, é possível ilustrar algumas fórmulas da álgebra linear com imagens, mais intuitivas. Por exemplo, a matriz A é chamada positiva definida se a seguinte propriedade é verdadeira para qualquer vetor x:

$$x^{\mathrm{T}}Ax > 0. \tag{6.6}$$

A figura 6.4 mostra as formas quadráticas para matrizes do tipo positiva definida (a), negativa definida (b), positiva indefinida (c) e indefinida (d).

A implicação da figura 6.4, é que se a matriz é positiva definida como em 6.4(a) então, ao invés de resolver o sistema da Equação 6.3, pode-se obter o mesmo resultado encontrando-

se o mínimo da sua função quadrática. Com o método dos gradientes conjugados isso pode ser realizado em *n* ou mais passos, de forma iterativa.

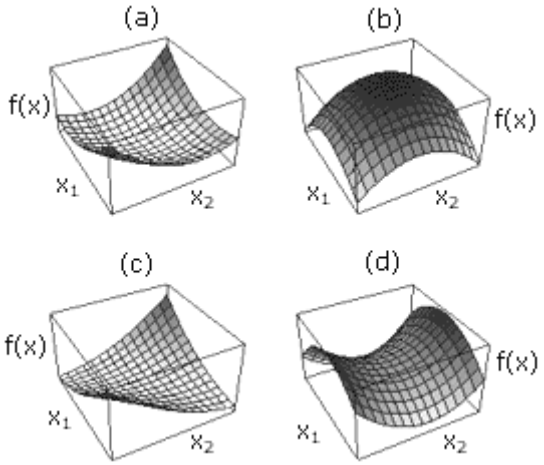

Figura 6.4: Formas quadráticas para tipos diferentes de matrizes;
(a) positiva definida; (b) negativa definida; (c) positiva indefinida;
(d) indefinida [1]

A busca do mínimo pode ser ilustrada através do método dos gradientes na figura 6.5 e basicamente inclui os seguintes passos:

- O gradiente é calculado no ponto inicial x(0) e o movimento continua, na direção de busca, na linha do antigradiente enquanto a função objetivo continua decrescendo.

- No ponto onde a função pára de decrescer, o gradiente é calculado de novo e o movimento continua em outra direção.

- O processo continua até que o ponto mínimo é encontrado [1].

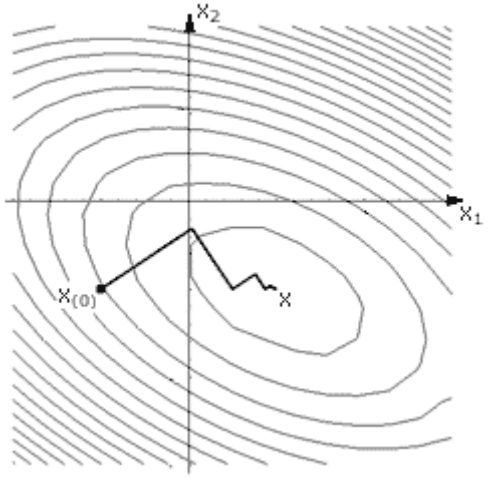

Figura 6.5: Busca de mínimo no método dos gradientes conjugados [1].

6.4 Código

A seguir serão apresentados os funcionamentos da versão pura MPI e híbrida do código de elementos finitos e explicados seus funcionamentos.

6.4.1 Aplicação Pura MPI

A entrada para a aplicação é uma malha constituída de valores nodais dos elementos finitos. Essa malha é gerada pela aplicação GID [7] para a geometria selecionada com as restrições e forças impostas a ela. O GID é uma ferramenta para modelagem geométrica para simulações numéricas que utilizam o método dos elementos finitos. Dentre as informações que descrevem uma malha têm-se as propriedades do material (módulo de elasticidade, coeficiente de Poisson), informações de forças e restrições, coordenadas de cada nó e conectividades.

Inicialmente, essa malha é particionada através da biblioteca METIS [20]. O número de partições é igual ao número de tarefas MPI. Assim, cada tarefa manipula aproximadamente o mesmo volume de dados. Cada tarefa então monta sua matriz de rigidez e algumas estruturas de dados auxiliares baseado na partição recebida. E então, o método dos gradientes conjugados paralelo é aplicado para resolver o sistema de equações. A figura 6.6 apresenta o fluxograma da aplicação.

6.4.1.1 – Decomposição de Domínio

Na implementação paralela do método dos gradientes conjugados a matriz de rigidez A deve ser subdividida e distribuída entre as várias tarefas. Cada tarefa possui também uma subdivisão do vetor x e uma subdivisão do vetor u [29] [59].

Quando ocorre divisão alguns graus de liberdade ficam compartilhados por mais de uma tarefa. Esses graus de liberdade são chamados de graus de liberdade de fronteira (ou compartilhados), enquanto todos os demais são chamados de graus de liberdade internos. Para fazer essa distinção, a matriz A de cada processador é subdividida em quatro Ap, As, Bp e Bp^T. Ap é uma matriz quadrada com dimensão igual ao número de graus de liberdade internos de cada subdomínio e nela estão armazenados os valores de rigidez referentes aos graus de liberdade internos. A matriz As também é quadrada e tem dimensão igual ao número de graus de liberdade compartilhados e nela estão os valores da matriz de rigidez para os graus de liberdade de fronteira. A dimensão da matriz B_p é função tanto dos graus de liberdade internos

quanto dos de fronteira. Nesta, estão os valores de rigidez que relacionam os graus de liberdade privados com os de fronteira. B_p^T é a matriz transposta de B_p.

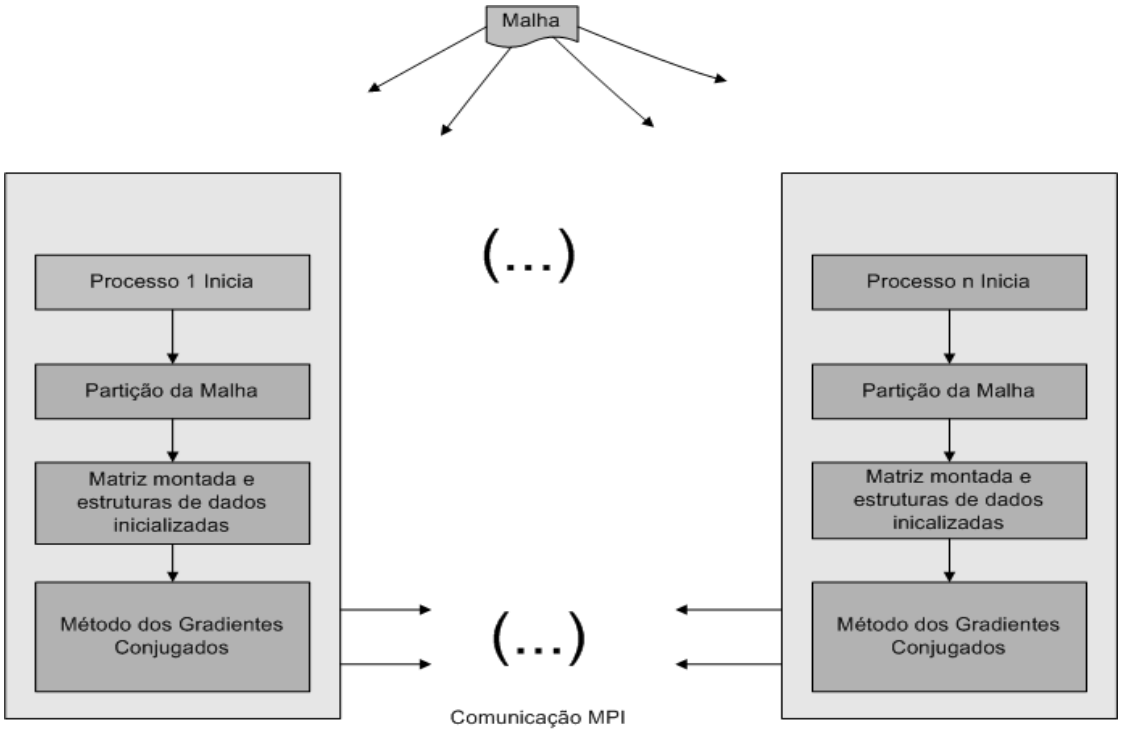

Figura 6.6: Fluxo da Aplicação

Devido à reestruturação da matriz A, os vetores x e b também são reformulados. Cada um deles é separado em outros dois. O vetor b é desmembrado em bp e bs, sendo que o primeiro tem dimensão igual aos graus de liberdade internos e o segundo aos de fronteira. O mesmo procedimento se aplica ao vetor x.

