JOSÉ LUIZ DE SOUZA GOMES

PARALELIZAÇÃO DE ALGORITMO DE SIMULAÇÃO DE MONTE CARLO PARA A ADSORÇÃO EM SUPERFÍCIES HETEROGÊNEAS BIDIMENSIONAIS

MARINGÁ

2009

JOSÉ LUIZ DE SOUZA GOMES

PARALELIZAÇÃO DE ALGORITMO DE SIMULAÇÃO DE MONTE CARLO PARA A ADSORÇÃO EM SUPERFÍCIES HETEROGÊNEAS BIDIMENSIONAIS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Ronaldo Augusto de Lara Gonçalves

MARINGÁ

2009

Dados Internacionais de Catalogação-na-Publicação (CIP)
(Biblioteca Central - UEM, Maringá – PR., Brasil)

G633p

Gomes, José Luiz de Souza
Paralelização de algoritmo de simulação de Monte Carlo para a adsorção em superfícies heterogêneas bidimensionais / José Luiz de Souza Gomes. -- Maringá : [s.n.], 2009.
88 p. : il. color.

Orientador : Prof. Dr. Ronaldo A. L. Gonçalves.
Dissertação (mestrado) - Universidade Estadual de Maringá, Programa de Pós-graduação em Ciência da Computação, 2009.

1. Paralelização - Particionamento de domínio. 2. Algoritmo de Monte Carlo. 3. Adsorção - Simulação. 4. Clusters - Computação. I. Universidade Estadual de Maringá, Programa de Pós-graduação em Ciência da Computação. II. Título.

CDD 21.ed.005.275

JOSÉ LUIZ DE SOUZA GOMES

PARALELIZAÇÃO DE ALGORITMO DE SIMULAÇÃO DE MONTE CARLO PARA A ADSORÇÃO EM SUPERFÍCIES HETEROGÊNEAS BIDIMENSIONAIS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Ronaldo A. L. Gonçalves

Aprovado em 27/02/2009.

BANCA EXAMINADORA

Prof. Dr. Ronaldo Augusto de Lara Gonçalves
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. João Angelo Martini
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. Rogério Luís Rizzi
Universidade Estadual do Oeste do Paraná – DME/UNIOESTE

Prof. Dr. Vladimir Ferreira Cabral
Universidade Estadual de Maringá – DEQ/UEM

AGRADECIMENTOS

Agradeço, primeiramente, a Deus, por sua infinita misericórdia e graça, sem as quais seria impossível terminar este trabalho.

À minha esposa e filhas, meus tesouros, pela compreensão, paciência, incentivo e amor demonstrados, mesmo durante minha ausência. Espero poder compensar os momentos que passei distante de vocês.

Aos meus pais, que sempre me incentivaram nos estudos.

Ao meu orientador, Prof. Dr. Ronaldo A. L. Gonçalves, pelo incentivo, paciência e confiança em mim depositadas. Seus conselhos me ajudaram muito a poder achar o caminho.

Ao Prof. Dr. Vladimir Ferreira Cabral, por ter apresentado o problema objeto deste trabalho e o confiado a mim, e pelos valiosos esclarecimentos sobre o fenômeno estudado.

Agradeço à chefia e aos meus colegas de trabalho da Pró-Reitoria de Pesquisa e Pós-Graduação da UEM, também pela paciência durante a realização deste trabalho e pela compreensão, mesmo nos momentos de crise devido ao volume de trabalho.

Aos meus colegas de turma, especialmente ao Walter Marcondes Filho, que me incentivou a ingressar no curso, ao Rogério, ao Mauro, ao Roberto e ao José Valderlei, que sempre estiveram dispostos a ajudar no que fosse possível. A ajuda de vocês foi inestimável.

Aos professores do Programa de Pós-Graduação em Ciências da Computação, à Coordenação e à Secretaria, que também sempre estiveram prontos a ajudar.

E, por fim, muito obrigado a todos que de alguma forma contribuíram, direta ou indiretamente, com esta fase de estudos e crescimento.

DEDICATÓRIA

A DEUS e à minha família. Sem vocês, eu nada sou.

RESUMO

Este trabalho discute questões relacionadas à paralelização de algoritmos sequenciais utilizados na solução de problemas científicos, muitos dos quais escritos em C ou FORTRAN, em uma época em que as facilidades de programação paralela e distribuída, tanto em *software* quanto em *hardware*, não eram tão disponíveis quanto atualmente. Muitos destes algoritmos requerem longo tempo de execução, embora apresentem resultados importantes durante as simulações. Áreas como Física, Biologia e Engenharia podem tirar vantagens com a execução paralela e distribuída desses algoritmos em um *cluster* de computadores, que pode ser adquirido a um baixo custo. Consequentemente, um volume maior de dados pode ser manipulado, provendo novos resultados e viabilizando o avanço das pesquisas nas áreas científicas. O objetivo principal deste estudo é a paralelização de um algoritmo da área de Engenharia Química que utiliza o método Monte Carlo para simular o processo de adsorção de moléculas em superfícies heterogêneas bidimensionais e que possa ser executado em um *cluster* de computadores. Esse algoritmo utiliza o método de Monte Carlo para calcular o estado de energia do sistema após movimentos das moléculas e os resultados são utilizados para a obtenção de gráficos de isotérmicas, comparando-os com os experimentos reais e dados conhecidos. Para isto, questões sobre a tarefa de paralelização encontradas na literatura foram estudadas e implementadas baseadas no modelo sugerido por Foster (1995). Foram implementadas 4 versões paralelas e discutidas as diferentes abordagens adotadas em cada uma, como particionamento de domínio, alocação dinâmica de carga e tolerância a falhas. A execução sequencial do referido algoritmo consome muitas horas de processamento e as versões paralelas apresentaram redução do tempo de processamento em aproximadamente 73,7%, 73,4%, 80% e 83,17%, respectivamente, sendo a 4.ª versão a mais eficiente, aproveitando melhor os recursos disponíveis no ambiente paralelo. Com isto simulações com um volume maior de dados poderão ser efetuadas. Espera-se, assim, que os resultados sejam mais significativos para a área em que o algoritmo é aplicado.

Palavras-chave: MPI. Processamento paralelo. Computação científica. Método de Monte Carlo. Simulação. Adsorção.

ABSTRACT

This work discusses issues related to the parallelization of sequential algorithms used in the solution of scientific problems, many of them written in C or FORTRAN, in a time where the parallel and distributed programming facilities, in hardware and in software, were not as available as nowadays. Many of these algorithms require a long execution time, even though they present important results during simulations. Areas such as Physics, Biology and Engineering can benefit from parallel and distributed execution of these algorithms on computer clusters, which can be acquired at low cost. Consequently, a larger volume of data can be processed, providing new results and enabling the research progress in scientific areas. The objective of this work is to parallelize a Chemical Engineering scientific algorithm for molecular adsorption on two-dimensional heterogeneous surfaces. This algorithm uses the Monte Carlo method to calculate the energy state of the system after molecular movements and the results are used to draw isotherm diagrams, comparing them with real experiments and known data. Therefore, questions about the task of parallelization found in the literature were studied and implemented based on the model suggested by Foster (1995). Four parallel versions have been implemented and discussed the different approaches taken in each one, such as domain partitioning, dynamic load allocation and fault-tolerance. Its sequential execution spends long processing time and the parallel versions showed a reduction of execution time by approximately 73.7%, 73.4%, 80% and 83.17%, respectively, where the 4th version is the most efficient, making better use of the available resources in the parallel environment. Thus, simulations with larger volume of data could be made. It is expected, therefore, that the results will be more significant for the area to which it applies.

Keywords: MPI. Parallel processing. Scientific computing. Monte Carlo method. Simulation. Adsorption.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS E TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CLUMPs	*Cluster of SMPs*
COW	*Cluster of Workstations*
CPU	*Central Processing Unit*
DNA	*Deoxyribonucleic Acid*
HPPCA	*High Performance Parallel and Computer Architectures Group*
LAM-MPI	*Local Area Multicomputer-Message Passing Interface*
LECAD	*Laboratório de Computação de Alto Desempenho*
MIMD	*Multiple Instruction, Multiple Data*
MISD	*Multiple Instruction, Single Data*
MPI	*Message Passing Interface*
MPMD	*Multiple Program, Multiple Data*
NASA	*National Aeronautics and Space Administration*
OSCAR	*Open Source Cluster Application Resources*
PC	*Personal Computer*
PCAM	*Particionamento, Comunicação, Aglomeração e Mapeamento*
PVM	*Parallel Virtual Machine*
SIMD	*Single Instruction, Multiple Data*
SISD	*Single Instruction, Single Data*
SMP	*Symmetric MultiProcessing*
SMT	*Simultaneous Multithreading*
SPMD	*Single Program, Multiple Data*

Capítulo I

Introdução

A ideia que envolve o conceito da computação paralela é a mesma aplicada a pessoas trabalhando juntas para realizar uma tarefa que demoraria muito tempo para ser executada por uma só pessoa, porém, ao invés de pessoas, temos processadores executando um algoritmo para resolver um problema (KARNIADAKIS, 2003). A crescente popularização dos computadores e avanços tecnológicos nos sistemas de comunicação têm possibilitado o acesso a recursos computacionais antes restritos a grandes centros de pesquisa. Computadores comuns, ao lado de softwares de padrões abertos, hoje permitem a construção de *clusters* de computadores com capacidade de processamento similar ao de grandes computadores de alguns anos atrás, porém com custos acessíveis, possibilitando processar grandes quantidades de dados e resolver problemas complexos em tempo muito menor. O primeiro *cluster* construído com computadores comuns foi patrocinado pela NASA em 1994 e era constituído por 16 PC's 486 interligados por uma rede Ethernet, a um custo de aproximadamente US$ 40.000 (KARNIADAKIS, 2003). Um supercomputador equivalente custava, na época, em torno de 1 milhão de dólares. As simulações de processos físicos, químicos e biológicos, entre outros, têm sido largamente utilizadas pelos cientistas nos últimos anos, sendo a computação científica o coração deste processo (KARNIADAKIS, 2003). Problemas como a dinâmica de fluidos e o sequenciamento de DNA, por exemplo, tipicamente envolvem grande quantidade

de dados e podem levar dias para serem resolvidos usando-se programação sequencial. Para reduzir este tempo, diferentes recursos de software e hardware estão disponíveis atualmente, tais como computadores paralelos, ferramentas e bibliotecas de programação paralela, compiladores com otimização de código, algoritmos de particionamento de domínio (MORETTI, 1998; CARVALHO, 2002), entre outros. Muitos programas ainda utilizados nas áreas da Física, Biologia e Engenharia, como por exemplo de geofísica e meteorologia, foram implementados sequencialmente há anos atrás, em linguagens como C ou FORTRAN, são potenciais candidatos a ser reprogramados visando aproveitar as facilidades atuais de programação paralela e distribuída. Contudo, isto não é um trabalho fácil e será objeto de discussão neste trabalho.

Para demonstrar isso, analisamos primeiramente o desenvolvimento das arquiteturas paralelas e a importância de *clusters* de computadores com a biblioteca MPI como ambiente de alto desempenho para a execução de aplicações paralelas, abordando questões que devem ser consideradas durante a análise de um algoritmo a ser paralelizado.

Tendo isto em mente, o objetivo deste trabalho é paralelizar um algoritmo da área de Engenharia Química para a simulação do fenômeno de adsorção em superfícies heterogêneas bidimensionais que usa o método Monte Carlo, o qual fornece uma representação estatística de um modelo microscópico para o fenômeno estudado (KARNIADAKIS, 2003) e que consiste, basicamente, na utilização intensa de números aleatórios para a sua resolução.

A adsorção é um fenômeno que pode ser utilizado para a separação de gases, através da introdução de um outro elemento, chamado de adsorvente, geralmente um sólido, que possui alguma afinidade química com um dos gases, o qual “adere” à superfície do sólido formando uma camada e propiciando uma separação altamente específica. Na simulação esse sólido é modelado em uma estrutura matricial e são analisadas as energias resultantes de cada movimento (locomoção na superfície, inserção e remoção) das moléculas adsorvidas em determinadas regiões, chamadas de sítios ativos. Após cada movimento, o algoritmo deve analisar exaustivamente a configuração de energia de toda a superfície. Esta análise é feita de milhões a bilhões de vezes, dependendo dos parâmetros iniciais, até chegar a uma condição de equilíbrio, a qual possui o menor nível energético.

O algoritmo analisado foi implementado em FORTRAN 90 (CABRAL, *et al*, 2003a) usando técnica de programação sequencial, e, dependendo dos dados de configuração inicial, pode consumir até vários dias de processamento em um único computador. O fator tempo de execução tem limitado bastante este tipo de simulação, pois os recursos computacionais (tempo de CPU) exigidos são muito grandes. Sua paralelização, portanto, além do benefício

da redução do tempo de execução, trouxe a possibilidade da ampliação do número de moléculas utilizadas, proporcionando, assim, que resultados mais expressivos sejam obtidos.

A tarefa de paralelizar um algoritmo não é fácil e requer do programador atenção especial quanto ao quê e como deve ser feito. Algumas abordagens encontradas na literatura foram analisadas e a metodologia escolhida baseou-se no modelo sugerido por Foster (1995). Foram implementadas 4 versões paralelas e discutidas as diferentes abordagens adotadas em cada uma, como particionamento de domínio, alocação dinâmica de carga e tolerância a falhas. As versões paralelas apresentaram redução do tempo de processamento em aproximadamente 73,7%, 73,4%, 80% e 83,17%, respectivamente.

O presente trabalho está organizado da seguinte forma: o Capítulo 2 explana o processamento e arquiteturas paralelas, destacando os *clusters* de computadores e descreve algumas características do Padrão MPI e da implementação LAM-MPI. O Capítulo 3 apresenta a revisão bibliográfica e discute algumas pesquisas que se utilizaram de aplicações paralelas em *clusters* de computadores e também tece algumas considerações que devem ser observadas pelo programador ao paralelizar um algoritmo, apresentando também, resumidamente, a metodologia proposta por Foster (1995). O Capítulo 4 apresenta o experimento de simulação de adsorção de moléculas, descrevendo sucintamente o fenômeno para que possa ser entendido por quem não é da área de Engenharia Química. O Capítulo 5 apresenta a metodologia utilizada para a implementação paralela do algoritmo nas 4 versões, cada uma com uma característica diferente. O Capítulo 6 mostra a análise dos resultados obtidos e o Capítulo 7 apresenta as conclusões.

Capítulo II

Processamento Paralelo

A arquitetura de von Neumann, surgida nos anos 50, estabeleceu os conceitos básicos da organização de computadores e por muitos anos foi o modelo seguido pela indústria. Nela, o computador é concebido como um conjunto basicamente composto pelo processador, memória e dispositivos de entrada/saída que executam tarefas sequencialmente, na ordem determinada por uma unidade de controle.

A evolução tecnológica trouxe a possibilidade de melhora desse modelo, sempre buscando maior eficiência. Como cada componente da arquitetura de von Neumann trabalha em velocidades diferentes, as operações executadas internamente no processador são muito mais rápidas que as operações de entrada e saída, por exemplo, numa execução sequencial o processador fica ocioso enquanto uma operação de leitura de dados do disco está sendo executada. Assim, o modelo sequencial desperdiça tempo precioso que poderia estar sendo usado para processamento.

Pensando nisso é que surgiu a ideia de aproveitar esse tempo que um processo usa para fazer E/S, por exemplo, para executar outro processo. Os projetistas, então, fizeram com que os computadores dividissem o uso da CPU entre os processos em execução, determinando uma fatia de tempo (*time slice*) para cada um, ou então, passando a executar outro processo

enquanto o atual está solicitando uma operação de E/S. Como esta troca de contexto (passar a execução de um processo para outro) é muito rápida, em sistemas monoprocessados, tem-se a impressão de que existem vários processos sendo executados ao mesmo tempo. Esse paralelismo não é real, uma vez que somente um processo está sendo executado em um determinado instante.

Para obter um paralelismo real é preciso adicionar, pelo menos, mais um processador[1]. Este outro processador poderia estar executando outro processo, mas não necessariamente. Ele pode estar executando o mesmo processo, contribuindo assim com a diminuição do tempo total de execução. Contudo, as arquiteturas paralelas trazem embutidas complexidades adicionais como o controle de processos em execução, controle de acesso à memória e comunicação entre os processos, por exemplo.

2.1 ARQUITETURAS PARALELAS

Segundo Hwang (1990), o processamento paralelo pode ser definido como: “Forma eficiente do processamento de informações com ênfase na exploração de eventos concorrentes no processo computacional”.

O principal objetivo da computação paralela é a redução do tempo computacional (ou obter maior desempenho) para a resolução de um problema (ALMASI, 1994; ZALUSKA, 1991). Podemos fazer uma analogia com a construção de uma estrada. Quanto mais gente executando a tarefa, mais rápida ela ficará pronta. Contudo, sempre existem pontos de estrangulamento (gargalos) que podem atrasar a tarefa e que precisam ser considerados. Na analogia da estrada, por exemplo, se houver 100 trabalhadores, mas 99 dependerem de que apenas 1 termine seu trabalho para os outros continuarem, então o ganho de tempo estará comprometido e será fortemente influenciado pelo tempo gasto por aquele único trabalhador. No caso dos computadores não é diferente: quanto mais processadores envolvidos, e cada um executando uma parte do problema ao mesmo tempo, mais rápido ele terminará, porém os gargalos (largura de banda, falha em alguma máquina, etc.) também influenciam no tempo total e precisam ser tratados. O ideal seria se, tendo n processadores, o tempo total de execução fosse reduzido para $1/n$ do tempo correspondente à execução com um único processador. Porém, isto não ocorre devido a diversos fatores, como latência de memória e de

[1] Não estamos considerando aqui o modelo SMT, em que em um único processador existem recursos para a execução simultânea de mais de uma *thread* (EGGERS, 1997).

rede e algoritmos pouco paralelizados.

A demanda pela computação paralela tem sido cada vez maior. Algoritmos complexos e volume de dados muito grandes, típicos de algumas aplicações científicas, industriais ou militares requerem cada vez mais poder computacional para ser processados (KIRNER, 1991).

2.2 CLASSIFICAÇÃO DE ARQUITETURAS PARALELAS

Flynn estabeleceu uma relação entre fluxos de instruções e fluxos de dados num processo computacional. Um fluxo de instruções corresponde a uma sequência de instruções executadas (em um processador) sobre um fluxo de dados aos quais estas instruções estão relacionadas (ALMASI, 1994; DUNCAN, 1990; HWANG, 1990; NAVAUX, 1989). As combinações desses dois fluxos resultam em 4 classes genéricas, que são:

- SISD (*Single Instruction, Single Data*): Fluxo único de instruções sobre um único conjunto de dados (Figura 1). Corresponde ao modelo de von Neumann.

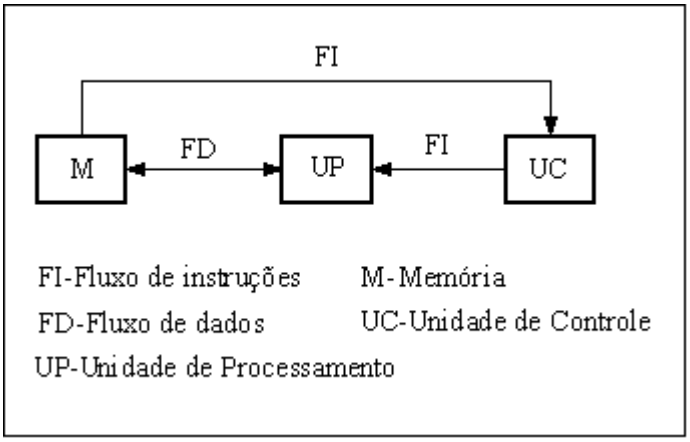


Figura 1. Modelo Computacional SISD.

- SIMD (*Single Instruction, Multiple Data*): Fluxo único de instruções em múltiplos conjuntos de dados. Múltiplos processadores (escravos) são controlados por uma única unidade de controle (mestre), sendo a mesma instrução executada simultaneamente sobre diversos conjuntos de dados (Figura 2). Manipulação de matrizes e processamento de imagens são exemplos do uso dessa arquitetura.

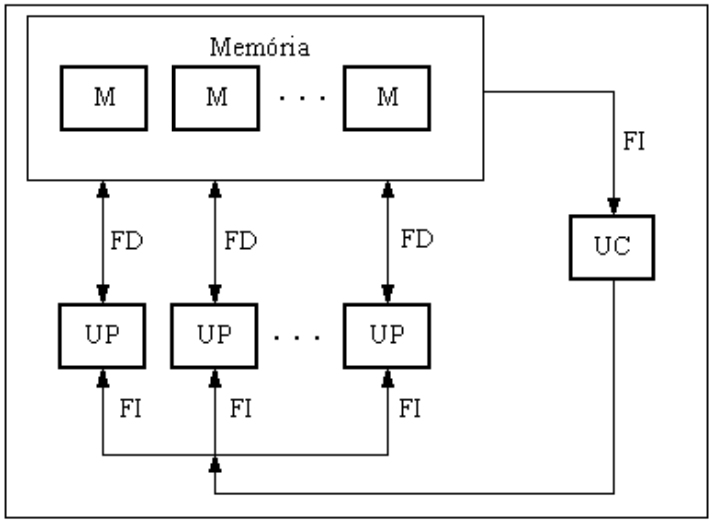


Figura 2. Modelo Computacional SIMD.

