ESCOLA POLITÉCNICA
DE PERNAMBUCO

d.s.c. UPE
Departamento de Sistemas e Computação

Avaliação de Desempenho de Ferramentas de Renderização de Imagens em *Clusters openMosix* e Arquiteturas *Multicore*

Trabalho de Conclusão de Curso

Engenharia da Computação

Péricles da Silva Bastos Sales
Orientador: Abel Guilhermino da Silva Filho
Co-Orientador: Wellington Pinheiro dos Santos

Recife, junho de 2008

UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

d.s.c. UPE
Departamento de Sistemas e Computação

Avaliação de Desempenho de Ferramentas de Renderização de Imagens em *Clusters openMosix* e Arquiteturas *Multicore*

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Péricles da Silva Bastos Sales
Orientador: Abel Guilhermino da Silva Filho
Co-Orientador: Wellington Pinheiro dos Santos

Recife, junho de 2008

UNIVERSIDADE
DE PERNAMBUCO

Péricles da Silva Bastos Sales

Avaliação de Desempenho de Ferramentas de Renderização de Imagens em *Clusters openMosix* e Arquiteturas *Multicore*

Resumo

As aplicações que são utilizadas nos dias de hoje exigem cada vez mais um grande poder de processamento e, nesse contexto, estão contidas as ferramentas de renderização de imagem. Com isso, a utilização de *clusters* e arquiteturas *multicore* já são uma tendência. Nesse trabalho, é feita uma avaliação do desempenho de ferramentas de renderização de imagens digitais fazendo-se uso de *clusters openMosix* e de computadores que possuem arquiteturas *multicore.*

Além do aprofundamento teórico feito com os assuntos pertecentes ao tema, foi feito um estudo de caso, usando principalmente a ferramenta de renderização denominada PovRay. Foi constatado através de experimentos realizados que o aplicativo de renderização em questão não conseguia tirar proveito nem da estrutura do *cluster* nem dos computadores de arquiteturas *multicore*, assim foi observado que esse aplicativo não era capaz de dividir a carga.

Nas pesquisas realizadas encontrou-se um aplicativo denominado POVmosix, que foi desenvolvido especialmente para paralelizar a renderização feita pelo PovRay, dividindo uma cena em um conjunto de sub-tarefas, que serão processos independentes, assim conseguindo a migração de processos entre os nós do *cluster* e fazendo uso dos núcleos presentes nos computadores *multicore*. Posteriormente, foi feita uma avaliação de desempenho utilizando a integração dessas duas ferramentas (PovRay-POVmosix) consistindo em uma seqüência de testes afim de se calcular o tempo de renderização de uma cena de exemplo. Concluiu-se que no *cluster* o desempenho melhora com aumento de número de nós até chegar um ponto que se estabiliza ou, por vezes, tem-se uma piora. No caso dos computadores *multicores,* foi observado que quando dividimos a renderização em um número de sub-tarefas igual ao número de núcleos, esta integração de ferramentas consegue usufrir de todos os núcleos presentes.

Abstract

The applications commonly used in these days require increasingly processing power and, in this context, are contained the tools to render images. Thus, the use of clusters and multicore architectures are already a tendency. In this work, is conducted a performance evaluation of tools to render digital images, using openMosix clusters and computers with multicore architecture.

In addition to deepening theoretical issues within the theme´s subject, it was made a case study using mainly the PovRay image rendering tool. It was observed that this rendering tool cannot take advantage of the cluster structure neither of the structure of computers with multicore architecture, therefore it was concluded that this application was not able to share the load.

In further researches, an application called POVmosix was found. This application was specially developed to parallelize the image rendering made by PovRay, splitting one scene into a number of sub-tasks, which are independent processes, thus making possible the migration of processes between the nodes of the cluster and making use of multiple cores in multicore computers. Subsequently, it was made a performance evaluation using the integration of these two tools (PovRay-POVmosix) consisting of a sequence of tests using one sample scene. It follows that when increasing the number of nodes of the cluster until a certain number, a performance gain on the image rendering could be realized. Beyond this number, the performance of the image rendering stabilizes or produces worse values. In the case of computers with multicore architecture, it was observed that when splitting the image rendering in a number of sub-tasks equal to the number of cores, the integration of those two rendering tools can make use of every single core of the multicore computer.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

CPU - *Central Processing Unit*

SMP - *Symmetric Multiprocessing*

E/S – Entrada e saída

S.O - Sistemas Operacionais

IP - *Internet Protocol*

TCP - *Transmission Control Protocol*

NFS - *Network File System*

DFSA - *Direct File System Access*

PVM - *Parallel Virtual Machine*

MPI - *Message Passing Interface*

PPE - *Parallel Programming Environment*

SSI - *Single System Image*

MPMD - *Multiple Program Multiple Data*

DHCP - *Dynamic Host Configuration Protocol*

Agradecimentos

Agradeço a todos os professores do Departamento de Sistemas e Computacão por contribuírem para minha formação acadêmica e moral. Agradecimentos especiais ao Prof. Abel Guilhermino e ao Prof. Wellington Pinheiro por seus ensinamentos e orientações neste trabalho.

Agradeço também a alguns colegas que fiz no decorrer da escrita dessa monografia, que apesar de distantes contribuíram bastante para realização deste projeto, como: O grande autor na área de *Clusters,* Marcos Pitanga; Alex Martini, que possui artigo na área que fiz minha monografia e Jean Rodrigo que já fez um trabalho de conclusão em área semelhante.

A todos os colegas e amigos com quem convivi nesses 5 anos de faculdade. Não podendo deixar de citar os que foram meus "braços-direito" no decorrer desses anos de vida acadêmica: André Luís, Axel Saraiva, Wilmar Feijó, Salomão Sampaio, Ricardo Ulisses e Rafael Albuquerque. Citando também alguns amigos que sempre estiveram me acompanhando nessa jornada: Mateus Peregrino, Sérgio Guerra, Thiago Brayner e Rubens José.

Capítulo 1

Introdução

A utilização de *clusters* e de arquiteturas *multicore* é uma tendência na área de arquitetura de computadores. As aplicações utilizadas atualmente exigem cada vez mais poder de processamento e nesse contexto os *clusters* e arquiteturas *multicore* até certo ponto podem trazer benefícios para aumentar o desempenho de aplicações que exigam um grande poder de processamento, como é o caso das aplicações de renderização de imagens, as quais serão analisadas nesse trabalho.

Tarefas que exigem grande poder computacional, até pouco tempo atrás, só podiam ser executadas por supercomputadores [1]. No entanto, nos últimos anos, a Tecnologia da Informação evoluiu rapidamente fazendo com que as organizações usufruíssem dessa evolução. Com o avanço tecnológico dos computadores pessoais e das estruturas de redes locais, cada vez mais velozes, surge a idéia de utilizar o poder individual destes computadores, conectando-os através de um meio físico de comunicação e dotado de *softwares* específicos para executarem aplicações paralelas e distribuídas, dando ao usuário transparência quanto ao seu funcionamento [2]. Assim, quando se utiliza dois ou mais computadores em conjunto para resolver um determinado problema, temos o que chamamos de *cluster*, que, do inglês, significa agrupamento, aglomerado ou concentração [3].

Neste trabalho foi utilizada a implementação do *cluster openMosix*, uma extensão do *kernel* do Linux. Este *kernel* transforma este *cluster* em uma rede de computadores que simula um supercomputador fazendo uso da migração preemptiva de processos e algoritmos de balanceamento dinâmico de carga. A migração preemptiva de processos faz com que os nós (nome dado aos computadores que fazem parte do *cluster*) do *cluster* consigam migrar qualquer processo, em qualquer instante, para qualquer outro nó, baseados principalmente nas informações geradas pelos algoritmos automáticos que estão

sempre em execução. O algoritmo de balanceamento de carga tenta sempre ficar reduzindo a diferença de carga entre pares de nós, fazendo a migração de processos de um nó que esteja muito sobrecarregado para outro que esteja menos, balanceando assim a carga total do *cluster*.

Assim como a tecnologia relacionada aos *clusters*, a indústria de processadores tem evoluído em uma velocidade bastante rápida nos últimos anos. A proliferação dos computadores e das tarefas que lhes incumbimos continua a pressionar a necessidade de processadores mais poderosos [4]. Assim a transição para processadores *multicore* torna-se um ponto crítico neste novo desenvolvimento. O *multicore* consiste em colocar dois ou mais núcleos (*cores*) de processador no interior de um único *chip*. O objetivo é possibilitar ao sistema executar várias operações em simultâneo e assim alcançar melhor desempenho para responder às necessidades dos dias de hoje. Assim, os processadores *multicore* representam uma grande revolução na tecnologia computacional, sendo capazes de prover maior capacidade de processamento com um custo/benefício melhor do que processadores *singlecore* [5].

O objetivo central dessa monografia é fazer a análise das potencialidades do uso de um *cluster* Linux baseado na distribuição *openMosix* na renderização de imagens digitais utilizando *softwares* específicos para essa tarefa. Além disso, avaliar o desempenho de ferramentas de renderização em diversos computadores que possuem arquitetura *multicore*.

O foco é analisar o comportamento das ferramentas de renderização, estudando mais aprofundadamente o PovRay, e investigar o impacto de quantidade de nós no *cluster* em função da velocidade de processamento. Também fazer a análise dessas ferramentas de renderização em arquiteturas *multicore*, para ser observado até que ponto esse tipo de ferramentas conseguem usufruir de núcleos adicionais que estão presentes em um computador que possui este tipo de arquitetura.

Capítulo 2

Paralelismo x Arquiteturas *Multicore*

A indústria de processadores tem evoluído em uma velocidade bastante rápida nos últimos anos. A proliferação dos computadores e das tarefas que lhes incumbimos continua a pressionar a necessidade de processadores mais poderosos.

Durante décadas uma conjectura ditou sobre o futuro dos processadores, a conhecida "Lei de Moore". Esta conjectura foi declarada pelo pesquisador Gordon Moore na década de 1970 e dizia que a cada 18 meses o poder dos processadores dobraria [6]. Nesses tempos, os fabricantes de processadores queriam aumentar cada vez mais a velocidade do *clock* (relógio) para se obter sempre um maior desempenho.

Nos dias de hoje, o aumento do *clock* do processador para obtenção de melhorias de desempenho não tem mais viabilidade devido ao consumo de energia e sérios problemas com a dissipação de calor. Assim, os fabricantes de processadores estão fazendo uma transição para tecnologias de processadores que possuem múltiplos núcleos (*cores*) no interior de um único *chip*, surgindo os processadores *multicore*, com objetivo de possibilitar ao sistema a execução de várias operações em simultâneo.

Com processadores *multicore* os programadores precisam reconsiderar a maneira como desenvolvem as aplicações se quiserem aproveitar as vantagens desse tipo de processadores. Essa evolução no *hardware*, especificamente nos processadores, forçou os desenvolvedores de *softwares* a tratar, cada vez mais, com computação paralela, concorrente, ou seja, a habilidade que um programa tem de se dividir em partes, sendo tratadas em separadas, e depois serem recombinadas [7].

2.1 Tecnologia *Multicore*

Antes de tudo, é preciso compreender quais são as principais partes de um processador.

Até pouco tempo, os processadores da Intel eram formados por dois componentes básicos [8]:

- Núcleo (em inglês, *Core*): lugar onde as instruções são executadas. É no núcleo que ocorrem as execuções de instruções, cálculos, etc.
- *Cache*: é uma memória de acesso rápido localizada dentro do processador. Ao invés do processador ir fazer a busca de dados na memória principal o tempo todo, ele possui um memória interna mais próxima a ele onde os dados são armazenados temporariamente. O processador sempre procura por um dado dentro da sua memória *cache* antes de ir buscar o dado na memória principal. Por esta razão, o *cache* também tem influência no desempenho do processador.

Uma analogia bastante interessante feita em relação ao significado de se adicionar a um novo núcleo de processamento a um processador é descrita a seguir: “adicionar um novo núcleo assemelha-se a abrir uma nova pista em uma estrada para aliviar o trânsito: os carros não precisam dirigir mais rápido para chegarem mais cedo ao seu destino, eles apenas não são atrasados tanto pelo gargalo de poucas pistas e congestionamentos”[9].

Os processadores de múltiplos núcleos podem trabalhar em um ambiente multitarefa. Nos sistemas que possuem somente um núcleo, as funções de multitarefa podem ultrapassar a capacidade da CPU (*Central Processing Unit* – Unidade Central de processamento), o que implica em queda no desempenho enquanto as operações aguardam para serem processadas. Em sistemas de múltiplos núcleos, como cada núcleo tem seu próprio *cache*, o sistema operacional dispõe de recursos suficientes para lidar com o processamento intensivo de tarefas executadas em paralelo. Assim, melhora-se a eficiência do sistema e o desempenho dos aplicativos em computadores que executam vários aplicativos em simultâneo [10].

Com isso, para melhor entendimento colocamos a figura abaixo para que ficasse mais fácil a compreensão básica da vantagem da tecnologia *multicore*. A figura 1(a) exemplifica o gargalo que surge quando vários aplicativos são executados em um único *core*. Já a figura 1(b) mostra o desaparecimento do gargalo quando adicionamos mais de um núcleo [10].

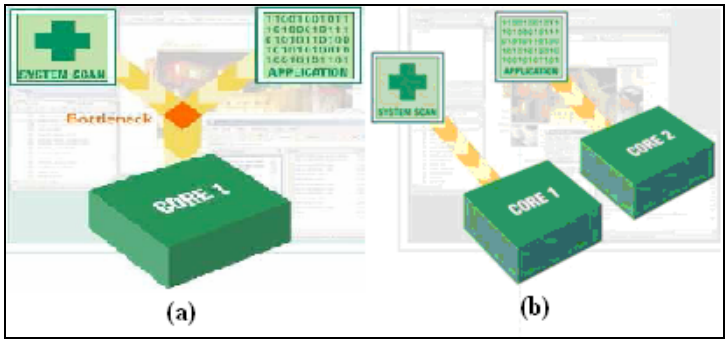

Figura 1. (a) Arquitetura *singlecore* (b)Arquitetura *multicore* [10].

Uma outra vantagem da tecnologia *multicore* é a redução da dissipação térmica quando comparada ao *singlecore*. Portanto, os picos de geração de calor e consumo de energia que existem quando processadores de núcleo único aumentam suas freqüências são reduzidos.

2.2 *Multicore* x SMP

Atualmente existe muita confusão no que diz respeito à classificação de sistemas multiprocessados, principalmente devido às diversas nomenclaturas adotadas pelas companhias que desenvolvem processadores, mas também em relação a questões puramente de mercado, como licenciamento de *software* por processador. Nos próximos parágrafos será abordada a questão das semelhanças e diferenças entre sistemas SMP (*Symmetric Multiprocessing* – multiprocessamento simétrico) e os recém-chegados sistemas *multicore*.

Basicamente, a tecnologia SMP foi desenvolvida pensando-se em adicionar processadores idênticos a uma mesma placa-mãe e provendo-se tecnologia adequada para intercomunicação entre estes, além de coordenação para acesso à memória e dispositivos de E/S (entrada e saída). Para que processadores em pastilhas fisicamente separadas – justamente o que ocorre com a tecnologia SMP – possam ter acesso coordenado à memória principal, memórias *cache* e demais dispositivos, uma grande quantidade de *hardware* de alta complexidade deve ser projetada e produzida.

Este *hardware* de comunicação deve levar em conta fatores como tecnologias SMP que trabalham com *caches* por processador, ou mesmo de maneira mais geral, como citado anteriormente, no acesso compartilhado à memória. Uma escrita à memória por parte de um dos processadores de uma arquitetura SMP faz com que seja necessária a atualização do conteúdo da *cache* desse processador, assim como provocar o descarte ou atualização de uma possível cópia privada desse mesmo pedaço de memória no *cache* do outro

processador, se existir essa cópia. Caso não haja a sincronização de *caches* privados e sem a coordenação de acesso à memória principal, as aplicações podem trabalhar com dados desatualizados, o que certamente gerará resultados incorretos [4].

Outro aspecto crucial para tecnologias de processadores em pastilhas separadas é que a comunicação entre processadores precisa ser muito rápida. Com isso, mais complexidade de *hardware* é adicionada, requerindo barramentos inter-processadores de altíssimo desempenho.

