Universidade de São Paulo
Instituto de Física de São Carlos
Departamento de Física e Informática

AMIGO:
Uma Contribuição para a Convergência na Área de Escalonamento de Processos

Paulo Sergio Lopes de Souza

Tese apresentada ao Instituto de Física de São Carlos, da Universidade de São Paulo, para a obtenção do título de Doutor em Ciências (Física Aplicada – opção em Física Computacional).

Orientador: Prof. Dr. Marcos José Santana

USP/IFSC/SBI
B-2-001331

São Carlos
2000

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

Souza, Paulo Sergio Lopes de
AMIGO: Uma Contribuição para a Convergência na Área de Escalonamento de Processos/ Paulo Sergio Lopes de Souza. São Carlos, 2000.
224p.

Tese (Doutorado) – Instituto de Física de São Carlos, 2000.

Orientador: Prof. Dr. Marcos José Santana

1. Escalonamento de Processos. 2. Computação Paralela.
3. Sistemas Distribuídos. 4. Computação Paralela Distribuída.
I. Título.

UNIVERSIDADE
DE SÃO PAULO
Instituto de Física de São Carlos

Av. Dr. Carlos Botelho, 1465
CEP 13560-250 - São Carlos - SP
Brasil

Fone (016) 273-9333
Fax (016) 272-2218

MEMBROS DA COMISSÃO JULGADORA DA TESE DE DOUTORADO DE PAULO SERGIO LOPES DE SOUZA APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO CARLOS, DA UNIVERSIDADE DE SÃO PAULO, EM 26 DE JUNHO DE 2000.

COMISSÃO JULGADORA:

Prof. Dr. Marcos José Santana/ICMC-SCE-USP

Prof. Dr. Edson dos Santos Moreira/ICMC-SCE-USP

Prof. Dr. Gonzalo Travieso/IFSC-FFI-USP

Prof. Dr. Ricardo de Oliveira Anido/UNICAMP

Prof. Dr. Moacyr Martucci Júnior/EP-USP

e-mail: wladerez@if.sc.usp.br

"Conhecimento sem transformação não é sabedoria."

(Paulo Coelho)

À Simone e ao Felipe.

Agradecimentos

"*Não mostre para Deus o tamanho do seu problema, mostre para o problema o tamanho do seu Deus*" (*autor desconhecido*). Deus, obrigado pelo conforto e proteção.

Ao Marcos Santana e à Regina Santana. Mais que agradecer a orientação precisa, agradeço a confiança e o exemplo de vida.

" *Deixa eu dizer que te amo*
Deixa eu pensar em você
Isso me acalma
me acolhe a alma
Isso me ajuda a viver..." (*Carlinhos Brown/Marisa Monte*)
Amiga, esposa, mãe. Simone, obrigado por seu amor e compreensão.

Ao nosso filho Felipe, por estar conosco e, assim, nos ensinar todos os dias que a vida pode ser simples e pura.

Às nossas famílias pela nossa formação e apoio. Em especial à D. Vera e à Ana Cristina, pela revisão de alguns textos gerados neste trabalho e à D. Leda, pela ajuda em Ponta Grossa nas questões práticas e necessárias da vida.

Ao Márcio e ao Renato, pela amizade e constante troca de idéias.

Ao Tomás, ao Arion e à Vera, pela ajuda na avaliação de desempenho.

À Aletéia, pela amizade e pelas implementações feitas durante o seu mestrado.

Ao Nivaldi, por sua visão crítica e positiva das coisas.

Ao Grupo de Sistemas Distribuídos e Programação Concorrente (Álvaro, Adriana, Ana Elisa, Andrezza, Boca, Célia, Masaki, Daniel Kawabata, Cláudia, Lia, Daniel Lobato, Edmilson, Lucas, Fernanda, Jorge, João Carlos, Kalinka, Luciano, Laís, Mara, Mário, Omar, Roberta, Renata, Ricardo, Sarita, Silmara e Tatiana), pela convivência harmônica e por fazerem do LaSDPC um dos melhores ambientes de trabalho que eu conheço.

Aos amigos do LabES (Lú, Adenilso, Tati, Andréia, Valéria, Auri, Ellen Barbosa, Thelma, Rejane, Janaína, Sandra, Elisa, Rurik e Plínio).

Aos amigos Rudinei, Garga, Paquito, Mirla, Jaque, Marisa e Rosângela.

À D.Maria, pelo carinho com que cuidou do Felipe durante esses anos. Sem ela não teríamos tranqüilidade para trabalhar.

À D.Olívia, um exemplo de solidariedade. Mais que uma vizinha, uma mãe para nós em São Carlos.

À Beth, sempre uma amiga.

Às "moças" da biblioteca do ICMC, ao Paulinho e ao Jacques, pelo suporte no levantamento bibliográfico, nos seminários e nas cópias necessárias.

À Wladerez, pela constante eficiência e simpatia com que realiza suas funções na seção de pós-graduação do IFSC.

À UEPG, em especial ao DeInfo e à PROPESP, pelo apoio aos nossos afastamentos.

À CAPES, pelo apoio financeiro.

Sumário

LISTA DE FIGURAS ..... *IV*

LISTA DE TABELAS ..... *VI*

RESUMO ..... *VII*

ABSTRACT ..... *VIII*

1. INTRODUÇÃO *1*
1.1. Computação Paralela Distribuída & Escalonamento *1*
1.2. O Problema Considerado nesta Tese *7*
1.3. Organização desta Tese *8*

2. ESCALONAMENTO DE PROCESSOS *11*
2.1. Considerações Iniciais *11*
2.2. Plataformas Computacionais Distribuídas e Heterogêneas *12*
2.3. O Escalonamento em Plataformas Computacionais Distribuídas *13*
2.4. Problemas com a Terminologia da Área de Escalonamento de Processos *16*
2.4.1. Divergências na Terminologia *17*
2.4.2. Razões para as Divergências na Terminologia *23*
2.5. Taxonomias da Área de Escalonamento de Processos *28*
2.5.1. Taxonomia de Casavant; Kuhl (1988) *28*
2.5.2. Taxonomia de Wang; Morris (1985) *36*
2.5.3. Taxonomia de Baumgartner; Wah (1991) *37*
2.5.4. Taxonomia de Xu; Lau (1997) *42*
2.5.5. Taxonomia de Feitelson; Rudolph (1995) *47*
2.6. Uma Convergência para as Taxonomias *53*
2.6.1. Taxonomias baseadas nos processos x baseadas na topologia *55*
2.6.2. Taxonomias baseadas nos processos x baseadas nos processadores *57*
2.7. Componentes de um Algoritmo de Escalonamento *60*
2.8. Índice de Carga *63*
2.9. Desempenho Alcançado com o Algoritmo de Escalonamento *66*
2.9.1. Objetivos e Métricas de Desempenho *66*
2.9.2. Fatores que Influenciam o Desempenho *68*
2.10. Escalonamento: a Teoria e a Prática *71*
2.10.1. Fatores que Afastam a Teoria da Prática *71*
2.10.2. Fatores que Convergem a Teoria e a Prática *73*
2.11. Considerações Finais *74*

3. AMIGO – UM AMBIENTE DE ESCALONAMENTO FLEXÍVEL E DINÂMICO. *76*
3.1. Considerações Iniciais *76*
3.2. Plataformas de Atuação do AMIGO *78*
3.3. O perfil dos usuários do AMIGO *79*
3.4. A Estrutura do AMIGO *81*
3.5. A Camada Superior do AMIGO *83*
3.5.1. Classes de Software *84*
3.5.2. Plataformas Computacionais *86*

3.5.3. Políticas de Escalonamento ... *90*
3.5.4. Monitoramento ... *97*
3.5.5. Simulação ... *100*
3.6. A Camada Inferior do AMIGO ... *101*
3.6.1. A Comunicação na Camada Inferior ... *102*
3.6.2. O AMIGOD ... *105*
3.6.3. Os Ambientes de Passagem de Mensagem ... *111*
3.6.4. As Políticas de Escalonamento ... *117*
3.7. Utilização do AMIGO ... *119*
3.8. Situação Atual do Projeto ... *120*
3.9. Considerações Finais ... *122*

4. AVALIAÇÃO DE DESEMPENHO COM O AMIGO* ... *126
4.1. Considerações Iniciais ... *126*
4.2. Avaliações Realizadas ... *128*
4.3. Técnica para Análise Estatística ... *129*
4.4. Análise dos Resultados Obtidos ... *131*
4.4.1. Desempenho com o AMIGO em uma aplicação CPU-Bound ... *132*
4.4.2. Desempenho com o AMIGO em uma aplicação I/O-Bound ... *138*
4.4.3. Desempenho em uma aplicação I/O e CPU-Bound ... *141*
4.5. Considerações Finais ... *144*

5. CONCLUSÕES* ... *146
5.1. Ponderações Finais desta Tese ... *146*
5.2. Contribuições desta Tese ... *149*
5.3. Sugestões para Trabalhos Futuros ... *150*

ANEXO A – ESPECIFICAÇÃO DAS MENSAGENS DA CAMADA INFERIOR DO AMIGO* ... *153
A.1. Mensagens entre o AMIGOD e o Ambiente de Passagem de Mensagem ... *153*
A.1.1. Mudanças no número de processadores ... *153*
A.1.2. Escalonamento inicial de processos ... *154*
A.1.3. Migração de Processos ... *154*
A.2. Mensagens entre os AMIGODs ... *155*
A.2.1. Mudanças no número de processadores ... *155*
A.2.2. Atualização da política de informação ... *156*
A.2.3. Mudança da política de escalonamento ... *156*
A.2.4. Política de Localização ... *156*
A.2.5. Migração de Processos ... *157*
A.3. Mensagens entre o AMIGOD e as Políticas ... *157*
A.3.1. Política de informação ... *157*
A.3.2. Política de localização ... *158*
A.3.3. Pedido de escalonamento inicial ... *159*
A.3.4. Inclusão de uma política ... *159*
A.3.5. Remoção de uma política ativa ... *159*
A.3.6. Remoção de um processador remoto ... *159*
A.3.7. Remoção de uma política remota ... *160*
A.3.8. Migração de Processos ... *160*

ANEXO B – ESPECIFICAÇÃO DAS ROTINAS DA CAMADA INFERIOR DO AMIGO *163*
B.1. Rotinas do AMIGOD *163*
B.1.1. Iniciação e Roteamento de Mensagens do AMIGOD *163*
B.1.2. Protocolo com o ambiente de passagem de mensagem local *165*
B.1.3. Protocolo com os outros AMIGODs. *167*
B.1.4. Protocolo com as políticas ativas. *174*
B.1.5. Interação com a camada superior do AMIGO *181*
B.2. Mudanças no feitas no PVM para a adaptação com o AMIGO *182*
B.2.1. Configuração inicial *182*
B.2.2. Inclusão de mais um processador *183*
B.2.3. Exclusão de um processador *183*
B.2.4. Escalonamento inicial de processos *184*
B.2.5. Migração de processos *185*

REFERÊNCIAS BIBLIOGRÁFICAS *189*

APÊNDICE A – RELAÇÃO DE NOVAS PROPOSTAS DE ALGORITMOS DE ESCALONAMENTO *1*

APÊNDICE B – ESTRUTURA DE DIRETÓRIOS DO AMIGO *5*

APÊNDICE C – CÓDIGOS FONTE DAS APLICAÇÕES USADAS NAS AVALIAÇÕES DE DESEMPENHO *7*

APÊNDICE D – PUBLICAÇÕES REALIZADAS COM BASE NOS TRABALHOS DESCRITOS NESTA TESE *19*

Lista de Figuras

Figura 2.1 - O escalonamento segundo Casavant; Kuhl (1988)..... *15*
Figura 2.2 - O escalonamento segundo Baumgartner; Wah (1991)..... *15*
Figura 2.3 – Taxonomia hierárquica de Casavant; Kuhl (1988)..... *29*
Figura 2.4 – Um sistema distribuído local segundo Wang; Morris (1985)..... *37*
Figura 2.5 – Relação hierárquica das políticas de escalonamento proposta por Baumgartner; Wah (1991)..... *39*
Figura 2.6 – Taxonomia proposta por Xu; Lau (1997)..... *45*
Figura 2.7 – Exemplo de um modelo iterativo gradiente com dezesseis processadores [Lüling et al. (1993)]..... *46*
Figura 2.8 – Extensão da taxonomia de Casavant; Kuhl (1988) por Muniz (1994)..... *57*
Figura 3.1 – Esquema estrutural do AMIGO..... *82*
Figura 3.2 – Exemplo de arquivo com o resultado da execução de um benchmark..... *89*
Figura 3.3 – Estrutura do arquivo de configuração policy.cfg..... *92*
Figura 3.4 – Comunicação na Camada Inferior..... *103*
Figura 3.5 – Cabeçalho das Mensagens Trocadas na Camada Inferior..... *103*
Figura 3.6 – Estrutura amigod_table existente no AMIGOD..... *106*
Figura 3.7 – Estrutura policy_table existente no AMIGOD..... *107*
Figura 3.8 – Esquema gráfico do escalonamento original no PVM..... *113*
Figura 3.9 - Esquema gráfico do escalonamento no PVM com o AMIGO..... *115*
Gráfico 4.1- Tempo médio com a execução do Método dos Trapézios Compostos em uma plataforma ociosa, com 10 processos (1 mestre e 9 escravos) e 6 processadores..... *132*
Gráfico 4.2 – Tempo médio com a variação da carga na plataforma durante a execução da aplicação Método dos Trapézios Compostos com 12.000.000 de divisões, com 10 processos (1 mestre e 9 escravos) e 6 processadores..... *135*
Gráfico 4.3 – Gráfico com sete variações de escalonamento para o Método dos Trapézios Compostos, determinadas de maneira estática, utilizando 10 processos (1 mestre e 9 escravos), 12.000.000 de divisões e plataforma ociosa. Os valores para "PVM – 6 processadores" e "AMIGO-DPWP – 6 processadores" representam o tempo para plataforma ociosa e com 12.000.000 de divisões, respectivamente 21,405s e 11,573s (vide Tabela 4.3)..... *138*
Gráfico 4.4 - Gráfico com a execução da aplicação Quicksort Paralelo em um vetor de 306.000 elementos inteiros, variando-se da carga da plataforma computacional, utilizando 10 processos (1 mestre e 9 escravos) e 6 processadores..... *139*
Gráfico 4.5 – Gráfico com as sete variações de escalonamento para o Quicksort Paralelo, determinadas de maneira estática, utilizando 10 processos (1 mestre e 9 escravos), vetor com 306.000 elementos inteiros e plataforma ociosa. Os valores para "PVM – 6 processadores" e "AMIGO-DPWP – 6 processadores" representam a média para plataforma ociosa, respectivamente 4,249s e 4,534s (vide Tabela 4.6)..... *141*

Gráfico 4.6 – Gráfico de uma aplicação I/O e CPU-Bound com a relação do tempo de comunicação x tempo de processamento para as execuções no PVM sem o AMIGO-DPWP, com 10 processos (1 mestre e 9 escravos) e 6 processadores. *143*
Gráfico 4.7 – Gráfico com o tempo de comunicação x tempo de processamento de uma aplicação I/O e CPU-Bound para as execuções com o AMIGO-DPWP, com 10 processos (1 mestre e 9 escravos) e 6 processadores. *143*
Gráfico 4.8 - Gráfico com o tempo total de execução de uma aplicação I/O e CPU-Bound para as execuções com o PVM e com o AMIGO-DPWP, com 10 processos (1 mestre e 9 escravos) e 6 processadores. *144*
Figura B.1 – Estrutura de diretórios usada pelo AMIGO. *5*