- MISD (*Multiple Instruction, Single Data*): Fluxo múltiplo de instruções em um único conjunto de dados. Múltiplos processadores executam diferentes instruções em um único conjunto de dados (Figura 3).

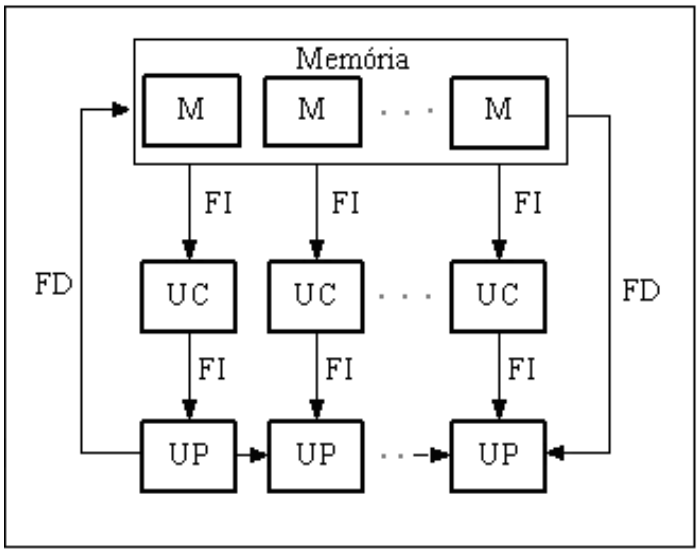


Figura 3. Modelo Computacional MISD.

- MIMD (*Multiple Instruction, Multiple Data*): Fluxo múltiplo de instruções sobre múltiplos conjuntos de dados. Múltiplos processadores executam diferentes instruções em diferentes conjuntos de dados, de maneira independente (Figura 4). A maioria dos computadores paralelos enquadra-se nesta classe.

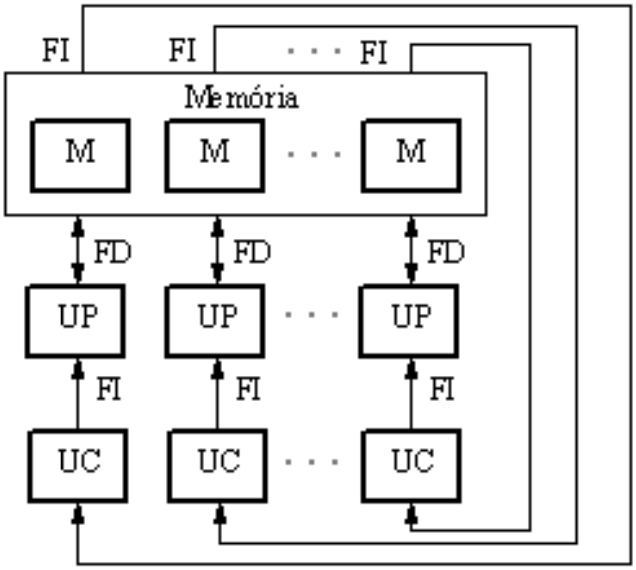


Figura 4. Modelo Computacional MIMD.

Embora seja antiga, esta classificação ainda é muito utilizada, mas é um tanto deficiente em relação a computadores mais modernos. Às vezes é difícil enquadrar um computador em uma classe ou outra, uma vez que várias tecnologias foram combinadas ao longo dos anos. Sem descartar, contudo, o que foi proposto por Flynn, Duncan (1990) propôs uma classificação mais abrangente, permitindo apresentar uma visão geral dos estilos de organização para computadores paralelos da atualidade, conforme mostra a Figura 5.

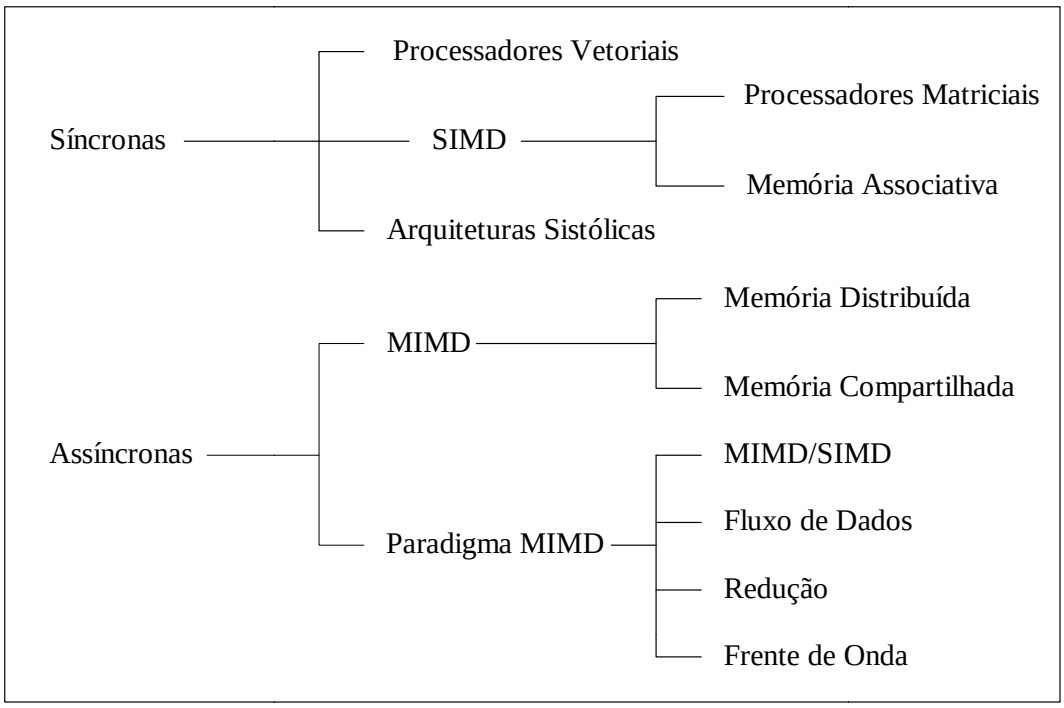


Figura 5. Classificação de Duncan (1990)

Duncan (1990) exclui de sua classificação arquiteturas que já se tornaram comuns nos computadores modernos e que apresentam apenas mecanismos de paralelismo de baixo nível, como unidades funcionais múltiplas em uma única CPU e o *pipeline* dos estágios de execução de uma instrução.

Devido ao escopo deste trabalho, não explanaremos todos os tipos propostos por Duncan e passaremos agora a tratar apenas de memória distribuída e compartilhada, para então discutirmos sobre a plataforma MPI.

Na arquitetura MIMD, como vimos, cada processador executa instruções independentes daquelas que estão nos outros processadores e acessam dados também independentes. Tal arquitetura apresenta duas formas de acesso à memória, a saber:

- **Memória Compartilhada:** Múltiplos processadores compartilham o acesso ao espaço de memória global via rede de interconexão de alta velocidade (Figura 6). Essa arquitetura também pode ser denominada SMP (multi-processamento simétrico) e permite aos processadores acessar e trocar dados, mas isso acaba gerando complexidade adicional para manter o controle e integridade da informação.

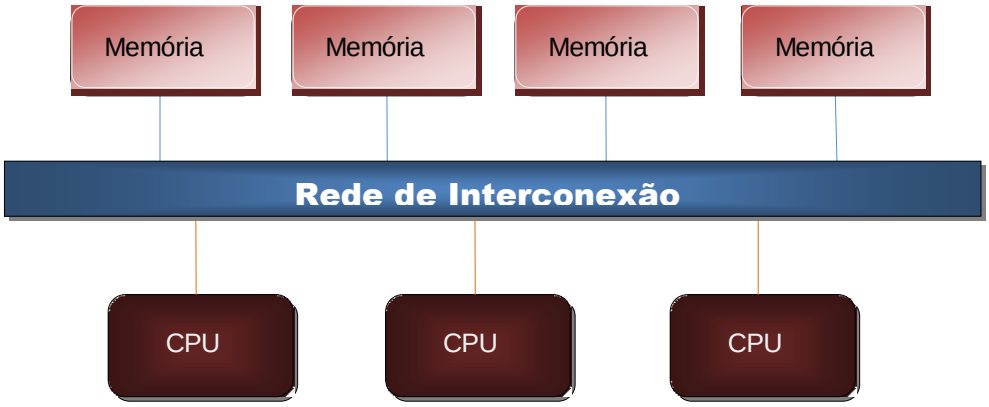


Figura 6. Exemplo de um sistema distribuído de memória compartilhada.

Da perspectiva do programador, sistemas de memória compartilhada simplificam o processo de converter código sequencial em paralelo, mas adicionam complexidade extra ao requerer sincronização explícita que podem resultar em erros não determinísticos difíceis de detectar e corrigir (DONGARRA *et al*, 2003).

- **Memória Distribuída:** Neste modelo temos uma coleção de computadores seriais (cada um designado como um nó) trabalhando em paralelo, conforme a Figura 7. Cada um tem sua própria memória local e acessa a memória dos outros nós via algum tipo

de rede de comunicação. A troca de dados é feita através da rede como mensagens entre os nós.

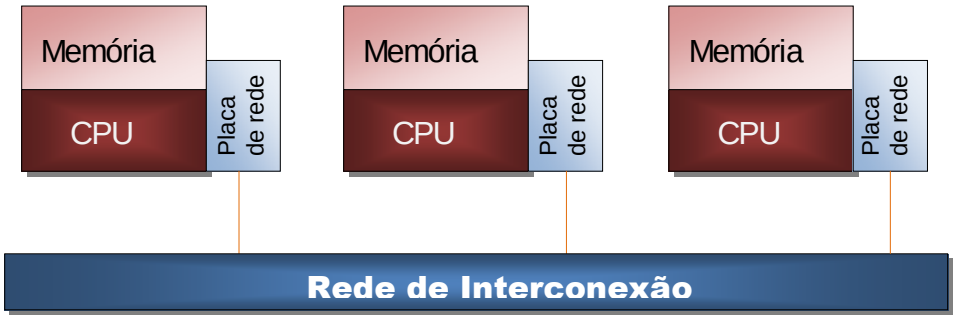


Figura 7. Exemplo de um sistema de memória distribuída.

Como cada processador acessa sua própria memória local, não há um espaço comum onde todos podem ler e escrever os dados. Daí surge a necessidade da troca (ou passagem) de mensagens entre os processos para que possa haver a coordenação das tarefas. Segundo Dongarra (2003), essa abordagem é a mais apropriada, mesmo que a conversão de um programa sequencial requeira mais trabalho para extrair a informação que deve ser paralelizada para a execução nos outros nós.

Essa troca de mensagens é efetuada via bibliotecas (API's) como o PVM e MPI, por exemplo. Devido à latência encontrada no meio de interconexão, a comunicação nestes sistemas torna-se o grande gargalo de desempenho, sendo objeto de pesquisas para sua otimização, como discutido em (BAER, 2004; KAMAL, 2005; LEE, 2004).

Contudo, o desenvolvimento de redes comerciais de alta velocidade tornou possível agrupar computadores e interligá-los com dispositivos comuns e a preços acessíveis, surgindo, então, o que chamamos de *cluster*, que é uma arquitetura que consiste de vários computadores interconectados por uma rede local, sendo um sistema paralelo/distribuído que apresenta desempenho comparável ao de supercomputadores de alguns anos atrás. Essa arquitetura é vista pelo usuário como uma só máquina com grandes recursos.

O uso de um *cluster* não deve ser pensado como um fim em si mesmo, mas como um passo adiante na efetiva computação paralela que é ao mesmo tempo confortável e de custo efetivo para muitas classes de problemas (HAYES, 1992).

A Figura 8 mostra um *cluster* típico, baseado no *cluster Beowulf*, que foi idealizado pela NASA em 1994 (KARNIADAKIS, 2003), com a finalidade de processar as informações espaciais. Além de oferecer baixo custo de implementação, apresenta alta disponibilidade e

tolerância a falhas, já que os recursos podem ser duplicados, sendo importantíssimos em sistemas de alto desempenho. Apresenta também alta escalabilidade. Para aumentá-la, basta acrescentar novas máquinas.

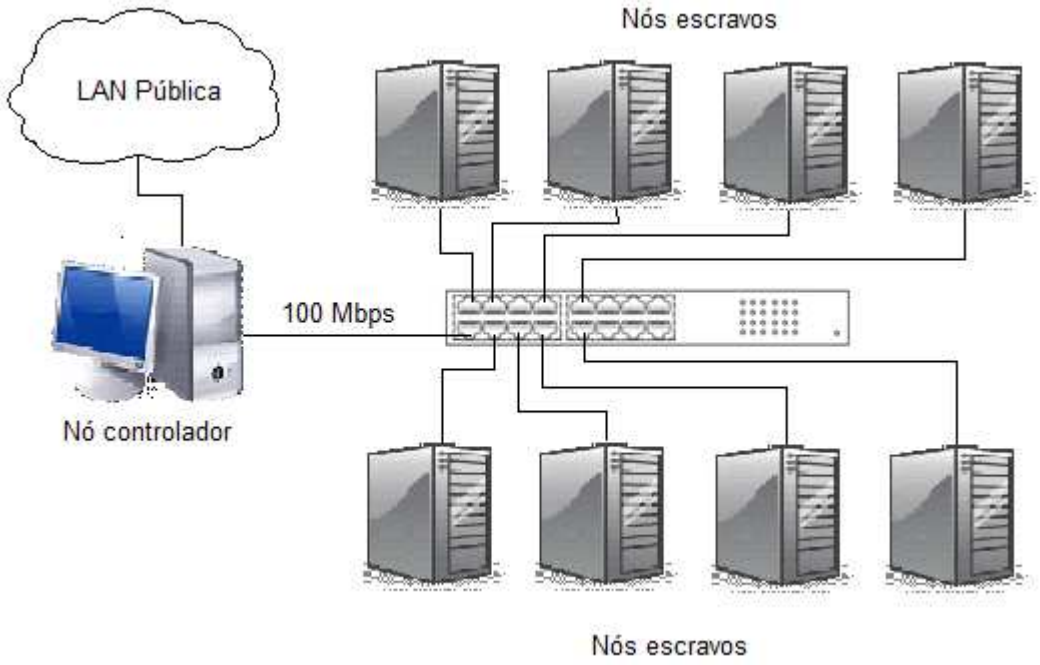


Figura 8. Exemplo típico de um *cluster* de PC´s.

O *cluster* disponível no Laboratório de Computação de Alto Desempenho (LECAD) do Departamento de Informática da Universidade Estadual de Maringá, pertencente ao grupo de pesquisa HPPCA e que foi utilizado neste trabalho, curiosamente, tem essa mesma configuração em termos de número de máquinas e tecnologia de interconexão. A configuração de cada nó é apresentada no Capítulo 6.

Um *cluster* é dito homogêneo se ele é constituído de equipamentos iguais em todos os nós e heterogêneo, caso contrário. Além dos *clusters* formados por computadores pessoais comuns existem também os *clusters* de *workstations* (COW) e *clusters* de SMPs (CLUMPs), onde cada nó possui um computador multiprocessado. Devido às suas características, o *cluster* tornou-se um ambiente ideal para a utilização de aplicações paralelas que usam troca de mensagens entre os processos. Para se ter uma ideia da popularidade dos *clusters*, na lista dos 500 maiores supercomputadores do mundo (TOP500, 2009), 82% deles fazem parte dessa arquitetura, somando 1.686.750 processadores, sendo que tem aumentado a cada nova listagem.

Como a tecnologia não para de avançar, a mais nova característica de *clusters* consiste no agrupamento entre eles, formando uma arquitetura denominada *Grid* (MARTINS, 2005). Esse conceito de computação distribuída em inúmeros nós foi recentemente utilizado no projeto SETI@HOME (SETI, 2009), no qual milhares de computadores conectados à Internet executaram o processamento das informações recebidas dos radiotelescópios, quando estivam ociosos, como se fossem um grande supercomputador virtual, à procura de inteligência extra-terrestre.

2.3 O PADRÃO MPI

O MPI (*Message Passing Interface*) (PACHECO, 1997; PACS, 2001) é uma biblioteca de funções (em C) ou sub-rotinas (em FORTRAN) usadas para executar a comunicação de dados entre processos em um ambiente paralelo distribuído. Surgiu em 1992 como um esforço conjunto de cerca de 60 profissionais de 40 organizações de várias áreas, chamado de MPI *Forum*, para tentar padronizar os diversos sistemas de trocas de mensagens existentes que não eram compatíveis entre si.

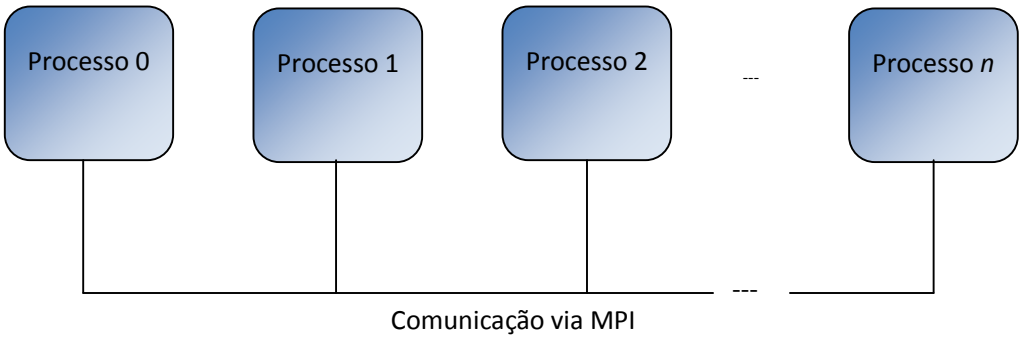


Figura 9. Processos MPI trabalhando juntos (KARNIADAKIS, 2003)

Tipicamente, programas de troca de mensagens consistem de múltiplas instâncias de programas seriais que se comunicam por chamadas de bibliotecas (PACS, 2001) (Figura 9). No padrão MPI, os processos que constituem uma aplicação se comunicam através de funções para o envio e recebimento de mensagens entre eles. Normalmente existe um conjunto fixo de processos na maioria das aplicações, porém, esses processos podem executar diferentes programas. Assim, o MPI pode seguir tanto o paradigma SPMD (*Single Program, Multiple Data*) no qual o mesmo programa é executado em todos os nós, como o MPMD (*Multiple Program, Multiple Data*), pois cada nó pode executar um programa diferente, ainda que sobre

os mesmos dados. O MPI pode executar também um processo concorrentemente no mesmo processador, sendo esta característica melhor aproveitada em processadores modernos, com dois ou mais núcleos, embora ocorra alguma perda de performance, pois, mesmo sendo local, a comunicação ainda é mais lenta que o acesso direto à memória.

Todo programa MPI deve seguir a estrutura exibida no Quadro 1. A primeira linha do fragmento de código em linguagem C mostrado serve para indicar ao compilador que deve incluir as bibliotecas do MPI ao gerar o executável do programa. Dentro do algoritmo, tudo aquilo que será executado em paralelo deve estar entre as linhas MPI_Init e MPI_Finalize.

```
#include <mpi.h>
...
int main(int *argc, char *argv[]) {
....
MPI_Init(&argc, &argv); //inicialização

      /* funções do MPI */

MPI_Finalize(); //finalização
....
}
```

Quadro 1. Estrutura básica de um programa MPI.

O MPI possui vários conjuntos de rotinas que servem para comunicação ponto-a-ponto, para comunicações globais, definição de tipos básicos e sincronização dos dados, tendo cada uma delas atributos que são importantes para a correta movimentação das informações. A partir da versão 2, lançada em 1995, o MPI ganhou novas características, como E/S paralela, comunicação unidirecional e comunicação coletivas não bloqueantes (MARTINS, 2005), entre outras.

2.3.1 Formas de comunicação da plataforma MPI

Os processos envolvidos na execução de programas paralelos no MPI são identificados por uma sequência de números inteiros não negativos. Assim, se existem *p* processos executando um programa, cada um possui um *rank* 0, 1, ..., *p*-1 (PACHECO, 1997). Usando o paradigma *Single Program, Multiple Data* (SPMD), temos programas iguais executando em diferentes processadores tomando ações baseadas no *rank* do processo: os

comandos executados pelo processo 0 são diferentes daqueles executados por outros processos, ainda que todos estejam executando o mesmo programa.

