Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Avaliação de desempenho de método para a resolução da evolução temporal de sistemas auto-gravitantes em dois paradigmas de programação paralela: troca de mensagens e memória compartilhada

Lorena Brasil Cirilo Passos

Orientador Prof. Dr. Gerson Henrique Pfitscher

Brasília
2006

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Lorena Brasil Cirilo Passos

Avaliação de desempenho de método para a resolução da evolução temporal de sistemas auto-gravitantes em dois paradigmas de programação paralela: troca de mensagens e memória compartilhada

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre.
Programa de Pós-graduação em Informática
Departamento de Ciência da Computação
Universidade de Brasília

Orientador: Prof. Dr. Gerson Henrique Pfistcher

Brasília, 07 de dezembro de 2006

Universidade de Brasília ─ UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenadora: Prof[a] Dr[a] Alba Cristina Magalhães Alves de Melo

Banca examinadora composta por:

Prof. Dr. Gerson Henrique Pfitscher (Orientador) ─ CIC/UnB
Prof[a] Dr[a] Alba Cristina Magalhães Alves de Melo ─ CIC/UnB
Prof. Dr. Tarcísio Marciano da Rocha Filho ─ IF/UnB

CIP – CATALOGAÇÃO INTERNACIONAL DA PUBLICAÇÃO

Lorena Brasil Cirilo Passos.

Avaliação de desempenho de método para a resolução da evolução temporal de sistemas auto-gravitantes em dois paradigmas de programação paralela: troca de mensagens e memória compartilhada/ Lorena Brasil Cirilo Passos. Brasília : UnB, 2006.
91 p. : il. ; 29,5 cm.

Tese (Mestre) ─ Universidade de Brasília, Brasília, 2006.

1. Programação Paralela, 2. Paradigmas de Programação, 3. Troca de Mensagens, 4. Memória Compartilhada Distribuída, 5. Avaliação de Desempenho, 6. Integrador Simplético

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro ─ Asa Norte
CEP 70910 - 900
Brasília ─ DF ─ Brasil

DEDICATÓRIA

Dedico este trabalho

Ao meu marido José Nelson, um companheiro de verdade.

AGRADECIMENTOS

Agradeço à minha família e amigos por compreender minhas ausências em vários momentos significativos e, além disso, incentivar a conclusão desta etapa em minha vida.

Aos amigos do mestrado: Daniela, Roberto, Alê!, Tânia, José Geraldo, Edward, José Nelson, Marcelo Sousa e Rodolfo pela ajuda mútua em momentos bastante críticos.

Ao professor Tarcísio Marciano por oferecer a oportunidade de paralelização do código seqüencial do integrador simplético e por apoiar o processo de desenvolvimento.

E enfim, ao meu orientador, professor Gerson Pfitscher, pois com seu bom-humor e paciência me ensinou, dentre outras coisas, a não levar a vida sempre tão a sério. :o)

RESUMO

Nesta dissertação, é apresentada a avaliação de desempenho de uma implementação paralela de um algoritmo seqüencial do integrador simplético para simular a evolução temporal de sistemas auto-gravitantes. Este algoritmo foi paralelizado e posteriormente implementado na linguagem C, utilizando-se dois paradigmas de programação paralela: a troca de mensagens empregando-se a biblioteca MPICH 1.2.6 e a memória compartilhada distribuída com o *middleware* JIAJIA.

Um *cluster* homogêneo de PCs foi o ambiente em que os testes de execução dos programas foram realizados. Um ambiente heterogêneo também foi utilizado para a realização de medidas de desempenho com um balanceamento empírico de carga, uma vez que a montagem deste tipo de sistema paralelo é prática freqüente entre usuários que necessitam de um maior poder computacional.

Para quantificar o desempenho da execução paralela das duas implementações distintas, foram realizados as medições dos tempos de execução e os cálculos dos *speedups* obtidos. Para mensurar o tempo de execução, foi inserida em cada um dos códigos-fonte a instrução *assembly* rdtsc que fornece ciclos de *clock* contabilizados em um registrador de *hardware*. Para o caso da implementação MPI, também foram realizadas medições de tempo de execução por meio da porta paralela utilizando-se a ferramenta PM^2P.

ABSTRACT

In this work it is presented the performance evaluation of a parallel implementation for the sympletic integrator to simulate the temporal evolution of a self-gravitating system. The algorithm of the sympletic integrator was parallelized and the source code was written in the C programming language. Two parallel programming paradigms were employed: message passing, using the MPICH 1.2.6 library specification, and distributed shared memory, using the JIAJIA middleware.

A homogeneous cluster of PCs was used to run the program tests. Due to the fact that users that need greater computational power tend to build heterogeneous computational environments, we also used a heterogeneous parallel system to take the performance measures using an empirical load balancing.

To quantify the parallel execution performance of the programs, execution time measures were taken and the speedups achieved were calculated. To measure the execution time, it was inserted into the source codes the assembly instruction rdtsc, which counts the clock cycles in hardware register. For the MPI implementation version, execution time measures made by the parallel port were also taken using a tool called PM^2P.

SUMÁRIO

RESUMO vi

ABSTRACT vii

LISTA DE FIGURAS x

LISTA DE TABELAS xi

1. INTRODUÇÃO 12

2. COMPUTAÇÃO DE ALTO DESEMPENHO 14
   2.1 - CLASSIFICAÇÃO DOS SITEMAS DE COMPUTAÇÃO DE ALTO DESEMPENHO 16
   2.2 – MODELO PARA PROGRAMAÇÃO EM SISTEMAS PARALELOS E DISTRIBUÍDOS 19

3. AVALIAÇÃO DE DESEMPENHO 25
   3.1 - SPEEDUP 26
      3.1.1 – Lei de Amdahl 26
      3.1.2 – Lei de Gustafson-Barsis 27
   3.2 – INSTRUMENTAÇÃO DE CÓDIGO 32
   3.3 – *TRACING* DE CÓDIGO 33

4. TROCA DE MENSAGENS 35
   4.1 - MPI 36
   4.2 - PVM 36
   4.3 – COMPARAÇÃO MPI E PVM 37

5. MEMÓRIA COMPARTILHADA DISTRIBUÍDA 39
   5.1 – JIAJIA 42

6. *CLUSTERS* DE COMPUTADORES 49
   6.1 – *CLUSTERS* BEOWULF 50

6.2 – CLASSIFICAÇÃO DOS *CLUSTERS* 51
6.3 – TENDÊNCIA DE UTILIZAÇÃO DOS *CLUSTERS* 52

7. ALGORITMO PARA SIMULAÇÃO DA EVOLUÇÃO TEMPORAL DE SISTEMAS AUTO-GRAVITANTES 55
7.1 – DESCRIÇÃO DO MODELO 55
7.2 – IMPLEMENTAÇÃO EM MPI 58
7.3 – IMPLEMENTAÇÃO EM JIAJIA 60

8. RESULTADOS EXPERIMENTAIS 62
8.1 AMBIENTE COMPUTACIONAL 62
8.2 Medidas de desempenho paralelo 63
8.2.1 *Speedups* relativos 64
8.2.2 Tempos de comunicação 72

9. CONCLUSÃO 76

REFERÊNCIAS 78

Anexo I – Código-fonte da implementação em MPI 82

Anexo II – Código-fonte da implementação JIAJIA 87

LISTA DE FIGURAS

Figura 2.1- Uma taxonomia para sistemas de computação de alto desempenho [7] 16
Figura 2.2 - Classificação de Flynn 17
Figura 2.3 - Classificação de Flynn-Johnson 18
Figura 2.4 – Grafo de tarefas com detalhamento de uma tarefa 20
Figura 2.5 – Quatro ações básicas das tarefas [6] 21
Figura 5.1 – Estruturação básica de um sistema DSM 41
Figura 5.2 - Organização de Memória do JIAJIA 43
Figura 5.3 - Transições de Estado do Protocolo [39]. 45
Figura 6.1 - Tendência de arquiteturas de sistemas [10] 53
Figura 6.2 - Tendência de utilização do *cluster* ao longo dos anos [10] 53
Figura 7.1 – Pseudocódigo do laço de repetição do integrador simplético 56
Figura 7.2 – Diagrama dos blocos de execução do algoritmo paralelo do integrador simplético para n tarefas. 58
Figura 7.3 – Exemplo do funcionamento da instrução MPI_AllReduce 59
Figura 7.4 – Exemplo do funcionamento do jia_barrier() 60
Figura 8.1 – Ambiente computacional utilizado incluindo *hardware* da ferramenta PM^2P. 63
Figura 8.2 – Tempo total de execução para 1, 2, 4, 6 e 8 máquinas 65
Figura 8.3 – *Speedup* relativo para 2, 4, 6 e 8 66
Figura 8.4 – Mapa de Gantt com os limites de tempo de execução e comunicação para 4 tarefas. Duas tarefas na máquina node102 e uma tarefa em cada uma das máquinas node01 e node02. 68
Figura 8.6 – Mapa de Gantt com os limites de tempo total de execução para 8 tarefas. Seis tarefas na node102 e uma tarefa em cada uma das máquinas node01 e node02. 68
Figura 8.7 – Tempos de execução para 1, 2, 4, 6 e 8 máquinas 71
Figura 8.8 – *Speedup* relativo alcançado para 2, 4, 6 e 8 máquinas. 72
Figura 8.9 – Tempos de comunicação para 1, 2, 4, 6 e 8 máquinas 73
Figura 8.10 – Tempo de comunicação para 2, 4, 6 e 8 máquinas 74

LISTA DE TABELAS

Tabela 6.1 – Atributos usados numa classificação de clusters [13].....51
Tabela 8.1 – Tempos de execução da implementação MPI.....64
Tabela 8.2 – *Speedup* relativo para 2, 4, 6 e 8 máquinas.....65
Tabela 8.3 - Instantes de tempo (s) em que ocorrem eventos internos da aplicação mostrados no mapa de Gantt da figura 8.1 67
Tabela 8.4 - Instantes de tempo (s) em que ocorrem eventos internos da aplicação mostrados no mapa de Gantt da figura 8.5 69
Tabela 8.5 - Instantes de tempo (s) em que ocorrem eventos internos da aplicação mostrados no mapa de Gantt da figura 8.6 69
Tabela 8.6 – Tempos de execução em microssegundos para 1, 2, 4, 6 e 8 máquinas 70
Tabela 8.7 – *Speedup* relativo alcançado para 2, 4, 6 e 8 máquinas.....71
Tabela 8.8 – Tempos de comunicação para 1, 2, 4, 6 e 8 máquinas.....73
Tabela 8.9 – Tempos de comunicação para 2, 4, 6 e 8 máquinas na implementação JIAJIA 74

1. INTRODUÇÃO

Ao longo da história dos computadores, o paralelismo tem sido utilizado como método principal para acelerar computações, ou mesmo fazer com que computações muito pesadas possam ser executadas [1, 2, 3]. A imagem clássica da computação paralela é um sistema em que vários processadores são conectados e colocados para trabalharem juntos em um mesmo problema. Idealmente, um sistema como esse deveria processar um programa que leva *T* unidades de tempo para ser executado seqüencialmente em *T/p* unidades de tempo utilizando-se *p* processadores. No entanto, vários fatores podem atrapalhar este ganho em desempenho [6].

O aumento da complexidade dos dados que devem ser analisados, em várias áreas do conhecimento humano, leva à construção de modelos de simulação mais e mais complexos. Neste cenário, a modelagem computacional apresenta-se como uma ferramenta para tratar a simulação de soluções para problemas científicos, analisando os fenômenos, desenvolvendo modelos matemáticos para sua descrição, e elaborando códigos computacionais para obtenção das soluções. O custo computacional dessas simulações, na maioria dos casos, é alto e, por isso, diversas áreas do conhecimento humano necessitam da computação de alto desempenho para viabilizarem seus estudos.

Como exemplos de áreas em que a computação de alto desempenho é ferramenta essencial para a evolução das análises estão: a Biotecnologia com o seqüenciamento de DNA, a Astrofísica, ou ainda a busca por números primos na Matemática. Todas essas são atividades que requerem um alto poder de processamento ou mesmo necessitam de execução paralela de tarefas para que possam ser executadas mais rapidamente.

Várias formas de se obter um alto poder de processamento foram desenvolvidas. Dentre estas estão os sistemas de processamento paralelo como o PVP (*Parallel Vector Processors*), SMP (*Symmetric Multiprocessors*), MPP *(Massive Parallel Processors*) e ainda os *clusters*, e agora, já no final do século XX, a computação em *grid* [1].

Há ainda muitos desafios a serem vencidos na área da computação de alto desempenho, por isso a importância da análise e avaliação de paradigmas de programação paralela, como memória compartilhada distribuída e troca de mensagens, para se entender melhor quais são os gargalos que os impedem de serem mais eficientes e para se buscar solução para estes.

Esta dissertação versa sobre a avaliação de desempenho de um método para a resolução da evolução temporal de sistemas auto-gravitantes em dois paradigmas de programação paralela: troca de mensagens e memória compartilhada.

O trabalho consistiu na migração de um algoritmo seqüencial, originalmente escrito em Fortran, para a linguagem C e sua implementação paralela por troca de mensagens, utilizando-se a biblioteca MPICH 1.2.6, e por memória compartilhada distribuída, utilizando o *middleware* JIAJIA. Foram realizadas execuções das duas implementações em um ambiente homogêneo que permitiram, por meio da análise dos dados resultantes, a comparação do desempenho das duas versões do programa paralelo.

Para o caso da implementação MPI foram realizadas medições de tempo de execução em um ambiente heterogêneo, uma vez que há uma tendência por parte dos usuários de várias ciências, que necessitam da computação de alto desempenho como ferramenta, de construírem ambientes heterogêneos formados por máquinas das mais diversas configurações e ligadas em redes locais.

Esta dissertação está organizada em nove capítulos que apresentam os temas resumidos a seguir. No capítulo 2 é apresentada a definição de computação de alto desempenho, bem como suas principais características, classificações, utilizações e modelos de programação. No capítulo 3 é conceituado desempenho e algumas de suas formas de mensuração e avaliação. A noção de *speedup* é discutida e um modelo para desempenho, englobando tempo de comunicação e de computação, é apresentado. Neste capítulo são ainda abordadas formas de captura dos tempos de execução.

No capítulo 4, são descritos o paradigma da troca de mensagens e as implementações de troca de mensagens PVM e MPI são apresentadas. No capítulo 5 é caracterizada a memória compartilhada distribuída, assim como o sistema escolhido para fazer a implementação do algoritmo do integrador simplético, o JIAJIA.

No capítulo 6, são apresentadas a noção de cluster de computadores e suas principais características e classificações. No capítulo 7 é descrita a aplicação implementada nos dois paradigmas de programação paralela utilizados neste trabalho e no capítulo 8 são apresentados os resultados experimentais obtidos das execuções das aplicações. No capítulo 9 são apresentados as conclusões e os trabalhos futuros.

2. COMPUTAÇÃO DE ALTO DESEMPENHO

A computação de alto desempenho é a área da Ciência da Computação voltada ao estudo de vários aspectos de *hardware* (*pipelining*, conjuntos de instruções, multiprocessadores, interconexões), algoritmos (eficiência, técnicas para concorrência) e *software* (compiladores otimização/paralelização, bibliotecas) [4]. O principal objetivo desta disciplina é o desenvolvimento de algoritmos de processamento paralelo e de programas que podem ser particionados de modo que cada uma das partes possa ser executada simultaneamente em processadores distintos.

O paralelismo se aplica a todos os níveis de projeto de computadores e interage essencialmente com todos os conceitos arquiteturais, apresentando uma dependência única da tecnologia de base. Os pontos básicos de localidade, largura de banda, latência e sincronização são abordados em muitos níveis do projeto de sistemas de computadores paralelos. Os desafios devem ser resolvidos no contexto de cargas de aplicações reais.

Um computador paralelo pode ser definido como sendo uma coleção de elementos com poder de processamento que cooperam e se comunicam para resolver problemas grandes de uma forma mais rápida [5]. Esta definição é ampla o bastante para incluir supercomputadores com centenas ou milhares de processadores, redes de estações de trabalho, estações de trabalho multiprocessadas e sistemas embarcados.

Computadores paralelos são interessantes porque oferecem o potencial de concentrar recursos computacionais, sejam estes processadores, memória, ou largura de banda de entrada e saída, em problemas computacionais de relevância [6].

Há a necessidade de se frisar que a computação usando computadores ligados por uma rede, computação distribuída, não é apenas uma subárea da computação paralela. A computação distribuída está profundamente preocupada com problemas como: confiabilidade, segurança, e heterogeneidade que geralmente são vistos de modo periférico na computação paralela. Um sistema distribuído pode ser definido como sendo um sistema em que a falha de uma máquina que nem se sabia que existia pode deixar seu próprio computador sem uso. No entanto, a tarefa de desenvolver aplicações que possam ser executadas em várias máquinas de uma só vez é um problema de computação paralela.

Neste sentido, os mundos da computação paralela e da computação distribuída convergem. Por isso, trataremos os computadores aqui referenciados indistintamente como paralelos.

Pode-se dizer, então, que computadores paralelos são máquinas velozes empregadas em aplicações especializadas que requerem uma grande capacidade de processamento, no entanto são de elevado custo financeiro. Vários tipos de aplicações necessitam dessas máquinas para serem executas, dentre estas estão a previsão do tempo, o seqüenciamento de DNA, cálculo dinâmico de fluidos, exploração petrolíferas, simulações astrofísicas e muitas outras. Estes tipos de simulações são essenciais para várias indústrias, a saber: automotiva (simulações de colisão, eficiência de combustíveis), aeronáutica (análises de fluxo de ar, eficiência de motores, mecânica estrutural, eletromagnetismo). Em todos estes tipos de aplicações, há uma grande demanda pela visualização de resultados, o que é em si uma grande demanda para a computação paralela. Nestes casos, pode-se notar a importância do modelo computacional paralelo, pois este permite a realização de análises profundas com relativo baixo custo [5].

Como dito anteriormente, a computação paralela é utilizada em várias áreas do conhecimento humano, e o componente de visualização que este tipo de computação possibilita colocou a computação científica e sua engenharia mais próxima da indústria do entretenimento. Em 1995, o primeiro longa metragem animado por computador, *Toy Story*, foi produzido em um sistema de computação paralela composto por centenas de estações de trabalho *Sun*. Esta aplicação só foi possível porque a tecnologia de base e a arquitetura cruzaram três limiares chave: o custo da computação caiu permitindo que coubesse no orçamento do filme, enquanto que o desempenho dos processadores individuais e a escalabilidade do paralelismo cresceram para atender ao prazo (vários meses = várias centenas de processadores). Cada aplicação científica ou de engenharia possui um limiar análogo de capacidade de processamento e custo nos quais se torna viável.

Do exemplo anterior, vê-se que mesmo as máquinas paralelas de larga escala são construídas com os mesmos componentes básicos das estações de trabalho e dos computadores pessoais. Essas máquinas são objeto dos mesmos princípios de engenharia e de desafios de custo versus desempenho. Assim sendo, para ter um desempenho máximo, uma máquina paralela precisa extrair desempenho máximo de seus componentes individuais. Desta forma, entender as arquiteturas paralelas exige um profundo

entendimento dos desafios de engenharia de computadores, e não apenas uma taxonomia descritiva de possíveis estruturas de máquinas [5].

2.1 - CLASSIFICAÇÃO DOS SITEMAS DE COMPUTAÇÃO DE ALTO DESEMPENHO

Há várias formas de categorizar os sistemas de computação de alto desempenho, uma bastante aceita e utilizada é classificação com base na arquitetura de *hardware* [7]. A figura 2.1 representa uma taxonomia dos sistemas de computação de alto desempenho.

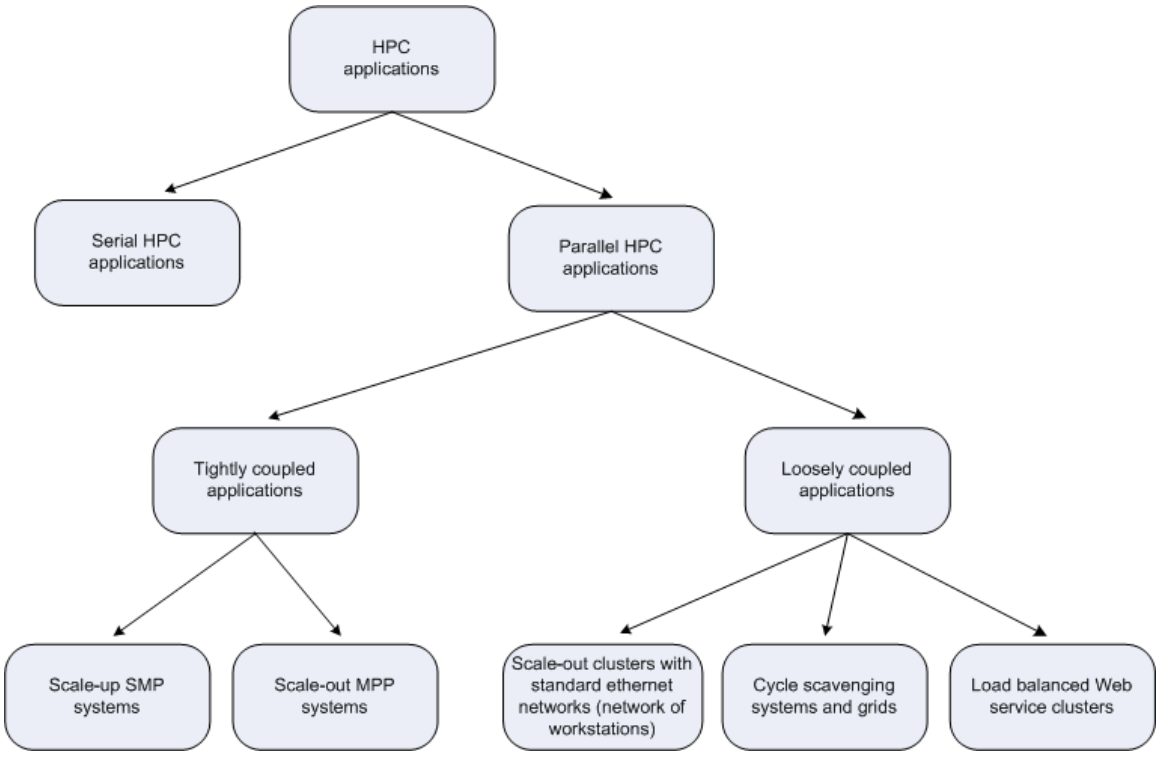

Figura 2.1- Uma taxonomia para sistemas de computação de alto desempenho [7]

Uma outra forma de classificação dos sistemas de computação é a proposta por Flynn [8], que é baseada no fluxo de dados e de instruções. Os termos SISD (*Single Instruction Multiple Data*), SIMD (*Single Instruction Multiple Data*), MISD (*Multiple Instruction Single Data*), e MIMD (*Multiple Instruction Multiple Data*) foram criados por Flynn para representar os quatro tipos de máquinas existentes segundo sua classificação. A figura 2.2 representa esta classificação:

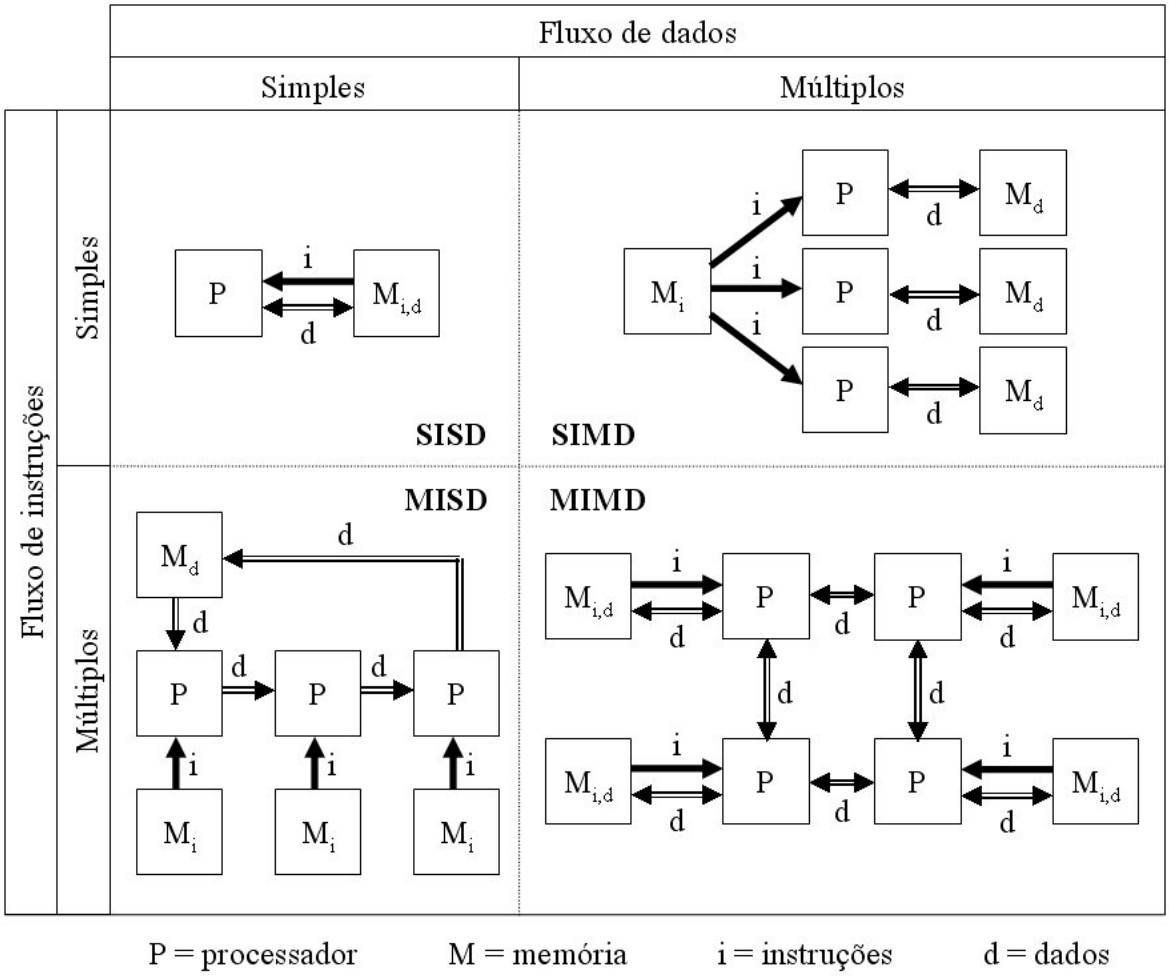

Figura 2.2 - Classificação de Flynn modificada

A classe SISD representa as máquinas mono processadas comuns. Computadores da classe SIMD, com vários processadores dirigidos por instruções oriundas da unidade central de controle, são caracterizados como *array processors*. As máquinas da categoria MISD não são amplamente utilizadas, mas podem ser vistas como *pipelines* generalizados, em que cada estágio executa uma operação relativamente complexa, em oposição aos *pipelines* comuns em que cada estágio executa instruções simples [8].