Lista de Tabelas

Tabela 2.1 – Revisão das taxonomias de escalonamento [Baumgartner; Wah (1991)]. . 38
Tabela 2.2 – Taxonomia ESR proposta por Baumgartner; Wah (1991). 41
Tabela 2.3 – Taxonomia das políticas space sharing proposta por Feitelson;Rudolph (1995). 49
Tabela 4.1 – Hardware utilizado para a avaliação de desempenho. 127
Tabela 4.2 – Execução da aplicação Método dos Trapézios Compostos em uma plataforma ociosa, com 10 processos (1 mestre e 9 escravos) e 6 processadores. 132
Tabela 4.3 – Variação da carga da plataforma durante a execução da aplicação Método dos Trapézios Compostos com 12.000.000 de divisões, com 10 processos (1 mestre e 9 escravos) e 6 processadores. 134
Tabela 4.4 – Sete variações de escalonamento realizadas nas aplicações para comparar a qualidade do escalonamento proposto pelo AMIGO-DPWP, usando 10 processos (1 mestre e 9 escravos) e um número variado de processadores (de 1 a 6 processadores). 136
Tabela 4.5 – Sete variações de escalonamento para a aplicação Método dos Trapézios Compostos, determinadas na aplicação de maneira estática, utilizando 10 processos (1 mestre e 9 escravos), 12.000.000 de divisões e processadores sem carga extra (plataforma ociosa). Os ganhos percentuais em relação ao PVM e em relação ao AMIGO-DPWP foram calculados com base no tempo médio para plataforma ociosa e com 12.000.000 de divisões, respectivamente 21,405s e 11,573s (vide Tabela 4.3). 137
Tabela 4.6 - Execução da aplicação Quicksort Paralelo em um vetor de 306.000 elementos inteiros, variando-se da carga da plataforma computacional, utilizando 10 processos (1 mestre e 9 escravos) e 6 processadores. 139
Tabela 4.7 – Sete variações de escalonamento para o Quicksort Paralelo, determinadas de maneira estática, utilizando 10 processos (1 mestre e 9 escravos), vetor com 306.000 elementos inteiros e plataforma ociosa. Os ganhos percentuais em relação ao PVM e em relação ao AMIGO-DPWP foram calculados com base na média para plataforma ociosa, respectivamente 4,249s e 4,534s (vide Tabela 4.6). 141
Tabela 4.8 –Resultados obtidos com a execução de uma aplicação com ambas as características I/O-Bound e CPU-Bound. As execuções tem como base os tempos obtidos para o PVM e assim são divididas em grupos desde 90% comunicação e 10% processamento até grupos com 10% comunicação e 90% processamento. 142

Resumo

Este trabalho propõe e descreve em detalhes o projeto do AMIGO (DynAMical FlexIble SchedulinG EnvirOnment), uma nova ferramenta de software capaz de viabilizar a união de diferentes algoritmos de escalonamento, de uma maneira completamente transparente ao usuário. O AMIGO é capaz de flexibilizar o escalonamento (em tempo de execução da aplicação) desde a sua configuração até a sua efetiva aplicação. Além da flexibilidade dinâmica e da transparência, o AMIGO também é modular: o seu projeto está dividido em módulos que, entre outras vantagens, facilitam sua execução em diferentes plataformas. Este trabalho também contribui apresentando uma análise crítica da literatura da área, apontando divergências e propondo pontos de convergência importantes. Assim, o levantamento bibliográfico apresentado atua como um material introdutório precioso para que os pesquisadores iniciantes formem um contexto geral sobre a área e, desse modo, aprofundem mais rapidamente seus estudos em outros trabalhos mais específicos. A avaliação de desempenho feita com o AMIGO demonstra que é possível a obtenção de ganhos de desempenho expressivos, com total transparência para o usuário final. Unindo-se desempenho, flexibilidade e transparência, espera-se contribuir para a redução da lacuna existente entre teoria e prática na área de escalonamento de processos.

Abstract

This thesis proposes and describes in details the design of the AMIGO (DynAMical FlexIble SchedulinG EnvirOnment), a novel software tool that makes possible the union of different algorithm scheduling proposals, in a way completely transparent to the user. The AMIGO is able to make flexible the scheduling activity (at run-time), covering all the steps from its configuration up to its effective application. Besides the dynamic flexibility and transparency, the AMIGO is also modular: it is split into modules that, among other advantages, facilitate its execution on different platforms. This work also contributes by presenting a critical analysis of the process-scheduling literature, pointing out the existing divergences and proposing important convergence points. Thus, the literature survey presented acts as a precious introductory material, which is able, on one hand, to give to the beginners a broad view of the process-scheduling area and, on the other hand, to facilitate the development of deeper studies in a quicker fashion when more specific works are needed. The performance evaluation of the AMIGO shows that is possible to have expressive performance gains, while having total user transparency. Joining flexibility and transparency it is hoped to contribute for the reduction of the existing gap between theory and practice in the scheduling process area.

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

1. Introdução

1.1. Computação Paralela Distribuída & Escalonamento

Várias mudanças ocorreram na computação nas últimas duas décadas (80 e 90). Dentre elas, a alta conectividade[1] dos recursos computacionais merece destaque, pois através da conexão desses recursos, diferentes e maiores problemas puderam ser resolvidos de uma maneira mais eficiente, elegante e/ou a um custo relativamente baixo.

A motivação para conectar diferentes recursos não é única. Ela pode ser desde o simples compartilhamento de discos e impressoras até a união de uma grande quantidade de processadores para viabilizar um aumento na potência computacional disponível. Outros motivos que levam a essa conexão podem ser: tolerância a falhas, mobilidade dos usuários e compartilhamento de determinados dispositivos (como processadores específicos para algum problema em particular). De um modo geral, quando o motivo é principalmente o aumento da potência computacional, diz-se que tais recursos são utilizados para a computação paralela e, por outro lado, quando o motivo para a conexão é o compartilhamento, diz-se que os recursos pertencem a um sistema distribuído.

---

[1] "Conectividade", aqui, refere-se à união entre recursos computacionais, como por exemplo, a união entre diferentes computadores ou diferentes processadores. Não se refere, por exemplo, à união entre um processador e a memória principal em uma arquitetura de von Neumann.

A computação paralela originou-se com o objetivo de proporcionar melhores desempenhos para aplicações específicas. Através de um grupo de processadores conectados que cooperam e comunicam-se entre si, grandes problemas são resolvidos em um tempo menor do que se estivessem sendo solucionados por computadores seqüenciais (arquiteturas de von Neumann) [Zaluska (1991), Almasi; Gottlieb (1994)].

Com o avanço da computação paralela, foram propostas diferentes maneiras de se conectar os recursos computacionais, criando-se diferentes arquiteturas paralelas. Cada arquitetura apresenta determinadas caraterísticas, porém, sempre com o intuito de buscar melhores desempenhos. Para acompanhar o desenvolvimento das arquiteturas paralelas e agrupar equipamentos com características comuns, foram propostas algumas taxonomias, dentre elas a de Flynn (1972).

A taxonomia de Flynn (1972), embora antiga, é amplamente adotada e baseia-se no fluxo de instruções e no fluxo de dados, criando quatro categorias: SISD (*Single Instruction, Single Data Stream*), SIMD (*Single Instruction, Multiple Data Stream*), MISD (*Multiple Instruction, Single Data Stream*) e MIMD (*Multiple Instruction, Multiple Data Stream*).

O hardware disponível para a utilização da computação paralela é fundamental para o desempenho obtido com uma aplicação concorrente, além de determinar como uma aplicação deve e/ou pode ser construída e executada. Além de considerar o hardware, a computação paralela também necessita da programação concorrente para viabilizar o desenvolvimento de aplicações que serão executadas sobre os vários processadores.

A programação concorrente visa a otimizar o desempenho de aplicações seqüenciais, explorando a concorrência em diferentes arquiteturas, permitindo assim uma melhor utilização dos recursos computacionais disponíveis.

Para que isso seja possível, a programação concorrente parte do princípio que determinadas aplicações podem ser divididas em partes menores e que cada parcela é responsável por uma porção do problema. Para essa divisão, há a necessidade de alguns recursos de software adicionais, não disponíveis na programação seqüencial e que viabilizam a iniciação/paralisação de processos concorrentes e coordenação (ou sincronização) dos mesmos [Almasi; Gottlieb (1994)].

Os recursos de software capazes de iniciar/paralisar e coordenar os processos concorrentes podem ser implementados por exemplo em: (1) linguagens de programação concorrente, como ADA, OCCAM ou adaptações em linguagens seqüenciais como Fortran e C, e (2) ambientes de passagem de mensagem.

Os ambientes de passagem de mensagem representam uma alternativa eficiente ao uso das linguagens de programação concorrente. Estes são compostos basicamente por uma biblioteca de comunicação que, atuando como uma extensão das linguagens seqüenciais (como C e Fortran), permitem a elaboração de aplicações paralelas. Exemplos de ambientes de passagem de mensagem são o PVM (*Parallel Virtual Machine*) e as implementações da especificação MPI (*Message Passing Interface*) como a implementação MPI-LAM (*MPI-Local Area Multicomputer*) [Geist et al. (1994), Dongarra et al. (1996), Snir et al. (1996)]. Ambientes de passagem de mensagem como o PVM e o MPI têm sido muito utilizados na computação paralela porque, além de viabilizarem a programação concorrente,

também fornecem portabilidade às aplicações desenvolvidas, permitem o uso de plataformas[2] heterogêneas, são softwares abertos e de domínio público.

Outra categoria de recursos computacionais conectados são os sistemas distribuídos. Os sistemas distribuídos são formados por computadores autônomos conectados por uma rede de comunicação e controlados por um sistema operacional distribuído. É esse sistema operacional distribuído o responsável por coordenar as atividades realizadas e por compartilhar os recursos computacionais disponíveis (hardware, software e dados). O sistema operacional distribuído fornece transparência (no sentido de encapsular os diferentes recursos), permitindo que os usuários tenham a impressão de estarem utilizando um sistema computacional único e integrado, embora esse sistema esteja implementado em vários computadores e em diferentes locais [Coulouris et al. (1994)].

Tanenbaum (1996) descreve um sistema distribuído como um caso especial de redes de computadores, pois permite o acesso a um conjunto de computadores autônomos e conectados, porém, de modo transparente (para o usuário não é visível a existência de múltiplos recursos). Para Tanenbaum (1996) é o sistema operacional distribuído o elemento responsável por manter as características de um sistema distribuído, utilizando a rede como meio de comunicação.

Mullender (1993) define um sistema distribuído como um grupo de computadores trabalhando juntos em alguma tarefa, destacando três características básicas: múltiplos computadores, conexões dos computadores e controle da concorrência. Mullender (1993) destaca a potência computacional inserida nos

---

[2] "Plataforma" está sendo usada como um termo genérico para se referir à união do hardware, software básico, topologia e demais características que compõe o ambiente computacional utilizado.

sistemas distribuídos, citando a tolerância a falhas e o paralelismo como aspectos importantes (além do compartilhamento).

A literatura para sistemas distribuídos apresenta diferentes características que um sistema deve ter para ser considerado um sistema distribuído. Embora essas caraterísticas variem de autor para autor, a transparência destaca-se como um conceito comum a todos os trabalhos. Dentre as possíveis características mencionadas estão: compartilhamento de recursos, ser composto por recursos "abertos" (*openness*), concorrência, capacidade escalar (*scalability*), tolerância a falhas e transparência [Coulouris et al. (1994)].

Embora com motivações diferentes e com algumas características incomuns, as duas áreas (computação paralela e sistemas distribuídos) vêm apresentando uma forte convergência nos últimos anos. Essa convergência deve-se, principalmente, ao avanço tecnológico e às linhas de pesquisa ocorridos a partir da década de 80. A computação paralela que empregava maciçamente arquiteturas SIMD, também começou a utilizar máquinas MIMD com memória distribuída, devido à versatilidade e alto desempenho dessas arquiteturas. Essas máquinas MIMD passaram a utilizar processadores de propósito geral, permitindo a sua abstração como um conjunto de computadores autônomos (com unidade de controle, unidade de processamento e memória), conectados por uma rede de comunicação.

A computação paralela sobre os sistemas distribuídos (ou computação paralela distribuída) vem sendo muito utilizada, principalmente por apresentar uma relação custo x benefício vantajosa, quando comparada ao uso de máquinas verdadeiramente paralelas. O uso de computadores de propósito geral, conectados por uma rede de comunicação (também de uso geral) e que normalmente já encontram-se instalados, reduz significativamente os custos para a execução de aplicações paralelas.

Além de uma relação custo x benefício atrativa, a popularização da computação paralela distribuída deve-se, em parte, à grande aceitação dos ambientes de passagem de mensagem, como o PVM e o MPI. Com o uso de tais ambientes em plataformas computacionais distribuídas, as aplicações paralelas puderam utilizar plataformas heterogêneas, além de se tornarem portáteis, pois podem ser executadas em outros lugares que não aqueles previstos inicialmente. Para que isso seja feito, basta que o ambiente de passagem de mensagem esteja implementado para a plataforma que se deseja. Outra questão para a grande aceitação dos ambientes de passagem de mensagem é a sua facilidade de uso e baixo custo. O desenvolvimento de aplicações com os ambientes de passagem de mensagem normalmente é feito com ferramentas de software (como editores de texto e compiladores) já empregadas para aplicações seqüenciais. Um usuário que esteja acostumado a usar tais ferramentas encontra menos dificuldades para se adaptar à nova situação. O baixo custo refere-se ao fato dos ambientes de passagem de mensagem serem de domínio público e também abertos. Cada implementação do ambiente de passagem de mensagem pode ser aperfeiçoada para uma determinada plataforma, sem alterar a funcionalidade para a aplicação desenvolvida.

Várias linhas de pesquisas vêm sendo desenvolvidas considerando as plataformas distribuídas, independentemente se elas são utilizadas para computação paralela, sistemas distribuídos ou ambos. Alguns exemplos destas são: protocolos de comunicação, redes de conexão, portabilidade das aplicações desenvolvidas, tolerância a falhas e escalonamento de processos. Dessas linhas de pesquisa, o escalonamento de processos é um dos fatores decisivos para melhorar o desempenho esperado com a utilização da plataforma computacional distribuída, sendo o tema principal abordado nesta tese.

A literatura sobre escalonamento de processos é abundante. Facilmente são encontrados centenas de artigos contendo informações que contribuem para o aprimoramento da área. No entanto, ao estudar esses artigos percebe-se que eles seguem caminhos distintos, tornando difícil a convergência e o relacionamento do conhecimento adquirido.

Embora a qualidade e a importância dos trabalhos sejam indiscutíveis, a literatura não fornece uma visão abrangente da área, a ponto de permitir a comparação apropriada das propostas e, assim, permitir a escolha correta da política de escalonamento a ser empregada.

Os problemas verificados na literatura e a complexidade para se considerar todos os vários fatores inseridos no escalonamento de processos (e que influenciam o desempenho final), dificultam a utilização prática do conhecimento gerado pelos trabalhos teóricos da área.

1.2. O Problema Considerado nesta Tese

Esta tese aborda o problema de se escalonar processos em plataformas computacionais distribuídas. O escalonamento de processos é discutido como atividade principal e não apenas como parte integrante de um sistema operacional distribuído ou ferramenta de escalonamento exclusiva à computação paralela. A meta é analisar o escalonamento de maneira independente da plataforma, dos seus objetivos e das suas implementações. Dessa forma, a literatura da área é apresentada nesta tese de uma forma abrangente e não voltada apenas para uma situação específica. São descritas as principais divergências encontradas na literatura, as quais contribuem para a falta de um contexto geral, dificultando a comparação entre diferentes propostas e, por conseqüência, a escolha apropriada da política de

escalonamento a ser empregada. Além de descrever essas divergências, esta tese cita os motivos que levam a elas e propõe pontos de convergência.