O MPI segue a abordagem de dividir e conquistar, que é um conceito chave em computação paralela (KARNIADAKIS, 2003). Essencialmente ocorre o seguinte (usando o paradigma SPMD):

- O usuário fornece a diretiva para o sistema operacional colocar uma cópia do executável em cada processador;
- Cada processador inicia a execução de sua cópia do executável;
- Diferentes processos podem executar diferentes ações dentro do programa. O algoritmo para a resolução de um problema deve ser elaborado de modo que cada processador execute uma parte do processamento. Tipicamente isto é baseado no *rank* do processo; e
- Os processos se comunicam para trocar dados.

Para que a comunicação ocorra, o MPI define o conceito de canais de comunicação. São dois os canais de comunicação padrão, *MPI_COMM_WORLD e MPI_COMM_SELF* (MARTINS, 2005). O primeiro é utilizado para a troca de mensagens com todos os processos e é criado no início da execução do programa. O segundo é utilizado apenas para comunicações que envolvam o próprio processo emissor. O grupo de processos associados a eles podem ser rearranjados em grupos distintos quando houver a necessidade de troca de mensagens entre nós específicos.

Como mostrado na Figura 9, os processos do MPI se comunicam e trocam dados através da estrutura fornecida pela biblioteca. Este processo é o cerne do MPI, pois quase tudo se resume em “enviar mensagem – receber mensagem” (KARNIADAKIS, 2003). A comunicação entre os processos pode ser feita de duas formas:

- **ponto-a-ponto:** comunicação estabelecida entre dois pontos, de modo que um envia e o outro recebe as mensagens. As diretivas do MPI que realizam esta tarefa são *MPI_Send()* e *MPI_Recv()*[2].

Para que a comunicação seja feita com sucesso, o MPI anexa algumas informações aos dados, chamado de envelope, e que consiste nas seguintes informações:

  - O dado que será transmitido/recebido;
  - O tipo do dado;

[2] Para o escopo deste trabalho, apenas as diretivas utilizadas na paralelização serão comentadas, mas sem detalhar os parâmetros. Mais informações podem ser obtidas em (PACHECO, 1997; SNIR, 1996).

- O tamanho em *bytes* do dado;
- O *rank* do processo emissor;
- O *rank* do processo destinatário;
- Um *tag* (etiqueta especificando o tipo da mensagem); e
- Um comunicador (domínio de comunicação). Neste trabalho foi utilizado somente o comunicador *MPI_COMM_WORLD*.

Um exemplo é mostrado no Quadro 2, em linguagem FORTRAN, onde o processo 0 envia uma mensagem para o processo 1. Se o processo emissor fica esperando até que o processo receptor confirme o recebimento da mensagem, a comunicação é dita síncrona (bloqueante), caso contrário, é dita assíncrona (não bloqueante).

```
CHARACTER*20 msg
INTEGER myrank, ierr, status(MPI_STATUS_SIZE)
INTEGER tag = 99
...
CALL MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr)
IF (myrank .EQ. 0) THEN
    msg = "Hello there"
    CALL MPI_SEND( msg, 11, MPI_CHARACTER, 1, tag, MPI_COMM_WORLD,
                   ierr)
ELSE IF (myrank .EQ. 1) THEN
        CALL MPI_RECV( msg, 20, MPI_CHARACTER, 0,tag,
                       MPI_COMM_WORLD, status, ierr)
END IF
```

Quadro 2. Exemplo de envio e recebimento de mensagens ponto-a-ponto no MPI. (SNIR, 1996)

- **coletivamente:** um processo envia uma mensagem para mais de um processo ou recebe mensagens deles. Serve para executar operações globais. Por exemplo, quando um mesmo dado precisa ser enviado para outros processos do grupo, incluindo o próprio emissor, é feito um *broadcast* (difusão dos dados) com a diretiva *MPI_BCast()*, conforme mostram o Quadro 3 e a Figura 10, onde o processo 0 lê os dados fornecidos pelo usuário via teclado e os transmite para os outros processos. Como cada um deve receber esses dados, esta diretiva também serve como ponto de sincronização. Somente quando a execução de todos estiver no primeiro *broadcast*, por exemplo, é que o dado será efetivamente transmitido e recebido, depois a execução de todos passará para o segundo *broadcast* e assim por diante.

```
...
if (my_rank .EQ. 0) then
      print *, 'Enter a, b, and n'
      read *, a, b, n
end if
call MPI_BCAST(a, 1, MPI_REAL , 0, MPI_COMM_WORLD, ierr )
call MPI_BCAST(b, 1, MPI_REAL , 0, MPI_COMM_WORLD, ierr )
call MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr )
...
```

Quadro 3. Exemplo de comunicação coletiva usando *broadcast*. (PACHECO, 1997)

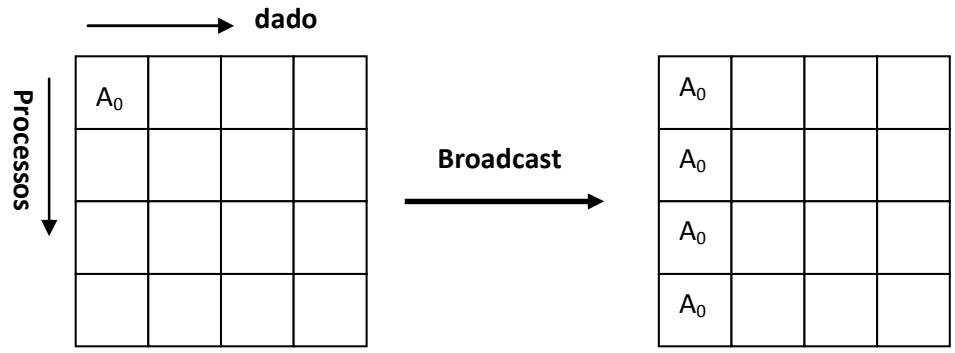


Figura 10. Exemplo gráfico do *broadcast*

Neste trabalho, logo após o início da execução, o processo mestre lê os dados do arquivo de configuração e os transmite via *MPI_Bcast()* para todos os outros nós e a si próprio, inclusive, para inicializar as variáveis com os mesmos dados. Quando cada escravo termina seu processamento, os dados são transmitidos de volta ao mestre via *MPI_Reduce()*, que pode executar vários tipos de operações sobre os dados. Os exemplos estão mostrados no Quadro 4 e Figura 11. Neste trabalho foi usada somente a operação de soma (MPI_SUM), que totaliza os dados para depois serem efetuados os cálculos finais. Todas as comunicações globais são pontos de sincronização, pois é preciso que todos os processos participantes estejam no mesmo ponto para que a comunicação ocorra.

```
...
call MPI_REDUCE(a1,a,1,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)
call MPI_REDUCE(b1,b,1,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)
call MPI_REDUCE(n1,n,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,
      ierr )
...
```

Quadro 4. Exemplo de comunicação coletiva usando *reduce*.(PACHECO, 1997)

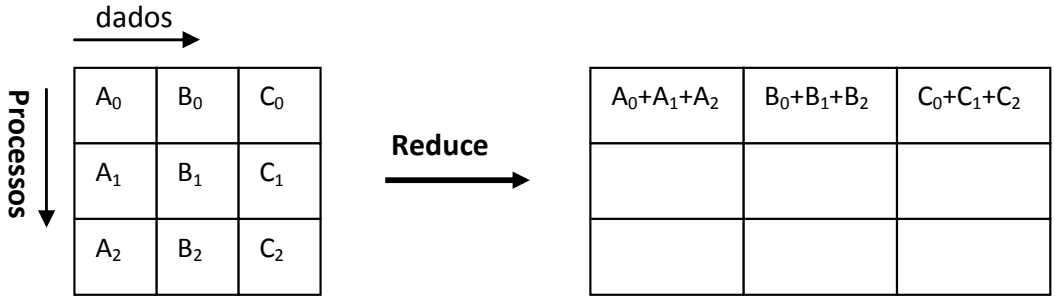


Figura 11. Exemplo gráfico do *reduce*.

O MPI foi projetado para computadores paralelos, *clusters* e redes heterogêneas, possibilitando a resolução de um algoritmo específico, e admite que a comunicação entre os nós é confiável, não se preocupando com o sistema de interconexão (SNIR, 1996). Entretanto, diversas implementações dessa plataforma têm utilizado redes de diferentes tecnologias, sendo a mais comum a padrão Ethernet. Em cada uma, problemas como latência de rede e largura de banda são analisados e pesquisas são voltadas para a busca da máxima eficiência na comunicação, de modo a reduzir os atrasos que possam ser provocados pela implementação na arquitetura adotada, como pode ser visto em (GROOVE, 2004; NUPAIROJ, 1994). Outros exemplos são (BAER, 2004), que propõe que as modificações podem ser feitas no protocolo TCP; nos algoritmos (KAMAL, 2005), na otimização de entrada/saída (LEE, 2004).

Outros enfoques têm sido abordados, como MPI orientado a objetos (MCCANDLESS, 1996), uso de linguagem Java (WEIQIN, 2000) e até mesmo metodologias de engenharia de software para construção de aplicações paralelas (LUKSCH, 1997; OJIMA, 2005). Com isto procuram também facilitar a vida do programador e aproveitar todo o potencial de tecnologias já estabelecidas.

2.4 A IMPLEMENTAÇÃO LAM-MPI

A LAM-MPI (*Local Area Multicomputer-MPI*) é uma implementação em código aberto do padrão MPI, desenvolvida e mantida pela Universidade de Indiana. Além de implementar a API do MPI, é um ambiente para execução de aplicações paralelas. Um pequeno programa chamado *daemon* (lamd) fica executando em *background* (segundo plano) monitorando as aplicações e fornecendo os serviços necessários, além de diversas ferramentas para depuração (MARTINS, 2005). Eles devem ser executados em todos os nós do *cluster* e são ativados pelo comando *lamboot*, como mostrado a seguir:

```
<prompt>lamboot –v hosts
```

O arquivo *hosts* é um arquivo texto que contém os nomes de todos os nós que formam o *cluster*, por exemplo, master, slave1, slave2, etc, um em cada linha, ou somente daqueles que serão usados para a execução da aplicação paralela. O parâmetro `–v` serve para exibir os passos na tela. Se a operação for realizada com sucesso, então a partir daí o ambiente do *cluster* estará pronto para executar a aplicação.

Os compiladores que são instalados com a implementação LAM-MPI, por padrão, são para as linguagens C e FORTRAN 77, mas é possível utilizar outros compiladores, com pequenas adaptações no ambiente.

Para compilar um programa em FORTRAN 77, por exemplo, usa-se o comando:

```
<prompt>mpif77 –o <nome_do_executável> <nome_do_fonte> [bibliotecas]
```

Em linguagem C:

```
<prompt>mpicc –o <nome_do_executável> <nome_do_fonte> [bibliotecas]
```

O parâmetro `–o` serve para indicar ao compilador que todas as máquinas onde o programa será executado pertencem à mesma arquitetura, não sendo necessária nenhuma conversão de dados.

Para executar efetivamente um programa MPI, utiliza-se o comando *mpirun* que é o responsável por colocar uma cópia do executável em cada nó, que iniciam, então, sua execução (MARTINS, 2005).

```
<prompt>mpirun –np <programa>
```

O parâmetro `–np` indica quais nós serão usados para executar o programa. Exemplo:

```
<prompt>mpirun –n0-4 pmix
```

No exemplo acima, os nós utilizados para executar o programa chamado `pmix` serão o 0 (mestre), slave1, slave2, slave3 e slave4. O programa deve conter as partes que serão

executadas pelo mestre e as que serão pelos escravos, indicando também os pontos de troca de mensagens e sincronização, quando necessário.

Para finalizar uma sessão MPI de modo correto, usa-se o comando abaixo:

```
<prompt>wipe –v hosts
```

Este comando faz com que todas as máquinas especificadas no arquivo *hosts* sejam retiradas da arquitetura paralela montada.

A LAM-MPI também possui a característica de ser portável para várias plataformas Unix, Linux e Windows, desde computadores pessoais até grandes supercomputadores. (MARTINS, 2005). Assim, um programa escrito em uma plataforma pode ser recompilado em outra e executará da mesma forma.

A seguir relacionamos algumas das principais implementações do MPI:

- Open MPI;
- LAM/MPI (Congelada. Desenvolvimento direcionado para a Open MPI.);
- Los Alamos MPI (LA-MPI) (Congelada. Desenvolvimento direcionado para a Open MPI.);
- FT-MPI (Também vai ser parte da Open MPI.);
- PACX-MPI (Também vai ser parte da Open MPI.);
- MPICH;
- MPICH2;
- MVAPICH (MPI sobre InfiniBand);
- Comerciais:
    - Scali MPI Connect;
    - WMPI II;
    - SGI Message Passing Toolkit;
    - Intel MPI;
    - Sun MPI;
    - HP-MPI; e
    - MPI/Pro.

Vale destacar que o projeto Open-MPI (http://www.open-mpi.org/) reúne tecnologias de diversas implementações anteriores, na tentativa de se obter a melhor biblioteca para o MPI.

2.5 DESEMPENHO E EFICIÊNCIA

Como o principal objetivo da computação paralela é aumentar o desempenho dos programas, tornando-os mais eficientes, algumas formas de medição são necessárias. Uma forma muito usada ainda hoje para medir o desempenho de um algoritmo paralelo é através do *speedup*, que mede o fator de redução do tempo de execução de um programa paralelizado em *P* processadores. Sua forma geral é mostrada na Equação 1 (Foster, 1995).

$$Speedup = \frac{T_S}{T_P}$$

Equação 1. *Speedup.*

onde:

- T_s = tempo de execução do programa sequencial;
- T_p = tempo de execução do programa paralelo em *P* processadores.

O *speedup* é dito relativo quando T_s é igual ao tempo de execução do programa paralelo em 1 processador e absoluto quanto é igual ao tempo de execução do "melhor" programa sequencial.

Alguns autores consideram outros fatores que devem ser levados em conta para obter uma melhor avaliação do desempenho e eficiência de um algoritmo paralelo, como *throughput* de rede, requisitos de memória, custos de projeto, implementação e reuso, etc., e chegam a propor novas formas, mas para os propósitos deste trabalho, a forma geral de uso da **Erro! Fonte de referência não encontrada.** é suficiente. Mais informações podem ser obtidas em (FOSTER, 1995)

Gene Amdahl fez uma observação importante sobre o *speedup*: "se o componente sequencial de um algoritmo consome *1/s* do tempo de execução do programa, então o máximo *speedup* possível que pode ser obtido com um computador paralelo é *s*" (FOSTER, 1995) (tradução nossa). Por exemplo, se 10% do programa for sequencial, então o máximo *speedup*

que se conseguirá é 10. Esta observação ficou conhecida como *Lei de Amdahl* e implica em que todo programa paralelo deve ter uma fração que é executada sequencialmente, que não pode ser paralelizada, por exemplo, a coleta e impressão dos resultados. Se chamarmos essa parte não paralelizável de ε, teremos:

$$Speedup = \frac{1}{1 - \varepsilon}$$

Equação 2. Outra forma do *speedup*.

Logo, $S = 1 - \varepsilon$ é a parte sequencial. Num programa totalmente sequencial, temos $\varepsilon = 0$, então *Speedup* = 1 (não há aumento do *speedup*). Por outro lado, se um programa fosse totalmente paralelizável ($\varepsilon = 1$), teoricamente teríamos um *speedup* muito grande. Se introduzirmos o número de processadores na parte paralelizável da Equação 1, teremos que o tempo de execução do programa paralelo será dado por:

$$T_p = \left(S + \frac{P}{N}\right) \cdot T_s$$

Equação 3. Tempo de processamento paralelo**.**

onde:

- P : % paralelizável;
- N : n° de processadores;
- S : % serial;
- T_S: tempo do processamento serial; e
- T_P: tempo do processamento paralelo.

Substituindo a Equação 3 na Equação 1, temos a forma da *Lei de Amdahl*:

$$Speedup = \frac{1}{S + \frac{P}{N}}$$

Equação 4. Lei de Amdahl.

A eficiência de um programa paralelo é determinada pela relação entre o *speedup* obtido e o número de processadores necessários para obtê-lo. Foster (1995) a define como “a fração do tempo que os processadores gastam fazendo trabalho útil” (tradução nossa).

$$E = \frac{Speedup}{N.o\ Processadores}$$

Equação 5. Eficiência.

É usada para medir a qualidade de um algoritmo paralelo. Ela caracteriza como um algoritmo utiliza os recursos de um computador paralelo independentemente do tamanho do problema.

Capítulo III

Aplicações Paralelas para *Clusters*

Neste capítulo são discutidas algumas pesquisas que foram desenvolvidas com o auxílio de *clusters* de computadores, ressaltando-se as contribuições obtidas, tanto na área de aplicação quanto no desenvolvimento da arquitetura em si.

Foi proposta por (LUKE, 1993) uma definição para a medida da escalabilidade em sistemas paralelos, introduzindo uma função custo de efetividade que relaciona a carga de trabalho com os recursos computacionais, independente da arquitetura. Em 1995, foi analisado o desempenho de sistemas de troca de mensagem em várias arquiteturas, entre elas um *cluster* de *workstations* (DONGARRA, 1995). Através da medição da latência e largura de banda entre dois nós de cada arquitetura, os autores as compararam com os valores nominais dos componentes empregados e concluíram que, no caso do *cluster*, tanto a latência qunto a largura de banda dependem muito da implementação do protocolo TCP/IP. Outra conclusão é que a troca de mensagens é um fator limitante para o desempenho de uma arquitetura paralela. Assim, a velocidade relativa da computação e comunicação pode ser usada na escolha da granulosidade (ou granularidade) do paralelismo na implementação de

uma aplicação, ou na otimização da movimentação de dados entre os processadores ou entre níveis da hierarquia de memória.

Muitas áreas têm apresentado trabalhos que usaram a tecnologia de *cluster* e MPI. Alguns são destacados a seguir. Na previsão meteorológica, (TEHRANIAN *et al*, 2006) os autores propuseram um *framework* robusto e tolerante a falhas para processamento em tempo real de dados gerados por satélite, os quais atingiam cerca de 1,5TB de dados por dia. Os resultados obtidos também contribuíram para a escolha dos algoritmos que devem ser utilizados para não comprometer a confiabilidade e alta disponibilidade exigida pelo *cluster*. Em (CARPENTER, 2001), é relatada uma aplicação comercial desenvolvida para análise de fenômenos meteorológicos cujos resultados são fornecidos para várias outras companhias, tais como as de aviação comercial e indústrias energéticas. A área de cobertura dos dados compreende uma região de 5.760 x 3.600 km, monitorada 4 vezes ao dia, totalizando 60 horas. Possui também grades horizontais de 40 km, com 32 níveis verticais. Os dados foram processados em várias arquiteturas, sendo duas delas formadas por *clusters* com MPI, a saber: Compaq Alpha (4 processadores Alpha 21264, 500 MHz, por nó) e outro com 32 processadores Pentium III (SGI 1200 PIII, 700 MHz, 2 por nó). Outro exemplo de aplicação nesta área foi apresentado por Fischer (1999), no qual ele descreve a paralelização de um algoritmo para previsão numérica do tempo, sendo fruto de uma cooperação internacional de 13 países iniciada em 1991. Esse sistema é executado duas vezes ao dia e abrange vários países da Europa e o Marrocos. Apresenta, também, eficiência computacional e portabilidade para várias plataformas.

Nas áreas de biologia molecular e genética, o sequenciamento de DNA tem se beneficiado enormemente do desempenho obtido com os *clusters*. Em (CHEN, 2003) é apresentado um algoritmo para calcular os alinhamentos ótimos e próximos do ótimo para duas sequências em tempo linear e quadrático, mostrando como o algoritmo pode ser paralelizado eficientemente para ser executado em um *cluster* ou *grid*, de modo a reduzir o tempo de execução significativamente. Usando um *cluster* com 64 nós os autores obtiveram um *speedup* de 41 ao estudar duas cadeias de DNA de comprimento 816.394 e 580.074, respectivamente. As implementações do MPI utilizadas foram a MPICH e MPICH-G2. Eles também observaram que, em *clusters* homogêneos, a divisão de tarefas entre os nós é mais eficiente quando se distribui equitativamente o número de colunas adjacentes da matriz para cada processador. Foi apresentada por Boukerche (2004) uma estratégia eficiente para implementar um algoritmo para comparação de longas sequências de DNA, ao invés de utilizar métodos heurísticos, que apesar de mais rápidos, não forneciam a precisão requerida

pelos biólogos. A avaliação do desempenho foi feita num *cluster* de *workstations* e os resultados indicaram uma solução melhor para o sequenciamento do DNA, quando comparado com resultados obtidos com os esquemas anteriores.

