SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito.....28.5.1999.....

Assinatura:.....*Marília Fudrader*.....

**DPWP – Uma nova Abordagem
– para o Escalonamento
Dinâmico em Computação
Paralela Virtual**

Aleteia Patrícia Favacho de Araújo

Orientação: Prof. Dr. Marcos José Santana

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo – ICMC/USP, como parte dos requisitos para a obtenção do título de Mestre em Ciências – Área de Ciências de Computação e Matemática Computacional.

**USP – São Carlos
Maio / 1999**

Dedicatória

*Aos meus pais, Aimê e José Bezerra,
Aos meus irmãos, Ádamo e Arethuza.
Por serem meu porto seguro...
Amo vocês!*

Agradecimentos

“Se tens um objectivo na vida, lute por ele, mesmo que tenhas que sofrer, pois mais vale uma lágrima no presente do que viver eternamente sofrendo porque nunca houveram lutas para ser feliz”.

Amigo é alguém muito especial, mesmo. Ele nos faz sorrir nos encoraja para sermos bem-sucedidos. Ele empresta um ouvido quando precisamos, ele compartilha uma palavra de elogio e sempre tem o coração aberto para nós. Ele divide com a gente aquilo que tem e se sente feliz com o que pode receber...Amigo é assim como vocês!

Agradeço a Deus pela benção, pelo grande amor que tem demonstrado ter por mim, e por ajudar-me a concretizar meu sonhos!

Mãe, todo mundo precisa de carinho, carinho puro. E o seu carinho é a forma mais bonita de transmitir o amor que tens por mim. Obrigada pela confiança, apoio, orações, palavras de estímulo e a sua eterna amizade, a qual foi fundamental para que eu conquistasse mais essa vitória!

Ao meu pai, por me ensinar a ser uma pessoa íntegra e batalhadora. Pai, ao senhor o meu muito obrigada!

Aos meus amados e queridos irmãos, pelo incentivo, força e acima de tudo pelos amigos que têm sido para mim. Agradeço a Deus a benção de tê-los como meus irmãos e de estarmos sempre juntos na luta pela felicidade de todos nós!

A todos meus familiares, em especial à minha vó Anna e a minha tia Madalena, pelo apoio e carinho!

Se você abre uma porta, você pode ou não entrar em uma nova sala. Você pode entrar e ficar observando a vida. Mas se você vence a dúvida, o temor, e entra, dá um grande passo....Ao Prof. Marcos José Santana e à Profa. Regina Helena C. Santana por terem sido as primeiras pessoas a abrirem a porta da USP para mim e acreditarem na minha capacidade!

Ao Paulo Sérgio Lopes de Souza, meu "co-orientador", que com extrema competência, sempre me mostrou pacientemente o “caminho das pedras”. Obrigada por toda a

orientação acadêmica e acima de tudo por me ensinar a “dizer para o meu problema o tamanho do meu Deus e não para o meu Deus o tamanho do meu problema”. À Simone e ao Felipe meu muito obrigada também.

Ao Cláudio Alex J. Rocha, pelo carinho, amizade, as palavras de estímulo, incentivo constante, apoio e enorme contribuição para a realização de mais uma vitória. De certo, tua participação foi imprescindível e, somente Deus pode recompensar-te por tudo que fizeste para mim!

Amigo é um irmão que agente escolhe...Agradeço ao meu irmão Paulo Sérgio Horst por compartilhar os tantos momentos felizes e tristes vividos em São Carlos. E lembre sempre, amigo é coisa para se guardar do lado esquerdo do peito, mesmo que o tempo e a distância digam não!

O dia mais importante não é quando conhecemos as pessoas, mas sim o momento em que elas passam a existir dentro de nós. Agradeço a Thereza P. P. Padilha e ao Gentil Veloso, pela amizade, carinho e partilha em todos os momentos!

Ao meu amigo Caldas pelo apoio no iniciar o mestrado e por me ouvir nos momentos mais difíceis. Amigo, esse dois anos de convívio serviram para ratificar a minha admiração por você. Obrigada por tudo...

À Andrezza Filizzola e ao Jorge Souza, pela amizade, incentivo, carinho, companheirismo, e apoio irrestrito e fundamental, sobretudo, na etapa final desta dissertação. Amigos, que Deus possa retribuir-lhes pela enorme contribuição!

Amigos são os que nas prosperidades aparecem sem receberem convite, e nas adversidades aparecem sem serem chamados. Obrigada a todos as amigas feitas em São Carlos, em especial: Selma, Silvio, Flávia, Mayb, Camilo, Maju, Jocélio, Marisa, Fabão, Alberto, Victor. E todos do *Engana Samba*, em especial ao Xande (Alexandre Barra), Topó e Ricardinho.

Aos amigos Fábio Melfi, Marco e Garga pela atenção e valiosa ajuda na linguagem C e na solução dos problemas encontrados no LINUX.

“As pessoas entram em nossa vida por acaso, mas não é por acaso que elas permanecem”. A todos os amigos de Belém que tanto torceram por essa vitória, em especial: Renato, Regiane, Lídia, Angela, Alexandre, Paulo André, Samira, Adalzira, Luciana, D. Francisca, Wilton, meu muito obrigada!

Ao pessoal do LASDPC, pela amizade, auxílio nos trabalhos e principalmente na dissertação: Adriana, Ana Elisa, Sarita, Célia, Silmara, Arion, Renata, Roberta, Edmilson,

Tatiana, Kalinka, Cláudia, Daniel, Boca. Em especial ao Márcio e ao Omar pela valiosa ajuda no término da dissertação.

A todos aqueles que constituem a família USP e propiciam “*educação para o Brasil*”. Em especial aos professores do ICMC, pela atenção e disposição com que sempre me trataram.

Às meninas da secretaria da Pós-Graduação Marília, Beth e Laura, pela disposição em ajudar, com grande vontade, em qualquer tipo de situação.

Aos funcionários do Instituto, em especial `a Adriana, ao Paulinho, ao Jacques (xerox), pela prestatividade e amizade.

Às meninas da biblioteca pela atenção, disposição e carinho com que sempre me trataram.

À FAPESP, pelo auxílio financeiro sem o qual este trabalho não seria possível.

1 INTRODUÇÃO	1
2 SISTEMAS DISTRIBUÍDOS E PARALELOS	5
2.1 CONSIDERAÇÕES INICIAIS	5
2.2 SISTEMAS DISTRIBUÍDOS	6
2.2.1 Características dos Sistemas Distribuídos	7
2.3 COMPUTAÇÃO PARALELA	10
2.3.1 Terminologia.....	11
2.3.2 Arquiteturas Paralelas.....	12
2.3.3 Programação Concorrente	17
2.3.4 Passagem de Mensagem	19
2.3.5 Ambientes de Passagem de Mensagem.....	20
2.4 PVM (PARALLEL VIRTUAL MACHINE).....	21
2.4.1 Componentes do PVM	23
2.4.2 PVM versão 3.4 e HARNESS	28
2.5 CONSIDERAÇÕES FINAIS	29
3 ESCALONAMENTO DE PROCESSOS.....	31
3.1 CONSIDERAÇÕES INICIAIS	31
3.2 CLASSIFICAÇÃO DOS ALGORITMOS	33
3.2.1 Classificação de Casavant	34
3.2.2 Classificação de Feitelson.....	38
3.2.3 Classificação de Lüling	38
3.3 COMPONENTES DE UM ALGORITMO.....	41
3.4 ÍNDICE DE CARGA.....	44
3.5 DESEMPENHO DE UM ESCALONADOR	46
3.6 CONSIDERAÇÕES FINAIS	48
4 AMIGO - UM AMBIENTE DE ESCALONAMENTO	49
4.1 CONSIDERAÇÕES INICIAIS	49
4.2 AMBIENTE DE ESCALONAMENTO (AMIGO)	50
4.2.1 A Estrutura do AMIGO.....	51
4.2.2 Utilização do Ambiente AMIGO	52
4.3 CAMADA SUPERIOR DO AMIGO	54
4.4 CAMADA INFERIOR DO AMIGO	59
4.4.1 Ambiente de Passagem de Mensagens.....	60
4.4.2 AMIGO Daemon (AMIGOD)	66
4.4.3 Políticas Ativas	72
4.5 CONSIDERAÇÕES FINAIS	73
5 POLÍTICA DE ESCALONAMENTO - DPWP	74
5.1 CONSIDERAÇÕES INICIAIS	74

5.2	POLÍTICA DE ESCALONAMENTO (DPWP).....	75
5.3	COMPONENTES DO ALGORITMO DPWP.....	78
5.4	CLASSIFICAÇÃO DO ALGORITMO DPWP	79
5.4.1	Classificação de Casavant	79
5.4.2	Classificação de Feitelson.....	81
5.4.3	Classificação de Lüling	81
5.5	PONTOS IMPORTANTES NO PROJETO	82
5.5.1	Índice de Cargas da DPWP.....	83
5.5.2	Heterogeneidade.....	85
5.5.3	Comunicação	86
5.6	DETALHES DE IMPLEMENTAÇÃO	87
5.6.1	Iniciação da DPWP	87
5.6.2	Manipulação da Tabela de Cargas	91
5.6.3	Iniciação do Escalonamento	93
5.6.4	Recebimento das Mensagens	95
5.6.5	Atualização Periódica da Tabela Local	96
5.7	CONSIDERAÇÕES FINAIS	96
6	AVALIAÇÃO DE DESEMPENHO DA DPWP.....	98
6.1	CONSIDERAÇÕES INICIAIS	98
6.2	PLATAFORMA UTILIZADA	99
6.3	APLICAÇÃO PARALELA UTILIZADA	101
6.4	ANÁLISE DOS RESULTADOS OBTIDOS	102
6.4.1	Execução da Multiplicação de Matrizes.....	103
6.4.2	Execução do Método do Trapézio.....	107
6.5	CONSIDERAÇÕES FINAIS	112
7	CONCLUSÕES	113
7.1	CONSIDERAÇÕES INICIAIS	113
7.2	ANÁLISE DA REVISÃO BIBLIOGRÁFICA	114
7.3	DISCUSSÃO SOBRE OS RESULTADOS OBTIDOS.....	115
7.4	CONTRIBUIÇÕES DESTES TRABALHOS	116
7.5	DIFICULDADES ENCONTRADAS	117
7.6	TRABALHOS FUTUROS	118
7.7	CONSIDERAÇÕES FINAIS	118
	REFERÊNCIAS BIBLIOGRÁFICAS	120
	APÊNDICE A - Interface do AMIGOD	129
	APÊNDICE B - Especificação das Mensagens.....	136
	APÊNDICE C - Interface da DPWP	141
	APÊNDICE D - Aplicações Paralelas Usadas nos Testes.....	147

Lista de Figuras

Figura 2.1 - Exemplo de Arquitetura SISD	13
Figura 2.2 - Exemplo de Arquitetura SIMD	14
Figura 2.3 - Exemplo de Arquitetura MISD	14
Figura 2.4 - Exemplo de Arquitetura MIMD	15
Figura 2.5 - Classificação de Duncan	16
Figura 2.6 - Modelo de Passagem de Mensagem Genérico	20
Figura 2.7 - Exemplo de Implementação PVM	24
Figura 2.8 - Identificador de Tarefas no PVM	26
Figura 3.1 - Classificação Hierárquica proposta por Casavant	35
Figura 3.2 - Classificação de Lüling	39
Figura 4.1 - Estrutura Interna do AMIGO	52
Figura 4.2 - Arquivo de Configuração dos Benchmarks	56
Figura 4.3 - Camada superior do AMIGO - escolha das Políticas de Escalonamento	58
Figura 4.4 - Implementações da Camada Inferior do AMIGOD	60
Figura 4.5 - Preparação dos hosts através da função <code>pvm_spawn()</code>	64
Figura 4.6 - Preparação dos hosts através da função <code>pvm_spawn</code> alterada()	65
Figura 4.7 - Relacionamento entre os módulos do AMIGO	67
Figura 4.8 - Cabeçalho das Mensagens trocadas no AMIGO	69
Figura 4.9 - Tabela de hosts do AMIGOD	70
Figura 4.10- Tabela de Políticas do AMIGOD	71
Figura 5.1 - Transferência do estado de uma máquina para outra	77
Figura 5.2 - Classificação de Casavant segundo a DPWP	80
Figura 5.3 - Classificação de Lüling adaptada para DPWP	82
Figura 5.4 - Troca de mensagem entre Políticas DPWP	87
Figura 5.5 - Passos a serem realizados pela DPWP	88
Figura 5.6 - Configuração do Sistema	90
Figura 5.7 - Tabela de Cargas da DPWP	91
Figura 5.8 - Passos Realizados na Atualização da Tabela de Cargas	92
Figura 5.9 - Passos Realizados no Escalonamento	94
Figura 6.1 - Ambiente do LASDPC usado nos testes	99
Figura D.1 - Paralelização do algoritmo do trapézio composto	148
Figura D.2 - Paralelização do algoritmo de multiplicação de matrizes	154

Lista de Tabelas

Tabela 4.1	- Relação de flags usados na função <code>pvm_spawn()</code>	64
Tabela 5.1	- Características do índice de Carga.	85
Tabela 6.1	- Configuração das máquinas usadas no LASDPC	100
Tabela 6.2	- Resultados com o PVM sem balanceamento (multiplicação de Matrizes).	103
Tabela 6.3	- Resultado com o PVM balanceado (multiplicação de matrizes)	104
Tabela 6.4	- Comparação entre os PVMs em um sist. carregado (multi. de matrizes).....	105
Tabela 6.5	- Comparação entre os PVMs em um sistema ocioso (multi. de matrizes).	106
Tabela 6.6	- Resultados com o PVM sem balanceamento (método do trapézio).....	107
Tabela 6.7	- Resultado com o PVM balanceado (método do trapézio).	108
Tabela 6.8	- Comparação entre os PVMs em um sistema carregado (mét. do trapézio).....	109
Tabela 6.9	- Comparação entre os PVMs em um sistema ocioso (método do trapézio).	110
Tabela 6.10	- Comparação entre os PVMs e a heterogeneidade.....	111

Lista de Gráficos

Gráfico 6.1 - PVM sem balanceamento (multiplicação de Matrizes).....	103
Gráfico 6.2 - PVM balanceado (multiplicação de matrizes)	104
Gráfico 6.3 - Um sistema carregado - multiplicação de matrizes.....	105
Gráfico 6.4 - Um sistema ocioso - multiplicação de matrizes.....	106
Gráfico 6.5 - PVM sem balanceamento (método do trapézio).....	107
Gráfico 6.6 - PVM balanceado (método do trapézio).....	108
Gráfico 6.7 - Um sistema carregado – método do trapézio.....	109
Gráfico 6.8 - Um sistema ocioso – método do trapézio.....	110
Gráfico 6.9 - Comparação entre os PVM's com e sem heterogeneidade.....	111

A computação distribuída sobre redes de estações de trabalho tem sido adotada como uma plataforma alternativa para a execução de aplicações paralelas. Essas redes não apresentam apenas uma boa relação custo/benefício, mas também fornecem um ambiente computacional de propósito geral, o qual pode ser usado tanto por aplicações paralelas quanto por aplicações não paralelas.

Nesses ambientes multiusuários ocorre uma grande variação da carga de trabalho manipulada por cada usuário trazendo sérios problemas para o desempenho global do sistema.

Dentro desse contexto, este trabalho descreve detalhadamente a implementação da DPWP, um algoritmo de escalonamento cuja principal finalidade é implantar balanceamento de cargas sobre uma rede de estações de trabalho heterogênea, para aplicações paralelas com grande quantidade de processamento.

Os estudos preliminares realizados sobre os testes desenvolvidos serviram, acima de tudo, para validar e avaliar o desempenho da DPWP. Os resultados obtidos demonstram que o algoritmo tem um comportamento seguro e que as aplicações paralelas executadas, tendo a DPWP como algoritmo de escalonamento, apresentaram um excelente resultado, levando a um aumento considerável em seu desempenho final.

Distributed computing on workstation networks has been considered as an alternative platform for parallel applications. These workstation networks are not only cheaper, but also provide a general-purpose computing environment that is typically shared by both parallel and non-parallel application developers and users.

Such a computing environment, where the use of resources varies as applications consume and release resources, brings important scheduling problems for global performance system.

Thus, this work presents the implementation of the DPWP, a process scheduling policy aiming at balancing the computing load on a network of heterogeneous workstations, for parallel computing applications, that have more processing and less communication.

Preliminary studies were developed in order to validate and to evaluate the performance reached by using the DPWP. The results showed that the scheduling policy behaves stable and the parallel applications reached excellent overall performance.

Introdução

Este capítulo apresenta o cenário global onde se enquadram os sistemas distribuídos e a computação paralela, enfatiza os objetivos desta dissertação e fornece a descrição de cada capítulo, dando uma visão geral de como o trabalho está organizado.

A computação ocupa posição de destaque no cenário global, resultante de uma grande evolução em um curto espaço de tempo. Essa evolução é impulsionada pelas necessidades provenientes da sociedade, condicionada às facilidades de utilização dos computadores e por um gradual aumento do leque de aplicações que podem ser computacionalmente resolvidas.

Essa transformação começou a ser moldada a partir da década de 70, quando veio à tona os sistemas computacionais distribuídos, que surgiram como uma evolução natural para substituir os sistemas centralizados (os tradicionais sistemas de compartilhamento de tempo). O avanço registrado na microeletrônica permitiu a integração em larga escala, contribuindo para o aparecimento dos primeiros microprocessadores, e o aprimoramento das redes de interconexão as quais, permitem a transmissão rápida e segura de milhões de bits por segundo.

Os computadores pessoais e as estações de trabalho revolucionaram a computação de tal forma que os objetivos iniciais dos sistemas distribuídos, de compartilhamento de recursos e informações, hoje são eminentemente uma realidade do dia a dia.

Dessa maneira, novos objetivos, novas aplicações e novos desafios surgem, principalmente no que concerne à busca de melhor desempenho e melhor confiabilidade. Isso tudo em consequência da evolução sem precedentes em que o planeta está envolvido.

A computação paralela é uma área de pesquisa que evoluiu de modo distinto aos sistemas distribuídos, mas de maneira paralela aos grandes avanços do progresso tecnológico. A

computação paralela sempre almejou em seus projetos a redução do tempo de processamento de algumas classes de programas aplicativos. Dentro da comunidade acadêmica (Kirner, 1991; Almasi & Gottlieb, 1994) a computação paralela e as arquiteturas paralelas sempre foram assuntos conjugados, isto é, computação paralela era realizada sobre arquiteturas paralelas.

Na década de 80, motivados pelo alto custo de aquisição e manutenção de máquinas paralelas, assim como a dependência estabelecida entre o usuário e o fabricante da máquina, a comunidade envolvida com pesquisas em sistemas distribuídos e computação paralela propuseram a utilização de sistemas distribuídos como plataformas de execução paralela. Isso possibilitou simular as arquiteturas paralelas do tipo MIMD, permitindo uma melhor relação custo/benefício para a computação paralela, pois fornece menor custo de implantação e maior potência computacional (em relação as arquiteturas sequenciais), a uma grande quantidade de aplicações (Beguelin et. al., 1994; Dongarra & Dunigan, 1997).

Nesse contexto, a convergência dessas duas áreas de pesquisa tem como marco a criação das chamadas Máquinas Paralelas Virtuais, que oferecem benefícios para as duas áreas. Entretanto, junto às benesses advindas com essa combinação, também veio uma gama de problemas peculiares à nova tecnologia. Por exemplo, o problema encontrado em sistemas distribuídos com a velocidade das redes de comunicação é similar ao problema encontrado nas arquiteturas paralelas com os barramentos de comunicação. Da mesma forma, os problemas com escalonamento de processos e suas implicações no desempenho final do sistema também ocorrem em ambas as áreas (Feitelson et. al., 1997; Wang et. al., 1998).

O problema de escalonamento de processos é particularmente importante em sistemas distribuídos com acoplamento fraco¹, uma vez que se tratando de ambientes multiusuários, a carga de trabalho manipulada por cada pessoa pode ser diferente e, dessa forma, a chegada aleatória de processos (tarefas) que requeiram recursos distintos, pode ocasionar um desequilíbrio no sistema.

Dessa maneira, o balanceamento de carga através de um algoritmo de escalonamento, tem como objetivo propiciar uma distribuição de carga na qual o equilíbrio do

¹ Nesse sistema cada máquina tem sua memória particular, isto é, seu próprio espaço de endereçamento (Tanenbaum, 1995).

sistema deve ser a meta principal. Com isso a tendência é que cada máquina receba carga proporcional ao seu estado e capacidade.

Por outro lado, a estratégia de balanceamento de carga adotada pode, em algumas situações, influenciar decisivamente no comportamento do desempenho final do sistema, devido às trocas de mensagens necessárias para transferir processos ou dados de uma máquina para outra, em busca de melhor balanceamento.

Diante do exposto, se faz necessário pesquisar métodos e/ou estratégias que proporcionem uma otimização do sistema e, conseqüentemente, uma melhor utilização da capacidade de processamento instalada (Gibbons, 1997; Togneri, 1998).

Este trabalho visa apresentar um novo algoritmo para o problema do balanceamento de cargas. A DPWP (**D**ynamic **P**olicy **W**ithout **P**reemption) é um algoritmo de escalonamento de processos construído com o intuito de melhorar a atividade de escalonamento em ambientes virtuais paralelos.

A DPWP considera em seu projeto importantes questões abordadas pela literatura para que se atinja o sucesso esperado. Dentre esses pontos destacam-se o fato de que um bom algoritmo de escalonamento deve traçar metas de escalonamento e conseqüentemente atuar sobre essas, ao invés de tentar ser um algoritmo genérico, capaz de escalonar todos os tipos de aplicações.

Em princípio, a DPWP foi projetada para atuar no escalonamento de aplicações paralelas que caracterizam-se por usar muito processamento e pouca comunicação. Diante disso, todos os pontos relevantes no projeto da DPWP foram traçados tendo isso como meta.

Além disso, a DPWP foi projetada para dar suporte ao ambiente AMIGO (**D**yn**A**Mical **F**lex**I**ble **S**chedul**I**n**G** **E**nvir**O**nment). Esse ambiente foi proposto na tentativa de diminuir a lacuna existente entre a teoria e prática nessa área (Souza et. al., 1999a).

Após o capítulo introdutório (capítulo 1), o capítulo 2 aborda diversos tópicos envolvidos na área de sistemas distribuídos e computação paralela. São apresentadas algumas definições encontradas na literatura especializada (Müllender, 1993; Coulouris et. al., 1994; Tanenbaum, 1995), assim como a caracterização dos sistemas distribuídos (Coulouris et. al., 1988). Em seguida, são abordados os principais tópicos envolvidos na computação paralela,

assim como, as classificações de arquiteturas paralelas apresentadas por Flynn (Flynn et. al., 1972) e mais recentemente por Duncan (Duncan, 1990).

No capítulo seguinte (capítulo 3), um estudo sobre escalonamento de processos é enfocado. Inicialmente é dada uma visão geral sobre o assunto, seguida das classificações apresentada por alguns autores. Na sequência do capítulo são abordados os componentes de um algoritmo e a importância do índice de carga (Kunz, 1991; Xu & Lau, 1997). O capítulo é concluído com a discussão dos pontos relevantes ao desempenho de um escalonador.

O capítulo 4 é responsável em apresentar o ambiente de escalonamento AMIGO. O estudo inicia com uma abordagem geral sobre o assunto, apresenta a estrutura do AMIGO e as duas maneiras de acesso a esse ambiente. Em seguida são abordadas a camada superior e a camada inferior do ambiente AMIGO. A camada inferior é discutida em detalhes (aspectos de implementação) no final desse capítulo.

Pela importância ímpar a este trabalho, um capítulo completo (capítulo 5) é destinado à DPWP. O capítulo inicia mostrando as idéias e objetivos dessa nova proposta e, na sequência, são apresentados os componentes e a classificação da mesma. Esse capítulo provê ainda questões consideradas importantes no projeto da DPWP, assim como os detalhes de implementação.

Na sequência do trabalho, o capítulo 6 aborda os resultados obtidos na execução da política aqui proposta sobre o ambiente AMIGO e a plataforma de passagem de mensagem PVM. Os dados obtidos são analisados e discutidos a fim de validar a DPWP.

Concluindo, o capítulo 7 apresenta uma análise final sobre o trabalho realizado, identificando-se as contribuições, as principais dificuldades encontradas e as sugestões para trabalhos futuros.

Sistemas Distribuídos e Paralelos

O presente capítulo aborda os diversos tópicos envolvidos na área de sistemas distribuídos, computação paralela e programação concorrente. Primeiramente, é apresentada uma visão geral sobre sistemas distribuídos e suas características essenciais. Em seguida, são abordadas algumas definições relevantes em computação paralela e a classificação das arquiteturas. Para finalizar, são discutidas as particularidades do ambiente PVM.

2.1 Considerações Iniciais

Os sistemas computacionais distribuídos e a computação paralela constituem duas áreas fundamentais dentro das ciências de computação que, juntas, proporcionam uma alternativa viável na busca da solução para se ter alto desempenho, a um custo relativamente baixo.

Em princípio, o sucesso dos sistemas computacionais distribuídos foi moldado graças ao avanço tecnológico dos microprocessadores e das redes de comunicação, os quais associados, facilitaram a construção de sistemas computacionais compostos por um grande número de processadores, ligados através de redes de alta velocidade.

Entretanto, a característica primordial de um sistema computacional distribuído está relacionado à sua capacidade de compartilhar recursos de hardware e software, de maneira transparente, através de um conjunto de computadores autônomos, interligados por uma rede de comunicação e apoiados por um sistema operacional distribuído (Müllender, 1993; Coulouris et. al., 1994; Tanenbaum, 1995).

Diferentemente dos sistemas distribuídos, a computação paralela surgiu objetivando a criação de novos modelos computacionais capazes de suprir os requisitos das aplicações que não podem ser satisfatoriamente executadas no Modelo de von Neumann. Dessa maneira, através de um conjunto de processadores que cooperam e comunicam-se entre si, grandes problemas são resolvidos mais rapidamente do que se estivessem sendo solucionados por computadores seqüenciais (Zaluska, 1991; Almasi & Gottlieb, 1994).

Apesar dos sistemas distribuídos e da computação paralela terem evoluído por razões distintas, ambos são portadores de características e problemas comuns (como por exemplo concorrência e balanceamento de cargas), capazes de induzir a convergência dessas duas filosofias (Zaluska, 1991).

Nesse contexto, a convergência deve-se principalmente ao avanço tecnológico da computação paralela, ocasionado pelos projetos bem sucedidos de processadores maciçamente paralelos (*Massively Parallel Processors* - MPP) e pelo desenvolvimento da computação distribuída (Zaluska, 1991; Beguelin et. al., 1994). A idéia é fazer com que os sistemas distribuídos possam ser utilizados como plataformas de memória distribuída para a execução de programas paralelos.

Essa idéia foi viabilizada a partir do desenvolvimento de ferramentas de software, conhecidas como ambientes de passagem de mensagem, que permitem a utilização do conceito de **Máquinas Paralelas Virtuais**.

Os ambientes de passagem de mensagem foram projetados originalmente para sistemas multiprocessadores, mas outros ambientes, como o PVM (*Parallel Virtual Machine*) (Geist et. al., 1994), foram desenvolvidos especialmente para atuarem sobre redes de estações de trabalho e microcomputadores.

2.2 Sistemas Distribuídos

Os sistemas computacionais distribuídos (ou simplesmente sistemas distribuídos) originaram-se no início da década de 70, após o surgimento de tecnologias de redes de computadores de maior desempenho e maior confiabilidade (Coulouris et. al., 1994). Com o constante melhoramento tecnológico das redes de comunicação (ATM e B-ISDN), assim como o

crescente aumento da qualidade e o barateamento do hardware, a tendência previsível é que a utilização de sistemas distribuídos aumentará significativamente e tornar-se-á uma das formas mais eficazes e dominantes de computação nos próximos anos.

Há na literatura especializada várias definições, mas nenhuma estabelece exatamente o conceito de sistemas distribuídos (Tanenbaum, 1995). Entretanto, a maioria delas é concordante no que se refere à idéia central de um sistema distribuído.

Coulouris (Coulouris et. al., 1994), descreve um sistema distribuído como sendo um conjunto de computadores autônomos interligados por uma rede de comunicação e equipados com um sistema operacional distribuído. O sistema operacional distribuído é o responsável por coordenar as atividades desenvolvidas e compartilhar os recursos do sistema (como hardware, software e dados).

Usuários de um sistema distribuído bem projetado devem ter a impressão de que eles estão utilizando um sistema computacional único e integrado, embora esse sistema esteja implementado em vários computadores e em diferentes locais (Coulouris et. al., 1994).

De maneira geral, os autores diferem no conjunto de características essenciais que um sistema deve possuir para que o mesmo seja considerado um sistema distribuído real. Müllender (Müllender, 1993) afirma que tentar definir exatamente o que é um sistema distribuído pode ser perigoso, pois alguma característica importante (ou que venha a ser considerada importante no futuro) pode ser esquecida.

Contudo, Coulouris (Coulouris et. al., 1994) propõe seis características, as quais têm sido bastante utilizadas na caracterização de um sistema distribuído. Essas características são: compartilhamento de recursos, abertura, concorrência, escalabilidade, tolerância a falhas e transparência.

2.2.1 Características dos Sistemas Distribuídos

Para prover suporte ao projeto de um sistema distribuído, se faz necessário enquadrá-lo nas características essenciais, responsáveis por sua funcionalidade e potência computacional.

A adoção de todas as características chaves dos sistemas distribuídos nem sempre é uma tarefa fácil. Alguns aspectos devem ser levados em consideração, quando se deseja projetar um sistema distribuído, pois algumas características são fundamentais a todo sistema distribuído bem projetado, além disso, há características que concorrem entre si.

Dessa maneira, Coulouris definiu as seguintes características: compartilhamento de recursos, abertura, concorrência, escalabilidade, tolerância a falhas e transparência.

- ❖ **Compartilhamento de recursos** é uma das características mais importantes na funcionalidade de um sistema distribuído, pois está relacionada à necessidade de compartilhar componentes de software, hardware e dados, a fim de aumentar a potência computacional do sistema, pois há um número elevado de casos em que compartilhar recursos é, além de interessante, fundamental.
- ❖ **Abertura** é a característica que determina com que facilidade um sistema pode ser expandido, sem que haja degeneração dos componentes já implantados. Um sistema pode ser aberto em relação a uma característica e fechado em relação a outra. Aberto no nível de hardware, significa que pode-se mudar, por exemplo, periféricos, memória e interface de comunicação, sem criar nenhum problema no sistema. Conseqüentemente, um sistema aberto no nível de software, traduz a possibilidade de incluir novas características do sistema operacional, protocolos de comunicação e serviços de compartilhamento de recursos.
- ❖ **Concorrência** é uma característica que transcende os sistemas distribuídos, pois pode-se obter concorrência a partir de um único processador, intercalando-se a execução de diversos processos, através da atribuição de parcelas de tempo de processamento a cada um deles. Dessa forma, a existência de concorrência e paralelismo em sistemas distribuídos é uma conseqüência natural, decorrente dos vários elementos de processamento, da independência dos recursos e da localização dos processos servidores em computadores separados (Coulouris et. al., 1994).
- ❖ **Escalabilidade** é a característica que permite um sistema distribuído operar com eficiência independente do seu tamanho. O tamanho de um sistema distribuído pode variar de um pequeno conjunto de computadores (por exemplo, um par), até centenas de computadores interligados através de uma ou mais redes. O objetivo é fazer com que tanto o sistema quanto

o software de aplicação, não precisam ser alterados quando o tamanho (escala) da rede aumenta (Coulouris et. al., 1994).

