Centro Universitário de Brasília – UNICEUB
Faculdade de Tecnologia e Ciências Sociais Aplicadas - FATECS
Disciplina: Projeto Final

Criação de um *cluster* Beowulf, desenvolvimento de uma aplicação utilizando processamento paralelo e análise de seu desempenho

Luis André Bazzi Morales

RA: 2036741/9

Monografia de Conclusão do Curso de Engenharia de Computação

Orientador(a): Professor Miguel Archanjo Bacellar Goes Telles Junior

Brasília – DF, 5 de junho de 2008.

Luis André Bazzi Morales

Criação de um cluster Beowulf, desenvolvimento de uma aplicação utilizando processamento paralelo e análise de seu desempenho

Trabalho de conclusão de curso apresentado como parte das atividades para obtenção do título de Bacharelado em Engenharia de Computação, do curso de Engenharia de Computação da Faculdade de Tecnologia e Ciências Sociais Aplicadas de Brasília – UniCEUB.

Profº orientador: Professor Miguel Archanjo Bacellar Goes Telles Junior
Brasília – DF, 2008

Autoria: Luis André Bazzi Morales
Título: Criação de um cluster Beowulf, desenvolvimento de uma aplicação utilizando processamento paralelo e análise de seu desempenho

Trabalho de conclusão de curso apresentado como parte das atividades para obtenção do título de Bacharelado em Engenharia de Computação, do curso de Engenharia de Computação da Faculdade de Tecnologia e Ciências Sociais Aplicadas do Centro Universitário de Brasília – UniCEUB.

Os componentes da banca de avaliação, abaixo listados, consideram este trabalho aprovado.				
	Nome	**Titulação**	**Assinatura**	**Instituição**
1				
2				
3				

Data da aprovação: ____ de _____ de _____.

"Dedico este trabalho à minha família, aos meus amigos, a minha namorada que teve muita paciência e em especial ao meu amigo Mario Gomide que me ensinou quase tudo que sei sobre o sistema operacional GNU/Linux."

The Devil finds work
For idle systems
Nature abhors a NoOp

Provérbio antigo da computação paralela

Resumo

O projeto descrito nesse documento tem como objetivo mostrar uma forma de: construir um *cluster* Beowulf, desenvolver uma aplicação que utiliza processamento paralelo e mensurar e analisar o desempenho do sistema a nível de *software*.

O *cluster* construído é composto de 8 computadores com processadores Pentium III 500Mhz interconectados por um *switch* de 100Mbps. Para a realização do processamento paralelo foi utilizada o MPICH2 que é uma implementação da interface de passagem de mensagens (Message Passing Interface – MPI). A aplicação desenvolvida foi escrita utilizando a linguagem de programação C padrão ANSI. Essa tem como proposta resolver uma série de fatoriais dividindo sua carga de trabalho entre os nós de processamento.

Os testes de desempenho foram realizados pelo programa High Performance Linpack (HPL), tendo ao final do mesmo, resultados de performance em Gflops.

Palavras-chave: computação de alto desempenho, *cluster*, Beowulf, processamento paralelo, MPI, HPL, GNU/Linux.

Abstract

The work described in this document has the objective of demonstrate a way of: building a Beowulf cluster, developing an application that uses parallel processing and measure and analyze the system performance at a software level.

The built cluster is composed by 8 computers with Pentium III 500Mhz processors interconnect by a switch with 100Mbps. The MPICH2, witch is a implementation of the message passing interface (MPI), was used in order to make possible the parallel processing. The application developed was made with the standard ANSI C programing language. This program solves a series of large number factorials dividing its workload with other processing nodes.

Performance tests were made by the application High Performance Linpack (HPL), presenting as final results the performance of the system measured in GFlops.

Key-words: high performance computing, cluster, Beowulf, parallel processing, MPI, HPL, GNU/Linux.

Sumário

Lista Figuras

Lista de tabelas

Lista de abreviaturas e siglas

APT - *Advanced Package Tool* (Ferramenta Avançada de pacotes)

ATLAS - *Automatically Tuned Linear Algebra Software* (Otimização automática do programa de álgebra linear)

BIOS – *Basic Input Output System* (Sistema básico de entrada e saída)

BLAS - *Basic Linear Algebra Software* (Programa básico de álgebra linear)

CAD – Computação de Alto Desempenho

CC-NUMA - *Cache-Coherent Non-Uniform Memory Access* (Acesso não uniforme à memória com coerência de cache)

COMA - *Cache-only memory architecture* (Arquitetura com somente memória cache)

COW - *Cluster of Workstations* (*Cluster* de estações de trabalho)

DHCP – *Dynamic Host Configuration Protocol* (Protocolo de configuração dinâmica de máquina)

DNS – *Domain Name System* (Sistema de nome de domínio)

E/S – Entrada/Saída

EP - Elemento Processador

GCC - GNU C *Compiler* (Compilador C do GNU)

GNU – GNU's *Not Unix* (GNU´s não é Unix)

GRUB - *Grand Unified Bootloader* (Grande carregador unificado)

HD – *Hard Drive* (Dico rígido)

HPL - *High-Performance Linpack* (Linpack de alto desempenho)

IBM - *International Business Machines* (Máquinas de negócio internacional)

IDE - *Integrated Development Environment (*Ambiente de desenvolvimento integrado)

IP – *Internet Protocol* (Protocolo de *internet*)

LILO - *Linux Loader* (Carregador do Linux)

M²COTS - *Mass Market Commodity-Of-The-Shelf* (Componentes disponíveis ao mercado consumidor comum tirados da prateleira)

MBR - *Master Boot Record* (Principal registro de inicialização)

MIMD - *Multiple Instruction Multiple Data* (Múltiplas instruções múltiplos dados)

MISD - *Multiple Instruction Single Data* (Múltiplas instruções único dado)

MPD - *MultiPurpose Daemon* (*daemon* de propósito múltiplo)

MPI – *Message Passing Interface* (Interface de passagem de mensagem)

MPP - *Massively Parallel Processors* (Processadores massivamente paralelos)

NASA - *National Aeronautics and Space Administration* (Administração espacial e aeronáutica nacional)

NCC-NUMA - *Non-Cache-Coherent Non-Uniform Memory Access* (Acesso não uniforme à memória sem coerência de cache)

NOW - *Network of workstations* (Rede de estações de trabalho)

NUMA - *Non-Uniform Memory Access* (Acesso a memória não uniforme)

PVM - *Parallel Virtual Machine* (Máquina virtual paralela)

SIMD - *Single Instruction Multiple Data* (Única instrução múltiplos dados)

SISD – *Single Instruction Single Data* (Única instrução único dado)

SO – *Sistema Operacional*

ULA - Unidade de Lógica e Aritmética

UMA - *Uniform Memory Access* (Memória de acesso uniforme)

VSIPL - *Vector Signal Image Processing Library* (Biblioteca de processamento de imagens de sinais vetoriais)

Introdução

A criação de computadores mais velozes não é uma questão de simples conveniência. O processamento mais rápido possibilita a solução de problemas maiores de forma menos demorada, e isso faz toda a diferença na evolução das áreas acadêmicas e industriais, de pesquisa e desenvolvimento (SLOAN, 2004).

Buscando um continuado progresso nessas áreas alguns fabricantes criaram computadores muito mais velozes que os convencionais, esses foram chamados de supercomputadores. Esses sistemas são muito caros, e não seguem padrões de desenvolvimento. Logo, cada modelo de supercomputador pode utilizar diferentes: grupos de instruções, sistemas operacionais e *softwares*. Esses fatos fazem com que muitos programas desenvolvidos para um sistema não funcionem em outro.

A solução desses problemas foi criada com o aumento do poder de processamento das máquinas convencionais. Essas máquinas seguem padrões de desenvolvimento (e.g. arquitetura x86) e são muito baratas quando comparadas a um supercomputador. Em 1994 Donald Becker e Thomas Sterling trabalhavam na *National Aeronautics and Space Administration* (NASA) quando surgiu a idéia de juntar computadores pessoais em uma rede de tecnologia convencional. Como fruto dessa idéia surgiu um supercomputador barato, eficiente e que seguia padrões de desenvolvimento, esse foi o início da história do *cluster Beowulf*, tema abordado nesse documento (GROOP; LUSK; STERLING, 2003).

Esse projeto tem como objetivo construir um *cluster* a partir de tecnologias convencionais utilizando somente *softwares* livres, desenvolver um algoritmo que utilize processamento paralelo provando que esse tipo de processamento pode ser vantajoso e realizar uma análise de desempenho do supercomputador criado.

O termo “tecnologias convencionais” indica que todos os componentes constituintes do *cluster Beowulf* são materiais não específicos para esse fim, e que podem ser adquiridos em lojas de informática convencionais. A aplicação construída deve comprovar a escalabilidade do sistema, ou seja, quantos mais nós forem utilizados na solução de um dado problema

menor será o tempo de execução do mesmo, provando assim que o processamento paralelo pode ser vantajoso. A aplicação desenvolvida nesse projeto tem como objetivo resolver um série de grandes cálculos fatoriais, dividindo a execução dos mesmos de diferentes formas entre os nós de processamento paralelo.

Por meio de uma análise de desempenho do sistema, deve ser mensurado o desempenho do *cluster* de forma numérica (bilhões de operações de ponto flutuante por segundo), analisando as configurações necessárias para que o *Beowulf* tenha um ganho de performance enquanto é aumentada a sua carga de trabalho.

1 Arquiteturas Paralelas

Existem atualmente vários tipos de sistemas com arquitetura paralela. Esses são utilizados para as diversas aplicações desde processadores de texto (*Dual Core, Quad Core)* á simulações nucleares (*Cray, clusters Beowulf)*. A idéia por trás disso não é nova. Em 1920 Vanevar Bush apresentava um computador analógico capaz de resolver equações diferenciais em paralelo. Na década de 40, John von Neumann sugere em um dos seus artigos que para resolver equações diferenciais deve ser utilizada uma grade em que os pontos são atualizados em paralelo (DE ROSE; NAVAUX, 2003).

1.1 Taxonomia de Flynn

Mesmo sendo uma idéia antiga e muito utilizada, ainda hoje não existe um tipo preciso de classificação das arquiteturas paralelas. Entretanto Michael Flynn em 1966 propôs uma taxonomia que é válida atualmente e muito difundida (PATTERSON, 1998)(DE ROSE; NAVAUX, 2003)(SIMA; FOUNTAIN; KACSUK, 1997). O contínuo uso dessa taxonomia pode ser explicada pelo fato da mesma ser simples, fácil de entender e por fornecer uma boa aproximação com a realidade. De fato, quase todos os computadores podem ser classificados por meio dessa taxonomia, com algumas exceções que possuem características de mais de uma classe (PATTERSON, 1998).

O esquema proposto por Flynn relaciona o número de fluxos de instruções executadas em paralelo sobre um número de fluxos de dados. Por exemplo, uma máquina convencional seguindo a arquitetura proposta por Von Neumann executa uma instrução sobre um dado de cada vez, logo existe um fluxo de instruções para um fluxo de dados. Essa arquitetura foi denominada por Flynn de *Single Instruction Single Data* (SISD). Segue abaixo a classificação completa (DE ROSE; NAVAUX, 2003).

Tabela 1.1: Taxonomia de Flynn

	SD (*Single Data*)	**MD** (*Multiple Data*)
SI (*Single Instruction*)	SISD	SIMD
MI (*Multiple Instruction*)	MISD	MIMD

Como se pode ver existem quatro classes de computadores. Essas (as classes) serão descritas a seguir com mais detalhes.

1.1.1 *Single Instruction Single Data* - SISD

Essa é constituída pela maioria dos computadores existentes atualmente, são computadores convencionais onde existe somente um processador e esse consegue tratar um fluxo de dados por vez. Essa pode ser definida como "um único fluxo de instruções atua sobre um único fluxo de dados."(DE ROSE; NAVAUX, 2003, p. 5), como pode ser visto abaixo, na figura 1.1.

Com base na definição vemos que esse tipo de computador não pode realizar múltiplas tarefas ao mesmo tempo não configurando assim uma arquitetura de processamento paralelo. Segue abaixo a topologia dessa arquitetura, onde C representa a unidade de controle, P a unidade central de processamento (UCP) e M a memória principal.

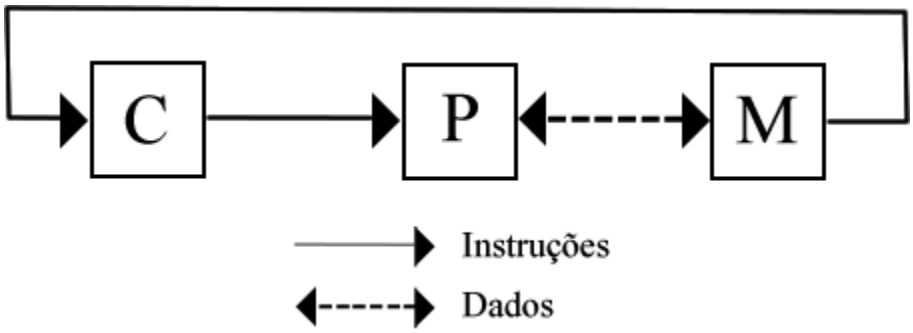

Figura 1.1: Topologia da arquitetura SISD.
Fonte: (DE ROSE; NAVAUX, 2003).

1.1.2 *Multiple Instruction Single Data* - MISD

Representa a ocorrência em paralelo de várias instruções sobre um mesmo fluxo de dados. Diferentes instruções são executadas sobre um único fluxo de dados. Assim diferentes instruções são executadas ao mesmo tempo sobre a mesma posição de memória. Como isso até hoje faz o menor sentido, além de ser impraticável, essa classe é considerada vazia (DE ROSE; NAVAUX, 2003)..A figura 1.2 mostra como essa arquitetura funciona.

Uma máquina MISD pode ser considerada uma arquitetura paralela mesmo sendo sem sentido, já que, a mesma trabalharia várias instruções concorrentemente sobre o mesmo espaço de memória.

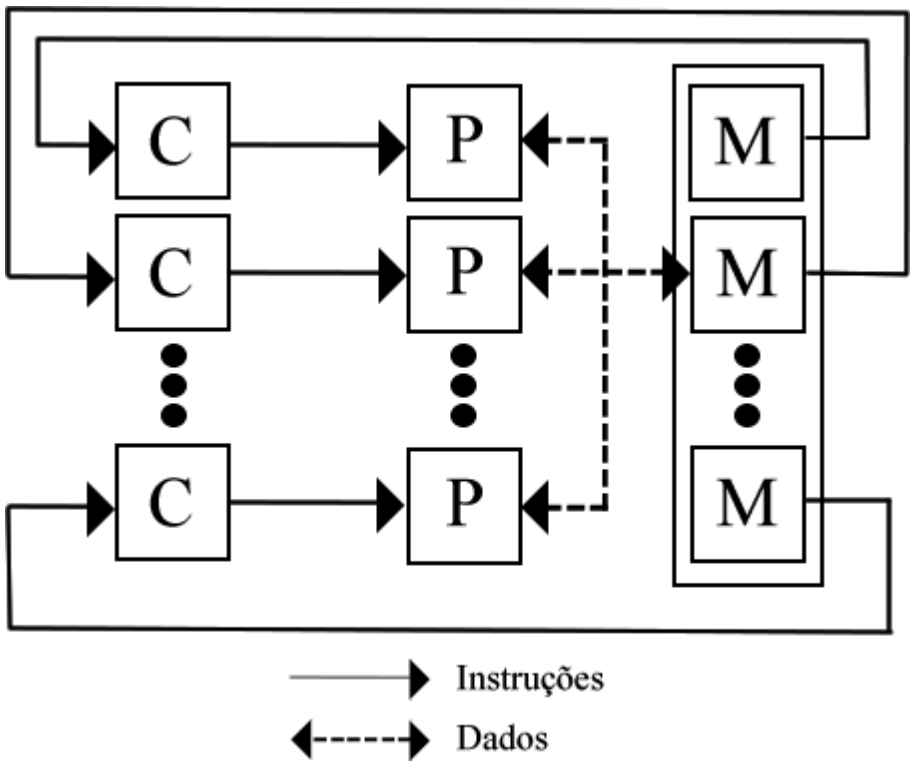

Figura 1.2: Topologia da arquitetura MISD.
Fonte: (DE ROSE; NAVAUX, 2003).

1.1.3 *Single Instruction Multiple Data* - SIMD

O precursor da arquitetura com múltiplos processadores, o *Single Instruction Multiple*

Data, consegue realizar uma única instrução em paralelo sobre múltiplos fluxos de dados. Essas são comumente chamadas de máquinas *array*, cujo nome foi dado, pelo fato de seus processadores serem "normalmente organizados em estruturas regulares como malhas, pois isso facilitava o mapeamento dos dados de aplicações numéricas como vetores e matrizes (também chamados de array – tabela de dados – daí o nome desse tipo de máquina)"(DE ROSE; NAVAUX, 2003, p. 68). Podemos visualizar a topologia da arquitetura paralela SIMD na figura 1.3.

As unidades de processamento utilizados pelo SIMD, na maioria dos casos, não eram processadores no sentido completo da palavra, eram EP's (elementos processadores), ou seja são formados por apenas uma ULA(unidade de lógica e aritmética) , memória local e uma conexão para comunicação de dados com o processador vizinho (DE ROSE; NAVAUX, 2003).

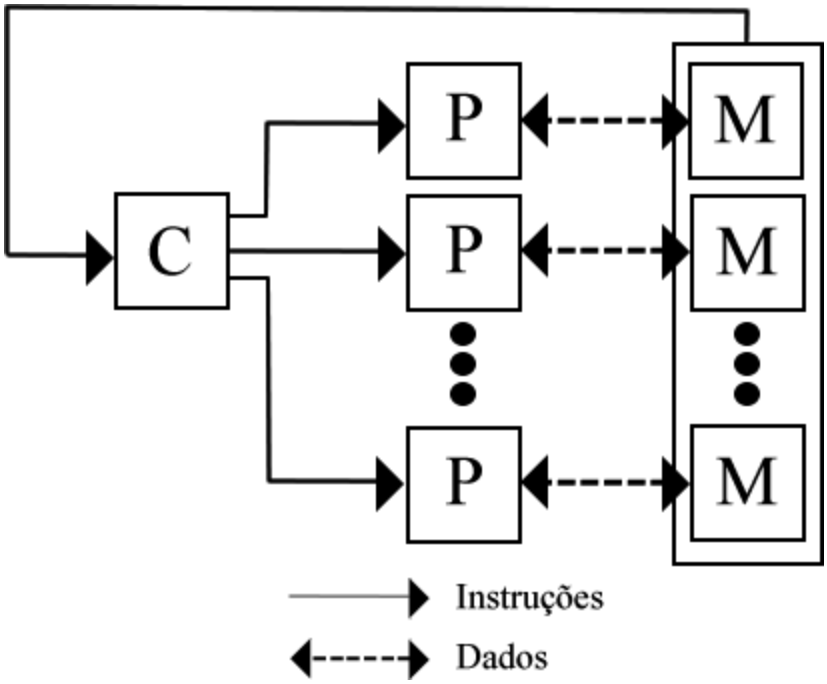

Figura 1.3: Topologia da arquitetura SIMD
Fonte: (DE ROSE; NAVAUX, 2003).

Os principais exemplos de sistemas SIMD são aqueles que utilizam processamento vetorial e processamento matricial. Exemplos clássicos desses tipos são o Cray-1 de 1976 e ILLIAC IV utilizado pela NASA (TANENBAUM, 2001 , p.333). Já que essa classe não

representa o foco do nosso trabalho não entraremos em mais detalhes aqui.

1.1.4 *Multiple Data Multiple Instruction* - MIMD

Multiple Data Multiple Instruction é caracterizado por máquinas que possuem um fluxo de dados e um fluxo de instruções para cada unidade de processamento. Esse tipo de sistema ficou popular com o avanço da tecnologia dos circuitos integrados o que tornou relativamente fácil e barato conectar vários processadores em um sistema. Com isso durante a década de 80 o número de elementos utilizados, nessa arquitetura, aumentou de dezenas para centenas e atualmente são encontrados *clusters* com milhares de unidades centrais de processamento (SIMA; FOUNTAIN; KACSUK, 1997). A figura 1.4 mostra a representação esquemática da topologia da classe MIMD.

