FUNDAÇÃO EURÍPIDES SOARES DA ROCHA
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MARCELO AUGUSTO OMOTO

CONFIGURAÇÃO DE UM CLUSTER PARA O ALGORITMO SHELLSORT DISTRIBUÍDO

MARÍLIA
2009

MARCELO AUGUSTO OMOTO

CONFIGURAÇÃO DE UM CLUSTER PARA O ALGORITMO SHELLSORT DISTRIBUÍDO

Trabalho de curso apresentado ao curso de Bacharelado em Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Ms. MAURÍCIO DUARTE

MARÍLIA
2009

Omoto, Marcelo Augusto
Configuração de um Cluster para o Algoritmo ShellSort Distribuído / Marcelo Augusto Omoto; Orientador: Maurício Duarte. Marília, SP: [s.n.], 2009.
67 f.

Trabalho de Curso (Graduação em Bacharelado em Ciência da Computação) - Curso de Ciência da Computação, Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília - UNIVEM, Marília, 2009.

1. Cluster 2. Arquitetura de Alto Desempenho 3. MPI

CDD 004.2

MARCELO AUGUSTO OMOTO

CONFIGURAÇÃO DE UM CLUSTER PARA O ALGORITMO SHELLSORT DISTRIBUÍDO

Banca Examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do grau de Bacharel em Ciência da Computação.

Resultado:

ORIENTADOR: _____
Prof. Ms. Maurício Duarte

1º EXAMINADOR: _____
Prof. Ms. Fábio Dacêncio

2º EXAMINADOR: _____
Prof. Ms. Leonardo Botega

Marília, ___ de _____ de 2009.

Dedico o presente trabalho...

À forma inteligente, a qual permitiu a existência da humanidade, inclusive a minha;

À meus pais, que me deram condições de chegar até aqui

AGRADECIMENTOS

Ao Prof. Maurício Duarte, pela excelente orientação à meu TCC, sua compreensão e ajuda, sem as quais, não terminaria meu trabalho.

Ao Prof. Marconato e à mestranda Priscila Saito, pela ajuda na configuração do cluster, que foi fundamental para a finalização do projeto.

A meus pais, que me originaram e me deram condições de tomar meu caminho, sempre ajudando no que precisei.

A meus colegas de turma, com os quais, nunca tive problemas de relacionamento e, em especial ao Mário, que sempre se esforçou ao máximo nos trabalhos que fazíamos em grupo, ao Lucão e ao Diego, companheiros de bar.

A meus amigos, que sempre estiveram comigo, em toda situação, em especial ao Lucas Faléco, Jorge Rampim, Cássio Xavier, Marcela Parra, João Rampim, Marcelo Balieiro, Sílvio Quintino, Luiz Henrique Lessa, Mauro Duarte, Alex Teren e Rafael Santos.

Aos professores e demais funcionários da UNIVEM

E aos meus colegas de trabalho do CDI.

“Para que levar a vida tão à sério, se a vida é uma alucinante aventura da qual jamais sairemos vivos”

“Vocês riem de mim por eu ser diferente e eu rio de vocês por serem todos iguais”

Bob Marley

OMOTO, Marcelo Augusto. **Configuração de um cluster para o algoritmo ShellSort distribuído**. 2009. 67 f. Trabalho de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2009.

RESUMO

A constante evolução da tecnologia que envolve os computadores e a crescente demanda de poder computacional para questões especialmente científicas, ocasionou o surgimento de novas arquiteturas, estas objetivando além do alto desempenho, um baixo custo. Este estudo tem como objetivo a configuração de uma arquitetura de alto desempenho baseada em cluster, apoiada pelo sistema operacional livre Linux e pelo ambiente de passagem de mensagens denominado MPI, que executa aplicações estritamente distribuídas. Trata-se de uma pesquisa exploratória, descritiva e prática, com revisão bibliográfica e a construção de uma arquitetura de alto desempenho baseada em computadores obsoletos. Um algoritmo paralelo será desenvolvido para validar o projeto. Demonstra-se na finalização do projeto, que a arquitetura de alto desempenho baseada em cluster, é uma opção extremamente viável para se obter um sistema de processamento de elevado desempenho a um baixo custo.

Palavras-chave: Cluster, Arquitetura de Alto Desempenho, MPI, Linux.

OMOTO, Marcelo Augusto. **Configuração de um cluster para o algoritmo ShellSort distribuído**. 2009. 67 f. Trabalho de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2009.

ABSTRACT

The constant evolution of technology involving computers and the growing demand of computational power for particular scientific issues, caused the arising of new architectures, which aimed a high performance and a low cost. This study aims to setting up an architecture based on high performance cluster, supported by the free operating system called Linux and by the environment for exchange messages called MPI, to run applications strictly distributed. This is an exploratory, descriptive and practical research, having a literature review and the construction of an architecture based on high performance obsolete computers. A parallel algorithm will be developed to validate the project. It will be expected at the end of the project, the demonstration that the architecture based on high performance cluster is a very viable option to obtain a system of processing of high performance at a low cost.

Keywords: Cluster, High Performance Architecture, MPI, Linux.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

EP: Elemento de Processamento

HA: High Availability

HPC: High Performance Computing

HS: Horizontal Scaling

MIMD: Multiple Instruction Multiple Data

MISD: Multiple Instruction Multiple Data

MMU: Memory Management Unit

MPI: Message Passing Interface

NUMA: Non-Uniform Memory Architecture

ORNL: Oak Ridge National Laboratory

PVM: Parallel Virtual Machine

SIMD: Single Instruction Multiple Data

SISD: Single Instruction Single Data

SMP: Simetric Multiprocessors

LISTA DE GRÁFICOS

SUMÁRIO

INTRODUÇÃO

A incessante demanda por poder computacional para questões estritamente científicas culminou com o desenvolvimento de arquiteturas com alto poder de processamento, estas, apesar de solucionarem parte do problema em questão, possuem um custo muito elevado, fazendo-se necessária a busca por novas soluções (TANENBAUM, 2002).

Surgiram então, as arquiteturas de alto desempenho baseadas em *cluster* de computadores, que além de seu alto poder de desempenho, possuem um custo muito baixo, haja vista ter base em software livre e poder ser construído a partir de computadores obsoletos (MORIMOTO, 2003).

A necessidade de alto desempenho e alto poder computacional para o processamento do *cluster* de computadores pode ser resolvida por meio do uso de sistemas distribuídos e de bibliotecas de passagem de mensagens, que viabilizam a computação paralela sobre sistemas distribuídos (computação paralela e distribuída).

A arquitetura denominada *cluster*, utiliza ambientes de passagem de mensagens para que a comunicação ocorra entre os computadores via rede. No presente trabalho será utilizado o ambiente *MPI* (*Message Passing Interface*) (SNIR et al., 1996). O *MPI* foi proposto para ser um padrão internacional de biblioteca de passagem de mensagens e tem tido destaque na literatura, não só pela flexibilidade, mas também pelo fato de constituir um tipo de solução para o problema da portabilidade de programas paralelos entre sistemas diferentes. Além disso, possibilita eficiência e segurança em qualquer plataforma paralela (CÁCERES, 2001). Existem muitas implementações de *MPI* em aplicações desenvolvidas em linguagens como Fortran, C e C++.

Objetivos

Este projeto visa a montagem e configuração de um cluster baseado em computadores comuns, auxiliado pelo ambiente de passagem de mensagens *MPI*, que posteriormente será validado com testes a partir de um algoritmo intrinsecamente paralelo. Um *cluster* pode ser empregado com sucesso em tarefas que exigem alto desempenho, como previsões meteorológicas, processamento de imagens médicas e até cálculos astronômicos.

Organização da monografia

Essa monografia está organizada da seguinte forma:

Capítulo 1 – Computação Paralela e Distribuída: apresenta a computação paralela e distribuída, destacando-se as arquiteturas utilizadas para a execução de aplicações paralelas. Conceitos essenciais referentes à programação paralela e sistemas distribuídos também são discutidos.

Capítulo 2 – Arquiteturas de Alto Desempenho: apresenta as principais arquiteturas de alto desempenho, dando ênfase ao *cluster*, arquitetura utilizada no presente trabalho. Conceitos primordiais para diferenciar cada arquitetura estão destacados neste capítulo.

Capítulo 3 – Ambientes de Passagem de Mensagens: aborda os ambientes de passagem de mensagens como suporte à computação paralela distribuída. Esse capítulo apresenta também as características e a descrição de algumas funcionalidades do ambiente de passagem de mensagens utilizado nesse trabalho.

Capítulo 4 – Metodologia e Projeto: apresenta as tecnologias utilizadas e a instalação e configuração do ambiente de passagem de mensagens *MPI*, que permite a comunicação do cluster. Além disso, é apresentada a proposta de algoritmo paralelo que servirá para validar o projeto.

Capítulo 5 - Análise de Desempenho e Conclusões: enfatiza as análises de desempenho que validam o projeto (testes seqüenciais e paralelos) e apresenta algumas conclusões finais e sugestões para atividades consideradas relevantes para a continuidade do projeto.

CAPÍTULO 1 – COMPUTAÇÃO PARALELA E DISTRIBUÍDA

Desde o prelúdio do surgimento dos computadores, quando estes pertenciam apenas a grandes corporações, buscou-se a redução de custos e a melhora do desempenho computacional e, com o crescente avanço tecnológico, diversos paradigmas importantes surgiram como grande solução para resolver este desafio.

Com o intuito de aumentar a potência computacional de um computador, chegou-se a utilizar máquinas multiprocessadoras, que caracterizaram a computação paralela. Embora estas possuíssem um perceptível desempenho superior aos computadores da época, eram muito dispendiosos e não permitiam uma ampliação do poder computacional, uma vez que eram limitados à quantidade de processadores que possuíam.

Posteriormente, surgiram as redes locais de alta velocidade que aliadas ao desenvolvimento dos microprocessadores, deram origem aos sistemas distribuídos (TANENBAUM, 2006).

1.1 Computação Paralela

A computação paralela, segundo Almasi e Gottllieb (1994), constitui-se de uma coleção de elementos de processamento (EPs) que se comunicam e cooperam entre si para resolver problemas de grande dimensão mais rapidamente; uma solução inovadora para se obter maior eficiência, se comparada às arquiteturas seqüenciais.

Assim, este paralelismo consiste na cooperação de elementos de processamento que através de comunicação e sincronismo, fragmentam a aplicação e a dividem entre si, o que torna o encargo muito mais rápido.

O processamento paralelo objetivou, além de um melhor desempenho (ALMASI, 1994), uma melhor relação custo benefício, quando comparado aos supercomputadores, a modularidade e a tolerância a falhas, obtida pela replicação de elementos de processamento.

Todavia, um computador paralelo pode não ter utilidade sem um programa paralelo e, a sincronização e divisão da aplicação em tarefas atribuídas aos elementos de processamento são incumbências muito trabalhosas. As tarefas manifestam a rotina de quando dois ou mais processos iniciados disputam a utilização do processador, descrevendo a concorrência, que ao contrário do paralelismo, ocorre tanto em sistemas paralelos, quanto uniprocessados.

Em sistemas uniprocessados, aflui-se um pseudo-paralelismo, onde o processador executa cada processo em parcelas de tempo muito diminutas, dando a impressão de ocorrerem ao mesmo tempo (ALMASI, 1994), já nos sistemas paralelos, têm-se diversos processadores, o que possibilita a sua execução simultânea.

Na busca de se obter máquinas paralelas, diferentes arquiteturas foram propostas, possuindo diferenças principalmente na forma interligá-las.

E, devido a essa diversidade de arquiteturas, diversas classificações foram propostas para classificá-las, não obstante, nenhuma com grande aceitação. Dentre eles, as mais conhecidas são a taxonomia de Flynn (1972) e a de Duncan (DUNCAN, 1990).

A taxonomia de Flynn baseia-se em fluxos de instruções e fluxos de dados. Um fluxo de instruções é uma seqüência de instruções executadas em que cada fluxo corresponde a um contador de programa (TANENBAUM, 1999) e um fluxo de dados representa um conjunto de operandos manipulados por um fluxo de instruções (TANENBAUM, 1999). Existem então, quatro combinações possíveis de arquiteturas: SISD (*Single Instruction / Single Data*), SIMD (*Single Instruction / Multiple Data*), MISD (*Multiple Instruction / Multiple Data*) e MIMD (*Multiple Instruction / Multiple Data*).

Single Instruction, Single Data (SISD) em que um único fluxo de instrução opera em único fluxo de dados. São exemplos dessa arquitetura, os computadores baseados no modelo de Von Neumann.

Single Instruction, Multiple Data (SIMD) em que um único fluxo de instrução é executado por múltiplos elementos de processamento sobre conjuntos de dados distintos. Processadores vetoriais e matriciais são representantes dessa arquitetura.

