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

CARACTERIZAÇÃO DE DESEMPENHO DE UMA APLICAÇÃO PARALELA DO MÉTODO DOS ELEMENTOS FINITOS EM AMBIENTES HETEROGÊNEOS DE PCs

Roberta Ribeiro Ferreira

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

Orientador: Prof. Dr. Gerson Henrique Pfitscher

Brasília
2006

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

CARACTERIZAÇÃO DE DESEMPENHO DE UMA APLICAÇÃO PARALELA DO MÉTODO DOS ELEMENTOS FINITOS EM AMBIENTES HETEROGÊNEOS DE PCs

ROBERTA RIBEIRO FERREIRA

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. Dr. Mário Antonio Ribeiro Dantas
INE/UFSC

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

Brasília, 27 de julho de 2006.

CIP – Catalogação Internacional na Publicação

Roberta Ribeiro Ferreira.

Caracterização de Desempenho de uma Aplicação Paralela do Método dos Elementos Finitos em Ambientes Heterogêneos de PCs / Roberta Ribeiro Ferreira. Brasília: UnB, 2006. 137 p. : il. ; 29,5 cm.

Dissertação (Mestrado) – Universidade de Brasília, Brasília, 2006.

1. Computação paralela, 2. *Clusters* heterogêneos de PCs, 3. Método dos elementos finitos, 4. Desempenho

Dedicado a

João Ribeiro, Ernestina Ribeiro e Evandro Guimarães,
pelos exemplos de amor ao próximo, de respeito e
de vida deixados.

AGRADECIMENTOS

A Deus, fonte de vida e de sabedoria, que me concedeu a graça de ter chegado até aqui.

Ao meu amigo e orientador professor Dr. Gerson Henrique Pfitscher pela dedicação, orientação, paciência mesmo nos momentos mais difíceis, atenção, apoio e por sua grande amizade que me foi tão importante durante a realização deste trabalho.

Aos meus pais, Helvécio e Maria Aparecida, que sempre me incentivaram e me apoiaram em todas as minhas decisões. Agradeço ao amor, ao carinho, à educação e aos ensinamentos que sempre me dedicaram.

Ao Marcos pela constante motivação e pelo incentivo em enfrentar e superar as dificuldades que existiram; agradeço também pela grande ajuda, pela paciência e pela dedicação.

À amiga Flávia Romano Villa Verde pela ajuda técnica e, principalmente, por sua amizade. Agradeço por ter-me feito acreditar, em vários momentos, que era válido o esforço necessário para a conclusão desta dissertação.

Aos grandes amigos Débora Nery, Helton Garcia, James Van, Jeovânio Bitencourte e Jorge de Oliveira pelas horas de estudo e de aprendizado que compartilhamos juntos.

Aos amigos Roberto Elias Cavalcante e Francisco Ricardo Lima pelo apoio e pela compreensão que favoreceram essa realização.

Aos professores e funcionários do Mestrado em Informática da UnB.

RESUMO

CARACTERIZAÇÃO DE DESEMPENHO DE UMA APLICAÇÃO PARALELA DO MÉTODO DOS ELEMENTOS FINITOS EM AMBIENTES HETEROGÊNEOS

Na área das Engenharias muitos problemas complexos de soluções extremamente trabalhosas e normalmente impossíveis de serem resolvidos por métodos analíticos exigem soluções numéricas. O método dos elementos finitos (MEF) é uma abordagem para solução destes problemas encontrados em análise de estruturas, fluídos, eletromagnetismo, modelagem de circuitos integrados, biomédica e transferência de calor que necessitam processamento de alto desempenho e trabalham com grandes volumes de dados.

A computação paralela aparece como uma alternativa viável para a obtenção do desempenho necessário para a solução de problemas através do MEF e a utilização de *clusters*, como alternativa para a obtenção deste desempenho a baixo custo, se comparados a outros sistemas de computação paralela. Contudo, em ambientes heterogêneos, para que o paralelismo seja explorado eficientemente é fundamental uma distribuição balanceada da carga de trabalho. Para isto, se faz necessário o conhecimento detalhado dos custos de execução e comunicação envolvidos no processamento da aplicação paralela, nas diferentes máquinas do ambiente.

Este trabalho tem como objetivo caracterizar o desempenho, através de medições de tempo de execução detalhadas, de um código paralelo para um problema de análise estrutural modelado pelo método dos elementos finitos e resolvido pelo método dos gradientes conjugados (MGC), em um ambiente heterogêneo de PCs.

Através dos resultados obtidos com as medições detalhadas, foi possível estabelecer um balanceamento de carga empírico para o ambiente heterogêneo, mostrando a viabilidade da utilização deste ambiente para a execução do código paralelo do método dos elementos finitos.

ABSTRACT

There are many complex problems of extremely difficult solutions in engineering area. These complex problems are usually impossible to be resolved for analytic methods and then they demand numeric solutions. The Finite Elements Method (FEM) is an approach for the solution of these problems found in analysis of structures, fluids, electromagnetism, assembling of integrated circuits, biomedical and transfer of heat, that need high performance processing and work with great volumes of data.

The parallel computing appears as a viable alternative for obtaining the necessary performance for the solution of problems through the FEM and the use of cluster appears as an alternative for obtaining this performance processing at a low cost, if compared with the other systems of parallel computing. However, in heterogeneous environments, for an efficiently exploration of the parallelism, it is fundamental a balanced distribution of the work load. For this, it is necessary the knowledge of the execution costs and the communication involved in the processing of the parallel application, in the different machines of the environment. This work has as objective to characterize the performance of a parallel code for a problem of structural analysis modeled by the Finite Elements Method and solved by the Conjugated Gradients Method, in a heterogeneous environment.

Through the results obtained with the detailed measurements, it was possible to establish an empiric load balancing for the heterogeneous environment, showing the viability of the use of this environment for the execution of the parallel code of the Finite Elements Method.

Sumário

Lista de Figuras

Lista de Tabelas

1 – INTRODUÇÃO

A busca por maior velocidade no processamento das informações foi sempre um desafio para a computação e ao mesmo tempo um dos maiores responsáveis pelo seu desenvolvimento.

Não são poucas as situações em que o uso do processamento paralelo de alto desempenho se torna necessário: problemas complexos como modelagem numérica para previsão de tempo, gerência de grandes bancos de dados, e também os problemas de engenharia modelados pelo método dos elementos finitos (MEF), dentre outros.

Na área das Engenharias muitos problemas complexos de soluções extremamente trabalhosas e normalmente impossíveis de serem resolvidos por métodos analíticos exigem soluções numéricas. O método dos elementos finitos (MEF) é uma abordagem para solução destes problemas encontrados em análise de estruturas, fluídos, eletromagnetismo, modelagem de circuitos integrados, biomédica e transferência de calor que necessitam processamento de alto desempenho e trabalham com grandes volumes de dados.

Por oferecer um elevado poder computacional, a computação paralela aparece como uma alternativa viável para a obtenção do desempenho necessário para a solução de problemas através do MEF e a utilização de *clusters*, como alternativa para a obtenção deste desempenho a baixo custo, se comparados a outros sistemas de computação paralela.

Um *cluster* é uma coleção de computadores completos, ou nós, que estão fisicamente interconectados por uma rede de alto desempenho ou uma rede local, LAN. Tipicamente, cada nó é um servidor SMP, ou uma *workstation*, ou um PC. O aspecto mais importante é que todos os nós devem ser capazes de trabalhar coletivamente como um recurso computacional único integrado, além de permitir o uso individual de cada nó por usuários [1]. Os *clusters* de PCs têm sido amplamente utilizados como máquinas paralelas. A disponibilização de bibliotecas de comunicação como o PVM (*Parallel Virtual Machine*) [2] [3] e o MPI (*Message Passing Interface*) [4] [5] [6] [7], que permitem a execução de programas paralelos em um conjunto de máquinas conectadas por uma rede local, favoreceu a utilização de *clusters*.

Um *cluster* homogêneo tende a se tornar heterogêneo pela adição de máquinas extras, devido à constante evolução e rápida obsolescência tecnológica. Como em ambientes heterogêneos as máquinas possuem poder computacional distinto, para que o paralelismo seja explorado eficientemente é fundamental uma distribuição balanceada da carga de trabalho. Para isto, se faz necessário o conhecimento dos custos de execução e comunicação envolvidos no processamento da aplicação paralela, nas diferentes máquinas do ambiente heterogêneo.

As aplicações paralelas estão normalmente relacionadas ao desempenho. Contudo, muitas vezes os valores de desempenho obtidos não são suficientes para caracterizar o completo funcionamento de uma aplicação paralela. Este trabalho tem como objetivo caracterizar o desempenho, através de medições de tempo de execução detalhadas, de um código paralelo para um problema de análise estrutural modelado pelo método dos elementos finitos e resolvido pelo método dos gradientes conjugados (MGC), em um ambiente heterogêneo de PCs.

Através dos resultados obtidos com as medições detalhadas, foi possível estabelecer um balanceamento de carga empírico para o ambiente heterogêneo, mostrando a viabilidade da utilização deste ambiente para a execução do código paralelo do método dos elementos finitos.

Neste trabalho, é utilizado como aplicação *benchmark* um código paralelo aplicado a um problema da engenharia. O código paralelo é uma solução do método dos elementos finitos aplicado à elasticidade linear, onde o sistema de equações algébricas é resolvido pelo método dos gradientes conjugados [8]. A aplicação foi implementada na linguagem C, com a utilização da biblioteca de troca de mensagens MPI (*Message Passing Interface*).

Esta dissertação está dividida como descrito a seguir. O capítulo 2 apresenta as motivações e conceitos sobre a computação paralela, as vantagens da utilização de *cluster* de computadores e as preocupações pertinentes à utilização de *clusters* heterogêneos. O capítulo 3 apresenta as medidas de desempenho a serem utilizadas neste trabalho. O capítulo 4 dá uma introdução ao método dos elementos finitos, ao método dos gradientes conjugados e apresenta a aplicação utilizada como *benchmark* para a avaliação de desempenho do *cluster* heterogêneo. O capítulo

5 apresenta os resultados obtidos, com as devidas análises destes. Finalmente, o capítulo 6 exibe a conclusão do trabalho e os trabalhos futuros.

2 – COMPUTAÇÃO PARALELA E ARQUITETURA DE *CLUSTERS*

2.1 – Evolução da arquitetura computacional

Em muitas áreas, tais como engenharia e automatização, há uma grande demanda de desempenho computacional [9]. As aplicações científicas utilizadas nestas áreas ainda são as forças que impulsionam a computação paralela necessária para manipular uma grande quantidade de informações, acelerar a execução do código e resolver problemas que consomem tempo excessivo [10].

Anteriormente, a busca pelo desempenho computacional obtido com a computação paralela se deu com base na utilização de computadores paralelos específicos, como aqueles com arquiteturas massiçamente paralelas (MPP) e os de memória compartilhada (SMP) [11]. Porém, a arquitetura MPP necessita de um alto investimento inicial, fazendo com que haja uma redução no número de pessoas que possam adquirir tais computadores, o que, por sua vez, faz com que exista pouco suporte de *software* para esta arquitetura. Como resultado, as arquiteturas MPP têm um custo elevado e pouca flexibilidade, apresentando dificuldade de atualização e de manutenção [9].

A arquitetura SMP apresenta uma limitação em relação ao número de CPUs suportadas, devido à utilização da memória compartilhada. Desta forma, esta arquitetura também apresenta pouca flexibilidade [9].

Os *clusters* aparecem como uma tendência na computação de alto desempenho. Nos *clusters*, *workstations* de alto poder computacional e baixo custo e/ou PCs são interligadas através de interfaces de comunicação rápidas para alcançar a computação paralela de alto desempenho. Os recentes aumentos de velocidade na comunicação e na freqüência de *clock*, juntamente com a disponibilidade de *softwares* de domínio público, incluindo sistemas operacionais, ferramentas de compilação e bibliotecas de troca de mensagens, fazem dos *clusters* uma alternativa economicamente viável para se obter alto desempenho [10].

As principais características dos *clusters*, como boa relação de desempenho *versus* custo, rapidez de processamento, confiabilidade e concorrência, serviram para colocá-los como uma das propostas mais atraentes e viáveis nos mais diversos ambientes.

2.2 – Computação distribuída e computação paralela

Existem atualmente vários conceitos a respeito do que vem a ser sistemas distribuídos, sistemas paralelos e sistemas concorrentes. Esses conceitos de alguma forma acabam se confundindo. Entre as definições existentes, há somente um ponto que todas elas concordam: todos esses sistemas requerem a presença de múltiplos processadores [12].

Segundo El-Rewini e Lewis [13], a computação distribuída é mais geral e universal do que a computação paralela. A distinção é sutil, mas importante. Paralelismo é uma forma restrita de computação distribuída. Computação paralela é uma computação distribuída onde todo o sistema está dedicado à solução de um único problema no menor tempo possível.

Existem basicamente três questões que distinguem a programação distribuída da programação seqüencial [12]:

1. O uso de múltiplos processadores.
2. A cooperação entre os processadores.
3. O potencial para falhas parciais: já que os processadores são autônomos, uma falha em um processador não afeta o funcionamento correto dos outros processadores.

Programas distribuídos executam, em paralelo, pedaços de seus códigos em processadores diferentes. Aplicações de alto desempenho usam este paralelismo para obter aumento de velocidade. O objetivo é obter o máximo de proveito dos processadores disponíveis, por isso, decisões relativas a quais computações devem ser executadas em paralelo são de grande importância. Em aplicações tolerantes a falhas, decisões para executar funções em diferentes processadores são baseadas em aumentar a confiabilidade ou a disponibilidade [12].

Para se conceituar a programação paralela, é preciso inicialmente abordar os conceitos de programa seqüencial e programa concorrente. Um programa seqüencial especifica a execução

seqüencial de uma lista de sentenças. Esta execução é chamada de processo. Um programa concorrente é um programa que define ações que podem ser executadas simultaneamente. O programa concorrente especifica dois ou mais programas seqüenciais que podem ser executados concorrentemente como processos paralelos [14].

Um programa concorrente pode ser executado permitindo que os processos compartilhem o processador ou que cada processo execute em seu próprio processador. A primeira abordagem é conhecida como multiprogramação. A segunda abordagem é conhecida como multiprocessamento, onde os processadores compartilham uma memória comum, ou é conhecida como processamento distribuído, no qual os processadores estão conectados por uma rede de comunicação [15].

Assim, programa concorrente é um termo genérico usado para descrever qualquer problema envolvendo comportamento paralelo real ou potencial. Programas paralelos e distribuídos são subclasses de programas concorrentes que são concebidas para execução em ambientes de processamento paralelo específicos [14].

Como um sistema distribuído tem, por definição, mais de um processador, é possível que se tenha mais de uma parte de um programa executando ao mesmo tempo, o que vem a constituir paralelismo. Quando um programa expresso de forma concorrente é executado em um único processador, a concorrência envolvida costuma ser chamada de pseudoparalelismo. Por outro lado, quando ele é executado em mais de um processador, a concorrência é chamada de paralelismo [12].

No projeto de um sistema distribuído, existem alguns aspectos-chave que devem ser tratados: transparência, flexibilidade, confiabilidade, desempenho e escalabilidade. A transparência provavelmente é o tópico mais importante de um sistema distribuído e diz respeito ao provimento de uma imagem única do sistema, ou seja, dar ao usuário a ilusão de estar usando um único sistema compartilhado composto por um único processador. Quanto à flexibilidade, um sistema distribuído deve ser flexível, isto é, novos recursos e serviços podem ser adicionados a ele com facilidade [16].

Um dos aspectos da confiabilidade é a disponibilidade, que se refere ao período de tempo em que o sistema pode ser utilizado plenamente. Outro aspecto é a tolerância a falhas, onde o sistema pode mascarar a ocorrência de uma falha. O desempenho é um dos principais objetivos buscados ao se construir um sistema distribuído, que deve apresentar um ganho em relação aos sistemas monoprocessados [16].

A escalabilidade se refere à capacidade de ter os recursos melhorados ou aumentados para absorver qualquer demanda de desempenho ou funcionalidade, ou reduzidos para diminuir custos [16].

Um dos maiores problemas relacionados à implementação de uma aplicação distribuída é a exploração eficiente dos recursos do sistema. Para atingir este objetivo, é necessário que a aplicação seja subdividida em módulos menores que possam ser distribuídos entre as unidades de processamento de forma a otimizar algum critério pré-estabelecido, que normalmente consiste em minimizar o tempo de execução total da aplicação. Alguns dos critérios mais utilizados são [17]:

- Minimização do tempo de execução total da aplicação.
- Balanceamento da carga computacional da aplicação.
- Maximização da utilização dos recursos do sistema.
- Minimização do número de processadores e de recursos adicionais necessários para a execução de um grupo de tarefas em um intervalo de tempo dado.
- Maximização do *throughput* do sistema.

Arquiteturas paralelas têm se tornado o suporte principal para a computação científica, incluindo física, química, biologia, astronomia e engenharias, entre outras. A engenharia de aplicação das ferramentas para modelar fenômenos físicos hoje em dia é essencial para muitas indústrias, incluindo indústria petrolífera (modelagem de reservatório), indústria automotiva (simulação de colisão, eficiência da combustão), indústria aeronáutica (análise da corrente de ar, eficiência do motor, eletromagnetismo), indústria farmacêutica (modelagem molecular) e outras. Em quase todas essas aplicações há uma grande demanda pela visualização dos resultados, o que as torna aplicações que necessitam de processamento paralelo [18].

2.2.1 – Tipos de paralelismo

O desempenho da solução vai depender da quantidade de paralelismo que for identificado no problema. Muitas vezes o processo de paralelização conduz a seqüências de funções ou operações similares, não necessariamente idênticas, sendo executadas em elementos de grandes estruturas de dados [13]. É o chamado paralelismo de dados. Neste caso, é requerido que o programador ofereça informação sobre como os dados serão distribuídos entre os processadores, em outras palavras, como os dados serão particionados entre as tarefas. Técnicas de decomposição de domínio deverão ser aplicadas sobre os dados sobre os quais se está operando [19].

Além desse tipo de paralelismo, as aplicações exibem também um paralelismo funcional, também chamado de paralelismo de controle ou paralelismo de tarefas: cálculos totalmente diferentes podem ser executados concorrentemente nos mesmos dados ou em conjuntos de dados distintos [19]. No paralelismo funcional, o foco está na computação (operação) que está sendo realizada e não nos dados que estão sendo manipulados pela computação. As computações são divididas em tarefas distintas. Os dados operados podem ou não ser distintos. No caso de estes não serem distintos, uma quantia considerável de comunicação será necessária para evitar a replicação dos dados comuns [19].

2.3 – Classificação das arquiteturas paralelas

Existem várias classificações possíveis para máquinas paralelas. A taxonomia proposta por Flynn, conforme ilustra a Figura 2.1, é a mais conhecida [20].

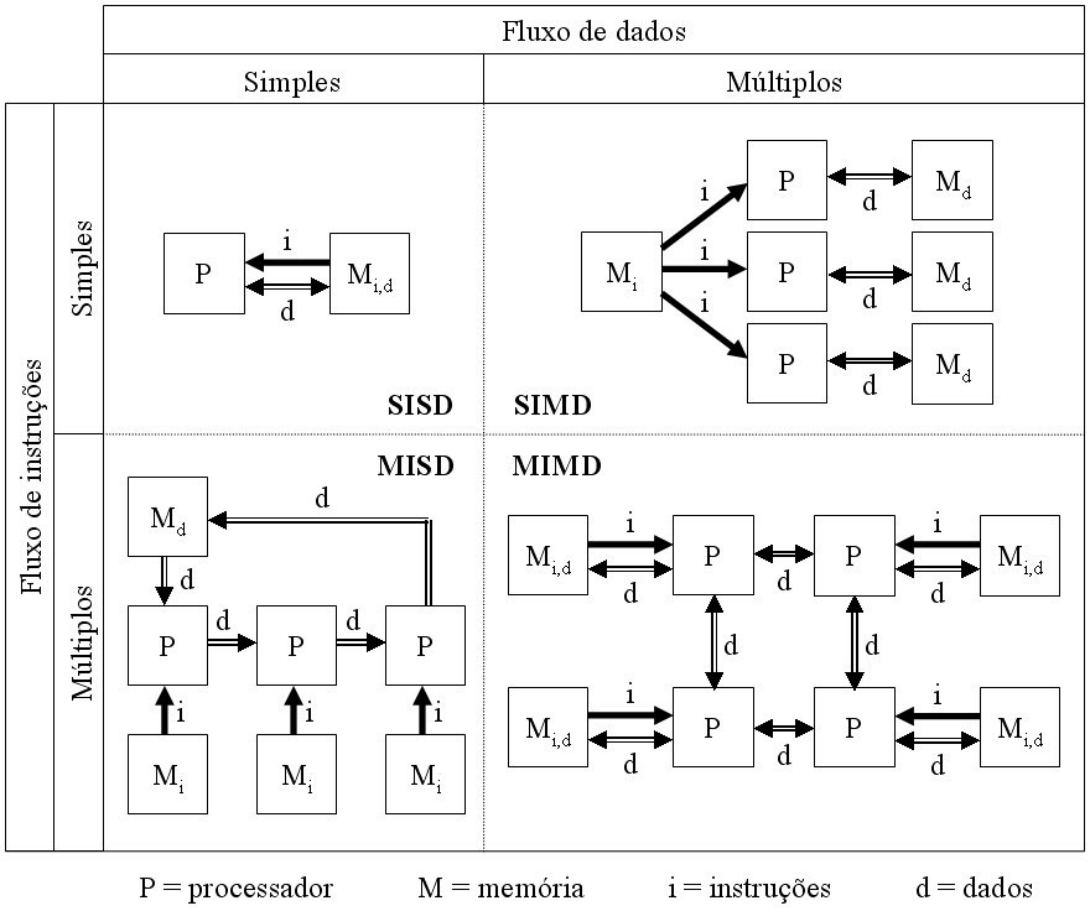

Figura 2.1 – Taxonomia de Flynn para computadores.

A classificação proposta por Flynn [20] distingue as arquiteturas com base no fluxo de instruções e no fluxo de dados a serem processados simultaneamente. Segundo a classificação de Flynn, são distinguidas quatro categorias de arquitetura existentes, descritas abaixo:

SISD – *Single Instruction, Single Data*. Esta categoria representa as máquinas monoprocessadas tradicionais. Nesta categoria, uma instrução é executada de cada vez, manipulando um único fluxo de dados. Caracteriza o modelo seqüencial (Arquitetura de von Neumann), cujo princípio básico de funcionamento identifica a grande maioria dos computadores.

SIMD – *Single Instruction, Multiple Data*. Caracterizado por um fluxo único de instruções, que são aplicadas sobre múltiplos fluxos de dados. Todos os processadores executam o mesmo fluxo de instrução em diferentes fluxos de dados. Constitui uma classe de arquiteturas muito importante na história da computação. Para certas classes de problemas, este tipo de arquitetura é perfeitamente apropriado para se atingir altas taxas de processamento, pois os

dados são separados em diversas partes, e as múltiplas unidades de instrução podem operar sobre elas ao mesmo tempo.

MISD – *Multiple Instruction, Single Data*. Não se tem conhecimento de arquitetura de máquinas desta categoria. Máquinas da categoria MISD possuem múltiplas funções executando independentemente operações num único fluxo de dados. Abstratamente, uma máquina MISD é um *pipeline* de múltiplas unidades funcionais de execução independente operando sobre um único fluxo de dados, repassando os resultados de uma unidade funcional para a próxima [21].

MIMD – *Multiple* Ins*truction, Multiple Data*. Esta classe provê simultaneamente múltiplos fluxos de instrução aplicados a múltiplos fluxos de dados, e é a categoria mais geral de todas e o tipo mais comum de computador paralelo. Por isso, Johnson [22] propôs uma sub-classificação para estas máquinas, baseada em sua estrutura de memória (global ou distribuída) e no mecanismo utilizado para comunicação e sincronização (variáveis compartilhadas ou troca de mensagens), conforme a Figura 2.2.

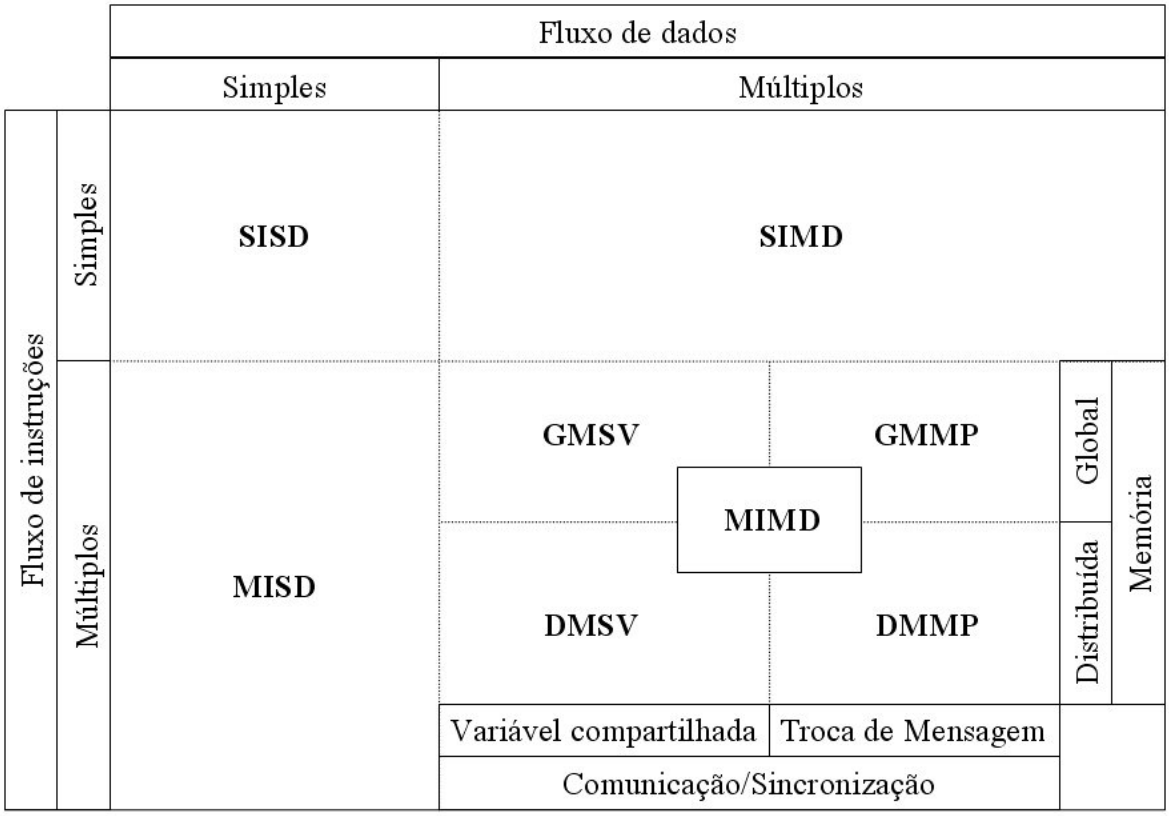

Figura 2.2 – Taxonomia de Flynn-Johnson para sistemas computacionais.

De acordo com a estrutura de memória e a comunicação/sincronização, a arquitetura MIMD de Flynn pode ser subdividida em quatro sub-classes: GMSV (*Global Memory Shared Variable*), DMSV (*Distributed Memory Shared Variable*), DMMP (*Distributed Memory Message Passing*) e GMMP (*Global Memory Message Passing*).

A classe GMSV é referenciada como sistemas multiprocessados com memória compartilhada, considerados sistemas fortemente acoplados. A classe DMMP é conhecida como sistemas multicomputadores com memória distribuída e fracamente acoplados.

A classe DMSV, que se tornou popular por combinar a implementação de uma memória distribuída com a facilidade de programação com variáveis compartilhadas, é conhecida como memória compartilhada distribuída. A categoria GMMP não é muito utilizada [40].

2.4 – Modelos físicos de máquinas

Fisicamente os sistemas computacionais podem ser classificados dentro de seis modelos implementáveis de máquinas [1]:

- Máquinas SIMD (S*ingle-Instruction, Multiple-Data*)
- Processador Vetorial Paralelo – PVP (*Parallel Vector Processor*)
- Multiprocessador Simétrico – SMP (*Symmetric Mutiprocessor*)
- Processador Maciçamente Paralelo – MPP (*Massively Parallel Processor*)
- Cluster de *workstations* – COW (C*luster of Workstations*)
- Multiprocessadores com Memória Distribuída Compartilhada – DSM (*Distributed Shared Memory*).

As máquinas SIMD são, na maioria dos casos, usadas para aplicações de propósito específico. Todos os outros modelos são máquinas MIMD.

Com exceção do PVP, que possui blocos customizados, a maioria das arquiteturas paralelas modernas é obtida por integração de elementos de prateleira (*commodity off-the-shelf)*.

Processadores vetoriais paralelos. A estrutura de um PVP típico pode ser visualizada na Figura 2.3. Estes sistemas contêm um número reduzido de processadores vetoriais customizados. Uma chave *crossbar* especificamente projetada e com grande largura de banda conecta os processadores aos módulos de memória compartilhada, o que provê alta velocidade no acesso aos dados. Estas máquinas normalmente não utilizam *caches*, mas um grande número de processadores vetoriais e um *buffer* de instruções. Um processador vetorial é uma máquina projetada para tratar operações aritméticas sobre elementos de matrizes ou vetores, de forma eficiente [23] [1].

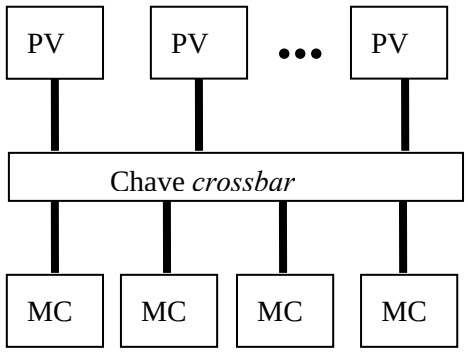

Figura 2.3 – Processadores vetoriais paralelos.

Multiprocessadores Simétricos. A arquitetura SMP está ilustrada na Figura 2.4. Diferente de um PVP, um SMP usa microprocessadores com memórias *cache on-chip* e *off-chip*. Estes processadores estão conectados a uma memória compartilhada através de um barramento de alta velocidade. Em alguns SMPs uma chave *crossbar* adicional é usada. Sistemas SMP são muito utilizados em aplicações comerciais, como bancos de dados, sistemas de transações *on-line* e *data warehouses*. Pelo fato de ser simétrico, cada processador tem igual acesso à memória compartilhada, aos dispositivos de E/S e aos serviços do sistema operacional. Também pelo fato de serem simétricos, um alto grau de paralelismo pode ser alcançado. Uma das desvantagens dos SMPs, é que por fazerem uso de memória compartilhada e de sistemas de interconexão baseados em chaves *crossbar* e barramentos, uma vez construídos eles acabam se tornando pouco escaláveis. Exemplos de SMPs incluem o IBM R-50, o SGI Power Challenge e o servidor DEC Alpha 8400 [1].

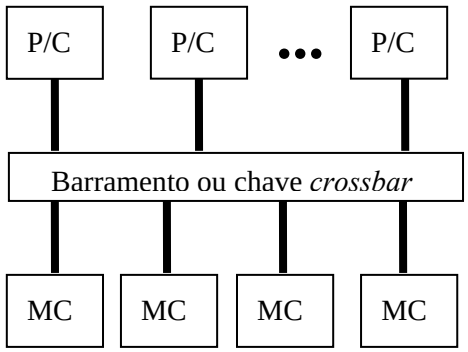

Figura 2.4 – Multiprocessador simétrico.

Processadores Maciçamente Paralelos. O termo MPP se refere a sistemas de computação de grande escala que utilizam microprocessadores *standard* (*commodity*) como nós de processamento, que usam memórias fisicamente distribuídas nos nós e que utilizam uma rede de interconexão de alta velocidade de comunicação e baixa latência e podem ser escalados para suportar centenas ou mesmo milhares de processadores (Figura 2.5). O programa consiste de múltiplos processos, cada um tendo o seu espaço de endereçamento privado. Os processos neste modelo, diferente dos PVPs e SMPs, se comunicam através de troca de mensagens [1].

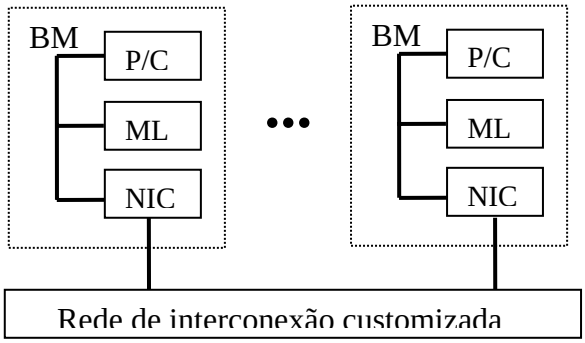

Figura 2.5 – Processador maciçamente paralelo.