Um novo ambiente de escalonamento flexível e dinâmico (AMIGO – *DynAMical FlexIble SchedulinG EnvirOnment*) é proposto com a intenção de viabilizar a união dos diversos fatores que influenciam a escolha da melhor política de escalonamento a ser empregada. O AMIGO encapsula para o usuário final as divergências verificadas na literatura, tornando transparente para esse usuário a escolha e utilização da política de escalonamento mais apropriada.

O AMIGO não implementa uma única política de escalonamento. Através deste, é possível implementar diferentes políticas, cada uma apropriada para determinada situação, retirando assim a visão algorítmica existente nas propostas de escalonamento desenvolvidas. Espera-se com a utilização do AMIGO, que haja uma maior convergência entre a teoria e a prática na área, por incluir no escalonamento: (1) flexibilidade para determinar a política a ser empregada, (2) a característica dinâmica para essa flexibilidade, pois esta é feita em tempo de execução da aplicação e (3) transparência total para o usuário.

1.3. Organização desta Tese

Este texto está organizado em cinco capítulos, dois anexos e quatro apêndices. Neste primeiro capítulo é feita uma introdução ao assunto tratado nesta tese, fornecendo um contexto geral para o escalonamento de processos dentro de plataformas computacionais distribuídas.

No Capítulo 2 é discutida a literatura da área de escalonamento de processos. Nesse segundo capítulo não houve a intenção de apenas descrever o que já existe, mas fornecer uma visão crítica e abrangente sobre o assunto, destacando

divergências e como convergir para um entendimento apropriado. Para fornecer uma visão ampla e convergente, são abordados alguns assuntos principais que compõem o conhecimento da área. Dentre esses assuntos, os problemas com a terminologia e as taxonomias são discutidos com maior ênfase, por serem representativos aos objetivos propostos nesta tese.

As divergências encontradas na terminologia são apresentadas, seguidas dos motivos que levam a essas divergências. A intenção é demonstrar através da literatura disponível, termos usados de maneira indiscriminada como sinônimos ou então atividades idênticas denominadas com termos distintos. Algumas das principais taxonomias da área são apresentadas, salientando-se suas vantagens/desvantagens e também suas divergências. Apresentadas as taxonomias, é descrita uma proposta de convergência, não com o objetivo de criar uma nova taxonomia, mas sim fornecer uma relação entre elas que possibilite comparar com eficiência dois algoritmos de escalonamento classificados sob duas taxonomias divergentes.

O AMIGO é descrito no Capítulo 3, sendo destacados seus objetivos e suas principais características. São apresentadas as plataformas onde essa ferramenta de software é utilizada, bem como o perfil necessário aos seus usuários. A estrutura do AMIGO é discutida, pormenorizando-se as suas camadas superior e inferior (maior divisão existente no AMIGO). Depois de detalhadas as principais questões do projeto da ferramenta, é apresentado como o AMIGO é iniciado pelos seus usuários e também é descrita a situação atual do projeto.

No Capítulo 4 é apresentada a avaliação de desempenho feita em aplicações que utilizaram o AMIGO como ferramenta de apoio ao escalonamento. Foram realizadas avaliações em aplicações pertencentes a diferentes classes, como *CPU-Bound* e

I/O-Bound, com o intuito de demonstrar o comportamento dessas aplicações com e sem a utilização do AMIGO, bem como demonstrar o impacto de uma política em particular sobre diferentes classes de aplicações.

No Capítulo 5 são apresentadas as conclusões desta tese, destacando-se as suas contribuições e algumas sugestões para trabalhos futuros.

Nos Anexos desta tese estão descritas as especificações das mensagens e das rotinas utilizadas pelo AMIGO. Os Apêndices contêm uma relação de trabalhos que propõem novos algoritmos de escalonamentos, a estrutura de diretórios utilizada pelo AMIGO, os códigos fonte das aplicações utilizadas na avaliação de desempenho feita com o AMIGO e uma relação com as principais publicações produzidas em função do desenvolvimento desta tese.

2.Escalonamento de Processos

2.1. Considerações Iniciais

O escalonamento é estudado e praticado em áreas do conhecimento onde se deseja atribuir ou distribuir serviços para recursos, os quais irão realizar algum processamento sobre esses serviços. Bons exemplos dessas áreas são a engenharia de produção e a computação.

A definição da atividade de escalonar é dada no dicionário como sinônimo de dispor, pôr, colocar ou distribuir ordenadamente [Ferreira (1986)]. Essa definição aplica-se perfeitamente à computação, onde para executar um serviço (como por exemplo um processo) há a necessidade de colocá-lo para execução em um recurso de processamento (ou processador[1]) de maneira apropriada.

Para se iniciar uma discussão sobre escalonamento de processos é importante destacar a diferença existente entre o escalonamento feito em um único processador e o escalonamento feito sobre vários processadores.

No escalonamento feito em um único processador disponível, também chamado de escalonamento local, a atribuição considera características como: mais de um usuário utilizando o mesmo processador (multiusuário), processos concorrentes

---

[1] Neste trabalho o termo "processador" é utilizado como sinônimo de elemento de processamento, recurso de processamento, nó, *host* e computador. O termo computador, quando qualificado como "computador paralelo" designa um equipamento com vários processadores, agrupados com o objetivo de executar aplicações paralelas. Um nó, segundo alguns autores, também pode conter vários processadores.

(multitarefa), tipos de aplicações e o algoritmo de escalonamento implementado pelo sistema operacional. Este trabalho não aborda o escalonamento local, o qual encontra-se descrito em Tanenbaum (1992).

No escalonamento sobre os vários processadores de uma plataforma computacional distribuída, também denominado de escalonamento global, devem ser consideradas, além das características do escalonamento local, qual(is) o(s) melhor(es) processador(es) para se executar o(s) processo(s), considerando os objetivos e a plataforma como um todo. Este capítulo aborda o escalonamento feito sobre vários processadores, com base no trabalho publicado por Souza et al. (2000).

2.2. Plataformas Computacionais Distribuídas e Heterogêneas

Sendo este trabalho direcionado ao escalonamento realizado sobre os vários processadores de uma plataforma computacional distribuída, cabe aqui definir o que significa tal "plataforma" dentro desse contexto.

Uma plataforma computacional distribuída é uma coleção de elementos de processamento, fisicamente distribuídos, conectados por uma rede de comunicação e utilizada com objetivos específicos. Essa definição apesar de assemelhar-se à definição de redes de computadores é mais abrangente, porque os elementos de processamento podem pertencer a uma *NOW* (*Network of Workstations*) composta por computadores pessoais e/ou estações de trabalho [Anderson et al. (1995)], ou até mesmo a computadores paralelos, todos podendo estar interligados e formando uma única plataforma computacional. Considera-se que tais plataformas são potencialmente (porém não necessariamente) heterogêneas, pois são compostas freqüentemente por recursos que diferem na sua arquitetura e/ou desempenho.

As plataformas podem ser fracamente acopladas (*loosely-coupled*) ou então fortemente acopladas (*tightly-coupled*). Nas plataformas fortemente acopladas, o envio de uma mensagem entre dois processadores apresenta um atraso (*delay)* pequeno e uma taxa de transferência alta (número de *bits* enviados), quando comparadas às plataformas fracamente acopladas [Tanenbaum (1995)].

Caso a plataforma distribuída e heterogênea possua na gerência das suas atividades um sistema operacional distribuído, tem-se um "sistema distribuído", caso contrário tem-se uma plataforma computacional distribuída fisicamente e com possíveis diferentes arquiteturas e/ou capacidades de processamento. Essa plataforma pode ter diferentes objetivos que vão desde o simples compartilhamento de recursos (como unidades de disco e impressoras) até a execução de aplicações paralelas através da união lógica (ou virtual) dos processadores, por intermédio de ferramentas de software como o PVM e o MPI [Geist et al. (1994), Snir et al. (1996)].

As plataformas distribuídas e heterogêneas possuem características interessantes que contribuem enfaticamente para a sua grande utilização e, algumas vezes, particularizam a atividade de escalonamento. A melhor relação custo x benefício é um dos fatores responsáveis pela grande utilização, principalmente quando a plataforma é composta basicamente por *NOWs*. As características multiusuária e heterogênea das *NOWs* são bons exemplos de fatores que particularizam as políticas de escalonamento.

2.3. O Escalonamento em Plataformas Computacionais Distribuídas

Vários autores definem a atividade de escalonamento de processos para plataformas computacionais distribuídas.

Tanenbaum (1995) refere-se ao escalonamento global como "Alocação de Processadores", definindo essa atividade como sendo o estudo dos "algoritmos usados para determinar quais processos serão atribuídos para quais processadores".

Mullender (1993) define o escalonamento global como sendo um serviço de um sistema distribuído que atribui processos para processadores. Esse serviço é denominado de serviço de gerenciamento de processos, enfatizando assim, os processos e não os processadores.

Saphir et al. (1995) definem escalonamento global como sendo "a atribuição de recursos específicos para uma aplicação paralela". Essa definição é voltada aos processadores e não aos processos, fato encontrado principalmente nos trabalhos pertencentes à computação paralela.

Shirazi; Hurson (1992) definem a área de escalonamento global como o estudo da distribuição dos processos entre os elementos de processamento, para atingir alguma meta de desempenho, tais como: maximizar o *throughput* ou encontrar o menor *turnaround time*. Essa definição, voltada à aplicação, cita uma importante característica do escalonamento: os objetivos a que eles se propõem. Esses objetivos inseridos no algoritmo de escalonamento, determinam a sua "personalidade" e restringem a sua área de atuação.

Casavant; Kuhl (1988) definem o escalonamento global, como sendo um recurso para o gerenciamento de recursos. Esse gerenciamento é composto basicamente por uma política usada para permitir o acesso e o uso de um recurso por seus vários consumidores, de modo eficiente (Figura 2.1). Os recursos são processadores, memórias, rede de comunicação, entre outros; e os consumidores são os usuários representados por suas aplicações seqüenciais distribuídas ou paralelas.

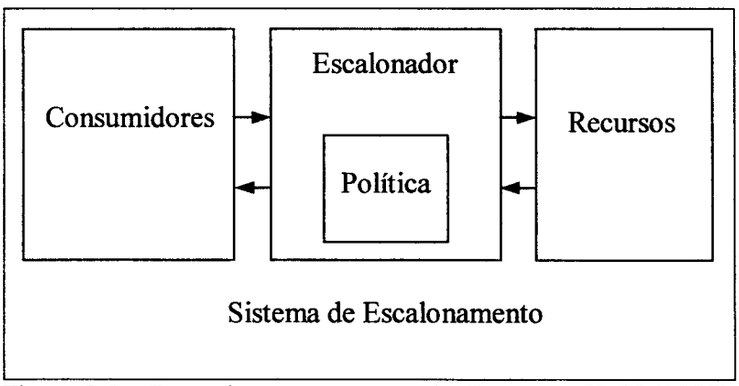


Figura 2.1 - O escalonamento segundo Casavant; Kuhl (1988).

Baumgartner; Wah (1991) definem o escalonamento como sendo o mapeamento de um conjunto de processos para um conjunto de processadores, sob o ponto de vista ilustrado pela Figura 2.2.

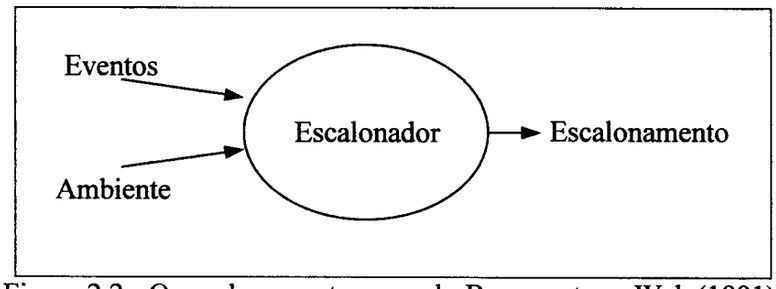


Figura 2.2 - O escalonamento segundo Baumgartner; Wah (1991).

Os eventos e o ambiente são as entradas para o escalonador e o escalonamento (ou mapeamento) é a saída do escalonador. Os eventos representam a menor entidade de escalonamento, como por exemplo processos. O ambiente engloba todas as demais características que podem afetar a escolha do escalonamento. Dentre essas características estão: número de processadores, características físicas dos processadores, meio de comunicação, e os objetivos de desempenho. Os objetivos podem ser, por exemplo, o prazo máximo para a conclusão de uma aplicação de tempo real ou então determinar se é possível obter alguma melhoria de desempenho com o escalonamento.

2.4. Problemas com a Terminologia da Área de Escalonamento de Processos

Para estabelecer uma taxonomia para a área de escalonamento de processos é necessário determinar por quais características os trabalhos devem ser organizados, assim como utilizar uma terminologia coesa que auxilie nessa organização. Entretanto, não há um conjunto de características que possam ser rotuladas como "padrão" para a comparação dos algoritmos de escalonamento propostos, nem tão pouco uma nomenclatura coesa que facilite essa comparação.

O desenvolvimento dos trabalhos na área é caracterizado por uma terminologia inconsistente, com diferentes formulações para o problema do escalonamento [Casavant; Kuhl (1988)]. Feitelson; Rudolph (1996) apontam a dificuldade de relacionar os vários trabalhos já desenvolvidos, citando que "em alguns casos tais comparações estão equiparando maçãs com laranjas, como se elas fossem a mesma coisa". Com isso, alguns desses trabalhos são desconsiderados, por serem vistos erroneamente como irrelevantes e, em outros casos, é simplesmente difícil verificar se a comparação faz sentido ou não.

As datas das publicações citadas acima (1988 e 1996) demonstram que a falta de uma terminologia consistente persiste. Analisando-se a literatura disponível (inclusive as posteriores a 1996), são encontrados artigos que criticam a terminologia confusa da área. No entanto, não foi encontrado um trabalho que detalhasse tais problemas através de exemplos concretos, nem tão pouco que apontasse os possíveis motivos que levam a essa discordância.

A presente seção tem por objetivo suprir essa deficiência, relacionando trabalhos que exemplificam as divergências citadas e que estabelecem os possíveis motivos que as originaram.

2.4.1. Divergências na Terminologia

Dentre os vários exemplos de divergências na terminologia da área, optou-se por escolher alguns grupos de termos que são abrangentes e significativos. Os grupos de termos escolhidos foram os seguintes: (1) aplicações, *jobs*, *threads*, tarefas, programas e processos; (2) políticas, estratégias, mecanismos e algoritmos de escalonamento; e (3) escalonamento de processos, balanceamento de cargas, alocação de recursos e outros afins.

Aplicações, *jobs*, *threads*, tarefas, programas e processos. O escalonamento pode ser usado tanto por aplicações seqüenciais quanto por concorrentes, sendo que o uso indiscriminado desses termos contribui para a falta de entendimento das publicações disponíveis. O problema está em não distinguir se, por exemplo, um *job* corresponde a um processo ou a um conjunto de processos, se uma tarefa tem o mesmo significado de um processo ou tem o significado de um *thread*.

Devarakonda; Iyer (1989), Mullender (1993) e Tanenbaum (1995) usam o termo processo como sendo uma abstração de um programa em execução, composta por um código fonte, uma entrada, uma saída e um estado (dados do programa e pilha, contador de programa e todas as demais informações necessárias para executar o programa). Para Tanenbaum (1995) *threads* ou *lightweight processes* são diferentes linhas de execução (concorrentes) que um mesmo processo pode ter, compartilhando o mesmo espaço de endereçamento de memória. Cada *thread* executa seqüencialmente e de maneira análoga a um processo normal, no entanto compartilha memória com outros *threads*. Ainda para Tanenbaum (1995), um *job* é um programa ou um conjunto de programas.

Feitelson; Rudolph (1995), Vesseur et al. (1995), Ferstl (1996), Islam et al. (1996), Russ et al. (1996) e Stellner; Trinitis (1997) definem *job* como sendo a maior

"entidade" de execução (um programa ou aplicação) constituída de uma ou mais tarefas (processos ou *threads*). Chen; King (1996) e Harchol-Balter et al. (1997) também usam o termo tarefa como sinônimo para processo.