Multiple Instruction, Single Data (MISD) em que múltiplos fluxos de instruções operam sobre um mesmo fluxo de dados. Existe uma divergência entre os autores sobre a existência de arquiteturas que possam ser classificadas como MISD.

Multiple Instruction, Multiple Data (MIMD) em que diferentes seqüências de instruções sobre diferentes conjuntos de dados são executadas simultaneamente por um conjunto de processadores. Grande parte dos computadores paralelos representam esta categoria. Esta categoria pode ser dividida em multiprocessadores e multicomputadores (STALLINGS, 2003).

Nos multiprocessadores, uma memória compartilhada é utilizada pelos processadores, para, haver alguma comunicação entre os processos de cada um, fato que caracteriza o multiprocessamento. Nos multicomputadores, uma memória distribuída é

utilizada, onde os computadores utilizam uma biblioteca de passagem de mensagens para se comunicarem.

Apesar disso, nos sistemas multiprocessados, caso os processadores compartilhem a mesma memória, ele é denominado fortemente acoplado (*tightly coupled*) e, caso os processadores possuam a sua própria memória privada, eles são denominados fracamente acoplados (*loosely coupled*).

A taxonomia de Flynn, sem dúvida, foi muito importante, mas não suficiente para classificar os novos computadores de forma adequada. Assim, Duncan propôs uma taxonomia dividida em arquiteturas síncronas e assíncronas (DUNCAN, 1990).

Nas arquiteturas síncronas, existe um relógio global, que manipula todas as suas operações. Esse grupo é formado pelas arquiteturas SIMD da taxonomia de Flynn, pelas arquiteturas vetoriais e sistólicas.

Nas arquiteturas assíncronas, os processadores não possuem um relógio global, ou seja, operam de forma autônoma. Esse grupo é composto, basicamente, pelas arquiteturas MIMD da taxonomia de Flynn, sejam elas convencionais (MIMD com memória distribuída ou compartilhada) ou não-convencionais (MIMD/SIMD, Fluxo de Dados, Redução ou Dirigidas à demanda e Frente de Onda) (DUNCAN, 1990).

1.2 Sistemas Distribuídos

Segundo Tanenbaum (2003), um sistema computacional distribuído é uma coleção de computadores autônomos, interligados por uma rede de comunicação e equipados com um sistema operacional distribuído que permitem compartilhamento transparente de recursos existentes no sistema.

Assim, pode-se dizer que os sistemas distribuídos permitem um agrupamento do poder computacional de diversos elementos de processamento interligados por uma rede de comunicação, de forma transparente, como se houvesse apenas um único e centralizado computador executando a tarefa.

Os sistemas distribuídos tornaram-se então, uma alternativa eficiente e de baixo custo, em relação aos sistemas centralizados. A utilização de sistemas distribuídos apresenta inúmeras vantagens em relação a um sistema centralizado, dentre as quais destacam-se:

- econômica: aproveita máquinas potencialmente ociosas e, um supercomputador é muito mais custoso que a interligação de vários processadores;
- potência de processamento: um sistema distribuído pode ter uma potência computacional maior do que um supercomputador;
- distribuição inerente: algumas aplicações são distribuídas por natureza, sendo assim, mais facilmente implementadas nesse tipo de ambiente;
- confiabilidade: em caso de falha de uma máquina, o sistema como um todo pode continuar executando a aplicação, apresentando apenas uma degradação de desempenho;
- crescimento incremental: o poder computacional pode ser aumentado com a inclusão de novos equipamentos.

Por outro lado, os sistemas distribuídos apresentam também algumas desvantagens relacionadas com:

- software: escassez de software para sistemas distribuídos;
- segurança: problemas no controle de acesso à um grande contingente de usuários;
- rede de intercomunicação: a rede pode saturar.

Segundo Tanenbaum (2006), os sistemas distribuídos devem apresentar em aspectos de projeto:

- transparência: um sistema distribuído é caracterizado transparente, quando este é visto como um centralizado e singular sistema, dissimulando a existência de componentes independentes e fisicamente separados;
- flexibilidade: possuir a facilidade de fazer reconfigurações;
- tratamento de falhas: deve exercer mecanismos de tratamentos de falhas, pois, quando um sistema falha, os demais componentes continuam ativos, podendo originar inconsistências;
- performance: é a avaliação do paralelismo e da comunicação para obter o nível de granularidade;

- escalabilidade: característica que prima a capacidade do sistema de expandir-se com o mínimo de degradação de desempenho;
- concorrência: devido à existência de múltiplos usuários e do compartilhamento de recursos, existe a necessidade da sincronização de processos, para que a consistência do sistema seja mantida;
- abertura: um sistema aberto, deve permitir a comunicação com outro sistema aberto por meio do emprego de regras padronizadas quanto a formato, conteúdo e significado das mensagens enviadas e recebidas (TANENBAUM, 2006). Em um sistema aberto, novos componentes de hardware e software podem ser adicionados, independentemente de fabricantes individuais.

Os sistemas distribuídos podem ser homogêneos ou heterogêneos. Sistemas homogêneos são formados por multicomputadores interligados por uma rede, onde todos os elementos de processamento possuem as mesmas características, já os sistemas heterogêneos possuem uma grande diversidade de características.

1.3 Computação Paralela/Distribuída

A recente e vasta utilização de microcomputadores monoprocessados, aliado à evolução das tecnologias de rede de computadores de alto desempenho (sistemas distribuídos), motivou o surgimento da computação paralela/distribuída.

A computação paralela/distribuída constitui a convergência de duas áreas surgidas com motivações e características distintas, a computação paralela e os sistemas distribuídos, trazendo benefícios para ambas (ZOMAIA, 1996).

Algumas características dos sistemas distribuídos são muito úteis à computação paralela, tais como a transparência de acesso aos recursos, confiabilidade e tolerância à falhas e, a evolução na tecnologia de redes mostrou que os sistemas distribuídos possuem uma infra-estrutura propícia à computação paralela.

Assim, com o intuito de se obter um maior proveito das características de cada um, sua utilização foi vista como uma alternativa à utilização de supercomputadores dispendiosos e uma forma de solucionar problemas com um retorno mais rápido e barato do que se utilizasse uma supermáquina seqüencial. Um sistema paralelo/distribuído é empregado com o uso de bibliotecas de passagem de mensagem.

Apesar de tantas vantagens, algumas técnicas e mecanismos dos sistemas paralelos não são adequados para um sistema paralelo/distribuído e, alguns fatores característicos dos sistemas distribuídos tais como a heterogeneidade e atraso na rede, tornam o gerenciamento desses sistemas, uma tarefa árdua.

1.4 Considerações Finais

A evolução tecnológica atual, caracterizada pela grande conectividade dos recursos, foi sem dúvida a principal responsável pelo uso dos sistemas distribuídos. Fato que se mostrou uma excelente alternativa ao uso de máquinas seqüenciais, que possuíam uma velocidade muito limitada.

Visando um poder maior de processamento, para realizar tarefas mais complexas, surgiu a computação paralela, tendo como finalidade, além de uma maior potência computacional, uma forma mais barata e aprimorada para realizar tarefas, principalmente as essencialmente paralelas.

A convergência, apresentada entre a computação paralela e os sistemas distribuídos, trouxe inúmeras vantagens, principalmente para a primeira, como a redução de custos, o aumento de desempenho, a utilização mais adequada de recursos evitando ociosidade no sistema, entre outras. Tudo isso tem feito com que cada vez mais apareçam trabalhos em que a computação paralela distribuída pode ser aplicada para solucionar uma variedade de aplicações que necessitam de alta potência computacional. O projeto a ser apresentado neste trabalho remete-se a uma arquitetura de alto desempenho denominada *cluster*, um protótipo de sistema distribuído que concentra todo o poder computacional em uma única tarefa.

CAPÍTULO 2 – ARQUITETURAS DE ALTO DESEMPENHO

A evolução tecnológica e científica acerca da computação paralela e dos sistemas distribuídos e a constante demanda de poder computacional, motivaram a idealização e posteriormente, o surgimento de novos tipos de arquiteturas, estas objetivando, atingir um alto desempenho a um baixo custo.

As arquiteturas de alto desempenho têm como principal finalidade, uma grande quantidade de poder computacional, e este contexto soluciona o problema de muitas questões estritamente científicas, tais como previsões meteorológicas, processamento de imagens médicas e até cálculos astronômicos, cujas tarefas demandam sua enorme capacidade computacional (TANENBAUM, 1999).

Seguindo este paradigma, concebem-se exclusivamente quatro arquiteturas: multiprocessadores simétricos (SMP), arquitetura de acesso não uniforme à memória (NUMA), *grid* e *cluster*, que será a arquitetura utilizada no projeto.

2.1 Multiprocessadores Simétricos (SMP)

A arquitetura baseada em multiprocessadores simétricos consiste de dois ou mais processadores similares com capacidades de processamento comparáveis, conectados entre si e à memória por um barramento ou outra forma similar de circuito de conexão interno.

Neste tipo de arquitetura, todos os processadores compartilham uma mesma memória e tem acesso aos mesmos dispositivos de entrada e saída, existindo um único sistema operacional, que torna transparente ao usuário, a existência de diversos processadores. O tempo de acesso à memória é o mesmo para todos os processadores.

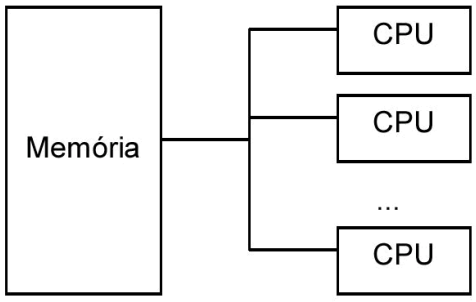

Figura 1 - SMP

Em geral, os processadores encontram-se ligados à memória por um barramento de tempo compartilhado, *crossbar* ou redes multiestágio.

2.1.1 Barramento de tempo compartilhado

No caso de se utilizar um barramento de tempo compartilhado, as estruturas e interfaces são praticamente as mesmas de um uniprocessador, existe um mecanismo de tempo compartilhado e este é limitado a 2 ou 3 processadores, justamente pelo fato de o barramento tornar-se um gargalo. Todavia, a utilização de uma memória *cache*, melhora o tráfego no barramento e possibilita a utilização de mais processadores, não obstante, fica propenso ao problema da coerência de *cachê* (PITANGA, 2002).

O problema da coerência de *cache* acontece, pois podem existir várias cópias de um mesmo dado nas *caches* de diferentes processadores, assim, a alteração em sua *cache* por um dos processadores pode tornar o conteúdo das outras *caches* incoerentes.

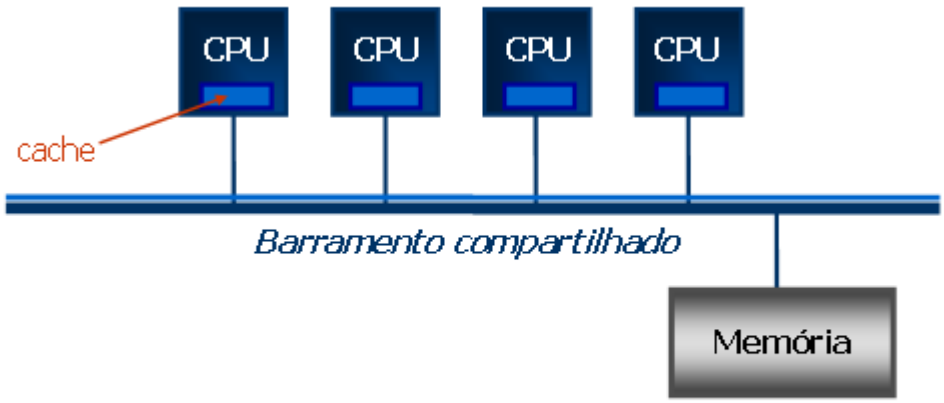

Figura 2 – SMP com Barramento de Tempo Compartilhado

A solução definitiva para a incoerência é feita através de protocolos de diretório e protocolos de monitoração. Na primeira abordagem, é empregado um diretório e um controlador central, que apesar de resolver o problema da coerência do *cache*, causa uma grande sobrecarga no sistema, pois a comunicação entre processadores e controlador central ocorre freqüentemente. Na segunda abordagem, todos os processadores ficam monitorando ativamente o tráfego no barramento e atualizam o estado em que se encontra um determinado

dado em sua *cache*, a coerência é mantida por escrita com atualização ou escrita com invalidação, no qual se destaca o protocolo MESI.

2.1.2 *Crossbar*

No esquema denominado *crossbar*, os processadores acessam a memória em paralelo e são interligados aos módulos de memória por meio de *crosspoints*. *Crosspoints* são *switches* eletrônicos utilizadas para conectar processador à memória (BARAK, 2008).