A tecnologia *multicore* é similar à tecnologia SMP no quesito de serem utilizados processadores – ou, na denominação da própria tecnologia, *cores* (ou núcleos) – idênticos. A principal diferença entre ambas está no fato de que a tecnologia *multicore* é uma especialização da tecnologia SMP onde ambas as unidades de processamento são dispostas num mesmo *chip*. Essa característica proporciona vantagens em termos de aumento de largura de banda para comunicação *inter-cores* e transferência da complexidade de comunicação entre processadores em pastilhas fisicamente separadas para comunicação intra-pastilha [4].

Devido ao fato de dispensar requisitos de *hardware* mais específicos como acontece na tecnologia SMP padrão por causa da localização das unidades de processamento, e com isso ter seu custo agregado total num sistema completo de computação menor, a tecnologia *multicore* é conhecida como uma versão barata de SMP. A maioria dos sistemas operacionais que suportam processadores *multicore* trabalha com eles como se fossem sistemas SMP padrão, e realmente é assim que eles funcionam, já que os processadores são fisicamente distintos, apenas estão localizados num mesmo *chip*, interagindo entre eles apenas quando necessário.

2.3 Programação Paralela: Maximizando Desempenho de Sistemas Multiprocessadores

Atualmente a programação paralela assume papel fundamental na evolução tecnológica dos sistemas computacionais devido à popularização dos sistemas multiprocessadores.

O objetivo é extrair desses sistemas um ganho de desempenho que não foi possível se obter com o desenvolvimento de uniprocessadores mais velozes. A idéia básica por trás disso é que problemas complexos podem ser quebrados em partes menores – no entanto, nem sempre é possível realizar cada uma dessas partes ao mesmo tempo devido à natureza seqüencial de algumas dessas.

2.3.1 Comunicação entre Processos

Uma aplicação paralela ou multiprocessada é constituída por um conjunto de processos que cooperam entre si em busca de um resultado. Esses processos precisam se comunicar para trocar dados e pedidos. Como cada processo progride de forma independente dos demais, na maioria das vezes é necessário definir pontos de sincronização entre eles, a fim de garantir a coerência da computação global.

Vários sistemas operacionais passaram a oferecer, nos últimos anos, suporte a um tipo de processos denominados *threads*, que podem se comunicar através de variáveis globais compartilhadas. Para o programador paralelo, *threads* são importantes em máquinas com vários processadores, onde eles permitem o desenvolvimento de aplicações paralelas cujas partes se comunicam por memória compartilhada e, muitas vezes executam de forma mais eficiente do que processos tradicionais. A comunicação por memória compartilhada exige uma atenção grande com o problema de sincronização. Para garantir uma relação de precedência entre comandos de processos distintos, o programador deve fazer uso de comandos explícitos de sincronização, que devem ser providos pela linguagem ou sistema operacional [11].

O desenvolvimento de aplicações baseadas em variáveis compartilhadas e condições podem ser bastante complexos. Os bloqueios e sinalizações associados a variáveis de condição podem estar espalhados por um código extenso, e qualquer acesso a uma variável global pode representar uma necessidade de sincronização, tornando difícil a tarefa de depurar o programa paralelo [11].

2.3.2 Identificação de Paralelismo

Os computadores *multicore* estão cada vez mais disponíveis no mercado. Para se obter vantagem de múltiplos processadores, os programadores e/ou compiladores precisam identificar operações que possam ser executadas em paralelo.

Um método para se identificar paralelismo é desenhar o grafo de dependência de dados. A figura 2 representa esses tipos de grafos, onde os nós desses grafos representam tarefas, e as setas de um nó para outro significa que uma tarefa precisa ser completada antes que a outra tenha início.

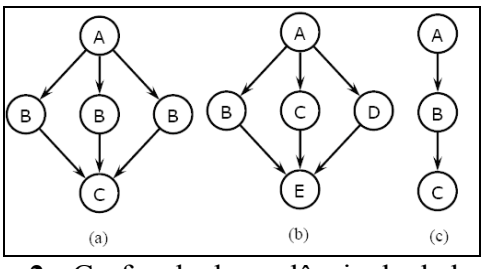

Figura 2. Grafos de dependência de dados [12].

Na figura 2(a) tem-se o chamado paralelismo de dados, em que tarefas independentes aplicam a mesma operação a diferentes elementos de dados. Já na figura 2(b) têm-se o paralelismo funcional, em que às tarefas independentes aplicando operações diferentes em diferentes elementos de dados. Por fim, na figura 2(c) ocorre o grafo puramente seqüencial. Se neste último caso só tivermos um problema a ser resolvido, não há como ter paralelismo [12].

2.3.3 Desenvolvimento de Programas Paralelos

McGraw e Axelrod, em 1988, identificaram quatro soluções diferentes para o desenvolvimento de aplicativos para computadores paralelos [12]:

1. **Extensão de um compilador existente para traduzir programas seqüenciais em paralelos:** Com este tipo de compilador o programador continua programando de forma totalmente seqüencial. A tarefa de paralelização é de total responsabilidade do compilador.
2. **Extensão de uma linguagem existente com novas operações para permitir a expressão do paralelismo:** A idéia é a criação de uma linguagem paralela, dando aos programadores acesso a funções de baixo nível que permitem máxima flexibilidade com respeito ao desenvolvimento de programas paralelos.
3. **Adição de uma nova camada de linguagem no topo de uma linguagem seqüencial:** A camada de baixo, que é o núcleo da solução do problema, é feita usando uma linguagem seqüencial existente. A camada de cima controla a criação e sincronização de processos e a partição de dados entre processos.
4. **Definição uma linguagem paralela totalmente nova e seu sistema de compilação:** O mais famoso exemplo de linguagem paralela é Occam. Esta linguagem suporta execução paralela e seqüencial de processos e sincronização e comunicação automática de processos.

2.3.4 Algoritmos Paralelos

A metodologia de desenvolvimento de algoritmos paralelos mostrada abaixo é baseada no modelo tarefa/canal descrito por Ian Foster. Esse modelo facilita o desenvolvimento de programas paralelos eficientes.

A Figura 3 abaixo representa o modelo tarefa/canal proposto por Foster através de um conjunto de tarefas (círculos) que interagem umas com as outras enviando mensagens através de canais (setas). Assim, cada tarefa é um programa, sua memória local e um conjunto de portas de E/S (entrada e saída). Um canal é uma fila de mensagens que

conecta a porta de saída de uma tarefa com a porta de entrada de outra tarefa. Nesse modelo, a transmissão de uma mensagem e a recepção de outra pode ser feita concorrentemente.

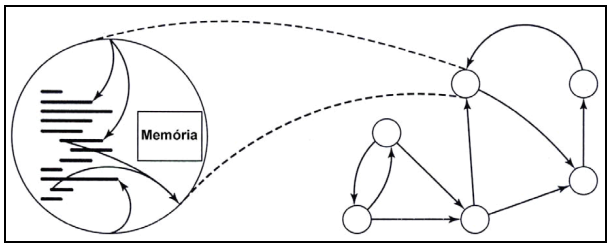

Figura 3. Modelo tarefa/canal [12].

O método de Foster é dividido em quatro etapas: Partição dos dados e do processamento com a criação das tarefas primitivas, estabelecimento do padrão de comunicação ente as tarefas, aglomeração de tarefas primitivas e mapeamento das tarefas nos processadores disponíveis [12,13].

A Figura 4 ilustra o método de Foster.

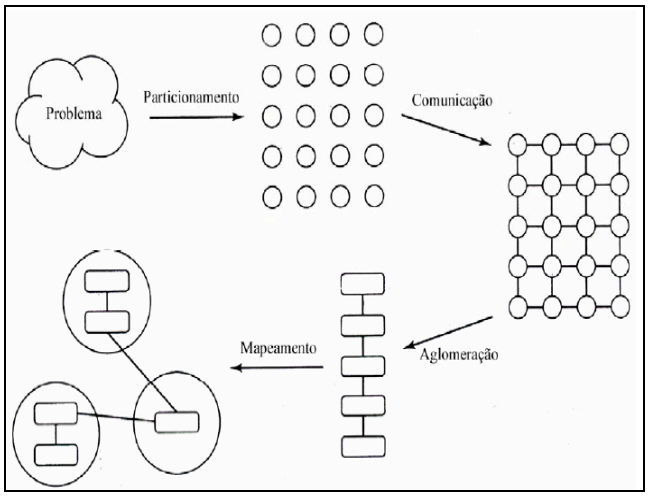

Figura 4. Metodologia de Foster para projetos de algoritmos paralelos [12].

Partição é o processo responsável pela divisão da computação e os dados em partes buscando descobrir o máximo de paralelismo possível. Já na comunicação é onde ocorre a identificação do padrão de comunicação entre as tarefas. Nesses dois processos iniciais o objetivo é identificar o máximo de paralelismo possível sem se preocupar com a plataforma em que o programa vai executar. Na aglomeração ocorre a combinação de tarefas primitivas para mapeá-las em processadores físicos de forma a reduzir a sobrecarga paralela, enquanto no mapeamento vai acontecer a designação das tarefas para os processadores [12].

2.4 Lei de Amdahl

A Lei de Amdahl nos mostra que o aumento de desempenho real em termos de processamento de um sistema multiprocessador depende de quanto o programa pode ser paralelizado e do número de processadores utilizados. Tomando como exemplo um programa que tem tempo de execução de 10 minutos e que tenha 80% de código paralelizado e 20% serializado. Temos que, se este for executado em um sistema quadriprocessado, a parte paralelizada poderá ser concluída em 8/4 = 2 min, enquanto que a parte serial será concluída em 2/1 = 2 min independente da quantidade de processadores. Ou seja, obtém-se um custo final de 4 minutos. Se o sistema fosse uniprocessador, o custo total seria de 8/1 + 2/1 = 10 min. Então, pode-se concluir que o ganho real obtido foi de 6 minutos, o que representa, segundo a Lei de Amdahl, um fator de aceleração de 10/4 = 2,5 – diferentemente do que poderia ser esperado com a quadruplicação da unidade de processamento, ou seja, um fator de aceleração igual a 4 [13].

Diante do exemplo anterior, a Lei de Amdahl mostra que existe um limitante superior para ganho de desempenho de sistemas independente do número de processadores adicionados: a quantidade de código que precisa ser executado serialmente. No exemplo pode-se notar que o maior fator de aceleração a ser obtido será de 10/2 = 5, considerando-se o fato de que o sistema tenha, no mínimo, 5 processadores, o que faria com que o código paralelo fosse executado em 8/5 minutos e, ao mesmo tempo, o código serial fosse executado em 2/1 minutos, sendo que para isso é necessário que o código serial não dependa do resultado final da parte paralela. Em suma, esse valor só é alcançável se a parte serial puder ser executada em paralelo (ainda assim de forma serial para a parte) com a parte paralela.

2.5 Multiprocessamento em Sistemas Operacionais e em Aplicativos

Um sistema operacional com suporte a múltiplos processadores gerencia o uso destes e dos demais recursos do computador tal qual um sistema uniprocessador, tornando transparente ao usuário os detalhes internos de gerenciamento adicionais necessários para sistemas multiprocessadores. Para tanto, é preciso que o sistema operacional proveja tanto as funcionalidades de um sistema multiprogramado quanto os recursos adicionais para suportar múltiplos processadores. Entre as principais questões de projeto, estão as seguintes [14]:

- Processos concorrentes simultâneos (rotinas reentrantes para permitir execução simultânea por processadores diferentes);
- Escalonamento por qualquer processador;
- Sincronização especial (devido a espaços de endereçamento e recursos de E/S compartilhados);
- Gerenciamento de memória (explorando paralelismo disponível no *hardware*);
- Confiabilidade e tolerância à falhas (suportar falhas de processadores).

Os sistemas operacionais mais usados hoje em dia – Windows, Linux e UNIX – já possuem suporte maduro a múltiplos processadores, realizando tanto os controles de recursos adicionais para esse tipo de sistema quanto decidindo entre a utilização de dois ou apenas um processador ao executar mais de uma tarefa do sistema.

Foram feitos grandes esforços para explorar o desempenho máximo em termos de utilização de recursos de multiprocessadores em sistemas operacionais, mas é sabido que trabalhar apenas em nível de S.O. (Sistemas Operacionais) não é o bastante para tirar total proveito do poder oferecido pelo *hardware* paralelo. Os aplicativos que serão executados no sistema preparado para múltiplos processadores também precisam ser pensados e estruturados de forma a facilitar o gerenciamento de recursos pelo S.O., seja através de algoritmos menos seqüenciais, com possibilidade de uso de vários *threads* de execução que podem ser distribuídos pelos processadores, ou mesmo através do uso de bibliotecas paralelas, que provejam mecanismos de controle necessários às áreas de memória compartilhada.

Capítulo 3

Clusters

Sabemos que, atualmente, existem vários tipos de aplicações que necessitam de um poder computacional muito maior do que um único computador, que possui somente um processador, pode proporcionar. Assim, o tempo de espera por respostas para esses tipos de aplicações nesses tipos de computadores pode durar de dias a anos. Alguns tipos de aplicações que demandam um alto poder de processamento são: mapeamento do genoma humano, exploração de petróleo, previsão do tempo, renderização de efeitos especiais em filmes, e outras.

Com o objetivo de preencher a falta de computadores que dão resultados mais rápidos quando são exigidos por aplicações que demandam por um alto poder computacional, foram surgindo os supercomputadores, como estações de trabalho dedicadas com múltiplos processadores, sendo capazes de distribuir a carga gerada pelas aplicações entre seus vários elementos de processamento. Mas, a obtenção de supercomputadores, nem sempre chega a ser a alternativa mais conveniente, devido a vários fatores, como, por exemplo: o custo muito elevado; possuem *softwares* proprietários e caros; um custo elevado para manutenção; a grande dependência dos fornecedores; grande dificuldade de fazer sua atualização [1].

Surgiu, assim, a computação baseada em *clusters*, que aparece então como uma forma relativamente simples e barata para suprir essa demanda, que até então somente supercomputadores seriam capazes de resolver. Uma definição simples para a computação em *cluster* seria: o agrupamento de dois ou mais computadores trabalhando em paralelo para solucionar determinado problema.

Uma outra definição de *cluster* pode ser resumida como sendo um conjunto de computadores autônomos e que interligados comportam-se como um único sistema do ponto de vista do usuário. Ou seja, todos os aspectos relativos à distribuição de tarefas,

comunicação, sincronização e organização física do sistema devem ser abstraídos do usuário [1].

3.1 Características dos *Clusters*

A computação baseada em *cluster* ainda supera os supercomputadores em diversos aspectos, onde podem-se destacar [1,15]:

- **Alto Desempenho:** resolução de problemas de grande complexidade em curto espaço de tempo;
- **Escalabilidade**: possibilidade de adicionar aplicações, periféricos, nós e até mesmo mais interconexões de rede sem interromper a disponibilidade dos serviços do *cluster*;
- **Custo Reduzido:** custos reduzidos com processamento de alto desempenho utilizando *hardware* de fácil disponibilidade, como computadores pessoais; utilização de *hardware* aberto, *software* livre e independência de fabricantes;
- **Confiabilidade:** o *cluster* tem que ser dotado da capacidade de detecção de falhas internas, e com isso, tomar a decisão cabível para solucionar o problema, para que o mesmo não venha a prejudicar os serviços que são oferecidos;
- **Gerenciamento e Manutenção:** mecanismos que permitem gerenciar de forma simples a complexidade de configurações e manutenção. Umas das grandes dificuldades de um *cluster* são a sua manutenção e principalmente a sua configuração, pois na maioria das vezes são tarefas morosas e que tem grande propensão a erros;
- **Transparência:** O *cluster*, que é construído por vários nós independentes e fracamente acoplados, deve parecer como se fosse apenas um computador para o cliente, ou seja, possuir a capacidade de se apresentar como um sistema único.

3.2 Tipos de *Clusters*

Clusters de computadores podem ser subdivididos em várias categorias, onde cada uma delas tem sua própria característica, sendo utilizada para uma finalidade específica, são elas:

3.2.1 Alta Disponibilidade

Um *cluster* de alta disponibilidade tem por finalidade principal prover os serviços de um servidor o maior tempo possível. A sua forma de funcionamento consiste em fazer a substituição de um nó do *cluster,* no momento que este vier a falhar, por outro, de modo transparente ao usuário. Isso é conseguido através da criação de réplicas dos serviços e servidores, onde os computadores agem como um único sistema. Cada um monitorando o outro através de *software* e, no caso de ocorrer uma falha, um deles assume os serviços daquele que ficou indisponível [16].

Os servidores mostrados na figura 5 abaixo são exatamente réplicas, tanto de *hardware* como de *software*. O servidor localizado na direita, através do esquema de monitoramento, detecta a indisponibilidade do servidor da esquerda, e logo assume o posto para fazer a manutenção do serviço disponível [17].

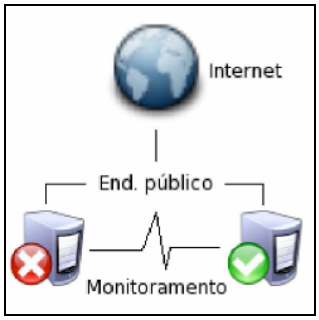

Figura 5. Esquema de alta disponibilidade [17].

3.2.2 Balanceamento de Carga

Esta solução tem como objetivo manter o equilíbrio de carga entre diversos servidores, distribuindo as requisições feitas aos elementos que o compõe, mantendo de forma equilibrada o tráfego gerado a determinados serviços, garantindo assim a disponibilidade do serviço em momentos de elevados acessos. A necessidade deste tipo de *cluster* se justifica pelo fato da grande quantidade de tráfego nos servidores que prestam serviço em uma rede, pois, atualmente, um dos maiores problemas enfrentados na Internet são os excessivos acessos quando se possui apenas um servidor [3].

A atividade principal é fazer o redirecionamento das requisições de entrada por meio de um elemento que fará o balanceamento entre os servidores e os usuários. Com isso, nesse tipo de estrutura têm-se múltiplos servidores, mas que parecem existir apenas um para o cliente, como ilustrado na figura 6 [17].

Figura 6. Cluster para balanceamento de carga [17].

3.2.3 Alto Desempenho

Sua utilização destina-se a atender a tarefas que exigem um grande poder computacional, esse tipo de *cluster* trabalha de modo a fazer a distribuição da carga de processamento entre os elementos que compõe o *cluster*, gerando uma redução do tempo que é gasto com seu processamento paralelo e distribuído. Sua utilização é bastante comum para resolução de problemas tipicamente paralelos, ou seja, aplicações desenvolvidas para processarem cálculos de forma paralela. É muito útil em áreas que tem como características a geração de um grande volume de dados, como simulações financeiras, renderização de efeitos especiais em filmes, meteorologia, e em várias outras que tem por necessidade um alto poder de processamento [3].

3.3 Componentes de um *Cluster*

Os principais componentes de um cluster de computadores são [17]:

- **Nós** – são os computadores que fazem parte do *cluster*. Estes podem ser computadores de vários tipos, desde computadores pessoais, com somente uma unidade de processamento, até os computadores multiprocessados.
- **Sistema operacional** – O sistema operacional mais utilizado em se tratando de cluster é o GNU/Linux, o qual possui código fonte aberto, permitindo assim realizar alterações no código fonte como acontece no caso do projeto *openMosix*.
- **Rede Local** – este componente do cluster influi diretamente no seu desempenho. A placa de rede, o *switch* e o cabeamento são os que possuem a função de transmitir e receber os pacotes entre os nós que compõe o *cluster*.
- **Protocolos** – conjunto de regras e procedimentos para se efetuar uma comunicação de dados rápida e confiável entre os nós do *cluster*. O protocolo

mais utilizado para implementação de um *cluster* é o TCP/IP, apesar de não ser otimizado para tarefas deste tipo, mas oferece um menor custo de implantação.

- **Sistema de arquivos** – um dos sistemas de arquivos mais conhecidos e utilizados é o NFS (*Network File System*), mas existem diversas bibliotecas de interfaces a sistemas de arquivos paralelos para redes de estações. Um exemplo típico é o DFSA (*Direct File System Access*), utilizado pelo sistema *openMosix*.
- **Ferramentas de comunicação** – é necessário que um *cluster* possua ambientes de programação que ofereçam meios para desenvolver programas paralelos, como é o caso do MPI (*Message Passing Interface*).
- **Ferramentas de gerenciamento** – formado por um conjunto de programas que realizam atividades de monitoramento e gerenciamento de *clusters*, como é o caso do *openMosixview* encontrado na distribuição *openMosix* .

3.4 Aspectos da Interligação em *Software*

A interligação de *clusters* recai fortemente no âmbito do *software* usado para permitir que o sistema funcione realmente como um único sistema interligando vários computadores distintos.

Quando temos um sistema de memória distribuída, no qual cada processador possui sua própria memória local, é necessário definir um modo de interação entre os processos em andamento. Para sistemas em *cluster*, o sistema de troca de mensagens é largamente usado para esse fim, onde as mensagens requerem os dados a serem copiados que serão executados em threads locais.

Para que o ambiente de troca de mensagens seja empregado em *clusters*, é preciso levar em conta alguns aspectos, como:

- Fraco acoplamento entre computadores;
- Possível diferença na largura de banda de interconexão de nós;
- Atrasos na comunicação de nós;
- Diferenças entre *hardwares*, sistemas operacionais e linguagens dos nós interligadas.

Os ambientes de programação mais importantes e independentes das características dos computadores utilizados são o PVM (*Parallel Virtual Machine*, Máquina Virtual Paralela) e o MPI (*Message Passing Interface*, Interface de Passagem de Mensagens).

Para uma maior abstração dos computadores em cluster, um pacote de software SSI (*Single System Image*, Imagem Única do Sistema) pode ser utilizado, funcionando como uma camada entre o sistema operacional e as aplicações.

3.4.1 PVM – Máquina Virtual Paralela

A biblioteca de programação paralela de passagem de mensagens PVM tem por objetivo prover um ambiente de programação paralela transparente para utilização em sistemas heterogêneos. Ou seja, é um pacote de software que ao ser utilizado em sistemas operacionais largamente conhecidos como Linux, UNIX e Windows, em um conjunto de máquinas heterogêneas interligadas em rede, permitem a agregação dessas máquinas em um único sistema, como um grande computador paralelo [18].

O PVM consiste em duas partes:

1. Biblioteca de programação paralela;
2. Ambiente Paralelo Computacional (PPE – *Parallel Programming Environment*).

A biblioteca de programação paralela provê serviços básicos em termos de chamadas de rotinas para a troca de mensagens entre sistemas distribuídos, devendo ser as rotinas solicitadas por alguma linguagem de programação. Isso faz com que os desenvolvedores de aplicações paralelas possam usar o ambiente distribuído de maneira transparente, sem se preocupar com a localização física dos computadores.

O PVM é largamente utilizado em *clusters* devido a sua praticidade, portabilidade, interoperabilidade, abstração dos controles e recursos dos processos e por ser *software* de domínio público.

3.4.2 MPI – Interface de Passagem de Mensagens

O MPI, assim como o PVM, foi um esforço para padronização de ambientes de passagem de mensagens entre computadores heterogêneos (*hardware* + S.O. + linguagens). No entanto, o MPI se restringe apenas ao conjunto de bibliotecas para utilização em diversas linguagens de programação para programas que utilizem passagem de mensagens [19].

A comunicação entre os processos com MPI requer vários pedaços de informações, tais quais:

- Processo transmissor e receptor;

- Endereço inicial de memória do destino;
- Mensagem de identificação;
- Grupo de processos que podem receber a mensagem.

Inicialmente, na maioria das implementações, um conjunto fixo de processos é criado. Porém, esses processos podem executar diferentes programas. Diante disso, muitas vezes esse padrão é chamado de *Multiple Program Multiple Data* (MPMD).

3.4.3 SSI – Imagem Única do Sistema

Quando se tem um conjunto de computadores que devem aparecer como um único recurso (ou seja, como um único sistema em *cluster*), é preciso usar uma camada de *middleware* denominada SSI, que deve estar localizada entre o sistema operacional e as aplicações de usuário.

Na camada que compõe o SSI existem basicamente duas sub-camadas como pode ser visto na figura 7:

1. Infra-estrutura de imagem única;
2. Infra-estrutura de disponibilidade.

A primeira sub-camada deve estar presente em todos os sistemas operacionais para criar a abstração de unificação dos recursos de todas as máquinas em uma única máquina. A segunda sub-camada tem por objetivo oferecer serviços ao *cluster* como recuperação e tolerância a falhas para todos os nós do *cluster* [1].

Figura 7. Camada de SSI em um nó cluster [1].

Algumas vantagens do uso de SSI estão listadas abaixo [1]:

- Transparência na execução de aplicações;
- Não necessidade de localização física dos nós;

- Robustez de configuração devido à menor necessidade de intervenção de operadores;
- Permite modelo de processo com espaço globalizado para balanceamento de carga.

3.5 Aspectos da Interligação em Hardware

A escolha das tecnologias de interconexão dos nós é um ponto determinante para um bom desempenho do *cluster*. As redes de interconexão de nós estão diretamente ligadas com o desempenho do *cluster*, já que devido à natureza do sistema, são responsáveis pela comunicação entre as CPUs e memórias dos computadores distribuídos. Se não houver um balanceamento adequado entre os nós do *cluster* e a comunicação entre eles, alguns nós podem deixar de usar recursos enquanto aguardam por uma resposta de outros nós [20].

Diante disso, a escolha da melhor solução de interconexão de *hardware* do *cluster* é muito importante, tendo implicações diretas sobre não apenas o desempenho, mas também sobre a dificuldade de montagem e configuração do sistema e o seu custo. Atualmente podemos listar várias tecnologias de interconexão capazes de serem usadas em projetos de clusters, como:

- *Fast Ethernet*;
- *Gigabit Ethernet*;
- *Myrinet*;
- *InfiniBand*;
- *Scalable Coherent Interface* (SCI).

Para se conseguir alto desempenho de comunicação entre nós, é necessário o uso de uma tecnologia de *hardware* de rede com altas taxas de transmissão de dados e baixa latência, e para conseguir alta disponibilidade é preciso que a rede de interconexão esteja apta a detectar e isolar falhas, fornecendo caminhos alternativos para os nós [20]. As redes de interconexão que mais se adequam a esses requisitos são: *Myrinet, InfiniBand e Gigabit Ethernet*. Estes tipos de redes de interligação de nós são amplamente usados nos sistemas de computadores do TOP500, que é a lista dos 500 supercomputadores mais potentes do mundo [21].

3.6 *Cluster Beowulf*

Esse tipo de cluster recebeu este nome de *Beowulf* devido a um antigo poema inglês que foi escrito por um anônimo cristão, que faz a narração das três batalhas do herói *Beowulf* contra o grande gigante Grendel, a mãe do gigante e um poderoso dragão que guardava um tesouro [22]. Assim, recebeu este nome, pois é comparado a esse herói inglês fazendo analogia ao fato dele vencer fortes inimigos, como, também, esse tipo de cluster venceu, ao bater de frente com o monopólio que até então era de grandes fornecedores, fornecendo uma alternativa eficiente e de baixo custo à computação que demanda um grande poder de processamento.

Um *cluster* do tipo *Beowulf* é considerado dedicado ao processamento paralelo. Ele possui uma arquitetura de diversos computadores formados por um nó controlador, também chamado de *front-end*, e nós clientes, denominados *back-ends*, sendo estes interligados por uma rede *Ethernet* ou qualquer outro tipo de tecnologia de rede, como ilustrado na figura 8.

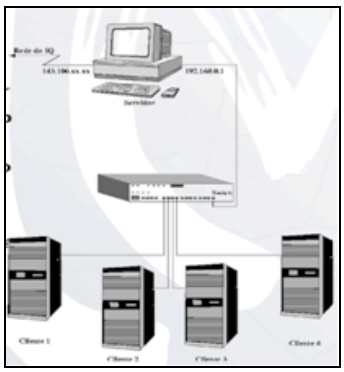

Figura 8. Cluster Beowulf [10].

Esta tecnologia de *cluster* é dedicada exclusivamente para a passagem de mensagens, com um sistema operacional que implemente estruturas PVM e/ou MPI, para realizar uma comunicação que otimize o processamento paralelo. A função do nó controlador é controlar, monitorar e distribuir tarefas para os nós do *cluster*, além de atuar como um elo entre ele e os usuários. Existe a possibilidade de ter mais de um nó controlador ou mestre no sistema. Os nós clientes ou escravos têm como finalidade exclusiva fazer o processamento das tarefas enviadas pelo nó controlador[1].

Quanto à utilização do cluster Beowulf, podemos citar algumas vantagens [17]:

- Independência de fornecedor, ou seja, sistema pode ser construído usando dispositivos de diferentes origens;

- Continuidade tecnológica, pois componentes atualizados são encontrados no mercado com facilidade;
- Flexibilidade de configuração;
- Escalabilidade;
- Alta disponibilidade.

Seguem a seguir alguns problemas que podem ocorrer na implementação [17]:

- Nem todas as aplicações podem ser paralelizadas;
- Difícil programação;
- Ocorrência de gargalos tradicionais em comunicações, como latência e largura de banda;
- É preciso experiência em ambientes de redes para a elaboração e configuração do *cluster*.

3.7 *Cluster openMosix*

O *openMosix* é um projeto de código aberto que se originou do projeto *Multicomputer Operating System Unix* (MOSIX) da Hebrew University. O openMosix é uma extensão do *kernel* do Linux para que os ambientes paralelos possuam uma imagem uniforme. A extensão provida pelo *openMosix*, faz com que uma rede comum de computadores executando estes pacotes, possa se tornar um supercomputador para aplicações executadas sob o sistema operacional Linux [16]. O *cluster OpenMosix* é utilizado para alta performance com o paralelismo de processamento

Após instalados os pacotes do *openMosix*, os nós que compõe a configuração começam um processo de comunicação, fazendo uma adaptação da carga de trabalho submetida para que seja executada. Com isso, a característica principal deste modelo de cluster é o método de migração de processos, na qual é equilibrada a carga entre os computadores de maneira que se consiga executar mais processos do que se fosse executado em apenas uma estação, contribuindo diretamente aplicações que possuem uma grande quantidade de processos independentes [23].

Dentre as aplicações que se beneficiam com o *openMosix*, destacam-se [23]:

- Aquelas cujos processos requerem longos tempos de execução e baixo volume de comunicação, como aplicações científicas, engenharia, etc;
- Grandes compilações;
- Banco de dados que não usam memória compartilhada;
- Processos na qual é possível fazer migração manualmente.

Algumas aplicações podem não obter vantagens utilizando este *cluster*, entre elas [23]:

- Aplicações que requerem memória compartilhada;
- Aplicativos que utilizam grande comunicação interprocessos;
- Aplicações que dependem de um *hardware* específico presente em algum nó do *cluster*;
- Aplicativos com grandes quantidades de *threads*;
- Aplicações que utilizam um único processo, por exemplo, um *browser*.

3.7.1 Algoritmos de Compartilhamento de Recursos

Para conseguir um melhor desempenho, este modelo de *cluster* é sempre verificado através de algoritmos de Balanceamento Dinâmico de Carga e algoritmo de Anunciação de Memória. O objetivo desses algoritmos é responder dinamicamente às variações da utilização dos recursos pelos diversos nós, garantindo sempre a escalabilidade.

O algoritmo de balanceamento de carga é utilizado quando um programa que está sendo executado em um nó específico está por alguma razão muito sobrecarregado, esse algoritmo irá agir com a finalidade de fazer uma migração desta atividade ou parte dela para um nó com mais recursos disponíveis, o sistema, conseqüentemente, ganhará em desempenho.

O novo algoritmo implementado no *openMosix* é, de certa forma, bastante interessante pelo fato de tentar conciliar as diferenças baseado em princípios da economia e análise de competitividade. Para o algoritmo chegar a alguma conclusão, os recursos de comunicação são medidos em termos de largura de banda, memória em termos de espaço e CPU em termos de ciclos. A idéia principal é fazer a conversão do uso total desses diversos recursos heterogêneos em um custo homogêneo. Assim, os processos são então transferidos para o local que possuir um menor custo. Esse algoritmo é executado de forma completamente independente em todos os nós do *cluster* [1].

Existe, também, um algoritmo que é usado com a finalidade de se evitar um total esgotamento da memória. Este algoritmo de anunciação de memória entra em ação quando um determinado nó do *cluster* começa a fazer uma paginação. Nesse caso específico, este algoritmo tem a preferência sobre os demais, mesmo que essa migração do processo possa causar um desbalanceamento da carga.

3.7.2 Migração de Processos

O *openMosix* implementa, para a migração de tarefas entre os nós, processos de modo preemptivo, ou melhor, os processos são divididos em duas partes, como mostrado na figura 9: a que é transferida para ser executada em outro nó, denominada de contexto do usuário (*remote*). A outra parte possui todas as descrições dos recursos dos quais o processo está utilizando ou está alocado para ele, bem como a pilha do sistema que controla a execução do programa. Essa parte é conhecida como contexto do sistema (*deputy*). É bom ressaltar que o processo remote pode ser transferido tantas vezes que se fizer necessário, para qualquer nó do cluster, mas o *deputy* nunca pode ser removido [1].

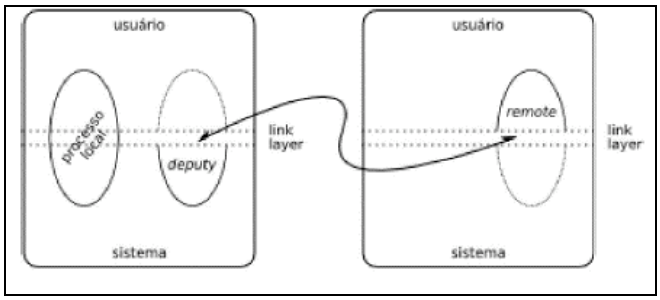

Figura 9. Processo local e processo remoto [1].

Ainda sobre a figura 9, podemos visualizar que existem dois contextos de execução para um processo: o modo usuário e o modo sistema. Como a interface dos dois modos é muito bem definida e conhecida, é possível, de um modo simples, interceptar toda e qualquer interação que existe entre esses dois modos e encaminhá-la para outros nós da rede. Este processo é realizado pela camada de adaptação, também conhecida como *link layer*, a qual contém um canal especial de comunicação para que exista essa interação dos dois contextos do sistema operacional [3].

O processo de migração de processos pode ser feito de duas formas: automático ou manual. Quando é da forma manual, o administrador faz a indicação de qual processo deverá migrar para determinado nó, podendo ainda indicar a capacidade máxima que um determinado nó deve tolerar antes de iniciar o processo de migração para outro nó do conjunto. Porém, existem muitos processos que podem migrar para outros nós, porém, isso não significa que os mesmos terão benefícios com esta migração.

Capítulo 4

Ferramentas de Renderização de Imagens

Sabe-se que a origem da palavra renderizar é originária da palavra inglesa *render* sendo um termo bastante usado na computação gráfica para fazer menção ao convertimento de um tipo de arquivo para outro, ou até mesmo a "tradução" de uma linguagem para outra [24].

Nesse processo de renderização de uma determinada cena é preciso fazer definição de várias "variáveis", por exemplo: os objetos presentes na cena, suas texturas, sua cor, reflexão, transparência e fazer a localização de pontos de iluminação. Podemos resumir o processo de renderização como sendo uma forma de converter modelos gráficos em uma imagem. Nesse ato de renderizar, os programas responsáveis, fazem cálculos diversos, como da perspectiva dos planos, das sombras e luzes do objeto.

O principal motivo da utilização de renderização nessa monografia é o fato desse ato sempre exigir uma grande capacidade computacional, podendo tirar proveito, até certo ponto, dos dois temas focos: *clusters* e arquiteturas *multicore*. Esse capítulo não tem por objetivo se aprofundar nos tipos de ferramentas de renderização, pois esse não é o foco do trabalho, e sim fazer uso dessas ferramentas para servirem de estudo para *clusters* e arquiteturas *multicore.*

Nos tópicos abaixo será dado destaque a uma técnica de renderização que é muito usada por várias ferramentas de renderização chamada de *Raytracing* e uma das ferramentas básicas de renderização utilizada nesta monografia.

4.1 Processo Físico de Geração de uma Imagem e Raytracing

Quando visualizamos um objeto ou uma determinada cena, o que acontece é o seguinte, como ilustrado na figura 10: uma fonte de luz emite raios em todas as direções, mas somente alguns desses raios atingem o objeto que está sendo observado. O objeto faz a absorção de parte da luz pelo qual é atingindo, e o resto é refletido pela sua superfície. Uma parte dessa luz que é refletida pelo objeto atinge os nossos olhos, assim, conseguimos ver o objeto em questão [25,26].

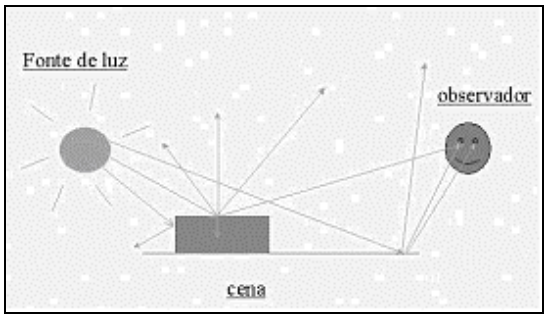

Figura 10. Processo físico de geração de uma imagem [25].

A técnica denominada de *raytracing*, ou seja, emissão de raios, é bastante utilizada na área da computação gráfica gerando resultados muito eficientes. Esta técnica de renderização de cena calcula a imagem desta cena simulando a forma como os raios de luz percorrem o seu caminho no mundo real.

A fonte de luz faz a emissão de uma grande quantidade de raios, mas somente poucos destes atingem realmente os nossos olhos para ocorrer à visualização do objeto. Fazer a simulação desse processo real no computador, seguindo o caminho de grande parte desses raios, apesar de apenas uma pequena parte contribuir para o resultado final, seria impraticável pelo fato da grande quantidade de processamento desnecessário envolvido. Por isso, o que é feito, é seguir o caminho do raio invertido, ou seja, a partir do ponto em que ele atinge o olho da pessoa que observa, e voltando para saber quais objetos foram atingidos desde que saiu da fonte de luz. Quando o raio atinge o objeto em um determinado ponto, podemos determinar se o ponto está sendo ou não iluminado pela fonte de luz, fazendo o traçado do raio a partir do ponto de intersecção até a fonte de luz. Se o raio atingiu a fonte, esse ponto está sendo diretamente iluminado. A idéia principal é lançar pelos menos um raio para cada *pixel* que compõe a imagem, e traçá-lo de volta, como também todos os raios que são gerados por ele na interação com os objetos da cena, até que ele intercepte uma fonte de luz [25,26].

O *raytracing* para fazer a geração dessas imagens foto-realistas, ou quase foto-realistas, depara-se com um custo computacional muito elevado, tornado-se esse processo bastante lento quando feito em um só computador.

4.2 PovRay

O PovRay é um renderizador de imagens do tipo *Raytracer*. Este tipo de renderizador consiste na simulação de iluminação, sintetizando imagens a partir de modelos geométricos tridimensionais de espaços físicos [27]. O PovRay é um aplicativo livre e está disponível para as plataformas Microsoft Windows, Linux e Mac OS [28].

Essa classe de programa faz uso intensivo da unidade de ponto flutuante que está presente no processador. Portanto, é um programa que pode gerar uma carga de processamento distribuído e ser bem aproveitada no âmbito dos *clusters* [28].

O PovRay consiste em um tipo de linguagem especial para descrever cenas. Para fazer uso é necessário dar como entrada comandos em um arquivo texto na inteface principal do programa para que o PovRay no processo de renderização possa gerar a imagem. Na figura 11 é mostrado a interface principal do programa na plataforma Windows dando ênfase a visualização de um código que descreve uma cena, que é o arquivo denominado landscape.pov, que gerará uma figura. Algumas das características de programa são: suporte para projeção ortográfica e perspectiva; possibilidade de inclusão de vários tipos de fontes de luz; radiosidade; primitivas, tais como cone, cilindro e esfera; técnicas de modelagem mais avançadas, tais como superfície paramétrica Bèzier, *sweep*, fractais; possibilidade de especificar as propriedades dos materiais incluindo diferentes tipos de texturas [29].

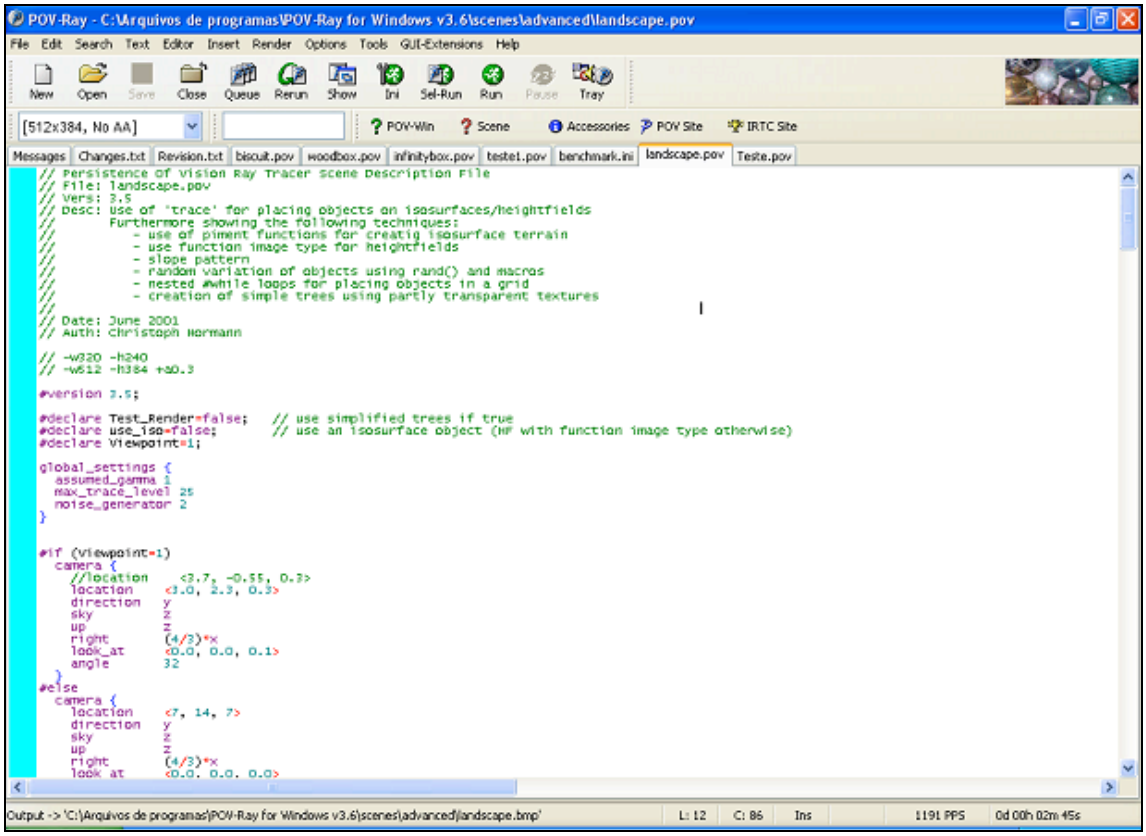

Figura 11. Interface principal do PovRay com arquivo landscape.pov.

Depois de executar o código, a ferramenta irá renderizar a imagem, varrendo linha por linha do código, gerando a imagem pouco a pouco, como podemos visualizar na figura 12.

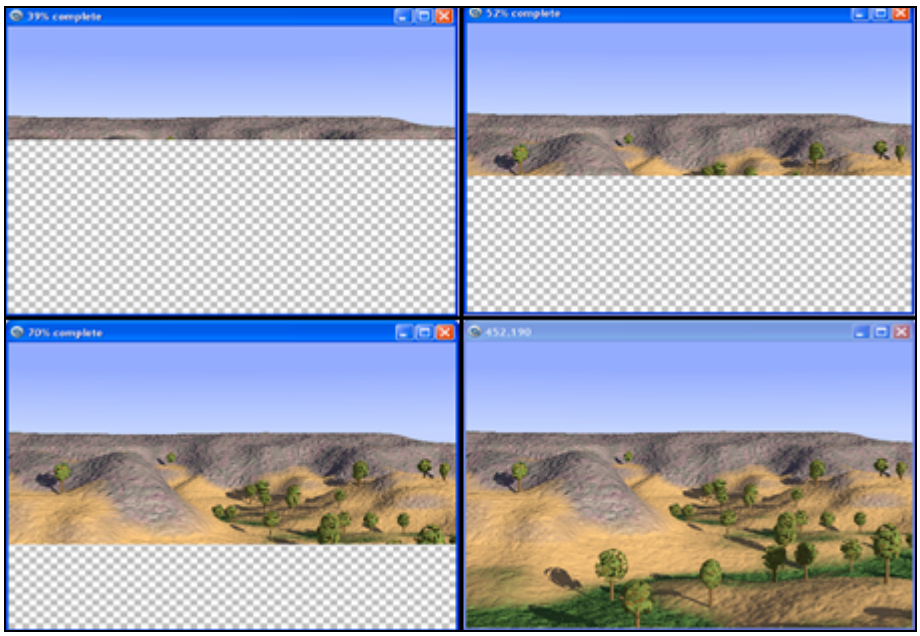

Figura 12. Etapas de geração da imagem.

Capítulo 5

Estudo de Caso

Neste capítulo será abordada a parte mais prática da monografia, ou seja, aspectos relacionados às atividades práticas desenvolvidas e os resultados obtidos. Assim, será descrito aspectos de configurações do *cluster*, estudo detalhado de questões que ficaram em aberto no estudo teórico, como o cálculo do *score* de cada nó do *cluster* e a parte dos testes que envolvem as ferramentas de renderização integrada a várias estruturas de *clusters* e arquiteturas *multicore* e conseqüentemente os resultados e conclusões obtidas após os testes realizados.

5.1 *Clusters* x Ferramentas de Renderização

5.1.1 Estrutura do *Cluster*

Foram montadas diversas estruturas de *clusters*, nas quais só variava o número de nós, com a finalidade de realizar testes com as ferramentas de renderização. O máximo número de nós utilizados foram 16, como pode ser visto na figura 13 abaixo.

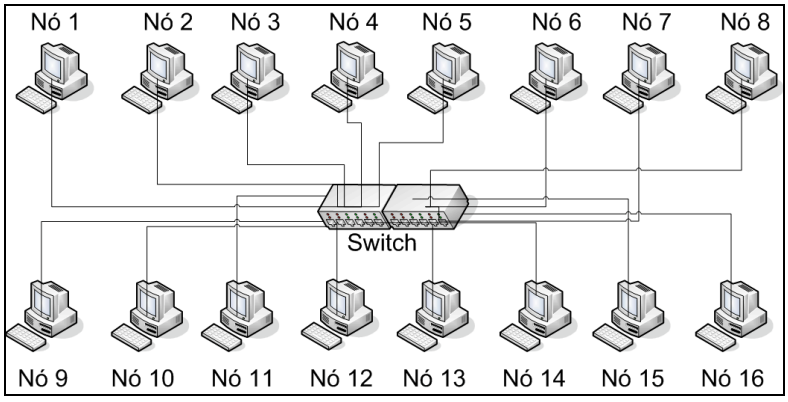

Figura 13. Estrutura do *cluster* montada.

Os nós pertencentes a todas as estruturas do *cluster* eram homogêneos com as configurações descritas na tabela 1.

Tabela 1. Configuração dos nós dos *clusters*.

Componente	Descrição	Quantidade
Processador	Intel Celeron – 2.53 GHz	1
Memória	512 Mbytes	1
Placa de rede	10/100 Mbps, padrão Ethernet	1

Nós com configurações iguais não são necessários, mas já ajudam a evitar, por exemplo, problemas decorrentes de compilação de um código para arquiteturas diferentes, ou por ter também uma carga de trabalho desbalanceada, onde os nós mais rápidos irão terminar primeiro que os mais lentos, de modo que os mais rápidos terão que ficar esperando. Assim, a homogeneidade não é uma condição necessária para um *cluster*, mas é um fator que irá reduzir bastante à quantidade de problemas.

Os nós estão interligados por um *switch* 10/100 Mbps e por cabos padrão *Ethernet* categoria 5 *enhanced*.

5.1.2 Configuração do *Cluster*

Desde o início da escolha do tema dessa monografia buscou-se um *software* que fosse uma alternativa prática e eficiente para montagem e realização dos estudos com *clusters*. A tecnologia de *cluster* escolhida foi o *OpenMosix* e a distribuição escolhida foi a *ClusterKnoppix* v3.6, rodando *kernel* Linux versão 2.4 modificado com o *patch openMosix* [1].

A configuração dos nós consistia basicamente na replicação do CD da distribuição *ClusterKnoppix*, havendo uma cópia do CD para cada computador componente do *cluster*, seguida da inicialização dos computadores com este sistema operacional através de *boot* via CD.

Como foi usada a estrutura do laboratório do DSC (Departamento de Sistemas e Computação), e este possui um servidor DHCP, que faz as atribuições de IPs automaticamente as maquinas, de inicio não se teve trabalho de fazer a configuração manual de IPs nas máquinas presentes no *cluster*, após a inicialização do sistema operacional. Mas, foi notado que assim alguns nós acabavam saindo da estrutura de *cluster* montada, pois sempre ficava rodando um processo (*pump*) que em determinados períodos de tempo alternava o IP dos nós.

A solução encontrada foi sempre que fosse configurado cada nó, primeiramente abortar-se-ia o processo *pump* para depois realizar-se a configuração manual dos IPs. Na tabela 2 abaixo tem-se o passo-a-passo dos comandos utilizados nesse processo.

Tabela 2. Configuração de rede dos nós do *cluster*.

```
$ su –
# killal -9 pump
# ifconfig eth0 192.168.168.1 netmask 255.255.255.0
# route add default gw 192.168.168.1
```

Os IPs eram sempre atribuídos variando o último octeto do endereço privado e atribuindo ao primeiro *host*, escolhido na ordem com que os computadores se encontravam distribuídos no laboratório, o valor 1, conforme mostra a tabela acima. Não há nenhuma restrição quanto a essa atribuição exigida pelo *OpenMosix*, esta foi apenas uma convenção adotada para saber qual IP estava relacionado a cada que máquina, com mais facilidade.

Apesar de não ser preciso alcançar outras redes a não ser a rede local do *cluster*, foi necessária a configuração do *gateway* padrão para o funcionamento do sistema de auto-descoberta de nós.

Em seguida, para finalizar a configuração do *cluster*, basta reiniciar o *daemon openMosix* para que a auto-descoberta de nós iniciasse, como mostrado na tabela 3.

Tabela 3. Reinicialização do daemon de auto-descoberta de nós do *openMosix*.

```
# /etc/init.d/openmosix restart
```

Depois de ser feito todo esse processo de configuração, foi executado o *software* de gerenciamento mais utilizado no *OpenMosix* conhecido como *openMosixview*. Através dessa ferramenta podemos visualizar todos os IPs dos nós que compõe o *cluster*, e poder saber se estes nós estão ativos ou não, através de uma numeração do lado esquerdo dos IPs. No caso da figura 14 abaixo, todos os nós estão ativos, ou seja, na cor verde; quando acontece alguma falha esse nó sai automaticamente do *cluster*, passa da cor verde para cor vermelha.

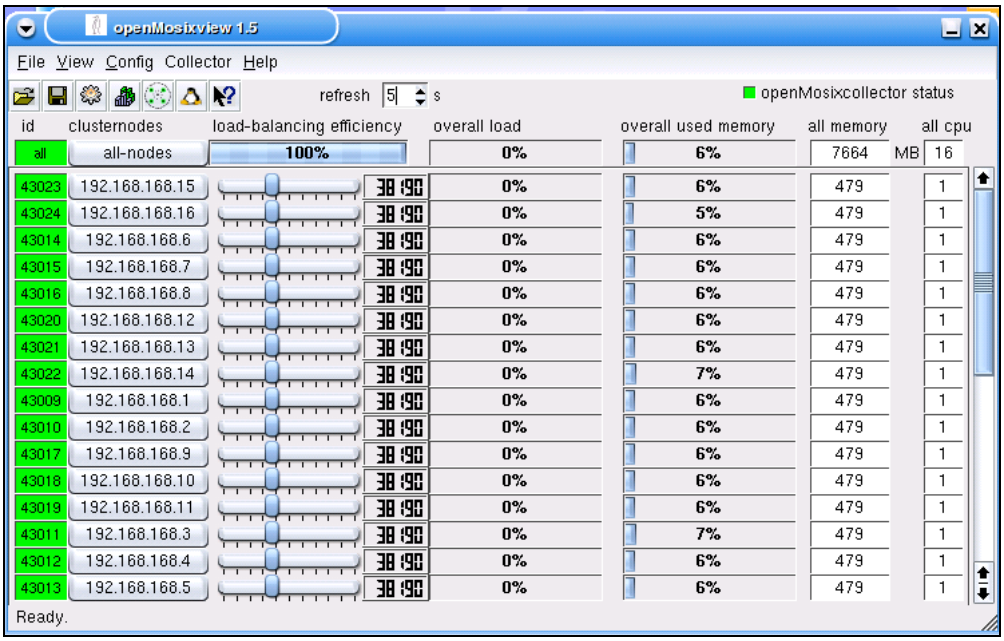

Figura 14. *openMosixview* após a configuração do *cluster*.

5.1.3 Computação do Score dos Nós do *Cluster* (*speed*)

Através da utilização da ferramenta *openMosixview*, notou-se que eram atribuídos valores numéricos aos nós que compunham o *cluster* como uma forma de classificar a potência computacional oferecida por cada nó, como mostrado quadriculado de vermelho na figura 15. Verificou-se que os maiores valores numéricos eram atribuídos aos computadores que possuíam configurações mais potentes. Desde então, denominamos genericamente este número de *score* do nó.

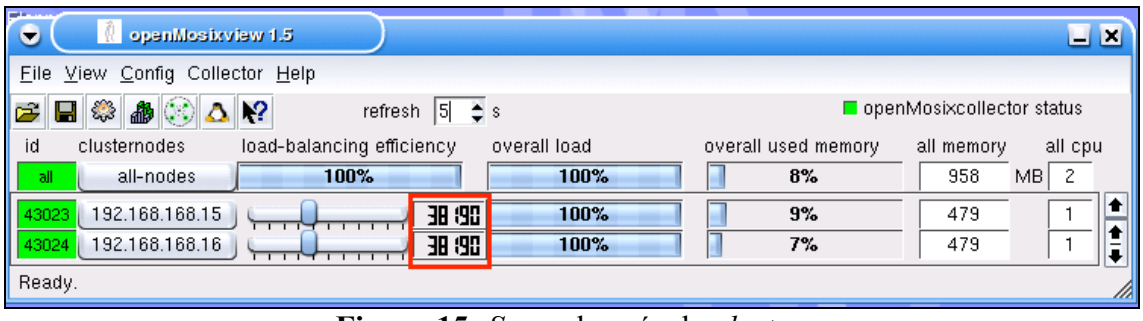

Figura 15. *Score* dos nós do *cluster*.

Desde os primeiros testes, em *clusters* diferentes (heterogêneos), com menos nós do que foram utilizados para os testes das ferramentas, foi observado que a atribuição dos processos aos nós sempre ocorria de forma que o nó com maior *score* sempre recebia os primeiros processos. Inteligentemente, o algoritmo de balanceamento de carga do *openMosix* faz essa atribuição baseada no *score* de forma a minimizar o tempo de

processamento global da tarefa, já que logicamente os nós mais potentes resolverão uma mesma tarefa mais rapidamente que nós menos potentes.

Até então, definiu-se o *score* calculado pelo *openMosix* em termos de "potência computacional", mas o que "potência" realmente significa para o cálculo deste valor? Algumas idéias surgiram como, por exemplo, se o algoritmo de determinação do *score* levava em consideração também a quantidade de memória disponível em cada nó ou mesmo a largura de banda usada pelo nó para se conectar aos demais nós que formam o *cluster*.

A partir dessas idéias, buscou-se inicialmente encontrar algo na literatura publicada na Internet sobre o *openMosix* que pudesse descrever mais detalhadamente o procedimento de geração do *score*. Com isso, encontrou-se várias ocorrências de quatro informações básicas em diversas fontes:

- O que chamamos genericamente de score é denominado *speed* na nomenclatura *openMosix*;
- O valor calculado para o *speed* é relativo ao valor padrão calculado para a "arquitetura padrão", onde "arquitetura padrão" significa um processador Pentium-III/1GHz;
- O valor padrão do *speed* é 10000;
- Esse valor pode ser encontrado examinando-se um arquivo do sistema de arquivos do *cluster*: /proc/hpc/admin/speed. Este é o arquivo lido pelo *openMosixview* para determinar o *speed* do nó.

Visto que basicamente foram encontradas apenas as quatro informações listadas acima, nada especificamente a respeito do algoritmo de determinação do *speed* foi encontrado. Com isso, outra abordagem foi utilizada: analisar o código-fonte do *kernel* do *openMosix*. Como o *openMosix* constitui-se de um *patch* (conjunto de linhas de texto, no caso, código, que representa adição, remoção, ou alteração do conteúdo de arquivos) para as versões 2.4 e 2.6 do *kernel* Linux padrão, ficou decidido que a análise precisava ser feita apenas no único arquivo de patch disponibilizado pelo *openMosix*, para a versão do *kernel* que foi usada nos experimentos, a versão 2.4.

A versão mais atual do *patch openMosix* para o *kernel* 2.4 pode ser obtida em [30]. Depois de baixado e descompactado, buscou-se encontrar alguns trechos de código que revelassem o verdadeiro cálculo do *speed*. Depois de concluída a leitura de alguns trechos que continham palavras-chave como "*speed*", "*processor*" e "*define*", foi possível obter as seguintes informações:

1. Apesar de não estar escrito explicitamente em lugar algum, todas as referências à processador citam apenas o Pentium-III/1GHz;
2. O valor padrão do *speed* no processador padrão (Pentium-III/1GHz) é 15000 e não 10000 como haviam sido encontrados em algumas referências na Internet;
3. O valor gerado do *speed* depende de um contador chamado loops_per_jiffy, presente no código-fonte do *kernel*;
4. O valor do contador loops_per_jiffy para o processador padrão é 9961472;
5. A fórmula usada para calcular o valor do *speed* está presente na função init_mosix() e é a seguinte:

Tabela 4. Fórmula para cálculo do *speed.*

```
cpuspeed = ((int64_t) loops_per_jiffy) * STD_SPD / STD_LOOPS;
```

O valor da variável loops_per_jiffy é calculado pelo próprio *kernel* do Linux no momento da inicialização do sistema [8] e é usada como base para o cálculo do speed. O intuito dessa variável é gerar atrasos de duração mais curta (na ordem de 1 microssegundo) do que os atrasos padrão usados pelo *kernel* como, por exemplo, tempo para verificação de resposta a interrupções (na ordem de milissegundos). Com isso, é natural que computadores com maior valor de *clock* tenham maior valor da variável loops_per_jiffy, o que implica diretamente em um maior valor de *speed*.

Assim, foi concluído que o valor do *speed* é dependente única e exclusivamente da velocidade de *clock* do processador, não levando em consideração outros valores como quantidade de memória ou largura de banda da tecnologia de rede utilizada para interligação dos nós.

5.1.4 *Clusters* x PovRay-PovMosix

O POVmosix é um programa auxiliar, desenvolvido para paralelizar a renderização feita pelo PovRay, num ambiente de *cluster openMosix*. Ele consiste na divisão de uma cena em um conjunto de sub-tarefas (que serão processos independentes) possibilitando a migração de processos entre os nós do *cluster* [31].

O programa PovRay já vem junto com a distribuição *ClusterKnoppix* utilizada nos experimentos, porém o POVmosix, não. Para utilizar o POVmosix, foi necessário baixar seu código-fonte e compilá-lo [31]. Após a compilação, o programa foi executado e configurado. Na configuração, é necessário especificar os seguintes valores:

1. O caminho para o binário povray: /usr/bin/povray;

2. As dimensões da imagem a ser gerada: 10000 x 5500 *pixels*;
3. O arquivo de entrada: /usr/share/doc/povray/pov3demo/anim/diffuse/rainbow1.pov;
4. O arquivo de saída: /home/knoppix/rainbow.png;
5. A quantidade de processos a ser utilizada.

É possível visualizar a divisão feita pelo POVmosix do arquivo de entrada em 16 arquivos distintos, devido ao fato da quantidade de processos ser definida como 16 na configuração, a partir da captura da tela pós-configuração do POVmosix exibida na figura 16 abaixo.

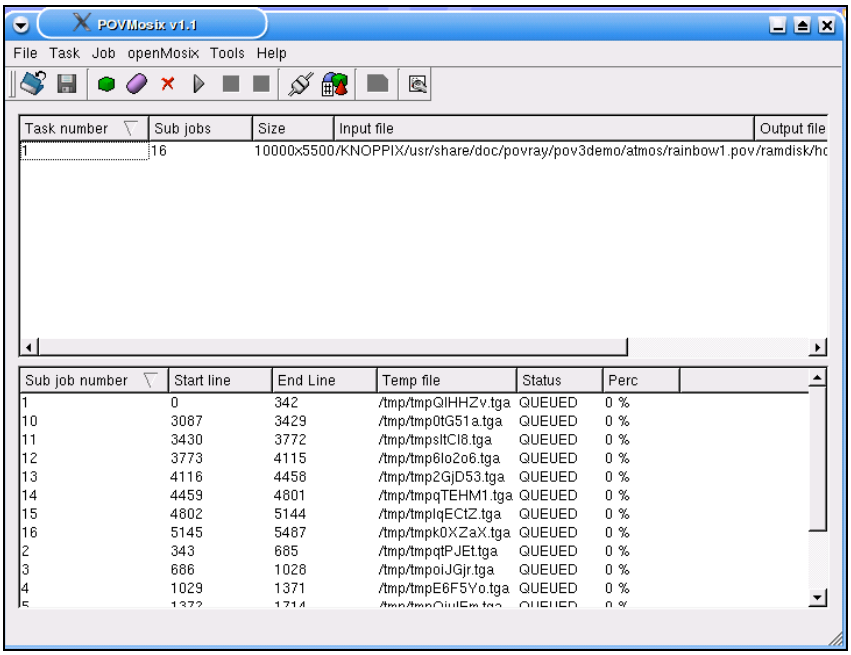

Figura 16. Interface do POVMosix após configuração.

Junto com o pacote de programa PovRay estão contidos vários exemplos de criação de imagens, ou seja, cenas que vem descritas na linguagem de descrição de cena que o próprio programa faz uso. Para os testes, foi utilizado a cena rainbow1.pov, como mostrado na figura 17 abaixo, que é o arquivo de entrada para o programa POVmosix onde está contida toda descrição geométrica para ser formada a imagem depois de ser renderizada, uma cena que acompanha o pacote de programa POVRay.

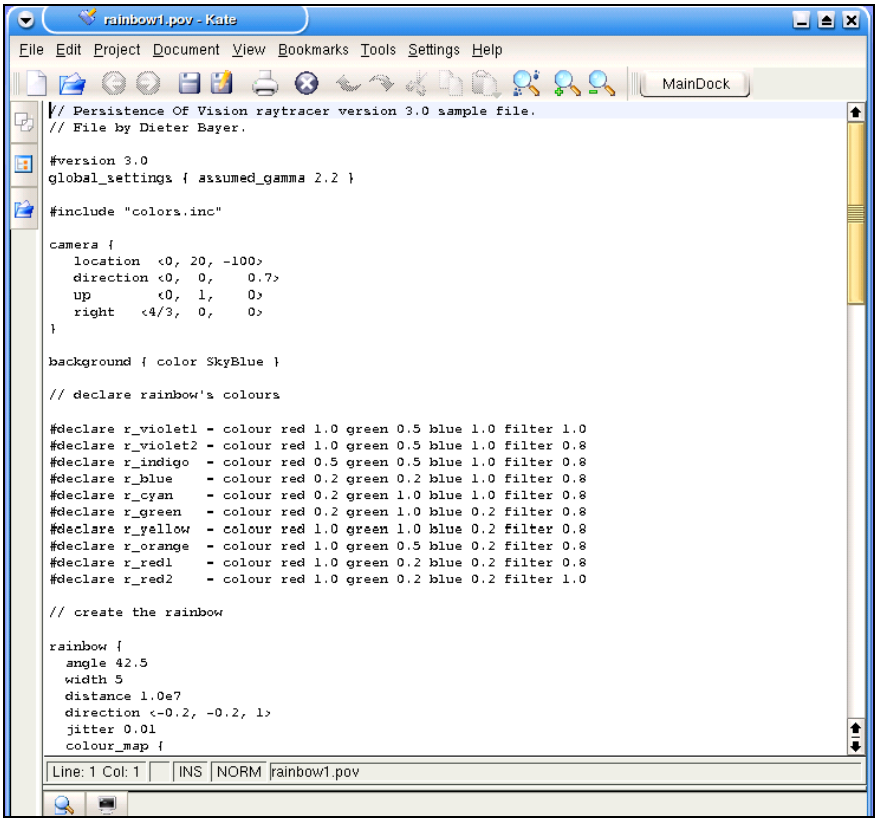

Figura 17. Arquivo de entrada: cena rainbow1.pov.

A cena rainbow1.pov foi renderizada na resolução 10000x5500, uma resolução considerada muito alta. Foi usada essa resolução uma vez que a cena escolhida continha poucos objetos e estes objetos possuiam pouca complexidade. Assim, com o aumento da resolução o custo computacional, durante a renderização, foi bastante elevado, compensando estes poucos objetos presentes na cena. O resultado da renderização pode ser vista na figura 18 abaixo.

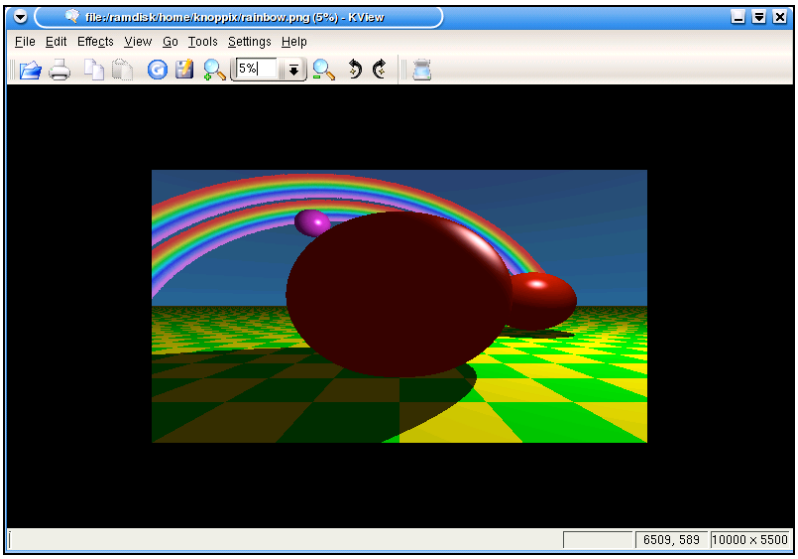

Figura 18. Cena teste após ser renderizada.

5.1.4.1 Comportamento dos Nós do *Cluster* Durante a Renderização

Quando iniciamos a execução do programa de renderização de imagem primeiramente o programa é executado somente no nó que está executando, como mostrado na figura 19 abaixo. Este nó que executa a operação possui o IP 192.168.168.16 e inicialmente fica totalmente sobrecarregado.

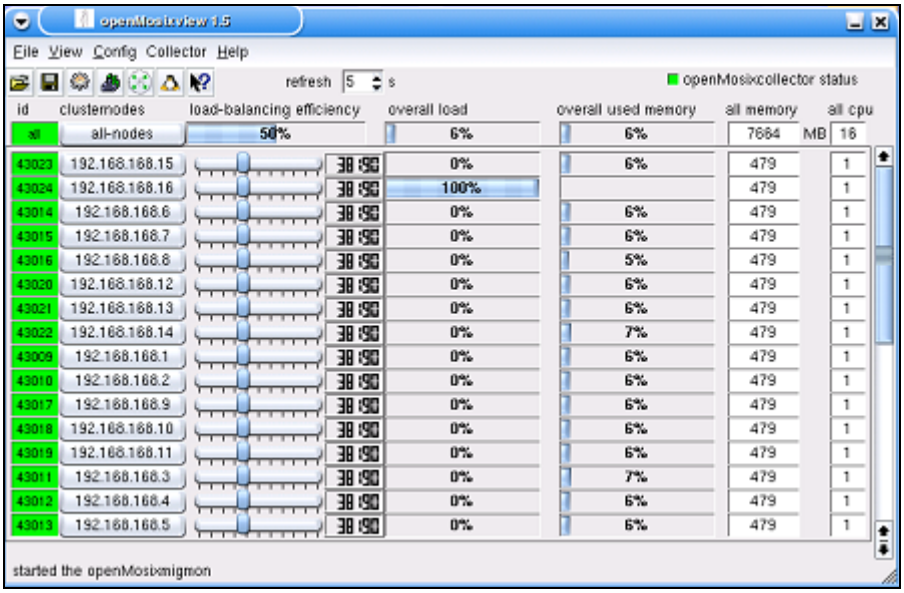

Figura 19. *openMosixView* - Comportamento dos nós quando estamos renderizando (1).

Depois de passar um tempo os processos começam a migrar para outras máquinas do cluster até chegar a um momento que todas as máquinas estão sendo utilizadas e desempenhando o seu papel, como podemos notar na figura 20 abaixo. Isso ocorre devido ao algoritmo de balanceamento de carga que fica executando em todos os nós, tentando fazer a redução da carga entre os nós, migrando os processos de um nó mais sobrecarregado para um nó que possua mais recursos em disponibilidade no *cluster*. Neste momento que foi capturada a tela do *openMosixview* a eficiência do balanceamento de carga foi de 85%.

Com o tempo, depois que todos os nós fizerem seu papel, voltará apenas um nó a ficar em atividade, normalmente este nó é aquele que executa a ferramenta de renderização, e depois deste terminar seu papel a imagem se formará por completo.

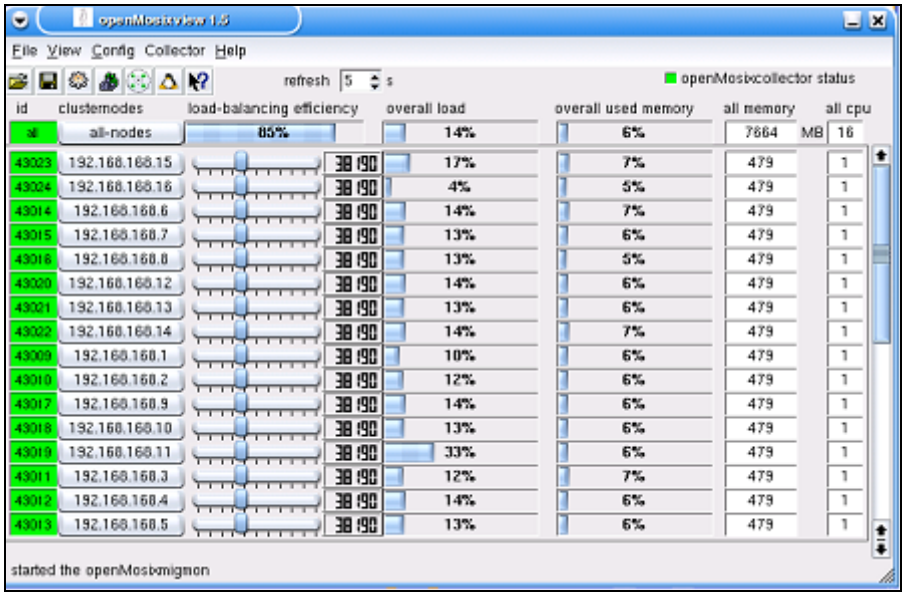

Figura 20. *openMosixView* - Comportamento dos nós quando estamos renderizando (2).

Durante a execução do programa para renderizar a imagem foi feito o uso de uma outra ferramenta chamada de *openMosixmigmon* para fazer a monitoração e as migrações dos processos efetuadas no cluster *openMosix*. A interface dessa ferramenta consiste basicamente em vários pingüins com um círculo em volta representando os nós do sistema, como podem visualizar na figura 21. O círculo maior, localizado no meio da figura, identifica a máquina que está executando o programa de renderização e em torno dessa circulo maior os outros 15 nós que compõem a estrutura maior do cluster, somando-se assim 16 nós no total. Os quadrados pretos, que compõem a circunferência do nó, representam os processos que podem ser migrados. Quando um processo migra para um dos nós, é mostrada uma ligação entre o processo principal (*deputy*) e o processo remoto (*remote*), onde este último é marcado por um quadrado verde, na figura abaixo ficou branco. Uma outra vantagem dessa ferramenta é a possibilidade de arrastar o processo e soltar em um outro nó, permitindo movimentar os processos sem muito esforço.

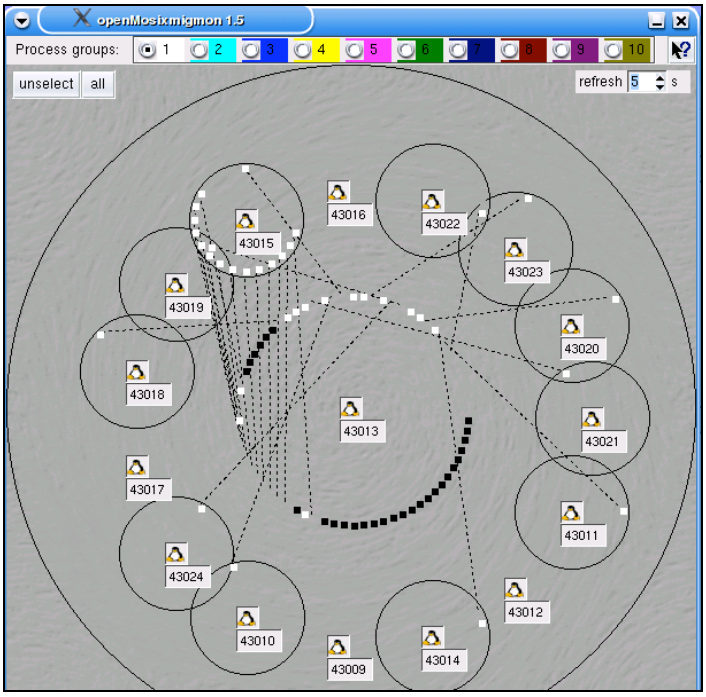

Figura 21. *openMosixmigmon* - Comportamento dos nós quando estamos renderizando .

5.1.4.2 Análise de desempenho do cluster durante a renderização

Nos testes realizados, foram feitas variações na quantidade de processos. Os primeiros testes foram feitos com o número de processos iguais ao número de nós, no segundo foram utilizados 8 processos, e no terceiro utilizados 32 processos.

Foram feitas 3 séries de teste e tirado a média entre eles para que se aumentasse a confiança nos resultados apresentados. O tempo total de simulação foi mais de 12 horas.

Nos estudos de casos foram feitas variações na quantidade de processos com a finalidade de se investigar se tinha alguma vantagem fazer o aumento do número de processos, assim foi escolhido oito processos, que é a metade do número máximo de nós do *cluster*, e de 32 processos, o dobro da quantidade máxima de nós do *cluster*.

5.1.4.2.1 Estudo de Caso 1 : Número de processos igual ao número de nós

Numa primeira situação foi adotada como critério a divisão de processos iguais ao número de máquinas, sendo assim, uma máquina, um processo, duas máquinas, dois processos, assim por diante.

Na tabela abaixo são apresentados os tempos obtidos com a renderização da imagem, a média e o desvio padrão.

Tabela 5. Tempo de renderização no *cluster* quando o número de processos são iguais ao número de nós.

Número de Nós	Teste 1 (s)	Teste 2 (s)	Teste 3 (s)	Média (s)	Desvio Padrão
1	296	296	293	295	2
2	209	209	220	213	6
3	173	173	162	169	6
4	146	144	153	148	5
6	114	111	110	112	2
8	95	92	98	95	3
10	92	85	86	88	4
12	103	86	92	94	9
14	75	99	101	92	14
16	93	92	97	94	3

Através dos dados apresentados acima podemos notar que houve uma diminuição no tempo de renderização, na média de aproximadamente 28% quando se adicionou um nó ao *cluster* e após ultrapassar a faixa de 10 nós observou-se uma estabilidade no tempo de renderização. A maior redução de tempo ocorre quando temos 10 nós chegando a uma redução de aproximadamente 70% do tempo com apenas um nó. No figura abaixo podemos visualizar melhor o comportamento geral do tempo de renderização em função da quantidade de nós do *cluster* para este caso.

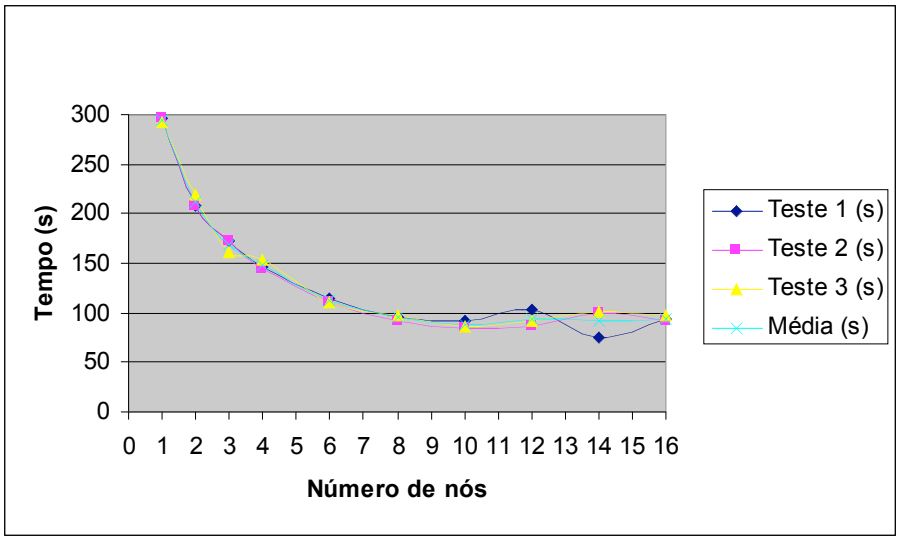

Figura 22. Número de nós x tempo de renderização no *cluster*, com o número de processos igual ao número de nós

5.1.4.2.2 Estudo de Caso 2: 8 Processos

Uma segunda experiência foi realizada deixando o número de processos fixos, nesse caso foram escolhidos oito processos, e os dados obtidos nesses testes estão apresentados na tabela abaixo.

Tabela 6. Tempo de renderização no *cluster* com o número de processos igual a oito.

Número de Nós	Teste 1 (s)	Teste 2 (s)	Teste 3 (s)	Média8 (s)	Desvio Padrão
1	279	279	276	278	2
2	172	170	168	170	2
3	135	131	133	133	2
4	116	118	115	116	2
6	103	99	99	100	2
8	94	92	97	94	3
10	91	93	92	92	1
12	94	93	88	92	3
14	92	92	92	92	0
16	93	95	94	94	1

Um fato a ser observado neste caso, é que quando temos oito nós ou mais, o tempo de renderização é igual ou muito próximo, isso é justificado pelo fato que como temos apenas 8 processos só são usados no máximo 8 nós do *cluster*, assim aumentando o número de nós a mais que oito não fará diferença.

Neste caso, observou-se que apesar de ter comportamento similar ao caso anterior, as reduções no tempo de renderizações, com o aumento da quantidade de nós do *cluster*, eram maiores quando comparados ao caso anterior. Comparando 1 com 2 nós apenas no *cluster* atinge reduções médias de aproximadamente 39%; para 3 nós no *cluster*, houve redução média de aproximadamente 22% comparado ao tempo anterior; quando se ultrapassa a faixa de 8 nós o tempo de renderização tem uma redução pouco significativa.

No gráfico da figura abaixo podemos visualizar mais claramente o comportamento dos dados da tabela.

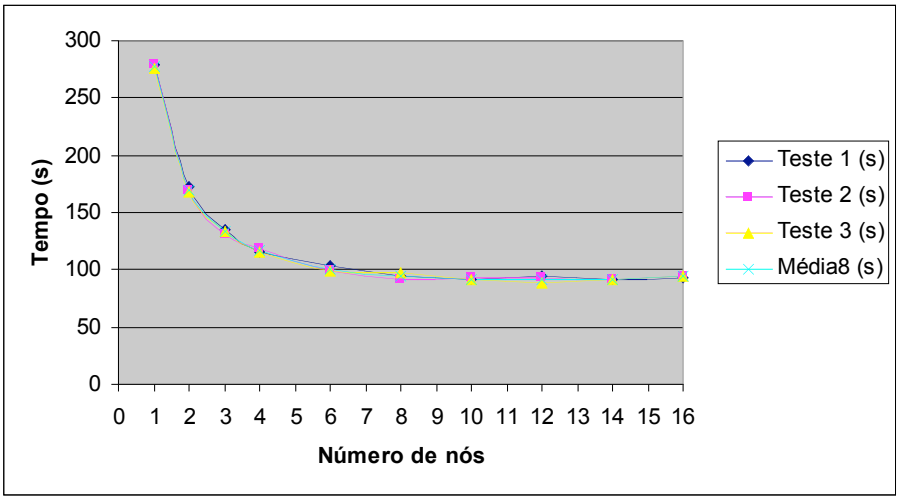

Figura 23. Número de nós x tempo de renderização no *cluster*, com o número de processos fixo e igual a oito.

5.1.4.2.3 Estudo de Caso 3: 32 processos

A terceira experiência resolveu adotar o número de processos muito grande, ou seja, trinta e dois processos. Na tabela abaixo estão os resultados adquiridos depois da realização das simulações provenientes desse caso.

Tabela 7. Tempo de renderização no *cluster* com o número de processos igual a trinta e dois.

Número de Nós	Teste 1 (s)	Teste 2 (s)	Teste 3 (s)	Média32 (s)	Desvio Padrão
1	278	277	273	276	3
2	160	162	161	161	1
3	122	124	123	123	1
4	104	103	104	104	1
6	87	83	92	87	5
8	76	79	95	83	10
10	93	84	95	91	6
12	89	90	92	90	2
14	90	89	91	90	1
16	92	86	89	89	3

Notamos que aconteceu uma redução muito grande se comparado aos dois casos anteriores. Comparando o tempo de renderização com 1 e 2 nós no *cluster*, pudemos observar que houve uma redução média no tempo de aproximadamente 42%. O menor tempo de renderização da imagem teste foi conseguido quando temos 8 nós no *cluster*, onde tivemos uma diminuição de aproximadamente 70% em média se comparado ao tempo com apenas um nó.

Na figura mostrada abaixo fica mais claro o comportamento geral do tempo quando são adicionados nós ao *cluster*.

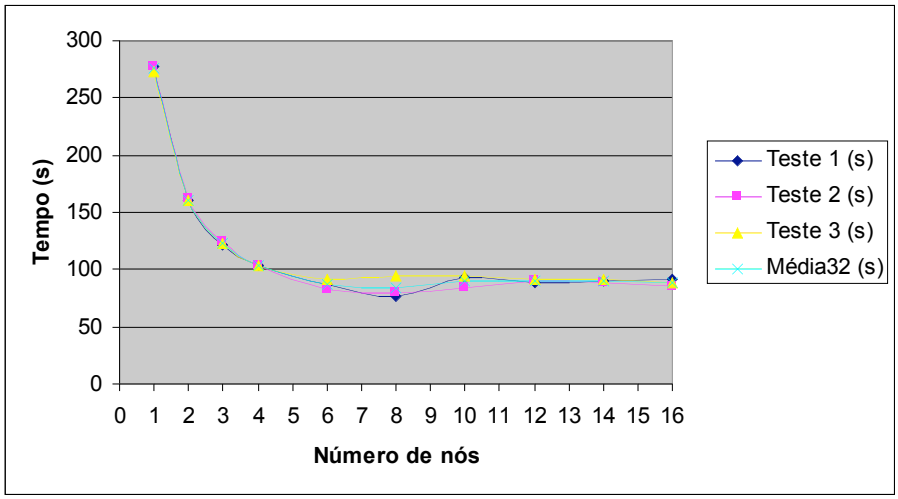

Figura 24. Número de nós x tempo de renderização no *cluster*, com o número de processos fixo e igual a trinta e dois.

5.1.4.2.4 Comparação entre os Estudos de Casos 1, 2 e 3

Fazendo-se uma comparação entre as três situações abordadas nos casos acima podemos visualizar pela figura 25 abaixo, que os três casos possuem gráficos muito parecidos. Mas, temos que considerar que quando temos entre 2 e 8 nós, quando dividimos em 32 processos, tivemos redução significativa no tempo de renderização se comparado ao dois outros casos.

Alguns pontos em comum podem ser notados nas três situações descritas, quando confrontamos os três gráficos e que vale a pena serem relatados:

- A maior redução percentual de tempo de renderização da imagem ocorre quando passamos de um para dois nós no *cluster*;
- Em um determinado momento chega-se a um ponto de saturação, quando aumentando o número de nós no *cluster* se tem nenhuma ou pouca redução significativa. Nos três casos analisados, esse ponto se saturação ficou perto dos 8 nós;
- O aumento da quantidade de nós no *cluster* reduz o tempo de renderização, no entanto, isso ocorre até uma certa quantidade de nós, quando observa-se uma certa estabilidade (8 para o caso analisado) para este caso específico de renderização de imagens;

- Quando se tem um grande número de nós no *cluster*, mais que 8, em alguns momentos, aumentando-se o número de nós tem-se um aumento no tempo de renderização se comparado com situações em que o *cluster* possui um menor número de nós;
- Comparando-se a redução máxima em percentual médio em relação quando só tinha um nó, esse percentual ficou em torno de 70% nas três situações.

Alguns fatos podem justificar o aumento do tempo de renderização quando temos grande quantidade de nós, como, por exemplo: a estrutura da rede utilizada que tem a velocidade de 100 Mbps, padrão *Fast-Ethernet*, o *delay* na rede, pois quando se tem mais máquinas fazendo parte da rede e mandando pacotes conseqüentemente aumenta o número de colisões.

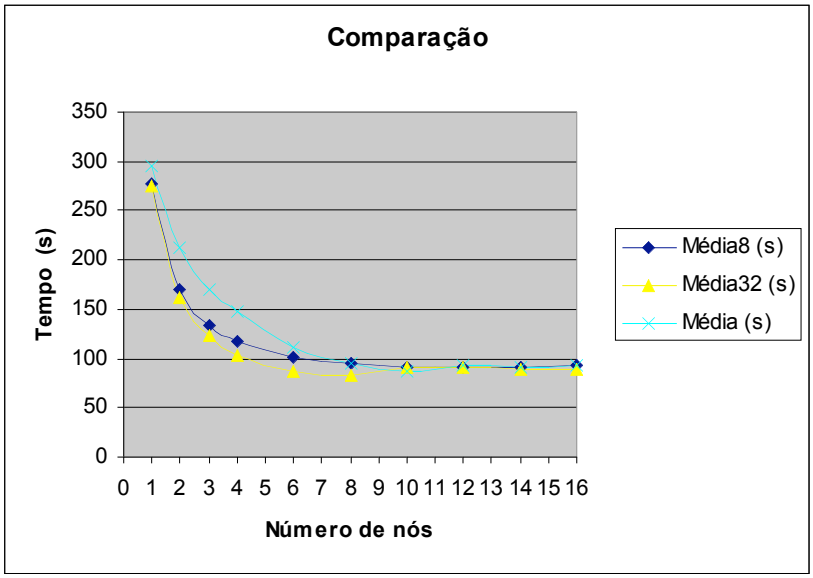

Figura 25. Número de nós x tempo de renderização no *cluster*, comparação entre os casos.

5.2 Arquitetura *Multicore* x Ferramentas de Renderização

Depois dos testes realizados com várias estruturas de *clusters*, foram feitos em arquiteturas *multicore*, assim, foi possível usar computadores com dois núcleos e quatro núcleos.

Um teste inicial foi feito com o renderizador PovRay para saber se ele tira proveito da arquitetura *multicore*, nesse teste foi usado o *quad core*. Foi renderizada uma imagem

exemplo e durante a renderização foi observado o comportamento das quatro CPUs como mostrado na figura abaixo.

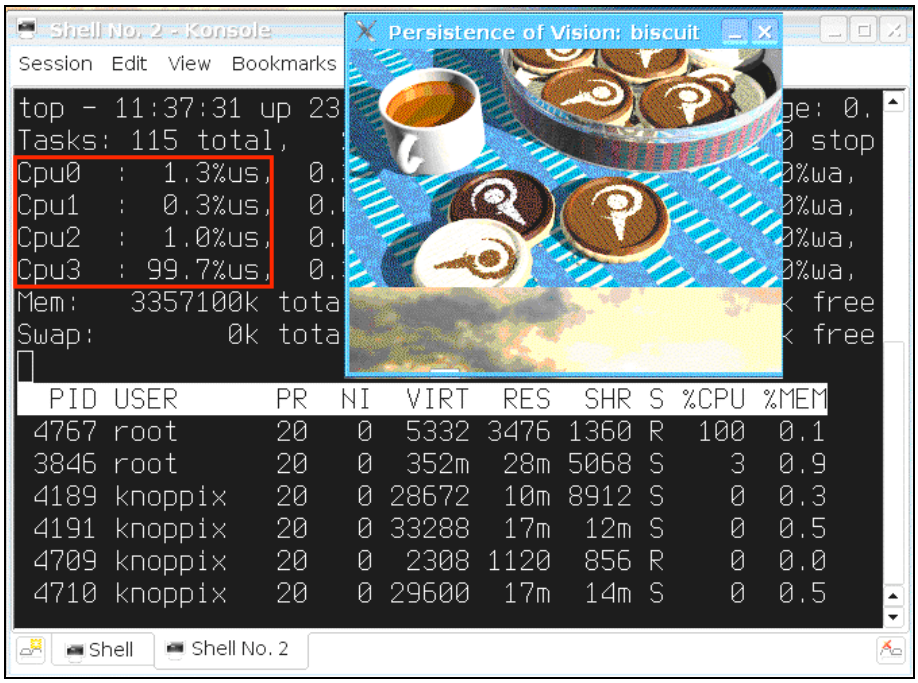

Figura 26. Comportamento das CPUs quando executamos o PovRay.

Fica bem claro que apenas uma CPU fica responsável pelo processo de renderização enquanto a outra não faz absolutamente nada em se tratando de renderizar. Assim, podemos concluir que o PovRay inicia apenas um processo e por isso pode trabalhar, isto é, tirar proveito de apenas uma CPU.

5.2.1 *Core Duo* x PovRay-PovMosix

Uma segunda situação a ser testada foi em um computador com dois núcleos, com as seguintes configurações descritas na tabela abaixo, mas agora usando o PovMosix acoplado ao PovRay.

Tabela 8. Configuração do computador *core duo*.

Componente	Descrição	Quantidade
Processador	Intel Pentium Dual CPU 1.80 GHz	2
Memória *cache* L2	1024 Kbytes	1
Memória	1024 MB	1

Como sabemos que o PovMosix divide a tarefa realizada pelo PovRay em tarefas ou processos bem definidos já era de esperar que se formos configurar a divisão das tarefas em mais de uma, as tarefas seriam divididas entre as CPUs, como mostrado na figura

abaixo. Fica claro que para se tirar proveito dos núcleos a carga deve ser dividida, senão os núcleos adicionais não são utilizados, e é exatamente isso que o PovMosix faz.

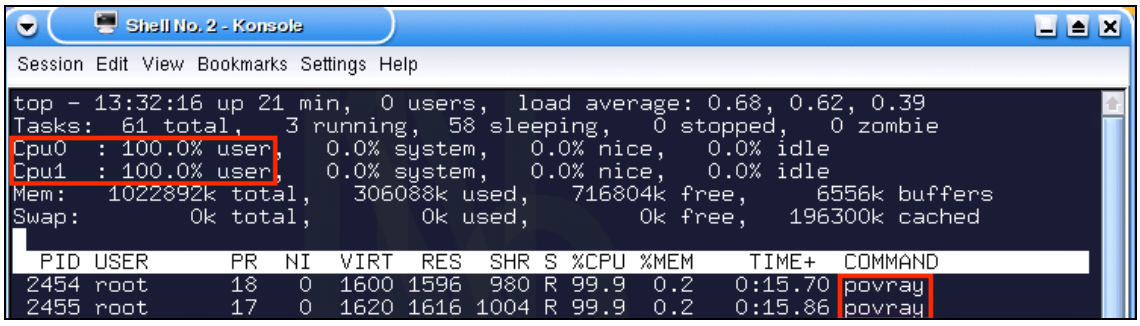

Figura 27. Comportamento das CPUs quando dividimos em 2 processos as atividades para renderização da imagem do processador com dois núcleos

Na tabela abaixo se tem os resultados das simulações feitas com o computador de dois núcleos fazendo alterações nos números de processos.

Tabela 9. Tempo de renderização no *core duo* variando o número de processos.

Número de Processos	Teste 1 (s)	Teste 2 (s)	Teste 3 (s)	Média2 (s)	Desvio Padrão
1	207	202	201	203	3
2	129	130	129	129	1
3	119	119	118	119	1
4	114	115	115	115	1
8	114	114	115	114	1
16	114	115	115	115	1

Podemos constatar pela tabela acima e na figura logo abaixo que o tempo para somente um processo é alto se comparado quando se tem dois, isso se justifica claramente pelo fato de quando se tem apenas um processo só ser usado uma CPU. A redução do tempo de um para dois processos foi aproximadamente de 37% em média, de dois para três processos tem uma queda de 10 segundos em média que chega até ser substancial, mas depois disso, ou seja, aumentando o número de processos para quatro, oito e dezesseis, a melhora é muito pouca, algumas vezes nula, ou até mesmo tem-se uma pequena piora, por isso quase uma constante após passar de 2 processos, como podemos visualizar no gráfico da figura abaixo.

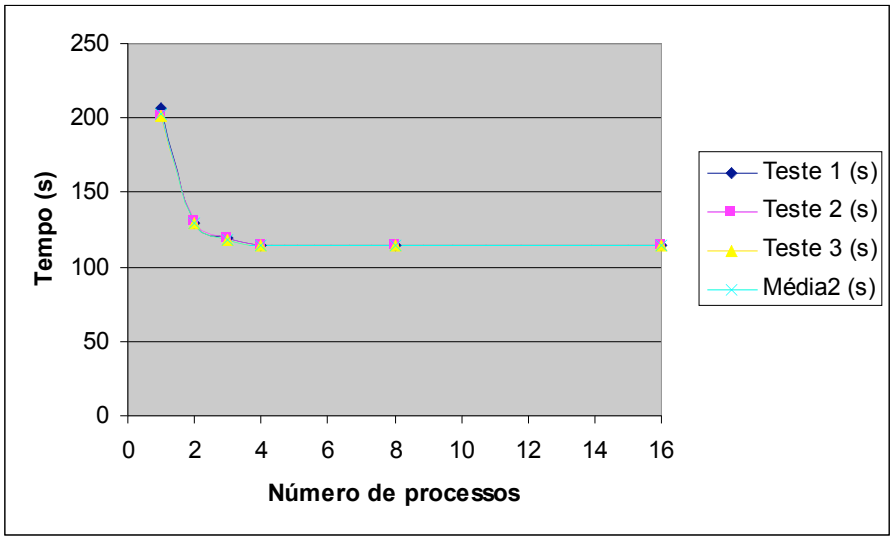

Figura 28. Número de processos x tempo de renderização no *core duo*.

5.2.2 *Quad Core* x PovRay-PovMosix

Depois de realizados os experimentos no *core duo*, foi feito os mesmos testes no *quad core* que possui a seguinte configuração mostrada na tabela abaixo.

Tabela 10. Configuração do computador *quad core*.

Componente	Descrição	Quantidade
Processador	Intel(R) Core(TM) 2 Quad CPU 2.40GHz	4
Memória *cache*	4096 KB	2
Mémoria	3278MB	1

Podemos notar nas três figuras abaixo o comportamento do processador *quad core* quando dividimos as atividades de renderização através do PovMosix. Quando temos dois processos, apenas duas CPUs são utilizadas, quando temos três processos três CPUs são utilizadas e com quatro processos faz usado das quatro CPUs. Para fazer uso de todas as CPUs é necessário no mínimo dividir a atividade de renderização em 4 processos.

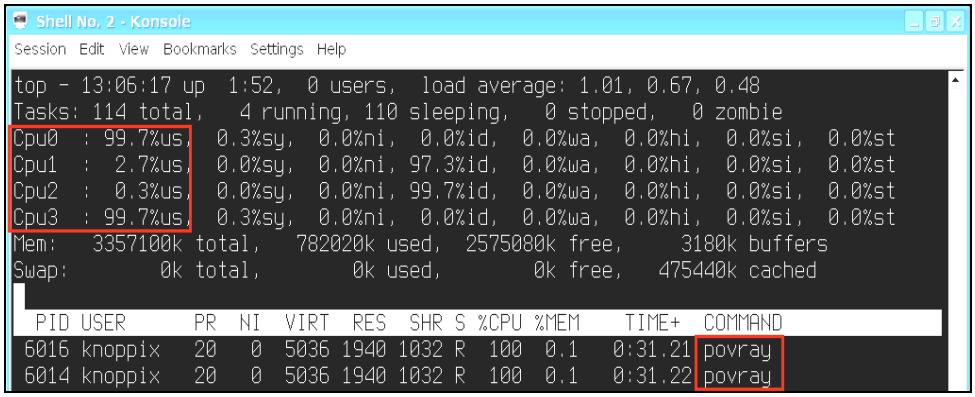

Figura 29. Comportamento das CPUs quando dividimos em 2 processos as atividades para renderização da imagem do processador com quatro núcleos

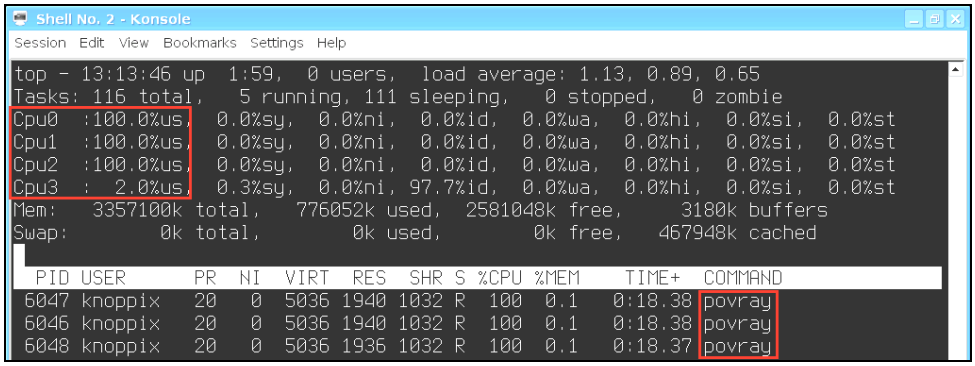

Figura 30. Comportamento das CPUs quando dividimos em 3 processos as atividades para renderização da imagem do processador com quatro núcleos

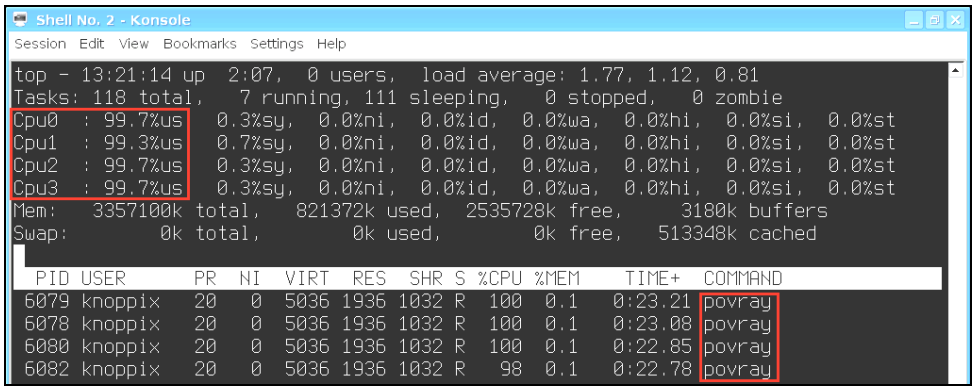

Figura 31. Comportamento das CPUs quando dividimos em 4 processos as atividades para renderização da imagem do processador com quatro núcleos

Na tabela abaixo podemos visualizar que quando temos apenas um processo no processo de renderização o tempo foi alto, pois usou apenas uma das CPUs. Quando dividimos em 2 processos esse tempo caiu em aproximadamente 34% em média, e quando dividimos em 3 processos houve uma queda de aproximadamente 53% em média, e 4 processos a redução foi de aproximadamente 60%. Dividindo em mais de quatro processos a redução de tempo é pouco substancial.

Tabela 11. Tempo de renderização no *quad core* variando o número de processos.

Número de Processos	Teste 1 (s)	Teste 2 (s)	Teste 3 (s)	Média4 (s)	Desvio Padrão
1	146	146	146	146	0
2	102	93	92	96	6
3	69	69	70	69	1
4	59	57	58	58	1
8	54	54	54	54	0
16	55	55	54	55	1

No gráfico da figura abaixo podemos constatar que as reduções no tempo são bastante expressivas até quando se chega a 4 processos, acima de 4 processos o tempo de renderização é praticamente constante.

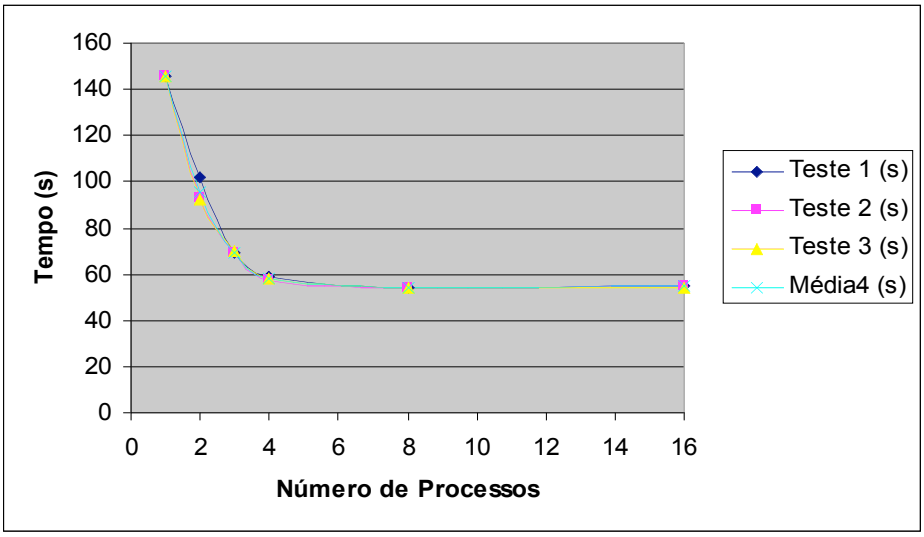

Figura 32. Número de processos x tempo de renderização no *quad core*.

5.3 *Clusters* x Arquiteturas *Multicore*

Fazendo-se uma comparação entre a estrutura de *clusters* quando se usa 32 processos, que foi quando se obteve melhor resultado, e no computador com *core duo* quando se usou 8 processos, podemos constatar que o melhor resultado para o tempo de renderização no *core duo* é semelhante quando se tem entre três e quatro nós no *cluster*. Com isso, cabe a empresa que utiliza esses tipo de ferramenta de renderização, se é mais vantajoso fazer uso de dois ou três computadores com configuração igual ao que foi utilizada na montagem do *cluster* ou comprar um *core duo* utilizado nos testes, isso vai depender dos preços dos computadores no momento da compra.

Quando feita uma comparação entre o melhor tempo de renderização do *quad core* que ocorreu quando foi feita a divisão em 8 processos e o melhor tempo no *cluster* quando se dividiu em 32 processos, podemos chegar ao gráfico da figura abaixo.

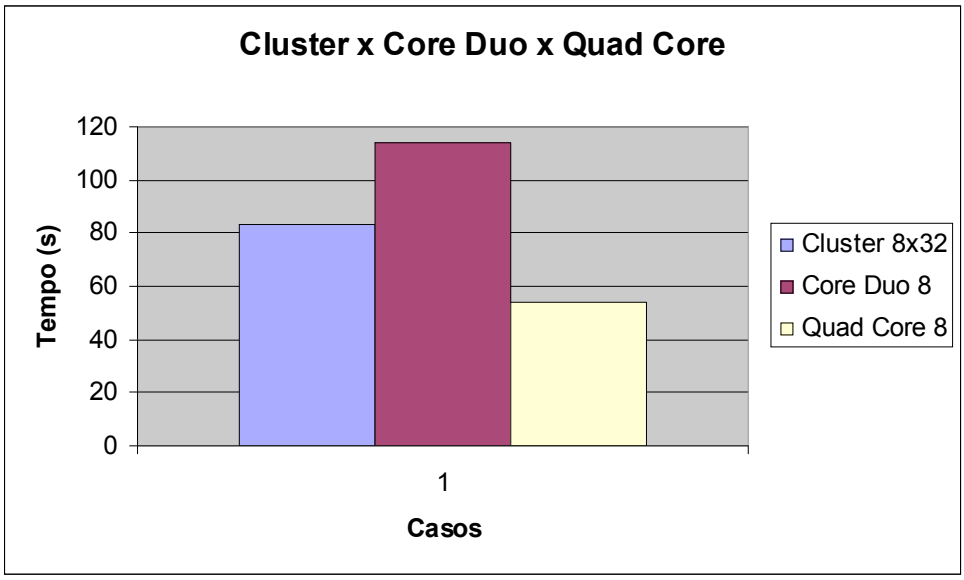

Figura 33. Comparação entre o melhores tempos no *cluster,* no *core duo* e no *quad core.*

Podemos notar pelo gráfico acima que é mais vantajoso, em termos de tempo de renderização, utilizar um *quad core* com as configurações vistas nos testes, do que montar um *cluster* com 8 computadores. A diferença é de 29 segundos a favor do *quad core.*

Capítulo 6

Conclusões e Trabalhos Futuros

Neste trabalho podemos concretizar um estudo aprofundado do ambiente de computação em cluster denominado *openMosix*, com a montagem de uma estrutura e configuração de um *cluster* utilizando esta tecnologia. Executando aplicações de renderização de imagem no *cluster* podemos constatar que aplicações como estas que exigem grande poder de processamento podem tirar vantagens de uma estrutura de *cluster* baseado em computadores pessoais.

Um dos pontos fundamentais que foram observados com o desenvolvimento deste trabalho é que nem todas as aplicações conseguem tirar proveito do alto poder de processamento proporcionado por um cluster *openMosix*. O *openMosix* não paraleliza a aplicação que será executada nele, ele identifica um nó mais livre na rede e migra o contexto de *software* para outro nó, ou seja, aplicações tem que ter como requisitos a compatibilidade com a migração de processos para tirar proveito de um *cluster openMosix*, senão será processada em apenas um nó da estrutura do *cluster*.

No capítulo de estudo de caso foi feito um estudo aprofundado da ferramenta de renderização chamada PovRay e notado que esta não consegue tirar o proveito da estrutura de *cluster* montada, algo que pode parecer um pouco decepcionante, mas é dessa forma que funciona no campo de atuação dos supercomputadores, ou seja, as aplicações precisam ser otimizadas para se tirar proveito de um sistema com computação paralela. Por isso, no estudo dessa ferramenta, foi encontrado um programa auxiliar, o POVmosix, desenvolvido especialmente para paralelizar a renderização feita pelo PovRay, dividindo uma cena em um conjunto de sub-tarefas, que serão processos independentes, assim conseguindo a migração de processos entre os nós do *cluster*.

Depois desse estudo foi feito uma avaliação de desempenho utilizando a integração dessas duas ferramentas (PovRay-POVmosix) e feito uma seqüência de testes utilizando uma cena exemplo para poderem ser tiradas algumas conclusões perante o desempenho do *cluster* tomando por base o tempo de renderização dessa cena e analisando o comportamento dos nós durante esse processo de renderização. Nesses experimentos, o foco era visualizar até que ponto a velocidade de processamento podia ser aumentada adicionando-se mais computadores ao *cluster* e verificando-se a existência de um ponto de saturação. Em outras palavras, observou-se a melhoria de desempenho da aplicação pela adição de um ou mais computaodores no *cluster*.

Algumas das conclusões observadas e comuns a todas as situações propostas nos testes para este modelo proposto para este tipo de aplicação são: o desempenho do *cluster* é não linear, de acordo com aumento do número de nós, ocorre um significante aumento de desempenho entre um e dois nós (chegando-se a quase 50% em algumas situações testadas), depois de oito nós o desempenho não tem um ganho substancial, ao adicionar mais de dez nós o desempenho já não irá aumentar devido a problemas de perfomance da rede.

Além do estudo e testes realizados em estruturas de clusters, foi feito um estudo na área de arquiteturas *multicore*. Foi constatada através dos testes realizados em computadores de dois e quatro núcleos que não é verdadeira a afirmação que núcleos adicionais, em todas as situações, aceleram perceptivelmente programas que são conhecidos por sua necessidade por poder de processamento. Assim como no *cluster*, o PovRay não consegue tirar vantagem da arquitetura *multicore*, ou seja, só consegue fazer uso de um núcleo enquanto o(s) outro(s) ficam sem fazer absolutamente nada em se tratando de tarefas de renderização.

Concluímos novamente, assim como nos experimentos com *clusters*, que a carga deve ser dividida, senão os núcleos adicionais não são utilizados. Mais núcleos nem sempre significam desempenho maior. Para que um aplicativo consiga verdadeiramente uma maior velocidade em sistemas de múltiplos núcleos, ele deve ser paralelizável, ou seja, ter a capacidade de produzir diferentes processos, para que estes possam ser executados simultaneamente.

Com isso foi feito uso novamente da integração PovRay-POVmosix para serem feitos testes para analisar o desempenho dessas ferramentas de renderização em arquiteturas *multicore*. Notou-se que quando dividimos a cena teste em um número de sub-tarefas menores que o número de núcleo, durante o processo de renderização da imagem, só eram utilizados um número de núcleos iguais ao número de sub-tarefas, e quando o número de sub-tarefas era maior que o número de núcleos o tempo de processamento era muito pouco

vantajoso se comparado ao tempo quando o número de sub-tarefas era igual ao número de núcleos.

Como trabalhos futuros podemos utilizar outras ferramentas de renderização de imagens, como o Kendel e/ou Blender, e avaliar seu desempenho fazendo uso de *clusters openMosix* e arquiteturas *multicore.* Também usar esses tipos de ferramentas em outra tecnologia de *cluster* que foi abordada na parte teórica da monografia chamada Beowulf.

Bibliografia

[1] PITANGA, Marcos. Computação em Cluster: O Estado Arte da Computação. Rio de Janeiro: Brasport, 2003.

[2] BUYYA, Rajkumar: *High. Performance Cluster Computing: Architectures and Systems.* Volume 1. New Jersey : Prentice-Hall, 1999. 849p.

[3] PITANGA, Marcos. Construindo supercomputadores com linux. 2ª edição. Rio de Janeiro: Brasport Livros e Multimídia Ltda, 2004. 292p.

[4] HOFF, "*Multicore, SMP and SMT Processors*", Disponível em : http://64.223.189.234/node/13/print Acesso em 24/03/2008.

[5] http://en.wikipedia.org/wiki/Multicore, *Multicore,* acesso em 24/03/2008.

[6] http://pt.wikipedia.org/wiki/Lei_de_Moore, Lei de Moore, acesso em 24/04/2008.

[7] KRAZIT, T.. Microsoft : *Multicore Chips Changing PC Softwares Design.* IDG News Service. Outubro, 2005.

[8] http://blogs.intel.com/brasildigital/ , Brasil Digital@Intel, acesso em 01/04/2008.

[9] http://www1.us.dell.com/content/, *Multi-Core Technology Brief,* acesso em 01/04/2008.

[10] www.ic.unicamp.br/~rodolfo/Cursos/mc722/2s2005/Trabalho/, acesso em 12/04/2008

[11] RIBEIRO, C.C., Rodriguez, N.R. Otimização e processamento paralelo de alto desempenho.

[12] MACHADO, M., Hofmam, M. Sistemas distribuídos MPI, 2005.

[13] Linux Magazine, número 33, pág. 44, edição de Agosto de 2007.

[14] STALLINGS, W., Arquitetura e organização de computadores, 5ª edição.

[15] FILHO, N. A. P. Linux, Clusters e Alta Disponibilidade. Dissertação (Mestrado) Universidade de São Paulo, 2002.

[16] DANTAS, Mario. Computação Distribuída de Alto Desempenho: Redes, Clusters e Grids Computacionais. Rio de Janeiro: Axcel Books, 2005. 262p.

[17] YOKOKURA, Alex Yuichi, Estudo de Viabilidade da Implantação de Técnicas de Cluster ao Projeto Servidor de Estações de Trabalho (SET).

[18] http://www.netlib.org/pvm3/, *Index for PVM3 Library*, acesso em 3/05/2008.

[19]http://www.inf.ufrgs.br/procpar/disc/cmp134/trabs/T2/981/mpi.html, acesso em 3/05/2008.

[20] FREITAS, Evandro Luiz de . Uma comparação entre os modelos de Message Passing MPI e PVM. Disponível em : http://www-usr.inf.ufsm.br/~andrea/elc888/artigos/artigo3.pdf, acesso em 3/05/2008.

[21] http://www.top500.org/, Top500 *Supercomputer sites*, acesso em 3/05/2008.

[22] http://www.beowulf.org , *Beowulf website*, acesso em 3/05/2008.

[23] http://openmosix.sourceforge.net , *openMosix: The openMosix Project*, acesso em 3/05/2008.

[24] http://pt.wikipedia.org/wiki/Renderiza%C3%A7%C3%A3o, Renderização, acesso em 12/05/2008.

[25] D.F. Rogers, *Procedural Elements for Computer Graphics*

[26] Foley et al., *Computer Graphics, Principles and Practice*

[27] Brevíssimo tutorial sobre PovRay . Disponível em: http://www.inf.ufsc.br/~awangenh/CG/raytracing/tutorial/index.html, acesso em 5/05/2008.

[28] http://www.povray.org, PovRay – *The Persistence of Vision Raytracer*, acesso em 3/05/2008.

[29] http://www.inf.pucrs.br/~manssour, Computação gráfica, acesso em 5/05/2008.

[30]http://ufpr.dl.sourceforge.net/sourceforge/openmosix/openMosix-2.4.26-1.bz2, *openMosix kernel patch*, acesso em 10/10/2007

[31] http://povmosix.sourceforge.net, POVmosix, acesso em 30/10/2007