6.4.1.2 – Computação

O MGC é um método iterativo que busca a convergência para a solução de um sistema de equações. Esse método, não calcula um valor exato e sim uma solução aproximada para o sistema. Para isso especifica-se um erro máximo ε que representa o critério de parada do algoritmo. Quando é obtida uma solução com um erro menor ou igual a ε, assume-se que a solução está próxima o suficiente do valor real e ela pode ser considerada a solução para o sistema de equações.

Inicialmente arbitra-se um valor aproximado para o vetor de resposta x. O valor arbitrado é então introduzido no sistema de equações para obter o resíduo γ. Se o resíduo é

menor ou igual ao erro, o algoritmo pára e o valor de *x* arbitrado inicialmente é considerado a resposta para o método. Caso contrário o método entra no *loop* de resolução. Em cada iteração, o resíduo γ é recalculado e se ele é menor ou igual ao erro ε, o algoritmo pára.

Não é possível prever quantas iterações o método realizará para chegar à resposta final. Eventualmente, o método pode falhar em convergir para uma solução e entrar em um *loop* infinito e por isso o número de iterações é limitado a um valor máximo.

A computação do método dos gradientes conjugados envolve um produto de matriz por vetor, duas operações de produto interno, três ajustes de vetores e uma operação de atualização do vetor *x*. O algoritmo é de ordem $O(n^2)$ e quando ele é executado por *p* processadores, cada um deles resolve um subproblema em tempo $O(n^2/p)$

6.4.1.3 – Comunicação

As tarefas MPI realizam comunicação apenas no método dos gradientes conjugados. Se uma malha é dividida entre alguns processadores, eles compartilham graus de liberdade em suas fronteiras e devem realizar comunicação intensiva para trocar informações durante os cálculos. Além disso, para calcular o erro, é necessário utilizar rotinas de computação global nos dados espalhados no diversos processadores. A comunicação interprocessos nessa aplicação é restrita a dois procedimentos: o produto interno e a atualização dos graus de liberdade compartilhados.

O procedimento de produto interno é realizado em vetores localizados em tarefas diferentes. Para isso é utilizada uma rotina de computação global de soma (*All-Reduce*) para calcular um escalar.

No procedimento de atualização dos graus de liberdade, cada tarefa envia para cada um das tarefas com os quais compartilha graus de liberdade de fronteira o seu valor para esses graus. Cada um deles então, soma o valor que possui com os valores que recebeu.

6.4.2 Solução Híbrida

Para obter a versão híbrida da aplicação foi utilizada uma abordagem incremental para obter uma versão de grão fino da aplicação MPI original. Inicialmente, foi realizada uma análise dos trechos de código que consumiam mais tempo para a execução. A aplicação híbrida foi então desenvolvida através da modificação do código com diretivas OpenMP. A partir de alguns testes iniciais, a paralelização foi gradualmente refinada para garantir que as *threads* fossem usadas para computação apenas nas partes do programa onde o benefício da

execução fosse maior que o custo de criação das *threads*. Isso naturalmente exclui seções de execução muito rápida e coloca o foco em trechos de código de longa duração e que podem ser executados concorrentemente (sem dependência de dados). Essa análise levou à paralelização da parte do programa que executa o cálculo do método dos gradientes conjugados já que essa consome 90% do tempo total da aplicação. Quanto à comunicação, não ocorre dentro das regiões paralelas de modo que a comunicação MPI da aplicação não é alterada. Nas regiões modificadas com OpenMP as diversas *threads* que realizam operações em paralelo se comunicam através da memória compartilhada dentro do nó SMP. A figura 6.7 mostra o fluxo da aplicação modificada.

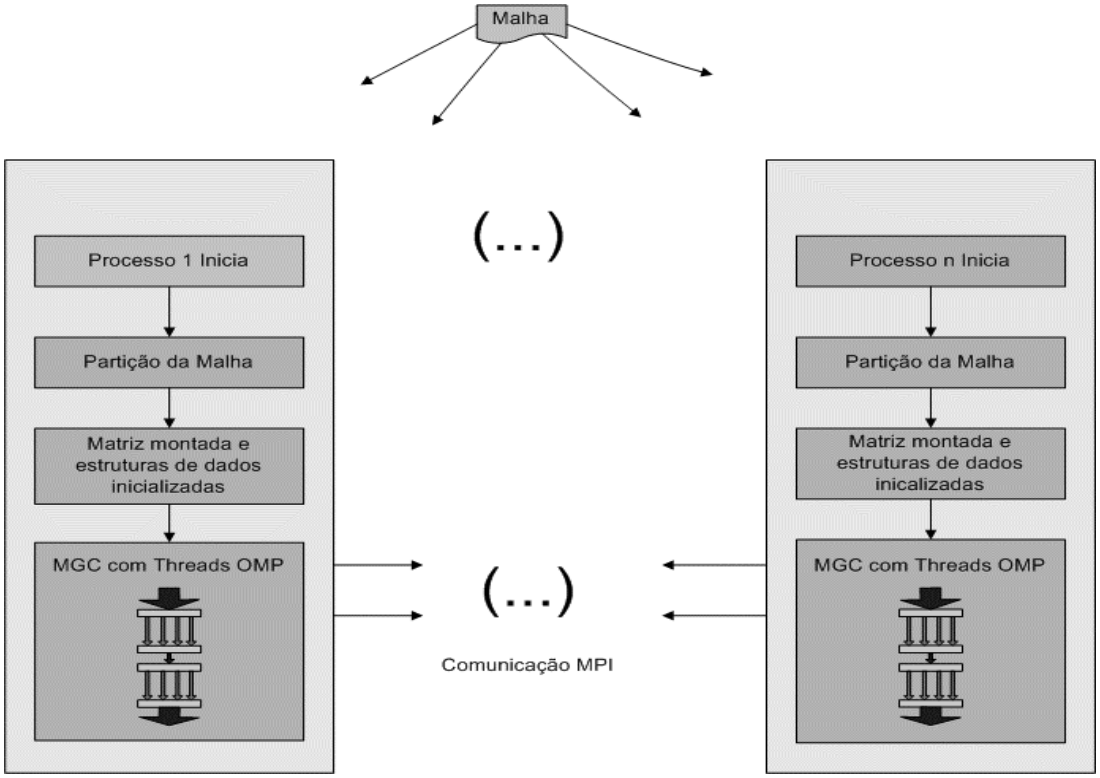

Figura 6.7: Fluxo da Aplicação Modificado (Híbrida).

O balanceamento de carga entre os nós SMP na aplicação depende diretamente da distribuição realizada nas tarefas MPI. Entretanto durante a execução, em pontos onde a computação é intensiva, a carga é subdividida entre as *threads* dentro do nó SMP, de acordo com as diretivas OpenMP inseridas no código.

Para um *cluster* com 16 processadores (2 por nó SMP) e que utiliza todos os processadores em uma execução da aplicação, na versão pura MPI o domínio será dividido em 16 partes (para 16 tarefas MPI, duas por nó SMP). Na implementação MPI/OpenMP, no

entanto, o domínio é dividido em 8 partes (usando uma tarefa MPI por nó SMP e duas *threads* nas regiões paralelas, uma por processador). A paralelização OpenMP acontecerá então em nível de *loop*, com cada *thread* realizando metade dos cálculos em paralelo. Nesse caso, se a paralelização OpenMP é eficiente para trechos de código que consomem grande quantidade de tempo, assumindo que a comunicação dentro do nó SMP é melhorada com a comunicação de memória compartilhada, então a versão híbrida deverá apresentar um desempenho melhor. A questão que se apresenta é se a paralelização OpenMP implementada é eficiente o bastante para tirar vantagem desse efeito na prática. Essa eficiência também deve ser tal que não seja prejudicada pelo *overhead* introduzido pela criação das *threads* nas regiões paralelas. A figura 6.8 apresenta a comparação entre a decomposição de dados na versão pura MPI (6.8(a)) e a versão híbrida MPI/OpenMP (6.8(b)).