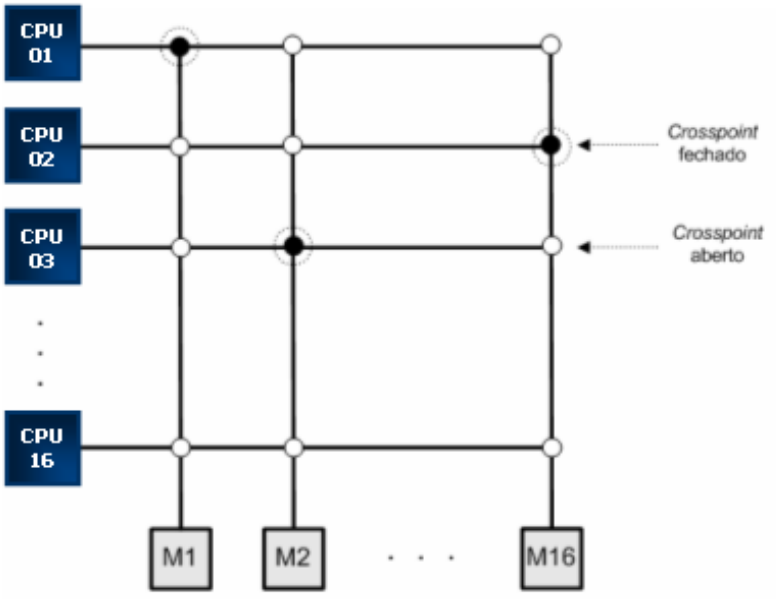

Figura 3 - Crossbar

Embora seja possível a utilização de um grande número de processadores neste esquema, isto não acontece na prática, pois seu elevadíssimo custo inviabiliza seu projeto.

2.1.3 Redes multiestágio

As redes multiestágio são desenvolvidas a partir de *switches* com determinado número de entradas e saídas, assim, diferentes classes destas redes podem ser construídos. O que varia de uma classe para outra é o tipo de *switch* utilizado e o padrão de conexão entre os estágios.

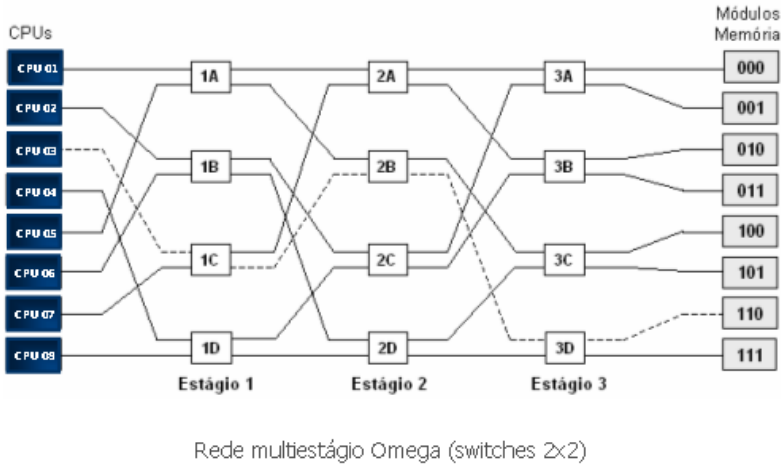

Figura 4 – Redes Multiestágio

A ilustração acima mostra *switches* 2x2 e um padrão de conexão determinado *perfect shuffle* (ALMASI; GOTTLIEB, 1994; TANENBAUM, 1999).

A principal desvantagem de uma rede multiestágio em relação à *crossbar* é que nem sempre dois acessos podem acontecer em paralelo, mesmo que sejam para módulos diferentes.

2.2 NUMA

Define-se por NUMA, uma arquitetura com dois ou mais processadores, que compartilham uma memória global. Neste tipo de organização, cada nó possui um ou mais processadores com sua própria memória principal e *cache*, conectados por um barramento ou outro meio de interconexão.

A principal característica de uma arquitetura NUMA é o acesso não uniforme à memória, ou seja, embora todos os processadores possam acessar qualquer posição da memória, o endereço acessado influi diretamente nos tempos de acesso. Com isso, tem-se que o acesso a uma posição de memória local é certamente mais rápido que o acesso a uma posição de memória remota.

Dessa forma, o sistema operacional de uma máquina com essa arquitetura deveria, sempre que possível, escalonar as *threads* de um processo entre os processadores do nó da memória usada pelo processo, assim, como o programador também deveria explorar a questão da localidade.

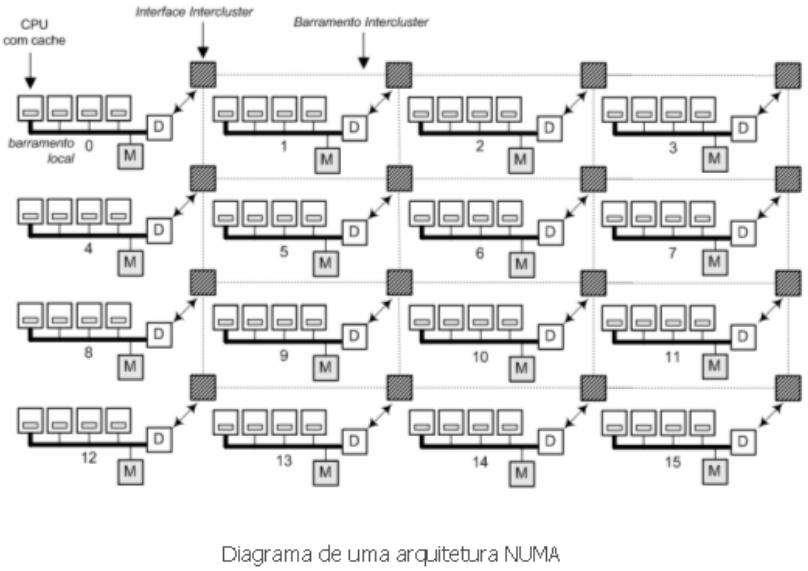

Figura 5 – NUMA

Existem dois tipos de arquitetura NUMA: NC-NUMA, que não utiliza *cache* e CC-NUMA, que utiliza *cache*.

2.2.1 NC-NUMA

Na arquitetura NC-NUMA, a coerência dos dados é garantida pela não existência de *cache*. Não obstante, um dado alocado em local incorreto, ocasiona muitas conseqüências, pois a quantidade de referências seguidas a uma posição de memória remota é igual ao número de buscas através do barramento do sistema. Na maioria das situações, a solução para este problema é implementada em software.

Uma das máquinas mais conhecidas pertencentes a esta categoria foi a Carnegie-Mellon Cm, que era composta por processadores LSI-11, cada uma com sua memória acessível diretamente através de um barramento local e seus nós interconectados por um barramento do sistema.

Nesta máquina, verifica-se que em cada referência à memória, a MMU verifica se o endereço pertence à memória local. Em caso afirmativo, a requisição é feita à memória local, caso contrário, é feita ao nó remoto, através do barramento do sistema.

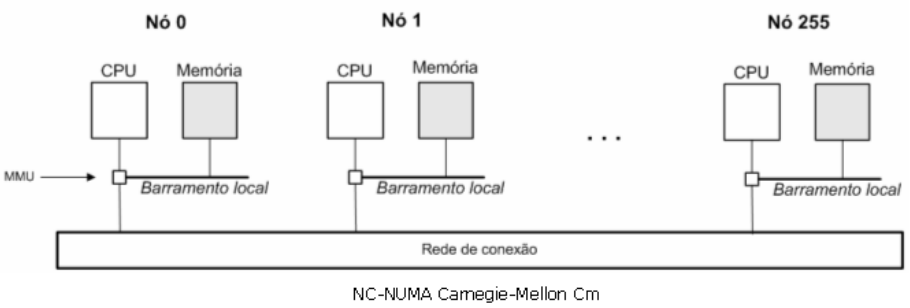

Figura 6 – NC-NUMA

2.2.2 CC-NUMA

A arquitetura CC-NUMA, por utilizar *cache*, requer algum protocolo de coerência. O método mais conhecido para solucionar o problema de coerência de *cache* é baseado no conceito de diretório.

O diretório funciona como uma espécie de banco de dados que indica a localização das várias porções de memória, assim como o status das *caches*.

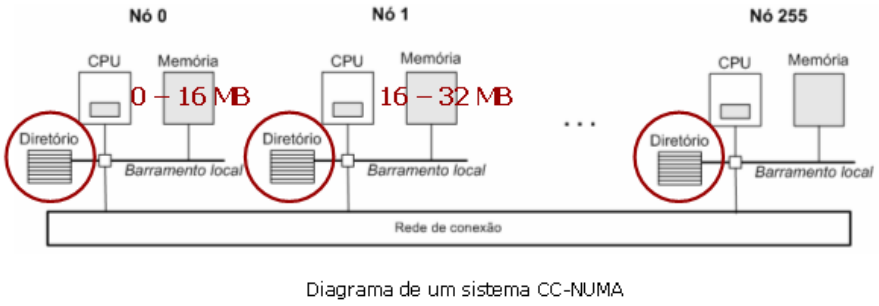

Figura 7 – CC-NUMA

2.3 Grids

Os *grids* computacionais se apresentam como uma das mais novas e inovadoras formas de computação de alto desempenho. Em comparação com a Internet, na qual computadores espacialmente dispersos se comunicam, o *grid* possui a característica de prover o compartilhamento de poder e recursos computacionais, tais como *software* e unidades de armazenamento entre os computadores.

Um *grid* computacional torna-se uma forma interessante e viável de se resolver o problema de insuficiência de poder computacional para cálculos de alta grandeza para computadores e *mainframes* (PITANGA, 2002).

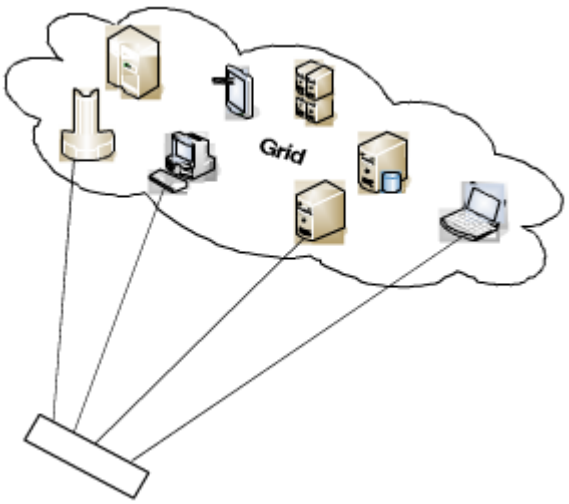

Figura 8 – Grids

As principais características de um *grid* são:

- Heterogeneidade: Os componentes que formam um *grid* tendem a ser extremamente heterogêneos, ou seja, qualquer solução para *grids* deve lidar com recursos de variadas gerações, *softwares* de distintas versões e serviços dos mais variados tipos;
- Alta dispersão geográfica: Um *grid* pode atingir uma escala global, agregando serviços localizados em diversas localizações do planeta;
- Compartilhamento: Um *grid* não pode ser dedicado a uma aplicação de forma exclusiva por um determinado intervalo de tempo;
- Múltiplos domínios administrativos: *Grids* congregam recursos de várias instituições, tornando-se passível a inúmeras políticas de acesso e uso dos serviços, de acordo com as diretrizes de cada domínio que faz parte do *grid*;
- Controle distribuído: Tipicamente não há uma entidade única para controlar o *grid.*

Apesar das características dos *grids* não o torná-lo tão desigual em relação às outras arquiteturas de alto desempenho, estes possuem aspectos e problemas interessantes e intrínsecos, tais como:

- Escalonamento de aplicação: pela incapacidade de se ter um escalonador global para controlar o *grid* por seu tamanho e dispersão, têm-se os escalonadores de recursos e os escalonadores de aplicação, responsáveis por escolher quais recursos utilizar e particionar o trabalho da aplicação, necessitando para isso, de sistemas de monitoramento;
- Acesso e autenticação: pelo aspecto de segurança e a existência de diversos modelos administrativos, utiliza-se o mapeamento de identificação para se ter um *login* único;
- Imagem do sistema: pela existência de diversos modelos administrativos, verifica-se uma imagem heterogênea do sistema, tendo-se como solução, implementar esta em nível de usuário;
- Economia: pela capacidade de se poder configurar manualmente o acesso aos componentes do *grid*, torna-se viável a compra e a venda de recursos computacionais.

2.4 Clusters

Entende-se por *cluster*, um conjunto de computadores (ou nós) autônomos, que podem trabalhar juntos, como um recurso de computação unificado, criando a ilusão de uma única máquina. Os nós são geralmente conectados por interfaces de rede de alto desempenho.