Saphir et al. (1995) citam que o termo aplicação paralela corresponde a "um programa MIMD – *Multiple-Instructions-Multiple-Data*, o qual é projetado para execução simultânea em vários processadores com memória distribuída". Para Saphir et al. (1995) "um *job* paralelo é uma seqüência de operações requisitadas por um usuário, onde uma aplicação paralela é a parte principal". A definição de uma aplicação paralela como "um programa MIMD" não é adequada. A sigla MIMD foi estabelecida por Flynn (1972) para classificar arquiteturas paralelas e não programas concorrentes.

Neste trabalho os termos processo e *thread* têm o mesmo significado apresentado por Tanenbaum (1995). O termo tarefa é empregado como sinônimo de um processo. Os termos aplicação e *job* são reservados para designar a maior "entidade" de execução. Dessa forma, uma aplicação pode conter vários processos que cooperam entre si para resolver um determinado problema (no caso uma aplicação concorrente) ou apenas um processo (no caso uma aplicação seqüencial).

Políticas, estratégias, mecanismos e algoritmos de escalonamento. Embora alguns autores como Casavant; Kuhl (1988) considerem erroneamente os termos política e mecanismo como sinônimos, na realidade não são. A política ou estratégia determina quais, quando e como os mecanismos serão empregados para que o escalonamento seja realizado, com base nos objetivos propostos para o escalonamento. Os mecanismos compreendem os comandos e/ou instruções que, atuando como "ferramentas", desempenham as atividades básicas para a realização do escalonamento. Diferentes grupos de políticas e de mecanismos

podem ser combinados e, assim, produzirem resultados completamente diferentes. [Tanenbaum (1992), Song et al. (1997)].

Exemplos de mecanismos são: (1) instruções que coletam informações sobre a carga do sistema, (2) instruções que trocam informações entre os processadores, (3) instruções que salvam o estado de um processo em execução para que o mesmo possa ser posteriormente transferido para outro processador, entre outros [Song et al. (1997)].

Exemplos de políticas são: (1) quando e quais informações sobre a carga do sistema devem ser coletas pelos mecanismos; (2) qual processo deve ser suspenso e transferido para outro processador; (3) qual critério usar para escolher um processador que receba o processo suspenso.

Um algoritmo de escalonamento será utilizado neste trabalho como um termo mais genérico, responsável por implementar políticas e mecanismos utilizados na atividade de escalonar.

Escalonamento de processos, balanceamento de cargas e alocação de recursos. Feitelson; Rudolph (1995) citam que o maior problema de se pesquisar sobre escalonamento de processos em computadores paralelos é que essa expressão possui significados diferentes para diferentes pessoas. Embora essa afirmação limite o problema aos computadores paralelos (deveria abranger as plataformas distribuídas como um todo), ela é verdadeira.

Muitos dos termos empregados são válidos porque são usados para distinguir políticas propostas, através de alguma característica do escalonamento em si. O problema reside em não se explicar o significado desses termos, utilizando-os diretamente como sinônimo da atividade de escalonamento. Outro grande problema é a falta de um contexto geral, onde as novas propostas e seus termos não são postos

em um contexto conhecido por todos, para que se possa entender o que foi desenvolvido e os seus objetivos [Souza (2000)].

Um dos problemas mais comuns é o uso do termo balanceamento de cargas como sinônimo de escalonamento realizado em tempo de execução da aplicação. A associação direta desses termos, mesmo que implícita, não é correta, porque o balanceamento das cargas de uma plataforma é um dos possíveis "objetivos" de um algoritmo de escalonamento, e não o único. Mullender (1993) exemplifica esse problema afirmando: "é possível, através de uma alocação prudente, otimizar o desempenho do sistema como um todo. Esse processo é conhecido como balanceamento de carga...". Uma alocação prudente pode ser feita com vários objetivos, não apenas o de balancear as cargas de uma plataforma.

Bestavros (1997) diverge de Mullender (1993) e é um exemplo de que balanceamento de cargas não deve nem ser sinônimo e nem ser o único objetivo do escalonamento. Bestavros (1997) propõe uma nova política de escalonamento para um sistema distribuído de tempo real, cujo objetivo é garantir que os processos forneçam as respostas esperadas dentro dos prazos máximos preestabelecidos. O balanceamento das cargas de trabalho nesse caso é irrelevante.

Lüling et al. (1993) exemplifica o uso conflitante de vários termos da área, mesmo propondo uma nova taxonomia. Os termos compartilhamento de recursos, escalonamento de recursos, escalonamento de *jobs*, migração de tarefas e balanceamento de cargas "dinâmico" são usados como sinônimos. A união de alguns desses termos, como migração de tarefa e escalonamento de *job* não é comum e diverge de vários outros autores.

Franklin; Govindan (1996), Graham; Barker (1994) e Tripathi; Karnik (1995) utilizam os termos balanceamento de carga "estático" para designar a atribuição

inicial de processos em tempo de execução e balanceamento de carga "dinâmico" para designar a transferência de processos de um processador para outro (migração de processos). O termo estático aqui diverge com o usado por outros autores para designar o escalonamento em tempo de compilação. Franklin; Govindan (1996) também utilizam o termo adaptável como sinônimo de dinâmico, fato encontrado somente nesse artigo, dentre os pesquisados neste trabalho. Alguns autores como Alanyali; Hajek (1995), Ferrari; Zhou (1987), Mehra; Wah (1993) e Shirazi et al. (1995a) utilizam o termo balanceamento de carga "dinâmico", apenas como sinônimo para escalonamento em tempo de execução (sem migração).

As propostas de escalonamento que incluem migração de processos ativos apresentam vários termos que exemplificam as divergências da terminologia. Alguns sinônimos para migração de processos são: escalonamento com preempção, balanceamento de carga, balanceamento de carga dinâmico/automático, preempção migratória, migração dinâmica de tarefa, alocação automatizada, reatribuição dinâmica, entre outros. Muitos desses termos não são usados exclusivamente como sinônimos de migração de processos, cabendo ao leitor a responsabilidade por distinguir tais diferenças [Souza et al. (2000)].

Shivaratri et al. (1992) e Xu; Lau (1997) distinguem "compartilhamento de carga" de "balanceamento de carga". O compartilhamento de carga denomina algoritmos que evitam estados de não-compartilhamento, aqueles em que um processador permanece ocioso enquanto ainda há tarefas para serem executadas em outros processadores. Por outro lado, o balanceamento de carga, embora também evite estados de não-compartilhamento, vai além, e equilibra a carga dos processadores. Wang; Morris (1985) adotam outra postura e afirmam que os termos balanceamento e compartilhamento de carga são sinônimos.

Os termos escalonamento *on-line* e *off-line* são usados respectivamente como sinônimos para o escalonamento decidido em tempo de execução (dinâmico) e o decidido em tempo de compilação (estático) por alguns autores como Feitelson et al. (1997), Schwiegelshohn (1996) e Shmoys et al. (1995). No entanto, não é estabelecido o relacionamento entre os termos *on-line* & dinâmico e *off-line* & estático nesses trabalhos, cabendo ao leitor essa responsabilidade.

"Escalonamento paralelo" é usado por Shu; Wu (1996) como um escalonamento realmente feito em paralelo e de uma maneira síncrona por todos os processadores envolvidos na execução da aplicação paralela. Esse escalonamento é dividido em uma "fase de sistema", onde todos os processadores, de maneira síncrona, coletam informações sobre a carga do sistema e efetivamente escalonam seus processos; e na "fase de usuário", onde todos os processadores executam os processos escalonados. Feitelson et al. (1997) e Gibbons (1997) também utilizam "escalonamento paralelo", porém, de maneira divergente. O termo é usado no sentido de "escalonamento de *jobs* paralelos" e em nada se assemelha ao trabalho de Shu; Wu (1996).

Outra questão é a preferência de certos autores com os termos "alocação/gerenciamento de recursos", em detrimento ao termo "escalonamento de processos". Casavant; Kuhl (1988) explicam que "escalonamento e alocação nada mais são do que dois termos que descrevem a mesma atividade, com diferentes pontos de vista". Enquanto o "escalonamento de processos" aborda a questão sob o ponto de vista dos processos, o termo "alocação de recursos" é direcionado aos processadores.

Ferstl (1996) não segue a mesma definição de Casavant; Kuhl (1988) para "alocação" e "escalonamento". Em Ferstl (1996) é utilizado o termo "sistema de gerenciamento de recurso e de *job*", separando explicitamente o termo "escalonador"

como um item desse sistema de gerenciamento. O escalonador é o responsável por atribuir a carga de trabalho para os processadores. Ferstl (1996) não é o único a destacar o "escalonamento" (ou escalonador), separando-o como se fosse um item diferente das demais atividades feitas nas ferramentas de software propostas. Saphir et al. (1995) e Krueger et al. (1994) também são exemplos de trabalhos que apresentam essa separação, cada um a seu modo [Souza et al. (2000)].

A separação do termo "escalonamento" é empregada normalmente nos trabalhos voltados à computação paralela. Nessas separações o termo "escalonamento" é empregado no sentido de "política de escalonamento". Outras partes como: "gerenciador de recursos", "alocador de processadores", "gerenciador de tarefas" e "servidor de usuários" são, na realidade, ou (1) mecanismos para coleta de informações sobre a plataforma computacional e "reserva" de processadores para um grupo de processos ou (2) uma interface de acesso ao escalonamento para, por exemplo, submissão de processos.

Neste trabalho será utilizado "escalonamento de processos", no lugar dessa grande quantidade de termos inseridos na literatura. A opção por esse termo deve-se à simplicidade e objetividade da terminologia.

2.4.2. Razões para as Divergências na Terminologia

Genericamente, o que pode explicar essa situação é a falta de uma teoria básica e comum sobre escalonamento, assim como a grande abrangência da área. As publicações sobre escalonamento na computação estão direcionadas para: sistemas operacionais (normalmente distribuídos), computação paralela e ferramentas de software responsáveis pelo escalonamento em plataformas distribuídas (implementadas no espaço do usuário). Assim, trabalhos em qualquer uma dessas

classes apresentam implementações, terminologias e/ou classificações próprias, que nem sempre aplicam-se perfeitamente a outras classes.

Um dos fatores implícitos nos trabalhos desenvolvidos é que as pessoas costumam utilizar uma terminologia que lhes é usual. Os pesquisadores acostumados a trabalhar diretamente com a computação paralela ou com os sistemas operacionais (distribuídos ou não), utilizam seus próprios termos, muitas vezes, para designar a mesma atividade. Novamente, o que falta é uma teoria única e básica para a área de escalonamento, independente da plataforma computacional e do pesquisador.

Tanenbaum (1992 e 1995) exemplifica esse caso. Para definir o escalonamento feito sobre um sistema distribuído, Tanenbaum utiliza o termo "alocação de processadores", destacando o escalonamento sob o ponto de vista da plataforma. Embora esses termos estejam direcionados à plataforma, a definição dada é direcionada à aplicação ("quais processos serão atribuídos para quais processadores"). A adoção dessa terminologia, justificada pelo autor como "... seguindo a tradição ...", é creditada neste trabalho ao fato do termo "escalonamento de processos" já ter sido utilizado anteriormente para definir exclusivamente o escalonamento local.

As explicações para as divergências podem ser encontradas em dois grandes grupos de trabalhos. No primeiro grupo os trabalhos divergem por serem voltados às necessidades ou da computação paralela ou dos sistemas distribuídos (escalonamento na computação paralela x escalonamento em sistemas distribuídos). O segundo grupo representa os trabalhos que divergem em função do local da sua implementação (implementações no *kernel* x no espaço do usuário). Os próximos parágrafos discutem o por quê das divergências nesses dois grupos e demonstram alguns casos onde as mesmas ocorrem.

Escalonamento na Computação Paralela x Escalonamento em Sistemas Distribuídos: Rapine et al. (1998) citam que o escalonamento na computação paralela é um problema de otimização da utilização dos recursos pelos processos e nos sistemas distribuídos é um problema de gerenciamento de requisições de vários usuários.

As duas áreas, computação paralela e sistemas distribuídos, embora apresentem semelhanças no que se refere à atividade de escalonamento, apresentam diferenças expressivas, tornando muitos dos algoritmos desenvolvidos para uma área impróprios para a outra.

Saphir et al. (1995) destacam o por quê dos algoritmos de escalonamento desenvolvidos para aplicações seqüenciais não serem apropriados para aplicações paralelas. Para esses autores o escalonamento feito para computadores paralelos com memória distribuída são significativamente mais complexos que aqueles feitos para plataformas com um processador. A maior complexidade deve-se, por exemplo: às solicitações por processadores nas aplicações paralelas, que variam o número e o tipo de nós; às exigências por outros recursos, como grande quantias de memória; à escolha do método para *time sharing* que afeta diretamente o desempenho da política de escalonamento e à maior dependência entre os processos pertencentes à aplicação.

Feitelson; Rudolph (1995) e Islam et al. (1996) são trabalhos voltados exclusivamente à computação paralela que apresentam uma análise sobre diferentes algoritmos pertencentes à política *space sharing*. Essa política designa algoritmos de escalonamento que dividem os processadores em grupos e permitem que somente os processos de uma aplicação paralela sejam atribuídos a esses grupos. Não encontrou-se na literatura pesquisada, nenhum escalonamento *space sharing*, nem tão pouco

diferentes tipos de partições de processadores (*processor partitioning*) feitas para "sistemas distribuídos".

Wang; Morris (1985) é um exemplo de trabalho voltado exclusivamente aos sistemas distribuídos. A proposta inclui apenas processos independentes, seqüenciais, sem migração e em plataformas homogêneas, apresentando, dessa forma, uma abordagem completamente distinta em relação às apresentadas anteriormente nesta seção. Os trabalhos de Graham; Barker (1994), Lim; Kim (1995), Shivaratri et al. (1992), também são exemplos direcionados aos sistemas distribuídos [Souza et al. (2000)]. Apesar de se encontrar na literatura alguns poucos trabalhos que se preocupam com o escalonamento e com a sincronização de vários *threads* em um sistema distribuído [Mullender (1993), Tanenbaum (1995)], a ênfase dada a esses problemas é pequena, quando comparada à ênfase desses mesmos problemas na computação paralela [Souza et al. (2000)].

Implementações no *kernel* e no espaço do usuário (*user space*): mesmo com objetivos semelhantes, diferentes implementações permitem que seus autores criem nomenclaturas próprias. Uma das possíveis variações na implementação pode ser o local onde o escalonamento é implementado: no *kernel* do sistema operacional ou no espaço do usuário.

Para Feitelson; Rudolph (1995) o escalonamento dinâmico pode ser feito no sistema operacional ou então em *runtime systems* (terminologia própria). *Runtime systems* são ferramentas no espaço do usuário com objetivo de satisfazer necessidades individuais de uma aplicação. Quando as necessidades passam a ser coletivas o escalonamento deve ser feito no sistema operacional. Paindaveine; Milojicic (1996) criam uma distinção entre os termos migração de processos e de tarefas, com o intuito de descrever a migração em diferentes níveis no sistema

operacional distribuído Mach (desde o *kernel* do sistema operacional até a aplicação). Feitelson; Rudolph (1995), Barak et al. (1996) e Paindaveine; Milojicic (1996) defendem o escalonamento feito no sistema operacional [Souza et al. (2000)].

Ferstl (1996) diverge nesse sentido, defendendo que o escalonamento seja feito por um sistema de gerenciamento de processos e recursos, independente do sistema operacional. Essa separação tem como vantagens: (1) permitir a utilização dos recursos de uma maneira estruturada, planejada e controlada pela administração; (2) oferecer recursos de uma plataforma computacional para os seus usuários de uma maneira transparente e de fácil utilização e compreensão; (3) fornecer uma interface ao usuário independente da plataforma computacional (portabilidade).