Em (REHN, 2003) são apresentados os resultados obtidos na simulação numérica de fluxos reativos de geometria complexa, que são aplicados em Dinâmica de Fluidos Computacional com o auxílio da supercomputação proporcionada pelos *clusters*. Nesse trabalho os processos que envolvem a combustão do hidrogênio foram analisados em detalhes, visando a melhoria da segurança dos sistemas e a redução de acidentes em ambientes que utilizam este gás como fonte de energia alternativa. Em outro trabalho nessa área, (LONG, 2001), foram estudadas as simulações dos fluxos das turbulências aeroacústicas geradas por helicópteros e navios para mostrar as capacidades do *cluster* em aplicações aeroespaciais. A grande demanda por pessoal capacitado para uso de aplicações de alto desempenho computacional levou a universidade onde eles trabalham a criar um curso específico nesta área.

Como exemplo de experiência no uso de *clusters*, no *Lawrence Livermore National Laboratory*, Eugene D. Brooks III (HAYES, 1992) relata que a computação em *cluster* pode oferecer ao usuário a ideia de um sistema com grande capacidade de armazenamento e memória que jamais seria obtida em um ambiente comum de PC’s, mas com a vantagem de parecer-se com ele. Duas questões fundamentais devem ser consideradas na construção de um *cluster*: primeiro, se o acesso dos usuários for complicado, com muitos nomes de *hosts* e que devam ser encontrados manualmente, então essa complexidade acabará afetando as vantagens da economia obtida com essa arquitetura. Segundo, o acesso aos recursos do *cluster* deve ser confiável. Mesmo naqueles com centenas de computadores, sempre há vários deles com algum problema e as chances de um usuário em particular não conseguir acessar os seus arquivos é pequena. No entanto, um usuário insatisfeito não se conformará com o sistema e dirá isto a todos os outros que encontrar, mesmo que os outros não enfrentem algum problema. Isto diminuirá a confiabilidade no sistema. Então, para ser aceito, o sistema de *software* para a computação em *cluster* deve estabelecer uma única visão do sistema e ser altamente tolerante à falhas. Em relação à computação massivamente paralela, os *clusters* não são a solução final, pois nem todas as aplicações podem ser executadas por *causa* do *overhead* típico dos sistemas de comunicação baseados em *software* existentes nessa arquitetura.

Em outro relato, Thomas Nash (HAYES, 1992) fala sobre a experiência do Fermilab, que desde os meados dos anos 80, buscava a paralelização de aplicações para estudo de dados

de física de alta energia e teoria da cromodinâmica quântica, para serem usadas nos *hardwares* que apresentassem custo e confiabilidade aceitável. Inicialmente baseados no processador Motorola 68020, mais de 500 módulos de processamento foram construídos, sendo que os dados eram transmitidos via servidores de E/S. Os dados eram posteriormente analisados pelos físicos. Em 1989, os processadores Motorola foram substituídos por um *cluster* de mais de 300 *workstations* baseados em processadores RISC, interconectados por uma rede Ethernet, e que processavam mais de 30 TB de dados por ano. Um outro *cluster*, composto por mais de 256 processadores e atingindo um pico de 5 GFLOPS foi utilizado por dois anos. O sistema, chamado ACPMAPS, foi atualizado para mais de 600 processadores Intel 860 e esperava-se que excedesse 50 GFLOPS. Este *cluster* seria usado na relaxação de Monte Carlo (processos de migração e relaxação energética do estado excitado), que é um método numérico para a resolução de problemas da teoria da cromodinâmica quântica. O Fermilab desenvolveu duas ferramentas que permitem aos cientistas identificarem a estrutura de seus algoritmos e então tê-los mapeados para a arquitetura em execução. São elas, o CPS (*Cooperative Process Software*), que fornece eficientemente as subrotinas de envio e recebimento de blocos de dados dos processos executando em cada nó, controlando um sistema único que enfileira os pedidos de transferência de dados e chamadas que estão aguardando um processo livre. Suporta ainda a sincronização dos processos. Outra ferramenta, a Canopy, permite aos físicos teóricos definir os algoritmos em termos de *grids*, locais nos *grids*, conectividade nos locais, campos nos locais e tarefas que executam cálculos nos campos. Esses programas podem ser depurados em uma *workstation* e são mapeados para os nós em um sistema paralelo em tempo de execução. A vantagem é que o cientista não se preocupa com nós e mensagens, somente com locais e campos do problema. Atualmente o principal objetivo do Fermilab na computação paralela é dar suporte à análise dos dados reconstruídos pelos *clusters*. Mais de 300 cientistas colaboram neste quesito.

Quatro orientações surgiram da experiência e sucesso desse laboratório na utilização de *clusters*:

- Encontre a menos ameaçadora abordagem para a computação paralela;
- Paralelize até o nível mais alto possível;
- Dê suporte aos cientistas em explicitamente direcionar o paralelismo com ferramentas que permitam a eles pensarem em termos de seus conceitos usuais; e

- Estabeleça contornos familiares dentro dos nós paralelos de modo que o *software* possa ser depurado numa *workstation*.

3.1 APLICAÇÕES COMERCIAIS

O uso de *clusters* é normalmente visto como um método de computação especializado usado somente pela comunidade científica para pesquisa em aplicações computacionais intensivas, especializadas e extremamente complexas (CHIOLA, 1998). Esta visão tem sido apoiada pela crescente popularização de *clusters* de PC's como alternativa de baixo custo aos tradicionais supercomputadores em laboratórios científicos das universidades e centros de supercomputação. Entretanto, segundo Chiola (1998), isto representa uma visão estreita da computação em *cluster*. Ela exclui uma grande porcentagem de aplicações que usam essa arquitetura hoje em dia, que são as de uso comercial. Por exemplo, uma área de aplicação em que os *clusters* estão fazendo sucesso é na indústria cinematográfica. O filme "*Titanic*" teve seus efeitos visuais renderizados em um grande *cluster* de máquinas Alpha rodando Linux. Outro projeto, a produção do filme de animação "*Toy Story*" também envolveu um grande *cluster* de *workstations* (CHIOLA, 1998).

Nos dias atuais, muitos filmes não poderiam sequer ser feitos sem o auxílio dos computadores, pois dependem totalmente de ambientes e personagens virtuais, como por exemplo, a recente trilogia de "*STAR WARS*", envolvendo os episódios iniciais da saga.

Em nosso país também temos um grande exemplo do uso intensivo de *clusters*, que é a sua utilização na exploração de jazidas de petróleo, onde são feitas simulações sísmicas sobre a existência de possíveis reservatórios em regiões submarinas. Os dados são interpretados por geógrafos e a empresa responsável decide ou não pela exploração. A PETROBRÁS mantém em seu centro de pesquisas aproximadamente 22% do quadro funcional com titulação de mestre ou doutor e cerca de 500 projetos de pesquisa em andamento (CENPES, 2009).

3.2 PARALELIZAÇÃO DE ALGORITMOS

Criar um algoritmo sequencial para resolver um problema computacional é a forma tradicional dos ensinamentos de programação e comumente vivenciada na prática. Estamos acostumados a pensar nas tarefas como tendo início, meio e fim. A execução sequencial de

um programa, na qual um processo é executado somente após o término de um anterior, ocorre entre o começo do seu processo inicial até o término do último processo (EVANS, 1995).

> "Um algoritmo paralelo deve possui quatro atributos: concorrência, escalabilidade, localidade e modularidade. *Concorrência* refere-se à habilidade de executar muitas ações simultaneamente; isto é essencial se um programa executa em muitos processadores. *Escalabilidade* indica capacidade de aumentar o número de processadores e é igualmente importante, visto como a contagem de processadores tende a crescer na maioria dos ambientes. *Localidade* significa a razão entre acessos a memória local e acessos à memória remota (comunicação); esta é a chave para alta performance em arquiteturas multicomputador. *Modularidade* --- a decomposição de entidades complexas em componentes mais simples --- é um aspecto essencial da engenharia de *software*, tanto em computação paralela quanto em sequencial." (FOSTER, 1995) (tradução nossa).

Ao analisar um algoritmo a ser paralelizado, o programador deve procurar descobrir quais são as partes que não dependem uma da outra. Se um resultado não depender de outros anteriores, então a execução paralela é possível, aumentando, assim, o desempenho do algoritmo.

Existem alguns tipos de paralelismo que o programador deve conhecer. São eles:

- Paralelismo de dados;
- Paralelismo funcional; e
- Paralelismo de objetos.
- Paralelismo de domínio.

No paralelismo de dados (Figura 12), o processador executa as mesmas instruções sobre dados diferentes. Por exemplo, na multiplicação de matrizes, as operações realizadas sobre as linhas e colunas são independentes e podem ser feitas em paralelo. Vejamos:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$A \cdot B = \{a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}\}$$

Equação 6. Multiplicação de matrizes.

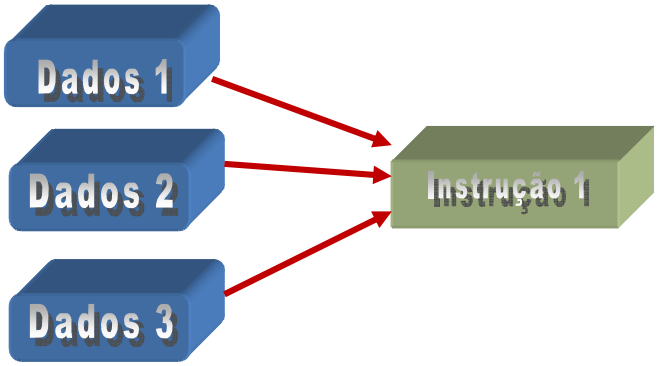


Figura 12. Paralelismo de dados.

Cada processador pode receber uma linha da matriz *A* e uma coluna da matriz *B* e fazer os cálculos independentemente dos outros processadores. O resultado pode ser enviado de volta para o processo que esteja controlando as informações.

No paralelismo funcional (Figura 13), o processador executa instruções diferentes que podem ou não atuar sobre o mesmo conjunto de dados. É aplicado em programas dinâmicos e modulares em que cada tarefa será um programa diferente. Como exemplo, temos o problema do "produtor-consumidor", no qual o processo "produtor" atua sobre os dados gerando alguma informação que será "consumida" (execução de alguma tarefa diferente sobre esses dados) pelo processo "consumidor".

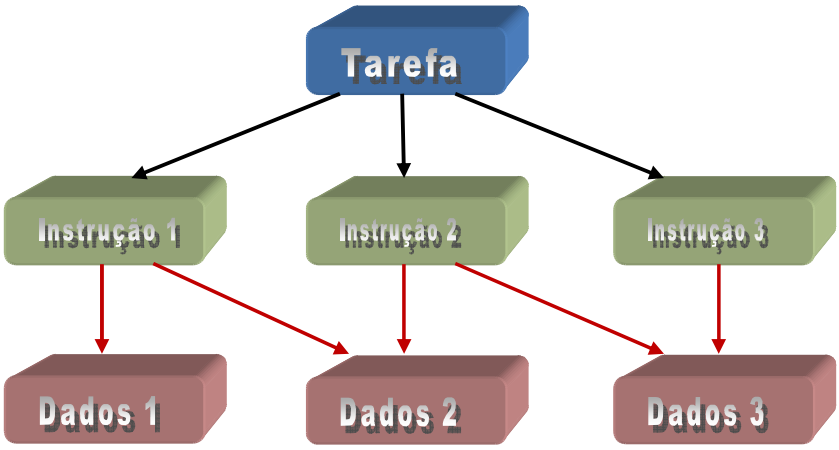


Figura 13. Paralelismo funcional.

Já no paralelismo de objetos (Figura 14), um conceito mais recente, os objetos estão distribuídos por uma rede (como a *Internet*), podendo ser acessados por métodos (funções) em processadores distintos para uma finalidade específica.

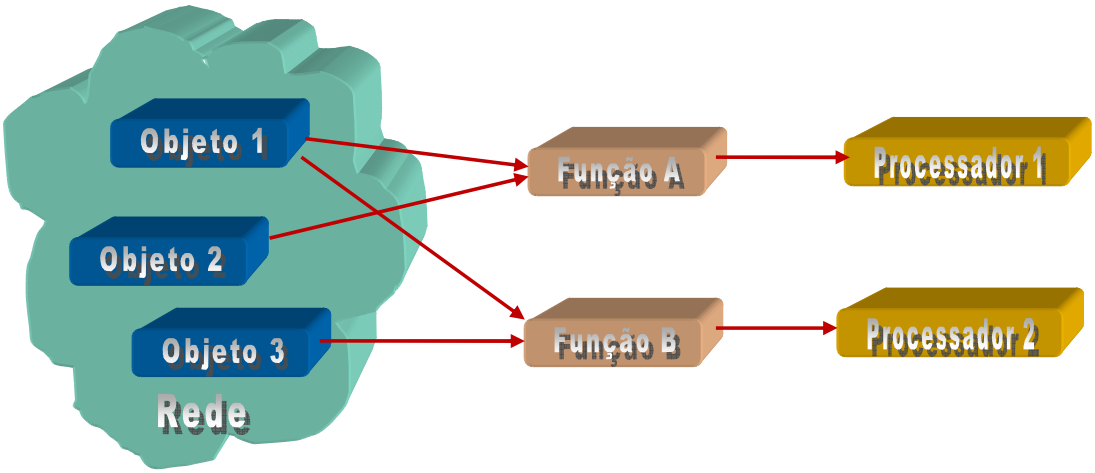
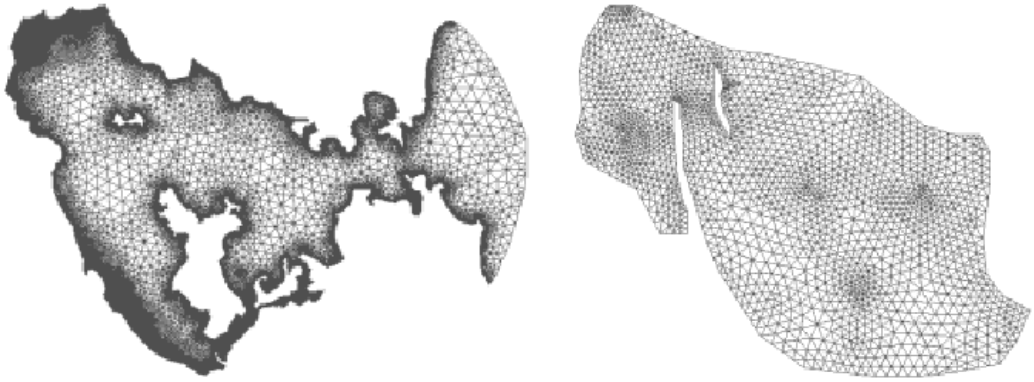


Figura 14. Paralelismo de objetos.

No paralelismo de domínio, um domínio pode ser representado por grafos (Figura 15), onde o paralelismo é obtido particionando-se o grafo, no qual as células do domínio são representadas pelos vértices e as arestas representam os dados trocados entre os subdomínios (CARVALHO, 2002).



(a) (b)

Figura 15. Paralelismo de domínio com grafos modelando: (a) a Baia da Guanabara e (b) um reservatório de petróleo (CARVALHO, 2002)

Maiores detalhes podem ser visto em Carvalho (2002), que apresenta o problema do particionamento de domínio e descreve vários algoritmos utilizados para resolvê-lo.

Toda a dificuldade encontrada na paralelização de um algoritmo compreende o entendimento sobre o que o algoritmo tenta resolver e também descobrir o que pode ser executado em paralelo.

Outro ponto a ser considerado é a granulosidade do código (também conhecida como *granularidade*), a qual pode ser classificada em um extremo como fina, quando o paralelismo ocorre em nível de instruções, e em outro extremo como grossa, quando o paralelismo ocorre

em nível de aplicações. Entretanto, existe uma variedade de graus de paralelização entre este dois níveis, podendo envolver *threads*, procedimentos ou estruturas lógicas do programa, de forma que será mais razoável definir a granulosidade como uma medida relativa: quanto mais fina, os cálculos por processador tendem a ser mais rápidos e a quantidade de troca de mensagem maior, causando sobrecarga no sistema de comunicação e prejudicando o desempenho do sistema paralelo como um todo. Melhor, então, que a granulosidade seja a mais grossa possível (HAYES, 1992), de modo que haja pouca comunicação entre as partes executando concorrentemente. Estruturas como matrizes, por exemplo, são fáceis de paralelizar, pois os cálculos sobre estas normalmente envolvem operações independentes sobre linhas ou colunas. Assim, cada processador pode executar operações sobre elas de forma independente e os resultados parciais podem ser juntados por um único computador.

Embora o MPI seja muito eficiente, a tarefa de paralelizar um programa sequencial não é fácil e nem todos os algoritmos podem ser paralelizados. O programador deve seguir alguns cuidados básicos, ficando atento à que partes do código serão executadas simultaneamente e como será feita esta troca de mensagens e sincronização dos dados. Por exemplo, se um processo está enviando alguma mensagem, o destinatário deve estar esperando por ela, caso contrário ocorrerá um erro de comunicação e o objetivo não será atingido, podendo haver até a interrupção da execução.

Outra questão importante a ser considerada é o balanceamento de carga, que consiste em distribuir as tarefas de acordo com a capacidade de processamento de cada nó. Esse conceito é muito parecido com o que é utilizado na *web* (o acesso a um site popular, por exemplo), onde uma requisição para abrir uma página pode estar sendo feita por muitos usuários ao mesmo tempo e é direcionada para o servidor que estiver menos ocupado, mas com a diferença de que cada processo pode estar em um ponto distinto de execução. Ao ser atingido um ponto de sincronização, os processos mais rápidos ficarão esperando os mais lentos. Então, mesmo que o *cluster* seja constituído por muitas máquinas, o desempenho global será determinado pela máquina mais lenta, principalmente no caso de um *cluster* heterogêneo, o que é muito fácil de acontecer, pois basta acrescentar novas máquinas de tempos em tempos.

Procurando facilitar o trabalho do programador, Foster (1995) propôs uma metodologia para o desenvolvimento de programas paralelos que consiste de quatro estágios: Particionamento, Comunicação, Aglomeração e Mapeamento (PCAM) (Figura 16). Cada estágio é descrito a seguir:

1. *Particionamento:* Consiste em decompor em tarefas menores tanto a computação que é executada quanto os dados que são computados nesta operação. Não se preocupa onde será executado e sim com o reconhecimento do quê pode ser executado em paralelo. Carvalho (2002) e Moretti (1998) apresentam alguns algoritmos para particionamento de domínio e discutem métodos para automatizar este processo;
2. *Comunicação:* Define as estruturas e algoritmos apropriados, pois a comunicação é requerida para coordenar a execução das tarefas;
3. *Aglomeração.* As estruturas definidas nos dois estágios anteriores são avaliadas em relação aos requisitos de performance e custos de implementação. As tarefas podem ser combinadas, se necessário, em tarefas maiores para aumentar a performance ou reduzir os cursos de desenvolvimento; e
4. *Mapeamento.* Para atingir os objetivos de maximização de processamento e minimização de comunicação, cada tarefa é atribuída para um processador de modo a tentar satisfazer essa competição. O mapeamento pode ser estático ou determinado em tempo de execução com a ajuda de algoritmos de balanceamento de carga.

A aplicação prática desta metodologia está exposta no Capítulo 5, onde descrevemos a forma como foi feita a paralelização do algoritmo estudado neste trabalho.

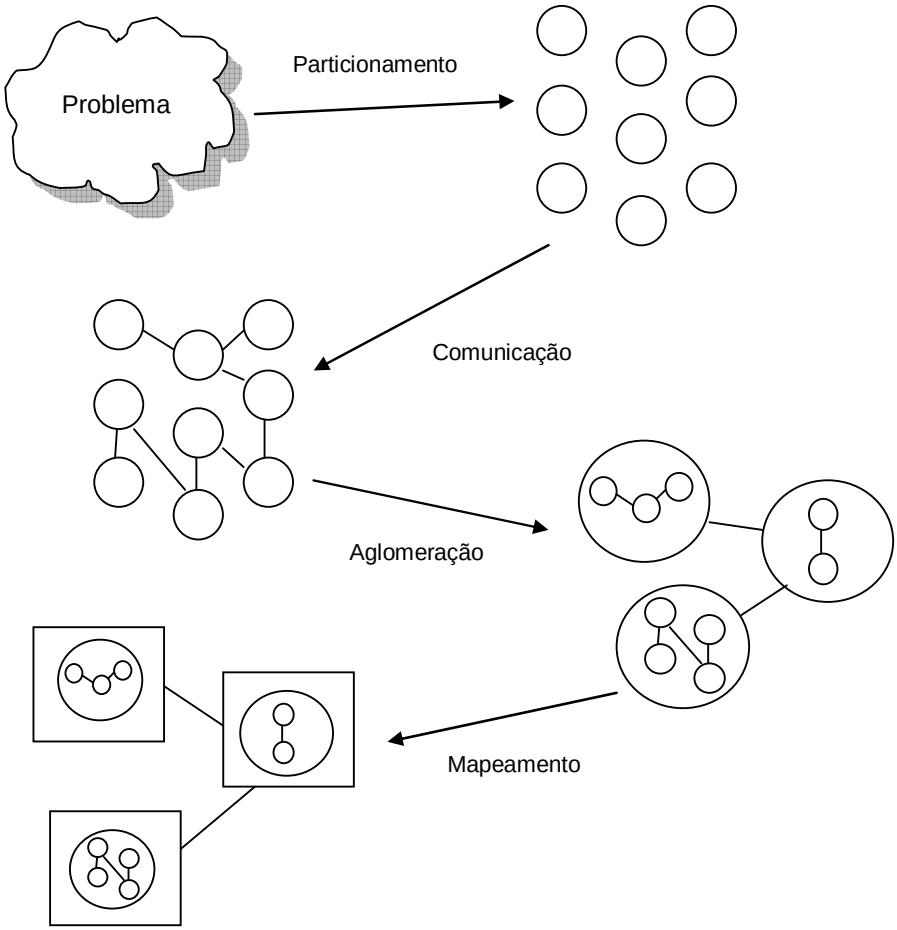


Figura 16. Metodologia PCAM (FOSTER, 1995).