Um *cluster* tem como objetivo fazer com que todo o processamento de uma aplicação seja distribuído aos computadores, de forma com que transpareça um único computador, realizando processamentos que até então apenas *mainframes* e outros computadores de alto desempenho eram capazes de realizar.

“Sendo assim, *clusters* ou suas combinações são usados quando a finalidade é a alta disponibilidade com conteúdos críticos e/ou alto desempenho quando a velocidade de processamento dos serviços deve ser a mais rápida possível” (BUYYA, 1999).

Atualmente, os *clusters* possuem grande emprego em bancos de dados com servidores WEB e, principalmente processamento paralelo.

As principais características de um *cluster* são:

- Configuração dos nós: os nós podem ser tanto uniprocessadores quanto multiprocessadores;
- Escalabilidade absoluta: é possível construir *cluster*s muito grandes, cuja capacidade de computação supera várias vezes a capacidade de uma máquina individual;
- Escalabilidade incremental: um *cluster* pode ser configurado de maneira que seja possível adicionar novos nós, expandindo-o de forma incremental;
- Disponibilidade: tomando-se por base que cada nó é um computador autônomo, uma falha em um nó não implica na perda total do serviço;
- Ótima relação custo/benefício: devido à facilidade de construir um *cluster* a partir de computadores obsoletos ou comercialmente disponíveis, é possível obter um *cluster* com poder de computação igual ou superior à um *mainframe* de grande porte, por um custo muito menor.

Em termos de aplicações, os *clusters* destacam-se pela empregabilidade em:

- Servidores de Internet: o incomensurável crescimento da utilização da Internet fragilizou *sites* muito visitados. Um *cluster* é capaz de distribuir a carga e aumentar a capacidade de resposta;
- Segurança: a grande capacidade que o processamento paralelo é capaz de oferecer beneficia qualquer processo para identificação e quebra na segurança;
- Bases de dados: pesquisas intensivas em banco de dados podem demorar muito tempo em um sistema comum. Um *cluster* pode reduzir drasticamente este período;
- Computação gráfica: o tempo de processamento pode ser muito alto em grandes projetos desta área. Um *cluster* pode reduzir o tempo de geração de imagens significativamente;
- Aerodinâmica: produção de novas capacidades tecnológicas e econômicas na pressão enfrentada em aeronaves, lançamento de naves espaciais e nos estudos de turbulência;

- Análise de elementos finitos: cálculos de barragens, navios, pontes, aviões, grandes edifícios e até veículos espaciais;
- Aplicações em sensoriamento remoto: análise de imagens de satélite para a obtenção de informações sobre a agricultura, fontes hidráulicas e geologia;
- Inteligência artificial e automação: processamento de imagens, reconhecimento de padrões, visão por computador, máquinas de interferência e reconhecimento de voz;
- Engenharia genética: projeto *Genoma*.

2.4.1 Funcionamento de um *cluster*

Os clusters são compostos intrinsecamente de duas partes. A primeira parte consiste em customizações do sistema operacional (como as modificações feitas no Kernel do Linux), compiladores especiais, e aplicações que permitem os programas obterem grande vantagem no cluster. O segundo componente, é o hardware que está interconectado entre os nós do cluster. Estas interconexões, em determinadas ocasiões, são feitas por dispositivos especiais, mas na maioria das implementações de cluster Linux, estas interconexões são feitas através de redes dedicadas, como Fast Ethernet, ou Gigabit Ethernet.

Associações de tarefas, rotinas de atualizações, requisições e dados de programas, podem ser compartilhadas através desta interface de rede, enquanto uma rede separada é empregada para conectar o cluster aos usuários. Às vezes, uma mesma estrutura de rede pode ser usada para ambas as tarefas, porém, isto pode causar uma degradação de desempenho quando a utilização da rede dos usuários for muito alta.

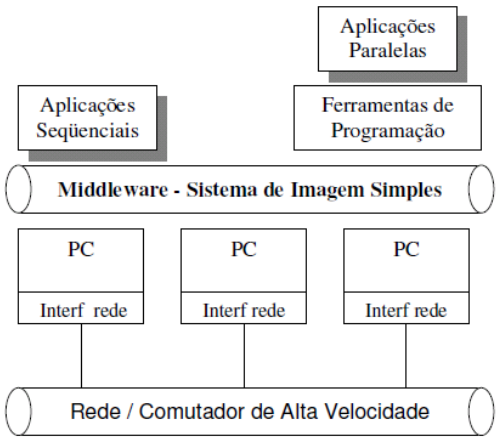

Figura 9 – Exemplo de Cluster

A camada designada de Middleware, situada entre o sistema operacional e as aplicações, é responsável pela criação do conceito de transparência do sistema, também designado “Sistema de Imagem Simples” (SSI - Single System Image), controle de transações e recuperação de falhas (PITANGA, 2002).

2.4.2 Classificação dos *Clusters*

Atualmente, podemos dividir os clusters em três categorias básicas quanto à aplicação final: Alta Disponibilidade (HA – High Availability), Alto Poder de Computação (HPC – High Performance Computing) e Balanceamento de Carga (HS – Horizontal Scaling). O cluster HA tem como objetivo, manter um determinado serviço de forma segura o maior tempo possível. O cluster de HPC é uma configuração designada a prover grande poder computacional, maior do que somente um único computador poderia oferecer em poder de processamento. O cluster de HS tem por finalidade, distribuir de forma equilibrada o tráfego e as requisições das máquinas que estão no sistema.

2.4.2.1 *Cluster* de Alta Disponibilidade

Nos dias de hoje, os computadores são utilizados em todos os ambientes empresariais, comerciais, e até mesmo domésticos e nenhum usuário espera ou deseja a paralização do funcionamento deste. A alta disponibilidade é uma alternativa para garantir a continuidade de operação do sistema em serviços de rede, armazenamento de dados ou processamento, mesmo se houver falhas em um ou mais dispositivos, sejam eles de hardware ou software. Nos clusters de alta disponibilidade, os equipamentos são usados em conjunto para manter um serviço ou equipamento sempre ativo, replicando serviços e servidores. Obviamente, com isso, pode-se ter a perda de desempenho durante o processamento, entretanto, o principal objetivo será alcançado, ou seja, não paralisar o serviço. Segundo Pitanga (2008, p.55): “Essa técnica foi desenvolvida a partir da necessidade de não deixar o sistema parar caso uma máquina, de forma inesperada, deixar de funcionar. Inúmeros são os casos em que o administrador do sistema de organizações é chamado às pressas em um fim de semana, no meio da madrugada ou fora do seu horário de expediente para recolocar o sistema no ar. Piores são as situações em que a organização não encontra seu administrador do sistema e permanece com o problema, podendo gerar prejuízos para ela.”

A disponibilidade dos sistemas torna-se então uma questão vital de sobrevivência empresarial. Um sistema de comércio eletrônico, como venda de livros, por exemplo, não admite indisponibilidade. De maneira geral, um servidor de sólida qualidade apresenta uma disponibilidade de 99,5%, enquanto que uma solução baseada em clusters de computadores apresenta uma disponibilidade de 99,99%.

2.4.2.2 Cluster de balanceamento de carga

O tipo de cluster baseado em balanceamento de carga tem a função de distribuir o tráfego e requisições de máquinas que estão no sistema. É usado para fornecer uma interface simplificada para um conjunto de recursos que podem aumentar ou diminuir com o passar do tempo e de acordo com a necessidade por processamento.

O cluster de Balanceamento de Carga tem como propósito a distribuição igualitária de processos ao longo dos nodos do agrupamento de computadores. Evidentemente algoritmos de escalonamento de processos se fazem necessários (PITANGA, 2002).

Os nós do cluster são interligados para suprir todas as solicitações de recursos e as requisições enviadas de clientes, sendo distribuídas de uma forma rápida e equilibrada entre os nós. Esse tipo de cluster não trabalha ao mesmo tempo em um único processo, fazendo uma espécie de redirecionamento das solicitações uniformemente, definido através de um algoritmo específico para escalonamento.

Para Pitanga (2008, p.68): “Sua grande especialidade é resolver problemas de serviços que tenham inúmeras solicitações em tempo real, como os serviços de comércio eletrônico, empresas que tenham seu sistema on-line, provedores de internet e para solucionar diferentes números de carga que possa ter num exato momento. Para se ter um cluster de característica escalável, é preciso garantir que cada servidor tenha sua utilização de uma forma completa. Portanto, não tendo um serviço e fazendo o balanceamento de carga entre máquinas, que dispõem ter a mesma capacidade de retorno para um usuário, podemos, inicialmente, ter problemas, como mais de uma máquina que possa responder à solicitação feita, em que a comunicação da rede fica mais congestionada e prejudicada.”

Segundo Buyya (1999, p.113), para fazer o balanceamento das requisições dos usuários, este tipo de cluster pode usar diferentes algoritmos de escalonamento, sendo os mais conhecidos:

- Least Connection: que encaminha as requisições para a máquina com o menor número de solicitações;
- Round Robin: que encaminha as solicitações de forma seqüencial;
- Weighted Fair: que encaminha as requisições para a máquina mais robusta ou com maior capacidade de carga.

2.4.2.3 Cluster de alto poder de computação

Esta categoria tem como foco, o desenvolvimento de soluções cujo objetivo principal é o ganho de desempenho através do agrupamento de sistemas de processamento, auxiliado por algoritmos de processamento paralelo. Um cluster HPC pode ser visto como uma solução alternativa para universidades e empresas de pequeno e médio porte, para obterem processamento de alto desempenho na resolução de problemas através de aplicações paralelizáveis, a um custo razoavelmente baixo se comparado com os altos valores necessários para a aquisição de um supercomputador da mesma classe de processamento.

Para Buyya (1999, p.77): "Seu desenvolvimento é uma alternativa aos caros supercomputadores e a facilidade está no custo e na montagem, já a performance está associada ao uso de hardware de alta velocidade (processadores ou periféricos de alta performance), à capacidade de efetuar tarefas de maneira mais eficiente através do uso de algoritmos que resolvam questões computacionais específicas e à condição de usar múltiplos computadores para executarem as sub- tarefas. Quando se fala em processamento de alto desempenho, o qual, neste contexto, pode ser entendido como processamento paralelo e processamento distribuído, imaginam-se muitas pessoas e grandes máquinas dedicadas, que custam milhões de dólares, difíceis de serem operadas e com salas super protegidas. Entretanto, hoje em dia, devido aos clusters, os custos foram reduzidos com pouca perda de desempenho, o que viabiliza o uso de processamento de alto desempenho na solução de problemas em diversas áreas.

O cluster HPC é um sistema paralelo ou distribuído com o objetivo de dividir grandes aplicações em tarefas menores para que possam ser processadas de maneira mais organizada. Os nós trabalham para disponibilizar recursos de maneira simples e integrada, provendo um grande poder computacional.

Aplicações como previsão do tempo, estudo de criptografia, astronomia, renderização, simulações e pesquisas que envolvam cálculos complexos, podem ser aproveitadas num cluster HPC. Nesta categoria máquinas com baixo poder de processamento podem ser aproveitadas, pois ao trabalharem em conjunto, irão conseguir um grande poder de processamento. Para este tipo de cluster, existem duas soluções que são amplamente conhecidas: o Projeto Beowulf e o projeto Mosix.

O Beowulf inaugurou um período de ouro para o Linux na área da supercomputação, justamente por permitir que grande poder computacional seja conseguido com hardware disponível comercialmente. É o que se chama de out of the shelf - direto das prateleiras. O experimento original era formado por máquinas PC 486 com hardware comum, encontrado em qualquer loja de equipamentos de informática.

O projeto Mosix, desenvolvido em Israel, é também especialmente interessante. Com a diferença de requerer uma maior modificação no kernel do Linux. Além disso, o Mosix possui uma característica importante, a migração automática entre várias máquinas de um cluster. Através desta característica, é possível que aplicações existentes funcionem num ambiente de cluster com pouquíssimas alterações. (PITANGA, 2002).

2.4.2.4 Outras classificações

Segundo Buyya (1998), existem outras classificações para os clusters, porém as classificações de destaque são quanto à utilização dos nós, quanto ao hardware e quanto à configuração dos nós.

A classificação quanto à utilização dos nós os clusters podem ser divididos em:

- Dedicados, onde todos os recursos do cluster são alocados para unicamente prover o processamento requisitado;
- Não-Dedicados, onde os recursos podem ser compartilhados, de modo que os nós que constituem o cluster possam ser utilizados para outros propósitos, além de estarem sendo utilizados para fins de processamento no cluster.