Segundo Krone et al. (1998) quando o escalonamento é feito em plataformas distribuídas, heterogêneas e multiusuárias, o escalonamento deve ser feito fora do sistema operacional e integrado à aplicação, em função da heterogeneidade em diferentes níveis (tanto no hardware quanto no software). Segundo os autores, "é praticamente impossível" adaptar todos sistemas operacionais dos diferentes computadores existentes em tais plataformas, para os objetivos do escalonamento. Ainda segundo Krone et al. (1998), uma das vantagens de se fazer o escalonamento no espaço do usuário é justamente o desempenho, qualidade destacada por outros autores como sendo do escalonamento no *kernel*.

Embora os defensores do escalonamento no *kernel* do sistema operacional aleguem falta de transparência no escalonamento feito no espaço do usuário, os trabalhos publicados nesse nível demonstram várias alternativas que implementam o escalonamento de modo completamente transparente à aplicação.

Para Eskicioglu (1989) há a possibilidade de implementar o escalonamento nos dois níveis: *kernel* e espaço do usuário. Implementando-se os mecanismos no *kernel*

e as políticas no espaço do usuário, minimiza-se a sobrecarga extra, gerenciando certas informações diretamente no *kernel*. Segundo o autor a desvantagem está no "chaveamento" necessário para a troca das informações entre os dois níveis.

2.5. Taxonomias da Área de Escalonamento de Processos

Para Baumgartner; Wah (1991) o objetivo de uma taxonomia deve ser aumentar e organizar o "conhecimento total" sobre uma classe de problemas, especificando-a e mostrando o relacionamento entre problemas. Os autores apontam ainda, no mínimo quatro características desejáveis de uma taxonomia: (1) identificar características significantes; (2) mostrar claramente o relacionamento entre os problemas; (3) capaz de expandir e contrair; e (4) separar a especificação do problema da sua solução. Esta última característica distingue a política empregada para solucionar um problema do problema em si, separando dessa maneira o "como é resolvida uma questão" do "o que deve ser resolvido".

A área de escalonamento de processos possui várias taxonomias, muitas delas criadas sem as características descritas acima por Baumgartner; Wah (1991) [Chen; King (1996), Gibbons (1997), Hwang; Xu (1998); Kemelmakher; Kremien (1998), Milojicic et al. (1993), Mullender (1993), Quinn (1994), Russ et al. (1996) e Tanenbaum (1995)]. Em função disso, esta seção descreve algumas taxonomias, escolhidas devido às suas abrangências e por serem aceitas em outros trabalhos.

2.5.1. Taxonomia de Casavant; Kuhl (1988)

A taxonomia de Casavant; Kuhl (1988) é híbrida, situando-se entre hierárquica e plana. São definidos dois conjuntos, um com termos que relacionam-se entre si de maneira hierárquica (Figura 2.3) e outro com termos que podem relacionar-se de maneira independente (taxonomia plana).

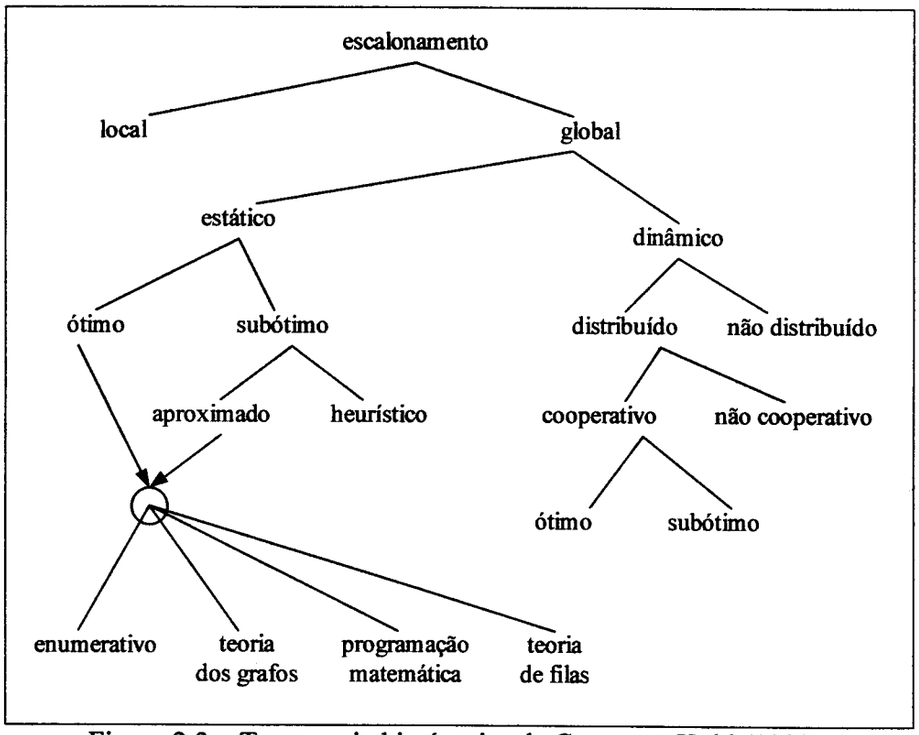


Figura 2.3 – Taxonomia hierárquica de Casavant; Kuhl (1988).

Taxonomia Hierárquica.

Local & Global: as diferenças entre o escalonamento local e global propostas por Casavant; Kuhl (1988) já foram apresentadas no início deste capítulo. Não houve nenhum comentário a respeito de um escalonamento global integrado com o escalonamento local nessa taxonomia, sendo citado apenas que o escalonamento global e local deveriam ficar separados, com o local a cargo do sistema operacional. Não houve uma preocupação com a sincronização criteriosa entre os diferentes processos pertencentes à aplicação. Essa sincronização é amplamente discutida nos trabalhos que envolvem a computação paralela.

Estático & Dinâmico: o escalonamento global estático e dinâmico indicam o tempo em que as decisões de atribuição são feitas e também indicam a quantia de informação que se dispõe para o escalonamento. A característica estática ou dinâmica é uma das mais destacadas pela literatura, embora não haja um consenso a

respeito desses termos. No escalonamento estático, o máximo de informações a respeito dos processos e dos processadores devem estar disponíveis em tempo de compilação/"linkedição". Assim, quando a aplicação for executada, o escalonamento já está determinado e não mudará em tempo de execução. O escalonamento estático é integrado à aplicação, onde, no máximo durante a compilação/*linkedição* da mesma é que o escalonamento é determinado. O escalonamento estático pode ser determinado ou pelos responsáveis pelo desenvolvimento da aplicação paralela, durante a sua construção e antes da compilação; ou então por uma ferramenta de software que analise sintática e semanticamente o código desenvolvido durante a fase de compilação. As principais vantagens do escalonamento estático (determinístico, *off-line* ou de tarefa) são: (1) ausência de sobrecarga para determinar o escalonamento em tempo de execução e (2) maior controle da aplicação em plataformas monousuárias, devido à grande quantidade de informações necessárias (tanto de hardware quanto de software) para que o escalonamento atinja seus objetivos. As desvantagens do escalonamento estático ficam por conta da falta de: flexibilidade, portabilidade, capacidade escalar (*scalability*), tolerância a falhas, adequabilidade às plataformas multiusuárias e também pela exigência de um alto conhecimento prévio a respeito do comportamento da aplicação em relação à plataforma usada [Souza et al. (2000)].

O escalonamento dinâmico é feito em tempo de execução da aplicação e possui muito pouco ou nenhum conhecimento prévio sobre as necessidades dos processos que serão executados nem da plataforma utilizada. O escalonamento dinâmico é amplamente pesquisado e possui inúmeras implementações, principalmente em função das suas vantagens que são, potencialmente: flexibilidade, portabilidade, tolerância a falhas, capacidade escalar e adequação às plataformas multiusuárias.

Considerando plataformas multiusuárias baseadas em *NOWs*, o escalonamento dinâmico é necessário para se atingir um desempenho melhor. Além da dificuldade de se antever o impacto causado pelos possíveis usuários simultâneos, essas plataformas computacionais podem ser alteradas mais facilmente quando comparadas, por exemplo, com os computadores paralelos. A maior desvantagem do escalonamento dinâmico é a sobrecarga gerada com as decisões em tempo de execução.

Ótimo & Subótimo: um escalonamento ótimo pode ser realizado quando se possui todas as informações necessárias a respeito dos processos que serão escalonados e da plataforma computacional utilizada. Entretanto, a geração de uma solução ótima é, na maioria dos casos, um problema NP-completo, inviabilizando a sua realização. Uma alternativa para esse problema NP-completo são as soluções subótimas, que procuram realizar "um bom escalonamento". O problema é determinar o que é "um bom escalonamento" e qual métrica avalia a solução encontrada.

Aproximado & Heurístico: o escalonamento aproximado utiliza algum modelo computacional formal para o algoritmo, porém, tem por objetivo alcançar um escalonamento aceitável, ao invés de buscar a melhor solução para o escalonamento. O escalonamento heurístico, por sua vez, utiliza parâmetros que afetam a plataforma computacional de maneiras indiretas. Normalmente os parâmetros estão relacionados indiretamente com o desempenho e são mais simples para monitorar e calcular. Casavant; Kuhl (1988) citam como um exemplo de heurística grupos de processos I/O-Bound que são escalonados no mesmo processador e grupos de processos CPU-Bound que são escalonados em processadores distintos. Essa heurística diminui

diretamente a sobrecarga envolvida na troca de informações entre os processos, enquanto se beneficia do paralelismo.

Distribuído & Não Distribuído: a responsabilidade pelo escalonamento dinâmico pode ser apenas de um processador (escalonamento não distribuído) ou fisicamente distribuída entre os processadores (escalonamento distribuído) [Souza et al. (2000)].

Cooperativo & Não Cooperativo: o escalonamento distribuído pode ser dividido considerando o grau de autonomia que cada processador possui para determinar como o escalonamento deve ser feito. No caso do não cooperativo, um processador toma as suas decisões de maneira independente das ações dos outros, não se preocupando com os efeitos das suas decisões sobre o restante da plataforma. No cooperativo cada processador é responsável por realizar a sua porção do escalonamento, onde todos os processadores agem seguindo um único objetivo global. Assim, com o cooperativo, os processadores não se preocupam apenas em melhorar o seu desempenho local, mas sim o desempenho do sistema como um todo.

Taxonomia Plana:

Adaptável & Não Adaptável: quando os algoritmos e/ou parâmetros usados para implementar a política de escalonamento são alterados dinamicamente em função do comportamento atual da plataforma computacional, diz-se que essa política é adaptável. Uma solução "não adaptável" é aquela onde a política de escalonamento (dinâmica ou não), considera sempre as mesmas regras e os mesmos parâmetros para realizar o escalonamento. Toda solução adaptável é uma solução dinâmica, mas uma solução dinâmica não necessariamente é adaptável [Souza et al. (2000)].

Balanceamento de Carga: os autores colocam o balanceamento de carga como uma possível política para o escalonamento, no mesmo nível de características como: adaptável, migração de processos, entre outros. Apesar dessa taxonomia ser seguida por alguns autores, este trabalho designa o balanceamento de cargas como um objetivo do escalonamento e não como uma característica de implementação. A questão aqui é como o escalonamento é feito e não o que o escalonamento faz.

***Bidding* (Licitação)**: o escalonamento através de licitação é feito por uma negociação (troca de informações) entre os processadores que possuem processos para serem executados remotamente e processadores que estão aptos a executarem processos. É citado como exemplo um escalonamento formado por um processador gerente (*manager*) e por processadores contratantes (*contractors*). O gerente representa o processador com processos ofertados para execução. Os contratantes representam processadores aptos a receber os processos. Após receberem o pedido para execução, os contratantes oferecem-se para executar os processos solicitados, indicando as suas características locais. Dessa maneira, o gerente escolhe a melhor opção de processador no momento e permite a autonomia dos contratantes (participam da negociação apenas os processadores que estiverem aptos).

Probabilístico: essa política tem por objetivo reduzir a sobrecarga existente com a geração de todas as possíveis soluções para o escalonamento de processos. A política probabilística realiza o escalonamento dos processos aleatoriamente (*randomly*), segundo alguma distribuição. Executando essa política repetidamente, diferentes escalonamentos são gerados e, analisados posteriormente, permitem que o melhor seja escolhido dentre aqueles criados.

Atribuição Inicial & Reatribuição Dinâmica: o escalonamento feito apenas através de uma atribuição inicial (não preemptivo ou sem migração), é aquele que

considera apenas uma única vez questões como onde e quando os processos devem ser executados. Para os autores, esse escalonamento deve considerar apenas as informações fornecidas pela aplicação, como: tempo estimado para execução ou demanda por recursos. O escalonamento feito com reatribuição dinâmica (mais conhecido como escalonamento preemptivo ou com migração de processos) corrige uma "atribuição inicial" já realizada que tornou-se incorreta devido a variações na plataforma computacional. A migração de processos suspende um processo em um processador e o conclui em outro processador. A correção pela transferência de um processo possui uma sobrecarga alta quando comparada com o escalonamento não preemptivo e deve ser bem parametrizada para que se consiga avaliar os ganhos reais com a migração [Downey; Harchol-Balter (1995)]. Todo o escalonamento que realiza a "migração de processos" é um escalonamento dinâmico, mas nem todo escalonamento dinâmico realiza a migração de processos.

A questão da migração de processos é uma das mais complexas dentro da área do escalonamento, com várias propostas já desenvolvidas. Sozinha, a migração de processos constitui-se uma ampla linha de pesquisa, por ser um dos recursos mais eficientes do escalonamento dinâmico para se adaptar às possíveis variações em uma plataforma multiusuária e tolerante a falhas.

Vantagens & Desvantagens: essa taxonomia é uma das mais completas, por agrupar inúmeros termos espalhados pelos trabalhos. Em relação à taxonomia hierárquica, a proposta é coerente e destaca-se das demais porque, apesar de existirem outras poucas classificações hierárquicas, essa estabelece um relacionamento entre termos muitas vezes conhecidos. A taxonomia plana é válida, pois agrega algumas características adicionais aos trabalhos propostos.

A proposta de Casavant; Kuhl (1988) apresenta alguns problemas que não comprometem o mérito do trabalho [Souza et al. (2000)]. Em alguns pontos houve uma limitação tecnológica da taxonomia, como por exemplo no item "balanceamento de carga". Nesse item é afirmado que o balanceamento é mais eficaz quando os processadores são homogêneos, em função de todos os processadores saberem a estrutura e potência dos outros. Isso poderia ser uma realidade em 1988, porém, com as plataformas heterogêneas atuais essa afirmação está defasada. Certas características foram omitidas, como por exemplo: quando são citadas políticas para troca de informação entre processadores, apenas as políticas periódica e sob demanda. Na realidade há também a por mudança de estado.

O item sobre migração de processos afirma que o escalonamento sem migração usa apenas informações fornecidas pela aplicação. Assim, não devem ser consideradas outras informações sobre as características atuais da plataforma, como o nível de utilização dos processadores. Essa afirmação não é válida, porque é possível ter-se um escalonamento sem migração e que considera a situação da plataforma no momento do escalonamento.

A taxonomia plana apresenta um número muito reduzido de termos. Poderiam ser incluídos itens a mais, como: quem é o responsável por iniciar o escalonamento (processadores transmissores, receptores ou iniciado simetricamente), se o escalonamento é feito no sistema operacional ou não, quanto há transparência para a aplicação, entre outros.

Falta estabelecer um relacionamento entre a taxonomia apresentada nesse trabalho com outras facilmente encontradas na literatura, como a taxonomia *space sharing* e *time sharing*. Mesmo com os autores citando que é possível adaptar outras

propostas, que enfocam mais o gerenciamento de recursos ao escalonamento de processos, essa analogia não é trivial e não foi feita no trabalho.