❖ **Tolerância a Falhas**, como afirmou Leslie Lamport (Müllender, 1993), os sistemas distribuídos são aqueles em que não se pode fazer um trabalho, porquê uma máquina que nem se conhece, não se sabe onde está, e nunca ouviu-se falar a respeito, encontra-se fora do ar. Apesar dessa afirmação ser muito forte, ela deve ser considerada, pois as falhas são imprevisíveis. Sendo assim, o projeto de sistemas tolerantes às falhas se processa sobre duas abordagens:

- **Redundância de Hardware:** devem-se utilizar componentes redundantes, isto é, aqueles componentes cuja queda venha provocar a queda de todo o sistema, devem ser replicados.
- **Redundância de Software:** deve-se projetar programas que sejam recuperados a partir do ponto de falha, isto é, um sistema computacional tolerante as falhas deve possibilitar que um programa após ter falhado possa retornar ao estado consistente anterior à falha.

❖ **Transparência** é a característica mais importante e cobiçada em um sistema distribuído, pois trata-se da maneira de como obter a imagem única do sistema (Tanenbaum, 1992). Em outras palavras, a transparência em um sistema distribuído, é a capacidade de ocultar do usuário e do programador de aplicação a separação de seus componentes, de forma que percebe-se o sistema como um todo e não como um conjunto de elementos independentes (Coulouris et. al., 1994).

A transparência pode ser obtida em diversos níveis. A maneira mais simples de obtê-la é escondendo dos usuários o fato deles estarem tratando com um sistema distribuído. Em um nível mais baixo, é possível também fazer com que o sistema pareça transparente para os programas, embora isto não seja muito simples (Tanenbaum, 1995).

No Manual de Referência ANSA (The Advanced Network Systems Architecture Reference Manual) e no Modelo de Referência ISO (*International Standards Organization*) para Processamento Distribuído Aberto, identificam-se oito formas de transparência (Coulouris et. al., 1994): transparência de acesso, transparência de localização, transparência de concorrência,

transparência de replicação, transparência a falhas, transparência de migração, transparência de desempenho e transparência de escala.

2.3 Computação Paralela

A computação paralela (ou processamento paralelo) segundo Almasi e Gottlieb (Almasi & Gottlieb, 1994), consiste basicamente em um conjunto de elementos de processamento que cooperam e comunicam-se entre si, para solucionar grandes problemas, de maneira mais rápida do que se estivessem sendo solucionados sequencialmente.

Para Quinn (Quinn, 1994), computação paralela é uma maneira de processamento de informação que enfatiza a manipulação concorrente de elementos de dados, capazes de solucionar um ou mais processos de um único problema.

O ímpeto do surgimento da computação paralela foi, portanto, a capacidade de aumentar o processamento em uma única máquina. Os problemas dos usuários aumentam a taxas crescentes, enquanto as máquinas sequenciais apresentam limitações tecnológicas para evoluir na velocidade necessária.

Além disso, a computação paralela ratificou-se no mercado devido ao seu desempenho na solução de problemas muito grandes, sua tolerância às falhas e sua capacidade de resolver naturalmente as aplicações inerentemente paralelas.

A computação paralela, entretanto, apresenta inúmeras características não encontradas na computação sequencial, como por exemplo, várias maneiras de se organizar os processadores, necessidade de um gerenciamento complexo dos elementos de processamento em relação aos sequenciais e manutenção da coerência da informação que trafega pela máquina. Dessa maneira, se faz necessário apresentar a terminologia particular usada em computação paralela.

2.3.1 Terminologia

Esta seção apresenta algumas definições necessárias ao entendimento da computação paralela e programação concorrente

- ❖ **Concorrência:** A concorrência acontece quando, em um determinado instante, dois ou mais processos começaram a sua execução, mas não terminaram (Almasi & Gottlieb, 1994). Por essa definição, concorrência pode ocorrer tanto em sistemas com um único processador, quanto em sistemas com múltiplos processadores.
- ❖ **Paralelismo (ou paralelismo físico):** Ocorre quando há mais de um processo sendo executado no mesmo intervalo de tempo e em sistemas com multiprocessadores. Na literatura usa-se ainda o termo pseudo-paralelismo (ou paralelismo lógico), quando vários processos são executados em um único processador.
- ❖ **Granulação (ou granulosidade):** A granulação diz respeito ao tamanho das unidades de trabalho submetidas aos processadores (Kirner, 1991; Almasi & Gottlieb, 1994). De maneira geral, divide-se a granulação em fina, média e grossa. Uma granulação fina indica que o paralelismo é feito em nível de operações. Na granulação média o paralelismo é atingido entre os blocos ou subrotinas do programa e, em uma granulação grossa são paralelizados os processos (Kirner, 1991).
- ❖ **Speedup:** É uma medida utilizada para calcular o quanto um programa paralelo utilizando p processadores é mais rápido do que sua execução seqüencial. Isso é representado através da fórmula abaixo, onde, T_{seq} é o tempo de execução seqüencial e T_{par} representa o tempo de execução em paralelo (Hwang, 1993). Pela fórmula observa-se que $Sp = p$ representa a situação ideal em termos de ganho. Porém, essa situação dificilmente é alcançada pois outros fatores influenciam nos cálculos, tais como: sobrecarga na comunicação, balanceamento de carga inadequado, algoritmos com um nível pequeno de paralelismo e granulação inadequada.

$$Sp = T_{seq} / T_{par}$$

- ❖ **Eficiência:** É outro conceito eminentemente importante para a computação paralela, pois é usada para quantificar a utilização do processador e é calculada através do *speedup* (Sp) e do número de processadores (p) (Quinn, 1994). Dessa forma, a máxima eficiência do processador é obtida por $Ep = 1$, que significa que sua capacidade foi totalmente aproveitada (100%). Entretanto, os valores dificilmente atingem 1 (um) e quando ocorre anomalia de *speedup* também ocorre uma exceção onde os valores de Ep serão maiores do que um (Foster, 1995).

$$Ep = Sp / p$$

- ❖ **Pipeline (ou paralelismo temporal):** Implica na execução de eventos sobrepostos no tempo. Uma determinada tarefa é dividida em uma sequência de subtarefas, sendo que cada uma delas é executada por um estágio especializado de hardware e software, que opera concorrentemente com os outros estágios do *pipeline* (Quinn, 1994).
- ❖ **Arquitetura Paralela:** A arquitetura paralela fornece uma estrutura de alto nível para o desenvolvimento de soluções através do processamento paralelo, adotando múltiplos processadores, simples ou complexos, que cooperam entre si para solucionar problemas através da execução concorrente.

Além de uma terminologia própria, a computação paralela apresenta alguns pontos essenciais em sua natureza, tais como, arquiteturas paralelas, programação concorrente, passagem de mensagem e ambientes de passagem de mensagem.

2.3.2 Arquiteturas Paralelas

Na presença do rápido desenvolvimento da computação paralela, se fez necessário produzir vários modelos de arquiteturas paralelas. Existem inúmeras maneiras de se organizar esses modelos. Para que se possa visualizar melhor todo o conjunto de possíveis opções de arquiteturas paralelas, é importante classificá-las, através de uma abordagem simples, hierárquica, sistemática e abrangente.

Na literatura há duas classificações que se destacam, a classificação de Flynn (Flynn, 1972, 1996), a qual baseia-se no fluxo de instruções e no fluxo de dados e a classificação de Duncan (Duncan, 1990), sugerida com o objetivo de agrupar as novas arquiteturas que não eram mais alcançadas pela classificação de Flynn.

Classificação de Flynn

Para Flynn o processo computacional deve ser entendido como uma relação entre fluxos de instruções e fluxos de dados. Desta maneira, as quatro formas de organizar uma arquitetura de computadores são (Flynn, 1972, 1996):

- ❖ **SISD (Single Instruction Single Data Stream):** Essa arquitetura está relacionada aos modelos que possuem apenas um fluxo de instruções e um fluxo de dados (veja Figura 2.1). As arquiteturas SISD representam as máquinas de von Neumann, pois as instruções são todas executadas sequencialmente.

Figura 2.1 - Exemplo de Arquitetura SISD

- ❖ **SIMD (Single Instruction Multiple Data Stream):** Diz respeito às máquinas com um único fluxo de instruções atuando sobre vários fluxos de dados. Nas arquiteturas SIMD há várias unidades de processamento e uma única unidade de controle (veja Figura 2.2).

Figura 2.2 - Exemplo de Arquitetura SIMD

- ❖ **MISD (Multiple Instruction Single Data Stream):** Essa arquitetura seria formada por múltiplos fluxos de instruções agindo sobre o mesmo fluxo de dados, isto é, o resultado de uma unidade de processamento seria a entrada para a próxima unidade, dando a idéia de um enorme *pipeline* (veja Figura 2.3).

Figura 2.3 - Exemplo de Arquitetura MISD

- ❖ **MIMD (Multiple Instruction Multiple Data Stream):** A classe MIMD é composta por inúmeros fluxos de instruções e fluxos de dados (veja Figura 2.4). Cada unidade de processamento atua sobre um conjunto de dados diferentes e possui uma unidade de controle para cada processador.

Figura 2.4 - Exemplo de Arquitetura MIMD

Classificação de Duncan

A classificação de Duncan procura manter e ampliar a classificação de Flynn, permitindo identificar as arquiteturas desenvolvidas posteriormente ao seu trabalho. Duncan considera apenas as arquiteturas paralelas, excluindo as arquiteturas que apresentam apenas mecanismos de paralelismo de baixo nível, já que essas técnicas são de domínio na indústria de hardware (Duncan, 1990).

A classificação completa de Duncan é apresentada na Figura 2.5, e todos os itens e suas respectivas subdivisões são posteriormente comentadas.

Figura 2.5 - Classificação de Duncan

Arquiteturas Síncronas

Essas arquiteturas coordenam suas operações concorrentes sincronamente em todos os processadores, através de relógios globais, unidades de controle únicas ou controladores de unidades vetoriais; enquadram-se nessa categoria (Duncan, 1990):

❖ **Processadores Vetoriais:** os processadores vetoriais caracterizam-se pela existência de um hardware específico para a execução de operações em vetores.

❖ **Arquiteturas SIMD:** já descritas anteriormente, são caracterizadas por apresentar múltiplos processadores, sobre a supervisão de uma unidade central de controle, que executam a mesma instrução sincronamente em conjuntos de dados distintos.

❖ **Arquiteturas Sistólicas:** essas arquiteturas se caracterizam pela presença de vários processadores enfileirados (*pipeline*), que formam uma cadeia na qual apenas os processadores localizados nos limites dessa estrutura possuem comunicação com a memória.

Arquiteturas Assíncronas

Essas arquiteturas, ao contrário das arquiteturas síncronas, caracterizam-se pelo controle descentralizado do hardware, de maneira que os processadores são independentes entre si. As arquiteturas assíncronas são compostas basicamente de máquinas MIMD que, por sua vez, podem ser convencionais ou não-convencionais (Duncan, 1990):

❖ **Arquiteturas MIMD:** esse tipo de organização pressupõe algoritmos de granulação mais grossa e com pouca comunicação entre processos. A organização de memória determina o sincronismo e a forma de comunicação entre os processadores. Essa arquitetura contempla máquinas com memória centralizada e memória distribuída. A primeira caracteriza-se pela existência de uma memória global e única, a qual é utilizada por todos os processadores, de maneira que através do compartilhamento de posições desta memória ocorre a comunicação entre os processos. A segunda caracteriza-se por cada processador possuir sua própria memória local, dessa forma, os processos comunicam-se através da troca de mensagem.

❖ **Paradigma MIMD:** essa classe engloba as arquiteturas assíncronas que, apesar de serem consideradas MIMD, são organizadas segundo conceitos tão fundamentais a seu projeto que torna-se difícil classificá-las apenas como MIMD. Por isso, tais arquiteturas denominam-se paradigmas arquiteturais MIMD. Essas arquiteturas são divididas em: arquiteturas MIMD/SIMD, Dataflow, Redução e Frente de Onda.

2.3.3 Programação Concorrente

A programação concorrente mostra-se um ponto inerente no apoio à construção de programas paralelos, de maneira a obter melhor desempenho e melhor utilização do hardware paralelo disponível.

Um programa concorrente especifica dois ou mais programas sequenciais que podem ser executados concorrentemente, explorando a concorrência monoprocessadora ou multiprocessadora. Quinn (Quinn, 1987), sugere 3 (três) maneiras de se construir um algoritmo paralelo:

❖ Detectar e explorar algum paralelismo inerente a um algoritmo seqüencial existente. Apesar dessa estratégia apresentar baixo *speedup*, é bastante utilizada.

❖ Criar um algoritmo paralelo novo. Essa técnica que possibilita melhor desempenho necessitando, porém, de uma reestruturação completa do algoritmo;

❖ Adaptar outro algoritmo paralelo que resolva um problema similar. Nessa caso, têm-se bom desempenho exigindo menor trabalho do programador em relação à construção completa do algoritmo.

Dessa forma, a programação concorrente é regida pela idéia de dividir determinadas aplicações em partes menores e que cada parcela resolva uma porção do problema. Entretanto, para essa divisão, há a necessidade de certos recursos adicionais, não necessários na programação seqüencial.

Para que os processos concorrentes sejam ativados, há a necessidade de um conjunto de ferramentas que determine exatamente que porção do código será paralela e qual será seqüencial. Inúmeras construções para a ativação e término de processos concorrentes têm sido propostas, apresentando características e finalidades distintas (Almasi & Gottlieb, 1994).

A escolha do método de ativação deve ser feita de acordo com os objetivos do programador. *Corotinas* são utilizadas para ativação de processos concorrentes, *fork/join* e *cobegin/coend* para a ativação de processos paralelos e *doall* para a ativação paralela de instâncias de *loops* (Almasi & Gottlieb, 1994).

Outro fator que exerce influência sobre a programação concorrente é a coordenação e especificação da interação entre os processos. Na programação concorrente os processos que cooperam entre si para a resolução de um determinado problema, devem comunicar-se e sincronizar-se, a fim de que haja interação entre eles. Dessa forma, a comunicação permite que a execução de um processo interfira na execução de outro, a qual pode ser feita ou pelo uso de variáveis compartilhadas (quando se trata de memória compartilhada), ou pelo uso de passagem de mensagem (quando se tem memória distribuída).

A sincronização se faz importante para que o acesso simultâneo não tome os dados, que são compartilhados, inconsistentes e para que seja possível o controle na seqüência da execução paralela.

Conhecendo-se os mecanismos de ativação de processos concorrentes e de comunicação, um ponto extremamente importante é a escolha do tipo de ferramenta que será usada para a construção de um programa fonte, que represente o algoritmo paralelo. Essas ferramentas podem ser implementadas de duas maneiras distintas: ou como linguagens de programação concorrente ou como ambientes de passagem de mensagens.

As linguagens de programação contêm os comandos específicos que os algoritmos paralelos necessitam. Os ambientes de passagem de mensagem (ou interfaces de passagem de mensagem) são compostos por bibliotecas de comunicação que implementam os mecanismos necessários à execução paralela, agindo como extensões das linguagens seqüenciais existentes (Almasi & Gottlieb, 1994).

2.3.4 Passagem de Mensagem

No paradigma de programação de memória compartilhada, os programadores enxergam seus códigos como um conjunto de processos tendo acesso a uma fila central de variáveis compartilhadas. Esse estilo é naturalmente aplicado para computadores com memória compartilhada.

No paradigma de programação de memória distribuída, os programadores vêem seus programas como uma coleção de processos com variáveis locais privadas, além disso, a comunicação entre processadores é feita via passagem de mensagem. Esse paradigma é adequado para arquiteturas de memória distribuída.

Segundo Angela Quealy (Quealy, 1994), passagem de mensagem é um método para a comunicação de processos quando não há compartilhamento de memória. Ela se faz necessária porque as memórias são locais aos processadores: não há compartilhamento de variáveis e é a única maneira de se obter dados de outras máquinas.

Um modelo genérico de passagem de mensagem é mostrado na Figura 2.6, onde n processos têm acesso direto à memória local e a comunicação entre os processos deve ser executada por passagem de mensagem através da rede.

Figura 2.6 - Modelo de Passagem de Mensagem Genérico

2.3.5 Ambientes de Passagem de Mensagem

Um programa que utiliza troca de mensagens pode ser definido, de maneira simples, como um conjunto de programas sequenciais distribuídos em vários processadores, que se comunicam através de um conjunto limitado e bem definido de instruções pertencentes a um Ambiente de Passagem de Mensagem.

Esses procedimentos tomam-se disponíveis para os programas na forma de uma biblioteca de passagem de mensagem, que pode ser acrescentada a uma linguagem sequencial comum (como por exemplo, C e FORTRAN), permitindo que os programas sequenciais sejam utilizados sem maiores adaptações para fazer processamento paralelo com alto desempenho (Mcbryan, 1994).

Os ambientes de passagem de mensagem foram desenvolvidos inicialmente para máquinas com processamento maciçamente paralelo, onde, devido à ausência de um padrão, cada fabricante desenvolveu seu próprio ambiente, sem se preocupar com a portabilidade do software gerado.

A fim de encontrar uma solução para esse problema, utilizou-se as chamadas **Plataformas de Portabilidade de Passagem de Mensagem**, que são ambientes de programação definidos e implementados em várias plataformas, as quais permitem a portabilidade dos programas paralelos de maneira fácil e direta (Mcbryan, 1994). Exemplos de ambientes de passagem de mensagem são PARMACS (Calkin et. al., 1994), Express (Flower & Kolawa, 1994), MPI (Walker, 1994; Geist et. al., 1996; Souza, 1996b) e PVM (Matrone et. al., 1993; Beguelin et. al., 1994).

O PVM é o ambiente de passagem de mensagem que alcançou maior aceitação e, atualmente, pode ser descrito como um padrão "de fato" nesse tipo de sistema (Geist et. al., 1996), tendo sido adotada como a plataforma de implementação neste trabalho, sendo descrita em maiores detalhes na próxima seção.

2.4 PVM (Parallel Virtual Machine)

PVM é um conjunto integrado de bibliotecas e de ferramentas de software, cuja finalidade é simular um sistema computacional concorrente heterogêneo, flexível e de propósito geral, sobre uma rede de computadores com arquiteturas variáveis (Beguelin et. al., 1994; Geist et. al., 1997).

Assim, o conceito de máquina virtual tem importância singular no desenvolvimento do PVM, devido à capacidade de juntar diferentes estações de trabalho, computadores pessoais e computadores maciçamente paralelos, formando uma única máquina paralela virtual (Geist et. al., 1996).

O projeto inicial do PVM foi desenvolvido em 1989 no *Oak Ridge National Laboratory* (ORNL) e, desde então, produziu três versões principais, sendo que a partir da segunda versão foi desenvolvido em conjunto com a *University of Tennessee* e a *Carnegie Mellon University*.

A fim de viabilizar a computação paralela para um maior número de possíveis usuários, os quais não possuem uma plataforma UNIX e/ou computadores paralelos, desenvolveram-se versões do PVM para a plataforma Windows (Win32). Exemplos dessas versões são: o WPVM desenvolvido pelo PVM-team (Fischer & Dongarra, 1996), o WPVM

desenvolvido na Universidade de Coimbra em Portugal (Alves et. al., 1995) e o PVM-W95, desenvolvido no grupo de Sistemas Distribuídos e Programação Concorrente do Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (Souza, 1996a).

O PVM fornece as funções para o usuário iniciar, comunicar e sincronizar tarefas na máquina virtual. Uma tarefa é definida como uma unidade computacional em PVM análoga aos processos UNIX (Beguelin et. al., 1994).

Dessa maneira, a idéia que rege o modelo computacional do PVM é baseada na noção de que as aplicações são formadas por tarefas, cada uma delas representando uma parte da carga de trabalho da aplicação. Cada aplicação pode ser paralelizada por dois métodos (Beguelin et. al., 1994):

- **paralelismo funcional:** onde a aplicação é dividida através das suas funções, isto é, cada tarefa desempenha um serviço diferente; esse método constitui o paradigma MPMD (Multiple-Program Multiple-Data).
- **paralelismo de dados:** onde todas as tarefas são as mesmas, mas cada uma resolve uma pequena parte dos dados; esse método é o mais usado e refere-se ao paradigma SPMD (Single-Program Multiple-Data).

Finalizando, os programas de aplicação enxergam o sistema de software PVM como um recurso computacional paralelo, flexível e geral, que dá suporte à memória compartilhada, passagem de mensagem e modelos de computação híbridos. Esse recurso pode ser utilizado em três níveis diferentes (Matrone et. al., 1993):

- ❖ **modo transparente,** no qual os computadores solicitados são automaticamente localizados no local mais apropriado;
- ❖ **modo dependente de arquitetura,** onde o usuário indica uma arquitetura específica na qual o componente está para ser executado;
- ❖ **modo de baixo nível,** permitindo que uma máquina particular possa ser especificada.

2.4.1 Componentes do PVM

O sistema de passagem de mensagem PVM é composto por duas partes. A primeira é um *daemon*², chamado *pvmd3* e na maioria das vezes abreviado por *pvmd*, o qual é executado em cada *host* que integra a máquina virtual, atuando como “gerenciador” da máquina e roteador de mensagens. A segunda parte é uma biblioteca de rotinas com a interface PVM, chamada *Libpvm*. Essa parte contém um conjunto de primitivas que agem como ponto de ligação entre uma tarefa e a máquina virtual (Beguelin et. al., 1994; Souza, 1996a) (veja Figura 2.7).

PVM Daemon (Pvmd)

Para que o sistema PVM possa ser executado, se faz necessário que um *pvmd* resida em cada *host* de uma máquina virtual e que todos sejam configurados para trabalhar em conjunto. Dessa forma, a aplicação pode então ser iniciada a partir do *prompt* do sistema operacional, em qualquer computador pertencente à máquina virtual.

No sistema PVM é possível que haja mais de uma máquina virtual utilizando os mesmos equipamentos simultaneamente, sendo que uma aplicação não interfere nas demais. Isso só é possível porque o *pvmd* que faz parte da máquina virtual de um usuário, não tem acesso a outro *pvmd* pertencente a outra máquina (Geist et. al., 1994).

O *pvmd* tem como função agir somente como roteador e controlador de mensagem. Ele fornece um ponto de acesso em cada *host*, assim como autenticação, controle de processos e detecção de falhas.

² *Daemons* são processos servidores que atendem às requisições de processos clientes.

Figura 2.7 - Exemplo de Implementação PVM

O primeiro **pvmd** (iniciado pelo usuário) é designado como mestre, enquanto os outros (iniciados pelo mestre) são chamados escravos. Durante a maioria das operações os **pvmds** não possuem diferença. Apenas quando há necessidade de operações de gerenciamento, como criar novos **pvmds** escravos e adicioná-los à máquina virtual, é que essa diferença aparece, pois somente o **pvmd** mestre tem permissão para fazer isso.

As estruturas de dados mais importantes no **pvmd** são as tabelas de *hosts* e as de tarefas, as quais descrevem a configuração da máquina virtual e determinam as tarefas que estão sendo executadas. Ligadas a essas estão as filas de pacotes e mensagens, e os salvadores de contexto, esses últimos são processadores e estruturas de dados que manipulam o estado da informação quando ocorre o escalonamento entre os processos (Beguelin et. al., 1994).

Como as finalidades principais do **pvmd** são o roteamento de mensagens e a configuração da máquina virtual, o primeiro passo do **pvmd** é criar um mecanismo de comunicação entre processos (os *sockets*) e permitir a sua utilização. Isso possibilita a comunicação com as tarefas e outros **pvmds**. O segundo passo do **pvmd** é criar as estruturas de dados que irão conter as informações sobre os outros *hosts* e sobre as tarefas PVM que estão em execução sob sua supervisão.

Biblioteca de Comunicação (Libpvm)

A biblioteca libpvm é um conjunto de funções que permite uma tarefa fazer contato com o pvmd e com outras tarefas. Ela contém funções para compor ou decompor mensagens, enviar requisições ao pvmd e receber respostas (Geist et. al., 1994).

A libpvm compartilha o espaço de endereçamento com as aplicações paralelas e dessa forma torna-se fundamental que a mesma seja tão pequena quanto possível. A libpvm contém 82 funções disponíveis para os usuários. Elas compreendem um elo de ligação entre a aplicação paralela e o PVM, simples e de fácil utilização, sendo talvez uma das razões pelas quais o PVM é tão aceito e utilizado (Geist et. al., 1994; Beguelin et. al., 1994).

As funções de mais alto nível da libpvm são as responsáveis pela interface com o usuário e são escritas independentemente da máquina onde o PVM está executando. Já as funções de mais baixo nível, que são dependentes da plataforma escolhida, foram separadas para permitir a sua substituição quando há a necessidade de portabilidade (Souza, 1996a).

Identificadores de Tarefas (TID)

O PVM adota o conceito de identificador de tarefa (*TID-Task Identifiers*), um inteiro de 32 bits, que tem por objetivo fornecer um único número para cada objeto pertencente à máquina virtual. Dessa maneira, os TID's agem sobre tarefas de usuários, Pvmd's (mestre ou escravo) e/ou grupos de tarefas.

O TID possui quatro campos, como mostrado na Figura 2.8, permitindo uma série de vantagens que retificam os objetivos de portabilidade e heterogeneidade. Por exemplo, as tarefas podem receber seus TIDs sem a necessidade de comunicação entre os Pvmds; as mensagens podem ser roteadas mais facilmente devido à hierarquia formada com o número do *host* precedendo o número de tarefas. Os campos S, G, H são interpretados da mesma forma em todos os Pvmd's da máquina virtual.

Figura 2.8 - Identificador de Tarefas no PVM

O campo S (1 bit) indica o servidor (*daemon*) do *host* H; assim sendo, o campo H possui o número do *host* e o campo L está vazio. O campo G (1 bit) é utilizado para referenciar tarefas pertencentes a um grupo, sendo bastante usado nas mensagens *broadcast*.

O campo L pode conter vários significados, dependendo do Pvmmd que o utiliza, porém, sua atividade genérica é designar uma tarefa local a um determinado *host* H. Para finalizar, o campo H contém o número do *host* na máquina virtual. Cada Pvmmd possui um único número, fornecido quando o mesmo é colocado em execução e conhecido por todos os outros Pvmmds da máquina virtual através da tabela de *hosts*.

Tratamento das Mensagens

O PVM possui rotinas para o "empacotamento" e o envio de mensagens entre tarefas. O modelo PVM foi construído de tal forma que, qualquer tarefa pode enviar uma mensagem para outra sem impor limites para o tamanho ou número de tais mensagens.

Para o usuário PVM, a comunicação empregada utiliza *send* bloqueante assíncrono, *receive* bloqueante assíncrono e *receive* não bloqueante. O *send* é chamado de bloqueante assíncrono porque retorna tão logo o *buffer* utilizado para enviar a mensagem esteja livre para ser novamente utilizado e é chamado de assíncrono porque não depende do receptor executar um *receive* para poder retornar (Geist et. al., 1994; Beguelin et. al., 1994).

Um *receive* é não bloqueante quando ele retorna imediatamente após ter verificado o *buffer* no *host* receptor. Por outro lado, um *receive* bloqueante não retorna enquanto a mensagem esperada não for recebida e inserida no *buffer*.

O PVM oferece comunicação ponto-a-ponto, *broadcasting* (para um grupo de tarefas definidos pelo usuário) e *multicasting* (para um conjunto de tarefas). A base do sistema de

comunicação do PVM é construída sobre os protocolos Internet TCP e UDP. Talvez outros protocolos existentes fossem mais apropriados, entretanto, não seriam tão difundidos quanto os escolhidos, o que poderia limitar a portabilidade do sistema (Geist et. al., 1994).

Paradigmas de Programação Usados no PVM

Contextualizado no tema deste trabalho, esta seção aborda os paradigmas de programação usados no sistema PVM, apresentando duas metodologias gerais. A primeira, chamada decomposição de dados ou particionamento, assume que todos os problemas envolvidos implicam em processamento sobre um ou mais conjuntos de dados. A segunda, intitulada decomposição de funções, divide a carga relacionada à aplicação em diferentes funções ou operações (Geist et. al., 1994)

Em suma, o escalonamento no PVM dá suporte tanto ao particionamento em nível de função, quanto em nível de dados, ou seja, na decomposição funcional, diferentes tarefas realizam diferentes operações e na decomposição de dados, tarefas idênticas operam sobre diferentes porções de dados.

Decomposição de Dados

Nessa estratégia os dados são distribuídos para os diferentes computadores que formam a máquina virtual. Essa distribuição pode ser feita de maneira estática, onde cada processador sabe *a priori* sua parte da carga de trabalho (Jackson & Humphres, 1997; Wang et. al., 1998), ou dinâmica, onde um processo controlador aloca sub-unidades da carga de trabalho para processadores quando esses estão livres (sem cargas) (Shirazi et. al., 1995).

Em muitas aplicações itens de dados requerem processamento idêntico. A maneira natural para paralelizar esse problema é distribuindo dados para diferentes processadores, os quais irão realizar processamento idêntico (Almasi & Gottlieb, 1994; Quinn, 1994). Essa estratégia, está fortemente relacionada com o modelo de programação, SPMD (Single Program Multiple Data) (Dantas, 1996).

Decomposição de Funções

A decomposição de funções é alcançada através do particionamento da carga total do sistema, em termos de diferentes operações (e/ou funções) (Geist et. al., 1994).

O paralelismo de funções é o modelo no qual a execução de diferentes funções é realizada concorrentemente sobre diferentes processadores, geralmente aplicado para diferentes itens de dados. Esse modelo de paralelismo é apropriado para arquiteturas MIMD devido requerer múltiplas instruções (Dantas, 1996).

2.4.2 PVM versão 3.4 e HARNESS

O projeto inicial do PVM passou por três versões, onde cada nova versão era fundamentada sobre diferentes projetos internos. Entretanto, há três princípios básicos que guiam e direcionam os objetivos nos novos projetos e implementações do PVM, que são (Geist et. al., 1997): simplicidade da interface, transparência de heterogeneidade e configuração dinâmica do sistema.

Dessa maneira, novas versões do ambiente de passagem de mensagens têm sido desenvolvidas, como a versão 3.4 do PVM e o HARNESS (Heterogeneous Adaptable Reconfigurable NEtworked SystemS), os quais mostram-se atentos a novos conceitos que sejam capazes de flexibilizar o maior número de recursos possíveis (Geist et al., 1997; Dongarra et al., 1998).

Como exemplo desses novos conceitos, tem-se o Generalized Plug-in Machine (GPM), o qual viabiliza a configuração dinâmica de uma máquina virtual, propiciando que a mesma possa adaptar-se às atuais necessidades das aplicações (Geist et. al., 1997).

A arquitetura do sistema HARNESS é projetada de tal forma que a mesma seja o mais aberta possível. Dessa maneira, para alcançar essas características, o projeto do sistema foi direcionado sobre dois aspectos :

- ❖ Gerenciamento do status de uma máquina virtual que é composta por um conjunto de *hosts* dinamicamente modificados.

❖ Capacidade de expandir o conjunto de serviços disponibilizados para os usuários, através do acréscimo no sistema de novos módulos (possivelmente definidos pelo usuário), sem comprometer a consistência do ambiente de programação.

Contudo, é importante ressaltar que, com a nova geração dos ambientes de passagem de mensagem, o problema de escalonamento das tarefas sobre uma plataforma heterogênea é, sem dúvida, uma das atividades mais beneficiadas com a flexibilidade imposta nesses ambientes, pois permite a inclusão de técnicas de escalonamento mais eficazes e específicas para cada realidade, podendo a mesma ser trocada de acordo com a realidade da máquina virtual.

2.5 Considerações Finais

Apesar das mudanças rápidas fazerem parte da indústria de computadores desde o seu nascimento, os novos desenvolvimentos parecem acontecer muito mais rapidamente nos dias de hoje que em anos anteriores. Muitas dessas mudanças no hardware têm impacto direto nos futuros sistemas distribuídos.