Multiprocessadores com Memória Distribuída Compartilhada. As máquinas DSM estão modeladas na Figura 2.6, baseada na arquitetura Stanford DASH. O diretório de *cache* é usado para dar suporte a *caches* distribuídas coerentes [1]. A idéia do uso de diretório é manter para cada bloco de memória um registro das memórias *cache* que atualmente contém uma cópia daquele bloco e dos estados do bloco nestas memórias *cache*. Este registro é

conhecido como a entrada de diretório para aquele bloco, e normalmente se localiza no mesmo nó onde se encontra o bloco [18]. A principal diferença entre máquinas DSM e SMP é que a memória está fisicamente distribuída entre os nós do sistema. Contudo, os sistemas de *hardware* e *software* criam a ilusão de um único espaço de endereçamento para as aplicações. Uma máquina DSM também pode ser implementada utilizando extensões de *software* em uma rede de *workstations*. O objetivo de um sistema DSM é tornar a comunicação entre processos transparente para os usuários finais. Ao permitir que o programador compartilhe objetos da memória sem ter que tratar o seu gerenciamento, os sistemas DSM pretendem propor um balanço entre a facilidade de programação das máquinas com memória compartilhada e a eficiência e escalabilidade dos sistemas com memória distribuída [1].

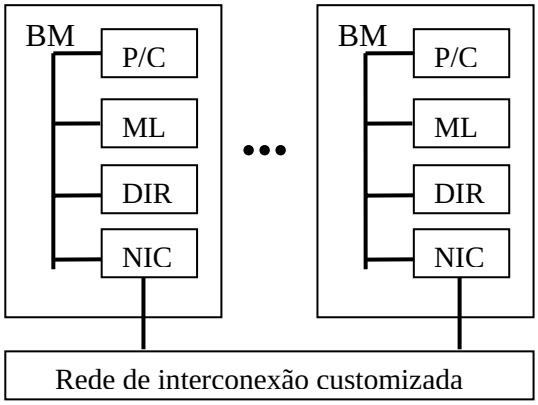

Figura 2.6 – Multiprocessadores com Memória Distribuída Compartilhada.

Os *clusters* de *workstations* (COW) também são um tipo de modelo físico, mas devido à sua relevância para este trabalho, eles serão detalhados na seção 2.6.

2.5 – Interação entre processos

Com o objetivo de cooperação, os processos executando concorrentemente em processadores distintos precisam se comunicar e sincronizar. A comunicação permite que a execução de um processo influencie na execução de outro processo. A comunicação entre processos é baseada no uso de variáveis compartilhadas (variáveis que podem ser referenciadas por mais de um processo) ou é baseada em troca de mensagens entre os processos [15].

No modelo de programação que utiliza memória compartilhada, as tarefas compartilham um espaço de endereçamento comum, no qual elas lêem e escrevem de modo assíncrono. Vários mecanismos, como *locks* e semáforos, podem ser usados para controlar o acesso à memória compartilhada. Uma vantagem deste modelo do ponto de vista do programador é que não há a noção de posse e, portanto, não há a necessidade de especificar explicitamente a comunicação de dados de produtores para consumidores. Este modelo pode simplificar o desenvolvimento do programa. Contudo, entender e gerenciar os conflitos de acessos às posições da memória compartilhada se torna mais difícil e também pode ser mais difícil escrever programas determinísticos [19].

A troca de mensagens é provavelmente o modelo de programação paralela mais utilizado atualmente. Programas que utilizam o paradigma de troca de mensagens criam múltiplas tarefas sendo que cada tarefa encapsula seus dados locais. Cada tarefa é identificada por um nome único e as tarefas interagem enviando mensagens para uma determinada tarefa e recebendo mensagens de uma determinada tarefa. O modelo de troca de mensagens não impede a criação de tarefas, a execução de múltiplas tarefas pelo processador ou a execução de diferentes programas por diferentes tarefas. Contudo, na prática, a maioria dos sistemas de troca de mensagens cria um número fixo de tarefas idênticas quando a execução do programa é iniciada e não permite que tarefas sejam criadas ou destruídas durante a execução do programa [19]. A troca de mensagens é baseada nas primitivas *send* e *receive*. Todos os processadores podem trocar informações com todos os outros processadores. Múltiplos *sends* e *receives* podem ocorrer simultaneamente em um computador paralelo. As mensagens são passadas entre os processadores através de um protocolo de comunicação [24].

2.5.1 – Troca de mensagem no MPI e PVM

O modelo funcional de um sistema que utiliza troca de mensagens é representado como um conjunto de processadores que têm sua memória local, mas cujos processos são capazes de se comunicar através do envio e recepção de mensagens [4].

Vantagens do modelo de troca de mensagem. A comunicação via troca de mensagem apresenta-se como uma das alternativas mais viáveis, devido a muitas vantagens, tais como [4]:

- **Generalidade (Universalidade):** Pode-se construir um mecanismo de passagem de mensagens para qualquer linguagem, como ferramentas de extensão das mesmas (bibliotecas) e sob qualquer tipo de rede (lenta ou rápida), para os mais diferentes tipos de máquinas, que vão desde os supercomputadores aos PCs mais simples.
- **Aplicabilidade:** A adequação a ambientes distribuídos faz deste modelo uma alternativa perfeitamente viável para uma grande gama de aplicações.
- **Expressividade:** O modelo de troca de mensagens tem sido utilizado como um modelo completo para o desenvolvimento dos mais diversos algoritmos para processamento paralelo.
- **Possibilidade de depuração:** Considerando-se as características de programação, é perfeitamente possível realizar a depuração no modelo de troca de mensagens, já que ela evita uma das causas mais comuns de erros: a corrida crítica e escrita indevida da memória, devido ao controle explícito do modelo.
- **Baixo custo:** As mais conhecidas bibliotecas de troca de mensagens (PVM (*Parallel Virtual Machine*) e MPI (*Message Passing Interface*)) são programas do gênero *freeware*, ou seja, de domínio público. Sua instalação requer apenas conhecimentos básicos e, portanto, permitem formar ambientes de processamento paralelo de baixíssimo custo.
- **Desempenho:** Finalmente, a mais importante razão permanece sendo a possibilidade de se obter um ganho de desempenho com a utilização de *clusters* de processamento intercomunicados via troca de mensagens.

Limitações do modelo de troca de mensagem. Algumas restrições apresentadas pelo modelo de troca de mensagens:

- **Necessidade de programação explícita:** Primeiro de tudo, o programador é diretamente responsável pela paralelização. Isto requer que haja uma mudança na forma algorítmica para desenvolvimento dos programas – de um modelo seqüencial, para uma visão cooperativa (distribuída) do processamento.

- **Custo de comunicação:** Os custos de comunicação podem tornar extremamente proibitiva a transmissão de mensagens em certos ambientes.

2.5.1.1 – Tipos de comunicação

Quanto ao bloqueio, os tipos de comunicação podem ser [2] [4]:

- **Comunicação "bloqueante" (*blocking communication*):** Quando a finalização da execução da rotina é dependente de determinados eventos, ou seja, existe uma espera por determinada ação antes de se permitir a continuação do processamento. Dois casos típicos são a espera pelo esvaziamento de um buffer antes de sua reutilização e a existência de barreiras de sincronização.
- **Comunicação "não-bloqueante" (*non-blocking communication*):** Neste caso, a finalização da chamada não espera qualquer evento que indique o fim ou sucesso da rotina. Rotinas de comunicação não bloqueantes não dependem de nenhum evento para que sejam finalizadas, isto é, a execução continua imediatamente após o início da comunicação.

Quanto à sincronização, a comunicação pode ser [2] [4]:

- **Comunicação síncrona:** É a comunicação na qual o processo que envia a mensagem, não retorna à execução normal enquanto não houver um sinal do recebimento da mensagem pelo destinatário, isto é, uma confirmação de êxito da rotina. Corresponde ao modelo mais seguro de transmissão de dados.
- **Comunicação assíncrona:** Quando não existe nenhuma restrição para a confirmação da rotina.

2.5.1.2 – MPI (*Message Passing Interface*) *– Interface para troca de mensagens*

A Interface de Troca de Mensagens, MPI, é uma padronização de um sistema de troca de mensagens proposta por um grupo de pesquisadores. Ela foi desenvolvida com o intuito de se tornar um padrão para a escrita de programas que utilizam troca de mensagem para se comunicar, tentando estabelecer um padrão prático, portável, eficiente e flexível para troca de

mensagens. Desde que foi finalizada, em junho de 1994, ela foi muito bem aceita e utilizada [6].

A principal vantagem da interface MPI, assim como da maioria dos padrões, é a portabilidade em máquinas distintas. Isto significa que o mesmo código-fonte pode ser executado em várias máquinas, desde que a biblioteca MPI esteja disponível. Outro tipo de compatibilidade oferecida pela MPI é a capacidade de executar de modo transparente em sistemas heterogêneos, ou seja, coleção de processadores que possuem arquiteturas distintas. Também provê comunicação eficiente e confiável [6].
MPI oferece uma coleção de rotinas de comunicação ponto-a-ponto e operações coletivas para movimentação de dados, computações globais e sincronização. Uma aplicação MPI pode ser visualizada como uma coleção de tarefas comunicantes e concorrentes [6].

As tarefas MPI podem executar no mesmo processador (Figura 2.7) ou em processadores diferentes concorrentemente. Para um programa aplicativo, enviar uma mensagem a uma tarefa na mesma máquina ou em outra é uma operação transparente [6].

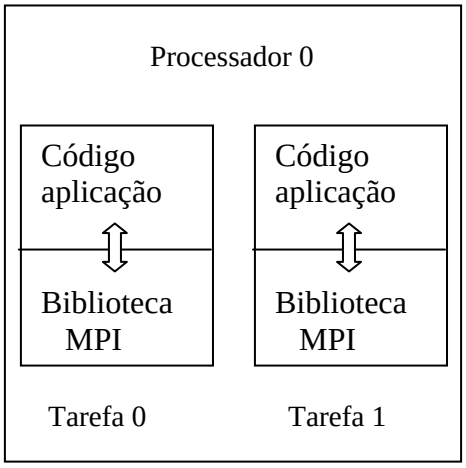

Figura 2.7 – Duas tarefas executando em um único processador.

Algumas características essenciais das implementações MPI mais utilizadas são apresentadas a seguir:

- Todo programa MPI deve incluir o arquivo *mpi.h* (ou *mpif.h* para uso com Fortran), que contém o protótipo de todas as funções, constantes e tipos MPI predefinidos que um programa pode necessitar.
- A primeira função MPI que um programa deve executar é o *MPI_Init*. Esta função permite que os procedimentos de inicialização sejam executados em uma máquina paralela particular ou em um *cluster* de *workstations*. A questão de como um programa MPI é iniciado não é endereçado pelo padrão MPI e pode variar de acordo com o sistema operacional e com a arquitetura.
- A última função MPI que um programa deve invocar é o *MPI_Finalize*. Ela limpa o estado paralelo antes de sair e nenhuma outra função MPI pode ser chamada depois dela.

O MPI, juntamente com o PVM, é um dos sistemas mais amplamente utilizados hoje em dia para programação paralela com troca de mensagens em clusters de computadores.

2.6 - *Cluster* de Computadores

O custo elevado e pouca flexibilidade das máquinas do tipo MPP e SMP, a grande oferta de computadores pessoais existentes, os recentes aumentos de velocidade na comunicação e na freqüência de *clock*, juntamente com a disponibilidade de *softwares* de domínio público de alto desempenho, incluindo sistemas operacionais, ferramentas de compilação e bibliotecas de troca de mensagens, fazem dos *clusters* uma alternativa economicamente viável para se obter alto desempenho [11] [10]. Por apresentar excelente custo-benefício e pela possibilidade de disponibilizar maiores e melhores recursos computacionais para os seus usuários, a cada dia mais organizações têm adotado este paradigma [11].

Os *clusters* computacionais objetivam a agregação de recursos computacionais para disponibilizá-los para a melhoria de aplicações [11].

Segundo Hwang [1], um *cluster* é uma coleção de computadores completos, ou nós, que estão fisicamente interconectados por uma rede de alto desempenho ou uma rede local, LAN. Tipicamente, cada nó é um servidor SMP, ou uma *workstation*, ou um PC. O aspecto mais importante é que todos os nós devem ser capazes de trabalhar coletivamente como um recurso computacional único integrado, além de permitir o uso individual de cada nó por usuários. Cinco conceitos arquiteturais devem estar presentes quando se fala em *cluster*: interconexão, nós, imagem única do sistema, disponibilidade e desempenho. Uma arquitetura conceitual de *cluster* é exibida na Figura 2.8.

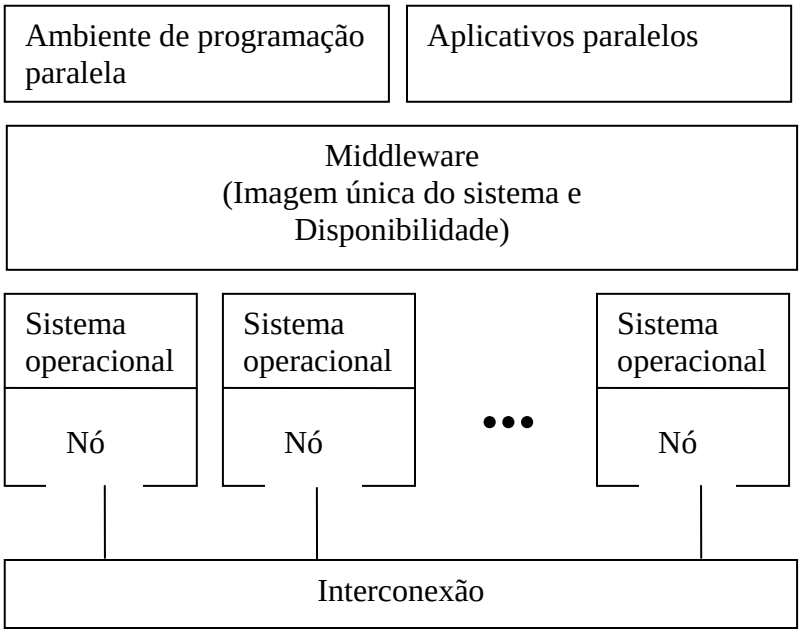

Figura 2.8 – Arquitetura ideal de um *cluster*.

Sterling [25] define um *cluster* como uma coleção de elementos computacionais fracamente acoplados, isto porque a memória está distribuída nos nós e um nó não tem acesso direto à memória de outro nó. Na configuração fortemente acoplada, os processadores compartilham uma mesma memória principal.

Gropp [26] define *cluster* como um computador paralelo construído com componentes de prateleira (*commodity components*) executando *software* de prateleira (*commodity software*). Um *cluster* é constituído de nós, contendo um ou mais processadores, memória, que é compartilhada pelos processadores dentro de um mesmo nó e dispositivos periféricos

adicionais (como discos), conectados por uma rede que permite a troca de informações entre os nós.

Para Sloan [27], uma configuração multicomputador, ou um *cluster*, é um grupo de computadores que trabalham em conjunto. Um *cluster* possui três elementos básicos: uma coleção de computadores individuais, uma rede de interconexão conectando esses computadores e um *software* que possibilita que um computador compartilhe trabalho com os demais computadores via a rede de interconexão.

Como mostra a Figura 2.9, cada nó de um *cluster* possui no mínimo um processador (P) e um módulo de memória *cache* (C) que se comunicam com uma memória local (ML), através de um barramento de memória (BM). Além disso, cada nó possui um disco local (DL) e um sistema operacional completo. Os programas paralelos trocam informações utilizando o paradigma de troca de mensagens [1]. Para garantir o potencial computacional que o *cluster* deve oferecer, os nós devem ser conectados através de uma interconexão de baixa latência. As redes de interconexão podem ser compartilhadas, como Ethernet, Fast Ethernet, Token Ring e FDDI ou as redes de interconexão podem ser dedicadas ponto-a-ponto, como por exemplo, a Myrinet [28], a Quadrics [29] e o Dolphin SCI [30]. [18] [11]

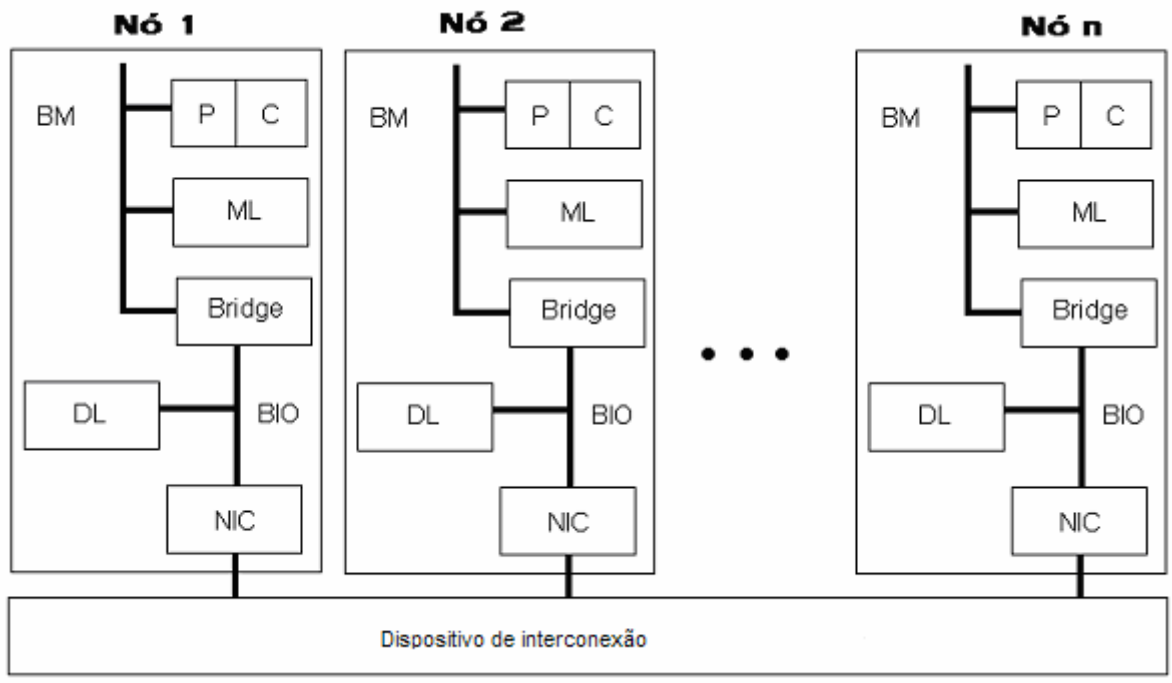

Figura 2.9 – Diagrama de blocos de um *cluster*.

O objetivo principal da utilização de *cluster* é obter uma configuração de *hardware* e *software* que permita conseguir um elevado desempenho a um baixo custo, durante a execução de aplicações paralelas. *Clusters* construídos a partir de elementos de prateleira (COTS) permitem flexibilidade de configuração. O número de nós, a capacidade de memória por nó, o número de processadores por nó e a topologia de interconexão são parâmetros da estrutura do sistema que podem ser especificados de forma detalhada. Ainda, a estrutura do sistema pode ser facilmente modificada ou acrescida de acordo com a necessidade sem perda do investimento já feito. Os *clusters* também permitem uma resposta rápida às melhorias da tecnologia. Assim que novos dispositivos, incluindo processadores, memória, discos e rede se tornam disponíveis, estes já podem ser integrados aos nós permitindo que o *cluster* possa usufruir dos avanços tecnológicos [25].

Outro motivo para se utilizar *cluster* é prover tolerância a falhas, ou seja, garantir que o poder computacional esteja sempre disponível. Pelo fato do *cluster* ser construído a partir de muitas cópias dos mesmos componentes ou de componentes similares, a falha de uma parte somente reduz o poder computacional do *cluster* [26].

A tolerância a falhas pode ser interpretada de várias maneiras. Para um servidor Web, o *cluster* pode ser considerado disponível (*up*) contanto que haja processadores suficientes e que a capacidade da rede possa suprir a demanda. Para aplicações científicas, a interpretação de disponibilidade (*uptime*) é diferente. Para *clusters* usados em aplicações científicas, particularmente aqueles usados para prover capacidade de memória adequada, a disponibilidade (*uptime*) é medida com relação ao tamanho mínimo do *cluster* (por exemplo, o número de nós) que permite que a aplicação execute [26].

Muito esforço foi aplicado anteriormente na construção de *cluster* de pequenas máquinas, tipicamente *workstations*, para se obter computadores paralelos de baixo custo. Mas, o projeto que verdadeiramente popularizou os *clusters* foi o projeto Beowulf do centro NASA Goddard Space Flight. Em 1994, Thomas Sterling, Donald Becker e outros construíram um *cluster* de PCs com 16 máquinas Intel 80486-based de 100MHz [26].

A classe Beowulf é uma categoria especial de *cluster*, construído a partir de PC’s e dispositivos facilmente encontrados no mercado (COTS). No cluster Beowulf os ambientes de

programação paralela, como MPI [4] e PVM [2] normalmente são utilizados para a construção de aplicações paralelas baseadas em troca de mensagens. Sistemas operacionais de distribuição livre, como Linux, normalmente são usados em arquiteturas Beowulf [25] [31].

2.6.1 – *Commodity cluster* para alto desempenho

A principal característica de um *commodity cluster* é seu baixo custo, sendo construído a partir de componentes facilmente encontrados no mercado (COTS). O *cluster* Beowulf é um *commodity cluster* [32]. O *commodity cluster* obtém vantagem de dois componentes de prateleira: CPUs rápidas projetadas principalmente para computadores pessoais e as redes de interconexão projetada para conectar computadores pessoais (LAN). Devido a estes componentes de prateleira, seu custo é relativamente baixo [26]. Gropp [26] afirma que, através de algumas medidas, incluindo velocidade de processamento, capacidade de memória, espaço disponível em disco e largura de banda, um único PC atual é mais poderoso do que os supercomputadores do passado.

Também referido como Linux *clusters* ou PC *clusters*, o Beowulf é talvez mais amplamente utilizado que qualquer outro tipo de computador paralelo devido ao seu baixo custo, flexibilidade e acessibilidade [33].

Os nós de um *cluster* podem incorporar um único processador ou múltiplos processadores numa configuração SMP [25]. Nos *commodity clusters* os nós são PCs ou pequenos SMPs de PCs interligados por uma rede de baixo custo e o sistema operacional utilizado é de código aberto, como Linux.

Vantagens da utilização de um *commodity* cluster:

- O uso de componentes de facilmente encontrados no mercado.
- Possibilidade de obter uma configuração de baixo custo, que atenda a necessidade da aplicação.
- Escalabilidade, oferecendo a possibilidade de acrescentar mais máquinas ao *cluster* caso haja a necessidade de mais poder de processamento.

O *commodity cluster* enquadra-se na taxonomia de Flynn [20] como MIMD (*Multiple Instruction Multiple Data*), do tipo multicomputador ou *hardware* fracamente acoplado, baseado no código monolítico Linux, utilizando a comunicação através de troca de mensagens. Este *cluster* é especificamente dedicado ao processamento paralelo, com uma rede de comunicações dedicada exclusivamente para a troca de mensagens.

Os *clusters* comerciais, diferentemente dos *commodity clusters*, utilizam computadores e *software* proprietários. Um exemplo é o SUN Ultra, que é um *cluster* proprietário. Em *clusters* proprietários, o *software* é fortemente integrado dentro do sistema. A principal desvantagem destes *clsuters* é o seu alto custo [27].

A lista do Top500 [35] representa duas classes de *clusters*: *cluster-NOW* e constelações. Ambas as classes são *clusters* não proprietários (*commodity clusters*) distinguidas pelo nível dominante de paralelismo. O primeiro nível é o número de nós conectados pela rede de comunicação global, na qual um nó contém todos os processadores do *cluster* e os recursos de memória. O segundo nível é o número de processadores em cada nó, normalmente configurado como um multiprocessador simétrico (SMP). Se um *cluster* não proprietário possui mais nós do que microprocessadores em cada um de seus nós, então o modo dominante de paralelismo é o primeiro nível (a categoria *cluster-NOW*). Se um nó tem mais microprocessadores do que os nós do *cluster*, o modo dominante de paralelismo é o segundo nível (uma constelação) [33]. Esta distinção não é arbitrária e pode levar a um sério impacto na programação. Um *cluster-NOW*, por exemplo, é programado quase que exclusivamente com interface de troca de mensagem (MPI), enquanto que uma constelação é programada, pelo menos em parte, com OpenMP usando um modelo de *treads*. Muito freqüentemente, uma constelação tem compartilhamento de espaço (*space-shared*) e não compartilhamento de tempo (*time-shared*), com cada usuário tendo seu próprio nó.

2.6.2 - Classificação de *Clusters*

Atualmente é comum efetuar a classificação dos tipos de *clusters* através de alguns de seus aspectos: limite geográfico, modo de utilização dos nós, tipo de *hardware* e conexão entre os nós, requerimento dos recursos solicitados pelas aplicações e tipo dos nós que constituem a configuração [11].

Limite geográfico. O nós do *cluster* podem estar distribuídos de forma compacta ou distribuída. Em um *cluster* compacto, os nós estão proximamente conectados, como por exemplo, dentro de uma sala e os nós não possuem periféricos. Em um *cluster* distribuído, os nós são estações completas, possuindo os periféricos e podem estar localizados em salas diferentes, prédios diferentes, mesmo em regiões geográficas diferentes. O empacotamento afeta diretamente a comunicação e a seleção da tecnologia de intercomunicação. Enquanto um *cluster* compacto pode utilizar uma rede de comunicação com alta largura de banda e baixa latência, que é sempre proprietária, os nós de um *cluster* distribuído normalmente são conectados através de LANs ou WANs padrões [1].

Utilização dos nós. A participação dos nós do *cluster* pode ser dedicada ou não-dedicada. Fatores como as políticas de gerenciamento, segurança, alta disponibilidade, escalonamento de processos, balanceamento de carga e o tipo de *middleware* do sistema de imagem única estão relacionados diretamente à forma de utilização dos nós [11].

Tipo de *hardware*. Os tipos de *hardware* empregados nos *clusters* têm sido classificados de uma forma empírica como NOWs (*Network of Workstations*), COWs (*Cluster of Workstations*) e Clumps (*Cluster of* SMPs). As NOWs são caracterizadas pela utilização de estações de trabalho distribuídas em uma rede local como elementos de *hardware* para compor os ambientes de *cluster*. Os COWs são geralmente constituídos de máquinas homogêneas e dedicadas à execução de aplicações específicas. Estas configurações dispõem de uma rede específica para conectar as máquinas. Os Clumps são *clusters* compostos de máquinas com arquitetura SMP [11].

Aplicações alvo. Existem dois alvos principais de requisitos para o uso de *clusters*. A primeira classe se refere às aplicações que necessitam de alto desempenho. Os *clusters*, por disponibilizar vários processadores, grande quantidade de memória e espaço em disco, se apresentam como um ambiente ideal para essas aplicações. A segunda classe são aplicações essenciais que não podem sofrer interrupções, sendo denominadas aplicações de alta disponibilidade [11].

Tipo dos nós. Esta classificação distingue os *clusters* entre homogêneos e heterogêneos, dependendo da similaridade de *hardware* e de *software* dos nós. Em um *cluster* homogêneo, todos os computadores possuem arquiteturas semelhantes como, por exemplo, PCs e SMPs e o mesmo sistema operacional. Os *clusters* heterogêneos possuem nós de arquiteturas distintas e/ou sistemas operacionais distintos [1] [11].

2.6.3 – Imagem Única do Sistema

Os sistemas de imagem única – SSI (*Single System Image*) – podem ser considerados *middlewares* que provêm uma abstração para os usuários de *clusters* quanto à sua configuração física e dos pacotes de *software* instalados no sistema [11].

A principal motivação da propriedade de unificação da imagem do sistema é fazer com que todos os recursos do sistema sejam vistos como um único recurso computacional. Esta propriedade pode ser definida como sendo uma ilusão criada por *hardware* ou por *software*, que apresenta uma coleção de recursos que atuam como se fossem um único componente [1].

A SSI é considerada uma camada *middleware,* que reside entre o sistema operacional e o ambiente do usuário, conforme pode ser visualizado na Figura 2.8. Além da subcamada de infra-estrutura de SSI, o *middleware* deve também conter a subcamada de infra-estrutura de disponibilidade. O sistema de disponibilidade disponibiliza serviços de *clusters* tais como *checkpoint*, recuperação e tolerância a falhas para todos os nós do *cluster*.

Alguns benefícios da SSI são [11]:

- Transparência na execução das aplicações;
- Transparência da localização dos dispositivos físicos.
- Redução na gerência do sistema;
- Alta disponibilidade pelo fato de o sistema ter a possibilidade de manter cópias de recursos em vários nós;
- Aumento da robustez da configuração pelo fato de que a SSI reduz a intervenção de operadores.

2.6.4 – *Cluster* Heterogêneo

Embora anteriormente os sistemas multicomputadores fossem, em sua maioria, homogêneos, atualmente vários destes sistemas são heterogêneos [36]. Pelo fato de os *clusters* permitirem flexibilidade de configuração, mais cedo ou mais tarde, um *cluster* homogêneo, quando for ampliado, adicionando-se máquinas extras ao *cluster* ou dispositivos mais atualizados, certamente se tornará heterogêneo, salvo alguns casos onde o *cluster* não mais será modificado e/ou se encontrarem máquinas exatamente iguais para se fazer a ampliação, algo quase impossível devido ao grande avanço tecnológico pelo qual a informática passa.

Em um sistema heterogêneo, os nós podem diferir na arquitetura, no *software*, na velocidade, etc [36].

A característica de um *cluster* ser ou não homogêneo tem implicações importantes no balanceamento de carga das aplicações. Para explorar de modo eficiente o poder computacional dos *clusters* de computadores heterogêneos, uma questão importante é como associar as tarefas aos computadores de modo que a carga fique balanceada e que o tempo total de computação seja minimizado [37].

No caso do particionamento de uma tarefa, ao se balancear a carga em um *cluster* homogêneo todos os computadores deverão receber cargas iguais, pois cada um deles possui as mesmas características e, conseqüentemente, a mesma capacidade de processamento.

Em um *cluster* heterogêneo é necessário ter uma preocupação maior com a distribuição da carga que cada nó deve receber. Deve ser feita uma análise levando-se em consideração as diferentes capacidades de processamento de cada uma das máquinas do ambiente. Após essa análise, cada nó deverá receber uma carga de trabalho compatível com a sua capacidade de processamento.

As redes de computadores heterogêneas são uma arquitetura de memória distribuída promissora. No caso mais geral, uma rede heterogênea inclui PCs, *workstations*, servidores multiprocessados, clusters de *workstations* e até mesmo supercomputadores. Diferente das plataformas homogêneas tradicionais, a arquitetura paralela heterogênea utiliza processadores

que executam em diferentes velocidades de processamento. Portanto, algoritmos paralelos tradicionais, com computações distribuídas uniformemente entre os processadores, não balancearão a carga entre os processadores com diferentes velocidades numa rede heterogênea. Os processadores mais rápidos executarão mais rapidamente suas partes e ficarão esperando por aqueles mais lentos nos pontos de sincronização [38].

Uma abordagem natural para o problema é distribuir os dados entre os processadores de maneira não uniforme para que cada processador execute o volume de computação proporcional à sua velocidade [38].

Obter um bom desempenho, contudo, pode ser um desafio. Além dos problemas encontrados no balanceamento de carga em sistemas homogêneos, precisa-se ainda levar em conta as velocidades diferentes dos processadores e suas arquiteturas e capacidades de memória diferentes, o que pode impactar significantemente no desempenho. Além disso, as cargas e disponibilidade das máquinas serão impossíveis de serem previstas, de forma que é difícil de saber com antecedência qual será a velocidade efetiva de cada máquina [39].

Os métodos de análise de desempenho de algoritmos paralelos homogêneos são bem estudados. Eles se baseiam em modelos e computadores paralelos, incluindo a *Parallel Random Access Machine* (PRAM), o modelo *Bulk-Synchronous Parallel* (BSP) e o *LogP model*. Todos os modelos assumem um computador paralelo como um multiprocessador homogêneo [38].

A análise teórica de um algoritmo paralelo homogêneo normalmente é acompanhada por um número relativamente pequeno de experimentos em um sistema paralelo homogêneo. O propósito destes experimentos é mostrar que a análise está correta e que o algoritmo analisado realmente é mais rápido do que suas contrapartes [38].

Uma análise teórica de desempenho para algoritmos paralelos heterogêneos é uma tarefa muito mais difícil do que a análise para algoritmos paralelos homogêneos. Enquanto algumas pesquisas nesta direção têm sido feitas, ainda não há nenhum modelo prático adequado para redes de computadores heterogêneas, o qual seria capaz de predizer o tempo de execução de algoritmos paralelos heterogêneos com uma precisão satisfatória. O problema de uma

distribuição de dados ótima numa rede heterogênea foi provado ser NP-completo até mesmo para uma álgebra linear simples como multiplicação de matrizes em redes heterogêneas [38]. Desta forma, algoritmos paralelos heterogêneos práticos são sub-ótimos.