A classe MIMD engloba uma vasta gama de computadores e, devido a este fato, foi proposta uma classificação baseada na estrutura de memória, global ou distribuída, e no mecanismo utilizado para a comunicação e sincronização, variáveis compartilhadas ou troca de mensagens. Esta classificação foi proposta por E.E. Johnson e é conhecida como Flynn-Johnson por unir a classificação anterior de Flynn com esta nova proposta [9]. A figura 2.3 mostra esta classificação:

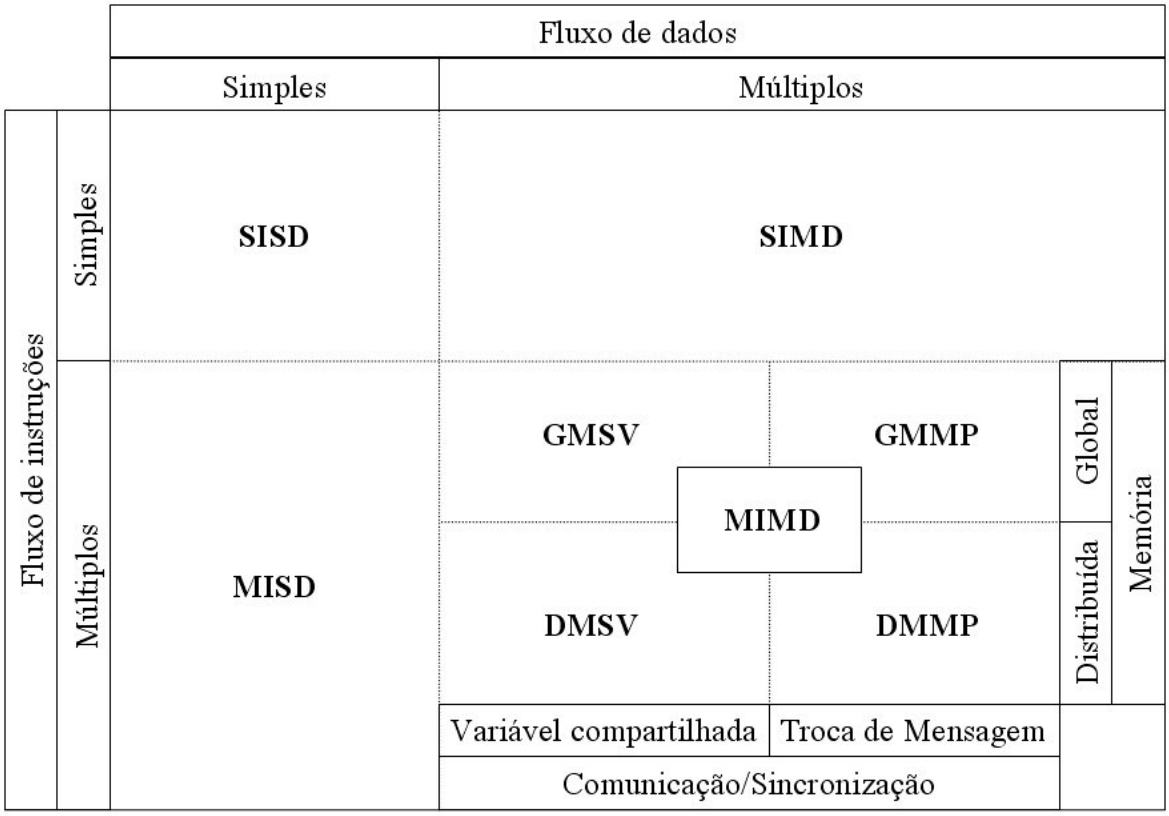

Figura 2.3 - Classificação de Flynn-Johnson

Na figura 2.3 podemos ver as quatro categorias desta classificação. A categoria GMMP (*Global Memory Message Passing*) é de fraca utilização. A classe GMSV (*Global Memory Shared Variable*) se refere a multiprocessadores de memória compartilhada. Já a classe DMMP (*Distributed Memory Message Passing*) é conhecida como multicomputadores de memória distribuída. Por fim, a classe DMSV (*Distributed Memory Shared Variable*) é a combinação da implementação da memória compartilhada com as facilidades da programação utilizando o esquema de variáveis compartilhadas, esta classificação é conhecida como memória compartilhada distribuída [9]. Quando todos os processadores em uma máquina MIMD executam o mesmo programa, este sistema pode ser classificado como SPMD (*Single Program Multiple Data*).

Pelo o que foi apresentado anteriormente, observa-se a existência de várias arquiteturas, e conseqüentemente, várias formas de se construir um supercomputador. No entanto, devido a seu alto custo, muitas vezes os supercomputadores tornam-se distantes da realidade de muitas instituições, fazendo com que o desenvolvimento de pesquisas sobre temas, como os previamente enumerados, tornem-se totalmente inviáveis. Alternativas

como *clusters* ou mesmo o *grid* podem ser utilizadas em casos assim para tornar acessível o poder computacional necessário.

Um *cluster* é um conjunto de computadores, ligados por uma rede de interconexão de alta velocidade, projetado para ser utilizado como um meio de computação integrada ou de processamento de dados. Seus componentes formam um sistema homogêneo gerido por uma entidade administrativa única que tem completo controle sobre o mesmo. É um paradigma de computação distribuída amplamente utilizado [10].

O *grid* é um conceito que vai além de idéias como computação em *cluster*, uma vez que se propõe a eliminar algumas barreiras das arquiteturas homogêneas e da concentração da administração dos recursos. *Grids* têm sido desenvolvidos por universidades, comércio e por entidades governamentais. Existem atualmente muitos projetos de ambientes de *Grid* em vários estágios de desenvolvimento, oferecendo vários tipos de vantagens. No entanto, este conceito de computação distribuída ainda não está tão maduro quanto o de *cluster*, e enfrenta vários problemas como o fato de funcionar bem apenas para um número muito limitado de aplicações devido a baixas taxas de comunicação das redes, grandes latências, diferentes velocidades de CPU e à necessidade de prover tolerância a falhas [11].

2.2 – MODELO PARA PROGRAMAÇÃO EM SISTEMAS PARALELOS E DISTRIBUÍDOS

Um modelo para a construção de aplicações nestes ambientes de programação distribuída e paralela pode ser derivado do modelo de máquina de Von Neumann, em que se assume que um processador é capaz de executar seqüências de instruções. Uma instrução pode especificar, além de várias operações aritméticas, o endereço do dado a ser lido ou escrito na memória e/ou o endereço da próxima instrução a ser executada. É possível programar um computador usando-se este modelo básico, escrevendo-se em linguagem de máquina, mas este método mostra-se complexamente proibitivo para muitos propósitos, pois seria necessário gerenciar milhões de endereços de memória e organizar a execução de milhares de instruções de máquina. Assim sendo, técnicas modulares de projeto são utilizadas, permitindo que programas complexos possam ser construídos a partir de componentes simples, e estes componentes são estruturados em abstrações de alto nível como estruturas de dados, laços iterativos e procedimentos. Abstrações como os procedimentos tornam a modularização mais fácil, pois permitem que objetos sejam manipulados sem preocupação com sua estrutura interna. Linguagens como Fortran e C

permitem que abstrações deste tipo sejam traduzidas automaticamente para código executável [6].

A programação paralela introduz fontes adicionais de complexidade. Se houver necessidade de programar no nível mais baixo, não apenas o número de instruções executadas cresce, mas também é necessário gerenciar explicitamente a execução de milhares de processadores e coordenar milhões de interações entre processos. Assim, na programação paralela, abstrações e modularização são pelo menos tão importantes quanto na programação seqüencial. A modularização pode ser vista como um quarto requisito fundamental para as aplicações paralelas, em conjunto com a concorrência, a escalabilidade e a localidade.

A figura 2.4 mostra um estado de uma computação e o detalhamento de uma tarefa. No grafo dirigido desta representação, uma computação consiste em um conjunto de tarefas simbolizadas por círculos e conectadas por canais (arestas). Uma tarefa encapsula um programa e uma memória local, e define um conjunto de portas que define sua interface com seu ambiente. Um canal é uma fila de mensagens no qual um remetente pode postar mensagens e do qual o destinatário pode removê-las.

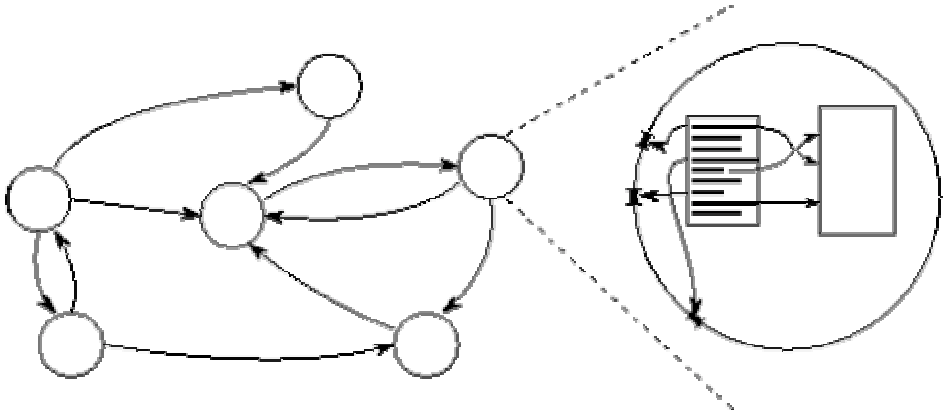

Figura 2.4 – Grafo de tarefas com detalhamento de uma tarefa

Faz-se necessário definir quais abstrações são apropriadas e úteis no modelo de programação paralela. Claramente são indispensáveis mecanismos que permitam explicitar a concorrência e a localidade e que facilitem o desenvolvimento de programas escaláveis e modulares. Também são necessárias abstrações simples de trabalhar e que sejam adequadas ao modelo arquitetural do ambiente paralelo. Inúmeras abstrações poderiam ser

consideradas para este propósito, mas duas se encaixam bem a estas necessidades: a tarefa e o canal [6, 12].

As transições representadas na figura 2.5 mostram que uma tarefa pode, além de ler/escrever na memória local, enviar/receber mensagens, criar tarefas filhas e terminar sua execução. Na computação paralela o número de tarefas pode variar durante a execução do programa. Uma tarefa encapsula um programa seqüencial e uma memória local, ou seja, pode ser vista como uma máquina de Von Neumann virtual. Um conjunto de portas de entrada e saída define a interface da tarefa com o ambiente. Pares de portas de entrada e saída são conectados por filas de mensagens (canais). Os canais podem ser criados e destruídos e as referências a eles podem ser incluídas nas mensagens, de forma que a conectividade pode variar dinamicamente. As operações de envio/recebimento de mensagens podem ser síncronas ou assíncronas. As tarefas podem ser mapeadas em processadores físicos de várias formas e o mapeamento empregado não deve afetar a semântica do programa. Múltiplas tarefas também podem ser mapeadas em um único processador.

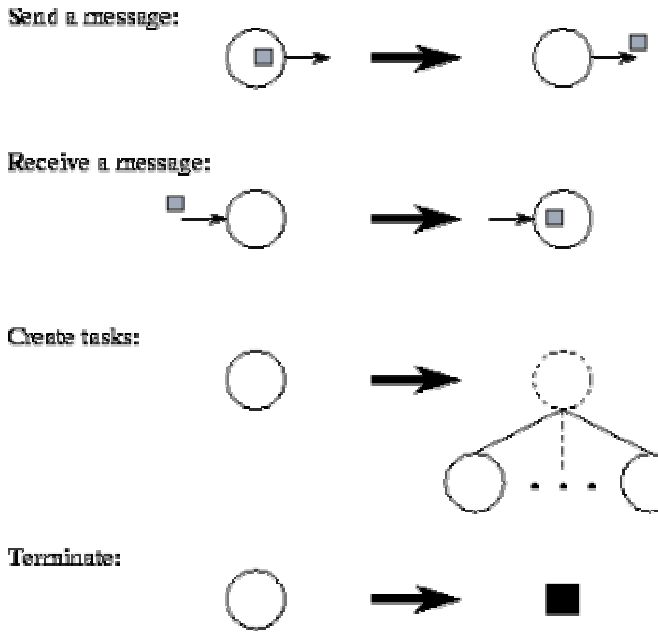

Figura 2.5 – Quatro ações básicas das tarefas [6]

Esta abstração de tarefa provê mecanismos para se falar de localidade: dados que estão na memória local da tarefa estão próximos, os outros dados estão remotos. A

abstração de canal provê um mecanismo para indicar que o prosseguimento da computação de uma tarefa requer dados de outra tarefa, ou seja, há uma dependência de dados.

Abstrações da programação seqüencial, tais como procedimentos e estruturas de dados, são efetivas porque podem ser mapeadas de modo simples e eficiente ao modelo de Von Neumann. A tarefa e o canal podem ser similarmente mapeados para o modelo do multicomputador. Uma tarefa representa uma parte do código que pode ser executada seqüencialmente, em um único processador. Se duas tarefas compartilham o mesmo canal e estão mapeadas em diferentes processadores, a conexão feita pelo canal é implementada como uma comunicação interprocessadores; se as tarefas estão mapeadas em um mesmo processador, um mecanismo de comunicação mais eficiente pode ser utilizado.

O resultado computado pelo programa não depende de onde as tarefas executam uma vez que estas interagem utilizando o mesmo mecanismo – os canais. Desta forma, os algoritmos podem ser projetados e implementados sem preocupação inicial com o número de processadores em que eles serão executados. Na verdade, os algoritmos são projetados para criar muito mais tarefas do que os processadores disponíveis. Esta é uma forma direta de se atingir a escalabilidade (à medida que o número de processadores cresce, o número de tarefas por processador é reduzido, sem necessidade de modificação do algoritmo). A criação de um número de tarefas maior do que o número de processadores também é útil para mascarar atrasos de comunicação, pois fornece um processamento que pode ser realizado enquanto é feita uma comunicação para acessar dados remotos se houver um processador auxiliar de comunicação.

No projeto de programas modulares, vários componentes de um programa são desenvolvidos separadamente, como módulos independentes, e depois são integrados para se obter o programa completo. As interações entre os módulos são restritas a interfaces bem definidas. Assim, a implementação de cada módulo pode ser alterada sem a modificação de outros componentes, e as propriedades do programa podem ser determinadas a partir da especificação de cada um dos módulos. Quando bem aplicada, a modularização reduz a complexidades do programa e facilita o reuso de código.

Um algoritmo ou programa é determinístico se sua execução com uma entrada particular sempre resulta na mesma saída. Um algoritmo é não determinístico se múltiplas execuções com a mesma entrada, resultam em saídas diferentes. Um modelo de

programação paralela que torne simples a construção de programas determinísticos é altamente desejável. Programas determinísticos tendem a ser mais fáceis de entender e também de verificar a corretude.

O modelo tarefa/canal não é o único para representar a computação paralela. Muitos outros modelos foram propostos, diferindo em sua flexibilidade, mecanismos de interação entre as tarefas, granulosidade de tarefas, suporte à localidade, escalabilidade e modularização.

Estilos de programação imperativo, funcional, lógica, *computing-by-learning* e orientação a objeto são alternativas aplicáveis à programação paralela, bastando que a esses estilos sejam adicionados comandos e diretivas destinadas ao paralelismo [13, 14]. Dentre eles, um mais amplamente utilizado é o imperativo por troca de mensagens e memória compartilhada [6].

Programas de troca de mensagens, assim como os programas do modelo tarefa/canal, criam múltiplas tarefas, com cada tarefa encapsulando dados locais. Cada tarefa é identificada por um nome único, e interagem entre si enviando e recebendo mensagens. O modelo de troca de mensagens não impede a criação dinâmica de tarefas, a execução de múltiplas tarefas por processador, nem a execução de programas diferentes por tarefas diferentes.

No entanto, na prática, a maioria dos sistemas de troca de mensagens cria um número fixo de tarefas idênticas no início da execução do programa e não permite que tarefas sejam criadas e nem destruídas durante execução de programa. Estes sistemas implementam um único programa com múltiplos dados (SPMD), ou seja, cada tarefa executa o mesmo programa, mas opera em dados diferentes. O modelo SPMD é suficiente para um vasto leque de problemas de programação de paralela. Este paralelismo de dados explora a concorrência derivada da aplicação da mesma operação a múltiplos elementos de uma estrutura de dados.

No modelo de programação de memória compartilhada, as tarefas compartilham um espaço comum de endereçamento, em que elas lêem e escrevem de modo assíncrono. Vários mecanismos tal como *locks* e semáforos podem ser utilizados para controlar o acesso à memória compartilhada. Uma vantagem deste modelo do ponto de vista do

programador é que não há a noção de “proprietário dos dados”, e assim não há nenhuma necessidade de especificar explicitamente a comunicação de dados entre produtores e consumidores. Este modelo pode simplificar desenvolvimento de programa. No entanto, entender e administrar a localidade dos dados torna-se mais difícil.

3. AVALIAÇÃO DE DESEMPENHO

De uma forma geral, a objetivo maior do desenvolvimento de *software* é projetar e implementar programas que satisfaçam os requisitos dos usuários no tocante à corretude e desempenho. No entanto, no caso dos programas paralelos, o desempenho é um assunto complexo e multifacetado. Deve-se considerar, além do tempo de execução e da escalabilidade dos *kernels* computacionais, os mecanismos pelos quais os dados são gerados, armazenados, transmitidos pelas redes, movido do e para o disco, e passado entre etapas diferentes de uma computação. Ademais, custos incorridos em fases diferentes do ciclo de vida de software, incluindo projeto, implementação, execução e manutenção devem ser contabilizados. Assim, a métrica usada para medir desempenho pode ser tão diversa quanto tempo de execução, eficiência paralela, requisitos de memória, *throughput*, latência, índices de entrada/saída, *throughput* de rede, custos de projeto, custos de implementação, custos de verificação, potencial para reuso, requisitos e custos de *hardware*, custos de manutenção, portabilidade e escalabilidade [6].

A importância relativa destas diversas métricas variará de acordo com a natureza do problema a ser tratado. Uma especificação pode valorizar algumas métricas, exigir que outras sejam otimizadas, e ignorar outras. Por exemplo, a especificação de projeto para um sistema de previsão do tempo estabelece o tempo máximo de execução (a previsão tem que se completar dentro de quatro horas) custo de *hardware* e custos de implementação, e exigem que a fidelidade do modelo esteja dentro destas limitações. No entanto, para um grupo de engenheiros desenvolvendo um programa paralelo de procura de base de dados para o uso ocasional, a satisfação está em encontrar algo que execute mais rápido que um programa seqüencial, e a preocupação poderá ser o longo tempo a ser gasto na implementação. Neste caso, a escalabilidade é menos crítica, mas o código deve se adaptar facilmente a mudanças na base de dados ou de tecnologia.

Como um terceiro exemplo, considere um *pipeline* de processamento de imagens que consiste em várias etapas simultâneas, cada uma executando uma transformação diferente numa *stream* de imagens. Aqui, a preocupação não é com o tempo total necessário para processar um certo número de imagens, mas sim com o número de imagens que pode ser processadas por segundo (*throughput*) ou o tempo que leva uma única imagem para atravessar o *pipeline* (latência). O *throughput* é importante numa

aplicação de compressão de vídeo, enquanto a latência é importante caso o programa faça parte de um sistema de sensores que deve reagir em tempo real a acontecimentos detectados numa *stream* de imagem.

A avaliação de desempenho na programação paralela é um importante estágio que visa à análise dos computadores de alto desempenho, dos algoritmos e das aplicações paralelas bem como à localização dos gargalos limitantes do desempenho e à busca de técnicas que permitam um melhoramento das soluções [15]. Esta avaliação possui um papel importante no direcionamento dos projetos e das melhorias nos sistemas de computação de alto desempenho [16].

3.1 - *SPEEDUP*

Um aspecto analisado na avaliação de desempenho de qualquer sistema ou aplicação é o *speedup*. Esta métrica para análise de escalabilidade pode ser baseada em manter um nível constante de eficiência, uma velocidade constante ou ainda uma taxa constante de utilização [13].

3.1.1 – Lei de Amdahl

O *speedup* baseado em um problema de tamanho fixo pode ser representado pela Lei de Amdahl:

$$S = \frac{1}{T_s + \frac{1 - T_s}{N}} \tag{3.1}$$

onde T_s = parte seqüencial do programa e N = número de máquinas.

Esta lei demonstra o fato de que o gargalo de uma aplicação paralela é sua parte seqüencial. Por ela, à medida que a parte seqüencial do programa aumenta, o *speedup* conseqüentemente diminui. Portanto para atingir um bom *speedup*, é necessário tornar a parte seqüencial do programa a menor possível. Ademais se um problema é constituído de partes paralelizáveis e seqüenciais, é necessário fazer com que a parte maior execute mais rápido [13].

Há ainda uma extensão da Lei de Amdahl que mostra que o desempenho de um programa paralelo não é limitado apenas pelo gargalo dos passos seqüenciais, mas também pelo overhead médio. Lei de Amdahl extendida:

$$S = \frac{N}{1 + (N-1)T_s + \frac{NT_0}{W}} \tag{3.2}$$

onde *To* = *overhead* total e *W* = carga de trabalho total.

Nesta fórmula, o overhead total (*To*) corresponde à soma dos *overheads* de paralelização (criação das tarefas, etc) e de interação (comunição, sincronização, etc).

Uma das principais suposições da Lei de Amdahl é o fato de que o tamanho do problema (*workload*) é fixo e, portanto não é escalável para atingir o poder de computação disponível à medida que o número de máquinas disponíveis aumenta. Isso leva quase sempre ao não aproveitamento dos recursos disponíveis quando uma máquina muito poderosa é empregada na resolução de um pequeno problema.