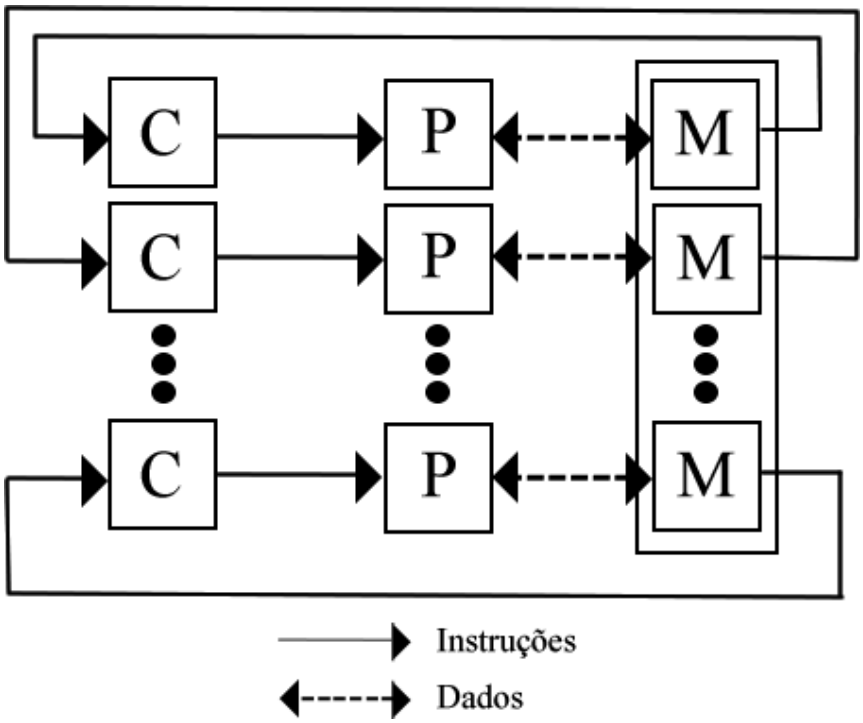

Figura 1.4: Topologia da arquitetura MIMD.
Fonte: (DE ROSE; NAVAUX, 2003).

Os sistemas do tipo MIMD são classificados pelo compartilhamento de memória. Existindo sistemas de memória compartilhada, também conhecidos como multiprocessadores, e sistemas de memória distribuída, também chamados de multicomputadores (DE ROSE e NAVAUX, 2003).

1.1.4.1 Multiprocessador

Nesse tipo de arquitetura, vários processadores dividem a memória do sistema seja essa uma única e centralizada memória física ou constituída de vários blocos distribuídos. Em qualquer um dos casos a memória é mapeada como uma só e é disponibilizada de forma global para todos os processadores (SIMA; FOUNTAIN; KACSUK, 1997).

A comunicação entre os processadores é realizada "através da memória compartilhada de forma bastante eficiente por operações do tipo *load* e *store*"(DE ROSE; NAVAUX , 2003 , p. 69). Por ter somente o elemento processador replicado, essa arquitetura leva o nome de múltiplos processadores (DE ROSE; NAVAUX, 2003).

Exemplos de multiprocessadores são os computadores ou servidores com mais de um processador (*Dual Core*, *Quad Core*, *Cell BE)*, entre outros.

Os multiprocessadores podem ser classificados em relação ao tipo de acesso à memória em três tipos principais:

- UMA
- NUMA
- COMA

1.1.4.1.1.*Uniform Memory Access* - UMA

Um sistema é classificado como *Uniform Memory Access* ou acesso uniforme a memória quando, o tempo de acesso à memória de todos os processadores é uniforme. Essa arquitetura pode utilizar memória centralizado ou distribuída, desde que, seja mantido igual o tempo de acesso a memória, para todas unidades de processamento (DE ROSE; NAVAUX, 2003).

Para evitar o gargalo de muitos acessos à memória principal, essas máquinas geralmente utilizam memórias cache nos processadores, fato que ameniza a diferença de velocidade entre o processador e a memória principal(DE ROSE; NAVAUX, 2003). Essa característica gera um problema de coerência de cache, pois toda vez que o processador precisa de um dado, ele verifica se o possui em seu cache, caso não, ele acessa a memória principal. Algumas vezes o

cache tem o dado necessário, no entanto, este não está atualizado, já existindo assim, uma versão mais recente.

1.1.4.1.2.*Non-uniform memory access* - NUMA

Non-uniform memory access são máquinas onde a memória é distribuída e como o nome já diz, os processadores não tem um tempo uniforme de acesso a memória. Esse também possui problemas de coerência de cache e o tratamento desse problema diferencia os vários tipos de multiprocessadores NUMA existentes. Esses são, de acordo com De Rose e Navaux(2003 , p. 71), classificados em:

- *Non-Cache-Coherent Non-Uniform Memory Access* (NCC-NUMA) ou acesso não uniforme à memória sem coerência de cache
- *Cache-Coherent Non-Uniform Memory Access* (CC-NUMA) ou acesso não uniforme à memória com coerência de cache

1.1.4.1.3.*Cache-only memory architecture* - COMA

Cache-only memory architecture (arquitetura com somente memória cache), implementa para cada processador uma grande memória cache, sendo o conjunto dessas a única memória do sistema, não possuindo assim, ao contrário dos sistemas UMA ou NUMA, uma memória principal. Nesse tipo de máquina, a alocação de memória é feita através da demanda dos processadores, de acordo com o sistema de coerência de cache utilizado (SIMA; FOUNTAIN; KACSUK, 1997).

1.1.4.2 Multicomputador

Sima, Fountain e Kacsuk (1997, p.536), defendem o conceito de que o multicomputador é um sistema com arquitetura MIMD onde ocorre a replicação do par memória/processador.

A diferença entre esses sistemas e os multiprocessadores NUMA ou UMA com memória distribuída é relativa, como já dito anteriormente, ao tratamento da memória principal. Ao contrário do multiprocessador, onde cada processador pode acessar qualquer parte da

memória, os processadores de um multicomputador possuem memória própria, que é privada, logo, nenhum outro elemento processador pode acessa-la diretamente.

Os multicomputadores ao contrário dos multiprocessadores, não possuem uma memória comum entre eles, a comunicação entre os processadores tem que ser feita através da troca de mensagens por meio da rede de interconexão. Este fato faz com que essas máquinas sejam também chamadas de sistemas de troca de mensagem (DE ROSE; NAVAUX, 2003). Assim cada EP precisa de um meio de comunicação, uma interface de rede, conectada a um *switch*, roteador, *hub,* que realizam todas as interconexões da rede do sistema. Vimos que nesse caso ocorre a replicação de toda a arquitetura convencional utilizada, e não somente do elemento processador como acontece nos multiprocessadores. A partir da forma de replicação surge o nome, múltiplos computadores (DE ROSE; NAVAUX, 2003). Sendo que em um multicomputador cada computador é normalmente referenciado como um nó.

Existem várias categorias de multicomputadores, alguns exemplos são:

- *Massively Parallel Processor*s (MPPs) ou Processadores Massivamente Paralelas.
- *Network of workstations* (NOW), também conhecidos como *Cluster of Workstations* (COW).
- *Clusters* Beowulf

1.1.4.2.1. Cluster Beowulf

O nome *Beowulf* foi escolhido com base na literatura inglesa, onde o herói épico *Beowulf* era descrito como tendo a força de muitos homens. Sendo assim o nome condiz com o *cluster* em si, pois o mesmo possui a força de muitos outros computadores que juntos formam um grande e forte sistema, o *cluster Beowulf.*(SLOAN, 2004).

1.1.4.2.1.1.História

A história do *cluster Beowulf* começou nos Estados Unidos no centro espacial Goddard da NASA em 1994, onde Thomas Sterling e Donald Becker construíram ao longo do projeto Beowulf, o primeiro supercomputador utilizando *Mass Market Commodity-Of-The-Shelf*

(M²COTS) ou seja, em uma tradução literal, componentes disponíveis ao mercado consumidor comum tirados da prateleira (BROWN, 2004). Em outras palavras, são equipamentos convencionais de uso doméstico, produtos que não são específicos para esse fim. Assim, o primeiro Beowulf construído por Becker e Sterling utilizava 16 nós Intel 80486 100MHz, cada um utilizando duas placas de rede 10Mbps para promover uma melhor comunicação entre os processadores. Este também utilizava uma das primeiras versões do Linux, com um *driver* de rede desenvolvido por Becker para esse fim e o PVM (pacote para programação paralela de computadores). Tudo muito simples com o custo muito baixo (GROOP; LUSK; STERLING, 2003) .

Becker durante esse trabalho desenvolveu a maioria dos *drivers* de rede presentes atualmente no Linux, trabalho que foi vital para o desenvolvimento do projeto. Brown (2004, p;29) defende o uso desse sistema operacional (S.O.), pelo fato do mesmo possuir código aberto, e assim caso seja necessário algum conserto ou melhoria, durante seu uso, seu código fonte pode ser modificado e recompilado, gerando uma nova versão do sistema que será mais adequada para as necessidades do usuário, que fará o seu *cluster* funcionar no jeito que ele gostaria.

A relativa facilidade de montagem e o baixo custo desse tipo de *cluster,* foram fatores essenciais para a difusão desse projeto por toda a NASA, por entidades acadêmicas e comunidades de pesquisa, sendo difundido posteriormente na esfera global.

1.1.4.2.1.2.Características do *cluster Beowulf*

Existem algumas características que atuam como fatores predominantes na diferenciação de um *Beowulf* dos outros multicomputadores. Brown (2004) defende uma série de características que segundo ele especificam e diferenciam esse sistema, são as seguintes:

- Os nós devem ser de uso dedicado ao *Beowulf* e não devem servir a nenhum outro propósito. No COW essas máquinas são estações de trabalho, logo servem outro propósito, não sendo dedicadas ao uso do *cluster*.

- A rede utilizada pelos nós é de uso exclusivo do *Beowulf* e não deve servir a nenhum outro propósito. No COW a rede é usada pelas estações de trabalho para outros fins.

- Os nós utilizados tem que ser formados a partir de M²COTS, para garantir assim que seu preço seja inexpressivo. Essa característica é essencial nesse tipo de *cluster*, pois o diferencia do MPP, já que esse é construído por empresas especializadas, usando tecnologia especializada para este fim, tendo assim um preço muito alto comparado ao do *Beowulf*.

- A rede utilizada para comunicação do *cluster* (*switches*, cabeamento e interfaces de rede) devem utilizar tecnologia M²COTS. Essa característica também diferencia o *Beowulf* do MPP, que utiliza tecnologia especializada para esse fim.

- Todos os nós devem utilizar *software* de código aberto.

- O *cluster* resultante da combinação dessas características deve ser utilizado para Computação de Alto Desempenho (CAD), também chamado de computação paralela.

Brown (2004) ressalta, que essas são características essenciais para esse tipo de *cluster*. Logo se o sistema em questão não possuir uma dessas, ele pode ser considerado um sistema de computação paralela de qualquer tipo menos um *cluster beowulf*.

Com base nas características apontadas acima, esse *cluster* tem sua características essenciais guiadas pelo preço de construção, a comodidade da compra dos recursos utilizados, o custo zero em relação ao *software* e a maleabilidade do mesmo, sempre buscando uma ótima relação entre desempenho e custo.

Essas características moldam um novo tipo de supercomputador, que ao contrário dos convencionais, que são construídos por poucas empresas e que muitas vezes não podem ser exportados para países menos desenvolvidos por questões políticas, o *beowulf* pode ser construído por qualquer nação com materiais disponíveis no mercado consumidor comum. Tendo ainda um preço bem menor, podendo alcançar ou até ultrapassar o poderio computacional dos supercomputadores clássicos, tornando possível assim, o desenvolvimento de áreas que demandam esses recursos em todos os países

1.1.4.2.1.3.Aplicações do *cluster Beowulf*

Antes de vermos onde esse tipo de *cluster* será empregado, vamos observar o porque da utilização do mesmo. As principais razões de utilizar um multicomputador *beowulf* são o alto desempenho e a tolerância a falhas. Uma aplicação normalmente precisa de um computador de alto desempenho quando:

- Existem limitações temporais sobre a execução de uma aplicação. Como na previsão do tempo, nos dados produzidos por um experimento, onde esses devem ser processados assim que forem gerados (GROOP; LUSK; STERLING, 2003).

- Uma grande vazão de dados. Por ser constituído de vários nós, o sistema pode processar várias entradas de forma simultânea, gerando assim várias saídas. Por exemplo, o Google utiliza 15.000 nós para realizar um serviço de procura na *web* de alto desempenho (GROOP; LUSK; STERLING, 2003).

- Quantidade de memória. Algumas das aplicações mais pesadas requerem uma grande massa de dados. Esse *cluster* pode fornecer, de forma eficiente, até terabytes (10^{12} bytes) de memória para o programa em execução.(GROOP; LUSK; STERLING, 2003).

Porém, existem algumas ressalvas na utilização desse sistema. Nem todos os problemas podem ser paralelizados de forma vantajosa (ou seja, o processamento será realizado mais rapidamente nesse do que em uma máquina sozinha), sendo assim essa abordagem não pode ser aplicada a todas as tarefas comuns. Além disso, um programa não será executado mais rapidamente em um *cluster* se o mesmo não for programado de forma a tirar vantagem do processamento em paralelo, isso claro, se o programa puder ser paralelizado. Um programa a ser executado em paralelo possui duas partes. Uma parte do código que pode ser paralelizado e outra que tem que ser executada em série. Quanto maior for a parte do código a ser executado em paralelo, mais eficiente será a sua execução em um ambiente de processamento paralelo. Essa relação entre processamento paralelo e em série durante a execução de uma aplicação, será vista com mais enfoque adiante, na Lei de Amdahl. Entretanto, de uma forma geral, problemas grandes que demorariam muito tempo para serem resolvidos em um computador tradicional, podem ser muito provavelmente reescritos para serem executados de

forma proveitosa em um *Beowulf* (BROWN, 2004).

Levando as características acima em consideração o *beowulf* pode ser usado para vários fins, entre eles:

- Previsões do tempo;
- Simulações (nucleares, aerodinâmicas e etc);
- Modelos de comportamento do mercado financeiro;
- Servidores de internet (de vídeo, jogos, serviços, entre outros).

1.1.4.2.1.4.Desempenho no *cluster Beowulf*

O desempenho tem como principal objetivo fazer com que um programa execute mais rapidamente em uma máquina paralela do que em uma uniprocessada. Isso respeitando a relação custo/benefício do sistema, não encarecendo demais o sistema para obter um desempenho um pouco superior.(TANENBAUM, 2001).

Tanenbaum (2001) ainda expõe que na perspectiva do *hardware* os fatores de maior interesse são a velocidade do processador, dos dispositivos de entrada e saída e das redes de interconexão. No entanto em um sistema multiprocessado onde ocorre a replicação de computadores, as velocidades de processamento e dos dispositivos de E/S serão iguais, em cada nó, restando assim as redes de interconexão como fator chave de desempenho. Tipicamente os principais limitadores de desempenho dessa são: a latência da rede de comunicação e a taxa de transmissão máxima. A análise de desempenho feita nesse trabalho tem como foco o desempenho do *software*, a ser executado, logo não iremos entrar em detalhes na análise de desempenho do hardware utilizado.

1.2 Lei de Amdahl

Voltando à proposta do trabalho, vamos analisar o desempenho que um determinado programa pode ter em sistema computacional de alto desempenho. Segundo Brown (2004), na

década de 60 quando os computadores eram mais lentos, a IBM, que era um dos principais fabricantes da época, queria utilizar o processamento paralelo para aumentar a velocidade dos computadores. No entanto, foram apontadas uma série restrições quanto ao ganho de velocidade que se poderia conseguir por meio do uso dos computadores em paralelo. Essas restrições formaram a lei de Amdahl, sendo Gene Amdahl, um funcionário da IBM, que na época trabalhou na solução desse problema.

A lei em questão nos informa o ganho máximo que um programa pode obter com a execução em paralelo. Para isso ela relaciona a parte do programa que tem de ser executada em série e parte que pode ser realizada em paralelo. Quanto maior foi o número de processadores adicionados ao sistema menor será o tempo de processamento paralelo. Essa colocação é feita sob um regime ideal, onde a parte executada em paralelo possa ser dividida em N partes para tirar proveito de N processadores presentes no sistema. Logo, podemos acelerar somente o processamento da parte que é executada em paralelo, assim o tempo necessário para a execução do trecho em série será o fator limitante da performance do programa. A representação algébrica da lei é

$$Ganho = \frac{100}{S + \frac{P}{N}} \tag{1.1}$$

onde S é a porcentagem do trabalho que será realizada em série, P a porcentagem do trabalho a ser realizado em paralelo e N o número de nós que serão utilizados. Para exemplificar o uso da fórmula vamos aplicá-la sobre um programa onde uma análise de seu código revelou que 10% dele será executado em série e 90% pode será executado em paralelo. Ao aumentarmos o número de processadores, a razão P/N diminui tendendo a zero, assim, em um caso ideal, onde todo o código paralelizado pode ser distribuído para um número infinito de máquinas, o ganho fica limitado somente pela parcela executada em série, tendo dessa forma, um ganho igual 10 (ou seja 10 vezes mais rápido) em relação ao processamento em um nó sozinho (SLOAN, 2004).

A lei de Amdahl é umas das métricas mais difundidas e úteis para descrever o desempenho do processamento paralelo. No entanto, existem muitas outras que podem ser utilizadas para o mesmo fim (SLOAN, 2004).

2. *Softwares* utilizados

2.1 *Message Passing Interface* e *Parallel Virtual Machine*

Como explicado anteriormente, ao contrário dos multiprocessadores, os multicomputadores não possuem acesso à memória dos outros nós, não possuem também uma memória compartilhada para a comunicação. Logo tem que efetuar a sua comunicação por meio de troca de mensagens. Os dois principais pacotes de comunicação para efetuar a computação de alto desempenho são o PVM (Parallel Virtual Machine) e o MPI (Message Passing Interface). Nesse trabalho utilizaremos somente o MPI.

O PVM como contam Groop, Lusk e Sterling (2003), teve sua primeira versão pronta para uso no começo da década de 90. Esse foi fruto do crescimento de um projeto computacional envolvendo o laboratório nacional Oak Ridge, a universidade do Tennessee e a universidade de Emory. Os objetivos principais desse projeto eram investigar problemas da utilização de máquinas heterogêneas no processamento distribuído e desenvolver soluções para os mesmos. O PVM foi uma das soluções que surgiram. Ele foi desenvolvido para permitir a combinação de diferentes sistemas operacionais, representações de dados, arquiteturas, linguagens e redes, todos trabalhando juntos sobre um único problema computacional.

A idéia básica do PVM era criar um *software* simples, que poderia ser carregado em qualquer coleção de computadores e que faria com que essa coleção parecesse um grande e único computador paralelo com memória distribuída. Atualmente essa prática é chamada de *Grid*. A contribuição real do PVM à ciência e a computação não foi na área dos supercomputadores. Esse pacote de alta confiança e facilidade de uso foi usado por milhares de usuários como meio de conexão entre COW'S e outros tipos de *clusters*, transformando eles em computadores paralelos.

O PVM continua popular, particularmente para aplicações que utilizam tolerância a falhas. O pacote de 1,5MB do PVM pode emular um computador de propósito geral, dinâmico, em um ambiente com máquinas heterogêneas em um grupo de computadores ligados por uma rede. Essa rede pode ser a internet (*Grid*) ou uma rede local dedicada

(*Beowulf*). Esse pacote ainda pode ser utilizado para combinar *clusters beowulf* na forma de uma grade computacional (*Grid*).

Já o MPI, de acordo com Groop, Lusk e Sterling (2003), foi criado em 1994 pelo fórum MPI, um grupo de vendedores de computadores paralelos, cientistas da computação, e usuários, que se juntaram cooperativamente e trabalharam em um padrão. Essa primeira fase de reuniões resultou na versão chamada de MPI-1 que foi liberada no mesmo ano. Uma vez implementado e com grande uso, uma segunda série de reuniões resultou em um grupo de extensões, referenciados como MPI-2. O nome MPI referencia os dois padrões MPI-1 e MPI-2. Os documentos referentes a esses padrões podem ser encontrados no site www.mpi-forum.org .

O objetivo do fórum MPI era criar uma biblioteca flexível que pudesse ser implementada eficientemente em grandes computadores e prover uma ferramenta de ataque para os mais difíceis problemas da computação paralela.

Neste trabalho será utilizado o MPICH2 desenvolvido inicialmente por William Groop e Ewing Lusk.(SLOAN, 2004).

2.2 Sistema Operacional GNU/Linux

Como foi citado anteriormente, uma das principais características do *cluster Beowulf* é o uso de *softwares* de código aberto. Assim ao longo desse projeto utilizaremos uma distribuição do sistema operacional GNU/Linux.