2.5.2. Taxonomia de Wang; Morris (1985)

Wang; Morris (1985) propõem uma taxonomia direcionada apenas ao compartilhamento de carga em sistemas distribuídos locais, apresentando algumas limitações como: o uso de plataformas homogêneas, execução de processos independentes, seqüenciais e sem migração [Souza et al. (2000)].

A taxonomia divide os algoritmos de escalonamento através de duas características. A primeira refere-se a quem tem a iniciativa de iniciar o escalonamento. Para tanto, os processadores foram classificados ou como origem ou servidor (Figura 2.4). Os termos origem e servidor também são conhecidos respectivamente como transmissor e receptor em vários outros trabalhos. Os processadores origem possuem processos para serem executados, enquanto os processadores servidores desejam executar processos remotos. Um processador pode ser simultaneamente origem e servidor, também conhecido como simétrico [Shivaratri et al. (1992)]. Se um processador origem é quem inicia o escalonamento, essa política é classificada como "iniciada pela origem" (*source-initiative*). Se um processador servidor é quem inicia o escalonamento essa política é classificada como "iniciada pelo servidor" (*server-initiative*).

A segunda distinção refere-se à dependência por informação que a política de escalonamento possui. Essa distinção indica a quantidade de informação que um processador origem possui (ou necessita) sobre os processadores servidores e vice-versa. Para determinar a quantidade de dependência por informação os autores criaram sete níveis (1-7), onde o número "um" representa a menor e o número "sete" a maior dependência. Para Wang; Morris (1985) no nível "um" estariam as políticas

de escalonamento estáticas e a partir do nível "cinco" estariam as políticas de escalonamento dinâmicas. Entre o nível "um" e o nível "cinco" estariam as políticas semidinâmicas , ou híbridas, conforme preferem alguns autores como Shirazi et al. (1995a).

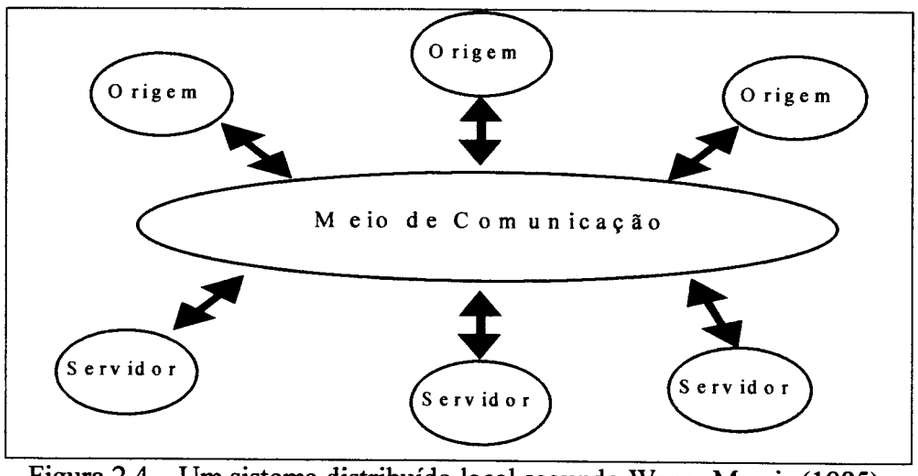


Figura 2.4 – Um sistema distribuído local segundo Wang; Morris (1985).

Vantagens & Desvantagens: As grandes vantagens desse trabalho ficam por conta da utilização de termos conhecidos e pela facilidade de compreensão. A taxonomia proposta por Wang; Morris(1985) é uma das mais antigas, onde, em função da sua simplicidade e limitações, ela é pouco referenciada em outros trabalhos. Apesar de ser anterior à taxonomia proposta por Casavant; Kuhl (1988) e de utilizar termos semelhantes (e conhecidos), outros autores preferem citar esta última, provavelmente em função da sua maior abrangência.

2.5.3. Taxonomia de Baumgartner; Wah (1991)

Baumgartner; Wah (1991) além de apresentarem uma nova taxonomia, fazem também um levantamento sobre as taxonomias propostas até então. Esse levantamento situa a proposta feita e permite uma comparação objetiva entre diferentes taxonomias.

A primeira distinção feita é entre os escalonamentos determinísticos, não determinísticos e indecidíveis. O determinístico caracteriza-se por ter, previamente, todas as informações necessárias e também por ser completamente definido em função do tempo (processamento) e espaço (memória e disco). O não determinístico pode ser completamente definido, no entanto, para encontrar uma solução exata leva-se muito tempo ou necessita-se muito espaço ou ambos. O indecidível é impossível de ser realizado, mesmo que haja infinitas capacidades de processamento e recursos de armazenagem.

Para situar a nova taxonomia (chamada ESR), os autores discutem sobre dois grupos de taxonomias. O primeiro grupo relaciona algumas taxonomias existentes até então, abordando tanto escalonamentos determinísticos quanto não determinísticos. Esse primeiro grupo enfoca os algoritmos não determinísticos. O segundo grupo é voltado para o escalonamento determinístico e é a base para a taxonomia proposta.

Tabela 2.1 – Revisão das taxonomias de escalonamento [Baumgartner; Wah (1991)].

Terminologia	
Característica	Valores
Nível do Escalonamento	Intraprocessador, Interprocessador
Flexibilidade das Normas	Estático, Dinâmico
Localização do Controle	Distribuído, Hierárquico, Centralizado
Cooperação	Negociação, Independente
Iniciação	Transmissor, Receptor, Ambos
Adaptabilidade	Adaptável, Não Adaptável

Considerando o primeiro grupo, seis características dos algoritmos de escalonamento são agrupadas, conforme a Tabela 2.1. A Figura 2.5 demonstra graficamente o relacionamento entre esses termos.

O "nível do escalonamento" distingue o escalonamento em intraprocessador interprocessador. Esses termos correspondem, respectivamente aos escalonamentos local e global da taxonomia de Casavant; Kuhl (1988).

A "flexibilidade das normas" pode ser estática ou dinâmica e refere-se à flexibilidade inserida nas regras do escalonamento para reagir ao estado atual da plataforma computacional.

A "localização do controle" é dividida em distribuída, hierárquica e centralizada e descreve como a responsabilidade pelas decisões do escalonamento é repartida entre os processadores pertencentes à plataforma.

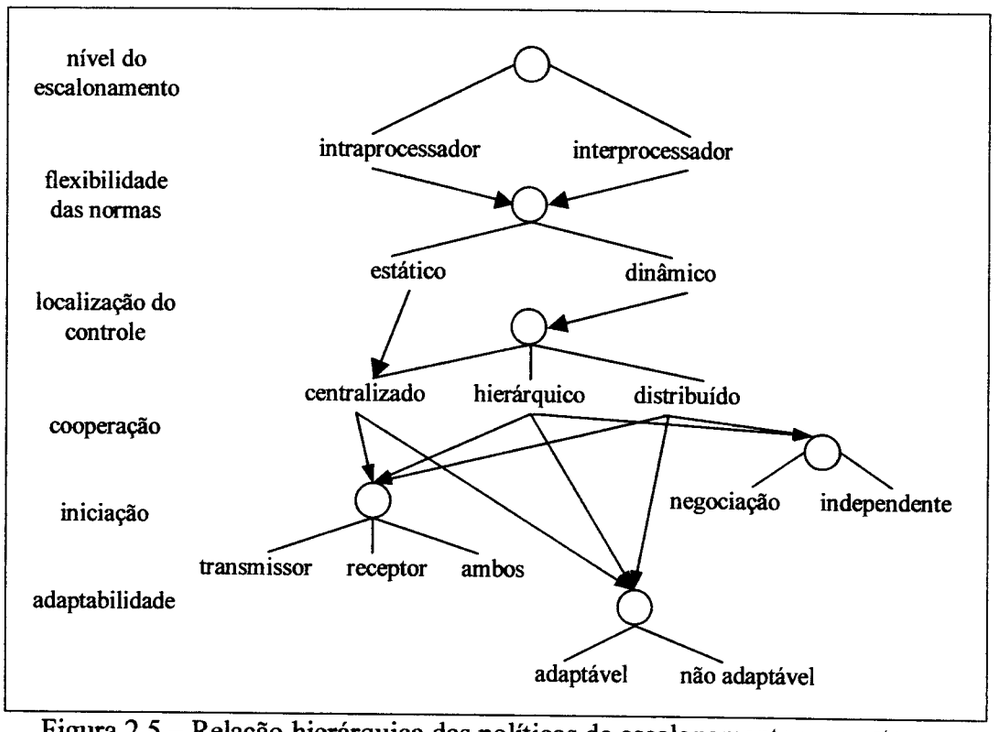


Figura 2.5 – Relação hierárquica das políticas de escalonamento proposta por Baumgartner; Wah (1991).

A "cooperação" descreve a quantia de interação realizada entre os processadores, podendo ser com negociação ou independente. Esses termos correspondem, respectivamente, aos termos cooperativos e não cooperativos da taxonomia de Casavant; Kuhl (1988).

A "iniciação" destaca o algoritmo de escalonamento através de quem é o responsável por iniciar o escalonamento. Pode ser iniciado pelo transmissor (ou origem, sobrecarregado), pelo receptor (ou destino, ocioso, servidor) ou ambos.

A "adaptabilidade" refere-se à mesma característica apontada por Casavant; Kuhl (1988), referindo-se à flexibilidade do algoritmo em mudar a sua política.

Considerando o segundo grupo de taxonomias (voltado para escalonamentos determinísticos) os algoritmos são classificados por características como: processos (α), ambiente de execução (β) e objetivos (γ). Essas características, quando divididas em outros componentes, permite uma especificação maior dos algoritmos classificados. Esse esquema é a base da nova taxonomia, onde são acrescentados itens que permitem o relacionamento de algoritmos com características determinísticas e também estocásticas.

ESR – a nova proposta de taxonomia: A nova proposta de taxonomia é chamada de ESR (*Event, Environment (Surroundings) and Requirements*) e está apresentada na Tabela 2.2.

A primeira categoria "E" corresponde aos eventos. Os eventos são as entidades que serão escalonadas, como por exemplo os processos. Considerando o relacionamento entre os eventos, estes podem ser: completamente independentes, com alguma ordem de precedência entre eles ou com a necessidade de trocar informações durante a execução. Em função da disponibilidade (taxa de chegada) os eventos podem ser: (1) estáticos, onde os processos chegam simultaneamente antes da sua execução, (2) periódicos, onde os processos chegam em intervalos conhecidos, mas distantes o suficiente para permitir que os processos já escalonados terminem a sua execução e (3) estocásticos, onde a chegada dos processos é determinada por uma distribuição de probabilidade. Em função da necessidade por recursos, os processos podem ser: (1) determinísticos, onde as futuras solicitações por recursos são especificadas como constantes, ou então (2) estocásticos, onde essas solicitações por recursos são regidas por alguma distribuição de probabilidade.

Tabela 2.2 – Taxonomia ESR proposta por Baumgartner; Wah (1991).

Categoria	Atributo	Valores
Evento (E)	Relação entre os eventos	Independente, precedente, comunicação
	Disponibilidade (taxa de chegada)	estática, periódica, estocástica
	Necessidades por recursos	determinística, estocástica
Ambiente (S)	Número	1, k, n
	Classes de recursos	recursos homogêneos (1 classe) recursos heterogêneos (> 1 classe)
	Características físicas	velocidade, tamanho da memória principal, capacidades especiais, etc. topologia
	Sobrecarga de comunicação	nenhuma, determinística, estocástica
	Mecanismo de comunicação	fluxo de eventos, broadcast, passagem de mensagem, outros
Objetivo (R)	Desempenho	qualquer solução, *deadlines (real time)*, solução boa (com melhoria) e solução ótima.

A segunda categoria "S" [*Environment (Surroundings)*] representa todas as demais características relativas à plataforma computacional, que afetam o desempenho do escalonamento. O número de recursos (processadores) pode ser: "um" indicando o escalonamento em apenas um processador (local ou intraprocessador), *k* (um número específico) indicando uma quantidade exata de recursos, ou *n* (um número arbitrário) indicando uma quantidade variada de recursos envolvidos. A classe de recursos indica a heterogeneidade contida na plataforma computacional: uma classe indica recursos homogêneos e mais de uma classe indica recursos heterogêneos. As características físicas das classes de recursos podem ser especificadas, se elas são decisivas para o escalonamento. A sobrecarga de comunicação, embora pudesse ser considerada uma das características das classes de recursos, indicam o peso da comunicação na plataforma utilizada. A comunicação pode ter um custo "zero", pode ser constante (determinística) ou então ser variada (estocástica). Os mecanismos de comunicação indicam como é feita a comunicação

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

entre os eventos escalonados. Esses podem ser: fluxo de eventos, *broadcast* de informação, passagem de mensagem ou qualquer outra forma de comunicação.

A terceira e última categoria "R", representa os requisitos do escalonamento, chamados neste trabalho de objetivos. O objetivo básico e mais simples é encontrar qualquer solução que execute os processos. Outros objetivos podem ser: redução do tempo de execução, execução de um processo dentro de um prazo máximo (*deadline*) ou qualquer outra métrica de desempenho.

Vantagens & Desvantagens: uma vantagem da taxonomia ESR é a capacidade de representar algoritmos de escalonamento de uma maneira flexível. A taxonomia pode conter maiores detalhes ou ater-se apenas a detalhes genéricos. Outra vantagem é a preocupação com o "como" o escalonamento é feito e não com "o que" ele deve resolver. As desvantagens ficam por conta do relacionamento entre as diferentes propostas (não há uma relação hierárquica entre as características da Tabela 2.2). Com a Tabela 2.1 e com a Figura 2.5, que representam não uma nova taxonomia, mas sim uma revisão de outras propostas, é possível estabelecer esse relacionamento entre diferentes algoritmos. O relacionamento ilustrado pela Figura 2.5 deveria limitar algumas possíveis conclusões errôneas ou então chamar a atenção para que as mesmas não fossem feitas. Com a Figura 2.5 pode-se concluir, erroneamente, que haja um escalonamento interprocessador, estático, centralizado e adaptável (para um escalonamento ser adaptável ele deve ser dinâmico e não estático); ou então um escalonamento intraprocessador, dinâmico, distribuído e com negociação (para um escalonamento ser distribuído ele deve ser interprocessador e não intraprocessador).

2.5.4. Taxonomia de Xu; Lau (1997)

Antes de apresentarem a taxonomia, os autores também citam que o escalonamento possui algumas "questões chave" (*key issues*). Dentre essas questões

aparece a regra de iniciação, a qual pode ser iniciada pelo transmissor, pelo receptor, simetricamente ou ainda periodicamente. As três primeiras (transmissor / receptor / simétrica) foram apresentadas de maneira semelhante aos trabalhos anteriores. A iniciação periódica (ou remapeamento) induz todos os processadores a participarem de um reescalonamento periódico.

Para fundamentar a taxonomia, os autores definem que o "domínio do escalonamento" representa o conjunto de processadores envolvidos no escalonamento dinâmico. O domínio do escalonamento pode ser global ou local. O domínio global permite que todos os processadores pertencentes à plataforma façam parte do escalonamento, enquanto o domínio local restringe o escalonamento apenas a um subconjunto de processadores da plataforma computacional. O domínio global ou local também determina a quantia de informação trocada entre os processadores. Dessa forma, há duas questões envolvidas no domínio do escalonamento: as regras de localização de um parceiro para o escalonamento e as regras para a troca de informações. Algoritmos de escalonamento que trocam informações e procuram parceiros apenas no domínio local são conhecidos como dinâmicos baseados nos vizinhos mais próximos. Os algoritmos baseados nos vizinhos mais próximos também são conhecidos como iterativos (repetitivos), porque transferem processos de um processador para seus vizinhos e assim sucessivamente até atingir os objetivos propostos para o escalonamento. Cada salto feito pelos processos é determinado por uma decisão local, no processador transmissor dos processos. Assim, em um escalonamento iterativo o processador origem (aquele que estava inicialmente com os processos) não sabe qual será o processador destino dos processos, por não fazer essa atribuição (origem/destino) diretamente.