Uma aproximação típica para a avaliação de um algoritmo paralelo heterogêneo é a sua comparação experimental com alguma contraparte homogênea em uma ou em várias plataformas heterogêneas. Os algoritmos heterogêneos distintos são também comparados principalmente experimentalmente. Devido à natureza complexa e irregular de redes heterogêneas, a avaliação experimental de algoritmos paralelos heterogêneos não é tão convincente como para as redes homogêneas. Pode-se facilmente discutir que a demonstração da vantagem de um algoritmo sobre outro em uma ou várias redes heterogêneas não prova que a situação não mudará se você executar os algoritmos em outras redes de computadores, com a diferença em relação à velocidade dos processadores e as diferentes estruturas e velocidade da rede de comunicação [38].

3 – DESEMPENHO EM COMPUTAÇÃO PARALELA

A meta de usar a computação paralela através de multicomputadores ou multiprocessadores é obter alto desempenho, pois o desempenho é uma questão central na computação paralela [18] [41]. Uma das principais motivações do processamento paralelo é resolver rapidamente problemas que exigem um processamento computacional elevado. Considerando o tempo de execução e o tamanho do problema, o que nós buscamos no processamento paralelo é velocidade, a qual é definida como trabalho dividido pelo tempo [42].

A avaliação de desempenho não é somente uma questão de medir o custo de uma computação específica, mas um processo de coletar informação sobre um conjunto de computações que informarão que decisão tomar para uma determinada meta. Uma das metas mais comuns é a velocidade [41].

Contudo, as tendências recentes no processamento paralelo sugerem que a questão de predição de desempenho está se tornando mais complexa e difícil. Os cientistas têm introduzido várias arquiteturas e algoritmos para fornecer escalabilidade de desempenho com a utilização de muitos processadores. Ao mesmo tempo, com várias arquiteturas e algoritmos disponíveis, a medição de desempenho está se tornando crítica na escolha de uma combinação apropriada de máquina-algoritmo para uma aplicação, principalmente quando a máquina tem uma arquitetura hierárquica sofisticada [42].

A análise de desempenho de aplicações paralelas é um desafio. Muitos fatores influenciam o desempenho e é difícil avaliar se e onde as aplicações experimentaram desempenho pobre [43].

O desempenho alcançado por uma aplicação paralela é o resultado de interações complexas entre os recursos de *hardware* e de *software* do sistema onde a aplicação está sendo executada. As características da aplicação, como a estrutura algorítmica, os parâmetros de entrada, o tamanho do problema, influenciam estas interações determinando como a aplicação explora os recursos disponíveis e os processadores alocados [43]. Assim como em sistemas monoprocessados, a maioria das questões sobre desempenho pode ser dirigida por técnicas algorítmicas e de programação, ou por técnicas arquiteturais ou por ambas [18].

Entender as técnicas de programação e as interdependências é importante não somente para projetistas de *software* paralelo, mas também para arquitetos. Além de nos ajudar a entender os programas paralelos, também nos ajuda a entender a relação entre o *hardware* e o *software*, isso é, em quais aspectos a arquitetura do sistema pode assistir e quais aspectos devem ser deixados para o *software*. As interdependências entre programa e sistema são mais complexas e têm maior impacto de desempenho em multiprocessadores do que em monoprocessadores, portanto, entende-se que são muito importantes para a nossa meta de desenvolver sistemas de alto desempenho que reduzam custo e esforço de programação [18].

Uma vez que um programa paralelo tenha sido escrito e que seus erros tenham sido eliminados, os programadores geralmente voltam a sua atenção para o desempenho do programa. A maioria dos programadores mede o desempenho de seus programas pelo tempo de *turnaround* e não por *throughput*. As ferramentas para medir o desempenho existem para ajudar os programadores a entender o motivo de seus programas não executarem rápido o suficiente [44].

3.1 – Métricas de desempenho

A operação fundamental da avaliação de desempenho de programa é medir o tempo requerido para executar a computação. Embora esta operação seja conceitualmente simples, o processo de medida está sujeito a muitos imprevistos. Para evitá-los os cuidados de que o programador precisa tomar são três [41]:

- Conhecer as ferramentas.
- Conhecer o programa.
- Conhecer o ambiente de execução.

As métricas de desempenho podem ser definidas como qualquer estatística sobre computação paralela projetada para ajudar os programadores a reduzirem o tempo de execução de seus programas.

As métricas de desempenho iniciaram na programação seqüencial como um simples *profile* do tempo de CPU que era consumido por cada *procedure* em um programa. Uma extensão

lógica desta técnica para programas paralelos é agregar os *profiles* do tempo de CPU de cada processo (ou *thread*) para prover um único *profile* para um programa paralelo [44].

O grande número de gargalos em potencial possíveis na programação paralela aumenta dramaticamente a necessidade de ferramentas de análise de desempenho se comparado às máquinas seqüenciais. As máquinas paralelas não só têm muitas instâncias dos recursos (CPU, registradores, sistemas de entrada e saída) que podem causar gargalos em programas seqüenciais, como também incluem características únicas como redes de interconexão e protocolos de coerência que podem contribuir para os problemas de desempenho [44].

No que se refere ao ambiente, uma questão que deve ser levada em conta são os efeitos arquiteturais. Muito mais do que em sistemas monoprocessados, o desempenho de máquinas paralelas depende de uma quantidade indefinida de questões arquiteturais envolvendo comunicação e compartilhamento de recursos. O programador que está desenvolvendo para uma máquina paralela deve ser cauteloso principalmente com as situações envolvendo quantidades diferentes de dados, computação ou comunicação [41].

Embora o processamento paralelo tenha se tornado uma abordagem comum para obter alto desempenho, as técnicas de avaliação de desempenho são fracas. Não existe nenhuma métrica bem estabelecida para medir o ganho de desempenho [45].

Simplesmente estender as métricas seqüenciais para os programas paralelos não é suficiente porque, em um programa paralelo, melhorar o procedimento que consome a maior quantidade de tempo pode não melhorar o tempo total de execução do programa. As dependências que existem entre os processos em um programa paralelo influenciam no momento de saber quais procedimentos são importantes para o tempo de execução do programa. As diferenças entre as métricas de desempenho de um programa paralelo podem ser notadas através do modo como elas respondem a estas dependências entre os processos [44].

O maior problema em ter muitas métricas é saber qual delas usar. Hollingsworth e Miller [44] compararam várias métricas distintas e concluíram que não havia uma métrica única que fosse a melhor. Na verdade, as orientações de várias métricas coincidiram. Contudo, eles foram capazes de caracterizar os tipos de programas onde cada métrica seria útil. Esta informação é

valiosa ao programador, e poderia ajudá-lo a selecionar as métricas apropriadas. Além disso, métricas diferentes têm custos de computação diferentes. Algumas vezes uma métrica mais barata (como uma prova) é suficiente e então não há a necessidade de calcular métricas complexas.

As métricas de desempenho provêm uma direção útil para se descobrir alguns tipos de gargalos; contudo, como métricas diferentes são requeridas para tipos diferentes de gargalos é deixada para o usuário a escolha de qual métrica utilizar [44].

A melhoria do desempenho de uma aplicação paralela pode ser visto como um processo iterativo consistindo de vários passos, envolvendo a identificação e localização de insuficiências, seus consertos e a verificação e validação do desempenho alcançado. O objetivo é definir as métricas de desempenho e um critério para explicar as propriedades e o comportamento de uma aplicação, identificando e localizando suas insuficiências de desempenho [31].

Durante o processo de medir o desempenho de uma aplicação, vários tipos de parâmetros podem ser medidos. Eles incluem parâmetros de medição de tempo, como, *wall clock times*, bem como parâmetros de contagem, como número de operações de entrada/saída, número de *bytes* lidos/escritos, número de acessos à memória, número de *bytes* enviados/recebidos. Estes parâmetros podem ser medidos em diferentes níveis de granulosidade, quer dizer, eles podem se referir à aplicação inteira, com suas atividades, por exemplo, computação, comunicação, acessos à memória, operações de E/S ou podem ser referir às suas regiões de código, por exemplo, laços de repetição, rotinas e declarações de código [43].

Um erro comum tanto na análise de programa serial como na análise de programa paralelo é medir o programa em um conjunto de dados e assumir que isso representa todos os conjuntos de dados. Para muitos programas, a velocidade pode depender fortemente dos dados de entrada. Neste caso, um programador pode desejar fazer execuções suficientes variando os dados de entrada para alcançar uma estimativa que seja estatisticamente útil do tempo de execução, ou pode tentar determinar o desempenho valendo-se do pior caso e do melhor caso das entradas [41].

Uma boa medida para o processamento paralelo é a velocidade média, a qual pode ser definida como sendo a velocidade alcançada de um dado sistema computacional dividida pelo número de processadores [42].

Idealmente, a velocidade média se mantém constante à medida que o tamanho do sistema cresce. O pico de desempenho do *hardware* especificado pelos vendedores normalmente está baseado nesta suposição ideal. Contudo, se o tamanho do problema é fixo, a situação ideal provavelmente não ocorre. O motivo é que para um problema que tenha o tamanho fixo, a razão comunicação/computação provavelmente cresce com o aumento do número de processadores, então, a velocidade média diminui com o aumento do tamanho do sistema. Por outro lado, se o tamanho do sistema é fixo, a razão comunicação/computação provavelmente decresce com o aumento do tamanho do problema para a maioria dos algoritmos práticos. Para estes algoritmos, aumentar o tamanho do problema proporcionalmente ao tamanho do sistema pode manter a velocidade média constante [42].

3.1.1 – *Speedup*

Suponha que tenha sido escolhido um programa paralelo com uma determinada carga de trabalho e que se queira usá-lo para avaliar uma máquina. Para uma máquina paralela, há duas coisas que é necessário medir: o desempenho absoluto e a melhoria de desempenho devido ao paralelismo. Esta última é tipicamente conhecida como *speedup*, que é definida como o desempenho absoluto obtido com a utilização de p processadores dividido pelo desempenho obtido com a utilização de um único processador. O desempenho absoluto (juntamente com o custo) é mais importante para o usuário final ou para a pessoa que vai adquirir a máquina. Contudo, ele sozinho não diz muito sobre o aumento de desempenho com a utilização do paralelismo e a efetividade de comunicação da arquitetura se comparado ao desempenho de um único processador. O *speedup* diz isto e é a métrica de desempenho mais comumente utilizada para o processamento paralelo [18] [45].

O desempenho absoluto será novamente abordado na Seção 3.3.

Tendo como medida de desempenho o tempo de execução, nós poderemos simplesmente executar o programa com uma mesma configuração de entrada em 1 e em *p* processadores e medir a melhora ou *speedup* como [18]

$$\frac{Tempo(1processador)}{Tempo(pprocessadores)} \qquad (3.1).$$

Já foi discutida a utilidade da melhoria de desempenho ou *speedup* para obter *insights* dentro da análise de desempenho de uma máquina paralela. A questão que ficou em aberto na medida de *speedup* para qualquer modelo escalar é o que o denominador na fração do *speedup* – desempenho em um processador – deveria realmente medir. Existem quatro escolhas [18]:

1. Desempenho do programa paralelo em um processador da máquina paralela;
2. Desempenho de uma implementação seqüencial do mesmo algoritmo em um processador da máquina paralela;
3. Desempenho da implementação do “melhor” algoritmo seqüencial em um processador da máquina paralela;
4. Desempenho do “melhor” programa seqüencial numa máquina padrão de acordo.

A diferença entre as opções 1 e 2 é que o programa paralelo incorre em *overheads* mesmo quando executado em um único processador, já que ele ainda executa sincronização, instruções de gerenciamento de paralelismo ou particionamento de código, ou testes para omitir estes. Estes *overheads* podem ser significantes algumas vezes. A diferença entre as opções 2 e 3 é que o melhor algoritmo seqüencial pode não ser possível ou pode não ser fácil de ser efetivamente paralelizado, assim, o algoritmo utilizado no programa paralelo pode ser diferente do melhor algoritmo seqüencial [18].

Utilizar desempenho como definido pela opção 3 claramente leva a uma métrica de *speedup* melhor e mais honesta do que os das opções 1 e 2. Do ponto de vista do arquiteto, contudo, em muitos casos pode-se usar a definição 2. A definição 4 resulta na métrica de comparação que é similar ao desempenho absoluto [18].

De acordo com as opções acima citadas, o *speedup* tem sido definido de forma distinta. Uma das definições foca em quão mais rápido um problema pode ser resolvido com a utilização de *p* processadores. Assim, ele compara o melhor algoritmo seqüencial com o algoritmo paralelo que está sendo considerado. Esta definição é referenciada como *speedup* absoluto. O *speedup* absoluto tem duas definições distintas, dependendo do fato de considerar ou não os recursos da máquina. No caso da independência da máquina, este *speedup* é definido como a razão do tempo decorrido do melhor algoritmo seqüencial em um processador sobre o tempo decorrido do algoritmo paralelo em *p* processadores. No caso da dependência da máquina, o *speedup* absoluto é definido como a razão do tempo decorrido do melhor algoritmo seqüencial na máquina seqüencial mais rápida sobre o tempo decorrido do algoritmo paralelo na máquina paralela [45].

Outro *speedup*, chamado *speedup* relativo, lida com o paralelismo inerente do algoritmo paralelo que está em consideração. Ele é definido como a razão do tempo decorrido do algoritmo paralelo em um processador sobre o tempo decorrido do algoritmo paralelo em *p* processadores. A razão de se utilizar o *speedup* relativo é que o desempenho de algoritmos paralelos varia de acordo com o número de processadores disponíveis. Comparar o algoritmo em si com números diferentes de processadores, dá a informação das variações e das degradações de paralelismo. O *speedup* absoluto e o *speedup* relativo são duas medidas de *speedup* freqüentemente utilizadas [45].

Entre todos os *speedups* definidos, o *speedup* relativo provavelmente seja o que tem tido a maior influência no processamento paralelo. Duas formulações bem conhecidas de *speedup* foram propostas baseadas no *speedup* relativo. Uma delas é a lei de Amdahl [46] e a outra é o *speedup* escalável de Gustafson [47]. Estas duas formulações de *speedup* usam um único parâmetro, a parte seqüencial do algoritmo paralelo. A lei de Amdahl fixa o tamanho do problema e está interessada em quão rápido pode ser o tempo de resposta. Ela sugere que o processamento maciçamente paralelo pode não alcançar um *speedup* elevado. Gustafson aborda o problema de outro ponto de vista. Ele fixa o tempo de resposta e está interessado em quão grande um problema pode ser para ser resolvido dentro deste tempo fixado. O argumento de Gustafson é que o tamanho do problema poderia aumentar até alcançar o poder computacional disponível para obter melhores resultados. Os resultados experimentais

mostram que o *speedup* poderia aumentar linearmente com o número de processadores disponíveis, baseado neste argumento [45].

Uma medida de custo, útil para as máquinas paralelas, é o número de processadores utilizados. Pode-se definir a medida de eficiência como [41]

$$E_p(n) = \frac{S_p(n)}{p} \qquad (3.2),$$

onde $S_p(n)$ é o *speedup* e p é o número de processadores.

A eficiência é o parâmetro de desempenho que mede a utilização média do processador, sendo definida como a razão do *speedup* sobre o tamanho do sistema. De acordo com a definição de diferentes *speedups*, tem-se diferentes definições para a eficiência, como a eficiência absoluta e a eficiência relativa [48].

O valor obtido com a expressão da eficiência (Equação 3.3) normalmente se encontra entre o intervalo [0 … 1] e é freqüentemente transformado em um valor percentual [0 … 100 por cento] [49].

3.1.1.1 – As leis de Amdahl e de Gustafson-Barsis

A Lei de Amdahl tem sido amplamente utilizada por projetistas e pesquisadores para alcançar uma estimativa inicial da melhoria de desempenho quando projetos e implementações variadas são experimentados [50].

A formulação original da lei de Amdahl define o impacto da porção inerentemente seqüencial de uma tarefa no *speedup* durante o multiprocessamento. Suponha que *f* represente a fração da tarefa que é inerentemente seqüencial, então, utilizando N processadores o *speedup* é dado por [46]

$$S = \frac{1}{(f + \frac{(1-f)}{N}} \qquad (3.3).$$

Quando **f = 0, S = N**, resultando em um *speedup* linear ideal.
Quando **f = 0.2, S < 5**, independente de **N**.
Quando **f = 0.5, S < 2**, independente de **N**.

Para **N** grande**, S = (1/f)**.

Este relacionamento gera um pessimismo em relação à viabilidade do processamento maciçamente paralelo especialmente se nós superestimamos o valor da fração *f*. Mas, os pesquisadores na comunidade de computação paralela começaram a suspeitar da utilidade e da validade da lei de Amdahl após observar *speedups* lineares impressionantes em algumas aplicações grandes. Gustafson obteve *speedups* quase lineares, em um hipercubo com 1024 processadores, para três aplicações práticas: análise de tensão de viga, simulação de onda de superfície, e fluxo de fluido instável. Isto o levou a suspeitar da natureza da formulação inicial de Amdahl e pareceu ter "quebrado" a lei de Amdahl e ter justificado o processamento maciçamente paralelo. Por exemplo, Gustafson discute que a lei de Amdahl é imprópria para as abordagens correntes de processamento maciçamente paralelo e sugere uma medida de *speedup* escalável para substituí-la. E. Barsis propôs uma fórmula de *speedup* escalável, que é freqüentemente referenciada como a lei de Gustafson. Ela é enunciada como: se a fração de tempo gasta pela parte seqüencial no sistema paralelo é **g**, então com **N** processadores o *speedup* escalável é

$$S = g + (1-g) * N \qquad (3.4),$$

um relacionamento linear simples [50] [51].

Em 1967, a lei de Amdahl foi usada como um argumento contra o processamento maciçamente paralelo. Já em 1988 a lei de Gustafson foi utilizada para justificar o processamento maciçamente paralelo. Yuan Shi [51] mostrou que a lei de Gustafson e a lei de Amdahl não são duas leis separadas e, de fato, provou a equivalência das duas leis. Yuan Shi

[51] concluiu que o uso do conceito de “porcentagem seqüencial” na avaliação de desempenho em ambiente paralelo é que conduz ao engano. Isto causou confusão na comunidade de processamento paralelo durante quase três décadas. Esta confusão desaparece quando os tempos de processamento são usados nas formulações. O que Yuan Shi sugeriu foi que as formulações baseadas em tempo seriam mais apropriadas para avaliação de desempenho paralelo.

Em 1967, a lei de Amdahl foi usada como um argumento contra o processamento maciçamente paralelo. Já em 1988 a lei de Gustafson foi utilizada para justificar o processamento maciçamente paralelo. Yuan Shi [51] mostrou que a lei de Gustafson e a lei de Amdahl não são duas leis separadas e, de fato, provou a equivalência das duas leis. Yuan Shi [51] concluiu que o uso do conceito de “porcentagem seqüencial” na avaliação de desempenho em ambiente paralelo é que conduz ao engano. Isto causou confusão na comunidade de processamento paralelo durante quase três décadas. Esta confusão desaparece quando os tempos de processamento são usados nas formulações. O que Yuan Shi sugeriu foi que as formulações baseadas em tempo seriam mais apropriadas para avaliação de desempenho paralelo.

Pela conclusão de Yuan Shi [51], Gustafson tinha, erroneamente, utilizado o valor de g como o valor de f na lei de Amdahl e suspeitou incorretamente da lei de Amdahl. As duas frações, f e g, são relacionadas em [51] como [50]

$$S = \frac{1}{\frac{(1 + (1 - g) * N}{g}} \qquad (3.5).$$

Por exemplo, Gustafson usou **g = 0.004** e calculou o *speedup* escalável como 1020 com **N = 1024**, mas utilizando-se a lei de Amdahl obteve um *speedup* de 201, usando o valor de g para f. Se ele tivesse utilizado o valor correto de f correspondente a g = 0.004, que é 0.0000039, então ele teria obtido o mesmo *speedup* de 1020 utilizando a lei de Amdahl. Assim, não há nada pessimista na lei de Amdahl. Na prática, para várias aplicações, a fração da parte serial é muito pequena, conduzindo à aproximação de *speedups* lineares [50].

Sabe-se que, dada uma arquitetura paralela e um problema de um tamanho fixo, o *speedup* de um algoritmo paralelo não continua a aumentar com o aumento do número de processadores, mas ele tende a saturar ou alcançar o cume com certo tamanho de sistema [48]. Desta forma, existem três relacionamentos possíveis entre um *speedup* e o número de processadores (P) [51]:

- ***Speedup* < *P***, ou *speedup* sub-linear;
- ***Speedup* = *P***, ou *speedup* linear;
- ***Speedup* > *P***, ou *speedup* super linear.

Como todo programa paralelo prático deve consolidar a(s) resposta(s) final (is) em um programa, a porcentagem seqüencial na lei de Amdahl nunca é zero, na prática. Assim, teoricamente *speedups* linear e super linear não são possíveis [51].

Na realidade, contudo, há dois fatores que podem ser utilizados para produzir *speedups* lineares ou super lineares [51]:

- O uso de uma execução seqüencial com recursos limitados como a base para o cálculo do *speedup*; e
- O uso de uma implementação paralela que pode deixar de executar uma grande quantidade de passos de cálculos e ainda assim produzir a mesma saída do algoritmo seqüencial correspondente [51].

Em [48] foi mencionado que *speedups* super lineares foram reportados por operações de busca, como algoritmos genéticos paralelos, etc. Obviamente, *speedups* super lineares podem também ser produzidos, especialmente em *workstations* modernas com memórias *caches* externas grandes, ou seja, se executar o programa em uma única *workstation*, o computador utiliza memória principal e talvez até mesmo memória *swap*; se executar em muitos computadores, a aplicação inteira executa na memória principal , talvez até mesmo execute integralmente na *cache* externa. Assim, poderá ter o *speedup* super linear [48].

3.1.2 – Escalabilidade

Segundo Hwang e Xu [1], um sistema computacional, incluindo todos recursos de hardware e software, é dito escalável se ele pode ter seus recursos melhorados/aumentados para absorver qualquer demanda de desempenho ou funcionalidade, ou reduzidos para diminuir custos.

Para [42], uma combinação algoritmo-máquina é escalável se a velocidade média alcançada por um algoritmo em uma dada máquina pode permanecer constante com o aumento do número de processadores, contanto que o tamanho do problema possa ser aumentado com o aumento do tamanho do sistema.

Como máquinas paralelas com mais e mais processadores têm se tornado disponível, a escalabilidade da métrica para desempenho está se tornando ainda mais importante. Escalabilidade mede como um algoritmo se comporta quando o tamanho do problema é aumentado linearmente com o número de processadores. Seja T(p, W) o tempo de execução para resolver um problema cujo trabalho é W (tamanho do problema) em p processadores. Na situação ideal, o número de processadores e a quantidade de trabalho podem ser aumentados N vezes, enquanto o tempo de execução permanece inalterado [42]:

$$T(Nxp, NxW) = T(p, W) \qquad (3.6).$$

A equação (3.7) é verdadeira se e somente se a velocidade média do sistema computacional for constante, onde a velocidade média é definida como o quociente da velocidade alcançada do dado sistema computacional e o número de processadores. A escalabilidade é formalmente definida como a capacidade de manter uma dada velocidade média.

Contudo, a escalabilidade não é sinônimo de tornar-se grande (*scale-up*). Envolve também a habilidade de reduzir (*scale down*) [1].

Dizer que um sistema é escalável envolve três pontos [1]:

- Funcionalidade e desempenho: O sistema escalável deve oferecer ou mais funcionalidade ou melhor desempenho. A potência total de processamento do sistema deve crescer proporcionalmente ao acréscimo de recursos.
- Escalabilidade do custo: O custo pago para escalar o sistema deve ser razoável. Regra prática: um escalonamento de n vezes não deve acarretar um custo superior a n ou *n log n* vezes.
- Compatibilidade: Os mesmos componentes incluindo *harware*, *software* de sistema e *software* de aplicação precisam continuar utilizáveis com pequenas alterações. As melhorias devem ser compatíveis com o resto do sistema: memórias, discos, interconexões, periféricos devem permanecer úteis.

3.1.2.1-Escalabilidade de recurso

A escalabilidade de recurso se refere ao ganho de desempenho ou funcionalidade obtido com o aumento do tamanho da máquina (número de processadores), com o aumento da capacidade de armazenamento (*cahe*, memória principal, discos), com a melhoria do software, etc [1].

Escalabilidade de tamanho da máquina significa fazer a máquina mais (ou menos) poderosa. Como o interesse aqui é o paralelismo, o tamanho da máquina é definido como o número de processadores [18]. A maneira mais imediata de escalonar um sistema é aumentar o tamanho da máquina (número de processadores). A escalabilidade de tamanho mede o número máximo de processadores que um sistema pode suportar [1].

Na escalabilidade de recursos pode-se manter o mesmo número de processadores e investir em mais memória, *caches* maiores, discos maiores, melhoria na arquitetura de comunicação ou no sistema de entrada/saída [1] [18].

Na escalabilidade de *software*, o *software* de um sistema pode ser melhorado: nova versão do SO, melhor compilador, bibliotecas mais eficientes, aplicativos mais eficientes, ambientes amigáveis, etc. [18]

3.1.2.2-Escalabilidade da aplicação

Para explorar completamente o potencial de computadores paralelos escaláveis, os programas aplicativos também devem ser escaláveis. Isto significa que o mesmo programa deve executar com um desempenho proporcionalmente melhor num sistema melhorado [1].

A escalabilidade do tamanho do problema indica a capacidade do sistema em suportar problemas maiores, tamanho de dados maiores, maior carga de trabalho [1].

3.1.2.3 - Escalabilidade tecnológica

A escalabilidade tecnológica se aplica a um sistema escalável que se adapta a mudanças tecnológicas. Pode ser dividido em três categorias: escalabilidade de geração, escalabilidade de espaço e heterogeneidade [1].

Na escalabilidade de geração (temporal), um sistema pode ser melhorado utilizando componentes de nova geração, tais como processador mais rápido, memória mais rápida, nova versão do sistema operacional, compilador com mais recursos, entre outros. Novamente, o poder computacional deveria aumentar quando o sistema migrasse para uma próxima geração. O restante do sistema deveria permanecer utilizável com o mínimo de modificações possíveis [1].

A escalabilidade de espaço se refere à habilidade de um sistema escalonar de múltiplos processadores em um gabinete, numa sala, num prédio, múltiplos prédios e regiões geográficas. Neste sentido, a Internet tem uma excelente escalabilidade de espaço [1].

A heterogeneidade é uma propriedade se refere à quão bem um sistema pode ser melhorado com a integração de componentes de *hardware* e *software* vindos de diferentes origens. Na área de *software* isto é chamado de portabilidade [1].

3.1.3 – Comparação entre escalabilidade e *speedup*

Existem as seguintes distinções e relacionamentos entre a escalabilidade e o *speedup* tradicional. Tanto a escalabilidade quanto o *speedup* dependem dos tamanhos iniciais do problema e do sistema e não são quantitativamente dimensionáveis, diferentemente das métricas de desempenho: tempo de execução, velocidade, etc. Quando o tamanho do sistema aumenta, o *speedup* provê a razão da variação da métrica de desempenho escolhida e a escalabilidade indica a capacidade do sistema para manter a métrica de desempenho escolhida. *Speedups* tradicionais requerem que o tamanho do problema seja fixo, mas a escalabilidade exige que o tamanho do problema seja escalável. *Speedup* mede o ganho de desempenho do processamento paralelo *versus* o processamento serial, enquanto a escalabilidade mede a degradação de desempenho de sistemas paralelos maiores *versus* sistemas paralelos menores [48].

3.2 – Particionando para obter desempenho

As etapas de particionamento – decomposição e atribuição – são em grande parte independente da arquitetura subjacente ou da abstração da comunicação e tem maior preocupação com questões de algoritmo que dependem somente de propriedades inerentes ao problema. Em particular, estas etapas vêm o multiprocessador simplesmente como um conjunto de processadores que comunicam um com o outro. Para estas etapas a máquina pode ser visualizada simplesmente como um conjunto de processadores cooperantes, ignorando seu modelo de programação e sua organização. Tudo o que se sabe neste estágio é que a comunicação entre os processadores é cara em termos de custo de processamento. As três questões algorítmicas que precisam ser consideradas para o particionamento são [18]:

- balancear a carga de trabalho e reduzir o tempo gasto na espera por sincronização de eventos,
- reduzir comunicação e
- reduzir o trabalho extra feito para determinar e gerenciar uma boa atribuição de tarefas a processadores.

Infelizmente, estas três metas algorítmicas primárias estão em contrariedade umas com as outras. Uma meta singular de minimizar a comunicação seria satisfeita executando o

programa num único processador, mas isso resultaria no desbalanceamento final da carga. Por outro lado, a aproximação de um balanceamento de carga perfeito poderia ser obtida a um grande custo de comunicação. Para diminuir a penalidade de gerenciamento de tarefas, pode-se fazer de cada operação primitiva no programa uma tarefa e associar as tarefas aleatoriamente aos processadores. Em muitas aplicações complexas, o balanceamento de carga e a comunicação podem ser melhorados se for gasto mais tempo determinando uma boa atribuição de tarefas a processadores. A meta de uma decomposição e atribuição é obter um compromisso entre estas demandas conflitantes [18].

3.2.1 – Balanceamento de carga e tempo de espera para sincronização

Em sua forma mais simples, balancear a carga de trabalho significa assegurar que cada processador faça a mesma quantidade de trabalho. Isso implica combinar concorrência suficiente com atribuição apropriada e redução de serialização, resultando no seguinte limite simples de *speedup* potencial [18]:

$$Speedup_{problema}(p) \leq \frac{TrabalhoSequencial}{\max(TrabalhoEmQualquer\,\mathrm{Pr}\,ocessador)} \qquad (3.7).$$

Trabalho, neste contexto, deveria ser interpretado liberalmente, porque o que importa não é somente quantas operações de cálculo são feitas, mas o tempo gasto para executá-las, o que envolve acesso a dados e comunicação [18].

De fato, balanceamento de carga é um pouco mais complicado do que simplesmente igualar o trabalho. Não significa somente os processadores distintos executarem a mesma quantidade de trabalho, mas estes devem estar trabalhando ao mesmo tempo. O ponto extremo seria se o trabalho fosse dividido igualmente entre os processos, mas somente um processo estivesse ativo por vez, assim, não haveria nenhum *speedup*. A meta real do balanceamento de carga é minimizar o tempo que os processos gastam esperando por pontos de sincronização, incluindo os implícitos no fim do programa. Isto também envolve minimizar a serialização de processos devido à exclusão mútua (espera para entrar em seções críticas) ou dependências [18].

O balanceamento de carga ótimo é computacionalmente caro e iria requerer um conhecimento prévio das características da carga de trabalho do sistema, durante o tempo de execução. Alternativamente, é desejável ter um algoritmo sub-ótimo que exija menos informação sobre a carga de trabalho, que possa lidar com a imperfeição do conhecimento sobre o estado do sistema e que não seja tão caro para ser utilizado [52].

O balanceamento de carga pode ser um mecanismo centralizado ou descentralizado. No mecanismo centralizado, a decisão de alocar processos aos *hosts* diferentes na rede é feita por um controlador central onde o estado do sistema é mantido [52].

No mecanismo descentralizado, os processos são escalonados nos *hosts* de chegada. Esta abordagem é mais rápida na tomada de decisões, contudo ela requer maior custo de comunicação para atualizar todos os *hosts* com o estado do sistema [52].

A política de balanceamento de carga pode ser estática ou dinâmica. Nas políticas de balanceamento de carga estáticas, a alocação de processos aos *hosts* é feita com base em um comportamento médio predeterminado da rede e não no estado corrente do sistema. Alternativamente, nas políticas dinâmicas, a alocação de processos aos *hosts* é feita baseada na estimativa atual do estado do sistema. Estas políticas associariam o processo na criação ou na chegada de um *host* externo para o *host* que pareça ser o melhor naquele momento [52].

Existem quatro etapas para balancear a carga de trabalho e reduzir o tempo de espera por sincronização [18]:

- Identificar concorrência suficiente durante a decomposição, e superar a Lei de Amdahl.
- Decidir como gerenciar a concorrência (estaticamente ou dinamicamente).
- Determinar a granulosidade na qual se possa explorar a concorrência.
- Reduzir os custos de serialização e de sincronização.

3.2.2 – Redução da comunicação inerente

O balanceamento de carga é por si só conceitualmente fácil desde que a aplicação tenha recursos para prover concorrência suficiente. Talvez o mais importante *tradeoff* com o

balanceamento de carga seja reduzir a comunicação entre os processadores. Decompor um problema em múltiplas tarefas normalmente implica em ter-se comunicação entre as tarefas. Se estas tarefas estão associadas a diferentes processos, podemos incorrer em comunicação entre os processos e, portanto, comunicação entre processadores. Nós focamos aqui na redução da comunicação que é inerente ao programa paralelo – isto é, um processo produz um dado de que outro necessita – enquanto ainda seja preservado o balanceamento de carga, retendo nossa visão da máquina como um conjunto de processadores cooperantes.