Quanto ao hardware, existem dois tipos principais de clusters:

- PCs (CoPCs – Cluster of PCs), onde os nós constituintes do cluster são computadores pessoais;
- Workstations (COW – Cluster of Workstation), onde o cluster é composto por nós do tipo estação de trabalho;

E por último, quanto à configuração de nós, os clusters podem ser:

- Homogêneos, onde os clusters são constituídos por nós com a mesma arquitetura e sistema operacional;
- Heterogêneos, onde os clusters são constituídos por nós com arquiteturas e sistemas operacionais distintos.

2.4.3 Justificativa para o uso de clusters

Há algum tempo, o paralelismo era visto como uma rara, exótica e interessante subárea da computação, mas de uma pequena relevância para o programador comum. Um estudo das tendências em aplicações, das arquiteturas de computadores e redes de comunicação, mostra que essa visão não é mais sustentável, pois o paralelismo tornou-se freqüente e a programação paralela está se tornando um fator imprescindível para a programação em ambientes de alto desempenho (PITANGA, 2002).

Empresas de menor porte e universidades, com poucos recursos financeiros, podem recorrer a uma alternativa cada vez mais freqüente para obtenção de processamento de elevado desempenho, a custos bem razoáveis, aproveitando o hardware existente. Essa alternativa pode ser concretizada através do uso de CoPCs, com desempenho da ordem de Teraflops4, o que se tornou possível através do desenvolvimento e barateamento da tecnologia das redes locais e da evolução dos processadores.

CAPÍTULO 3 – AMBIENTES DE PASSAGEM DE MENSAGENS

O emprego da computação paralela sobre os sistemas distribuídos necessita de uma camada de *software* para gerenciar o uso paralelo, um ambiente de passagem de mensagens, pela necessidade de troca de informações entre as máquinas que compõem o ambiente.

Os ambientes de passagem de mensagens são ambientes de programação paralela para memória distribuída portáteis, pois permitem o transporte de programas paralelos entre diferentes arquiteturas e sistemas distribuídos de maneira transparente (SANTANA, 1997). Estes ambientes são responsáveis por prover recursos à computação paralela, tais como, comunicação, criação e sincronização entre processos.

Atuando também como extensões de linguagens seqüenciais, as bibliotecas de passagem de mensagens mais utilizadas são o *MPI (Message Passing Interface)* e o *PVM (Parallel Virtual Machine)*. Ambas provêem rotinas para iniciar e configurar o ambiente e, efetuar o recebimento de entrega de mensagens de dados entre os elementos de processamento do sistema.

Os ambientes *MPI* e *PVM* obtiveram uma grande aceitação por proporcionar o desenvolvimento de aplicações paralelas a um custo relativamente baixo em relação à arquiteturas paralelas (SANTANA, 1997).

Os mesmos estão citados a seguir:

3.1 *PVM (Parallel Virtual Machine)*

O *PVM* é um ambiente paralelo virtual caracterizado como uma das plataformas mais utilizadas e é um padrão de fato, haja vista possuir uma grande utilização e popularidade em diversos setores, tais como o acadêmico, o industrial e o comercial (GEIST et al., 1994).

Sua criação ocorreu em 1989 no *ORNL (Oak Ridge National Laboratory)*, permitindo que um grupo de computadores e diferentes arquiteturas fosse conectado, formando assim, uma máquina paralela virtual (GEIST, 1994).

O *PVM* é composto por um conjunto de bibliotecas e ferramentas, que tem por objetivo a emulação de um sistema computacional concorrente heterogêneo, flexível e de propósito geral (BEGUELIN, 1994). Este ambiente pode ser dividido em duas partes principais: uma formada por um conjunto de processos *daemon*, mais conhecido como *pvmd3* ou *pvmd*, e estes são executados em todos os elementos de processamento, com o objetivo de

formar uma máquina paralela virtual, realizar a comunicação entre os processos criados e coordenar as tarefas em execução, e a segunda parte sendo uma biblioteca de programação que trata funções básicas para geração do paralelismo como troca de mensagens, criação e eliminação de processos, sincronização de tarefas, modificação da máquina virtual e envio e recebimento de mensagens (GEIST, 1999).

Existem inúmeras vantagens na utilização do *PVM*:

- Capacidade de paralelização escalável e dinâmica;
- Facilidade de instalação e uso;
- Software de domínio público;
- Relativa difusão e aceitação;
- Flexibilidade;
- Variedade de arquiteturas e redes de trabalho;
- Recurso computacional facilmente expansível;
- Independência das aplicações;
- Facilidade de programação com a ajuda de bibliotecas para linguagens geralmente utilizadas na computação científica.

Com o *PVM*, uma rede de computadores heterogêneos pode desempenhar as funções de um computador com memória distribuída e alto desempenho. O *PVM* oferece funções para inicialização, comunicação e sincronização de tarefas na máquina virtual. Uma tarefa é uma unidade computacional em *PVM*, análoga aos processos em UNIX (SUNDERAM, 1994).

O modelo do *PVM* é baseado na noção de que uma aplicação consiste em diversas tarefas. Cada tarefa é responsável pela execução de uma parte da aplicação. Uma aplicação pode ser paralelizada através de dois métodos: o paralelismo funcional e o paralelismo de dados. No paralelismo funcional, a aplicação é dividida através de suas funções, ou seja, cada tarefa desempenha um serviço diferente. O paralelismo de dados refere-se ao paradigma de arquiteturas *SIMD*.

Os programas executados nos diversos processadores devem, necessariamente, serem iguais, podendo, com isso utilizar os conceitos do paradigma *SIMD*. Esse paradigma implica que o mesmo código fonte seja distribuído pelos processadores, e cada processador deve executá-lo de forma independente, o que implica na execução de diferentes partes desse

programa, em cada um dos processadores. Quando se distribui códigos fontes distintos para os processadores, utiliza-se o paradigma de arquiteturas *MIMD*.

3.2 *MPI (Message Passing Interface)*

O *MPI* (*Message Passing Interface*) é um padrão internacional de interface para troca de mensagens em máquinas paralelas com memória distribuída (FOSTER, 1995), haja vista a imensa quantidade de bibliotecas de passagem de mensagens encontradas atualmente.

Nesta interface, uma aplicação é constituída de um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos, garantindo dessa forma, a comunicação e a sincronização entre os processos.

Existem diversas versões do *MPI*, sendo as pioneiras, a versão 1.0, lançada em 1993 e que teve como exemplos de implementação, o *MPICH* e o *LAN*, e a versão 2.0, lançada em 1997 e que em comparação à versão anterior, teve correções e novas funcionalidades, tais como a criação dinâmica de processos, E/S paralela e comunicação unilateral (MPI, 1997). A versão mais recente do *MPI*, denominado *OpenMPI*, possui novas primitivas, as quais facilitam ainda mais a programação paralela.

Pela inexistência da sobrecarga na carga de processos em tempo de execução, os programas escritos em *MPI* tendem a ser mais eficientes que os escritos em *PVM*. Entretanto, necessita-se explicitar a criação das tarefas, suas comunicações e destruição, o que pode ser feito através de algumas funções.

Nota-se a existência de diversas variações nas funções básicas da biblioteca de troca de mensagens dentre as diferentes implementações existentes, sejam comerciais de ou de domínio público. As implementações de domínio público mais conhecidas são o *LAM/MPI* (LAM, 2006) e MPICH.

Segundo David Walker (1994), existem uma série de motivos que explicam a necessidade de um padrão para esse tipo de sistema:

- Portabilidade e facilidade de uso: Com o *MPI* é possível transportar aplicações entre diferentes plataformas, de maneira rápida e transparente;
- Fornecer uma especificação detalhada: Manter um conjunto bem definido de padrões e rotinas;

- Crescimento da indústria de *software* paralelo: A existência de um padrão torna a criação de *software* paralelo por empresas, uma opção comercialmente viável;
- Incentivar uma maior utilização nas arquiteturas paralelas: Difundir o uso de computadores paralelos;

O *MPI* baseia-se nas melhores características de todos os ambientes de passagem de mensagens e tenta explorar as vantagens de cada um deles.

3.2.1 Especificação do *MPI*

O *MPI* (2008) define um conjunto de 129 comandos para a biblioteca *MPI*, que oferecem os seguintes serviços:

- Comunicação ponto-a-ponto e coletiva: O *MPI* implementa diversos tipos de comunicação.
- Suporte para grupos de processos: O *MPI* relaciona os processos em grupos, e esses processos são identificados pela classificação deste grupo. Essa classificação dentro do grupo é denominada *rank.*
- Suporte para contextos de comunicação: Contextos podem ser definidos como escopos que relacionam um determinado grupo de processos. Esses tipos de instâncias são implementadas com o intuito de garantir que não existam mensagens que sejam recebidas ambiguamente por grupos de processos não relacionados. Então, um grupo de processos ligados a um contexto não consegue comunicar-se com um grupo que esteja definido em outro contexto;
- Suporte para topologias: O *MPI* fornece primitivas que permitem ao programados definir a estrutura topológica que descreve o relacionamento entre processos.

O padrão *MPI* possui ainda as seguintes características:

- Eficiência: Foi cuidadosamente projetado para executar com eficiência em diferentes máquinas, especifica somente o funcionamento lógico das operações e deixa a implementação em aberto, permitindo aos desenvolvedores aperfeiçoarem o código usando características específicas de cada máquina;
- Facilidade: Define uma interface não muito diferente dos padrões *PVM*, *Express*, etc;
- Portabilidade: É compatível com sistemas de memória distribuída, memória compartilhada, redes de servidores e uma combinação deles;
- Transparência: Permite que o programa seja executado em sistemas heterogêneos sem mudanças significativas;
- Segurança: Provê uma interface de comunicação confiável;
- Escalabilidade: O *MPI* permite o crescimento em escala sob diversas formas, por exemplo, uma aplicação pode criar subgrupos de processos que permitem operações de comunicação coletiva para melhorar o alcance dos processos.

3.2.2 Comunicação do *MPI*

As rotinas de comunicação ponto-a-ponto e coletiva, formam o núcleo básico do *MPI*. Antes de apresentar essas rotinas, é importante entender como o *MPI* organiza uma mensagem.

Uma mensagem no *MPI* é definida como um vetor de elementos de um determinado tipo. Ao enviar uma mensagem, deve-se informar o endereço do primeiro elemento e a quantidade de elementos que formam o vetor. Esses elementos devem ser do mesmo tipo.

Deve-se indicar na mensagem o tipo dos elementos que são enviados. Essa característica é essencial quando a comunicação é realizada entre sistemas heterogêneos, tornando necessária a conversão de dados.

O *MPI* permite que se criem tipos definidos pelo usuário, tornando possível enviar mensagens compostas por elementos de tipos distintos, como por exemplo, estruturas.

Para as operações de comunicação, o *MPI* (2008) define:

Rotina Send(): A maneira eficiente de implementação de uma rotina de envio depende, de certa maneira, do protocolo e da plataforma paralela sobre as quais o *MPI* está sendo executado. Para garantir a eficiência em qualquer plataforma, o *MPI* define vários modos de comunicação, que definem diferentes semânticas para a rotina. Os modos disponíveis, que apresentam versões bloqueantes (ficam bloqueados o recebimento de uma mensagem) e não-bloqueantes (não esperam por respostas) são:

- Síncrono: O transmissor, ao enviar uma mensagem, espera uma confirmação de recepção de mensagem;
- Com *buffer*: Mensagens são transmitidas via uma área temporária explicitamente criada pelo programador;
- Padrão: O modo mais eficiente de comunicação, podendo ser síncrono o com *buffer*;
- *Ready:* necessita que uma rotina de recebimento corrspondente tenha sido iniciada.

Rotina Receive(): Uma mensagem é selecionada para recebimento pelo número (*rank)* do processo que a enviou e pelo seu identificados (*tag*), dentro de um determinado contexto. Esses dois itens podem ser *wild-cards*, isto é, podem utilizar rotinas que recebem mensagens de qualquer processo e com qualquer *tag*. A rotina recebimento não possui desdobramento em modos, podendo ser bloqueantes ou não. Além disso, existem em todas as rotinas coletivas que garantem operações coletivas eficientes em todas as plataformas paralelas. Todas as operações coletivas no *MPI* são bloqueantes e executam no modo padrão.

3.3 Diferenças entre *PVM* e *MPI*

O *MPI* e o *PVM* são os modelos de passagem de mensagens mais utilizados atualmente e apresentam diferenças importantes, que serão apresentadas a seguir:

- Portabilidade: Programas escritos para uma arquitetura podem ser compilados para outra arquitetura sem grandes mudanças, ou às vezes nenhuma, tanto no *MPI* quanto no *PVM*;

- Máquina virtual: O *MPI* não utiliza esta abstração, pois usa fielmente a visão de passagem de mensagens, enquanto o *PVM* agrupa máquinas, a fim de criar a abstração de um único recurso existente;
- Manipulação de mensagens: O *PVM* utiliza o conceito de empacotamento de dados e não permite a passagem de mensagens formadas por estruturas e vetores não contíguos. Porém, o *MPI* pode ou não empacotar os dados, assim permite a passagem de estruturas complexas;
- Tolerância a falhas: O *PVM* apresenta esquemas básicos de notificação de falhas para alguns casos. Porém, permite flexibilidade de forma que, ainda em certas situações onde não existe resposta de uma determinada máquina, uma aplicação pode continuar a receber resultados das outras máquinas.