Os algoritmos de escalonamento dinâmicos que localizam seus parceiros e que utilizam informações através do domínio global, são denominados algoritmos de escalonamento globais (ou diretos). O termo direto deve-se ao fato desses algoritmos determinarem o destino dos processos diretamente entre o processador transmissor e o receptor.

Segundo os autores, os algoritmos globais são mais indicados em plataformas que possuem *broadcast* ou políticas de escalonamento centralizadas, em função do custo para determinar e distribuir de maneira eficiente a carga de trabalho de todos os processadores. Já os algoritmos iterativos são mais flexíveis, onde a cada salto dos processos uma nova decisão sobre o escalonamento é tomada por um processador. Os algoritmos iterativos também são mais apropriados para plataformas que possuem a rede de comunicação direta (ou ponto-a-ponto) e com o roteamento do tipo *store-and-forward.*

Xu; Lau (1997) citam que há na literatura várias taxonomias sobre algoritmos de escalonamento em plataformas de computação distribuída baseadas em redes locais, fazendo referência aos trabalhos de Baumgartner; Wah (1991), Casavant; Kuhl (1988) e Wang; Morris (1985). No entanto, esses trabalhos são incompletos por excluírem os computadores paralelos baseados em uma rede de conexão direta.

A Figura 2.6 ilustra a taxonomia proposta. Os algoritmos determinísticos e estocásticos indicam quais serão as regras de distribuição aplicadas ao escalonamento. Os algoritmos de escalonamento iterativos determinísticos possuem regras predefinidas que determinam para qual(is) processador(es) vizinho(s) a carga de trabalho deve ser transferida, além de também determinar qual a quantia dessa carga deve ser transferida. Os algoritmos iterativos estocásticos distribuem os processos na plataforma de tal modo que haja uma alta probabilidade de se atingir os

objetivos propostos, sem utilizar regras e parâmetros predefinidos pela política de escalonamento. Essas definições de determinístico e estocástico divergem das citadas anteriormente por Casavant; Kuhl (1998) e Baumgartner; Wah (1991).

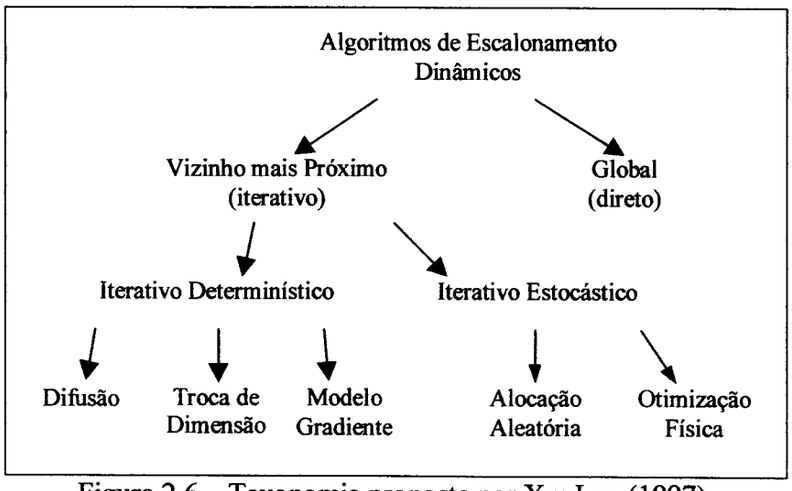
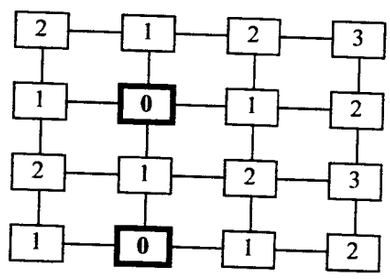


Figura 2.6 – Taxonomia proposta por Xu; Lau (1997).

Os três tipos de algoritmos de escalonamento pertencentes ao escalonamento iterativo e determinístico são: difusão (*diffusion*), troca de dimensão (*dimension exchange*) e modelo gradiente (*gradient model*). Os algoritmos por difusão e por troca de dimensão são parecidos, em função de examinarem todos os seus vizinhos em toda operação de escalonamento. Porém, um algoritmo por difusão permite que, em cada operação de escalonamento, um processador contate todos os seus vizinhos ao mesmo tempo, propagando porções da sua carga de trabalho para um ou mais processadores, enquanto solicita carga de trabalho de outros vizinhos. No algoritmo troca de dimensão, um processador também realiza o escalonamento com seus vizinhos e entra em contato com apenas um processador vizinho por vez. Depois de uma operação de transferência e/ou recebimento de carga de trabalho com um vizinho, o processador comunica a sua carga atual para o próximo vizinho e então uma nova distribuição de carga é iniciada. Maiores detalhes sobre a política de escalonamento baseada em difusão podem ser encontrados em Corradi et al. (1997).

Os algoritmos de escalonamento baseados no modelo gradiente, propostos inicialmente por Lin; Keller (1987), restringem o escalonamento apenas ao vizinho que apresenta a menor carga de trabalho. Segundo Lüling et al. (1993) gradientes são vetores que possuem a menor distância (em saltos) entre o processador atual e um processador ocioso (com carga de trabalho baixa ou nenhuma), permitindo assim, que os processos a serem escalonados saibam a distância e saltem sobre os processadores até encontrarem esse processador ocioso.

A Figura 2.7 demonstra uma superfície gradiente (*gradient surface*) que representa a menor distância de cada processador para o processador ocioso mais próximo. Para se determinar essa menor distância entre os processadores, a carga local é coletada por todos os processadores e enviada sistematicamente para os seus vizinhos mais próximos. Com isso, o escalonamento usando o modelo gradiente pode ser feito sucessivamente com o objetivo de se atingir um balanceamento de carga global para a plataforma.



Sendo **0** um processador com pouca carga de trabalho, os valores dentro dos retângulos representam a distância mínima para um processador ocioso.

Figura 2.7 – Exemplo de um modelo iterativo gradiente com dezesseis processadores [Lüling et al. (1993)].

Os dois tipos de algoritmos de escalonamento pertencentes ao escalonamento iterativo e estocástico são: alocação aleatória e otimização física. A alocação

aleatória é a mais simples das duas políticas, onde os processos são transferidos para um processador vizinho escolhido de maneira arbitrária. Se o processador escolhido, determinar que ficou sobrecarregado após receber os processos, pode transferi-los para outro processador vizinho, também escolhido de maneira arbitrária. Os algoritmos pertencentes ao tipo "otimização física" são baseados em analogias com sistemas físicos. Segundo Xu; Lau (1997) os algoritmos de otimização física fazem o mapeamento do escalonamento de processos sobre sistemas físicos e então solucionam esse escalonamento através de simulação ou técnicas advindas da física experimental ou teórica. A grande vantagem desses algoritmos de otimização física sobre os que utilizam a alocação aleatória é que os primeiros melhoram o controle sobre a maneira aleatória da redistribuição dos processos.

Vantagens & Desvantagens: a taxonomia anexa a algumas taxonomias anteriores a questão da localização dos processadores, considerando também a topologia da plataforma computacional. Embora a topologia também tenha sido considerada em um outro trabalho [Lüling et al. (1993)], a taxonomia de Xu; Lau (1997) possui uma especificação maior e também uma relação hierárquica entre os diferentes tipos de algoritmos de escalonamento. A terminologia usada nas taxonomias de Lüling et al. (1993) e de Xu; Lau (1997) não divergem, fato observado com os termos "local" e "global" utilizados por ambas e com o mesmo significado [Souza et al. (2000)]. O fato da taxonomia ser apenas parcial, estando vinculada ao escalonamento dinâmico e detalhando somente o escalonamento iterativo é uma desvantagem e restringe a sua abrangência.

2.5.5. Taxonomia de Feitelson; Rudolph (1995)

Os autores classificam os algoritmos de escalonamento direcionados à computação paralela, em duas dimensões (tempo e espaço) e em dois níveis (*jobs* e

threads). O termo "nível" aqui, não tem o mesmo sentido apresentado por Baumgartner; Wah (1991) [Souza et al. (2000)].

Considerando as duas dimensões: tempo e espaço, os algoritmos de escalonamento podem ser, respectivamente: *time sharing* ou *space sharing*[2]. Na classe *time sharing* as aplicações paralelas compartilham o uso dos mesmos processadores com outras aplicações, enquanto que na classe *space sharing* um (sub)conjunto específico de processadores são separados e a eles são atribuídas aplicações paralelas que os usam exclusivamente até o final da execução. A classe *space sharing* também é citada como *space partitioning* por alguns autores como Tripathi; Karnik (1995).

Os algoritmos *time sharing* são utilizados onde há alta variação na carga computacional, onde há pouco ou nenhum conhecimento sobre as aplicações que serão executadas e onde deve-se garantir o tempo de resposta de aplicações interativas. No entanto, *time sharing* nem sempre é uma opção viável devido a problemas como: garantir domínios de proteção às aplicações concorrentes e os custos com o "chaveamento de contexto".

Os algoritmos *space sharing* são utilizados para se obter melhores desempenhos com a utilização exclusiva dos recursos computacionais [Feitelson et al. (1997), Hwang; Xu (1998)]. S*pace sharing* também pode tornar-se inviável se a plataforma utilizada não dispõe dos recursos de software necessários para garantir essa divisão; se a proteção do (sub)conjunto de processadores não pode ser garantida ou se as aplicações executadas necessitam da plataforma computacional como um todo. Harchol-Balter et al. (1997) citam que na computação paralela há a preferência pelo

---

[2] Feitelson; Rudolph (1995) foram os únicos autores, encontrados na literatura, que utilizam o termo *space* e *time slicing* ao invés de *space* e *time sharing*.

uso das políticas *space sharing* em função do alto custo para o chaveamento de contexto nas políticas *time sharing*.

Feitelson; Rudolph (1995) descrevem grupos de políticas pertencentes a cada tipo de compartilhamento (tempo e espaço). Para a classe *space sharing* são descritas as políticas: partição fixa, variável, adaptável e dinâmica (Tabela 2.3). Para a classe *time sharing* são descritas as políticas: fila global e *gang scheduling*.

Tabela 2.3 – Taxonomia das políticas *space sharing* proposta por Feitelson;Rudolph (1995).

	Parâmetros Considerados		
Políticas *Space sharing*	Solicitada pelo Usuário	Carga Computacional	Alterações na Partição
Partição Fixa	não	não	não
Partição Variável	sim	não	não
Partição Adaptável	sim	sim	não
Partição Dinâmica	sim	sim	sim

Space sharing. Na **partição fixa** o número de processadores de cada partição é determinado previamente pelo administrador da plataforma. Somente um "reboot" permite a repartição. Na **partição variável**, também conhecida como *space sharing* pura (*pure space sharing*) ou partição estática (*static partitioning*), o tamanho de cada partição é definido previamente em função das necessidades de cada aplicação. Depois que cada aplicação termina a sua execução a partição é desfeita. Na **partição adaptável** o número de processadores de cada partição é definido em função da utilização da plataforma (carga computacional atual), durante a submissão da aplicação. Depois de feita, a partição não é alterada até o final da execução da aplicação paralela. Na **partição dinâmica** o tamanho de cada partição, além de também considerar a carga computacional na submissão, também permite a alteração durante a execução da aplicação.

A divisão das políticas *space sharing* em quatro classes de políticas não foi encontrada em outros trabalhos (também voltados para a computação paralela). Islam

et al. (1996), Squillante (1995), por exemplo, dividem as políticas *space sharing* em três políticas: partição estática (ou variável), adaptável e dinâmica.

A utilização dessas políticas *space sharing* depende diretamente da habilidade da aplicação paralela em adaptar-se a esses esquemas de divisão em grupos de processadores. Se uma aplicação não é capaz de executar com um número flexível de processadores, não há a necessidade de uma partição dinâmica ou adaptável. Essa "habilidade" pode estar na própria aplicação ou então encapsulada em uma ferramenta de software que forneça-lhe maior flexibilidade e transparência [Souza et al. (2000)].

As desvantagens das políticas *space sharing* dependem de qual política está sendo utilizada. Um dos maiores problemas é a fragmentação dos processadores disponíveis, principalmente com as políticas de partição fixa e variável. A fragmentação ocorre quando o número de processadores disponíveis é insuficiente para executar quaisquer aplicações paralelas. Outro problema é o tempo de espera imposto às aplicações, na tentativa de obter acesso aos processadores solicitados não disponíveis no momento da submissão. Esses problemas agravam-se quando são executadas aplicações com maior processamento e que necessitam de todos os processadores disponíveis [Souza et al. (2000)].

A partição adaptável é uma alternativa viável para as desvantagens apontadas, porque evita que processos "absorvam" muitos processadores quando a plataforma está com alta utilização no momento da submissão. A desvantagem da partição adaptável é não permitir alterações nas partições durante a execução, limitando, assim, seus benefícios quando há uma alta variação das cargas computacionais. A partição dinâmica, além de considerar a carga computacional no momento da submissão, também permite que as partições sejam alteradas durante a execução das

aplicações, refletindo assim as mudanças da carga computacional [Souza et al. (2000)].

Time sharing: a fila global é uma das mais simples políticas de escalonamento *time sharing*, onde uma única fila de execução recebe todas as submissões para a plataforma. Os processadores recebem o primeiro processo da fila, executam-no por um intervalo de tempo e depois devolvem-no à fila. A política fila global é normalmente utilizada em computadores paralelos com memória compartilhada e com pequeno número de processadores. Um dos seus principais problemas é a falta de capacidade escalar, onde o aumento do número de processos e processadores aumenta de maneira considerável a fila global. A maior vantagem da fila global é o compartilhamento de carga "automático", onde nenhum processador fica parado enquanto há algum processo esperando para ser executado.

Tripathi; Karnik (1995) citam que utilizando-se o escalonamento através de uma fila global, há a possibilidade da implementação de algumas variações, tais como: FCFS (*First Come First Served*) e SNPF (*Smallest Number of Process First*) [Souza et al. (2000)]. Para Feitelson;Rudolph (1995) também há a possibilidade de se incluir o objetivo de balancear a carga do sistema, considerando assim a diferença entre as cargas computacionais dos processadores.

Dentre os possíveis problemas com a fila global, a falta de coordenação com o escalonamento dos diferentes processos que pertencem a uma aplicação paralela é um dos mais significativos. Considerando que em muitas aplicações paralelas os seus processos interagem e sincronizam com os demais, a falta de um escalonamento coordenado entre esses processos implica que, enquanto um processo está tentando sincronizar/comunicar com outro, este segundo pode não estar executando. Essa falta de coordenação impedirá que o(a) sincronismo/comunicação seja efetuado(a)

momentaneamente, necessitando para tanto "chaveamentos de contextos" extras e, conseqüentemente, aumento da sobrecarga.

A política de escalonamento *gang scheduling*, criada por J.K. Ousterhout em 1982 [Ousterhout (1982) *apud* Tanenbaum (1995)] é, segundo Feitelson; Rudolph (1995), a maneira mais usual de se implementar escalonamento *time sharing*. Com a *gang scheduling* todos os processos pertencentes a uma aplicação são escalonados e suspensos simultaneamente, de uma maneira coordenada. Sincronizando o escalonamento dos vários processos de uma aplicação, a interação entre os mesmos é otimizada. O escalonamento coordenado permite uma interação melhor entre processos, principalmente naqueles com menor granulosidade. Outra questão é o fato desses processos serem escalonados permanentemente para os mesmos processadores, retirando-se a necessidade de transferência do estado dos processos.