O impacto da comunicação é mais bem estimado não pela quantidade de comunicação, mas por uma quantidade chamada de razão entre comunicação e computação. Isto é definido como a quantidade de comunicação (que pode ser em bytes) dividida pelo tempo de computação, ou pelo número de instruções executadas. Por exemplo, um gigabyte de comunicação tem um impacto muito maior no tempo de execução e na ocupação da banda de comunicação de uma aplicação se o tempo requerido para que a aplicação execute é de 1 segundo do que o impacto se o tempo requerido para que a aplicação execute é de 1 hora. A razão entre a comunicação e a computação pode ser calculada como um número por processo, ou pode ser acumulado por todos os processos [18].

A comunicação inerente à razão da computação é primariamente controlada pela atribuição de tarefas a processos. Para reduzir a comunicação, nós deveríamos tentar assegurar que as tarefas que acessem os mesmos dados ou que necessitem se comunicar muito sejam atribuídas ao mesmo processo. Por exemplo, numa aplicação de banco de dados, a comunicação seria reduzida se as consultas e as atualizações que acessem os mesmos registros do banco de dados fossem atribuídas ao mesmo processo.

Em adição à redução do volume de comunicação, é também importante manter a comunicação balanceada entre os processadores, não somente a computação. Já que a comunicação tem um alto custo, desbalanceamentos na comunicação podem-se traduzir diretamente em desbalanceamentos no tempo de execução dos processadores [18].

Incluir a comunicação como um custo explícito de desempenho refina nosso limite básico de *speedup* para:

$$Speedup_{problema}(p) \leq \frac{TrabalhoSeqüencial}{\max(Trabalho + TempoEsperaSinc. + CustoComunic.)} \quad (3.8).$$

3.2.3 – Redução do Trabalho Extra

A discussão da decomposição de domínio mostra que quando uma computação é irregular, computar uma atribuição eficiente que forneça um balanceamento de carga e que reduza a comunicação pode ser bastante cara. Este trabalho extra não é requerido na execução seqüencial e é um *overhead* [18].

Outro exemplo de trabalho extra é computar valores de dados redundantemente ao invés de ter um processo que os compute e comunique aos demais, o que vem a ser um *tradeoff* favorável quando o custo de comunicação é alto [18].

Finalmente, muitos aspectos existentes em programas paralelos – tais como criar processos, gerenciar tarefas dinâmicas, distribuir código e dados ao longo da máquina, executar operações de sincronização e instruções de controle de paralelismo, estruturar a comunicação apropriadamente para a máquina, empacotar e desempacotar dados para e das mensagens de comunicação – envolvem trabalho extra [18].

Torna-se necessário considerar cuidadosamente os *tradeoffs* entre o trabalho extra, o balanceamento de carga e a comunicação quando vamos tomar nossas decisões de particionamento. A arquitetura pode nos ajudar a reduzir o trabalho extra ao fazer de modo mais eficiente a comunicação e o gerenciamento das tarefas e, portanto, reduzir a necessidade por trabalho extra. Baseado somente nessas questões algorítmicas, o limite de *speedup* pode ser refinado para [18]:

$$Speedup_{problema}(p) \leq \frac{TrabalhoSeqüencial}{\max(Trabalho + TempoEsperaSinc. + CustoComunic. + Trab.Extra)} \quad (3.9).$$

3.3 – Desempenho absoluto

Desempenho absoluto é mais bem medido como trabalho executado por unidade de tempo.

Para um usuário de um sistema o desempenho absoluto é a medida mais importante. Suponha que o tempo de execução seja a nossa métrica de desempenho absoluto. O tempo pode ser medido de diferentes modos. Primeiro, existe uma escolha entre o tempo do usuário (*user time*) e o *wall-clock time* para uma carga de trabalho. O *user time* é o tempo que a máquina gasta executando o código de uma carga de trabalho particular ou um programa, excluindo, assim, atividade do sistema e outros programas que podem também compartilhar do tempo de processamento, enquanto o *wall-clock time* é o tempo total decorrido para a execução da carga de trabalho – incluindo todas as atividades que possam intervir – medido por um relógio que se encontra na parede. Segundo, há a questão de se deve ser utilizado o tempo de execução médio ou o máximo de todos os processos do programa [18].

Como, finalmente, o que importa aos usuários é o *wall-clock time*, nós devemos medir e apresentar este tempo na comparação de sistemas. Contudo, se outros programas de usuários – não somente o sistema operacional – interferir na execução do programa em decorrência da multiprogramação, então o *wall-clock time* não vai nos ajudar a entender os gargalos de desempenho naquele programa particular em que estamos interessados. Note que o *user time* para aquele programa pode não ser também muito útil neste caso, uma vez que a execução intercalada com processos sem conexão rompe com as interações do programa, assim como sua sincronização e balanceamento de carga, no sistema de memória. Nós deveríamos, então, apresentar o *wall-clock time* e descrever o ambiente de execução (*batch* ou multiprogramado), se ou não nós apresentamos informações mais detalhadas para realçar a compreensão [18].

Similarmente, como um programa paralelo não termina até que o último processo tenha terminado, é o tempo deste ponto que é importante, e não o tempo médio dos processos. Claro, se nós realmente queremos entender os gargalos de desempenho nós precisaríamos verificar os perfis de execução de todos os processos – ou como um exemplo – quebrar em componentes de tempo distintos.

Em resumo, do ponto de vista do usuário na comparação de máquinas a medida de desempenho de maior interesse é o tempo de execução *wall-clock* (havendo, claro, questões de custos a serem consideradas). Contudo, do ponto de vista de um arquiteto, para um programador entender o desempenho de um programa ou para um usuário mais interessado no desempenho de uma máquina, é melhor utilizar o tempo de execução e o *speedup* [18].

4 – A APLICAÇÃO *BENCHMARK*

A aplicação utilizada como *benchmark* neste trabalho para analisar o desempenho do ambiente heterogêneo é um código paralelo para uma solução do método dos elementos finitos (MEF) aplicados à elasticidade linear, onde o sistema de equações algébricas é resolvido por uma versão paralela do método dos gradientes conjugados (MGC) [8].

4.1 – Método dos Elementos Finitos (MEF)

4.1.1 – Introdução

A Engenharia Mecânica utiliza-se de uma enorme gama de materiais para as várias construções e experiências profissionais. Dentre esses materiais, as placas, metálicas ou não, possuem grande utilização, desde desenvolvimentos mais resistentes como caçambas, caldeiras, autoclaves até peças sensíveis como próteses humanas, placas para controle, entre outros. Para o desenvolvimento de tais peças é necessária uma grande quantidade de testes preliminares com as placas utilizadas, prevendo o desgaste e possíveis falhas na fabricação e no decorrer do uso. Tais testes demandam grande custo financeiro e na maioria das vezes um tempo vasto nas experiências.

O Método dos Elementos Finitos (MEF) foi desenvolvido como um meio de baratear os custos e diminuir o tempo de teste. No âmbito da Engenharia de Estruturas, o Método dos Elementos Finitos tem como objetivo a determinação do estado de tensão e de deformação de um sólido de geometria arbitrária sujeito a ações exteriores [54].

O método dos elementos finitos se tornou uma das mais poderosas ferramentas de análise para solução de problemas de engenharia. Um número muito grande de autores tem tratado das formulações, aplicações e implementações computacionais desta técnica nos últimos 50 anos. Na análise estrutural, área onde mais tem se desenvolvido, a base do método de elementos finitos consiste em transformar o sólido contínuo em uma associação de elementos discretos e escrever equações de equilíbrio e compatibilidade entre eles, admitindo funções contínuas capazes de representar o campo de deslocamentos no domínio do elemento. A partir de então,

obtém-se o estado de deformações e, através de relações constitutivas chega-se ao estado de tensões nos elementos [55] [56].

Pelo método dos elementos finitos um modelo matemático descrito por equações diferenciais em um domínio contínuo, é convertido em um modelo discreto de pequenos elementos, com um número finito de graus de liberdade (GLs), pelo uso de algum método variacional [57] [58].

O advento da computação digital motivou a aplicação dos métodos numéricos na solução dos problemas de engenharia. Através de métodos como elementos finitos, equações algébricas são obtidas para estimar, em um número finitos de pontos, a solução analítica. Esta estimativa pode ser aprimorada com o aumento do número de equações [8].

A formulação matemática do método dos elementos finitos pode ser obtida através de métodos variacionais ou de métodos de resíduos ponderados. Todas estas técnicas permitem transformar o conjunto de equações diferenciais parciais em um sistema de equações algébricas. A abordagem usada na aplicação é fundamentada em um método variacional, o Princípio dos Trabalhos Virtuais e está baseada na forma adotada por Bathe [59].

4.1.2 – Mecânica computacional (Mecânica do contínuo)

A área da mecânica computacional resolve problemas específicos por simulação através de métodos numéricos implementados em computadores digitais [60].

Dentre os ramos da mecânica computacional que podem ser distinguidos de acordo com a escala física do foco de atenção, nossa atenção será voltada para a mecânica do contínuo.

Grande parte dos problemas de engenharia pode ser formulada através dos princípios gerais da mecânica do contínuo. Este ramo da mecânica trata a matéria como sendo um meio contínuo, sem vazios interiores, desconsiderando sua estrutura molecular. Na mecânica do contínuo, os princípios da física são escritos sob a forma de equações diferenciais. Os efeitos da constituição interna molecular dos materiais são levados em conta de forma macroscópica através das equações constitutivas do material.

A mecânica do contínuo estuda os corpos ao nível macroscópico, utilizando modelos contínuos nos quais a micro-estrutura é homogeneizada por médias calculadas do fenômeno. As duas áreas tradicionais de aplicação da mecânica do contínuo são a mecânica dos sólidos e a dos fluidos. A primeira inclui as estruturas que, por razões óbvias, são fabricadas com sólidos. A mecânica computacional dos sólidos tem uma abordagem científica aplicada, considerando que a mecânica computacional estrutural enfatiza aplicações tecnológicas para a análise e modelagem de estruturas. A mecânica computacional dos fluidos lida com problemas que envolvem o equilíbrio e o movimento de líquidos e gases [60].

O Método dos Elementos Finitos (MEF) é seguramente o processo que mais tem sido usado para a discretização de meios contínuos.

A discretização de sistemas contínuos tem como objetivo particionar o domínio - o sistema - em componentes cujas soluções são mais simples e, depois, é necessário unir as soluções parciais para obter a solução do problema.

4.1.3 – Tipo de análise

Quando surge a necessidade de resolver um problema de análise de uma estrutura, a primeira questão que se coloca é a sua classificação quanto à geometria, modelo do material constituinte e ações aplicadas. O modo como o MEF é formulado e aplicado depende, em parte, das simplificações inerentes a cada tipo de problema [54].

Análise dinâmica e análise estática. As ações sobre as estruturas são em geral dinâmicas, devendo ser consideradas as forças de inércia associadas às acelerações a que cada um dos seus componentes fica sujeito. Por este motivo, seria de esperar que a análise de uma estrutura teria, obrigatoriamente, que considerar os efeitos dinâmicos. Contudo, em muitas situações é razoável considerar que as ações são aplicadas de um modo suficientemente lento, tornando desprezáveis as forças de inércia. Nestes casos a análise se caracteriza como estática [54]. Somente serão considerados os problemas que se encaixem na análise estática.

Análise linear e análise não linear. Os problemas estáticos podem ser classificados em lineares e não lineares. A análise estática linear lida com problemas estáticos nos quais a resposta é linear no sentido de causa e efeito, ou seja, ao nível do material que constitui a estrutura, a relação entre tensões e deformações é linear [60] [54]. Por exemplo, se as forças aplicadas são dobradas, os deslocamentos internos também dobram [60]. Materiais metálicos, quando sujeitos a tensões suficientemente pequenas, exibem comportamento elástico linear e apresentam valores de deformações e deslocamentos diretamente proporcionais às forças aplicadas. A forma geral de uma equação constitutiva para materiais elásticos lineares é

$$\text{Tensão} = (\text{uma constante}) \times \text{deformação} \qquad (4.1)$$

As constantes de proporcionalidade são determinadas experimentalmente.

Os problemas fora deste domínio são classificados como não linear [60]. Somente serão considerados os problemas que se encaixem na análise linear.

4.1.4 – Detalhamento do Método dos Elementos Finitos

O conceito de MEF será parcialmente ilustrado através de um exemplo bem antigo: encontrar o perímetro L de um círculo de diâmetro d. Como $L = \pi.d$, isto é equivalente a obter um valor numérico para π [60].

A Figura 4.1 (a) mostra o desenho de um círculo de raio r e diâmetro $d = 2r$. A Figura 4.1 (b) mostra a inscrição de um polígono regular de n lados, onde $n = 8$. Renomeando os lados do polígono como *elementos* e os vértices como *nós*, numerando os nós com inteiros de 1, ..., 8, um elemento típico, por exemplo o segmento que une os vértices 4 e 5 pode ser extraído, conforme ilustra a Figura 4.1 (c). Uma instância do elemento genérico i-j está ilustrado na Figura 4.1 (d). O comprimento do elemento é $L_{ij} = 2r\,sen(\pi/n)$. Como todos os elementos têm o mesmo comprimento, o perímetro do polígono é dado por $L_n = nL_{ij}$, de onde a aproximação para π é $\pi_n = L_n / d = n\,\text{sen}(\pi/n)$ [60].

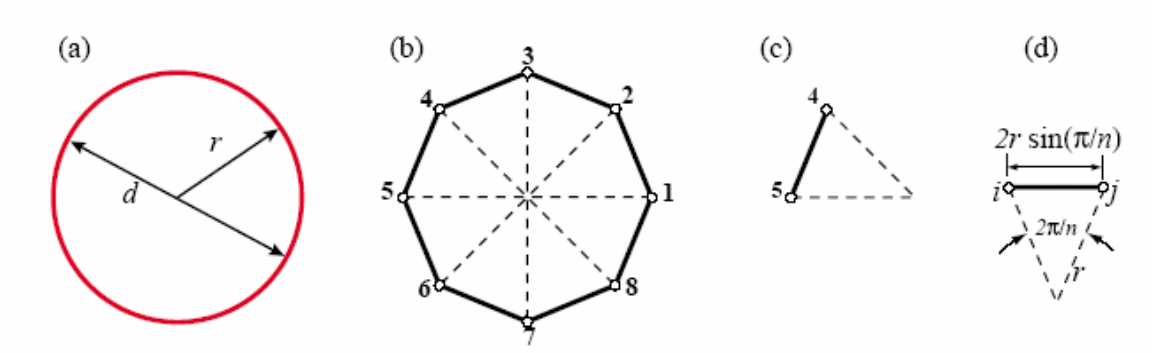

Figura 4.1 – O problema de "encontrar o π", tratado com os conceitos do MEF: objeto contínuo, (b) uma aproximação discreta por polígonos regulares, (c) elemento desconectado, (d) elemento genérico.

Algumas idéias chaves por trás do MEF podem ser identificadas neste exemplo. O círculo, visto como um *objeto matemático fonte*, é substituído por polígonos. Estes são aproximações discretas para o círculo. Os lados, aqui chamados de *elementos*, são especificados por seus *nós*. Os elementos podem ser separados desconectando nós, um processo chamado de separação (desmontagem) no MEF. A separação (desmontagem) de um elemento genérico pode ser definida, independentemente do círculo original, pelo segmento que conecta dois nós i e j. A propriedade relevante do elemento: o comprimento do lado $L_{i\,j}$, pode ser computado no elemento genérico independentemente dos outros, uma propriedade chamada de *suporte local* no MEF. A propriedade alvo: o perímetro do polígono é obtido reconectando-se n elementos e adicionando seus comprimentos; os passos correspondentes à *montagem* e *solução* no MEF, respectivamente. A mesma técnica pode ser utilizada para o plano [60].

O exemplo não ilustra o conceito de graus de liberdade, quantidades conjugadas e coordenadas local-global [60].

Pode-se discutir o fato de que um círculo é um objeto mais simples do que um polígono de 128 lados, por exemplo. Apesar destas falhas o exemplo é útil em um respeito: mostrar uma escolha na substituição de um objeto matemático por outro. Esta é a raiz do processo de análise do MEF [60].

4.2 – Solução numérica para sistemas de equações lineares

O processo de discretização pelo método dos elementos finitos leva ao estabelecimento e à solução de um conjunto de equações algébricas da forma:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{4.2}$$

que aparece na análise de estruturas. Para problemas lineares, *A* é matriz de rigidez - uma matriz quadrada, positiva definida e simétrica, *x* é o vetor de deslocamentos nodais e *b* é o vetor de forças aplicadas aos nós.

Para resolver esse sistema de equações, duas classes de algoritmos podem ser identificadas: os métodos diretos e os métodos iterativos. Neste capítulo são descritos estes dois métodos e o método dos gradientes conjugados (MGC) é descrito mais detalhadamente por se tratar do método utilizado neste trabalho para solucionar o sistema de equações e também por ser o método mais popular para resolver sistemas de equações lineares como os descritos na Equação (4.2) [61] [62].

4.2.1 – Métodos diretos

Quando o processo de solução de um sistema de equações lineares envolve a decomposição da matriz dos coeficientes, obtendo-se a solução sem o uso de iterações, este processo é conhecido como método direto. A solução do sistema é alcançada em um número conhecido de operações, ao contrário dos métodos iterativos. A principal desvantagem desses métodos, quando aplicados a matrizes esparsas e de ordem elevada, é a grande quantidade de memória requerida e um aumento exagerado da quantidade de trabalho computacional. Dois exemplos de métodos diretos de solução que podem ser citados são o método de Crout e o método de Cholesky.

Basicamente há três passos durante o processo de solução por um método direto. Uma fatoração simbólica para determinar o padrão de esparsidade (*sparsity pattern*) do fator, uma fatoração numérica e retro-substituições (*back-substituition*) triangulares.

A implementação paralela dos métodos diretos é difícil devido principalmente à característica seqüencial inerente dos processos de decomposição e retro-substituição. Esses procedimentos requerem uma grande quantidade de comunicação entre processos, além de apresentar dificuldade na distribuição da carga de trabalho entre os processadores, principalmente quando a matriz apresenta um perfil muito irregular.

4.2.2 – Métodos iterativos

O termo método iterativo se refere às técnicas que utilizam aproximações sucessivas para obter, a cada passo, soluções mais acuradas para o sistema de equações lineares [63].

Os métodos iterativos consistem basicamente na geração de uma seqüência de soluções aproximadas do sistema de equações. Esses métodos partem de uma estimativa inicial para a solução da Equação (4.2) e realiza, então, sucessivas aproximações até que a convergência seja alcançada. A convergência não é garantida para todos os métodos e alguns métodos convergem mais rapidamente do que outros para um problema específico. Ao utilizar a maioria dos métodos iterativos, não se conhece previamente o número de iterações necessárias para alcançar a solução. Para alguns métodos, entretanto, o número máximo de iterações até se obter a solução é conhecido.

Para que os métodos iterativos sejam eficientes é necessário que a convergência seja obtida com o menor número possível de iterações. O número de condição da matriz *A* afeta a taxa de convergência da iteração. O número de condição da matriz *A* é definido como

$$\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}} \tag{4.3}$$

onde $\lambda_{\max}$ e $\lambda_{\min}$ são, respectivamente, o maior e o menor autovalor (*eigenvalue*) da matriz *A*. Quanto mais próximo da unidade estiver o número de condição da matriz *A* mais rapidamente irão convergir os algoritmos iterativos.

A quantidade de trabalho computacional necessária por iteração geralmente é linear ou quadrática em relação ao número de incógnitas, enquanto nos métodos diretos esta relação

geralmente é cúbica. Portanto, os métodos iterativos são mais apropriados quando o número de incógnitas é grande.

4.2.2.1 – Métodos iterativos básicos

De um modo geral, a solução iterativa da Equação 4.2 tem a seguinte forma

$$x^{k+1} = Gx^k + c \quad \text{k = 0, 1, 2, ...} \tag{4.4}$$

onde k é o contador de iterações e G e c são, respectivamente, uma matriz e um vetor, os quais são definidos pelo método a ser utilizado e não dependem de k. Esta matriz e este vetor podem ser expressos em termos de uma matriz não singular Q tal que

$$G = I - Q^{-1}A \text{ e } c = Q^{-1}b \tag{4.5}$$

onde I é a matriz identidade.

Geralmente, esses métodos efetuam uma decomposição da matriz A da seguinte forma

$$A = L_s + D_s + U_s \tag{4.6}$$

onde D_s é uma matriz diagonal, L_s é uma matriz triangular inferior com os elementos da diagonal nulos e U_s é uma matriz triangular superior com os elementos nulos na diagonal principal. Esta decomposição tem um custo computacional mínimo em relação às decomposições efetuadas pelos métodos diretos. Além disso, a matriz A é simétrica e, portanto, $U_s = L_s^T$.

Alguns métodos iterativos básicos são descritos e comentados a seguir.

Método de Richardson

Este é o mais básico dos algoritmos iterativos. A matriz *Q* para este método é definida como

$$Q = I \tag{4.7}$$

portanto, $G = I - A$ e $c = b$.

Apesar da facilidade de implementação, este algoritmo converge apenas quando o autovalor da matriz *A* é menor do que 2, impossibilitando o seu uso para a maioria dos problemas que aparecem na análise de estruturas [64].

Método de Jacobi

Neste método as matrizes são definidas como

$$Q = D_s \tag{4.8}$$

portanto, $G = -D_s^{-1}(L_s + U_s)$ e $c = D_s^{-1}b$.

O método de Jacobi converge sempre que o raio espectral da matriz *A* for menor do que 1. Esta condição é frequentemente respeitada na análise de estruturas, mas a sua ocorrência não é garantida. Define-se raio espectral da matriz *A* como o maior autovalor de *A* em valor absoluto, ou seja, $max\{|\lambda(A)|\}$.

Como apenas uma matriz diagonal deve ser invertida, todas as equações podem ser resolvidas independentemente a cada iteração, o que é apropriado para a implementação em paralelo. O método de Jacobi examina cada uma das equações do sistema $Ax = b$ isoladamente e independente de ordem. Por esta razão, este método é também conhecido como o *método de deslocamentos simultâneos*. Se cada processo ficar responsável por uma parte do sistema total, as trocas de informações necessárias devem corresponder apenas à montagem da matriz *A* no inicio da iteração, à troca da solução x^k com os processos vizinhos após cada iteração e a passagem de escalares durante a checagem da convergência [63].

Método de Gauss-Seidel

Neste método as matrizes têm a seguinte forma

$$Q = D_s + L_s \tag{4.9}$$

portanto,

$G = -(I + D_s^{-1} L_s)^{-1} D_s^{-1} U_s$ e

$c = (I + D_s^{-1} L_s)^{-1} D_s^{-1} b$.

O método de Gauss-Seidel sempre converge se as matrizes A e D_s forem simétricas e positivas definidas.

Como a matriz $(L_s + D_s)$ é invertida, em cada iteração do método de Gauss-Seidel é necessária uma operação de retro-substituição e esta operação é essencialmente seqüencial e requer um grande número de comunicação entre os processos, além de apresentar dificuldade na distribuição equilibrada do trabalho entre os processadores. Pelo fato da existência de dependência entre as iterações, este método é também conhecido como *método de deslocamentos sucessivos* [63].

Método da Sobre-Relaxação Sucessiva

Este método incrementa as propriedades de convergência do método de Gauss-Seidel, multiplicando a solução encontrada em cada iteração por um escalar ω, cujo valor deve ficar entre 0 e 1 se a solução deve sofrer uma sub-relaxação ou, o que é mais comum, entre 1 e 2 se se a solução deve sofrer uma sobre-relaxação (a convergência é garantida somente para $0 < \omega < 2$). Existem vários procedimentos para estimar o valor de relaxação apropriado, entretanto, não existe um que seja ideal para todas as aplicações. As matrizes, neste caso, possuem a seguinte forma

$$Q = \omega^{-1} D_s - L_s \tag{4.10}$$

portanto,

$G = (I + \omega D_s^{-1} L_s)^{-1} (-\omega D_s^{-1} U_s + (1 - \omega) I)$ e

$$c = (I + \omega D_s^{-1} L_s)^{-1} \omega D_s^{-1} b \, .$$

Assim como o método de Gauss-Seidel, neste método também aparecem operações essencialmente seqüenciais durante cada iteração, o que torna este método inapropriado para o ambiente paralelo.

4.2.3 – O Método dos Gradientes Conjugados (MGC)

Para matrizes simétricas positivas definidas, o método dos gradientes conjugados (MGC) é efetivo e o método iterativo mais popular para resolver sistemas grandes de equações lineares [63]. Este método foi desenvolvido por Hestenes e Stiefel no início da década de 50 [65]. A competitividade do MGC considerando os métodos diretos é baseada na necessidade de pouca memória para armazenamento, na pouca quantidade de trabalho computacional necessária por iteração e no fato de ser um algoritmo que possui um paralelismo implícito.

Uma dificuldade associada a vários métodos iterativos é o fato deles, muitas vezes, dependerem de parâmetros que são difíceis de serem escolhidos apropriadamente. Um exemplo é o método da sobre-relaxação sucessiva, apresentado anteriormente, no qual é necessário estimar um valor para a relaxação adequado ao problema em análise. Entretanto, não existe a garantia de que o procedimento utilizado para estimar este valor seja o mais apropriado. O método dos gradientes conjugados, por sua vez, não apresenta este tipo de dificuldade, o que é uma vantagem em relação aos demais métodos iterativos.

Neste trabalho, o método dos gradientes conjugados é utilizado para solucionar o sistema de equações, discretizado através do método de elementos finitos e esta seção é dirigida ao entendimento da solução do método dos gradientes conjugados (MGC) em suas formulações seqüencial e paralela.

O método dos gradientes conjugados garante que o número de iterações necessárias até que a solução correta seja alcançada seja, no máximo, igual ao número de equações do sistema (N_{eq}). Entretanto, essa garantia existe numa máquina ideal, na qual não ocorrem erros de arredondamento durante as iterações. Dessa forma, principalmente quando o número de condição do sistema for elevado (10^4 a 10^5), ou seja, quando o sistema apresenta-se mal

condicionado, os erros de arredondamento podem dificultar a convergência do método, podendo ocorrer casos onde o número de iterações ultrapasse bastante o N_{eq} até que a precisão desejada seja alcançada. Técnicas de pré-condicionamento, apesar de não incluídas na implementação realizada, são empregadas para aumentar a taxa de convergência.

4.2.3.1 – Forma quadrática

A Equação 4.2 pode ser reescrita como:

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{4.11}$$

onde A é uma matriz de dimensão n x n, x e b vetores de dimensão n.

Shewchuk [61] afirma que uma forma quadrática é uma função quadrática, escalar de um vetor que tem a seguinte forma:

$$f(x) = \frac{1}{2}\mathbf{x}^{\mathbf{T}} \cdot \mathbf{A} \cdot \mathbf{x} - \mathbf{b}^{\mathbf{T}} \cdot \mathbf{x} + c \tag{4.12}$$

onde A é uma matriz, x e b são vetores e c é um escalar. Shewchuk [61] mostra ainda que se A é simétrica e positiva definida, então $f(x)$ é minimizado pela solução para $Ax = b$.

Assumindo, por exemplo, os seguintes valores para A, b e c.

$$\mathbf{A} = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 2 \\ -8 \end{bmatrix}, \quad c = 0. \tag{4.13}$$

A solução do sistema de equações, neste caso, é de fácil obtenção e o vetor x é igual a [2, -2]. O funcional da forma quadrática para os valores mostrados acima está representado na Figura 4.2 [61].

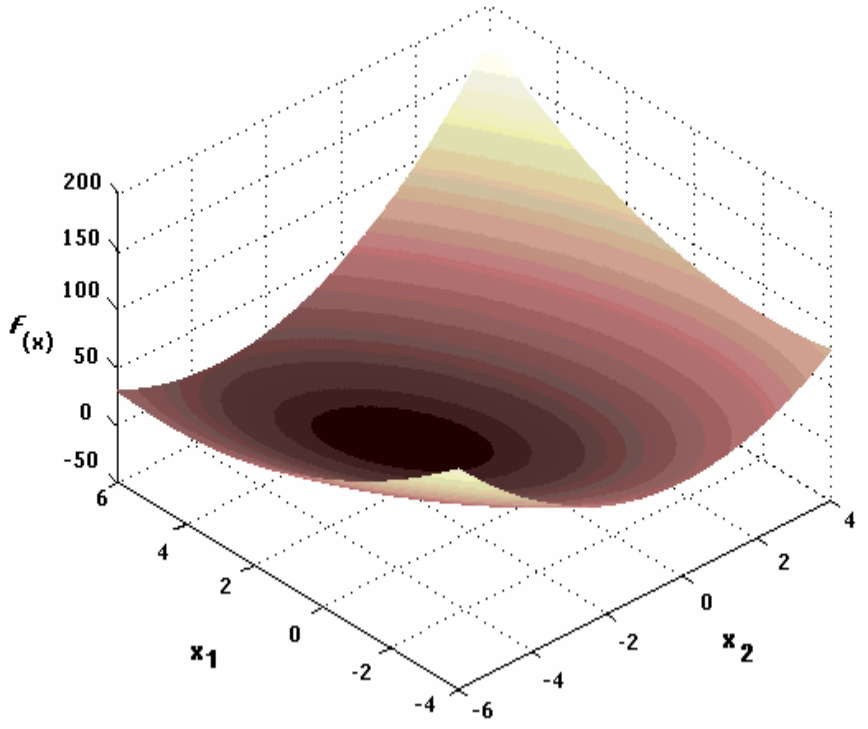

Figura 4.2 – Forma quadrática f(x): x_1 e x_2 são as componentes do vetor x.

Na Figura 4.2 pode-se visualizar que a solução do sistema apresentado é o ponto de mínimo do parabolóide. Ainda pode ser observado que, sendo A uma matriz positiva definida, a superfície determinada por $f(x)$ tem a forma de um parabolóide com concavidade para cima [61].

O gradiente da forma quadrática é definido por [61]:

$$f'(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \frac{\partial}{\partial x_2} f(x) \\ \vdots \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix} \tag{4.14}$$

O gradiente é um campo vetorial que, para um dado ponto x, aponta para a direção de maior crescimento da função $f(x)$. No ponto inferior do parabolóide o gradiente é nulo. Pode-se minimizar a função $f(x)$ impondo que $f'(x)$ seja igual a zero [61].

Aplicando a Equação 4.14 à Equação 4.12 obtém-se [61]:

$$f'(x) = \frac{1}{2}\mathbf{x} \cdot \mathbf{A} + \frac{1}{2}\mathbf{A} \cdot \mathbf{x} - \mathbf{b} \tag{4.15}$$

sendo *A* simétrica, a Equação 4.14 pode ser reescrita na forma

$$f'(x) = \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \tag{4.16}$$

Atribuindo-se zero ao gradiente, obtém-se a equação 4.1, que é o sistema linear que se deseja resolver. Portanto, a solução para $Ax = b$ é o ponto crítico de *f(x)*. Se *A* é positiva definida além de ser simétrica, então esta solução é um mínimo de *f(x)*, assim $Ax = b$ pode ser resolvido encontrando-se um *x* que minimize *f(x)* [61].

4.2.3.2 – Algoritmo paralelo para o Método dos Gradientes Conjugados

No método dos gradientes conjugados a busca pela solução se inicia em um ponto arbitrário $\mathbf{x_0}$ e iterativamente procura-se o ponto de mínimo de $f(x)$, solução do sistema. Para tanto é realizada uma série de passos $\mathbf{x_0}$, $\mathbf{x_1}$, $\mathbf{x_2}$ ..., até que o valor de $\mathbf{x}_k$ esteja próximo o suficiente da solução **x**. Pode-se definir: erro, resíduo e direção de busca através das Equações 4.17, 4.18 e 4.19, respectivamente [66]:

$$\mathbf{e}_k = \mathbf{x}_k - \mathbf{x} \tag{4.17}$$

onde $\mathbf{x}_k$ é o ponto em análise, $\mathbf{e}_k$ é o erro relativo a $\mathbf{x}_k$ e **x** é a solução.

$$\mathbf{r_0} = \mathbf{b} - \mathbf{A} \cdot \mathbf{x_0} \tag{4.18}$$

onde $\mathbf{r_0}$ é o vetor resíduo do ponto de partida $\mathbf{x_0}$.

$$\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \frac{\|\mathbf{r}_{k+1}\|^2}{\|\mathbf{r}_k\|^2}\mathbf{d}_k \tag{4.19}$$

ou

$$\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \beta_{k+1}\,\mathbf{d}_k \tag{4.20}$$

O comprimento do deslocamento na direção de busca é obtido por meio do seguinte cálculo

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k \tag{4.21}$$

onde,

$$\alpha_k = \frac{\left\| \mathbf{r}_k \right\|^2}{\mathbf{d}_k \cdot \mathbf{A} \cdot \mathbf{d}_k} \tag{4.22}$$

O resíduo **r** é atualizado pela seguinte relação:

$$\mathbf{r}_{k+1} = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_{k+1} = \mathbf{b} - \mathbf{A}(\mathbf{x}_k - \alpha_k\,\mathbf{d}_k) = \mathbf{r}_k - \alpha_k \mathbf{A} \cdot \mathbf{d}_k \tag{4.23}$$

Para se aproveitar o paralelismo para a solução do sistema de equações lineares $Ax = b$, tanto a matriz A quanto o vetor de incógnitas x e o vetor de termos independentes b estão particionados e distribuídos entre os processadores.