Baseando-se nas diferenças existentes entre eles, conclui-se que o *MPI* é mais adequado para sistemas homogêneos em que necessitam de muita comunicação e o desempenho é o fator decisivo. Já o *PVM* é indicado para sistemas heterogêneos ou que possuem uma granulosidade grossa com pouca comunicação. Entretanto, encontra-se atualmente uma quantidade muito maior de desenvolvedores em *MPI* do que em *PVM*.

3.4 Considerações finais

Os ambientes virtuais constituem uma abordagem eficiente para a implementação de programas paralelos utilizando troca de mensagem. A possibilidade de transportar programas diretamente entre sistemas distribuídos é um grande atrativo.

Dentre as plataformas de portabilidade destaca-se o *PVM* por sua generalidade e adequabilidade a uma gama de aplicações, entretanto, faz-se ainda necessário uma plataforma de portabilidade que possibilite eficiência e segurança em qualquer plataforma paralela e que seja um padrão, de modo a ser aceito largamente na comunidade computacional, além de oferecer ferramentas não proporcionadas pelo *PVM*, devido a sua simplicidade.

O fato de o *MPI* ter sido projetado para fornecer eficiência, não implica que necessariamente sua implementação o será. O Fórum *MPI* define apenas indicações de possíveis maneiras eficientes de implementação de determinadas partes do padrão. Portanto, aos poucos as características e funcionalidades do *PVM* estão sendo incorporadas ao *MPI* e

vice-versa, e em breve será disponibilizado o *PVMPI*, uma unificação dos dois métodos, para que se tenha de fato um padrão para ambientes paralelos distribuídos.

CAPÍTULO 4 – METODOLOGIA E PROJETO

No presente capítulo, são apresentadas as características de desenvolvimento do projeto, tais como: tecnologias utilizadas, montagem e configuração do *cluster* e a proposta do algoritmo distribuído para validar o projeto.

4.1 Tecnologias utilizadas

Para o desenvolvimento do projeto, foram utilizados, os laboratórios disponíveis no UNIVEM com os *softwares* necessários para o pleno funcionamento da arquitetura e conseqüentes testes de desempenho do *cluster*.

Foi empregado o sistema operacional Linux Fedora Core 9 sobre 3 máquinas homogêneas (Pentium IV de 2.7 GHz com 512 Mbytes, interligadas por um switch de 20 portas, com 100 Mb/s), onde foi montado, instalado e configurado o *cluster*, com a apoio do ambiente paralelo distribuído *MPI* e desenvolvida a proposta de implementação do algoritmo paralelo e as avaliações de desempenho do algoritmo em questão.

Para a implementação da proposta do algoritmo distribuído e avaliações de forma seqüencial e paralela, foi empregado o ambiente *MPICH2*.

O desempenho do algoritmo implementado foi validado por meio de análises estatísticas (Teste de Hipóteses), a fim de comprovar seu grau de significância. Os testes de desempenho realizados, encontram-se no capítulo 5.

A bibliografia apresenta-se com pesquisas na internet, projetos apresentados em simpósios e livros, que foram de grande importância para a aquisição de conhecimento sobre o assunto abordado no projeto de pesquisa.

4.2 Montagem e configuração do *cluster*

Para a montagem do *cluster Beowulf*, foi instalado e configurado o ambiente de passagem de mensagens *MPI*, cuja instalação está descrita a seguir.

Inicialmente, foi revisada uma versão do *MPI* para o projeto. Nesse caso, foi utilizada a versão mpich2-1.0.5. O arquivo encontra-se a princípio em formato tar, portanto, moveu-se o arquivo para um local adequado à descompactação com o comando:

mv mpich2-1.0.5.tar.gz /usr/local

Feito isso, com o arquivo já em */usr/local*, é feita a descompactação do mesmo, com o comando:

tar –zxvf mpich2-1.0.5.tar.gz

Assim, o sub-diretório mpich-1.0.5, foi criado em /usr/local, podendo dar prosseguimento à instalação. Nesse diretório, deve-se configurar a biblioteca de passagem de mensagens *MPI*. O que pode ser feito com os seguintes comandos:

cd mpich-1.0.5
./configure –prefix=/usr/local

O primeiro comando entrou no diretório citado acima e o segundo, configurou a biblioteca de passagem de mensagens. Em que: o *configure* analisa se todos os requisitos para a instalação estão disponíveis para a compilação e configura os parâmetros de compilação de acordo com o sistema, o *prefix* serve para especificar o diretório onde o pacote será instalado e */usr/local* é o caminho do diretório onde deverá ser instalada a aplicação.

Em seguida, deve-se compilar o *software* com o comando:

make

Para copiar os arquivos gerados na compilação, no local especificado utiliza-se o comando:

make install

Após a instalação, deve-se adicionar a pasta bin do mpich à declaração *PATH*. Para tanto, deve-se acessar o arquivo "*.bash_profile*", que localiza-se na pasta */root.* O que pode ser feito com os seguintes comandos:

cd /root
gedit .bash_profile

A seguir, basta acrescentar as seguintes linhas no arquivo:

MPIR_HOME=/usr/local/mpich2-1.0.5
PATH=$MPIR_HOME/bin:$PATH
export PATH

Feito isso, salva-se o arquivo. Para que as modificações nas variáveis de ambiente sejam efetivamente realizadas, deve-se reinicializar o *Linux.*

É obrigatória, a criação do arquivo *mpd.conf* para o usuário *root* (*.mpd.conf* para qualquer outro usuário), sendo que para o usuário *root*, o arquivo deve ser criado em */etc*. Dentro do arquivo, deve-se armazenar o seguinte texto:

MPD_SECRETWORD=mpipassword

No caso, "*mpipassword*" foi a senha utilizada no projeto, o qual, pode ser substituída por qualquer outra senha. Definida a senha, deve-se configurar as permissões do arquivo *mpd.conf* para que o mesmo só possa ser visualizado e modificado pelo proprietário, o que pode ser feito pelo comando:

chmod 600 mpd.conf (ou .mpd.conf)

No diretório */etc*, deve-se criar um arquivo com a lista de todos os nós participantes do *cluster*, este arquivo informa ao *MPI*, quais as máquinas do *cluster* estão acessíveis ao mesmo, este arquivo tem o nome *mpd.hosts*. Este arquivo deve ser replicado em todas as máquinas.

gedit /etc/mpd.hosts

A lista de nós a ser listada fica a critério de quais nós deseja-se utilizar no *cluster*:

no1
no2
no3

Para testar se até o momento, o sistema está configurado corretamente, pode-se utilizar os seguintes comandos:

mpd &
mpdtrace
mpdallexit

Em que, o "*mpd &"* inicializa a execução do servidor *MPI*, o "*mpdtrace*" mostra as conexões ativas que no caso é apenas um servidor e "*mpdallexit*" fecha o servidor *MPI*.

Até o momento, caso os três comandos acima tenham ocorrido com sucesso em todas as máquinas, tem-se a certeza de que o *MPI* está instalado e configurado corretamente nas máquinas para execução seqüencial, sendo necessária ainda, a configuração dos arquivos para a comunicação entre as máquinas (sem a autenticação por senha).

Antes de começar a configuração para fazer *rsh* sem senha é necessário verificar se o *rsh-server* está instalado no sistema, se não estiver é necessário instalá-lo. Todos os nós que farão *rsh* sem senha devem ter o mesmo usuário com mesmo id e mesmo grupo. Isso pode ser feito pelo comando:

rpm -qa | grep rsh-server

Caso o mesmo não esteja instalado, deve-se instalá-lo pelo comando:

yum install rsh-server

A seguir, para configurar o rsh-server, deve-se modificar alguns arquivos que encontram-se no diretório */etc/xinetd.d/*, são eles *rexec*, *rlogin* e *rsh*. Seguem os comandos:

cd /etc/xinetd.d/
gedit rexec

Deve ser alterada a linha para *DISABLE=no*, em *rexec*, *rlogin* e *rsh*.

O próximo arquivo a ser alterado encontra-se no diretório */etc/pam.d/*, denominado *rsh*.

cd /etc/pam.d/

gedit rsh

Troca-se a linha:

auth required pam_rhosts_auth.so

Por:

auth sufficient pam_rhosts_auth.so

No mesmo arquivo, a seguinte linha deve ser comentada:

 # auth required pam_securetty.so

Devido ao sistema de segurança do Fedora ser autoritário, é necessário desabilitar os iptables e inicializar os serviços do rsh. Faz-se:

service iptables stop

ntsysv

A tela apresentada, mostra todos os serviços que são inicializados com o *boot*, portanto a opção *iptables* deve ser desativada e as opções *rlogin*, *rsh* e *sshd* devem ser habilitadas.

Os arquivos a seguir, devem ser criados e replicados em todas as máquinas participantes do processo.

Os primeiros arquivos a serem editados são o "*hosts*" e o "*hosts.allow*", que podem ser encontrados no diretório */etc*. Os arquivos *hosts* e *hosts.allow* devem ter os *IP*s, os nomes e apelidos atribuídos às máquinas participantes do processo. Caso haja necessidade, o comando *ifconfig* retorna os *IP*s de cada máquina, assim como o comando *hostname* retorna os nomes e apelidos conforme foram definidos no *DNS (Domain Name System)*. Assim, consegue-se prover uma maior transparência ao usuário, pois o conhecimento entre as máquinas participantes ocorre pelos nomes atribuídos a elas, mascarando o *IP*. A

configuração do arquivo *hosts* e do *hosts.allow* utilizados no projeto são idênticas e estão listada abaixo:

 # <IP> <hostname> <hostname>
192.168.1.32 no01 no01
192.168.1.55 no02 no02
192.168.1.23 no03 no03

O arquivo seguinte a ser configurado é o *hosts.equiv* que também está localizado no diretório */etc*. Estabelece-se assim a relação de confiança entre as máquinas, para que haja relacionamento de equivalência entre elas sem a necessidade de autenticação por senha. A configuração do arquivo *hosts.equiv* segue abaixo:

 # <hostname>
no01
no02
no03

A seguir, configura-se o arquivo *.rhosts*, que deverá constar em cada diretório de trabalho do usuário como */home* e */root*. O arquivo encontra-se oculto ao comando *ls* por causa do ".". Este arquivo será utilizado pelo protocolo *rsh* para a execução dos comandos remotos e por algumas aplicações de monitoramento. A configuração do arquivo está descrita abaixo:

no01
no02
no03

O último arquivo referente à comunicação entre as máquinas que deverá ser configurado é o *securetty*. O arquivo encontra-se no diretório */etc* e necessita apenas que se acrescente *rsh*, *rexec* e *rlogin* ao final do arquivo, um em cada linha. Incluem-se dessa forma as chamadas aos protocolos *rsh*, *rexec* e *rlogin*, habilitando o acesso sem senhas entre as máquinas. Um exemplo de configuração está descrito abaixo:

tty1

tty2

tty3

...

rsh

rexec

rlogin

Feito isso, deve-se verificar se o acesso entre as máquinas, por meio dos protocolos *rsh* e *ssh*, está funcionando. A sintaxe do comando é:

rsh <hostname> (em que *hostname* é o nome da máquina a se acessar)

O *rsh* necessita de senha para o acesso à outra máquina, porém, caso o *ssh* também necessite de senha, é necessário gerar um par de chaves (pública e privada) para o usuário que terá permissão de efetuar ssh sem a digitação de senha, para isso, é usado o comando ssh-keygen.

```
cd /root
ssh-keygen -b 1024 -t rsa
```

Quando for solicitada a digitação de senha, a tecla *ENTER* deve ser pressionada para que nenhuma senha seja armazenada.

Na pasta */root/.ssh* surgiram dois arquivos: *id_rsa* (chave privada) e *id_rsa.pub* (chave pública). Após a geração das chaves, é necessário copiar a chave pública para as outras máquinas.

```
scp /root/.ssh/id_rsa.pub no2:/home/usuario/.ssh/
scp /root/.ssh/id_rsa.pub no3:/home/usuario/.ssh/
```

Deve-se efetuar um *ssh* para cada máquina de destino e criar o arquivo *authorized-keys*, com a respectiva chave pública:

```
cat /root/.ssh/id_rsa.pub >>/ root/.ssh/authorized_keys
```

A seguir, devem ser alteradas as permissões sobre o arquivo *authorized_keys*:

chmod 600 /root/.ssh/authorized_keys

Efetuando um *exit* e um *ssh* para a máquina destino, não deve ser mais necessária a inserção da senha.