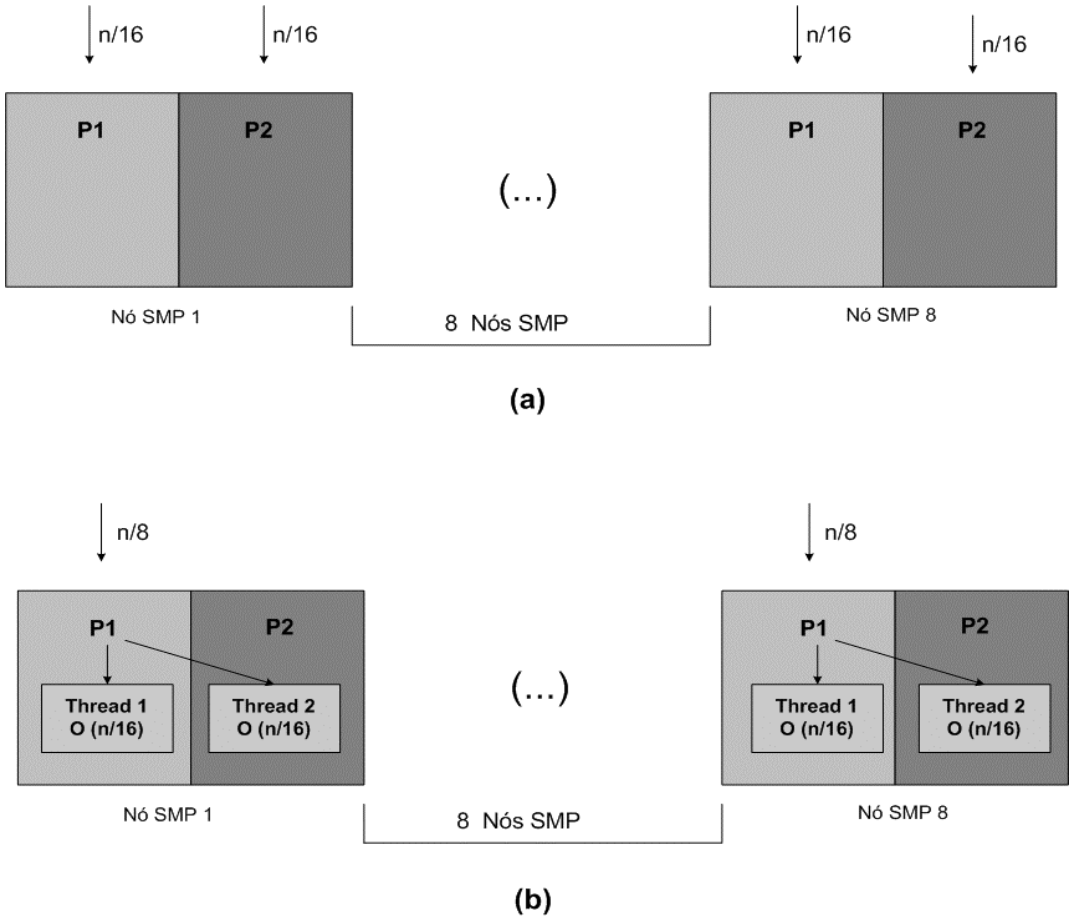

Figura 6.8: Decomposição de dados; (a) versão pura; (b) versão híbrida.

CAPÍTULO 7

RESULTADOS EXPERIMENTAIS

Este capítulo apresenta os resultados experimentais obtidos a partir dos testes realizados para a versão híbrida da aplicação, os resultados obtidos para a versão original e a comparação entre eles. Fazem parte desses dados informações como *speedup*, tempo de execução e tempo de comunicação. Na seção 7.1 são apresentadas as configurações de *hardware* e na seção 7.2 é apresentada a metodologia utilizada. Seguem-se a essas, na seção 7.3, as medidas de *speedup* relativo; na seção 7.4, a comparação dos tempos de execução; na seção 7.5 é apresentada a comparação entre os tempos de execução divididos em comunicação e computação e finalmente na seção 7.6 é apresentada a interpretação dos resultados.

7.1 Configurações de *Hardware*

Os resultados foram obtidos em um *cluster* de SMPs constituído de 8 máquinas *dual-processor* AMD rodando o Sistema Operacional Linux Red Hat 9. Na tabela 7.1 segue-se a configuração detalhada do *cluster*.

TABELA 7.1: Especificação do *Cluster*

Número de Processadores	16 (8 x 2)
Processador	AMD Athlon MP 1900+
Freqüência de *clock*	1,6 GHz
Memória *RAM*	1,0 GB
Memória *cache*	256 KB
Sistema operacional	Linux Red Hat 9
Rede de conexão	*Switch* Ethernet 1,0 Gb/s

7.2 Metodologia

Aplicamos inicialmente a abordagem incremental ao código original MPI para obter a versão híbrida da aplicação. O código foi então instrumentado para distinguir entre tempo de comunicação e tempo de computação. O tempo de comunicação inclui as chamadas de sincronização como sincronizações de barreira, por exemplo. Essa separação é importante

para investigar nossa hipótese inicial de que com a solução híbrida, o tempo total consumido com comunicação seria reduzido.

Neste capítulo, por simplificação de notação, na comparação dos dois modelos chamamos processos MPI e *threads* OpenMP de tarefas. Assim uma execução pura, com dois processos MPI é considerada como possuindo duas tarefas e uma execução híbrida com duas *threads* OpenMP também.

As medidas foram realizadas para tamanhos diferentes de malhas de uma mesma geometria. Para cada uma das malhas, variamos o número de tarefas entre 2, 4, 8 e 16. Na versão pura MPI, cada nó SMP inicia sua execução com dois processos MPI e continua com esse número constate até o final da execução. Na versão híbrida OpenMP/MPI por outro lado, em cada nó SMP apenas um processador inicia sua execução com um *thread* e à medida que, durante o fluxo do programa, são encontradas regiões paralelas, uma outra *thread* é criada para no processador disponível executar código em paralelo com a *thread* principal.

A geometria escolhida para os testes foi um bloco conforme Figura 7.1. Uma das faces do bloco foi fixada para ter seu movimento restringido e foi aplicada uma carga concentrada de 5.000 *N* na direção negativa do eixo *z* em um nó localizado em um dos vértices de forma a provocar alguma deformação.

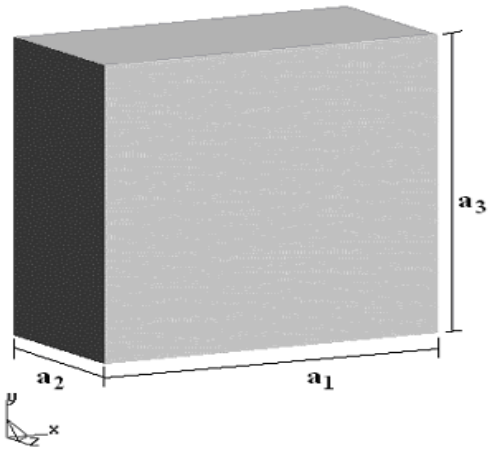

Figura 7.1 - Geometria Selecionada [59]

Os resultados foram obtidos para quatro malhas tridimensionais formadas por tetraedros lineares, especificadas na Tabela 7.2.

TABELA 7.2: Malhas utilizadas

Malha	Quantidade de Nós	Elementos	Equações
1	467	1882	1266
2	933	4014	2577
3	1984	9126	5567
4	3717	18031	10581

A figura 7.2, apresenta a geometria dividida em 933 elementos para a malha 2 e a figura 7.3 apresenta a deformação obtida no bloco.