O algoritmo paralelo do método dos gradientes conjugados implementado em [8] e utilizado neste trabalho tem como referência os trabalhos publicados por [67] [68].

Para paralelizar a solução, a geometria discretizada pelo método dos elementos finitos é particionada em subdomínios e estes, então, são distribuídos entre os processos (tarefas). Cada tarefa resolve, então, as equações para os elementos que pertencem subdomínio pelo qual ficou responsável durante o particionamento. Desta forma, os elementos da matriz A**,** da Equação 4.2, ficam distribuídos entre as várias tarefas. Existem graus de liberdade que são compartilhados por mais de uma tarefa, chamados graus de liberdade de fronteira e outros contidos apenas em um único subdomínio, denominados graus de liberdade internos (ou privados). A matriz A deve ser reestruturada de maneira que os valores dos produtos necessários para a solução do MGC não se alterem. A Figura 4.3 [8] ilustra uma malha discretizada em elementos e particionada em subdomínios.

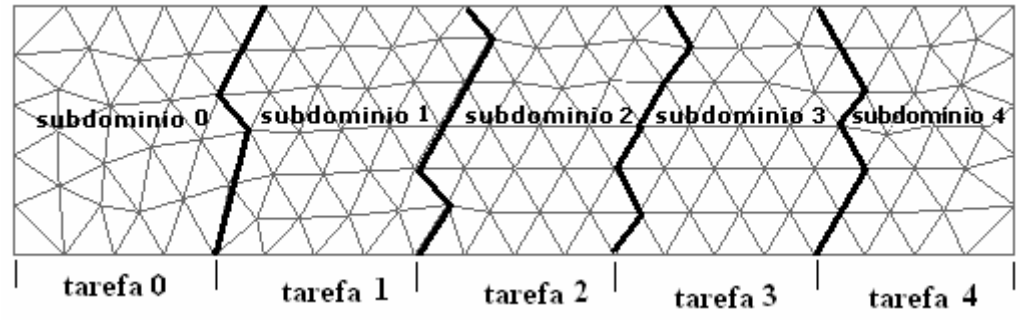

Figura 4.3 – Partição de uma malha em 5 outras sub-malhas.

Assim, a matriz A de cada subdomínio computacional é desmembrada em quatro outras, que são A_p, A_s, B_p e B_p^T. A_p é uma matriz quadrada com dimensão igual ao número de graus de liberdade internos de cada subdomínio, na qual, estão armazenados os valores de rigidez referentes aos graus de liberdade internos. Os graus de liberdade internos são aqueles que pertencem a um único subdomínio. A matriz A_s também é quadrada e seu tamanho é definido pelo número de graus de liberdade que estão na fronteira de cada partição. Em A_s estão os valores da matriz de rigidez pertinentes aos graus de liberdade de fronteira. Os graus de liberdade de fronteira são aqueles que pertencem a dois ou mais subdomínios. 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. Durante o processo de partição das malhas, os nós e os graus de liberdade são renomeados de forma a permitir que cada tarefa veja seu subdomínio como um domínio, sem perder a relação com o domínio original.

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 b_p e b_s, sendo que o primeiro tem dimensão igual aos graus de liberdade internos e o segundo aos de fronteira. O mesmo procedimento é feito para o vetor x.

O algoritmo paralelo para o método dos gradientes encontra-se na Figura 4.4. Inspecionando este algoritmo (e tomando como referência as Figuras 4.11 e 4.12 e a Tabela 4.1) é possível observar que cada processo (tarefa) i, para $i = 0$ *até* $p-1$ (onde p é o número de processos (tarefas)) executa exatamente o mesmo algoritmo. Pode-se observar também que a comunicação entre as tarefas ocorre somente em dois subprogramas: Atualização e Produto

Interno. Cada passo realizado dentro do laço de solução do MGC é chamado de módulo (M1 até M12).

Para um valor de tolerância conhecido ε

1. $\mathbf{x}_i^0 = \mathbf{0}$ $\quad \mathbf{x}_{Si}^0 = \mathbf{0}$ $\quad \beta^0 = 0$

2. $\mathbf{b}_{Si}$ = Atualização ($\mathbf{b}_{Si}$)

3. $\mathbf{r}_i^0 = \mathbf{b}_i$ $\quad \mathbf{r}_{Si}^0 = \mathbf{b}_{Si}$

4. $\mathbf{d}_i^0 = \mathbf{b}_i$ $\quad \mathbf{d}_{Si}^0 = \mathbf{b}_{Si}$

5. γ^0 = Produto Interno $(\mathbf{r}_i^0, \mathbf{r}_{Si}^0; \mathbf{r}_i^0, \mathbf{r}_{Si}^0)$

Para **k = 0, 1, ...** Repetir os passos de 6 a 17

6. $\mathbf{q}_i^k = \mathbf{A}_{Pi} \cdot \mathbf{d}_i^k + \mathbf{B}_{Pi} \cdot \mathbf{d}_{Si}^k$ //M1

7. $\mathbf{q}_{Si}^k = \mathbf{B}_{Pi}^t \cdot \mathbf{d}_i^k + \mathbf{A}_{Si} \cdot \mathbf{d}_{Si}^k$ //M2

8. Atualização($\mathbf{q}_{Si}^k$) //M3

9. τ^k = Produto Interno $(\mathbf{d}_i^k, \mathbf{d}_{Si}^k; \mathbf{q}_i^k, \mathbf{q}_{Si}^k)$ //M4

10. $\alpha^k = \gamma^k / \tau^k$ //M5

11 $\mathbf{x}_i^{k+1} = \mathbf{x}_i^k + \alpha^k \mathbf{d}_i^k$ $\quad \mathbf{r}_i^{k+1} = \mathbf{r}_i^k - \alpha^k \mathbf{q}_i^k$ //M6

12 $\mathbf{x}_{si}^{k+1} = \mathbf{x}_{si}^k + \alpha^k \mathbf{d}_{si}^k$ $\quad \mathbf{r}_{si}^{k+1} = \mathbf{r}_{si}^k - \alpha^k \mathbf{q}_{si}^k$ //M7

13 Armazena γ^k //M8

14. γ^{k+1} = Produto Interno $(\mathbf{r}_i^{k+1}, \mathbf{r}_{Si}^{k+1}; \mathbf{r}_i^{k+1}, \mathbf{r}_{Si}^{k+1})$ //M9

15. Se $\left(\sqrt{\gamma^{k+1}} \leq \varepsilon\right)$ **PARE** //M10

16. $\beta^k = \gamma^{k+1} / \gamma^k$ //M11

17. $\mathbf{d}_i^{k+1} = \mathbf{r}_i^{k+1} + \beta^k \mathbf{d}_i^k$ $\quad \mathbf{d}_{Si}^{k+1} = \mathbf{r}_{Si}^k + \beta^k \mathbf{d}_{Si}^k$ //M12

Figura 4.4 – Algoritmo de solução do gradiente conjugado paralelo [68].

```
Atualização(a){
    Faça i = 1 até nº de tarefas
            Faça j = 1 até Comum[i]
                    Buf [j] = a [Vizinhos[i, j]]
            Se Comum[i] > 0
            Send (Buf, Comum[i], i)
    Faça de i = 1 até nº de tarefas
            Se Comum[i] > 0
                        Receive (Buf, Comum[i], i)
            Faça j = 1 até Comum[i]
            a [Vizinhos[i, j]] + = Buf[j]
    }
```

Figura 4.5 – Algoritmo da função de Atualização [68].

Produto Interno ($\mathbf{a}_1$, $\mathbf{a}_2$; $\mathbf{c}_1$, $\mathbf{c}_2$) {

$$\gamma = \mathbf{a_1} \cdot \mathbf{c_1} + \sum (\mathbf{a_2}[i] \times \mathbf{c_2}[i]) / (\textbf{Compartilhados}[i])$$

$$\text{Allreduce}(\gamma, soma, MPI\_SUM)$$

Retorne (*soma*)

}

Figura 4.6 – Algoritmo da função Produto Interno para MPI [68].

Tabela 4.1 – Estruturas de dados de cada tarefa (associada com as Figuras 4.5 e 4.6)

Compartilhado[i]	Vetor que armazena informação sobre quantas tarefas compartilha determinado grau de liberdade. Sua dimensão é igual ao número de graus de liberdade de fronteira.
Comum[i]	Vetor que armazena a quantidade de graus de liberdade compartilhada entre tarefas. Sua dimensão é dada pelo número de tarefas.
Vizinhos[i,j]	Matriz que armazena o número local de graus de liberdade de fronteira que o nó *j* compartilha com a tarefa *i*. Sua dimensão igual ao número de tarefas por número de graus de liberdade de fronteira.

A função Produto Interno (Figura 4.6) é usada para calcular o produto interno entre dois vetores que se encontram distribuídos entre as tarefas e requer uma comunicação global. Esta comunicação é uma redução que determina a soma do produto interno de cada tarefa e então fornece a cada processo uma cópia desta soma. Neste caso, todas as tarefas enviam a variável γ e a soma de todos γ é retornada para todas as tarefas.

A função Atualização (Figura 4.5) utiliza comunicações ponto-a-ponto entre os processos que compartilham graus de liberdade. O propósito desta função é permitir que os valores dos graus de liberdade dos nós de fronteira possam ser montados.

4.3 – A aplicação *benchmark*

O código da aplicação *benchmark* foi escrito na linguagem C, utilizando a biblioteca de comunicação MPI [8].

O pré-processamento e o pós-processamento do programa desenvolvido são realizados através de exportação e importação de arquivos entre este e o aplicativo GID® [34]. Sendo assim, numa primeira etapa em uma única malha, é gerado o modelo de elementos finitos, que envolve o desenho da geometria alvo, a geração da malha de elementos finitos e a aplicação das condições de contorno do problema. Por meio do aplicativo são organizados os dados de entrada para o programa. A estrutura de dados gerada pelo aplicativo GID® é armazenada em um arquivo compartilhado por todas as máquinas do *cluster*, assim cada processador (ou tarefa) pode acessar e trabalhar estas informações.

Em seguida, o programa executa o particionamento da malha em subdomínios, realizado através do programa METIS® [69]. Esta decomposição de domínio é feita em paralelo por todas as tarefas. Como as partições possuem aproximadamente o mesmo número de elementos, cada tarefa efetua basicamente o mesmo número de operações [57]. Neste momento, cada tarefa já conhece o subdomínio que lhe pertence e, portanto, quais elementos lhe pertence e, então, cada tarefa cria suas matrizes de elementos A_p, A_s, B_p e B_p^T. Finalmente, o programa resolve o sistema de equações algébricas gerado pelo método dos gradientes conjugados.

Quando o aplicativo METIS® efetua o particionamento da malha entre as tarefas, vão existir nós de alguns elementos finitos que estarão compartilhados entre mais de uma tarefa e, portanto, os graus de liberdade destes nós estarão também compartilhados (graus de liberdade de fronteira). Isso pode ser observado na Figura 4.3, que apresenta uma geometria discretizada dividida em subdomínios. Para que o sistema de equações algébricas possa ser corretamente solucionado pelo método dos gradientes conjugados, é necessário que ocorram comunicações entre as tarefas que compartilham graus de liberdade para que sejam trocadas informações a respeito destes graus de liberdade de fronteira.

A Figura 4.7 apresenta uma malha de elementos finitos tridimensional decomposta em duas, quatro e oito partes. Estas partições foram definidas pelo programa METIS e a visualização gerada pelo programa PMVis (*Partitioned Mesh Visualizer*) [57].

A Figura 4.8 (adaptada de [8]) mostra resumidamente o funcionamento do programa.

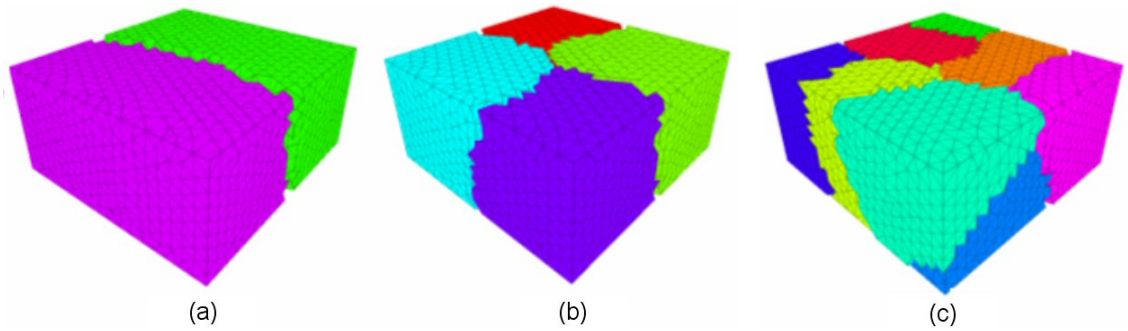

Figura 4.7 - Partição de uma malha em: (a) duas, (b) quatro e (c) oito partes

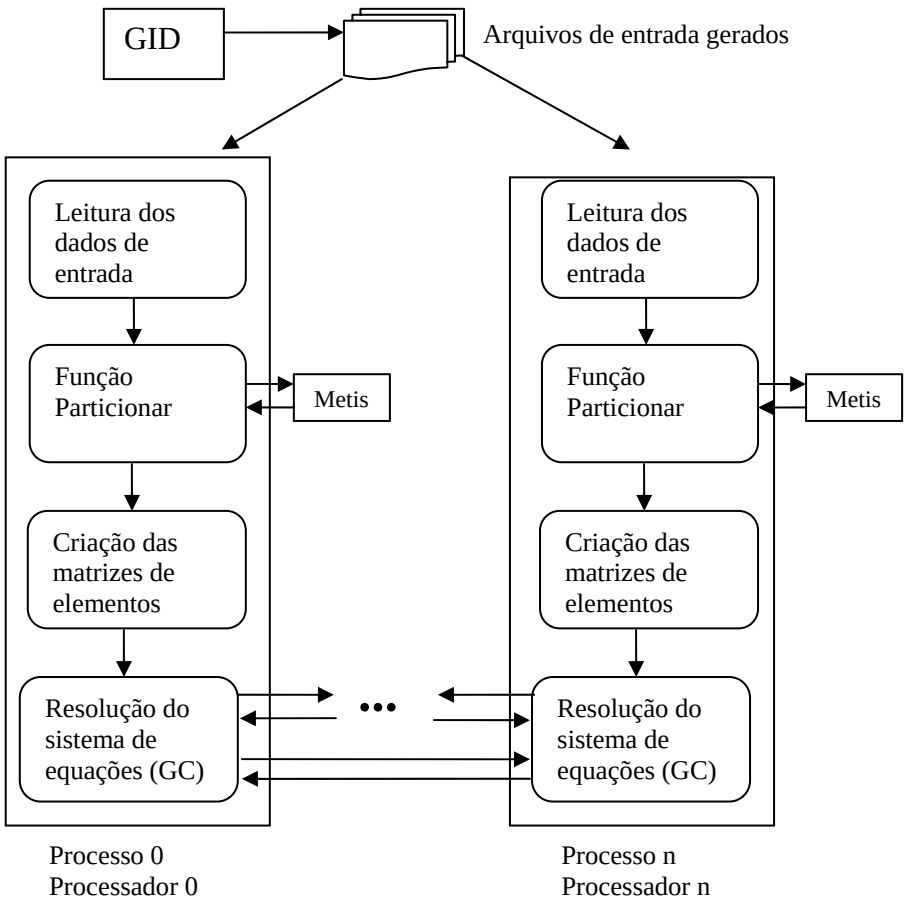

Figura 4.8 – Esquema resumido do funcionamento do programa.

4.4 – Trabalhos relacionados

A aplicação utilizada neste trabalho foi implementada em [8], com o objetivo de verificar o ganho de desempenho obtido com a paralelização do algoritmo do método dos gradientes conjugados. Entretanto, o foco do trabalho realizado em [8] foi apresentar os resultados baseados nos tempos totais de execução da aplicação. Além disso, a aplicação foi executada somente em ambiente distribuído homogêneo.

Neste trabalho, o objetivo é apresentar uma medição de tempo de execução mais detalhada, através da inserção de diversos pontos de leitura de tempo de execução. Esta medição detalhada tem a intenção de caracterizar o tempo gasto por cada uma das principais atividades desempenhadas pela aplicação. Desta forma, foi possível identificar o percentual do tempo total de execução que é gasto com comunicação e o ganho de desempenho obtido com a concorrência existente em máquinas SMP, quando comparado a máquinas monoprocessadas.

Além disso, a aplicação foi executada em ambiente heterogêneo, com o objetivo de verificar a viabilidade da utilização destes ambientes para a simulação do método dos elementos finitos aplicado à elasticidade linear, para a solução de problemas estruturais. Com base nos tempos obtidos através da medição detalhada, foi possível estabelecer um balanceamento de carga empírico para o ambiente heterogêneo no qual a aplicação foi executada.

5 – RESULTADOS

Este capítulo apresenta os resultados obtidos de medições efetuadas em pontos que demandam maior quantidade de tempo em suas execuções, dentro do código da aplicação *benchmark,* objetivando a caracterização de desempenho desta aplicação. Os valores obtidos provêm da execução da aplicação em dois *clusters* distintos de PCs e no ambiente heterogêneo formado por máquinas destes dois *clusters*.

5.1 – Captura do tempo

Uma maneira de obter os tempos é capturar diretamente a hora do sistema. Neste caso, a precisão do tempo capturado obtida é dependente do sistema operacional e existe um *delay* que corresponde ao intervalo de tempo necessário para que um processo requisitado para execução pelo sistema operacional seja atendido pelo processador. A outra maneira de capturar o tempo é indiretamente, através da utilização de ciclos de *clock* e posterior divisão destes pela freqüência do processador no qual o programa está sendo executado. Utilizando ciclos de *clock*, é possível capturar o tempo no nível do hardware e, portanto, o tempo capturado tende a ser mais rápido e mais preciso. A utilização da instrução *assembly rdtsc* provida pelas máquinas Intel desde o Pentium II elimina o *overhead* de chamadas de funções, otimizando o desempenho.

A contagem dos ciclos de *clock* que ocorrem no processador é armazenada no contador de *time-stamp* e para acessar este contador, é utilizada a instrução *assembly* rdtsc (*read time-stamp counter*). Nos processadores Intel o contador de *time-stamp* é um registro de 64 bits, que é incrementado a cada ciclo de *clock* do processador. O contador é reinicializado a cada vez que a máquina é reiniciada [73].

A existência do registro TSC permite uma monitoração de tempo independente do compilador utilizado, pois se trata de uma contagem por *hardware*. Assim, a temporização de um determinado evento pode ser realizada de forma bem precisa.

Foram comparadas as seguintes funções (ou instruções) que capturam tempo, direta ou indiretamente:

- a função *gettimeofday*( ), provida pelo sistema operacional Linux, que retorna o tempo decorrido, expresso em segundos e micro-segundos,
- a função *MPI_Wtime*, que é um comando da biblioteca MPI que retorna, em segundos, o tempo decorrido desde um tempo arbitrário no passado,
- a função *rdtscll( )* que retorna um *time stamp*, em ciclos de *clock* e
- a instrução *assembly rdtsc*, que retorna a contagem dos ciclos de *clock* que ocorrem no processador, armazenada no contador *time-stamp*.

Para efetuar as comparações, foram implementados quatro programas na linguagem C que fazem chamadas a cada uma das funções e/ou instruções acima. Cada programa faz cem chamadas a uma das funções ou instrução acima e entre estas chamadas são colocadas intruções *assembly rdtsc* com o objetivo de medir, em ciclos de *clock*, o custo de execução de cada uma das cem chamadas de cada função ou instrução. Estes programas foram executados em 3 máquinas com processadores distintos e com distintas capacidades de memória, conforme descritas na Tabela 5.1.

Tabela 5.1 – Configurações das máquinas

	Máquina 1	**Máquina 2**	**Máquina 3**
Processador	AMD Athlon MP 1900+ 1,7 GHz	Pentium II 350 MHz	Celeron 2,4 GHz
Memória RAM	1,0 GB	128 MB	512 MB
Sistema operacional	Linux Red Hat 9	Debian Linux	Linux Red Hat 9

O objetivo desta comparação é descobrir qual das funções ou instrução tem menor custo de execução durante sua chamada para ser utilizada nas medidas dos tempos de determinadas operações dentro do código da aplicação *benchmark*.

Os valores obtidos, em ciclos de *clock*, como resultados das execuções dos quatro programas nas diferentes máquinas são apresentados na Tabela 5.2 e nas Figuras 5.1 a 5.3. A Tabela 5.3 apresenta os valores médios da Tabela 5.2 convertidos para milisegundos e a Figura 5.4 apresenta graficamente os valores expostos na Tabela 5.3.

Tabela 5.2 – Valores obtidos, em ciclos de *clock*, para as execuções da instrução *assembly rdtsc* e das funções *rdtscll( )*, *gettimeofday( )* e *MPI_Wtime ( )*.

		Máquina 1	**Máquina 2**	**Máquina 3**
Instrução *assembly rdtsc*	Melhor caso	11	34	84
	Pior caso	26	74	240
	Média	12	37	87
Função *Rdtscll( )*	Melhor caso	32	88	180
	Pior caso	356	176	768
	Média	94,4	97	216
Função *gettimeofday*( )	Melhor caso	399	519	3.164
	Pior caso	9.063	4.031	3.884
	Média	513	556	3.392
Função M*PI_Wtime( )*	Melhor caso	1.167	1.410	7.992
	Pior caso	18.446	45.728	34.100
	Média	1.293	1.440	8.289

Tabela 5.3 – Valores médios convertidos em milisegundos, para as execuções da instrução *assembly rdtsc* e das funções *rdtscll( )*, *gettimeofday( )* e *MPI_Wtime ( )*.

	Instrução *assembly rdtsc*	Função *rdtscll( )*	Função *gettimeofday*( )	Função *MPI_Wtime ( )*
Máquina 1	0,00000882	0,00005552	0,00030217	0,00076058
Máquina 2	0,00010571	0,00027714	0,00158965	0,00411428
Máquina 3	0,00003708	0,00009125	0,00141333	0,00345375

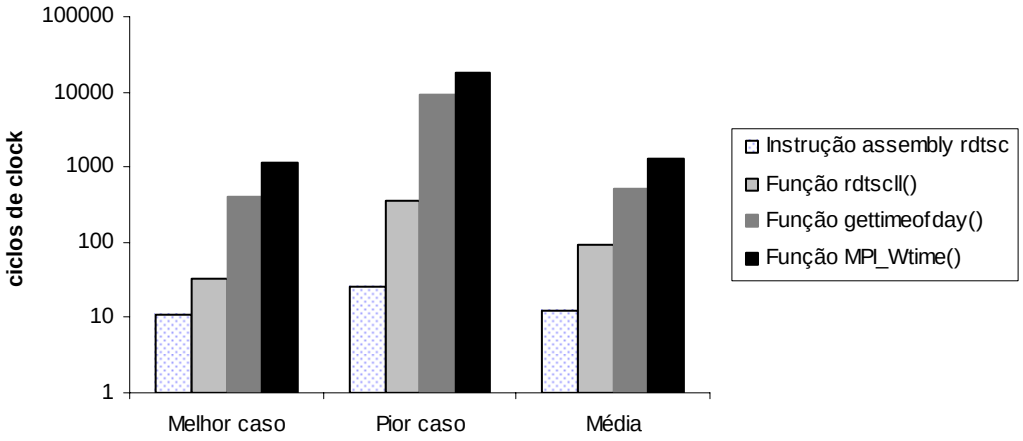

Figura 5.1 - Custos de execução obtidos, em ciclos de *clock*, para a instrução *assembly rdtsc* e para as funções *rdtsc( )*, *gettimeofday( )* e *MPI_Wtime( )* na Máquina 1.

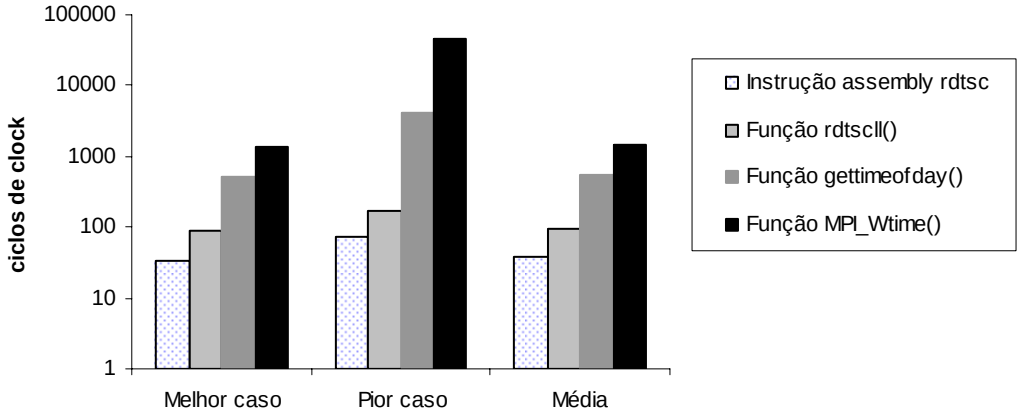

Figura 5.2 - Custos de execução obtidos, em ciclos de *clock*, para a instrução *assembly rdtsc* e para as funções *rdtsc( )*, *gettimeofday( )* e *MPI_Wtime( )* na Máquina 2.

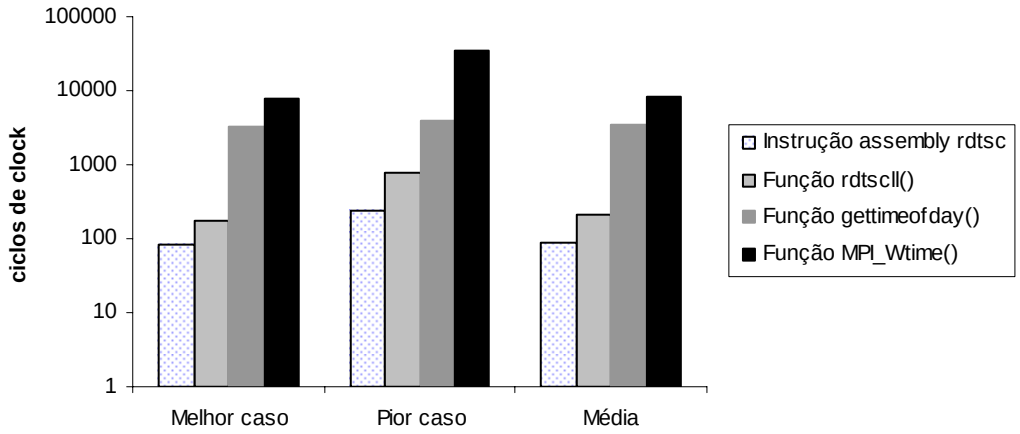

Figura 5.3 - Custos de execução obtidos, em ciclos de *clock*, para a instrução *assembly rdtsc* e para as funções *rdtsc( )*, *gettimeofday( )* e *MPI_Wtime( )* na Máquina 3.

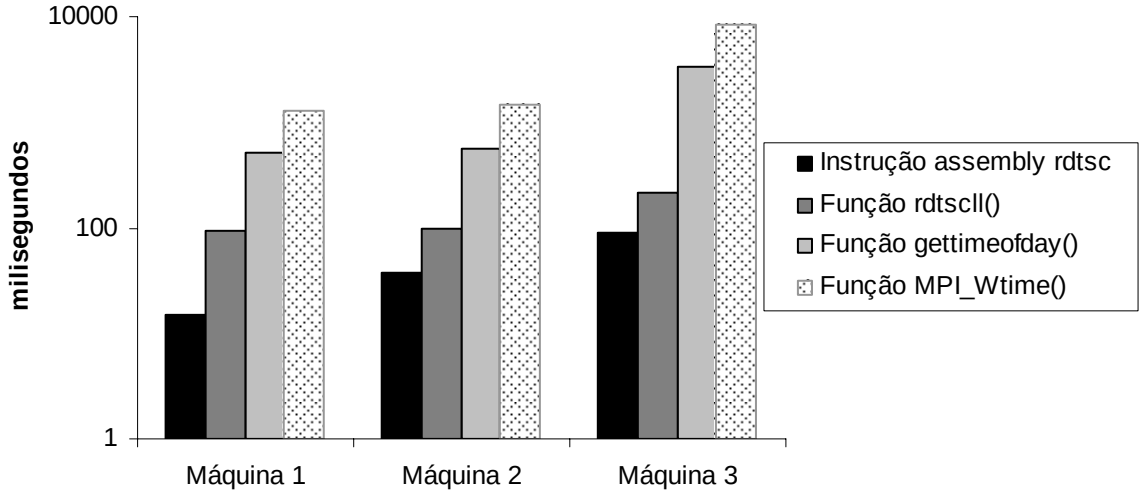

Figura 5.4 – Tempos médios de execução obtidos, em milisegundos, para as execuções da instrução *assembly rdtsc* e das funções *rdtsc( )*, *gettimeofday( )* e *MPI_Wtime( )*, nas máquinas 1, 2 e 3.

Como pode ser observado, o número de ciclos de *clock* e também os tempos gastos para a execução da instrução *assembly rdtsc* é consideravelmente o menor em todas as três máquinas e em todos os três casos: melhor, pior e média, sendo, portanto, a menos intrusiva para ser colocada dentro do código da aplicação *benchmark* de modo a obter os custos de execução de todo o código ou de trechos deste.

Como o objetivo deste trabalho é caracterizar o desempenho de uma aplicação, será utilizado predominantemente, para a captura dos tempos de execução do código paralelo ou de trechos deste, os ciclos de *clock* (capturados com a utilização da instrução *assembly rdtsc*), os quais serão convertidos para unidade de tempo. Estes são mais precisos que a hora do sistema, pois eliminam o *delay* ocasionado pelo sistema operacional na requisição de processos (tempo de requisição, espera e concessão para execução) ao processador, criando uma independência do tipo de sistema operacional. A instrução *assembly rdtsc* evita o risco de as medidas de tempo serem afetadas pelo sistema operacional.

5.2 – Infra-estrutura utilizada para testes experimentais

A aplicação *benchmark* foi executada em máquinas pertencentes a dois *clusters* de computadores localizados no IE, Instituto de Ciências Exatas da Universidade de Brasília. As características dos dois *clusters* estão relacionadas na Tabela 5.4.

Tabela 5.4 – Características dos *clusters* utilizados durante as execuções

	Cluster 1	*Cluster* 2
Número de máquinas	8	8
Processador	AMD Athlon MP 1900+	Intel Pentium II
Freqüência de *clock*	1,7 GHz	350 MHz
Memória RAM	1,0 GB	128 MB
Memória *cache*	256 KB	512 KB
Sistema operacional	Linux Red Hat 9	Debian Linux
Rede de interconexão	*Crossbar* - Ethernet 1,0 Gb/s	*Crossbar* - Ethernet 100 Mb/s

Cada uma das oito máquinas do primeiro *cluster* (*Cluster* 1, Figura 5.5) possui dois processadores. O segundo *cluster* (*Cluster* 2) é exibido na Figura 5.6.

Figura 5.5 – *Cluster* 1.

Figura 5.6 – *Cluster* 2.

5.3 – Metodologia e geometria utilizadas para ensaio

A geometria das malhas escolhidas para execução da aplicação é o paralelepípedo mostrado na Figura 5.7 e uma malha de elementos finitos correspondente com as condições de contorno aplicadas é apresentada na figura 5.8.

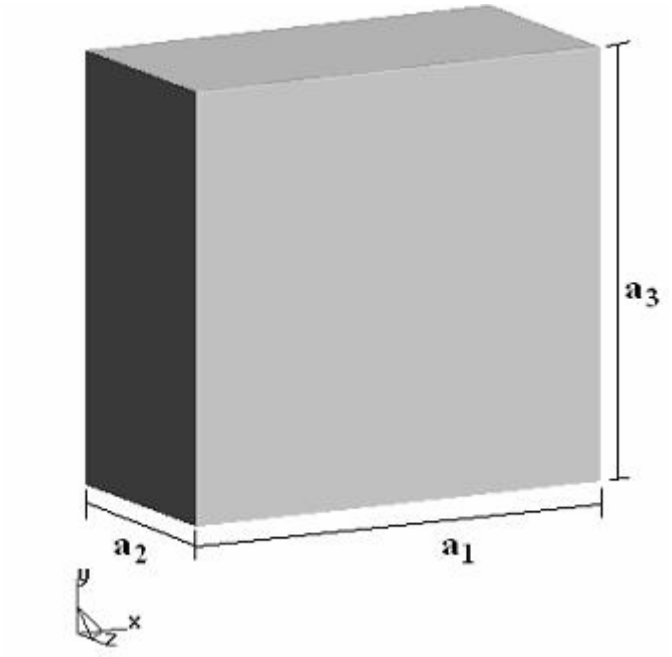

Figura 5.7 - Geometria tridimensional utilizada para execução do código, onde $a_1 = a_3 = 1{,}0m$ e $a_2 = 0{,}5m$.