3.1.2 – Lei de Gustafson-Barsis

Devido às limitações da Lei de Amdahl, em 1988, John Gustafson e Ed Barsis propuseram o que veio a ser chamado de Lei de Gustafson-Barsis afirmando que o gargalo seqüencial pode ser diminuído se a suposição de tamanho fixo para o problema for deixada de lado. Eles propuseram a fixação do tempo para a resolução do problema e não o seu tamanho. Em outras palavras, o que se quer é resolver o maior problema possível em uma máquina com elevado poder de processamento com aproximadamente o mesmo tempo de execução gasto para resolver um problema menor em uma máquina menor [13].

Exemplos representativos de aplicações que se beneficiam com esta proposta é o uso do método de elementos finitos para a análise estrutural ou ainda o uso do método de diferenças finitas para resolver problemas de simulação dinâmica de fluidos para a previsão do tempo, entre outros.

A equação 3.3 representa a Lei de Gustafson-Barsis:

$$S = \frac{T(1)}{T(N)} \tag{3.3}$$

onde T(1) = tempo de execução em uma máquina e T(N) = tempo de execução em N máquinas.

Um bom modelo desempenho é capaz de explicar observações disponíveis e prediz circunstâncias futuras, enquanto abstrai detalhes sem importância [6].

O modelo de desempenho considerado aqui especifica uma métrica cujos parâmetros são T - tempo de execução - como uma função de N - tamanho de problema - número P de processadores, número de U de tarefas, e outras características de um algoritmo específico ou do hardware envolvido (*T = f(N, P, U, ...)*).

Define-se o tempo de execução de um programa paralelo como sendo o tempo gasto desde que o primeiro processador começa a executar o programa até o tempo em que o último processador completa a execução. Esta definição não é totalmente adequada para um computador paralelo de tempo compartilhado, mas é o suficiente para o que será proposto a seguir. Durante a execução, cada processador está computando, comunicando ou está ocioso. O tempo gasto em cada uma dessas ações pode ser representado por T^i_{comp}, T^i_{comm} e T^i_{ocioso} respectivamente para um processador i. Desta forma, o tempo total de execução T pode ser definido de duas maneiras: como a soma dos tempos de computação, comunicação e ocioso de um processador j qualquer, ou como a soma destes tempos de todos os processadores dividida pelo número de processadores P.

$$T = T^i_{comp} + T^i_{comm} + T^i_{ocioso} \tag{3.4}$$

ou

$$T = \frac{1}{P}(T^i_{comp} + T^i_{comm} + T^i_{ocioso}) = \frac{1}{P}(\sum_{i=0}^{p-1} T^i_{comp} + \sum_{i=0}^{p-1} T^i_{comm} + \sum_{i=0}^{p-1} T^i_{ocioso}) \tag{3.5}$$

A última definição é freqüentemente mais útil, uma vez que tipicamente é mais fácil determinar a computação e a comunicação total de um algoritmo paralelo do que o tempo de computação e comunicação individual de cada processador.

Assim sendo, o objetivo é desenvolver expressões matemáticas que especifiquem tempo de execução em função de N, P, etc. Estes modelos devem ser o mais simples

possível, provendo uma acurácia aceitável. As seguintes técnicas podem ser utilizadas para reduzir a complexidade do modelo:

- Usar a análise de escalabilidade para identificar efeitos insignificantes que podem ser ignorados na análise. Por exemplo, se um algoritmo consiste de um passo de inicialização seguido por milhares de iterações de um passos de computação, ao menos que a inicialização seja muito cara, deve-se considerar apenas o passo de computação na análise.

- Usar estudos empíricos para calibrar modelos simples, ao invés de desenvolver modelos muito complexos.

Como visto no modelo exposto anteriormente, o tempo de execução de um programa paralelo possui três componentes básicos: tempo de computação (T_{comp}), tempo de comunicação(T_{comm}) e tempo ocioso(T_{ocioso}). Cada um destes componentes será analisado a seguir.

O tempo de computação de um algoritmo é aquele gasto realizando o processamento descontados os tempos de comunicação ou de ociosidade. Se há um programa seqüencial que realiza a mesma computação que o programa paralelo, pode-se determinar T_{comp} medindo o tempo gasto na execução deste programa seqüencial.

Este tempo de computação vai depender, normalmente, do tamanho do problema representado por um único parâmetro N ou por um conjunto de parâmetros N_1, N_2, N_3, ..., N_m. Se o algoritmo paralelo replica a computação, então o tempo de computação também vai depender do número de tarefas e de processadores. Em um computador paralelo heterogêneo, como uma rede de estações de trabalho, o tempo de computação pode variar de acordo com o processador em que a computação está sendo realizada.

As características do processador e da memória também influenciam o tempo de computação. Por exemplo, a escalabilidade do tamanho do problema ou o número de processadores, pode alterar o desempenho do *cache* ou a efetividade do *pipeline* do processador. Como conseqüência, não se pode assumir automaticamente que o tempo total de computação permanecerá o mesmo se o número de processadores for alterado.

O tempo de comunicação de um algoritmo é aquele que suas tarefas gastam enviando e recebendo mensagens. Dois tipos de comunicação podem ser diferenciados:

comunicação interprocessadores e comunicação intraprocessador. Na comunicação interprocessadores, duas tarefas que se comunicam estão alocadas a diferentes processadores. Este sempre será o caso quando um algoritmo cria uma tarefa por processador. Na comunicação intraprocessador, duas tarefas que se comunicam estão alocados no mesmo processador. Para simplificar, assume-se que os custos das comunicações intra e interprocessadores são comparáveis. Esta suposição não é totalmente absurda para a maioria dos multicomputadores, a não ser para aqueles em que a comunicação intraprocessador é altamente otimizada. Isto porque o custo de cópias memória/memória e de trocas de contexto realizadas em uma típica implementação de comunicação intraprocessador é comparável ao custo de uma comunicação interprocessadores. Em ambientes, como as estações de trabalho conectadas por redes Ethernet, a comunicação intraprocessador é bem mais rápida.

Considerando-se a arquitetura ideal de um multicomputador, o custo de envio de uma mensagem entre duas tarefas localizadas em processadores diferentes pode ser representado por dois parâmetros: o tempo de *startup* da mensagem (tempo necessário para iniciar a comunicação, t_s), e o tempo de transferência de cada unidade da mensagem (considerando-se que esta unidade seja uma palavra de quatro *bytes*, t_w) determinado pela largura de banda física do canal de comunicação que liga a remetente e a destinatária. Assim sendo, o tempo requerido para enviar a mensagem de L palavras é:

$$T_{msg} = t_s + t_w L \tag{3.6}$$

Os tempos de computação e de comunicação são especificados explicitamente em um algoritmo paralelo, sendo mais direta a determinação da contribuição destes tempos para o tempo de execução. O tempo ocioso é mais difícil de ser determinado uma vez que ele depende da ordem que as operações são executadas.

Um processador pode ficar ocioso por falta de instruções para serem processadas ou por falta de dados. No primeiro caso, a ociosidade pode ser evitada utilizando-se técnicas de balanceamento de carga. No segundo caso, o processador fica ocioso enquanto aguarda o processamento ou a comunicação que vai gerar dados remotos. Este tempo ocioso pode ser aproveitado estruturando-se o programa de modo que os processadores realizem outro

processamento ou comunicação enquanto esperam por dados remotos. Esta técnica é conhecida como computação e comunicação de *overlapping*, uma vez que a computação local é executada concorrentemente com a computação e comunicação remotas. Este *overlapping* pode ser feito de duas maneiras. Uma abordagem simples é criar várias tarefas em cada processador. Quando uma tarefa se bloqueia esperando por dados remotos, a execução de uma outra tarefa cujos dados já estão disponíveis pode acontecer. Esta abordagem tem a vantagem da simplicidade, mas só é eficiente caso o custo de agendar uma nova tarefa seja menor do que o tempo de ociosidade a ser evitado. Alternativamente, uma única tarefa pode ser estruturada de modo que as requisições de dados remotos sejam explicitamente intercaladas com outros processamentos.

O tempo de execução nem sempre é a métrica mais conveniente para se avaliar o desempenho de um algoritmo paralelo, uma vez que ele pode variar de acordo com o tamanho do problema e deve ser normalizado quando se compara o desempenho do algoritmo aplicado a diferentes tamanhos de problema. A eficiência, fração do tempo que os processadores gastam fazendo processamentos úteis, é uma métrica que pode prover uma medida mais conveniente da qualidade de um algoritmo paralelo. Ela caracteriza a efetividade com a qual um algoritmo usa os recursos computacionais de um computador paralelo, sendo assim independente do tamanho do problema. A eficiência relativa pode ser definida por [6]:

$$E_{relativa} = \frac{T_1}{PT_p} \tag{3.7}$$

em que T1é o tempo de execução em um processador e Tp é o tempo de execução em P processadores. O *speedup* relativo que é o fator pelo qual o tempo de execução é reduzido em P processadores pode ser dado como [6]:

$$S_{relativo} = PE_1 \tag{3.8}$$

As quantidades definidas nas duas equações acima são chamadas de eficiência relativa e *speedup* relativo porque são definidas em relação a um algoritmo paralelo sendo executado em um único processador. Elas são úteis na exploração da escalabilidade de um algoritmo, mas não podem ser levadas em consideração em termos absolutos. Por exemplo, suponha um algoritmo paralelo que leva 10.000 segundos para ser executado em um

processador e 20 segundos para ser executado em 100 processadores. Um outro algoritmo leva 1000 segundos em um processador e 5 segundos em 1000 processadores. Sendo assim, o segundo algoritmo atinge um *speedup* de apenas 200 e o primeiro algoritmo tem um *speedup* de 500.

Quando se comparam dois algoritmos, pode ser útil possuir uma métrica independente de algoritmos e que não seja o tempo de execução. Sendo assim, a eficiência e o *speedup* absolutos são definidos usando-se, como linha de base, o tempo em um único processador do melhor algoritmo conhecido. Em vários casos, o melhor algoritmo será um algoritmo seqüencial.

3.2 – INSTRUMENTAÇÃO DE CÓDIGO

Como visto, para a obtenção do *speedup* é necessário obter o tempo de execução da aplicação, e para isso, é necessário instrumentar o código. A instrumentação é a inserção de instruções específicas do código para detectar e gravar os eventos do programa. Cada registro de um *trace* contem, pelo menos, o tipo do evento, a data de sua ocorrência e a identificação de que processo executou o evento. O ajuste de desempenho, a eliminação de erros e os testes requerem a instrumentação para prover informações sobre a execução. Por exemplo, a informação sobre o uso de características arquiteturais é requerida para o ajuste de desempenho. Já informações detalhadas a respeito do estado global da computação são essenciais para testar e eliminar erros [17].

A instrumentação do código é uma das formas de coletar dados sobre um programa e há muitas maneiras fazer esta instrumentação. Pode-se escolher introduzir a instrumentação diretamente no código fonte da aplicação, ou deixar o compilador fazer automaticamente este trabalho [1]. Uma outra maneira de instrumentar o código é usar bibliotecas de tempo de execução, ou até mesmo modificar o código executável. Neste trabalho, o interesse é coletar uma informação de granulosidade fina sobre a aplicação e, por isso, decidiu-se utilizar a instrumentação direta da aplicação. Quando este tipo de instrumentação é comparado à instrumentação de *kernel* ou de biblioteca, percebe-se que esta é a melhor maneira de coletar informações nesta granulosidade [17]. Por exemplo, dados em nível dos laços de repetição e em nível de blocos básicos só podem ser coletados com a técnica de instrumentação direta.

A instrução de *assembly* rdtsc retorna o número de ciclos de *clock* decorridos desde que foi feito o último *reboot* da máquina. Esta instrução está presente nos processadores de

arquitetura x86 da Intel, e, portanto é adequada ao nosso ambiente de testes. Devido a sua precisão de nanossegundos e tempo de execução mínimo, esta instrução fornece uma melhor informação sobre o tempo de execução quando comparada às funções gettimeofday da linguagem C, ou ao mpiwtime da biblioteca de MPI que fornecem precisão apenas de microssegundos [18, 19]. Utilizando, nesta dissertação, a instrução rdtsc para obter os tempos de execução do programa, observou-se que o erro da precisão das outras funções de medição de tempo, como jia_clock do *middleware* de memória compartilhada distribuída JIAJIA, foi em torno de 10%.

3.3 – *TRACING* DE CÓDIGO

Além de mensurar o tempo de execução e o *speedup* das máquinas, também é importante verificar o comportamento da aplicação em vários pontos da execução com o intuito de identificar possível gargalos que estejam comprometendo o desempenho ou, até mesmo, o correto funcionamento do programa [20]. Para isso, é necessário fazer o *tracing* do programa. O *tracing* pode ser feito por meio de três técnicas básicas: *tracing* de *hardware*, híbrido e de *software* [21]. Cada uma dessas técnicas possui pontos positivos e negativos.

O *tracing* por *software* é a técnica mais barata e a mais portátil, no entanto, é difícil obter *tracings* da alta qualidade devido à falta de relógios globais e às intrusões causadas pela própria atividade de *tracing*. O *tracing* por *hardware* requer o desenvolvimento de *hardware* específico o que pode ser muito caro, por outro lado, esta é a maneira menos intrusiva de fazer *tracing*, o que significa que tende a não interferir na execução do código.

O *tracing* híbrido combina as abordagens de *tracing* por *software* e por *hardware*. Este tipo de *tracing* é iniciado por instruções em nível da aplicação, o programa que está sendo monitorado inicia a gravação de eventos fazendo um pedido ao coletor do hardware, e as informações geradas são escritas em portas de *hardware* dedicadas conectados ao *hardware* que efetua as monitorações por um barramento ou uma rede de interconexão, o que mantém a intrusão causada pela monitoração em um nível muito baixo. Esta abordagem híbrida fornece a visibilidade dos eventos da instrumentação direta de uma aplicação com os baixos *overheads* dos monitoramentos por *hardware* dedicado.

Para obter medidas precisas do tempo de execução, é necessário sincronizar os relógios das máquinas em que o programa é executado. No entanto, nos *clusters* da classe Beowulf, a sincronização de relógios é um dos problemas mais freqüentes. O relógio

individual de cada nó é sensível às condições externas, tais como a temperatura, e sofre o efeito de diferentes variações, tornando difícil manter uma única visão de *clock*. Várias abordagens de *software*, *hardware* e híbridas foram propostas para melhorar a sincronização dos relógios entre os nós dos clusters, mas os problemas, como a atrasos de comunicação nas redes, não permitem uma visão perfeita de um relógio unificado de referência.

Como os relógios locais das máquinas são assíncronos, é necessária uma maneira externa para obter as medidas do tempo em que ocorrem os eventos da aplicação. Uma abordagem interessante para isso é o uso da porta paralela para enviar sinais a uma máquina (*front end*) responsável por coletar os dados de todas as outras máquinas que estão executando a aplicação cujo comportamento se quer observar.

Neste trabalho, julgou-se que seria interessante utilizar a medição por meio da porta paralela para mapear o comportamento das várias partes do programa. A ferramenta escolhida para realizar esta tarefa foi a PM^2P, desenvolvida no Departamento de Ciência da Computação da Universidade de Brasília [22]. Além de permitir mapear o comportamento do programa, esta ferramenta possibilitou um balanceamento empírico de carga entre máquinas heterogêneas.

4. TROCA DE MENSAGENS

É um paradigma de programação distribuída que consiste em enviar dados de um programa executando em um nó da rede para um programa executando em algum outro nó.

A troca de mensagens é típica de programas executados em um conjunto de sistemas de computação, cada um com sua memória privada, conectados por meio de uma rede de comunicação, qualquer interação entre processos é feita por meio da troca explícita de mensagens [23].

Mensagens de diferentes tipos são trocadas entre os processos e estes, por sua vez, possuem a capacidade de distinguir cada uma delas para que ações diferentes sejam tomadas a cada tipo de mensagem recebida. Um campo próprio para identificar o tipo da mensagem (*tag*) pode ser reservado em todas as mensagens para permitir a identificação [24].

A estrutura básica das primitivas de troca de mensagens é a seguinte:

- *send*(*tag*, *buffer*, comprimento, destino)
- *tag*: tipo da mensagem
- *buffer*: endereço do *buffer* que contem a mensagem a ser enviada
- comprimento: tamanho dos dados
- destino: identificador do processo ao qual a mensagem será enviada.
- *receive*(*tag*, *buffer*, comprimento, remetente)
- *tag*: tipo da mensagem a ser recebida
- *buffer*: endereço do *buffer* em que a mensagem será gravada
- comprimento: tamanho esperado dos dados
- remetente: identificador do processo que enviou a mensagem

O modelo de troca de mensagens se encaixa bem para sistemas distribuídos com um grande número de processadores. Controlando-se cuidadosamente a interação entre os processadores, o desempenho de certas aplicações, que não requerem muita comunicação, é capaz de melhorar progressivamente à medida que o número de processadores aumenta.

Um outro aspecto positivo sobre a troca de mensagens é que ela permite que um *cluster* heterogêneo formado por máquinas com um único processador se comporte como um supercomputador. No entanto, o bom desempenho neste paradigma está intimamente ligado à latência da rede. O tempo gasto para transmitir uma mensagem de um processador a outro varia num intervalo de centenas a milhões de ciclos de *clock*. Se a aplicação requer comunicação freqüente entre os processadores participantes, o ganho em desempenho pode ser seriamente limitado [15].

Várias implementações de troca de mensagens foram desenvolvidas, dentre as mais utilizadas atualmente estão: MPI – *message passing interface* e PVM – *parallel virtual machine*.

4.1 - MPI

O MPI é uma especificação de bibliotecas para troca de mensagens proposta por uma comissão mista de empresas, implementadores e usuários. Esta especificação foi projetada para alto desempenho tanto em máquinas paralelas como em *clusters* de estações de trabalho [25].

Dentre as características da especificação MPI estão as seguintes [26]:

- É uma biblioteca para escrever aplicações, não um sistema operacional distribuído.
- Não exige implementações *thread-safe*, mas as permite.
- É capaz de fornecer alto desempenho em sistemas de alto desempenho.
- Suporta uma computação heterogênea
- É modular, para acelerar o desenvolvimento de bibliotecas paralelas portáveis. A modularização tem muitas implicações. Por exemplo, todas as referências devem ser relativas a um módulo, não ao programa todo. Assim sendo, o processo fonte ou destino deve ser especificado por um *rank* em um grupo, ao invés de ser identificado por um identificador absoluto, e o contexto não deve ser um valor visível.

4.2 - PVM

O PVM é um pacote de software que permite que um conjunto heterogêneo de máquinas Unix e/ou Windows conectadas por uma rede possam ser usadas como uma

única máquina paralela. É composto de duas partes: um *daemon* (pvmd3) que reside em todas as máquinas utilizadas pela aplicação que se quer executar e que formarão a máquina virtual e a outra parte é uma biblioteca de rotinas de interface PVM que contem todas as primitivas de funcionalidades necessárias à cooperação entre tarefas de uma aplicação [27].

O PVM é mais antigo que o MPI, tendo surgido em 1989 nos laboratórios da Emory *University* e Oak Ridge *National Laboratory* com o objetivo de criar e executar aplicações paralelas em um hardware já existente. Sua versão mais atual tem como principal característica a interoperabilidade entre máquinas UNIX e máquinas NT [28].

O PVM foi amplamente difundido aceito contando com milhares de usuários e tornando-se, assim, um padrão de fato devido a sua flexibilidade, uma vez que habilita uma coleção de computadores heterogêneos a comportar-se como se fosse um único recurso computacional expansível e concorrente. Desta forma, problemas computacionais podem ser resolvidos por meio da agregação e do compartilhamento de processadores e memórias de outros computadores com um menor custo efetivo.

Ademais, o PVM dá suporte a diversas arquiteturas e redes de trabalho e oferece a capacidade de utilização efetiva de computação paralela com paralelização escalável e dinâmica, e a utilização de linguagens de programação Fortran e C.

4.3 – COMPARAÇÃO MPI E PVM

Várias diferenças podem ser apontadas entre o PVM e o MPI, a seguir são abordadas quatro delas. Em relação à portabilidade das aplicações, tanto no PVM quanto no MPI programas escritos para uma arquitetura podem ser compilados para uma outra arquitetura diferente sem necessitar de grandes ajustes. No tocante à interoperabilidade o PVM possui um grau maior do que em relação ao MPI, uma vez que sua versão atual permite a integração de máquinas com sistemas operacionais UNIX e Windows NT.

No PVM existe a abstração de máquina virtual permitindo que uma coleção de máquinas seja considerada como sendo um único recurso computacional. No MPI esta abstração não existe, pois não possui o conceito de máquina virtual paralela, estando mais voltado para o conceito de troca de mensagens em si. Sendo assim, o controle de processos é muito maior no PVM permitindo iniciar, interromper e controlar processos em tempo de execução, ao passo que no MPI o controle é bastante restrito, permitido apenas o controle

de grupo de tarefas. Pela existência da abstração da máquina paralela, o PVM é bastante dinâmico em relação ao controle dos recursos disponíveis. Já no MPI, este o controle de recursos é totalmente estático.

No tocante à topologia, o PVM exige que o programador organize manualmente as tarefas em grupos obedecendo a uma determinada organização. O MPI, embora não possua o conceito de máquina virtual, provê um alto nível de abstração em termos de topologia.

Para a realização de comunicação segura, a máquina virtual no PVM é controlada por meio de um *daemon* utilizado também para criar rótulos de contextos únicos. Processos também podem se comunicar com grupos permitindo a recuperação de falhas. No MPI, o nível segurança é mais alto, pois é possível diferenciar as mensagens das bibliotecas das mensagens de usuários.

Para a implementação de tolerância a falhas, no PVM existem esquemas básicos de notificação de falha para alguns casos. Apesar disso, há flexibilidade suficiente para que uma aplicação receba resultados de outras máquinas caso não haja resultado de uma determinada máquina esperada. No MPI, somente nas versões mais recentes é que tais serviços estarão disponíveis, atualmente o modelo seguido é o estático.

Esta pequena comparação demonstra que o PVM possui alguns recursos mais evoluídos que os do MPI, levando a concluir que este seria a melhor opção para a implementação de uma aplicação paralela. No entanto, estas facilidades oferecidas pela PVM possuem um custo que aparece no desempenho das aplicações como observado em [29], em que uma mesma aplicação foi implementada utilizando-se MPI e PVM, e o desempenho da implementação MPI mostrou desempenho notavelmente superior ao do PVM. Por isso, no presente trabalho, o MPI foi o modelo de troca de mensagens escolhido para a implementação da aplicação.

5. MEMÓRIA COMPARTILHADA DISTRIBUÍDA

No começo da utilização dos sistemas distribuídos, os processos em máquinas com memória distribuída executavam em espaços de endereçamento disjuntos, sendo a comunicação entre eles vista em termos de troca de mensagens. Em 1986, Kai Li propôs um esquema diferente, conhecido por DSM (*Distributed Shared Memory*) [30]. Neste trabalho, Li e Hudak propuseram uma coleção de estações de trabalho conectadas por uma LAN (*Local Area Network*) compartilhando um espaço de endereçamento virtual, uma página virtual única.

Entende-se que a memória compartilhada distribuída é uma abstração que cria uma memória global acessível a todos os processadores [31]. Neste paradigma de programação paralela, vários processos compartilham um único espaço virtual de memória. Eles podem ler/escrever nesta memória compartilhada independentemente de onde realmente residam [32]. A comunicação entre os processos é feita por meio de leituras e escritas de dados compartilhados, a semântica desta comunicação é definida pelo modelo de consistência da memória e usa primitivas de sincronização.

Neste modelo de programação, as páginas de memória podem ser residentes (locais) ou não residentes (remotas), os *page faults* gerados pelo acesso a páginas não residentes são tratados pelo sistema operacional ou pelo *middleware* de memória compartilhada distribuída. Há vários aspectos críticos a serem tratados nos DSMs, são eles: desempenho, modelo de consistência de memória e implementação das primitivas de sincronização [33].

Na maioria dos sistemas DSM, a mínima unidade de comunicação é a página, o que gera uma série de questões relativas aos acessos concorrentes a uma mesma página uma vez que isso pode levar à inconsistência de dados e a repetidos *page faults* e transferências de páginas. Uma possível solução para este problema seria restringir o acesso à pagina a um único processo por vez, mas isso aumenta a latência em relação aos outros processos. O desempenho é também afetado pela troca de mensagens dirigidas ao serviço de tratamento de recuperação de páginas. A quantidade de mensagens trocadas é dependente das estratégias de *caching* e do modelo de consistência de memória utilizados [32].