Capítulo IV

A Simulação da Adsorção de Moléculas

O algoritmo em estudo neste trabalho foi desenvolvido em FORTRAN 90, sendo utilizado para a simulação da adsorção de moléculas polissegmentadas usando a técnica de Monte Carlo no *ensemble* grande canônico, que é o conjunto de todos os estados possíveis (configurações) para sistemas no qual os potenciais químicos, a área para adsorção e a temperatura estão fixos (CABRAL *et al*, 2003a, 2003b). Existem várias técnicas de Monte Carlo, mas a mais utilizada é a de Metropolis, que recebeu esse nome devido ao seu criador, Nicholas Metropolis, em 1953. O nome Monte Carlo também foi atribuído por ele, em homenagem ao famoso cassino da cidade de Monte Carlo, em Mônaco (AMAR, 2006). Essa técnica faz uso intenso de números aleatórios para estabelecer distribuições probabilísticas uniformes. Segundo Amar (2006) o Método de Monte Carlo desempenha um papel muito importante em simulações na Ciência da Computação e Engenharia, pois é utilizado para aproximar, de forma tão precisa quanto possível, as propriedades de um sistema dentro de uma determinada distribuição estatística.

A adsorção é um fenômeno que pode ser usado na separação ou purificação de uma mistura de gases. Nesse fenômeno, uma mistura de gases entra em contato com um sólido (adsorvente) que possui maior afinidade química com um dos componentes da mistura

gasosa. Dessa maneira, um dos componentes “adere” preferencialmente sobre a superfície do sólido, propiciando uma separação altamente específica. O algoritmo desenvolvido simula o fenômeno da adsorção em um modelo de sólido bidimensional.

A primeira etapa do algoritmo consiste em fazer a leitura dos parâmetros iniciais, os quais são especificados em um arquivo de configuração (Quadro 5) e, então, modela as superfícies bidimensionais em uma estrutura matricial quadrada de ordem 100x100, por exemplo, chamada Lattice.

```
Tipo de simulação
 1
 Nome do Arquivo de Configuração:
 simRC1_seq.cfg
 Nome do Arquivo de Saía dos Resultados:
 simRC1_seq.dat
 Dimensão do Lattice:
 100
 Número Inicial de Moléculas:
 25 25
 Número de Segmentos de Cada Molécula:
 8    8
 Razão de Sítios Ácidos:
 0.45
 Dimensão dos Grãos de Sítios Ácidos:
 1
 Energia de sítios  dividida por K*T:
 0.0  1.0D0  0.0  1.0D0
 Energia de contato dividida por  K*T:
 0.5d0   0.0  0.0   0.5d0
 Constante de adsorção do componente1:
 1.0D0
 Constante de adsorção do componente2:
 1.0D0
 Número de Sólidos em Cada Ponto Simulado:
 32
 Número de Configurações para Equilíbrio:
 100000000
 Numero de Configurações para Cada Média:
 1000000
 Número de Cálculos de Média:
 10
 Intervalo para Impressão dos Resultados:
 5000000
 Mínimo Valor de LogP:
 -5
 Máximo Valor de LogP:
 5
 Número de Pontos Simulados:
 30
 N_inf: Número de Pontos Simulados (auxiliar)
 1.00d0   1.00d0
 r1 – dif. entre as energ.dos sítios 'a' e 'b' em relação ao comp(1)
 1.0d0
 r2 – dif. entre as energ.dos sítios 'a' e 'b' em relação ao comp(2)
 1.0d0
```

Quadro 5. Arquivo de configuração inicial.

Os parâmetros que influem diretamente no tempo de processamento são: a dimensão do Lattice (reticulado), o número de segmentos de cada molécula, a razão dos sítios ácidos, a dimensão dos mesmos, o número de sólidos em cada ponto simulado, o número de configurações para equilíbrio, número de configurações para cada média e o número de cálculos de média, sendo estes três últimos os responsáveis pela quantidade de repetições que o algoritmo fará para atingir o número de configurações necessárias para o equilíbrio.

A Figura 17, no eixo horizontal, exemplifica a quantidade de configurações necessárias para uma dada configuração de sólido (para um reticulado de 100X100 com grãos ácidos de dimensão Dg=1 na fração de 0,22 para a adsorção de uma molécula com 4 segmentos) (CABRAL *et al*, 2003a).

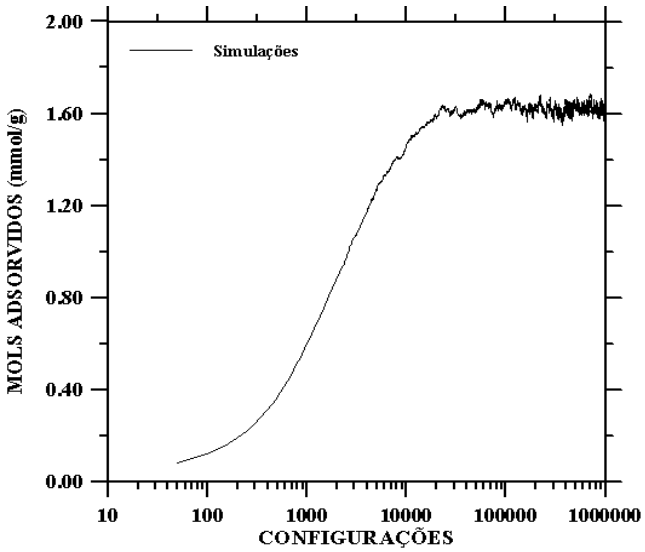


Figura 17. Determinação do número mínimo de configurações necessárias para o cálculo das propriedades médias

A Figura 18 apresenta alguns exemplos de superfícies heterogêneas (Lattice) geradas pela simulação, com quatro reticulados quadrados de 40x40 com 30% de sítios ativos, randomicamente distribuídos em grãos com dimensões (A) Dg=1, (B) Dg=4, (C) Dg=8, e (D) Dg=12 (CABRAL *et al*, 2003a). Os pontos escuros formam os sítios ativos, que são áreas onde ocorrem as ligações mais fortes entre as moléculas adsorvidas e o adsorvente. A localização destes sítios ativos é determinada aleatoriamente em tempo de execução e eles podem ser agrupados, formando grãos de várias dimensões (Dg).

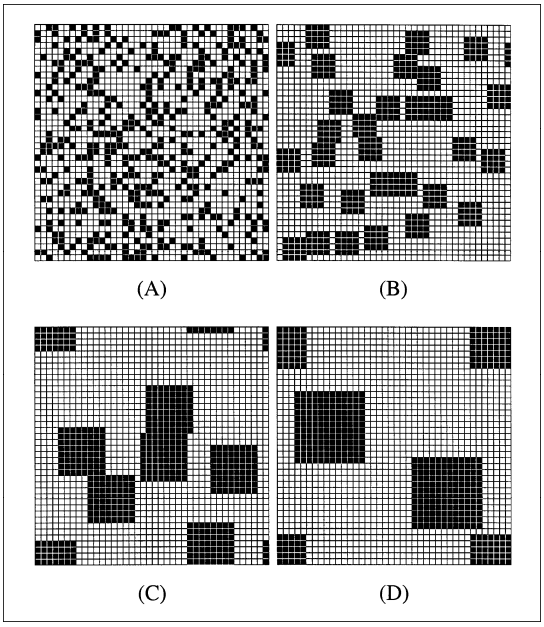


Figura 18. Exemplos de superfícies heterogêneas.

A próxima etapa do algoritmo realiza a execução exaustiva de três movimentos sobre o sólido adsorvente, na seguinte ordem: locomoção de uma molécula; inserção de uma molécula; remoção de uma molécula adsorvida.

Cada um desses movimentos segue o algoritmo de Metropolis (CABRAL *et al*, 2003a, 2003b) para o *ensemble* grande canônico. O objetivo é calcular o número médio de moléculas adsorvidas em determinada medição de pressão e temperatura, obtendo-se, dessa forma, as isotermas de adsorção. Os resultados destas simulações podem ser correlacionados com dados experimentais da adsorção de substâncias puras, objetivando-se, assim, a determinação de parâmetros do modelo de simulação. Conhecidos tais parâmetros, é possível, então, prever a adsorção de misturas binárias (2 gases diferentes) e ternárias (3 gases diferentes).

O algoritmo amostra os estados (ou configurações) das moléculas segundo a função de distribuição de probabilidades do *ensemble* grande canônico. A cada movimento é verificada a diferença de energia entre duas configurações consecutivas. Por não levar em conta a probabilidade das configurações em si, mas sim a razão entre elas. A Equação 7 mostra a razão entre duas probabilidades de configurações, uma anterior (*Um*) e uma nova (*Un*), para o caso da locomoção de uma molécula sobre o sólido adsorvente.

$$\frac{P_n}{P_m} = \exp\left[-\frac{U_n - U_m}{k_b T} \right]$$

Equação 7. Razão entre duas probabilidades de configurações.

onde :

k_b : constante de Boltzmann (é a constante física que relaciona temperatura e energia de moléculas);

T : Temperatura;

U_m : energia da configuração do estado *m* (anterior); e

U_n : energia da configuração do estado *n* (atual).

O trecho do algoritmo que toma mais tempo de processamento pode ser resumido nos seguintes passos, que resumem o algoritmo de Metropolis (também conhecido como Metropolis-Hastings) e que também pode ser usado nas mais diversas áreas como arquitetura, economia e imagens por computador (AMAR, 2006).

1. Uma configuração (ou estado) inicial aleatória, ou seja, com valores aleatórios para todos os graus de liberdade do sistema é gerada respeitando as suas restrições. Atribui-se o índice *m* a essa configuração, a qual é aceita para a amostra.
2. Geração de uma nova configuração de índice *n*, através de um dos movimentos (locomoção, inserção ou retirada) que apresenta pequenas alterações nas coordenadas da configuração *m*.
3. Se a energia da configuração *n* (atual) for menor que a da configuração *m* (anterior), então inclui-se a configuração *n* na amostra e se atribui a ela o índice *m*. Caso contrário, realizam-se os passos descritos nos subitens (a) e (b) abaixo:
    a. Gera-se um novo número aleatório entre 0 e 1;
    b. Se esse número for menor que *Pn/Pm*, a configuração *n* é aceita na amostra, e ela passa a ter o índice *m*. Caso contrário, o índice *m* permanece na configuração original.
4. Repete-se os passos 2 e 3 até que seja satisfeito algum critério de parada. Essas repetições são ditas como um passo de Monte Carlo (MC).

A execução deste algoritmo consome horas de processamento em uma única máquina, pois, segundo (CABRAL *et al*, 2003a), cada passo de Monte Carlo deve ser analisado milhões ou até bilhões de vezes, dependendo das condições simuladas, sendo que periodicamente são calculadas as propriedades médias do sistema.

Toda a simulação também é repetida várias vezes, de acordo com os dados iniciais, para diferentes sólidos com a mesma fração de sítios ativos e mesma dimensão, porém, com diferentes topologias de sólido adsorvente. Esse procedimento é aplicado para evitar que as configurações do sólido tenham alguma influência sobre os resultados obtidos nas simulações (CABRAL *et al,* 2003a).

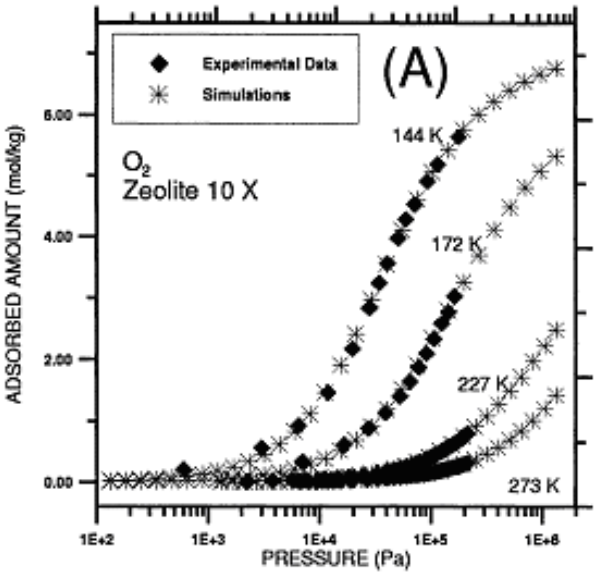


Figura 19. Gráfico comparativo da simulação da quantidade de gás adsorvido com os resultados experimentais, utilizando ainda o algoritmo sequencial (CABRAL, 2003a).

Esses resultados são posteriormente plotados em gráficos e comparados com valores já conhecidos da área, conforme mostrado na Figura 19.

Essas simulações exigem grande esforço computacional e isto tem limitado algumas aplicações em problemas da Engenharia Química (CABRAL, 2003a). Portanto, sua paralelização trouxe benefícios diretos para a área, possibilitando a obtenção dos resultados mais rapidamente, além da utilização de muito mais informações que as que hoje são analisadas, como por exemplo, moléculas e superfícies maiores, que seriam inviáveis na execução sequencial.

Capítulo V

Metodologia

A metodologia adotada para conseguir a paralelização do algoritmo de simulação está descrita neste capítulo e foi baseada na metodologia proposta por Foster (1995), tendo consistido das seguintes fases (cada fase será melhor detalhada mais adiante):

Fase 1: Análise do algoritmo sequencial: Esta fase serviu para determinar o domínio do problema e quais partes serão beneficiadas com a execução paralela, conforme (EVANS, 1995). Segundo Foster (1995), particionar o domínio é a tarefa de dividir os dados que serão distribuídos no processamento paralelo, procurando descobrir o máximo de paralelismo possível, podendo ser aplicada tanto à computação quantos aos dados. Quanto menores as partes de computação e de dados, melhor é a qualidade desse particionamento. Então, foi feito um estudo sobre o particionamento do algoritmo descrito no capítulo anterior, estabelecendo quais partes são executadas pelo nó mestre e quais as executadas pelos escravos. Também foram definidos os estágios de Comunicação, Aglomeração e Mapeamento, sugeridos por Foster em sua metodologia.

Fase 2: Implementação do algoritmo paralelo utilizando 4 abordagens diferentes.

Fase 3: Validação das implementações paralelas através da comparação com os dados obtidos pelo algoritmo original. Nesta fase os resultados obtidos devem concordar com os experimentos realizados pelo pesquisador da área de Engenharia Química que idealizou o algoritmo (CABRAL, 2003a), atestando, assim, a validade do algoritmo para uso em ambiente de *cluster* de computadores.

Fase 4: Comparação das execuções das diversas versões, levando em consideração o tempo de execução e a quantidade de moléculas envolvidas na simulação. Esta fase serve para determinar qual abordagem traz maior vantagem para a aplicação em termos de redução do tempo de execução.

5.1 ANÁLISE DO ALGORITMO SEQUENCIAL

Para entender melhor a lógica do programa e a forma de resolução adotada em cada versão, apresentamos no Quadro 6 um pseudo-código resumido do algoritmo sequencial estudado.

```
1)Definição do ambiente (variáveis, arquivos, etc.)
2)Leitura do arquivo de configuração
3)Alocação das estruturas matriciais e de controle
4)Início da simulação:
5)    Gerar configuração inicial
6)    Para cada ponto P, faça:
7)          Para cada sólido, faça:
8)                Defina novo sólido
9)                Para cada Fase, faça:
10)                     Para cada Ciclo, faça:
11)                           Para cada Configuração, faça:
12)                                 Mova molécula
13)                                 Insira molécula
14)                                 Remova molécula
15)                           Calcule as médias do Ciclo
16)                           Atualize Acumuladores
17)         Calcule médias, variância e covariância
18)         Salve os dados do Ponto
19)Fim da simulação.
```

Quadro 6. Pseudo-código resumido da versão sequencial.

Após ler os dados de entrada do arquivo de configuração, são alocadas as estruturas que irão receber as informações, por exemplo, a matriz dos sólidos, a das coordenadas dos sítios ácidos, a dos lugares ocupados e assim por diante. A configuração inicial (sólido inicial) é então gerada e o processamento ocorre variando-se os pontos de pressão e, para cada ponto, vários outros sólidos são criados. Para cada sólido são feitas as movimentações e calculadas as variações de energia e número de moléculas, entre outras variáveis.

As linhas 5, 8, 12, 13 e 14 resumem sub-rotinas que contêm muitos laços de repetição, cada um deles chamando a rotina de geração de números aleatórios *n* vezes, além de percorrer as matrizes que representam as superfícies do sólido várias vezes em cada iteração, de acordo com os dados de configuração lidos do arquivo de entrada. As linhas 6, 7, 9, 10 e 11

dependem também das configurações iniciais e determinam quantas vezes haverá a repetição dos cálculos, podendo chegar até a bilhões de vezes (CABRAL, 2003a). Estas são as partes do algoritmo responsáveis pela demora na execução. A cada iteração dos pontos de pressão os dados são totalizados e as configurações finais das moléculas são gravadas em arquivo, sendo que os dados de quantidade de moléculas adsorvidas e variação da energia total é gravada em um único arquivo.

5.2 PARALELIZAÇÃO DO ALGORITMO

O ponto de partida para a paralelização do algoritmo analisado foi particionar o domínio do problema. Após estudá-lo e entender seu funcionamento, verificou-se que os sólidos gerados para o cálculo dos pontos de pressão apresentavam a independência de dados, durante os cálculos, necessária para a execução paralela. Isto está de acordo com (HAYES, 1992), que sugere paralelizar ao nível mais alto possível, conforme suas conclusões obtidas com a experiência do uso de *clusters* no Fermilab, onde também foi utilizado o método Monte Carlo para resolução de problemas de Cromodinâmica Quântica.

Na versão sequencial, para cada ponto de pressão, vários sólidos são gerados baseados em uma configuração inicial e então são efetuados os movimentos de moléculas sobre ele para obtenção da energia total, em um processo que refaz os cálculos estatísticos cerca de bilhões de vezes para obter as médias desejadas. A seguir, para o próximo ponto, um novo conjunto de sólidos é gerado, completamente diferente e independente do anterior e os cálculos são refeitos, e assim sucessivamente, até que todos os pontos tenham sido processados.

Dividir o número de sólidos calculados em cada ponto de pressão pelo número de máquinas do *cluster* parece ser uma forma interessante de se obter o particionamento do domínio. Podemos pensar nessa alocação de carga entre os processadores utilizando esse número de sólidos sempre como um múltiplo do número de máquinas utilizadas na simulação (Figura 20). Assim, se temos 8 máquinas escravas, podemos processar 8, 16, 32, 64, ... sólidos por simulação.

Neste ponto, precisamos estabelecer uma diferença sutil entre os termos alocação de carga de trabalho e balanceamento de carga. O primeiro, no contexto deste trabalho, refere-se à distribuição dos dados entre os nós, não levando em conta as características individuais de cada nó nem qualquer análise de capacidade de processamento. Quanto ao segundo, existem

na literatura muitos algoritmos para balanceamento de carga, como citados em (GORINO, 2006), que geralmente consistem em soluções baseadas em análise do *hardware* e refere-se à capacidade do *cluster* distribuir as tarefas de acordo com o poder de processamento e ociosidade de cada nó.

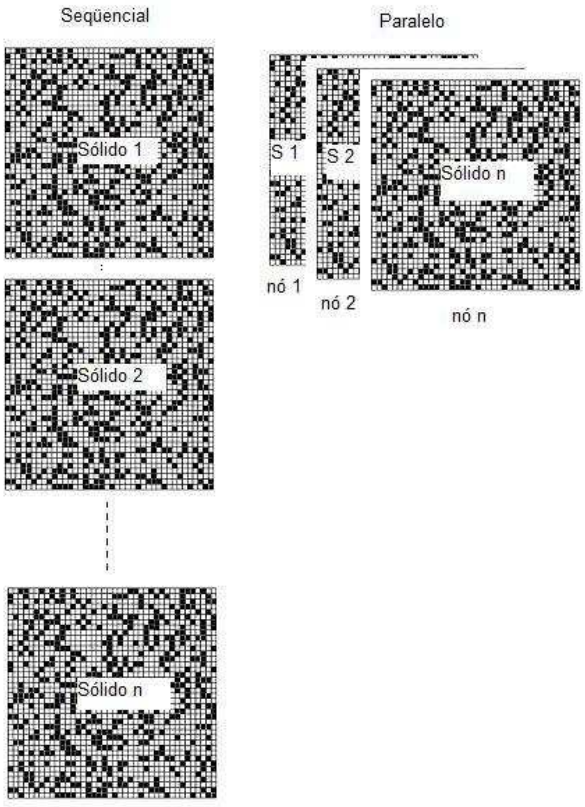


Figura 20. Distribuição dos sólidos nas versões sequencial e paralela (1.ª versão).