O ensaio consistiu em restringir uma das faces do paralelepípedo e aplicar uma carga concentrada de 500 N, na direção negativa do eixo y, em um nó localizado em um dos vértices, de forma a provocar uma flexão. As condições de contorno aplicadas estão indicadas na Figura 5.8.

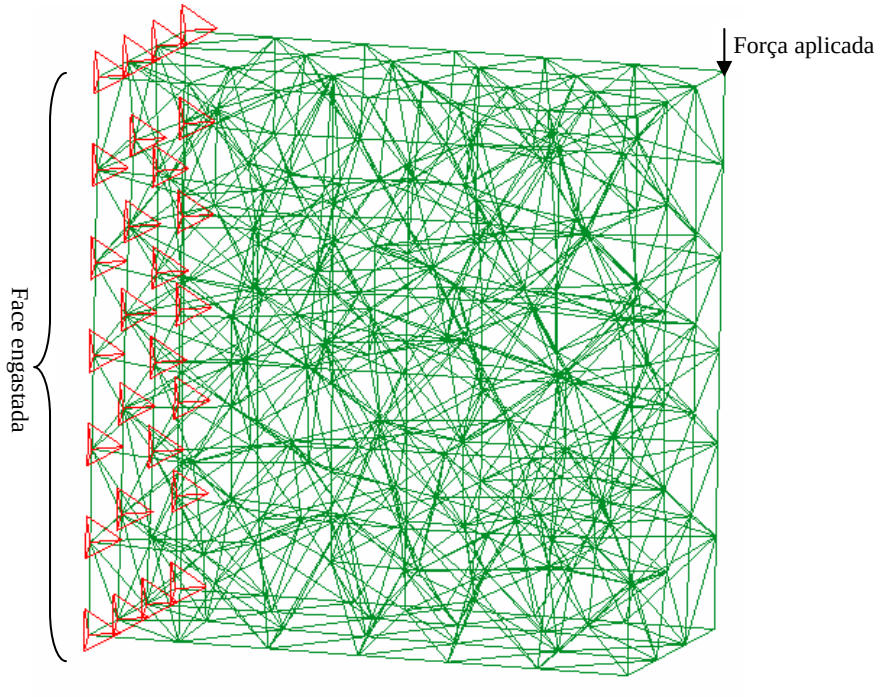

Figura 5.8 – Geometria com as condições de contorno aplicadas.

As principais características das três malhas de elementos finitos utilizadas para análise de desempenho são apresentadas na Tabela 5.5.

Tabela 5.5 – Características das malhas utilizadas

	Malha 1	**Malha 2**	**Malha 3**
Número de nós	233	933	1.984
Número de elementos	855	4.014	9.126
Número total de graus de liberdade	699	2.799	5.952
Número de graus de liberdade restritos	81	222	387
Número de equações	618	2.577	5.565

5.4 – Caracterização de desempenho da aplicação *benchmark*

Os valores aqui exibidos se referem à execução do código da aplicação utilizando a biblioteca de comunicação MPI (mpich-1.2.6), para uma tolerância de convergência do método dos gradientes conjugados de 1×10^{-6}.

Todos os valores de tempos de execução apresentados resultam da média dos valores obtidos a partir de todas as tarefas em execução para que possa haver uma maior precisão destes.

5.4.1 - Comparação de tempos de execução da aplicação no *Cluster* 1 e no *Cluster* 2

Inicialmente, foram medidos os tempos de execução expressos em milisegundos para três etapas do programa: particionamento da malha, montagem das matrizes de elementos e solução do sistema de equações pelo método dos gradientes conjugados. Nesta execução, cada tarefa foi alocada a um nó distinto, nos *clusters* 1 e 2. Estas medidas, juntamente com a medida de tempo total de execução são apresentadas nas Tabelas 5.6 a 5.11 e nas Figuras 5.9 a 5.14.

Tabela 5.6 – Tempos de execução para a Malha 1 no *Cluster* 1, sendo cada tarefa alocada a um nó (SMP) distinto.

Nº de tarefas	Tempo de particionamento (milisegundos)	Tempo de montagem (milisegundos)	Tempo de solução do MGC (milisegundos)	Tempo total de execução (milisegundos)
1	10,983399	81,089356	822,512801	914,586506
2	33,677250	24,000125	515,197944	572,878216
4	15,885343	12,029543	254,890687	282,807269

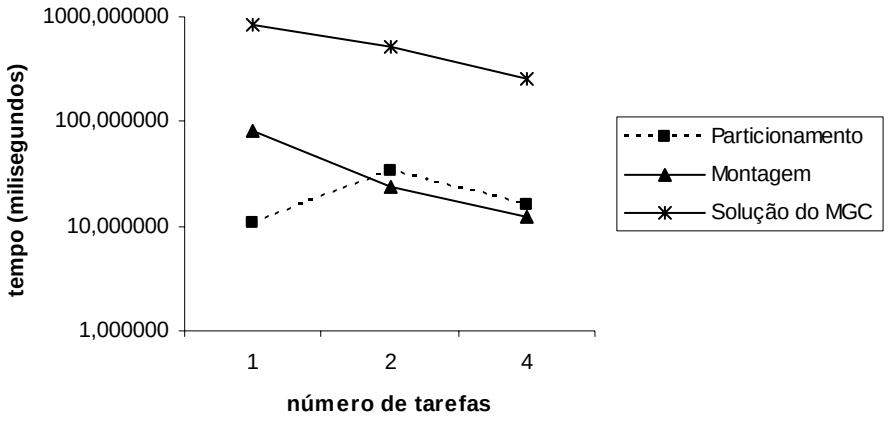

Figura 5.9 – Tempos de execução para a Malha 1 no *Cluster* 1, sendo cada tarefa alocada a um nó (SMP) distinto.

Tabela 5.7 – Tempos de execução para a Malha 1 no *Cluster* 2, sendo cada tarefa alocada a um nó distinto.

Nº de tarefas	Tempo de particionamento (milisegundos)	Tempo de montagem (milisegundos)	Tempo de Solução do MGC (milisegundos)	Tempo total de execução (milisegundos)
1	15,070020	280,456234	6.146,843209	6.442,371311
2	29,816100	138,125717	3.024,004363	3.291,970573
4	32,82930786	66,91257357	1.187,250309	1.287,00393

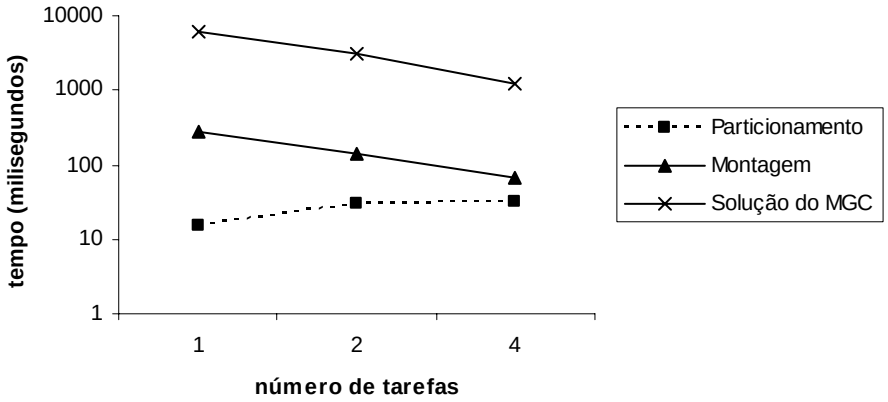

Figura 5.10 – Tempos de execução para a Malha 1 no *Cluster* 2, sendo cada tarefa alocada a um nó distinto.

Tabela 5.8 – Tempos de execução para a Malha 2 no *Cluster* 1, sendo cada tarefa alocada a um nó (SMP) distinto.

Nº de tarefas	Tempo de particionamento (milisegundos)	Tempo de montagem (milisegundos)	Tempo de solução do MGC (milisegundos)	Tempo total de execução (milisegundos)
1	15,064388	343,509846	22.290,128377	22.648,703560
2	38,508328	167,253720	8.942,463421	9.148,233616
4	31,673379	61,547763	3.702,036315	3.795,261000

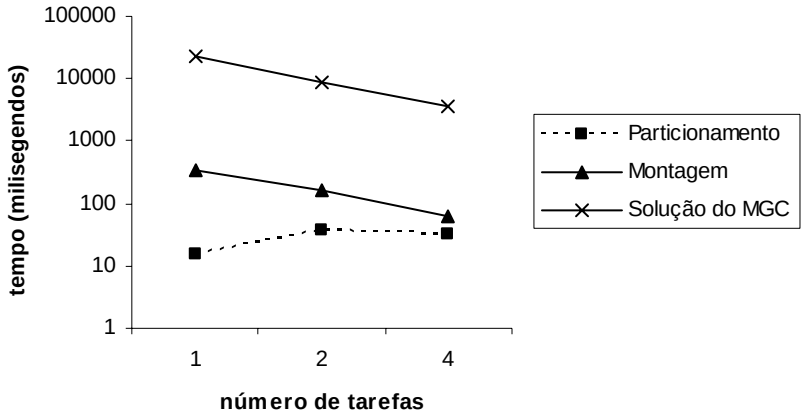

Figura 5.11 – Tempos de execução para a Malha 2 no *Cluster* 1, sendo cada tarefa alocada a um nó (SMP) distinto.

Tabela 5.9 – Tempos de execução para a Malha 2 no *Cluster 2* , sendo cada tarefa alocada a um nó distinto.

Nº de tarefas	Tempo de particionamento (milisegundos)	Tempo de montagem (milisegundos)	Tempo de Solução do MGC (milisegundos)	Tempo total de execução (milisegundos)
1	110,772466	1.625,167986	176.958,572194	178.694,514591
2	95,535166	717,441386	61.877,003023	62.689,994321
4	93,886874	340,957269	24.228,050533	24.662,934057

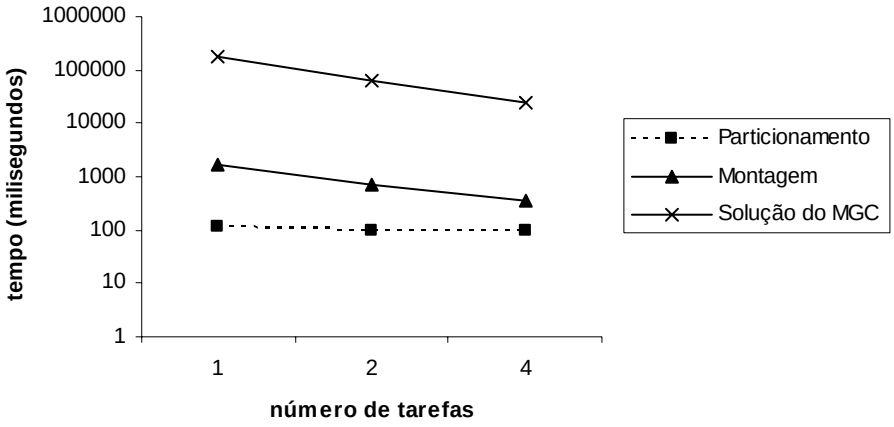

Figura 5.12 – Tempos de execução para a Malha 2 no *Cluster* 2, sendo cada tarefa alocada a um nó distinto.

Tabela 5.10 – Tempos de execução para a Malha 3 no *Cluster* 1, sendo cada tarefa alocada a um nó (SMP) distinto.

Nº de tarefas	Tempo de particionamento (milisegundos)	Tempo de montagem (milisegundos)	Tempo de solução do MGC (milisegundos)	Tempo total de execução (milisegundos)
1	65,763922	1.035,438408	138.316,488641	139.417,691645
2	69,247299	399,527420	53.818,605990	54.287,389058
4	42,294679	154,185366	20.625,133120	20.821,621845

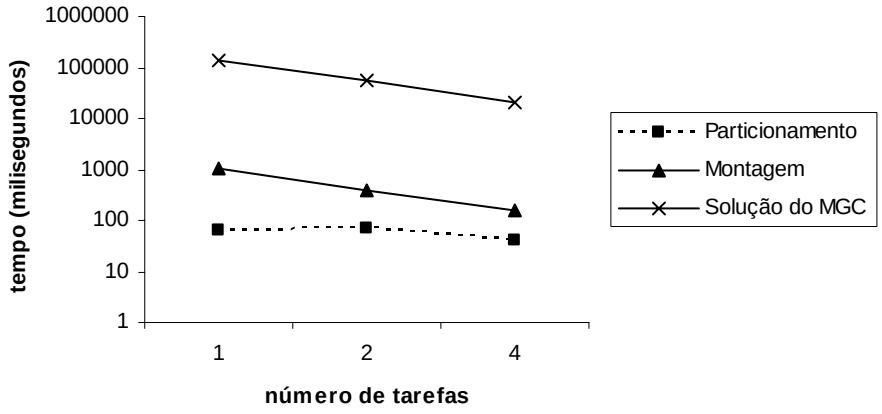

Figura 5.13 – Tempos de execução para a Malha 3 no *Cluster* 1, sendo cada tarefa alocada a um nó (SMP) distinto.

Tabela 5.11 – Tempos de execução para a Malha 3 no *Cluster* 2, sendo cada tarefa alocada a um nó distinto.

Nº de tarefas	Tempo de particionamento (milisegundos)	Tempo de montagem (milisegundos)	Tempo de solução do MGC (milisegundos)	Tempo total de execução (milisegundos)
1	150,253954	6.201,251220	1.119.382,584037	1.125.734,097360
2	206,425511	1.881,229441	378.398,207580	389.485,917057
4	353,341332	1.603,460262	255.051,300994	257.008,139626

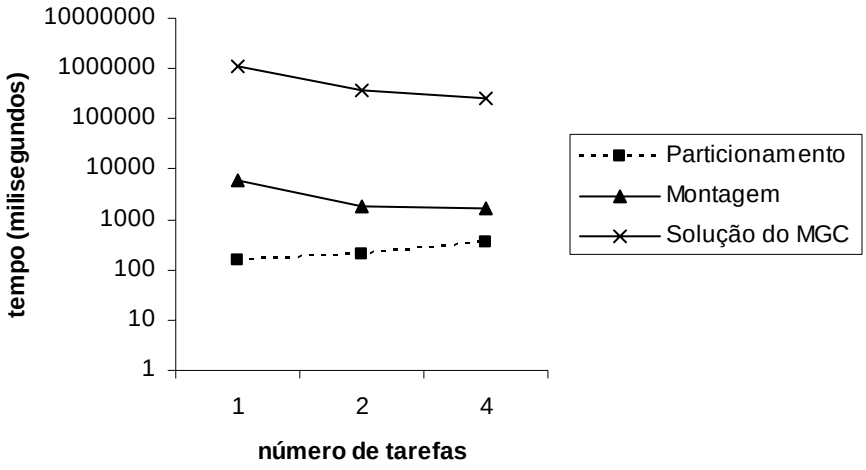

Figura 5.14 – Tempos de execução para a Malha 3 no *Cluster* 2, sendo cada tarefa alocada a um nó distinto.

Como pode ser observado pelos resultados obtidos, os tempos de particionamento tendem a aumentar com o aumento do número de tarefas. Isto ocorre porque aumentando o número de tarefas, o número de partições aumenta, fazendo com que o aplicativo Metis® despenda uma maior quantidade de tempo realizando a partição da malha computacional em subdomínios. Este tempo também tende a crescer com o aumento do tamanho da malha utilizada durante a execução, pois quanto maior a malha maior o número de elementos que terão que ser distribuídos entre as tarefas.

Os valores medidos do tempo de montagem da matriz decaem com o aumento do número de tarefas, pois fica atribuída a cada tarefa uma menor quantidade de graus de liberdade e, consequentemente, um menor número de equações para serem resolvidas. O tempo de

montagem tende a aumentar com o aumento do tamanho da malha devido ao fato de aumentarem os números de graus de liberdade atribuídos às tarefas.

O tempo de solução dos gradientes conjugados é o tempo gasto na solução do sistema de equações. É nesta etapa onde ocorrem as comunicações ponto-a-ponto (*send* e *receive*) e as comunicações globais (*allreduce*). É possível observar que o tempo solução dos gradientes conjugados tem um maior decréscimo quando o número de tarefas passa de 1 para 2. Quando o número de tarefas passa de 2 para 4 o decréscimo é menor para todas as três malhas e para os dois *clusters*. Tal fato ocorre porque com o aumento do número de tarefas, aumentam os graus de liberdade compartilhados entre as tarefas, aumentando, consequentemente, também as comunicações.

O tempo total para todas as malhas e em ambos os *clusters* diminuiu com o aumento do número de tarefas.

5.4.2 - Comparação de tempos de execução da aplicação no *Cluster* 1 com os nós mono ou dual processados

Cada nó do *Cluster* 1 é um multiprocessador simétrico (SMP), conforme já foi descrito na Seção 5.2. Foram executadas 1, 2, 4 e 8 tarefas em um único nó dual processado (SMP) e também foram executadas 1, 2, 4 e 8 tarefas neste mesmo nó monoprocessado, no *Cluster* 1.

As medidas obtidas destas execuções, para as etapas de particionamento da malha, montagem das matrizes de elementos e solução do método dos gradientes conjugados, são apresentadas nas Tabelas 5.12 a 5.17 e nas Figuras 5.15 a 5.20.

Tabela 5.12 – Tempos de execução para a Malha 1 no *node8* (SMP) do *Cluster* 1.

Número de tarefas	Tempo de particionamento (milisegundos)	Tempo de montagem (milisegundos)	Tempo de solução do MGC (milisegundos)	Tempo total de execução (milisegundos)
1	10,983399	81,08935647	822,5128011	914,586505
2	4,257215	25,03653	494,7236	524,0205
4	4,692661	15,512971	361,364329	381,571701
8	62,791920	18,085809	1.705,515803	1.786,396291

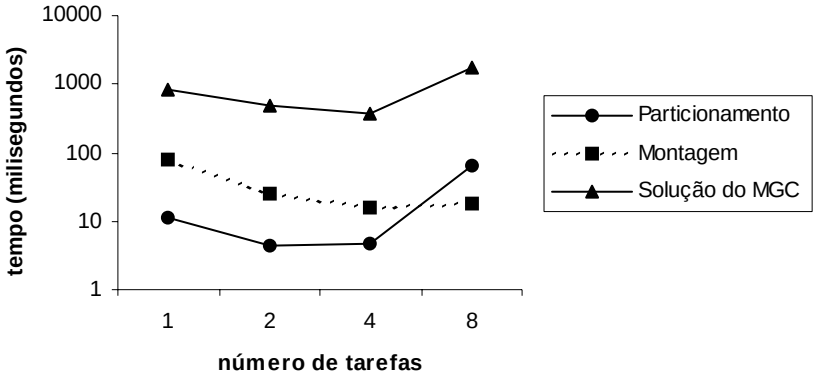

Figura 5.15 – Tempos de execução para a Malha 1 no *node8* (SMP) do *Cluster* 1.

Tabela 5.13 – Tempos de execução para a Malha 1 *node8* (monoprocessado) do *Cluster* 1.

Número de tarefas	Tempo de particionamento (milisegundos)	Tempo de montagem (milisegundos)	Tempo de solução do MGC (milisegundos)	Tempo total de execução (milisegundos)
1	87,594978	52,470486	842,559801	982,625919
2	128,626256	24,228551	833,813186	986,671193
4	66,794143	11,821128	919,363287	997,980441
8	318,626299	5,682648	3.293,829959	3.618,141578

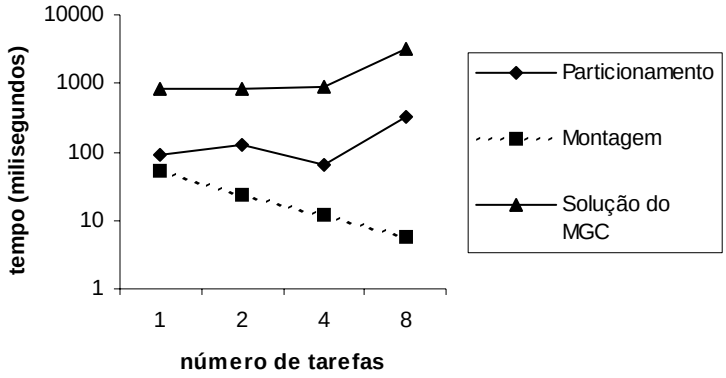

Figura 5.16 – Tempos de execução para a Malha 1 no *node8* (monoprocessado) do *Cluster* 1.

Tabela 5.14 – Tempos de execução para a Malha 2 no *node8* (SMP).

Número de tarefas	Tempo de particionamento (milisegundos)	Tempo de montagem (milisegundos)	Tempo de solução do MGC (milisegundos)	Tempo total de execução (milisegundos)
1	12,515423	333,155444	22.075,841395	22.421,512867
2	20,553950	203,420004	9.176,798800	9.397,570976
4	55,085474	76,998480	6.650,775687	6.782,863157
8	48,964171	165,472347	5.705,982455	5.920,422720

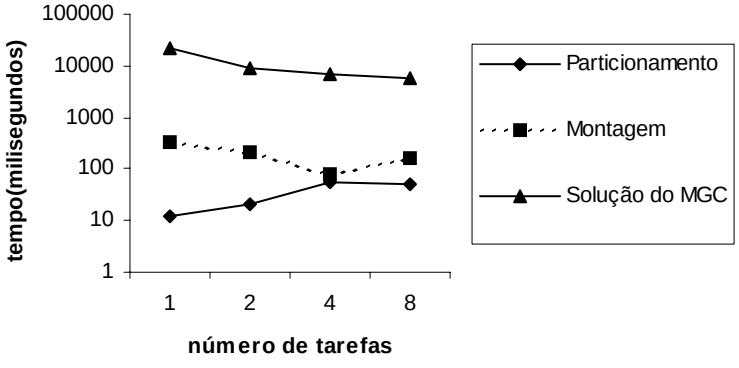

Figura 5.17 – Tempos de execução para a Malha 2 no *node8* (SMP) do *Cluster* 1.

Tabela 5.15 – Tempos de execução para a Malha 2 no *node8* (monoprocessado) do *Cluster* 1.

Número de tarefas	Tempo de particionamento (milisegundos)	Tempo de montagem (milisegundos)	Tempo de solução do MGC (milisegundos)	Tempo total de execução (milisegundos)
1	55,604776	332,856415	22.441,694902	22.830,157083
2	35,714642	301,781582	20.136,340046	20.473,839363
4	30,151193	125,667376	11.588,355683	11.744,177690
8	158,044407	171,279945	10.382,543116	10.711,871031

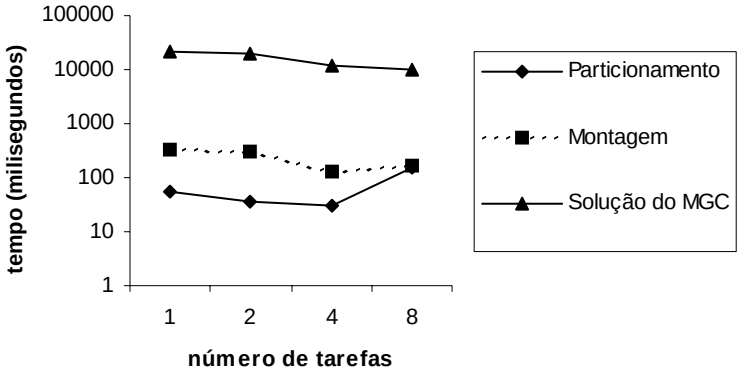

Figura 5.18 – Tempos de execução para a Malha 2 no *node8* (monoprocessado) do *Cluster* 1.

Tabela 5.16 – Tempos de execução para a Malha 3 no *node8* (SMP) do *Cluster* 1.

Número de tarefas	Tempo de particionamento (milisegundos)	Tempo de montagem (milisegundos)	Tempo de solução do MGC (milisegundos)	Tempo total de execução (milisegundos)
1	65,763922	1.035,438408	138.316,488641	139.417,691645
2	55,768689	478,411601	55.018,881896	55.553,070543
4	37,430016	273,274944	37.905,515961	38.216,226022
8	265,778955	247,496646	23.903,845939	24.417,125862

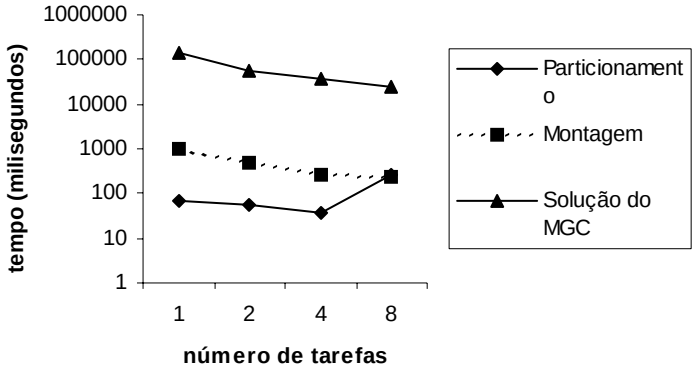

Figura 5.19 – Tempos de execução para a Malha 3 no *node8* (SMP) do *Cluster* 1.

Tabela 5.17 – Tempos de execução para a Malha 3 no *node8* (monoprocessado) do *Cluster* 1.

Número de tarefas	Tempo de particionamento (milisegundos)	Tempo de montagem (milisegundos)	Tempo de solução do MGC (milisegundos)	Tempo total de execução (milisegundos)
1	239,360728	1.111,398458	142.347,938919	143.698,698837
2	84,902996	755,921283	106.536,141062	107.376,973591
4	104,901787	445,625780	70.736,306202	71.286,838787
8	812,344549	815,013691	44.646,058009	46.273,420646

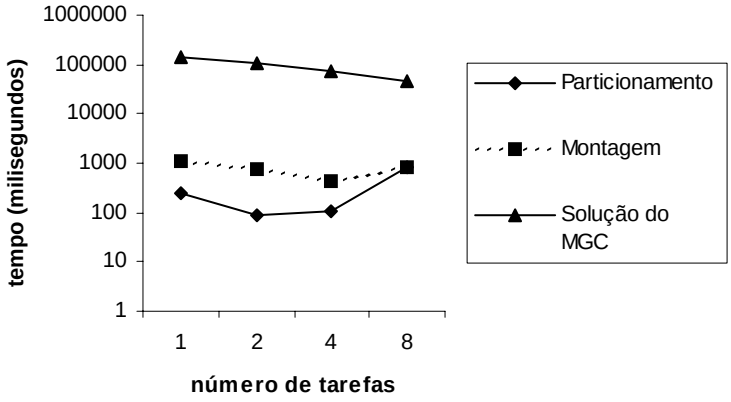

Figura 5.19 – Tempos de execução para a Malha 3 no *node8* (monoprocessado) do *Cluster* 1.

Em algumas execuções, os valores de tempo de montagem da matriz de rigidez são superiores quando executados no nó dual processado (SMP). Isso é explicado pelo fato de que, durante a

etapa de montagem da matriz, o acesso à memória compartilhada no modo SMP acaba se tornando um gargalo, fazendo com que seja gasto mais tempo para a execução desta etapa.

Para a simulação utilizando a Malha 1 e o nó dual processado, o menor valor do tempo de solução do MGC e do tempo total ocorre com a execução de 4 tarefas; com 8 tarefas o tempo já aumenta quase três vezes. Com o nó monoprocessado, o tempo de solução do MGC já começa a aumentar com 4 tarefas em execução e o tempo total já aumenta a partir de 2 tarefas em execução, mostrando que o ideal neste caso, é que se execute apenas uma tarefa em cada nó monoprocessado.

Para a Malha 2, o decréscimo de tempo vai se tornando menos significativo à medida que o número de tarefas vai aumentando, principalmente quando passa de 4 para 8 tarefas em execução. O tempo de execução tende a aumentar quando o número de tarefas ultrapassa 8 tarefas. O mesmo comportamento pode ser observado para a Malha 3.

Conforme já mostrado na Tabela 5.5, a Malha 3 possui um maior número de nós e um maior número de elementos do que a Malha 2, e esta possui um maior número de nós e um maior número de elementos do que a Malha 1. Este maior número de nós resulta em um maior número de equações a serem solucionadas. Com isto, a Malha 3 é beneficiada pela distribuição do domínio em um maior número de tarefas, já que a redução do tempo gasto na solução das equações vai compensar o tempo gasto na comunicação entre as tarefas. Por outro lado, a Malha 1 já não apresenta um bom desempenho quando o número de tarefas ultrapassa 4 tarefas, pois, por possuir um número menor de equações, o custo de comunicação terá uma maior influência no tempo total de execução, mesmo que a comunicação esteja ocorrendo na mesma máquina.

Para um maior detalhamento do comportamento da execução do programa dentro de um nó, a Figura 5.20 apresenta, em ordem cronológica, as etapas de particionamento, montagem e os primeiros eventos de *send* e *receive* entre duas tarefas em execução em um mesmo nó, sendo este nó dual processado, *node8* do *Cluster* 1. A Figura 5.21 se refere à execução de uma tarefa em um nó monoprocessado. Foram colocadas instruções *assembly rdtsc* dentro do código para capturar os *times stamps* dos eventos ocorridos. A malha utilizada nesta simulação foi a Malha 1.

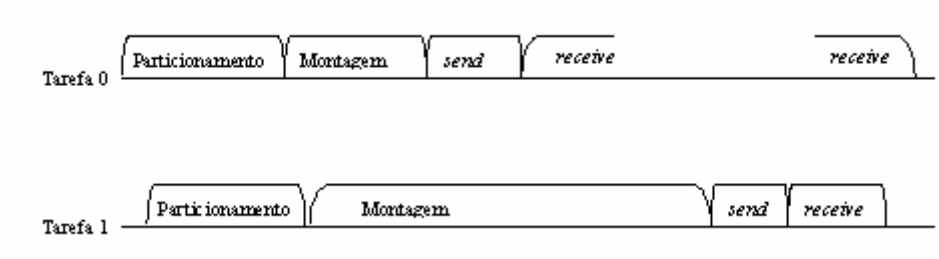

Figura 5.20 – Ordem cronológica para execução de duas tarefas no *node8* (SMP).

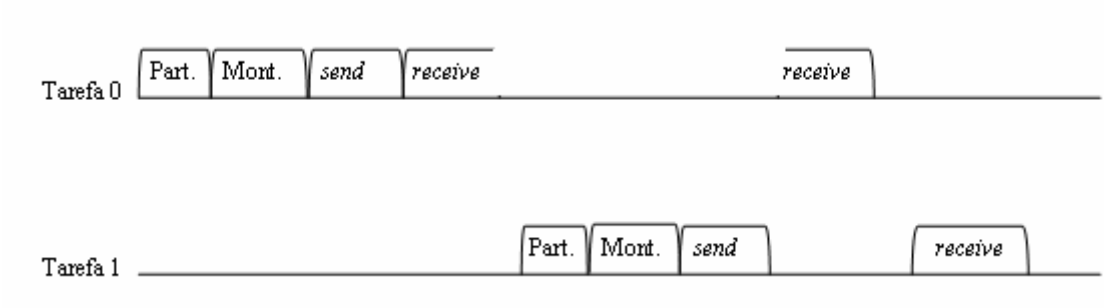

Figura 5.21 – Ordem cronológica para execução de duas tarefas no *node8* (monoprocessado).

Destas duas figuras acima, é possível verificar uma justificativa para o fato de os tempos totais de execução são sempre maiores para o caso de os nodos monoprocessados quando comparados aos nodos SMP, visto que as duas tarefas executam concorrentemente quando na execução no nó dual processado.

5.4.3 - Caracterização de desempenho da etapa de solução do Método dos Gradientes Conjugados

A Figura 5.22 apresenta a relação entre o número de equações e o número de iterações realizadas durante a solução do MGC para as três malhas testadas. É possível observar que o número de iterações aumenta com o aumento do número de equações.

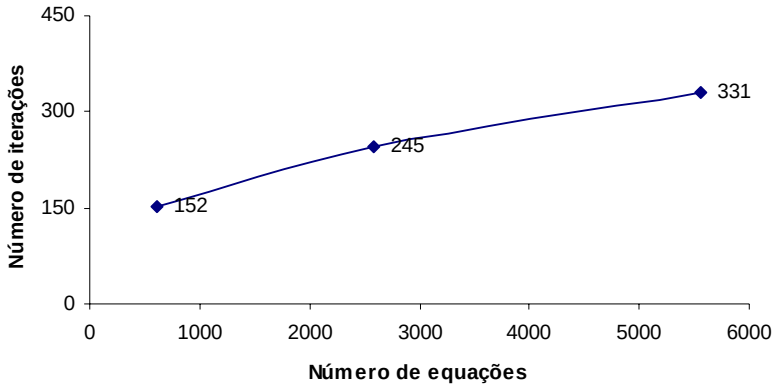

Figura 5.22 – Número de iterações do MGC para as três malhas.