Entretanto, em uma arquitetura maciçamente paralela, todos os processadores são normalmente iguais em capacidade, recursos, software e velocidade de comunicação, o que não acontece em um sistema distribuído.

Os computadores disponíveis em um sistema distribuído podem ser de diferentes fabricantes, com diferentes compiladores. Portanto, quando se deseja explorar um conjunto de computadores interligados por uma rede, poderá haver vários tipos de heterogeneidade (Beguelin et. al., 1994).

Máquinas com potências computacionais diferentes podem gerar desempenhos fracos, fazendo com que haja no mesmo sistema máquinas ociosas enquanto há outras sobrecarregadas. Como sistemas distribuídos possuem vários usuários, cada um executando suas aplicações, a carga de trabalho atribuída a cada máquina e à(s) rede(s) pode variar muito, afetando aplicações que desejam explorar o paralelismo no sistema.

Dessa forma, para garantir o desempenho de um sistema distribuído faz-se necessário implementar estratégias de escalonamento de processos, capazes de assegurar que as máquinas

com uma potência maior não fiquem ociosas aguardando o processamento das máquinas com menor potência computacional, garantindo um ganho direto no desempenho do sistema como um todo.

Escalonamento de Processos

Este capítulo é dedicado ao estudo do Escalonamento e Balanceamento de Cargas em sistemas distribuídos e paralelos. Inicialmente são apresentadas as classificações propostas para os algoritmos. Em seguida, dá-se ênfase aos componentes de um escalonador e ao índice de cargas. Finalizando, destacam-se os componentes responsáveis pelo desempenho do escalonador.

3.1 Considerações Iniciais

Computadores maciçamente paralelos (MPP) estão se tornando uma classe cada vez mais popular e comum de sistemas de memória distribuída, que são destinados a executar grandes aplicações. Entretanto, a idéia de explorar o poder computacional das redes locais de estações de trabalho, permitindo que a mesma possa ser usada como um sistema distribuído tem experimentado um aumento considerável nos últimos anos (Clark et. al., 1995). Esse aumento é impulsionado pela vantajosa relação custo/benefício e pela relativa aceitação dos ambientes de passagem de mensagem, como o PVM (Geist et. al., 1997) e o MPI (Snir et. al., 1996).

Contudo, se em máquinas maciçamente paralelas, como multicomputadores³ e multiprocessadores⁴, um bom escalonamento de processos e distribuição de cargas são

³ São máquinas do tipo MIMD com memória distribuída e caracterizam-se por cada processador possuir sua própria memória local. Os processos comunicam-se através da troca de mensagem.

⁴ São máquinas MIMD com memória compartilhada e caracterizam-se pela existência de uma memória global, de maneira que através do compartilhamento de posições de memória ocorre a comunicação entre os processos.

importantes (Wang & Morris, 1995), em redes de estações de trabalho isso é particularmente mais importante (Chow & Abraham, 1995), visto que em um sistema MPP, todos os processadores são iguais, e isso não é necessariamente verdade em uma rede de computadores.

Dessa maneira, deve-se lidar com vários tipos de heterogeneidade de hardware (tais como, potências computacionais, arquiteturas, memória e dispositivos de entrada/saída) e de software básico (sistemas operacionais) (Mehra & Wah, 1993). Além disso, tratando-se de ambientes multiusuários, a carga de trabalho manipulada por cada usuário pode ser diferente, e dessa forma, a chegada aleatória de processos criados recentemente e que requeiram recursos distintos pode causar um desequilíbrio no sistema.

Evidentemente, com essas características a mesma quantidade de carga não pode ser distribuída para todas as máquinas⁵ de maneira igual, ou seja, para cada máquina deve ser alocada uma quantidade de carga proporcional ao seu estado e capacidade.

O escalonamento de processos é a atividade responsável por atribuir processos aos *hosts* para que esses sejam executados e, conseqüentemente, melhorem o desempenho do sistema (Shirazi et. al., 1995; Feitelson et. al., 1997).

Quando o escalonamento envolve somente um *host*, são considerados apenas fatores locais como entrada/saída, porções de tempo disponíveis no processador, entre outros. Entretanto, o escalonamento feito para os sistemas computacionais distribuídos, envolvendo vários *hosts*, tem por finalidade distribuir os processos entre esses processadores, seguindo alguns objetivos previamente estabelecidos (Feitelson & Rudolph, 1996).

O balanceamento de cargas é um desses objetivos, o qual tem a finalidade de manter o sistema em equilíbrio, transferindo cargas (ou tarefas) de máquinas sobrecarregadas para máquinas ociosas ou menos carregadas. Assim sendo, o algoritmo de balanceamento de cargas provê um aumento considerável no desempenho final do sistema, visto que a otimização e eficácia do sistema estão diretamente relacionados com a melhor utilização da capacidade de processamento instalada (Shivaratri et. al., 1992).

⁵ Máquinas, elemento de processamento, estação de trabalho, *hosts* e nó são termos que, neste trabalho designam um hardware composto por no mínimo um processador com acesso à memória.

Em suma, as estratégias de escalonamento são na maioria das vezes, direcionadas na tentativa de obter-se diminuição do tempo médio de resposta, diminuição dos atrasos na comunicação e aumento da utilização dos recursos disponíveis (Shivaratri et. al., 1992). Todavia, as atividades mais importantes do escalonamento de processos são (Feitelson et. al., 1997): satisfação dos seus usuários e aumento de produtividade com a utilização de seus recursos.

Faz-se necessário ressaltar que a organização de conceitos e termos existentes na área de escalonamento de processos não é uma tarefa fácil. Isso é condicionado pela existência de várias terminologias imprecisas, pela falta de uma nomenclatura única e pela falta de clareza, dificultando, assim, o estudo e relacionamento correto dos diferentes trabalhos (Feitelson & Rudolph, 1996; Feitelson et. al., 1997).

Na tentativa de unir o maior número de características, da maneira mais coerente possível, vários autores propõem diversas classificações, todas elas objetivando alcançar uma taxonomia única e clara, capaz de representar os mais diversos conceitos empregados na área (Casavant & Kuhl, 88; Feitelson et. al., 1997; Martin, 1999).

3.2 Classificação dos Algoritmos

A árdua tentativa dos mais diversos autores em produzir uma taxonomia ampla o suficiente para contemplar o maior número de características, não é uma tarefa simples. A grande dificuldade, entretanto, está no relacionamento dos trabalhos desenvolvidos, em função da diversidade dessas características e, principalmente, devido à falta de clareza de seus autores sobre as semelhanças e diferenças do seu trabalho em relação aos demais (Feitelson et. al., 1997; Souza et. al., 1998).

Dentre todas as classificações propostas, esta seção aborda apenas as classificações mais citadas e referenciadas na literatura.

3.2.1 Classificação de Casavant

Casavant, em 1988 (Casavant & Kulh, 88), propõe uma classificação que tenta ser abrangente para sistemas distribuídos de propósito geral. Essa taxonomia tem o objetivo de apresentar o problema de escalonamento nos mais diversos aspectos, assim como possibilitar que futuros trabalhos possam ser inseridos nesse contexto, mesmo que modifique o conjunto de características consideradas importante atualmente.

Essa taxonomia é dividida em duas partes: uma classificação hierárquica (onde seus elementos possuem uma relação de dependência entre si) e uma classificação plana (onde seus elementos possuem liberdade para se adaptarem aos mais diversos tipos de escalonamento) (Casavant & Kulh, 1988).

Classificação Hierárquica

A classificação hierárquica é apresentada em sua totalidade na Figura 3.1, onde no mais alto nível, pode-se distinguir entre escalonamento local e global.

O *escalonamento local* é realizado pelo sistema operacional de um computador, o qual atribui os períodos de tempo de um único processador entre todos os processadores em execução (*time-slice*). O *escalonamento global* é executado em um sistema computacional distribuído e corresponde à tomada de decisão de qual elemento do processamento executará um determinado processo. Essa decisão pode ser realizada de maneira estática ou dinâmica (Casavant & Kulh, 1988; Shirazi et. al., 1995; Xu & Lau, 1997).

No *escalonamento global estático* (ou determinístico) a tomada de decisão de escalonamento é realizada em tempo de compilação. Assim sendo, as informações sobre os tempos de execução das tarefas e recursos de processamento devem ser conhecidos antes que os processos comecem sua execução (Jackson & Humphres, 1997). Em um sistema onde a carga varia imprevisivelmente, o *escalonamento estático* não é viável, visto que não pode tomar decisões em tempo de execução do processo.

Figura 3.1 - Classificação Hierárquica proposta por Casavant.

A geração de *escalonamentos ótimos* é um problema NP-completo (Lewis, 1996). Devido ao tempo exponencial para se encontrar a solução nesses casos, a busca por soluções ótimas é empregada apenas em casos específicos e que envolvam, geralmente, poucas combinações.

O método de *escalonamento estático sub-ótimo* é uma alternativa ao método ótimo, o qual tenta encontrar uma solução "aceitável" para o problema, sendo que o termo "aceitável" varia de acordo com cada situação. As soluções *sub-ótimas* podem ser encontradas através de duas abordagens: *aproximada* e *heurística*. Na *aproximada* a solução é pesquisada por algum método de busca conhecido. Na abordagem *heurística* são utilizadas regras especiais (como Inteligência Artificial), para produzir o processo de escalonamento na direção aproximada de uma solução ótima.

O *escalonamento dinâmico*, também denominado (erroneamente) como balanceamento de cargas por alguns autores (Mehra & Wah, 1993; Milojicic et. al., 1993;

Franklin & Govindan, 1996; Stellner & Trinitis, 1997), utiliza muito pouco ou até mesmo nenhum conhecimento sobre as aplicações, durante o tempo de compilação. A decisão sobre a distribuição dos processos entre os processadores é feita durante a execução. A principal vantagem do *escalonamento dinâmico* é que não é preciso se preocupar com o comportamento de uma aplicação antes que ela seja executada, tomando o sistema capaz de adaptar-se às mudanças ambientais durante a execução dos programas (Shirazi et. al., 1995; Jackson & Humphres, 1997).

Devido a essa capacidade do *escalonamento dinâmico*, esse método é largamente empregado em ambientes multiusuários, como por exemplo os sistemas distribuídos, em virtude da grande variação de tipos de usuários e tipos de aplicações.

No *método dinâmico fisicamente distribuído*, o escalonamento é realizado nos diversos elementos de processamento integrantes do sistema. Cada processador autônomo possui as informações da carga dos demais processadores e, assim, decide o escalonamento.

O *escalonamento dinâmico distribuído* pode ser *cooperativo* ou *não cooperativo*. Como o próprio nome sugere, no *escalonamento não cooperativo* cada processador toma as decisões independentes dos demais, sem haver preocupação sobre os resultados que isso possa ter sobre o sistema inteiro. Entretanto, no *escalonamento cooperativo*, as decisões individuais devem levar em consideração um objetivo comum para o sistema como um todo.

O *escalonamento não distribuído* possui um elemento de processamento que é responsável pelas decisões para o sistema inteiro. Os outros processadores atuam apenas como coletores de informações locais, enviando-as para o processador central (Casavant & Kuhl, 1988; Shirazi et. al., 1995).

Classificação Plana

A classificação plana, apresentada por Casavant (Casavant & Kuhl, 1988), não segue nenhuma hierarquia, classificando os algoritmos de escalonamento de acordo com outras características, as quais não são exclusivas a um determinado tipo de escalonamento verificado acima. As características usadas para a classificação plana dos algoritmos de escalonamento são: se é adaptativo ou não, se possui preempção e, finalizando, se é estável e apresenta eficiência.

Os algoritmos adaptativos são capazes de adaptar as suas políticas durante a execução, em função de mudanças ocorridas no sistema. Por consequência, um sistema não adaptativo é aquele que não leva em consideração o histórico do sistema em suas decisões e, dessa forma, apresenta sempre o mesmo comportamento previamente estabelecido (Casavant & Kuhl, 1988).

Nos algoritmos de escalonamento preemptivos, também chamados de algoritmos com migração de processos, um processo é atribuído novamente para outro processador, mesmo já tendo iniciado sua execução (Eskicioglu, 1989; Shirazi et. al, 1995). Essa operação geralmente é dispendiosa e difícil de ser realizada, pois todas as informações necessárias sobre o processo ativo devem ser transferidas para o outro processador, para que o processo possa ser retomado no elemento de processamento destino (Theimer & Hayes, 1991; Shivaratri et. al., 1992). O principal motivo para a existência dos algoritmos com preempção é permitir a correção de um escalonamento já efetuado, devido a mudanças ocorridas no sistema e não previstas anteriormente.

Por outro lado, a estabilidade de um sistema está fortemente relacionado à sua capacidade de responder à sua carga de trabalho. Na área de transferência de tarefas, a estabilidade se divide em duas formas (Shivaratri et. al., 1992):

- ❖ a perspectiva da teoria de filas;
- ❖ a perspectiva dos algoritmos de distribuição.

Quanto à teoria de filas, um sistema é considerado instável quando as tarefas de um sistema apresentam uma taxa de chegada maior do que sua taxa de conclusão. Entretanto, um algoritmo de distribuição de carga é eficiente sobre um determinado conjunto de condições, se ele consegue aumentar o desempenho relativo de um sistema. Dessa maneira, um algoritmo eficiente é estável, mas o contrário nem sempre é verdadeiro (Shivaratri et. al., 1992).

Analisando-se pela perspectiva dos algoritmos de distribuição, se um algoritmo consegue realizar operações improdutivas indefinidamente, com uma probabilidade diferente de zero, então esse algoritmo é instável.

3.2.2 Classificação de Feitelson

Feitelson, em 1995 (Feitelson & Rudolph, 1995), classifica os algoritmos de escalonamento como estáticos, em tempo de execução (*Runtime System*), feitos pelo sistema operacional e com finalidades administrativas.

O escalonamento estático apresenta a mesma definição dada através da classificação de Casavant (Casavant & Kuhl, 1988). O escalonamento realizado em tempo de execução segue a mesma idéia do escalonamento dinâmico proposta anteriormente, mas, com a diferença de que o mesmo deve ser obrigatoriamente feito em nível de usuário. Todavia, há os algoritmos de escalonamento feitos pelo sistema operacional, em nível de sistema, responsáveis pelas necessidades coletivas, não apenas as individuais dos seus usuários.

Os algoritmos com finalidades administrativas fazem o escalonamento não apenas considerando as questões técnicas, mas sim, questões administrativas na hora de se atribuir um processo a um elemento de processamento.

Em suma, a classificação apresentada por Feitelson (Feitelson & Rudolph, 1995) difere da proposta de Casavant (Casavant & Kuhl, 1988), fundamentalmente, pela primeira estar voltada às plataformas paralelas, enquanto que a segunda está mais voltada aos sistemas distribuídos de propósito geral.

3.2.3 Classificação de Lüling

Nessa classificação o objetivo primordial de Lüling (Lüling et. al., 1993) foi propor um esquema baseado em tomada de decisão, espaço de alocação e iniciativa da distribuição da carga. Para Lüling, qualquer algoritmo de escalonamento de carga dinâmico apresenta duas fases principais: uma fase de decisão e outra de alocação.

Na fase de decisão, escolhe-se qual unidade de carga deve ser distribuída. A decisão pode ser baseada na situação local de um elemento de processamento e de seus vizinhos (decisão local) ou de algum subconjunto de elementos de processamento da rede inteira (decisão global) (Lüling et. al., 1993; Martin, 1999).

Figura 3.2 - Classificação de Lüling

Na fase de distribuição, uma unidade de carga é transferida para outro processador com a finalidade de reduzir o desequilíbrio do sistema. A distribuição de cargas pode ser feita para os processadores vizinhos (espaço de distribuição local) ou para qualquer outro processador pertencente ao sistema (espaço de distribuição global).

Em síntese, Lüling (Lüling et. al., 1993) classificou os algoritmos em quatro grandes grupos: DLML (Decisão Local Migração Local), DLMG (Decisão Local Migração Global), DGML (Decisão Global Migração Local) e por último DGMG (Decisão Global Migração Global), mostrados na Figura 3.2.

No caso do algoritmo DLML, a decisão de distribuir a carga (Lüling usa o termo migrar) é tomada localmente e essa carga só pode ser transferida para processadores próximos ao processador que inicia a distribuição. No segundo caso, a decisão continua sendo tomada por um processador local, mas a carga pode ser transferida para qualquer processador pertencente ao sistema como um todo.

No algoritmo DGML a tomada de decisão de distribuir uma tarefa é baseada em informações de todos os *hosts* do sistema, porém a carga é distribuída apenas para os *hosts* próximos ao *host* que tomou a decisão de transferir a carga. Para finalizar, no algoritmo DGMG a decisão pode ser realizada por qualquer *host* do sistema, e após tomada a decisão a carga pode

ser transferida para qualquer máquina, independente da distância dessa para a máquina de origem da carga.

Lüling também faz distinção em relação a quem inicia a atividade de escalonamento: transmissor, receptor ou simétrico (uma forma combinada dos dois).

Iniciado pelo Transmissor

Nesse algoritmo a atividade de distribuição de carga é iniciada por um processador sobrecarregado (transmissor), tentando enviar uma tarefa para um nó ocioso. Existem três técnicas que utilizam esse algoritmo:

- ❖ Aleatório: nessa estratégia a tarefa é transferida para um processador selecionado aleatoriamente, ou seja, não se usa nenhuma informação de estado. Dessa forma, uma transferência inútil ocorre caso a tarefa seja distribuída para um processador que já está sobrecarregado.

- ❖ Limite: essa estratégia tem como principal vantagem a capacidade de evitar uma distribuição inútil, pois inspeciona um processador (selecionado aleatoriamente) para determinar se a distribuição da tarefa excederá o seu limite de cargas. Em princípio, se não ultrapassar, a tarefa é transferida para o processador selecionado, o qual deve executar a tarefa sem importar-se com o seu estado quando a tarefa realmente chegar.

- ❖ Menor Limite: nesse caso, um número de processadores é selecionado aleatoriamente e inspecionado para determinar os tamanhos de suas filas de processos pendentes esperando para serem executados. O processador com a menor fila é selecionado como o processador destino da tarefa transferida.

Iniciado pelo Receptor

No algoritmo iniciado pelo receptor, a atividade de distribuição de carga é iniciada a partir de um processador ocioso (receptor), o qual tenta receber uma tarefa de um processador sobrecarregado (transmissor) na rede.

Iniciado Simetricamente

Nesse algoritmo, ambos (transmissor e receptor) iniciam a atividade de distribuição de carga de uma tarefa. Algoritmos desse tipo podem apresentar as vantagens das duas categorias. Quando a carga do sistema é baixa, os elementos transmissores podem ter mais sucesso em encontrar processadores com pouca carga de trabalho. Porém, quando a carga é alta, os receptores são os mais indicados para encontrar processadores sobrecarregados (Shivaratri et. al., 1992).

Em síntese, há inúmeras outras classificações adotadas, considerando características mais específicas ou mais genéricas, dependendo das necessidades de seus autores. Entretanto, as classificações apresentadas aqui são o suficiente para exemplificar os diferentes rumos que o escalonamento de processos pode seguir, assim como a falta de uma nomenclatura única (Martin, 1999; Souza et. al., 1999a).

Uma explicação para as terminologias diferentes, deve-se, talvez, às diferentes finalidades que o escalonamento se propõe. Os sistemas computacionais distribuídos envolvem desde uma rede local de estações controlada por um sistema operacional distribuído, até uma máquina paralela com vários elementos de processamento. Esses sistemas, que possuem características e finalidades diferentes, necessitam de técnicas de escalonamento aprimoradas, porém diferenciadas (Souza et. al., 1999b).

Nas plataformas verdadeiramente paralelas, observa-se que os algoritmos de escalonamento se preocupam com questões como: dividir os elementos de processamento considerando duas dimensões: tempo e espaço, ou seja, quanto tempo os elementos de processamento devem (ou podem) ficar destinados a uma aplicação e o número mínimo de elementos requisitados por uma aplicação paralela (Feitelson et. al., 1997).

3.3 Componentes de um Algoritmo

Na literatura disponível, encontram-se inúmeros artigos propondo novos algoritmos de escalonamento, os quais demonstram a importância e interesse sobre o assunto. Todavia, há

pontos de importância ímpar a serem considerados no projeto de uma nova estratégia (também usada como política neste trabalho) de escalonamento.

Os usuários de um escalonador de processos, em geral, possuem características e necessidades mais amplas que as encontradas em um escalonador, pois na prática a implementação de um algoritmo sempre limitará uma maior flexibilidade no tratamento do escalonamento dos processos (Souza et. al., 1999b).

Dessa maneira, espera-se que existam algoritmos de escalonamento cada vez mais direcionados para atuar sobre certas aplicações particulares, tentando contemplar o tipo de trabalho da aplicação e, conseqüentemente, elevar a eficiência e eficácia do escalonador.

Um algoritmo de escalonamento é composto segundo uma política e um mecanismo que manipulam os objetivos do algoritmo. O mecanismo é a parte do esquema de escalonamento responsável em definir como o escalonamento será efetuado (Song et. al., 1997).

O segundo componente, a política, é responsável por refletir o que tem que ser feito para que haja o escalonamento de processos. Shivaratri (Shivaratri et. al., 1992) dividiu sabiamente as políticas em: política de informação, política de seleção, política de localização e política de informação.

Política de Transferência

Essa política está diretamente relacionada à distribuição de cargas pelo sistema, ou seja, é a responsável em determinar se um nó está em um estado adequado para receber cargas (receptor) ou para enviar cargas (transmissor) (Shivaratri et. al. 1992; Shirazi et. al., 1995). Em geral, as políticas de transferência se baseiam em limites de cargas para determinar o estado de um *host*.

Caso a carga do *host* esteja acima do limite, então esse é considerado pela política de transferência um *host* transmissor. Todavia, quando a carga do *host* está abaixo do valor limite, esse é considerado pela política um *host* receptor, capaz de aceitar cargas de outras máquinas existentes no sistema.

Política de Seleção

Como o próprio nome sugere, essa política de seleção tem por objetivo selecionar as tarefas que devem participar da distribuição de cargas (Shivaratri et. al., 1992). Caso a política de seleção considere também as tarefas já em execução para participarem da distribuição, então o escalonamento pode ser realizado de maneira preemptiva ou não preemptiva.

No primeiro caso, uma tarefa mesmo após já ter iniciado sua execução pode ser transferida para um outro processador, objetivando retificar um balanceamento de carga não ideal. No segundo, uma tarefa após ter iniciado sua execução em um processador não pode ser transferida. Dessa maneira, em um algoritmo sem preempção geralmente a tarefa selecionada é a tarefa criada mais recentemente e conseqüentemente a responsável pela sobrecarga do sistema (Xu & Lau, 1997).

Política de Localização

Essa política tem por finalidade identificar um parceiro para transmitir ou receber a(s) tarefa(s). Um *host* pode ser escolhido basicamente de 3 (três) maneiras: aleatoriamente, através da coleta de informações ou através da análise dos vizinhos. Isso tudo pode ser feito de maneira serial, ou em paralelo (por exemplo, *multicast*).

No caso aleatório, o *host* que deseja encontrar um parceiro para a sua carga, escolhe aleatoriamente qualquer outro *host* do sistema. Essa técnica pode provocar instabilidade. No caso da coleta de informações, o estado de todos os *hosts* do sistema é analisado e o melhor *host* é o escolhido. No último caso, análise dos vizinhos, apenas os *hosts* que estão próximos à máquina que deseja distribuir carga são analisados.

Política de Informação

Essa política decide quando as informações sobre o estado de outros *hosts* no sistema devem ser coletadas, de onde serão coletadas e quais informações coletar. Existem três tipos de política de informação (Shivaratri et. al., 1992):

❖ Política dirigida por demanda: nessa política um *host* coleta o estado de outro *host* somente quando ele se torna também um transmissor ou receptor disponível para iniciar e compartilhar a carga. Nesse caso, a troca de informação entre os *hosts* do sistema ocorre sempre que um *host* precisa distribuir e/ou compartilhar sua carga, pois há necessidade de conhecer o estado do sistema como um todo.

❖ Política periódica: essa política pode ser implementada de maneira centralizada ou descentralizada e realiza a coleta de informações periodicamente, ou seja, de tempos em tempos. No caso da política periódica centralizada, de tempos em tempos, os *hosts* enviam informações do seu estado para um *host* centralizado. No segundo caso, a informação de cada *host* é distribuída entre as demais máquinas pertencentes ao sistema.

❖ Política dirigida pela mudança de estado: nessa política os *hosts* difundem as informações sobre os seus estados todas as vezes em que seu estado muda por um certo grau. Na política centralizada os *hosts* enviam as informações para um único *host* coordenador, e no caso descentralizado, as informações são enviadas para os outros *hosts* do sistema.

A diferença básica entre a política dirigida por demanda e a política dirigida pela mudança de estado consiste na natureza do algoritmo, ou seja, no caso da política dirigida por demanda sempre que um *host* deseja realizar uma distribuição de carga, esse recupera o estado de todos os demais *hosts* do sistema. No segundo caso, a informação do estado é difundida sempre que houver uma mudança de estado em algum *host* do sistema.

3.4 Índice de Carga

Índice de carga é definido como um valor não negativo, o qual tende a zero ou a valores próximos de zero, caso a máquina esteja ociosa e, a valores maiores representando máquinas sobrecarregadas (Ferrari & Zhou, 1987; Xu & Lau, 1997).

De modo geral, o índice de carga quantifica o conceito de carga nos recursos do sistema. Dessa forma, um bom índice deve representar a estimativa do tempo de resposta dos processos residentes no processador. Assim, o índice de carga é usado para prever um

comportamento futuro, baseando-se em uma situação presente ou de um passado recente (Kunz, 1991; Schnor et. al., 1997).

A escolha de um índice de carga adequado é considerada um ponto essencial no projeto de um algoritmo de distribuição de cargas, pois o mesmo apresenta uma relação direta com o desempenho final do escalonamento. Para ser eficiente, o índice de carga deve refletir precisamente o estado de um nó. Além disso, a técnica usada para medir a carga deve ser eficaz e impor a menor sobrecarga possível (Shivaratri et. al., 1992; Dantas, 1996).

Há necessidade de considerar alguns pontos importantes, quando a escolha do índice de carga fizer-se necessária. Dentre esses pontos têm-se (Ferrari & Zhou, 1987):

- ❖ a necessidade de quantificar não somente o uso da CPU, mas também as requisições a memória e dispositivos de Entrada/Saída.

- ❖ adotar cargas que sejam capazes de prever o estado do sistema em um futuro próximo, pois o tempo de resposta de um processo será afetado mais pela carga futura do que pela carga presente.

- ❖ adotar índices relativamente estáveis, ou seja, valores flutuantes devem ser desconsiderados ou ignorados.

- ❖ ter uma relação simples (e idealmente linear) com o índice de desempenho, tal que seus valores possam ser facilmente traduzidos dentro daquela performance esperada.

Além disso, o índice de carga deve ser voltado aos objetivos do algoritmo de distribuição de cargas e às características da aplicação paralela que se deseja otimizar. Por exemplo, caso o índice de carga represente a utilização da CPU e essa esteja congestionada, esse sistema deve ser considerado sobrecarregado para aplicações do tipo *CPU-bound*, mas não teria sentido ser considerada sobrecarregada para aplicações do tipo *I/O-bound* (Ferrari & Zhou, 1987).

Entretanto, não há um consenso quanto à eficiência de um único índice de carga. Consequentemente, há na literatura um grande número de índices de cargas diferentes. Os índices que têm sido mais estudados e usados são (Ferrari & Zhou, 1987; Kunz, 1991; Dantas,

1996; Xu & Lau, 1997): tamanho da fila da CPU, a média do tamanho da fila da CPU em um determinado período, utilização da CPU, a quantidade de memória disponível, a taxa de chamadas do sistema, tempo de resposta e funções híbridas unindo mais de um índice de carga.

Pesquisas têm demonstrado diferenças na eficiência de cada índice de carga, e que os índices de cargas mais simples são particularmente os mais eficientes (Shivaratri et. al., 1992). Além disso, tem-se notado que os índices de carga baseados no tamanho da fila de algum recurso representam melhor o estado do sistema do que os índices de cargas que adotam a utilização do recurso. Isso ocorre porque quando uma máquina está com a sua carga alta, sua utilização de CPU, por exemplo, tende a 100%, não refletindo dessa forma o valor exato da carga. Por outro lado, caso seja usado a fila de processos na CPU, o valor da carga é refletido completamente, espelhando com exatidão a situação do sistema.

Kunz, em 91 (Kunz, 1991), realizou uma pesquisa importante sobre índice de carga e concluiu que o tamanho da fila de processos prontos na CPU é o índice de carga mais eficiente dentre os mencionados anteriormente. Isso se explica pela sua simplicidade de cálculo e, conseqüentemente, por não acarretar um alto custo na aplicação paralela. Todavia, apesar da simplicidade de quantificar-se esse índice de carga, Ferrari & Zhou (Ferrari & Zhou, 1987) chamam a atenção para o perigo de obter-se esse valor de maneira instantânea no sistema, visto que valores flutuantes (ruídos) podem ser capturados, dando uma falsa visão do estado do sistema.

Em suma, é importante adotar-se um índice de carga que seja a média de algum recurso sobre um determinado período de tempo, pois reduz-se a possibilidade de tomar-se algum valor que não corresponda ao estado real do sistema (Ferrari & Zhou, 1987).

3.5 Desempenho de um Escalonador

Os usuários de um escalonador de processos possuem características e necessidades mais abrangentes que as encontradas em um escalonador. Uma proposta de algoritmo de escalonamento tem por meta atuar sobre certos tipos de aplicações, tentando-se prever o tipo e a quantidade de trabalho que irão ocorrer. Mudando-se essas características, por algum motivo

qualquer, o algoritmo de escalonamento pode não apresentar a eficiência esperada. Considerar que serão executadas aplicações de apenas um tipo é, portanto, uma grande limitação (Feitelson et. al., 1997; Souza et. al., 1998).

Sumarizando, todo algoritmo de escalonamento deve possuir seus objetivos e, aliadas a esses objetivos, devem-se estar as métricas, as quais avaliam a eficiência do escalonamento (Souza et. al., 1998). Na literatura observa-se, entretanto, que as métricas desenvolvidas nem sempre são as mais apropriadas para o objetivo proposto ao algoritmo.

As métricas devem estar voltadas para os objetivos do escalonamento e ser tão flexíveis quanto ele. Por exemplo, nem sempre a redução no tempo de execução das aplicações é a métrica mais apropriada para indicar se o algoritmo de escalonamento está sendo eficiente ou não. Quando o objetivo é maximizar a utilização dos processadores, dá-se prioridade às aplicações "mais apropriadas" e, conseqüentemente, o tempo de resposta das aplicações "mais leves" deve aumentar.

Além das diferentes necessidades, o escalonamento, deve preocupar-se com inúmeros outros fatores, os quais influenciam decisivamente no seu desempenho final (Shivaratri et. al., 1992; Loh et. al., 1996). Dentre esses fatores, destacam-se (Souza et. al., 1998): a finalidade da plataforma computacional utilizada, o hardware disponível, ambientes mutiusuários, diferentes classes de aplicações e diferentes classes de trabalho.

A finalidade da plataforma computacional utilizada é um fator importante, pois o escalonamento adotado deve diferenciar de acordo com a finalidade da plataforma computacional utilizada. Por exemplo, em uma arquitetura paralela rodando aplicações paralelas os objetivos do escalonamento adotado mudam em relação aos objetivos para um sistema distribuído com aplicações sequenciais (Squillante, 1995; Feitelson & Rudolph, 1996; Harchol-Balter et. al., 1997).