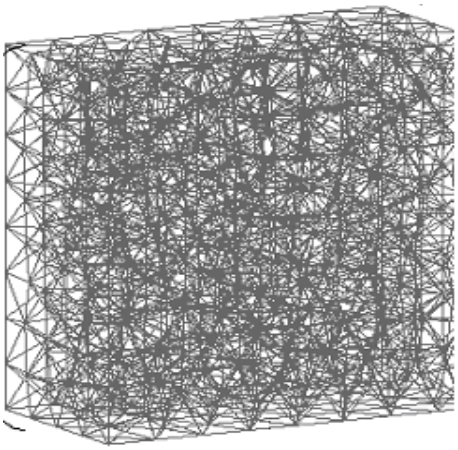

Figura 7.2: Malha dois com 933 nós e 4014 elementos [59].

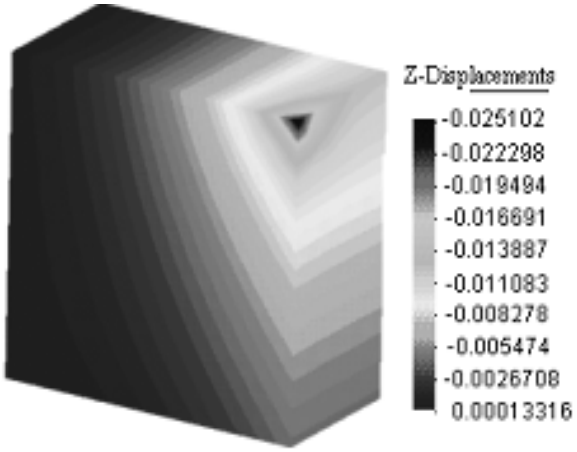

Figura 7.3: Deformação obtida [59]

As medidas de tempo foram obtidas indiretamente com o uso da instrução *assembly RDTSC* [27] que conta o número de ciclos de *clock* ocorridos desde o instante em que a máquina foi ligada ou desde sua última reinicialização. As grandes vantagens dessa estratégia de instrumentação são a alta resolução nas medidas de tempo obtidas e o baixo *overhead* introduzido na aplicação para realizá-las.

7.3 Tempos de Execução: Versão Híbrida x Versão Pura MPI

A tabela 7.3 apresenta os tempos de execução em segundos para as duas versões da aplicação para as quatro malhas utilizadas. A figura 7.4 apresenta os gráficos com as comparações dos tempos de execução.

TABELA 7.3: Valores de tempo total de execução (em segundos)

Malha	2 Threads OpenMP	2 Processos MPI	4 Threads OpenMP	4 Processos MPI	8 Threads OpenMP	8 Processos MPI	16 Threads OpenMP	16 Processos MPI
1	2,7096	2,5448	1,6209	0,9681	0,6831	1,0171	0,6457	2,295
2	13,4715	10,0369	5,8126	3,8745	1,9638	1,6984	1,2497	3,0508
3	80,6773	61,3336	30,8334	22,5368	11,8089	8,0598	4,2986	4,744
4	348,772	458,1793	244,8423	92,4036	85,683253	45,937	36,5828	14,1057

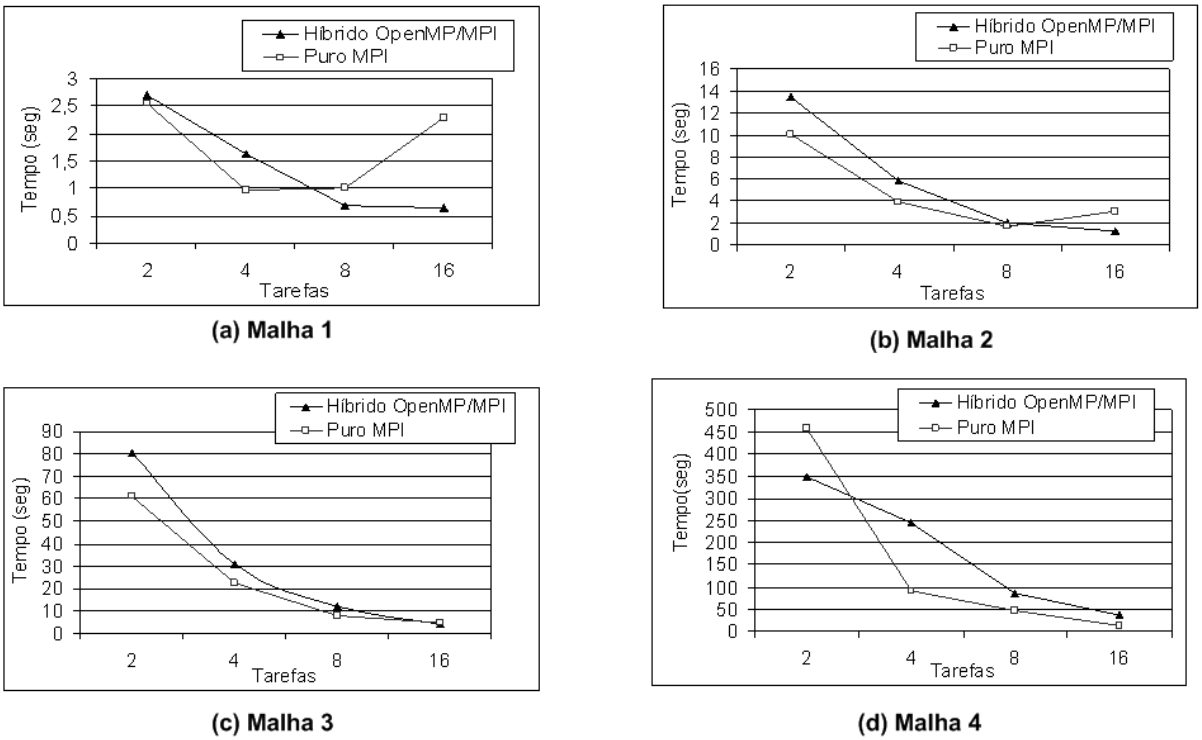

Figura 7.4 – Gráfico de comparação do tempo total de execução para o programa híbrido e para o programa puro MPI.

7.4 Tempos de Execução Por Etapas

As tabelas 7.4, 7.5, 7.6 e 7.7 apresentam o tempo de execução decomposto em tempo de computação e tempo de comunicação, respectivamente para as malhas 1, 2, 3 e 4. A figura 7.5 apresenta o gráfico do tempo decomposto para as quatro malhas.

TABELA 7.4: Tempo total de Execução Decomposto para Malha 1 (em segundos)

	2 Threads OpenMP	2 Processos MPI	4 Threads OpenMP	4 Processos MPI	8 Threads OpenMP	8 Processos MPI	16 Threads OpenMP	16 Processos MPI
computação	2,6945	2,3799	1,4784	0,6806	0,3925	0,1947	0,0876	0,0767
comunicação	0,0151	0,1649	0,1425	0,2875	0,2906	0,8224	0,5581	2,2183

TABELA 7.5: Tempo total de Execução Decomposto para Malha 2 (em segundos)

	2 Threads OpenMP	2 Processos MPI	4 Threads OpenMP	4 Processos MPI	8 Threads OpenMP	8 Processes MPI	16 Threads OpenMP	16 Processos MPI
computação	13,1639	9,6668	5,4144	2,62	1,6026	1,3143	1,1995	1,2371
comunicação	0,30757	0,3701	0,3982	1,2545	0,3612	0,3841	0,0502	1,8137

TABELA 7.6: Tempo total de Execução Decomposto para Malha 3 (em segundos)

	2 Threads OpenMP	2 Processos MPI	4 Threads OpenMP	4 Processos MPI	8 Threads OpenMP	8 Processos MPI	16 Threads OpenMP	16 Processos MPI
computação	80,6366	61,1241	30,5638	13,3801	9,9192	6,2511	3,3115	1,0491
comunicação	0,0407	0,2095	0,2696	9,1567	1,8897	1,8087	0,9871	3,6949

TABELA 7.7: Tempo total de Execução Decomposto para Malha 4 (em segundos)

	2 Threads OpenMP	2 Processos MPI	4 Threads OpenMP	4 Processos MPI	8 Threads OpenMP	8 Processos MPI	16 Threads OpenMP	16 Processos MPI
computação	348,724	457,864	244,3755	73,1356	73,437153	33,5724	30,9984	6,6594
comunicação	0,0481	0,3153	0,4668	19,268	12,2461	12,3646	5,5844	7,4463