No modelo de programação de memória compartilhada distribuída, o programador especifica a execução paralela marcando o código com diretivas. As diretivas, geralmente, são uma ou mais linhas indicando execução seqüencial ou paralela, tipos de variáveis (compartilhadas ou privadas), esquema de escalonamento entre outros.

A comunicação e a sincronização entre processos é implícita nas regiões paralelas, o que quer dizer que estas operações de comunicação são transparentes e não aparecem no código fonte. Deve-se ainda notar que a paralelização é localizada, ou seja, paralelizar uma área do código não têm efeito lógico sobre o resto do programa. A sincronização transparente e as áreas paralelas localizadas oferecem uma forma mais fácil de trabalhar, especialmente para programadores inexperientes [32].

A organização básica de um sistema de DSM é similar a um multicomputador que utiliza troca de mensagens. Geralmente envolve um conjunto de nós conectados por uma rede de interconexão escalável, a qual estabelece o suporte básico para troca de mensagens eficientes. Cada nó pode ser um sistema mono ou multiprocessado, normalmente organizado com barramento compartilhado. *Caches* privadas aos processadores são utilizadas para reduzir a latência de memória. Cada nó do sistema contém um módulo local de memória física, o qual aponta parcial ou inteiramente para o espaço de endereçamento global de DSM. Cada nó precisa ter também um controlador específico de interconexão para interligá-lo ao restante do sistema. A figura 5.1 mostra esta organização básica dos sistemas DSM.

O sistema de DSM precisa armazenar informações sobre os blocos de dados do espaço de endereçamento virtual compartilhado, tais como estados e localizações correntes. Para isto, normalmente são utilizadas tabelas de sistema ou diretório. A organização do diretório pode ser feita de várias formas, dentre elas: armazenamento totalmente mapeado, ou diferentes organizações dinâmicas (listas simples ou duplamente encadeadas e árvores). O nó pode fornecer armazenamento para o diretório inteiro, ou para apenas uma parte dele. O sistema de diretório pode ser distribuído por meio do sistema e estruturado de forma hierárquica. A organização do diretório e a semântica das informações mantidas por ele, depende do método utilizado para manutenção da consistência de dados.

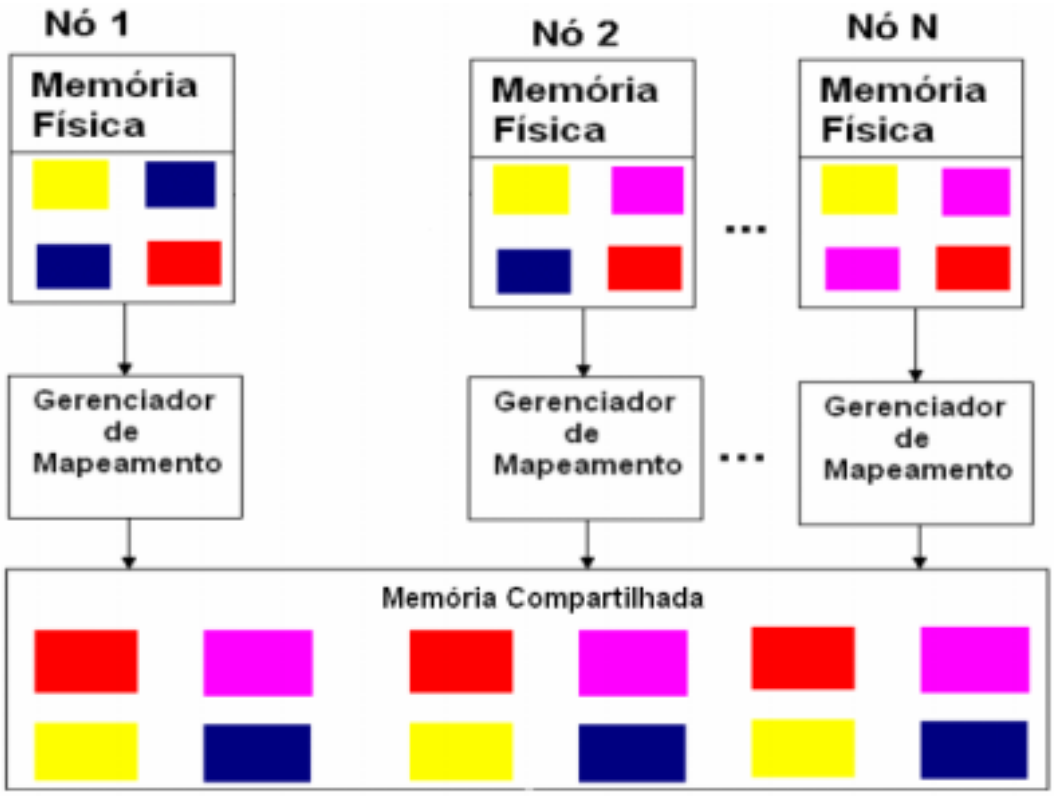

Figura 5.1 – Estruturação básica de um sistema DSM

Uma vez que a unidade mínima de comunicação em DSM geralmente é a página, um aspecto de grande relevância nestes sistemas é a latência de memória, ou seja, o intervalo de tempo compreendido entre o momento no qual um processador iniciou um acesso a um dado compartilhado até que este acesso seja satisfeito. O problema da latência de memória é inerente aos sistemas DSM por causa do grande *overhead* para localização de dados e acesso em sistemas com memória fisicamente distribuída. Este *overhead* é mais sério em DSM baseados em *software*, visto que o *overhead* para acessar a memória remota pode envolver chamadas ao *kernel*, troca de contexto, buferização, além da latência do *software* de comunicação e do *hardware*.

Como a latência da comunicação é fator bastante relavante para os sistemas DSM, o modelo da rede de interconexão é de crucial importância para estes sistemas, assim como o é para os sistemas que utilizam a troca de mensagens. Redes de interconexão devem oferecer uma pequena latência e uma alta banda passante a um custo razoável. Esta relação de desempenho por custo depende freqüentemente do número de nós do sistema. Em sistemas com poucos nós, barramentos e anéis podem ser escolhidos devido seu baixo custo e latência. Em sistemas com cem ou mais nós, outras soluções com maior banda

passante são necessárias. O modelo *crossbars* usa uma grade de barramentos e oferece grande banda passante e baixa latência, entretanto é inviável para centenas de nós.

Uma vez que os sistemas DSM permitem centenas de nós, as redes de interconexão normalmente trocam a complexidade do modelo por maiores latências limitando a conectividade apenas a poucos nós adjacentes. Exemplos populares de tais esquemas são *mesh* e *tori*. No caso de não usar nenhum esquema especial de modelo de interconexão, alternativas com custo razoável incluem tecnologia LAN, tais como Ethernet e ATM, o que é adequado, uma vez que geralmente os sistemas DSM são construídos em redes de estações de trabalhos ou computadores pessoais [34].

Existem vários sistemas que provêem o *middleware* para a memória compartilhada distribuída, a saber: Brazos [35], JIAJIA [36], ADSM [37], e outros [38].

O *middleware* de memória compartilhada distribuída utilizado neste trabalho é o JIAJIA, que é um sistema baseado no conceito *home* para as páginas compartilhadas e que utiliza o modelo de consistência do escopo.

5.1 – JIAJIA

O JIAJIA foi implementado em nível de usuário, como uma biblioteca de tempo de execução, que atua como uma interface para que o sistema operacional detecte e solucione *page faults*. Não são necessárias quaisquer alterações em nível de *kernel* para seu funcionamento, o que facilita sua portabilidade.

O protocolo de coerência, mecanismo para propagar alterações em dados replicados de forma que todos os processadores tenham uma visão coerente da memória compartilhada, foi concebido baseado em *home*. Sendo assim, todas as informações de coerência são passadas por meio de *locks* tornando desnecessárias as informações de diretório.

A maneira como a memória distribuída compartilhada é organizada em cima de um espaço isolado de memória tem grande influência no protocolo de coerência e conseqüentemente no *overhead* do sistema [39].

Neste modelo, as páginas são distribuídas entre todos os nós de forma que cada *host* seja a *home* de uma parte específica do espaço de endereçamento compartilhado. O nó

home de uma página é aquele que possui o seu valor mais recente e, por isso, é de onde a página deverá ser trazida caso ocorra um *page fault*. A figura 5.2 mostra a organização de memória do JIAJIA.

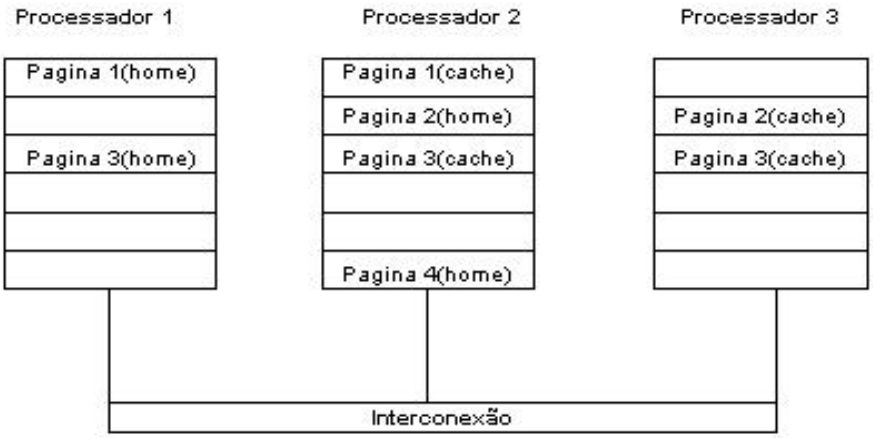

Figura 5.2 - Organização de Memória do JIAJIA

Quando uma página é referenciada pelo seu nó *home*, o acesso ocorre localmente. Mas se uma página remota é referenciada, ocorre um *page fault*, então a página é trazida de sua *home* é armazenada em *cache* para acessos subsequentes. No JIAJIA, uma página em *cache* é sempre armazenada no mesmo endereço que sua cópia em *home*, isto é, os endereços de páginas são idênticos em todos os processadores, como é mostrado na figura 5.2. Esta estratégia elimina a necessidade de tradução de endereço em um acesso remoto. Outra vantagem dessa organização de memória é que, como o nó *home* de uma posição de memória pode ser determindo unicamente pelo seu endereço, não são necessárias informações de diretório para encontrar o dono de uma página.

O JIAJIA permite uma memória compartilhada maior que a memória física de uma única máquina, limitada a quantidade de memória apenas pelo limite da memória virtual [39]. Todos os nós possuem uma estrutura de dados de controle das páginas mantidas em *cache*. Essa estrutura contém o endereço, o estado corrente e um *twin* para cada página em *cache*.

Como dito anteriormente, o protocolo de coerência do JIAJIA é baseado em *lock* e usa uma estratégia de propagação de escritas por invalidação com múltiplos escritores. Nesse protocolo toda a coerência é mantida por meio da leitura e escrita de *write notices* em *locks*, não sendo necessárias quaisquer informações de diretório.

De acordo com a consistência do escopo, modificações realizadas dentro de um escopo devem ser visíveis para os outros processadores que entrem em uma sessão desse mesmo escopo. O protocolo de coerência implementa a semântica desse modelo de consistência propagando as informações a respeito de modificações feitas por um processo saindo da sessão crítica usando o próprio *lock*.

No JIAJIA cada página compartilhada tem uma *home* fixa localizada por uma função de *hash* simples sobre seu endereço. Cada uma delas pode ainda ser mantida em *cache* pelos outros processadores em um dos três estados: Inválida(INV), *Read-only*(RO), *Read-write* (RW).

Os *locks* também são considerados objetos compartilhados e por isso também possuem um nó *home* fixo. Essa associação entre o *lock* e seu nó *home* é feita, assim como para as páginas, por meio de uma função de *hash* simples. A garantia de acesso exclusivo às páginas compartilhadas não é o único papel desempenhado por essas variáveis no sistema, o *lock* também guarda as notificações sobre escritas em páginas durante a sessão crítica correspondente, as chamadas *write notices*.

A seguir é apresentada uma descrição resumida do funcionamento do protocolo de coerência.

Durante uma sessão crítica, se ocorre um *write fault* em uma página, antes que esta seja liberada para a escrita é criado um *twin* para essa página. Um *twin* é uma cópia gêmea da página e é uma cópia do estado exato desta antes da modificação. No momento do *release*, o processo liberando o *lock*, compara todas as páginas em *cache* modificadas durante a sessão crítica com seus respectivos *twins* e codifica as diferenças existentes em estruturas chamadas *diffs*. Cada um dos *diffs* é então enviado para a *home* da linha de memória codificada. Em seguida, é enviada para a *home* do *lock* uma mensagem que o libera e leva anexadas as *write notices* da seção crítica correspondente ao *lock*. O processador *home* do *lock* ao receber essa mensagem, grava os *write notices* localmente e entrega o *lock* a outro processador.

No momento do *acquire*, o processador que está requisitando o *lock*, o faz por meio de uma mensagem para sua *home* e se bloqueia até recebê-lo. O processador *home* de um *lock*, mantém as requisições feitas a este em uma fila de espera. Quando a *home* recebe uma requisição ao *lock*, a coloca no final da fila correspondente e quando o *lock* é liberado, o processador na frente da fila o recebe.

Quando o *lock* é entregue a um processador, junto com a mensagem de confirmação são enviados os *write notices* gravados no *lock*. Assim, ao receber a mensagem de

confirmação, processador que está adquirindo o *lock* invalida todas as páginas em *cache* que, segundo os *write notices* anexados, foram modificadas. Se uma dessas páginas for então referenciada posteriormente será gerado um *page fault* e uma cópia atualizada será trazida da *home*.

Em uma *page fault* de leitura, a página de memória associada é trazida e armazenada localmente em estado *read only* (RO). No caso de um *page fault* de escrita, se a página não está presente ou se está no estado INV, ela é trazida em estado *read/write* (RW) e é criado um *twin* para ela. Se esta página estiver em estado RO na memória local, então seu estado é mudado para RW, é gravado um *write notice* para a página e um *twin* é criado antes da escrita.

A figura 5.3 apresenta um diagrama de transição de estados do protocolo. A transição de RW para RO em um *release* e um *acquire* tem como única finalidade reconhecer escritas em um intervalo e gravar os respectivos *write notices*.

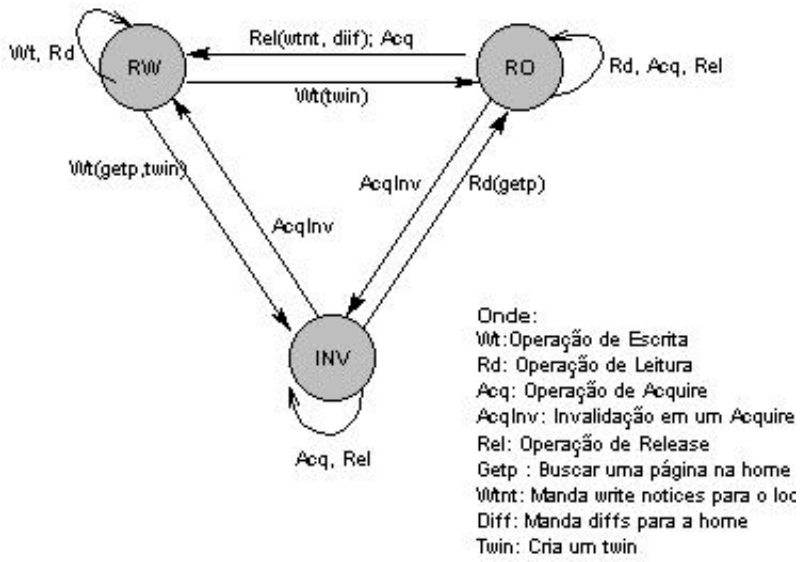

Figura 5.3 - Transições de Estado do Protocolo [39].

Para realizar a alocação de memória, inicialmente o sistema verifica se há memória compartilhada suficiente para atender à requisição. Então o tamanho requisitado é convertido para um múltiplo do tamanho de página e o espaço é alocado na *home* da página, com a chamada de sistema *mmap* do UNIX e de acordo com a distribuição especificada pelo programador. As páginas são inicialmente alocadas como RO em suas

homes, apenas para detectar escritas realizadas pelo processador *home*. Por fim, a variável *globaladdr*, que armazena a quantidade de bytes alocados é incrementada.

O comportamento do manipulador do sinal SIGSEGV é descrito a seguir. Inicialmente uma página é mapeada no seu endereço global apenas pelo seu processador *home*. Referências para páginas não *home* provocarão um sinal SIGSEGV. O JIAJIA utiliza um novo manipulador para esse sinal para criar uma interface que possibilite ao sistema operacional detectar e servir *page faults* no contexto do sistema distribuído.

Quando um *signal* SIGSEGV é gerado, o manipulador inicialmente obtém do sistema o endereço e o tipo de falta. Se a falta ocorreu na *home* da página, então certamente foi uma *write fault* (uma vez que as páginas na *home* são inicialmente mapeadas como RO). Basta alterar a proteção da página para RW e registrar o fato de que a página foi modificada pelo seu nó *home*. Se a falta não ocorreu na *home* da página, esta é mapeada como RW no mesmo endereço da cópia em *home* e uma posição para ela é encontrada na estrutura de dados de *cache*.

Se a falta é uma *read fault*, a página é requisitada a sua *home* por meio de uma mensagem GETP e após o recebimento da página (GETPGRANT), seu conteúdo é copiado para o endereço associado. O estado de proteção da página é então mudado para RO e seu endereço e estado são gravados na estrutura de dados de *cache*.

Se a falta, no entanto, é uma *write fault* e a página não está como RO em *cache*, segue-se o mesmo processo com a diferença de que ao fim o estado da página é mudado para RW e ainda, é criado um *twin* para a página.

Para realizar a sincronização, no JIAJIA os *locks* são mantidos em um *array* de *Maxlocks* posições. Cada elemento do *array* é uma estrutura que guarda os *write notices* relacionados àquele *lock* [39]. Para o correto funcionamento do protocolo de coerência, é importante identificar as páginas modificadas quando há seções críticas aninhadas. No JIAJIA, uma pilha de *locks* é usada para guardar os *diffs* e *write notices* de seções críticas aninhadas. A pilha guarda o *lock* id e os *write notices* e *diffs* produzidos na sessão crítica protegida pelo *lock*. O *lock* é empilhado quando o processador entra em uma seção crítica e desempilhado quando o processador sai. *Diffs* e *write notices* são calculados e armazenados no topo da pilha antes que sejam empilhados ou desempilhados, dessa forma o topo da pilha sempre armazena os *diffs* e *write notices* da sessão atual.

Além disso, como modificações de uma sessão interna também são modificações de sua sessão externa, todos os *write notices* do topo da pilha são copiados para o próximo nível antes da pilha ser desempilhada ou quando a sessão crítica mais interna é

abandonada. Há ainda um *lock* (*hidelock*) que armazena os *write notices* relacionados a barreiras.

Os passos executados no jia_lock são descritos a seguir:

1- Calcula os *diffs* e *write notices* para todas as páginas RW e os coloca no topo da pilha.
2- Muda o estado das páginas RW para RO.
3- Manda uma mensagem requisitando o *lock* (ACQ) para a *home* do *lock* e espera até que a mensagem de confirmação (ACQGRANT) chegue.
4- Quando a mensagem de confirmação chega, as páginas em *cache* são invalidadas de acordo com os *write notices* anexados a mensagem ACQGRANT.
5- O *lock* é colocado no topo da pilha. (topo da pilha é inicializado).

Quando recebe uma requisição de *acquire* ACQ a *home* do *lock* executa as seguintes ações:
1- Adiciona o processador que requereu o *lock* ao fim da fila de espera.
2- Se há apenas um processador na fila de espera então envia uma mensagem de confirmação para o processador que requisitou. Os *write notices* são enviados juntos.

Os passos executados no jia_unlock() são descritos a seguir:
1- Calcula os *diffs* e *write notices* para todas as páginas RW e os coloca no topo da pilha.
2- Muda o estado das páginas RW para RO.
3- Manda os *diffs* para a *home* das páginas modificadas e espera até que uma mensagem de confirmação DIFFGRANT seja recebida.
4- Manda uma mensagem de *release* (REL) para a *home* do *lock* anexada dos *write notices* de páginas modificadas na sessão crítica correspondente.
5- Copia os *write notices* no topo da pilha para o próximo nível da pilha.
6- Desempilha o *lock* (limpa os *diffs* e *write notices* no topo da pilha).

Quando recebe uma requisição de *release* REL a *home* do *lock* executa as seguintes ações:
1- Remove o *host* que enviou a mensagem da frente da fila de espera.
2- Se a fila não está vazia, então manda uma mensagem de confirmação para o processador na frente da fila de espera, anexada dos *write notices*.

Os passos implementados no jia_barrier() são descritos a seguir:
1- Calcula os *diffs* e *write notices* para todas as páginas RW e os coloca no topo da pilha.

2- Muda o estado das páginas RW para RO.

3- Manda os *diffs* para a *home* das páginas modificadas e espera até que uma mensagem de confirmação DIFFGRANT seja recebida.

4- Manda uma requisição de barreira BARR, para a *home* da barreira anexada das *write notices* e espera até que um BARRGRANT seja recebido.

5- Invalida as páginas em *cache* de acordo com os *write notices* anexados na mensagem de BARRGRANT.

6- Limpa os *write notices* de todos os *locks.*

7- Pop Stack (Limpa os *diffs* e *write notices* no topo da pilha).

8- Push Stack (topo da pilha é inicializado).

Quando recebe uma requisição BARR, a *home* da barreira, toma as seguintes ações:

1- Incrementa o contador de barreira.

2- Se o contador de barreiras é igual ao número de processadores, envia uma mensagem de confirmação BARRGRANT para todos os *hosts*, tendo como anexos todos os *write notices* gravados na barreira.

3- Após o BARRGRANT ser enviado para todos os processadores, o contador de barreiras é zerado e todos os *write notices* gravados na barreira são apagados.

6. *CLUSTERS* DE COMPUTADORES

Cluster é um conjunto de máquinas (nós) fisicamente interligadas por uma rede de alto desempenho ou por uma LAN (*local area network*). Tipicamente cada nó do cluster é uma estação de trabalho, uma máquina do tipo PC (*personal computer*) ou um servidor SMP (*symmetric multiprocessor*). Um ponto muito importante a ser observado é que além de poderem ser utilizados individualmente por usuários interativos, todos os nós de um *cluster* devem estar aptos a trabalhar em conjunto coletivamente como uma única fonte integrada de computação [13].

O surgimento dos *clusters* está ligado à necessidade de expansão do poder de processamento e também a necessidade de replicação de cópias de segurança – *backups*. Pode-se dizer que as bases de engenharia para a computação em *cluster*, como forma de processamento paralelo, foram inventadas por Gene Amdahl com a publicação em 1967 da Lei de Amdahl que, como visto no capítulo sobre avaliação de desempenho, descreve matematicamente o *speedup* que pode ser esperado da paralelização de uma tarefa antes executada seqüencialmente. Este trabalho de Amdahl definiu as bases tanto para a computação de multiprocessada quanto para a computação em *cluster*.

Por ser o *cluster* um conjunto de máquinas interligadas por uma rede, pode-se também dizer que a história dos *clusters* de computadores está ligada à história das redes de computadores. Como a motivação primária para o desenvolvimento de uma rede era ligar recursos computacionais, criando assim um cluster. As redes de comutação de pacotes foram conceitualmente inventadas pela corporação RAND em 1962. Usando o conceito de rede de comutação de pacotes, o projeto ARPANET prosperou ao criar, em 1969, o que era o primeiro cluster de computadores conectados por uma rede. Neste projeto foram ligados quatro diferentes centros de computação. O projeto ARPANET evoluiu para a Internet, que pode ser pensada como “a mãe de todos *clusters* de computadores”. Este projeto também estabeleceu o paradigma em uso por todos *clusters* de computadores atualmente: o uso de redes de comutação de pacotes para realizar as comunicações entre os processadores.