O projeto GNU ou sistema operacional GNU foi desenvolvido por Richard Stallman. Onde a palavra GNU seria referente ao mamífero proveniente do continente africano. Richard acreditava no futuro do *software* livre e por isso, o GNU também representa um trocadilho “GNU'S Not Unix”, pois o Unix nessa época era um sistema operacional proprietário e o sistema operacional de Stallman é livre. Ele começou o desenvolvimento de seu S.O. pelo compilador, o famoso GCC (GNU C Compiler), entre outros aplicativos compatíveis com o Unix (MOTA FILHO, 2006). O GCC é utilizado até hoje, sendo aplicado no desenvolvimento desse projeto para compilar programas na linguagem C.

O Linux foi criado por Linus Torvalds, baseando em parte no Minix, sistema operacional desenvolvido por Andrew Tanenbaum. O nome Linux é uma mistura do nome de seu criador Linus com Unix. Linus começou o seu desenvolvimento pelo kernel e utilizou o padrão POSIX para garantir que o seu sistema fosse compatível com o Unix. Na época o projeto GNU ainda não tinha terminado o desenvolvimento de seu kernel (HURD), porém com o desenvolvimento do Linux e a sua compatibilidade com os aplicativos do projeto GNU, isso não era mais necessário. Nascia então o sistema operacional GNU/Linux (MOTA FILHO, 2006).

2.2.1 Distribuição Debian

Uma distribuição Linux é definida como "a união do *kernel* com vários programas compatíveis com ele (MOTA FILHO, 2006, p. 61)". Sendo um sistema livre qualquer pessoa pode fazer uso do *kernel* e de outros aplicativos para assim criar a sua própria distribuição.

A distribuição Debian foi criada em 1993 por Ian Murdock. Esse foi criado com o objetivo de ser totalmente aberto e sem fins comerciais. Atualmente o Debian é atualizado por uma rede organizada de quase 1500 programadores e serve de base para muitas outras distribuições como Ubuntu, Kurumin, entre outras. Essa distribuição foi escolhida para servir de base para o trabalho, pois, é uma distribuição que existe há quinze anos, sendo assim uma das mais antigas, por seu projeto continuar sendo atualizado e por existirem muitos *clusters Beowulf* que utilizam a mesma distribuição, fato que facilita a utilização do Debian para esse fim. Utilizaremos aqui a versão mais nova do Debian a versão 4 codinome "Etch", liberada em 2007 (MOTA FILHO, 2006).

2.2.1.1 MBR

No Debian, o *Master Boot Record* (MBR) corresponde a um espaço de 512 bytes, existente no começo do HD, esse acomoda gerenciadores de *boot*, como o Linux Loader (LILO) e o *Grand Unified Bootloader* (GRUB), e parte do esquema de particionamento do disco (HDs e pendrives) (MOTA FILHO, 2006). O LILO e o GRUB, realizam, o gerenciamento do *boot* (inicialização do sistema), e permitem a coexistência de vários sistemas operacionais em uma mesma máquina. A interface do GRUB pode ser vista na

inicialização da máquina como um menu onde seus itens correspondem a opções de sistemas operacionais. Após a escolha do mesmo a máquina é inicializada (MOTA FILHO, 2006).

2.3 Pacotes utilizados

Para utilizarmos um grupo de computadores como um *cluster Beowulf* precisamos que uma série de pacotes sejam instalados e configurados de forma correta. Segue abaixo uma descrição básica de cada programa utilizado no desenvolvimento.

- XFCE4 – interface gráfica leve
- gcc – compilador C
- g++ - compilador C++
- g77 – compilador Fortran (utilizado na compilação do HPL)
- make – utilitário para compilação de programas
- SSH (Servidor e cliente) – comunicações seguras, *login* remoto, transferência de arquivos, entre outros serviços
- Partimage – programa utilizado para clonagem dos nós
- PYTHON 2.2 ou superior – linguagem de programação. O MPICH2 e XFCE4 dependem dessa, para funcionar corretamente.
- MPICH2 – versão 2 do pacote MPICH de passagem de mensagens. Utilizado no processamento paralelo
- ATLAS – programa que possibilita a criação da biblioteca BLAS (*Basic Linear Algebra Software*), de forma otimizada para a arquitetura sendo utilizada. Essa biblioteca é um dos requisitos para o funcionamento do HPL.
- HPL – programa utilizado para realizar testes de desempenho em sistemas de memória distribuída

3. Estrutura Física

3.1 Planejamento Elétrico

Nesse projeto não foi necessário o estudo elétrico do *cluster,* pois o mesmo contou com a infra-estrutura do laboratório de *hardware* 8006 do Uniceub que possui um estabilizador com potência de 5kVA, voltagem de entrada de 220 V, voltagem de saída 110V, fornecendo assim energia estabilizada para todo o laboratório. Nessas condições a infra-estrutura do laboratório pode manter mais de 30 computadores ligados ao mesmo tempo, contando com os computadores nativos do laboratório de *hardware*.

3.2 Ventilação

A ventilação da sala é feita por dois aparelhos de condicionamento de ar, deixando o ar sempre frio e evitando o aumento da temperatura dos computadores utilizados.

3.3 Construção do *cluster*

Para a construção do *cluster* foram utilizados 8 computadores com a seguinte configuração:

- Principais características da placa mãe 3748LMRT - AMPTRON
    - O primeiro barramento de suporte ao processador (*Slot-1*)
        - Suporta os processadores Pentium III com taxa de *clock* em 450MHz à 550MHz, Pentium II 233MHz até 450MHz e SEPP Celeron 266MHz até 433MHz.
        - Suporte ao *front side bus* de 66MHz ou 100MHz.
    - Segundo barramento de processamento (*Socket-370*)

        - Suporta no máximo um PPGA Celeron com taxas de clock de 300MHz até 466MHz.
        - Suporte ao *front side bus* de 66MHz
    - Somente um dos barramentos de processamento pode ser utilizado
    - 3 barramentos para memória SDRAM 168 pinos, tendo no máximo 3 pentes de 256 MB (768 MB de memória total)
    - 1 barramento 32-bit PCI
    - 1 barramento 8/16-bit ISA
    - Canais primário e secundário de PCI IDE
    - Suporta fonte de energia padrão AT ou ATX
    - Placa de vídeo on-board AGP
        - 64-bit
        - AGP Ver. 1 operando a 66/133MHz
        - Memória compartilhada máxima de 8 MB
    - Uma porta serial
    - Uma porta paralela para ECP e EPP
    - Dimensões *Baby* AT (22cm x 22cm)

Conforme as características acima foram usados os seguintes componentes compatíveis:

- Processador Pentium III 500 Mhz
- 256 MB de memória princiapal SDRAM 100MHz (PC-100)
- Placa de rede 3com *off-board* padrão IEEE 802.3u (*FastEthernet* 100Mbps)

- Placa de vídeo *on-board* descrita acima
- Leitores de CD-ROM (52x) e de disquete (1.44 MB)
- Disco rígidos ATA de tamanhos variando entre 6 GB e 40 GB
- Fonte de energia com potência de pico 250W (aprox. 144W RMS)

Esses foram interconectados por um *switch* 3com *Superstack* II de 24 portas 10/100Mbps e por cabos de par trançado sem blindagem com conectores RJ45. O sistema utiliza um monitor no nó principal, com o intuito de utilizar o *cluster* de dentro do laboratório e realizar a sua manutenção, por meio dessa máquina. O acesso a *internet* é feito por meio de um cabo de par trançado sem blindagem com conector RJ45 configurado como *cross-over* que conecta o *switch* utilizado à um ponto de rede do laboratório (ligação *switch* à *switch).*

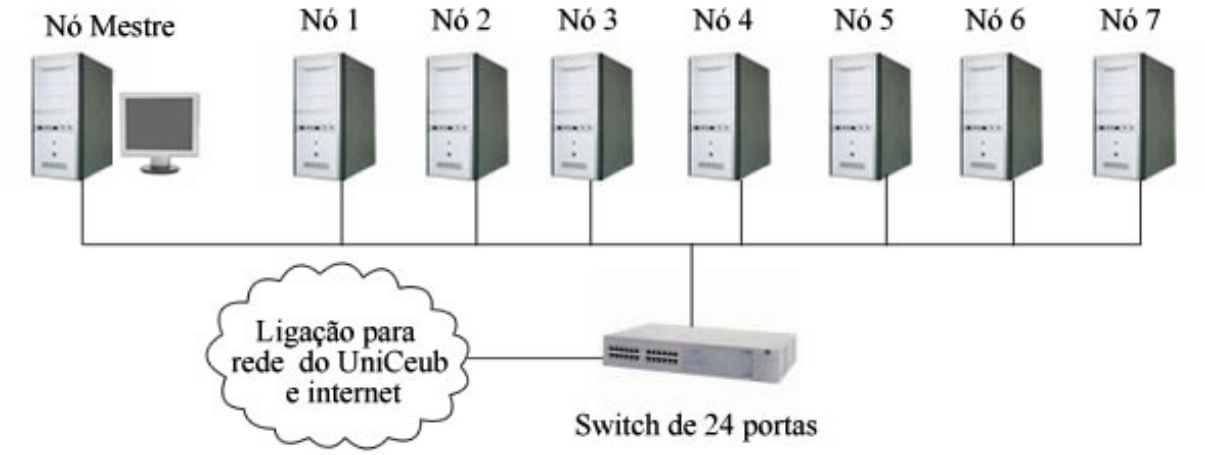

Figura 3.1: Topologia do cluster

Todos os nós foram dispostos em uma bancada do laboratório próximo do fornecimento de energia já estabilizado em 110V.

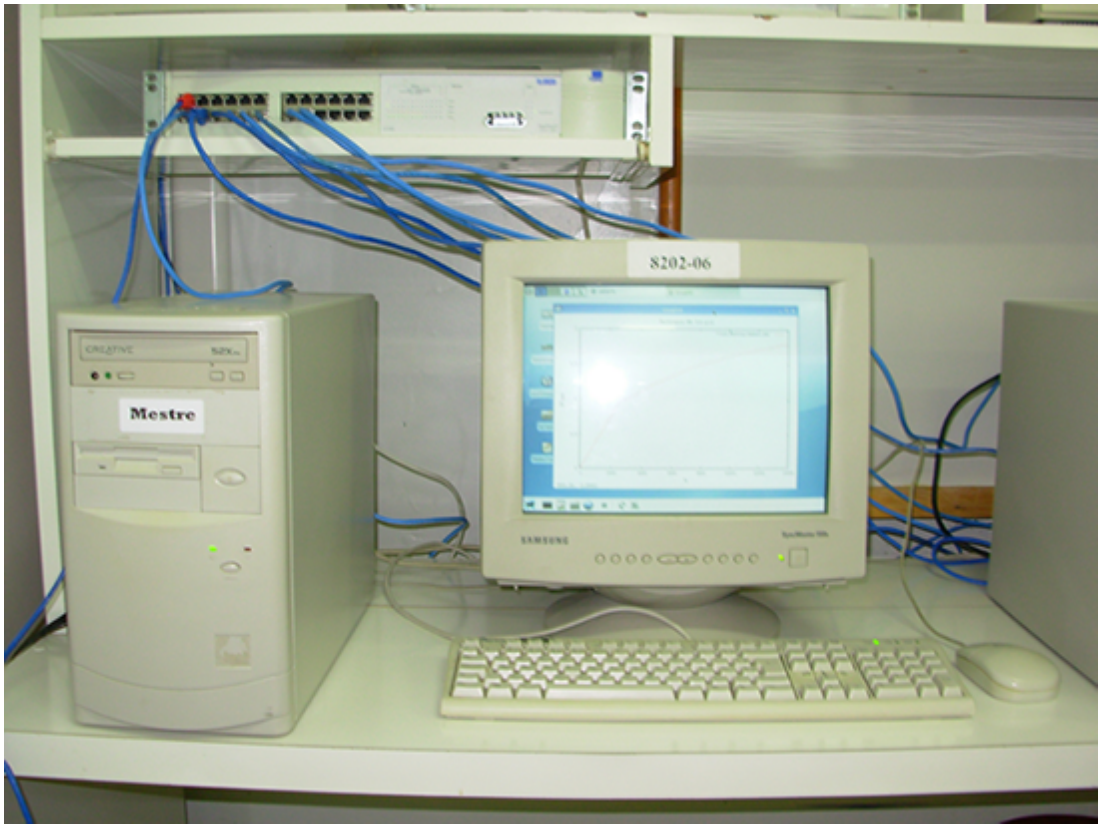

Figura 3.2: Visão frontal do nó principal (mestre)

Figura 3.3: Visão lateral do cluster construído

3.4 Instalação do sistema

Antes de começar a instalação do *cluster* verifique se cada máquina funciona corretamente, formate todos os discos e exclua qualquer partição existente. Conecte todas as máquinas ao *switch*.

3.5 Instalando o Debian

Antes de instalar o Debian devemos verificar se a máquina em questão atende seus pré-requisitos básicos.

Segundo site (www.debian.org) o Debian Etch, com Desktop(GNOME ou KDE) requer no mínimo um processador com o clock de 1GHz, com somente o uso do SHELL ainda não foi encontrado no guia da instalação. No entanto Mota (2006), afirma que o Debian Etch no modo SHELL (ou seja, sem modo gráfico) requer no mínimo um Pentium 100 sendo ideal um Pentium II. Além disso, o computador deve possuir um leitor de CD-ROM e conexão a internet (sendo o segundo somente necessário para o tipo de instalação netInstall, que foi utilizada nesse projeto).

Tabela 3.1:Requisitos de memória e disco rígido da distibuição Debian 4.0 "Etch":

Tipo de Instalação	RAM (mínimo)	RAM (recomendado)	Disco Rígido
Sem desktop	64 megabytes	256 megabytes	1 gigabyte
Com Desktop (GNOME ou KDE)	64 megabytes	512 megabytes	5 gigabyte

Fonte: http://www.debian.org/releases/stable/i386/ch03s04.html.pt , acesso: 8 maio, 2008.

Para ver as informações completas sobre os requisitos de *hardware* da arquitetura i386, ou caso tenha alguma dúvida sobre o procedimento de instalação, acesse o guia de instalação Debian Etch disponível em http://www.debian.org/releases/stable/i386/index.html.pt (acessado: 8 maio, 2008).

Levando em consideração os requisitos mínimos de instalação, podemos instalar o Debian utilizando somente o modo SHELL, já que para a instalação do modo gráfico nosso poder de processamento não seria suficiente. No entanto é interessante disponibilizarmos de uma interface gráfica no nó principal, para isso, vamos instalar no computador em questão uma interface gráfica mais leve chamada de XFCE4.

Existem várias formas de instalar o Debian, nesse projeto não utilizamos muitos pacotes e também não poderíamos utilizar por limitações do *hardware* utilizado. Caso o sistema ficasse muito pesado o processamento paralelo realizado nele perderia desempenho. Levando isso em consideração foi baixado o instalador netInstall do Debian para arquitetura i386, com somente 159 MB de tamanho, esse foi gravado em um CD-R(W) convencional para a sua instalação nos computadores. A arquitetura dos computadores utilizados nesse trabalho é a i686 o que a princípio seria um problema levando em consideração a arquitetura do arquivo de instalação. No entanto Mota (2006), afirma que podemos utilizar pacotes com arquiteturas menores em computadores com arquiteturas maiores, o que é nosso caso (i386 em i686), no entanto o contrário não é possível.

Caso tenha alguma dúvida no processo de instalação do sistema operacional GNU/Linux distribuição Debian, consulte o apêndice A – Processo de instalação do Debian.

3.5.1 Configurando o Debian

Entre no Debian no modo multi-usuário (iniciação normal), e entre como *root* no sistema, já que iremos configurar alguns arquivos de acesso restrito do Debian. Para isso use o *login*: *root* e o *password* definido durante a instalação para esse usuário.

Nosso projeto teve como intuito deixar o mais simples possível à construção de um cluster *beowulf*. Logo, a rede utilizada terá uma configuração estática diferentemente dos sistemas que utilizam Dynamic Host Configuration Protocol (DHCP).

Para realizar essa comunicação com IP'S estáticos, todos nós devem possuir o arquivo /etc/hosts, que lista todas as máquinas da rede com IP, nome com domínio e nome da máquina, sempre atualizados. Após a instalação o arquivo mostrará somente o IP, o nome de sua máquina com o domínio, e o nome de sua máquina sem o domínio, respectivamente. O arquivo *hosts* deve ser semelhante ao arquivo mostrado abaixo:

```
127.0.0.1       localhost
172.18.86.115   mestre.cluster.uniceub   mestre

# The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
```

```
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts
```

O primeiro endereço que aparece é o endereço de *loopback* da placa de rede e o segundo corresponde aos dados da sua máquina. Não se preocupe com as outras informações contidas nesse arquivo, pois essas só serão utilizadas com máquinas que fazem uso do Ipv6, que não será utilizado aqui. Complete o arquivo acima, com os nomes e IP's dos outros computadores do seu *cluster*. Nosso arquivo completo ficou assim:

```
127.0.0.1       localhost
172.18.86.115   mestre.cluster.uniceub      mestre
172.18.86.116   no1.cluster.uniceub         no1
172.18.86.117   no2.cluster.uniceub         no2
172.18.86.118   no3.cluster.uniceub         no3
172.18.86.119   no4.cluster.uniceub         no4
172.18.86.120   no5.cluster.uniceub         no5
172.18.86.121   no6.cluster.uniceub         no6
172.18.86.122   no7.cluster.uniceub         no7

# The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts
```

Com esse arquivo configurado já podemos utilizar, o nome da máquina para referenciá-la, ao invés do endereço de IP.

Tendo finalizado a etapa anterior, vamos realizar a atualização do sistema, testando assim o APT, ferramenta responsável pelo gerenciamento de pacotes on-line do Debian, e a conexão com a internet. Primeiro precisamos configurar o arquivo que lista os repositórios do APT, esse não está completo, pois o espelho de rede não foi utilizado na instalação e outros repositórios como os relativos ao CD-ROM de instalação precisam ser comentados. Para editar o arquivo de configurações execute o comando nano /etc/apt/sources.list . O nano é um poderoso editor de arquivos que já vem instalado mesmo nessa versão mínima do Debian. O arquivo deve parecer com esse:

```
#
#deb cdrom:[Debian GNU/Linux 4.0 r3_Etch_ - Official i386 NETINST Binary-1 20$

deb cdrom:[Debian GNU/Linux 4.0 r3_Etch_ - Official i386 NETINST Binary-1 2008$

# Line commented out by installer because it failed to verify:
#deb http://security.debian.org/ etch/updates main contrib.
# Line commented out by installer because it failed to verify:
#deb-scr http://security.debian.org/ etch/updates main contrib
```

Altere o arquivo acima, comentando o segundo repositório que começa com texto "deb cdrom", para comentar a linha coloque "#" no começo dela. Tire os comentários dos dois repositórios de segurança abaixo e adicione a lista os seguintes repositórios oficiais (sem aspas) "deb http://ftp.us.debian.org/ etch main" e "deb-src http://ftp.us.debian.org/ etch main". O arquivo deve ficar assim, retirando comentários realizados pelo instalador:

```
#
#deb cdrom:[Debian GNU/Linux 4.0 r3_Etch_ - Official i386 NETINST Binary-1 20$

#deb cdrom:[Debian GNU/Linux 4.0 r3_Etch_ - Official i386 NETINST Binary-1 2008$

deb http://ftp.us.debian.org/ etch main
deb-src http://ftp.us.debian.org/ etch main

deb http://security.debian.org/ etch/updates main contrib.
deb-scr http://security.debian.org/ etch/updates main contrib
```

Depois disso execute os comandos:

```
apt-get update
apt-get upgrade
```

Caso tenha algum problema na conexão com a internet veja o apêndice B – problemas com a rede. Os comandos acima realizam o *update* e *upgrade* do sistema operacional. Durante o processo de *upgrade* o APT perguntará se deve ou não realizar alguns *downloads*, isso fica ao critério do instalador, no nosso caso sempre instalamos todos os *upgrades* do sistema.