O segundo estágio da metodologia de Foster, o de comunicação, envolveu a definição de como as trocas de mensagens seriam executadas. A determinação da granulosidade procurou minimizá-las entre os processos, conforme (EVANS, 1995). Foi estabelecido, então, que o nó mestre é o responsável pela leitura dos dados iniciais e geração da configuração inicial, a qual é transmitida a todos os outros nós. Dessa forma, a troca de mensagens entre os processos foi minimizada.

5.2.1 Primeira versão paralela

Inicialmente, foi utilizado um nó exclusivamente para gerar os números aleatórios para o método de Monte Carlo (MC), onde todos os outros nós trocavam mensagens com ele,

passando os parâmetros para o cálculo pelo servidor, que devolvia o número gerado para o nó solicitante. Cada número aleatório serve como semente para um posterior. Isto foi feito para tentar garantir a distribuição uniforme dos números gerados, que é uma característica importante para o método de Monte Carlo, pois cada máquina, mesmo que idêntica a outra no *cluster* quanto aos componentes internos, pode gerar números aleatórios diferentes para uma mesma semente, influenciados por vários fatores (ROSENTHAL, 2000), por isso, os números aleatórios gerados por computadores são chamados de *pseudoaleatórios*. Devido ao intenso uso que esse algoritmo faz desses números e a granulosidade tornar-se extremamente fina dentro de cada laço (linhas 12, 13 e 14), o *overhead* introduzido pela camada de comunicação tornou essa versão paralela muito mais lenta que a versão sequencial, o que comprovou a sugestão exposta por Hayes (1992), de que devemos procurar paralelizar ao nível mais alto possível, evitando a granulosidade muito fina. Logo, essa abordagem tornou-se inviável.

Para contornar essa situação e ainda assim manter a característica aleatória e uniformemente distribuída requerida pelo método de Monte Carlo, foi utilizado um vetor inicial de números aleatórios, gerados em sequência, cada um a partir do anterior, pelo nó mestre, com tantos elementos quantos eram os pontos da simulação (Figura 21). Esse vetor é então transmitido aos outros nós, que por sua vez, selecionam o elemento correspondente ao ponto de pressão que está sendo processado.

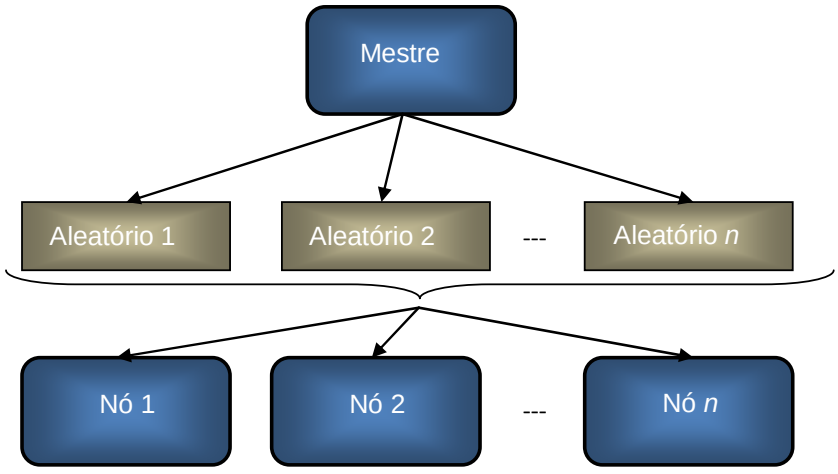


Figura 21. Geração e distribuição dos números aleatórios iniciais.

A partir desse número, cada nó calcula seus próprios números aleatórios durante a execução do algoritmo e não há troca de mensagens com essa finalidade. Esta abordagem é semelhante à utilizada em (KORNISS, 1999). De posse da configuração inicial, cada nó gera os seus próprios sólidos, em número igual para todos, ou seja, se o número total de sólidos a

processar for 32, cada escravo será responsável por 4 deles (em uma simulação com 8 escravos, por exemplo), executando as linhas 9 a 21 do pseudo-código do Quadro 6.

Ao final, todos os resultados são enviados de volta ao mestre, que é o responsável também pela recepção, totalização e gravação dos dados calculados por cada nó (representado pelas linhas 22 a 24 do pseudo-código do Quadro 6). Estas etapas contemplam os estágios de aglomeração e mapeamento da metodologia de Foster (1995). O pseudo-código desta versão paralela está mostrado no Quadro 7.

```
Nó mestre:
      1)Definição do ambiente (variáveis, arquivos, etc.)
      2)Leitura do arquivo de configuração
      3)Alocação das estruturas matriciais e de controle
      4)Início da simulação:
      5)    Gerar números aleatórios iniciais -> Num. Pontos
      6)    Gerar configuração inicial
      7)    Transmitir dados para os escravos
      8)    Para cada ponto P, faça:

Nó escravo:
      9) Receber dados de controle e da configuração inicial
      10)   Para cada ponto P, faça:
      11)         Para cada sólido, faça:
      12)               Defina novo sólido
      13)               Para cada Fase, faça:
      14)                     Para cada Ciclo, faça:
      15)                           Para cada Configuração, faça:
      16)                                 Mova molécula
      17)                                 Insira molécula
      18)                                 Remova molécula
      19)                           Calcule as médias do Ciclo
      20)                           Atualize Acumuladores
      21)Transmitir dados calculados para nó mestre

Nó mestre:
      22)   Receber os dados dos nós escravos e totalizar
      23)   Calcular médias, variância e covariância
      24)   Salvar os dados do Ponto
      25)Fim da simulação.
```

Quadro 7. Pseudo-código resumido da 1.ª versão paralela.

Essa primeira versão serviu para percebermos que, embora tenhamos conseguido o principal objetivo, que era a execução paralela do algoritmo, ele ainda poderia ser melhorado. Apesar da divisão dos sólidos pelo número de processadores parecer distribuir a carga de trabalho de maneira equitativa, na prática, esta não foi obtida de forma satisfatória. Em um *cluster* heterogêneo, como é o caso do que foi usado na simulação (ver página 73), ocorre que as máquinas com maior poder de processamento executarão os cálculos com maior rapidez e, consequentemente, terminarão primeiro. Como, nesta versão, o algoritmo foi estruturado de

modo que o nó mestre recebe todos os cálculos dos nós escravos para só então processar as médias e gravar os dados finais, as máquinas mais rápidas têm que ficar aguardando as mais lentas chegarem ao mesmo ponto, onde os dados são sincronizados. Isto implica que, na prática, a velocidade de processamento de todo o *cluster* é equivalente à velocidade da máquina mais lenta! Mesmo sendo melhor que a execução em apenas uma máquina, esta limitação pode ser contornada para reduzir ainda mais o tempo de processamento. O algoritmo deve levar em conta a capacidade de processamento de cada nó, a fim de obter uma alocação de carga melhor distribuída, pois, afinal, queremos explorar todo o potencial oferecido pelo *cluster*!

Todos os resultados e conclusões desta versão e seguintes estão expostos e comentados no próximo Capítulo.

5.2.2 Segunda versão paralela

A segunda versão implementada teve o objetivo de verificar a viabilidade de outra abordagem para o particionamento do domínio. Tanto a versão sequencial quanto a primeira paralela executam variando-se os pontos de pressão. Para cada ponto são gerados *n* sólidos e realizados os cálculos.

Nesta segunda versão optou-se por inverter o *loop* das linhas 8 e 11 do pseudo-código do Quadro 7. Então, um sólido não é trocado para cada ponto simulado, os pontos é que variam. Vale ressaltar que a relação pontos de pressão/sólidos não foi alterada, garantindo, assim, que todas as versões processam a mesma quantidade de dados. O novo pseudo-código é mostrado no Quadro 8.

A linha 8 da primeira versão paralela foi mudada de lugar nesta segunda versão para a linha 23, porque o nó mestre agora só recebe o resultado final para o sólido como um todo. Para isto, foi criada uma estrutura vetorial em cada nó que acumula os dados de cada ponto para depois enviá-los ao mestre. Este, por sua vez, percorre estes vetores na ordem dos pontos (cada ponto é representado pelo índice do vetor), fazendo os cálculos necessários. Assim, esta parte do algoritmo original não precisou ser alterada.

```
Nó mestre:
      1)Definição do ambiente (variáveis, arquivos, etc.)
      2)Leitura do arquivo de configuração
      3)Alocação das estruturas matriciais e de controle
      4)Início da simulação:
      5)    Gerar números aleatórios iniciais -> Num. de sólidos
      6)    Gerar configuração inicial
      7)    Transmitir dados para os escravos

Nó escravo:
      8) Receber dados de controle e da configuração inicial
      9)    Para cada Sólido S, faça:
      10)         Defina novo sólido
      11)         Para cada ponto P, faça:
      12)               Para cada Fase, faça:
      13)                     Para cada Ciclo, faça:
      14)                           Para cada Configuração, faça:
      15)                                 Mova molécula
      16)                                 Insira molécula
      17)                                 Remova molécula
      18)                           Calcule as médias do Ciclo
      19)                           Atualize Acumuladores
      20)Transmitir dados calculados para nó mestre

Nó mestre:
      22)   Receber os dados dos nós escravos e totalizar
      23)   Para cada ponto P, faça:
      24)         Calcule médias, variância e covariância
      25)         Salve os dados do Ponto
      25)Fim da simulação.
```

Quadro 8. Pseudo-código resumido da 2.ª versão paralela.

5.2.3 Terceira versão paralela

A segunda versão mostrou-se ligeiramente mais lenta que a primeira, mas ainda assim produziu os resultados corretos, conforme apresentado no próximo capítulo. O importante é que ela mostrou-se viável quanto ao particionamento e serviu de base para a próxima versão do programa. O pseudo-código da terceira versão está mostrado no Quadro 9.

Para não fugir do que foi estabelecido na fase de particionamento do domínio, a solução adotada foi que todos os sólidos são previamente calculados pelo mestre e armazenados em estruturas matriciais. A seguir, o mestre fica aguardando a solicitação de um sólido por um nó escravo (linhas 11 a 15).

Cada sólido enviado é registrado para que seja feito o controle sobre qual está sendo processado e qual está disponível. Desta forma, no início, é feita uma sincronização após a geração dos sólidos pelo mestre e todos os escravos solicitam um sólido e o recebem. Aquele

que terminar primeiro solicitará outro, até que todos tenham sido processados ou estejam em processamento.

```
Nó mestre:
      1)Definição do ambiente (variáveis, arquivos, etc.)
      2)Leitura do arquivo de configuração
      3)Alocação das estruturas matriciais e de controle
      4)Início da simulação:
      5)    Gerar números aleatórios iniciais -> Num. de pontos
      6)    Gerar configuração inicial
      7)    Transmitir dados para os escravos
      8)    Para cada solido, faça:
      9)          Definir novo sólido
      10)         Armazenar sólido
      11)   Enquanto não acabar os sólidos
      12)         Aguardar solicitação de sólidos
      13)         Se há sólido disponível, então
      14)               Enviar solido para solicitante
      15)         senão, avisar que não há
      16)   Aguardar resultados

Nó escravo:
      17) Receber dados de controle e da configuração inicial
      18) Faça:
      19)   Solicite sólido
      20)   Se recebeu aviso de fim, então
      22)         saia do loop
      23)   Receba sólido
      24)   Para cada ponto P, faça:
      25)         Para cada Fase, faça:
      26)               Para cada Ciclo, faça:
      27)                     Para cada Configuração, faça:
      28)                           Mova molécula
      29)                           Insira molécula
      30)                           Remova molécula
      31)                     Calcule as médias do Ciclo
      32)                     Atualize Acumuladores
      33)         Acumule dados do ponto
      34)Transmitir dados calculados para nó mestre

Nó mestre:
      35)   Receber os dados dos nós escravos e totalizar
      36)   Para cada ponto P, faça:
      37)         Calcule médias, variância e covariância
      38)         Salve os dados do Ponto
      39)Fim da simulação.
```

Quadro 9. Pseudo-código resumido da 3.ª versão paralela.

Assim, esta alocação de carga de trabalho e em nível de aplicação possibilita que as máquinas mais rápidas calculem mais sólidos que as mais lentas, aproveitando o seu poder de computação maior, e consequentemente, reduzindo o tempo de processamento global. Nenhuma análise de *hardware* é feita. Simplesmente as mais rápidas processarão mais dados!

Embora tenha havido um ligeiro aumento da troca de mensagens e a complexidade de controle dos processos também tenha aumentado, diminuíram as chances de que uma máquina mais rápida fique muito tempo esperando pelas mais lentas para sincronizar os dados. É interessante frisar que esta abordagem ficou muito semelhante ao problema do "produtor-consumidor", muito comum nas aulas sobre Sistemas Operacionais. Aqui, o nó mestre "produz" os sólidos e os escravos os "consomem", mas com a diferença de que a produção é fixa, acontecedo somente no início da simulação e os "consumidores" devolvem o resultado.

5.2.4 Quarta versão paralela

A quarta versão, baseada na terceira, surgiu devido a alguns problemas encontrados durante o desenvolvimento do trabalho. Procuramos, agora, também implementar a característica de tolerância a falhas. Isto foi necessário porque muitas simulações (algumas das que levaram dias para terminar) foram abortadas quando uma ou outra máquina parou de funcionar, seja por problema de *hardware* ou de queda de energia. O pseudo-código é mostrado no Quadro 10.

A complexidade aumentou mais ainda porque agora o nó mestre executa vários procedimentos de controle. Por exemplo, para cada sólido, ele fica em um *loop* aguardando até que uma mensagem chegue. Isto foi preciso para evitar que houvesse um *deadlock* caso o mestre e um escravo estivessem em um ponto de recebimento ao mesmo tempo (na implementação da LAM-MPI, a diretiva *MPI_Send()* não é bloqueante, mas a *MPI_Recv()* é). Ao receber uma mensagem, o mestre sai do *loop* e verifica qual o tipo, através da *tag*. Se for uma solicitação de sólido, o mestre irá percorrer uma lista mantida para controlar os que estão em processamento e os disponíveis, armazenando também os tempos de inicio e fim de processamento de cada um. Enquanto houver um sólido disponível, ele o enviará para o nó solicitante. Caso contrário, encontrará o primeiro que ainda não terminou e verificará o tempo de processamento médio de todos os sólidos, multiplicando-o por 3,5 (este fator foi estimado experimentalmente por inspeção, após sucessivas simulações). Caso o tempo atual seja maior que isto, pode ser um sinal de que o escravo que começou a processar aquele sólido esteja fora do ar (não necessariamente). Então o mestre redireciona os dados desse sólido para o processo que está solicitando e registra esse redirecionamento.

```
Nó mestre:
      1)Definição do ambiente (variáveis, arquivos, etc.)
      2)Leitura do arquivo de configuração
      3)Alocação das estruturas matriciais e de controle
      4)Início da simulação:
      5)    Gerar números aleatórios iniciais -> Num. de sólidos
      6)    Gerar configuração inicial
      7)    Transmitir dados para os escravos
      8)    Para cada solido, faça:
      9)          Definir novo sólido
      10)         Armazenar sólido
      11)   Enquanto não acabar os sólidos
      12)         Enquanto o n. de resultados < n.o sólidos, faça:
      13)               Aguardar solicitação de sólidos
      14)         Se for pedido de solido
      15)               Verifica solido disponivel
      16)                     Se há sólido disponível, então
      17)                           Enviar solido para solicitante
      18)                     senão, verificar tempo processamento
      19)                     Se tempo processamento > media *3,5
      20)                           Redireciona solido
      21)         Se for aviso de resultado, então
      22)               Recebe resultado
      23)               Atualiza numero resultados
      24)         Se for aviso de processamento, então
      25)               Atualiza tempo do processo
      26)         Se n.o de resultados = n.o sólidos, então
      27)               Avisa processos para encerrar
      28)   Aguardar resultados

Nó escravo:
      29) Receber dados de controle e da configuração inicial
      30) Faça:
      31)   Solicite sólido
      32)   Se recebeu aviso de fim, então
      33)         saia do loop ( e encerra)
      34)   Receba sólido
      35)   Para cada ponto P, faça:
      36)         Para cada Fase, faça:
      37)               Para cada Ciclo, faça:
      38)                     Para cada Configuração, faça:
      39)                           Se mod(Conf,IPrint) == 0, então
      40)                                 Avise p/ atualizar tempo
      41)                           Mova molécula
      42)                           Insira molécula
      43)                           Remova molécula
      44)                     Calcule as médias do Ciclo
      45)                     Atualize Acumuladores
      46)         Acumule dados do ponto
      47)         Avisar o mestre dos resultados
      48)         Se recebeu aviso de continuar, então
      49)               Transmitir dados calculados para nó mestre

Nó mestre:
      50)   Receber os dados dos nós escravos e totalizar
      51)   Para cada ponto P, faça:
      52)         Calcule médias, variância e covariância
      53)         Salve os dados do Ponto
      54)Fim da simulação.
```

Quadro 10. Pseudo-código resumido da 4.ª versão paralela.

Quando cada processo termina de calcular seu sólido, envia um aviso ao mestre de que está pronto para mandar os resultados. O mestre, então, sinaliza e se prepara para a recepção. Ao recebê-los, verifica se tinha sido redirecionado para outro. Caso positivo, descarta-o, para evitar duplicidade de dados para aquele mesmo sólido (o que acabaria gerando erro de cálculo quando o mestre fosse computar as médias) e envia uma mensagem para que este nó termine seu processamento, "dispensando-o", dessa forma, de continuar o trabalho, mesmo que seja com outro sólido. É um ponto a ser considerado em versões futuras, para tentar não desperdiçar esse tempo que foi gasto no processamento. Caso seja um resultado "normal" (que não foi redirecionado para outro), ou do nó que recebeu o sólido redirecionado, recebe-o e guarda os valores. Quando o número de resultados recebidos for igual ao de sólidos, avisa a todos os nós de que não há mais nada a fazer, para que terminem, e então passa para a fase final, a de cálculo das médias.

Nesta versão não há mais o ponto de sincronização dos dados via *MPI_Reduce(),* que agora são enviados via *MPI_Send()* e recebidos com *MPI_Recv(),* porém o mestre deve acumular o dado recebido com os anteriores. Assim, a cada sólido terminado, o mestre receberá os dados e distribuirá outro, se houver. Isto melhora ainda mais o tempo de execução, pois, na versão anterior, mesmo que as máquinas mais rápidas processassem mais informações, ainda havia o ponto de sincronização dos dados com o mestre, as chamadas ao *MPI_Reduce(),* o que causava ainda alguma ociosidade. Infelizmente, essa versão não é 100% tolerante a falhas, pois se o nó mestre sair do ar, todo o processamento deverá ser refeito. A ideia é que, mesmo que quase todas as máquinas do *cluster* tenham algum problema, com pelo menos duas máquinas, o mestre e um escravo, a simulação vá até o final sem precisar da intervenção do usuário. Contudo, dependendo do quanto já tiver sido processado, pode não ser viável manter a execução, pois o tempo necessário para recalcular os sólidos perdidos pode ser muito maior do que recomeçar com todas as máquinas. Quanto menos máquinas "saírem do ar" durante o processamento, melhor. A boa notícia é que existem procedimentos implementados no ambiente LAM-MPI, conforme descritos por (MARTINS, 2005), que podem ser utilizados para retomar o processamento a partir de um ponto previamente salvo (*checkpoint*). Isto facilita ainda mais a utilização e aumenta a confiabilidade das simulações com os *clusters* de computadores.

Capítulo VI

Resultados Obtidos

Todas as simulações foram feitas num *cluster* heterogêneo tipo *Beowulf* pertencente ao grupo de pesquisas HPPCA e instalado no LECAD do Departamento de Informática da Universidade Estadual de Maringá, sendo constituído por 9 máquinas (1 mestre e 8 escravos). A configuração de cada nó pode ser vista na Tabela 1.