Caso as configurações acima tenham ocorrido com sucesso, o *mpich* está praticamente pronto para executar aplicações paralelas, não obstante, é recomendado efetuar algumas configurações do nível de segurança para evitar problemas na execução das aplicações. Nesse caso, as configurações referentes ao nível de segurança devem ser verificadas. Isso por ser feito em *Desktop*, Configurações do Sistema, Nível de Segurança.

Na janela de configuração aberta, na aba de "Opções de *Firewall*", deve-se desabilitar o *firewall* (caso esteja habilitado), já na aba *SELINUX*, deve-se deixar desabilitadas as duas opções. E, no diretório */etc/sysconfig*, deve-se conter a linha *SELINUX=disabled*.

Feito isso, o *mpich2* deve ser configurado para trabalhar em conjunto e não apenas sequencialmente, para tanto, deve-se editar o arquivo *mpiconf.sh* a ser criado no diretório */etc/profile.d/*.

gedit /etc/profile.d/mpiconf.sh

O seguinte *script* deve ser inserido no arquivo:

PATH=$PATH:/usr/local/mpich2-1.0.5/bin
export PATH

Para que o *script* possa ser executado por todos os usuários, deve-se alterar a permissão do arquivo:

chmod og+x mpiconf.sh

Com sucesso em todas as configurações acima, inicia-se a operação do *cluster*. Antes de executar qualquer aplicação, é necessário iniciar a execução dos servidores em todas as máquinas que participarão do processo.

No nó mestre, executa-se o comando "*mpd &*", nos nós escravos, executa-se o comando "*mpd –h hostname –p port &"*, em que *hostname* é o nome do servidor mestre e *port* é a porta em que ele está sendo executado.

Caso haja necessidade, as duas informações acima podem ser encontradas através dos comandos *hostname* e *mpdtrace –l*, respectivamente.

Antes de executar a proposta do algoritmo paralelo a ser apresentada, pode-se testar o correto funcionamento do *MPICH* através de um dos aplicativos de teste que acompanham a ferramenta. O exemplo demonstrado abaixo localiza-se em */usr/local/mpich2-1.0.5/examples* e é denominado *CPI*.

A compilação e execução podem ser realizadas com os seguintes comandos, respectivamente:

mpicc cpi.c –o cpi

mpirun –np 3 /usr/local/mpich2-1.0.5/examples/cpi

Caso tudo tenha ocorrido com sucesso, o resultado da execução do programa será algo parecido com o trecho abaixo:

Process 0 of 3 on no01

Process 1 of 3 on no02

Process 2 of 3 on no03

pi is approximately 3.1415926544231323, Error is 0.0000000008333392

Nesse momento, tem-se a certeza de que tudo está funcionando perfeitamente.

4.3 Proposta do algoritmo distribuído

Para a validação do presente projeto, foi implementado um algoritmo que faz a distribuição do algoritmo *ShellSort*, que foi escolhido devido a apresentar um desempenho médio entre todos os algoritmos de ordenação existentes (não tão ruim quanto o *BubbleSort* e nem tão bom quanto o *QuickSort*). A idéia foi basicamente distribuir partes do vetor para os que os nós escravos efetuem a ordenação enquanto o mestre efetua a ordenação da sua parte e

em seguida, o nó mestre recebe as partes do vetor e efetua a intercalação entre eles, para que o vetor fique completamente ordenado. Foram efetuados testes com vetores de tamanhos 150, 150000 e 900000, respectivamente, a fim de comprovar que vetores pequenos são favoráveis à execução seqüencial, enquanto vetores grandes são favoráveis à execução distribuída. O algoritmo utilizado no projeto com 3 nós e vetor de tamanho 900.000, segue no ANEXO A.

Este algoritmo possui funções para preencher o vetor com valores aleatórios (utilizado pelo nó mestre), exibir o vetor (para checagem do perfeito funcionamento da ordenação), *shellSort* (algoritmo de ordenação) e intercalar dois vetores (para juntar as partes ordenadas por cada nó em um único vetor).

Para executar as comparações de tempo entre os algoritmos com vetores de diferentes tamanhos e diferentes quantidades de nós no *cluster*, foi efetuada a subtração do tempo final e o tempo inicial de execução do algoritmo através da função *gettimeofday()*.

As diferenças entre os algoritmos resumiram-se basicamente em alterar o valor da constante TAM com os valores 150, 150.000 e 900.000. No algoritmo seqüencial, não existiu comunicação entre nós, já no algoritmo com dois nós, o nó mestre enviou metade do vetor ao nó filho, ambos efetuaram o *shellSort* e o nó mestre efetuou a intercalação dos vetores, para obter o resultado final. No algoritmo com três nós, seguiu-se a mesma lógica, o vetor foi divido em 3 e intercalado 2 vezes para se obter o vetor final ordenado.

4.4. Considerações Finais

Este capítulo apresentou a metodologia utilizada no projeto, demonstrando o ambiente e os equipamentos utilizados para o feito. Foi apresentado o passo a passo da instalação do ambiente de passagem de mensagens denominado *MPI*, o qual permitiu que as máquinas se comunicassem e também, a proposta de algoritmo de validação do projeto, bem como uma análise do funcionamento do algoritmo e seus resultados esperados.

Foram encontradas dificuldades durante a configuração do *cluster*, tais como a necessidade de se reconfigurar os repositórios, pois o mesmo estava desatualizado e não encontrava os arquivos para baixá-los, a reconfiguração dos arquivos de rede, pois os computadores por padrão, recebiam IP do servidor, não obstante, por fim, os problemas não apresentaram muita dificuldade em serem resolvidos.

Com intuito de avaliar o desempenho das implementações em paralelo, o próximo capítulo faz uma análise de desempenho dos resultados obtidos por meio da comparação das implementações em paralelo e em seqüencial e também apresenta as conclusões finais.

CAPÍTULO 5 – ANÁLISE DE DESEMPENHO E CONCLUSÕES

A análise de desempenho da proposta de algoritmo paralelo foi desenvolvida por meio de diferentes testes reais em um ambiente paralelo distribuído composto por máquinas homogêneas (Pentium IV de 2.7 GHz com 512Mbytes, interligadas por uma rede *ethernet* de 100Mb/s), em que inicialmente foi utilizada uma máquina e posteriormente foram inseridas novas máquinas, atingindo o limite de 3 máquinas.

Foi implementado um algoritmo paralelo que lê valores armazenados em vetores de diferentes tamanhos (150, 150.000, 900.000), respectivamente e os ordena de forma crescente.

A análise de desempenho foi feita com base em medidas relacionadas ao tempo de processamento (em milissegundos).

As próximas seções apresentam os resultados da execução seqüencial e paralela das técnicas implementadas utilizando-se vetores de diferentes tamanhos.

5.1 Vetor de Tamanho 150

A figura a seguir apresenta os resultados da execução seqüencial e paralela da proposta de algoritmo, com as médias de tempo da execução de 1, 2 e 3 nós, respectivamente.

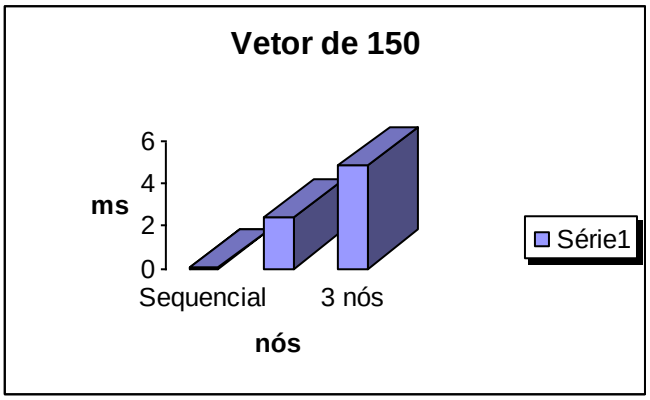

Gráfico 1 – Vetor de tamanho 150

Tempos:

Seqüencial: 0,032 ms 2 nós: 2,37 ms 3 nós: 4,865 ms

Pela figura é possível observar que o tempo de execução do algoritmo seqüencial é significativamente melhor que o tempo do mesmo em paralelo independentemente do número de máquinas e da quantidade de processos iniciados em paralelo.

Uma vez que a quantidade de ordenações efetuadas pelos nós é relativamente pequena, a paralelização do mesmo não impõe melhoria, pois se consome mais tempo com comunicação para envio de dados por meio da rede do que no processamento da ordenação propriamente dita, ou seja, o tempo utilizado para o envio de cada parte do vetor é maior que o tempo gasto pelos escravos para realizar o processamento (ordenação desses subvetores), o que a torna uma aplicação mais voltada para comunicação do que para processamento.

5.2 Vetor de Tamanho 150.000

A figura a seguir, refere-se à execução com vetor de 150.000 elementos:

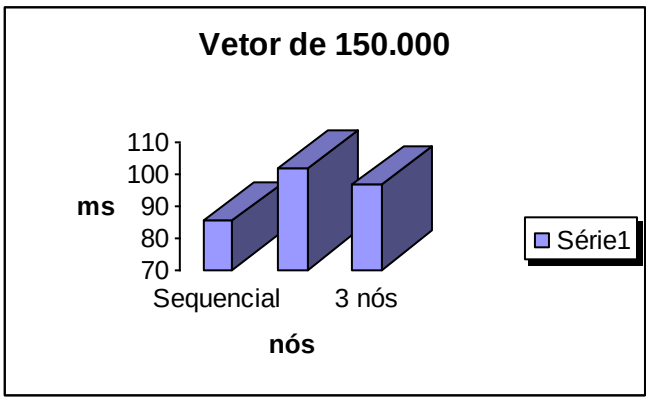

Gráfico 2 – Vetor de tamanho 150.000

Tempos:

Seqüencial: 85,419 ms 2 nós: 101,52 ms 3 nós: 96,693 ms

Nota-se na figura anterior que o tempo de execução dos nós, de forma seqüencial, apresentou um desempenho superior aos outros, não obstante, a execução com 3 nós obteve um melhor desempenho que a execução com 2 nós. O vetor utilizado possui um bom tamanho, assim, mesmo que a execução seqüencial ainda seja melhor, verifica-se que o aumento do número de nós passa a ser um fator relevante para o aumento de desempenho de forma gradativa, ou seja, embora a comunicação ainda seja um fator que influencia na perda

de desempenho, o aumento do número de nós vai diminuindo gradativamente seu tempo de execução.

5.3 Vetor de Tamanho 900.000

Pela figura a seguir, é possível observar que o tempo de execução dos nós, independentemente do número de máquinas utilizadas, apresenta um melhor desempenho se comparado ao tempo seqüencial. O vetor utilizado é grande, dessa forma, as ordenações efetuadas pelos escravos são volumosas, garantindo dessa forma, uma melhoria bastante significativa do uso em paralelo uma vez que a comunicação imposta se torna desprezível quando comparada às ordenações realizadas.

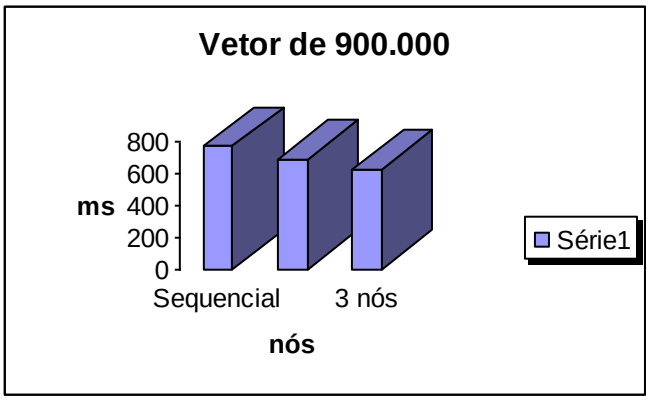

Gráfico 3 – Vetor de tamanho 900.000

Tempos:

Seqüencial: 766,407 ms 2 nós: 690,222 ms 3 nós: 624,752 ms

5.4 Conclusões

O projeto de pesquisa teve como objetivo principal demonstrar que os *clusters* são uma excelente opção de baixo custo ao uso de supercomputadores comerciais, se igualando e até, as vezes ultrapassando seu poder computacional a um custo muitas vezes inferior ao valor de uma máquina comercial. Para tanto, utilizou-se da programação paralela e distribuída, viabilizada por sistemas distribuídos e pela biblioteca de passagem de mensagens *MPI*.

Tal fato se comprova com a novidade de que recentemente o Departamento de Defesa dos Estados Unidos montou um *cluster Beowulf*, com computadores pessoais e

rodando *Linux*, para substituir seu supercomputador comercial e suas simulações de movimentações de tropas em caso de guerra.