3.5.2 Instalando Pacotes

Com o APT funcionando, vamos para a próxima etapa, a instalação dos pacotes. Para isso vamos utilizar o comando apt-get passando o parâmetro install e em seguida o nome do

pacote. Ao efetuar o comando ele irá descarregar o pacote e o instalará em seguida. Para instalar os pacotes necessários execute os seguintes comandos:

```
apt-get install gcc
apt-get install g++
apt-get install g77
apt-get install make
apt-get install python
apt-get install ssh
```

Alguns pacotes possuem dependências, logo se o APT perguntar se deseja realizar o *download* dessas responda as perguntas com “S”. Caso esteja instalando o computador central, no nosso caso o mestre, é interessante o uso de uma interface gráfica. No nosso caso instalaremos uma interface gráfica leve chamada XFCE4. Essa interface depende do pacote xorg para funcionar. Execute o seguinte comando para instalá-lo:

```
apt-get install xorg
```

Após a instalação surgirá uma tela perguntando as resoluções de vídeo que o servidor X deve utilizar. No nosso caso foram escolhidas somente as opções 800x600 e 640x480. Para instalar a interface gráfica execute o comando:

```
apt-get install xfce4
```

O *mouse serial* não é reconhecido automaticamente pela interface gráfica sendo assim, deve-se re-configurar o Xorg através do arquivo xorg.conf na pasta /etc/X11/ . Devem ser editadas as linhas referentes ao protocolo e ao dispositivo utilizado pelo *mouse*. Troque o nome do dispositivo pelo dispositivo serial que está utilizando, no nosso caso foi a “/dev/ttyS0” e troque o protocolo utilizado para “Auto”. A parte do arquivo relativo ao *mouse* deve ficar parecida com o abaixo:

```
Section "InputDevice"
   Identifier "Configured Mouse"
   Driver "mouse"
   Option "CorePointer"
   Option "Device" "/dev/ttyS0"
   Option "Protocol" "Auto"
   Option "Emulate3Buttons" "true"
EndSection
```

Caso tudo esteja instalado corretamente, entre no modo gráfico executando o comando "startxfce4". Finalizada essa etapa podemos instalar o pacote de maior importância do trabalho o MPICH2. Para isso, precisamos baixar os arquivos fontes do mesmo no sitío oficial, e uma das formas de realizar o *download* é através de um navegador. No nosso caso utilizamos o Mozilla Firefox, para instalá-lo, execute o comando apt-get install firefox . Após a instalação, faça o *download* pelo link http://www.cs.mtsu.edu/~zach/debian/current/mpich2-1.0.3.tar.gz (acesso: 09 maio, 2008). Note que o pacote vai estar compactado. Para descompactá-lo execute o comando:

```
tar xfz mpich2-1.0.3.tar.gz
```

Depois dessa etapa deve ser gerada uma pasta chamada "mpich2-1.0.3" no local de descompactação. Entre nesse diretório e execute os seguintes comandos:

```
./configure | tee inst_configure.log
make | tee make.log
make install | tee install.log
```

O primeiro comando executa a configuração do MPICH2 para instalação, o segundo gera os binários para instalação (compila o código fonte) e o terceiro executa o instalador gerado. Todos os comandos geram um arquivo com o *log* da etapa realizada, o nome desses está ao lado de cada um dos comandos "inst_configure.log", "make.log" e "install.log". Caso ocorra algum problema durante uma dessas etapas, veja o respectivo arquivo de *log*. Caso contrário confirme se realmente o MPICH2 foi instalado com sucesso, isso pode ser realizado através dos comandos:

```
which mpd
which mpicc
which mpiexec
which mpirun
```

Se cada um retornar o seu caminho absoluto no sistema à instalação foi feita com sucesso.

3.6 Instalação nos nós

3.6.1 Processo de Clonagem

O processo de instalação do sistema operacional feito na máquina principal deve ser repetido, no nó que possua o menor espaço de disco rígido. O motivo dessa escolha é bem simples, esse nó será usado como imagem no processo de clonagem dos computadores, logo suas partições tem que ser pequenas o suficiente para caberem em qualquer um dos outros nós. No nosso projeto essa máquina foi o "no1" que possui apenas 6GB de espaço em disco.

Repita toda a instalação feita no nó principal exceto à instalação do pacote MPICH2 e a interface gráfica XFCE4. O MPICH2 deve ser instalado após a clonagem em cada nova máquina e a interface gráfica será desnecessária nos nós escravos. Com o nó base pronto (no nosso caso, o "no1") faça alguns testes de funcionamento, como ping <nome_do_nó_principal>, ssh <nome_do_nó_principal>. Caso esses funcionem podemos seguir em frente, caso contrário verifique o apêndice B – problemas com a rede.

No processo de clonagem utilizaremos um *Live-CD* de um sistema operacional GNU/Linux da distribuição Slax e o partimage, composto por um programa cliente chamado de "partimage" e um servidor chamado de "partimaged". O *Live-CD* será utilizado pela necessidade do partimage funcionar sem nenhuma partição montada no sistema, ou seja, só é possível gerar as imagens de todas as partições do nó escravo se, o Debian do mesmo não estiver sendo executado. Um *Live-CD* corresponde a um sistema operacional inicializável em um CD, que utiliza somente a memória RAM do computador para ser executado. Sendo assim todas as partições ficam livres para cópia. Essa estratégia também será utilizado em máquinas novas (sem sistema operacional e formatadas), para realizar a restauração das imagens, completando assim o processo de clonagem. Outra característica fundamental do *Live-CD* Slax é seu serviço inerente de SSH, que será utilizado para cópia de alguns arquivos essenciais e pelo próprio partimage.

A primeira etapa desse processo é o *download* do *Live-CD* do Slax e do partimage. A página oficial do slax é www.slax.org (acesso: 9 maio, 2008) onde pode-se realizar o *download* referente ao arquivo em http://www.slax.org/get_slax.php?download=iso (acesso: 9

maio, 2008). Utilizamos aqui a versão 5.1.8 do Slax. No entanto não devem ocorrer problemas na utilização das versões futuras do Slax, entretanto caso ocorram, procurem utilizar a versão 5.1.8 do mesmo ou outro dos vários *Live-CD* de outras distribuições, mas lembre-se ele deve possuir o serviço de SSH.

O partimage utilizado nesse trabalho pode ser encontrado no CD em apêndice, no entanto existem muitas outras ferramentas para desempenhar o mesmo papel. Essa versão do partimage foi utilizada pela sua facilidade de uso e por sua portabilidade. Guarde uma cópia do partimage e do partimaged no nó principal, assim os nós poderão carrega-lo através da rede utilizando o SSH.

Após o *download* do *Live-CD* do Slax, grave o arquivo em um CD, e realize a inicialização do seu nó, o qual servirá de imagem, pelo *drive* de CD-ROM (mesmo procedimento realizado na instalação do Debian). Digite o *login* "root " e o *password* "toor", para entrar no sistema.

Com o Slax inicializado, vamos configurar a rede e copiar o programa cliente do partimage para o sistema. Para configurar a rede utilize o comando ifconfig da seguinte forma:

ifconfig <interface_de_rede> <IP> netmask <máscara_de_rede>

No nosso caso ficou assim:

ifconfig eth0 172.18.86.116 netmask 255.255.0.0

Confirme se o comando acima obteve sucesso executando o comando ifconfig, caso seja listada a interface eth0 com o IP especificado ele funcionou, caso contrário veja o apêndice B – problemas com a rede. Utilizamos o seguinte comando SCP para realizar a cópia desse arquivo:

scp <arquivo_de_origem> <arquivo_destino>
scp 172.18.86.115:/root/Desktop/partimage /root/

O SSH irá pedir uma confirmação de conexão e depois pedirá a senha para acessar o nó principal. Responda "*yes"* e depois digite a senha do usuário *root* do nó principal. No nosso caso o arquivo será carregado para a pasta /root/ onde pode ser executado pelo caminho

completo /root/partimage ou de dentro da pasta /root/ por meio de ./partimage . Inicie o servidor do partimage o partimaged no computador principal. Esse aguardará a conexão do computador escravo. Como dito anteriormente as partições do Debian não podem estar em uso ou montadas no sistema. Por padrão as partições Debian serão carregadas, para descarregá-las utilize o comando:

umount <nome_do_dispositivo>

No nosso caso terão de ser "desmontadas" os dispositivos /dev/hda1, /dev/hda5, /dev/hda6, /dev/hda8 e /dev/hda9. Para visualizar as partições montadas execute o comando "mount". No nosso caso executamos os comandos, para desmontar as partições:

umount /dev/hda1
umount /dev/hda5
umount /dev/hda6
umount /dev/hda8
umount /dev/hda9

Execute o comando "*mount*" novamente e cheque se não há nenhuma partição montada, relativa ao disco rígido hda (/dev/hdaN), caso nenhuma seja mostrada, podemos executar o partimage. Execute o partimage como já dito anteriormente por /root/partimage ou de dentro da pasta /root/ execute ./partimage . Uma tela de diálogo SHELL será aberta:

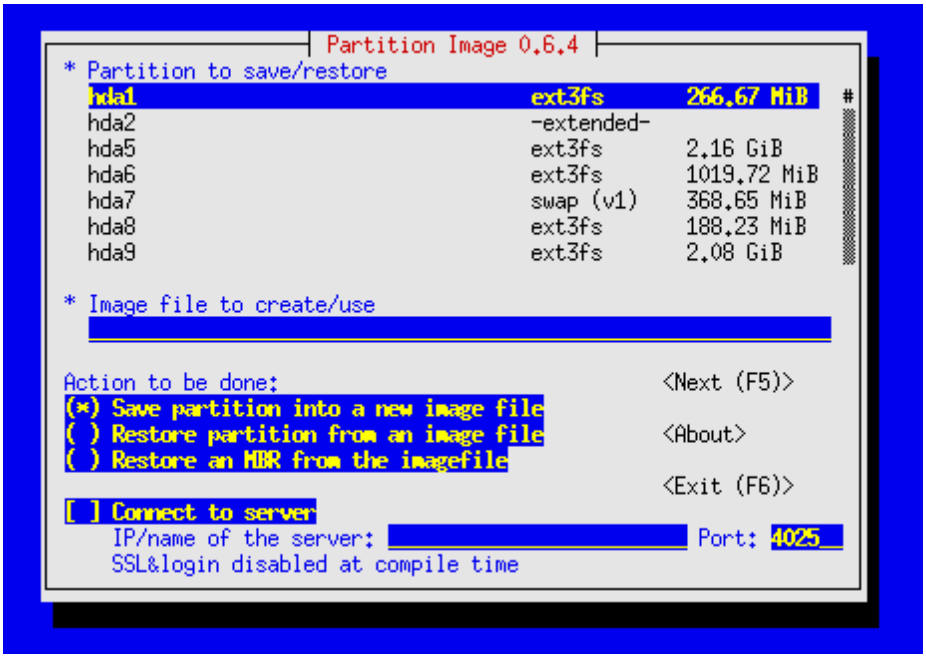

Figura 3.4: Mostra a tela inicial do partimage.

Selecione no quadro da parte superior, a partição que deseja converter em imagem. Essa estando selecionada aperte a tecla “Tab” para navegar entre as opções. O campo abaixo do quadro de seleção é do endereço destino da imagem (onde no servidor será salva a imagem. No lado esquerdo abaixo desse campo, existem três opções, a primeira irá salvar a partição escolhida em um arquivo de imagem, a segunda opção serve para restaurar o arquivo de imagem na partição e a terceira restaura a MBR de um arquivo de imagem. A caixa de seleção abaixo desse deve ser marcada para conectar-se com o servidor, onde será gravada a imagem da partição. Ao lado direito abaixo do campo anterior existem dois campos o primeiro para especificar o IP do servidor e o segundo a porta de comunicação.

No nosso projeto foram selecionados os dispositivos /dev/hda1, /dev/hda5, /dev/hda6, /dev/hda8 e /dev/hda9, cada um de uma vez. Preenchendo o caminho do servidor remoto com “/home/andre/imagenno/imagemhda1”, “/home/andre/imagenno/imagemhda5” e assim sucessivamente. Selecionando a primeira opção, salvar partição em um novo arquivo de imagem. Conectando ao servidor mestre pelo IP 172.18.86.115 e utilizando a porta padrão 4025. Observe que o dispositivo /dev/hda2 é uma partição estendida logo, não é possível a criação de uma partição a partir dela e o /dev/hda7 é uma partição de swap do disco rígido que também não pode ser “clonada”. Essas serão clonadas utilizando uma cópia da MBR e da estrutura de partições desse nó.

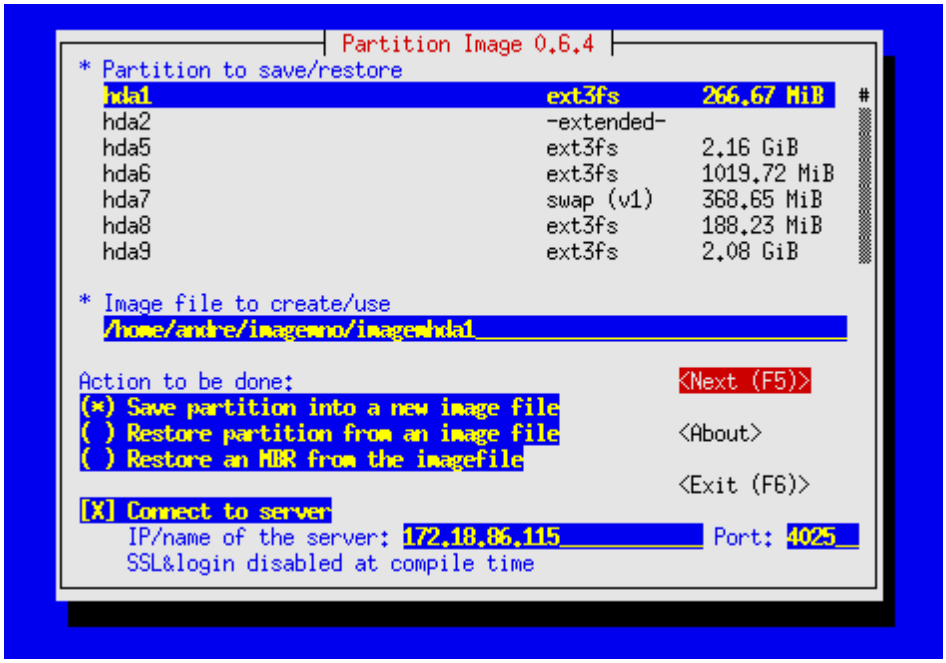

Figura 3.5:Partimage configurado para gerar a imagem da partição hda1

Após preencher os campos aperte a tecla “F5” para continuar. A segunda tela trará várias opções, essas podem ser deixadas com seus valores padrão e podemos seguir em frente. A terceira tela perguntará a descrição do arquivo que será gerado, no nosso caso, inserimos nesse campo “hda1”, “hda5” e assim sucessivamente. Esse campo é importante pois, no futuro ele servirá como forma de confirmação de que estamos carregando a imagem certa. Selecione o botão “OK” para continuar a operação. O programa mostrará os dados da partição e quanto espaço essa ocupará no disco rígido do servidor, pressione “OK” para continuar o processo. Dependendo do tamanho da partição o processo pode demorar um pouco. Note que o servidor “partimaged” estará recebendo a imagem através da rede. No término da transferência o “partimage” será encerrado automaticamente. O procedimento deve ser repetido para todas as outras partições especificadas acima.

Com as imagens das partições geradas, reinicie o computador e entre no Debian. Para isso configure a BIOS para iniciar através do IDE-0. Agora criaremos a imagem da MBR e da estrutura de partições. Para isso vamos utilizar o programa “dd” que “entre outras tarefas, cria imagens de dispositivos, copiando os seus dados bloco a bloco (MOTA FILHO, 2006, p.508)”. Execute os comandos abaixo

```
dd if=/dev/hda of=hda.mbr count=1 bs=512
sfdisk –d /dev/hda > hda.sf
```

O primeiro comando gera uma imagem dos primeiros 512 bytes do dispositivo “hda” (disco rígido utilizado), que são a MBR do disco, e os salva dentro do arquivo “hda.mbr”. O segundo comando gera uma cópia da tabela de partições do disco rígido e grava o resultado em hda.sf. Por uma questão de organização, vamos manter todos as imagens no nó principal. Para isso utilize o programa “scp”, no nosso caso esse foi utilizado dessa forma:

```
scp hda.mbr mestre:/home/andre/imagemno/
scp hda.sf mestre:/home/andre/imagemno/
```

Agora vamos iniciar um novo nó, que receberá as imagens geradas, por meio do *Live-CD* do Slax. Depois da inicialização, temos que ter certeza que não existem partições no antigo disco rígido. Para isso execute o programa fdisk dessa forma:

```
fdisk /dev/hda
```

O programa será aberto e uma opção será requisitada. Aperte a tecla “p” para visualizar a tabela de partições. Caso essa mostre alguma partição, aperte a tecla “d” para excluí-la, no entanto tenha cuidado, esse procedimento irá apagar a referência dos dados que o disco rígido contém. Caso tenha algum dado importante nesse disco rígido tire um *backup* antes. Após excluir a partição ou partições, aperte a tecla “q” para sair do programa.

Após concluir a etapa anterior vamos copiar os arquivos referentes às imagens da MBR e do particionamento do disco rígido. Caso não esteja utilizando DHCP, re-configure a rede como já explicado anteriormente. Caso tenha alguma dúvida na configuração da rede consulte o apêndice B. Em seguida vamos utilizar o programa “scp” para carregar as imagens relativas a MBR e a tabela de partições do disco.

```
scp 172.18.86.115:/home/andre/imagemno/hda.mbr /root/
scp 172.18.86.115:/home/andre/imagemno/hda.sf /root/
```

Note que não utilizamos o nome “mestre” para referenciar o nó principal, já que, o Slax não teve seu arquivo de “hosts” configurado com esse objetivo. Com os arquivos copiados vamos restaurar a MBR e o particionamento do disco rígido. Para isso vamos executar os comandos:

```
dd if=hda.mbr of=/dev/hda
sfdisk –force /dev/hda < hda.sf
```

Reinicie o computador para que as partições sejam automaticamente reconhecidas e montadas pelo Slax. No entanto, para que as partições sejam restauradas é necessário que essas não estejam montadas. Para desmontar as partições execute os comando.

```
umount –t ext3 –a
umount /dev/hda1
```

Já que estamos utilizando um *Live-CD* e o mesmo utiliza somente a memória RAM para funcionar, a cada inicialização a rede deve ser re-configurada. Depois de configurada vamos copiar o programa “partimage” (programa cliente) e executar o programa “partimaged” no nó principal. Utilize o seguinte comando para copiar o “partimage”:

```
scp 172.18.86.115:/root/Desktop/partimage /root/
```

Execute o programa partimage. Vamos configurá-lo da mesma forma que anteriormente,

só que dessa vez, vamos realizar a restauração das partições e não a criação das imagens. Selecione a partição que deseja restaurar no quadro na parte superior, uma vez selecionado aperte a tecla “Tab” e preencha o caminho completo referente a imagem dessa partição dentro do nó principal especificando também o nome completo do arquivo desejado, no nosso caso esse foi “/home/andre/imagemno/imagemhda1.000”. Selecione a opção “Restore partion from na imagem file” (Restaurar uma partição por meio de um arquivo de imagem), marque a opção conectar ao servidor, coloque o IP de seu servidor (no nosso caso 172.18.86.115) e utilize a porta padrão 4025. As configurações utilizadas, no nosso projeto, foram as seguintes:

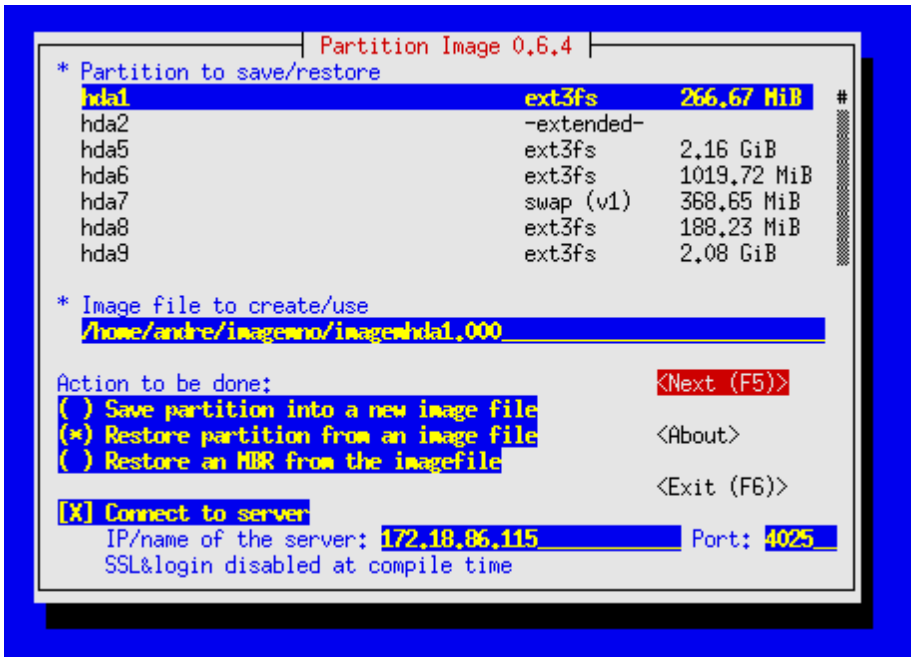

Figura 3.6:Partimage configurado para restaurar a imagem da partição hda1

Outra tela será aberta, deixe as opções apresentadas com a configuração padrão e aperte a tecla “F5” para continuar. Um quadro de informações sobre a partição será apresentado e após um quadro de confirmação, verifique as informações dispostas, e caso tudo esteja certo, aperte a tecla “Enter” para continuar. Realize esse todo esse procedimento para as partições hda1, hda5, hda6, hda8 e hda9.