O desenvolvimento dos *clusters* feitos por usuários ou por centros de pesquisa foi evoluindo em conjunto com as redes e o sistema operacional Unix na década de 70, assim como foram criados e formalizados protocolos para a comunicação baseada em rede, como o TCP/IP. No entanto, só por volta de 1983 é que os protocolos e ferramentas que

facilitaram a distribuição remota de tarefas e o compartilhamento de arquivos foram definidos, principalmente no contexto do BSD Unix desenvolvido pela *Sun Microsystems*, e foram disponibilizados comercialmente, em conjunto com um sistema de arquivos compartilhado.

O primeiro produto comercial para a montagem de *clusters* foi ARCnet, desenvolvido pela Datapoint em 1977. O ARCnet não foi um sucesso comercial, e a computação em clusters só decolou quando a empresa DEC lançou o VAXcluster em 1984 para o sistema operacional VAX/VMS. Os produtos ARCnet e VAXcluster não davam suporte apenas à computação paralela, mas também ao sistema de arquivos compartilhado e a dispositivos periféricos. Eles foram desenvolvidos para fornecer as vantagens do processamento paralelo, mantendo a confiabilidade e a unicidade. O VAXcluster ainda é um produto disponível, agora com o nome de VMScluster, para sistemas OpenVMS da HP que rodam em sistemas Alpha ou Itanium.

Outro ponto muito importante na história da computação em *cluster* foi o desenvolvimento do *software Parallel Virtual Machine* (PVM) em 1989. Como visto no capítulo 4, este *software* de código aberto baseado em comunicações TCP/IP permite a criação instantânea de um supercomputador virtual, ou seja, um *cluster* de computares de alto desempenho feito de qualquer tipo de sistema interligado por TCP/IP. *Clusters* montados sobre este modelo rapidamente atingiram o número de FLOPS que excederam aqueles disponíveis nos mais poderosos supercomputadores. O PVM e o advento de redes de computadores de baixo custo impulsionaram a NASA na criação de um projeto, em 1993, para a construção de supercomputadores com clusters formados por máquinas de prateleira. Em 1995, surgiu o *cluster* Beowulf – *cluster* de computadores construído sobre uma rede e computadores de prateleira com o objetivo específico de se um supercomputador capaz de realizar processamento paralelo de alto desempenho.

6.1 – *CLUSTERS* BEOWULF

Os *clusters* Beowulf são *clusters* de desempenho escalável baseados em *hardware* de prateleira, em uma rede de comunicação privada, com uma infra-estrutura de *software* de código aberto – Linux. O desempenho pode ser melhorado proporcionalmente com a adição de novas máquinas. O hardware de prateleira pode ser qualquer um do mercado: simples como dois computadores pessoais (PCs) interconectados, com sistema operacional

Linux e um sistema de arquivos compartilhado, ou complexo como 2000 nós em uma rede de baixa latência e alta velocidade.

Podem-se classificar estes *clusters* em: classe I e classe II. Os *clusters* de classe I são montados inteiramente com hardware e software de prateleira, utilizando tecnologias como SCSI, Ethernet e IDE. Este tipo de cluster é tipicamente mais barato dos que os clusters de classe II que podem fazer uso de hardware especializado para atingir um melhor desempenho.

A utilização mais freqüente dos *clusters* Beowulf é para a execução de aplicações tradicionais como: simulações, biotecnologia, mineração de dados, servidores de áudio e jogo na Internet, etc. As aplicações para este tipo de cluster são geralmente escritas em liguagens como o C e o Fortran.

6.2 – CLASSIFICAÇÃO DOS *CLUSTERS*

Uma classificação dos *clusters* pode ser feita por meio de quatro atributos ortogonais destes: *Packaging*, Controle, Homogeneidade e Segurança. A tabela 6.1 mostra os valores que cada um destes atributos pode assumir:

Tabela 6.1 – Atributos usados numa classificação de clusters [13]

Atributo	Valor do atributo	
Packging	Compacto	Distribuído
Controle	Centralizado	Descentralizado
Homogeneidade	Homogêneo	Heterogêneo
Segurança	Isolado	Exposto
Exemplo	*Cluster* dedicado	*Cluster* de uma empresa

Podem-se ainda categorizar os *clusters* como sendo de alta disponibilidade, de balanceamento de carga e de alto desempenho. A seguir uma breve descrição de cada um destes tipos.

Clusters de alta disponibilidade são implementados com o intuito de aumentar a disponibilidade dos serviços providos pelo cluster. Este tipo de *cluster* opera com nós redundantes usados para prover serviços quando algum componente falha, eliminando

assim os pontos únicos de falha. Há várias implementações comerciais de *clusters* de alta disponibilidade para vários sistemas operacionais. Um exemplo é o Linux-HA projeto de software livre para *clusters* de alta disponibilidade usando o sistema operacional Linux [40].

Clusters de balanceamento de carga têm sua carga de trabalho advinda de um ou mais *front ends* de balanceamento de carga que distribuem o trabalho em uma coleção de servidores de *back end*. Ainda que sejam primariamente implementados para um desempenho melhorado, também incluem características de alta disponibilidade. Este tipo de *cluster* é também conhecido como “fazenda de servidores”. Podem ser citados vários exemplos do uso deste tipo de *cluster*, dentre eles o Moab Cluster Suíte [41], Maui Cluster Scheduler [42] e o Linux Virtual Server [43].

Clusters de alto desempenho são construídos primariamente para prover desempenho crescente por meio da dispersão das tarefas computacionais em vários nós diferentes do *cluster* e são mais freqüentemente utilizados na computação científica. Uma das implementações mais populares deste tipo de *cluster* é aquela com nós executando o sistema operacional Linux e *sofwares* livres para implementar o paralelismo. Como dito anteriormente, esta configuração é conhecida como *cluster* Beowulf [44, 45]. Nestes *clusters* são executadas aplicações personalizadas projetadas para explorar o paralelismo disponível. Muitas destas aplicações utilizam bibliotecas, como o MPI, especialmente projetadas para a construção de aplicações científicas para computadores de alto desempenho.

Os *clusters* de alto desempenho são otimizados para cargas de trabalho que requerem comunicação entre processos que executam em nós distintos do cluster. Isto inclui processamentos em que resultados intermediários de um nó afetam cálculos futuros em outros nós.

6.3 – TENDÊNCIA DE UTILIZAÇÃO DOS *CLUSTERS*

O *cluster* é uma tendência em computadores paralelos escaláveis de baixo custo [10], isto se deve ao seu alto grau de escalabilidade e vantagem em termos de custo em relação a outras arquiteturas de máquinas paralelas. A Figura 6.1 demonstra esta tendência em termos percentuais:

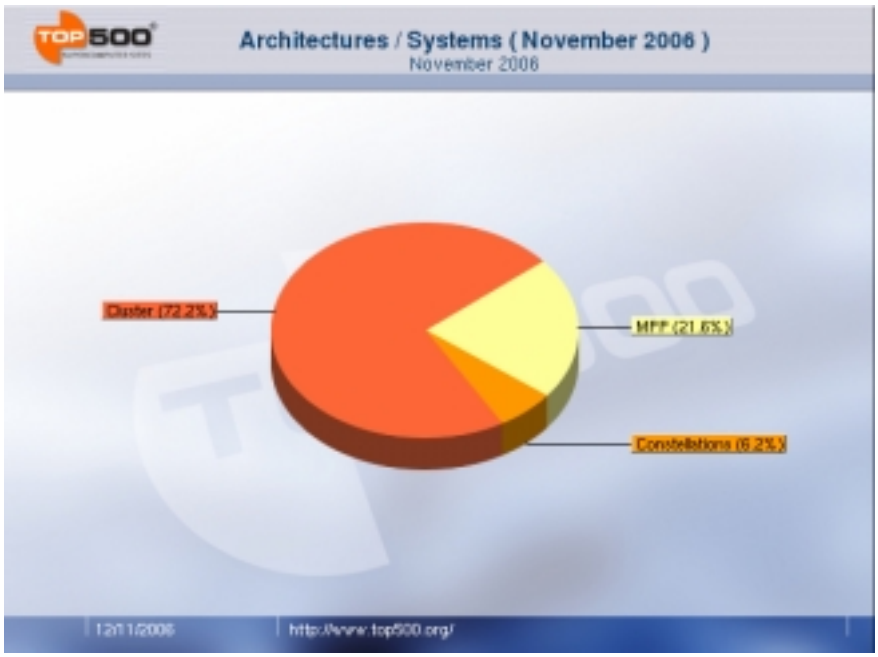

Figura 6.1 - Tendência de arquiteturas de sistemas [10]

Como pode ser visto na figura 6.1, os *clusters* superam as outras arquiteturas de alto poder de processamento totalizando 58,8% dos sistemas paralelos.

Da figura 6.2 pode-se confirmar que o *cluster* é uma tendência crescente ao longo dos últimos anos.

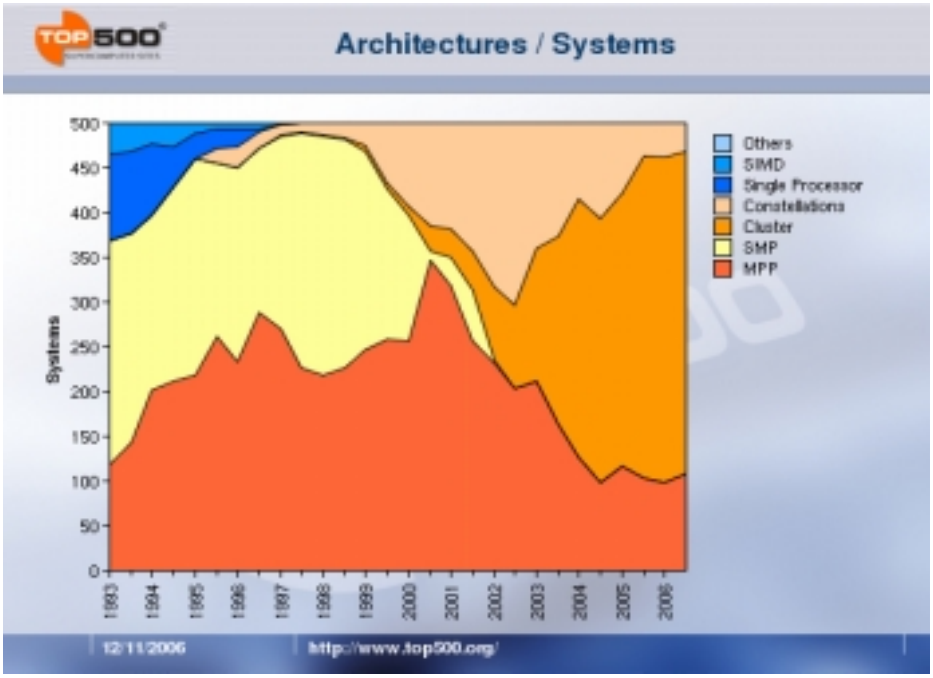

Figura 6.2 - Tendência de utilização do *cluster* ao longo dos anos [10]

Para a realização do presente trabalho, foram utilizados dois *clusters* Beowulf ambos homogêneos. A configuração de cada um destes clusters é apresentada no capítulo 8.

7. ALGORITMO PARA SIMULAÇÃO DA EVOLUÇÃO TEMPORAL DE SISTEMAS AUTO-GRAVITANTES

7.1 – DESCRIÇÃO DO MODELO

Em astrofísica e cosmologia, os sistemas gravitantes de muitos corpos são de grande importância. Exemplos típicos destes sistemas são os *clusters* globulares[1] e as galáxias elípticas reconhecidos como sistemas estelares auto-gravitantes. Neste trabalho, utiliza-se um modelo experimental (*toy model*) representando um sistema auto-gravitante de N-corpos. Neste modelo, simula-se o movimento circular de partículas de massa ***m*** e velocidade ***v*** em um raio ***r*** com uma distribuição esférica de massa e densidade constante.

Para sistemas mecânicos conservativos que podem ser descritos pelas equações de Hamilton, o integrador simplético é o mais adequado uma vez que estes sistemas preservam seu volume em espaço de fase[2] e podem ser explicitamente revertidos. O algoritmo escolhido é um integrador de quarta ordem descrito por Omelyan et al [46]. O código implementado é aplicado ao modelo de anel auto-gravitante (modelo HMF) para ***n*** partículas, extensivamente estudado na literatura como um modelo simplificado de sistemas de interações de longos intervalos em Física. A interação potencial nestes sistemas conduz a uma simplificação na implementação numérica, de modo que o tempo de CPU cresce linearmente com o número de partículas do sistema. Esta propriedade é particularmente útil para testar o código paralelo para um grande número de partículas.

As principais grandezas envolvidas no sistema modelo são: posição, velocidade, massa e força. Para os testes realizados, a massa foi considerada como sendo igual a 1. Na implementação do algoritmo, a posição, a velocidade e a força são representadas por três vetores distintos: ***p***, ***r*** e ***f*** respectivamente, em que cada posição de cada um dos vetores corresponde aos dados relativos a cada uma das partículas do sistema. A figura 7.1 representa o pseudocódigo do laço de repetição, que é a parte principal do algoritmo implementado.

[1] *Clusters* globulares são concentrações esféricas de estrelas geralmente com 100 anos luz de diâmetro e contém milhares de estrelas. Estas concentrações têm pelo menos 10 bilhões de anos de idade e surgiram quando a estava sendo formada. Estão espalhados por um halo esférico em torno de nossa galáxia e leva centenas de milhões de anos para percorrê-la.

[2] Um espaço de fase é o espaço multidimensional constituído pelos pontos que correspondem aos estados de cada partícula. . A posição de cada ponto desse espaço abstrato é definida por coordenadas que correspondem às coordenadas generalizadas e aos momentos generalizados.

```
Enquanto (tempo < tempoFinal)
{
        // 1º passo

        Mx=0.0;
        My=0.0;
        para (i=0;i<tamanhoVetores;i++)
        {
                f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)n;
                p[i]=p[i]+CSI*dt*f[i];
                r[i]=r[i]+(1.0-2.0*LAMBDA)*dt*p[i]/2.0;

                Mx += cos(r[i]);
                My += sin(r[i]);
        }
        MxTotal=0.0;
        MyTotal=0.0;

        Operação de Comunicação (Mx);
        Operação de Comunicação (My);
        .....

        // 4º passo

        Mx=0.0;
        My=0.0;
        para (i=0;i<tamanhoVetores;i++)
        {
                f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)n;
                p[i]=p[i]+XI*dt*f[i];
                r[i]=r[i]+(1.0-2.0*LAMBDA)*dt*p[i]/2.0;

                Mx += cos(r[i]);
                My += sin(r[i]);
        }
        MxTotal=0.0;
        MyTotal=0.0;

        Operação de Comunicação (Mx);
        Operação de Comunicação (My);

        para (i=0;i<tamanhoVetores;i++)
        {
                f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)n;
                p[i]=p[i]+CSI*dt*f[i];
        }

        tempo = tempo + dt;
}
```

Figura 7.1 – Pseudocódigo do laço de repetição do integrador simplético

Na implementação do algoritmo, quatro blocos de operações similares às

apresentadas na figura 7.1 são executados, diferindo nos cálculos dos vetores ***f***, ***p*** e ***r*** de acordo com os passos para integração simplética de quarta ordem. Nesta implementação do algoritmo, as partículas do sistema são distribuídas igualitariamente entre as tarefas do sistema, ou seja, cada tarefa é responsável por atuar sobre um subconjunto de partículas e operações de redução, no caso da implementação por troca de mensagens, são utilizadas para agregar e distribuir valores globais entre as tarefas. Para a implementação por memória compartilhada, os valores globais são diretamente escritos em variáveis compartilhadas com os necessários mecanismos de sincronização.

A seguir são descritos os vários passos que compõem o algoritmo do integrador simplético de Omelyan. Primeiro, em um passo de inicialização, são atribuídos valores aleatórios a cada uma das posições dos vetores velocidade e posição (***r*** e ***f*** respectivamente) e a energia inicial do sistema ($\boldsymbol{e_0}$) é calculada. As energias cinética e potencial também são calculadas neste passo de inicialização, para cada partícula é calculada individualmente a energia cinética parcial e depois se somam os valores parciais da energia cinética, obtendo o valor da energia cinética total que é distribuído para todas as tarefas. Para o cálculo da energia potencial, utilizam-se as magnetizações no eixo x e no eixo y ($\boldsymbol{M_x}$ e $\boldsymbol{M_y}$) que são, respectivamente, a soma dos valores do co-seno e do seno das posições de cada uma das partículas (***r[i]***). Após o cálculo das energias cinética e potencial, calcula-se a energia total do sistema que será periodicamente calculada ao longo dos cálculos como forma de verificar a propriedade conservativa do sistema.

Depois de realizado o passo de inicialização, o próximo passo é um laço de repetição que representa a decorrência do tempo em que se quer simular a evolução do sistema. Dentro deste laço de repetição são calculados por quatro vezes os valores de força, velocidade e posição, bem como as magnetizações $\boldsymbol{M_x}$ e $\boldsymbol{M_y}$ para cada uma das partículas do sistema.

A figura 7.2 representa, de modo geral, a seqüência de execução do algoritmo do integrador simplético de forma paralela para ***n*** tarefas. Dessa figura pode-se notar que a comunicação entre as tarefas ocorre na porção do código correspondente aos cálculos necessários à resolução das equações envolvidas (cálculo da velocidade, posição e força de cada uma das partículas).

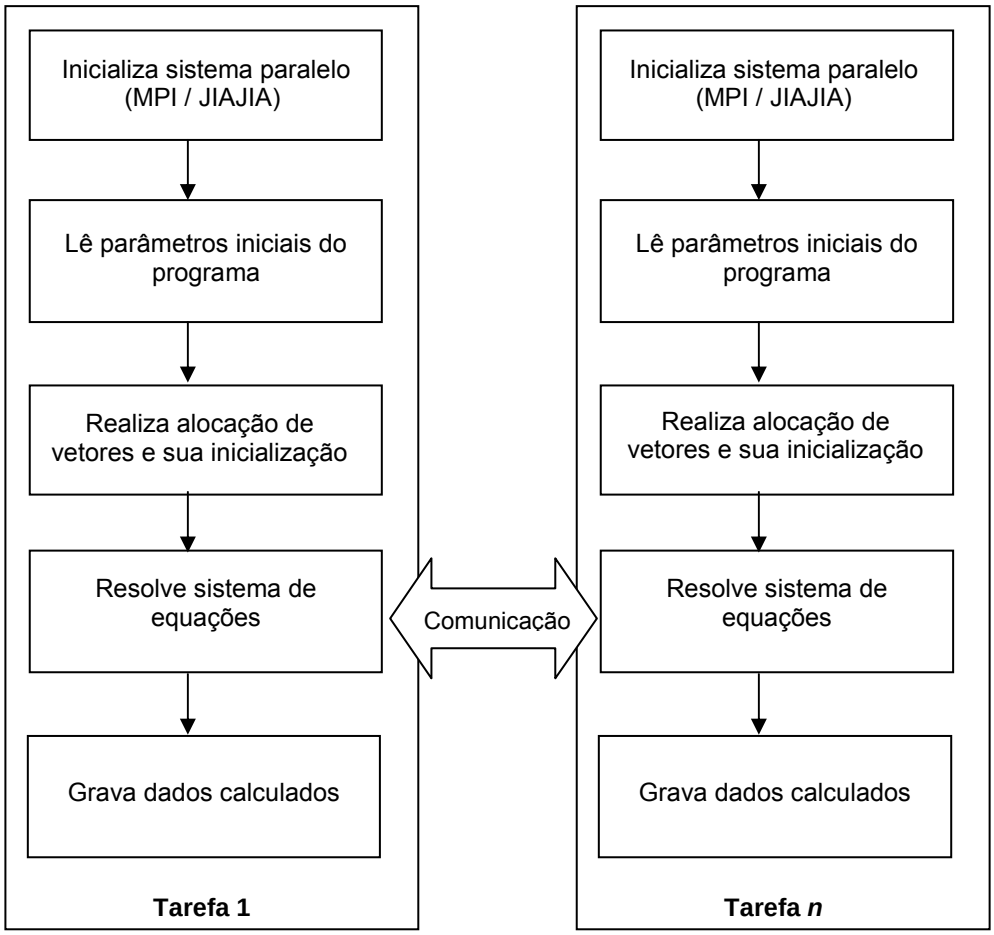

Figura 7.2 – Diagrama dos blocos de execução do algoritmo paralelo do integrador simplético para n tarefas.

7.2 – IMPLEMENTAÇÃO EM MPI

Um importante aspecto da implementação paralela do código do integrador simplético é a comunicação. Na implementação MPI, para realizar esta comunicação foi utilizada a instrução *AllReduce* ao invés de *Sends* e *Receives* pela facilidade oferecida por esta operação de realizar o cálculo deseja, neste caso a soma, e partilhar o resultado da soma com todos as tarefas participantes da execução.

A instrução *AllReduce* consiste em combinar os valores de todas as tarefas e distribuir o resultado para todas elas. A assinatura da função MPI_AllReduce é a seguinte:
int MPI_Allreduce(void ****sendbuf***, void ****recvbuf***, int ***count***, MPI_Datatype ***datatype***, MPI_Op ***op***, MPI_Comm ***comm***)
em que

- ***sendbuf*** é o endereço do *buffer* de envio

- ***count*** é o número de elementos no *buffer* de envio
- ***datatype*** é o tipo de dados dos elementos do ***sendbuffer***
- ***op*** é a operação a ser realizada com os dados (soma, subtração, divisão, etc.)
- ***comm*** comunicador utilizado para transmitir a mensagem.
- ***recvbuf*** *buffer* de recebimento do resultado da operação

A figura 7.3 exemplifica o funcionamento da instrução MPI_AllReduce para a soma de números para 3 tarefas.

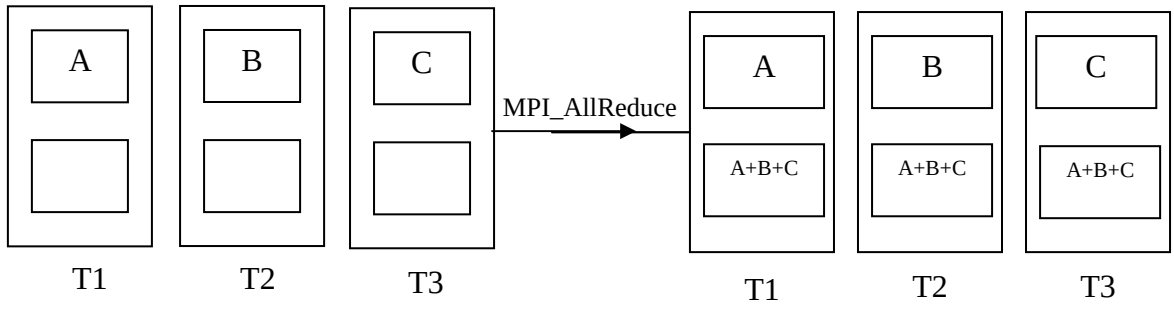

Figura 7.3 – Exemplo do funcionamento da instrução MPI_AllReduce

No caso da presente implementação, a operação MPI_AllReduce foi usada para o recolhimento e soma dos valores das variáveis ***Mx*** e ***My*** parcialmente calculados pelas tarefas. Como cada tarefa calcula apenas a parte de ***Mx*** ou ***My*** correspondente às partículas que foram alocadas a ela, é necessário somar os valores parciais de cada uma das tarefas para a obtenção dos valores totais de ***Mx*** e ***My*** a serem utilizados nos cálculos de ***p***, ***r*** e ***f***. Sendo assim, os parâmetros passados na função MPI_AllReduce para o cálculo de ***Mx*** foram os seguintes:

MPI_AllReduce(***&Mx***,***&MxTotal***,1, ***MPI_DOUBLE***,***MPI_SUM***,
MPI_COMM_WORLD) em que:

- ***&Mx*** é o endereço do *buffer* de envio
- 1 é o número de elementos no *buffer* de envio
- ***MPI_DOUBLE*** é o tipo de dados dos elementos do ***sendbuffer***
- ***MPI_SUM*** indica que a operação de soma deve ser realizada
- ***MPI_COMM_WORLD*** comunicador global do MPI, significa que todas as tarefas enviam e recebem o resultado da operação efetuada em ***Mx***.
- ***&MxTotal*** *buffer* de recebimento do resultado da operação

O cálculo de ***My*** é realizado da mesma forma.

7.3 – IMPLEMENTAÇÃO EM JIAJIA

Como a comunicação em memória compartilhada distribuída é feita por meio da leitura/escrita dos dados nas áreas de memória compartilhada, na implementação JIAJIA é necessário sincronizar estes acessos à memória e isso é feito por meio de barreiras e de acesso exclusivo de uma tarefa em determinados trechos do programa.