Em princípio, uma das maneiras para incentivar a convergência entre a teoria e a prática nessa área é diminuir a dificuldade em relacionar trabalhos já existentes; analisar algoritmos propostos, buscando aproveitar boas soluções e resolvendo deficiências pendentes; combinar as idéias de diferentes escalonadores, pois as políticas de escalonamento dependem de certas hipóteses e quando as hipóteses falham, o escalonador não apresenta a mesma eficiência.

E acima de tudo usar funções que realmente reflitam as características e objetivos da organização, otimizando assim a utilização do sistema computacional (Feitelson et. al., 1997).

3.6 Considerações Finais

A importância do escalonamento de processos é uma realidade que tem dominado cada vez mais os objetivos das plataformas computacionais, quer sejam através das arquiteturas paralelas ou através dos sistemas distribuídos.

A atividade de escalonamento de processos pode ser realizada de inúmeras maneiras, dependendo da política de escalonamento empregada e, conseqüentemente, dos objetivos almejados com essa distribuição de processos entre as máquinas pertencentes ao sistema computacional.

A escolha correta da política de escalonamento influencia decisivamente no desempenho final da plataforma computacional. Caso não seja empregada uma política de escalonamento adequada às necessidades da plataforma, essa técnica pode transformar-se em uma estratégia êmula ao desempenho final. Por outro lado, o escalonamento sendo executado de forma correta é, sem dúvida, um grande aliado para que os recursos disponíveis sejam utilizados com eficácia.

Dessa forma, como pode ser observado durante o decorrer deste capítulo, há na comunidade científica inúmeros artigos que abordam, de alguma forma, a atividade de escalonamento de processos. Inúmeras políticas com as mais diversas características e objetivos são propostas, testadas e avaliadas. Todavia, os trabalhos de pesquisa desenvolvidos sobre esse tema têm sido direcionados na tentativa de diminuir a grande distância entre a teoria e prática e, além disso, criar ambientes que permitam explorar ao máximo a característica particular de cada aplicação.

Ultrapassando esses objetivos, o próximo capítulo (capítulo 4) apresenta um novo ambiente de escalonamento, capaz de introduzir flexibilidade aos algoritmos de escalonamento desenvolvidos, além de tornar transparente aos usuários finais os diversos fatores inclusos na escolha de um algoritmo de escalonamento (Araújo et. al., 1999a).

AMIGO - um Ambiente de Escalonamento

Este capítulo apresenta o Ambiente de Escalonamento AMIGO. Primeiramente, são abordadas suas características gerais e, em seguida, são descritos os componentes e os detalhes mais relevantes de sua implementação.

4.1 Considerações Iniciais

O escalonamento de processos sobre plataformas distribuídas/paralelas tem ratificado sua importância na comunidade científica por propiciar um considerável ganho no desempenho final do sistema.

Nesse contexto, inúmeros projetos de software têm sido propostos com o intuito de otimizar o escalonamento em plataformas computacionais com vários elementos de processamento, principalmente as heterogêneas (Milojicic et. al., 1993; Squillante, 1995; Schnor et. al., 1997).

Contudo, o estudo e análise desses projetos demonstram que, embora várias políticas de escalonamento, comprovadamente eficazes, tenham sido propostas, poucas são as que realmente são utilizadas na prática pelos usuários finais (Feitelson & Rudolph, 1996). Dentre os fatores responsáveis por isso, destacam-se a falta de uma nomenclatura coesa, a dificuldade de se relacionar trabalhos diferentes e a falta de uma maior flexibilidade dos objetivos inseridos nos algoritmos (Saphir et. al., 1995; Feitelson & Rudolph, 1996).

Na tentativa de solucionar esses problemas foi proposto o ambiente de escalonamento AMIGO (DynAMical FlexIble SchedulIng EnvirOnment), o qual tem por

objetivo viabilizar, de modo coerente, fácil e organizado, a utilização das melhores políticas e mecanismos, em uma única ferramenta de escalonamento de processos para aplicações paralelas (Souza et. al., 1999b).

O ambiente AMIGO está sendo especificado e parcialmente implementado através de um projeto de doutorado em andamento no grupo de Sistemas Distribuídos e Programação Concorrente do Instituto de Ciências Matemáticas e de Computação-ICMC .

Vários projetos de mestrado e doutorado estão em andamento e visam contribuir para a implementação do AMIGO. O projeto descrito nesta dissertação compreende um desses trabalhos, trazendo uma contribuição importante para a implementação do AMIGO. As informações apresentadas neste capítulo foram obtidas através das publicações (Souza et. al., 1998, 1999a, 1999b, 1999c).

4.2 Ambiente de Escalonamento (AMIGO)

O AMIGO é uma ferramenta de software capaz de agrupar diferentes algoritmos de escalonamento, permitindo que o usuário escolha o algoritmo mais apropriado para a sua aplicação paralela .

O termo flexível do ambiente AMIGO refere-se à capacidade de inserir e escolher a política e o mecanismo mais adequados às necessidades reais do usuário. Para viabilizar a flexibilidade, o AMIGO deve ser totalmente “aberto”, ou seja, definir interfaces claras e objetivas para permitir futuras inclusões.

O termo dinâmico refere-se à capacidade de mudar completamente a política e o mecanismo de escalonamento adotados, em tempo de execução da aplicação. Assim sendo, o termo dinâmico no AMIGO não tem nenhuma relação com o termo dinâmico usado para referenciar os “algoritmos de escalonamento dinâmicos” e sim à capacidade do AMIGO de alterar dinamicamente as políticas que estão sendo usadas no escalonamento durante a execução da aplicação do usuário.

A intenção é retirar a visão algorítmica contida nos algoritmos de escalonamento e, dessa forma, fornecer recursos que apoiem os usuários em sua árdua busca por um escalonador de processos que atenda às suas necessidades.

O AMIGO foi projetado, inicialmente, para abranger aplicações paralelas desenvolvidas para redes de estações de trabalho e com o sistema operacional UNIX (MIMD com memória distribuída), gerenciadas por um ambiente de passagem de mensagem como o PVM ou o MPI (Geist et. al., 1994, 1997; Snir et. al., 1996). Todavia, a sua utilização não está restrita a essa plataforma, podendo ser usado sobre um sistema distribuído, executando aplicações sequenciais, ou então para máquinas paralelas de fato.

4.2.1 A Estrutura do AMIGO

O ambiente AMIGO é estruturado em uma camada superior e em uma camada inferior, conforme mostra a Figura 4.1. Na primeira, encontra-se a interface gráfica, pela qual o usuário terá acesso às várias opções para configuração do ambiente, além de determinar como serão realizadas as políticas de escalonamento.

Na camada inferior estão os módulos responsáveis pelo escalonamento em si, assim como, toda a interface que relaciona as aplicações dos usuários com as políticas de escalonamento disponíveis e atuantes no momento.

A estrutura do AMIGO será detalhada nas seções 4.3 e 4.4 a fim de demonstrar como trabalha cada camada e como ocorre a interação entre as mesmas, principalmente entre o AMIGO e a aplicação paralela.

Figura 4.1 - Estrutura Interna do AMIGO

4.2.2 Utilização do Ambiente AMIGO

Para a construção de um ambiente de escalonamento para aplicações paralelas, com os objetivos que o AMIGO se propõe, deve-se considerar qual é o perfil do usuário que irá operá-lo. Existem duas possibilidades para essa questão: o usuário de uma aplicação paralela, que deseja executar a sua aplicação de modo mais eficiente, ou então o gerente dos recursos

computacionais que supervisiona a execução de várias aplicações paralelas na plataforma sob sua responsabilidade (Souza et. al., 1998).

Para usar o AMIGO, o usuário deve ter consciência da importância de se realizar um escalonamento de processos otimizado, não precisando, obrigatoriamente, saber como essa tarefa é realizada.

Independente de quem seja essa pessoa (usuário de uma aplicação paralela ou gerente), sempre haverá vantagens e desvantagens na escolha feita. Caso o usuário do AMIGO seja o usuário de uma aplicação paralela, esse terá maior facilidade em determinar a política e o mecanismo mais apropriados para a sua aplicação, pois terá o controle total do escalonamento feito para a sua aplicação. Porém, visto que há a possibilidade de se inserir e utilizar várias políticas para o escalonamento, esse usuário pode ter uma visão puramente egoísta da plataforma utilizada e talvez prejudicar outros possíveis usuários.

Caso o usuário do AMIGO seja o gerente da plataforma, o escalonamento deve ser orientado através de uma visão mais democrática dos recursos disponíveis, visto que um gerente preocupa-se não somente com a satisfação de um usuário, mas sim com a satisfação coletiva. Essa visão, entretanto, pode dificultar algumas atividades como, por exemplo, medições de desempenho, onde uma aplicação necessita de um controle total da plataforma utilizada.

Outro fator envolvido com a utilização do AMIGO por um "gerente da plataforma" é a maior transparência obtida, visto que os usuários finais (responsáveis apenas por suas aplicações), estariam utilizando o escalonamento gerenciado pelo AMIGO, sem preocupar-se em como isso está sendo feito de fato.

Acesso à Camada Superior

O usuário, independente de quem seja (usuário de uma aplicação ou gerente), tem acesso ao AMIGO através da interface gráfica pertencente à camada superior. Nesse caso, o usuário pode configurar todo o ambiente de escalonamento, definindo as classes de software, os *benchmarks* dentre outras opções, nas quais a escolha pela política mais adequada será feita.

Em síntese, nessa opção do AMIGO é interessante que o usuário tenha algum conhecimento para que possa configurar o ambiente mais adequado para a sua realidade. É importante salientar que usando apenas essa opção o usuário não fará o escalonamento propriamente dito, pois isso é uma função exclusiva da camada inferior.

Acesso à Camada Inferior

O acesso do usuário à camada inferior se dá através do ambiente de passagem de mensagem utilizado, ou seja, o usuário executa da maneira normal o ambiente de passagem de mensagem e este, por sua vez, deve iniciar a execução do AMIGO, através do AMIGOD (AMIGO Daemon).

No caso particular do PVM, o usuário chama através da linha de comando o PVM, que inicia a máquina virtual e, conseqüentemente, o AMIGOD. O AMIGOD ao iniciar deve ter acesso à camada superior em busca dos arquivos de configuração.

É importante ratificar que o funcionamento da camada inferior independe do funcionamento da camada superior, havendo apenas a necessidade da existência dos arquivos de configuração pois, baseando-se em informações contidas nos mesmos, o AMIGOD realiza sua configuração.

Após a execução e configuração do ambiente de passagem de mensagem o acesso do usuário ao AMIGO se dá por intermédio das aplicações paralelas.

4.3 Camada Superior do AMIGO

A interface gráfica que compõe a camada superior do AMIGO permite a escolha de políticas de escalonamento, assim como sugere qual seria a política mais apropriada a ser adotada, considerando questões como: plataformas, carga atual do sistema e natureza da aplicação.

Para que essas atividades possam ser realizadas, o AMIGO possui depósitos de dados, os quais permitem a inserção de classes de software, do hardware utilizado, das políticas de escalonamento disponíveis e das métricas que serão utilizadas para o monitoramento das políticas de escalonamento do AMIGO.

A inclusão de classes de software foi projetada para integrar políticas de escalonamento com classes de aplicações, viabilizando dessa maneira que cada aplicação seja escalonada com a política de escalonamento mais adequada, visto que não há políticas perfeitas para todos os tipos de aplicações.

Criando classes de software é possível executar e estudar o comportamento dos algoritmos de escalonamento, atuando sobre diferentes tipos de aplicações. Isso permite classificar tipos de aplicações independentemente das características que se deseja abstrair.

A manutenção de um cadastro de hardware possibilita que o mesmo possa ser avaliado antecipadamente sobre questões como, por exemplo, desempenho e localização na rede de comunicação. Dessa maneira é contemplada a heterogeneidade encontrada nas plataformas, principalmente as distribuídas.

A detecção do hardware pode ser manual ou automática. Na detecção manual o usuário indica quais elementos de processamento devem ser considerados para o cadastro. Na detecção automática o AMIGO fornece ao usuário a relação de elementos de processamento disponíveis para serem inseridos ou alterados no cadastro.

A análise da potência computacional de cada elemento de processamento é feita através de *benchmarks* cadastrados pelos usuários. Os *benchmarks* podem apresentar grandes variações, dependendo do que se deseja analisar em uma plataforma. Pode-se analisar um mesmo elemento de processamento, por exemplo, sob a ótica do processador ou da comunicação entre os processadores, usando, para tanto, *benchmarks* diferentes.

Os resultados obtidos com os *benchmarks* são inseridos em um arquivo de configuração e disponibilizados para as políticas de escalonamento. As políticas de escalonamento, com esse arquivo de configuração, contemplam a heterogeneidade existente na plataforma, de maneira simples e direta. A ausência do arquivo de configuração não impede a execução da aplicação paralela, apenas trata todos os elementos de processamento com igual capacidade.

A idéia original é que o usuário tenha nesse arquivo um apoio na tomada de decisão, isto é, para cada tipo de aplicação sejam adotados os valores fornecidos pelo *benchmark*, caracterizado em analisar os recursos mais usados por essa aplicação (veja Figura 4.2). Atualmente, há um *benchmark* implementado neste trabalho de mestrado, sendo usado pelo ambiente AMIGO para contemplar a diferença da capacidade de processamento de todos os *hosts* pertencentes à máquina virtual.

Figura 4.2 - Arquivo de Configuração dos Benchmarks

Outra característica importante no escalonamento é a sua capacidade de monitoramento, pois permite que a eficiência da política utilizada seja comprovada através de métricas bem definidas e inseridas no AMIGO. Por meio do monitoramento da eficiência é possível verificar possíveis problemas que estejam ocorrendo e corrigi-los, se possível, através da utilização de outras políticas.

O monitoramento é viabilizado no AMIGO através da inclusão de métricas que definem qual será o índice de desempenho a ser adotado em cada caso. Essas métricas devem estar relacionadas aos objetivos propostos à plataforma computacional, ou seja, não se tem um único índice de desempenho o qual possa ser empregado em qualquer algoritmo de escalonamento (Feitelson et. al., 1997).

As políticas de escalonamento no AMIGO são determinadas através de um cadastro de políticas disponíveis. Segundo os princípios do AMIGO (já discutidos anteriormente), as políticas devem ser específicas e simples para que, uma vez instanciadas as características da plataforma computacional e da aplicação, atinja-se um desempenho melhor, se comparado às políticas genéricas e “mais pesadas”.

A maneira como as políticas de escalonamento devem ser construídas para a interligação com os demais componentes do AMIGO é abordada na seção sobre a camada inferior do AMIGO (seção 4.4). A escolha da política de escalonamento pode ser feita de três modos distintos: padrão, assistida ou livre.

Com a opção padrão, a política de escalonamento é determinada pela heurística adotada no AMIGO, sem interferência do usuário. A heurística aplicada aqui toma as decisões baseadas nas informações contidas no banco de dados do AMIGO e, dessa forma, considera fatores tais como classes de software, eficiência dos meios de comunicação, distância entre elementos de processamento, carga de trabalho da plataforma, horário de atuação, questões administrativas, entre outros.

A opção assistida permite que o usuário intervenha no processo de seleção da política de escalonamento feita pelo AMIGO. A heurística seguida por essa opção é a mesma da opção padrão, no entanto, o usuário pode dar maior prioridade para determinadas características, mudando a escolha da política.

A opção livre deve ser utilizada preferencialmente pelos usuários que dominam a teoria envolvida na área de escalonamento de processos. Nessa opção o usuário tem total domínio e responsabilidade na escolha da política de escalonamento a ser adotada na distribuição dos processos.

Figura 4.3 - Camada superior do AMIGO - escolha das Políticas de Escalonamento

Independente de qual opção tenha sido adotada, o resultado da escolha da política de escalonamento é depositado em um arquivo de configuração (semelhante ao responsável pelos resultados dos *benchmarks*) (Figura 4.3). Esse arquivo deve conter todas as políticas que se fizerem necessárias para esse escalonamento, que serão intituladas neste trabalho como políticas ativas ou correntes.

Esse arquivo é recuperado pela camada inferior do AMIGO quando este começar a executar ou quando houver mudanças na política de escalonamento empregada. Portanto, a cada alteração desse arquivo de configuração, a camada inferior do AMIGO é avisada e realiza as devidas substituições em tempo de execução.

4.4 Camada Inferior do AMIGO

Enquanto a camada superior do AMIGO responsabiliza-se por toda a configuração envolvida com a flexibilidade do escalonamento de processos, a camada inferior oferece o suporte, através de seus módulos, ao escalonamento flexível e dinâmico proposto.

Dito isso, é importante ressaltar que a camada inferior do AMIGO funciona de maneira independente da camada superior (como já explicado). Os itens pertencentes à camada superior e utilizados na camada inferior são apenas os arquivos de configuração (*benchmark.cfg* e *policy.cfg*) já gerados pela execução dos *benchmarks* e pela escolha das políticas de escalonamento que devem ser ativadas. Caso esses arquivos, por algum motivo, não forem localizados, valores *default* são assumidos, permitindo que a camada inferior funcione e o usuário não seja impossibilitado de executar a sua aplicação.

A camada inferior do AMIGO está estruturada em três módulos bem definidos: o ambiente de passagem de mensagens, o AMIGO *Daemon* (também chamado AMIGOD) e as políticas de escalonamento (Apêndice A).

É importante destacar que este trabalho de mestrado foi responsável pela implementação do AMIGOD, com exceção das funções relativas à migração de processos, além de implementar as mudanças no PVM necessárias para a interface com o AMIGOD, novamente, excluindo-se a parte pertinente à migração de processos. A Figura 4.4 mostra a camada inferior do AMIGO e as partes implementadas neste trabalho (que aparecem em cor escura).

Figura 4.4 - Implementações da Camada Inferior do AMIGOD

4.4.1 Ambiente de Passagem de Mensagens

No AMIGO, o ambiente de passagem de mensagens faz o papel de cliente, visto que, no caso da computação paralela distribuída, os ambientes de passagem de mensagens requisitam ao AMIGO os elementos de processamento necessários à execução de suas aplicações paralelas. Portanto, o ambiente de passagem de mensagem age como a interface entre a aplicação paralela e o AMIGO.

A aplicação paralela do usuário não necessita ter nenhuma linha de código alterada em um programa PVM ou MPI que já esteja executando. Todas as especificações do AMIGO foram realizadas tendo a transparência como uma de suas principais metas. Os exemplos e explicações apresentados aqui fazem referência ao ambiente de passagem de mensagem PVM, mas dada sua proporcionalidade, são válidos também para o MPI.

As mudanças necessárias no ambiente de passagem de mensagens, para que esse possa interagir com o AMIGO são poucas. De modo geral, essas mudanças consistem em:

- ❖ na iniciação e paralisação de novos *hosts* nos ambientes de passagem de mensagens, pois novos AMIGOD's devem ser executados e retirados respectivamente;

- ❖ na atribuição de novas tarefas aos elementos de processamento, onde a aplicação paralela solicita recursos computacionais;

- ❖ na migração de tarefas, através da preempção dessas tarefas após já terem iniciado suas execuções (Franklin & Govindan, 1996).

A migração de tarefas é, sem dúvida, uma das estratégias de escalonamento mais complexas, pois envolve mais detalhes para controlar, tais como, o TID da tarefa (dependente do TID original) e dos descritores de arquivos (*sockets*, arquivos, etc.).

Há diversos trabalhos que implementam migração de tarefas sobre a plataforma PVM. Alguns desses trabalhos são MPVM (Casas et. al., 1995), Dynamic PVM (Overeinder et. al., 1996) e UPVM (Casas et. al., 1994) dentre muitos outros. Entretanto, o problema principal a ser solucionado nessas diferentes implementações são a relação entre o identificador da tarefa (TID) e seu PVMD local e as mensagens que chegam de uma tarefa remota devem ser redirecionadas, pois nenhuma mensagem deve ser perdida e, além disso, todas as mensagens devem ser recebidas na mesma ordem em que foram enviadas, como definido pelo PVM (Casas et. al., 1995).

A migração de tarefas especificada pelo AMIGO, não foi implementada neste projeto de mestrado (como já exposto). Em função disso, não serão apresentados os detalhes envolvidos na sua implementação e, de um modo mais específico, as próximas seções descrevem as demais alterações necessárias no código original do PVM.

Iniciando e Removendo um *Host*

Um AMIGOD é iniciado sempre por intermédio do ambiente de passagem de mensagens, ou seja, o ambiente de passagem de mensagens deve ter seu código alterado, a fim de que o mesmo seja capaz de iniciar o módulo central da camada inferior do ambiente AMIGO.

Quando o PVMD (*PVM Daemon*) começa a sua execução, o AMIGOD é executado através da função *StartAMIGOD()*, inserida na função *main()*, do código do PVMD.

A *StartAMIGOD()* é responsável por criar todo o ambiente necessário para a execução do AMIGOD. Primeiramente, é criado um *port* TCP para a comunicação AMIGOD $\Leftrightarrow$ Ambiente de Passagem de Mensagem, depois executar o AMIGOD com alguns parâmetros de entrada e, caso não seja o primeiro AMIGOD a ser criado, passa também o *port* UDP do AMIGOD para comunicação AMIGOD $\Leftrightarrow$ AMIGOD. Em seguida, aguarda o AMIGOD se configurar e conectar com o ambiente de passagem de mensagem.

Além disso, o *StartAMIGOD()* recebe o *port* UDP criado pelo AMIGOD local para a comunicação entre AMIGOD $\Leftrightarrow$ AMIGOD. O *port* UDP juntamente com o IP local será armazenado e enviado posteriormente para um PVMD remoto, durante a inclusão de um novo processador na máquina paralela virtual e conseqüentemente na inclusão de outro AMIGOD. Com isso, quando um novo AMIGOD for criado, o PVMD passa como parâmetro o endereço IP necessário para a comunicação entre um outro AMIGOD pertencente a máquina virtual. O IP e o *port* UDP são enviados para um novo PVMD por meio do parâmetro “-aIP:UDP” na função *phase1()* no arquivo *startup.c* do código fonte do PVM.

Dessa maneira, após o AMIGOD ter sido iniciado ele cria *sockets* de comunicação TCP, para conectar com o ambiente de passagem de mensagem local e para comunicar-se com as políticas ativas no ambiente. Um *socket* UDP é criado para propiciar a comunicação desse AMIGOD com os outros AMIGOD's pertencentes a máquina virtual.

Após criar os mecanismos de comunicação, o AMIGOD configura suas estruturas de dados e inicia as políticas ativas que estão no arquivo de configuração. Isso é feito independentemente do PVMD e em nada afeta sua execução normal.

Por outro lado, quando o PVMD é finalizado, ele chama a função *RemoveAMIGOD()* que tem por objetivo mandar uma mensagem do tipo AM_REM para o AMIGOD local, informando sua morte e removendo os mecanismos IPC usados para comunicação. Essa função é acrescentada no início da função original do PVM *pvm bailout()*.

Visto que não há o conceito de mestre/escravo no AMIGO, a “morte” de um AMIGOD não acarreta na finalização do ambiente de escalonamento como um todo, entretanto, caso a morte desse AMIGOD tenha sido provocada pela “morte” do PVM mestre, isso implica na finalização de toda a máquina virtual, pois não há no PVM 3.3.11 tratamento para essa condição.

Escalonando Novas Tarefas

O escalonamento será ativado, no caso do PVM, sempre que houver a execução de uma função *pvm_spawn()* na aplicação paralela, pois essa função é responsável por criar novas tarefas nos *hosts* pertencentes à máquina virtual. Para um melhor entendimento, é importante apresentar o protótipo da função *pvm_spawn()*.

int numt = pvm_spawn (char *task, char **argv, int flag, char *where, int ntasks, int *tids), onde:

task = *string* de caracteres contendo o nome do arquivo executável do processo PVM para ser iniciado;

argv = ponteiro para um *array* de argumentos para o programa executável, com o fim do array apontado para NULL;

flag = inteiro que especifica opções do *spawn*;

where = *string* de caracteres especificando onde iniciar o processo PVM, dependendo do valor de *flag*, “*where*” pode ser o nome de um *host* ou de uma classe de arquiteturas PVM;

ntasks = inteiro especificando o número de tarefas que devem ser criadas;

tids = *array* de inteiros que retorna os TID’s das tarefas PVM criadas por essa chamada do *pvm_spawn()*;

numt = inteiro que retorna o número de tarefas criadas pelo *pvm_spawn()*;

No PVM tradicional sempre que há uma chamada para a função *pvm_spawn()*, essa função empacota os dados e os envia para o PVMD local. No PVMD local essa mensagem é recebida pela função *TM_spawn()*, que por sua vez deve desempacotar a mensagem e preparar uma cópia da estrutura *host table* com os *hosts* que irão receber as tarefas.

Tabela 4.1 – Relação de *flags* usados na função *pvm_spawn()*

Flag	Valor	Significado
PvmTaskDefault	0	PVM escolhe Qualquer máquina para iniciar as tarefas.
PvmTaskHost	1	Where especifica um host particular.
PvmTaskArch	2	Where especifica um tipo de arquitetura.
PvmTaskDebug	4	Inicia processos sobre Debugger.
PvmTaskTrace	8	Processo produzirá dados PVM.
PvmMppFront	16	Inicia processos sobre front-end.
PvmHostCompl	32	Usa conjunto de host complementar.

Essa estrutura contendo os *hosts* é montada de acordo com o *flag* ajustado (Tabela 4.1) na função original. Em seguida, diante dessa relação de *hosts*, é chamada a função *assign_tasks()* que, através do algoritmo *round-robin*, define quantas tarefas devem ser executadas por cada *host*. Finalizando é enviada uma mensagem para cada *host* da relação, para que estes comecem a executar as novas tarefas, o que é feito através da função *DMEXEC()* no *host* remoto. A Figura 4.5 mostra graficamente todos os passos realizados pelo PVM tradicional na busca dos *hosts* que recebem as tarefas.

Figura 4.5 - Preparação dos *hosts* através da função *pvm_spawn()*

Procurando alcançar uma transparência total para o usuário final, foram adotados o *flag PvmTaskDefault* e o *PvmTaskArch* como *flags* usados através do AMIGO. Esses *flags* foram escolhidos porque é através deles que o PVM usa a sua política *round-robin*. Atuando sobre esses *flags* o AMIGO substitui essa política do PVM por uma mais adequada, segundo os critérios já mencionados.

Como pôde ser observado no protótipo da função *pvm_spawn()*, essa função passa como parâmetro além do nome do arquivo executável a quantidade de instâncias (tarefas) a serem criadas sobre a máquina virtual.

Portanto, estando essa função configurada com *PvmTaskDefault()*, é chamada a função *GetHostsFromAMIGOD()*, a qual é responsável por enviar uma mensagem com o código AM_SCHED para o AMIGOD local, solicitando uma relação de *hosts* apropriados a receber novas tarefas. Essa mensagem contém o nome do código executável, o número de processos PVM a serem criados e o tipo de arquitetura.

Figura 4.6 - Preparação dos *hosts* através da função *pvm_spawn* alterada()

O AMIGOD após receber essa mensagem manda uma mensagem com o código AP_SCHED requisitando para a política adequada a relação de máquinas que devem ser usadas

no escalonamento das tarefas. Quando a lista está pronta, a política envia para o AMIGOD local a relação de máquinas e o AMIGOD, por sua vez, retransmite essa mensagem com o código AM_SCHEDACK para o PVM. Esse por sua vez, desempacota a mensagem recebida e retira da mesma a lista das máquinas que serão usadas como receptoras das novas tarefas. A Figura 4.6 apresenta essas alterações feitas no código tradicional do PVM.

As alterações feitas no *pvm_spawn()* são bastante simples e claras, porém, são decisivas para mudar a política de escalonamento usada pelo PVM tradicional, o qual escalona as tarefas através do algoritmo *round-robin* entre os *hosts* usados para o escalonamento. A principal vantagem desse método são as poucas mudanças requeridas no código do PVM, possibilitando uma adaptação fácil e clara em novas versões do código PVM.

4.4.2 AMIGO Daemon (AMIGOD)

O AMIGOD é o componente central da camada inferior do AMIGO, e é o responsável por interligar o ambiente de passagem de mensagem às políticas de escalonamento ativas. O AMIGOD, portanto, não realiza o escalonamento em si, ele apenas atua como um roteador e controlador de mensagens entre os diferentes componentes envolvidos no escalonamento.

Outra grande característica do AMIGOD é que ele é completamente distribuído, ou seja, não existe uma hierarquia (como mestre/escravo) entre os AMIGOD's, os quais devem ser executados em todos os elementos de processamento alvos do escalonamento e podem realizar quaisquer tarefas sem distinção de poderes.

Apesar da possibilidade de se aumentar a complexidade do projeto e a sobrecarga de comunicação entre os módulos, um AMIGOD distribuído permite uma maior tolerância a falhas e evita “gargalos” no sistema. Além disso, as políticas ativas têm permissão para trocarem mensagens diretamente entre elas sem ter que envolver o AMIGOD. Todavia, a responsabilidade dessas mensagens é toda da política, pois o AMIGOD não tem conhecimento das mesmas.

A estrutura interna do AMIGOD é formada basicamente por alguns mecanismos de comunicação entre processos (IPC) responsáveis pelas trocas de mensagens e por estruturas de

dados responsáveis pelas tabelas de roteamento e pelas políticas ativas no momento. O código do AMIGOD foi projetado para ser pequeno, porém eficiente na sua tarefa de dar suporte ao ambiente AMIGO.

Protocolos de Comunicação

A comunicação entre processos realizada pelo AMIGOD é baseada em *sockets* TCP (*Transmission Control Protocol*) e *sockets* UDP (*User Datagrama Protocol*) através do protocolo de comunicação TCP/IP (Comer, 1995). Dessa maneira, todos os *hosts* pertencentes à máquina virtual e conseqüentemente ao AMIGO são capazes de se conectarem através desses mecanismos de comunicação. A Figura 4.7 apresenta o esquema de comunicação adotado pelo AMIGO.

Como pôde ser notado, no AMIGOD a comunicação entre os módulos pertencentes ao mesmo elemento de processamento são feitos através de *sockets* TCP e a comunicação feita entre elementos de processamento distintos é feita através de *sockets* UDP (Apêndice B).

Figura 4.7 - Relacionamento entre os módulos do AMIGO

A escolha do protocolo TCP para os módulos pertencentes a um mesmo elemento de processamento deve-se, principalmente, em função da confiabilidade e a reduzida quantidade de módulos. Por outro lado, para a comunicação entre os AMIGOD's optou-se por *sockets* UDP, e essa escolha foi feita por dois motivos principais,

O primeiro diz respeito à “escalabilidade”, pois em uma máquina com n *hosts*, cada AMIGOD deveria ter $n-1$ conexões, onde cada transmissão aberta com TCP consome um descritor de arquivos no PVMD e alguns sistemas operacionais limitam o número de arquivos abertos. Um *socket* UDP aberto, entretanto, pode comunicar-se com qualquer outro *socket* UDP remoto. O segundo é tempo extra (sobrecarga) gasto para fazer as conexões necessárias com TCP. Essa mesma escolha (pelo TCP e/ou UDP) é feita pelo PVM (Geist et. al., 1994).

A comunicação na camada inferior do AMIGO deve considerar 3 (três) conexões: AMIGOD $\leftrightarrow$ POLITICA, AMIGOD $\leftrightarrow$ AMBIENTE e AMIGOD $\leftrightarrow$ AMIGOD. As mensagens trocadas entre AMIGOD $\leftrightarrow$ AMIGOD são identificadas através de macros cujo prefixo é AA_, as mensagens entre AMIGOD $\leftrightarrow$ POLITICA são identificadas pelo prefixo AP_, e finalmente, as mensagens entre AMIGOD $\leftrightarrow$ AMBIENTE comunicam-se através de mensagens rotuladas por macros iniciadas por AM_.

Cabeçalho das Mensagens

As mensagens trocadas na camada inferior do AMIGO são compostas por um cabeçalho (*header*) e um conteúdo (*body*). O cabeçalho tem a finalidade de identificar a mensagem, independente de qual seja a sua origem e/ou destino.

Conforme mostra a Figura 4.8 o cabeçalho é formado por uma estrutura com 16 (dezesesseis) *bytes*, dividida em 4 (quatro) campos, os quais representam o código da mensagem, o módulo destino, o módulo origem e o tamanho da mensagem.

O código da mensagem é um campo que ocupa 2 (dois) *bytes* e sua finalidade é identificar a função correta que deverá ser chamada para tratar a mensagem (por exemplo, se é uma requisição para adicionar novos *hosts*, enviar dados entre políticas, confirmação de um serviço executado, entre outras).

Figura 4.8 - Cabeçalho das Mensagens trocadas no AMIGO

O módulo origem e destino são estruturas que ocupam 6 (seis) *bytes* cada. O primeiro *byte* define quais são os módulos envolvidos na comunicação, isto é, a letra A define um AMIGOD. A letra P determina uma política e por último, a letra M define um ambiente de passagem de mensagem.

Os próximos 4 (quatro) *bytes* indicam o endereço IP da máquina que está participando da troca de mensagem, o último campo só é preenchido se a comunicação estiver sendo feita entre alguma política, pois é usado para identificar dentre as políticas ativas qual está enviando e/ou recebendo a mensagem.

No último campo são empacotados 2 (dois) *bytes* para o tamanho da mensagem o qual, como o próprio nome diz, indica o tamanho da mensagem a ser recuperada, além de ser usado para identificar se a mensagem foi recebida e/ou enviada corretamente.

Tabela de Hosts

O AMIGOD manipula uma tabela de *hosts* que tem a finalidade de manter a configuração da máquina virtual. Essa tabela contém, além da configuração da máquina, a identificação de cada AMIGOD na máquina virtual, pois o AMIGOD é identificado através do endereço IP de cada *host*, pois sempre haverá, para cada máquina virtual, apenas um AMIGOD em cada *host*.

Com essa tabela, o AMIGOD tem condições de manipular conjuntos de *hosts* que poderão vir a ser considerados candidatos em potencial para iniciar as tarefas geradas pelo usuário. Essa tabela é implementada através da estrutura *amigod_table*, definida a partir do arquivo *amigod.h* (Figura 4.9).

Essas tabelas são mantidas sincronizadas em toda a máquina virtual, embora isto não possa ser verdade durante todo o tempo de execução. Quando ocorrem determinados problemas com um AMIGOD e ele é retirado de operação, podendo demorar algum tempo para que os outros AMIGODs notem a sua falta. Durante esse intervalo de tempo as tabelas de *hosts* não estão sincronizadas.

Figura 4.9 - Tabela de *hosts* do AMIGOD

Essa tabela é criada localmente em cada AMIGOD que inicia a sua execução. O AMIGOD verifica, logo após ser iniciado, se recebeu um parâmetro com a *string* contendo IP: PORT UDP e, caso não tenha recebido, o AMIGOD supõe que é o primeiro AMIGOD “no ar”. Por outro lado, o recebimento desse parâmetro, indica que há mais algum AMIGOD pertencente à máquina virtual, embora o AMIGOD não saiba ainda quantas existem.

Após detectar a existência de mais um AMIGOD, o AMIGOD local chama a função *PutHost()*, que tem por finalidade atualizar a tabela de *hosts* locais com os *hosts* remotos. Essa função empacota as informações do *host* local e manda uma mensagem com o código AA_ADD para o *host* remoto avisando da sua existência e solicitando a tabela de roteamento do *host* remoto. O AMIGOD remoto, ao receber essa mensagem, chama a função *GetHost()* que é responsável por inserir o novo *host* na tabela local, assim como empacotar todos os elementos da tabela local, menos é claro, o próprio AMIGOD local e repassá-la ao AMIGOD remoto de quem recebeu a mensagem AA_ADD.

Além disso, a função *GetHost()* ainda deve fazer um *broadcast* para todos os *hosts* inseridos em sua tabela, avisando a existência do novo *host*. No final dessas mensagens, todas as

máquinas terão uma tabela local contendo informação de todas as máquinas pertencentes à máquina virtual.

A operação inversa ocorre sempre que por algum motivo um AMIGOD sair do ar. Entretanto, nesse caso, após um AMIGOD detectar a morte de algum AMIGOD remoto, esse chama a função *DetectRemoteAMIGOD()*, a qual é responsável em mandar uma mensagem com o código AA_REM para todos os *hosts* inseridos em sua tabela local, indicando a morte do AMIGOD remoto. Os AMIGODs ao receberem essa mensagem chamam a função *RemoveRemoteAMIGOD()*, que removerá da tabela local o *host* que teve sua morte detectada.

Tabela de Políticas

O AMIGOD, além da tabela de *hosts*, mantém também informações sobre todas as políticas ativas. Essa tabela é mantida através de uma estrutura estática preenchida através da função *GetActivePolicy()*, que busca, a partir do arquivo de configuração *policy.cfg*, quais as políticas devem estar ativas no momento.

Essa tabela possui, portanto, todas as informações necessárias para que o AMIGOD conheça as políticas ativas que estão sobre sua supervisão. A Figura 4.10 mostra a tabela de políticas em detalhes.

Figura 4.10 - Tabela de Políticas do AMIGOD

4.4.3 Políticas Ativas

Outro módulo da camada inferior são as políticas de escalonamento, as quais foram implementadas como processos independentes, para permitir a característica dinâmica do AMIGO. A política de escalonamento, portanto, pode ser substituída sem que a aplicação necessite ser interrompida ou, nem mesmo, informada à respeito da troca (Souza et. al., 1999b).

Embora o módulo AMIGOD seja completamente distribuído, não necessariamente as políticas de escalonamento também devam ser. As políticas podem, perfeitamente, ser centralizadas, sendo justamente essa flexibilidade “a alma” do AMIGO.

As políticas de escalonamento, implementadas independentemente do AMIGOD, podem possuir quaisquer finalidades e características, desde que obedeçam alguns critérios tais como: sejam políticas dinâmicas, ou seja, atuantes em tempo de execução da aplicação do usuário, e sigam o protocolo de comunicação especificado para a troca de mensagens entre os módulos do AMIGO. As características mais marcantes das políticas implementadas para o AMIGO são:

- ❖ a sua independência do ambiente de passagem de mensagens utilizado, visto que as políticas não têm acesso a outro módulo que não seja o AMIGOD. Uma política funciona para o AMIGO e não para o ambiente de passagem de mensagens;

- ❖ o seu código pequeno e específico, visto que o objetivo é se ter políticas concisas porém eficientes.

Em princípio, a idéia é se ter inúmeras políticas implementadas e que essas sejam objetivas e específicas para um determinado tipo de problema, possibilitando assim políticas ótimas para cada caso. Dependente do tipo de aplicação e do *status*, o ambiente escolhe a melhor política para cada caso.

Os mecanismos, responsáveis por implementar certas tarefas específicas à plataforma utilizada (como por exemplo coletar os índices de carga), estão inclusos nos códigos das políticas nesta primeira fase do projeto. Entretanto, esses mecanismos estão bem modularizados e documentados, facilitando assim a sua manutenção.

4.5 Considerações Finais

O AMIGO é uma proposta que tem a finalidade de diminuir a grande distância entre a teoria e a prática na área de escalonamento de processos.

Para isso o ambiente AMIGO vem ao encontro das soluções aos problemas encontrados na área, tais como, falta de uma nomenclatura coesa, dificuldades de se relacionar o grande números de trabalhos diferentes e principalmente a falta de uma maior flexibilidade dos objetivos inseridos nos algoritmos.

A flexibilidade do AMIGO é alcançada através das inúmeras políticas que podem ser implementadas e direcionadas para resolver problemas específicos, aumentando potencialmente, portanto, o desempenho final do sistema.

Além disso, os detalhes de implementação e as decisões tomadas no escalonamento podem ser totalmente transparentes para o usuário final. Através de uma interface gráfica, o responsável pela plataforma computacional pode tomar todas as decisões necessárias para escalonar a aplicação paralela desejada.

O próximo capítulo apresenta a primeira política desenvolvida para o ambiente AMIGO, a qual é voltada para o escalonamento de aplicações paralelas que se caracterizam como *CPU-bound*, por realizarem muito processamento e pouca comunicação. Essa política foi construída seguindo a especificação adotada no AMIGO, a fim de que haja uma completa interação entre os mesmos.

Política de Escalonamento - DPWP

O presente capítulo aborda a política de escalonamento DPWP. Primeiramente são apresentadas suas características mais relevantes e sua classificação, segundo alguns autores. Em seguida, são apresentados os pontos mais importantes de seu projeto. Para finalizar, detalhes pertinentes à sua implementação, são abordados.

5.1 Considerações Iniciais

O escalonamento de processos pode ser realizado de diversas maneiras, dependendo dos objetivos almejados e das políticas empregadas. Dentre esses objetivos, encontra-se o balanceamento de cargas, que tem como finalidade distribuir as cargas entre os elementos do sistema para que o mesmo possa alcançar o maior equilíbrio possível.

Dessa maneira, o algoritmo de balanceamento de cargas provê um aumento considerável no desempenho final do sistema, visto que a otimização do sistema e um melhor desempenho estão relacionados com a melhor utilização da capacidade de processamento instalada (Feitelson et. al., 1997).

Entretanto, a escolha correta da política de balanceamento de cargas empregada influencia decisivamente o desempenho final da plataforma computacional, pois, caso a política de escalonamento adotada não seja otimizada, pode, em algumas situações, comprometer o desempenho global, devido à sobrecarga necessária em busca de uma melhor distribuição (Shirazi et. al., 1995; Feitelson & Rudolph, 1996).

Na literatura disponível encontram-se vários artigos propondo algoritmos de escalonamento. Isso demonstra a importância e interesse sobre o assunto, porém, dificulta uma análise mais apropriada sobre o que existe e, quais seriam as vantagens reais no desenvolvimento de uma nova abordagem (Kaplan & Nelson, 1994; Harchol-Balter et. al., 1997; Harchol-Balter & Downey, 1997).

Todavia, diferente das demais políticas de balanceamento de cargas, este projeto apresenta a DPWP (*Dynamic Policy Without Pre-emption*), que foi implementada e especificada para evitar a ocorrência de problemas comuns associados aos escalonadores de processos, tais como (Grant & Skjellum, 1992): sobrecarga do algoritmo, uso de informações velhas e imprecisas e redução no número excessivo de mensagens transferidas na rede.

5.2 Política de Escalonamento (DPWP)

A DPWP é um escalonador desenvolvido com o objetivo de melhorar e facilitar a atividade de balanceamento de cargas em ambientes computacionais distribuídos sobre uma rede de estações de trabalho (Araújo et. al., 1999b).

O algoritmo DPWP, de modo geral, considerou em seu projeto algumas questões importantes e que devem ser tratadas por todos os algoritmos de escalonamento. Por exemplo, uma proposta de algoritmo de escalonamento tem por meta atuar sobre certos tipos de aplicações, tentando-se prever os tipos e a quantidade de trabalho que irão ocorrer, pois dessa maneira, o algoritmo proposto se torna mais eficiente e eficaz na distribuição de cargas sobre uma classe de software em particular, ao invés de tentar ser um algoritmo de escalonamento genérico, capaz de balancear todas as classes de software (Ferrari & Zhou, 1987; Souza et. al., 1999b).

Segundo trabalhos apresentados por Harchol-Balter & Downey (Harchol-Balter & Downey, 1997) e por Kunz (Kunz, 1991), as aplicações geralmente podem ser classificadas em *CPU-bound* e *I/O-bound*. A primeira se caracteriza por usar muito processamento e pouca comunicação, e a segunda por representar as aplicações que realizam bastante chamadas aos dispositivos de entrada e saída.

Dessa maneira, o algoritmo aqui apresentado está voltado para aplicações caracterizadas como *CPU-bound* e, portanto, preocupado em distribuir a carga nas máquinas portadoras da melhor potência computacional, visto que os dispositivos de entrada e saída não interferem de maneira significativa no desempenho final do sistema.

Após definir a classe de software, outras duas tomadas de decisões se fizeram necessárias no decorrer do projeto do algoritmo DPWP: a primeira diz respeito à escolha de um algoritmo de escalonamento estático ou dinâmico e a segunda está relacionada à preempção ou não do algoritmo.

Vantagem oriunda dos algoritmos de escalonamentos dinâmicos é a não necessidade de informações sobre as aplicações antes que essas sejam executadas, ou seja, no algoritmo dinâmico, o que importa é o estado do sistema no momento em que as tarefas começarem a executar. Essa característica foi relevante no momento de se optar por trabalhar com um algoritmo dinâmico ao invés do algoritmo estático.

Optou-se por implementar um algoritmo dinâmico, por este adaptar-se às mudanças ambientais durante a execução dos programas (Shirazi et. al., 1995; Jackson & Humphres, 1997). Devido à adaptação do escalonamento dinâmico, esse método é largamente empregado em ambientes multiusuários, como os sistemas distribuídos, haja vista a grande variação de tipos de usuários e tipos de aplicações (Xu & Lau, 1997).

A Segunda decisão tomada no projeto do algoritmo DPWP, diz respeito à implementação de escalonamento de processos sem migração (algoritmo sem preempção), ao invés de implementar-se escalonamento de processos com migração (algoritmo preemptivo ou com preempção) (Shirazi et. al., 1995; Dantas, 1996; Alfaro et. al., 1997).

A escolha pelo algoritmo com preempção não foi adotada, pois migrar uma tarefa consiste em transferir sua execução de uma máquina para outra, e isso é uma atividade muito

cara, pois para que a tarefa possa ser restabelecida na nova máquina é necessário que seu estado⁶ seja transferido (Theimer & Hayes, 1991; Shivaratri et. al., 1992) (Figura 5.1).

Figura 5.1 - Transferência do estado de uma máquina para outra

Em suma, a escolha foi feita pelo algoritmo sem preempção por não ocasionar no sistema grande sobrecarga de comunicação, e com isso propiciar uma carga ainda maior no sistema.

A DPWP se caracteriza também pela sua capacidade em contemplar a heterogeneidade dos sistemas distribuídos, e isso é uma característica que a difere das demais, pois poucas são as políticas de escalonamento que se preocupam com essa diferença. Essa foi uma preocupação no projeto da DPWP, pois a mesma não adota o conceito de algoritmo sem preempção, com isso, caso a máquina escolhida não seja a melhor máquina a atividade de escalonamento não poderá ser refeita enquanto as tarefas distribuídas estiverem executando.

⁶ O estado de uma tarefa compreende todas as informações essenciais para a retomada dessa na nova máquina, como por exemplo memória virtual, mensagens, identificadores dos arquivos, *buffers* de entrada e saída.

5.3 Componentes do Algoritmo DPWP

No projeto do algoritmo DPWP, optou-se por seguir as idéias propostas por Shivaratri (Shivaratri et. al., 1992), o qual propõe dividir o algoritmo de distribuição de cargas em 4 (quatro) componentes (como visto na seção 3.3): política de transferência, política de seleção, política de localização e política de informação.

A adoção dessa divisão tem grande importância na definição da DPWP, pois viabiliza, em grande parte, a sua simplicidade e eficácia em tornar o programa modular, claro e objetivo. Isso possibilita um futuro reuso do código fonte por outras políticas, assim como permite um menor grau de dificuldade na alteração e manutenção da DPWP.

A política de transferência, no caso particular da DPWP, assume um *host* apto para participar do escalonamento de processos sempre que este está com a sua carga acima de um limite estipulado. Dessa maneira, a política de transferência é iniciada sempre por um *host* sobrecarregado (transmissor), tentando enviar uma tarefa para um *host* ocioso, ou menos sobrecarregado. Essa decisão foi tomada por ser considerada a mais natural diante da ausência de preempção adotada pela DPWP e pelas características inerentes aos ambientes de passagem de mensagens. Esses ambientes geralmente se caracterizam por receber um certo número de tarefas e precisar distribuí-las de uma única vez.

A política de seleção na DPWP é realizada de maneira automática, ou seja, todas as tarefas recentemente criadas no ambiente de passagem de mensagem são consideradas aptas a serem escalonadas, fazendo com que o algoritmo de escalonamento escolha o processador onde a tarefa será executada, antes que as mesmas possam iniciar sua execução, justificando dessa forma a ausência de preempção no nome do algoritmo (*Without Preemption*).

Quanto à política de localização, a DPWP assume a decisão baseada na técnica da menor carga, ou seja, seleciona o processador com a menor carga corrente. Quando houver necessidade de escolher para qual processador a carga será alocada, o algoritmo DPWP verifica dentre todas as máquinas pertencentes ao sistema, qual máquina está no estado de ociosa e dentre essas qual possui a menor carga, pois essa será então a escolhida.

Essa decisão é tomada a partir de uma tabela que contém a carga de todos os *hosts* pertencentes à máquina virtual paralela. A atualização e manutenção dessa tabela é realizada pela própria DPWP (seção 5.6.2).

A política de informação é crucial para o desempenho final de qualquer algoritmo de escalonamento, pois apresenta um alto custo em relação às demais políticas. O algoritmo DPWP implementa a política de informação dirigida por mudança de estado. Essa escolha foi adotada por ser, dentre todas as políticas de informação (periódica, por mudança de estado e por demanda), a mais adequada à realidade do algoritmo DPWP. Além disso, essa política tende a produzir menos comunicação pela rede, diminuindo assim a sobrecarga e a latência produzidas pelas trocas dessas mensagens.

5.4 Classificação do Algoritmo DPWP

Esta seção pretende localizar (classificar) a DPWP entre todas as classificações abordadas na seção 3.2 do capítulo 3 (três), objetivando apresentar o relacionamento deste trabalho com os demais trabalhos da área.

5.4.1 Classificação de Casavant

Segundo a classificação hierárquica apresentada por Casavant & Kuhl (Casavant & Kuhl, 1988), o algoritmo DPWP é um algoritmo de escalonamento global, dinâmico, fisicamente distribuído e não cooperativo. A Figura 5.2 mostra a classificação de Casavant segundo a ideologia do algoritmo DPWP.

Figura 5.2 - Classificação de Casavant segundo a DPWP

A DPWP é classificada como global, pois o algoritmo é executado em um sistema com vários processadores e, conseqüentemente, corresponde à tomada de decisão de qual elemento de processamento executará um determinado processo. É dinâmica porque todas as tomadas de decisão são feitas em tempo de execução, fisicamente distribuída pois todos os *host* pertencentes ao sistema têm o mesmo poder de decisão. Em seguida a DPWP, é considerada cooperativa, pois a tomada de decisão de um nó leva em consideração os resultados que isso possa causar sobre o sistema.

Quando a DPWP escolhe os *hosts* para o escalonamento é levado em consideração a situação e o estado dos demais *hosts* no sistema. Dessa maneira, são escolhidas as melhores máquinas e assim o sistema diminui o risco de tornar-se desbalanceado.

Para finalizar, a DPWP se classifica como sub-ótimo, pois a geração de escalonamentos ótimos é um problema do tipo NP-completo (Lewis, 1996). Nesse tipo de problema, só é possível gerar soluções ótimas em casos específicos (por exemplo, quando o tempo de execução de todas as tarefas é igual e somente dois processadores são utilizados). Em

casos gerais, isso é muito difícil, visto que um problema NP-completo requer um período de tempo polinomial (exponencial) para ser solucionado.

A DPWP adota a abordagem heurística, pois os métodos heurísticos utilizam regras especiais para conduzir o processo de escalonamento na direção aproximada de uma solução ótima. Esses métodos representam os algoritmos mais bem estruturados do processo de escalonamento, já que as heurísticas são baseadas no conhecimento produzido por inúmeras pesquisas feitas sobre essa área (Shirazi et. al., 1995).

5.4.2 Classificação de Feitelson

Seguindo a classificação apresentada por Feitelson & Rudolph (Feitelson & Rudolph, 1995), o algoritmo de escalonamento DPWP é classificado como um algoritmo realizado em tempo de execução (*Runtime System*).

A idéia de algoritmo realizado em tempo de execução apresentada por Feitelson & Rudolph segue a mesma linha de raciocínio do algoritmo dinâmico proposto anteriormente, contudo, com a diferença de que o mesmo deve ser obrigatoriamente feito em nível de usuário.

5.4.3 Classificação de Lüling

Nessa classificação o algoritmo DPWP é um algoritmo do tipo DLMG, pois a decisão de transferência de processos é adotada baseando-se no estado do processo local, e a tarefa selecionada para participar da distribuição de cargas pode ser transferida para qualquer *host* pertencente ao sistema. Em princípio, caso o processador local seja considerado um transmissor o mesmo inicia o processo de distribuição de carga, e essa carga será transferida para a melhor carga do sistema, independente da localização dessa máquina.

Além dessa abordagem, Lüling (Lüling et. al., 1993) ainda classifica os algoritmos pela iniciativa da distribuição da carga, nesse caso a DPWP é classificada como um algoritmo iniciado pelo transmissor, pois a atividade de distribuição da carga é sempre tomada por um nó

que está sobrecarregado e procura um nó ocioso para transferir a sua mais nova tarefa. A Figura 5.3 apresenta a localização da DPWP segundo a classificação de Lüling.

Como apresentado acima, a DPWP é um algoritmo de escalonamento de processos iniciado pelo transmissor, pois imediatamente após uma nova tarefa ter sido gerada pela aplicação paralela, inicia-se o processo de busca por uma máquina capaz de receber a carga relacionada com a nova tarefa.

Figura 5.3 - Classificação de Lüling adaptada para DPWP

5.5 Pontos Importantes no Projeto

No projeto do escalonador DPWP foram abordadas algumas particularidades que o diferem das tantas políticas de escalonamento propostas na literatura. Esta seção mostra esses pontos em maiores detalhes.

5.5.1 Índice de Cargas da DPWP

Há na literatura disponível um consenso quanto à importância do índice de carga sobre o desempenho final do escalonamento. Portanto, um bom índice de carga deve ser capaz de refletir precisamente o estado da máquina; além disso, o custo gasto para obtenção desse índice não deve interferir no desempenho do sistema (Ferrari & Zhou, 1987; Kunz, 91; Dantas, 1996).

Com isso, o índice de carga deve estar voltado para o objetivo ao qual se propõe o algoritmo de escalonamento de processos. Nesse caso particular, como o escalonador DPWP se propõe a escalonar aplicações paralelas caracterizadas como *CPU-bound*, o índice de carga mais adequado e que melhor reflete o estado do sistema é o número de processos prontos na fila da CPU. Pois, dessa forma, o índice de carga está representando exatamente o recurso mais importante para o tipo de aplicação que está sendo escalonada.

Esse índice de carga é obtido através da função *getindexload()*. Essa função foi implementada a partir de alterações feitas no código fonte do programa “*top*”, usado pelo UNIX para fornecer dados estatísticos sobre o estado da máquina local. Esse programa fornece, por exemplo, a porcentagem de utilização da CPU, a porcentagem de utilização da memória, o número de processos prontos para executar na fila da CPU, dentre outras informações.

Assim sendo, houve a necessidade de se alterar o código a fim de que o mesmo não viesse a provocar sobrecarga no sistema, na captura de informações que não seriam aproveitadas na DPWP.

Entretanto, em se tratando de redes de estações de trabalho, o fato de tomar-se como índice de carga um valor obtido instantaneamente pode acarretar em uma diminuição do desempenho final, pois valores incompatíveis com a realidade do sistema podem ser capturados, ou seja, se em algum instante o sistema sofrer uma flutuação e se a carga vier a ser capturada exatamente nesse momento, estará representando mais ruído de informações de carga do que informações de cargas válidas (Ferrari & Zhou, 1987).

Todavia, decidiu-se adotar como índice de carga da política DPWP o tamanho médio da fila de processos na CPU nos últimos 20 (vinte) segundos. Essa média é calculada através da

função *get_averageload()*, essa por sua vez calcula a média através da implementação da técnica da “janela deslizante”.

Essa técnica tem como fundamento calcular a média sobre um conjunto de dados, onde o valor mais antigo é sempre substituído pelo valor mais novo e assim sucessivamente. Por exemplo, tem-se um vetor com 4 (quatro) elementos, em um primeiro instante o índice de carga será o valor capturado de maneira aleatória, pois o vetor está vazio.

No segundo instante, a carga capturada novamente de forma aleatória é armazenada na segunda posição do vetor, e a média agora será os dois valores somados e divididos pelo número de elementos no vetor (dois). E assim sucessivamente, até que o vetor venha a ser preenchido. Em um quinto instante a carga capturada será armazenada na posição onde está o primeiro valor obtido. E em um sexto instante, será armazenada na posição da segunda carga guardada, e assim sucessivamente.

Dessa forma, a média será sempre calculada sobre os valores mais recentes, descartando-se, portanto, os valores mais antigos.

Ao determinar-se a carga da máquina local, há a necessidade de se determinar se esse índice obtido é bom ou não para o sistema. Para isso, deve-se determinar se essa máquina está ociosa, carregada ou moderada.

O uso de limites é uma prática comum em algoritmos de escalonamento de processos, devido à imprecisão em definir-se termos como máquinas sobrecarregadas e máquinas ociosas. Além disso, os limites são usados frequentemente em métricas para avaliar a carga de um *host* (Dantas, 1996).

Os limites adotados pela DPWP para classificar um *host* como sobrecarregado, ocioso e moderado foram baseados na quantificação feita por Dantas (Dantas, 1996), cuja plataforma usada se assemelha à plataforma usada para implementar o DPWP. A Tabela 5.1 demonstra os valores usados no algoritmo DPWP para delimitar as máquinas como sobrecarregadas, ociosas e moderadas.

Tabela 5.1 - Características do índice de Carga

Estado da Máquina	Carga Média	Utilização da CPU
Sobrecarregada (Heavy -H)	1.8 - 3.0	41% - 50%
Moderada (Moderate -M)	1.0 - 1.7	21% - 40%
Ociosa (Light-L)	0.0 - 0.9	0% - 20%

5.5.2 Heterogeneidade

No projeto da DPWP houve uma preocupação em contemplar a heterogeneidade do sistema, em busca de um balanceamento mais justo entre as diversas máquinas, pois não se pode distribuir a mesma carga para todas as máquinas do sistema, se existem diferenças entre elas.

Todavia, a heterogeneidade é um conceito amplo que se aplica tanto a hardware quanto a software. No hardware há inúmeros recursos que podem ser contemplados; no caso da DPWP, foi avaliado apenas a diferença de potência computacional, dada sua característica de atuar para aplicações do tipo *CPU-bound*.

A DPWP trata essa heterogeneidade, buscando informações a partir de um arquivo de configuração, que contém a potência relativa de cada máquina do sistema, valores que traduzem a capacidade de processamento. Após essa informação ter sido capturada, faz-se uma normalização com o índice de carga obtido, gerando um valor que traduza a carga relativa de cada máquina pertencente ao sistema.

Com esse valor normalizado, pode-se comparar as máquinas do sistema de igual para igual, sem a preocupação de estar sendo injusto com determinadas máquinas e alocando para essas cargas maiores do que elas possam suportar.

5.5.3 Comunicação

O algoritmo de balanceamento de cargas DPWP possui rotinas para o empacotamento e o envio de mensagens para o AMIGO. O algoritmo assume que qualquer tarefa pode enviar mensagens para ele através do AMIGO, que age como roteador de mensagens entre o ambiente de passagem de mensagem e o algoritmo. Não há limites para o número de mensagens trocadas entre a DPWP e o AMIGOD, ou entre duas políticas DPWP.

Toda a interface de comunicação implementada na DPWP segue a especificação adotada pelo AMIGOD, pois, como o escalonador DPWP foi construído sobre o AMIGOD, é de fundamental importância que o mesmo siga as regras pré-determinadas.

Na DPWP, enviar uma mensagem consiste em determinar um *buffer* que deve ser criado, onde as mensagens são depositadas temporariamente, empacotar a mensagem dentro do *buffer*, e finalmente, enviar a mensagem (o conteúdo do *buffer*). A mensagem deve ser recebida por uma função bloqueante pertencente ao AMIGOD, que deverá desempacotar essa mensagem e enviar para o destino correto, isto é, para o ambiente de passagem de mensagem local e/ou para outro AMIGOD em outra máquina.

O *send* usado no algoritmo DPWP é chamado de *send* bloqueante, pois retorna tão logo o *buffer* utilizado para enviar a mensagem esteja livre para ser novamente utilizado.

Sumarizando, a comunicação realizada pelo DPWP é baseada em TCP (*Transmission Control Protocol*), e assume que todas as máquinas pertencentes à máquina virtual sejam capazes de se comunicarem através desse mecanismo. A interface de comunicação usada durante o processo de troca de mensagens para escalonar um processo (ou tarefa) é a mesma usada pelo AMIGOD.

A DPWP se comunica diretamente com o AMIGOD, e este se torna responsável por entregar as mensagens destinadas para uma outra máquina (*sockets* UDP), ou para o ambiente de passagem de mensagem local (*sockets* TCP).

Caso uma política DPWP queira comunicar-se com uma “irmã” em outra máquina, a mesma deve enviar uma mensagem para o AMIGOD local que desempacota, envia para o

AMIGOD remoto, e esse por sua vez entrega a mensagem para a política destino. Apesar desse método causar uma quantidade extra de comunicação pela rede, optou-se por esta implementação pois o AMIGOD tem total controle das mensagens trocadas, o que não ocorreria caso fosse criado uma conexão TCP entre todas as políticas DPWP ativas no sistema. A Figura 5.4 mostra um exemplo dessa troca de mensagem.

Figura 5.4 - Troca de mensagem entre Políticas DPWP

5.6 Detalhes de Implementação

O código do algoritmo DPWP foi implementado de tal forma a fornecer todos os recursos necessários para que quaisquer aplicações concorrentes (e em especial as aplicações *CPU-bound*) possam ser escalonadas de maneira simples e direta sobre uma rede de estações de trabalho (Apêndice C). Como já exposto, a comunicação é toda baseada no AMIGOD.

5.6.1 Iniciação da DPWP

O escalonador DPWP é iniciado sempre através do AMIGOD, isto é, logo após o AMIGOD ser iniciado pelo ambiente de passagem de mensagem através da função *StartAMIGODD()*, ele lê o arquivo de configuração contendo todas as políticas que devem ser iniciadas e dentre essas está a DPWP (Figura 5.5 passo 1).

Figura 5.5 - Passos a serem realizados pela DPWP

O AMIGOD recupera do arquivo *policy.cfg* todas as políticas a serem criadas através da função *GetActivePolicy()* e as inicia através da função *StartPolicy()*. Dessa forma, o escalonador DPWP é iniciado e recebe em sua iniciação um parâmetro indicando o *port* TCP criado pelo AMIGOD para comunicar-se com as políticas criadas.

Em seguida, o DPWP cria um *socket* de comunicação TCP para conectar com o AMIGOD local, e cria a estrutura *dpwp_table* que é responsável em armazenar as informações sobre todas as máquinas pertencentes ao sistema.

Depois de criar a *dpwp_table*, o escalonador recupera do arquivo de configuração *benchmark.cfg* (mantido pelo AMIGO) a potência computacional relativa às demais máquinas e a armazena em uma variável global que sempre será usada quando a carga local for calculada. Em seguida, recupera a carga local através da função *getindexload()*, e insere todas as informações pertinentes ao *host* local na tabela chamando a função *updatehost()*. Dessa maneira, sempre o primeiro elemento a ser inserido na tabela é a máquina local (Figura 5.5 passo 2).