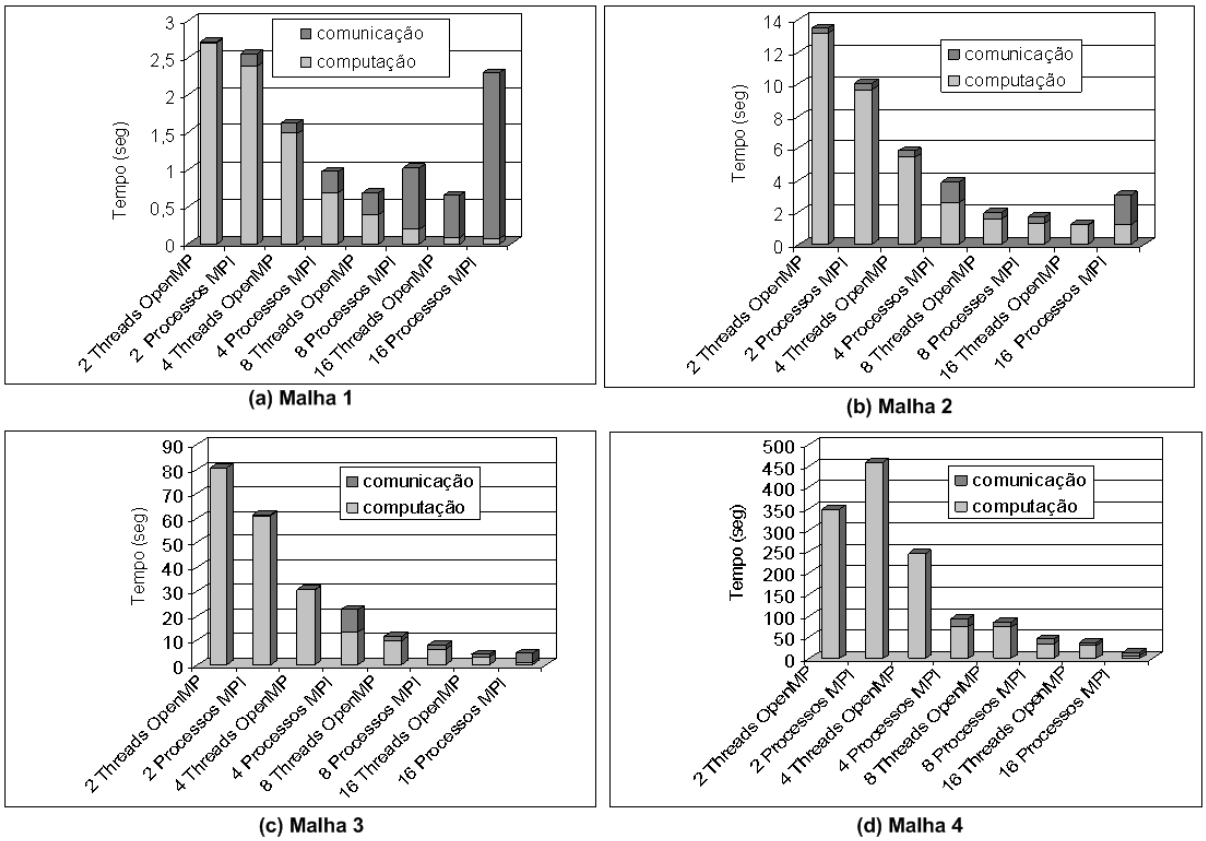

Figura 7.5 – Tempo de Execução Decomposto em Tempo de Computação e Tempo de Comunicação.

7.4.1 Tempos de Comunicação

Para uma melhor visualização, a figura 7.6 apresenta o gráfico com a comparação apenas do tempo de comunicação para as duas versões da aplicação e para cada uma das malhas utilizadas.

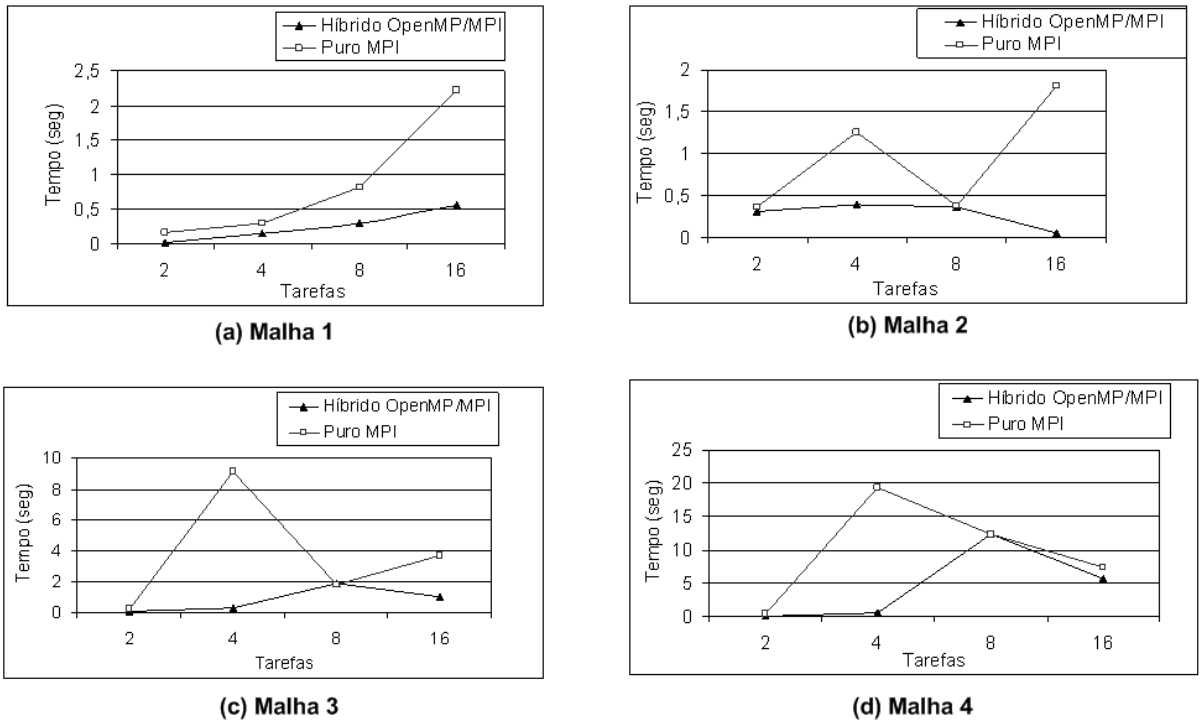

Figura 7.6 - Gráfico de comparação do tempo de comunicação para o programa híbrido e para o programa puro MPI.

7.5 Speedup Relativo: Versão Híbrida x Versão Pura MPI

A tabela 7.8 apresenta os valores das medidas dos *speedups* relativos para as quatro malhas utilizadas na versão híbrida e na versão pura MPI. A figura 7.7 apresenta os gráficos com as comparações dos *speedups*.

TABELA 7.8: Valores de *speedup* relativo.

Malha	2 Threads OpenMP	2 Processos MPI	4 Threads OpenMP	4 Processos MPI	8 Threads OpenMP	8 Processos MPI	16 Threads OpenMP	16 Processos MPI
1	1,9	2	3,4	5,3	7,4	5	7,9	2,2
2	1,9	2,6	4,8	7	12,8	14,8	20,2	10,3
3	1,9	2,5	5,1	7	13,2	19,4	36,4	33
4	1,7	1,5	2,8	7,3	7,9	14,7	18,5	48

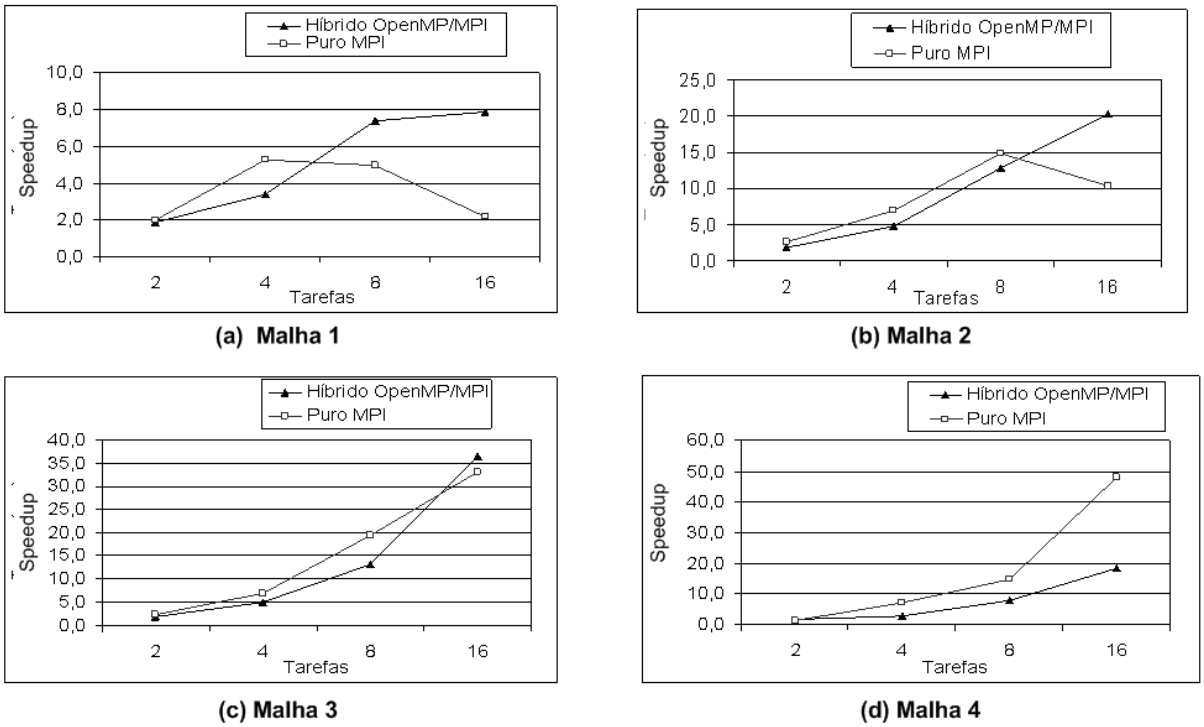

Figura 7.7 – Gráfico de *speedup* relativo para programa híbrido e puro MPI.

7.6 **Desempenho: Versão Híbrida x Versão Pura MPI**

A tabela 7.9 apresenta os valores das medidas de desempenho para as quatro malhas utilizadas. A figura 7.8 apresenta os gráficos com as comparações de desempenho.

TABELA 7.9: Valores de desempenho.

Malha	2 Threads OpenMP	2 Processos MPI	4 Threads OpenMP	4 Processos MPI	8 Threads OpenMP	8 Processos MPI	16 Threads OpenMP	16 Processos MPI
1	0,95	1,00	0,85	1,32	0,93	0,63	0,49	0,13
2	0,95	1,30	1,20	1,75	1,60	1,85	1,26	0,64
3	0,95	1,25	1,27	1,75	1,65	2,43	2,27	2,06
4	0,85	0,75	0,70	1,82	0,98	1,83	1,15	3,00

7.7 Análise dos Resultados

Os resultados experimentais mostram que na maioria dos casos o desempenho geral da versão pura MPI é melhor que o desempenho da versão híbrida OpenMP/MPI.

Na seção 7.5 referente ao *speedup* relativo, pode-se perceber que a versão híbrida alcança um bom desempenho, no entanto na maioria das situações, ele não é melhor do que aquele alcançado pela versão original em MPI da aplicação. Para o tempo total de execução na seção 7.3, ocorre o mesmo e a versão MPI obtém um melhor tempo de execução na maioria dos casos.

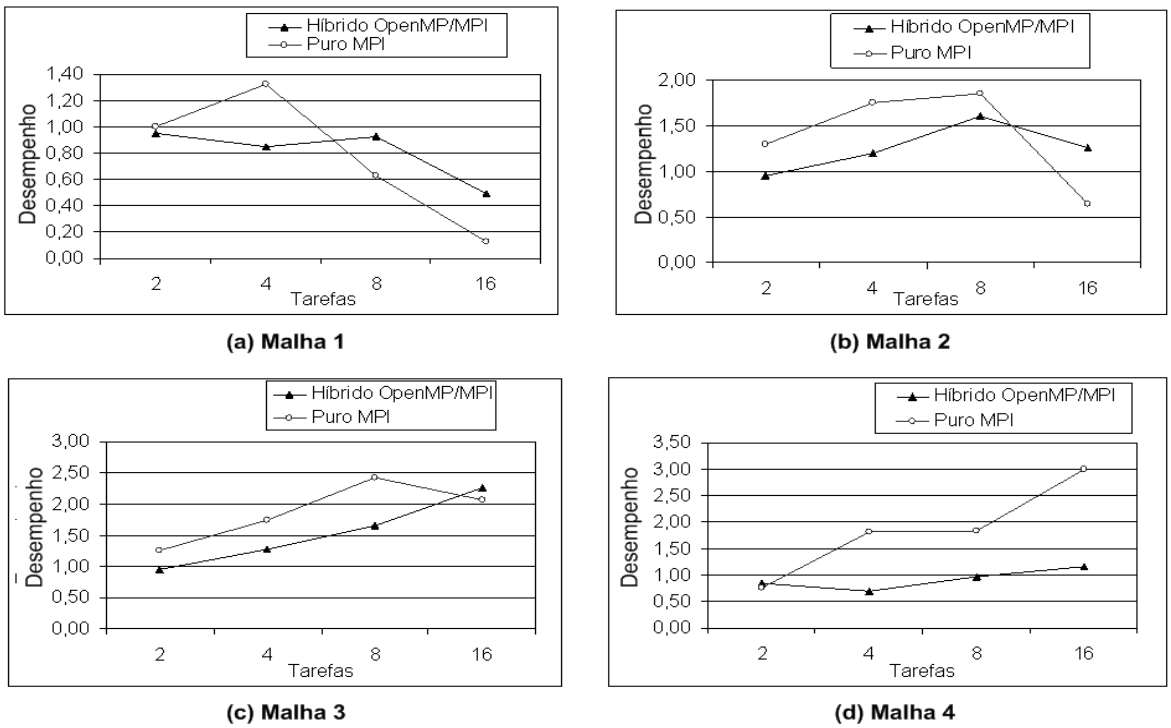

Figura 7.8 – Gráfico de desempenho para o programa híbrido e puro MPI.

Entretanto, para as malhas menores e com uma grande quantidade de máquinas, o tempo de execução da versão híbrida foi menor. Isso se deve ao fato de que para as execuções com essas malhas o tempo consumido é pequeno e a redução no tempo de comunicação é maior que o *overhead* introduzido.

Essa relação pode ser melhor visualizada observando-se os tempos de comunicação e de computação separadamente. Na seção 7.4 pode-se perceber que, em 99% das situações a versão híbrida possui tempo de comunicação menor que a versão pura MPI. No entanto, o tempo de computação da versão híbrida é maior na maioria dos casos o que indica que o programa híbrido, apesar de diminuir o tempo de comunicação, introduziu um *overhead* no tempo de execução maior do que o benefício obtido.

A introdução desse *overhead*, excessivo, pode ser explicada pelas principais fraquezas do modelo OpenMP que são [30]:

1) A multiplicação de regiões paralelas OpenMP implica em um custo de gerência de *thread* considerável.
2) As regiões paralelas provocam uma má utilização da hierarquia de memória.

O primeiro fator é especialmente importante nessa aplicação porque foi utilizada uma abordagem de grão fino. Além disso, pelo fato da aplicação usar um método iterativo para resolver o sistema de equações, *threads* precisam ser criadas muitas vezes.

A respeito do tamanho das malhas, a maior malha utilizada levou um tempo total seqüencial de aproximadamente 678 segundos. Seria interessante a investigação do comportamento da aplicação para malhas maiores e que consequentemente levassem mais tempo para serem executadas, porém execuções de malhas maiores que 3717 exigem mais memória do que está disponível atualmente no *cluster*.

É importante ressaltar que os resultados e observações obtidos a partir dessa versão híbrida da aplicação dizem respeito à abordagem de grão fino utilizada. Uma abordagem de grão grosso pode levar a resultados diferentes. Existem na literatura casos que obtiveram bons e maus resultados utilizando cada uma das duas abordagens [6]. Uma abordagem de grão fino, no entanto possui algumas vantagens sobre a abordagem de grão grosso, o que levou à escolha dessa a técnica para este trabalho. Dentre essas vantagens, talvez a mais importante é que a abordagem de grão fino possibilita a paralelização incremental do código. Isso significa que um código MPI já existente pode ser transformado em uma aplicação híbrida sem a necessidade de modificações na estrutura original do programa, apenas com a anotação de trechos do código com diretivas. Nessa abordagem pode-se inclusive obter as duas versões da aplicação a partir do mesmo código, compilado com ou sem suporte ao padrão OpenMP. A abordagem de grão grosso, por outro lado exige que a aplicação seja completamente reescrita para adotar um modelo SPMD para as *threads* OpenMP.

Modelos híbridos de programação para *clusters* de SMPs foram usados em diversas aplicações. Apesar de em alguns trabalhos como [42], [31] e [57] terem obtido um ganho de desempenho comparado a uma versão pura da mesma aplicação, na maior parte dos casos, observou-se uma perda de desempenho. Essa perda de desempenho foi reportada em trabalhos como [23], [33] e [50] entre outros. Em [36], um problema que utiliza o método dos gradientes conjugados, a solução híbrida superou o desempenho da versão pura apenas para um número considerável de *nós*.

Em [4], é demonstrado que o ganho de desempenho de uma aplicação híbrida é claramente dependente de aplicação. Nesse trabalho foram realizadas comparações para 6 diferentes *benchmarks* e em quatro deles o desempenho da versão original pura MPI foi melhor enquanto em outros dois a versão híbrida apresentou um melhor desempenho. O desempenho de cada modelo depende da aplicação, do tamanho dos dados e das características dos diferentes componentes da arquitetura (CPU, sistema de memória, sistema de interconexão).

CAPÍTULO 8

CONCLUSÃO

Esse trabalho desenvolveu e avaliou um modelo híbrido de programação paralela para uma aplicação de engenharia baseada no método dos elementos finitos. Para isso foi utilizada uma abordagem incremental para transformar a aplicação MPI de simulação desenvolvida em [59] em uma aplicação híbrida. Essa aplicação é capaz de utilizar troca de mensagens entre os *nós* de um *cluster* SMP e memória compartilhada dentro do *nó* SMP.

A hipótese inicial era que o tempo consumido com comunicação na aplicação seria reduzido, pois o uso desse modelo substitui a comunicação por troca de mensagens dentro do *nó* SMP por uma comunicação de memória compartilhada, mais rápida. Como conseqüência poderia se supor que o tempo total de execução também seria reduzido.

Essa hipótese se confirmou em parte já que o tempo de comunicação de fato foi reduzido. Entretanto, na maioria dos casos, o tempo total de execução aumentou em relação à aplicação pura por troca de mensagens.

Isso acontece porque a transformação da aplicação para usar memória compartilhada introduz um *overhead* maior do que o ganho obtido com comunicação. Essa perda foi observada em outros trabalhos na literatura como [23], [33] e [50]. Embora outros trabalhos como [42], [31] e [57] tenham obtido um ganho de performance, o que se constata é que o ganho de desempenho de uma aplicação híbrida depende de características da aplicação. Portanto, a escolha entre uma abordagem pura MPI e uma híbrida, para uma aplicação voltada para *clusters* de SMPs não é trivial.

Apesar disso, a redução no tempo de comunicação obtida, indica que esta pode ser uma estratégia útil para aplicações com requisitos de comunicação mais restritos ou com sérios problemas de escalabilidade. A aplicação desenvolvida, de fato, obteve um bom desempenho geral, com um *speedup* satisfatório, apesar de na maioria dos casos ter sido menos eficiente no tempo total de execução do que a versão pura MPI.

O padrão OpenMP se apresentou como uma estratégia de programação paralela extremamente simples e elegante. Por isso sugere-se que, em trabalhos futuros, esse padrão seja utilizado para resolução de problemas semelhantes de simulação em máquinas SMP como uma maior quantidade de *nós.*

Outros trabalhos futuros possíveis, especificamente voltados para modelos híbridos de programação, incluem a transformação dessa aplicação em uma versão de grão grosso para avaliar o desempenho dessa estratégia. E ainda a utilização da abordagem de sobreposição de comunicação apresentada em [58] para buscar um desempenho melhor do que aquele obtido neste trabalho.

REFERÊNCIAS

[1] – BASEGROUP LABS. **Method of Conjugate Gradients: Mathematical Apparatus**, 2006. Disponível em: http://www.basegroup.ru/neural/conjugate.en.htm

[2] – BATHE, K.-J. **Finite Element Procedures**. Prentice-Hall, Upper Saddle River, N.J. 1996.

[3] – BRUNSCHEN, C.; BRORSSON M. **OdinMP/CCp: A portable implementation of OpenMP for C**. In: European Workshop on OpenMP, September 1999.

[4] – CAPPELLO, F.; ETIEMBLE, D. **MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks**. In Supercomputing 2000, 2000

[5] – CHANDRA, R. et al. **Parallel Programming in OpenMP**. Morgan-Kaufmann, ISBN: 1558606718, 2000

[6] – CHOW , E.; HYSOM , D. **Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters**. In: Lawrence Livermore National Laboratory Technical Report UCRL-JC-143957, May 2001.

[7] – CIMNE - INTERNATIONAL CENTER FOR NUMERICAL METHODS IN ENGINEERING **GiD 8.0 reference manual**: **Pre and post processing system for F.E.M. calculations.** May 2006. Disponível em: http://gid.cimne.upc.es/support/gid_toc.subst

[8] – COMPUTATIONAL SCIENCE EDUCATION PROJECT. **Computer Architecture.** 1995**.** Disponível em: http://www.ipp.mpg.de/de/for/bereiche/stellarator/Comp_sci/CompScience/csep/csep1.phy.ornl.gov/ca/ca.html

[9] – COOK, R. D. (1995). **Finite Element Modeling for Stress Analysis**. John Wiley, New York.

[10] – CULLER, D.E.; SINGH, J.P.; Gupta, A. **Parallel Computer Architecture: A *Hardware*/Software Approach**. Morgan Kaufmann Publishers Inc. August 1998

[11] – DAGUM, L. ; MENON, R.. OpenMP**: An industrystandard API for shared-memory programming**. In: IEEE Computational Science & Engineering, 5(1):46--55, 1998.

[12] – DROSINOS, N. **Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters.** In IPDPS 2004, 2004

[13] – DROSINOS, N. ; KOZIRIS, N. **Advanced Hybrid MPI/OpenMP Parallelization Paradigms for Nested Loop Algorithms onto Clusters of SMPs.** In: PVM/MPI 2003, 2003.

[14] – EL-REWINI, H; LEWIS, T**. Distributed and Parallel Computing,** Manning, Greenwich, CT.

[15] – FELIPPA, A. **Introduction to Finite Element Methods**. Disponível em: http://www.colorado.edu/engineering/CAS/courses.d/IFEM.d/Home.html

[16] – FLYNN, M. **Some Computer Organizations and Their Architectures**. IEEE Transaction on Computers. Vol. C-21, 1972

[17] – FOSTER, I. **Designing and Building Parallel Programs.** Addison-Wesley, 1995

[18] – GATLIN, K. S.; ISENSEE , P. **Reap the Benefits of Multithreading Without All The Work.** In: MSDN Magazine, October 2005

[19] – GHARACHORLOO, K.; SCALES, D. J.; AGGARWAL, A. **Fine-grain Software Distributed Shared Memory on SMP clusters**. In: Proc. of the 4th IEEE Symp. On High-Performance Computer Architecture (HPCA-4), February 1998.

[20] – GEORGE, K.; VIPIN, K. METIS: **A Software Package for Partitioning Unstructured Graph, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices**. In: Manual, University of Minnesota, Department of Computer Science 1988.

[21] – GUSTAFSON, J.L. **Reevaluating Amdahl's Law.** In: Communications of ACM. Volume 31, Issue 5, 1998

[22] – HE, Y.; DING, C.H.Q. **MPI and OpenMP Paradigms on Cluster of SMP Architectures: the Vacancy Tracking Algorithm for Multi-Dimensional Array Transposition**

[23] – HENTY, D.S. **Peformance of Hybrid message-passing and shared-memory parallelism for discrete element modeling** In: Proc. Supercomputing 2000, Dallas, TX.

[24] – HIBBELER, R.C. **Mechanics of Materials**. Prentice-Hall. 1997. Upper Saddle River, N. J.

[25] – HONGHUI, C. H. ;COX, L. A.; ZWAENEPOEL, W. **OpenMP for Networks of SMPs**. In: Proc. of the Second Merged Symp. IPPS/SPDP 99, 1999.

[26] – HWANG, K.; XH, Z. **Scalable Parallel Computing: Technology, Architectures, Programming.** McGraw-Hill, 1998.

[27] – INTEL CORPORATION. **Using the RDTSC instruction for performance monitoring.** 1997. Disponível em:
cedar.intel.com/software/idap/media/pdf/rdtscpm1.pdf

[28] – INTEL CORPORATION. **Extending OpenMP to Clusters**. Techinical Report, 2006. Disponível em:
http://cache-www.intel.com/cd/00/00/28/58/285865_285865.pdf

[29] – JIMACK , P.K.; TOUHEED, N. **Developing Parallel Finite Element Software Using MPI**. In: High Performance Computing for Computational Mechanics, ed. B.H.V. Topping and L. Lammer (Saxe-Coburg Publications), 15--38, 2000.

[30] – KRAWEZIK, K.; CAPELLO, F. **Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors**. In: SPAA 03, June 7-9, 2003. San Diego California, USA

[31] – LOFT, R. D.; THOMAS, S. J.; DENNIS, J. M. **Terascale Spectral Element Dynamical Core for Atmospheric General Circulation Models**. In: proc. SC 2001, Nov 2001, Denver USA.

[32] – LUMETTA S. S.; MAINWARING, A.; CULLER, D. E. **Multi-protocol active messages on a cluster of SMPs**. In: SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/IEEE SC97 Conference: November 15–21, 1997, San Jose, California, USA., 1997.

[33] – MAVRIPLIS, D.J. **Parallel Performance Investigation of an Unstructured Mesh Navier-Stokes Solver.** Technical Report. ICASE, Hampton, VA, 2000.

[34] – MOORE, G. **Crammin More Components Onto Integrated Circuits**. In: Eletronic Magazine, 1965.

[35] **–** MPI FORUM. **MPI: A Message Passing Interface**. In: Supercomputing '93, November 1993.

[36] **–** NAKAJIMA, K. **OpenMP/MPI Hybrid vs. Flat MPI on the Earth Simulator: Parallel Iterative Solvers for Finite Element Method.** In: RIST/TOKYO GeoFEM Report Research Organization for Information Science & Technology, October 2003

[37] – NI, L.M.; SUN, X**. Another View On Parallel Speedup.** In:Proceedings of the 1990 conference on Supercomputing. New York, United States, 1990.

[38] – **OpenMP Application Programming Interface Version 2.5, 2005.**
Disponível em:
http://www.openmp.org.

[39] – PARAMOUNT GROUP. **Program Parallelization and Tuning Methodology,** 2000. Disponível em:
http://peak.ecn.purdue.edu/ParaMount/UMinor/methodology.html

[40] – PATTERSON, D. A.; HENNESSY, J. **Arquitetura de Computadores uma Abordagem Quantitativa**. Ed Campus: 2003.

[41] **–** RABENSEIFNER, R; **Hybrid Parallel Programming: Performance Problems and Chances**. In: The Proceedings of the 45th CUG Conference 2003, Columbus, Ohio, USA, May 2003

[42] – RABENSEIFNER, R..; GRABYSZ, A. D. **Nesting OpenMP in MPI to Implement a Hybrid Communication Method of Parallel Simulated Annealing on a Cluster of SMP Nodes**. In: Proceedings , EuroPVM/MPI 2005 – 18-21, Sorrento Italy, LNCS 2005.

[43] – RAO, P.; JAGDISH, D.; NAIK, S.; SHIRODKAR, V. S.; BHALCHANDRA, M.; KHANDEPARKER, S. **The Finite Element Method**. Department Of Mechanical Engineering; Goa College Of Engineering – Farmagudi, Goa.

[44] – ROYLANCE, D. **Introduction to Elasticity**. Department of Materials Science and Engineering, Cambridge, January 2000. Disponível em:
http://web.mit.edu/course/3/3.11/www/modules/elas_1.pdf.

[45] – SAMANTA, R.; BILAS, A.; IFTODE, L.; SINGH, J. P. **Homebased SVM protocols for SMP clusters: Design and performance**. In Proc. of the 4th IEEE Symp. On High-Performance Computer Architecture (HPCA-4), February 1998.

[46] – SEVERANCE, C.; DOWD, K. **High Performance Computing,** Second Edition, OReilly, July 1998.

[47] – SHAN, H. S.; SINGH, J. P. **A Comparison of MPI, SHMEM, and Cache-Coherent Shared Address Space Programming Models on the SGI Origin2000**. In: Proc. of the International Conference on Supercomputing, ICS, June 1999

[48] – SHEWCHUK, J.R.. **An Introduction to the Conjugate Gradient Method Without the Agonizing Pain**. Technical Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, 1994.

[49] – SMITH, L.; BULL, M. **Development of Mixed Mode MPI/ OpenMP Applications**. In: Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2000), July 2000.

[50] – SMITH, L.; KENT, P. **Development and Performance of a Mixed OpenMP/MPI Quantum Monte Carlo Code**. In: Concurrency: Practice and Experience, 12:1121-1129, 2000.

[51] – SNIR, M.; OTTO, S.; HUSS-LEDERMAN, S.; WALKER, D.; DONGARRA, J. **Mpi The Complete Reference**. The MIT Press, Cambridge Masacucets, 1996.

[52] – STROHMAIER, M. **Recent Trends in Marktplace of High Performance Computing.** In: CTWatch Quarterly Volume 1, Number 1, February 2005

[53] – TANEMBAUM, A. S. **Distributed Operating Systems**. Prentice Hall, 1st edition, August 25 1994.

[54] – THE OPEN GROUP. **Threads and the Single Unix Specification, Version 2**. May 1997. X/Open Company Ltd., U.K.

[55] – **TOP 500 Supercomputers**. Novembro de 2006
Disponível em:
www.top500.org

[56] – VAN DER PAS , R. **Using OpenMP to parallelize for CMT.**
Disponível em:
http://blogs.sun.com/ruud/entry/using_openmp_to_parallelize_for

[57] – VIET, T. Q., YOSHINAGA, T.; ABDERAZEK, B. A.; and SOWA, M. **A Hybrid MPI-OpenMP Solution for a Linear System on a Cluster of SMPs**.
In: Symposium on Advanced Computing Systems and Infrastructures 2003.

[58] – VIET, T. Q.; YOSHINAGA, T.; ABDERAZEK, B. A.; SOWA, M.: **Construction of Hybrid MPI-OpenMP Solutions for SMP Clusters**.In: *IPSJ Digital Courier*, Vol. 1, pp.153-165. (2005).

[59] – VILLA VERDE, F.R. **Parallel solution for a finite element code applied to a linear elasticity on a cluster of PCs**, 2004. MSc. Dissertation in Mechanical Engineering, University of Brasilia

[60] – VILLA VERDE, F. R. ; FERREIRA, R.; PFITSCHER, G. H. **Solução Paralela em Agregado de PCs de um Código de Elementos Finitos Aplicado à Elasticidade Linear**. In: I Congresso Sul Catarinense de Computação

[61] – WOLF, F. **Automatic Performance Analisys on Parallel Computers with SMP Nodes**. In: NIC Series Volume 17, NIC- Directors , Germany 2003.