Inicialmente, a exclusividade de acesso de uma tarefa a uma sessão crítica foi delimitada por *locks*, mas isso acarretou em um grande *overhead* na aplicação. Então, substituem-se os *locks* por barreiras e IF's que testam se o identificador da tarefa é igual a zero (tarefa mãe), em caso positivo, a tarefa executa o código delimitado pelo IF enquanto as outras tarefas aguardam na barreira. A utilização de barreira é mais adequada à natureza da aplicação uma vez que é necessário que todas as tarefas conheçam os valores finais de todas as posições dos vetores para que possam passar para uma nova etapa de cálculos.

A instrução de barreira utilizada foi o jia_barrier(), que faz com que todas as tarefas fiquem aguardando em determinado ponto do programa (o ponto em que está a barreira) até que todas cheguem à barreira para continuar a execução. A figura 7.4 exemplifica a utilização da barreira para a sincronização de três processos (representados por setas). A diretiva WAIT() bloqueia até que todos os processos tenham atingido a barreira.

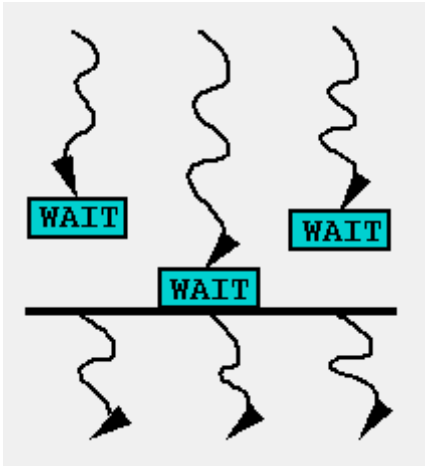

Figura 7.4 – Exemplo do funcionamento do jia_barrier()

A barreira, cujo funcionamento foi detalhado no capítulo 5, foi utilizada para que o cálculo de ***Mx*** e ***My*** fosse feito apenas pela tarefa mãe, e depois os valores escritos em um

vetor de ***n*** partículas posições devidamente atualizado que pode ser utilizado por todas as demais tarefas.

8. RESULTADOS EXPERIMENTAIS

O código do integrador simplético foi implementado na linguagem C, em dois paradigmas de programação paralela distintos: MPI para troca de mensagens e JIAJIA para a memória compartilhada distribuída. Para a verificação do desempenho das duas implementações, foram realizados testes de execução em ambientes de *hardware* homogêneo e heterogêneo. Neste capítulo, são apresentados os ambientes computacionais em que os testes foram realizados, bem como os procedimentos adotados, os dados utilizados e os resultados obtidos.

8.1 AMBIENTE COMPUTACIONAL

Para implementação e teste do integrador simplético foi utilizado um cluster Beowulf dedicado. Este cluster, denominado cluster IE, é constituído por oito máquinas cada uma com dois processadores AMD Athlon MP 1900+ com freqüência de *clock* de 1600 MHz e 256KB de *cache* L2. Cada nó deste cluster possui 1GB de memória RAM local e 40GB de disco rígido. Estas máquinas estão interconectadas por uma rede com velocidade de transmissão de 1 gigabit/s, com sistema operacional Red Hat 9. Quando a porta paralela foi utilizada para coletar informações para a análise de desempenho, foi empregada como *front end* uma máquina com as seguintes características: processador Intel Pentium II 350 MHz com 512 KB de *cache* L2, 160 MB de memória RAM e disco rígido de 6,5 GB. Esta máquina está conectada ao *cluster* IE por meio de uma rede de 100 megabits/s. Para a realização dos testes em ambiente heterogêneo, foram utilizadas duas máquinas com configuração igual à da máquina *front end* citada anteriormente e uma das máquinas do *cluster* IE, estas três estações e a máquina *front end* estão interligadas por uma rede de megabit. A figura 8.1 mostra a representação gráfica do ambiente computacional utilizado.

Na versão de implementação por troca de mensagens, foi utilizada a biblioteca MPICH versão 1.2.6. Para implementar a versão com memória compartilhada distribuída, o *middleware* escolhido foi o JIAJIA versão 2.1. Para a medição de tempos de execução na implementação de memória compartilhada distribuída foi utilizada a instrução *assembly* RDTSC (*read time-stamp counter*), uma vez que esta instrução mostrou-se menos intrusiva

e com maior resolução [18, 19]. Na implementação por troca de mensagens, os tempos apresentados foram obtidos com medição por meio da porta paralela utilizando-se a ferramenta PM^{2}P [22]. Além disso, medições com o uso da instrução RDTSC também foram efetuadas como meio de validar a comparação de desempenho entre os dois paradigmas de programação.

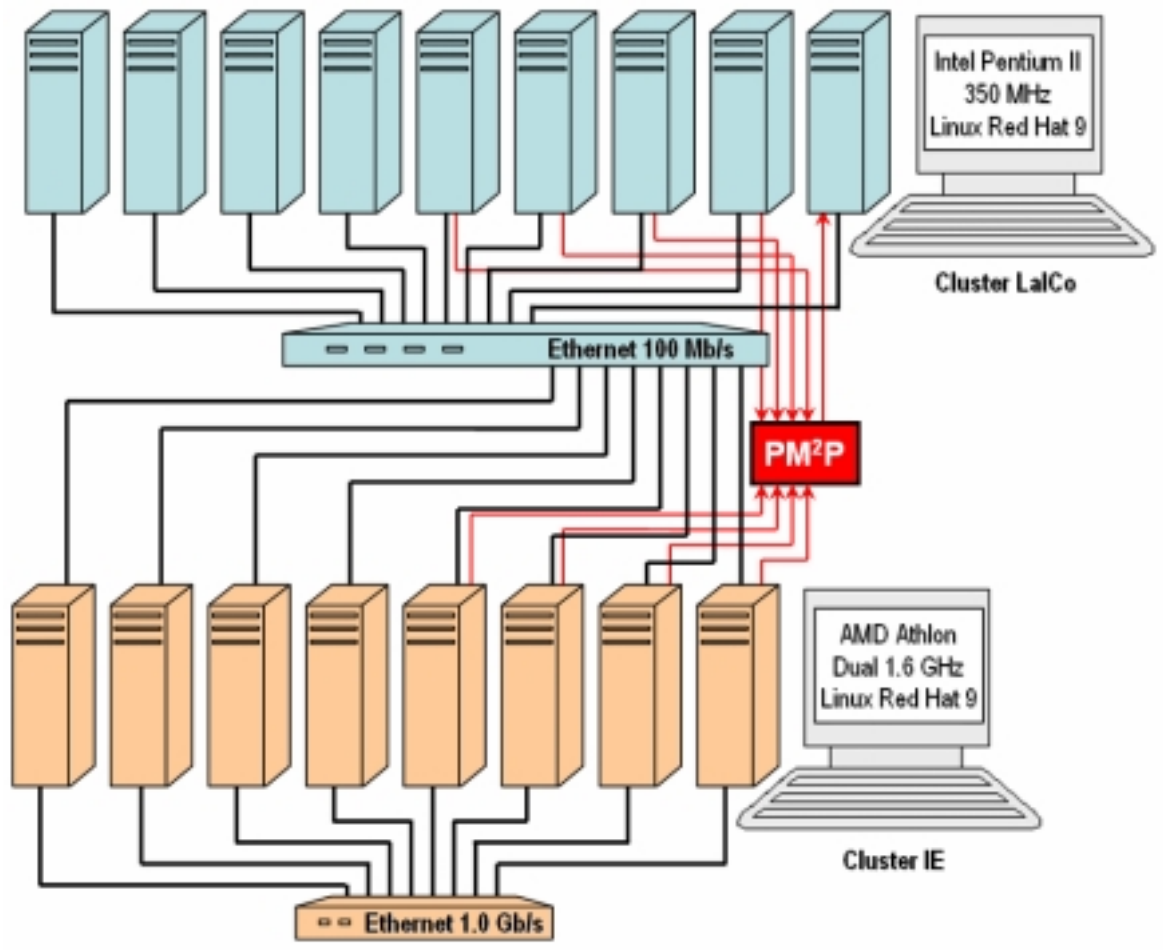

Figura 8.1 – Ambiente computacional utilizado incluindo *hardware* da ferramenta PM^{2}P.

8.2 Medidas de desempenho paralelo

As implementações com troca de mensagens e memória compartilhada foram testadas para 1, 2, 4, 6 e 8 máquinas, cada uma executando apenas 1 tarefa. Os testes foram realizados de modo que as partículas fossem igualitariamente distribuídas por todas as máquinas. Assim sendo, o número de partículas do sistema variou de 24 mil a 98 milhões. A opção por iniciar os testes com 24 mil partículas deve-se ao fato deste ser o primeiro número

resultante do MMC (mínimo múltiplo comum) entre os números 6 e 8 multiplicado por um fator de escala, neste caso 10^3, que é significativo em termos de tempo de execução.

8.2.1 *Speedups* relativos

8.2.1.1 MPI

A tabela 8.1 a seguir mostra os tempos totais de execução expressos em segundos da implementação MPI para 1 iteração do algoritmo nas máquinas no *cluster* IE. O gráfico loglog da Figura 8.2 permite a visualização gráfica do comportamento dos tempos de execução da implementação por troca de mensagens do algoritmo.

Tabela 8.1 – Tempos de execução da implementação MPI

Número de partículas	Número de máquinas				
	1	2	4	6	8
24 x10^3	0,055338429	0,029670311	0,025590111	0,027549789	0,125588511
48 x10^3	0,110421969	0,057645349	0,035300597	0,0357131	0,130497549
96 x10^3	0,220530866	0,113359003	0,038202906	0,045943374	0,13598406
192 x10^3	0,485261063	0,223983263	0,125658034	0,072039397	0,160986806
384 x10^3	0,880500417	0,488891737	0,2300784	0,169940291	0,224616346
768 x10^3	1,766101006	0,8854091	0,562964105	0,309419729	0,241550494
1036 x10^3	3,543616906	1,767637023	0,895849809	0,596182277	0,509356346
3072 x10^3	7,072499497	3,532233763	2,218901619	1,200720134	0,936331417
6144 x10^3	14,13663543	7,092949589	3,541953429	2,3695725	1,89234452
12288 x10^3	28,20532013	14,11260543	7,098588943	4,72411496	3,55118476
24576 x10^3	56,45811365	28,27478303	14,14620363	9,464272037	7,310004583
49152 x10^3	1062,855414	56,54956606	28,30137953	18,8669209	14,18900364
98304 x10^3	-	-	-	37,75598629	28,33791351

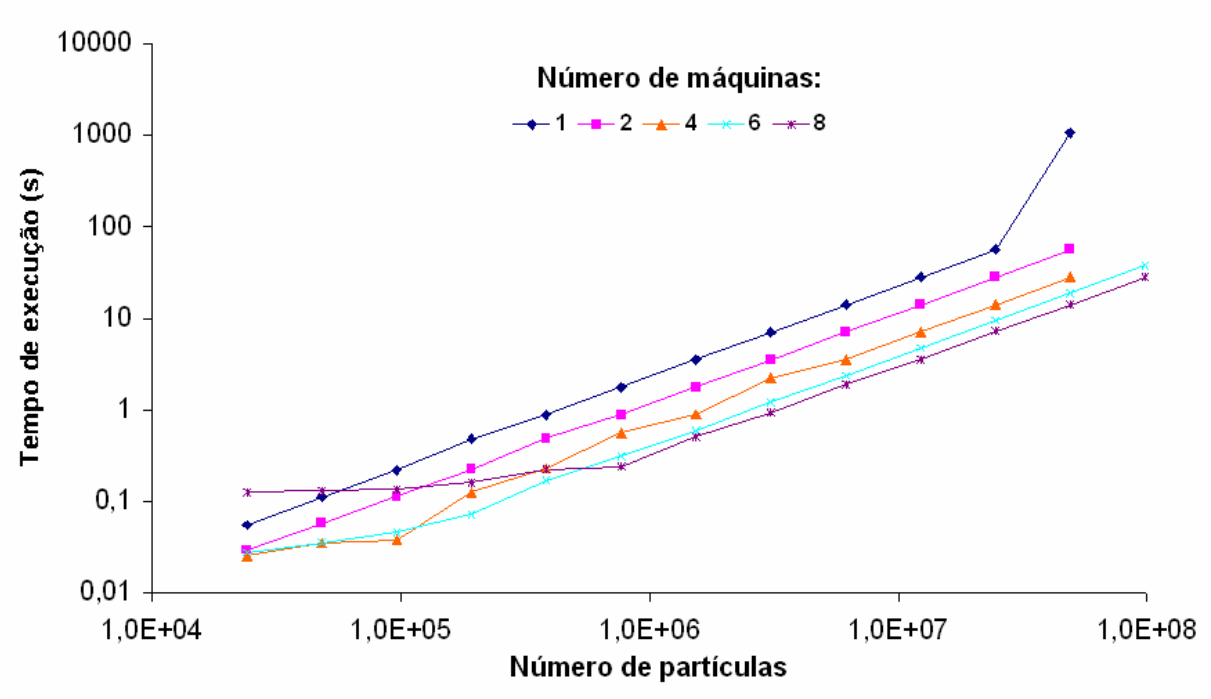

Figura 8.2 – Tempo total de execução para 1, 2, 4, 6 e 8 máquinas

A tabela 8.2 mostra o *speedup* relativo obtido com as execuções da implementação utilizando a biblioteca MPI:

Tabela 8.2 – *Speedup* relativo para 2, 4, 6 e 8 máquinas

Número de partículas	Número de máquinas				
	1	2	4	6	8
2,40E+04	1	1,91190854	1,9178663	2,48658759	1,6897776
4,80E+04	1	1,96019778	3,7616343	5,30120758	4,3398392
9,60E+04	1	1,99549971	3,87068	5,62477203	7,3114077
1,92E+05	1	2,18902391	4,3337231	6,37866591	8,378041
3,84E+05	1	1,79704966	3,9434221	5,87966016	7,7542947
7,68E+05	1	1,98997169	3,573508	5,91951188	7,5636184
1,04E+09	1	1,9906421	3,9721129	5,94259003	7,1544143
3,07E+09	1	2,00040483	3,9929345	5,98399276	7,9607089
6,14E+09	1	1,98269421	3,991235	5,96705929	7,9578792
1,23E+10	1	1,99669997	3,9913488	5,98295263	7,9622563
2,46E+10	1	1,99295397	3,9834175	5,98176147	7,9442965
4,92E+10	1	0,60613932	25,966782	38,9921481	51,960076
9,83E+10	-	-	-	-	-

O gráfico da Figura 8.3 apresenta a visualização loglog do *speedup* relativo. Não é possível calcular o *speedup* relativo para as execuções com 98 milhões de partículas, pois o programa não executa em apenas uma máquina. Pode-se notar que valores bastante

significativos de *speedup* são obtidos para um número de máquinas a partir de quatro com um sistema constituído de 49 milhões de partículas. Uma possível explicação para esse fato é que com um menor número de máquinas há a sobrecarga de cada uma delas no processamento dos vetores, o que diminui com a adição de novas máquinas.

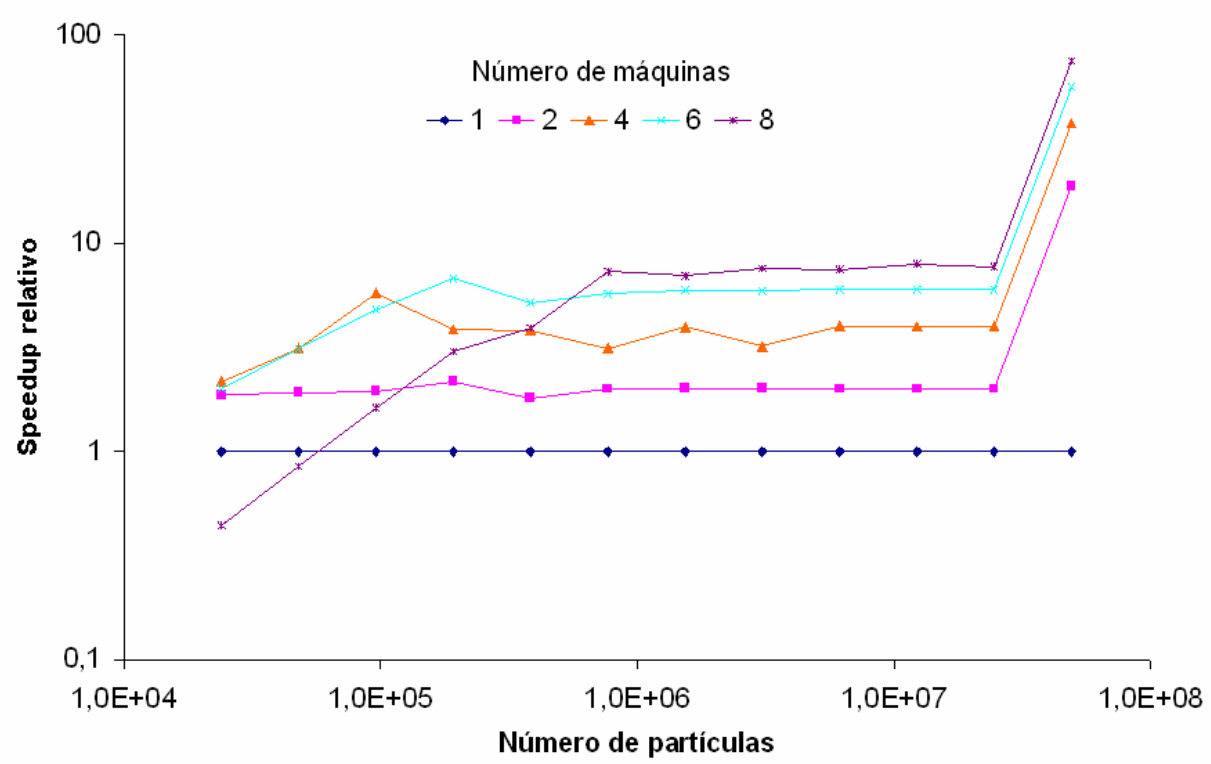

Figura 8.3 – *Speedup* relativo para 2, 4, 6 e 8

Para o caso da implementação MPI, também foram realizados testes em um ambiente heterogêneo. A medição dos tempos totais de execução e a determinação dos limites dos intervalos de tempo relativos às computações e comunicações do programa foram realizadas por meio da porta paralela.

O objetivo dos testes em um ambiente heterogêneo foi determinar empiricamente a distribuição de carga quando uma máquina mais rápida é agregada a um sistema paralelo já existente.

Nas figuras 8.4 a 8.6 está representado o tempo de execução, expresso em segundos, para um sistema com 24.576 milhões de partículas executado em duas máquinas mais lentas (node01 e node02) e uma máquina mais rápida (node102), nestes testes manteve-se constante a quantidade de tarefas executadas nas máquinas mais lentas, uma

tarefa por máquina, e aumentou-se progressivamente o número de tarefas atribuídas à máquina mais rápida (duas, quatro e seis tarefas).

A ferramenta PM^{2}P permite que sejam visualizados intervalos de tempo que delimitam marcações de eventos na aplicação, além do intervalo total de execução. Nos mapas de Gantt apresentados a seguir, estes intervalos de tempo são representados pelos traços verticais em cada uma das barras. Nas figuras 8.4, 8.5 e 8.6 podem-se observar quais eventos internos foram evidenciados pela marcação do intervalo de tempo.

A tabela 8.3 apresenta os valores, em segundos, relativos às medições dos instantes de tempo em que ocorrem determinados eventos internos da aplicação, que são mostrados na figura 8.4.

Tabela 8.3 - Instantes de tempo (s) em que ocorrem eventos internos da aplicação mostrados no mapa de Gantt da figura 8.1

Tarefas	Início execução	Inicialização		Cálculo ***Mx*** e ***My***		Comunicação	Fim execução
t1	0,00183	5,86537	5,86537	15,54931	170,41623	195,95149	196,13480
t3	0	5,86541	5,86541	15,54936	141,54837	144,00498	196,13482
t4	0,00076	5,86543	5,86543	15,54937	141,54791	144,00419	196,13495
t2	0,00191	5,86546	5,86546	15,54933	157,94273	178,02852	196,13489

Para ilustrar o modo como estes intervalos de tempo podem ser observados no mapa de Gantt apresentado pela ferramenta, pode-se verificar, na figura 8.4, o comportamento das linhas de eventos internos relativas à tarefa t2. Contando-se a partir da extremidade esquerda da figura, têm-se ao todo sete linhas verticais, sendo que as linhas iniciais e finais da barra delimitam o intervalo total de execução da aplicação. As linhas 4 e 5 delimitam o intervalo de tempo relativo a um dos cálculos dos vetores ***Mx*** e ***My*** do algoritmo, enquanto as linhas 5 e 6 delimitam o intervalo de tempo relativo a duas operações de comunicação.

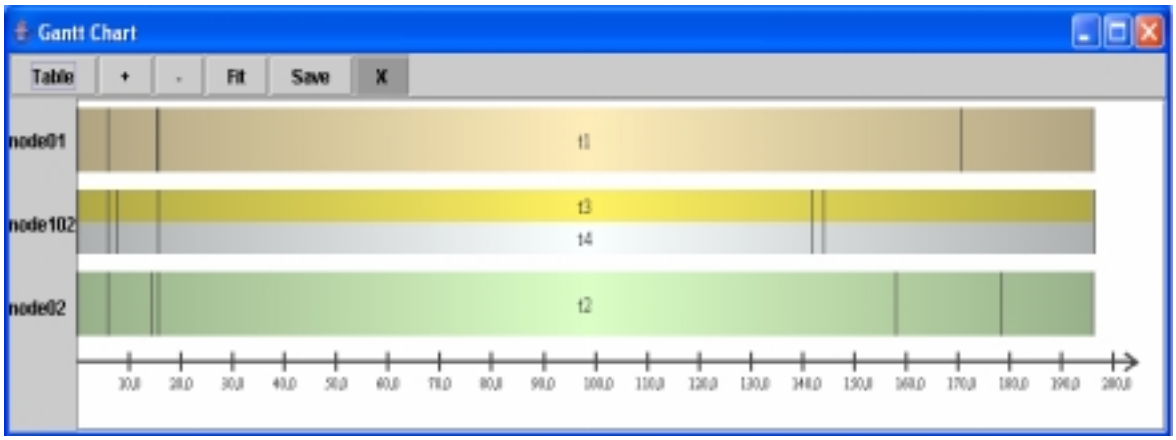

Figura 8.4 – Mapa de Gantt com os limites de tempo de execução e comunicação para 4 tarefas. Duas tarefas na máquina node102 e uma tarefa em cada uma das máquinas node01 e node02.

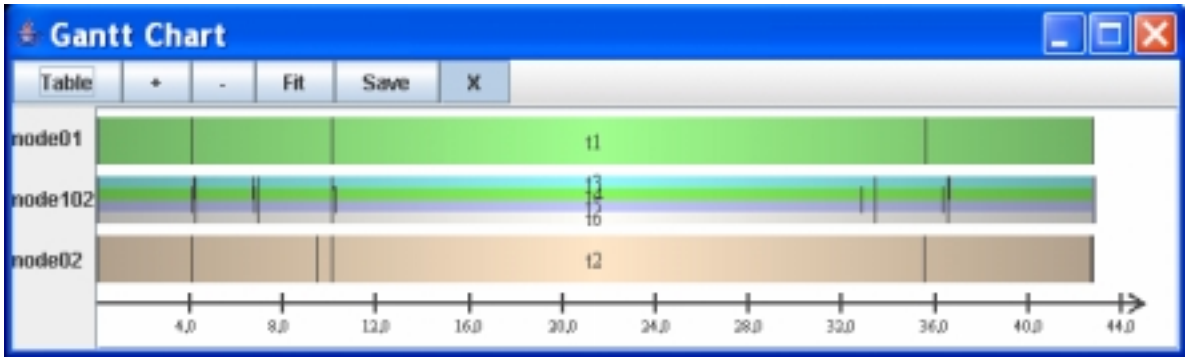

Figura 8.5 – Mapa de Gantt com os limites de tempo total de execução para 6 tarefas. Quatro tarefas na node102 e uma tarefa em cada uma das máquinas node01 e node02.

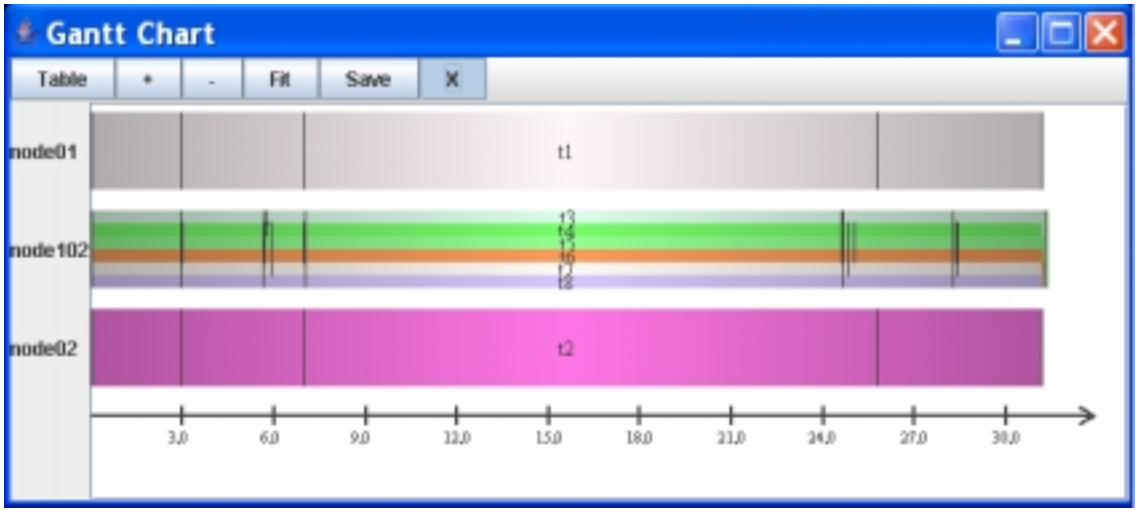

Figura 8.6 – Mapa de Gantt com os limites de tempo total de execução para 8 tarefas. Seis tarefas na node102 e uma tarefa em cada uma das máquinas node01 e node02.

Tabela 8.4 – Instantes de tempo (s) em que ocorrem eventos internos da aplicação mostrados no mapa de Gantt da figura 8.5

Tarefas	Início execução	Inicialização		Cálculo ***Mx*** e ***My***		Comunicação	Fim execução
t1	0,001830349	4,12319	4,12317	10,10305	35,58840	42,78231	42,78377
t3	0	4,18292	4,18292	10,10292	33,40351	36,64374	42,78761
t4	0,001536283	4,12309	4,12309	10,17717	33,45235	36,606374	42,78358
t5	0,002275291	4,20620	4,20620	10,10293	33,46896	36,53862	42,82085
t6	0,000798606	4,12311	4,12311	10,22700	32,88734	36,36347	42,78358
t2	0,001893326	4,12314	4,12314	10,10293	35,55075	42,70663	42,78365

Tabela 8.5 – Instantes de tempo (s) em que ocorrem eventos internos da aplicação mostrados no mapa de Gantt da figura 8.6

Tarefas	Início execução	Inicialização		Cálculo ***Mx*** e ***My***		Comunicação	Fim execução
t1	0,001808	2,93147	2,93147	6,96880	25,7936	31,16469	31,1802
t3	0	2,93143	2,93143	7,03308	24,63784	28,26166	31,26602
t4	0,002315	2,93144	2,93144	6,96867	24,82272	28,39665	31,18013
t5	0,00306	2,94732	2,94732	6,99991	24,65348	28,27637	31,31584
t6	0,000801	3,01040	3,01040	6,96879	25,03161	28,43935	31,27934
t7	0,001545	2,97056	2,97056	6,98336	24,60300	28,38464	31,18015
t8	0,003884	2,96063	2,96063	7,00986	24,81694	28,44606	31,23952
t2	0,002608	2,93222	2,93222	6,96942	25,79850	31,17939	31,18046

Pelos mapas de Gantt apresentados, comprova-se que o aumento de tarefas na máquina mais rápida, node102, contribui para a diminuição do tempo de execução da aplicação.

8.2.1.2 JIAJIA

Os dados mostrados a seguir referem-se à execução de uma iteração da implementação JIAJIA do algoritmo do integrador simplético em 1, 2, 4, 6 e 8 máquinas, com um número de partículas variando de 24 mil a 3,072 milhões. No decorrer dos testes, foi verificado que para 2 máquinas é possível executar a aplicação para até 12,288 milhões de partículas. No entanto, como não foi possível utilizar este mesmo número de partículas nas simulações com as demais quantidades de máquinas, optou-se por analisar os dados referentes apenas às execuções até 3,072 milhões de partículas para cada conjunto de máquinas. O JIAJIA foi utilizado exatamente como disponibilizado pelo Centro de Computação de Alto

Desempenho da Academia Chinesa de Ciências, nenhuma melhoria foi adicionada ao seu código.

A tabela 8.6 mostra os tempos totais de execução, em microssegundos, obtidos nas simulações realizadas.

O gráfico da Figura 8.7 exibe o comportamento destes valores para cada conjunto de máquinas, permitindo a visualização da relação entre número de máquinas, número de partículas e tempos de execução. Pode-se notar que o comportamento da relação entre número de máquinas e tempos de execução é o inverso do que acontece com a implementação MPI. Na implementação JIAJIA, à medida que o número de máquinas cresce, cresce também o tempo de execução para um mesmo número de partículas. Já no MPI, à medida que máquinas são acrescentadas ao *cluster*, o tempo de execução diminui. Esta diferença pode dever-se ao fato de que com o aumento do número de máquinas no *cluster*, o JIAJIA tem um custo maior no gerenciamento da memória compartilhada, o que leva a uma significativa perda de desempenho.

Tabela 8.6 – Tempos de execução em microssegundos para 1, 2, 4, 6 e 8 máquinas

Número de partículas	Número de máquinas				
	1	2	4	6	8
2,40E+04	43822,8884	108582,8042	162391,3095	129692,2947	160321,7516
4,80E+04	87352,5895	166116,5137	223650,2232	252762,0042	251106,3579
9,60E+04	158545,381	359689,1621	396389,3221	410600,2863	485104,3705
1,92E+05	352204,261	706409,0947	716618,9137	811818,5768	854865,3811
3,84E+05	616383,326	1157572,716	1415991,512	1586109,171	1750018,156
7,68E+05	1249875	2227258,206	2802595,301	3256104,421	3555224,522
1,54E+06	2763480,66	4926375,613	6005029,187	6746758,737	7308298,779
3,07E+06	5514750,92	10965069,61	12377611,86	13863278,48	15294584,72

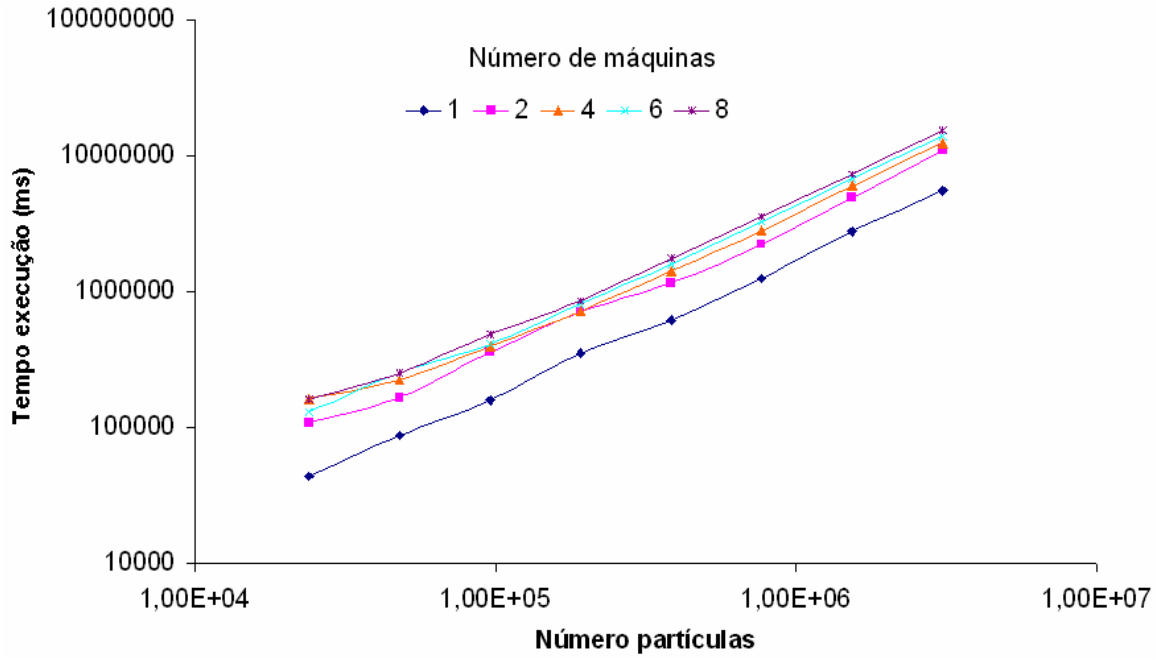

Figura 8.7 – Tempos de execução para 1, 2, 4, 6 e 8 máquinas

O *speedup* relativo obtido para o caso da implementação em memória compartilhada com o JIAJIA é mostrado na tabela 8.7.

Tabela 8.7 – *Speedup* relativo alcançado para 2, 4, 6 e 8 máquinas

Número de partículas	Número de máquinas				
	1	2	4	6	8
2,40E+04	1	0,403589581	0,269859813	0,337898936	0,273343373
4,80E+04	1	0,525851329	0,390576804	0,345592249	0,347870879
9,60E+04	1	0,440784427	0,399973895	0,386130712	0,326827361
1,92E+05	1	0,498583984	0,491480554	0,433846023	0,411999677
3,84E+05	1	0,532479142	0,435301569	0,388613431	0,352215389
7,68E+05	1	0,561172025	0,445970561	0,383855932	0,351560075
1,54E+06	1	0,560956142	0,460194376	0,409601227	0,37812913
3,07E+06	1	0,502938068	0,445542402	0,397795581	0,360568856

No gráfico da Figura 8.8 é mostrado o comportamento deste *speedup* e observa-se uma aparente dificuldade do JIAJIA em gerenciar a memória compartilhada distribuída, pois a paralelização para esta implementação não adicionou nenhum ganho ao desempenho da aplicação, acarretando, na realidade, uma queda no desempenho em relação à execução em apenas uma máquina.

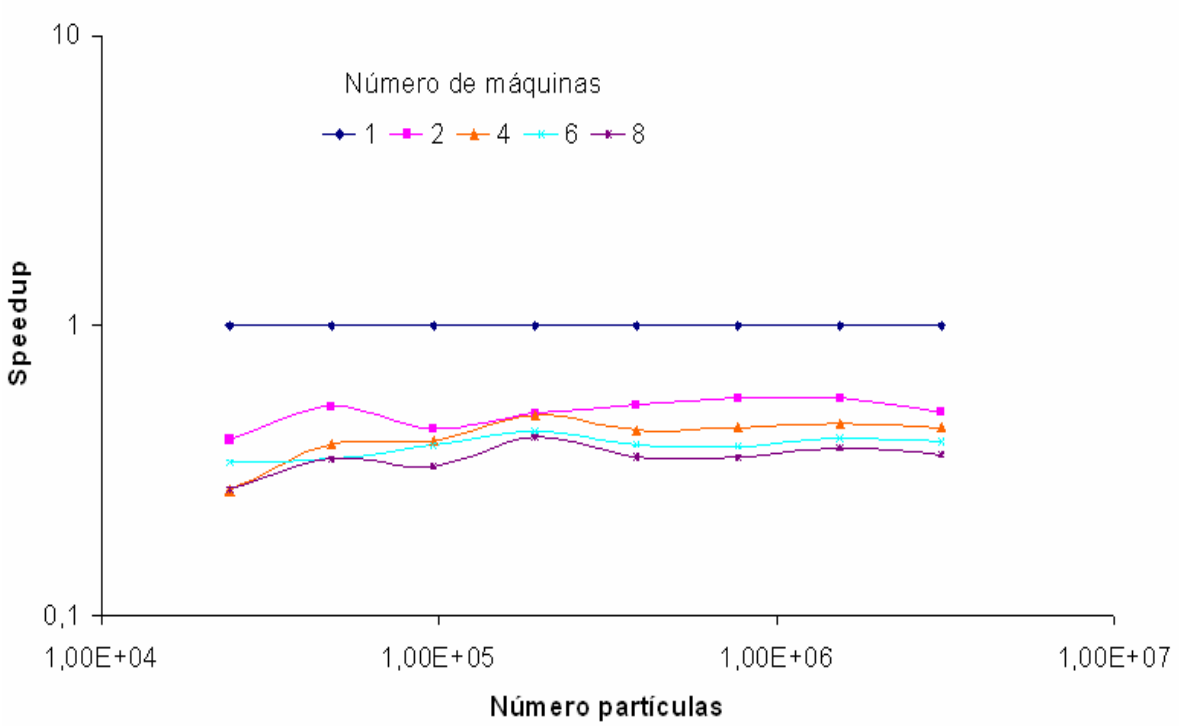

Figura 8.8 – *Speedup* relativo alcançado para 2, 4, 6 e 8 máquinas.

8.2.2 Tempos de comunicação

8.2.2.1 MPI

Como para o tempo de execução, os dados mostrados referem-se às execuções da implementação MPI em 1, 2, 4, 6 e 8 máquinas, com um número de partículas que variou de 24 mil a 98304 milhões de partículas. Para medir o tempo de comunicação nesta implementação, foi utilizado um instrumento de medição ligado à porta paralela de cada uma das máquinas descrito e implementado em [22]. A tabela 8.8 apresenta os tempos de comunicação obtidos nos testes realizados.

No gráfico da Figura 8.9 observa-se o comportamento do tempo de comunicação para 1, 2, 4, 6 e 8 máquinas.

Do gráfico da Figura 8.9 pode-se observar que o custo de comunicação é inversamente proporcional ao número de máquinas para uma mesma quantidade de partículas. Acreditamos que isso se deve ao fato de que com um conjunto maior de máquinas, o subconjunto de dados para cada uma das máquinas é menor e, portanto, menos tempo é requerido para a troca destes dados.

Tabela 8.8 – Tempos de comunicação para 1, 2, 4, 6 e 8 máquinas.

Número de partículas	Número de máquinas				
	1	2	4	6	8
2,40E+04	0,03806786	0,019020812	0,00947029	0,006301612	0,004722264
4,80E+04	0,07609212	0,03813446	0,01894464	0,012603224	0,009458376
9,60E+04	0,15268051	0,076242332	0,02079078	0,025206448	0,018953816
1,92E+05	0,33729221	0,152613648	0,07614843	0,050412896	0,038013088
3,84E+05	0,61116475	0,337205804	0,15273544	0,101769108	0,076241304
7,68E+05	1,22664211	0,609793828	0,38156794	0,2034017	0,152841952
1,04E+09	2,45012186	1,220009932	0,61040044	0,406792856	0,337439672
3,07E+09	4,91992794	2,440534996	1,52574372	0,812924844	0,609743212
6,14E+09	9,83289098	4,894351668	2,44108700	1,62652006	1,220163784
1,23E+10	19,6661926	9,759941804	4,89741934	3,263735528	2,44234366
2,46E+10	39,1078390	19,63933299	9,76726994	6,504736012	4,893706252
4,92E+10	576,9836505	-	19,53673782	13,03206138	9,76623866
9,83E+10	-	-	-	26,04627676	19,54737437

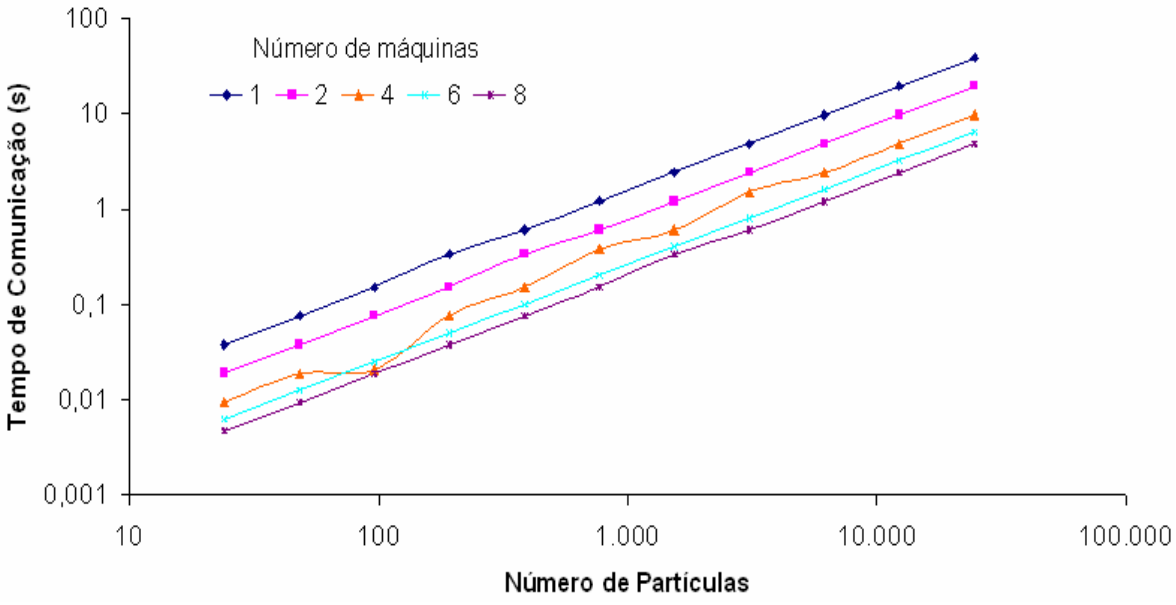

Figura 8.9 – Tempos de comunicação para 1, 2, 4, 6 e 8 máquinas

8.2.2.2 JIAJIA

A tabela 8.9 contem os dados obtidos sobre o tempo de comunicação das execuções em 2, 4, 6 e 8 máquinas.

Tabela 8.9 – Tempos de comunicação para 2, 4, 6 e 8 máquinas na implementação JIAJIA

Número de partículas	Número de máquinas			
	2	4	6	8
2,40E+04	689,8526	1.517,6757	1.655,6463	2.207,5284
4,80E+04	1.379,7052	1.931,5873	2.483,4694	2.483,4694
9,60E+04	551,8821	3.449,2631	4.690,9978	44.150,5684
1,92E+05	3.449,2631	1.517,6757	8.830,1136	7.726,3494
3,84E+05	2.897,3810	3.863,1747	3.863,1747	4.690,9978
7,68E+05	4.690,9978	12.417,3473	7.726,3494	15.452,6989
1,54E+06	39.597,5410	39.045,6589	45.806,2147	51.049,0947
3,07E+06	144.800,0673	157.010,4589	166.116,5136	171.911,2757

O gráfico loglog da Figura 8.10 mostra o tempo de comunicação medido na implementação em memória compartilhada distribuída para 2, 4, 6 e 8 máquinas.

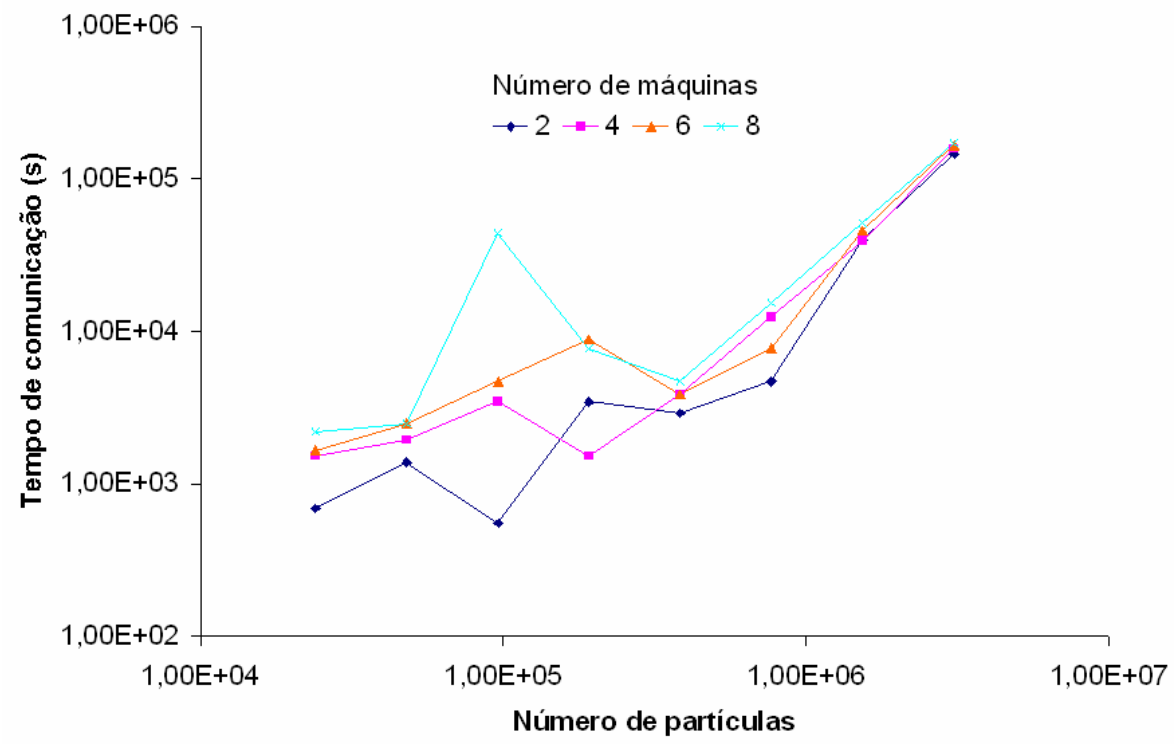

Figura 8.10 – Tempos de comunicação para 2, 4, 6 e 8 máquinas

Como se pode notar do gráfico 8.10, o tempo de comunicação cresce à medida que o número de máquinas aumenta. O acréscimo de máquinas ao sistema paralelo, acarreta em um maior custo de gerência da memória compartilhada distribuída, o que acreditamos ser a principal causa deste aumento no custo de comunicação.

9. CONCLUSÃO

Neste trabalho é apresentada a avaliação do desempenho de uma implementação do integrador simplético em dois paradigmas de uso corrente para programação paralela, que não são restritos apenas a profissionais da área de Computação. Pesquisadores como físicos, matemáticos, engenheiros, biólogos também têm necessidade de computação de alto desempenho para viabilizar suas simulações.

Para quantificar o desempenho da execução paralela das duas implementações distintas, foram realizados as medições dos tempos de execução e o cálculo do *speedup* obtido, variando-se o número de máquinas participantes do *cluster* e o tamanho do problema, por meio da variação do número de partículas do sistema simulado. Para mensurar o tempo de execução, foi inserida em cada um dos códigos-fonte a instrução *assembly* rdtsc.

À medida que o número de partículas foi sendo aumentado em cada uma das implementações, observou-se uma limitação na implementação JIAJIA em relação ao número de partículas que podem ser consideradas na simulação. De um modo geral, para qualquer número de máquinas, só foi possível simular um sistema de até três milhões de partículas. Na implementação por troca de mensagens, o número de partículas constituintes do sistema simulado chegou até cem milhões de partículas.

Dos resultados obtidos observou-se que a implementação do integrador simplético em troca de mensagens apresenta melhores resultados em termos de tempo total de execução do que a implementação em memória compartilhada distribuída. Uma vez que o acesso à memória remota envolve chamadas ao *kernel*, trocas de contexto, buferização, além da latência do *software* de comunicação e do *hardware*, o *overhead* envolvido na manutenção da memória compartilhada distribuída em DSMs baseados em *software*, como é o JIAJIA, parece ser o fator que contribui para um desempenho inferior da implementação do algoritmo por memória compartilhada distribuída em relação à implementação por troca de mensagens.

Um outro aspecto relevante em favor da implementação MPI é a facilidade encontrada para a construção do código, apesar da necessidade de explicitar as comunicações. Além disso, a comunidade de usuários desta biblioteca de troca de

mensagens é bem mais ampla que a do JIAJIA o que facilita encontrar documentação e exemplos.