Tabela 1. Configuração do *cluster* HPPCA

Nó	CPU	Clock (GHz)	RAM (MB)	HD (GB)
Mestre	Athlon	1	512	20
Escravo 1	Pentium IV	1.8	1024	40
Escravo 2	Pentium IV	1.8	1024	40
Escravo 3	Pentium IV	1.8	1024	40
Escravo 4	Pentium IV	1.8	1024	40
Escravo 5	Pentium IV HT	3.0	1024	80
Escravo 6	Pentium IV HT	3.0	1024	80
Escravo 7	Pentium IV HT	3.0	1024	80
Escravo 8	Pentium IV HT	3.0	1024	80

O ambiente de execução em todos os nós é formado pelo sistema operacional Red Hat Linux, versão 2.4.29-smp, com implementação LAM-MPI versão 7.0.4, utilizando também o sistema de administração de *clusters* OSCAR (*Open Source Cluster Application Resources*) (OSCAR, 2009). O compilador utilizado para a linguagem FORTRAN 90 foi o GNU g95 (G95, 2009), que não é padrão da instalação LAM-MPI, por isso foi necessária uma adaptação no ambiente, redirecionando o comando de compilação `mpif77` para esse novo compilador. Maiores informações podem ser obtidas no *site* www.lam-mpi.org.

6.1 VALIDAÇÃO DOS RESULTADOS

Para demonstrar a eficiência dessas abordagens, foi estipulada uma configuração inicial (ver Quadro 5) para as simulações com 30 pontos de pressão, 32 sólidos e uma superfície de dimensões 200x200 (portanto, quatro vezes maior que o tamanho de sólidos normalmente simulado, que é 100x100), e foi executada nas quatro versões. Esses valores foram determinados pelo autor do algoritmo original, baseado em sua experiência na utilização do mesmo.

A validação dos algoritmos paralelos foi feita comparando-se os dados obtidos com os equivalentes gerados pela versão sequencial. É preciso ter certeza que o algoritmo está produzindo os resultados corretos. A validação é mostrada no gráfico da Figura 22, que apresenta as curvas de isoterma obtidas, tanto pela versão sequencial como pelas paralelas.

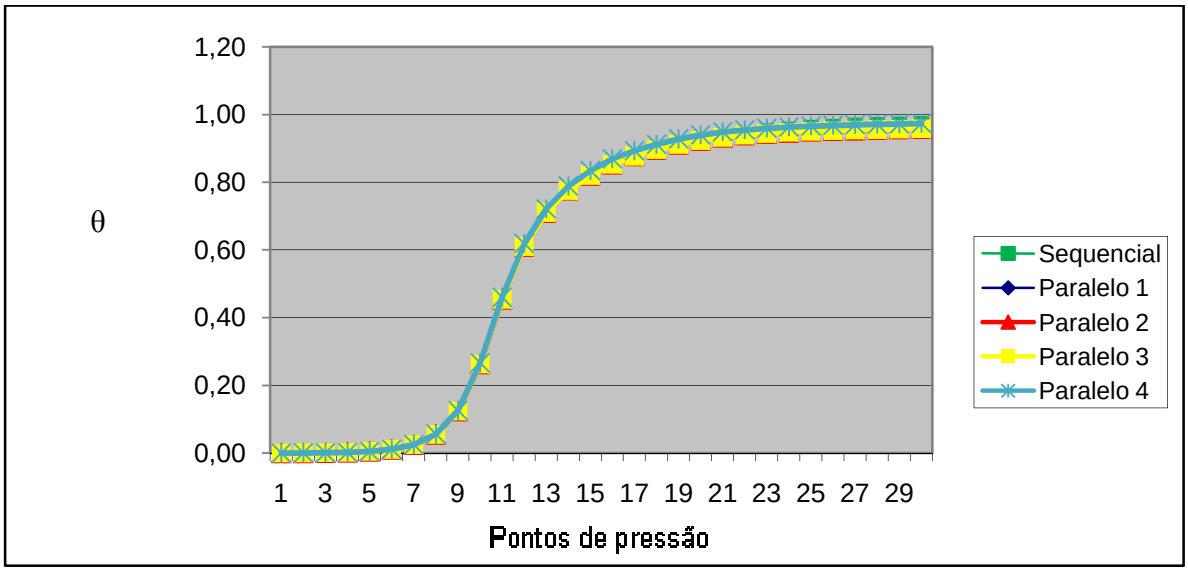


Figura 22. Gráfico das isotermas obtidas com as simulações sequencial e paralelas.

A coluna vertical mostra os valores da fração de cobertura da superíficie *(θ)*, que devem estar entre 0 e 1. Na horizontal estão os pontos de pressão. À medida que a pressão aumenta, mais moléculas são “forçadas” contra o sólido, aumentando também a fração de corbertura da superfície e, portanto, favorecendo a adsorção. Podemos perceber que no ponto de maior pressão, a fração de cobertura fica bem próxima do valor 1, o qual indica a total cobertura do sólido.

Propositalmente este gráfico foi colocado nesta disposição para mostrar visualmente que há somente pequenas variações em alguns pontos, porque, como as simulações não foram executadas ao mesmo tempo e pelas próprias características do Método de Monte Carlo, que é produzir uma distribuição estatística uniforme, além de os números aleatórios apresentarem variações quando se usa mais de uma máquina (ROSENTHAL, 2000), alguma diferença nos resultados dos pontos é até esperada, mas é tão insignificante a ponto de não afetar a curva. A Tabela 2 mostra os valores de θ obtidos por todas as versões e a Tabela 3 apresenta as diferenças obtidas pelas versões paralelas em comparação com os valores obtidos pela versão sequencial (coluna 2 da Tabela 2).

Tabela 2. Valores de θ obtidos nas simulações

Ponto	Sequencial	Paralelo 1	Paralelo 2	Paralelo 3	Paralelo4
1	1,2454730000E-04	1,2453150000E-04	1,2454290000E-04	1,2454950000E-04	1,2844020000E-04
2	3,4293260000E-04	3,4301450000E-04	3,4284960000E-04	3,4290780000E-04	3,5168350000E-04
3	8,8459010000E-04	8,8478180000E-04	8,8438050000E-04	8,8467180000E-04	9,0321710000E-04
4	2,1081420000E-03	2,1085580000E-03	2,1083200000E-03	2,1084570000E-03	2,1464100000E-03
5	4,8338640000E-03	4,8334770000E-03	4,8329120000E-03	4,8333210000E-03	4,9193360000E-03
6	1,0901440000E-02	1,0902700000E-02	1,0905450000E-02	1,0903200000E-02	1,1094140000E-02
7	2,4487130000E-02	2,4496510000E-02	2,4494000000E-02	2,4493440000E-02	2,4903410000E-02
8	5,5156040000E-02	5,5161160000E-02	5,5149840000E-02	5,5135610000E-02	5,6059190000E-02
9	1,2398210000E-01	1,2390120000E-01	1,2394990000E-01	1,2395590000E-01	1,2572750000E-01
10	2,6291280000E-01	2,6279930000E-01	2,6278060000E-01	2,6299010000E-01	2,6639320000E-01
11	4,5392740000E-01	4,5410030000E-01	4,5399500000E-01	4,5419640000E-01	4,5989650000E-01
12	6,1022090000E-01	6,1033210000E-01	6,1038410000E-01	6,1018740000E-01	6,1926500000E-01
13	7,1075170000E-01	7,1077260000E-01	7,1062580000E-01	7,1079150000E-01	7,2163780000E-01
14	7,7548350000E-01	7,7596920000E-01	7,7599370000E-01	7,7555390000E-01	7,8910930000E-01
15	8,2029440000E-01	8,2148870000E-01	8,2039710000E-01	8,2062240000E-01	8,3451700000E-01
16	8,5307970000E-01	8,5320900000E-01	8,5276360000E-01	8,5337900000E-01	8,6915770000E-01
17	8,7767890000E-01	8,7771640000E-01	8,7795870000E-01	8,7736180000E-01	8,9302040000E-01
18	8,9673460000E-01	8,9673080000E-01	8,9668670000E-01	8,9684670000E-01	9,1204060000E-01
19	9,1221380000E-01	9,1185410000E-01	9,1209940000E-01	9,1186330000E-01	9,2790780000E-01
20	9,2478590000E-01	9,2405490000E-01	9,2345120000E-01	9,2376870000E-01	9,4000590000E-01
21	9,3422270000E-01	9,3416410000E-01	9,3254770000E-01	9,3254070000E-01	9,4899270000E-01
22	9,4364800000E-01	9,4327220000E-01	9,3869570000E-01	9,3923910000E-01	9,5488040000E-01
23	9,5194110000E-01	9,4853930000E-01	9,4371340000E-01	9,4374300000E-01	9,5942560000E-01
24	9,5666800000E-01	9,5325520000E-01	9,4689270000E-01	9,4753700000E-01	9,6326750000E-01
25	9,6195080000E-01	9,5705110000E-01	9,5001260000E-01	9,5032130000E-01	9,6627260000E-01
26	9,6473860000E-01	9,5928900000E-01	9,5209670000E-01	9,5245280000E-01	9,6825740000E-01
27	9,6843810000E-01	9,6162910000E-01	9,5373410000E-01	9,5416630000E-01	9,6979990000E-01
28	9,7076490000E-01	9,6352740000E-01	9,5525280000E-01	9,5576820000E-01	9,7146150000E-01
29	9,7177530000E-01	9,6491450000E-01	9,5672340000E-01	9,5704990000E-01	9,7254820000E-01
30	9,7324370000E-01	9,6603360000E-01	9,5768730000E-01	9,5804000000E-01	9,7369840000E-01

Tabela 3. Diferenças nos Valores de θ obtidos nas versões paralelas em comparação com os da versão sequencial

Ponto	Paralelo 1	Paralelo 2	Paralelo 3	Paralelo4
1	0,0001268594	0,0000353279	-0,0000176640	-0,0312563982
2	-0,0002388224	0,0002420301	0,0000723174	-0,0255178423
3	-0,0002167105	0,0002369459	-0,0000923592	-0,0210572106
4	-0,0001973302	-0,0000844345	-0,0001494207	-0,0181524774
5	0,0000800602	0,0001969439	0,0001123325	-0,0176819207
6	-0,0001155811	-0,0003678413	-0,0001614466	-0,0176765638
7	-0,0003830584	-0,0002805555	-0,0002576864	-0,0169999506
8	-0,0000928275	0,0001124084	0,0003704037	-0,0163744533
9	0,0006525135	0,0002597149	0,0002113208	-0,0140778387
10	0,0004317021	0,0005028283	-0,0002940138	-0,0132378492
11	-0,0003808979	-0,0001489225	-0,0005926058	-0,0131499002
12	-0,0001822291	-0,0002674441	0,0000548982	-0,0148210263
13	-0,0000294055	0,0001771364	-0,0000559971	-0,0153163193
14	-0,0006263189	-0,0006579121	-0,0000907821	-0,0175707156
15	-0,0014559407	-0,0001251990	-0,0003998564	-0,0173384092
16	-0,0001515685	0,0003705398	-0,0003508465	-0,0188470081
17	-0,0000427263	-0,0003187954	0,0003612939	-0,0174796272
18	0,0000042376	0,0000534160	-0,0001250091	-0,0170685953
19	0,0003943155	0,0001254092	0,0003842301	-0,0172043001
20	0,0007904532	0,0014432530	0,0010999303	-0,0164578634
21	0,0000627259	0,0017929344	0,0018004272	-0,0158099348
22	0,0003982417	0,0052480374	0,0046721871	-0,0119031673
23	0,0035735404	0,0086430768	0,0086119824	-0,0078623562
24	0,0035673818	0,0102180694	0,0095445860	-0,0068984224
25	0,0050935037	0,0124104060	0,0120894956	-0,0044927454
26	0,0056487840	0,0131039641	0,0127348486	-0,0036474129
27	0,0070309088	0,0151832110	0,0147369254	-0,0014061818
28	0,0074554612	0,0159792551	0,0154483336	-0,0007175785
29	0,0070600683	0,0154890745	0,0151530915	-0,0007953485
30	0,0074083192	0,0159840747	0,0156216783	-0,0004672006

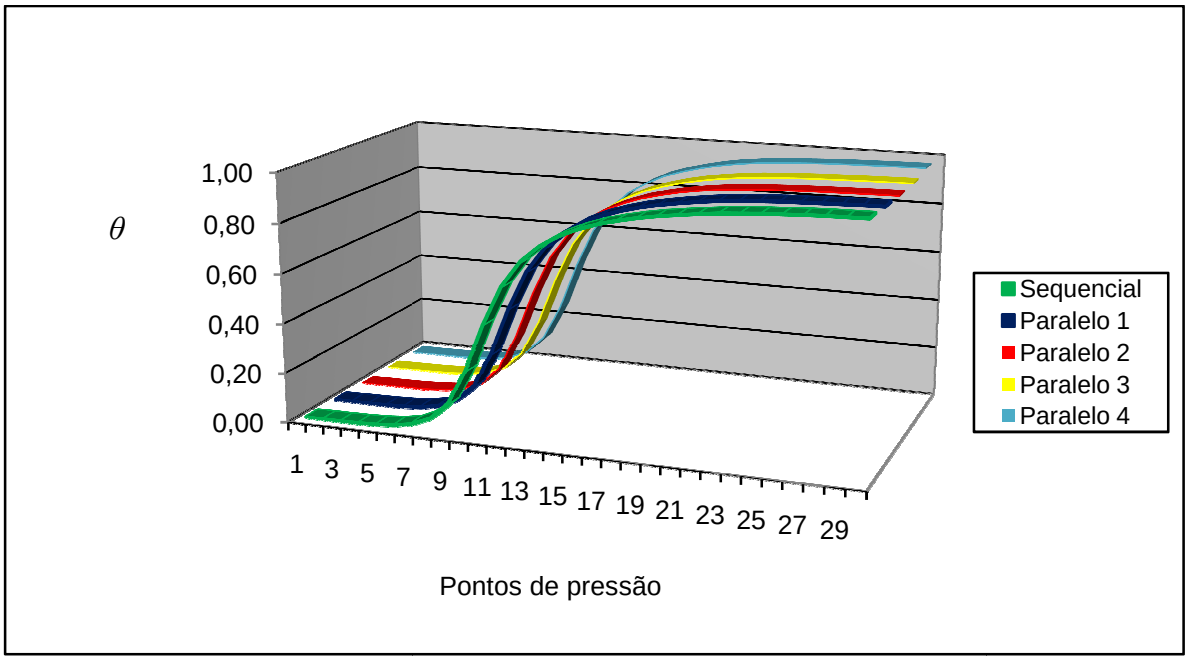


Figura 23. Gráfico das isotermas em outra perspectiva, para distinguir as versões.

A cada vez que uma versão paralela for executada os resultados apresentarão esta pequena variação em pontos distintos ou não, mesmo que seja feita uma simulação imediatamente após a outra. O gráfico da Figura 23 mostra as mesmas isotermas da Figura 22 de outra perspectiva, para podermos distingui-las melhor.

Podemos verificar que os dados produzidos correspondem aos resultados obtidos com a versão sequencial, com pequenas variações (menor que 0,01, para mais ou para menos, na maioria dos pontos, conforme mostrado na Tabela 3). Após constatada a validade desses resultados, resta-nos avaliar o desempenho e a eficiência das versões paralelas.

6.2 AVALIAÇÃO DO DESEMPENHO E DA EFICIÊNCIA

A execução sequencial foi escalonada para a máquina mais rápida do *cluster* (Escravo 8) e terminou após 10 dias, 21 horas e 31 minutos, aproximadamente, após processar todos os pontos e sólidos. Todas as versões paralelas foram executas com 3, 5 e 9 máquinas (contando com o mestre), com os mesmos dados de entrada da versão sequencial.

6.2.1 Desempenho

Os tempos de processamento de cada versão estão listados na Tabela 4 e o gráfico correspondente é mostrado na Figura 24.

Tabela 4. Tempos de processamento (em minutos)

N.º de Máquinas	Sequencial	Paralelo 1	Paralelo 2	Paralelo 3	Paralelo 4
1	15691	---	---	---	---
3	---	7731	8497	7256	6322
5	---	6994	7193	5385	4738
9	---	4124	4172	3133	2640

Foster (1995) define o tempo de execução de cada processador como a soma do tempo de computação propriamente dito, tempo de comunicação e tempo ocioso, porém, neste trabalho eles não são discriminados e estão sendo utilizados os tempos desde o início da simulação até a geração dos arquivos de resultados, sendo que este último consumiu apenas

alguns segundos (informação obtida dos arquivos de *log* das simulações). Estes tempos para leitura e escrita de arquivos não influenciaram negativamente o desempenho dos algoritmos.

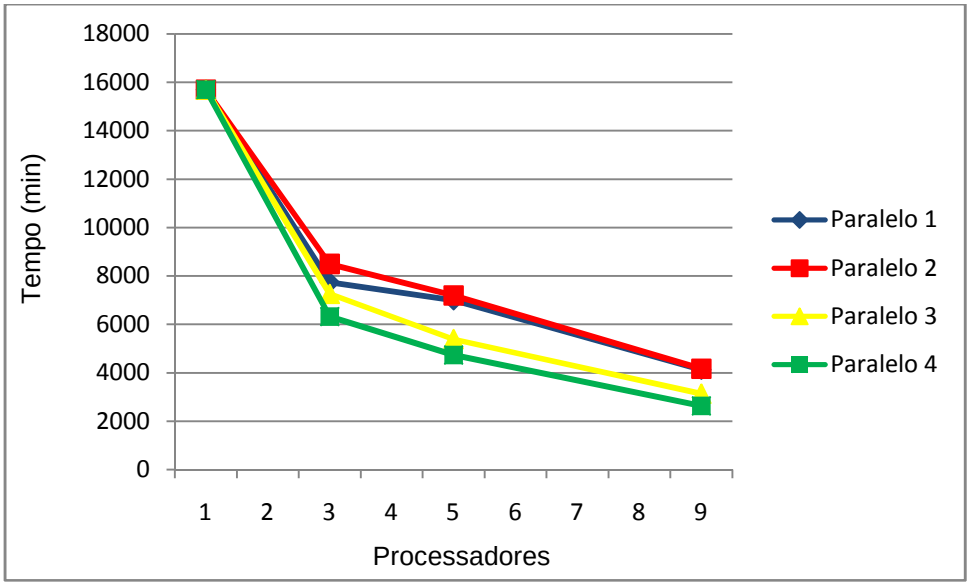


Figura 24. Tempos de processamento.

O *speedup* foi, então, calculado conforme definido no Capítulo 2. Para exemplificar, vejamos o valor para a 1.ª versão paralela com 3 máquinas (mestre + 2 escravos):

$$S = \frac{T_S}{T_P} = \frac{15691}{7731} \sim 2{,}029$$

Equação 8. Cálculo do *speedup*.

Então, com 3 máquinas (1 processo por máquina) obtém-se pouco mais que o dobro da velocidade de processamento, e o tempo cai para pouco menos da metade. O ideal seria obtermos um valor 3, mas devemos lembrar que agora o tempo gasto na comunicação dos processos influi negativamente no tempo de execução total, além do fato de que o nó mestre não efetua os cálculos sobre os sólidos; apenas distribui os dados iniciais e fica aguardando os resultados, mas mesmo assim ele entra no cálculo do *speedup*. A Tabela 5 mostra os outros valores do *speedup* e o gráfico é mostrado na Figura 25, onde é exibido também o valor teórico que seria obtido em uma configuração ideal, na qual não houvesse atrasos devido à camada de comunicação e ociosidade dos processadores. Os valores intermediários foram interpolados.

Tabela 5. Valores de *speedup* obtidos

Versão Paralela	N. º de Máquinas	*Speedup*
1	3	2,029
	5	2,243
	9	3,804
2	3	1,924
	5	2,181
	9	3,761
3	3	2,162
	5	2,913
	9	5,008
4	3	2,57
	5	3,43
	9	5,943

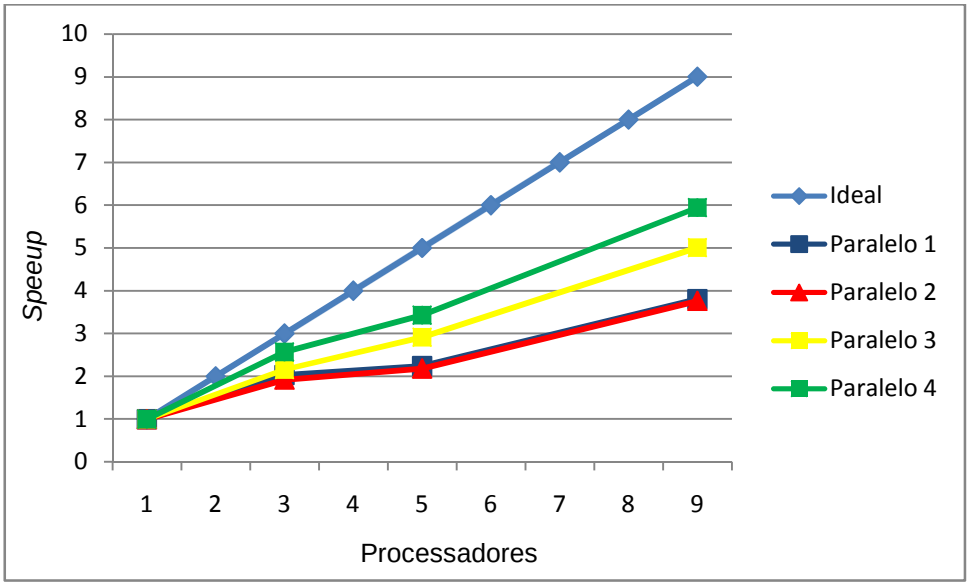


Figura 25. Gráfico do *speedup*.