Conforme pode ser observado através das medidas já exibidas até aqui, a solução das equações pelo MGC é a etapa da aplicação que mais consome tempo de execução, consumindo, em média, cerca de 90% do tempo total da aplicação. Por esse motivo, esta etapa do programa foi medida mais detalhadamente, medindo, principalmente, os tempos gastos com comunicações ocorridas dentro da mesma.

Nesta execução, cada tarefa foi alocada a um nó distinto, nos *clusters* 1 e 2, sendo os nós do *Cluster* 1 multiprocessadores simétricos (SMP). As medidas obtidas com o custo de comunicação, juntamente com o custo da etapa de solução do MGC e o percentual do tempo de solução do MGC ao qual o tempo de comunicação corresponde, são mostradas nas Tabelas 5.18 a 5.23 e nas Figuras 5.23 a 5.28.

Tabela 5.18 – Custo de comunicação para a Malha 1 no *Cluster* 1, sendo cada tarefa alocada a um nó (SMP) distinto.

Número de tarefas	Tempo de solução do MGC (milisegundos)	Custo de comunicação (milisegundos)	Percentual
2	515,197944	174,129897	33,80%
3	278,6419098	187,413540	67,26%
4	254,8906874	180,560117	70,84%
5	214,964074	179,555895	83,53%
6	248,1422556	219,438110	88,43%

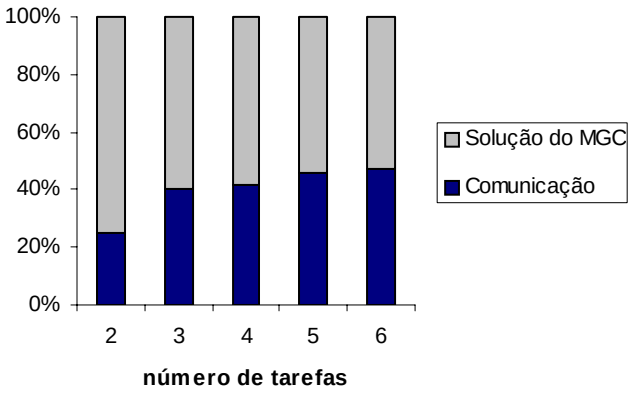

Figura 5.23 – Percentual do custo de comunicação em relação ao tempo de solução do MGC da Malha 1 no *Cluster* 1, sendo cada tarefa alocada a um nó (SMP) distinto.

Tabela 5.19 – Custo de comunicação para a Malha 1 no *Cluster* 2, sendo cada tarefa alocada a um nó distinto.

Número de tarefas	Tempo de solução do MGC (milisegundos)	Custo de comunicação (milisegundos)	Percentual
2	3.024,004363	682,916270	22,58%
3	1.745,514112	510,636115	29,25%
4	1.187,250309	437,553916	36,85%

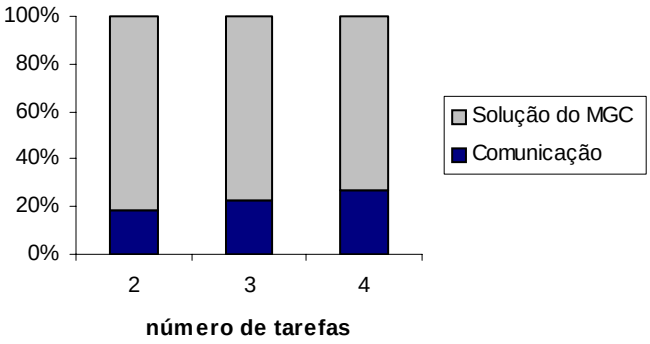

Figura 5.24 – Percentual do custo de comunicação em relação ao tempo de solução do MGC da Malha 1 no *Cluster* 2, sendo cada tarefa alocada a um nó distinto.

Tabela 5.20 – Custo de comunicação para a Malha 2 no *Cluster* 1, sendo cada tarefa alocada a um nó (SMP) distinto.

Número de tarefas	Tempo de solução do MGC (milisegundos)	Custo de comunicação (milisegundos)	Percentual
2	8.942,463421	277,306933	3,10%
3	5.001,829780	609,569673	12,19%
4	3.702,036315	980,038046	26,47%
5	2.808,501028	2.130,905186	75,87%
6	2.287,800475	1.768,875843	77,32%

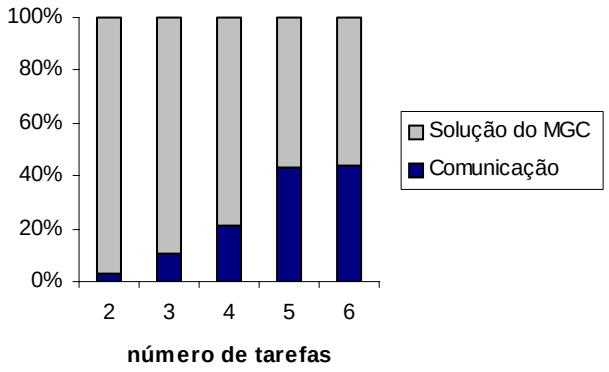

Figura 5.25 – Percentual do custo de comunicação em relação ao tempo de solução do MGC da Malha 2 no *Cluster* 1, sendo cada tarefa alocada a um nó (SMP) distinto.

Tabela 5.21 – Custo de comunicação para a Malha 2 no *Cluster* 2, sendo cada tarefa alocada a um nó distinto.

Número de tarefas	Tempo de solução do MGC (milisegundos)	Custo de comunicação (milisegundos)	Percentual
2	61877,00302	2.516,470937	4,07%
3	35.177,87173	2.607,559583	7,41%
4	24.228,05053	2.357,598101	9,73%

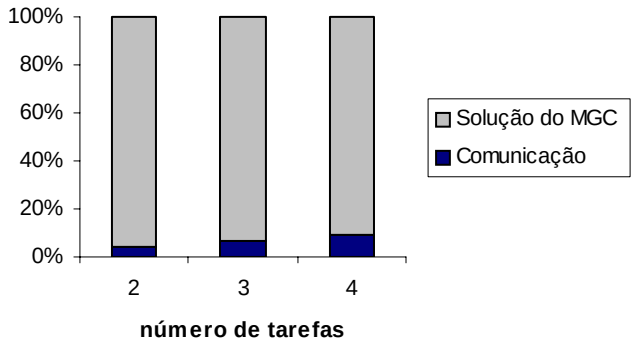

Figura 5.26 – Percentual do custo de comunicação em relação ao tempo de solução do MGC da Malha 2 no *Cluster* 2, sendo cada tarefa alocada a um nó distinto.

Tabela 5.22 – Custo de comunicação para a Malha 3 no *Cluster* 1, sendo cada tarefa alocada a um nó (SMP) distinto.

Número de tarefas	Tempo de solução do MGC (milisegundos)	Custo de comunicação (milisegundos)	Percentual
2	53.818,60599	2.093,965723	3,89%
3	31.166,32692	4.072,904394	13,07%
4	20.625,13312	5.700,632141	27,64%
5	25.379,52016	9.313,163455	36,70%
6	12.111,32165	10.717,6656	88,49%

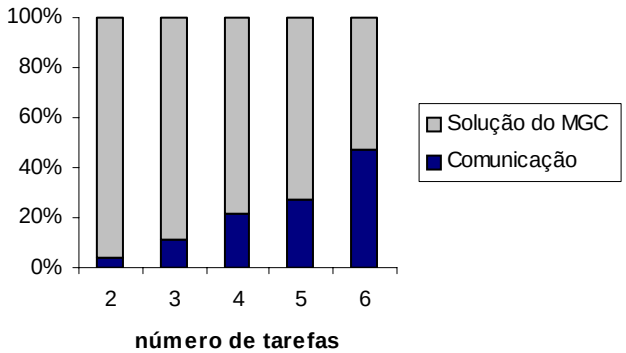

Figura 5.27 – Percentual do custo de comunicação em relação ao tempo de solução do MGC da Malha 3 no *Cluster* 1, sendo cada tarefa alocada a um nó (SMP) distinto..

Tabela 5.23 – Custo de comunicação para a Malha 3 no *Cluster* 2, sendo cada tarefa alocada a um nó distinto.

Número de tarefas	Tempo de solução do MGC (milisegundos)	Custo de comunicação (milisegundos)	Percentual
2	378.398,207580	11.293,525983	2,98%
3	209.392,953604	14.943,208966	7,14%
4	255.051,300994	59.588,012386	23,36%

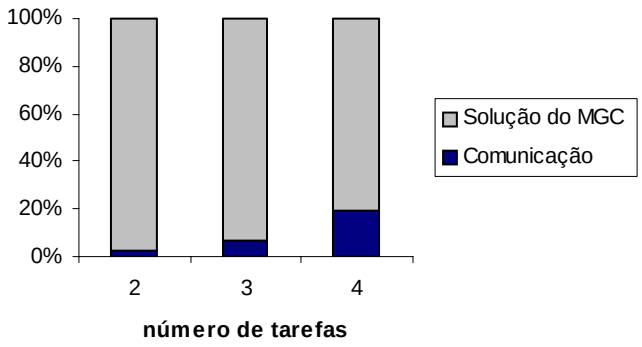

Figura 5.28 – Percentual do custo de comunicação em relação ao tempo de solução do MGC da Malha 3 no *Cluster* 2, sendo cada tarefa alocada a um nó distinto.

Para as execuções das três malhas nos dois *clusters*, o comportamento foi o mesmo: houve um acréscimo do tempo de comunicação com o aumento do número de tarefas, ou seja, o percentual do custo de comunicação em relação ao tempo de solução do Método dos Gradientes Conjugados aumentou à medida que mais tarefas foram executadas.

Para a execução em ambos os *clusters*, utilizando a Malha 1, o custo de comunicação representa um percentual elevado em relação ao tempo de solução das equações pelo MGC. Este percentual elevado se deve ao fato desta malha não possuir um número muito elevado de equações a serem resolvidas, fazendo com que o custo de comunicação tenha uma influência significativa no tempo de solução do MGC, ou seja, para a simulação desta malha, a comunicação aparece como um gargalo.

Para as outras duas malhas, Malha 2 e Malha 3, o tempo de comunicação já não representa um percentual tão significativo em relação ao tempo de solução das equações pelo MGC. Isto porque estas duas malhas possuem um número maior de equações a serem resolvidas, o que faz com que o custo de comunicação não seja representativo.

Uma outra verificação interessante ao comparar as execuções de uma mesma malha nos dois *clusters* é o fato de os valores percentuais do custo de comunicação serem sempre menores no *Cluster* 2 do que no *Cluster* 1. Este comportamento pode ser explicado pelo menor poder computacional do *Cluster* 2 (nós com processador de menor freqüência e memória RAM com menor capacidade de armazenamento) em relação ao *Cluster* 1, o que faz com que o maior tempo gasto na etapa de solução do MGC seja na solução das equações; sendo o custo de comunicação pouco significativo no *Cluster* 2. Já no *Cluster* 1, a comunicação para alguns casos pode ser considerada um gargalo, já que este *cluster* tem um maior poder computacional e resolve mais rapidamente as equações.

Visualizações gráficas da relação entre a comunicação e o número de tarefas são apresentadas nas Figuras 5.29 e 5.30, para *Clusters* 1 e 2.

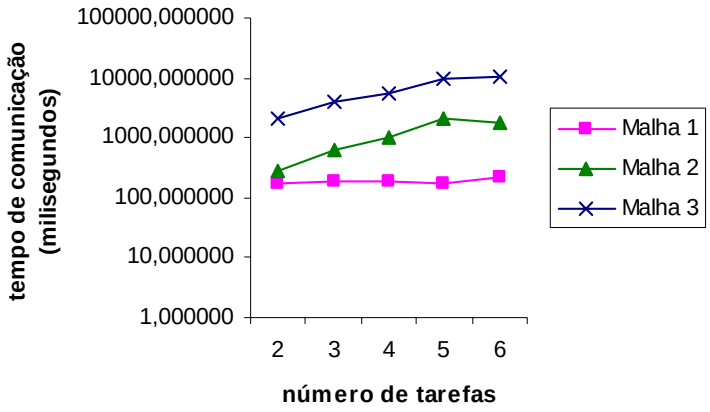

Figura 5.29 – Custo de comunicação para as três malhas no *Cluster* 1.

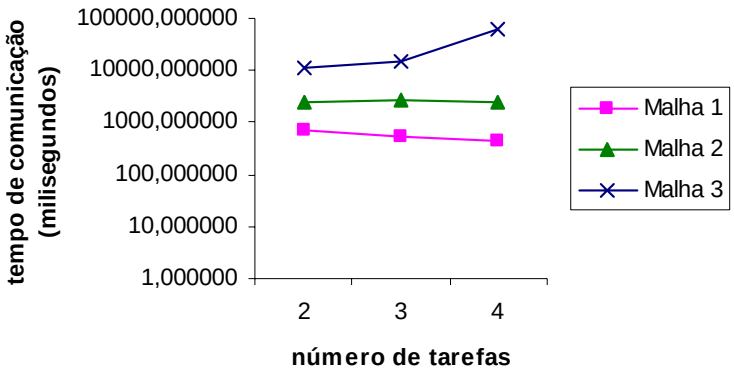

Figura 5.30 – Custo de comunicação para as três malhas no *Cluster* 2.

5.4.4 - Verificação do custo de comunicação entre tarefas em execução em um único nó

Nesta execução, mais de uma tarefa foi alocada ao *node8* do *Cluster* 1 e a malha utilizada foi a Malha 2. A Tabela 5.24 e a Figura 5.31 exibem as medições de tempos gastos com comunicações ponto a ponto (*send* e *receive*).

Tabela 5.24 – Comunicação ponto a ponto no *node8* do *Cluster* 1 para a Malha 2.

Nº de tarefas	Tempo de *send* (milisegundos)	Tempo de *receive* (milisegundos)	Tempo total (milisegundos)
2	17,779120	333,938710	351,717830
4	192,1730851	1.940,730377	2.132,903462
8	314,418839	1.886,912941	2.201,331780

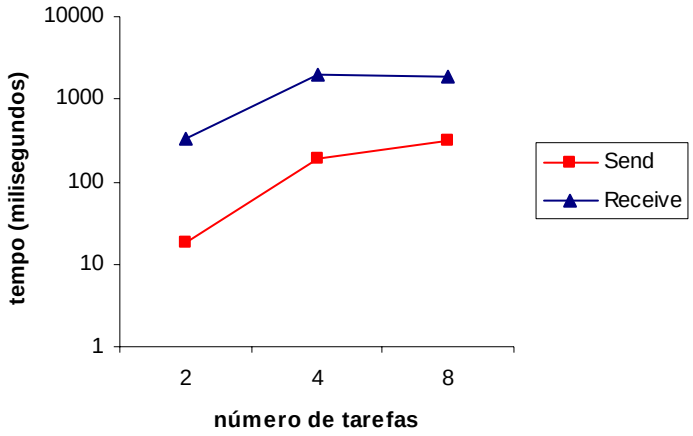

Figura 5.31 – Comunicação ponto a ponto no *node8* do *Cluster* 1 para a Malha 2.

Os tempos de comunicação são acrescidos com o aumento do número de tarefas, mesmo com as tarefas sendo executadas em uma única máquina. Uma observação interessante é o fato de o tempo de *receive* ser sempre maior do que o de *send*. Tal fato ocorre porque o *receive* fica sempre um tempo a mais esperando pelo *send* correspondente, conforme é pode ser observado através das figuras 5.20 e 5.21. A Tarefa 0 ficou em execução até iniciar o *receive*. Como não havia nenhum *send* enviado pela Tarefa 1, a Tarefa 0 fica parada esperando e a Tarefa 1 inicia sua execução realizando o seu *send*. Só depois de a Tarefa 1 ter enviado o *send* é que a Tarefa 0 poderá concluir o seu *receive*.

5.4.5 - Verificação dos tempos de execução no ambiente heterogêneo

Através dos resultados já apresentados e analisados, pode-se perceber que os valores de tempos de execução no *Cluster* 1 são sempre inferiores aos valores de tempos de execução no *Cluster* 2, como era de se esperar pelas características das máquinas constituintes de cada um dos dois *clusters*.

As Tabelas 5.25 e 5.26 e a Figura 5.32 exibem os valores totais dos tempos obtidos a partir da execução da aplicação para as três malhas numa única máquina de cada um dos dois *clusters*. Foram executadas 1, 2, 4 e 8 tarefas em cada uma das máquinas de forma independente.

Tabela 5.25 – Tempos totais de execução de tarefas no *node8* do *Cluster* 1.

Número de tarefas	Malha 1 (milisegundos)	Malha 2 (milisegundos)	Malha 3 (milisegundos)
1	914,586505	22.421,512867	139.417,691645
2	524,020535	9.397,570976	55.553,070543
4	381,571701	6.782,863157	38.216,226022
8	1.786,396291	5.920,422720	24.417,125862

Tabela 5.26 – Tempos totais de execução de tarefas no *node00* do *Cluster* 2.

Número de tarefas	Malha 1 (milisegundos)	Malha 2 (milisegundos)	Malha 3 (milisegundos)
1	6.442,371311	178.694,514591	1.125.734,097360
2	5.277,342147	122.232,137024	740.547,439699
4	4.096,185234	83.423,108431	466.273,914186
8	7.997,566956	59.986,057192	316.657,647988

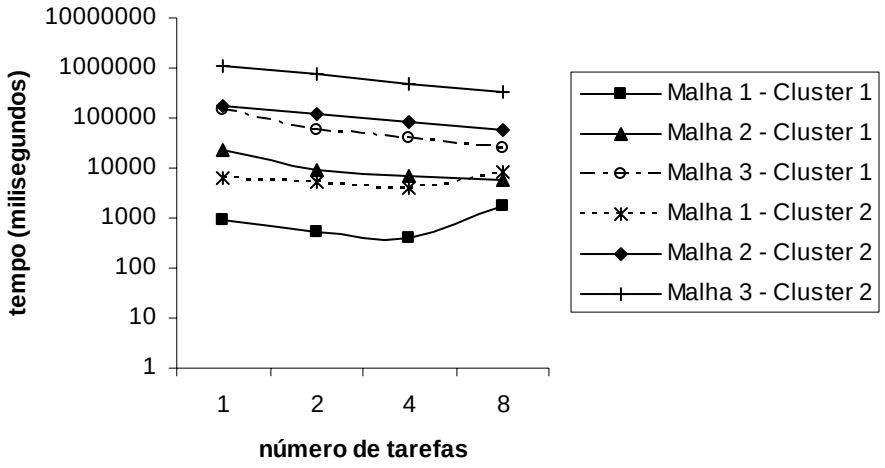

Figura 5.32 – Tempos totais de execução nos *clusters* 1 e 2.

Como pode ser observado nas duas tabelas acima, o tempo gasto no *Cluster* 2 para a execução de 1 tarefa é, em média, 7,7 vezes maior do que o tempo gasto no *Cluster* 1 para a execução de 1 tarefa. Para 2 tarefas, o tempo gasto no *Cluster* 2 é, em média, 12,1 vezes maior do que o

tempo gasto no *Cluster* 1. Para 4 tarefas, este valor é, em média, 11,7 vezes maior e para 8 tarefas, 9,2 vezes maior, em média.

O decréscimo deste último valor se deve ao fato de que a execução da Malha 1 com 8 tarefas, em ambos os *clusters*, apresenta uma queda de desempenho devido à grande influência do custo de comunicação, como explicado anteriormente. Se forem desconsiderados os tempos obtidos para a execução da Malha 1 com 8 tarefas, o tempo gasto no *Cluster* 2 para a execução com 8 tarefas é, em média, 11,5 vezes maior do que o tempo gasto no *Cluster* 1, aproximando-se dos valores anteriores.

Para a caracterização de desempenho da aplicação *benchmark* no ambiente heterogêneo, foram feitas execuções utilizando uma máquina (*node8*) do *Cluster* 1 e uma máquina (*node00*) do *Cluster* 2 interligadas, formando um *cluster* heterogêneo. Variou-se o número de tarefas alocadas a cada máquina. O detalhamento das execuções efetuadas segue abaixo:

- Execução 1: 1 tarefa executando no *node00* e 1 tarefa executando no *node8*.
- Execução 2: 1 tarefa executando no *node00* e 3 tarefas executando no *node8*.
- Execução 3: 1 tarefa executando no *node00* e 7 tarefas executando no *node8*.
- Execução 4: 2 tarefas executando no *node00* e 2 tarefas executando no *node8*.
- Execução 5: 2 tarefas executando no *node00* e 6 tarefas executando no *node8*.
- Execução 6: 3 tarefas executando no *node00* e 1 tarefa executando no *node8*.
- Execução 7: 3 tarefas executando no *node00* e 5 tarefas executando no *node8*.
- Execução 8: 4 tarefas executando no *node00* e 4 tarefas executando no *node8*.
- Execução 9: 5 tarefas executando no *node00* e 3 tarefas executando no *node8*.
- Execução 10: 6 tarefas executando no *node00* e 2 tarefas executando no *node8*.
- Execução 11: 7 tarefas executando no *node00* e 1 tarefa executando no *node8*.

A Tabela 5.27 e as Figuras 5.33 a 5.35 apresentam os valores de tempos, em segundos, obtidos das execuções detalhadas acima.

Tabela 5.27 – Valores dos tempos de execução no *cluster* heterogêneo.

	Malha 1	Malha 2	Malha 3
Execução 1	3,221081	60,15758	372,81553
Execução 2	0,815899	16,81075	89,1673023
Execução 3	1,871412	10,40932	49,8016146
Execução 4	1,616054	34,75037	226,241982
Execução 5	2,905506	17,23698	89,809218
Execução 6	2,736709	59,03364	346,090435
Execução 7	3,026862	23,9888	140,487559
Execução 8	3,673973	30,97071	177,370295
Execução 9	4,933285	39,54501	207,468925
Execução 10	5,189419	46,93692	238,505192
Execução 11	6,54722	54,03075	285,66603

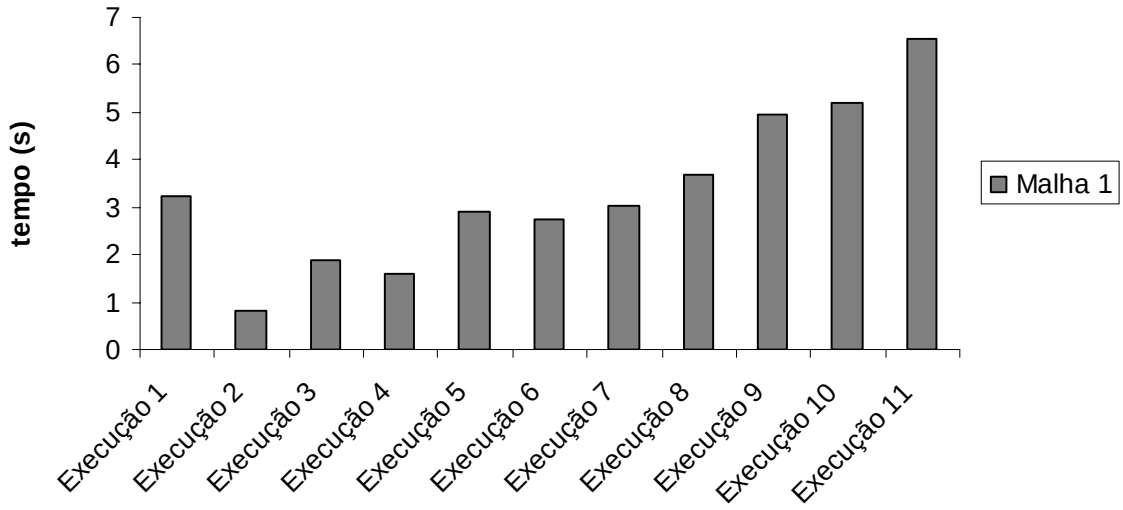

Figura 5.33 – Tempos de execução no *cluster* heterogêneo para a Malha 1.

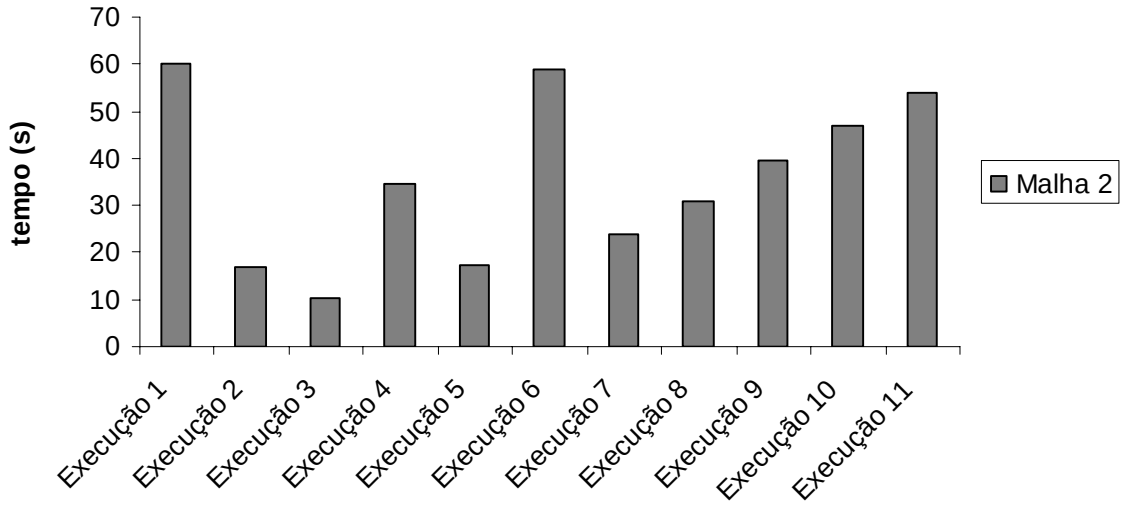

Figura 5.34 – Tempos de execução no *cluster* heterogêneo para a Malha 2.

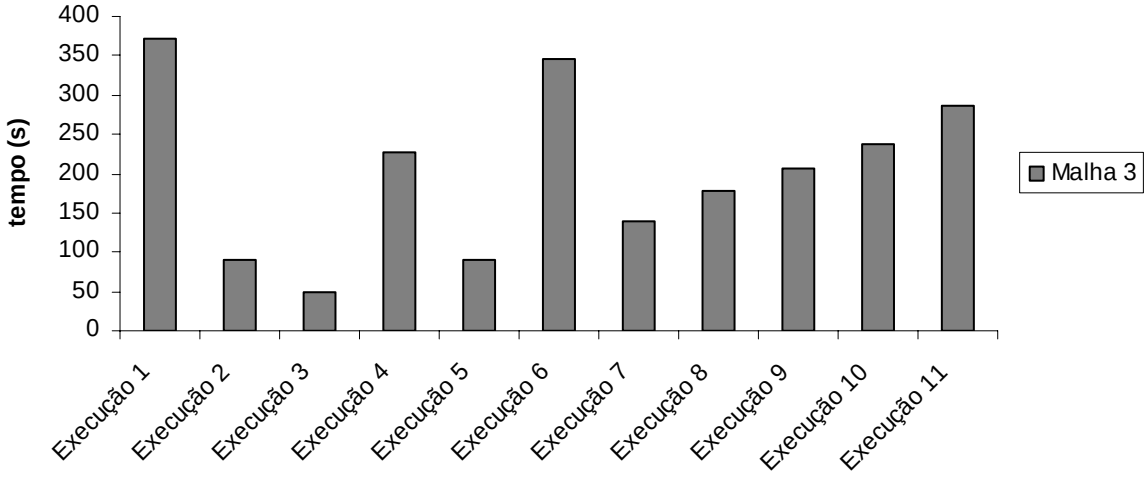

Figura 5.35 – Tempos de execução no *cluster* heterogêneo para a Malha 3.

O menor valor obtido para a Malha 1 no *cluster* heterogêneo foi com a Execução 2 (1 tarefa executando no *node00* e 3 tarefas executando no *node8*) e o maior valor foi com a Execução 11 (7 tarefas executando no *node00* e 1 tarefa executando no *node8*). De modo geral, para esta malha, os melhores valores obtidos foram com as execuções de 4 tarefas no total, distribuídas entre as duas máquinas do ambiente heterogêneo, o que é explicado pelo tamanho pequeno desta malha, conforme já discutido. Com o total de 8 tarefas em execução, a simulação desta malha não apresenta um bom desempenho, a não ser com a Execução 3.

Para as malhas 2 e 3, o menor valor obtido foi com a Execução 3 (1 tarefa executando no *node00* e 7 tarefas executando no *node8*) e o maior valor foi com a Execução 1 (1 tarefa executando no *node00* e 1 tarefa executando no *node8*). O menor valor obtido com a Execução 3 se deve ao fato da menor influência da comunicação com a máquina mais lenta (*node00*), já que somente uma tarefa foi alocada a esta máquina. O pior valor obtido com a Execução 1 é justificado pelos tamanhos das malhas 2 e 3, o que faz com que apenas duas tarefas em execução não sejam suficientes para obter um bom tempo de execução; e também é justificado pela influência da comunicação entre as duas máquinas. Haverá muita comunicação entre as duas tarefas e a máquina mais lenta do ambiente heterogêneo (*node00*) influencia no custo de comunicação.

Pode-se observar que o tamanho da malha que está sendo simulada influencia nos valores dos tempos de execução tanto para os *clusters* homogêneos 1 e 2 quanto para o ambiente heterogêneo. No ambiente heterogêneo, para uma malha que tenha um maior número de equações a serem resolvidas, quanto maior o número de tarefas alocadas à máquina com maior poder de processamento, neste caso o *node8* do *Cluster* 1, melhor o desempenho do *cluster* heterogêneo. Este fato se explica por dois motivos. O primeiro deles é que mesmo havendo custo de comunicação entre tarefas alocadas a um mesmo nó este custo não se apresenta tão relevante para as malhas com um grande número de equações, quando comparado ao custo de comunicação entre o *node8* e o *node00*. O segundo motivo é o próprio número de tarefas em execução. Ao dividir o domínio em maior quantidade de subdomínios cada tarefa fica com um menor número de equações para resolver.

No caso das malhas 2 e 3, o melhor desempenho foi alcançado ao alocar 7 vezes mais tarefas para a máquina do *Cluster* 1 do que para a máquina do *Cluster* 2, estando de acordo com o valor obtido nas Tabelas 5.25 e 5.26 e na Figura 5.31.

As Tabelas 5.28 a 5.30 e a Figura 5.36 apresentam os valores das comparações das execuções de duas tarefas em duas máquinas (uma tarefa alocada a cada um das máquinas) nos *clusters* 1, 2 e no *cluster* heterogêneo, para as três malhas.

Tabela 5.28 – Tempos de execução, em segundos, de duas tarefas em duas máquinas no *Cluster* 1, *Cluster* 2 e no c*luster* heterogêneo, da Malha 1, sendo cada tarefa alocada a uma máquina.

	Cluster 1	*Cluster* heterogêneo	*Cluster* 2
Duas tarefas em execução em duas máquinas	3,22	3,29	5,72

Tabela 5.29 – Tempos de execução, em segundos, de duas tarefas em duas máquinas no *Cluster* 1, *Cluster* 2 e no *cluster* heterogêneo, da Malha 2, sendo cada tarefa alocada a uma máquina.

	Cluster 1	*Cluster* heterogêneo	*Cluster* 2
Duas tarefas em execução em duas máquinas	9,15	60,15	62,68

Tabela 5.30 – Tempos de execução, em segundos, de duas tarefas em duas máquinas no *Cluster* 1, *Cluster* 2 e no *cluster* heterogêneo, da Malha 3, sendo cada tarefa alocada a uma máquina.

	Cluster 1	*Cluster* heterogêneo	*Cluster* 2
Duas tarefas em execução em duas máquinas	54,28	382,81	389,48

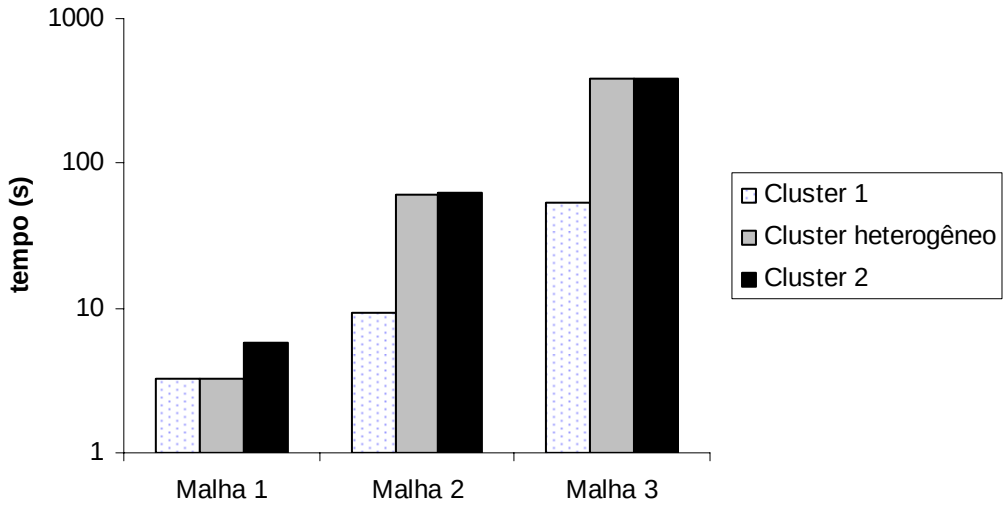

Figura 5.36 – Tempos de execução nos *clusters* 1 e 2 e no *cluster* heterogêneo para as malhas 1, 2 e 3.