Considerando que, de um modo geral, há uma tendência de utilização de ambientes heterogêneos formados por máquinas das mais diversas configurações e ligadas em redes locais, para o caso da implementação MPI foram realizadas medições de tempo de execução em um ambiente heterogêneo. A utilização da ferramenta PM^2P para medição de intervalos de tempo no sistema paralelo heterogêneo possibilitou um balanceamento de carga empírico entre as máquinas do sistema, contribuindo para uma melhor distribuição do trabalho entre elas e conseqüente melhora no desempenho em termos de tempo total de execução do programa.

Como trabalhos futuros, os resultados da monitoração de trechos do código paralelo do integrador simplético poderiam ser utilizados para uma identificação dos gargalos, de modo a possibilitar uma melhora na forma como a paralelização é feita. Testes podem ser realizados substituindo-se, por exemplo, a instrução *AllReduce* da implementação MPI por *sends* e *receives* visando à comparação do desempenho das diretivas de comunicação. Além disso, poderia ser ampliado o *cluster* heterogêneo visando ao aprofundamento do estudo sobre balanceamento de carga empírico com o auxílio de ferramentas de visualização.

REFERÊNCIAS

[1] Parhami, B.; Introduction to Parallel Processing - Algorithms and Architectures. Plenum Series in Computer Science; Kluwer Academic Publishers; University of California at Santa Barbara; Santa Barbara, California 2002.

[2] Fernandes , A. L.; Current Architectures for Parallel Processing. In: ICCA'04, Departamento de Informática**,** Universidade do Minho, 2004.

[3] Gobbert, M. K.; Configuration and Performance of a Beowulf cluster for Large Scale Scientific Simulations. Computing in Science and Engineering, pp. 14-26, 2005.

[4] McCaughan, D.; Foundations of Parallel Programming. EXCELerate'04; York University. Disponível em: <http://www.hpcc.ecs.soton.ac.uk >. Acesso em: 2005

[5] Culler, D.; Singh, J.P.; Gupta, A. **Parallel Computer Architecture – A Hardware/Software Approach**; University of California; Berkeley, California 1997.

[6] Foster, I. **Designing and Building Parallel Programs**. Addison-Wesley; 1995. Disponível em: http://www-unix.mcs.anl.gov/dbpp/. Acesso em: 2006.

[7] Microsoft Corporation; Microsoft Solution Guide for Migrating High Performance Computing (HPC) Applications from UNIX to Windows; Junho 2004. Disponível em: <http://www.microsoft.com/technet/itsolutions/cits/interopmigration/unix/hpcunxwn/prefhpc.mspx>. Acesso em: 2006.

[8] Flynn, M. J.; Rudd K.W.; Parallel Architectures. ACM Computing Surveys, 28(1):67-70, March 1996.

[9] Introduction to Parallel Computing. Disponível em: <http://www.llnl.gov/computing/tutorials/parallel_comp/>. Acesso em: Novembro de 2006.

[10] Top 500 Supercomputer Sites. Disponível em: <http://www.top500.org>. Acesso em: Novembro de 2006.

[11] Turner, D. Introduction to Parallel Computing and Cluster Computers; Ames Laboratory. Disponível em: <http://www.scl.ameslab.gov/Projects/parallel_computing/>. Acesso em: Novembro de 2006.

[12] Lynch, N. A.; Distributed Algorithms, Morgan Kaufman, 1996.

[13] Kai Hwang, Zhiwei Xu; **Scalable Parallel Computing**, 1ª Edição, Mc Graw Hill, 1998.

[14] Joe, et al; **Concurrent Programming in ERLANG**; 2ª edição; Prentice Hall; 1996.

[15] Mucci, P. J.; Performance Analysis and HPC; In: High Performance Computing Summer School at PDC ; University of Tennessee, Knoxville 13, 18 de Agosto de 2004.

[16] Performance Evaluation and Optimization of High Performance Computing. Disponível em: http://www.ncic.ac.cn/paper/paper_other50.htm. Acesso em: Novembro 2005.

[17] Hollingsworth, Jeffrey K., Lumpp, James E., Miller, Barton P.; Techniques for Performance Measurement of Parallel Programs; in: Casavant, Th.L., Tvrdik, P., Plail, F., Parallel Computers: Theory and Practices; IEEE Computer Society Press, 1995.

[18] Ferreira, R. R., Caracterização de desempenho de uma aplicação paralela do método dos elementos finitos em ambientes heterogêneos, Dissertação de Mestrado, CIC/UnB, 2006.

[19] VILLA VERDE, F. R.; PFITSCHER, G. H.; VIANA, D. M., Performance Characterization of a Parallel Code Based on Domain Decomposition on PCs Cluster. In: I2TS´2006 – 5th International Information and Telecommunication Technologies Symposium, Cuiabá, MT, Brazil, 2006.

[20] Calzarossa, M., Massari, L. e Tessera. D.; A Methodology Towards Automatic Performance Analysis of Parallel Applications, Journal: Parallel Computing, Vol. 30, n. 2, pp. 211-223, 2004.

[21] Kergommeaux, J.C., Maillet, É, Vincent, J-M. Monitoring parallel programs for performance tuning in cluster environments.

[22] Haridasan, M., Pfitscher, G. H.; PM^2P: A tool for performance monitoring of message passing applications in COTS PC clusters; Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03), 2003.

[23] UCLA Academic Technologies Services; An Introduction to Parallel Computing on Clusters of Machines; Disponível em: <http://www.ats.ucla.edu/at/hpc/parallel_computing/default.htm>. Acesso em: 2005.

[24] Introduction to HPC; University of Greenwich; 1995; Disponível em: <http://www.gre.ac.uk/~selhpc/pvm_course/hpc_intro/hpc_intro.html>. Acesso em: 2005.

[25] Argonne National Laboratory; The Message Passing Interface (MPI) Standard. Disponível em: <http://www-unix.mcs.anl.gov/mpi/>. Acesso em: Novembro de 2006.

[26] Gropp, W., Lusk, E.; Why are PVM and MPI so different?. Mathematics and Computer Science Division – Argone National Laboratory

[27] Al Beguelin, A. et al. PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing; Massachusetts Institute of Technology; 1994. Disponível em: <http://www.csm.ornl.gov/pvm/>. Acesso em: 2005.

[28] Parallel Virtual Machine version 3. Disponível em: <http://www.netlib.org/pvm3/>

[29] Villa Verde, F. R.; Solução paralela em clusters PCs de um código de Elementos Finitos aplicado à elasticidade linear; Dissertação de mestrado em Engenharia Mecânica; Universidade de Brasília; Brasília-DF, Novembro 2004.

[30] LI, K., HUDAK, P. Memory Coherence in Shared Virtual Memory Systems; ACM Transactions on Computer Systems, New York, v. 7, n. 4, p. 321-359, Novembro 1989.

[31] Melo, A. C. M. A.; Sistemas Distribuídos; Mestrado em Informática; 2003; Universidade de Brasília. Disponível em: <wwwhttp://www.cic.unb.br/~albamm/>. Acesso em: 2005.

[32] Park, I.; Parallel programming methodology and environment for the shared memory programming model, Dezembro 2000, Purdue University

[33] Walpole, J.; Distributed Shared Memory; Department of Computer Science & Engineering OGI/OHSU.

[34] MILUINOVIC, V., STENSTRÖN; P.; Scanning the Issue: Special Issue on Distributed Shared Memory Systems; Proceedings of the IEEE. March 1999, vol. 87, Nº 3.

[35] Brazos Project. Disponível em: <http://capricorn.csl.cornell.edu/Brazos/>. Acesso em: 2004.

[36] JIAJIA Distributed Shared Memory Project; Institute of Computing Technology, CAS; Disponível em: <http://www.ict.ac.cn/chpc/dsm/>. Acesso em: 2004.

[37] Monnerat, L., Bianchini, R.; The ADSM Project; Universidade Federal do Rio de Janeiro; Disponível em: <http://www.cos.ufrj.br/~ricardo/adsm.html>. Acesso em: 2004.

[38] Javid; Distributed Shared Memory Home Pages; Disponível em: <http://www.ics.uci.edu/~javid/dsm.html>. Acesso em: 2004.

[39] Weiwu H., Weisong S., Zhimin T.; The JIAJIA Software DSM System; Center of High Performance Computing, Institute of Computing Technology, Chinese Academy of Sciences; January, 1998.

[40] The High Availability Linux Project. Disponível em: <http://www.linux-ha.org/>. Acesso em: 2004.

[41] Moab Cluster Suíte. Disponível em: <http://www.clusterresources.com/pages/products/moab-cluster-suite.php>. Acesso em: 2005.

[42] Maui Cluster Scheduler. Disponível em: <http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php>. Acesso em: 2005.

[43] The Linux Virtual Server. Disponível em: <http://www.linuxvirtualserver.org/>. Acesso em: 2005.

[44] Brown, R. G.; Engineering a Beowulf-style Compute Cluster; Duke University Physics Department; May 24, 2004.

[45] Beowulf.org. Disponível em: <http://www.beowulf.org/>. Acesso em: 2006.

[46] Omelyan I.P., Mryglod I.M., Folk R.; Optimized Forest-Ruth- and Suzuki-like algorithms for integration of motion in many-body systems, Computer Phys. Comm., 2002, V.146, No 2.- P. 188-202.

ANEXO I – CÓDIGO-FONTE DA IMPLEMENTAÇÃO EM MPI

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include "forca.h"

#define CSI 0.1644986515575760
#define LAMBDA -0.2094333910398989
#define XI 0.1235692651138917
#define frequenciaClock 1900

int main (int argc, char **argv)
{

	FILE *dados, *resultados, *dadosConferencia;
	char nomeArquivo[30],nomeArquivo1[30];
	float randomico;//número randômico para a geração do vetor velocidade
	int i, j, numProcessos;
	int tamanhoVetores, rank, n;
	double dt, tf, ts, tempo, tc, e0, energia, Mx, MxTotal, My, MyTotal,
energiaCineticaTotal;
	double pInicial = 2.02;//velocidade inicial
	long semente = 1;

	double *r, *p, *f; //r = posicao, p = velocidade e f = forca.


	MPI_Init (&argc, &argv);

	//numero do processo que esta sendo executado
	MPI_Comm_rank (MPI_COMM_WORLD, &rank);

	//numero total de processos.
	MPI_Comm_size (MPI_COMM_WORLD, &numProcessos);

	//leitura dos valores de dt, tf e ts de um arquivo
	if ((dados = fopen("data.in", "r")) == NULL)
	{
		fprintf(stderr, "Nao é possível abrir o arquivo dados.in\n");
	}
	fscanf(dados, "%lf\n", &dt); //
	fscanf(dados, "%lf\n", &tf); //tempo final do sistema
	fscanf(dados, "%lf\n", &ts); //
	fscanf(dados, "%d\n", &n); //número de partículas do sistema
	fscanf(dados, "%d\n", &tamanhoVetores); //tamanho do vetor a ser alocado
	fclose(dados);

	tempo = 0.0;
	tc=0.0; //contador que determina de quanto em quanto tempo é escrito no arquivo

	if(rank == 0)
	{
	printf("************************** MPI ***************************** \n");
```

```
        printf("\t\tInicializacao realizada com sucesso.\n\t\tSistema com %d particulas. \n",
n);
                printf("\t\tTamanho vetores: %d\t\t \n", tamanhoVetores);
           printf("********************************************************************* \n");
        }

        //alocação de vetores
        r = (double *) malloc (tamanhoVetores * sizeof (double));
        p = (double *) malloc (tamanhoVetores * sizeof (double));
        f = (double *) malloc (tamanhoVetores * sizeof (double));


        //inicializa vetor posição e vetor velocidade
        for(i=0;i<tamanhoVetores;i++)
        {
                r[i] = 0.0;
                randomico = ran0(&semente);//funçãoo que gera números aleatórios
                p[i] = randomico * 2*pInicial - pInicial;
        }

        // cálculo da energia inicial
        //e0 = calcula_energia(r,p,n,tempo);

        double energiaCinetica = 0.0;
        //cálculo energia cinética parcial para cada uma das tarefas
        for(i=0; i<tamanhoVetores; i++)
        {
                energiaCinetica = energiaCinetica + p[i]*p[i]/2.0; //massa=1
        }
        //energia cinética total
        MPI_Allreduce (&energiaCinetica, &energiaCineticaTotal, 1, MPI_DOUBLE,
MPI_SUM, MPI_COMM_WORLD);

        //cálculo Mx e My
        Mx=0.0;
   My=0.0;

        for (i=0;i<tamanhoVetores;i++)
        {
                Mx += cos(r[i]);
                My += sin(r[i]);
        }

        MPI_Allreduce (&Mx, &MxTotal, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
        MPI_Allreduce (&My, &MyTotal, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);


        //cÃ¡lculo energia potencial
        double energiaPotencial = 0.0;
        energiaPotencial = 1.0/2.0 * ((double)n - MxTotal*MxTotal/(double)n -
MyTotal*MyTotal/(double)n);

        //energia total
```

```
        e0 = (energiaCineticaTotal + energiaPotencial)/(double)n;
        printf("Processo %d - energia = %lf \n",rank, e0);

        while(tempo < tf)
        {
                Mx=0.0;
        My=0.0;
                for (i=0;i<tamanhoVetores;i++)
                {
                        f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)n;
                        p[i]=p[i]+CSI*dt*f[i];
                        r[i]=r[i]+(1.0-2.0*LAMBDA)*dt*p[i]/2.0;

                        Mx += cos(r[i]);
                        My += sin(r[i]);
                }

        MxTotal=0.0;
        MyTotal=0.0;

                MPI_Allreduce (&Mx, &MxTotal, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
                MPI_Allreduce (&My, &MyTotal, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);


                Mx=0.0;
        My=0.0;
                for (i=0;i<tamanhoVetores;i++)
                {
                        f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)n;
                        p[i]=p[i]+XI*dt*f[i];
                        r[i]=r[i]+LAMBDA*dt*p[i];

                        Mx += cos(r[i]);
                        My += sin(r[i]);
                }


        MxTotal=0.0;
        MyTotal=0.0;

                MPI_Allreduce (&Mx, &MxTotal, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
                MPI_Allreduce (&My, &MyTotal, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

                Mx=0.0;
        My=0.0;
                for (i=0;i<tamanhoVetores;i++)
                {
                        f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)n;
                        p[i]=p[i]+(1.0-2.0*(XI+CSI))*dt*f[i];
                        r[i]=r[i]+LAMBDA*dt*p[i];
```

```
                        Mx += cos(r[i]);
                        My += sin(r[i]);
                }


        MxTotal=0.0;
        MyTotal=0.0;

                MPI_Allreduce (&Mx, &MxTotal, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
                MPI_Allreduce (&My, &MyTotal, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);


                Mx=0.0;
        My=0.0;
                for (i=0;i<tamanhoVetores;i++)
                {
                        f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)n;
                        p[i]=p[i]+XI*dt*f[i];
                        r[i]=r[i]+(1.0-2.0*LAMBDA)*dt*p[i]/2.0;

                        Mx += cos(r[i]);
                        My += sin(r[i]);
                }

        MxTotal=0.0;
        MyTotal=0.0;

                MPI_Allreduce (&Mx, &MxTotal, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
                MPI_Allreduce (&My, &MyTotal, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

                for (i=0;i<tamanhoVetores;i++)
                {
                        f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)n;
                        p[i]=p[i]+CSI*dt*f[i];
                }

                tempo = tempo + dt;
                tc = tc + dt;

                if(tc >= ts)
                {
                        energia = calcula_energia(r,p,n,tempo);
                        tc = 0;
                }
        }

        printf("[simpletico] Fim execucao.\n");
        printf("[simpletico] Iniciando gravacao de arquivos..... ");

        //Abertura de arquivos
        sprintf (nomeArquivo, "resultSemChamada%d.out", rank);
```

```
	sprintf (nomeArquivo1, "dadosConferencia%d.out", rank);

	if ((resultados = fopen(nomeArquivo, "w+")) == NULL)
	{
	fprintf(stderr, "Nao eh possivel criar o arquivo resultSemChamada.out\n");
	}


	if ((dadosConferencia = fopen(nomeArquivo1, "w+")) == NULL)
	{
	fprintf(stderr, "Nao eh possivel criar o arquivo dadosConferencia.out\n");
	}

	fprintf(resultados, "---------- SimplÃ©tico Sequencial - SEM chamada de função --
---------- \n\n");


	//impressao dos dados calculados
	for (i=0;i<tamanhoVetores;i++)
	{
		fprintf(dadosConferencia, "r[%d]: %lf \n", i, r[i]);
		fprintf(dadosConferencia, "p[%d]: %lf \n", i, p[i]);
		fprintf(dadosConferencia, "f[%d]: %lf \n", i, f[i]);
	}

	fclose(resultados);
	fclose(dadosConferencia);

	free(p);
	free(r);
	free(f);

	p = NULL;
	r = NULL;
	f = NULL;

	MPI_Finalize ();
	return 0;
}
```

ANEXO II – CÓDIGO-FONTE DA IMPLEMENTAÇÃO JIAJIA

```
#include <jia.h>
#include <stdio.h>
#include "forca.h"


#define N 32000 //Numero de partículas do sistema

#define CSI 0.1644986515575760
#define LAMBDA -0.2094333910398989
#define XI 0.1235692651138917
#define frequenciaClock 1900

double dt, tf, ts, e1, tempo, tc, en, en0, de, ppi;
double (*r),(*p),(*f), (*Mx),(*My);
long semente = 1;


/**
 * Faz a leitura do arquivo de dados e inicializa as variaveis
 */
void inicializaDados()
{
        int i,j;
        double pInicial = 2.02;//velocidade inicial
        float randomico;//número randômico para a geração do vetor velocidade
        FILE *dados;

        if (jiapid==0)
        {
                //leitura dos valores de dt, tf e ts de um arquivo
                if ((dados = fopen("data.in", "r")) == NULL)
                {
                fprintf(stderr, "Nao eh possivel abrir o arquivo dados.in\n");
                }
                fscanf(dados, "%lf\n", &dt); //
                fscanf(dados, "%lf\n", &tf); //tempo final do sistema
                fscanf(dados, "%lf\n", &ts); //
                //fscanf(dados, "%d\n", &n); //nÃºmero de partÃculas do sistema
                fclose(dados);


                for (i=0;i<N;i++)
                {
                        r[i]=0.0;
                        randomico = ran0(&semente);
                        p[i] = randomico * 2*pInicial - pInicial;
                }

                tempo = 0.0;
                tc=0.0; // determina de quanto em quanto tempo é escrito no arquivo
    }
    printf("************************** JIAJIA ****************************** \n");
```

```
    printf("\t\tInicializacao realizada com sucesso.\n\t\tSistema com %d particulas. \n",N);
    printf("********************************************************************* \n");
}

/**
 * Integrador simpletico de 4a. ordem
 */
void integradorSimpletico()
{
        int i,j;
        int start, end;
        double temp, energiaCinetica = 0.0, energiaPotencial = 0.0, e0=0.0;
        double MxTotal, MyTotal;


        start=(N/jiahosts)*jiapid;
        end=start+(N/jiahosts);

        int restoDivisao = N%jiahosts;

        if(restoDivisao != 0 && jiapid == jiahosts -1)
                end = end + restoDivisao;

        //cálculo energia cinética
        for(i=0; i<N; i++)
        {
                energiaCinetica = energiaCinetica + p[i]*p[i]/2.0; //massa=1
        }

        MxTotal=0.0;
        MyTotal=0.0;

        for (i=start;i<end;i++)
        {
                Mx[i] = cos(r[i]);
                My[i] = sin(r[i]);

   }

  jia_barrier();

  if(jiapid==0)
  {
                for (i=0;i<N;i++)
                {
                        MxTotal = MxTotal + Mx[i];
                        MyTotal = MyTotal + My[i];

            }
  }
   jia_barrier();

        // cálculo energia potencial
        energiaPotencial = 1.0/2.0 * ((double)N - MxTotal*MxTotal/(double)N -
MyTotal*MyTotal/(double)N);
```

```
        //energia total
        e0 = (energiaCinetica + energiaPotencial)/(double)N;
        printf("energia inicial do sistema: %lf \n", e0);

        jia_barrier();

        printf("Iteracoes: ");
        while(tempo < tf)
        {
                if(jiapid==0)
                        printf("%da. ",j);

                for (i=start;i<end;i++)
                {
                        f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)N;
                        p[i]=p[i]+CSI*dt*f[i];
                        r[i]=r[i]+(1.0-2.0*LAMBDA)*dt*p[i]/2.0;

                        Mx[i] = cos(r[i]);
                        My[i] = sin(r[i]);
                }

          jia_barrier();

if(jiapid == 0)
{
           MxTotal=0.0;
                MyTotal=0.0;

           for (i=0;i<N;i++)
                {
                        MxTotal = MxTotal + Mx[i];
                        MyTotal = MyTotal + My[i];

           }
}
jia_barrier();



                for (i=start;i<end;i++)
                {
                        f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)N;
                        p[i]=p[i]+XI*dt*f[i];
                        r[i]=r[i]+LAMBDA*dt*p[i];

                        Mx[i] = cos(r[i]);
                        My[i] = sin(r[i]);
                }

           jia_barrier();

if(jiapid == 0)
{
```

```
                MxTotal=0.0;
                MyTotal=0.0;

                for (i=0;i<N;i++)
                {
                        MxTotal = MxTotal + Mx[i];
                        MyTotal = MyTotal + My[i];

        }
}
jia_barrier();


                for (i=start;i<end;i++)
                {
                        f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)N;
                        p[i]=p[i]+(1.0-2.0*(XI+CSI))*dt*f[i];
                        r[i]=r[i]+LAMBDA*dt*p[i];

                        Mx[i] = cos(r[i]);
                        My[i] = sin(r[i]);
                }

        jia_barrier();

if(jiapid == 0)
{
                MxTotal=0.0;
                MyTotal=0.0;

                for (i=0;i<N;i++)
                {
                        MxTotal = MxTotal + Mx[i];
                        MyTotal = MyTotal + My[i];

        }
}jia_barrier();


                for (i=start;i<end;i++)
                {
                        f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)N;
                        p[i]=p[i]+XI*dt*f[i];
                        r[i]=r[i]+(1.0-2.0*LAMBDA)*dt*p[i]/2.0;

                        Mx[i] = cos(r[i]);
                        My[i] = sin(r[i]);
                }

        jia_barrier();

if(jiapid == 0)
{
                MxTotal=0.0;
                MyTotal=0.0;
```

```
                for (i=0;i<N;i++)
                {
                        MxTotal = MxTotal + Mx[i];
                        MyTotal = MyTotal + My[i];

        }
}jia_barrier();


                for (i=start;i<end;i++)
                {
                        f[i] = -(sin(r[i])*MxTotal +cos(r[i])*MyTotal)/(double)N;
                        p[i]=p[i]+CSI*dt*f[i];
                }


                tempo = tempo + dt;
                tc = tc + dt;


                if(tc >= ts)
                {
                        energia = calcula_energia(r,p,n,tempo);
                        tc = 0;
                }

        }
}



main(int argc,char **argv)
{
        int i,j;
        float t1,t2;
        FILE *dadosConferencia;

        jia_init(argc,argv);

        r=(double (*))jia_alloc3(N*sizeof(double),(N*sizeof(double))/jiahosts,0);
        p=(double (*))jia_alloc3(N*sizeof(double),(N*sizeof(double))/jiahosts,0);
        f=(double (*))jia_alloc3(N*sizeof(double),(N*sizeof(double))/jiahosts,0);
        Mx=(double (*))jia_alloc3(N*sizeof(double),(N*sizeof(double))/jiahosts,0);
        My=(double (*))jia_alloc3(N*sizeof(double),(N*sizeof(double))/jiahosts,0);

        jia_barrier();

        inicializaDados();

        jia_barrier();
        //jia_startstat();
        t1=jia_clock();
```

```
	integradorSimpletico();

	jia_barrier();
	t2=jia_clock();

	//grava os dados calculados
	if(jiapid==0)
	{
		printf("\nTempo total de execucao == %10.2f seconds\n", t2-t1);

		if ((dadosConferencia = fopen("dadosConferencia.out", "w+")) == NULL)
		{
		fprintf(stderr, "Nao eh possivel criar o arquivo dadosConferencia.out\n");
		}

		for (i=0;i<N;i++)
		{
			fprintf(dadosConferencia, "r[%d]: %lf \n", i, r[i]);
			fprintf(dadosConferencia, "p[%d]: %lf \n", i, p[i]);
			fprintf(dadosConferencia, "f[%d]: %lf \n", i, f[i]);
		}
	}

	jia_exit();
}
```