As simulações executadas com 3 e 5 máquinas foram realizadas, respectivamente, nas máquinas mais rápidas e nas mais lentas, devido à heterogeneidade do *cluster* e para aproveitar o tempo realizando duas simulações simultâneas. Isto explica a ligeira queda das curvas entre 3 e 5 processadores para estas versões. Nas versões 1 e 2, o nó mestre fica quase o tempo todo ocioso até sincronizar e receber os dados dos processos escravos, que também apresentam maior ociosidade, devido ao ponto de sincronização. Como a versão 2 difere da versão 1 apenas na forma como os dados foram particionados, suas curvas são muito parecidas, com uma ligeira vantagem da 1.ª versão, em termos de *speedup*. Já para a versão 3, o nó mestre, apesar de não efetuar os cálculos diretamente sobre os sólidos propriamente ditos e por realizar um melhor controle sobre a distribuição do processamento, fazendo uma

alocação de sólidos para os processadores mais eficiente, contribuiu para o aumento do *speedup.* Isto aconteceu também na versão 4, que ainda possui a característica de ser tolerante a falhas em alguns nós escravos. As versões paralelas não foram executadas em apenas 1 máquina porque a troca de mensagens certamente iria atrasar o processamento, resultando em um tempo maior que a versão sequencial (mais de 11 dias, provavelmente); por isso, o tempo inicial da versão sequencial foi utilizado como ponto inicial também para as versões paralelas, apenas para efeito visual do gráfico.

Com relação aos objetivos propostos e após realizar todas as simulações, podemos observar na Tabela 4 que o melhor tempo foi conseguido com a 4.ª versão paralela executada nas 9 máquinas (2640 minutos), que representa uma redução do tempo de execução em torno de 83,17% (diminuiu para 1 dia e 20 horas) em relação à versão sequencial, conforme é apresentado pela Tabela 6. Como na versão 3, a versão 4 também fez uma melhor distribuição de carga de trabalho entre as máquinas, aproveitando o poder de processamento de cada uma e conseguiu um ganho de tempo adicional pelo fato de que não precisa esperar que todas atinjam o mesmo ponto para sincronizar os dados.

Tabela 6. Ganho de Tempo de processamento (em %)

N.º de Máquinas	Paralelo 1	Paralelo 2	Paralelo 3	Paralelo 4
1	---	---	---	---
3	50,73	45,84	53,75	59,71
5	55,42	54,15	65,68	69,80
9	73,71	73,41	80,03	83,17

Sobre a tolerância a falhas, ressaltamos que essa simulação foi executada sem que houvesse algum tipo de problema com alguma máquina. Assim, essa característica acabou não sendo utilizada, mas seria interessante passar por mais testes para ser aperfeiçoada. Se muitas máquinas pararem, é preciso fazer uma avaliação se compensa reiniciar a simulação ou não, dependendo do que já tiver sido processado. Por exemplo, em uma simulação que leve alguns dias para ser executada, se as máquinas que pararem forem algumas das mais lentas, melhor, mas se for o contrário, as mais lentas vão demorar muito mais para retomar os cálculos que foram perdidos. Neste caso, caberá ao usuário decidir. Foi detectado, nos testes preliminares, que quanto mais cedo ocorrer alguma falha que tire um nó do ar, menor será o impacto no tempo de execução global.

6.2.2 Eficiência

A Tabela 7 mostra a eficiência conseguida com as diferentes versões e o gráfico é mostrado na Figura 26.

Tabela 7: Eficiência das versões paralelas.

Versão Paralela	N.º de Processos	*Eficiência*
1	3	0,6763
	5	0,4486
	9	0,4226
2	3	0,6413
	5	0,4362
	9	0,4178
3	3	0,7206
	5	0,5826
	9	0,5564
4	3	0,79
	5	0,72
	9	0,66

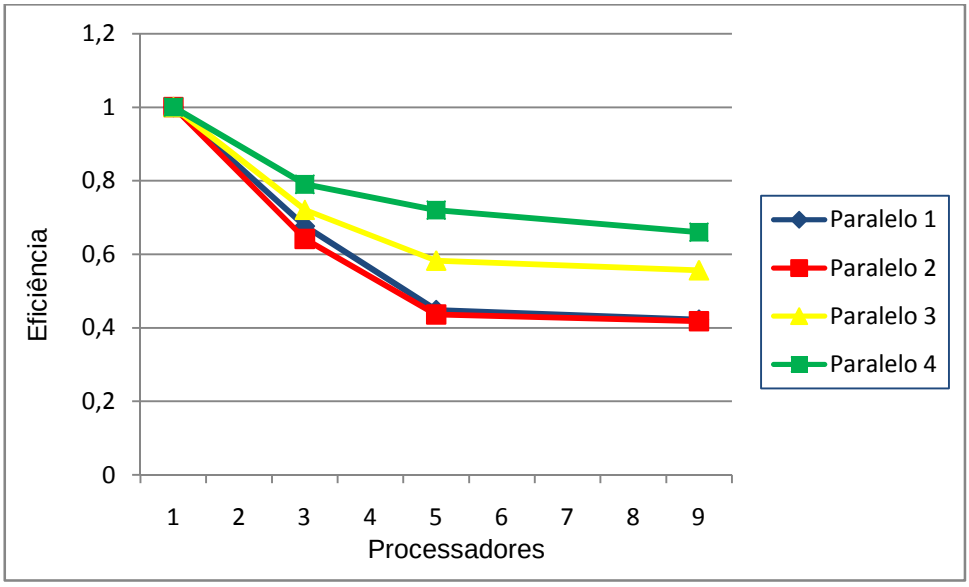


Figura 26. Gráfico da Eficiência das versões paralelas.

Na 4.ª versão do algoritmo praticamente não há ociosidade das máquinas, inclusive do nó mestre, que agora fica quase o tempo todo monitorando a chegada de mensagens, usando diretivas não bloqueantes. Como a sincronização global foi substituída pela diretiva *MPI_Recv()*, as máquinas mais rápidas não precisam mais esperar pelas mais lentas,

diminuindo sua ociosidade. Assim, tivemos uma elevação do *speedup* e, consequentemente, isso fez com que aumentasse ainda mais a eficiência.

É interessante notar que a eficiência diminui à medida que mais processadores são usados. Isto porque quando acrescentamos mais processadores aumentamos também a comunicação entre os processos, o que acaba gerando ociosidade em cada processador, pois cada um tem que esperar que os dados fiquem disponíveis em seus registradores. Embora o aumento do número de processadores faça com que haja redução do tempo de processamento, haveria mais processos trocando dados. Então, a partir de determinado número de processadores, a sobrecarga da comunicação irá atrasar a chegada dos dados ao destino, consumindo mais tempo que o processamento propriamente dito, anulando a vantagem do processamento paralelo. Para sabermos qual o número limite de processos que garanta a vantagem da execução em paralelo é preciso que o algoritmo seja executado em um *cluster* com talvez algumas centenas de processadores, o que, infelizmente, não foi possível neste trabalho.

Capítulo VII

Conclusão

Aplicações paralelas e distribuídas estão se tornando uma solução viável para a solução de problemas complexos, tendo os *clusters* de computadores surgidos como grandes ferramentas capazes de tornar possível o estudo desses problemas. A simulação feita em computadores é uma forma muito útil para que os pesquisadores consigam resultados muito próximos àqueles obtidos com experimentos reais, sem, contudo, ter os custos e muitas vezes também os riscos inerentes a cada experiência.

Dada esta disponibilidade de recursos em termos de *hardware*, é preciso que estudos mais profundos sobre os modelos e as técnicas atualmente empregadas na escrita de programas paralelos sejam feitos. O problema apresentado neste trabalho requer longo tempo de processamento sequencial, o que dificulta o avanço das pesquisas na área, pois limita a quantidade de informação que realmente pode ser obtida. Com a paralelização, um número maior de moléculas poderá ser analisado, contribuindo para um resultado mais expressivo e até mesmo promover novas descobertas para a área em que é aplicado.

Como resultado deste trabalho, as simulações executadas em paralelo mostraram que a redução do tempo de processamento foi extremamente significativa, ainda levando em conta que os dados utilizados superam em muito as simulações executadas normalmente na versão sequencial.

O objetivo da redução do tempo de execução foi atingido já na primeira versão paralela, que foi 74,7% mais rápida que a sequencial, porém a questão da ociosidade das máquinas mais rápidas precisou ser resolvida. Na segunda versão, com uma abordagem de particionamento de domínio diferente, os cálculos foram feitos variando-se os pontos de pressão pelos sólidos, perdeu-se um pouco de desempenho em relação à versão anterior (73,4%), mas foi mínimo (0,3%) e serviu para demonstrar a viabilidade dessa nova abordagem. Na terceira versão foi implementado um algoritmo de distribuição dos sólidos que propiciou uma alocação de carga dinâmica, propiciando que as máquinas com maior poder de processamento fossem servidas com novos dados à medida que terminavam os cálculos de cada sólido, numa abordagem semelhante ao problema do “produtor-consumidor”, porém com algumas diferenças. Essa diminuição da capacidade de computação ociosa mostrou-se eficaz, fazendo com que essa versão atingisse 80% de redução do tempo de execução sequencial. A quarta e última versão buscou reduzir ainda mais esse tempo. Deixando de utilizar as diretivas de comunicação globais para obtenção dos resultados parciais e implementando um algoritmo de tolerância a falhas nos nós, essa versão apresentou-se como a mais eficiente de todas, obtendo uma diferença entre o tempo de execução sequencial e a paralela em torno de 83,17% (com 9 máquinas), o que representa um ganho expressivo. Embora não seja 100% tolerante a falhas, essa versão consegue terminar a simulação mesmo que algumas máquinas tenham problemas e sofram alguma parada, redirecionando o trabalho para aquelas que estiverem funcionando. Contudo, dependendo da quantidade de sólidos processados e quais máquinas tenham saído do ar, pode ser mais vantajoso reiniciar a simulação com todas as máquinas ao invés de continuar e esperar pelo recálculo dos sólidos perdidos.

Após analisarmos os dados obtidos com as simulações pudemos comprovar que cada abordagem adotada apresentou desempenho e eficiência satisfatórios sendo a última, embora um tanto mais complexa que as outras, a que atingiu plenamente os objetivos requeridos para um *cluster* de computadores.

Como trabalhos futuros sugerimos o estudo para a utilização do algoritmo com superfícies tridimensionais. Na área da Ciência da Computação, pode ser feita a aplicação e avaliação de técnicas de tratamento de *loops* (fusão e inversão, por exemplo), procurando aproveitar melhor as características da linguagem FORTRAN. Outro aspecto que pode ser melhorado é o algoritmo de tolerância a falhas, associado aos recursos existentes no próprio ambiente de execução, como o *checkpoint*.

Referências

ALMASI, G. S.; GOTTLIEB, A. **Highly Parallel Computing.** 2.a Edição: The Benjamin Cummings Publishing Company, Inc, 1994, ISBN: 0-8053-0443-6.

AMAR, J. G. The Monte Carlo Method in Science and Engineering. In: *Computing in Science and Engineering.* Vol. 8, n. 2, pp.9-19, IEEE Educational Activities Department, 2006, ISSN: 1521-9615

AMORIM C. L.; BARBOSA V. C.; FERNANDES E. S. T. **Uma Introdução à Computação Paralela e Distribuída.** 1.a Edição. Campinas, SP: Unicamp/IMECC, 1988.

BAER T.; WYCKOFF P. A parallel I/O mechanism for distributed systems. In: *IEEE International Conference on Cluster Computing*, pp. 63-69, Sep 20-23, 2004.

BOUKERCHE, A. *et al*. Performance Evaluation of a Local DNA Sequence Alignment Algorithm on a Cluster of Workstations. In: *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, vol. 7, n. 7, p. 164a, 2004.

CABRAL, V. F.; CASTIER M.; TAVARES F. W. Monte Carlo simulations of adsorption using 2-D models of heterogeneous surfaces. In: *AIChE Journal*, Vol. 49, n. 3., pp. 753-763, 2003a

CABRAL, V. F. *et al*. Monte Carlo simulations of the adsorption of chainlike molecules on two-dimensional heterogeneous surfaces. In: *Langmuir*, Vols. 19, n. 4. - pp. 1429-1438, 2003b.

CARPENTER JR., R. L.; BASSETT G. M. Commercial Application of the Advanced Regional Prediction System (ARPS). In: *14th Conference on Numerical Weather Prediction*, Fort Lauderdale, FL, USA, 2001.

CARVALHO, E. C. A. **Particionamento de Grafos de Aplicações e Mapeamento em Grafos de Arquiteturas Heterogêneas**. 2002. 146f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal do Rio Grande do Sul, Porto Alegre, 2002.

CENPES. Disponível em <http://www2.petrobras.com.br/tecnologia2/port/centro_pesquisas dapetrobrasapresentacao.asp>. Acesso em 20/01/2009.

CHEN, C.; SCHMIDT, B. Computing Large-Scale Alignments on a Multi-Cluster. In: *Fifth IEEE International Conference on Cluster Computing (CLUSTER'03)*, p. 38, IEEE, 2003

CHIOLA G. Some research projects on clusters of personal computers. In: *Proceedings of 24th Euromicro Conference*. Vol. 2. - pp. 47-49, 1998, ISBN: 0-8186-8646-4-2.

DONGARRA, J. J.; DUNIGAN, T. Message-Passing Performance of Various Computers. *Technical Report*: UT-CS-95-299. Knoxville, TN, USA, University of Tennessee, 1995.

DONGARRA JACK *et al*. **Sourcebook of parallel computing**. San Francisco, CA: Morgan Kaufmann Publishers Inc., p. 789, 2003, ISBN: 1-55860-871-0.
DUNCAN, R. A Survey of Parallel Computer Architectures. In: *Computer*. IEEE Computer Society Press,.Vol. 23, n. 2, pp. 5-16, 1990, ISSN: 0018-9162.

EGGERS, S. J. *et al*. Simultaneous Multithreading: A Platform for Next-generation Processors. In: *IEEE Micro.* IEEE, Vols. 17, n. 5, pp. 12-19, 1997.

EVANS, D. J. V.; GOSCINSKI, A. M. A Survey of Basic Issues of Parallel Execution on a Distributed System. *Technical Report*: C95-03. School of Computing and Mathematics, Deakin Universtiy, Victoria, 1995.

FISCHER, C. ESTRADE, J. F.; JERMAN, J. **Implementation of the Limited-Area Numerical Weather Prediction Model Aladin in Distributed Memory.** London: Springer-Verlag, pp. 1411-1416, 1999, ISBN: 3-540-66443-2.

FOSTER, I. **Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.** Mathematics and Computer Science Division - Argonne National Laboratory. 1995. Disponível em <http://www-unix.mcs.anl.gov/dbpp/text/node4.html>. Acesso em 13 de janeiro 2009.

G95. Disponível em <http://sourceforge.net/projects/g95/>. Acesso em 13/01/2009.

GORINO, F. V. R. **Balanceamento de Carga em Clusters de Alto Desempenho:** Uma extensão para a LAM/MPI. 2006. 94f. Dissertação (Mestrado) – Programa de Pós-Graduação em Ciências da Computação. Universidade Estadual de Maringá, Maringá, 2006.

GROOVE, D. A.; CODDINGTON, P. D. Communication Benchmarking and Performance Modelling of MPI programs on cluster computers. In: *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, p. 249, IEEE, 2004.

HAYES A. *et al*. The role of computational clusters. In: *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*. IEEE, Dec 1992.

HWANG, K.; BRIGGS, F. A. **Computer Architecture and Parallel Processing**. 1.a Edição, New York, NY: McGraw-Hill, Inc., 1990, ISBN: 0070315566.

KAMAL, H.; PENOFF B.; WAGNER, A. SCTP versus TCP for MPI. In: *Proceedings of the ACM/IEEE SC 2005 Conference,* p. 30, 2005.

KARNIADAKIS, G. E.; KIRBY, R. M. **Parallel Scientific Computing in C++ and MPI**: A seamless approach to parallel algorithms and their implementation. Cambridge: Cambridge University Press, p. 650, 2003, ISBN: 9780521520805.

KIRNER, C. Arquiteturas de sistemas avançados de computação. JORNADA EPUSP/IEEE EM SISTEMAS DE COMPUTAÇÃO DE ALTO DESEMPENHO**. Anais...,** 1991, pp. 307-353.

KORNISS, G.; NOVOTNY, M. A.; RIKVOLD, P. A. Parallelization of a Dynamic Monte Carlo Algorithm: a Partially Rejection-Free Conservative Approach. In: *Journal of Computational Physics*: Academic Press Professional, Inc., Vol. 153, n. 2, pp. 488-508, 1999, ISSN: 0021-9991.

LEE, J. *et al*. RFS: efficient and flexible remote file access for MPI-IO. In: *IEEE International Conference on Cluster Computing*, pp. 71-81, 2004.
LONG, L. N.; MODI, A. Turbulent Flow and Aeroacoustic Simulations using a Cluster of Workstations. In: *NCSA Linux Revolution Conference*, Illinois, 2001.

LUKE, E. A. Defining and measuring scalability. In: *Proceedings of the Scalable Parallel Libraries Conference*. Mississippi State, MS, USA, pp. 183-186, 1993, ISBN: 0-8186-4980-1.

LUKSCH, P. Software Engineering Methods for Designing Parallel and Distributed Applications from Sequential Programs in Scientific Computing.In: *Proceedings of HICSS-30, Minitrack on Software Engineering for Distributed Systems*. IEEE, 1997.

MARTINS JR., A. S. **Checkpoint automático de aplicações distribuídas em cluster MPI**. 2005. 103f. Dissertação (Mestrado) – Programa de Pós-Graduação em Ciências da Computação. Universidade Estadual de Maringá, Maringá, 2005.

MCCANDLESS, B. C.; SQUYRES, J. M.; LUMSDAINE, A. Object Oriented MPI (OOMPI): a class library for the Message Passing Interface In: *Proceedings of Second MPI Developer's Conference*. Notre Dame, IN, USA, pp. 87-94, 1996, ISBN: 0-8186-7533-0.

MORETTI C. O. *et al*. Algoritmos Automáticos de Partição de Domínio. Escola Politécnica da Universidade de São Paulo - Boletim Técnico, BT/PEF-9803, 1998, ISSN: 0103-9822.

NAVAUX, P. O. A. Introdução Ao Processamento Paralelo. In: *Revista Brasileira de Computação*, Vol. 5, n. 2, pp. 31-44, 1989.

NUPAIROJ, N.; NI, L. M. Performance Evaluation of Some MPI Implementations on Workstation Clusters. In: *Proceedings of the Scalable Parallel Libraries Conference*: IEEE, pp. 98–105, Oct, 12-14, 1994.

OJIMA, Y. *et al*. Design of a Software Distributed Shared Memory System using an MPI communication layer. In: *ISPAN '05: Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks*. Washington, DC, USA: IEEE Computer Society, pp. 220-229, 2005, ISBN: 0-7695-2509-1.

OSCAR. Disponível em < http://oscar.sourceforge.net>. Acesso em 13/01/2009.

PACHECO, P. S. **Parallel Programming with MPI**, Morgan Kaufmann Publishers, CA, 1997

PACS Training Group. **Introduction to MPI**: University of Illinois, 2001.

REHM, W; WANG, B; BINNINGER, B.; NAE, C. Modern Field Code Cluster of Fluid Dynamics and Structure Mechanisms with Meta-Computing Grids for Safety and Environment Research Studies. In: *4th. European LS-DYNA Users Conference*, 2003.

ROSENTHAL, J. S. Parallel computing and Monte Carlo algorithms. In: *Far East Journal of Theoretical Statistics*, Vol. 4, pp. 207-236, 2000.

SETI. Disponível em <http://setiathome.berkeley.edu/sah_about.php>. Acesso em 13/01/2009.

SNIR, M. *et al*. **MPI**: The Complete Reference . Cambridge: The MIT Press, 1996.

TEHRANIAN, S. *et al.* A robust framework for real-time distributed processing of satellite data. In: *Journal of Parallel and Distributed Computing*. Vols. 66, n. 3. , pp. 403-418, 2006. ISSN:0743-7315.

TOP500 *Supercomputer site.* Disponível em <http://www.top500.org>. Acesso em 19/01/2009.

WEIQIN, T.; HUA, Y.; WENSHENG, Y. PJMPI: pure Java implementation of MPI. In: *Proceedings of The Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*. Beijing, China, Vol. 1, pp. 533-535, 2000, ISBN: 0-7695-0589-2.

ZALUSKA, E. J. Research Lines in Distributed Computing. In: *Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software*. São Carlos: ICMC/USP, pp. 132-155, 1991.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo