

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE GOIÁS
DEPARTAMENTO DE TELECOMUNICAÇÕES
CURSO DE REDES DE COMUNICAÇÃO

ESLI PEREIRA FAUSTINO JÚNIOR
REINALDO BORGES DE FREITAS

Construindo Supercomputadores com Linux
- Cluster Beowulf -

Goiânia
2005

ESLI PEREIRA FAUSTINO JÚNIOR
REINALDO BORGES DE FREITAS

Construindo Supercomputadores com Linux
- Cluster Beowulf -

Monografia apresentada ao Curso de Redes de Comunicação da CEFET-GO, como requisito para a obtenção do grau de TECNÓLOGO em Redes de Comunicação.

Orientador: Cloves Ferreira Júnior

Mestre

Goiânia
2005

JÚNIOR e FREITAS, ESLI E REINALDO

Construindo Supercomputadores com Linux / ESLI E REINALDO

JÚNIOR e FREITAS - 2005

190.p

1.Construindo Supercomputadores com Linux-Redes. I.Título.

CDD 536.7

CDU 536.21

ESLI PEREIRA FAUSTINO JÚNIOR
REINALDO BORGES DE FREITAS

Construindo Supercomputadores com Linux
- Cluster Beowulf -

Monografia apresentada ao Curso de Redes de Comunicação da CEFET-GO, como requisito para a obtenção parcial do grau de TECNÓLOGO em Redes de Comunicação.

Aprovado em 17 de março de 2005

BANCA EXAMINADORA

Cloves Ferreira Júnior

Mestre

Édio Cardoso de Paiva

Mestre

Paulo César Bezerra Bastos

Mestre

À Deus o autor da vida.

Aos nossos familiares.

À nossa carreira.

Resumo

Trata-se de um trabalho cujo objetivo é propor a construção de supercomputadores com alto poder de processamento com *softwares* livres.

Discorremos sobre conceitos básicos sobre Clusters, vantagens e aplicações. Apresentamos dois tipos de Clusters, Beowulf e OpenMosix, e ferramentas de administração.

Depois expomos um roteiro de implementação de um Cluster Beowulf, no estilo passo-a-passo. Também demonstramos algumas aplicações e funcionamentos das bibliotecas de programação paralela, apresentando os resultados obtidos.

Por fim, tratamos dos ajustes necessários para obtenção de rendimento máximo do Cluster.

Abstract

One is about a work whose objective is to consider the construction of supercomputers with high being able of processing with free softwares.

We discourse on basic concepts on Clusters, advantages and applications. We present two types of Clusters, Beowulf and OpenMosix, and tools of administration.

Later we display a script of implementation of a Cluster Beowulf, in the style step-by-step. Also we demonstrate some applications and functionings of the libraries of parallel programming, presenting the gotten results.

Finally, we deal with the necessary adjustments for attainment of maximum income of the Cluster.

Agradecimentos

À Deus o autor e o consumidor de nossa fé.

Aos nossos pais, irmãos, tios, primos, nossas respectivas namoradas e amigos.

Aos nossos colegas que foram até o final com a gente (Luciano, Maria Amélia, Cristiane, Érika, Sérgio, Náyron), e aos nossos colegas que não puderam concluir conosco.

Aos professores, orientadores pelas críticas e sugestões e por nos honrar com suas presenças em nossa banca examinadora.

Aos professores do Departamento de Redes de Telecomunicações pelos seus ensinamentos e aos nossos colegas e amigos de curso.

Ao professor Cloves por nos ajudar e lutar por esse trabalho. Ao Célio, Wagsley e Cristhian (Universidade Salgado de Oliveira).

Ao Fábio pela força e ânimo para nos ajudar a carregar e montar o cluster.

*” O caminho do homem justo é como a
luz da aurora que vai brilhando mais e
mais até ser dia perfeito”.*

Provérbio de Salomão

Índice

Lista de Figuras	9
Lista de Tabelas	10
1 Introdução	11
2 Cluster	12
2.1 O que é	12
2.2 Tipos de Clusters	13
2.2.1 Cluster de Alta Disponibilidade	13
2.2.2 Cluster de Alta Performance de Computação	14
2.3 Vantagens	15
2.4 Aplicações	17
3 Por que o Linux?	20
4 Beowulf	24
4.1 BProc	24
4.1.1 Processos Ghost	25
4.1.2 Mascaramento de PID	26
4.1.3 Daemons	26
4.2 PVM	27
4.3 MPI	30
5 Administração do Cluster	34
5.1 bWatch - Beowulf Watch	34

6	O Cluster OpenMosix	36
6.1	Cluster OpenMosix	36
6.2	OpenMOSIXVIEW	39
7	Implementação	41
8	Demonstração do Cluster Beowulf	50
8.1	Demonstração do BProc	50
8.2	Aplicações	53
8.2.1	PovRay	53
8.2.2	Compilador Distribuído - Pvmmake	54
8.3	PVM	55
8.4	MPI	59
9	Tuning	63
9.1	O que é	63
9.2	Como fazer	64
9.3	Outras Considerações	67
10	Considerações para trabalhos futuros	68
10.1	Distribuição baseada em Beowulf	68
10.2	Boot Remoto	68
10.3	Distribuição em Floppy	69
10.4	LinuxBios	69
10.5	Sistemas de Arquivos Distribuído	69
11	Conclusão	70
I	Programas Utilizados	71
I.1	Stress1	71

I.2	Stress2	75
II	Arquivos de Configuração	81
II.1	machines.LINUX	81
II.2	.rhosts	81
II.3	.xpvms_hosts	81
II.4	bashrc	82
II.5	mpipov	83
II.6	Makefile.aimk	83
	Bibliografia	99

Lista de Figuras

4.1	Visão Geral - BProc	27
5.1	Bwatch com carga	34
5.2	Bwatch sem carga	35
5.3	Bwatch sem escravo1	35
6.1	OpenMosixView	39
8.1	XPVM - Nós do Cluster	56
8.2	XPVM - Saída	57
8.3	Divisão dos Processos - pvmmake	57
8.4	Utilização - pvmmake	58
8.5	XPVM - Linha do Tempo	60
8.6	Renderização Parcial do POV-Ray	60
8.7	Renderização completa do POV-Ray	61

Lista de Tabelas

8.1	Stress2	53
8.2	Tempo de Compilação do Kernel	58
8.3	Renderização POVRay	59

1 Introdução

O mercado necessita de investimentos tecnológicos, e que geralmente são caros. O Cluster Beowulf pode ser uma solução para empresas de médio porte e universidades. Essas soluções são baratas, o que torna fácil e acessível a criação de supercomputadores. Em consequência promove o desenvolvimento de novas tecnologias, permite renderização de gráficos em alta velocidade, viabiliza simulações e outras tarefas que utilizam grande poder de processamento.

O principal objetivo desse trabalho é avaliar o funcionamento de um Supercomputador Beowulf, apresentando vantagens, desvantagens, viabilidades. Todas as etapas serão demonstradas e explicadas nesse processo de construção do cluster.

2 Cluster

2.1 O que é

Do inglês, cluster significa: grupo compacto de coisas do mesmo tipo; agrupar(-se); juntar(-se). Neste caso, Cluster é um conjunto de máquinas (especificamente, PC's) interligadas via rede que trabalham em conjunto trocando informações entre si em torno de uma única tarefa.[01]

O termo "*clustering*" refere-se atualmente a um número de diferentes tecnologias e configurações.

Um cluster pode ser definido como um conjunto de nós processadores (Personal Computers ou estações) autônomos e que interligados comportam-se como um sistema de imagem única. O conceito de imagem única dita que um sistema paralelo ou distribuído, independente de ser composto por vários processadores ou recursos geograficamente distribuídos, deve comportar-se com um sistema centralizado do ponto de vista do usuário. Dessa forma, todos os aspectos relativos à distribuição de dados e de tarefas, comunicação e sincronização entre tarefas e a organização física do sistema devem ser abstraídos do usuário, ou seja, devem ser transparentes a ele.

Os nós de uma rede tendem a ser menos complexos do que os nós de um cluster, uma vez que em sua maioria correspondem a PCs ou estações monoprocessadas. Os nós de um cluster podem conter dois ou quatro processadores, sendo de igual ou maior complexidade do que máquinas MPP (máquinas proprietárias de processamento massivo), se levarmos em consideração a presença de discos e sistemas operacionais completos no primeiro em comparação com a ausência de discos e sistemas operacionais baseados em microkernel no segundo. Máquinas SMP (máquinas multiprocessadas) geralmente são mais complexas, pois podem conter um número maior de processadores.

O fornecimento de imagem única ou transparência é maior em máquinas SMP, que o suportam em praticamente todos os níveis da arquitetura. Máquinas MPP suportam esse conceito em apenas alguns níveis (por exemplo, aplicação). Os clusters provêem transparência em um nível comparável ao de máquinas MPP - o sistema operacional encarrega-se da gerência dos vários processadores e as ferramentas mais recentes de geren-

ciamento permitem uma visão centralizada de todos os nós do cluster. As redes de estações não suportam esse conceito, visto que a autonomia dos nós impõe múltiplas imagens.

Essas arquiteturas ainda podem ser comparadas em relação ao tipo de sistema operacional que utilizam (monolítico ou modular e heterogêneo ou homogêneo), em relação ao modelo de comunicação empregado (memória compartilhada ou troca de mensagens) ou ainda ao protocolo e tecnologia de rede utilizada.

2.2 Tipos de Clusters

Existem três tipos de clusters: Alta Disponibilidade (HA- High Availability), Alta Performance (HPC- High Performance Computing) e cluster Combo que combina as características dos anteriores. O cluster HA tem finalidade de manter um determinado serviço de forma segura o maior tempo possível. O cluster de HPC é uma configuração designada a prover grande poder computacional, maior que somente um único computador poderia oferecer em capacidade de processamento.

2.2.1 Cluster de Alta Disponibilidade

Um cluster de alta disponibilidade tem como função essencial deixar um sistema no ar vinte e quatro horas por dia, sete dias por semana, ou que não suporte paradas de meia hora ou alguns minutos. São estas paradas não planejadas que influenciam diretamente na qualidade do serviço e nos prejuízos financeiros que estas podem acarretar.

Atualmente os computadores crescem em todos os ambientes empresariais, comerciais, e até mesmo domésticos e nenhum usuário quer que seu equipamento pare de funcionar. E a alta disponibilidade vem a tópicos para resolver esse tipo de situação, garantindo a continuidade da operação do sistema em serviços de rede, armazenamento de dados ou processamento, mesmo se houver falhas em um ou mais dispositivos, sejam eles hardware ou software.

Nos clusters de alta disponibilidade, os equipamentos são usados em conjunto para manter um serviço ou equipamento sempre ativo, replicando serviços e servidores, o que evita máquinas paradas, ociosas, esperando apenas o outro equipamento ou serviço paralisar, passando as demais a responder por elas normalmente. É claro que com isso poderemos

ter perda de performance e poder de processamento, mas o principal objetivo será alcançado, ou seja, não paralisar o serviço.

A disponibilidade dos sistemas torna-se então uma questão vital de sobrevivência empresarial, ou seja, se o sistema parar a empresa pára. Um sistema de comércio eletrônico, como venda de livros por exemplo, não admite indisponibilidade. De maneira geral, um servidor de boa qualidade apresenta uma disponibilidade de 99,5% enquanto que uma solução através de cluster de computadores apresenta uma disponibilidade de 99,99%.[02] Alto desempenho e balanceamento de carga em um cluster não significa necessariamente alta disponibilidade. Como referência podemos citar alguns trabalhos em linux open source sobre alta disponibilidade:

- LVS - Linux Virtual Server: O objetivo neste projeto é o balanceamento de carga e alta disponibilidade em um cluster de servidores, criando a imagem de um único servidor virtual.
- Eddiware : Atua com escalabilidade de servidores web, consistindo em um servidor de balanceamento HTTP, um servidor DNS aperfeiçoado, usa técnicas de duas camadas com o frontend e *backend*, fazendo roteamento de tráfego.
- TurboLinux Cluster: Baseado na solução da distribuição asiática TurboLinux, provê alta disponibilidade para roteadores e servidores, e é baseado no LVS, inserções no kernel(patch), *tunneling*, ferramentas de instalação e configuração.

Então conclui-se que os clusters de alta disponibilidade e balanceamento de carga compartilham diversos módulos como uma solução para uma variedade de problemas relacionados a alto tráfego de sites muito acessados, aplicações de missão crítica, paradas programadas, serviços contínuos, dentre outros.

2.2.2 Cluster de Alta Performance de Computação

Este tipo de cluster tem como foco o alto desempenho(o que é a meta do nosso trabalho), algoritmos de processamento paralelo, construções de aplicações paralelas.

Um cluster de computadores pode ser visto como uma solução alternativa para universidades e empresas de pequeno e médio porte, para obterem processamento de alto desempenho na resolução de problemas através de aplicações paralelizáveis, a um custo

razoavelmente baixo se comparado com os altos valores necessários para a aquisição de um supercomputador na mesma classe de processamento.

Quando se fala em processamento de alto desempenho, que pode ser traduzido em processamento paralelo e processamento distribuído, muitas pessoas pensam em grandes máquinas dedicadas (supercomputadores), que custam milhões de dólares, difíceis de serem operados e com salas superprotegidas. Mas hoje em dia, devido aos clusters, os custos foram reduzidos e existem máquinas muito rápidas, o que viabiliza o uso do processamento de alto desempenho na solução de problemas em diversas áreas.

A vantagem do custo é óbvia, pois uma dúzia de estações de trabalho custa muito menos do que o mais barato dos supercomputadores. Outra vantagem é a facilidade de se montar um cluster: tudo o que se precisa são duas ou mais estações de trabalho ou computadores pessoais conectados por uma rede padrão, como *Ethernet* por exemplo, e todo o software necessário pode ser obtido em um domínio público através da Internet.

O cluster HPC é um tipo de sistema para processamento paralelo ou distribuído que consiste de uma coleção de computadores interconectados, trabalhando juntos como um recurso de computação simples e integrado. Um nó do cluster pode ser um simples sistema multiprocessado (PCs, estações de trabalho ou SMPs) com memória, dispositivo de entrada/saída de dados de um sistema operacional. No entanto, esse sistema pode fornecer características e benefícios (serviços rápidos e confiáveis) encontrados somente em sistemas com memória compartilhada (Multiprocessadores Simétricos- SMP), como os supercomputadores.

2.3 Vantagens

Com a adoção de um cluster, sua empresa passa a dispor de recursos computacionais equivalentes aos de um supercomputador de milhões de dólares com custos incomparavelmente menores.

Grandes corporações precisam de grande poder computacional a um baixo custo para renderizar gráficos em alta velocidade, prever o clima, fazer simulações de explosões atômicas e outras tarefas que exigem máquinas com alto desempenho. Empresas que trabalham com serviços de missão crítica precisam de alta disponibilidade para garantir que equipamentos hospitalares e sistemas públicos de emergência estejam sempre acessíveis e funcionando. Manter apenas um computador realizando apenas uma tarefa importante, não é garantia

segura de que o serviço vai estar sempre disponível, pois problemas de hardware ou software podem causar a interrupção do serviço.

Os clusters fornecem desempenho e tolerância a falhas, não encontradas em qualquer sistema com Multiprocessamento Simétrico (SMP). O desempenho é escalável através do simples acréscimo de computadores ao sistema. Os clusters também oferecem maior disponibilidade, pelo fato de que se um nó falhar, os outros podem continuar fornecendo serviços de dados e a carga de trabalho dos nós defeituosos pode ser redistribuída para os nós restantes.

Dentre inúmeras vantagens em se utilizar clusters, pode-se destacar algumas:

- **Alto Desempenho** - possibilidade de se resolver problemas muito complexos através do processamento paralelo, diminuindo o tempo de resolução do mesmo;
- **Escalabilidade** - possibilidade de que novos componentes sejam adicionados à medida que cresce a carga de trabalho. O único limite é capacidade da rede;
- **Tolerância a Falhas** - o aumento de confiabilidade do sistema como um todo, caso alguma parte falhe;
- **Baixo Custo** - a redução de custo para se obter processamento de alto desempenho utilizando-se simples PCs;
- **Independência de fornecedores** - utilização de hardware aberto, software de uso livre e independência de fabricantes e licença de uso.

Há algum tempo atrás, o paralelismo era vista como uma rara, exótica e interessante sub área da computação, mas de uma pequena relevância para o programador comum.

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

Atualmente, diversas instituições científicas possuem clusters com um porte e poder computacional razoável, com número de computadores variando desde dois até milhares de

nós. No site *www.top500.org*, de atualização semestral, encontra-se uma relação dos supercomputadores proprietários e dos clusters que possuem a maior capacidade de processamento do mundo [03][04].

2.4 Aplicações

Clusters são recomendados para empresas que rodam sistemas pesados, banco de dados robustos, servidores de redes sobrecarregados, engenharia e aplicações de cálculos científicos, processamento de logs em provedores, portais de grande acesso e sites que enviam diariamente milhões de e-mails.

A área de aplicações dos clusters é muito diversificada. Em qualquer local onde tivermos um grande problema computacional em que o processamento seja considerado uma vantagem, pode ser indicada a utilização de um cluster. O processamento paralelo pode ser explorado através do desenvolvimento de aplicações paralelas, que é uma tarefa significativamente mais complexa e difícil que o desenvolvimento de aplicações sequenciais (aplicações processadas em um único processador)[05]. Algumas necessitam de enormes quantidades de memória, algumas possuem tempo de processamento gigantesco, outras fazem o uso elevado de comunicação. Em geral, elas resolvem problemas fundamentais que possuem grande impacto econômico e científico. São tidas como impossíveis sem o uso de modernos computadores paralelos, por causa do tamanho das suas necessidades em matéria de tempo de processamento, memória e de comunicação.

Os computadores cada vez se tornam mais rápidos, devido ao fato de que uma nova tecnologia em particular satisfaça as aplicações conhecidas, novas aplicações ultrapassarão o limite destas tecnologias e demandarão o desenvolvimento de novas tecnologias. Tradicionalmente, o desenvolvimento de computadores tem sido motivado por simulações numéricas de sistemas complexos como tempo, clima, dispositivos mecânicos, circuitos eletrônicos, processos de manufaturamento e reações químicas.

Estas aplicações incluem videoconferência, ambientes de trabalhos cooperativo, diagnósticos em medicina, base de dados paralelos utilizados para suporte a decisões e gráficos avançados em realidade virtual, particularmente na indústria do entretenimento. Por exemplo, integração de computação paralela, redes de alto desempenho e tecnologias de multimídia estão conduzindo para o desenvolvimento de servidores de vídeo, projeto de computadores para servir centenas ou milhares de requisições simultâneas para vídeos de tempo real.

Alguns exemplos de áreas onde a utilização de clusters pode ser indicada são:

- **Servidores de Internet** - o grande crescimento de utilização de Internet revelou fragilidades nos sites muito visitados. Um cluster pode distribuir a carga e aumentar a capacidade de resposta;
- **Segurança** - a grande capacidade que o processamento paralelo oferece beneficiará qualquer processo para identificação, quebra na segurança (criptografia) e verificação de possíveis soluções;
- **Bases de Dados** - pesquisas intensivas (cujo o tempo-resposta seja pequeno) em banco de dados podem demorar muito tempo em um sistema comum. A utilização de um cluster pode reduzir esse tempo significativamente;
- **Computação Gráfica** - nesta área, é muito comum o tempo de processamento ser uma limitação para a evolução da qualidade dos projetos. A utilização de um cluster pode diminuir o tempo de renderização de imagens durante a elaboração de um filme, por exemplo;
- **Aerodinâmica** - produções de novas capacidades tecnológicas e econômicas na pressão enfrentada em aeronaves, lançamento de naves espaciais e nos estudos de turbulência;
- **Análise de elementos finitos** - cálculos de barragens, pontes, navios, aviões, grandes edifícios, veículos espaciais;
- **Aplicações em sensoriamento remoto** - análise de imagens de satélite para a obtenção de informações sobre a agricultura, florestas, geologia, fontes híbridas;
- **Inteligência artificial e automação** - processamento de imagens, reconhecimento de padrões, visão por computador, reconhecimento de voz, máquinas de interferência;
- **Engenharia Genética** - projeto Genoma;
- **Exploração sísmica** - empregado especialmente pelas companhias petrolíferas para determinação de local de poços de petróleo;

- **Oceanografia e astrofísica** - exploração de recursos dos oceanos, estudo da formação da terra, dinâmica das galáxias;
- **Previsão do tempo** - um processo demorado e com pouca precisão quando gerado através de computadores seqüenciais;
- **Pesquisas militares** - projeto de armas nucleares, simulação dos efeitos das armas, em especial as radioativas, processamento de sinais de radares para o comando de mísseis antibalísticos, geração automática de mapas, acompanhamento de submarinos;
- **Problemas de pesquisa básica** - em química, física e engenharia, tais como mecânica quântica, mecânica, estatística, química de polímeros, crescimento de cristais, análise de trajetórias de partículas, dinâmica dos fluidos, teoria do campo quântico, dinâmica molecular, equações de circuitos de alta escala, distribuição de conexões em circuitos VLSI;
- **Segurança de reatores nucleares** - análises das condições do reator, controle automático, treinamento através de simulação de operações, atuação rápida em caso de acidente;[01]

3 Por que o Linux?

Hoje em dia o Linux é um clone do Unix, capaz de rodar X Windows, TCP/IP, Emacs, UUCP, mail e muitos outros recursos consagrados. Quase todos os programas de domínio público têm sido portados para o Linux. *Softwares* comerciais também têm recebido versões para Linux. A quantidade de dispositivos suportados pelas versões de hoje (2.6.x) também foi consideravelmente ampliada em relação às suas primeiras versões.

O Unix é um dos sistemas operacionais mais populares disponíveis hoje em dia. Um dos motivos dessa popularidade é o grande número de arquiteturas que rodam alguma versão Unix. Ele foi originalmente desenvolvido na década de 1970, nos laboratórios Bell, como um sistema multitarefa para minicomputadores e computadores de grande porte. As primeiras versões do Unix tiveram seu código licenciado para diversas universidades e centros de pesquisa, onde gerou uma legião de fãs e desenvolvedores. Isso contribuiu também para o desenvolvimento e implementação de várias versões de Unix, muitas das quais continuaram a ser desenvolvidas depois que o código do Unix passou a não ser mais licenciado academicamente, como é o caso da Universidade de Berkeley (BSD- Berkeley Software Distribution).

A partir do momento em que o código do Unix não pode mais ser utilizado livremente, alguns programadores partiram para implementações próprias do sistema operacional Unix, como é o caso do Minix (Tanenbaum, 1986) e do Linux por Linus Torvalds.

O sistema operacional Linux é uma implementação independente da especificação para sistema operacional POSIX, com extensões System V e BSD. O linux é distribuído gratuitamente nos termos da Licença Pública GNU (*GNU General Public Licence*), sendo que não existe código de propriedade em seu pacote de distribuição. Ele funciona em IBM PCs e compatíveis com barramentos ISA, EISA ou PCI e processadores 386 ou superiores, existindo também implementações para outras arquiteturas.

O Linux foi originalmente desenvolvido como passatempo por Linus Torvalds na Universidade de Helsinky, na Finlândia. O desenvolvimento do Linux contou, no entanto, com a colaboração e ajuda de muitos programadores e especialistas em Unix através da Internet. Ele foi inspirado no sistema operacional Minix, um sistema Unix de pequeno porte desenvolvido por Andrew Tanenbaum (1986), sendo que as primeiras discussões sobre o

Linux apareceram na lista de discussão do Minix (`comp.os.minix`). A idéia era desenvolver um sistema operacional Unix para máquinas de pequeno porte que explorasse novas funcionalidades disponíveis nos processadores 80386, como interface de modo protegido. Facilidades estas que não eram exploradas pelo Minix.[06]

A distribuição Linux que adotamos neste trabalho é o Conectiva, devido a facilidade de instalar e remover aplicativos através do `apt-get` (aplicação que facilita a instalação e remoção de pacotes) e principalmente por ser uma distribuição brasileira que possui manuais traduzidos para o português. Mais observa-se que todos aplicativos utilizados neste trabalho estão no formato RPM e tar.gz.

Muitas pessoas novas ao software livre encontram-se confusas porque a palavra "free" no termo "free software" não é usada como elas esperam. Para eles 'free' significa "sem custo". Um dicionário de inglês lista quase vinte significados para a palavra "free". Apenas uma delas é "sem custo". O resto se refere à liberdade e falta de obrigação. Quando falamos de Software Livre falamos de liberdade, não de preço.

Software que é 'livre' apenas no sentido de não ter de pagar para usar é dificilmente 'livre' no final das contas. Você pode ser impedido de passá-lo para frente e quase certamente impedido de melhorá-lo. Software licenciado sem custo é normalmente uma arma numa campanha de marketing para promover um produto ligado a ele ou para tirar um competidor menor do mercado. Não há garantia de que ele continuará 'livre'.

O verdadeiro *software* livre será sempre livre. Software que é colocado no domínio público pode ser pego e colocado em programas não-livres. Quaisquer melhoras feitas, assim, são perdidas para a sociedade. Para ficar livre, o software precisa ter copyright e licença.

Para os não-iniciados, ou um software é livre ou não. Para entender que tipos de coisas as pessoas estão colocando quando chamam o software de 'software livre' apresentaremos conceitos sobre licenças de *software*.

Os *copyrights* são um método de proteger os direitos do criador de certos tipos de trabalhos. Na maioria dos países, o software que você escreve tem automaticamente um copyright. Uma licença é o jeito de o autor permitir o uso de sua criação (*software* nesse caso) a outros, de maneiras aceitáveis para ele. Cabe ao autor incluir uma licença que declara em que maneiras o software pode ser usado.[07]

Claro, diferentes circunstâncias chamam por diferentes licenças. As companhias de software estão procurando proteger-se para apenas lançarem código compilado (que não é legível para humanos) e colocar muitas restrições no uso do software.

Software livre (open-source) é um conceito de extrema importância no mundo da computação. De forma básica, quando um software é livre, significa que seu código-fonte está disponível para qualquer um e você pode alterá-lo para adequá-lo às suas necessidades, sem ter que pagar. Portanto, software livre, é de fato gratuito, mas usar este termo somente para designar *softwares* sem custo, é um erro grosseiro.

Uma das maiores vantagens do Software Livre é sua colaboração com qualquer projeto ou trabalho de inclusão digital, seja por meio da redução de custos de licenças de software, seja por redução no preço da aquisição de hardware ou ainda na otimização de pessoal. Mas nem só disso vive a inclusão digital. Também se faz necessária a disponibilização de código-fonte como forma de ampliação e compartilhamento do conhecimento para seus usuários. E essa é uma das premissas do software livre, a partilha do código e sua disponibilização para quem desejar.

No caso do Linux, a sua licença de uso, é a GPL. A GPL, sigla para GNU Public License, é uma das formas mais conhecidas de distribuição de programas. A maior parte dos *softwares* para Linux é baseada na licença GPL. Vale dizer que uma licença é um documento que permite o uso e distribuição de programas dentro de uma série de circunstâncias. É uma espécie de copyright (direitos autorais) que protege o proprietário do programa. Tendo copyright, o dono pode vender, doar, tornar freeware, enfim.

No nosso caso, a licença GPL faz exatamente o contrário. Ela permite que você copie o programa, instale em quantos computadores quiser, veja, estude, altere o código-fonte e não pague nada por isso. A GPL não é simplesmente um texto que diz o que você deve fazer para desenvolver um software livre. É, resumidamente, um documento que garante a prática e existência do mesmo. Sua filosofia consiste em defender vários pontos, dentre os quais, destacam-se os mais importantes abaixo:

- Liberdade para executar um programa para qualquer finalidade;
- Liberdade para estudar um programa, e adaptá-lo às suas necessidades;
- Liberdade de distribuir cópias e assim ajudar um colega, uma instituição qualquer;
- Liberdade de melhorar o programa e entregá-los à comunidade.

Para um software ter licença GPL, deve seguir essas quatro liberdades. Esta é uma licença pertencente à Free Software Foundation, que como o próprio nome diz, é uma organização que trabalha em prol do software livre. É válido dizer que o conceito de software livre

não se aplica somente ao Linux. Qualquer programa, independente da plataforma, pode ter código aberto. O navegador de Internet Mozilla por exemplo, possui código fonte disponível tanto para Linux, como para Windows e outros sistemas operacionais.

O software livre, sem dúvida, é essencial não só para a concepção e uso de programas, mas também por ser de grande importância em pesquisas e avanços tecnológicos, principalmente em países com problemas financeiros.

4 Beowulf

Em 1993, Donald Becker e Thomas Sterling começaram a esboçar um cluster que seria uma alternativa de baixo custo aos grandes supercomputadores. Em 1994 o projeto Beowulf foi iniciado no CESDIS (Center of Excellence in Space Data and Information Sciences) da NASA, sob o patrocínio do projeto HPCC/ESS (High Performance Computing Cluster/Earth and Space Sciences).

O protótipo inicial consistia em um conjunto de 16 computadores Intel 486 DX4 conectados por uma rede Ethernet 10Mbps. O cluster teve um sucesso imediato e sua idéia de usar "produtos de prateleira" para satisfazer exigências computacionais específicas fez com que ele se espalhasse dentro da NASA e nas comunidades acadêmicas e de pesquisa.[08]

O Beowulf é portanto uma pilha de PCs comuns, com algumas características especiais:

- Nenhum componente feito sob medida;
- Independência de fornecedores de hardware e software;
- Periféricos escaláveis;
- Software livre e de código aberto.

O Beowulf pode ser implementado através de modificações no kernel do Linux, ou através do uso de ferramentas e bibliotecas de programação específicas para esse fim[09][10]. Em todos os casos, o objetivo é permitir a distribuição das tarefas entre os PCs que fazem parte do cluster.[11][03]

4.1 BProc

O BPROC(Beowulf Distributed Process) tem como objetivo criar um sistema de visão única para os processos (Single Process Space) em um cluster Beowulf. BProc consiste em quatro partes básicas. No nó mestre, há o "processo ghost" que suporta todos os processos remotos. Há também o daemon mestre que determina as mensagens para o

sistema e também mantém a informação do estado sobre a existência e a localidade dos processos. Nos nós slave há o "ID process" que engana o sistema fazendo com que os processos que estão no nó sejam enxergados no mestre. Há também um daemon simples no lado slave que comunica o kernel dos nós escravos com a rede. O BPROC é um conjunto de modificações no kernel (núcleo do sistema operacional), utilitários e biblioteca para a função de iniciar processos de usuários em outras máquinas em um cluster Beowulf.

4.1.1 Processos Ghost

Estes representam no *frontend*, os processos remotos, que executam nos restantes nós sobre a monitoração dos slaves daemons. Estes processos não têm espaço de memória, arquivos associados nem contexto. No entanto, possuem signal handlers e filas de mensagens.

A sua funcionalidade é o seguinte:

- (1) Encaminham os sinais que recebem para os respectivos processos remotos que representam;
- (2) Efetuam `fork()` (função da linguagem de programação "C" que divide processos) a pedido do respectivo processo remoto. Quando um processo remoto pretende efetuar `fork()` é necessário obter o PID do processo filho a ser criado a partir do master daemon, que é quem gera o espaço de processos distribuídos. Isto é conseguido através do seu processo ghost, que efetuam `fork()` e retorna o PID do filho ao nó remoto através do Mascaramento de PIDs efetuado pelo slave daemon no nó de destino;
- (3) Sempre que um processo efetua `wait()` o respectivo ghost deve fazer o mesmo para impedir a acumulação de processos ghost zombies na árvore de processos;
- (4) Sempre que um processo remoto efetua um `exit()` o seu código de terminação é enviado ao respectivo ghost que também efetua o `exit()` com mesmo código de terminação, o que permite que `wait()` no processo pai remoto receba a informação sobre o término do respectivo processo filho (que poderia encontrar-se em execução em qualquer outro nó do sistema).

No entanto, estes processos espelham o estado dos processos remotos que são representados. A informação acerca desses processos remotos é representada no `/proc` file system em vez da informação acerca dos próprios processos ghost, que são invisíveis para o utilizador. Essa informação é atualizada a pedido (por consulta de `/proc` utilizando o comando `top` por exemplo) e também periodicamente. Neste sentido, `/proc` passa a ser designado Unified `/proc` File System, ou espaço unificado no sistema de arquivos `proc`.

4.1.2 Mascaramento de PID

Os nós remotos executam em uma parte da sua árvore de processos de frontend. Para além desses processos podem existir outros processos iniciados por `rsh` por exemplo, que devem coexistir com os processos remotos controlados pelo BPROC, iniciados utilizando as primitivas indicadas. Quando se desloca um processo de um nó para o outro, o seu PID não deve mudar devido as dependências que existem entre diversos processos. Para que tal seja possível, o slave daemon cria e controla um espaço de processos local que é um subconjunto do espaço de processos do master daemon. Neste subespaço, os processos têm o mesmo identificador dos processos ghost correspondentes - isto designa-se de Mascaramento de PID. Podem eventualmente existir outros processos no nó associados a diferentes daemons e conseqüentemente a diferentes espaços de processos que não interferem entre si.

4.1.3 Daemons

O master daemon armazena a informação sobre as configurações dos nós e conhece a localização de todos os processos no seu espaço de endereçamento. É responsável pela criação da imagem de uma única árvore de processos. Os slaves daemons atuam como intermediários entre os processos remotos e o master daemon. Representam um subconjunto do espaço de processos global e efetuam o Mascaramento PID nesse subespaço.

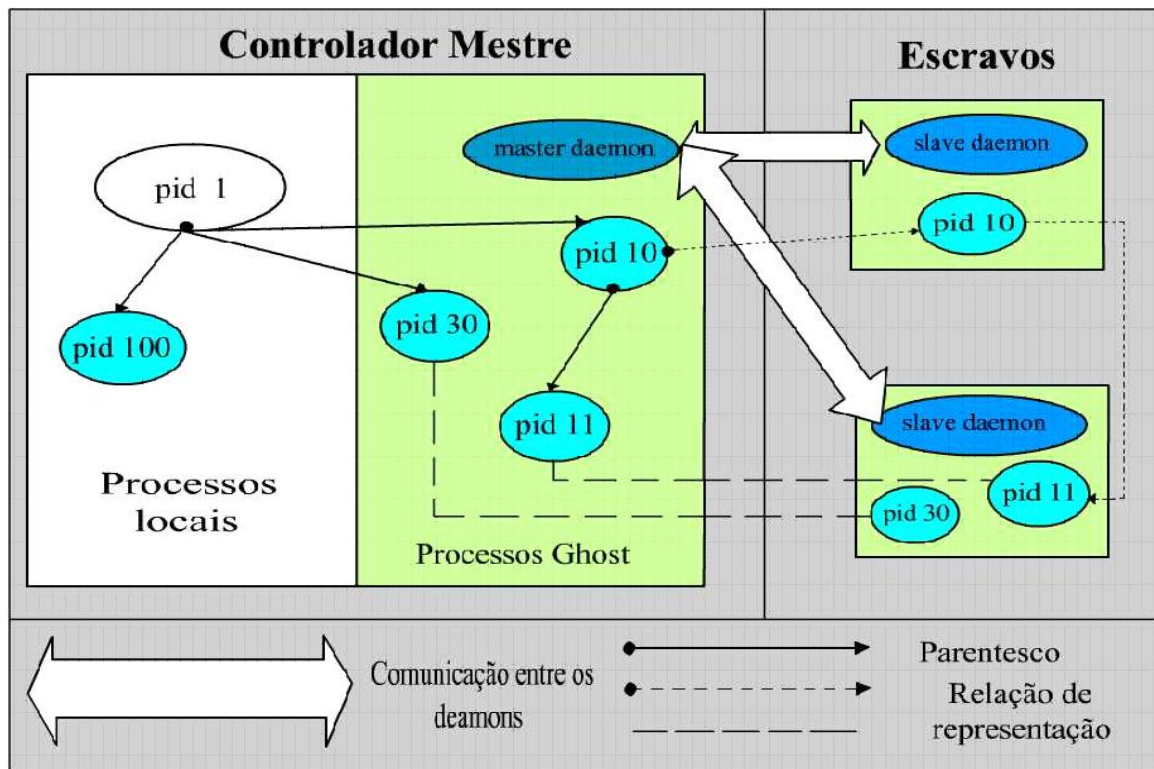


Figura 4.1: Visão Geral - BProc

4.2 PVM

A idéia do Parallel Virtual Machine consiste em montar uma máquina virtual de n processadores e usá-los para executar tarefas simultâneas. O PVM é dividido em três partes principais:

- **console:** usado para montar a máquina paralela virtual.
- **daemon:** um programa que roda em cada máquina do ambiente estabelecendo a comunicação entre as diversas máquinas.
- **biblioteca:** é na biblioteca que reside as funções e sub-rotinas que instruem o código a dividir as tarefas entre os daemons.

A biblioteca dispõe de recursos que possibilitam manipular qualquer coisa do seu ambiente virtual, pode-se manipular o tempo de execução, embora não seja muito bom. Devido ao custo computacional de se adicionar e retirar máquinas em tempo de execução. O

ideal é criar a máquina virtual fora do código, através do console, e usá-la várias vezes, ou mesmo deixá-la ativa enquanto as máquinas estiverem ligadas, além de possibilitar disparar e matar processos a partir do console.

A biblioteca de programação paralela PVM (Parallel Virtual Machine) foi produzido pelo Heterogeneous Network Project, um esforço conjunto da Oak Ridge National Laboratory, University of Tennessee, Emory University e Carnegie Mellon University, em 1989, para facilitar o campo da computação paralela heterogênea. O PVM foi um dos primeiros sistemas de software a possibilitar que programadores utilizassem uma rede de sistemas heterogêneos (máquinas diferentes com sistemas operacionais diversos) ou sistemas MPP (Massively Parallel Processors) para desenvolver aplicações paralelas sobre o conceito de passagem de mensagens. Esta biblioteca ou API de programação tem sido muito difundida em ambientes computacionais científicos e recentemente tem ganho muitos adeptos também no campo de aplicações comerciais devido à sua capacidade de portabilidade, sendo referenciada como o "padrão de facto" em sua área.

Um cluster consiste basicamente em processos trocando mensagens sobre uma rede de comunicação e o PVM possui muitas funções de recebimento e envio de mensagens. O pacote PVM é relativamente pequeno (cerca de 4.5Mb de código fonte em C), roda sobre ambiente Unix, seus derivados e Windows NT, e necessita ser instalado apenas uma vez em cada máquina para ser acessível a todos os usuários. Além disso, a instalação não requer privilégios especiais e pode ser efetuada por qualquer usuário.

O daemon é chamado pvmd3, que reside em todas as máquinas que compõem o cluster, criando o que se refere como uma *máquina virtual paralela*. Quando um usuário deseja usar uma aplicação PVM, ele executa o daemon pvmd3 em um dos seus computadores do cluster, no qual responsabiliza-se por chamar outros processos daemons pvmd3 escravos em cada computador da máquina paralela virtual. Uma aplicação PVM pode então ser iniciada em um prompt Unix em qualquer console do cluster.

A biblioteca de rotinas contém o padrão de comunicação do PVM. Esta biblioteca contém rotinas chamáveis pelo usuário para passagem de mensagens, criação de processos, sincronização de tarefas e modificação da máquina virtual. As aplicações devem ser ligadas com esta biblioteca para poderem usufruir do ambiente paralelo criado pelo PVM.

A biblioteca PVM pode ser baixada no site www.epm.ornl.gov/pvm/ e pode ser utilizada em uma rede com Linux e Windows NT, podendo-se, com isso utilizar um combinado de computadores com Windows NT e Unix.[12][13]

A principal idéia por trás do PVM é utilizar um conjunto de computadores heterogêneos interconectados, como um recurso virtualmente único. Cada computador existente na rede pode ser utilizado como sendo um nó da máquina paralela virtual. O papel de console da máquina paralela é assumido pelo próprio nó local onde o usuário está localizado fisicamente. O usuário pode alocar qualquer nó da rede localmente, ou a longa distância, desde que o mesmo esteja devidamente autorizado. Este ambiente é constituído de uma biblioteca de rotinas em C e FORTRAN, onde o usuário pode escolher qual protocolo de camada de transporte deseja utilizar, TCP ou UDP, no envio e recebimento de mensagens. Em ambos os casos as mensagens são empacotadas (STUB) antes do envio e desempacotadas pelo processo receptor. Estes procedimentos geram uma sobrecarga extra (overhead).

Um daemon mestre cria os diversos daemons escravos na máquina do cluster via remote shell - rsh, os quais devem estar devidamente configurados em todos os host antes do uso do PVM.

Vejamos algumas características importantes do PVM:

Quanto à interoperabilidade: além da portabilidade, os programas podem ser executados em arquiteturas completamente diferentes;

Abstração completa: o PVM permite que a rede seja totalmente heterogênea, administrada como uma única máquina virtual;

Controle de processo: capacidade de iniciar, interromper e controlar processos, em tempo de execução;

Controle de recursos: totalmente abstrato, graças ao conceito de máquina virtual paralela;

Tolerância a falhas: comporta esquemas básicos de notificação de falhas, para alguns casos. Porém, permite flexibilidade, de forma que, ainda em certas situações onde não existe resposta de um nó, uma aplicação receba os resultados das outras.

Há muitas formas de se melhorar a performance com o PVM, mas a verdade é que não há uma "receita" ou método a ser seguido, pois tudo depende da arquitetura do cluster, da velocidade da rede, das configurações dos sistemas e de muitos outros fatores determinantes que devem ser avaliados no momento de se escolher uma metodologia para resolver determinado problema.[14][15]

4.3 MPI

O MPI (Message Passing Interface) é constituído por um padrão de troca de mensagens com sintaxe definida, mas preservando características exclusivas de cada arquitetura, inclusive para arquiteturas de memória compartilhada. O MPI define um conjunto de rotinas para facilitar a comunicação (troca de dados e sincronização) entre processos em memória, ela é portátil para qualquer arquitetura, tem aproximadamente 125 funções para programação e ferramentas para análise de performance. A biblioteca MPI possui rotinas para programas em linguagem C/C++, Fortran 77/90. Os programas são compilados e ligados à biblioteca MPI.

O principal objetivo do MPI é otimizar a comunicação e aumentar o desempenho computacional das máquinas, não possuindo dessa forma gerenciamento dinâmico de processos e processadores.

Embora exista a restrição citada acima, os programas escritos em MPI tendem a serem mais eficientes pelo fato de não haver overhead na carga de processos em tempo de execução.

O padrão para Message Passing, denominado MPI (Message Passing Interface), foi projetado em um forum aberto constituído de pesquisadores, acadêmicos, programadores, usuários e vendedores, representando 40 organizações ao todo.

No MPI Forum foram discutidos e definido a: Sintaxe, Semântica e o Conjunto de rotinas padronizadas para Message Passing.[16][17]

As principais características do MPI são:

- Eficiência - Foi cuidadosamente projetado para executar eficientemente em máquinas diferentes. Especifica somente o funcionamento lógico das operações. Deixa em aberto a implementação. Os desenvolvedores otimizam o código usando características específicas de cada máquina.
- Facilidade - Define uma interface não muito diferente dos padrões PVM, NX, Express, P4, etc. e acrescenta algumas extensões que permitem maior flexibilidade.
- Portabilidade - É compatível para sistemas de memória distribuída, NOWs (network of workstations) e uma combinação deles.

- Transparência - Permite que um programa seja executado em sistemas heterogêneos sem mudanças significativas.
- Segurança - Provê uma interface de comunicação confiável. O usuário não precisa preocupar-se com falhas na comunicação.
- Escalabilidade - O MPI suporta escalabilidade sob diversas formas, por exemplo: uma aplicação pode criar subgrupos de processos que permitem operações de comunicação coletiva para melhorar o alcance dos processos.

A diferença básica entre o MPI e o PVM é que, ao contrário do anterior, no MPI existe um único código fonte igual para todas as máquinas e conseqüentemente um único processo rodando.

O MPI funciona da seguinte forma: cada máquina (node) recebe uma cópia do código fonte e um nome. Cada nó começa a executar o programa a partir da primeira linha de comando utilizando as seguintes diretrizes: Executar todas as linhas de comando não nomeadas; Executar as linhas de comando nomeadas com o mesmo nome do node; Não executar as linhas de comando com o nome de outro node.

Muitas implementações de bibliotecas do padrão MPI têm sido desenvolvidas, sendo algumas proprietárias e outras de código livre, mas todas seguindo o mesmo padrão de funcionamento e uso. O esforço de padronização MPI envolve cerca de 60 pessoas de 40 organizações, principalmente dos Estados Unidos e da Europa. A maioria dos vendedores de computadores paralelos e concorrentes estão envolvidos com o padrão MPI, através de pesquisas em universidades e laboratórios governamentais e industriais. O processo de padronização começou com um seminário sobre o Centro de Pesquisas em Computação Paralela, realizada em 29 e 30 de abril de 1992 em Williamsburg, Virginia. Nesse seminário, as características essenciais para uma padronização de uma interface de passagem de mensagem foram discutidas, sendo estabelecido também um grupo de trabalho para continuar o processo de padronização.

As funcionalidades que MPI designa são baseadas em práticas comum de paralelismo e outras bibliotecas de passagem de mensagens similares, como Express, NX/2, Vertex, Parmacs e P4. A filosofia geral do padrão MPI é implementar e testar características novas que venham a surgir no mundo da computação paralela. Assim que uma determinada gama de características novas tenham sido testadas e aprovadas, o grupo MPI reúne-se novamente para considerar sua inclusão na especificação do padrão. Muitas das

características do MPI têm sido investigadas e usadas por grupos de pesquisas durante muitos anos, mas não em ambiente de produção ou comerciais. Sendo assim, existe um grande espaço de tempo entre novidades do padrão e sua implementação por vendedores e colaboradores. Entretanto, a incorporação destas características dentro do padrão MPI é justificada pela expressivas inovações que elas trazem.

O MPI implementa um excelente mecanismo de portabilidade e independência de plataforma computacional. Por exemplo, um código MPI, que foi escrito para uma arquitetura IBM RS-600 utilizando o sistema operacional AIX, pode ser portado para a arquitetura SPARC com o S.O.Solaris ou PC com Linux com quase nenhuma modificação no código fonte da aplicação.

O MPI-1.2 é uma extensão para o padrão MPI-1 de 1992. Há uma terceira especificação chamada MPI-2, que adiciona várias extensões à versão 1.2, sendo uma das mais importantes o controle de processos dinâmicos. Devido a dificuldade de se encontrar uma implementação MPI-2 que seja distribuída livremente, este trabalho restringe-se ao padrão ao padrão MPI-1.2. Portanto, qualquer menção do padrão MPI, refere-se na verdade, ao padrão MPI-1.2.

A execução de uma aplicação executando esta API (Interface de Programação de Aplicações) inicia um procedimento de disparar através de um processo "pai" seus processos "filhos" para serem executados remotamente nos computadores escravos do cluster.

Cada processo executa e comunica-se com outras instâncias do programa, possibilitando a execução no mesmo processador e diferentes processadores. A melhor performance quando esses processos são distribuídos entre diversos processadores. A comunicação básica consiste em enviar e receber dados de um processador para o outro. Esta comunicação se dá através de uma rede de alta velocidade, no qual os processos estarão em um sistema de memória distribuída.

O pacote de dados enviados pelo MPI requer vários pedaços de informações: o processo transmissor, o processo receptor, o endereço inicial de memória onde os ítems de dados deverão ser mandados, a mensagem de identificação e o grupo de processos que podem receber a mensagem, ou seja, tudo isto ficará sob responsabilidade do programador em identificar o paralelismo e implementar um algoritmo utilizando construções com o MPI.[18]

Algumas implementações do MPI:

- MPI-F: IBM Research

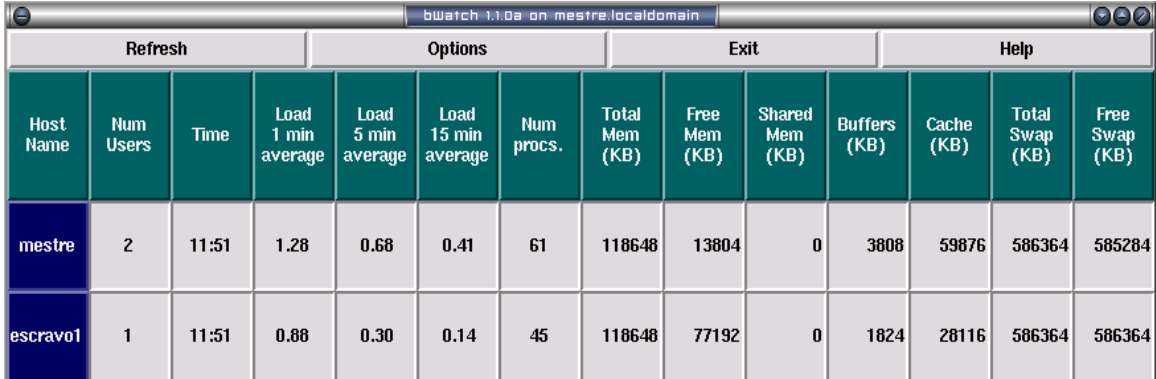
-
- MPICH: ANL/MSU - Argonne National Laboratory e Missipi State University
 - UNIFY: Missipi State University
 - CHIMP: Edinburg Parallel Computing Center
 - LAM: Ohio Supercomputer Center
 - MPL: IBM Research

5 Administração do Cluster

5.1 bWatch - Beowulf Watch

O bWatch é um script escrito na linguagem interpretada gráfica Tcl/Tk designado para monitorar clusters. Sua tarefa é observar a carga e o uso da memória em todos os nós do supercomputador.[19]

Este programa assume que na máquina na qual ele esteja rodando possa executar o remote shell (rsh) em todos os computadores listados na variável \$listOfHosts. Ele também assume que seu interpretador wish esteja em /usr/bin/wish. Se você não estiver com o Tcl/Tk instalado em seu computador, então faça-o.



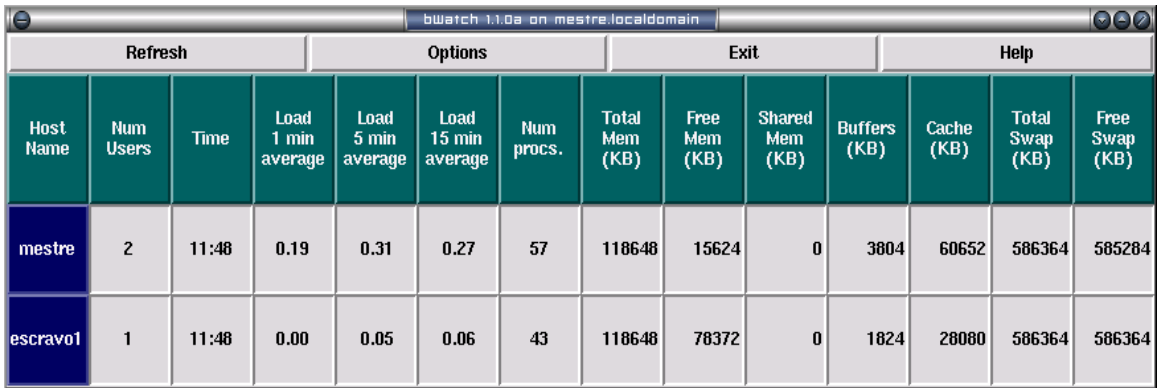
The screenshot shows a window titled "bWatch 1.1.0a on mestre.localdomain". It contains a table with columns for Host Name, Num Users, Time, Load averages (1, 5, 15 min), Num procs., Total Mem (KB), Free Mem (KB), Shared Mem (KB), Buffers (KB), Cache (KB), Total Swap (KB), and Free Swap (KB). The table lists two nodes: "mestre" and "escravo1".

Host Name	Num Users	Time	Load 1 min average	Load 5 min average	Load 15 min average	Num procs.	Total Mem (KB)	Free Mem (KB)	Shared Mem (KB)	Buffers (KB)	Cache (KB)	Total Swap (KB)	Free Swap (KB)
mestre	2	11:51	1.28	0.68	0.41	61	118648	13804	0	3808	59876	586364	585284
escravo1	1	11:51	0.88	0.30	0.14	45	118648	77192	0	1824	28116	586364	586364

Figura 5.1: Bwatch com carga

Esta ferramenta não necessita que você esteja "logado" como root para funcionar, e pode ser executado por qualquer usuário cadastrado no sistema que possua a permissão de executar comandos remotos via protocolo rsh (remote shell) para as outras máquinas.

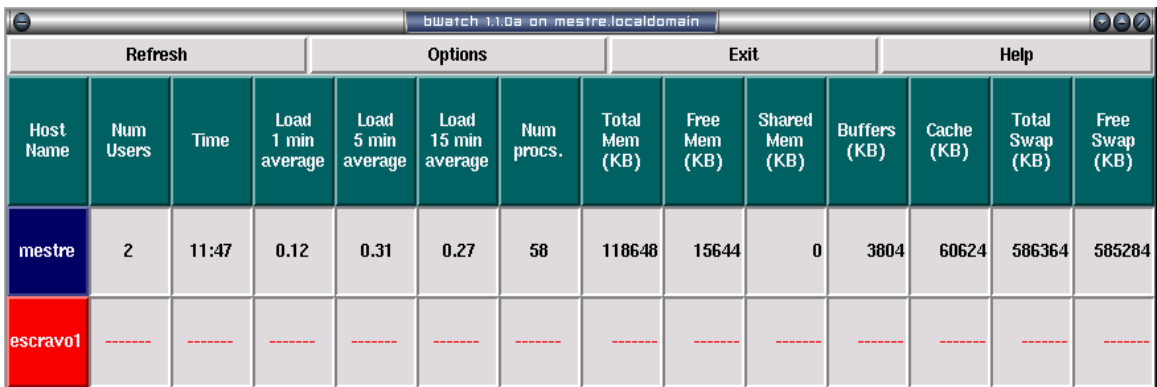
O BWatch foi escrito especificamente para a plataforma Linux e poderá não ser executado em qualquer outro sistema Unix a menos que seu sistema de arquivo possua o diretório /proc; exatamente igual ao Linux. Esta ferramenta confia fielmente no sistema de arquivo /proc; assim, deve-se compilar este suporte dentro do kernel de todas



Refresh			Options				Exit			Help			
Host Name	Num Users	Time	Load 1 min average	Load 5 min average	Load 15 min average	Num procs.	Total Mem (KB)	Free Mem (KB)	Shared Mem (KB)	Buffers (KB)	Cache (KB)	Total Swap (KB)	Free Swap (KB)
mestre	2	11:48	0.19	0.31	0.27	57	118648	15624	0	3804	60652	586364	585284
escravo1	1	11:48	0.00	0.05	0.06	43	118648	78372	0	1824	28080	586364	586364

Figura 5.2: Bwatch sem carga

as máquinas listadas no arquivo listofhosts.



Refresh			Options				Exit			Help			
Host Name	Num Users	Time	Load 1 min average	Load 5 min average	Load 15 min average	Num procs.	Total Mem (KB)	Free Mem (KB)	Shared Mem (KB)	Buffers (KB)	Cache (KB)	Total Swap (KB)	Free Swap (KB)
mestre	2	11:47	0.12	0.31	0.27	58	118648	15644	0	3804	60624	586364	585284
escravo1	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Figura 5.3: Bwatch sem escravo1

6 O Cluster OpenMosix

6.1 Cluster OpenMosix

O projeto Mosix - Multicomputer Operating System unIX - é um sistema operacional distribuído, desenvolvido originalmente pelos estudantes do professor Amnom Barak, na Universidade Hebrew em Jerusalém, Israel. Foi utilizado nos anos 80 pela força área americana para a construção de um cluster de computadores PDP11/45. O projeto foi desenvolvido em sete fases, para diferentes versões de UNIX e arquiteturas de computadores. A primeira versão para PC foi desenvolvida para o BSD/OS. A última versão foi para o sistema operacional Linux em plataforma Intel.

O OpenMosix é uma extensão do projeto Mosix, baseado no GPLv2, iniciado em 10 de fevereiro de 2002, coordenado pelo Ph.D Moshe Bar, para manter os privilégios desta solução Linux para cluster disponível com software de código aberto.

Este agrupamento de máquinas Linux é o que poderíamos classificar de verdadeiro sistema de imagem simples (SSI - Single System Image), pois já é claro que a idéia de que não se tem um cluster verdadeiro enquanto não existir um SSI. Podemos ter como referência os primeiros clusters SSI como o IBM SysPlex e o cluster DEC. Em um cluster DEC você executa um telnet para um endereço no cluster e essa chamada será atendida por qualquer nó do cluster, e o usuário não precisa se preocupar com qual nó que irá atender esta chamada, e qualquer programa iniciado pelo usuário será executado no nó que possuir maior disponibilidade de recursos para atender ao programa.

O OpenMosix é uma extensão do núcleo do Linux, que faz com que um cluster de computadores se comporte como um grande e único supercomputador através da utilização de migração preemptiva de processos e balanceamento dinâmico de carga.

A implementação da Migração Preemptiva de processos é capaz de migrar qualquer processo do usuário, em qualquer instante e para qualquer nó disponível de maneira transparente. Para atingir um melhor desempenho este é controlado por Algoritmos de Balanceamento Dinâmico de Carga e de prevenção contra falta de memória. Estes algoritmos são projetados para responder dinamicamente as variações da utilização dos recursos nos diversos nós. Isto garante que o cluster se comporte muito bem, seja numa configuração

com poucas ou com muitas máquinas, propiciando uma maior escalabilidade. Ou seja, se o programa que estamos rodando em uma máquina consumir muitos recursos, o sistema varre toda a rede e procura uma máquina mais "disponível no momento" em termos de memória e CPU, e desloca seu "programa" ou parte dele para ser executado remotamente. Com isso, o sistema ganha desempenho.

Estes algoritmos são descentralizados, ou seja, não existe a configuração de Controlador Mestre e nós escravos como ocorre no Cluster Beowulf para computação paralela. Cada nó é um mestre para os processos que são criados localmente, e um servidor para processos remotos, migrados de outros nós do cluster. Isto significa que podemos acrescentar ou remover as máquinas do cluster em qualquer momento, com um mínimo de distúrbio no sistema. Este cluster possui também algoritmos de monitoramento que identificam a velocidade de cada nó, a carga da CPU, a memória livre disponível, a comunicação interprocessos IPC e a velocidade de acesso de cada processo.

Como o OpenMosix opera de forma silenciosa e as operações são transparentes para as aplicações, ou seja, pode-se executar aplicações sequenciais e paralelas como se fosse um único computador SMP (Symmetric Multi-Processor - Multiprocessamento simétrico). Você não precisa conhecer onde seus processos estão sendo executados, nem se preocupar com que os outros usuários estão fazendo na rede por isso ele usa o acrônimo "fork and forget". O que ele faz é, pouco tempo depois de iniciar os processos, o OpenMosix envia-os para um melhor computador da rede, o OpenMosix continua a monitorar os novos processos e os demais, e poderá movimentá-los pelos computadores com pouca carga de trabalho maximizando o trabalho e melhorando a performance.

Aplicações que se beneficiam com o OpenMosix:

- Processos CPU-bound: processos com longos tempos de execução e baixo volume de comunicação entre processos, ex: aplicações científicas, engenharia e outras aplicações que demandam alta performance de computação.
- Grandes compilações.
- Para processos que misturam longos e rápidos tempos de execução ou com quantias moderadas de comunicação interprocessos, é recomendado o uso das bibliotecas MPI/PVM amplamente utilizadas em computação paralela.
- Processos I/O bound misturados com processos da CPU: executados através do servidor de arquivos, usando o sistema de arquivos distribuídos do OpenMosix, o

MFS (Mosix File System) e o DFSA (Distributed File System Architecture).

- Banco de dados que não usem memória compartilhada.
- Processos que podem ser migrados manualmente.

As desvantagens do OpenMosix:

- Processos com baixa computação, como aplicativos com alta comunicação interprocessos.
- Aplicações com memória compartilhada.
- Aplicações dependentes do hardware que necessitam de acesso a um periférico de um nó em especial. Aplicações com muitas threads não ganham desempenho.
- Não se ganha desempenho quando se roda um único processo, tal como seu browser.

Aplicações testadas que não migraram sobre OpenMosix:

- Programas em Java usando threads nativas não migram desde que eles utilizem memória compartilhada. Green Threads JVMs, entretanto, podem ser migradas porque cada thread Java é um processo separado.
- Aplicações que usam pthreads.
- MySQL, Apache, Oracle, Postgres, SAP, Baan, usam memória compartilhada.
- Python com threading habilitada.
- VMware, este ao rodar o Win98, algumas vezes travou e em outras o emulador do SO parou. Você deverá ter muito cuidado quando usar o VMware com o OpenMosix.
- A característica de não migrar é uma situação normal para programas que falhariam ao serem movimentados pelo OpenMosix. Estes programas devem rodar como planejado no nó onde foram iniciados.

6.2 OpenMOSIXVIEW

O OpenMosixview é um gerenciador gráfico (GUI - Graphic User Interface), ele é um front-end para os comandos "mosctl". Com esta aplicação é possível gerenciar e ajustar os principais parâmetros deste cluster tais como: visualizar a carga de todos os nós do cluster, visualização de processos remotos em outros nós, bem como executar ou transferir processos para outros computadores que compõem o cluster.

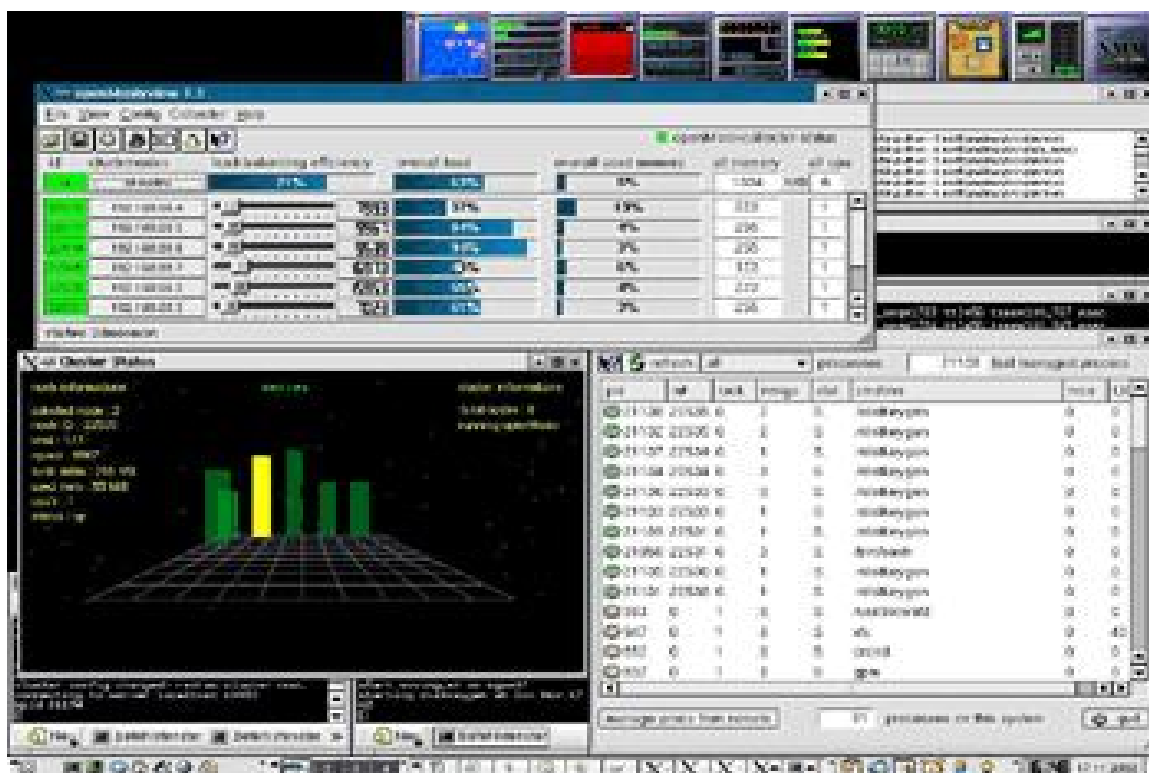


Figura 6.1: OpenMosixView

O OpenMosixview é um conjunto de cinco ferramentas utilizadas para administração e monitoramento do Cluster OpenMosix. São eles:

- OpenMosixview: a aplicação principal de administração/monitoramento.
- OpenMosixprocs: um box para gerenciamento de processos.
- OpenMosixcollector: coleta informações dos nós do cluster através de um processo daemons para geração de logs do sistema.
- OpenMosixanalyzer: utilizado para a análise dos dados coletados pelo OpenMosix-collector.

-
- OpenMosixhistory: histórico dos processos no nosso cluster.

7 Implementação

INSTALANDO O CONECTIVA 8

Siga os passos normais.[20][21][22]

Em "PERFIL DE INSTALAÇÃO" escolha "Instalação REALMENTE Mínima", e marque a opção "Forçar seleção de pacotes".

Em "SELEÇÃO DE COMPONENTES", marque:

- + Minimal
- + Development Workstation
- + Linuxconf
- + C Development
- + C++ Development
- + Linuxconf modules
- + Sistema X Window
- + Window Maker
- + Aplicações para Console
- + Extra Development Files
- + 'Selecionar pacotes individuais'

Em "SELEÇÃO DE PACOTES", acrescente:

Administração:

- + apt-listchanges

Bibliotecas:

- + glibc
- + gmp2

```
+ libpcap
+ libpcap-devel
+ mcrypt
```

Rede:

```
+ iptables
+ iptraf
+ libnet
+ nfs-server
+ nmap
+ rsh
+ tcpdump
```

Utilitários:

```
+ unzip
+ zip
```

X11:

```
+ xterm
```

Configure as interfaces de rede: IP, máscara, nome, gateway e DNS.

Configure o arquivo `/etc/hosts` com os IPs dos nós do cluster:

127.0.0.1	mestre	mestre
127.0.0.2	escravo1	escravo1
127.0.0.3	escravo2	escravo2
127.0.0.4	escravo3	escravo3

127.0.0.5	escravo4	escravo4
127.0.0.6	escravo5	escravo5
127.0.0.7	escravo6	escravo6
127.0.0.8	escravo7	escravo7

Para que o protocolo RSH (Remote Shell) funcione corretamente, configure os arquivos `/etc/host.equiv` e `/root/.rhosts`, informando os nomes dos nós do cluster:

```
mestre
escravo1
escravo2
escravo3
escravo4
escravo5
escravo6
escravo7
```

Acrescente em `/etc/securetty`:

```
rsh
rlogin
```

Selecione o GRUB como gerenciador de boot.

Para poder gerenciar os nós de forma segura, veja o manual do SSH (Secure Shell) e configure o servidor SSH, editando o arquivo `/etc/ssh/sshd_config`.

Execute o `ntsysv` para configurar os serviços que devem ser inicializados automaticamente:

```
+ atd
+ crond
+ gpm (se quiser usar mouse no console)
+ keytable
```

```
+ netfs
+ network
+ nfs
+ nfslock
+ portmap
+ random
+ rawdevices
+ rlogin
+ rsh
+ sshd
+ syslog
+ xinetd
```

Reinicie a máquina.

Neste ponto o Conectiva 8 estará pronto para funcionar.

PREPARANDO PARA COMPILAR O KERNEL

Baixe a fonte do kernel 2.4.19, disponível em www.kernel.org

Descompacte e crie um link simbólico:

```
/usr/src# tar zxvf /caminho/para/linux-2.4.16.tar.gz
/usr/src# ln -s linux-2.4.16.tar.gz linux
```

Baixe o iptables, disponível em www.iptables.org, assim como o patch-o-magic (disponível no mesmo site). Descompacte o iptables e o patch-o-magic. Execute o patch-o-magic e siga as instruções.

Baixe e descompacte o fonte do BProc. Depois aplique o patch[23] no kernel:

```
/usr/src/linux# patch -p1 </caminho/para/BProc/BProc-patch-2.4.19"
```


COMPILANDO O KERNEL COM BProc

```
/usr/src/linux# cp /caminho/para/BProc/kernel-2.4.19-i386.config .config
/usr/src/linux# make menuconfig
```

No menu de configuração do kernel:

- Selecione o modelo do processador.
- Ajuste os drivers de rede e demais necessários (na dúvida, marque todos como MÓDULO)

Desative Network Device Support -> PCMCIA (*)

Desative ISDN (*)

Desative File Systems - Reiserfs (*)

(*) Se estes módulos forem necessários para seu equipamento, consulte as instruções dos fabricantes para saber quais as ferramentas necessárias para compilar o kernel com estes recursos.

Saia do menu e salve a configuração.

```
/usr/src/linux# make dep
/usr/src/linux# make clean
/usr/src/linux# make bzImage
/usr/src/linux# make modules
/usr/src/linux# make modules_install
/usr/src/linux# mkinitrd /boot/initrd-2.4.19.img 2.4.19
/usr/src/linux# cp -aRdiv arch/i386/boot/bzImage /boot/vmlinuz-2.4.19
/usr/src/linux# cp System.map /boot/System.map-2.4.19
/usr/src/linux# cp include/linux/kernel.h /boot/kernel.h-2.4.19
/usr/src/linux# cd /boot
/boot# rm System.map
/boot# rm kernel.h
/boot# ln -s System.map-2.4.19 System.map
/boot# ln -s kernel.h-2.4.19 kernel.h
```

Acrescente uma entrada em `/boot/grub/menu.lst` para a nova imagem do kernel que foi gerada:

```
title = Beowulf
    kernel = (hd1,1)/boot/vmlinuz-2.4.19 root=/dev/hdb2 3
    initrd = (hd1,1)/boot/initrd-2.4.19.img
```

Reinicie com o novo kernel.

INSTALANDO O BProc

Vá para o diretório do fonte do BProc e execute:

```
# make
# make install
# cp clients/libBProc.so.2 /usr/lib/.
# cd/boot
/boot# rm -f vmlinuz
/boot# ln -s vmlinuz-2.4.19 vmlinuz
```

Reinicie com o novo kernel.

Neste ponto sua máquina está pronta para atuar em cluster beowulf.

CONFIGURANDO O BProc###

Edite o arquivo de configuração do mestre: `/etc/beowulf/config`

As diretivas de configuração estão na documentação do BProc.

```
# Interface de escuta do MESTRE
interface lo 127.0.0.1 255.0.0.0
# Numero de nos
nodes 9
# Faixa de IPs para os nos
```

```
iprange 1 127.0.0.1 127.0.0.8
# Bibliotecas que serao compartilhadas entre os nos
libraries /lib/ld-*
libraries /lib/libc-*
libraries /lib/libBProc*
libraries /lib/libtermcap*
libraries /usr/lib/libBProc*
```

TESTANDO A INSTALAÇÃO DO BProc

Execute tanto no mestre como nos escravos:

```
# modprobe BProc
```

No mestre, execute:

```
# bpmaster
```

Nos escravos, execute:

```
# bpslave <MESTRE> 2223
```

Onde MESTRE representa o IP ou o NOME do mestre na rede.

De volta ao mestre, execute:

```
# bpstat
```

e verifique se todos os escravos estão listados como "UP".

AUTOMATIZANDO A INICIALIZAÇÃO

No MESTRE, crie o arquivo /etc/init.d/BProc :

```
!/bin/sh
/sbin/modprobe BProc
/usr/sbin/bpmaster
```

Nos ESCRAVOS, crie o arquivo `/etc/init.d/BProc` :

```
!/bin/sh
/sbin/modprobe BProc
/usr/sbin/bpslave MESTRE 2223
```

Onde 'MESTRE' representa o IP ou o NOME do mestre na rede.

Em TODOS os nós do cluster, inclusive MESTRE:

```
# cd /etc/rcN.d/
```

(N pode ser 2,3,4 ou 5. No Conectiva, quando iniciado no console, use 3)

```
/etc/rc3.d# ln -s ../init.d/BProc S98BProc
```

Quando as máquinas forem reiniciadas, entrarão automaticamente no cluster. Lembre-se apenas de iniciar o MESTRE antes dos ESCRAVOS.

INSTALANDO O PVM

Faça o download da versão 3.4.4, disponível em www.csm.ornl.gov/pvm e descompacte no diretório `/usr/share/pvm3`

Edite o arquivo `/etc/bashrc` e acrescente estas linhas:

```
export PVM_ROOT=/usr/share/pvm3
export PVM_ARCH=LINUX
export PVM_TMP=/tmp/pvm
```

```
export PATH=$PATH:$PVM_ROOT/lib:$PVM_ROOT/lib/$PVM_ARCH:$PVM_ROOT/bin/$PVM_ARCH
export PVM_DPATH=$PVM_ROOT/lib/pvmd
```

Compile o PVM:

```
cd /usr/share/pvm3
/usr/share/pvm3# make
```

Inicie o PVM no MESTRE:

```
# pvm
pvm> add mestre
```

Acrescente os escravos:

```
pvm> add escravo1
pvm> add escravo2
pvm> add escravo3
pvm> add escravo4
pvm> add escravo5
pvm> add escravo6
pvm> add escravo7
```

Verifique se todos estão no cluster:

```
pvm> conf
```

8 Demonstração do Cluster Beowulf

As estações utilizadas nesse projeto foram:

- 3 máquinas Pentium 4, 1.8 GHz; (2 máquinas para o cluster e outra apenas para monitorar)
- cada uma com 40GB de HD;
- cada uma com 128M de RAM;
- Teclados 102 teclas (ABNT2);
- Mouses e monitores padrão IBM;

8.1 Demonstração do BProc

Para demonstrar o funcionamento da biblioteca BProc, utilizamos dois programas: Stress1 e Stress2. Ambos realizam um número arbitrário de operações de multiplicação com números em ponto flutuante de 80 bits. As operações, não correlatas, foram divididas em subprocessos também de forma arbitrária. Stress1 utiliza o método `fork()` para criar os subprocessos. Esperávamos que o kernel com BProc fosse distribuir esses subprocessos no cluster, mas isso não aconteceu. Todos os subprocessos foram executados na máquina que os invocou. No programa Stress2 substituímos o método `fork()` por `bproc_rfork()`, e com isso os subprocessos foram distribuídos no cluster. Mas a distribuição não ocorreu de forma automática. Tivemos que utilizar outros métodos do BProc para obter uma lista dos nós ativos, e a cada chamada de `bproc_rfork()` tivemos que especificar em qual nó o subprocesso deveria ser executado.

Como utilizamos duas máquinas com o mesmo poder de processamento, distribuímos de forma uniforme os processos e obtivemos ganhos da ordem de 100%. Durante a execução do Stress2 distribuído, capturamos os processos em execução em cada máquina, e pudemos ver a distribuição. Para o teste executamos 250 milhões de operações, divididas em dez subprocessos de 25 milhões de operações. O resultado da execução é o seguinte:

Início: Wed Jul 21 08:33:44 2004

Filho 1 -> pid 1162 -> no 0

Filho 2 -> pid 1164 -> no 1

Filho 3 -> pid 1165 -> no 0

Filho 4-> pid 1167 -> no 1

Filho 5 -> pid 1168 -> no 0

Filho 6 -> pid 1170 -> no 1

Filho 7 -> pid 1171 -> no 0

Filho 8 -> pid 1173 -> no 1

Filho 9 -> pid 1174 -> no 0

Filho 10 -> pid 1176 -> no 1

Fim do pid 1164 (status = 0)

Fim do pid 1167 (status = 0)

Fim do pid 1170 (status = 0)

Fim do pid 1162 (status = 0)

Fim do pid 1173 (status = 0)

Fim do pid 1176 (status = 0)

Fim do pid 1165 (status = 0)

Fim do pid 1168 (status = 0)

Fim do pid 1171 (status = 0)

Fim do pid 1174 (status = 0)

Fim: Wed Jul 21 08:35:19 2004

Tempo gasto (s): 95 Numero total de operações: 250000000

Processos na máquina **mestre** durante a execução:

```

895  ?      S      0:00  /usr/sbin/bpslave -s 192.168.200.1 mestre 2223
899  ?S 0:00 /usr/sbin/bpslave -s 192.168.200.1 mestre 2223"
1163 ?      R      0:05  ./stress2 5000 10
1166 ?      R      0:04  ./stress2 5000 10
1169 ?      R      0:04  ./stress2 5000 10
1172 ?      R      0:04  ./stress2 5000 10

```

```

1175 ?      R      0:04      ./stress2 5000 10

908  ?      S      0:00      login -- root
914  tty1    S      0:00      -bash
1161 tty1    S      0:00      ./stress2 5000 10
1162 tty1    RW     0:05      [stress2]
1164 tty1    RW     0:05      [stress2]
1165 tty1    RW     0:04      [stress2]
1167 tty1    RW     0:04      [stress2]
1168 tty1    RW     0:04      [stress2]
1170 tty1    RW     0:04      [stress2]
1171 tty1    RW     0:04      [stress2]
1173 tty1    RW     0:04      [stress2]
1174 tty1    RW     0:04      [stress2]
1176 tty1    RW     0:04      [stress2]

```

Processos na máquina **escravo1**:

```

925  ?      S      0:00      /usr/sbin/bpslave mestre 2223
929  ?      S      0:00      /usr/sbin/bpslave mestre 2223
1045 ?      R      0:06      ./stress2 5000 10
1046 ?      R      0:06      ./stress2 5000 10
1047 ?      R      0:06      ./stress2 5000 10
1048 ?      R      0:06      ./stress2 5000 10
1049 ?      R      0:05      ./stress2 5000 10

```

Podemos ver que para cada instância do daemon escravo (**bpslave**) tivemos cinco subprocessos em execução. Observe que a máquina **mestre** também executou o daemon **bpslave** para que fizesse parte da distribuição de tarefas.

Também podemos ver nesse exemplo o conceito de mascaramento de PID. Observe que cada processo filho obteve um número PID que o identifica na máquina **mestre**. Cada um desses PIDs está representado de uma forma especial pelo sistema operacional,

Resultados		
	1 Máquina	2 Máquinas
Stress2	188 s	94 s

Tabela 8.1: Stress2

por se tratar de processos fantasmas. Os processos fantasmas não são executados, eles servem para transmitir sinais entre processos de forma transparente, pois não é necessário saber em qual nó um processo está realmente em execução para que se possa enviar um sinal ao mesmo.

A desvantagem da utilização da biblioteca BProc é que sua utilização implica no desenvolvimento de métodos de controle da distribuição de tarefas, assim como para troca de mensagens, pois essas tarefas não fazem parte do BProc.

8.2 Aplicações

8.2.1 PovRay

O PovRay é uma ferramenta de alta qualidade, totalmente livre para criar gráficos tridimensionais. Está disponível em versões oficiais para OS X e Windows, do mac OS/Mac e i86 Linux. O Persistence of Ray-Tracer da visão cria imagens 3D, foto-realísticas usando uma técnica de renderização chamada ray-tracing. Lê uma informação dentro do arquivo fonte que descreve os objetos, ilumina uma cena e gera uma imagem dessa cena do ponto da vista de uma câmera descrita também no código. O Ray-Tracer não é um processo rápido, mas produz imagens da qualidade muito elevada com reflexões realísticas [24].

Para instalar o POVray, foram executados os seguintes passos:

```
# cd /usr/local
```

```
# mkdir povray31 # cd povray31 # cp /lugar/pacotes/de/instalação/povuni* .
```

Depois foram aplicados patches ao programa para rodarem sobre PVM e MPI. Foram copiados todos os patches para o diretório /usr/local/povray31 .

Como são dois patches específicos, foram criados dois diretórios dentro de /usr/local/povray31 e copiados todos os dados descompactados.

Para o patch do MPI foram feitos os seguintes passos:

Dentro do /usr/local/povray31:

```
# mkdir mpipov
# cd mpipov
# tar zfvx ../povuni_s.tgz
# tar zfvx ../povuni_d.tgz
# cp /lugar/de/origem/do/patch/mpi-povray-1.0.patch.gz .
# gunzip mpi-povray-1.0.patch | patch -p1
# cd source/mpi-unix
# make newxwin
```

Para o patch do PVM foram feitos os seguintes passos:

```
$ cd /usr/local/povray31
$ cp /lugar/de/origem/do/patch/pvmpov-3.1g2.tgz .
$ tar zfvx pvmpov-3.1g2.tgz
Será criado o diretório pvmpov3_1g_2
$ cd pvmpov3_1g_2
$ tar zfvx ../povuni_s.tgz
$ tar zfvx ../povuni_d.tgz
$ ./inst-pvm
$ cd povray31/source/pvm
$ aimk newunix
$ aimk newsvga
$ aimk newxwin
$ make install
```

8.2.2 Compilador Distribuído - Pvmmake

PVMmake é um compilador que usa a biblioteca de passagem PVM para executar duas operações básicas:

- *Transferência de Arquivo* - O PVMmake envia dados de texto (formato ASCII), possibilitando a comunicação entre dois hosts;
- *Execução de Comando* - PVMmake pode emitir um comando arbitrário para um host arbitrário. Assim, o PVMmake pode executar comandos do UNIX e shell scripts

bem como programas compilados. Ademais, o output destes comandos, scripts ou programas podem ser vistos nos Nós onde o PVMmake foi lançado ou iniciado.

Essas duas operações básicas fazem com que o PVMmake se torne bastante flexível, poderoso, e de fácil uso.

8.3 PVM

Para demonstrar o funcionamento da biblioteca PVM, utilizamos o **pvmmake** para realizarmos a compilação do kernel do linux. Processos na máquina **mestre** durante a execução:

```

9013 tty5    S    0:03 /usr/share/pvm3/lib/LINUX/pvmd3
14279 tty5    S    0:00  /usr/share/pvm3/bin/LINUX/make_pvm 525749
14280 tty5    S    0:00    make -C drivers fastdep
14281 tty5    S    0:00  /usr/share/pvm3/bin/LINUX/make_pvm 263622
14282 tty5    S    0:00    make _sfdep_acpi _sfdep_atm _sfdep_block _sfdep
14286 tty5    S    0:00  /usr/share/pvm3/bin/LINUX/make_pvm 525753
14287 tty5    S    0:00    make _sfdep_dispatcher _sfdep_events _sfdep_exe
14354 tty5    S    0:00  /usr/share/pvm3/bin/LINUX/make_pvm 525789
14355 tty5    S    0:00  /bin/sh -c /usr/src/linux-teste/scripts/mkdep -
14356 tty5    R    0:00    /usr/src/linux-teste/scripts/mkdep -D__KERNEL

```

Processos na máquina **escravo1**:

```

4377 ?      S    0:00 /usr/share/pvm3/lib/LINUX/pvmd3 -s -d0x0 -nescravo1
4454 ?      S    0:00  /usr/share/pvm3/bin/LINUX/make_pvm 0
4455 ?      S    0:00    /root/mestre/linux-2.4.19/./pvmgmake/make _sfd
4468 ?      S    0:00  /usr/share/pvm3/bin/LINUX/make_pvm 524324
4469 ?      S    0:00    /root/mestre/linux-2.4.19/./pvmgmake/make -C d
4724 ?      S    0:00  /usr/share/pvm3/bin/LINUX/make_pvm 262248
4725 ?      S    0:00    /root/mestre/linux-2.4.19/./pvmgmake/make _sfd
4848 ?      S    0:00  /usr/share/pvm3/bin/LINUX/make_pvm 262199

```

```

4849 ?      S    0:00      /root/mestre/linux-2.4.19/./pvmgmake/make -C q
4853 ?      S    0:00      /usr/share/pvm3/bin/LINUX/make_pvm 262186
4854 ?      R    0:00      /root/mestre/linux-2.4.19/./pvmgmake/make -C i

```

Podemos observar um funcionamento análogo ao funcionamento do **BProc**: cada nó executa um daemon que recebe as tarefas, sem a utilização de remote **shell**.

O grande diferencial da biblioteca PVM é que a distribuição de tarefas fica a cargo do daemon do **nó mestre**, e não do programa em execução. Isso permite ao programador uma abstração da arquitetura do cluster, o que torna esse sistema bastante flexível.

Na figura adiante podemos ver a representação da distribuição de tarefas dessa compilação, obtida através da ferramenta XPVM.

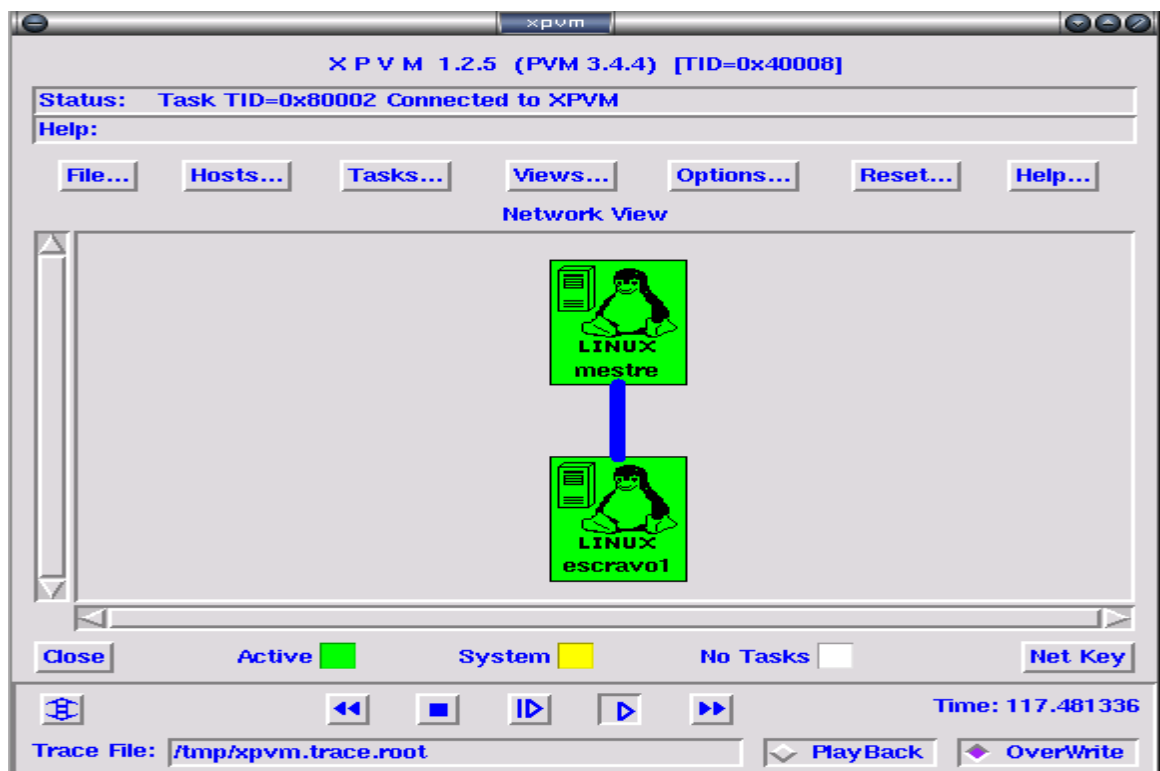


Figura 8.1: XPVM - Nós do Cluster

Na tabela seguinte temos os tempos totais de execução dessa compilação. Observe que o ganho foi de apenas 28,67%. O ganho não foi maior porque a compilação do kernel não foi adequada ao processamento paralelo. Isso significa que em muitas vezes o processo em um dos nós era paralisado até que o outro nó terminasse parte da compilação.

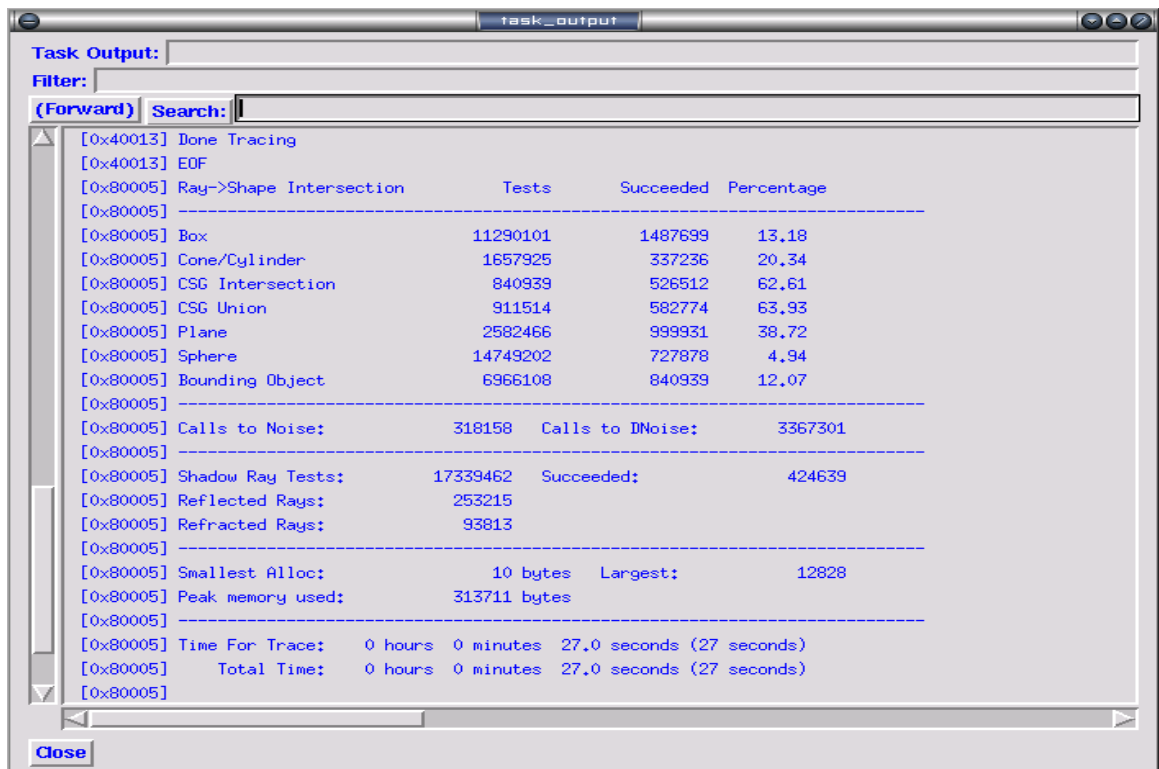


Figura 8.2: XPVM - Saída

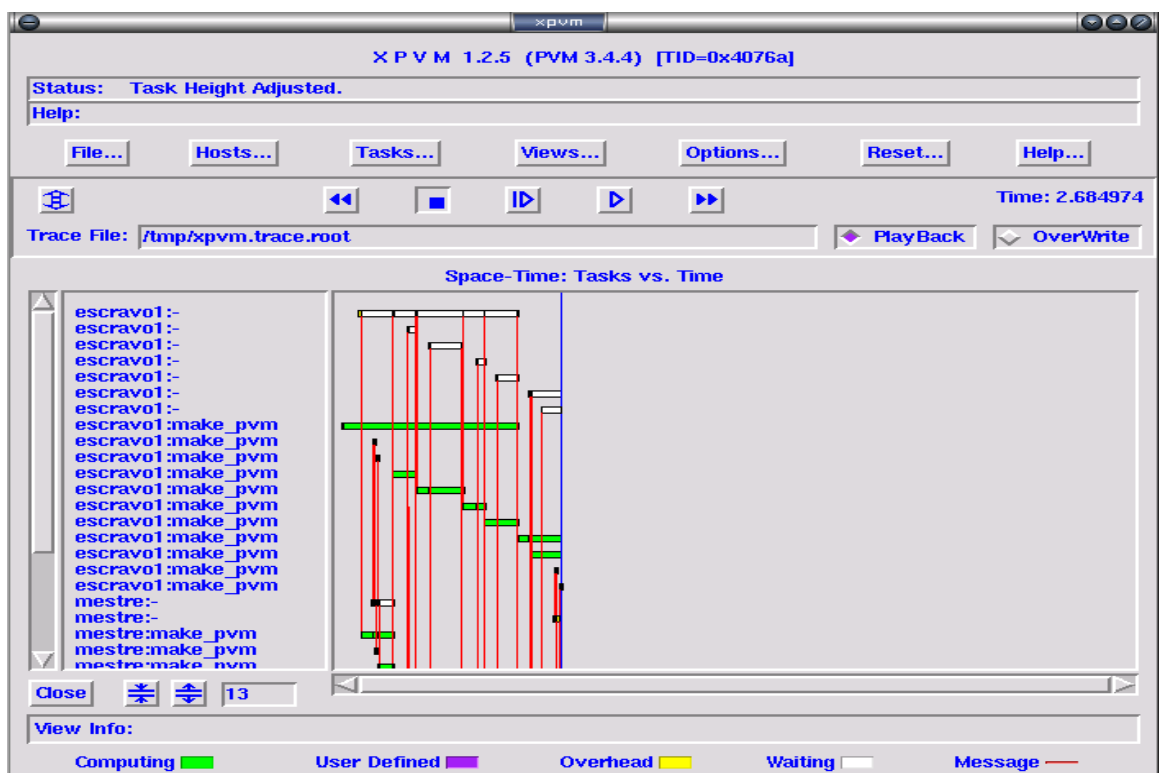


Figura 8.3: Divisão dos Processos - pvmmake

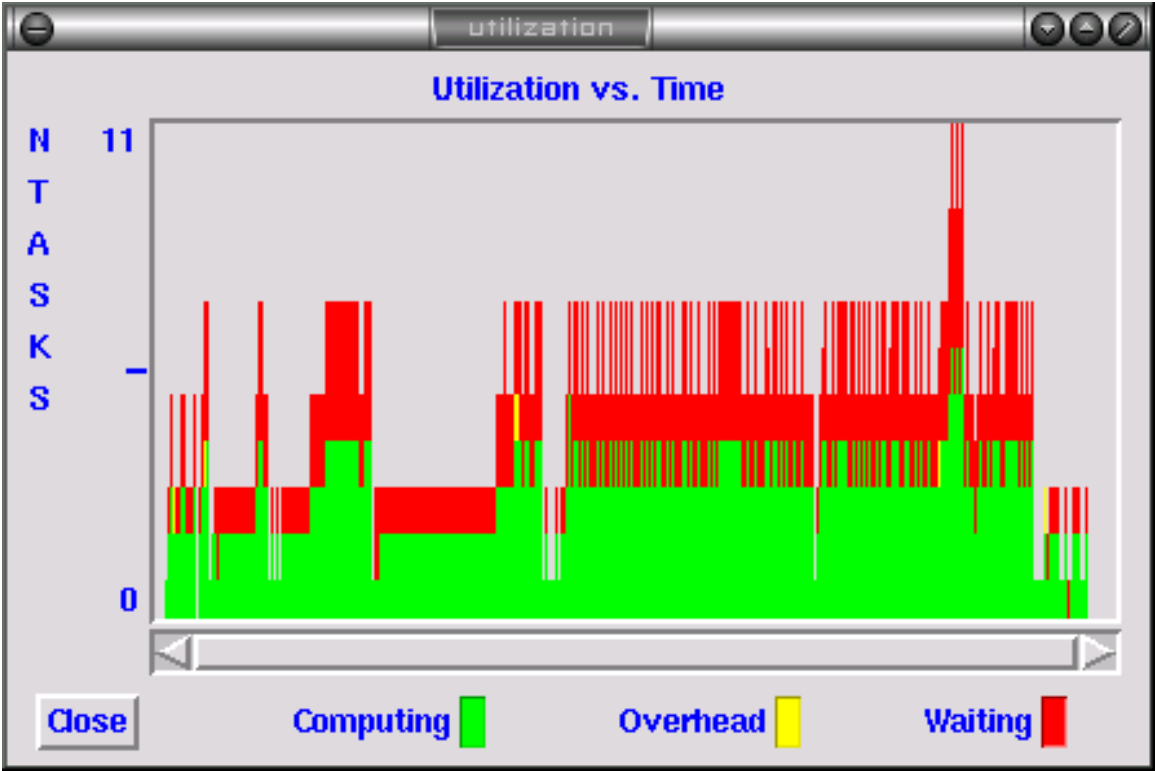


Figura 8.4: Utilização - pvmmake

Resultados		
	mestre	mestre+escravo
Tempo	22 min e 22 seg	17min e 23 seg

Tabela 8.2: Tempo de Compilação do Kernel

Também realizamos a renderização de uma imagem tridimensional utilizando uma adaptação do **PovRay**. O **PovRay** precisou ser adaptado para fazer uso dos recursos do PVM de processamento paralelo. A imagem a ser renderizada foi dividida em pequenos blocos, e cada bloco renderizado por um subprocesso.

Os subprocessos do **PovRay** foram distribuídos entre os nós do cluster. Os tempos gastos na execução estão na Tabela IV.2. Podemos observar que o ganho do processamento no cluster foi de 72,22% em relação ao processamento em apenas um nó.

Resultados		
	mestre [tempo (s)]	mestre+escravo [tempo (s)]
MPI	187	94
PVM	31	18

Tabela 8.3: Renderização POVRay

As condições da rede eram as mesmas do momento em que fizemos a execução do Stress2 e obtivemos ganhos de 100,00%. O ganho com o PVM não é tão grande porque o próprio PVM consome recursos do cluster para realizar o gerenciamento das tarefas e de trocas de mensagens. Essa é a desvantagem em se utilizar o PVM, se comparado ao BProc.

A Figura seguinte foi capturada durante a renderização da imagem no cluster. As próximas duas figuras mostram a imagem renderizada.

8.4 MPI

Realizamos uma renderização de imagem para demonstrar o funcionamento da biblioteca MPI, usando uma adaptação do **PovRay**: o **mpipov**.

O princípio é o mesmo usado na renderização com o **PVM**: a figura é dividida em blocos, e os blocos são distribuídos entre os nós do cluster.

O **MPI** não utiliza daemons para distribuição de tarefas, mas sim o **remote shell**. E isso mostrou-se uma desvantagem, devido à necessidade de execução de tarefas de login cada vez que um processo é enviado a um nó escravo. Entretanto, esse método pode ser mais seguro que o uso de daemons, pois pode-se efetuar o login remoto usando o **ssh** (security

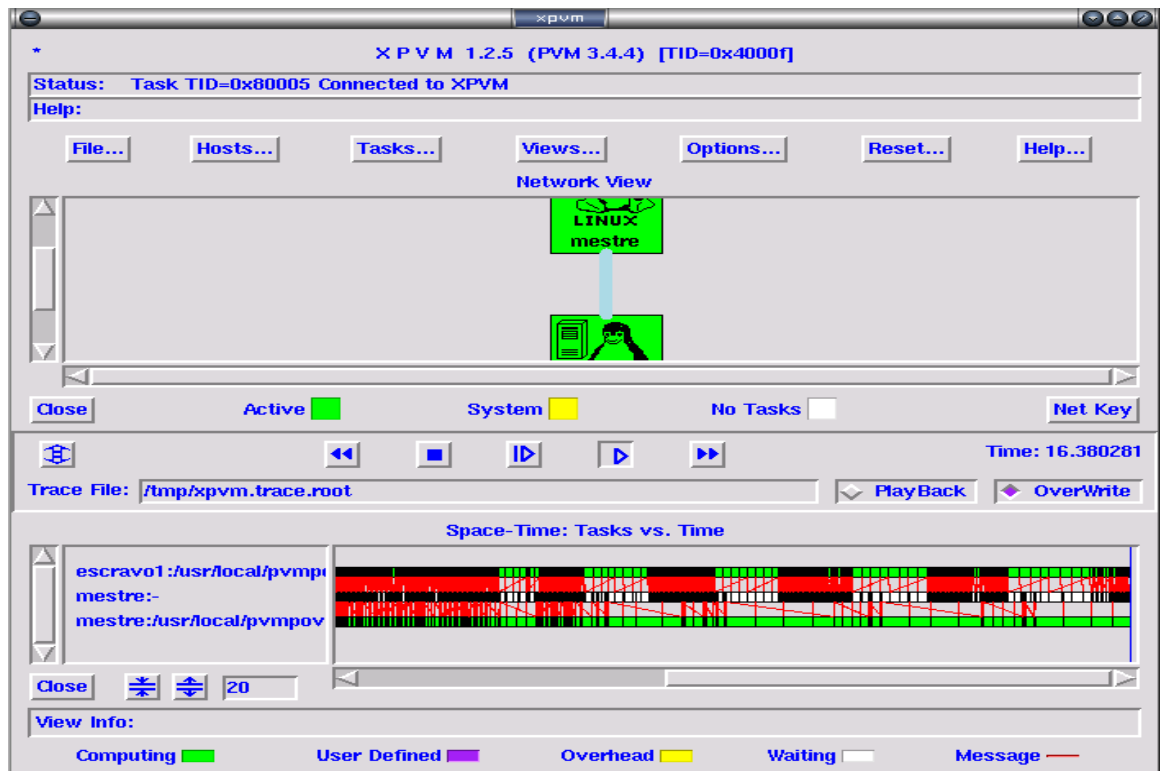


Figura 8.5: XPVM - Linha do Tempo

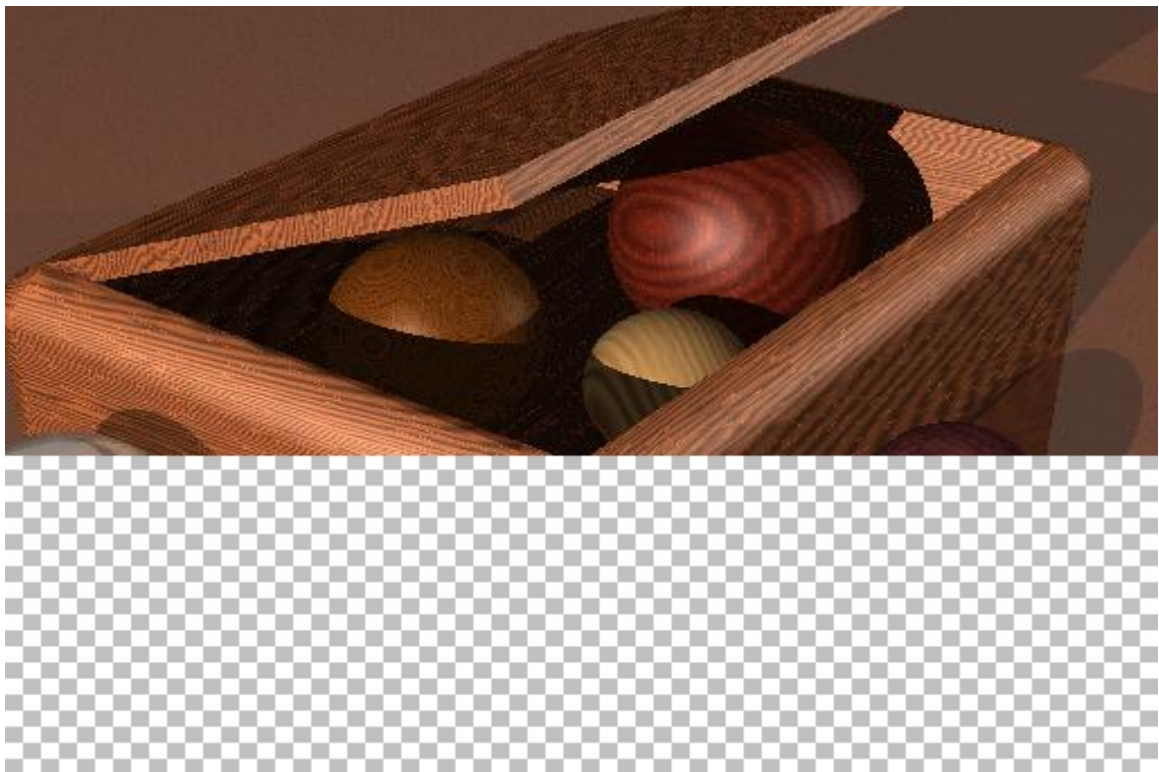


Figura 8.6: Renderização Parcial do POV-Ray



Figura 8.7: Renderização completa do POV-Ray

shell) com sistemas de chaves assimétricas para autenticação.

Processos na máquina **mestre** durante a execução:

```

19175 tty1  S    0:00    /bin/sh ./mpipov skyvase.pov 4
19176 tty1  S    0:00    /bin/sh /usr/local/mpich-BProc/bin/mpirun -np
19304 tty1  S    0:00    /root/mestre/povray31/source/mpi-unix/mpi-x
19305 tty1  S    0:00    /root/mestre/povray31/source/mpi-unix/mpi
19306 tty1  S    0:00    /usr/bin/rsh escravo1 -l root -n /root/me
19307 tty1  S    0:00    /usr/bin/rsh mestre.localdomain -l root -
19311 tty1  S    0:00    /usr/bin/rsh escravo1 -l root -n /root/me

 732 ?      S    0:00    xinetd -reuse
19308 ?      S    0:00    in.rshd
19309 ?      R    0:35    /root/mestre/povray31/source/mpi-unix/mpi-x-pov
19310 ?      S    0:00    /root/mestre/povray31/source/mpi-unix/mpi-x-p

```

Processos na máquina **escravo1**:

767	?	S	0:00	xinetd -reuse
4460	?	S	0:00	in.rshd
4461	?	R	0:14	/root/mestre/povray31/source/mpi-unix/mpi-x-pov
4462	?	S	0:00	/root/mestre/povray31/source/mpi-unix/mpi-x-p
4463	?	S	0:00	in.rshd
4464	?	R	0:14	/root/mestre/povray31/source/mpi-unix/mpi-x-pov
4465	?	S	0:00	/root/mestre/povray31/source/mpi-unix/mpi-x-p

Ao executarmos o **mpipov**, a biblioteca **MPI** se encarregou de distribuir os subprocessos entre os nós utilizando o **rsh** (remote shell). Observe a existência de um processo no **mestre** disparado pelo **remote shell** através do serviço **xinetd**. No **escravo1** podemos observar dois processos disparados pelo **xinetd**. A biblioteca **MPI** tem a mesma abstração da biblioteca **PVM**, ou seja, cria a imagem de uma máquina virtual para o programador, e a própria biblioteca se encarrega de distribuir as tarefas. Na Tabela IV.2 temos os tempos gastos para a renderização em uma máquina apenas, e no cluster: ganho de 98,94%. Esse ganho é um excelente resultado, se analisado de forma isolada. Vamos comparar agora a renderização utilizando o **PVM** e utilizando o **MPI**. A renderização, utilizando apenas uma máquina, com **PVM** obteve um ganho de 503,23% em relação a renderização com **MPI**. Utilizando duas máquinas, o ganho do **PVM** em relação ao **MPI** foi de 422,22%. Esses resultados reforçam a teoria de que a utilização de daemons para distribuição de processos, como ocorre com o **BProc** e o **PVM**, é mais ágil do que a utilização de **remote shell**, como no **MPI**. Quando se utiliza daemons, cada tarefa é executada por um subprocesso desse daemon. Ao usar **remote shell**, um novo processo **shell** é iniciado para cada tarefa, e isso requer a execução de uma série de tarefas de login, tais como verificação de cotas, configuração de variáveis de ambiente, registros nos logs de acesso ao sistema, dentre várias executadas pelo sistema operacional. A execução dessas tarefas de login prejudicam o desempenho do cluster. No caso do **mpipov**, o próprio desenvolvedor recomenda a utilização do **pvmov**[25][26]. A biblioteca **MPI** oferece mais recursos ao programador do que a biblioteca **PVM**. Mas, devido ao problema descrito, o **PVM** é a melhor opção. Está em fase de implementação uma biblioteca **PVM** que incorpore todos os recursos oferecidos pelo **MPI**, numa tentativa de unificar as bibliotecas utilizando o conceito do **PVM** de daemons para distribuição de tarefas [27].

9 Tuning

9.1 O que é

Uma tradução para *tunning* seria "afinação". *Tunning* são os ajustes necessários para a obtenção do máximo desempenho possível. Esse conceito não se aplica apenas em clusters, mas em diversas áreas.

Inúmeros são os fatores que influenciam a performance do cluster. O poder de processamento dos nós é o fator mais importante e notório. A princípio, imagina-se que o poder de processamento de um cluster é a soma das capacidades de cada nó. Isso seria o ideal, mas não acontece na prática, pois a administração do cluster e das tarefas consome recursos das máquinas.

O desempenho da rede também é de suma importância, pois depende dela a distribuição de tarefas e a coleta de resultados.

Esses dois fatores são controlados pelo administrador do cluster e dependem diretamente da quantidade de recursos financeiros disponíveis para a montagem do cluster. Outro fator importante, mas que não está sob controle do administrador, é a granulação das tarefas. Para que uma tarefa possa ser executada em um cluster, ela precisa ser dividida em subprocessos que possam ser distribuídas. A tarefa precisa estar adaptada para sofrer essa fragmentação.

O "tamanho" de cada subprocesso é que determina a granulação. Entenda-se "tamanho" como número de instruções necessárias para executar o subprocesso. Uma tarefa com alta taxa de granulação (grande quantidade de "grãos", ou fragmentos) terá uma grande quantidade de subprocessos de pequenos tamanhos (ou pequenas quantidades de instruções). O fator granulação é controlado pelo programador, pois só ele pode definir os pontos de divisão de cada tarefa.

9.2 Como fazer

Como o fator granulação não depende de recursos financeiros, ele é o fator que deve ser ajustado no processo de *tunning* para se obter o melhor desempenho.

Vamos ilustrar uma situação para abordar esse tema:

Um cluster com dois nós. O nó "A" tem poder de processamento de um milhão (1M) de instruções por segundo. O nó "B" pode processar dois milhões (2M) de instruções por segundo. Existe ainda o nó mestre, que apenas coordena a distribuição e execução de tarefas. Os nós estão interligados por uma rede, com capacidade de transmissão de 10 mensagens por segundo. A tarefa a ser executada tem 100 milhões (100M) de instruções. Na primeira situação, vamos dividir a tarefa em 4 subprocessos independentes (que não dependem do término de outro subprocesso para serem processados), de 25M instruções cada. Vejamos as etapas do processo:

Resultados		
Relógio	Etapas	Tempo Gasto
0,0s	SuBProcesso enviado para nó A	0,1s
0,1s	Nó A inicia o subprocessamento (subprocesso 1)	25,0s
0,1s	SuBProcesso 2 enviado para nó B	0,1s
0,2s	Nó B inicia processamento (subprocesso 2)	12,5s
12,7s	Nó B envia resultado (subprocesso 2)	0,1s
12,8s	SuBProcesso 3 enviado para nó B	0,1s
12,9s	Nó B inicia processamento (subprocesso 3)	12,5s
25,1s	Nó A envia resultado (subprocesso 1)	0,1s
25,2s	SuBProcesso 4 enviado para nó A	0,1s
25,3s	Nó A inicia processamento (subprocesso 4)	25,0s
25,4s	Nó B envia resultado (subprocesso 3)	0,1s
50,3s	Nó A envia resultado (subprocesso 4)	0,1s
50,4s	Fim do processamento	

Se a tarefa estivesse sendo executada toda em "A", o tempo gasto seria de 100 segundos. Se fosse executada em "B", o tempo necessário seria de 50 segundos. Observe que o

tempo necessário para a execução na situação ilustrada foi maior do que o necessário para a execução no nó mais rápido do cluster. Houve uma perda de 0,8%. Isso ocorreu porque o nó "B" ficou ocioso a partir do tempo 25,4s, totalizando 25,0s de ociosidade.

Vamos repetir essa simulação, mas agora dividindo a tarefa em 10 subprocessos de 10M.

Vejamos as etapas do processo:

Resultados		
Relógio	Etapa	Tempo Gasto
0,0s	Subprocesso 1 enviado para nó A	0,1s
0,1s	Subprocesso 2 enviado para nó B	0,1s
0,1s	Nó A inicia processamento (subprocesso 1)	10,0s
0,2s	Nó B inicia processamento (subprocesso 2)	5,0s
5,2s	Nó B envia resultado (subprocesso 2)	0,1s
5,3s	Subprocesso 3 enviado para nó B	0,1s
5,4s	Nó B inicia processamento (subprocesso 3)	5,0s
10,1s	Nó A envia resultado (subprocesso 1)	0,1s
10,2s	Subprocesso 4 enviado para nó A	0,1s
10,3s	Nó A inicia processamento (subprocesso 4)	10,0s
10,4s	Nó B envia resultado (subprocesso 3)	0,1s
10,5s	Subprocesso 5 enviado para nó B	0,1s
10,6s	Nó B inicia processamento (subprocesso 5)	5,0s
15,6s	Nó B envia resultado (subprocesso 5)	0,1s
15,7s	Subprocesso 6 enviado para nó B	0,1s
15,8s	Nó B inicia processamento (subprocesso 6)	5,0s
20,3s	Nó A envia resultado (subprocesso 4)	0,1s
20,4s	Subprocesso 7 enviado para nó A	0,1s
20,5s	Nó A inicia processamento (subprocesso 7)	10,0s
20,8s	Nó B envia resultado (subprocesso 6)	0,1s
20,9s	Subprocesso 8 enviado para nó B	0,1s
21,0s	Nó B inicia processamento (subprocesso 8)	5,0s
26,0s	Nó B envia resultado (subprocesso 8)	0,1s
26,1s	Subprocesso 9 enviado para nó B	0,1s
26,2s	Nó B inicia processamento (subprocesso 6)	5,0s
30,5s	Nó A envia resultado (subprocesso 7)	0,1s
30,6s	Subprocesso 10 enviado para nó A	0,1s
30,7s	Nó A inicia processamento (subprocesso 1)	10,0s
31,2s	Nó B envia resultado (subprocesso 6)	0,1s
40,7s	Nó A envia resultado (subprocesso 1)	0,1s
40,8s	Fim do processamento	

Veja que agora houve um ganho de 22,5%, e que o tempo de ociosidade do nó "B" foi de apenas 9,6 segundos.

O tempo máximo de ociosidade de um nó é o tempo necessário para processar o maior subprocesso no nó com menor poder de processamento. Na primeira simulação esse tempo máximo era de 25 segundos. Já na segunda esse tempo diminuiu para 10 segundos.

Se desconsiderarmos o tempo gasto com a troca de mensagens, quanto maior a taxa de granulação (ou número de subprocessos), menor será o tempo máximo de ociosidade de um nó, e melhor será o desempenho do cluster.

Entretanto, aumentar a granulação implica em um número maior de troca de mensagens. Se a rede ficar congestionada, os nós podem ficar ociosos aguardando a chegada de mensagens, e ociosidade significa menor desempenho.

Outro fator importante é o tempo necessário para administrar cada subprocesso. Em tarefas pequenas esse tempo é desprezível, mas em grandes tarefas deve ser considerado se não existir um nó dedicado a essa função.

Portanto, o objetivo do *tunning* é aumentar a granulação da tarefa até atingir o limite da capacidade de transmissão da rede que interliga os nós do cluster. Quando essa situação for atingida, o cluster estará operando com desempenho total.

9.3 Outras Considerações

O ajuste da granulação também pode levar à conclusão de que a rede que interliga os nós está impondo um tempo máximo de ociosidade elevado demais, e que precisa ser melhorada. Também pode levar à conclusão de que o nó com menor poder de processamento precisa ser retirado para diminuir o tempo máximo de ociosidade.

Outro fato importante ocorre em clusters que executam várias tarefas simultaneamente. Mesmo que as tarefas não tenham a granulação ideal, o cluster pode operar com desempenho máximo. As tarefas não serão executadas no menor tempo possível, mas os diferentes "tamanhos" de subprocessos podem levar a uma condição de baixa ociosidade do cluster no geral. É por esse motivo que clusters multi-tarefas de propósito geral não se preocupam em determinar a granulação com precisão.

10 Considerações para trabalhos futuros

Durante o desenvolvimento do trabalho, identificamos alguns assuntos que podem contribuir para o desenvolvimento e aplicação de cluster Beowulf, especialmente no que diz respeito a automatização do processo de implementação do cluster.

10.1 Distribuição baseada em Beowulf

Assim como a OpenMosix, a idéia básica é construir uma distribuição "bootável" através de um CD baseada em Beowulf. A intenção principal é que o mestre fique com o CD "bootável", e que essa distribuição possua scripts que ativem os servidores e auto-configurem os arquivos das aplicações. Os compiladores dessa distribuição são todos distribuídos e próprio para cada biblioteca de programação paralela (MPI, PVM, BProc e etc.)

Nessa distribuição deve ter aplicações ou scripts que crie disquetes para boot remoto dos escravos, utilizando apenas o floppy, poupando assim o uso do HD e drive de CD dessas estações.

O mestre deve ter a capacidade de auto-detectar a inicialização dos escravos.

10.2 Boot Remoto

A idéia principal é desenvolver maneiras que facilitem o boot dos escravos para o cluster usando técnicas de boot BOOTP/DHCP, TFTP e pela EPROM da placa de rede.

10.3 Distribuição em Floppy

Assim como o OpenMosixLOAF, existe a possibilidade de criar uma distribuição que caiba em um disquete, e que torne mais prático a configuração das estações.

10.4 LinuxBios

LinuxBIOS é um projeto aberto que visa criar um sistema de boot simples e rápido. O projeto foi iniciado como parte do trabalho de pesquisa no Laboratório Nacional de Los Alamos. A idéia é ganhar o controle dos nós(estações) através de uma imagem do sistema operacional (kernel-Linux) gravada na BIOS (reprogramação). Isto permite uma maior rapidez para carregar o sistema operacional, o que é denominado "carregar a frio". O tempo mais rápido de um boot já registrado é de 3 segundos. Outra vantagem de usar LinuxBIOS é a economia de energia, pois apenas dois motores trabalham na máquina: O ventilador do processador central e da fonte de alimentação.

LinuxBIOS é um Linux bem simples:

- Possui 10 patches inclusos no kernel.
- Possui 500 linhas em Assembler e 5000 linhas em C - para inicialização do kernel.
- Executa 16 instruções para entrar no modo 32-bit, acessar a memória e inicializar outros hardwares que serão necessários após o Linux iniciar.

10.5 Sistemas de Arquivos Distribuído

Existe um projeto chamado V9FS cujo o objetivo é criar uma visão única de disco. O sistema funciona como o NFS, porém, fazendo com que vários discos se unam virtualmente, criando uma imagem única de disco.

11 Conclusão

Este trabalho fornece algumas soluções para a construção de supercomputadores, utilizando *softwares* livres, reaproveitando máquinas antigas, possibilitando a construção de supermáquinas caseiras. Essas soluções podem suprir as necessidades de grandes Universidades e empresas de médio porte.

Todas essas soluções apresentadas possuem uma total viabilidade, algumas chegaram a 100% de ganho, porém, existe a dificuldade para encontrar aplicações para esta plataforma. Nos nossos testes o BPROC obteve um grande desempenho, por ser leve e trabalhar diretamente no kernel do sistema operacional. O nó escravo chegou a processar em 100%, aproveitando todos os recursos dele. Outra vantagem foi o não congestionamento da rede. O MPI não alcançou as expectativas deste trabalho, deixando a desejar na renderização de imagens. Isto acontece devido a grande troca de mensagens o que gera um alto overhead. Além de ser prático o PVM agradou em desempenho chegando a render 200%, diminuindo o tempo de processamento pela metade.

Dentre essas três implementações, o BPROC e o PVM atingiram as nossas expectativas. Um grande problema encontrado nessas três implementações, é a grande complexidade de programar para ambientes paralelos.

Para empresas e universidades é uma porta aberta para investimentos em tecnologia, principalmente as que requerem recursos computacionais. Pode-se abrir várias áreas de estudo em relação a essas soluções como : programação para sistemas distribuídos, programação paralela, escalonamento de processos, processamento de imagens distribuídas, sistemas de arquivos, arquiteturas de redes, gerência de rede, criação de protocolos mais eficientes.

I Programas Utilizados

I.1 Stress1

```
/******
```

CEFET-GO

Redes de Computadores

Trabalho de Conclusão de Curso

Esli P. Júnior

Reinaldo B. Freitas

M. Sc. Cloves F. J. (orientador)

```
*****
```

stress1.cpp

Realiza multiplicações com números em ponto flutuante
do tipo long double (80 bits para i586), divididas em
subprocessos.

Forma de uso:

```
./stress1 <fator> <processos>
```

<processos> Indica o número de subprocessos que serão
criados para realizar os cálculos

Cada processo executará (fator*fator) multiplicações
em ponto flutuante.

O processo pai indicará a hora do início do processamento
e do fim, assim como o tempo gasto (em segundos).

*/

```
#include <stdio.h>
```

```
#include <wait.h>
```

```
#include <signal.h>
```

```
#include <string.h>
```

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
#include <time.h>
```

```
#define OK          0
```

```
#define ERRO        1
```

```
#define MAX_ OPER    50000
```

```
#define MAX_ PROC    50
```

```
int main( int  argc, char *argv[] ) {
```

```
    long double fator, acumulado;           // Usados no cálculo
```

```
    unsigned int operacoes, processos;      // Parâmetros
```

```
int temp = 0, i = 0, j = 0, k = 0;      // Uso geral

int filhos = 0; //                      Número de subprocessos em execução

time_t ini, fim;                       // Marcação do tempo de início e de término


// Escolhidos ao acaso

fator = 56.5873615e20;

acumulado = 9867.37464425e35 * fator;


// Verificando parâmetros

if( argc != 3 ) {

printf( "Forma de uso: %s <num operera> <num procs>\n", argv[0] );

    exit( ERRO );

}

operacoes = atoi( argv[1] );

if( operacoes <= 0 || operacoes > MAX_OPER ) operacoes = MAX_OPER;

processos = atoi( argv[2] );

if( processos <= 0 || processos > MAX_PROC ) processos = MAX_PROC;


// Marcando hora inicial

ini = time( &ini );

printf( "Inicio: %s\n", ctime(&ini) );


// Criando subprocessos

for( i = 0; i < processos; i++ ) {

    // Utilizada a função fork() padrão para criar subprocessos

    temp = fork();
```

```
        if( temp < 0 ) {

            perror( "Erro ao criar subprocesso\n");

            exit( ERRO );

        }

        // Processo filho: temp == 0

        if( temp == 0 ) {

            for( j = 0; j < operacoes; j++ ) {

                for( k = 0; k < operacoes; k++ ) {

                    // Cálculo propriamente dito

                    acumulado *= acumulado;

                }

            }

            exit( OK );      // Fim do processo filho

        }

        // Processo pai                filhos++;

        printf( "Filho %d -> pid %d\n", filhos, temp );

    }      // Aguardando o fim de todos os processos filhos

    while( filhos > 0 ) {

        i = wait( &temp );

        if( i > 0 ) {

            printf( "Fim do pid %d (status = %d)\n", i, temp );

            filhos--;

        }

    }

}

// Marcando hora final e apresentando resultados
```

```
        fim = time( &fim );

printf( "Fim: %s\n",  ctime(&fim) );

printf( "Tempo gasto (s): %d\n", fim - ini );

printf( "Numero total de operacoes: %d\n", operacoes * operacoes * processos);

}
```

```
/******
```

stress1.cpp

```
*****/
```

I.2 Stress2

```
/******
```

CEFET-GO

Redes de Computadores

Trabalho de Conclusão de Curso

Esli P. Júnior

Reinaldo B. Freitas

M. Sc. Cloves F. J. (orientador)

```
*****/
```

stress2.cpp

Realiza multiplicações com números em ponto flutuante do tipo long double (80 bits para i586), divididas em subprocessos.

PS: esse programa utiliza a biblioteca BPROC, e precisa ser linkado com essa biblioteca:

```
$ gcc -lBProc -o stress2 stress2.cpp
```

Forma de uso:

```
./stress1 <fator> <processos>
```

<processos> Indica o número de subprocessos que serão criados para realizar os cálculos

Cada processo executará (fator*fator) multiplicações em ponto flutuante.

O processo pai indicará a hora do início do processamento e do fim, assim como o tempo gasto (em segundos).

```
*/
```

```
#include <stdio.h>
```

```
#include <wait.h>
```

```
#include <signal.h>
```

```
#include <string.h>
```



```
#include <stdlib.h>

#include <unistd.h>

#include <time.h>

#include <sys/BProc.h>


#define OK                0

#define ERRO              1


#define MAX_OPER          50000

#define MAX_PROC          50


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


    long double fator, acumulado;           // Usados no cálculo

    unsigned int operacoes, processos;     // Parâmetros

    int temp = 0, i = 0, j = 0, k = 0;      // Uso geral

    int filhos = 0;                         // Número de subprocessos em execução

    time_t ini, fim;                        // Marcação do tempo de início e de término

    struct BProc_node_info_t *hosts; // Matriz com as informações dos nós

    int nhosts;                             // Número de nós do cluster

    int no = 0;


    // Consultando cluster

    nhosts = BProc_nodelist( &hosts );

    if( nhosts <= 0 ) {

        perror( "Erro ao consultar cluster.\n"
```

```
    );

    exit( ERRO );

}

// Escolhidos ao acaso

fator = 56.5873615e20;

acumulado = 9867.37464425e35 * fator;


// Verificando parâmetros

if( argc != 3 ) {

printf( "Forma de uso: %s <num operera> <num procs>\n", argv[0] );

    exit( ERRO );

}

operacoes = atoi( argv[1] );

if( operacoes <= 0 || operacoes > MAX_OPER ) operacoes = MAX_OPER;

processos = atoi( argv[2] );

if( processos <= 0 || processos > MAX_PROC ) processos = MAX_PROC;


// Marcando hora inicial

ini = time( &ini );

printf( "Inicio: %s\n", ctime(&ini) );


// Criando subprocessos

for( i = 0; i < processos; i++ ) {

    // Procurando um nó ativo no cluster para executar o

    // próximo subprocesso
```

```
while( hosts[no].status != BProc_node_up ) {  
    no++;  
    if( no >= nhosts ) no = 0; // Voltando ao início da fila  
}  
  
// Criando subprocesso com a função do BPROC  
// O nó de destino já está indicado por hosts[no].node  
temp = BProc_rfork( hosts[no].node );  
  
if( temp < 0 ) {  
printf( "Erro ao criar subprocesso (nhosts=%d) (no=%d)\n", nhosts, no );  
exit( ERRO );  
}  
  
// Processo filho: temp == 0  
if( temp == 0 ) {  
    for( j = 0; j < operacoes; j++ ) {  
        for( k = 0; k < operacoes; k++ ) {  
            // Cálculo propriamente dito  
            acumulado *= acumulado;  
        }  
    }  
    exit( OK ); // Fim do processo filho  
}  
  
// Processo pai  
filhos++;  
  
printf( "Filho %d -> pid %d -> no %d\n", filhos, temp, hosts[no].node );  
  
// Avançando na fila  
no++;
```

```
        if( no >= nhosts ) no = 0; // Voltando ao início da fila
    }

    // Aguardando o fim de todos os processos filhos

    while( filhos > 0 ) {

        i = wait( &temp );

        if( i > 0 ) {

            printf( "Fim do pid %d (status = %d)\n", i, temp );

            filhos--;

        }

    }

    // Marcando hora final e apresentando resultados

    fim = time( &fim );

    printf( "Fim: %s\n", ctime(&fim) );

    printf( "Tempo gasto (s): %d\n", fim - ini );

    printf("Numero total de operacoes:%d\n", operacoes * operacoes * processos);

}

/*****

stress2.cpp

*****/
```

II Arquivos de Configuração

II.1 machines.LINUX

```
#Change this file to contain the machines that you want to use
#to run MPI jobs on. The format is one host name per line, with either
#hostname
#or
#hostname:n
#where n is the number of processors in an SMP. The hostname should
# be the same as the result from the command hostname
mestre.localdomain
escravo1
```

II.2 .rhosts

```
mestre beowulf
mestre root
escravo1 beowulf
escravo1 root
```

II.3 .xpvm_hosts

```
mestre
escravo1 lo=beowulf
```

II.4 bashrc

/etc/bashrc

System-wide functions and aliases

Environment configuration on/etc/profile

```
PS1="[u@h W]$"
```

```
alias which="type -path" alias l="ls -laF --color=auto"
```

```
alias ls="ls --color=auto" alias m="minicom -s -con -L"
```

```
alias minicom="minicom -s -con -L"
```

```
alias tm="tail -f /var/log/messages"
```

```
alias tmm="tail -f /var/log/maillog"
```

```
alias tms="tail -f /var/log/secure"
```

```
alias cds="cd /etc/rc.d/init.d ls"
```

```
alias fd="mount /dev/fd0 /mnt/floppy; cd /mnt/floppy ls"
```

```
alias ufd="cd /mnt umount floppy ls"
```

```
alias ldir="mount /mnt/floppy 1 /mnt/floppy umount /mnt/floppy"
```

```
export PVM_ROOT=/usr/share/pvm3
```

```
export PVM_ARCH=BEOLIN
```

```
#export PVM_TMP=/tmp/pvm
```

```
#export PVM_TMP=/root/escravo1/pvmtmp
```

```
export PVM_TMP=/tmps
```

```
export PATH=$PATH:$PVM_ROOT/lib:$PVM_ROOT/lib/$PVM_ARCH:$PVM_ROOT/bin/$PVM_ARCH
```

```
export PVM_DPATH=$PVM_ROOT/lib/pvmd
```

```
export PROC_LIST=mestre:escravo1 lo=beowulf
```

```
export XPVM_ROOT=/usr/local/xpvm
```

```
export TCL_LIBRARY=/usr/lib/tcl
```

```
export TK_LIBRARY=/usr/lib/tk
```

```
export PATH=$PATH:/usr/local/mpich-1.2.4/bin export
```

```
MPIR_HOME=/usr/local/mpich-1.2.4
```

II.5 mpipov

```
#!/bin/sh

/usr/local/mpich-BProc/bin/mpirun -np \${2}/root/mestre/povray31-mpi/source/mpi-
unix/mpi-x-povray $3 -I $1 +v1 +ft -x +mb25 +a.0.300 +j1.000 +r3 -q9 -w640 -H480 -S1 -
E480 -k0.000 -mv2.0 +b1000 +L/root/mestre/povray31-mpi/include/ +NH128 +NW128
$4 $5 $6 $7 $8"
```

II.6 Makefile.aimk

```
#

# \${Id}: Makefile.aimk,v 1.40 2001/05/11 18:58:10 pvmsrc Exp \${
#
# Generic Makefile body to be concatenated to config header.
#
# Imports:
# PVM_ARCH    = the official pvm-name of your processor
# ARCHCFLAGS  = special cc flags
# ARCHDLIB    = special libs needed for daemon
# ARCHDOBJ    = special objects needed for daemon
# HASRANLIB   = 't' or 'f'
# AMEM        = Memory page locks from Convex
# PLOCKFILE   = Page Lock in line code from SUN
#
# Compatibility defines (usually in conf/*.def):
# FDSETPATCH   if system includes don't have fd_set stuff
# USESTRERROR   if sys_errlist, sys_nerr not public, use strerror()
# HASERRORVARS  if errno, sys_nerr, sys_errlist already declared
# HASSTDLIB     if system has stdlib.h
# NEEDENDIAN    to include <endian.h>
# NEEDMENDIAN   to include <machine/endian.h>
# NOGETDTBLSIZ if system doesn't have getdtablesize()
```

```

# NOEXEC      if system doesn't have rexec()
# NOSOCKOPT    if system doesn't have setsockopt() / it doesn't work
# NOSTRCASE    if system doesn't have strcasecmp, strncasecmp
# NOTMPNAM     if system doesn't have tmpnam() or it's hosed
# NOUNIXDOM    to disable use of Unix domain sockets
# NOWAIT3      if system doesn't have wait3()
# NOWAITPID    if system doesn't have waitpid() either
# RSHCOMMAND=  for rsh command other than /usr/ucb/rsh""
#              (can override using PVM_RSH environment variable)
# SHAREDTMP    if /tmp is shared between machines (yecch)
# SOCKADHASLEN if struct sockaddr has an sa_len field
# SYSVBFUNC    if system uses memcpy() instead of bcopy(), etc.
# SYSVSIGNAL   if system has sysV signal handling
# SYSVSTR      if system uses strchr() instead of index()
# UDPMAXLEN=   for alternate max udp packet size
# NEEDSSELECTH if you need to include select.h to get fd_set (IBMs)
# SOCKLENISUINT if socket parameter is unsigned int (or size_t),
#              generally -> recvfrom() accept() getsockname()
#              remove flag for AIX 4.1 compiler complaints...
# FDSETNOTSTRUCT if fd_set var declarations do not require struct""
# CTIMEISTIMET if ctime() takes an arg of type time_t""
# SYSERRISCONST if char *sys_errlist[] defined as const
#
# Options defines:
# CLUMP_ALLOC  allocates several data structures in big chunks
# MCHECKSUM    to enable crc checksums on messages
# RSHNPLL=     for number of parallel rsh startups (default is 5)
# RSHTIMEOUT=  for rsh timeout other than default (60 sec)
# STATISTICS   to enable collection of statistics in pvmd
# TIMESTAMPLOG to enable timestamps in pvmd log file
# USE_PVM_ALLOC to enable instrumented malloc functs (for pvm debug)
# USE_GNU_REGEX to enable use of GNU Regex for Mbox Lookup
#              -> requires installation of GNU regex, as well as

```



```

#                               modifying the following defines (see below):
#                               REGEXCONF, REGEXCONFOS2, REGEXOBSJS
#

SHELL      =    /bin/sh
PVMDIR     =    ../..
SDIR       =    \$(PVMDIR)/src
LIBDIR     =    \$(PVMDIR)/lib/\$(PVM_ARCH)
CFLOPTS    =    -O
#CFLOPTS   =    -g
#OPTIONS   =    -DCLUMP_ALLOC
#OPTIONS   =    -DSTATISTICS
#OPTIONS   =    -p
OPTIONS    =    -DCLUMP_ALLOC -DSTATISTICS \
               -DTIMESTAMPLOG -DSANITY
CFLAGS     =    \$(CFLOPTS) \$(OPTIONS) -I\$(PVMDIR)/include \
               -DARCHCLASS=\"\$(PVM_ARCH)\" -DIMA_\$(PVM_ARCH) \
               \$(ARCHCFLAGS)

LIBPREFIX  =    lib
LIBPVM     =    \$(LIBPREFIX)pvm3

#
# GNU Regular Expression Defines - set if needed
#

REGEXCONF  =
#REGEXCONF =    regexconfig

REGEXCONFOS2 =
#REGEXCONFOS2 =    regexconfig-os2

REGEXOBSJS =

```

```
#REGEXOBSJS      =    pvmregex.o regex.o
```

```
#
```

```
# PVM Daemon & Library Objects
```

```
#
```

```
DOBJ              = \
    ddpro.o \
    host.o \
    hoster.o \
    imalloc.o \
    msgbox.o \
    pkt.o \
    pmsg.o \
    pvmalloc.o \
    pvmcruft.o \
    pvmd.o \
    pvmdpack.o \
    pvmdtev.o \
    pvmmerr.o \
    pvmmfrag.o \
    pvmmlog.o \
    sdpro.o \
    task.o \
    tdpro.o \
    waitc.o \
    global.o \
    \$(REGEXOBSJS)
```

```
SOCKDOBJ          = \
    pvmdabuf.o \
    pvmdunix.o
```

```
SHMEMDOBJ    = \  
    \$(AMEMOBJ) \  
    nmdclass.o \  
    pvmdshmem.o \  
    pvmdshmem.o
```

```
MPPDOBJ      = \  
    mppchunkhost.o \  
    mppmsgghost.o \  
    lmsg.o \  
    pvmdabuf.o \  
    pvmdmimd.o \  
    pvmdunix.o
```

```
BEODOBJ      = \  
    pvmdmimd.o
```

```
OS2DOBJ      = \  
    deathapi.o \  
    sthoster.o \  
    stdlog.o \  
    rexec.o \  
    ruserpas.o \  
    os2spawn.o
```

```
LOBJ         = \  
    imalloc.o \  
    tev.o \  
    lpvmcat.o \  
    lpvmgen.o \  
    lpvmpack.o \  
    lpvmglob.o \  
    pmsg.o \  
    pmsg.o
```

```
pvmalloc.o \  
pvmcruft.o \  
pvmerr.o \  
pvmfrag.o \  
waitc.o \  
global.o  
  
LPVMSOCK      =   lpvm.o  
  
SOCKLOBJ      = \  
    pvmdabuf.o  
  
LPVMSHMEM     =   lpvmshmem.o  
  
SHMEMLOBJ     = \  
    \$(AMEMOBJ) \  
    pvmsmem.o  
  
LPVMMIMD      =   lpvmmimd.o  
  
MPPLOBJ       = \  
    mppchunknode.o \  
    mppmsgnode.o \  
    lmsg.o \  
    lpvmmpp.o \  
    pvmdabuf.o  
  
OS2LOBJ       =   stdlog.o  
  
REGEXSRC      = \  
    \$(SDIR)/regex/Makefile.in \  
    \$(SDIR)/regex/configure \  
    \$(SDIR)/regex/configure.in \  

```

```
$(SDIR)/regex/pvmregex.c \
$(SDIR)/regex/regex.c \
$(SDIR)/regex/regex.h
```

```
REGEXDIR      =    ./regex
```

```
REGEXCP       = \
    $(REGEXDIR)/Makefile.in \
    $(REGEXDIR)/configure \
    $(REGEXDIR)/configure.in \
    $(REGEXDIR)/pvmregex.c \
    $(REGEXDIR)/regex.c \
    $(REGEXDIR)/regex.h
```

```
REGEXOPTS     =    CC=$(CC) \
                   CFLAGS=(CFOPTS) -DREGEX_MALLOC(ARCHCFLAGS)"""
```

```
SHMEMTARGETS  =    $(LIBDIR)/pvmd3-shm \
                   $(LIBDIR)/lib-shm $(LIBDIR)/$(LIBPVM)s.a
```

```
MPPTARGETS    =    $(LIBDIR)/pvmd3-mpp \
                   $(LIBDIR)/$(LIBPVM).a $(LIBDIR)/$(LIBPVM)pe.a
```

```
BEOTARGETS    =    $(LIBDIR)/pvmd3-beo $(LIBDIR)/$(LIBPVM).a
```

```
OS2TARGETS    =    $(LIBDIR)/pvmd3-os2 $(LIBDIR)/lib-os2
```

```
all:          pvmd3$(EXESFX) $(LIBPVM).a
```

```
all-shm:      pvmd3-shm lib-shm $(LIBPVM)s.a
```

```
all-mpp:      pvmd3-mpp $(LIBPVM).a $(LIBPVM)pe.a
```

```
all-beo:      pvmd3-beo $(LIBPVM).a

all-os2:      pvmd3-os2 lib-os2

install:      $(LIBDIR) $(LIBDIR)/pvmd3$(EXESFX) $(LIBDIR)/$(LIBPVM).a

install-shm:   $(LIBDIR) $(SHMENTARGETS)

install-mpp:   $(LIBDIR) $(MPPTARGETS)

install-beo:   $(LIBDIR) $(BEOTARGETS)

install-os2:   $(LIBDIR) $(OS2TARGETS)

$(LIBDIR):
    - mkdir $(LIBDIR)

# libdir sock pvmd3 & libpvm3.a

$(LIBDIR)/pvmd3$(EXESFX):  pvmd3$(EXESFX)
    cp pvmd3$(EXESFX) $(LIBDIR)

$(LIBDIR)/$(LIBPVM).a:  $(LIBPVM).a
    cp $(LIBPVM).a $(LIBDIR)

# libdir shm pvmd3 & libpvm3*.a

$(LIBDIR)/pvmd3-shm:      pvmd3-shm
    cp pvmd3$(EXESFX) $(LIBDIR)
    touch $(LIBDIR)/pvmd3-shm

$(LIBDIR)/lib-shm:  lib-shm
    cp $(LIBPVM).a $(LIBDIR)
```

```
touch $(LIBDIR)/lib-shm

$(LIBDIR)/$(LIBPVM)s.a: $(LIBPVM)s.a
    cp $(LIBPVM)s.a $(LIBDIR)/$(LIBPVM)s.a

# libdir mpp pvmd3 & libpvm3*.a

$(LIBDIR)/pvmd3-mpp:    pvmd3-mpp
    cp pvmd3$(EXESFX) $(LIBDIR)
    touch $(LIBDIR)/pvmd3-mpp

$(LIBDIR)/$(LIBPVM)pe.a:    $(LIBPVM)pe.a
    cp $(LIBPVM)pe.a $(LIBDIR)

# libdir Beowulf pvmd3

$(LIBDIR)/pvmd3-beo:    pvmd3-beo
    cp pvmd3$(EXESFX) $(LIBDIR)
    touch $(LIBDIR)/pvmd3-beo

# libdir os2 pvmd3 & libpvm3*.a

$(LIBDIR)/pvmd3-os2:    pvmd3-os2
    cp pvmd3$(EXESFX) $(LIBDIR)
    touch $(LIBDIR)/pvmd3-os2

$(LIBDIR)/lib-os2:    lib-os2
    cp $(LIBPVM).a $(LIBDIR)
    touch $(LIBDIR)/lib-os2

# sock pvmd3 & libpvm3.a

pvmd3$(EXESFX): $(REGEXCONF) $(DOBJ) $(SOCKDOBJ)
```

```

$(CC) $(CFLAGS) -o pvmd3$(EXESFX) $(DOBJ) $(SOCKDOBJ) \
    $(LOPT) $(ARCHDLIB)

$(LIBPVM).a:    $(REGEXCONF) $(LOBJ) $(LPVMSOCK) $(SOCKLOBJ)
    rm -f $(LIBPVM).a
    $(AR) cr $(LIBPVM).a $(LOBJ) $(LPVMSOCK) $(SOCKLOBJ)
    case x$(HASRANLIB) in xt ) echo ranlib; ranlib $(LIBPVM).a ;; esac

# shm pvmd3 & libpvm3*.a

pvmd3-shm:    $(REGEXCONF) $(DOBJ) $(SHMEMDOBJ)
    $(CC) $(CFLAGS) -o pvmd3$(EXESFX) $(DOBJ) $(SHMEMDOBJ) $(ARCHDLIB)
    touch pvmd3-shm

lib-shm:    $(REGEXCONF) $(LOBJ) $(LPVMSHMEM) $(SHMEMLOBJ)
    rm -f $(LIBPVM).a
    $(AR) cr $(LIBPVM).a $(LOBJ) $(LPVMSHMEM) $(SHMEMLOBJ)
    case x$(HASRANLIB) in xt ) echo ranlib; ranlib $(LIBPVM).a ;; esac
    touch lib-shm

$(LIBPVM)s.a:    $(REGEXCONF) $(LOBJ) $(LPVMSOCK) $(SOCKLOBJ)
    rm -f $(LIBPVM)s.a
    $(AR) cr $(LIBPVM)s.a $(LOBJ) $(LPVMSOCK) $(SOCKLOBJ)
    case x$(HASRANLIB) in xt ) echo ranlib; ranlib $(LIBPVM)s.a ;; esac

# mpp pvmd3 & libpvm3*.a

pvmd3-mpp:    $(REGEXCONF) $(DOBJ) $(MPPDOBJ)
    $(CC) $(CFLAGS) -o pvmd3$(EXESFX) $(DOBJ) $(MPPDOBJ) \
    $(LOPT) $(ARCHDLIB)
    touch pvmd3-mpp

$(LIBPVM)pe.a:    $(REGEXCONF) $(LOBJ) $(LPVMMIMD) $(MPPLOBJ)

```



```

rm -f $(LIBPVM)pe.a

$(AR) cr $(LIBPVM)pe.a $(LOBJ) $(LPVMMIMD) $(MPPLOBJ)

# Beowulf(LINUX) pvmd3 & libpvm3.a

pvmd3-beo: $(REGEXCONF) $(DOBJ) $(BEODOBJ) $(SOCKDOBJ)
    $(CC) $(CFLAGS) -o pvmd3$(EXESFX) $(DOBJ) $(BEODOBJ) $(SOCKDOBJ) \
        $(LOPT) $(ARCHDLIB)

# os2 pvmd3 & libpvm3.a

pvmd3-os2: $(REGEXCONFOS2) $(OS2DOBJ) $(DOBJ) $(SOCKDOBJ)
    $(CC) $(CFLAGS) -o pvmd3$(EXESFX) $(OS2DOBJ) $(DOBJ) $(SOCKDOBJ) \
        $(LOPT) $(ARCHDLIB)
    touch pvmd3-os2

lib-os2: $(REGEXCONFOS2) $(LOBJ) $(OS2LOBJ) $(LPVMSOCK) $(SOCKLOBJ)
    rm -f $(LIBPVM).a
    $(AR) cr $(LIBPVM).a $(LOBJ) $(OS2LOBJ) $(LPVMSOCK) $(SOCKLOBJ)
    case x$(HASRANLIB) in xt ) echo ranlib; ranlib $(LIBPVM).a ;; esac
    touch lib-os2

#

clean: tidy
    rm -f pvmd3$(EXESFX) $(LIBPVM).a $(LIBPVM)s.a $(LIBPVM)pe.a
    rm -rf $(REGEXDIR)

tidy:
    rm -f $(DOBJ) $(SOCKDOBJ) $(SHMEMDOBJ) $(MPPDOBJ) \
        $(LOBJ) $(LPVMSOCK) $(SOCKLOBJ) $(LPVMSHMEM) $(SHMEMLOBJ) \
        $(LPVMMIMD) $(MPPLOBJ) $(REGEXDIR)/*.o \
        pvmd3-shm lib-shm pvmd3-mpp pvmd3-os2 lib-os2 \

```

```

        regexconfig regexconfig-os2

lint:    lintd lintl

lintd:
        lint -DARCHCLASS=\"$(PVM_ARCH)\" \
        ddpro.c host.c hoster.c imalloc.c pkt.c pmsg.c \
        pvmmalloc.c pvmcruft.c pvmd.c pvmdtev.c pvmmfrag.c pvmmlog.c \
        task.c tdpro.c waitc.c global.c > Ld

lintl:
        lint -I$(PVMDIR)/include \
        lpvm.c0 lpvmshmem.c lpvmmimd.c \
        imalloc.c lpvmcat.c lpvmgen.c lpvmpack.c lpvmglob.c \
        pvmmalloc.c pvmcruft.c pvmmerr.c pvmmfrag.c tev.c global.c > Ll

amem.o: $(SDIR)/amem.s
        $(AS) -o amem.o $(SDIR)/amem.s

ddpro.o:    $(SDIR)/ddpro.c
        $(CC) $(CFLAGS) -c $(SDIR)/ddpro.c

host.o: $(SDIR)/host.c
        $(CC) $(CFLAGS) -c $(SDIR)/host.c

vhoster.o:    $(SDIR)/hoster.c
        $(CC) $(CFLAGS) -c $(SDIR)/hoster.c

imalloc.o:    $(SDIR)/imalloc.c
        $(CC) $(CFLAGS) -c $(SDIR)/imalloc.c

lmsg.o: $(SDIR)/lmsg.c
        $(NODECC) $(CFLAGS) -DIMA_NODE -c $(SDIR)/lmsg.c

lpvm.o: $(SDIR)/lpvm.c
        $(CC) $(CFLAGS) -c $(SDIR)/lpvm.c

lpvmshmem.o:    $(SDIR)/lpvmshmem.c
        $(CC) $(CFLAGS) -c $(SDIR)/lpvmshmem.c $(PLOCKFILE)

lpvmmimd.o: $(SDIR)/lpvm.c

```

```
$(CC) $(CFLAGS) -DIMA_MPP -o lpvmmimd.o -c $(SDIR)/lpvm.c
lpvmmpp.o: $(SDIR)/lpvmmpp.c
$(NODECC) $(CFLAGS) -DIMA_MPP -c $(SDIR)/lpvmmpp.c
lpvmcat.o: $(SDIR)/lpvmcat.c
$(CC) $(CFLAGS) -c $(SDIR)/lpvmcat.c
lpvmgen.o: $(SDIR)/lpvmgen.c
$(CC) $(CFLAGS) -c $(SDIR)/lpvmgen.c
lpvmpack.o: $(SDIR)/lpvmpack.c
$(CC) $(CFLAGS) -c $(SDIR)/lpvmpack.c
lpvmglob.o: $(SDIR)/lpvmglob.c
$(CC) $(CFLAGS) -c $(SDIR)/lpvmglob.c
msgbox.o: $(SDIR)/msgbox.c
$(CC) $(CFLAGS) -c $(SDIR)/msgbox.c
mppchunkhost.o: $(SDIR)/mppchunk.c
$(CC) $(CFLAGS) -c -o mppchunkhost.o $(SDIR)/mppchunk.c
mppchunknode.o: $(SDIR)/mppchunk.c
$(NODECC) $(CFLAGS) -DIMA_NODE -c -o mppchunknode.o \
$(SDIR)/mppchunk.c
mppmsgghost.o: $(SDIR)/mppmsg.c
$(CC) $(CFLAGS) -c -o mppmsgghost.o $(SDIR)/mppmsg.c
mppmsgnode.o: $(SDIR)/mppmsg.c
$(NODECC) $(CFLAGS) -DIMA_NODE -c -o mppmsgnode.o $(SDIR)/mppmsg.c
nmdclass.o: $(SDIR)/nmdclass.c
$(CC) $(CFLAGS) -c $(SDIR)/nmdclass.c
pkt.o: $(SDIR)/pkt.c
$(CC) $(CFLAGS) -c $(SDIR)/pkt.c
pmsg.o: $(SDIR)/pmsg.c
$(CC) $(CFLAGS) -c $(SDIR)/pmsg.c
pvmmalloc.o: $(SDIR)/pvmmalloc.c
$(CC) $(CFLAGS) -c $(SDIR)/pvmmalloc.c
pvmcruft.o: $(SDIR)/pvmcruft.c
$(CC) $(CFLAGS) -c $(SDIR)/pvmcruft.c
pvmd.o: $(SDIR)/pvmd.c
```

```
$(CC) $(CFLAGS) -c $(SDIR)/pvmd.c
pvmdabuf.o: $(SDIR)/pvmdabuf.c
$(CC) $(CFLAGS) -c $(SDIR)/pvmdabuf.c
pvmdshmem.o: $(SDIR)/pvmdshmem.c
$(CC) $(CFLAGS) -c $(SDIR)/pvmdshmem.c $(PLOCKFILE)
pvmdmimd.o: pvmdmimd.c
$(CC) $(CFLAGS) -I.. -c pvmdmimd.c"
pvmdpack.o: $(SDIR)/pvmdpack.c
$(CC) $(CFLAGS) -c $(SDIR)/pvmdpack.c
pvmdunix.o: $(SDIR)/pvmdunix.c
$(CC) $(CFLAGS) -c $(SDIR)/pvmdunix.c
pvmmerr.o: $(SDIR)/pvmmerr.c
$(CC) $(CFLAGS) -c $(SDIR)/pvmmerr.c
pvmmfrag.o: $(SDIR)/pvmmfrag.c
$(CC) $(CFLAGS) -c $(SDIR)/pvmmfrag.c
pvmmlog.o: $(SDIR)/pvmmlog.c
$(CC) $(CFLAGS) -c $(SDIR)/pvmmlog.c
pvmmshmem.o: $(SDIR)/pvmmshmem.c
$(CC) $(CFLAGS) -c $(SDIR)/pvmmshmem.c $(PLOCKFILE)
pvmmpp.o: $(SDIR)/pvmmpp.c
$(CC) $(CFLAGS) -c $(SDIR)/pvmmpp.c
sdpro.o: $(SDIR)/sdpro.c
$(CC) $(CFLAGS) -c $(SDIR)/sdpro.c
task.o: $(SDIR)/task.c
$(CC) $(CFLAGS) -c $(SDIR)/task.c
tev.o: $(SDIR)/tev.c
$(CC) $(CFLAGS) -c $(SDIR)/tev.c
tdpro.o: $(SDIR)/tdpro.c
$(CC) $(CFLAGS) -c $(SDIR)/tdpro.c
waitc.o: $(SDIR)/waitc.c
$(CC) $(CFLAGS) -c $(SDIR)/waitc.c
pvmdtev.o: $(SDIR)/pvmdtev.c
$(CC) $(CFLAGS) -c $(SDIR)/pvmdtev.c
```

```

global.o: $(SDIR)/global.c
    $(CC) $(CFLAGS) -c $(SDIR)/global.c

deathapi.o: $(SDIR)/$(PVM_ARCH)/src/deathapi.c
    $(CC) $(CFLAGS) -c $(SDIR)/$(PVM_ARCH)/src/deathapi.c
stdlog.o: $(SDIR)/$(PVM_ARCH)/src/stdlog.c
    $(CC) $(CFLAGS) -c $(SDIR)/$(PVM_ARCH)/src/stdlog.c
sthoster.o: $(SDIR)/$(PVM_ARCH)/src/sthoster.c
    $(CC) $(CFLAGS) -c $(SDIR)/$(PVM_ARCH)/src/sthoster.c
rexec.o: $(SDIR)/$(PVM_ARCH)/src/rexec.c
    $(CC) $(CFLAGS) -c $(SDIR)/$(PVM_ARCH)/src/rexec.c
ruserpas.o: $(SDIR)/$(PVM_ARCH)/src/ruserpas.c
    $(CC) $(CFLAGS) -c $(SDIR)/$(PVM_ARCH)/src/ruserpas.c
os2spawn.o: $(SDIR)/$(PVM_ARCH)/src/os2spawn.c
    $(CC) $(CFLAGS) -c $(SDIR)/$(PVM_ARCH)/src/os2spawn.c

pvmregex.o: $(REGEXDIR)/pvmregex.o
    cp $(REGEXDIR)/pvmregex.o .
regex.o: $(REGEXDIR)/regex.o
    cp $(REGEXDIR)/regex.o .
$(REGEXDIR)/pvmregex.o: $(REGEXCP)
    cd $(REGEXDIR) ; $(MAKE) $(REGEXOPTS) pvmregex.o
$(REGEXDIR)/regex.o: $(REGEXCP)
    cd $(REGEXDIR) ; $(MAKE) $(REGEXOPTS) regex.o

regexconfig: $(REGEXDIR) $(REGEXCP) $(REGEXDIR)/Makefile
    @ touch regexconfig

regexconfig-os2: $(REGEXDIR) $(REGEXCP)
    cp $(SDIR)/$(PVM_ARCH)/Makefile.reg $(REGEXDIR)/Makefile
    @ touch regexconfig-os2

$(REGEXDIR)/Makefile:

```

```
    cd $(REXEXDIR) ; CC=$(CC) ./configure
$(RESEXCP): $(RESEXSRC)
    cp $(RESEXSRC) $(REXEXDIR)
$(REXEXDIR):
    @- mkdir $(REXEXDIR)

#
# Source File Dependencies
#

include $(PVMDEPPATH)$(SDIR)/pvmdep
```

Bibliografia

- [01] <http://www.linuxclube.com/colunas/13,06,2004.php>, acesso em 20 de março de 2004.
- [02] Pitanga, Marcos, *Construindo Supercomputadores com Linux*, Brasport, Rio de Janeiro , 2002.
- [03] <http://clustering.foundries.sourceforge.net/>, acesso em 26 de março de 2004.
- [04] <http://www.beowulf-underground.org/>, acesso em 26 de março de 2004.
- [05] <http://www.inf.aedb.br/download/biblio/sbc2000/eventos/semish/semi008.pdf>, acesso em 06 de maio de 2004.
- [06] Tanenbaum, Andrew S., *Sistemas Operacionais Modernos*, Makron Books, São Paulo , 2003.
- [07] <http://www.copyright.gov/>, acesso em 11 de abril de 2004.
- [08] <http://www.beowulf.org>, acesso em 05 de junho de 2004.
- [09] <http://www.vivaolinux.com.br/artigos/verArtigo.php?codigo=840>, acesso em 05 de junho de 2004.
- [10] <http://www.vivaolinux.com.br/artigos/verArtigo.php?codigo=733>, acesso em 03 de julho de 2004.
- [11] <http://www.beowulf.gr>, acesso em 22 de março de 2004.
- [12] http://www.csm.ornl.gov/pvm/pvm_home.html, acesso em 07 de agosto de 2004.
- [13] <http://www.csm.ornl.gov/pvm/man/manpages.html>, acesso em 20 de agosto de 2004.
- [14] <http://www.canonical.org/kragen/beowulf-faq.html>, acesso em 22 de agosto de 2004.
- [15] <http://www.netlib.org/pvm3/book/node1.html>, acesso em 22 de agosto de 2004.
- [16] <http://www-unix.mcs.anl.gov/mpi/>, acesso em 02 de setembro de 2004.

- [17] <http://www-unix.mcs.anl.gov/mpi/mpich2/>, acesso em 03 de setembro de 2004.
- [18] <http://www.cacr.caltech.edu/beowulf/>, acesso em 10 de setembro de 2004.
- [19] <http://es.tldp.org/Manuales-LuCAS/doc-cluster-computadoras/doc-cluster-computadoras-html/node59.html>, acesso em 22 de junho de 2004.
- [20] <http://www.cacr.caltech.edu/beowulf/tutorial/building.html>, acesso em 26 de agosto de 2004.
- [21] <http://www.ats.ucla.edu/at/clustering/default.htm>, acesso em 17 de maio de 2004.
- [22] <http://www.fysik.dtu.dk/CAMP/cluster-howto.html>, acesso em 17 de maio de 2004.
- [23] <http://www.clustermatic.org>, acesso em 07 de julho de 2004.
- [24] <http://www.povray.org/>, acesso em 10 de novembro de 2004.
- [25] <http://pvmpov.sourceforge.net/>, acesso em 02 de abril de 2004.
- [26] <http://www.verrall.demon.co.uk/mpipov/>, acesso em 04 de junho de 2004.
- [27] <http://www.linuxjournal.com/article.php?sid=5690>, acesso em 06 de junho de 2004.