Comparando as execuções de duas tarefas alocadas a duas máquinas do *Cluster* 1 e duas tarefas alocadas a duas máquinas do *cluster* heterogêneo, percebe-se que os tempos apresentam valores superiores quando executados nas máquinas do *cluster* heterogêneo, ou seja, há uma degradação no desempenho, como esperado. Isto porque as duas máquinas do *Cluster* 1 oferecem maior capacidade de processamento em relação a uma das máquinas do *cluster* heterogêneo. Assim, não é lucrativo, em termos de desempenho, tornar um *cluster* heterogêneo com a adição de máquinas mais lentas, pois estas farão com que haja perda de desempenho.

Comparando as execuções de duas tarefas alocadas a duas máquinas do *Cluster* 2 e duas tarefas alocadas a duas máquinas do *cluster* heterogêneo, ocorre uma diminuição nos tempos de execução para o *cluster* heterogêneo, ou seja, uma melhora no desempenho. Este fato se deve ao ganho no poder de processamento obtido com a adição da máquina do *Cluster* 2 ao *cluster* heterogêneo. A diminuição nos tempos de execução é muito pequena devido ao atraso na comunicação. O modo como o Metis® efetua o particionamento do domínio influencia na quantidade de comunicação a ser executada. Espera-se que haja um ganho maior de desempenho à medida que mais máquinas do *Cluster* 1 sejam adicionadas ao *cluster* heterogêneo.

Conforme será mostrado em seguida, o Metis® divide o domínio de maneira que cada subdomínio resultante da partição fique com aproximadamente o mesmo número de elementos e com o mesmo número de graus de liberdade compartilhados. Dessa forma, todas as tarefas efetuarão aproximadamente a mesma quantidade de comunicação. Por este motivo, quando somente duas tarefas estão em execução no *cluster* heterogêneo, haverá muita comunicação entre estas tarefas e a máquina com menor poder computacional vai causar um custo de comunicação alto, degradando o desempenho.

As Tabelas 5.31, 5.32 e 5.33 apresentam as quantidades de elementos e as quantidades de graus de liberdade compartilhados entre tarefas após a etapa de particionamento.

Tabela 5.31 - Execução da Malha 3 com duas tarefas

	Número de elementos	Número de gl's compartilhados
Tarefa 0	4566	594
Tarefa 1	4560	594

Tabela 5.32 – Execução da Malha 3 com quatro tarefas

	Número de elementos	Número de gl's compartilhados
Tarefa 0	2243	546
Tarefa 1	2329	666
Tarefa 2	2349	684
Tarefa 3	2205	582

Tabela 5.33 – Execução da Malha 3 com oito tarefas

	Número de elementos	Número de gl's compartilhados
Tarefa 0	1162	654
Tarefa 1	1197	462
Tarefa 2	1094	741
Tarefa 3	1117	432
Tarefa 4	1049	459
Tarefa 5	1194	486
Tarefa 6	1127	747
Tarefa 7	1186	402

Pode-se perceber que o número de graus compartilhados é aproximadamente o mesmo para cada tarefa. Comparando os valores das Tabelas 5.31, 5.32 e 5.33, é possível verificar o aumento do número de graus de liberdade compartilhados com o aumento do número de tarefas, justificando o aumento da comunicação com o aumento do número de tarefas, conforme mostrado anteriormente. Ainda, verifica-se a diminuição do número de elementos para cada tarefa com o aumento do número de tarefas.

5.4.6 – *Speedup* obtido nas execuções no *cluster* heterogêneo

Para calcular os valores de *speedup* para as Malhas 1, 2 e 3, foram tomados os valores dos tempos de execução com 1 tarefa de cada uma das máquinas do *cluster* heterogêno. Estes valores foram divididos pelos valores dos tempos das Execuções 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 e 11 quando se apresentavam menores do que os valores dos tempos de execução com 1 tarefa. Este *speedup* calculado e aqui utilizado é o *speedup* relativo.

Com o objetivo de facilitar a leitura, o detalhamento das execuções efetuadas é repetido aqui:

- Execução 1: 1 tarefa executando no *node00* e 1 tarefa executando no *node8*.
- Execução 2: 1 tarefa executando no *node00* e 3 tarefas executando no *node8*.
- Execução 3: 1 tarefa executando no *node00* e 7 tarefas executando no *node8*.
- Execução 4: 2 tarefas executando no *node00* e 2 tarefas executando no *node8*.

- Execução 5: 2 tarefas executando no *node00* e 6 tarefas executando no *node8*.
- Execução 6: 3 tarefas executando no *node00* e 1 tarefa executando no *node8*.
- Execução 7: 3 tarefas executando no *node00* e 5 tarefas executando no *node8*.
- Execução 8: 4 tarefas executando no *node00* e 4 tarefas executando no *node8*.
- Execução 9: 5 tarefas executando no *node00* e 3 tarefas executando no *node8*.
- Execução 10: 6 tarefas executando no *node00* e 2 tarefas executando no *node8*.
- Execução 11: 7 tarefas executando no *node00* e 1 tarefa executando no *node8*.

Conforme apresentado nas Figuras 5.37 a 5.39, houve um ganho de desempenho para quase todas as execuções (acima detalhadas) efetuadas no *cluster* heterogêneo.

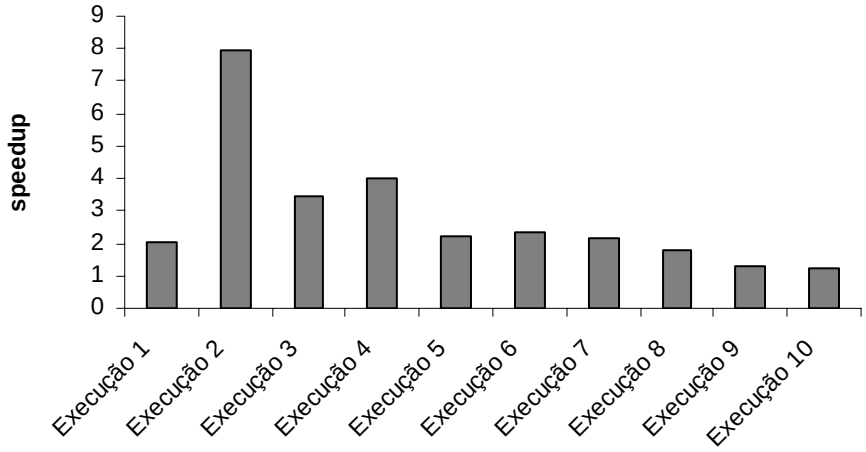

Figura 5.37 – *Speedup* da Malha 1 no *cluster* heterogêneo.

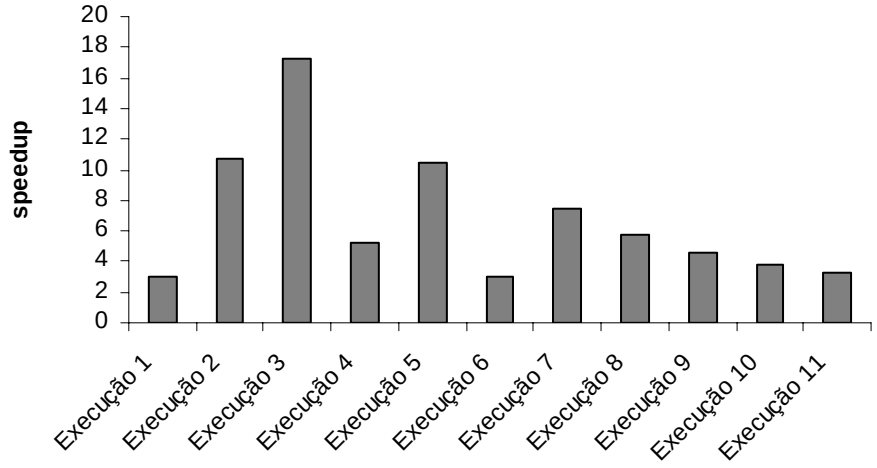

Figura 5.38 – *Speedup* da Malha 2 no *cluster* heterogêneo.

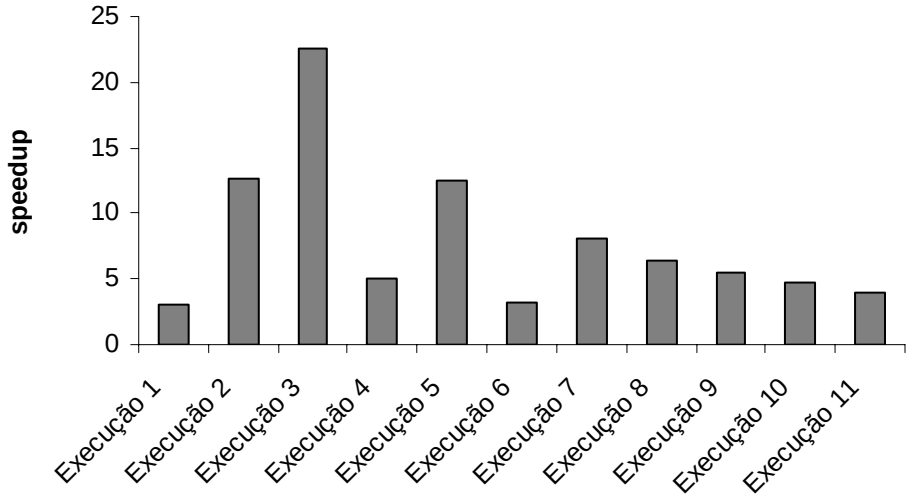

Figura 5.39 – *Speedup* da Malha 3 no *cluster* heterogêneo.

Para as Malhas 2 e 3, o maior valor de *speedup* obtido foi na Execução 3 como esperado, uma vez que esta execução apresentou os menores tempos.

5.4.7 – Eficiência obtida nas execuções no *cluster* heterogêneo

Os valores de eficiência obtida com as execuções das Malhas 1, 2 e 3 no *cluster* heterogêneo são mostrados nas Figuras 5.40 a 5.42. A eficiência é a relação entre o *speedup* e o número de tarefas utilizadas na execução.

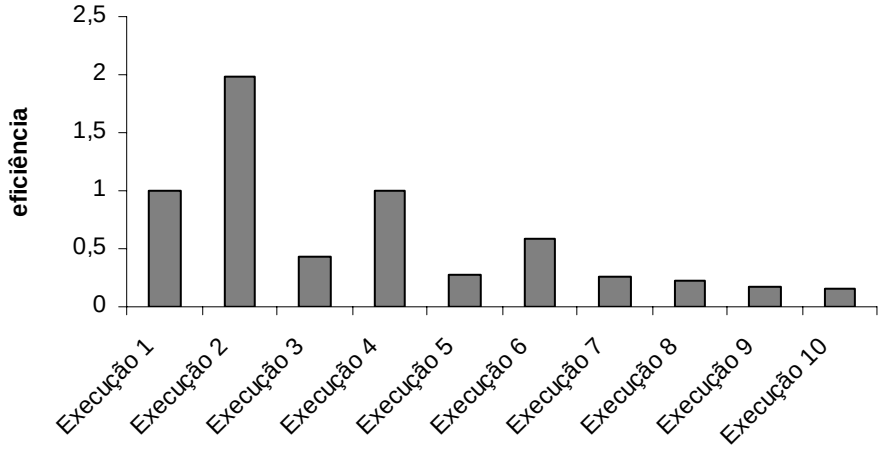

Figura 5.40 – Eficiência da Malha 1 no *cluster* heterogêneo.

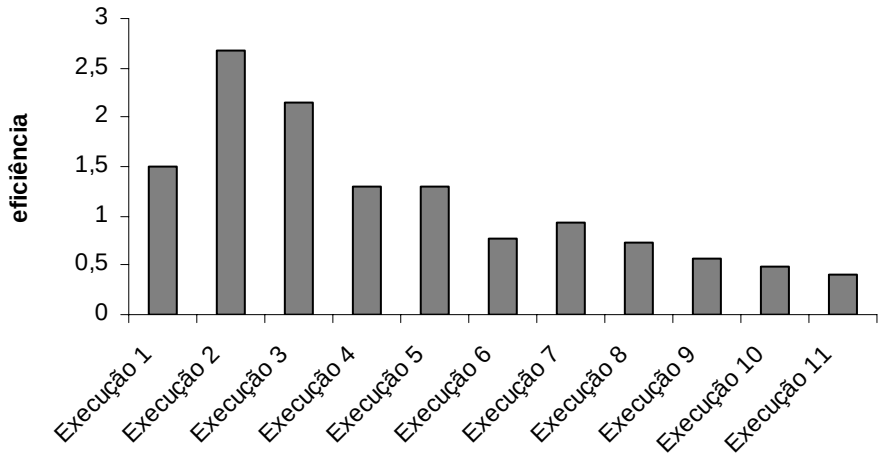

Figura 5.41 – Eficiência da Malha 2 no *cluster* heterogêneo.

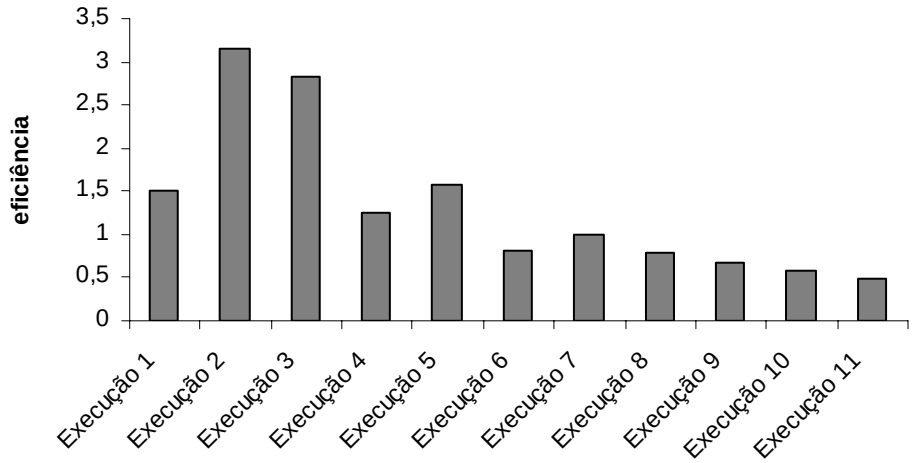

Figura 5.42 – Eficiência da Malha 3 no *cluster* heterogêneo.

6 – CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho são apresentados resultados que caracterizaram o desempenho de uma aplicação paralela para simulação do método dos elementos finitos aplicado à elasticidade linear, para a solução de problemas estruturais, que utiliza o método dos gradientes conjugados para a solução do sistema de equações lineares.

Para captar os tempos das medições neste trabalho, foi utilizada a instrução *assembly rdtsc*, que retorna o contador de *time stamp* em ciclos de *clock*. Isto porque pôde ser observado, através de execuções em três máquinas distintas, que o número de ciclos de *clock* gastos para a execução da instrução *assembly rdtsc* é consideravelmente menor quando comparado aos números de ciclos de *clock* gastos pelas funções *rdtsc( )*, *gettimeofday( )* e *MPI_Wtime ( )*.

Os valores resultantes das medidas tomadas dentro da aplicação e aqui apresentados foram produzidos por execuções paralelas da aplicação, para simulação de três malhas, em dois *clusters* homogêneos de PCs e num ambiente heterogêneo, sendo este formado a partir da interligação de máquinas dos *clusters* homogêneos. Para uma melhor caracterização de desempenho foram exibidos os valores dos tempos de execução das três principais etapas do programa: particionamento, montagem da matriz e solução do método dos gradientes conjugados e também foram apresentados os valores totais de tempo de execução, os valores de *speedup* e eficiência.

Foi possível observar que há um ganho de desempenho com a execução paralela do código. Na maioria das simulações efetuadas o tempo total de execução decresce com o aumento do número de tarefas, mesmo havendo um maior custo de comunicação. Neste caso a maior distribuição do domínio compensa o tempo de comunicação. Somente para o caso da simulação com a Malha 1 é que a execução com 8 tarefas apresentou um acréscimo no tempo em relação à execução com 4 tarefas. Este comportamento demonstrou que o ganho de desempenho obtido com a partição do domínio tende a saturar num determinado número de tarefas e que este número depende do tamanho da malha. Quando a malha possui um número pequeno de equações a serem solucionadas, o custo de comunicação acaba superando mais rapidamente o ganho obtido com o particionamento.

A etapa de solução do método dos gradientes conjugados foi medida mais detalhadamente para que fosse verificado o custo de comunicação entre as tarefas que compartilham graus de liberdade e foi constatado que, percentualmente, este custo tem maior influência quando a execução ocorre no *Cluster* 1 do que quando ocorre no *Cluster* 2. Esta maior influência pode ser explicada pelo menor poder computacional do *Cluster* 2 (nós com processador de menor freqüência e memória RAM com menor capacidade de armazenamento) em relação ao *Cluster* 1, o que faz com que o maior tempo gasto na etapa de solução do MGC seja na solução das equações; sendo o custo de comunicação pouco significativo no *Cluster* 2. Já no *Cluster* 1, a comunicação para alguns casos pôde ser considerada um gargalo, já que este *cluster* tem um maior poder computacional e resolve mais rapidamente as equações.

Verificou-se que os valores de tempos de execução no *Cluster* 1 são sempre inferiores aos valores de tempos de execução no *Cluster* 2, como era de se esperar pelas características das máquinas constituintes de cada um dos dois *clusters*.

As medidas obtidas com as execuções no ambiente heterogêneo demonstraram um ganho de desempenho quando comparadas às execuções realizadas no *Cluster* 2. Ficou evidente que a utilização de *clusters* heterogêneos é uma alternativa viável, econômica e eficiente para a solução de um problema de elasticidade utilizando o método dos elementos finitos. Os valores de *speedup* obtidos confirmam o ganho de desempenho alcançado.

Como sugestões para trabalhos futuros é possível citar a implementação de uma modelagem que permita predizer o desempenho em ambientes heterogêneos, através da utilização de balanceamento de carga, aplicando a mesma metodologia apresentada neste trabalho para um número maior de máquinas no ambiente heterogêneo.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Hwang, K., Xu, Z., “Scalable Parallel Computig: Technology, Architectures, Programming”, McGraw-Hill, 1998.

[2] Geist, A., Beguelin, A., Dongarra, J., Jang, W., Manchek, R., Sunderam, V., “PVM: Parallel Virtual Machine A Users’ Guide and Tutorial for Networked Parallel Computing”, MIT Press, Cambridge, Massachusetts, 1994.

[3] “Parallel Virtual Machine”, http://www.csm.ornl.gov/pvm/, disponível em agosto de 2006.

[4] Gropp, W., Lusk, E., Skjellum, A., “Using MPI: Portable Parallel Programming with Message-Passing Interface”, MIT Press, Cambridge, Massachusetts, 1999.

[5] “The Message Passing Interface (MPI) Standard”, http://www-unix.mcs.anl.gov/mpi/, disponível em agosto de 2006.

[6] Snir, M., Otto, S., Huss_Lederman, S., Walker, D., Dongarra, J., “MPI: The Complete Reference”, MIT Press, Cambridge, Massachusetts, 1996.

[7] Pacheco, P.S., “Parallel programming with MPI”, Morgan Kaufmann, San Francisco, 1997.

[8] Villa Verde, F. R., “Solução Paralela em *Clusters* de PCs de um Código de Elementos Finitos Aplicado à Elasticidade Linear”, Dissertação de Mestrado em Engenharia Mecânica, Publicação ENM.DM - 80A/04, Departamento de Engenharia Mecânica, Universidade de Brasília, DF, 2004.

[9] Song, A., Peng, Q., Hu, B., “Research-on Parallel Computing Model of BSP in the NOWS Adapt Environment”, Int. Conf. Neural Networks and Signal Processing, IEEE, China, Volume 1, 2003.

[10] Fadlallah, G., Lavoie, M., Dessaint, “Parallel computing environments and methods”, Parallel Computing in Electrical Engineering, 2000, PARELEC 2000, Proceedings of International Conference IEEE, 2000.

[11] Dantas, M., “Computação Distribuída de Alto Desempenho – Redes, Clusters e Grids Computacionais”, Editora Axcel, 2005.

[12] Bal, H. E., Steiner, J. G., Tanenbaum, A. S., “Programming Languages for Distributed Computing Systems”, ACM Computing Surveys, Vol. 21, No. 3, September 1989.

[13] El-Rewini, H., Lewis, T. G., “Distributed and Parallel Computing”, Manning, Greenwich, CT, 1997.

[14] Bustard, D. W., “Concepts of Concurrent Programming”, Carnegie-Mellon University, 1990.

[15] Andrews, G. R., Schneider, F. B., “Concepts and Notations for Concurrent Programming”, ACM, Computing Surveys, Vol. 15, N^{o} I, March 1983.

[16] Tanembaum, A. S., “Sistemas Operacionais Modernos”, Rio de Janeiro, Prentice/Hall, 1995.

[17] Miklosko, J., Kotov V. E., “Algorithms, Software and Hardware of Parallel Computers”, Springler-Verlag, 1984.

[18] Culler, D. E., Singh, J. P., “Parallel Computer Architecture: A hardware/Software Approach”, Morgan Kauffman Publishers, 1999.

[19] Foster, I., “Designing and Building Parallel Programs”, Addison-Wesley, 1995.

[20] Flynn, M., “Some Computer Organizations and Their Effectiveness”, IEEE Transaction on Computers, Vol. C-21, 1972.

[21] Flynn, M. J., Rudd, K. W., “Parallel Architectures”, ACM Computing Surveys, Vol. 28, No. 1, 1996.

[22] Johnson, E. E., “Completing an MIMD Multiprocessor Taxonomy”, In: ACM SIGARCH Computer Architecture News, Volume 16, Issue 3, 1998.

[23] Fosdick, L. D., Schauble, C. J. C., Jessup, E. R., “Tutorial: General Architecture of Vector Processors”, HPSC Group of the University of Colorado, 1999. Disponível na Internet pelo site: http://www-ugrad.cs.colorado.edu/~csci4576/VectorArch/VectorArch.html

[24] Konchady, M., Sood, A., Schopf, P. S., “Implementation and Performance Evaluation of a Parallel Ocean Model”, In: Parallel Computing, 181-203, 1998.

[25] Sterling, T., “An Introduction to PC Clusters for High Performance Computing”, The International Journal of High Performance Computing Applications, Volume 15, No. 2, Summer 2001, pp. 92-101, Sage Publications, Inc.

[26] Gropp W., Lusk, E., Sterling, T., “Beowulf Cluster Computing with Linux”, Second Edition, The Mit Press, 2003.

[27] Sloan, J. D., “High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI”, Publisher: O'Reilly, 2004.

[28] http://www.myri.com/, disponível em agosto de 2006.

[29]http://www.quadrics.com/quadrics/QuadricsHome.nsf/DisplayPages/Homepage, disponível em agosto de 2006.

[30] http://www.dolphinics.com/, disponível em agosto de 2006.

[31] Beowulf Project Overview. Disponível na Internet pelo site: http://www.beowulf.org/

[32] Montante, R., Department of Mathematics, Computer Science, and Statistics, Bloomsburg University, CCSC: Northeastern Conference, "Beowulf and Linux – An Integrated Project", 2002.

[33] Dongarra, J., Sterling, T., Simon, H., Strohmaier, E., "High-Performance Computing: Clusters, Constellations, MPPs and Future Directions", Perspectives in Computational Science, IEEE, 2005.

[34] http://gid.cimne.upc.es, disponível em maio de 2005.

[35] http://www.top500.org, disponível em agosto de 2006.

[36] Banawan, S. A., Zeidat, N. M., "A Comparative Study of Load Sharing in Heterogeneous Multicomputer Systems", 1992 IEEE.

[37] Chronopoulos, A. T., Andonie, R., Benche, M., Grosu, D., "A Class of Loop Self-Scheduling for Heterogeneous Clusters", Proceedings of the 2001 IEEE International Conference on Cluster Computing.

[38] Lastovetsky A., Reddy, R., "On performance Analysis of heterogeneous parallel algorithms", Parallel Computing 30 (2004) 1195–1216, Elsevier.

[39] Hummelt, S. F., Schmidt, J., Uma, R. N., Wein, J., "Load-Sharing in Heterogeneous Systems via Weighted Factoring", SPAA'96, Padua, Italy, 1996 ACM.

[40] Parhami, B., "Introduction to Parallel Processing: Algorithms and Architectures", Kluwer Academic Publishers, 1999.

[41] Fisher, A. L., Gross, T., "Teaching Empirical Performance Analysis of Parallel Programs", ACM, 1992.

[42] Xian-He Sun, **"**Performance Prediction: A Case Study Using a Scalable Shared-Virtual-Memory Machine", IEEE Parallel & Distributed Technology, Winter 1996.

[43] Calzarossa, M., Massari, L., Tessera, D., "A methodology towards automatic performance analysis of parallel applications", Parallel Computing 30 (2004), Elsevier.

[44] Hollingsworth, J. K., Lumpp. J. E., Miller, B. P., Computer Sciences Department, University of Wisconsin, Madison WI 53706, USA, Department of Electrical Engineering, University of Kentucky, Lexington, KY 40506-0046, USA, "Techniques for Performance Measurement of Parallel Programs".

[45] Xian-He, S., Ni, L. M., Supercomputing 90, Proceedings of 12-16, Nov. 1990, Page(s):324 – 333; "Another view on parallel speedup".

[46] Amdahl, G. M., "Validity of Single Processor Approach to Achieving Large Scale Computer Capabilities", AFIPS Conference Proceedings 30, 1967.

[47] Gustafson, J. L., "Reevaluating Amdahl's Law", Communications of ACM, Volume 31, Issue 5, 1988.

[48] Xingfu Wu, Wei Li, ELSEVIER, Journal of Systems Architecture 44, 1998, "Performance models for scalable cluster computing".

[49] Burkhart, H., Millen, R, IEEE Transactions on Computers, Vol. 38, No. 5, May 1989, "Performance-Measurement Tools in a Multiprocessor Environment".

[50] Krishnaprasad, S., Mathematical, Computing, and Information Sciences, Jacksonville State University, Jacksonville, AL 3626. JCSC 17, 2 (December 2001), CCSC: Southeastern Conference, "Uses and Abuses of Amdahl's Law".

[51] Shi, Y., Computer and Information Sciences department, Temple University, (MS:38-24), Philadelphia, PA 19122, "Reevaluating Amdahl's Law and Gustafson's Law", Disponível na Internet pelo site: http://joda.cis.temple.edu/~shi/docs/amdahl/amdahl.html

[52] Ezzat, Ahmed K., "Load Balancing in Nest: A Network of Workstations"

[53] Meyer, B. H., Pieper, J. J., Paul, J. M., Nelson, J. E., Pieper, S. M., Rowe, A. G., "Power-Performance Simulation and Design Strategies for Single-Chip Heterogeneous Multiprocessors", IEEE TRANSACTIONS ON COMPUTERS, Vol. 54, NO. 6, June 2005.

[54] Azevedo, A. F. M., "Método dos Elementos Finitos", 1 Edição, Abril 2003. Disponível na Internet: http://civil.fe.up.pt/pub/apoio/ano5/mnae/Livro_MEF_AA.htm

[55] Gurtin, M.E., "An Introduction to Continuum Mechanics", Mathematics in Science and Engineering, Vol. 158.Academic Press, New York, 1981.

[56] Hughes, T.J.R., "The Finite Element Method - Linear Static and Dynamic Finite Element Analysis", Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[57] Villa Verde, F. R., Ferreira, R. R., Pfitscher, G. H., "Solução Paralela em Agregado de PCs de um Código de Elementos Finitos Aplicado à Elasticidade Linear", I Congresso Sul Catarinense de Computação, 2005.

[58] Villa Verde, F. R., Pfitscher, G. H., Viana, Dianne M., Ferrreira, Roberta R., "Simulação de Problemas Estruturais Aplicados à Elasticidade Linear em Agregados de PCs", IV Congresso Nacional de Engenharia Mecânica, Agosto 2006, Recife –PE.

[59] Bathe, K. J, “Finite Element Procedures”, Prentice-Hall, Upper Saddle River, N.J.

[60] Felippa, C. A., “Introduction to Finite Element Methods”, Department of Aerospace Engineering Sciences and Center for Aerospace Structures - University of Colorado Boulder, Colorado 80309-429, USA - Last updated Fall 2004.

[61] Shewchuk, J. R., “An Introduction to the Conjugate Gradient Method Without the Agonizing Pain”, Carnegie Mellon University, Pittsburgh, PA, 1994.

[62] Press, W. H., Teukolsky, S. A., Vetterling, W. T., Flannery, B.P., “Numerical Recipes in C: The Art of Scientific Computing”, Second Edition, Cambridge University Press, 1992.

[63] White, R. E. “Computational Modeling with Methods and Analysis”, CRC Press, 2003.

[Andersen, P. H., Antonio, J. K.] Andersen, P. H., Antonio, J. K., Department of Computer Science, Texas Tech University, Lubbock, “Implementation and Utilization of a Heterogeneous Multicomputer Cluster for the Study of Load Balancing Strategies”.

[64] Hageman, L. A., Young, D. M., “Applied Iterative Methods”, Academic Press, New York, 1981.

[65] Hestenes, M. R., Stiefel, E., “Methods of conjugate gradients for solving linear systems.”, Journal of Research of the National Bureau of Standards 49 (1952), 409–436.

[66] Buchanan, J.L., Turner, P.R., “Numerical Methods and Analysis, McGraw-Hill, New York, 1992.

[67] Jimack, P.K., Touheed, N., “An Introduction to MPI for Computational Mechanics”. In: Parallel and Distributed Processing for Computational Mechanics Systems and Tools, ed. B.H.V. Topping and L. Lamner (Saxe-Coburg Pub.), 24-45, 1999.

[68] 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. Lamner (Saxe-Coburg Pub.), 15-38, 2000.

[69] Karypis, G., Kumar, V., “METIS – A Software Package for Partitioning Unstructured Graph, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices”, Manual, University of Minnesota, Department of Computer Science, 1998.

[70] Jimack, P. K., Hodgson, D. C., “Parallel Preconditioners Based Upon Domain Decomposition”. In: Parallel and Distributed Processing for Computational Mechanics: Systems and Tools, ed. B.H.V. Topping (Saxe-Coburg Publications), 207-223, 1999.

[71] Hughes, T.J.R., “The Finite Element Method. Linear Static and Dynamic Finite Element Analysis”, Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[72] Dickinson, J. K., Forsyth, P. A., "Preconditioned conjugate gradient methods for three-dimensional linear elasticity". In: *International Journal for numerical methods in engineering*. 37, 2211-2234, 1994.

[73] Intel Corporation, “Using the RDTSC Instruction for Performance Monitoring”, 1997.

APÊNDICE

APÊNDICE A – EXECUÇÃO DO CÓDIGO EM MPI

1. Compilar o código

Ir à pasta onde estão salvos os fontes (corpo_programa.c, elemento_trilinear.c, leitura_dados.c, funções.c, particionar.c, estruturas.h e def.h) e os arquivos de entrada (entrada_tri.txt, contorno_tri.txt, forças_tri.txt, nodes_tri.txt e gcoord_tri.txt).

Dentro da pasta, digitar o comando de compilação do código abaixo:

mpicc corpo_programa.c leitura_dados.c particionar.c funções.c elemento_trilinear.c –o corpo_programa –I/usr/local/metis-4.0/Lib –lm –lmetis

Obs.1: Para o exemplo de compilação (descrito acima) supõe-se que o metis-4.0 esteja instalado em /usr/local

2. Executar

Para executar a aplicação, deve-se digitar o comando *mpirun*.

2.1 – Execução utilizando o argumento -np
Se desejar que o próprio MPI efetue a alocação das tarefas aos nós, basta utilizar o argumento –np <np>, conforme o exemplo abaixo:

mpirun –np <np> corpo_programa

onde <np> é o número de processos que se deseja criar.

2.2 – Execução utilizando o argumento -map:
Este argumento permite a especificação de qual *rank* executará em qual máquina.

mpirun –map <list> corpo_programa

onde <list> representa a lista dos nomes das máquinas que participarão da execução e devem estar separados por “:”

Exemplo: mpirun –map node8:node7:node6 corpo_programa
Neste exemplo, serão executadas três tarefas e a tarefa de *rank* 0 será executada no *node8*; a tarefa de *rank* 1 executada no *node7* e a tarefa de *rank* 2 será executada no *node6*.