Após criar a estrutura de dados e se configurar, o escalonador DPWP conecta com o AMIGOD através da função *connect()* do UNIX (Stevens, 1991). Nesse momento o escalonador não tem nenhum conhecimento quanto à existência ou não de outra máquina no sistema, ele sabe apenas que a sua máquina está no ar. Por essa razão, faz-se necessário que o escalonador peça do AMIGOD a sua tabela de roteamento (*amigod_table*), atividade feita por intermédio da função *Get_Table()* (passo 3).

Essa função tem a finalidade de mandar uma mensagem para o AMIGOD solicitando do mesmo toda a sua tabela de roteamento, pois nesse momento o AMIGOD já se configurou e possui o endereço IP de todos os *hosts* que estão inseridos na máquina virtual. Essa mensagem é nula e tem em seu cabeçalho o código da mensagem AP_GETTABLE.

Depois de enviar a solicitação o escalonador DPWP chama a função *WorkForever()*, a qual é responsável por receber e tratar todas as mensagens enviadas pelo AMIGOD. Após o AMIGOD receber a mensagem com o código AP-GETTABLE, o mesmo chama a função *GetTable()* que enviará para a DPWP uma mensagem com o código AP_PUTTABLE contendo todos os *hosts* inseridos em sua tabela, com exceção do *host* local. Dessa forma, caso não haja

nenhum *host* inserido na máquina virtual essa mensagem será enviada para a DPWP com o conteúdo nulo.

A DPWP recebe essa mensagem através da função *WorkForever()* que chama a função *get_message()*, e essa por sua vez desempacotará a mensagem e ativará a função *Put_Table()*. Essa função sendo ativada, desempacota a mensagem recebida e retira todos os *hosts* recebidos na mensagem. As cargas das máquinas desses *hosts* serão inseridas na tabela local com a carga “-1” através da função *updatehost()* (passo 4).

Entretanto, caso essa mensagem seja nula, implica que não há nenhum novo *host* a ser inserido. Após inserir os novos *hosts* na tabela, o algoritmo DPWP chama a função *getloadremote()* que atualizará a tabela local com a carga remota de todas as máquinas no sistema. A Figura 5.6 espelha a configuração do sistema nesse momento.

Figura 5.6 - Configuração do Sistema

5.6.2 Manipulação da Tabela de Cargas

Como dito na seção anterior, a DPWP após ter sido iniciado cria uma tabela que será capaz de armazenar todos os endereços IP e cargas correspondentes a cada máquina (veja Figura 5.7). As funções responsáveis por manipular essa tabela são: *count_host()*, *freehost()*, *updatehost()*, *inserthost()*, *deletehos()*, *getdpwp_tale()*, *gethost()*, *searchhost()*.

A função *count_host()* é responsável por fornecer o número de elementos inseridos na tabela *dpwp_table* com exceção do *host* local. A próxima função, *freehost()*, trabalha em conjunto com *deletehost()*, isto é, sempre que houver necessidade de retirar um elemento da lista, é importante que o espaço ocupado por este *host* seja liberado, para isso é chamada a função *freehost()*.

A função *updatehost()* é usada para inserir e/ou atualizar elementos na tabela, isto é, quando essa função é chamada a mesma verifica através da função *searchhost()* se o elemento recebido como parâmetro já existe na tabela. Caso verdadeiro esse elemento terá a sua carga atualizada e, possivelmente, a sua posição na tabela também é atualizada.

Figura 5.7 - Tabela de Cargas da DPWP

Portanto, caso o *host* ainda não esteja na tabela, este será inserido na mesma através da função *inserthost()*. Assim como a *deletehost()* precisa liberar espaço, a *inserthost()* precisa alocar espaço para guardar o novo elemento, através da função *gethost()*. Contudo, a função

getdpwp_table() será chamada sempre que houver necessidade de recuperar todos os endereços IP contidos na tabela local, com exceção, é claro, do IP local.

A tabela *dpwp_table* será atualizada em duas condições: primeiramente quando o escalonador está se configurando, ou seja, logo após receber do AMIGOD a sua tabela de roteamento; a segunda situação é quando houver uma mudança de estado na máquina, pois essa deve informar aos demais *hosts* a sua nova carga.

No primeiro caso, o escalonador chama a função *getloadremote()*. Essa função manda uma mensagem com o código *AP_GETLOADREMOTE* para o AMIGOD local pedindo que o mesmo recupere as cargas de todos os *hosts* contidos na mensagem. Essa mensagem contém ainda a carga local que será informada aos AMIGODs remotos (passo 1 da Figura 5.8). O AMIGOD local, após receber essa mensagem, chama a função *PolGetLoadRemote()*, a qual enviará para cada *host* contido na mensagem *AP_GETLOADREMOTE* uma outra mensagem com o código *AA_GETLOAD* (passo 2).

O AMIGOD remoto, após receber essa mensagem, chama a função *AMIGODGetLoadLocal()* que repassará a mensagem com o código *AP_GETLOADLOCAL* para o escalonador DPWP local, pedindo sua carga (passo 3). O DPWP local, por sua vez, recebe essa mensagem através da função *getloadlocal()*, a qual é responsável por desempacotar a carga remota e atualizar a sua tabela; em seguida deve empacotar a sua carga atual e enviá-la ao *host* que solicitou por meio da mensagem com o código *AP_PUTLOADLOCAL* (passo 4).

Figura 5.8 - Passos Realizados na Atualização da Tabela de Cargas

O AMIGOD local recebendo essa mensagem sabe que deve repassá-la ao AMIGOD original e o faz enviando a mensagem `AA_PUTLOAD`, através da função `PolPutLoadLocal( )` (passo 5). O AMIGOD remoto, ao receber essa mensagem, repassa-a para a política local com o código `AP_PUTLOADREMOTE`, por meio da função `AMIGODPutLoadRemote( )` (passo 6).

Para finalizar essa atualização, a política original que solicitou a carga da máquina remota recebe a mensagem `AP_PUTLOADREMOTE` através da função `putloadremote( )`, retira então o valor da carga remota e atualiza em sua tabela. A Figura 5.8 apresenta todos os passos aqui abordados.

A outra atualização na tabela `dpwp_table` ocorre sempre que o estado da carga da máquina local muda, ou seja, passa de um estado ocioso para carregado ou vice-versa, pois nesse caso há necessidade de que as cargas das demais máquinas sejam atualizadas (política de informação atualizada por mudança de estado).

Essa atualização é ativada pelo escalonador local, logo após ter notado a sua alteração de estado. Nesse momento a DPWP chama a função `putloadlocal( )`, essa por sua vez empacota a sua carga local e a envia ao AMIGOD para que esse possa repassá-la a todos os outros *hosts*, após a função `putloadlocal( )` ser chamada; os próximos passos são os mesmos a partir do (passo 4) apresentado anteriormente.

5.6.3 Iniciação do Escalonamento

Após ter sido apresentado como o escalonador DPWP inicia e como a tabela de cargas é mantida sempre atualizada no sistema, será discutido como é feito o escalonamento em si.

O escalonamento inicia sempre após o ambiente de passagem de mensagem mandar uma mensagem com o código `AM_SCHED` para o AMIGOD. No caso do PVM, isso é feito através da função `GetHostsFromAMIGOD( )` (Figura 5.9 passo 1). O AMIGOD recebe essa mensagem através da função `RequestReceivers( )`, que tem a finalidade de desempacotar a mensagem `AM_SCHED` e enviar uma nova mensagem para o escalonador DPWP com o código

AP_SCHED. Essa mensagem deve conter o nome do arquivo executável, o número de instâncias a serem criadas e o tipo de arquitetura.

O DPWP recebe essa mensagem através da função *get_receivers()*. Essa função, após ser ativada, desempacota a mensagem recebida e retira da mesma o arquivo executável, o número de tarefas que deverão ser criadas e a arquitetura. Em seguida, inicia na tabela de cargas a busca pelas melhores máquinas na rede. A idéia aqui é distribuir a carga da melhor maneira possível a fim de que se obtenha um equilíbrio no sistema.

Baseando-se nesse princípio, o escalonador DPWP verifica o número de máquinas ociosas no sistema. Caso esse número não seja suficiente para contemplar todas as tarefas, continua analisando a tabela, agora buscando máquinas com a carga moderada (nem ociosa e nem carregada), caso ainda não seja suficiente volta considerando as máquinas ociosas e fica realizando *round-robin* dentre as máquinas do sistema que estão no estado de ociosa e/ou moderada até que a quantidade de tarefas seja alcançada.

Figura 5.9- Passos Realizados no Escalonamento.

As máquinas no estado de sobrecarregadas não são selecionadas por esse algoritmo. Entretanto, caso ocorra uma situação de saturação do sistema e todas as máquinas se encontrem no estado sobrecarregado, nesse caso em particular as tarefas serão distribuídas dentre essas máquinas, pois estas são as únicas disponíveis.

Após terem sido selecionadas as n melhores máquinas no sistema, a função *get_receivers()* empacota essas máquinas através da função *send_receivers()*, que após empacotar todos os *hosts* os envia para o AMIGOD através da mensagem com o código AP_SCHEDACK (passo 4).

O AMIGOD, após receber essa mensagem, chama a função *ReplyReceivers()* que tem a finalidade de repassar as máquinas receptoras recebidas do DPWP ao ambiente de passagem de mensagem através da mensagem AM_SCHEDACK (passo 5).

O ambiente de passagem de mensagem, após receber essa relação de máquinas, realiza a distribuição das tarefas propriamente dita. A Figura 5.9 apresenta a seqüência na qual esses passos ocorrem.

5.6.4 Recebimento das Mensagens

No escalonador DPWP todas as mensagens enviadas pelo AMIGOD são recebidas através da função *WorkForever()*, que é responsável por aguardar mensagens, utilizando a função *select()* do UNIX, a qual determina quais mensagens estão prontas para serem recebidas (Stevens, 1991; Comer, 1995).

Após verificar que existe uma mensagem pronta para ser recebida, essa função chama a *recv()*, a qual receberá o cabeçalho da mensagem enviada, desempacotará o tamanho da mensagem sem o cabeçalho e chamará novamente a função *recv()*, agora para receber o restante (o corpo) da mensagem. Fez-se necessário receber a mensagem através de dois *receivers*, porque em se tratando de conexão TCP, quando o AMIGOD precisa mandar mais de uma mensagem para o DPWP em um espaço de tempo muito curto, o mesmo recebe a nova mensagem e a coloca no final da mensagem anterior, com isso, enquanto o *buffer* de recebimento tiver espaço, as mensagens enviadas são todas enfileiradas, perdendo a informação original.

Resolvido esse problema e tendo a mensagem sido recebida corretamente, é ativada a função *get_message()*, a qual recebe a mensagem inteira, desempacota o seu cabeçalho, recupera o seu código e chama a função responsável por tratar cada mensagem recebida.

5.6.5 Atualização Periódica da Tabela Local

A seção 5.5.2 tratou dos detalhes de implementação pertinentes à manipulação da tabela de cargas, mas nada foi falado quanto à atualização periódica que ocorre na tabela de carga local a cada DPWP. Para que a informação manipulada na tabela de cargas não seja velha e/ou imprecisa, a tabela local é atualizada periodicamente através de um *timeout* implementado pela própria função *select()*, isto é, além do recebimento de mensagens a função *select()* é usada para ativar o processo de atualização da tabela.

Esse processo, quando ativado, chama primeiramente a função *getindexload()*, a qual irá recuperar instantaneamente o índice de carga adotado na DPWP (nesse caso, a fila de processos prontos da CPU), em seguida é ativada a função *get_averageload()*, responsável por calcular a média da carga nos últimos 20 (vinte) segundos. Esse período foi escolhido a partir dos trabalhos apresentados por Kunz (Kunz, 1991) e por Ferrari & Zhou (Ferrari & Zhou, 1987) e adaptados à realidade da plataforma usada no desenvolvimento deste trabalho.

Como já explicado anteriormente, trabalhar com a média da carga ao invés de valores instantâneos, traz todas as vantagens já expostas. Dessa maneira, após ter-se obtido a média, a mesma é repassada para a função *update_table()*, que é responsável em atualizar a tabela local e chamar a função *get_status()*, a qual analisará se houve mudança de estado ou não. Caso positivo é chamada a função *putloadlocal()*, que se responsabiliza em avisar os demais *hosts* como já exposto na seção 5.5.2.

5.7 Considerações Finais

A DPWP apresentada neste capítulo é um novo algoritmo de escalonamento de processos desenvolvido para o AMIGO, o qual constitui uma nova ferramenta de suporte ao escalonamento, que oferece para os usuários um completo, simples e eficiente ambiente.

A DPWP foi projetada para ser usada em um ambiente distribuído paralelo, sobre sistemas baseado em redes de estações de trabalho, e sua principal característica é propiciar um bom desempenho nesse sistema.

Além disso, o algoritmo DPWP foi implementado de tal forma que o mesmo não exija nenhum esforço para mudar de uma plataforma para outra, podendo ser utilizado tanto sobre o PVM (Geist et. al., 1994) quanto sobre o MPI (Dantas, 1996), pois o algoritmo DPWP tem uma interface bem definida com o *daemon* do AMIGO e esse, por sua vez, com as plataformas de passagem de mensagem.

De modo geral, a DPWP demonstra sua capacidade de tratar a heterogeneidade constante nas redes de estações de trabalho para permitir que as tarefas criadas pela aplicação paralela possam ser iniciadas sobre *hosts* de diferentes arquiteturas.

A eficiência e eficácia do algoritmo DPWP é demonstrada através da forma pela qual o mesmo normaliza o índice de carga nas diferentes máquinas pertencentes ao sistema. Essa eficiência pode ser constatada no próximo capítulo através dos resultados obtidos com a execução da DPWP e do AMIGO sobre uma rede de estações de trabalho. Essa rede se caracteriza por apresentar um alto grau de heterogeneidade em termos de potência computacional das máquinas.

Avaliação de Desempenho da DPWP

Este capítulo descreve os resultados obtidos no ambiente distribuído através da implementação do algoritmo de escalonamento DPWP implementado para o AMIGO e tendo o PVM como a plataforma de passagem de mensagem. A plataforma computacional utilizada é apresentada e em seguida são levantados os resultados obtidos nos testes.

6.1 Considerações Iniciais

O escalonamento de processos é uma realidade que tem conquistado cada vez mais o seu espaço nas pesquisas desenvolvidas na área de sistemas distribuídos e computação paralela (Feitelson et. al., 1997).

As aplicações paralelas desenvolvidas com o escalonamento tendem a apresentar uma otimização no desempenho em relação às aplicações que não implementam nenhuma estratégia de escalonamento.

Por essa razão, é importante mostrar o desempenho alcançado no PVM modificado para realizar balanceamento de cargas, com relação ao PVM original com a sua técnica padrão para distribuir suas tarefas entre as máquinas pertencentes ao sistema.

Para essa avaliação de desempenho do PVM com balanceamento foram utilizadas algumas aplicações paralelas desenvolvidas em um trabalho de mestrado no grupo de Sistemas Distribuídos e Programação Concorrente do ICMC (Cortés, 1999).

6.2 Plataforma Utilizada

As máquinas componentes do sistema distribuído utilizado são 7 (sete) computadores do tipo PC, que constituem uma máquina do tipo MIMD com memória distribuída. Todas as máquinas estão localizadas no Laboratório de Sistemas Distribuídos e Programação Concorrente, no Instituto de Ciências Matemática e de Computação da USP e executam o sistema operacional LINUX (versão de *kernel* 2.0.30) (Welsh, 1995), que é um sistema operacional largamente utilizado na comunidade acadêmica e que, mais recentemente, tem sido apontada como alternativa para vários outros setores (veja a Figura 6.1).

Figura 6.1 - Ambiente do LASDPC usado nos testes

As máquinas estão conectadas através de uma rede *Ethernet* de 10 Mbits, utilizando o protocolo de comunicação TCP/IP. Além disso, esse sistema caracteriza-se por sua considerável heterogeneidade em relação à potência computacional das máquinas. A Tabela 6.1 mostra o nome e a configuração de cada máquina pertencente ao sistema.

Essa heterogeneidade teve suma importância nos testes da DPWP, pois, acima de tudo, serviu para validar a estratégia adotada pelo algoritmo para normalizar essa diferença, principalmente no que tange à potência computacional das máquinas do sistema. A Tabela 6.1 mostra o nome e a configuração particular de cada máquina do sistema.

Tabela 6.1 - Configuração das máquinas usadas no LASDPC

Nome	Configuração
Lasd02	486 DX2 66 MHz, 8 Mb, 512 de cache.
Lasd03	Pentium 200 MHz MMX, 32 MBytes, 512 de cache.
Lasd04	486 DX4 100 MHz, 12 Mb, 512 de cache.
Lasd06	Pentium 100 MHz, 32 MBytes, 512 de cache.
Lasd07	Pentium 200 MMx, 32 MBytes, 512 de cache.
Lasd08	Pentium 166 MHz, 32 Mbytes, 512 de cache.
Lasd09	486 DX2 66 MHz, 8 Mb, 512 de cache.

É importante ressaltar que os tempos foram coletados em horário de pouca utilização do laboratório (finais de semana, à noite e feriados) minimizando, portanto, a existência de processos que pudessem sobrecarregar outros recursos do sistema (como por exemplo, memória e dispositivos de Entrada/Saída) e, dessa forma, implicar em resultados não reais para esta pesquisa. Entretanto, mesmo reduzindo o número de possíveis usuários devido ao horário, não se pode garantir que, depois de iniciada a execução de uma aplicação, outros usuários não tenham acesso remoto às máquinas utilizadas para formar a máquina virtual.

Todavia, a existência de outros usuários não implica em uma diferença considerável nos resultados finais apresentados a seguir, pois, dada a característica da DPWP, a qual atualiza a sua tabela de carga de tempos em tempos, caso haja uma sobrecarga da máquina causada por usuários remotos isso é diretamente espelhado pelo índice de carga adotado e, assim, o estado da máquina será divulgado através da política de informação.

Optou-se em realizar os testes em um horário não convencional devido à facilidade de se controlar a carga do sistema e, assim sendo, o sistema ora era carregado e ora era descarregado. Isso é importante para que a operação da DPWP possa ser avaliada em qualquer estados do sistema.

Neste trabalho a expressão PVM balanceado implica no PVM que teve seu código alterado para que o mesmo pudesse interagir com o AMIGO, e o AMIGO interagir com a DPWP.

6.3 Aplicação Paralela Utilizada

As aplicações usadas para testar o algoritmo de escalonamento DPWP, assim como o AMIGO, foram todas voltadas para algoritmos numéricos⁷, por necessitarem de grande capacidade de processamento e, dessa forma, preencherem os objetivos das aplicações paralelas desejáveis para participarem do escalonamento (Apêndice D).

A primeira aplicação paralela escolhida para avaliar o desempenho da DPWP é o Método dos Trapézios Compostos, o qual soluciona uma integral definida pela aproximação de diversos trapézios. Esse algoritmo é considerado pouco preciso quando são utilizadas poucas divisões; por esse motivo, as integrais foram divididas em um número grande e crescente de trapézios.

A paralelização foi realizada através de uma propriedade das integrais, onde, dada uma função qualquer $f(x)$ que pode ser integrada nos pontos de a até b , ela pode ser dividida na soma de duas outras integrais.

Dessa forma, essa integral pode ser dividida no somatório de n outras integrais definidas, onde cada uma delas pode dar origem a uma nova tarefa e, conseqüentemente, ser calculada em processadores diferentes.

Como exposto, esse método é um exemplo típico de aplicação caracterizada como *CPU-bound*, pois utiliza muito processamento e pouquíssima comunicação. A comunicação se faz necessária apenas 2 (duas) vezes, primeiramente quando o mestre manda os dados para os escravos e posteriormente quando os escravos retornam os dados para os mestre.

⁷ Algoritmos numéricos são algoritmos que procuram encontrar soluções aproximadas de problemas através da representação do mesmo por um modelo matemático (Cortés, 99).

A segunda aplicação escolhida foi a de multiplicação paralela de matrizes, por apresentar uma grande quantidade de operações em ponto flutuante, exigida para a solução utilizando o algoritmo tradicional de complexidade n^3 .

A paralelização dessa aplicação inicia com a geração de duas matrizes quadradas A e B. O processo mestre gera a matriz A e, ao mesmo tempo, os processos escravos geram a mesma matriz B. Após terminarem o processo, o mestre envia a sua matriz gerada A para os processos escravos e um dos processos escravos envia a matriz B para o mestre.

Com esses dados, o mestre envia aos escravos a posição da linha onde cada processo iniciará a sua parte da multiplicação. Todos os processadores executam a multiplicação incluindo o mestre. Para finalizar, os escravos mandam os respectivos pedaços calculados para o mestre.

Em suma, essa aplicação também consome bastante processamento, entretanto, há um número bem maior de comunicação se comparado com a primeira aplicação. Isso propicia uma avaliação do comportamento do algoritmo DPWP em aplicações que não são puramente *CPU-bound*.

6.4 Análise dos Resultados Obtidos

Essa seção mostra todos os resultados obtidos nos testes realizados para validar o AMIGO e o algoritmo DPWP. Cada tabela agrupa o tempo necessário para a execução da aplicação testada, assim como as plataformas de passagem de mensagem usadas, isto é, indica se a aplicação foi executada usando o PVM sem a DPWP ou o PVM com a DPWP.

O segundo (seg.) é a unidade de tempo usada para representar a média de execução da aplicação sobre multiplicação de matrizes (da Tabela 6.2 até a Tabela 6.5). Em seguida são abordados os resultados equivalentes à execução da aplicação relacionada ao método do trapézio, nesse caso a unidade de tempo adotada é o minuto (min.) visto o elevado tempo dado em segundos.

6.4.1 Execução da Multiplicação de Matrizes

A Tabela 6.2 e o gráfico 6.1 mostram os resultados obtidos na execução da multiplicação de matrizes com o PVM sem balanceamento. Nesse caso, é claro a influência da carga em um sistema distribuído, pois quando o sistema apresenta na média uma carga ociosa (ou seja, a média da carga está abaixo do limite inferior) e/ou moderada (a média está acima do limite inferior e abaixo do limite superior) o tempo de execução da aplicação foi levemente menor se comparado aos resultados obtidos a partir de um sistema carregado.

Tabela 6.2 - Resultados com o PVM sem balanceamento (multiplicação de Matrizes)

Tamanho da Matriz	50	100	150	200	250	300
PVM Normal Carregado	0,98	3,43	7,82	17,27	25,46	48,76
PVM Normal Ocioso	0,58	1,82	4,12	7,16	10,55	20,91

Gráfico 6.1 - PVM sem balanceamento (multiplicação de Matrizes)

Por outro lado, a Tabela 6.3 e o Gráfico 6.2 mostram os resultados alcançados na execução da multiplicação de matrizes usando o PVM balanceado tanto em um sistema intitulado como sobrecarregado (a média da carga está acima do limite superior) quanto no sistema ocioso (a média da carga está abaixo do limite inferior). Percebe-se que apesar da carga também influenciar no tempo de execução da aplicação rodando com o balanceamento empregado, há um ganho da aplicação executada no PVM balanceado sobre a execução no PVM sem considerar o estado do sistema.

Tabela 6.3 - Resultado com o PVM balanceado (multiplicação de matrizes)

Tamanho da Matriz	50	100	150	200	250	300
PVM Balanceado Carregado	0,72	3,12	6,91	16,48	26,64	40,19
PVM Balanceado Ocioso	0,37	1,44	3,29	6,26	9,41	14,38

Gráfico 6.2 - PVM balanceado (multiplicação de matrizes)

A Tabela 6.4, assim como o Gráfico 6.3, mostram o quão importante é a aplicação de balanceamento de cargas sobre plataformas computacionais distribuídas que apresentam uma variação considerável em suas cargas. O ganho no desempenho da aplicação executada com o PVM balanceado foi em média de 12,24% sobre a execução usando o PVM normal. Essa diferença não foi mais significativa dada a natureza da aplicação aqui apresentada.

Nesse caso, o retrato de um sistema carregado não é um bom diferencial para a aplicação, pois a mesma, apesar de usar bastante processamento, consome também considerável uso de dispositivos de Entrada/Saída para a comunicação mestre/escravo.

Tabela 6.4 - Comparação entre os PVMs em um sist. carregado (multi. de matrizes)

Tamanho da Matriz	50	100	150	200	250	300
PVM Normal Carregado	0,98	3,43	7,82	17,27	25,46	48,76
PVM Balanceado Carregado	0,72	3,12	6,91	16,48	26,64	40,19

Gráfico 6.3 – Um sistema carregado - multiplicação de matrizes

Para finalizar, a Tabela 6.5 e o Gráfico 6.4 apresentam a comparação entre o PVM balanceado e o PVM normal em um sistema considerado ocioso. Assim como no caso do sistema carregado (apresentado anteriormente) o PVM balanceado teve um desempenho melhor que o PVM normal. Nesse caso, o ganho foi em média 24,19%. Isso é explicado como antes, ou seja, o índice de carga adotado para espelhar a carga do sistema não tem grande expressividade diante do algoritmo de multiplicação de matrizes.

Isso mostra que o algoritmo de escalonamento deve ser voltado para aplicações particulares e que o índice de carga adotado deve ser voltado para os interesses da aplicação. No caso da multiplicação de matrizes é interessante adotar um índice de carga que seja representante tanto da utilização de CPU quanto de dispositivos de Entrada/Saída.

Tabela 6.5 – Comparação entre os PVMs em um sistema ocioso (multi. de matrizes)

Tamanho da Matriz	50	100	150	200	250	300
PVM Normal Ocioso	0,58	1,82	4,12	7,16	10,55	20,91
PVM Balanceado Ocioso	0,37	1,44	3,29	6,26	9,41	14,38

Gráfico 6.4 - um sistema ocioso - multiplicação de matrizes

6.4.2 Execução do Método do Trapézio

A Tabela 6.6 e o Gráfico 6.5 mostram a execução dessa aplicação sobre o PVM sem balanceamento. Nesse caso há uma grande diferença no tempo de execução da aplicação, pois o PVM normal não usa nenhum tipo de seleção para escolher as máquinas que receberam as novas tarefas, com isso, quando o sistema espelha uma sobrecarga implica que na média as máquinas estão apresentando um desempenho não aceitável. Dessa maneira, caso o PVM inicie alguma tarefa (carga) para uma dessas máquinas, mesmo que o mestre seja colocado em uma ótima máquina (máquina ociosa) esse tem que esperar pelo término do filho, que está executando em uma máquina carregada.

Tabela 6.6 - Resultados com o PVM sem balanceamento (método do trapézio)

Número de Integrais (*1000)	300	600	1200	2400	4800
PVM Normal Carregado	91,32	94,87	95,76	100,01	102,79
PVM Normal Ocioso	29,23	32,05	37,46	42,81	<u>56,44</u>

Gráfico 6.5 - PVM sem balanceamento (método do trapézio)

Em seguida são apresentados a Tabela 6.7 e o Gráfico 6.6, os quais mostram o desempenho do algoritmo do trapézio quando executado pelo PVM que implementa o balanceamento de carga. Nota-se que a diferença de desempenho entre o sistema carregado e o sistema ocioso é ligeiramente grande, na média 100%, mas ainda assim esse desempenho é muito melhor do que o desempenho fornecido na execução do PVM sem balanceamento.

Tabela 6.7 - Resultado com o PVM balanceado (método do trapézio)

Número de Integrais (*1000)	300	600	1200	2400	4800
PVM Balanceado Carregado	9,36	9,38	9,40	9,44	9,48
PVM Balanceado Ocioso	3,28	4,13	4,31	4,55	4,83

Gráfico 6.6 - PVM balanceado (método do trapézio)

A Tabela 6.8 e o Gráfico 6.7 mostram a grande importância de se implementar um balanceamento de carga com uma estratégia bem definida e específica. É claro o excelente ganho de desempenho da aplicação, quando essa é escalonada pela DPWP. Isso explica-se pelo fato de se estar diante de uma aplicação caracterizada como *CPU-bound*, usando um escalonador totalmente voltado para essa realidade. A Tabela 6.4 mostra a diferença de desempenho entre os dois PVMs (em média 90,27%). Quando executando a multiplicação de matrizes. Entretanto, o ganho naquele caso não foi tão significativo quanto aqui, dada a diferença na natureza de cada aplicação.

Tabela 6.8 - Comparação entre os PVMs em um sistema carregado (mét. do trapézio)

Número de Integrais (*1000)	300	600	1200	2400	4800
PVM Normal Carregado	91,32	94,87	95,76	100,01	102,79
PVM Balanceado Carregado	9,36	9,38	9,40	9,44	9,48

Gráfico 6.7 - Um sistema carregado – método do trapézio

A seguir, a Tabela 6.9 e o Gráfico 6.8 apresentam os tempos de execução obtidos pelo método do trapézio tanto em uma plataforma balanceada quanto em uma plataforma sem balanceamento, ambas estando no estado ocioso. O ganho de desempenho (em média 89,03%) também ocorre nesse caso de uma maneira gratificante, pois, mesmo que o sistema represente na média uma situação de ociosidade, a implantação de técnicas que sejam capazes de escolher as melhores máquinas disponíveis para a execução da aplicação, diminui-se consideravelmente o risco de se ter alguma máquina que não esteja tão ociosa assim, e dessa forma possa vim causar um retardo no desempenho final do sistema, ocasionado por uma máquina apenas.

Tabela 6.9 - Comparação entre os PVMs em um sistema ocioso (método do trapézio)

Número de Integrais (*1000)	300	600	1200	2400	4800
PVM Normal Ocioso	29,23	32,05	37,46	42,81	56,44
PVM Balanceado Ocioso	3,28	4,13	4,31	4,55	4,83

Gráfico 6.8 - Um sistema ocioso – método do trapézio

Finalizando, a tabela 6.10 e o Gráfico 6.9 mostram o quão importante é o tratamento da heterogeneidade em um sistema distribuído. O desempenho alcançado com o algoritmo de balanceamento sem contemplar a heterogeneidade é, sem dúvida, maior do que o alcançado com o algoritmo sem escalonamento (na média o aumento foi de 82,25%). Entretanto, quando comparado ao algoritmo que contempla a heterogeneidade esse desempenho é maior (cerca de 89,03%). Isso explica-se facilmente pelo fato de se terem realizados os testes em um ambiente totalmente heterogêneo e, conseqüentemente, as máquinas que têm grande capacidade de processamento são diretamente comparadas às máquinas com baixa capacidade. Essa situação implica em um falso balanceamento de carga, visto que as máquinas com baixa capacidade de processamento ficaram carregadas muito mais rapidamente que as demais.

Tabela 6.10 – Comparação entre os PVMs e a heterogeneidade

Número de Integrais (*1000)	300	600	1200	2400	4800
PVM Normal	29,23	32,05	37,46	42,81	56,44
PVM Balanceado sem heterog.	3,30	6,21	8,03	8,75	9,12
PVM Balanceado com heterog.	3,28	4,13	4,31	4,55	4,83

Gráfico 6.9 – Comparação entre os PVM's com e sem heterogeneidade

6.5 Considerações Finais

Neste capítulo foram apresentados os resultados obtidos com a execução de 2 (dois) algoritmos numéricos (multiplicação de matrizes e método do trapézio) sobre o PVM sem balanceamento de carga (PVM normal) e o PVM com o código alterado (PVM balanceado).

Os resultados aqui apresentados são de suma importância, pois mostram o ganho de desempenho de uma aplicação paralela que foi escalonada através de algoritmos de balanceamento, sobre a aplicação que não teve nenhum tipo de estudo ou análise para escalonar suas tarefas entre os processadores da máquina virtual.

Além disso, ratifica a importância de se contemplar a heterogeneidade existente em um sistema distribuído. Todavia, espelha a necessidade de se tratar a heterogeneidade de acordo com a característica da aplicação paralela que se deseja distribuir.

No caso deste trabalho, contempla-se a diferença de potência computacional, pois os testes realizados foram todos voltados para aplicações paralelas que consomem bastante processamento, mesmo que às vezes não só processamento, mas comunicação também.

Os testes realizados servem para validar o algoritmo de balanceamento de cargas sugerido neste trabalho (a DPWP) bem como a proposta apresentada pelo trabalho de doutorado que propõe o Ambiente de Escalonamento Flexível e Dinâmico – AMIGO (Souza et. al., 1999a).

O próximo capítulo conclui esta dissertação, discutindo as dificuldades encontradas na implementação dos módulos e listando sugestões para que seja dada continuidade a este trabalho, que é pioneiro no que diz respeito à utilização do AMIGO como um todo.

Conclusões

Este capítulo apresenta uma discussão final sobre o trabalho desenvolvido, abordando uma análise da revisão bibliográfica. As contribuições relevantes e as dificuldades encontradas são apresentadas em seguida. Posteriormente, sugestões para trabalhos futuros, que servirão como diretrizes para a continuidade do trabalho aqui iniciado, são também propostas.

7.1 Considerações Iniciais

Neste trabalho, apresentou-se a DPWP, um algoritmo de escalonamento de processos que objetiva alcançar o maior equilíbrio possível das cargas distribuídas sobre as máquinas pertencentes ao ambiente paralelo virtual.

No projeto da DPWP houve necessidade de tomar-se duas decisões relevantes na implementação da mesma. A primeira concerne ao fato de se implementar uma política de escalonamento dinâmica, e a segunda está relacionada à não preempção nesse algoritmo.

Além de apresentar todos os detalhes de implementação da DPWP, este trabalho discute também o AMIGO, o qual é uma ferramenta de software cujo objetivo é agrupar de modo fácil, coerente e organizado as melhores políticas de escalonamento.

Por último, a avaliação de desempenho da DPWP executando através do AMIGO e do ambiente de passagem de mensagem PVM, foi apresentada e discutida, permitindo que os tempos de execução sejam comparados e principalmente permitindo que algumas questões interessantes possam ser levantadas.

7.2 Análise da Revisão Bibliográfica

Sistemas Distribuídos e Paralelos

Os tópicos mais relevantes e relacionados tanto aos sistemas distribuídos, quanto à programação paralela foram discutidos. Em termos de sistemas distribuídos, muita pesquisa tem sido desenvolvida com o intuito de aumentar as velocidades das redes e a transparência para o usuário, fazendo com que o usuário final não necessite ter nenhum conhecimento sobre o sistema que está usando.

Em termos de arquiteturas paralelas, vêm sendo muito utilizados os sistemas distribuídos simulando uma arquitetura paralela virtual. Seu custo relativamente baixo transformou a computação paralela extremamente atrativa e cada vez mais popular, sendo usada em diversos setores.

Em relação à parte de software dessas arquiteturas paralelas, tem havido uma tendência mais forte à utilização de programas MPMD e SPMD, apoiados por plataformas paralelas de passagem de mensagem, tais como PVM e MPI.

Escalonamento de Processos

Neste tópico foi apresentado o escalonamento de processos como uma estratégia para se obter melhor desempenho de aplicações paralelas. Tem havido uma união entre a computação paralela e os sistemas distribuídos. Entretanto, quando se unem as duas plataformas, unem-se também seus problemas, fazendo com que o escalonamento de processos para aplicações paralelas se preocupe com questões antes inexistentes como a heterogeneidade, multiusuários, variações de carga de trabalho, entre outras.

Por essa razão, pesquisando-se na literatura disponível sobre escalonamento de processos é fácil encontrar artigos que tratam, de alguma maneira, dessa atividade. Entretanto, há uma falta de consenso entre os autores da área sobre uma nomenclatura única e simples, e isso é um ponto negativo na tentativa de unir a teoria que há com a prática que ainda deixa a desejar.

7.3 Discussão sobre os Resultados Obtidos

Os resultados obtidos no desenvolvimento deste trabalho demonstram, acima de tudo, a importância do escalonamento de processos em sistemas computacionais. Essa importância fica clara pelos resultados mostrados no capítulo 6.

Ao projetar um algoritmo de escalonamento, deve-se preocupar com os objetivos almejados, isto é, o projetista deve traçar metas e baseando-se nessas metas deve definir os pontos essenciais no projeto.

Como mostrado pelos resultados obtidos, uma boa política de escalonamento deve ser voltada para um certo grupo de aplicações, melhorando ao máximo o desempenho da mesma e não tentar alcançar um bom escalonamento para todas as classes de software. Cada classe de software tem particularidades que devem ser tratadas de maneira eficiente, e só assim o escalonamento “ideal” poderá ser obtido.

A DPWP mostrou bom resultado no desempenho da aplicação com as características que essa política se propõe a balancear. Por outro lado, os resultados apresentados pela aplicação (multiplicação de matrizes) que foge um pouco das características de uma aplicação *CPU-bound*, apesar de terem sido melhores do que os resultados obtidos com o PVM sem balanceamento, não foram tão bons quanto os resultados obtidos com o método do trapézio, que é uma aplicação puramente *CPU-bound*.

A heterogeneidade é um outro ponto que mostrou, através dos resultados, sua importância. A DPWP tratou essa questão através de um método simples, mas que mostrou ser eficiente, pois os resultados obtidos nos testes onde a diferença de potência computacional não foi contemplada mostram um resultado ineficaz quando relacionados aos resultados que contemplam essa diferença.

Concluindo, o escalonamento de processos é uma estratégia que deve ser considerada em qualquer situação. Porém, essa estratégia deve estar voltada aos interesses aos quais se propõe a aplicação escalonada, pois, caso contrário, esse escalonamento pode comprometer o desempenho global do sistema, sobrecarregando-o com atividades que em nada apoiam a aplicação como, por exemplo, uma troca excessiva de mensagens na busca do escalonamento adequado.

7.4 Contribuições deste Trabalho

Dentre as contribuições apresentadas por este trabalho destacam-se:

- ❖ Apontar dificuldades na literatura, após o levantamento bibliográfico realizado notou-se a falta de uma nomenclatura simples e concisa na área de escalonamento de processos;
- ❖ Propor uma nova política de escalonamento para balanceamento de cargas específica, concisa e eficiente;
- ❖ Fornecer a primeira política para o AMIGO contribuindo para a validação do mesmo;
- ❖ Implementar o AMIGOD e as alterações no PVMD, os quais poderão ser usados por outras políticas de escalonamento de processos que serão implementadas para o AMIGO;
- ❖ Demonstrar as diferenças entre classes de aplicações. Os testes demonstraram diferenças no desempenho de acordo com a natureza de cada aplicação paralela executada, pois cada escalonador de processos deve ser projetado para agir em classes de software específicas;
- ❖ Demonstrar a importância de se tratar a heterogeneidade existente no ambiente paralelo virtual;
- ❖ Ratificar a importância de se implementar o balanceamento de cargas em um sistema distribuído, onde a variação de cargas é uma característica constante no sistema dada a natureza multiusuária;
- ❖ A DPWP, por ser pioneira, serve como base para novas pesquisas sobre o balanceamento dinâmico de cargas em plataformas paralelas/distribuídas no grupo de Sistemas Distribuídos e Programação Concorrente.

7.5 Dificuldades Encontradas

Inicialmente, um dos maiores empecilhos para o desenvolvimento do projeto foi a falta de uma nomenclatura clara e concisa, que possibilite um melhor entendimento da área. Como consequência dessa falta de nomenclatura, houve dificuldades em relacionar-se os inúmeros trabalhos existentes nessa área de pesquisa e, dessa forma, encontrar os pontos positivos e negativos que deveriam ser tratados.

No desenvolvimento da camada inferior do AMIGO, encontrar falhas no código fonte foi uma dificuldade à parte, pois a falta de ferramentas para depuração causou dificuldades extras, as quais consumiram bastante tempo.

Ainda na implementação do código do AMIGO, houve alguns problemas encontrados na implementação das funções do UNIX. O primeiro problema ocorreu quando houve necessidade de implementar-se a função *WorkUntilDeath()* do AMIGOD, a qual deve ficar em *loop*, preparada para receber e enviar mensagens. As mensagens recebidas são primeiramente detectadas pela função *select()* do UNIX, em seguida é efetivamente recebida por um *recv()*. Nesse caso, quando várias mensagens são enviadas em um intervalo de tempo muito pequeno e, se o tamanho dessas mensagens não fosse do tamanho exato do *buffer* localizado na função *recv()*, as mensagens eram todas recebidas e “emendadas” umas nas outras até que o *buffer* completasse a sua execução.

O segundo problema está relacionado a um erro na prototipação da função *recvfrom()*, o qual indica em um de seus parâmetros o tamanho de uma determinada estrutura, e isso não é real, pois quando se preenche esse parâmetro dessa maneira, o mesmo funciona para alguns casos e para outros não. Há necessidade de preencher esse parâmetro com o tamanho de uma outra estrutura mais geral que a especificada no protótipo.

O outro problema ocorreu na implementação da função *gethostbyaddr()*, a qual funcionava perfeitamente bem quando chamada a primeira vez, entretanto quando chamada pela segunda vez em uma mesma função, essa função não apresenta um comportamento confiável.

7.6 Trabalhos Futuros

Para a continuidade deste projeto, são sugeridos alguns trabalhos nessa linha de pesquisa:

- ❖ Implementação de políticas de escalonamento que tratem a migração de processos e possibilitem, dessa forma, a correção de algum escalonamento realizado erroneamente;
- ❖ Desenvolvimento de outras políticas de escalonamento que não tenham o objetivo de balancear cargas;
- ❖ Implementação de políticas de balanceamento de cargas para outras classes de aplicação como, por exemplo, *I/O-bound*;
- ❖ Comparação dos resultados obtidos neste trabalho com os resultados obtidos por outras políticas, desenvolvidas para a mesma plataforma;
- ❖ Implementar no MPI a interface necessária para a execução da DPWP;
- ❖ Investigar os limites de cargas usados para delimitar um sistema como sobrecarregado, ocioso ou moderado;
- ❖ Analisar métodos mais eficazes para a obtenção de índices de cargas que possam aumentar o desempenho da aplicação paralela;
- ❖ Analisar o impacto da atualização do índice de carga no desempenho do escalonador.

7.7 Considerações Finais

Diante dos objetivos e motivações inicialmente propostos para este trabalho, verifica-se que a implementação da DPWP apresentou ótimos resultados, pois a mesma propiciou a implantação de escalonamento dinâmico em ambientes paralelos virtuais, o que permite que um número maior de possíveis usuários possa familiarizar-se com as técnicas de escalonamento de processos.

A DPWP também mostrou ser uma alternativa eficiente para balancear cargas, pois tem-se comportado de maneira estável em todos os testes realizados no Laboratório de Sistemas Distribuídos e Programação Concorrente (LASDPC) do ICMC.

De modo geral, o projeto como um todo teve muitas compensações, inclusive a primazia de ter sido a primeira política implementada para o AMIGO, assim como ter sido responsável pela implementação parcial do código do AMIGO.

Ao término deste projeto, fica a certeza da contribuição dada para a comunidade científica e da importância do mesmo para o grupo de Sistemas Distribuídos e Computação Concorrente, que agora detém, na prática, o conhecimento necessário para que este trabalho pudesse ser concluído de maneira eficiente e eficaz.

Referências Bibliográficas

- Alfaro et. al., 1997 ALFARO, F. J., Gallud, J. A., Sánchez, J. L., "A Function to Dynamic Workload Allocation in Distributed Application", Fourth European PVM-MPI Conference - Euro PVM/MPI'97, Lecture Notes in Computer Science, 1332, 1997, pp 219-225.
- Almasi & Gottlieb, 1994 ALMASI, G. S., Gottlieb A., "Highly Parallel Computing", 2^a ed., The Benjamin Cummings Publishing Company, Inc., 1994.
- Alves et. al., 1995 ALVES, A., Silva, L., Carreira, J., Silva, J. G., "WPVM: Parallel Computing for the People", Proceedings of High Performance Computing and Networking Conference (HPCN'95), in Springer Verlag Lecture Notes in Computer Science, Milan, Italy, pp. 582-587, 1995.
- Araújo et. al., 1999a ARAÚJO, A. P. F., Souza, P. S. L., Santana, M. J., Santana, R. H. C., "DPWP - A New Load Balancing Algorithm", Third World Multiconference on Systemics, Cybernetics and Informatics (SCT'99) and the Fifth International Conference on information Systems Analysis and Synthesis (ISAS'99), Orlando, USA, 1999 (no prelo).
- Araújo et. al., 1999b ARAÚJO, A. P. F., Souza, P. S. L., Santana, M. J., Santana, R. H. C., "A New Dynamical Scheduling Algorithm, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), 1999, Monte Carlo, Las Vegas, Nevada, USA, 1999 (no prelo).
- Beguelin et. al., 1994 BEGUELIN, A., Geist, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V., "PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing", The MIT Press, 1994.
- Calkin et. al., 1994 CALKIN, R., Hempel, R., Hoppe, H. C., Wypior, P., "Portable programming with the PARMACS message-passing library", Parallel Computing, v. 20, pp. 615-632, 1994.

- Casas et. al., 1994 CASAS, J., Konuru, R., Otto, S. W., Prouty, R., Walpole, J., "Adaptive Load Migration Systems for PVM", Supercomputing'94 Proceedings, pp. 390-399, 1994.
- Casas et. al., 1995 CASAS, J., Clark, D., Konuru, R., Otto, S., Prouty, R., Walpole, J., "MPVM: A Migration Transparent Version of PVM", Computer System, v. 8, pp. 171-216, 1995.
- Casavant & Kuhl, 1988 CASAVANT, T. L., Kuhl, J. G., "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems", IEEE Transactions on Software Engineering, pp. 141-154, Fevereiro de 1988.
- Chow & Abraham, 1995 CHOW, T. C.K., Abraham, J. A., "Load Balancing in a Network Flow Optimization Code", Springer Lecture Notes in Computer Science, 1041, 1995, pp. 214-222.
- Clark et. al., 1995 CLARK, D. L., Casas, J., Otto, S. W., Prouty, R. M., Walpole, J., "Scheduling of Parallel Jobs on Dynamic, Heterogeneous Networks", <http://www.cse.ogi.edu/DISC/projects/cpe/>
- Comer, 1995 COMER, D., "Internetworking with TCP/IP – Principles, Protocols and Architecture", Vol. 1, Third Edition, Prentice Hall, 1995.
- Cortés, 1999 CORTÉS, O. A. C., "Desenvolvimento e Avaliação de Algoritmos Numéricos Paralelos", Dissertação de Mestrado, Instituto de Ciências e Matemáticas e de Computação (ICMC), Universidade de São Paulo (USP), Março, 1999.
- Coulouris et. al., 1988 COULOURIS, G., Dollimore, J., Kindberg, T., "Distributed Systems Concepts and Design", Addison-Wesley Publishing Company, 1988.
- Coulouris et. al., 1994 COULOURIS, G., Dollimore, J., Kindberg, T., "Distributed Systems Concepts and Design", 2ª ed., Addison-Wesley Publishing Company, 1994.
- Dantas, 1996 DANTAS, M. A. R., *Efficient Scheduling of Parallel Applications on Workstation Clusters*", Thesis of Doctor, University of Southampton, 1996.
- Dongarra & Dunigan, 1997 DONGARRA, J. J., Dunigan, T., "Message-passing performance of various computers", Concurrency: Practice and Experience, Vol. 9(10), pp. 915-926, October 1997.

- Dongarra et. al., 1998 DONGARRA, J., Geist, A., Kohl, J. A., Papadopoulos, P. M., Sunderam, V., *"HARNESS: Heterogeneous Adaptable Reconfigurable Neyworked SystemS"*, <http://www.EPM.ORNL>
- Duncan, 1990 DUNCAN, R., *"A Survey of Parallel Computer Architectures"*, IEEE Computer, pp. 5-16, 1990.
- Eskicioglu, 1989 ESKICIOGLU, M. R., *"Design Issues of Process Migration Facilities in Distributed Systems"*, IEEE Technical Committee on Operating Systems Newsletter, v. 4, pp. 3-13, Inverno de 1989.
- Feitelson & Rudolph, 1995 FEITELSON, D.G., Rudolph, L., *"Parallel Job Scheduling: Issues and Approaches"*, IPPS' 95 Workshop on Job Scheduling Strategies for Parallel Processing, Santa Barbara, CA, USA . Lecture Notes in Computer Sciences 949, April 1995.
- Feitelson & Rudolph, 1996 FEITELSON, D. G., Rudolph, L., *"Toward Convergence in Job Schedulers for Parallel Supercomputers"*, IPPS' 96 Workshop on Job Scheduling Strategies for Parallel Processing, Santa Barbara, Honolulu, Hawaii, Lecture Notes in Computer Sciences 1162, April 1996.
- Feitelson et. al., 1997 FEITELSON, D. G., Rudolph, L., Schwiegelshohn, U., Sevcik, K. C., Wong, P., *"Theory and Practice in Parallel Job Scheduling"*, IPPS' 97 Workshop on Job Scheduling Strategies for Parallel Processing, Santa Barbara, Geneva, Switzerland, Lecture Notes in Computer Sciences 1291, April 1997.
- Ferrari & Zhou, 1987 FERRARI, D., Zhou, S., *"An Empirical Investigation of Load Indices for Load Balancing Applications"*, Proc. Performance'87, Amsterdam, 1987, pp. 515-528.
- Fischer & Dongarra, 1996 FISCHER, M., Dongarra, J., *Another Architecture: PVM on Windows95/NT*, 3^o EuroPVM'96, Third Euro PVM User's Group Meeting, Munich, Germany, October, 1996.
- Flower & Holawa, 1994 FLOWER, J., Kolawa A., *"Express is not just a message passing system. Current and future directions in Express"*, Parallel Computing, v. 20, pp. 597-614, 1994.
- Flynn, 1972 FLYNN, M. J., *"Some Computer Organizations and Their Effectiveness"*, IEEE Transactions on Computers, v. C-21, pp. 948-960, 1972.

- Flynn, 1996 FLYNN, M. J., Rudd, K. W., *"Parallel Architectures"*, ACM Computing Surveys, v. 28, n° 1, 1996.
- Foster, 1995 FOSTER, Y., *"Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering"*, Addison-Wesley Publishing Company Inc., 1995.
- Franklin & Govindan, 1996 FRANKLIN, M.A., Govindan, V., *"A general matrix iterative model for dinamic load balancing"*, Parallel Computing, Vol. 22, 1996, pp. 969-989.
- Geist et. al., 1994 GEIST, A., Beguelin, A., Dongarra, J., Jiang, W. Manchek, R., Sunderam, V., *"PVM 3 User's Guide and Reference Manual"*, Oak National Laboratory, 1994.
- Geist et. al., 1996 GEIST, G. A., Kohl, J. A., Papadopoulos, P. M., *"PVM and MPI: a comparison of feature"*, U. S. Department of Energy, 1996.
- Geist et. al., 1997 GEIST, G. A., Kohl, J. A., Papadopoulos, P. M., Scott, S. L., *"Beyond PVM 3.4: What We've Learned, What's Next, and Why"*, Proceedings of EuroPVM-MPI'97, Cracow-Poland, LNCS 1332, pp. 116-126, 1997.
- Gibbons, 1997 GIBBONS, R., *"A Historical Application Profiler for Use by Parallel Schedulers"*, IPPS' 97 Workshop on Job Scheduling Strategies for Parallel Processing, Santa Barbara, Geneva, Switzerland, Lecture Notes in Computer Sciences 1291, April 1997.
- Grant & Skjellum, 1992 GRANT, B. K., Skjellum, A., *"The PVM Systems: An In-Depth Analysis and Documenting Study-Concise Edition"*, Relatório Técnico, Lawrence Livermore National Lab., DE 93011959, 20 pp., Novembro de 1992.
- Harchol-Balter & Downey, 1997 HARCHOL-BALTER, M., Downey, A., *"Exploiting Process Lifetime Distributions for Dynamic Load Balancing"*, ACM Transactions on Computer Systems Vol. 15, No. 3, August 1997.
- Harchol-Balter et. al., 1997 HARCHOL-BALTER, M., Crovella, M., and Murta, C., *"On Choosing a Task Assignment Policy for a Distributed Server System"*, <http://http.cs.berkeley.edu/~harchol/Papers/papers.html>.
- Hwang, 1993 HWANG, K., *"Advanced Computer Architecture: Parallelism, Scalability, Programmability"*, Illinois: McGraw-Hill, 1993.

- Jackson & Humphres, 1997 JACKSON, D. J., Humphres, C. W., "A Simple yet Effective Load Balancing Extension to the PVM Software System", *Parallel Computer*, v. 22, 1997, pp. 1647-1660.
- Kaplan & Nelson, 1994 Kaplan, J.A., Nelson, M.L., "A comparison of Queueing, Cluster and Distributed Computing Systems", Nasa Langley Research Center, June 1994.
- Kimer, 1991 KIRNER, C., "Arquitetura de Sistemas Avançados de Computação", Anais da Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho, pp. 307-353, 1991.
- Kunz, 1991 KUNZ, T. "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme", *IEEE Transactions on Software Engineering*, Vol. 17, No.7, pp. 725-730, July 1991.
- Lewis, 1996 LEWIS, F. D., "Solving NP-Complete Problems", Texto do Curso de Algoritmos Heurísticos, páginas HTML (<http://al.cs.engr.uky.edu:80/~lewis/cs-heuristic>), 1996.
- Loh et. al., 1996 LOH, P.K.K., Hsu, W.J., Wentong, C., Sriskanthan, N., "How Network Topology Affects Dynamic Load Balancing", *IEEE Parallel & Distributed Technology*, Vol. 04 No. 03, 1996, pp. 25-35.
- Lüling et. al., 1993 LÜLING, R., Monien, B., Ramme, F., "Load Balancing in Large Networks: A Comparative Study", Relatório Técnico do Departamento de Matemática e Ciência da Computação, Universidade de Paderborn (Alemanha), 1993.
- Martin, 1999 MARTIN, P., "Research plan: a new taxonomy for load distribution systems". Home Page available at: www.mcs.vuw.ac.nz/~pmar.
- Matrone et. al., 1993 MATRONE, A., Schiano, P., Puoti, V., "Linda and PVM: A comparison between two environments for parallel programming", *Parallel Computing*, v.19, pp. 949-957, 1993.
- Mcbryan, 1994 MCBRYAN, O. A., "An overview of message passing environments", *Parallel Computing*, v. 20, pp. 417-444, 1994.

- Mehra & Wah, 1993 MEHRA, P., Wah, B. W., "*Automated Learning of Load-Balancing Strategies for a Distributed Computer System*", University of Illinois at Urbana-Champaign, 1993. Available on <ftp.ibr.cs.tu-bs.de> (directory with online thesis and papers).
- Milojicic et. al., 1993 MILOJICIC, D.S., Giese, P., Zint, W., "*Load Distribution on Microkernels*", IEEE Workshop "Future Trends in Distributed Computing Systems", September 1993.
- Müllender, 1993 MÜLLENDER, "*Distributed Systems*", 2^a. Ed., ACM PRESS Frontier Series, Addison-Wesley Publishing Company, 1993.
- Overeinder et. al., 1996 Overeinder, B. J., Sloot, P. M. A., Heederik, R. N., Hertzberger, L. O., "*A dynamic load balancing system for parallel cluster computing*", FGCS, Vol. 12, pp. 101-115, 1996.
- Quealy, 1994 QUEALY, A., "*An Introduction to Message Passing*", slides, NASA Lewis Research Center, http://www.lerc.nasa.gov/Other_Groups/IFMD/2620/tutorialPP.html, 1994.
- Quinn, 1987 QUINN, M. J., "*Designing Efficient Algorithms for Parallel Computers*", McGraw Hill, 1987.
- Quinn, 1994 QUINN, M. J., "*Parallel Computing: Theory and Practice*", McGraw Hill, 1994.
- Saphir et. al., 1995 SAPHIR, W., Tanner, L. A., Traversat, B., "*Job Management Requirements for NAS Parallel Systems and Clusters*", IPPS' 95 Workshop on Job Scheduling Strategies for Parallel Processing, Santa Barbara, CA, USA, Lecture Notes in Computer Sciences 949, April 1995.
- Schnor et. al., 1997 SCHNOR, B., Petri, S., Langendörfer, H., "*Load Management for Load Balancing on Heterogeneous Platforms: A Comparison of Traditional and Neural Network Based Approaches*", Lecture Notes in computer Sciences 1124, pp. 615-620.
- Schnor et. al., 1997 SCHNOR, B., Petri, S., Langendörfer, H., "*Load Management for Load Balancing on Heterogeneous Platforms: A Comparison of Traditional and Neural Network Based Approaches*", Lecture Notes in computer Sciences 1124, pp. 615-620.

- Shirazi et. al., 1995 SHIRAZI, B. A., Hurson, A. R., Kavi, K. M., "*Scheduling and Load Balancing in Parallel and Distributed Systems*", IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 1-6, 1995.
- Shivaratri et. al., 1992 SHIVARATRI, N. G., Kreuger, P., Singhal, M., "*Load Distribution for Locally Distributed Systems*", Computer, v. 25, December, 1992, pp. 33-44.
- Snir et. al., 1996 SNIR, M., Otto, S., Steven, H., Walker, D., Dongarra, J. J., MPI: The Complete Reference, The MIT Press, Massachusetts, 1996.
- Song et. al., 1997 SONG, J., Choo, H. K., Lee, K. M., "*Application-Level Load Migration and its Implementation on top of PVM*", Concurrency: Practice and Experience, v. 9, pp. 1-19, 1997.
- Souza et. al., 1998 SOUZA, P. S. L., Santana, M. J., Santana, R. H. C., "*Escalonamento de Processos: características, divergências e soluções*", Semana Paraense de Informática (SEPAI'98), Belém, Pa, 1998.
- Souza et. al., 1999a SOUZA, P. S. L., Santana, M. J., Santana, R. H. C., "*AMIGO-A Dynamical Flexible Scheduling Environment*", Third World Multiconference on Systemics, Cybernetics and Informatics (SCI'99) and the Fifth International Conference on information Systems Analysis and Synthesis (ISAS'99), Orlando, USA, 1999 (no prelo).
- Souza et. al., 1999b SOUZA, P. S. L., Santana, M. J., Santana, R. H. C., "*A New Scheduling Environment for Near-Optimal Performance*", International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), 1999, Monte Carlo, Las Vegas, Nevada, USA, 1999 (no prelo).
- Souza et. al., 1999c SOUZA, P. S. L., Santana, M. J., Santana, R. H. C., "*PVM and its Relationship with a Dynamical and Flexivel Scheduling*", artigo submetido para EuroPVM/MPI'99, Espanha, 1999 (no prelo).
- Souza, 1996a SOUZA, P. S. L., "*Máquina Paralela Virtual em Ambiente Windows*", Dissertação de Mestrado, Instituto de Ciências Matemáticas e Computação (ICMC-USP), Maio de 1996.

- Souza, 1996b SOUZA, M. A., "*Avaliação das Rotinas de Comunicação Ponto-a-Ponto do MPI*", Dissertação de Mestrado, Instituto de Ciências Matemáticas e Computação (ICMC-USP), 1996.
- Squillante, 1995 SQUILLANTE, M.S., "*On the Benefits and Limitations of Dynamic Partitioning in Parallel Computer Systems*", IPPS' 95 Workshop on Job Scheduling Strategies for Parallel Processing, Santa Barbara, CA, USA, Lecture Notes in Computer Sciences 949, April 1995.
- Stellner & Trinitis, 1997 STELLNER, G., Trinitis, J., "*Load Balancing Based on Process Migration for MPI*", Euro - Par'97 Parallel Processing – Third International Euro-Par Conference, Passau, Germany, Lecture Notes in Computer Sciences 1300, August 1997.
- Stevens, 1991 STEVENS, W. R., "*UNIX Network Programming*", Prentice Hall Software Series, 1991.
- Tanenbaum, 1992 TANENBAUM, A. S., "*Modern Operating Systems*", Prentice Hall, 1992.
- Tanenbaum, 1995 TANENBAUM, A. S., "*Distributed Operating Systems*", Prentice Hall, 1995.
- Theimer & Hayes, 1991 THEIMER, M. M., Hayes, B., "*Heterogeneous Process Migration by Recompilation*", Anais da 11ª Conferência Internacional IEEE sobre Sistemas de Computação Distribuída, pp. 18-25, 1991.
- Togneri, 1998 TOGNERI, R., "*Parallel Program Analysis on Workstation Clusters: Memory Utilisation and Load Balancing*", Concurrency: Practice and Experience, Vol. 10(8), pp. 631-653, 1998.
- Walker, 1994 WALKER, D. W., "*The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers*", Parallel Computing, V. 20, pp. 657-673, 1994.
- Wang & Morris, 1995 WANG, Y. T., Morris, J. T., "*Load Sharing in Distributed Systems*", IEEE Transactions on Parallel and Distributed Systems, March, 1995, pp. 204-217.
- Wang et. al., 1998 WANG, K. Y., Marinescu, D. C., Carbutar, O. F., "*Dynamic Scheduling of Process Groups*", Concurrency: Practice and Experience, Vol. 10(4), pp. 265-283, April 1998.

- Welsh, 1995 WELSH, M., "*Linux Installation and getting started*", 1995.
- Xu & Lau, 1997 XU, C., Lau, F. C. M., "*Load Balancing in Parallel Computers-Theory and Practice*", Kluwer Academic Publishers, 1997.
- Zaluska, 1991 ZALUSKA, E. J., "*Research Lines in Distributed Computing Systems and Concurrent Computation*", Anais do Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software, ICMC-USP, pp. 132-155, 1991.

Apêndice A

Interface do AMIGOD

Este apêndice fornece uma descrição sucinta das rotinas especificadas pelo doutorado mentor do AMIGO e responsáveis pela interface da camada inferior desse ambiente de escalonamento. O apêndice está organizado de acordo com a funcionalidade de grupos de rotinas.

A.1 Iniciar o AMIGOD

NULL main(CHAR * IP:PORT TCP, CHAR * IP:PORT UDP)

Tem por objetivo preparar o ambiente para a execução do AMIGOD. Recebe 2 argumentos da linha de comando: PORT TCP para conexão com ambiente de passagem de mensagens local e pode receber também IP:PORT UDP para conexão com um AMIGOD remoto, se este existir.

A.2 Função para troca de mensagens - loop infinito

WorkUntilDeath()

Responsável pelo roteamento de todas as mensagens trocadas na camada inferior do AMIGO, recebendo e enviando mensagens para o ambiente de passagem de mensagem local, às políticas locais ativas ou para o AMIGOD remoto.

A.3 Interface com o ambiente de passagem de mensagem local.

NULL RemoveLocalAMIGOD(void)

Na remoção de um AMIGOD, cancela o IPC com ambiente de passagem de mensagem, pára a execução do AMIGOD local e avisa todos os interessados (políticas locais e AMIGODs remotos).

*INT RequestReceivers(BYTE *Message)*

No escalonamento inicial de tarefas, recebe o pedido do ambiente de passagem de mensagem, através da mensagem AM_SCHED, solicitando uma relação de possíveis receptores para atribuir *n* tarefas, e a repassa às políticas, através da mensagem AP_SCHEDACK. Essa mensagem deve conter o nome do executável, o número de instâncias a criar e a arquitetura desejada.

*INT ReplyReceivers(BYTE *Message)*

Recebe da política a relação de receptores (mensagem AP_SCHEDACK) e envia uma mensagem TCP ao ambiente de passagem de mensagem local informando a relação de *hosts* escolhidos para serem os receptores das novas tarefas, através da mensagem AM_SCHEDACK.

A.4 Interface com os outros AMIGODs.

*INT PutHost(INT NumElemHostDest, LONG INT *HostDest, Char Fwd, INT NumHostsInserir, INT * HostsInserir)*

Na inclusão de um AMIGOD, informa a existência e o port UDP do novo *host*, através da mensagem de código AA_ADD. Essa mensagem deve conter a relação dos novos *hosts* a inserir (*NumHostsInserir*) na tabela de roteamento do(s) *HostDest*(s) e se os valores informados devem ser repassados aos *hosts* existentes na tabela do(s) *HostDest*(s), parâmetro *fwd*. Retorna 0 se sucesso da operação e menor que zero se fracasso.

INT GetHost(BYTE * Message)

Receber a relação informando a existência e o port UDP de novos *hosts* (mensagem AA_ADD) que devem ser inseridos na tabela roteamento do *host* local. Se o parâmetro fwd estiver setado, então essa função deve enviar o conteúdo para todos os *hosts* previamente existentes na tabela, através da função *PutHost()*. Além disso, essa função deve enviar para o *host* que solicitou a tabela completa, excluindo a indicação local e a do novo *host*.

NULL RemoveRemoteAMIGOD(BYTE *Message)

Na remoção de um AMIGOD, remove da tabela de roteamento local os *hosts* inseridos na mensagem AA_REM e envia uma mensagem para as políticas ativas avisando e pedindo a remoção dos *hosts* (mensagem AP_REMHOST).

NULL DetectRemoteAMIGODDeath(CHAR *NameHost)

Elimina o *host* remoto da tabela de roteamento e envia mensagem AA_REM para todos AMIGOD's existentes, informando a perda do *host*.

NULL AMIGODGetLoadLocal(BYTE *Message)

Na atualização da Política de Informação, solicita carga local para a sua política, retira da mensagem AA_GETLOAD a carga remota e envia a mensagem AP_GETLOADLOCAL para a política local informando, se for válida, a carga remota.

NULL AMIGODPutLoadRemote(BYTE *Message)

Responsável por informar carga remota, retira da mensagem AA_PUTLOAD a carga remota para ser atualizada e envia a mensagem AP_PUTLOADREMOTE para a política local contendo carga remota.

NULL ChangePolicy(BYTE *Message)

Para mudar as políticas ativas, retira da mensagem AA_CHANGEPOL o número e a relação de políticas a serem excluídas e a serem incluídas, para isso chama as funções *RemovePolicies* e *StartPolicy*(), respectivamente, e, por fim, atualiza a tabela de políticas.

NULL DetectRemotePolicyDeath(ORIGEM Pip:rótulo)

Informa a política local da "morte" da sua política "irmã", envia mensagem AP_POLICYDEATH para a política local, solicitando a exclusão da política remota das tabelas existentes.

NULL AMIGODGetPartner(BYTE *Message)

Na política de localização, repassa à política local o pedido de parceiros, feito através da mensagem AA_GETPART, enviando a mensagem AP_SCHEGETPART à política solicitando parceiros.

NULL AMIGODPutPartner(BYTE *Message)

Depois de receber a mensagem AA_PUTPART de um AMIGOD remoto, envia mensagem AP_SCHEPUTPART para a política local informando a relação de parceiros solicitados anteriormente.

A.5 Interface com as políticas ativas

VOID StartPolicy (INT NumberPolExec, CHAR * NamePol, INT PORT TCP)

Objetivando incluir políticas ativas, executa a(s) política(s) solicitada(s) nos parâmetros e aguarda o pedido de conexão TCP vindo de cada nova política local criada.

INT RemovePolicies(INT NumberPol, BYTE *Policies)

Para a remoção de políticas ativas, verifica se todas as políticas locais relacionadas são válidas (estão executando) e envia uma mensagem AP_REMPOL para cada uma dessas políticas para que elas parem a sua execução.

NULL DetectPolicyDeath ()

Depois de detectar a "morte" de uma política local, envia mensagem AA_POLICYDEATH para as políticas remotas que constam na relação de AMIGOD's remotos, informando a perda da política e solicitando a exclusão desta política local das tabelas no *host* remoto.

NULL PolGetLoadRemote(BYTE *Message)

Tem por objetivo atualizar a política local com cargas remotas. Para tanto, recebe mensagem AP_GETLOADREMOTE da política local pedindo a carga de outras máquinas (todas ou as especificadas no vetor), retira a carga local dessa mensagem e envia mensagem AA_GETLOAD para cada *host* relacionado, solicitando a carga destes e informando, se houver, a carga local.

NULL PolPutLoadLocal(BYTE *Message)

Recebe a mensagem AP_PUTLOADLOCAL da política pedindo que a carga local seja enviada para outras máquinas (todas ou as especificadas no vetor) e/ou fique no cache do AMIGOD e envia a carga local, através da mensagem AA_PUTLOAD, para os *hosts* relacionados na mensagem AP_PUTLOADLOCAL.

NULL PolGetPartner(BYTE *Message)

Recebe a mensagem AP_POLGETPART da política local pedindo uma relação de parceiros e repassa o pedido ao AMIGOD coordenador, através da mensagem AA_GETPART.

NULL PolPutPartner(BYTE *Message)

Recebe a mensagem AP_POLPUTPART da política local informando a relação de parceiros solicitados e, por intermédio do AMIGOD remoto, repassa a relação para a política remota usando a mensagem AA_PUTPART.

A.6 Mudanças no ambiente de passagem de mensagens

NULL StartAMIGOD(VOID)

Essa função deve ser incluída no código fonte do ambiente de passagem de mensagem, e tem por objetivo criar todo o ambiente necessário para que o AAMIGODD possa ser iniciado. Essa função deve iniciar o AMIGOD com o port TCP para a conexão AMIGOD/ambiente de passagem de mensagem local, caso esse não seja o primeiro AMIGOD criado, então essa função deve repassar também o port UDP de um AMIGOD remoto, para que esses possam se comunicar.

NULL RemoveAMIGOD()

Na exclusão de um *host*, desfaz o ambiente necessário para executar o AMIGOD, enviando a mensagem AM_REM para que o AMIGOD local possa se auto eliminar. Essa função deve cancelar o port TCP usado para comunicação entre o ambiente de passagem de mensagem e o AMIGOD.

Char * GetReceivers(CHAR *FileNameExec, INT num, CHAR *Arch)

No escalonamento inicial das tarefas, obtêm uma relação de *hosts* receptores. Para isso, envia uma mensagem AM_SCHED ao AMIGOD local e recebe a mensagem AM_SCHEDACK com a relação de receptores.

A.7 Interação com a camada superior do AMIGO

*policies *GetActive Policy()*

Deve encontrar e ler o arquivo contendo relação de políticas ativas para retornar uma estrutura de dados com relação de políticas padrão e demais informações. Caso essa função não encontre o arquivo de configuração, deve obter o arquivo PADRÃO de configuração.

*policies *ChangeActive Policy()*

Em caso de mudanças nas políticas ativas, trata o recebimento de sinal assíncrono vindo do AMIGO, configurando quais políticas devem estar ativas. Para isso, verifica quais políticas atualmente ativas devem deixar de ser executadas, inicia a execução de novas políticas (aquelas que não estavam executando e que agora estão no arquivo de configuração). Envia mensagem AA_CHANGEPOL para os AMIGOD's remotos informando quais políticas devem ser paralisadas e quais políticas devem ser iniciadas e atualiza tabela de políticas.

Especificação das Mensagens

Este apêndice apresenta uma descrição de todas as mensagens especificadas e trocadas na camada inferior do AMIGO. As mensagens estão divididas de acordo com o grupo que está mandando e/ou recebendo mensagens em relação ao módulo AMIGOD.

B.1 Protocolo com o Ambiente de Passagem de Mensagem

AM_REM

Essa mensagem é enviada pelo ambiente de passagem de mensagem para o AMIGOD local solicitando a remoção de um AMIGOD.

Conteúdo: NULO.

AM_SCHED

O ambiente de passagem de mensagem local envia para o AMIGOD local pedindo relação de possíveis receptores.

Conteúdo: nome executável, núm. de instâncias a criar e a arquitetura.

AM_SCHEDACK

O AMIGOD informa ao ambiente de passagem de mensagem local a relação das máquinas que receberão as novas tarefas.

Conteúdo: número de receptores, relação de *hosts* receptores para as tarefas.

B.2 Protocolo com os outros AMIGOD's

AA_ADD

O AMIGOD local solicita para um AMIGOD remoto a sua inclusão na tabela de roteamento.

Conteúdo: fwd para indicar se o conteúdo é para ser repassado, relação de AMIGOD's a incluir.

AA_REM

Um AMIGOD informa a remoção de um outro AMIGOD.

Conteúdo: número de *hosts* a excluir, relação de *hosts* a excluir.

AA_GETLOAD

Um AMIGOD solicita a carga remota de um outro AMIGOD.

Conteúdo: carga local para atualização remota.

AA_PUTLOAD

AMIGOD local informa carga local para AMIGOD's;

Conteúdo: carga local para atualização remota.

AA_CHANGEPOL

Pedido para remoção/inclusão de políticas.

Conteúdo: número de políticas a excluir, relação de políticas a excluir,
número de políticas a incluir, relação de políticas a incluir.

AA_POLICYDEATH

AMIGOD detecta "morte" de uma política e avisa os demais AMIGOD's.

Conteúdo: política a excluir.

AA_GETPART

AMIGOD local após receber pedido de parceiros da política local, repassa o pedido para outros AMIGOD's remotos.

Conteúdo: determinado pela política de localização

AA_PUTPART

AMIGOD após receber relação de parceiros da política local repassa relação para o AMIGOD remoto que solicitou.

Conteúdo: determinado pela política de localização.

B.3 Protocolo com as Políticas Ativas

AP_GETLOADREMOTE

Política local ativa solicita do AMIGOD local cargas remotas.

Conteúdo: carga local para atualização remota, quantidade e a relação dos *hosts* solicitados.

AP_PUTLOADLOCAL

Política local informa para o AMIGOD local a sua carga.

Conteúdo: carga local, número de *hosts*, relação de *hosts* para informar a carga.

AP_GETLOADLOCAL

AMIGOD local pede a carga local da política ativa.

Conteúdo: carga remota do *host* que solicitou a carga local

AP_PUTLOADREMOTE

AMIGOD local após receber mensagem de um AMIGOD remoto com a sua carga informa carga remota para a política local.

Conteúdo: carga remota.

AP_POLGETPART

Política de localização pede parceiros para o AMIGOD local.

Conteúdo: o nome do AMIGOD coordenador, núm. parceiros desejados, tipo de parceiros (transmissor ou receptor).

AP_SCHEGETPART

AMIGOD solicita parceiros.

Conteúdo: núm. parceiros desejados, tipo de parceiros (transmissor/receptor).

AP_POLPUTPART

Política de localização informa parceiros.

Conteúdo: número de parceiros, relação de parceiros solicitados.

AP_SCHEPUTPART

AMIGOD informa parceiros .

Conteúdo: número de parceiros, relação de parceiros solicitados.

AP_SCHED

AMIGOD após receber mensagem do ambiente de passagem de mensagem solicitando receptores repassa essa mensagem para a política local ativa.

Conteúdo: nome executável, núm. instâncias a criar e arquitetura .

AP_SCHEDACK

Após receber mensagem do AMIGOD local a política ativa repassa a relação de receptores.

Conteúdo: número de receptores, relação de possíveis receptores.

AP_PUTTABLE

AMIGOD envia a tabela de roteamento à política.

Conteúdo: número de *hosts* da relação, relação de *hosts*.

AP_REMPOL

AMIGOD solicita que a política pare a sua execução.

Conteúdo: nulo.

AP_REMHOST

AMIGOD solicita que os *hosts* remotos inseridos na mensagem sejam eliminados das tabelas.

Conteúdo: número de *hosts*, relação de *hosts* remotos a excluir.

AP_POLICYDEATH

AMIGOD remoto comunica perda de uma política.

Conteúdo: nome (Pip:rótulo) da política remota a eliminar das tabelas.

Apêndice C

Interface da DPWP

O objetivo deste apêndice é fornecer uma descrição de todas as funções implementadas na DPWP. O apêndice está organizado de acordo com a funcionalidade de cada função.

C.1 Iniciar a DPWP

*NULL main(CHAR * IP:PORT TCP)*

Tem por objetivo preparar o ambiente para a execução da DPWP. Recebe como argumento da linha de comando um port TCP para conexão com o AMIGOD local. Em seguida, conecta com o AMIGOD e cria a estrutura de dados para armazenar informações sobre os demais *hosts*

C.2 Passagem de Mensagem

Void WorkForever(void)

Responsável pelo roteamento de todas as mensagens trocadas na DPWP, recebendo e enviando mensagens para o AMIGOD local.

*Void put_header(char *buffer, short int cod_message, struct module_addr dest, struct module_addr source, short int len_message, char *message)*

Essa função é responsável em empacotar todos os dados do cabeçalho junto com o conteúdo da mensagem e armazená-los no *buffer* que será enviado.

*Void get_header(char *buffer, short int cod, char *dest.module, int *dest.ip, char *dest.label, char *src.module, int *src.ip, char *src.label, short int len_message, char *message)*

Essa função é responsável em desempacotar a mensagem recebida e conseqüentemente retirar todas as informações contidas no cabeçalho da mensagem.

*Void get_message(char * message)*

Essa função retira o código da mensagem (*message*) recebida e chama a função responsável por tratar cada mensagem recebida.

C.3 Manipulação da *dpwp_table*

Int count_host(void)

Essa função tem a finalidade de contar quantos *hosts* existem na tabela *dpwp_table* com exceção do *host* local.

*Void freehost(struct dpwp_table *info)*

Essa função quando ativada libera o espaço alocado pela variável apontada pela estrutura **info*.

*Void updatehost(struct dpwp_table *info)*

Essa função tem a finalidade de atualizar a tabela *dpwp_table* todas as vezes que for chamada. Essa função pesquisa se a variável **info* já existe na tabela, caso positivo atualiza sua carga, caso negativo insere esse elemento na tabela.

Int inserthost(struct dpwp_table *info)

Essa função insere o elemento apontado por **info* na tabela local. A diferença entre essa função e a *updatehost()* é que essa função não pesquisa para saber se esse elemento já existe, ele sempre insere, e caso já exista haverá dois iguais, e no caso da *updatehost()* o elemento só será inserido caso não exista na tabela.

Void deletehost (unsigned long iphost)

Essa função pesquisa na tabela local a estrutura que tem o ip igual ao *iphost* e retira esse elemento da tabela local.

Void getdpwp_table(unsigned long **tablehost)

Essa função deve preparar um vetor com todos os elementos existentes na tabela local, com exceção do *host* local.

Struct dpwp-table *gethost(void)

Essa função aloca espaço na tabela local para que um novo elemento possa ser inserido.

Struct dpwp_table *searchhost(unsigned long ip_host)

Essa função tem por objetivo pesquisar em toda a tabela *dpwp_table* local o elemento que contém o ip igual ao *ip_host* e devolver toda essa estrutura.

Int Get_Table(void)

Essa função deve mandar uma mensagem para o AMIGOD com o código AP_GETTABLE, solicitando a tabela de roteamento local do AMIGOD. Devolve 0 se sucesso da operação e menor que zero se a operação falha.

Void Put_Table(char *buffer)

Essa função tem a finalidade de desempacotar a mensagem recebida em *buffer* e inserir na tabela local os ip's dos *hosts* pertencentes a máquina virtual. Em seguida, essa função solicita a carga remota dos demais AMIGODs.

C.4 Manipulação do Índice de Carga

Int putloadlocal(void)

Essa função tem a finalidade de empacotar a carga local e repassa-la aos AMIGODs pertencentes à máquina virtual. Retorna zero se sucesso e menor que zero se aconteceu algum problema com a função.

Int getloadremote(void)

O algoritmo DPWP deve empacotar e enviar a mensagem solicitando a carga remota dos outros AMIGODs.

Int get_status(void)

Essa função quando ativada recupera o estado atual da máquina. É retornado um inteiro que representa o estado no qual a máquina está, isto é, sobrecarregado, ocioso e moderado.

Int update_table(float load)

Atualiza as tabelas remotas com a nova carga local. Essa função é ativada sempre que ocorre uma mudança de estado no *host* local. Essa Função realiza o papel da política de informação da DPWP.

Void get_loadlocal(char *buffer)

Essa função deve desempacotar a mensagem recebida no *buffer*, retirar a carga remota, atualizar a tabela local. Além disso, essa função deve empacotar a carga local e enviar ao *host* que solicitou a carga.

Void put_loadremote(char *buffer)

Essa função deve desempacotar a mensagem recebida no *buffer*, retirar a carga remota e atualizar a tabela local. Essa função difere da *get_loadlocal()*, pois não há necessidade de mandar a carga local para o *host* origem.

Float get_averageload(int carga, int *vetorccarga, int tamanho)

Essa função é a responsável por implementar a técnica da janela deslizante, a qual calcula a média da carga nos últimos *n* segundos e diminui a possibilidade de adotarmos uma carga que não corresponda a realidade do sistema.

C.5 Seleção das Máquinas Receptoras

Void get_receivers(char *buffer)

Essa função deve desempacotar a mensagem contida no *buffer*. Em seguida deve retirar o número de tarefas que devem ser criadas, o arquivo executável e o tipo de arquitetura. Com essas informações seleciona-se as melhores máquinas para receber essas tarefas. Essas máquinas são armazenadas em um vetor de ip's.

Int send_receivers(int tamanho, unsigned long *vet)

Essa função recebe uma lista de máquinas receptoras e tem a finalidade de empacotá-las e enviá-las para o AMIGOD que por sua vez deve repassar a mensagem para o ambiente de passagem de mensagem.

C.6 Cálculo do Índice de Carga

*Void *xalloc(void *a, int b)*

*Void *xmalloc(unsigned int b)*

*Void *xrealloc(void *inf, unsigned int p)*

Void closeproc(PROCTAB tab)*

*Void freeproc(proc_t *th)*

*Void stat2proc(char *stat, proc_t *th)*

*Void statm2proc(char *stat, procc_t *th)*

*Void nulls2sep(char *sp, int p, char h)*

*Int file2str(char *fh, char *tp, char *inf, int th)*

*Char **file2strvec(char *fs, char *st)*

*proc_t *readproc(PROCTAB* tab, proc_t *pt)*

*proc_t **ReadProcTab(proc_t **pt, int p)*

*PROCTAB *openproc(int tb, ...)*

Int getindexload(void)

Todas essas funções são responsáveis em calcular o número de processos prontos na fila da CPU, o qual é o índice de carga adotado pela DPWP.

Aplicações Paralelas Usadas nos Testes

Este apêndice descreve as duas aplicações paralelas utilizadas nos testes para validar e avaliar o algoritmo DPWP. O primeiro exemplo mostra a aplicação do método do trapézio que caracteriza-se como uma aplicação CPU-bound. A Segunda aplicação é a multiplicação de matrizes, que apesar de consumir bastante processamento não é tida como uma aplicação CPU-bound devido a sua comunicação.

D.1 Método do Trapézio

Essa aplicação tem por objetivo calcular uma integral através da aproximação de diversos trapézios. A Figura D.1 mostra os passos necessários para que essa aplicação seja paralelizada.

É importante ressaltar que a quantidade de trapézio dentro da área desejada deve ser divisível pelo número de tarefas criadas.

Uma integral que no modo seqüencial foi dividida, por exemplo, em 9 (nove) milhões de trapézios em 3 (três) processadores cada um foi responsável por calcular uma integral com 3 (três) milhões de divisões (ou trapézios).

A seguir é mostrado o código dessa aplicação, primeiramente é detalhado o código do programa mestre e em seguida o código dos programas escravos.

Figura D. 1 – Paralelização do algoritmo do trapézio composto

```

01 //Codigo Fonte: integralpvm.c
02 // O processo integralpvm é o processo mestre. Ele gera os
03 // processos escravos, responsaveis por resolver uma parte
04 // das integrais.
05
06 #include <stdio.h>
07 #include <stdlib.h>
08 #include "time.h"
09 #include <math.h>
10 #include "pvm3.h"
11
12 #define ESCRAVO "/home/aleteia/teste/integral/escravo"
13 #define N 300000 /*quantidade de divisoes*/
14 #define NPROCS 5 /*quantidade de escravos*/
15 #define NP 6 /*quantidade de tarefas*/
16
17 //static char *hosts[]={"lasd06","lasd08"};
18 double s = 0,sum = 0;
19 struct timeval tempo1, tempo2;
20 struct timezone tzp;
21 double tempo;
22 double a,b;
23 double limite_inferior,limite_superior;
24 double x;
25 double resultado,res,valor_retorno;
26 int info, infos;
27
28 float integral(double a, double b, int n)
29 {
30     double h,var,r,s = 0, aux;
31     int i; // numero de termos
32     h = (double)(b-a)/n;
33     s = s + sin(a) + exp(a);
34     s = s + sin(b) + exp(b);
35     aux = a;
36     for(i=1;i<n-1;i++){
37         var = aux + h;
38         r = 2 * (sin(var) + exp(var));
39         s = s + r;
40         aux = var;
41     }
42     return((h/2) * s);
43 }
44
45
46 main(int argc, char **argv)
47 {
48     int numtasks; /* numero total de processos */

```

```

49     int mytid;                /* Identif. do processo mestre */
50     int tids[NPROCS];        /* Ident. processos escravos 51*/
52     int cc;                  /* Numero de processos criados */
53     int tid;                 /* Ident do processo escravo */
54     int i;                   /* Contador */
55
56
57     mytid = pvm_mytid ();
58     numtasks = NPROCS + 1;
59     printf("Processo mestre: %x\n", mytid);
60     // info = pvm_addhosts(hosts,2,&infos);
61     // for(i=0;i<NP-1;i++)
62         cc=pvm_spawn(ESCRAVO, (char**)0, PvmTaskHost,
63             hosts[i],1,&tids[i+1]);
64     // if (cc < 1){
65     //     printf("Nao consegui gerar escravos...\n");
66     //     exit(0);
67     // }
68
69     // for(i=0;i<NP-1;i++)
70         cc=pvm_spawn(ESCRAVO, (char**)0, PvmTaskDefault, NULL, 71
NPROCS,&tids[i+1]);
72     if (cc < NPROCS){
73         printf("Nao consegui gerar algum escravo...\n");
74         exit(0);
75     }
76     printf("Iniciando calculos...\n");
77     gettimeofday(&tempol,&tzp);
78
79     a = 0.0;
80     b = 1.2;
81
82     x = (double)(b - a)/numtasks;
83
84     limite_inferior = a;
85     limite_superior = a + x;
86     for(i=0;i<NP-1;i++){
87         pvm_initsend(PvmDataDefault);
88         pvm_pkdouble(&limite_inferior,1,1);
89         pvm_pkdouble(&limite_superior,1,1);
90         pvm_send (tids[i+1],0);
91         limite_inferior = limite_superior;
92         limite_superior = limite_superior + x;
93     }
94
95     res= integral(limite_inferior,limite_superior,
96         N/numtasks);
97

```

```

78   for(i=0;i<NP-1;i++){
79       pvm_recv(tids[i+1],4);
80       pvm_upkdouble(&valor_retorno, 1,1);
81       sum = sum + valor_retorno;
82   }
83
84   res = res + sum;
85
86   gettimeofday(&tempo2,&tzp);
87
88   tempo = (double)(tempo2.tv_sec - tempo1.tv_sec) +
89((double)(tempo2.tv_usec-tempo1.tv_usec))/1000000);
90   printf("Tempo em segundos = %f\n",tempo);
91   printf("Quantidade de divisoes = %d\n",N);
92   printf("-----\n");
93   fflush(stdout);
94   //   info = pvm_delhosts(hosts,2,&infos);
95
96   pvm_exit ();
97 }

```

```
01 //Codigo Fonte: escravo.c
02 // Os processos escravo.c sao os escravos gerados pelo
04 // mestre (integralpvm.c).
05
06 #include <stdio.h>
07 #include "pvm3.h"
08 #include <stdlib.h>
09 #include "time.h"
10 #include <math.h>
11
12 #define N 2400000 /*quantidade de divisoes*/
13 #define NPROCS 5 /*numero de tarefas*/
14
15 float integral(double a, double b, int n)
16 {
17     double h,var,r,s = 0, aux;
18     int i; // numero de termos
19     h = (double) (b-a)/n;
20     s = s + sin(a) + exp(a);
21     s = s + sin(b) + exp(b);
22     aux = a;
23     for(i=1;i<n-1;i++){
24         var = aux + h;
25         r = 2 * (sin(var) + exp(var));
26         s = s + r;
27         aux = var;
28     }
29     return((h/2) * s);
30 }
31
32 main()
33 {
34     int mytid; /* Identificacao do processo escravo */
35     int master; /* Identificacao do processo mestre */
36     double a,b,res;
37
38     mytid = pvm_mytid ();
39     master = pvm_parent ();
40
41     pvm_recv(-1,-1);
42     pvm_upkdouble(&a,1,1);
43     pvm_upkdouble(&b,1,1);
44
45     res = integral(a,b,(N/NPROCS));
46
47     pvm_initsend(PvmDataDefault);
48     pvm_pkdouble(&res,1,1);
```



```
49     pvm_send(master,4);  
50     pvm_exit ();  
51 }
```

D.2 Multiplicação de Matrizes

Essa aplicação é bastante simples de ser paralelizada e foi escolhida por usar uma grande quantidade de operações em ponto flutuante, e dessa maneira, consumir muito processamento. Entretanto, não é caracterizada como uma aplicação *CPU-bound* devido as trocas de mensagens que se fazem necessárias entre mestre/escravo. A Figura D.2 mostra graficamente a paralelização do algoritmo de multiplicação de matrizes para 3 (três) processadores.

Figura D. 2 – Paralelização do algoritmo de multiplicação de matrizes

```
01 // Codigo Fonte: matrixpvm.c
02 // O processo matrixpvm.c e o processo mestre. Ele gera os
03 // processos escravos escravomatrix, responsaveis pela
04 // multiplicacao de alguns elementos da matrix.
05
06 #include <stdio.h>
07 #include <stdlib.h>
08 #include "time.h"
09 #include <math.h>
10 #include "pvm3.h"
11
12 #define ESCRAVO "/home/aleteia/teste/matrix/escravo"
13 #define NPROCS 4
14 #define TAM 50
15 #define NP 5
16
17 void gera_matriz(double [][][TAM], int);
18
19 //static char *hosts[]={"lasd06","lasd08"};
20 /* static char *hosts[]={"lasd06"}; */
21
22 main(int argc, char **argv)
23 {
24     int numtasks; /* numero total de processos */
25     int mytid; /* Identificacao do processo mestre 26 */
26     int tids[NP]; /* Identificacao dos processos escravos */
27     int cc; /* Numero de processos criados */
28     int tid; /* Identificacao do processo escravo */
29     double A[TAM][TAM], B[TAM][TAM], C[TAM][TAM];
30
31
32     int i,j,k;
33     int n_lin,total;
34
35     struct timeval tempo1, tempo2;
36     struct timezone tzp;
37     double tempo;
38     int pos,info,infos;
39
40     mytid = pvm_mytid ();
41     numtasks = NPROCS + 1;
42     printf("Processo mestre: %x\n", mytid);
43     // info = pvm_addhosts(hosts,NP-1,&infos);
44     for(i=0;i<numtasks-1;i++)
```

```

45     cc = pvm_spawn (ESCRAVO, (char**)0,
46     PvmTaskDefault,NULL,1,&tids[i+1]);
47
48     n_lin = TAM/NP;
49     total = TAM*TAM;
50     printf("Gerando matrix...\n");
51     gettimeofday(&tempo1,&tzp);
52     gera_matriz(A,1235);
53     pos = 0;
54     for(i=0;i<numtasks-1;i++){
55         pvm_init send(PvmDataDefault);
56         pvm_pkdouble(&A[0][0],(TAM*TAM),1);
57         pvm_pkint(&pos,1,1);
58         pvm_pkint(&tids[0],1,1);
59         pvm_send (tids[i+1],0);
60     }
61
62     pvm_recv(-1,-1);
63     pvm_upkdouble(&B[0][0],(TAM*TAM),1);
64
65     for(k=pos;k<pos+n_lin;k++){
66         for(i=0;i<TAM;i++){
67             for(j=0;j<TAM;j++){
68                 C[k][i] = C[k][i] + (A[k][j] * B[j][i]);
69             }
70         }
71     }
72     pos = 0;
73     for(i=0;i<numtasks-1;i++){
74         pvm_recv(tids[i+1],7);
75         pvm_upkdouble(&C[pos][0],(n_lin*TAM),1);
76         pos = pos + n_lin;
77     }
78
79     gettimeofday(&tempo2,&tzp);
80
81     tempo = (double)(tempo2.tv_sec - tempo1.tv_sec) +
82     (((double)(tempo2.tv_usec-tempo1.tv_usec))/1000000);
83     printf("Tamanho da matrix %d x %d \n",TAM,TAM);
84     printf("Tempo em segundos = %f\n",tempo);
85     printf("-----\n");
86     pvm_exit ();
87 }
88
89
90 void gera_matriz(double mat[][TAM], int init){
91
92     int i, j;

```

```
93         double norma = 0.0;
94
95         for (j = 0; j < TAM; j++) {
96             for (i = 0; i < TAM; i++) {
97                 init = 3125*init % 65536;
98                 mat[i][j] = fabs((init - 32768.0)/16384.0);
99                 norma = (mat[i][j] > norma) ? mat[i][j] :
100 norma;
101                 if (i == j)
102                     if (mat[i][j] == 0) mat[i][j] = 1;
103             }
104         }
105 }
```

```

01 // Código Fonte: escravomatrix.c
02 // Os processos escravomatrix.c são os processos escravos
03 // gerados pelo mestre (matrixpvm.c).
04 //
05
06 #include <stdio.h>
07 #include "pvm3.h"
08 #include <stdlib.h>
09 #include "time.h"
10 #include <math.h>
11
12 #define TAM 50
13 #define NP 5
14
15 void gera_matriz(double [][][TAM], int);
16
17 main()
18 {
19     int mytid;      /* Identificação do processo escravo */
20     int master;     /* Identificação do processo mestre */
21     double A[TAM][TAM], B[TAM][TAM], C[TAM][TAM];
22
23     int i,j,k,id,pos;
24     int n_lin,total;
25     mytid = pvm_mytid ();
26     master = pvm_parent ();
27     n_lin = TAM / NP;
28
29     pvm_recv(-1,-1);
30     pvm_upkdouble(&A[0][0], (TAM*TAM), 1);
31     pvm_upkint(&pos, 1, 1);
32     pvm_upkint(&id, 1, 1);
33
34     gera_matriz(B, 1235);
35     if (id == mytid){
36         pvm_initsend(PvmDataDefault);
37         pvm_pkdouble(&B[0][0], (TAM*TAM), 1);
38         pvm_send(master, 4);
39     }
40
41     for(k=pos; k<pos+n_lin; k++){
42         for(i=0; i<TAM; i++){
43             for(j=0; j<TAM; j++){
44                 C[k][i] = C[k][i] + (A[k][j] * B[j][i]);
45             }
46         }

```

```

47     }
48     pvm_initsend(PvmDataDefault);
49     pvm_pkdouble(&C[pos][0], (n_lin*TAM), 1);
50     pvm_send(master, 7);
51
52     pvm_exit ();
53 }
54
55 void gera_matriz(double mat[][TAM], int init){
56
57     int i, j;
58     double norma = 0.0;
59
60     for (j = 0; j < TAM; j++) {
61         for (i = 0; i < TAM; i++) {
62             init = 3125*init % 65536;
63             mat[i][j] = fabs((init - 32768.0)/16384.0);
64             norma = (mat[i][j] > norma) ? mat[i][j] :
65 norma;
66             if (i == j)
67                 if (mat[i][j] == 0) mat[i][j] = 1;
68         }
69     }
70 }

```