Por motivos ainda desconhecidos o GRUB falha em inicializar o sistema após esse procedimento. Conseguimos resolver esse problema reinstalando o mesmo. Para que isso seja feito remonte a partição hda1. Executando o seguinte comando:

mount /dev/hda1 /mnt/hda1

Agora vamos utilizar o comando “chroot” para mudar a base do sistema que está sendo utilizada do Live-CD para a partição hda1. Para que assim podermos reinstalar o GRUB.

```
chroot /mnt/hda1
```

O comando de reinstalação do GRUB, o grub-install, está na partição /usr, logo temos que monta-la para utilizá-lo. Ao executar esse programa devemos passar qual disco rígido queremos reinstalar o GRUB.

```
mount /dev/hda5 /usr/
grub-install /dev/hda
```

Finalizada essa etapa, vamos reiniciar o computador. Para realizar essa ação utilize os comandos abaixo:

```
exit
reboot
```

Configure a BIOS para utilizar o dispositivo IDE-0 para inicializar o sistema e vamos configurar o novo nó. Entre com o usuário *root* no sistema. Primeiramente vamos mudar o nome da máquina e sua senha de acesso. Para isso edite o arquivo /etc/hostname trocando o nome antigo que está dentro do arquivo pelo novo nome. Para isso utilize os comandos:

```
nano /etc/hostname
passwd
```

Após o último comando acima serão requisitados a nova senha e a confirmação da mesma. Para configurar a rede entre no arquivo “interfaces” com o comando:

```
nano /etc/init.d/interfaces
```

Nesse estarão dispostos configurações de rede da máquina. Para que a rede funciona-se normalmente tivemos que mudar a interface de rede de eth0 para eth. e modificar o IP da máquina para um novo IP. Segue o arquivo “interfaces”, com as partes que devem ser modificas em **negrito.**

```
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).
```

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
allow-hotplug eth1
iface eth1 inet static
   address 172.18.86.117
   netmask 255.255.0.0
   network 172.18.0.0
   broadcast 172.18.255.255
   gateway 172.18.0.1
   # dns-* options are implemented by the resolvconf package, if installed
   dns-nameservers 172.18.0.1
   dns-search cluster.uniceub
```

Após esse procedimento reinicie a rede do Debian através do comando:

```
/etc/init.d/networking restart
```

Caso a rede não funcione corretamente veja o apêndice B – problemas com a rede. Finalizada essa etapa instalaremos agora o MPICH2. Para isso siga o mesmo procedimento descrito para o nó principal e o primeiro nó. Execute os comandos a seguir dentro da pasta de descompactação do MPICH2.

```
./configure | tee inst_configure.log
make | tee make.log
make install | tee install.log
```

Finalizada a clonagem de todos os nós, vamos configurar o SSH.

3.6.2 Configurando o SSH

Para que o MPICH2 funcione de forma correta os nós devem ser capazes de executar comandos de SSH entre eles sem o pedido de senha. Para realizar essa tarefa vamos gerar uma chave pública e uma privada para cada nó através do programa “ssh-keygen”. Esse pode utilizar dois tipos de criptografia, RSA ou DSA. No nosso projeto utilizaremos a RSA. Execute em cada um dos nós, inclusive na máquina principal, o seguinte comando:

```
ssh-keygen –t rsa
```

Após a execução do comando acima, o “ssh-keygen” irá perguntar em qual diretório deve ser salvo o arquivo, deixe o diretório padrão, e depois irá perguntar e pedir a confirmação da senha de acesso para a máquina (*passphrase*), deixe essa em branco. Esse comando irá gerar dois arquivos um com a chave pública e um com a privada em cada máquina. Essas poderão ser encontradas em:

/root/.ssh/id_rsa (Chave privada)
/root/.ssh/id_rsa.pub (Chave pública)

Após termos gerado todas as chaves, as juntaremos, por uma questão de organização, no computador principal. As chaves que devem ser enviadas são as chaves públicas, e essas devem ser inseridas dentro do arquivo “/root/.ssh/authorized_keys” de cada máquina, para serem reconhecidas pelo SSH. Utilize o comando SCP para juntar todas as senhas no computador central, no nosso caso esse procedimento foi feito, através do computador central, dessa forma:

scp <nome_do_nó>:/root/.ssh/id_rsa.pub /home/andre/senhasSSH/<nome_do_nó>_id_rsa.pub

Ao final desse processo devemos juntar todas as senhas públicas em um arquivo chamado authorized_keys, para isso utilize o comando:

cat <nome_do_nó>_id_rsa.pub >> authorized_keys

Uma vez juntas utilize o programa “scp” para colocar o arquivo “authorized_keys” no diretório “/root/.ssh/” de todas as máquinas. Uma vez concluído, teste se o SSH está funcionando sem senhas. Para isso execute um dos comandos abaixo através do computador central para todos os outros nós.

ssh <nome_do_nó> date
ssh <nome_do_nó> hostname

Caso o SSH não peça senha de acesso à configuração foi concluída com sucesso, caso contrário o processo deve ser refeito.

3.6.3 Configurando o MPICH2

As configurações aqui descritas devem ser realizadas somente no computador principal, no nosso caso o nó chamado de mestre. Por razões de segurança, o MPICH2 procura um

arquivo de configuração chamado “mpd.conf” na pasta “/etc/”. Esse arquivo deve possuir uma palavra chave para o seu funcionamento. A senha deve constar dentro desse arquivo da seguinte forma:

```
secretword=MinhaSenha
```

De preferência essa senha não deve ser igual à de acesso ao Debian e o arquivo deve poder ser alterado e lido somente pelo usuário em questão. Para isso, realize o seguinte comando sobre o arquivo recém criado e o copie para todos os nós utilizando o “scp”:

```
chmod 600 mpd.conf
scp mpd.conf no1:/etc/
scp mpd.conf no2:/etc/
scp mpd.conf no3:/etc/
scp mpd.conf no4:/etc/
scp mpd.conf no5:/etc/
scp mpd.conf no6:/etc/
scp mpd.conf no6:/etc/
scp mpd.conf no7:/etc/
```

Ao final dessa etapa podemos realizar o primeiro teste do MPICH2. Execute os seguintes comandos:

```
mpd &
mpdtrace
mpdallexit
```

O primeiro comando acima executa o gerenciador de processos padrão, o MPD (multipurpose daemon). O comando seguinte, “mpdtrace”, retorna como resposta quais máquinas tiveram seus gerenciadores inicializados com sucesso. O terceiro comando, “mpdallexit”, fecha os gerenciadores de processos inicializados. Ao realizar esses comandos, o nome da máquina principal deve ser emitido, já que somente o gerenciador de processos do nó principal foi ativado.

Agora vamos criar outro arquivo chamado “mpd.hosts” em “/root/” com uma lista dos nomes de todos os computadores do *cluster*. Caso seja necessário, especifique o nome completo de cada nó (nome da máquina seguido do nome do domínio). No nosso caso, especificamos o nome completo para evitar problemas futuros. Nosso arquivo de configuração foi escrito dessa forma:

mestre.cluster.uniceub
no1.cluster.uniceub
no2.cluster.uniceub
no3.cluster.uniceub
no4.cluster.uniceub
no5.cluster.uniceub
no6.cluster.uniceub
no7.cluster.uniceub

Agora vamos iniciar os MPD's de todas as máquinas do cluster. Para isso vamos utilizar o comando "mpdboot" que possui a seguinte sintaxe básica:

mpdboot –n <número_de_máquinas> –f <caminho_do_arquivo>

O argumento –n corresponde ao número de maquinas que devem ter o MPD inicializado e o argumento –f corresponde ao caminho completo do arquivo "mpd.hosts", que será lido pelo "mpdboot". No nosso projeto, o seguinte comando foi utilizado para realizarmos o *boot* em todas as máquinas do cluster:

mpdboot –n 8 –f /root/mpd.hosts

Teste a inicialização dos MPD's, executando o programa "mpdtrace". Esse deve retornar o nome de todos os nós que tiveram o MPD inicializado com sucesso. Vamos testar agora uma aplicação teste do MPICH2. A aplicação utilizada no teste seguinte, faz uso do processamento paralelo, para calcular o valor da constante π(PI) com 25 casas decimais. Para executar esse programa em paralelo utilize o comando "mpiexec". No nosso caso, esse foi executado dessa forma:

mpiexec –n 8 /usr/local/bin/examples/cpi

O argumento –n nesse comando corresponde a quantos processos devem ser criados. Esse número pode ser superior ao número de máquinas. Sendo assim, serão criados mais de um processo por computador. Após o valor do argumento –n, segue o caminho do programa que deve ser executado em paralelo. Note que no nosso caso a pasta "examples", foi movida para dentro da pasta "/usr/local/bin/", para que os arquivos ficassem mais organizados. O resultado desse comando deve ser semelhante ao que obtivemos abaixo:

Process 0 of 8 is on mestre
Process 2 of 8 is on no4

```
Process 1 of 8 is on no3
Process 3 of 8 is on no2
Process 4 of 8 is on no1
Process 5 of 8 is on no6
Process 6 of 8 is on no5
Process 7 of 8 is on no7
pi is approximately 3.1415926544231247, Error is       0.0000000008333316
wall clock time = 0.039721 segundos
```

O exemplo acima mostra o cálculo aproximado de π(PI), utilizando os 8 computadores do *cluster*, mostrando também qual processo ficou com qual computador. Ao final, exibe o erro que essa aproximação teve em relação a constante π(PI), com 25 casas decimais, e o tempo em segundos que levou o processamento (*wall clock time*).

3.7 Compiladores

Para realizarmos a edição do código fonte de forma facilitada, utilizamos a IDE (Integrated Development Environment) Kdevelop. Esse programa é livre e pode ser descarregado via APT pelo comando:

```
apt-get install kdevelop
```

Para compilar os arquivos fontes vamos utilizar o compilador disponibilizado pelo pacote MPICH2, o chamado MPICC. Esse não é um compilador em si, ele utiliza o GCC para realizar a compilação do programa. Utilizamos MPICC, pois este já realiza a chamada das bibliotecas necessárias a serem utilizadas, como a “mpi.h”, que é fundamental para compilação de um programa que utiliza o MPI. Além dessa biblioteca, configuramos o programa MPICC para realizar a chamada da “math.h” que geralmente é utilizada em nossas aplicações.

Para adicionar essa biblioteca altere o MPICC executando o comando

```
nano /usr/local/bin/mpicc
```

e modifique o valor da variável MPI_OTHERLIBS adicionando a string “-lm”.

```
MPI_OTHERLIBS="     -lpthread  -lrt -lm  "
```

Vamos compilar o programa “hellow.c”, presente na pasta de exemplos do MPICH2 (/usr/local/bin/examples/). Para facilitar essa tarefa, entre na pasta de exemplos especificada anteriormente e execute os comandos abaixo.

```
mpicc <nome_do_arquivo_fonte> -o <nome_da_aplicação>
mpicc hellow.c -o hellow
```

Ao final desse comando, deve ser criado um executável chamado “hellow”, que pode ser executado em paralelo. Para executá-lo dessa forma, todas as máquinas devem possuir uma cópia deste, no mesmo local. Copie o arquivo para todas as máquinas por meio do “scp”, esse procedimento é cansativo e pode ser substituído por um simples *SHELL Script*, que enviará o arquivo especificado para todas as máquinas.

Durante nosso projeto foi desenvolvido o seguinte *SHELL Script*, chamado distribuiPrograma.sh.

```
#!/bin/bash

echo Digite o nome do programa que deseja distribuir..
read NOME_PROGRAMA

for i in `cat /etc/hosts | grep no[0-9] | awk '{print $3}'`; do
      if [ $i != mestre ]; then
      echo $i
      scp /usr/local/bin/examples/$NOME_PROGRAMA $i:/usr/local/bin/examples/
      fi
done
```

Por mais simples que esse programa pareça, ele realiza em poucos segundos a tarefa de copiar uma aplicação para todos os nós do *cluster*. Já que SHELL Script não é o escopo desse trabalho, vamos somente explicar como a aplicação funciona. Quando executamos o programa(./distribuiPrograma.sh), ele irá perguntar qual é o nome do arquivo que queremos enviar para todas as máquinas. Após inserir o nome do arquivo, o aplicativo irá ler o arquivo “/etc/hosts” e selecionará a terceira coluna das linhas que possuem a expressão regular no[0-9](ou seja, no1,no2,no3... no9), assim será montado, de forma automática, uma lista dos nomes dos nós utilizados no projeto. Com os nomes das máquinas carregados, ele executará o comando “scp”, enviando o arquivo especificado do diretório de exemplos do MPICH2, do computador principal, para os diretórios de exemplos de todos os outros nós, exceto o nó principal que já terá essa aplicação. Ao executarmos o programa, inserindo a *string*

“hellow”como nome do arquivo, foi mostrada a seguinte saída:

```
Digite o nome do programa que deseja distribuir..
hellow
no1
hellow                    100%  526KB 526.3KB/s   00:00
no2
hellow                    100%  526KB 526.3KB/s   00:00
no3
hellow                    100%  526KB 526.3KB/s   00:00
no4
hellow                    100%  526KB 526.3KB/s   00:00
no5
hellow                    100%  526KB 526.3KB/s   00:00
no6
hellow                    100%  526KB 526.3KB/s   00:00
no7
hellow                    100%  526KB 526.3KB/s   00:00
```

Na saída acima, nas linhas iniciadas pelo nome do programa (nesse caso *hellow),* as outras colunas que seguem, são respectivamente, à medida do progresso da transferência do arquivo, o tamanho total do arquivo, a velocidade de transferência e o tempo gasto para realizar a transferência desse arquivo para máquina especificada na linha acima. No caso, já que a transferência foi efetuada em menos de um segundo, o tempo de cada operação consta como “00:00”, já que o menor tempo que poderia ser exibido seria de 1 segundo.

As transferências de arquivos são intercaladas com os nomes das máquinas, para que, caso haja alguma falha em uma das transferências, seja facilmente especificado qual máquina falhou em recebe-lo. Após essa etapa, podemos executar o programa “hellow” da mesma forma que foi mostrada anteriormente. No nosso projeto, essa pode ser realizada pelos seguintes comandos:

```
mpdboot -n 8 -f /root/mpd.hosts
mpiexec -n 8 /usr/local/bin/examples/hellow
```

Essa é uma aplicação muito simples e gerará como saída somente uma mensagem de “Hello World”, de cada processo.

4. A aplicação

O propósito de nossa aplicação é resolver um grande cálculo fatorial, no qual suas partes são processadas de forma paralela entre vários nós.

O fatorial de um número é descrito pela expressão

n! = n * (n-1) * (n-2) * (n-3) * (n-4) * ... * 2 * 1

logo o fatorial do número 6 seria

6! = 6 * 5 * 4 * 3 * 2 * 1 = 720.

No começo do desenvolvimento pensávamos em dividir o processamento de um fatorial entre vários nós. Para isso utilizamos números pequenos para testar a divisão de tarefas.

Após os primeiros testes com o programa funcionando corretamente, um computador sozinho estava processando mais rapidamente o algoritmo desenvolvido do que todos os 8 nós juntos. A primeira vista isso contraria a lógica, entretanto, pode ser explicado por um simples fato. Para as máquinas aqui utilizadas, um fatorial de um número baixo (abaixo de 50) não corresponde a um trabalho de alto processamento. Logo, quando o algoritmo divide um fatorial baixo em várias partes distribuindo o cálculo realizado entre os nós ele gasta mais tempo realizando a comunicação entre os computadores do que um nó trabalhando sozinho para resolver o problema.

Tentamos assim aumentar o valor do fatorial sendo resolvido para que cada máquina tivesse uma porção de trabalho grande o suficiente para compensar o tempo de comunicação através da rede.

Ao chegarmos ao fatorial de 170! (7,2574e^306) os resultados obtidos foram os mesmos. Acima desse valor tivemos que trocar a definição do tipo de dado que armazenava a resposta, já que o resultado desse fatorial é o maior que o tipo de dado *double* pode armazenar. Assim passamos a utilizar o tipo de dado *long double* que possui um limite de numérico muito superior.

Tabela 4.1: Limites das representaç de ponto flutuante

Tipo	Tamanho (em bits)	Intervalo
float	32	3,4E-38 a 3,4E+38
double	64	1,7E-308 a 1,7E+308
long double	80	3,4E-4932 a 1,1E+4932

Fonte: http://apostilacpp.awardspace.com/index.php?pagina=modulo03 , acesso: 05 de junho, 2008.

Após a troca de tipo de dado continuamos a aumentar o valor do fatorial com a esperança de chegarmos a um valor no qual a solução em vários computadores fosse vantajosa.

Os resultados obtidos continuaram os mesmos, sendo mais rápida a execução de um fatorial em um nó do que em vários nós. Nesse ponto já havíamos chegado ao final do limite de armazenamento de uma variável do tipo *long double* com o fatorial de 1754! (1,979e^4930).

Já que o processamento de um fatorial com tamanho máximo de *long double* não foi suficiente, modificamos o programa para que o mesmo realiza-se a mesma operação cerca 100.000 vezes. As modificações realizadas no programa seguem destacadas em negrito no código fonte utilizado.

```
// Biblioteca de entrada e saída da linguagem C
#include <stdio.h>
// Biblioteca de funções do MPI
#include "mpi.h"
// Declaração da função “fat” que realiza o cálculo fatorial
long double fat(int,int);
// Declaração da função principal do programa
int main ( int argc, char *argv[] );

int main ( int argc, char *argv[] )
{
// Contador i
long int i=0;
/* Respectivamente, número a ser fatorado, início do intervalo de fatoração de um processo e fim do intervalo de fatoração de um processo */
int num=1754,ini,fim;
/* Respectivamente, o valor do fatorial total (resultado final) e o valor do fatorial de cada processo */
long double fatorial,parte;
/* Respectivamente, o número de processos em execução e o identificador do processo que está executando o programa */
int numprocs,rank;
```

```
    /* Respectivamente o tempo de execução inicial e o tempo de execução final */
    double tempoInicial = 0.0, tempoFinal;
    // Inicialização do MPI
    MPI_Init(&argc,&argv);
    // Grava na variável "numprocs" a quantidade de processos sendo utilizados
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    // Grava na variável "rank" o número identificador do processo sendo executado
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    // O processo está sendo executado no computador principal (mestre)?
    if(rank==0){
        // Pega o tempo de execução inicial no computador principal
        tempoInicial = MPI_Wtime();
    }
    // Realiza a iteração de 0 a 100.000
    for(i=0;i<=100000;i++){
        parte=0.0;
        fatorial=0.0;
        // Divide o intervalo do fatorial em dois limites
        /* O primeiro limite é a variável "ini". Ele representa onde o processo deve
começar o processo de fatoração */
        ini = 1 + rank*(num)/numprocs;
        /* O segundo limite é a variável "fim". Ele representa onde o processo deve
terminar o processo de fatoração */
        fim = (rank+1)*(num)/numprocs;
        /* A parte do resultado do fatorial realizado com base nos limites determinados
acima é atribuída a variável "parte" */
        parte=fat(ini,fim);
        // Existe mais de um processo em execução?
        if(numprocs>1){
            /* O MPI_Reduce junta as partes do fatorial realizado por cada processo e
as multiplica, atribuindo o resultado dessa operação à variável "fatorial" */
            MPI_Reduce(&parte, &fatorial, 1, MPI_LONG_DOUBLE, MPI_PROD,
0, MPI_COMM_WORLD);
        }else{
            // Caso só exista um processo não há necessidade do uso do MPI_Reduce.
            fatorial = parte;
        }
    }
    // o processo está sendo executado no computador principal (mestre)?
    if(rank == 0){
        // Grava na variável "tempoFinal" o tempo final de execução
        tempoFinal = MPI_Wtime();
        /* Imprime o número cujo fatorial foi determinado e o resultado do último fatorial
processado */
        printf("O fatorial de %ld e %.11Le\n",num, fatorial);
        /* Imprime o tempo que levou para o processamento ser realizado */
        printf("O processamento demorou %.16f\n",tempoFinal-tempoInicial);
    }
```

```
    // Finaliza o MPI
    MPI_Finalize();

    return 0;
    }
    // Função que calcula as partes do fatorial
    long double fat(int ini,int fim)
    {
    // Contador
    int i;
    // Resultado obtido pelo fatorial dos dois limites passados
    long double soma=1;
    // Realiza a iteração entre os limites "ini" e "fim"
    for(i=ini;i<=fim;i++){
        /* Atribui à variável "soma" o valor do contador "i" multiplicado pelo seu próprio
valor. Realizando assim, ao longo da iteração, o cálculo fatorial entre os limites
especificados */
        soma=soma*i;
    }
    // Retorna o valor do fatorial que foi calculado
    return soma;
    }
```

Como podemos visualizar acima, a única diferença de um arquivo tradicional da linguagem C para esse é o uso da biblioteca "mpi.h". Essa contém uma coleção de funções desenvolvidas para possibilitar o uso da arquitetura de passagem de mensagens.

O algoritmo é composto de duas partes: a função *main* e a função *fat*. Na função principal a comunicação do MPI é inicializado pela função "MPI_Init(&argc, &argv)", assim o SO (Sistema Operacional) faz as inicializações necessárias para possibilitar o uso do MPI. Essa função é necessária para utilização de todas funções do MPI e recebe como parâmetros os endereços dos parâmetros da função *main* (&argc, &argv).

Em seguida é executada a função "MPI_Comm_size(MPI_COMM_WORLD ,&numprocs)". O primeiro corresponde ao responsável pela comunicação entre os processos, ele representa a coleção de processos que podem se comunicar entre si. O segundo parâmetro é o endereço da variável "numprocs" que após a execução dessa função irá conter o número de processos sendo executados pelo programa.

A função "MPI_Comm_rank(MPI_COMM_WORLD,&rank)" é utilizada para saber qual é a identificação do processo sendo executado pelo nó. Esse número varia de 0 a (numprocs –

1). O primeiro parâmetro da função é o comunicador “MPI_COMM_WORLD” e o segundo é o endereço da variável “&rank” onde será gravado o valor do *ranking* desse processo. Essa função é de grande importância, já que a partir desse valor, podemos designar trabalhos diferentes a cada computador que executar o programa. O computador onde foi feita a execução, no nosso caso o nó chamado de “mestre”, terá o *rank* igual a 0. Esse comportamento é o padrão do MPI.

Nas próximas linhas o começo do trabalho é marcado no computador principal (rank==0) com o comando MPI_Wtime(). Esse retorna o tempo atual de execução em segundos.

```
if(rank==0){
        tempoInicial = MPI_Wtime();
}
```

Após essa etapa, o programa entra em uma iteração onde ocorre a parte principal do algoritmo.

```
parte=0.0;
fatorial=0.0;

ini = 1 + rank*(num)/numprocs;
fim = (rank+1)*(num)/numprocs;
```

A cada vez que é executada a iteração são iniciados os valores das variáveis “parte”, que armazena a parte do fatorial executado pelo processo e a variável “fatorial”, que armazena o valor da multiplicação de todas as partes (resultados obtidos por cada processo), contendo assim o valor final do cálculo fatorial. As variáveis “ini” e “fim” recebem valores diferentes para cada processo dependendo do *rank* do processo, do número cujo fatorial deve ser processado e do número de processos utilizados. Dividindo assim os trabalhos a serem realizados por cada nó de forma uniforme. Por exemplo, o fatorial de 32 (32!) seria divido para dois processos nos seguintes intervalos.

```
//Primeiro Processo no “mestre”
ini = 1 + 0 * (32)/2; // ini = 1
fim = (0+1)*(32)/2; // fim = 16
//Segundo Processo no “no1”
ini = 1 + 1 * (32)/2; // ini = 17
fim = (1+1)*(32)/2; // fim = 32
```

Após a determinação dos limites do fatorial a ser realizado o processo executa a função *fat* que retorna como resultado o fatorial da parte especificada pelos argumentos “ini” e “fim”.

parte=fat(ini,fim);

Caso esteja sendo utilizado mais de um processo na execução da aplicação, utilizaremos a função

MPI_Reduce(&parte, &fatorial, 1, MPI_LONG_DOUBLE, MPI_PROD, 0, MPI_COMM_WORLD)

para retornar a multiplicação de todas as partes do fatorial, tendo assim a resposta final do cálculo requisitado. A função "MPI_Reduce" possui a seguinte sintaxe:

int MPI Reduce(
void* operand /* entrada - no nosso caso o endereço da variável "parte" */,
void* result /* saída – no nosso caso o endereço da variável "fatorial" */,
int count /* entrada – número de elementos sendo enviados, no nosso caso somente 1 */,
MPI Datatype datatype /* tipo de dado do *buffer* de entrada e saída – no nosso caso MPI_LONG_DOUBLE */,
MPI Op operator /* operação a ser executada com as partes – o fatorial é igual a multiplicação de todas as partes, logo será utilizada a operação "MPI_PROD" */,
int root /* entrada – *ranking* do processo que irá obter o resultado da operação, nesse caso esse será o "0" */,
MPI Comm comm /* entrada – comunicador a ser utilizado – MPI_COMM_WORLD */) (KARNIADAKIS; KIRBY II, 2003)

A operação "MPI_Reduce" faz com que cada processo envie para um computador central a resposta de parte do problema, depois efetua a operação especificada entre as parcelas e guarda a resposta na variável especificada.

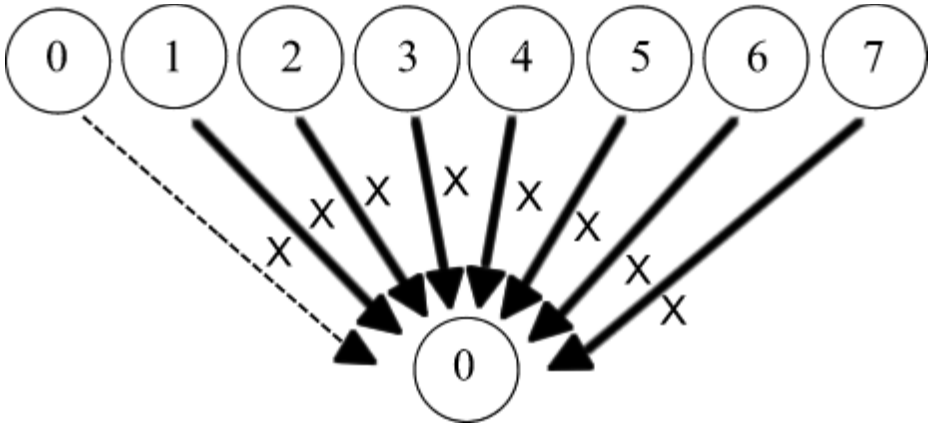

Figura 4.1: MPI_Reduce com 8 processos utilizando a operação MPI_PROD

Como definido no algoritmo, caso tenha somente um processo em execução o

“MPI_Reduce” não será utilizado. Isso é feito porque não ocorrerá divisão de tarefas com somente um processo e assim a variável “parte” determinada para o processo “0” corresponderá à resposta total do sistema (variável “fatorial”).

Ao final dos 100.001 cálculos do fatorial especificado (1754!), o computador principal emitirá uma saída com o número cujo fatorial foi calculado, o valor do fatorial da última iteração realiza e tempo total de execução representado pela subtração do tempo medido no final com o tempo inicial.

```
if(rank == 0){
        tempoFinal = MPI_Wtime();
        printf("O fatorial de %ld e %.11Le\n",num,fatorial);
        printf("O processamento demorou %.16f\n",tempoFinal-tempoInicial);
}
```

Todo programa que utiliza a interface de passagem de mensagens deve ter ao seu final a finalização do mesmo. Isto é realizado pela função “MPI_Finalize()” que avisa ao sistema operacional que os processos abertos pelo “MPI_Init()” já podem ser fechados.

Com essas alterações a carga dos cálculos realizados pelo algoritmo cresceu cerca de 100.000 vezes. Entretanto ele continua a executar de maneira mais rápida em um computador do que em vários nós.

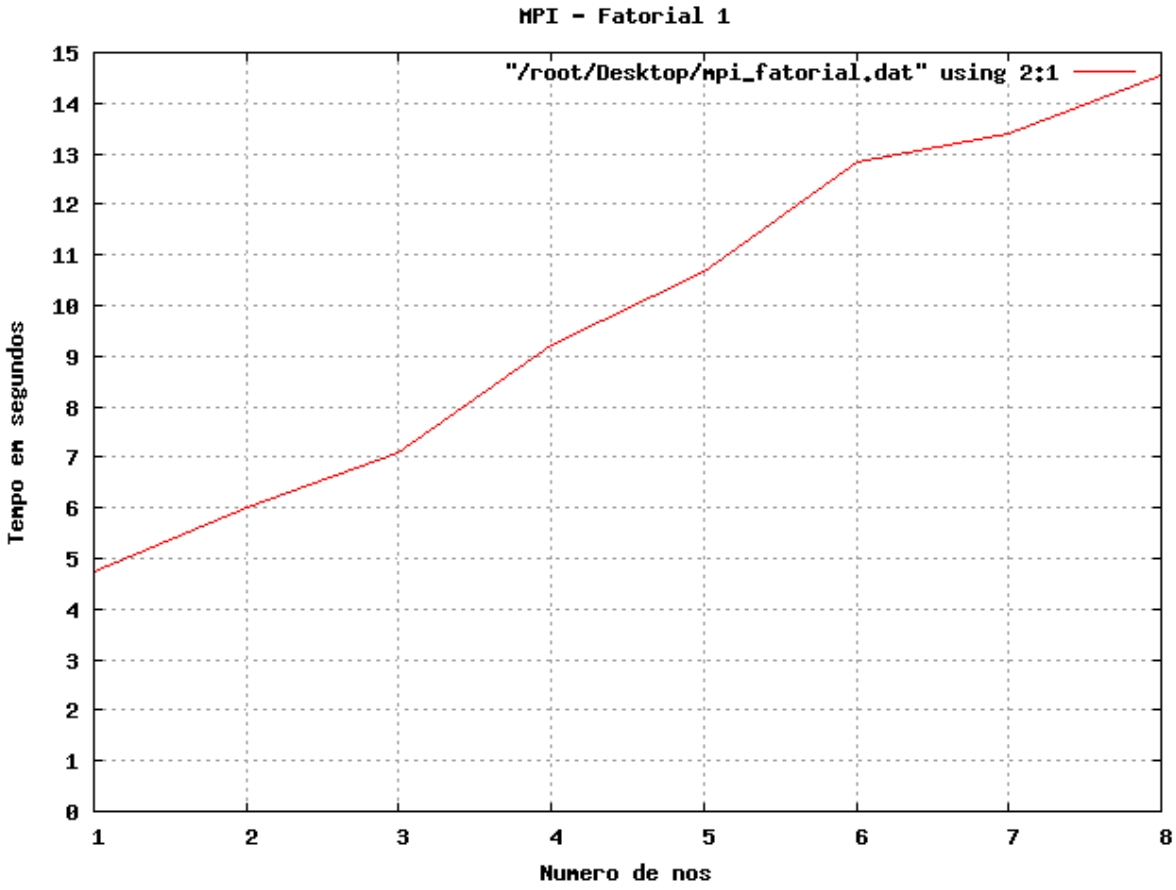

Figura 4.2: Gráfico de desempenho do algoritmo fatorial-1
Número de nós versus tempo gasto no processamento

O gráfico acima demonstra que conforme são adicionados novos nós na execução do programa, maior é o tempo de execução. Logo, a carga dos 100.001 cálculos do fatorial especificado está sendo paralelizada de forma não vantajosa.

O programa não está sendo eficiente em um *cluster* pela quantidade de comunicação ocorrendo para "pequenos" cálculos que ocorrem milhares de vezes. Assim, cada nó adicionado ao processamento faz com que a comunicação realizada através da operação "MPI_Reduce()" gaste mais tempo. Já que essa operação ocorre toda vez que o fatorial é repetido, o tempo de execução só cresce, sendo esse crescimento proporcional a adição de novos nós ao processamento.

Para resolver esse problema temos que realizar uma nova abordagem na solução dos cálculos efetuados.

Levando em consideração que cada fatorial realizado é uma operação "simples" para

cada máquina sozinha, e que, o tempo gasto na comunicação pela função “MPI_Reduce()” é muito alto quando o mesmo é utilizado milhares de vezes com muitos nós, vamos realizar uma nova divisão do problema. Só que, nesse novo programa, vamos dividir as 100.001 tarefas entre os processos, assim, cada computador irá realizar sozinho algumas dezenas de milhares de fatoriais, ao contrário do que ocorria anteriormente, onde cada fatorial tinha sua execução dividida pelo número de processos milhares de vezes.

Para tornar isso possível foram feitas algumas alterações no código fonte previamente apresentado.

```
// Biblioteca de entrada e saída da linguagem C
#include <stdio.h>
// Biblioteca de funções do MPI
#include "mpi.h"
// Declaração da função “fat” que realiza o cálculo fatorial
long double fat(int,int);
int main ( int argc, char *argv[] );

int main ( int argc, char *argv[] )
{
// Respectivamente o contador utilizado e o número de fatoriais a serem calculados
long int i=0,numfatoriais=100000;
/* Respectivamente número a ser fatorado, início do intervalo de fatoração e fim do
intervalo de fatoração */
int num=1754,ini,fim;
/* Respectivamente o resultado do fatorial e o resultado da parte a ser calculada que
nesse caso é igual ao resultado final*/
long double fatorial,parte;
/* Respectivamente o número de processos em execução e o identificador do processo
que está executando o programa */
int numprocs,rank;
/* Respectivamente o tempo de execução inicial e o tempo de execução final */
double tempoInicial = 0.0, tempoFinal;
// Inicialização do MPI
MPI_Init(&argc,&argv);
// Grava na variável “numprocs” a quantidade de processos sendo utilizados
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
// Grava na variável “rank” o número identificador do processo sendo executado
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
// O processo está sendo executado no computador principal (mestre)?
if(rank==0){
        // Pega o tempo de execução inicial no computador principal
        tempoInicial = MPI_Wtime();
}
/* Divide através de dois limites o total de cálculos a serem realizados no processo em
```

```
execução */
	/* O primeiro limite é a variável “ini”. Essa define qual é o primeiro cálculo a ser realizado pelo processo */
	ini = 1 + rank*(numfatoriais)/numprocs;
	/* O segundo limite é a variável “fim”. Essa define qual é o último cálculo a ser realizado pelo processo */
	fim = (rank+1)*(numfatoriais)/numprocs;
	// Realiza a iteração entre os limites especificados
	for(i=ini;i<=fim;i++){
		parte=0.0;
		fatorial=0.0;
		/* Já que só existe um processo realizando esse fatorial faremos o fatorial de 1 ao número “num”.*/
		parte=fat(1,num);
		/* Já que o fatorial é resolvido somente por um processo, a variável “parte” é igual a variável “fatorial” */
		fatorial = parte;
	}
	// O processo está sendo executado no computador principal (mestre)?
	if(rank == 0){
		// Grava na variável “tempoFinal” o tempo final de execução
		tempoFinal = MPI_Wtime();
		/* Imprime o número cujo fatorial foi determinado, o número de vezes que o mesmo fatorial foi resolvido e o resultado do último fatorial processado */
		printf("O fatorial de %d foi realizado %ld vezes e seu resultado foi %.11Le\n",num,numfatoriais, fatorial);
		/* Imprime o tempo que levou para o processamento ser realizado */
		printf("O processamento demorou %.16f\n",tempoFinal-tempoInicial);
	}
	// Finaliza o MPI
	MPI_Finalize();

	return 0;
	}

	long double fat(int ini,int fim)
	{
	// Contador
	int i;
	// Resultado obtido pelo fatorial dos dois limites passados
	long double soma=1;
	// Realiza a iteração entre os limites “ini” e “fim”
	for(i=ini;i<=fim;i++){
		/* Atribui à variável “soma” o valor do contador “i” multiplicado pelo seu próprio valor. Realizando assim, ao longo da iteração, o cálculo fatorial entre os limites especificados */
		soma=soma*i;
	}
```

```
// Retorna o valor do fatorial que foi calculado
return soma;
}
```

As principais alterações nessa nova versão do programa foram feitas no seguinte trecho.

```
ini = 1 + rank*(numfatoriais)/numprocs;
fim = (rank+1)*(numfatoriais)/numprocs;

for(i=ini;i<=fim;i++){
        parte=0.0;
        fatorial=0.0;

        parte=fat(1,num);

        fatorial = parte;
}
```

Agora a divisão de tarefas feito pelas variáveis *ini* e *fim* são baseadas na quantidade de fatoriais a serem realizados e não no número cujo cálculo fatorial deve ser realizado por vez (1754). Nessa nova abordagem cada processo em cada nó executa de forma independente (sem o uso do MPI_Reduce), dezenas de milhares de operações fatoriais. Assim o tempo de comunicação é reduzido a quase zero e a carga de processamento é distribuída de uma melhor forma entre os nós.

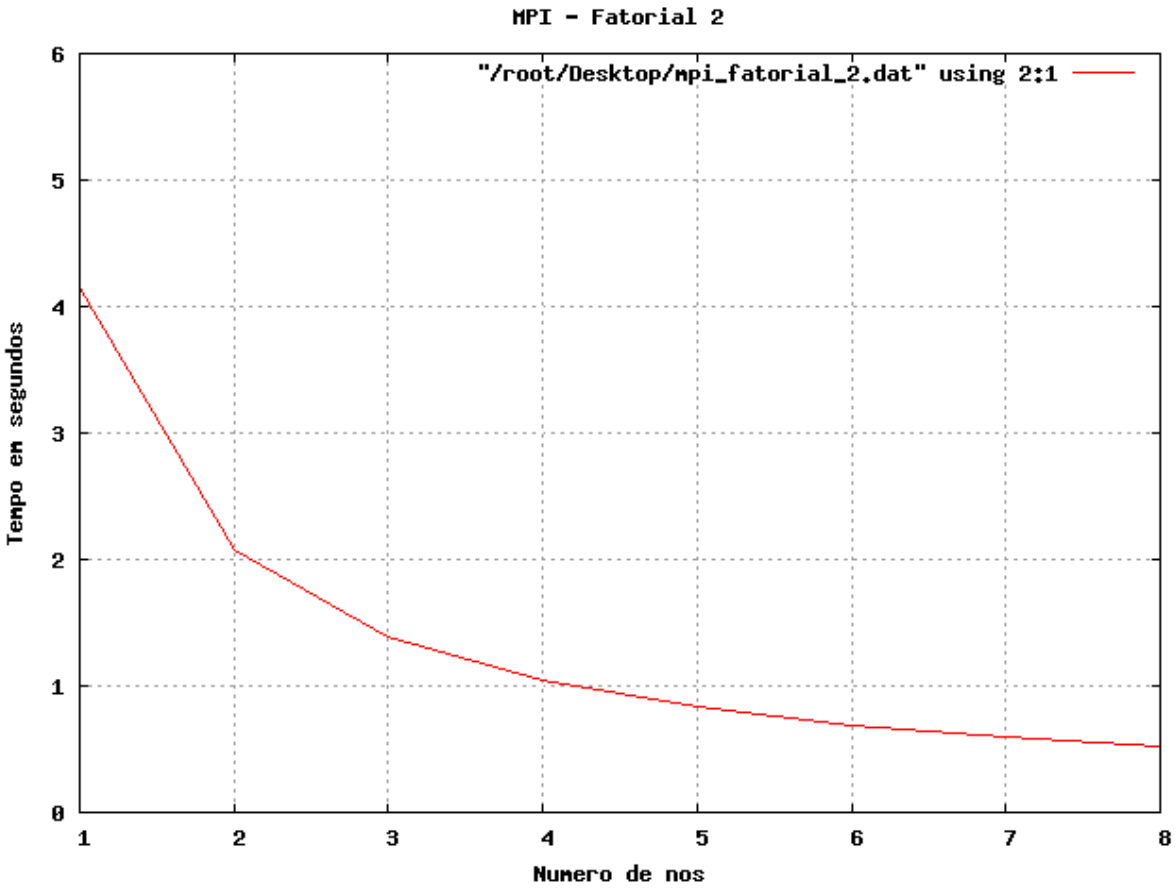

Figura 4.3: Gráfico de desempenho do algoritmo fatorial-2 número de nós versus tempo gasto no processamento

Como dito anteriormente, nem toda tarefa pode ser paralelizada de forma proveitosa. No primeiro algoritmo (fatorial-1) dividimos tarefas simples milhares de vezes e assim perdemos desempenho no sistema quando utilizado com mais de um nó. No entanto, no segundo algoritmo (fatorial-2), dividimos o trabalho a ser realizado em uma série de tarefas completas independentes. Assim a aplicação pôde ser executada em paralelo de forma vantajosa, aumentando assim o desempenho do sistema global, sendo várias vezes superior a de um nó sozinho.

5. Desempenho

Nesse trabalho vamos medir o desempenho do sistema através do programa HPL (*High-Performance Linpack*). Esse é um aplicativo que segundo sítio oficial (http://www.netlib.org/benchmark/hpl/, acesso: 15 maio, 2008), resolve em sistemas de memória distribuída cálculos envolvendo densos sistemas lineares aleatórios utilizando precisão de 64-bits. Este pacote contém um programa que pode ser utilizado para testes e como cronômetro de execução, para quantificar a precisão da resposta obtida e o tempo que o computador levou para resolver o problema.

Para que possa funcionar, esse *software* requer uma implementação do MPI (em conformidade com o padrão 1.1) e uma implementação do BLAS (*Basic Linear Algebra Subprograms*) ou VSIPL (*Vector Signal Image Processing Library*).

Optamos pelo pacote BLAS por ele possuir uma documentação mais extensa e um guia de instalação mais completo. Para instala-lo de forma otimizada utilizaremos o ATLAS (*Automatically Tuned Linear Algebra Software*). Esse pode ser descarregado no site oficial do ATLAS (ATLAS, 2008). Após o *download* mova o arquivo para um diretório onde possa descompactá-lo e construir o programa, em seguida vamos descompactar o arquivo. No nosso caso o diretório escolhido foi o /home/andre/libraries/. Para descompactar o arquivo utilize os seguintes comandos dentro do diretório:

```
gunzip atlas3.8.1.tar.gz
tar -xvf  atlas3.8.1.tar
```

Após essa etapa, os itens previamente compactados serão extraídos para uma nova pasta, no mesmo diretório, chamada de “ATLAS”. Dentro dessa criaremos uma outra pasta para separar os arquivos de instalação dos arquivos fontes. No nosso caso criamos a pasta chamada “buildATLAS”. Ao entrar na pasta “buildATLAS” vamos executar o programa de configuração do ATLAS. Normalmente não é necessária a passagem de outros parâmetros de configuração na instalação do ATLAS, no entanto no nosso caso, não utilizaremos o compilador Fortran, já que nossos programas só utilizarão a linguagem C. Isso deve ser passado para o utilitário de configuração, através do argumento “-Si nof77 1”. Para

visualizar as escolhas feitas na configuração, execute o “configure” passando os parâmetros “-v 2”, para assim visualizar a detecção do seu sistema. No nosso projeto utilizaremos o seguinte comando, dentro da pasta de construção do ATLAS (buildATLAS) para configura-lo.

/home/andre/libraries/ATLAS/configure -v 2 -Si nof77 1

Após essa etapa vamos compilar os códigos fontes com as configurações geradas. Para isso utilize o programa make dentro da pasta de construção. Esse comando pode demorar algumas horas para ser terminado. Ao término da etapa anterior a instalação estará finalizada. Caso queira testar o desempenho de sua instalação, execute o comando “make time” dentro da pasta de construção do programa. Esse comando retornará uma lista comparando o desempenho de sua instalação com outra que utiliza a mesma arquitetura. Esse comparativo é baseado em dados de referência do programa, e normalmente é representada por valores mais altos do que os alcançados pelo seu sistema.

Agora vamos instalar o HPL. Para isso, faça o *download* do mesmo do sítio oficial http://www.netlib.org/benchmark/hpl/hpl.tgz, vamos realizar o mesmo processo de descompactação realizado no pacote ATLAS. Criando no nosso caso uma pasta chamada “hpl” dentro /home/andre/libraries/. Para instalarmos o HPL precisamos criar um arquivo de configuração de nosso sistema. Nesse serão indicados os caminhos, dos compiladores, da implementação do MPI e da biblioteca BLAS, que serão utilizados. Existem vários arquivos de configuração que podem servir de modelo. Escolha o arquivo que tenha maior semelhança ao *cluster* em que está sendo instalado. No nosso projeto esse foi o ”Make.Linux_PII_CBLAS” (PentiumII utilizando linguagem C para o BLAS). Esse arquivo está dentro da pasta recém descompactada, dentro do diretório “*setup*”. Vamos copiar o arquivo para uma pasta acima (/hpl/), renomeando-o para ”Make.Linux_PIII_CBLAS”. Agora vamos editá-lo, seguem abaixo as partes editadas em negrito:

ARCH = **Linux_PIII_CBLAS**
#
--
- HPL Directory Structure / HPL library ------------------------------
--
#
TOPdir = **/home/andre/libraries/hpl**
INCdir = $(TOPdir)/include

```
BINdir       = $(TOPdir)/bin/$(ARCH)
LIBdir       = $(TOPdir)/lib/$(ARCH)
#
HPLlib       = $(LIBdir)/libhpl.a
#
# ----------------------------------------------------------------------
# - Message Passing library (MPI) --------------------------------------
# ----------------------------------------------------------------------
# MPinc tells the  C  compiler where to find the Message Passing library
# header files,  MPlib  is defined  to be the name of  the library to be
# used. The variable MPdir is only used for defining MPinc and MPlib.
#
MPdir        = /usr/local
MPinc        = -I$(MPdir)/include
MPlib        = $(MPdir)/lib/libmpich.a
#
# ----------------------------------------------------------------------
# - Linear Algebra library (BLAS or VSIPL) -----------------------------
# ----------------------------------------------------------------------
# LAinc tells the  C  compiler where to find the Linear Algebra  library
# header files,  LAlib  is defined  to be the name of  the library to be
# used. The variable LAdir is only used for defining LAinc and LAlib.
#
LAdir        = /home/andre/libraries/ATLAS/buildATLAS/lib
LAinc        =
LAlib        = $(LAdir)/libcblas.a $(LAdir)/libatlas.a
.........
```

Agora vamos executar o programa "*make"* na pasta /hpl/ passando como argumento o nome do arquivo de configuração. No nosso projeto, esse comando foi executado dessa forma:

```
make arch=Linux_PIII_CBLAS
```

Caso esse comando tenha sucesso, será criada uma pasta com o mesmo nome do parâmetro "arch" dentro da pasta /hpl/bin/. No nosso projeto essa pasta foi criada com o nome Linux_PIII_CBLAS dentro de /home/andre/libraries/hpl/bin/. Essa pasta irá conter um programa chamado "xhpl" e um arquivo de configuração do mesmo o "HPL.dat". Ambos serão utilizados nos testes de desempenho. O programa "xhpl" deve ser copiado para todas as máquinas do *cluster*, e na máquina principal, esse deve estar acompanhado do arquivo de configuração "HPL.dat"(ambos na mesma pasta). No nosso caso essa operação foi feita copiando-se ambos para a pasta /usr/local/bin/examples/ no computador central e depois executando o programa "distribuiPrograma.sh" para distribuir o programa xhpl. Depois de

finalizado o processo de cópia, pode-se executar o programa “xhpl” de forma paralela por meio do “mpiexec”.

O programa “xhpl” realizará uma série e de testes com a sua configuração padrão. No entanto o desempenho nesses não indicará o desempenho real do sistema utilizado. Pois esse não estará configurado para tirar o máximo de proveito da estrutura disponibilizada.

5.1 Configurando o XHPL

Para configurar o “xhpl”, precisamos modificar os parâmetros do arquivo “HPL.dat”. Esse arquivo é composto de 31 linhas, nas quais 29 correspondem a argumentos utilizados pelo programa. Configurar o HPL para obter o maior desempenho possível é um processo demorado, pois muitas configurações são escolhidas de forma empírica. Dependendo do volume de dados utilizado, o “xhpl” pode demorar dias para completar a execução, fato que dificulta ainda mais os testes de configuração.

Nesse trabalho modificamos somente as linhas principais desse arquivo. Acreditamos que por meio dessas modificações foi possível realizar um teste de desempenho perto do ideal. Começamos os ajustes na linha 4 do arquivo, modificando o valor 6 para 5. Essa modificação fará com que os resultados do teste sejam gravados em um arquivo, cujo o nome foi especificado na linha acima (HPL.out). Essa configuração é importante, porque muitas vezes não é possível visualizar os resultados obtidos de forma clara no terminal.

A linha 5 especifica a quantidade de problemas que o *cluster* terá de resolver durante a execução, essa constante é chamada de N e pode chegar ao valor máximo de 20. Para diminuir o tempo de espera de cada execução utilizamos aqui o valor 1. Na linha 6 será especificado o tamanho do problema a ser executado, nesse caso essa corresponde a quantidade de linhas ou colunas (matriz quadrada) que a matriz a ser utilizada terá. Já que vamos executar somente um problema por vez essa linha terá somente um valor. O cálculo do valor a ser utilizado segue uma fórmula que leva em conta a quantidade total de memória disponibilizada pelo *cluster*. Não podemos ultrapassar o máximo de memória disponível, pois caso isso ocorra, a partição de *swap* do disco rígido será utilizada e a performance cairá drasticamente, já que, o acesso ao disco é muito mais lento do que o acesso a memória principal.

No nosso projeto utilizamos a seguinte fórmula para chegar ao tamanho máximo aproximado do problema a ser executado.

$$N = \sqrt{\left(\frac{T * 0{,}8}{8}\right)} \tag{5.1}$$

Na fórmula 5.1 a variável T corresponde ao total de memória principal disponível no *cluster* em *bytes* e N corresponde ao tamanho máximo aproximado do problema. Abaixo utilizamos a fórmula em partes.

8X256MB = 2048MB * (1024^2) * 0.8 = 1717986918,4 bytes/8 bytes = 214748364,8 (204,8MB)

sqrt (214748364,8) = 14654

Utilizando 80% da memória total disponível na solução do problema, o "xhpl" pode realizar cálculos com, no máximo, uma matriz quadrada de 14654 linhas por 14654 colunas de elementos de precisão dupla (8 *bytes*). Esse cálculo informa um valor aproximado do máximo. Veremos mais adiante que a maior matriz executada com sucesso no *cluster* possui 14000x14000. Note que não podemos utilizar 100% da memória do sistema pois temos que deixar uma parcela da memória para o uso do sistema operacional.

Na linha 7 vamos configurar qual será a quantidade de blocos a serem executados pelo "xhpl". Esse valor deve ser um inteiro com o valor entre 1 e 20. No nosso caso utilizamos somente 1 para que o processo de teste demore menos. Na linha seguinte definimos o tamanho desse bloco. Esse é um dado que deve ser testado de forma empírica, pois deve ser aumentado de acordo com o tamanho do problema, para aumentarmos assim o desempenho.

Na linha 10 definimos em quantas configurações de *grids* a aplicação será executada. No nosso caso para serem realizados testes de desempenho mais rapidamente utilizamos somente um valor de *grid*. As próximas duas linhas devem ser preenchidas de acordo com a linha 10. No nosso caso utilizamos o valor 2 para Ps e 4 para Qs. Isso quer dizer que o "xhpl" será executado em um *cluster* com a configuração na forma de uma grade 2x4.

Na fase de configuração do *cluster* utilizamos um valor de *threshold* negativo (linha 13). Assim o "xhpl" não realiza testes de precisão sobre os resultados obtidos, o que deixa o processo de configuração mais rápido.

5.2 Testes realizados

O HPL precisa de no mínimo 4 nós para ser executado. Os testes feitos nesse projeto utilizam o a potencial total do *cluster* (8 nós).

Foram realizados vários testes de desempenho com diversas configurações. Nesse trabalho mostraremos as configurações utilizadas nesses testes e o desempenho atingido nos mesmos.

Na tabela 5.1 a coluna “N” corresponde ao tamanho da matriz utilizada em número de linhas. A segunda coluna corresponde ao tamanho dos blocos de execução utilizados. A coluna “GRID” indica que tipo de configuração de grade computacional foi aplicada na execução do HPL. A quarta coluna corresponde a porcentagem de memória utilizada em cada nó durante a execução e a última coluna corresponde ao melhor desempenho obtido na execução de cada teste. Vale ressaltar que cada Flop medido pelo HPL representa uma operação de ponto flutuante em precisão dupla.

Tabela 5.1: Valores utilizados nos testes realizados com o High Performance Linpack

N	NB	GRID	Memória	GFlops
2500	54	2x4	4%	0,881
2500	72	2x4	4%	0,912
2500	100	2x4	4%	0,854
5000	72	2x4	11%	1,238
5000	100	2x4	11%	1,250
5000	108	2x4	11%	1,232
7500	100	2x4	27%	1,462
7500	132	1x8	27%	1,368
7500	132	2x4	27%	1,469
7500	192	2x4	27%	1,447
10000	132	2x4	41%	1,606
10000	164	2x4	41%	1,624
10000	180	2x4	41%	1,612
13500	164	2x4	77%	1,780
14000	164	2x4	80%	1,787
14500	150	2x4	88%	ERRO
14500	164	2x4	88%	ERRO

O desempenho máximo medido (Rmax) foi de 1,787 Gflops realizando operações de precisão dupla com uma matriz de 14000 linhas. Os testes realizados nas duas últimas linhas da tabela apresentaram “ERRO” na execução, por falta de memória principal.

Como podemos ver na tabela 5.1, a medida que aumentamos o tamanho da tabela utilizada maior é a performance do sistema, no entanto para isso acontecer deve utilizar um crescente NB que seja compatível com o tamanho do problema sendo resolvido. O valor dessa variável é determinada de forma empírica e por isso foram realizados vários testes com tamanhos variados para determinarmos o NB ideal para cada tipo tamanho de matriz. O valor máximo que permite um maior desempenho para *cluster* construído foi de 164. Acima desse valor os testes resultaram em erro ou em quedas de desempenho medido em Gflops.

Os valores de desempenho apresentados na tabela 5.1 correspondem aos valores máximos obtidos em cada teste. Durante cada teste são realizados séries de operações e cada uma possui um desempenho diferente, medido em GFlops. Para termos uma melhor noção do

desempenho do *cluster* na execução de cada teste, foi construída uma tabela com os resultados de desempenho médio para cada teste realizado.

Tabela 5.2: Valores utilizados nos testes realizados e a performance média obtida

N	NB	GRID	Memória	Média em GFlops
2500	54	2x4	4%	0,876
2500	72	2x4	4%	0,904
2500	100	2x4	4%	0,850
5000	72	2x4	11%	1,235
5000	100	2x4	11%	1,245
5000	108	2x4	11%	1,228
7500	100	2x4	27%	1,458
7500	132	1x8	27%	1,364
7500	132	2x4	27%	1,463
7500	192	2x4	27%	1,442
10000	132	2x4	41%	1,600
10000	164	2x4	41%	1,617
10000	180	2x4	41%	1,607
13500	164	2x4	77%	1,776
14000	164	2x4	80%	1,781

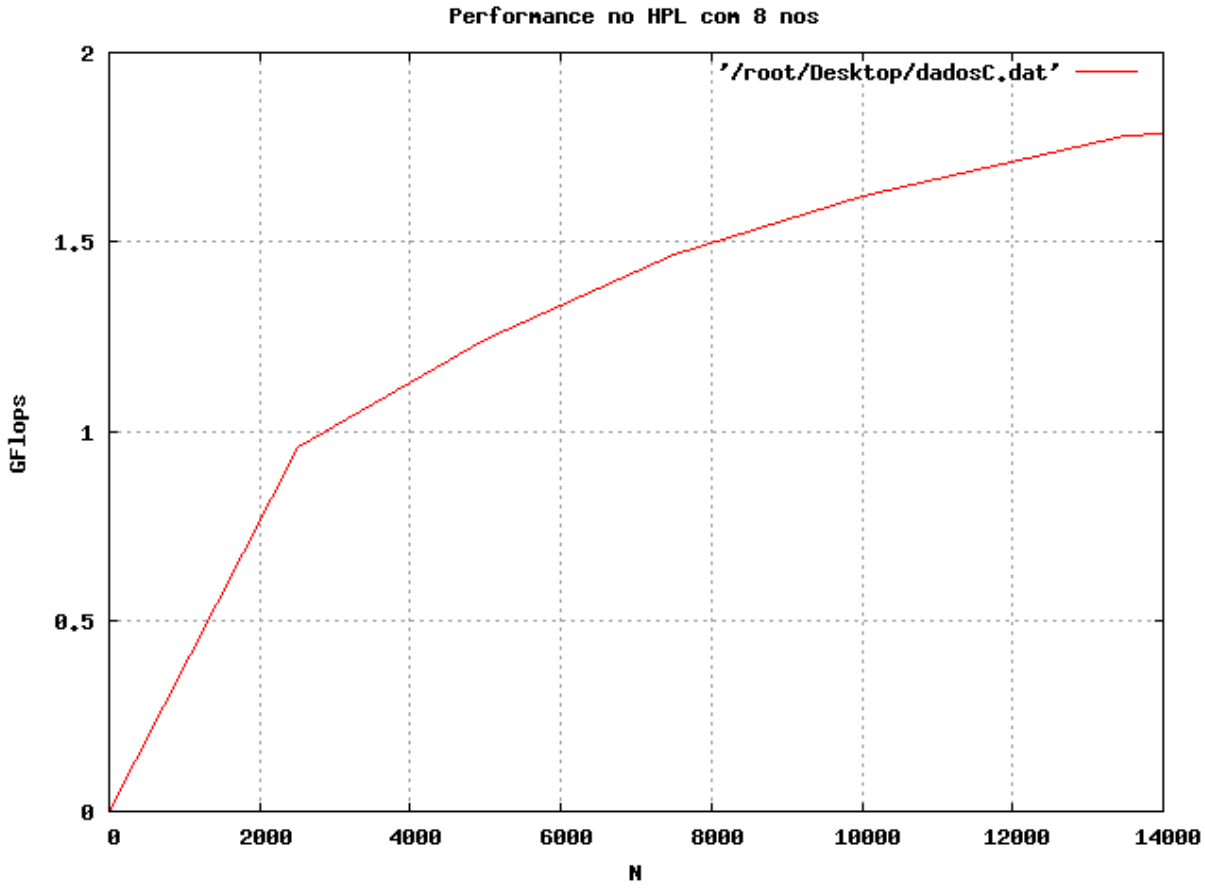

Figura 5.1: A relação entre o valor de GFlops obtido e o tamanho da matriz utilizada

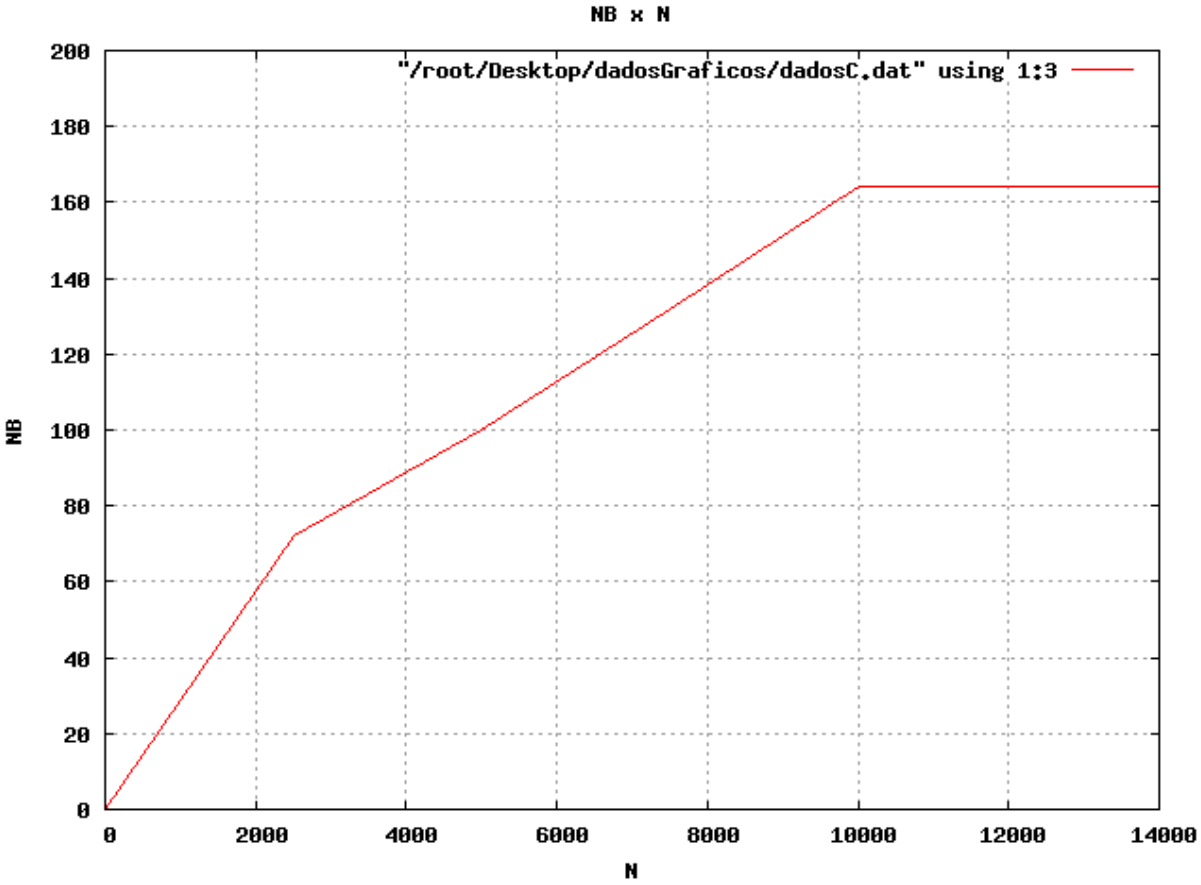

Figura 5.2: A relação entre o valor de NB e o tamanho da matriz utilizada enquanto aumenta-se o desempenho do sistema

O desempenho teórico atingido pelo *cluster* é igual ao número de operações de ponto flutuante de precisão dupla por ciclo de *clock*, multiplicado pela soma do *clock* de todos os processadores envolvidos. No nosso projeto o cálculo foi esse.

Rpeak = 500Mhz x 8 x 1Flop de precisão dupla/ciclo = 4GFlops

O desempenho teórico do sistema será sempre maior do que a performance máxima obtida. Isso é explicado pelo fato do desempenho teórico não levar em consideração o desempenho da rede utilizada. Caso fossem utilizadas duas placas de rede em cada nó o *Rmax* (desempenho máximo medido pelo HPL), seria maior, ficando mais próximo do teórico.

A eficiência do *cluster* é a medida do desempenho máximo obtido sobre o desempenho teórico.

Eficiência = *Rmax* / *Rpeak*= 1,8/4,0 = 0,45

Logo, a eficiência do *cluster* construído chega ao máximo aproximado de 45% em relação ao seu desempenho teórico.

Conclusão

Ao final do projeto foi efetuada a construção de um *cluster* Beowulf contemplando todas as características essenciais de sua construção. Essas são:

- Os nós e a rede utilizados são de uso exclusivo do *cluster*.
- Os nós e a rede utilizados são compostos de tecnologias convencionais, não especializadas para o fim proposto.
- Todos os nós utilizaram somente *softwares* de código aberto.
- O *cluster* é utilizado para computação de alto desempenho (CAD).

A aplicação desenvolvida prova que o processamento paralelo é vantajoso, se o algoritmo for implementado de forma correta. Nesse projeto uma máquina sozinha demorou cerca de 4.144 segundos para terminar a execução do programa desenvolvido, enquanto os 8 nós trabalhando em paralelo foram capazes de reduzir esse tempo para 0.52 segundos. Logo, o uso do cluster para o processamento paralelo ofereceu um ganho de aproximadamente 8 vezes na aplicação realizada. Isso prova que 8 nós processando em paralelo executam o mesmo programa cerca de 8 vezes mais rápido do que uma máquina sozinha.

O desempenho do sistema apresentou acréscimos contínuos a medida que o tamanho do problema a ser resolvido pelo *cluster* era aumentado. Essa mudança foi acompanhada pelo aumento do tamanho dos blocos de execução enviados pela rede que chegaram ao tamanho máximo de 164. O tamanho da matriz utilizada chegou ao valor máximo de 14000X14000, sendo que acima desse valor o processamento resultou em erro de execução.

O desempenho máximo atingido foi 1,787GFlops que representa aproximadamente 45% do desempenho teórico do *cluster* utilizado. O resultado foi interpretado de forma positiva já que a rede é o fator limitante da eficiência do *cluster* utilizado e que cada nó utilizou somente uma placa de rede com taxa de transferência máxima igual a 100Mbps, o que teoricamente causaria um impacto maior no desempenho do sistema.

Além das conclusões relativas aos objetivos do trabalho podemos ressaltar que, o *cluster*

Beowulf é um supercomputador com grande potencial computacional. Isso pode ser comprovado por intermédio da lista dos 500 supercomputadores mais poderosos do mundo (www.top500.org), onde aparentemente não existem limitações arquiteturais aos multicomputadores Beowulf, podendo-se construir sistemas com centenas de milhares de processadores e com centenas de milhares de GB de memória principal. Tudo de forma facilitada pelo uso de tecnologias convencionais e de software de código aberto que, a qualquer momento podem ser re-configurados de forma a agradar o gosto do freguês. Apesar de todas as outras, a característica mais marcante do *Beowulf* é o custo/performance, fator que leva muitas companhias a investirem nesse tipo de tecnologia.

Esse foi o primeiro projeto do UniCeub a abordar tópicos como *cluster Beowulf* e processamento paralelo. Nos próximos anos, vários outros estudantes de várias faculdades, poderão aproveitar o *cluster* já construído, no laboratório de *hardware* 8006, para realizar seus projetos. Podendo ser utilizado como uma ferramenta, resolvendo grandes problemas, ou como base para novos projetos, tanto de alunos da engenharia de computação como da ciência da computação.

Como sugestões de projetos futuros podem ser realizados a disponibilização dos recursos do *cluster* através da rede do UniCeub por meio de um programa de gerenciamento dos recursos do mesmo, pode ser construída uma grade computacional (*Grid*), seja dividindo o *cluster* construído em partes ou criando um novo *cluster* para que seja formado um *grid* de supercomputadores, podem ser criados muitos tipos de aplicações para serem implementadas utilizando o *cluster* construído, como *web farms*, bancos de dados distribuídos entre muitas outras.

Referências Bibliográficas

- ATLAS. **Automatically Tuned Linear Algebra Software** .Disponível em: <http://downloads.sourceforge.net/math-atlas/atlas3.8.1.tar.gz?modtime=1203673630&big_mirror=0> . Acesso em 05/05 de 2008.

- BROWN, ROBERT G. **Engineering a Beowulf-style Compute Cluster**. 1 ed. Durham. Disponível em:<http://www.phy.duke.edu/~rgb/Beowulf/beowulf_book/beowulf_book.a4.pdf>. Acesso em 5 maio. 2008. 16:30.

- Computer Science Departmente – Stony Brook. **Compiling with Math Library**. Disponível em: <http://www.cs.sunysb.edu/facilities/FAQ/UnixCompilingMathLibrary.html> . Acesso em 15/05 de 2008.

- DE ROSE, CÉSAR A.F. ; Navaux, Philippe O.A. . **Arquiteturas Paralelas**. Porto Alegre: Sagra Luzzatto, 2003.

- Debian Admin. **SSH your Debian servers without password.** Disponível em: <http://www.debianadmin.com/ssh-your-debian-servers-without-password.html> . Acesso em 12/05 de 2008.

- Debian. **O sistema operacional universal** .Disponível em:

<http://www.debian.org/index.pt.html> . Acesso em 05/05 de 2008.

- Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy. **Revisiting Iterative Refinement for Linear Systems**. Disponível em:<http://www.cs.utk.edu/~library/TechReports/2006/ut-cs-06-574.pdf> .Acesso em 17/05 de 2008.

- Gnuplot Tips. **2-D Plot** .Disponível em: <http://t16web.lanl.gov/Kawano/gnuplot/index-e.html . Acesso em 28/05 de 2008>.

- GROOP, WILLIAN; LUSK, EWING; STERLING, THOMAS. **Beowulf Cluster Computing with Linux**. 2 ed. Cambridge: The MIT Press, 2003.

- Guia do hardware. **Usando o partimage**. Disponível em: <http://www.guiadohardware.net/tutoriais/usando-partimage/> .Acesso em 12/05 de 2008.

- High Performance Computing. **Performance Characteristics of Intel Architecture based Servers**. Disponível em: <http://www.dell.com/downloads/global/power/4q03-ali.pdf> . Acesso em 17/05 de 2008.

- HPL. **HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers**. Disponível em: <http://www.netlib.org/benchmark/hpl/> . Acesso em: 5 jun. 2008.

- Huss, Eric. **The C Library Reference Guide**. Disponível em:

<http://www.acm.uiuc.edu/webmonkeys/book/c_guide/> . Acesso em: 8 jun. 2008.

- IBM. **High performance Linux clustering, Part 2: Build a working cluster** . Disponível em: <http://www.ibm.com/developerworks/linux/library/l-cluster2/index.html> . Acesso em 15/05 de 2008.

- IBM. **Parallel Environment (PE) library** . Disponível em: <http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=/com.ibm.cluster.pe431.mpiprog.doc/am106_pmd.html> . Acesso em: 06/06 de 2008.

- KARNIADAKIS, GEORGE EM; KIRBY II, ROBERT M. .**Parallel Scientific Computing in C++ and MPI:** A seamless approach to parallel algorithms and their implementation. Cambridge: Cambridge University Press, 2003.

- MOTA FILHO, JOÃO ERIBERTO. **Descobrindo o Linux: Entenda o sistema operacional GNU/Linux**. 2 ed. São Paulo: Novatec Editora, 2007.

- MPICH2. **MPICH2: High-performance and Widely Portable MPI** .Disponível em: <http://www.mcs.anl.gov/research/projects/mpich2/> . Acesso em 05/05 de 2008.

- PATTERSIB, DAVID A.; HENNESSY, JOHN L.. **Organização e projeto de computadores: A interface HARDWARE/SOFTWARE**. 2 ed. Rio de Janeiro: LTC, 2000.

- Pentium III = Pentium II + SSE. **Internet SSE Architecture Boosts Multimedia**

Performance. Disponível em: <http://studies.ac.upc.edu/ETSETB/SEGPAR/microprocessors/pentium3%20(mpr).pdf>. Acesso em 17/05 de 2008.

- SIMA, DEZSÕ; FOUNTAIN,TERENCE; KACSUK, PÉTER. **Advanced computer architectures: A design space approach**. Harlow: Addison Wesley Longman, 1997.

- SLOAN, JOSEPH D. **High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI**. Sebastopol: O'Reilly Media, 2005.

- TANENBAUM, ANDREW S. . **Sistemas operacionais modernos**. 2 ed. São Paulo: Pearson Education do Brasil, 2003.

- TANENBAUM, ANDREW S. .**Organização estruturada de computadores**. 4 ed. Rio de Janeiro: LTC, 2001.

- TOP500. **TOP 500 Supercomputer Sites**. Disponível em: <http://www.top500.org/> . Acesso em: 5 jun. 2008.

Apêndice A – Processo de instalação do Debian

Com o CD-R(W) gravado, configuramos a BIOS do sistema para realizar *boot* pelo CD-ROM. Para realizar esse procedimento aperte a tecla Delete depois que a máquina esteja ligada, e surgirá um menu com as configurações da BIOS. O programa de configuração da BIOS varia de placa mãe para placa mãe, no nosso caso entramos no item *Advanced Setup* e mudamos com as teclas + ou -, o *1st Boot Device* para CD-ROM. Feito isso aperte a tecla Esc e saia do programa escolhendo o item *Exit*. O computador reiniciará irá ser inicializado pelo CD-ROM.

A primeira tela da instalação é a tela de boot do Debian onde aparece o seu logo, aperte F1 para visualizar todas as opções de inicialização.

Figura A1: Tela de inicialização do instalador Debian.
Fonte: http://wiki.forumdebian.com.br/images/b/b9/Netinstall01.jpg , acesso: 11 de junho 2008.

Aperte a tecla F3, digite install e aperte a tecla “Enter” para instalar o Debian utilizando o

modo SHELL. Não foi possível por limitações de hardware instalar por meio do modo gráfico “installgui”, logo recomendo o uso modo SHELL pois esse é mais leve.

Na primeira tela da instalação pelo modo SHELL você deve escolher a língua a ser utilizada no procedimento de instalação e no sistema final. Existem muitas línguas, no nosso caso escolhemos o Português do Brasil. Na próxima tela escolha o país, região ou área. No nosso caso escolhemos o Brasil. Na próxima tela deve-se escolher o padrão de teclado a ser utilizado, para nós brasileiros existe uma regra informal básica que quase sempre funciona, se o teclado possuir a tecla “Ç”, escolha o padrão Português Brasileiro (br-abnt2), caso contrário utilize a Português Brasileiro (br-latin1). Na próxima tela ele irá começar o procedimento de instalação detectando o *hardware*, posteriormente carregando componentes adicionais e detectando os componentes de rede.

No nosso caso não possuímos um servidor DHCP, logo a rede não será configurada automaticamente. Nesse caso será emitido um aviso relatando esse fato, pressione a tecla “Enter” e depois escolha a opção “Configurar a rede manualmente”. Agora vamos colocar o IP do computador, no nosso caso vamos utilizar um IP proveniente da rede já estabelecida no laboratório para assim conseguir acesso á internet, requisito nesse tipo de instalação. O endereço da rede já estabelecida é 172.18.0.0 com a máscara de rede 255.255.0.0 e *gateway* padrão 172.18.0.1. Escolhemos nesse trabalho começar com o IP 172.18.86.115. Depois será perguntado a máscara de rede que no nosso caso será 255.255.0.0. Na próxima tela deve ser definido o endereço do *gateway* que no nosso caso é 172.18.0.1. Depois serão perguntados os endereços dos servidores de nomes, nesse pode-se deixar o endereço do gateway, no nosso caso foi necessário adicionar além desse o servidor de DNS do Uniceub 192.168.2.5. Após essa tela será perguntado o nome de máquina, no nosso projeto foram utilizados o nome mestre para o nó principal e no1 à no7, para os nós “escravos”. Depois escolha o nome do seu domínio a ser utilizado, no nosso caso o domínio foi “cluster.uniceub”.

Após a configuração de rede vamos a configuração do disco. Essa etapa da instalação começa pelo particionamento dos discos, na primeira tela, foi escolhida a opção “Assistido - usar disco inteiro” já que essas máquina serão dedicadas ao uso do *cluster*. Na próxima tela serão listados os discos rígidos detectados pelo sistema, deve ser escolhido o disco a ser instalado o Debian, no nosso caso, cada máquina só possui um HD, o que facilita a escolha. Na tela seguinte escolheremos o esquema de particionamento do disco, no nosso caso para

garantimos um espaço para o *swap* e para que seja facilitado o *backup* de partições distintas no futuro, utilizamos à opção "Partições home, /usr, /var e /tmp separadas". Esse tipo também é recomendado para sistemas multi-usuários, esse seria mais um motivo para essa escolha, pois no futuro, o computador central deverá ser acessado por vários usuários ao mesmo tempo. Após a escolha o programa de instalação fará um modelo de particionamento do disco e mostrará o resultado. No nosso caso o mestre ficou com a seguinte configuração:

IDE1 principal (hda) – 20.8 GB Maxtor 6E020L0

#1 primária	279.6 MB **B**	**f ext3**	/
#5 lógica	5.0 GB	**f ext3**	/usr
#6 lógica	3.0 GB	**f ext3**	/var
#7 lógica	740.2 MB	f swap	swap
#8 lógica	403.0 MB	**f ext3**	/tmp
#9 lógica	11.4 GB	**f ext3**	/home

A *flag* "**B**"(em negrito acima) na partição primária quer dizer que essa partição será a partição de *boot* (onde o sistema será inicializado) o "**f ext3**" é a especificação do tipo de formatação que essas partições terão. Caso concorde com o modelo apresentado escolha a opção "Finalizar o particionamento e gravar as mudanças no disco". Essa escolha tem que ser confirmada na tela seguinte.

Após o particionamento escolha uma cidade com base em seu fuso horário, já que São Paulo possui o mesmo fuso horário que o nosso escolheremos São Paulo.

Na próxima tela deve ser definida a senha do usuário *root*, essa deve ser escolhida com cuidado, pois o usuário *root* pode realizar alterações em qualquer arquivo do sistema operacional, o que é potencialmente perigoso. As senhas de *root* e de usuário desse *cluster* serão entregues ao final do projeto em um anexo separado ao professor orientador. Após a senha de *root* e confirmação da senha, serão requisitadas o nome completo para o novo usuário (além do *root* o debian precisa ter um usuário). Após o nome de login para sua conta, será requisitada a senha e a confirmação de senha para sua conta.

Após a instalação do sistema básico, será configurado o gerenciador de pacotes APT (*Advanced Package Tool*). Na primeira tela da configuração será perguntado se você deseja utilizar espelho de rede, essa opção possibilita a instalação de vários pacotes adicionais e *updates* do sistema. Para deixarmos o sistema o mais leve possível não utilizaremos espelho de rede, assim escolheremos a opção "não". A tela seguinte será de advertência, dizendo que

não foi possível baixar as atualizações de segurança do Debian, esse fato foi causado pela não utilização do espelho de rede. Não se preocupe, após a instalação podemos efetuar o *update* do sistema básico e com isso as atualizações de segurança. Escolha a opção “Continuar”.

Por causa da falha na atualização da segurança o instalador voltará ao menu principal onde deve ser escolhido o item “Selecionar e instalar *software*”, na tela seguinte haverão duas opções de software a ser instalado, Laptop e/ou sistema básico, no nosso caso escolhemos “Sistema básico” e pressionamos a tecla “Enter” para continuar. A opção representa a instalação dos programas básicos do sistema operacional.

Finalizada a etapa acima vamos instalar o GRUB em um disco rígido, na primeira tela desse item, pergunta-se se desejamos instalar o carregador de inicialização GRUB no registro de inicialização principal, já que nessas máquinas só haverá um sistema operacional, respondemos “Sim”. Após o término da instalação do GRUB, a instalação do Debian será finalizada. O CD-R(W) será ejetado e um aviso surgirá na tela, deve-se escolher a opção “Continuar” para reiniciar o computador no novo sistema operacional, o Debian Ecth. No entanto para que isso aconteça deve-se re-configurar a BIOS para realizar o *boot* por meio do disco rígido. Entre na BIOS e mude o *1st Boot Device* para a interface do disco rígido onde foi instalado o Debian, no nosso caso essa interface é a IDE-0.

Apêndice B - Problemas com a rede

O primeiro passo para configurar a rede no sistema operacional GNU/Linux distribuição Debian 4.0 (Etch) ou na distribuição Slax, é verificar se a interface de rede está ativada. Para isso execute o comando "ifconfig". Este listará as interfaces de rede ativas. Caso nenhuma interface esteja habilitada esse só retornará o endereço de *loopback* da placa de rede, representado pelo nome de interface "lo". Para ver todas as interfaces de rede, habilitadas ou não, execute o comando "ifconfig -a". Caso não apareça nenhuma interface de rede com o nome, eth0, eth1, eth2, e assim sucessivamente, sua placa de rede não está ativada ou apresenta algum defeito de *hardware*. Caso a interface apareça somente no comando "ifconfig -a" temos que ativa-la.

Para ativar uma interface de rede utilize o comando:

ifconfig <nome_da_interface_de_rede> up

Agora vamos atribuir um endereço IP para essa e uma máscara de rede. Podemos realizar essas atribuições por meio do "ifconfig". Pelo comando:

ifconfig eth0 172.18.86.115 netmask 255.255.0.0

Agora vamos testar as configurações acima. Para isso vamos utilizar o comando "ping". Esse possui a seguinte sintaxe:

ping <ip_válido>

No nosso caso, utilizaremos o IP do "no1".

ping 172.18.86.116

O comando acima deve retornar respostas do computador com o endereço IP especificado, se o mesmo estiver ligado e não tiver um *firewall* bloqueando o comando "ping".

Agora vamos configurar o acesso a *internet*. No nosso projeto não é necessário o uso

dessa em nós funcionando com o *Live-CD* Slax, portanto essas configurações serão específicas para a distribuição Debian.

Após a termos configurado da rede, o acesso a *internet* deve funcionar automaticamente. No entanto, caso não funcione, devem ser checados o roteamento utilizado na rede e o IP do servidor de DNS.

Para visualizar as rotas existentes, utilize o comando “route -n”, caso nenhum *Gateway* esteja definido, execute o comando abaixo especificando o endereço do *gateway* de sua rede:

route add default gw <endereço_do_gateway>

O comando acima define um *gateway* padrão para as comunicações. Os dados do servidor de DNS, podem ser definidos por meio de dois arquivos o /etc/resolv.conf ou o /etc/network/interfaces. Caso no arquivo “interfaces” os dados de DNS estejam comentados o arquivo “resolv.conf” deve ser alterado, caso contrário altere os valores apresentados no arquivo “interfaces”. Esse deve conter o nome do domínio utilizado e o endereço dos servidores de DNS. Ao edita-lo o acesso a *internet* deve ser restabelecido.

Em várias ocasiões as configurações de rede não são carregadas automaticamente, logo o serviço de rede deve ser reinicializado. Para isso utilize o comando:

/etc/init.d/networking restart

Deve ser observado que comando “ifconfig” realiza uma configuração temporária, que é ideal para testes de conexão, ao ser reinicializado o computador perderá essas configurações. Para manter a rede configurada, preencha os dados de configuração no arquivo /etc/network/interfaces, como especificado na instalação do sistema.