Na análise de desempenho realizada por meio de testes das implementações dos algoritmos na forma seqüencial e paralela, foram obtidos os seguintes resultados e informações:

- O processamento seqüencial só obtém um desempenho superior à execução paralela quando a quantidade de processamento executada pelos escravos é baixa, pois a comunicação entre os nós consome um tempo maior que o pouco processamento feito pelos escravos.
- Os resultados apresentados na seção anterior demonstraram que a proporção de aumento do tamanho do vetor se reflete no tempo de processamento paralelo de forma exponencial. Quanto maior o tamanho do vetor, melhor o resultado obtido com a paralelização.
- À medida que se aumenta o número de processos, diminui-se o desempenho, sendo um resultado já esperado, demonstrando o impacto que o tempo de comunicação entre as máquinas exerce sobre o tempo de processamento.

Embora a computação paralela distribuída viabilize a transferência de dados e o uso de múltiplas máquinas, possibilitando os resultados obtidos e já apresentados, ela não provê por si própria (o ambiente de passagem de mensagem utilizado provê somente o escalonamento *round-robin*) mecanismos eficientes para a distribuição de processos a processadores (escalonamento de processos objetivando o balanceamento de cargas) (BRANCO, 2004).

Como sugestão para futuros trabalhos, podemos citar a instalação e configuração de outros tipos de *clusters*, por exemplo, o *Openmosix*, e fazer uma comparação de desempenho em relação à arquitetura *Beowulf*. Também como sugestão, poderia efetuar-se a execução de testes de *benchmarks* para a avaliação da performance do conjunto com variação na quantidade de nós utilizado.

Com a elaboração e desenvolvimento deste trabalho, foi adquirido um bom conhecimento acerca do sistema operacional *Linux*, aprendido muitos conceitos sobre arquiteturas paralelas, ambientes de computação paralela e vivenciados na prática, as dificuldades e o aprendizado sobre a implementação de um *cluster* de computadores.

CAPÍTULO 6 – REFERÊNCIAS BIBLIOGRÁFICAS

ALMASI, G. S., GOTTLIEB, A. **Highly Parallel Computing**. 2a. ed. The Benjamin Cummings Publishing Company, Inc., 1994.

BEGUELIN, A.; GEIST, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAM, V. **PVM: Parallel Virtual Machine. A User's Guide and Tutorial for NetWork Parallel Computing**. The MIT Press, s. 1, 1994.

BRANCO, K. R. L. J. C.; SANTANA, M. J.; SANTANA, R. H. C.; BRUSCHI, S. M. PIV and WPIV: **Performance Index For Heterogeneous Systems Evaluation. In: 2006 IEEE International Symposium on Industrial Electronics**. Volume: 1. p. 323-328, 2006.

BRANCO, K. R. L. J. C. **Índices de carga e desempenho em ambientes paralelos/distribuídos – modelagem e métricas**. São Paulo: Instituto de Ciências Matemáticas e Computação, 2004, 260 f. Grau: Tese (Doutorado em Ciência da Computação e Matemática Computacional) Instituto de Ciências Matemáticas e de Computação. Universidade de São Paulo, São Carlos, 2004.

BRANCO, K. R. L. J. C. **Extensão da Ferramenta de Apoio à Programação Paralela (F.A.P.P.) para Ambientes Paralelos Virtuais**. 1999, 152f. Dissertação (Mestrado) – Instituto de Ciências Matemáticas e de Computação de São Carlos, Universidade de São Paulo, São Carlos, 1999.

BUYYA, R. High **Performance Cluster Computing: Architectures and Systems**. Volume 1, Prentice Hall, 1999.

CÁCERES, E. N.; MONGELLI, H.; SONG, S. W. Algoritmos Paralelos Usando CGM/PVM/MPI: **Uma Introdução. In: As Tecnologias da Informação e a Questão Social**. Ed. Porto Alegre: Sociedade Brasileira de Computação, 2001.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. Distributed **Systems: Concepts and Design**. 3. ed. Harlow: Addison Wesley, 771 p, 2001. Distributed Systems: Concepts and Design. 4. ed. Harlow: Addison Wesley, 944 p, 2006.

DONGARRA, J.; FOSTER, I.; FOX, G.; GROPP, W.; KENNEDY, K.; TORCZON, L.; WHITE, A. (Ed.). **Sourcebook of Parallel Computing**. San Francisco: Morgan Kaufmann Publishers, 842 p, 2003.

DONGARRA, J. J.; OTTO, S. W.; SNIR, M.; WALKER, D. **An Introduction to the MPI Standard**. University of Tennessee Technical Report CS-95-274, Disponível em: <http://www.netlib.org/utk/papers/intro-mpi/intro-mpi.html>,1995. Acesso em 09 out. 2009.

DUNCAN, R. **A Survey of Parallel Computer Architectures**. In: Computer. Los Alamitos, CA, USA: IEEE Computer Society Press, (Survey & Tutorial Series, 2). P. 5–16, 1990. FLYNN, M. J. Some Computer Organizations and Their Effectiveness. IEEE Transactions on Computers, v. C, n. 21, pp. 948-960, 1972.

FLYNN, M. J.; RUDD, K. W. **Parallel Architectures**. ACM Computing Surveys, v. 28, n. 1, p. 67–70, mar. 1996.

FOSTER, I. **Designing and Bulding Parallel Programs: Concepts and Tools for Parallel Software Engineering**. New York: Addison-Wesley Publishing Company, 1995. Disponível em: <http://www-unix.mcs.anl.gov/dbpp/text/book.html>. Acesso em: 5 jun. 2009.

GEIST, A.; BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAM, V. **PVM 3 User's Guide and Reference Manual**. Oak National Laboratory, Setembro, 1994.

GIESS, C.; MAYER, A.; EVERS, H.; MEINZER, H. P. "**Medical Image Processing and Visualization on Heterogeneous Clusters of Symmetric Multiprocessors using MPI and POSIX Threads**". Parallel Processing Symposium, 1998. Proceedings of the First Merged International and Symposium on Parallel and Distributed Processing 1998.

HWANG, K.; BRIGGS, F. A. **Computer Architecture and Parallel Processing**. McGraw-Hill International Editions, 1984.

HWANG, K.; XU, Z. Scalable **Parallel Computing: Technology, Architecture, Programming**. 1. ed. New York: McGraw-Hill, 802 p, 1998.

ISO, INTERNATIONAL STANDARDS ORGANIZATION. **Basic Reference Model of Open Distributed Processing, Part 1: Overview and Guide to Use**. ISO/IEC JTC1/SC212/WG7 CD 10746-1, 1992.

LAM. **LAM/MPI Parallel Computing**. Disponível em: <http://www.lam-mpi.org>. Acesso em: 05 Nov 2008.

MORIMOTO, C.E.; **Brincando de Cluster** – Disponível em: <http://www.guiadohardware.net/artigos/cluster/>. – Acesso em 18 Fev 2009.

MORIN, S.; KOREN, I.; KRISHNA, C. M. JMPI: **Implementing the Message Passing Standard in Java**. In IPDPS 02: Proceedings of the 16th International Parallel and Distributed Processing Symposium, p.191, IEEE Computer Society, 2002.

MPI. **MPI-2: Extensions to the Message-Passing Interface**. 1997. Disponível em: <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html>. Acesso em: 14 Out 2008.

MPICH2. **Download Document Home Page**. Disponível em: http://www.unix.mcs.anl.gov/mpi/mpich2/. Acesso em 15 Dez 2008.

NEVIN, N. J. **The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster**. Columbus, Ohio, 1996.

PITANGA, Marcos J., **Construindo Supercomputadores com Linux**, 1a ed., Brasport Livros e Multimídia Ltda, 2002.

QUINN, M.J. **Designing Efficient Algorithms for Parallel Computers. McGraw Hill, 1987. Parallel Computing: Theory and Practice**. 2. ed. New York: McGraw Hill, 1994. 446 p.109

STALLINGS, W. **Arquitetura e Organização de Computadores: Projeto para o Desempenho**. 5. ed. São Paulo: Prentice Hall, 786 p. Tradução: Carlos Camarão de Figueiredo e LucíliaCamarão de Figueiredo, 2003.

SUNDERAM, V. S.; GEIST, G. A. **Heterogeneous Parallel and Distributed Computing. Parallel Computing**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, n. 25, p. 1699–1721, 1999.

SUNDERAM, V. S., GEIST, A., DONGARRA, J., MANCHEK, R**. The PVM Concurrent Computing System: Evolution, Experiences and Trends**. Parallel Computing, v. 20, pp. 531- 545, 1994.

TANENBAUM, A. M. **Modern Operating Systems**. 2ª ed., Prentice Hall International Inc, 2003.

TANENBAUM, A. S. **Distributed Systems: Principles and Paradigms**. Prentice Hall, 704p, 2006.

TANENBAUM, A. S. **Structured Computer Organization**. 5. ed. Upper Saddle River, NJ: Prentice Hall, 2006.

ZALUSKA, E. J. **Research Lines in Distributed Computing Systems and Concurrent Computation**. Anais dos Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software, pp. 132-155, 1991.

ZOMAYA, A. Y. (Ed.). **Parallel and Distributed Computing Handbook**. New York: McGraw-Hill, 1198 p. (Computer Engineering Series), 1996.

WALKER, D. W. **The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers**. Parallel Computing, v. 20, pp 657-673, 1994.

ANEXO A – IMPLEMENTAÇÂO COM 3 NÓS

```
#include <stdio.h>
#include <sys/time.h>
#include "mpi.h"
#define TAM 900000


/* função de inicialização do vetor */
void inicializaVetor(int vetor[])
{
    int i;
    for (i = 0; i < TAM; i++)
        vetor[i] = rand() % TAM;
}


/* função de exibição do vetor */
void exibeVetor(int vetor[])
{
    int i;
    for (i = 0; i < TAM; i++)
        printf("%d\n",vetor[i]);
}


/* função do algoritmo shellsort */
void shellSort(int * vetor, int tamanho)
{
   int i, j, valor;
   int gap = 1;
   do {
      gap = 3 * gap + 1;
```

```
   } while(gap < tamanho);
   do {
          gap /= 3;
          for (i = gap; i < tamanho; i++) {
         valor = vetor[i];
         j = i - gap;
         while (j >= 0 && valor < vetor[j])
         {
            vetor [j + gap] = vetor[j];
            j -= gap;
         }
         vetor [j + gap] = valor;
       }
   } while (gap > 1);
}


/* função para intercalar 2 vetores ordenados */

void intercala (int *v1, int t1, int *v2, int t2, int *v3)
{

    int i=0, j=0, k;

    for (k=0; k<t1+t2; k++)
   {
         if (i < t1 && j < t2)
         {
                  if (v1[i] < v2[j])
                        v3[k] = v1[i++];
                  else
                        v3[k]= v2[j++];
         }
          else
```

```
            if (i==t1)
                v3[k] = v2[j++];
                else
                v3[k] = v1[i++];
        }
}

int main(int argc, char **argv)
{
        int i, envia, recebe, procs, meurank, aux;
        int tag = 10;
        int vetor[TAM];
        int vetor2[TAM], vetor3[TAM];
        struct timeval tinicial, tfinal;
        struct timezone tz;
        int ti_ms;



        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &procs);
        MPI_Comm_rank(MPI_COMM_WORLD, &meurank);

    if (meurank == 0)   // este é o mestre
    {
        gettimeofday(&tinicial, &tz);

        aux = TAM / procs;
        inicializaVetor(vetor);


           for(i = 1; i < procs; i++) {
```

```
                MPI_Send(&vetor[(i-1)*aux], aux, MPI_INT, i, tag,
MPI_COMM_WORLD);

            }
            shellSort(&vetor[2*aux], aux);


           for(i = 1; i < procs; i++) {
              MPI_Recv(&vetor[(i-1)*aux], aux, MPI_INT, i, tag, MPI_COMM_WORLD,
&status);
           }


            // intercalar os vetores ordenados
            intercala(&vetor[0], aux, &vetor[aux], aux, vetor2);
            intercala (&vetor2[0], 2*aux, &vetor[2*aux], aux, vetor3);
            gettimeofday(&tfinal, &tz);


            printf("Tempo de execucao: %f \n", (double)tfinal.tv_usec-
(double)tinicial.tv_usec);
            //exibeVetor(vetor3);

        }

        else
        {
           aux = TAM/procs;
           MPI_Recv(vetor2, aux, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
           shellSort(vetor2, aux);
           MPI_Send(vetor2, aux, MPI_INT, 0, tag, MPI_COMM_WORLD);


        }
```

```
    MPI_Finalize();
    return(0);
}
```