Analisando-se a freqüência com que os processos interagem, a *gang scheduling* pode apresentar algumas variações. *Coscheduling* é uma dessas, na qual são escalonados subconjuntos de processos (os que mais interagem), principalmente se é impossível escalonar todos os processos ao mesmo tempo. *Family scheduling* permite que sejam escalonados mais processos que processadores, utilizando para tanto um segundo nível de *time sharing*. Os termos *hard* e *soft scheduling* também são encontrados na literatura como sinônimos de *gang* e *coscheduling*, respectivamente [Feitelson; Rudolph (1996)]. Os termos *group*, *global* e o próprio *coscheduling* também podem ser encontrados como sinônimos de *gang scheduling* [Saphir et al. (1995), Schnor (1995), Schwiegelshohn (1996), Hwang; Xu (1998)].

As desvantagens da *gang scheduling* ficam por conta: (1) da sobrecarga com o "chaveamento de contexto" coordenado, (2) da possibilidade de interferência com o *cache* de cada processador, devido às trocas constantes de processos e (3) da possível

fragmentação dos processadores, problema considerado de menor importância devido à natureza *time sharing* dessa política.

Vantagens & Desvantagens: a taxonomia é voltada à computação paralela, fato que também ocorre em vários outros trabalhos disponíveis na literatura, como por exemplo: Islam et al. (1996), Krueger et al. (1994), Saphir et al. (1995), Schnor (1995), Schwiegelshohn (1996) e Squillante (1995). Dentro desse contexto (computação paralela) a taxonomia é coerente com os demais artigos encontrados na literatura, usando uma terminologia semelhante. A principal falha, não só desse trabalho mas também dos demais, é a ausência de um relacionamento explícito dessa taxonomia voltada mais para os processadores, com as outras já discutidas neste trabalho, direcionadas mais para os processos.

2.6. Uma Convergência para as Taxonomias

As taxonomias apresentadas neste trabalho demonstram que elas são, em alguns casos, muito divergentes e podem ser divididas em três grupos diferentes. As taxonomias de Casavant; Kuhl (1988), Wang; Morris (1985) e Baumgartner; Wah (1991) classificam os algoritmos de escalonamento sob o ponto de vista dos processos, considerando como os processos têm acesso aos processadores. Assim sendo, essas taxonomias podem ser chamadas de baseadas nos processos.

A taxonomia de Xu; Lau (1997), apesar de também ser baseada em processos, preocupa-se com a distância entre os processadores que farão parte do escalonamento, considerando a topologia utilizada pela plataforma computacional. Portanto, taxonomias como essa podem ser chamadas de baseadas na topologia.

A taxonomia de Feitelson; Rudolph (1995) pode ser denominada de baseada nos processadores, porque divide os algoritmos de escalonamento pela maneira com que os processadores serão disponibilizados aos processos de uma aplicação paralela.

Considerando as diferentes linhas das taxonomias apresentadas, acredita-se que não seja uma tarefa trivial para um leitor (principalmente se iniciante), orientar-se e ainda estabelecer com rapidez e segurança um relacionamento entre dois ou mais algoritmos de escalonamento, classificados através de duas ou mais taxonomias diferentes. Não é uma tarefa trivial analisar uma taxonomia, como por exemplo de Feitelson; Rudolph (1995), relacionando-a com as taxonomias baseadas em processos.

Para exemplificar essa situação, considere um algoritmo de escalonamento "A" classificado segundo as taxonomias baseadas em processos e um algoritmo "B" classificado segundo as taxonomias baseadas em processadores. O algoritmo de escalonamento "A", segundo a sua taxonomia, é global, dinâmico, distribuído, não determinístico e iniciado pelo transmissor. O outro algoritmo de escalonamento, segundo a sua taxonomia, é *space sharing,* partição variável e quando a partição é determinada, usa também a política *gang scheduling.* Com base nas taxonomias apontadas para os dois algoritmos, essas duas propostas de escalonamento são iguais ou são diferentes? A maneira de responder essa questão é analisar um dos algoritmos de escalonamento sob o ponto de vista da outra taxonomia.

Analisando o algoritmo "B", não sob o enfoque dos processadores, mas sim dos processos, tem-se que o algoritmo de escalonamento determina para onde os processos serão atribuídos antes da execução dos mesmos. Ao analisar a definição da partição variável (ou pura): "o tamanho de cada partição é definido previamente em função das necessidades de cada aplicação (depois que cada aplicação termina a sua execução a partição é desfeita)", conclui-se que o algoritmo "B" é estático por determinar o escalonamento somente antes da execução da aplicação. A política *gang scheduling*, que também está presente no algoritmo "B", sincroniza a

preempção entre os processos pertencentes à aplicação paralela. Mesmo com a política *gang scheduling,* o algoritmo "B" continua sendo estático, porque a *gang scheduling* não considera quando e onde os processos serão executados e também não se preocupa com a situação atual da carga de trabalho dos processadores divididos. Apesar de ser preemptiva, a *gang scheduling*, preocupa-se tão somente em sincronizar as preempções locais, feitas em cada processador.

Para estabelecer uma convergência entre as diferentes taxonomias, as seções seguintes determinam uma relação entre as taxonomias baseadas nos processos com as outras duas linhas de taxonomias: baseadas na topologia e nos processadores.

2.6.1. Taxonomias baseadas nos processos x baseadas na topologia

Esses dois grupos de taxonomias são os mais fáceis de serem relacionados, porque as taxonomias baseadas na topologia podem ser consideradas como uma extensão das baseadas nos processos. Conforme citado na taxonomia de Xu; Lau (1997), os algoritmos classificados pelo domínio do escalonamento como locais ou globais são dinâmicos (segundo Casavant; Kuhl (1988)), porque determinam o escalonamento e consideram a carga computacional durante a execução. A diferença está apenas se o processador analisa a sua situação local e/ou no máximo dos seus processadores vizinhos; ou então analisa a situação da carga computacional de toda a plataforma.

O significado dos termos "local e global" nas taxonomias baseadas nos processos é diferente do usado nas taxonomias baseadas na topologia. As taxonomias de Casavant; Kuhl (1988) e de Xu; Lau (1997) exemplificam bem as diferenças.

Considerando as taxonomias baseadas em processos como referência, os algoritmos de escalonamento baseados nos vizinhos mais próximos (ou iterativos ou locais) [Xu; Lau (1997)] são equivalentes aos classificados como distribuídos

[Casavant; Kuhl (1988)]. Isso acontece porque as decisões tomadas para o escalonamento são feitas por vários processadores em tempo de execução, durante os vários "saltos" dos processos sobre os processadores.

Os algoritmos diretos (ou globais) [Xu;Lau (1997)] podem ser tanto distribuídos quanto não distribuídos. Isso depende da maneira com que eles são implementados.

Os algoritmos denominados como iterativos/determinísticos são cooperativos, porque tomam suas decisões considerando a situação de outros processos (mesmo que apenas seus vizinhos) e são também subótimos, porque a cada iteração um novo escalonamento é decidido somente entre os vizinhos (um escalonamento ótimo com essas características tornar-se-ia oneroso). Esses mesmos algoritmos podem ser iniciados pelo transmissor/receptor/simétrico, implementar ou não migração e ser ou não adaptáveis, todas essas características dependentes da implementação.

Os algoritmos denominados como iterativos/estocásticos não são cooperativos, porque determinam seus parceiros (considerando apenas os vizinhos) de maneira arbitrária (aleatória) e sem se preocupar em como está o processador que será parceiro no escalonamento. Esses mesmos algoritmos podem ser iniciados pelo transmissor / receptor / simétricos, implementarem ou não migração de processos, serem ou não adaptáveis, características essas dependentes da implementação.

Muniz (1994) apresenta uma extensão à taxonomia baseada nos processos feita por Casavant; Kuhl (1988), unindo características de taxonomias baseadas na topologia, como a de Lüling et al. (1993). Muniz (1994) é um exemplo de trabalho na área de escalonamento com a preocupação de situar-se dentro de um contexto conhecido na literatura. A Figura 2.8 ilustra a extensão feita, onde as partes sublinhadas e em negrito demonstram as principais alterações. A classificação em globalmente distribuído e localmente distribuído (ou semidistribuído) pode ser

extrapolada para a classificação global (ou direta) e em vizinhos mais próximos (ou iterativas) de Xu; Lau (1997), porque ambas as classificações preocupam-se ou com o uso de todos os processadores ou então de um subconjunto destes, respectivamente.

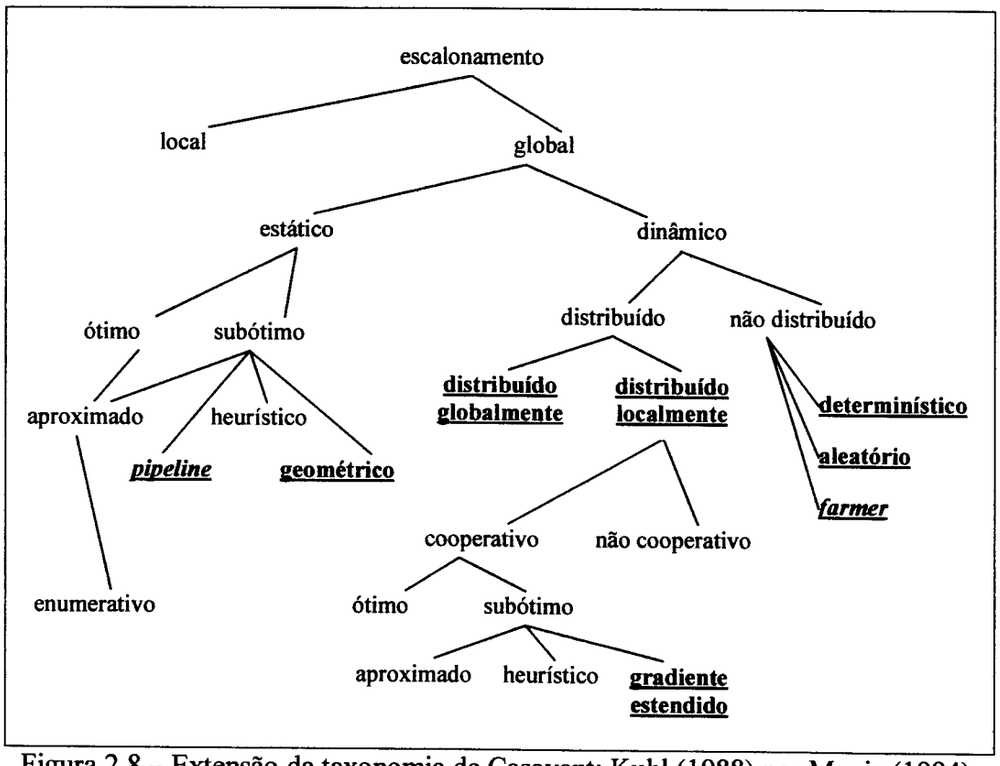


Figura 2.8 – Extensão da taxonomia de Casavant; Kuhl (1988) por Muniz (1994).

A classificação em cooperativos e não cooperativos pode ser extrapolada para a classificação em determinísticos e estocásticos de Xu;Lau (1997). Os determinísticos cooperam entre si para atingirem um objetivo comum, enquanto que os estocásticos não. Finalizando, o modelo gradiente (estendido, no caso) está subordinado à classificação em cooperativos da mesma forma como o modelo gradiente está subordinado à classificação determinística em Xu; Lau (1997).

2.6.2. Taxonomias baseadas nos processos x baseadas nos processadores

Analisando as políticas *space sharing* com partições fixas ou variáveis e mudando o enfoque para os processos que serão escalonados nessas partições,

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

percebe-se que o escalonamento é estático, porque os processadores são determinados antes da aplicação executar, quer seja através de informações fornecidas pelo próprio usuário/administrador, quer seja através de informações fornecidas por uma ferramenta de software que analise o código fonte e determine quais serão os processadores disponíveis à aplicação. Segundo o que está exposto na literatura, durante a execução da aplicação não é feita nenhuma consideração que pondere o destino dos processos. A principal diferença entre as duas políticas fixa e variável é que a primeira determina o número de processadores de maneira arbitrária e sem considerar (nem estaticamente) a aplicação a ser executada, enquanto que a segunda tem condições de pelo menos indicar antes da execução os seus requisitos de hardware.

Considerando as políticas *space sharing* com partições adaptáveis ou dinâmicas (sob o enfoque dos processos que serão escalonados), conclui-se que o escalonamento é dinâmico, em virtude dos processadores serem determinados durante a execução dos processos e também por considerarem a situação atual da carga trabalho da plataforma computacional. As adaptáveis, embora determinem o escalonamento durante a execução, o fazem apenas na submissão e não implementam a migração de processos. Já as dinâmicas, mesmo tendo determinado o subconjunto de processadores que irão servir à aplicação paralela durante a submissão da mesma, podem rever o escalonamento feito, alterar a partição e migrar os processos já escalonados.

Para as políticas *space sharing* com partições adaptáveis ou dinâmicas, questões como por exemplo: distribuído & não distribuído, cooperativo & não cooperativo, ótimo & subótimo e quem é o responsável por iniciar o escalonamento (transmissor/receptor/simétrico) são dependentes da implementação do algoritmo.

A política *time sharing* com fila global é uma política dinâmica [Casavant; Kuhl (1988)], em função da atribuição dos processos ser feita em tempo de execução. Se um processador torna-se ocioso, ele é o responsável (através do algoritmo de escalonamento) por retirar o processo que está na "cabeça" da fila de execução da plataforma, executá-lo por uma quantia de tempo e posteriormente retorná-lo à "cauda" da fila. Apesar desse escalonamento contar com um único ponto de entrada para os processos (a fila global), a decisão por executar processos pode ser feita de maneira distribuída, dando a cada processador receptor a responsabilidade por iniciar o escalonamento. Considerando a responsabilidade por iniciar o escalonamento, esse algoritmo é iniciado pelo receptor. As outras questões como: cooperativo & não cooperativo, implementar ou não migração, ser ou não adaptável ficam por conta da implementação.

As políticas *time sharing* implementadas com *gang scheduling* são mais difíceis de serem relacionadas em outras taxonomias. A *gang scheduling* tem o objetivo de sincronizar o escalonamento dos vários processos pertencentes à aplicação paralela, para melhorar a sincronização entre os mesmos. Se uma política implementa *gang scheduling,* não está determinado se o escalonamento é feito em tempo de execução, quem é o responsável por iniciá-lo, se a decisão é distribuída ou não (caso seja dinâmica), entre outras questões. Assim, pode-se afirmar que a política *gang scheduling* indica uma classe de algoritmos de escalonamento globais, segundo a taxonomia de Casavant; Kuhl (1988), a qual se preocupa com a sincronização da preempção local em cada processador (aquela feita por todo sistema operacional multitarefa/multiusuário). Não obstante, essa preocupação não determina que um algoritmo deva ser dinâmico ou estático. O escalonamento pode ser determinado em

tempo de compilação e a preempção feita em cada processador ser sincronizada pela *gang scheduling* durante a execução.

2.7. Componentes de um Algoritmo de Escalonamento

Há várias maneiras de se construir um algoritmo de escalonamento, dividindo-se o mesmo em módulos, chamados aqui de componentes. Cada componente de um algoritmo de escalonamento é responsável por uma porção da atividade maior que é "escalonar processos" e a combinação desses diferentes componentes é que determinam como o escalonamento será classificado.

Determinar com exatidão quantos e quais são os componentes de um algoritmo de escalonamento não é uma tarefa trivial. Há várias maneiras de se fazer essa divisão, algumas mais modulares (com um maior grau de divisão) e outras menos [Souza et al. (2000)].

Shivaratri et al. (1992) estabelece quatro componentes para o escalonamento de processos, utilizados como referência neste trabalho devido à simplicidade e à modularidade que essa divisão proporciona e também em função da aceitação desses componentes por outros trabalhos como Harchol-Balter; Downey (1997), Shu; Wu (1996) e Tanenbaum (1995).

Shivaratri et al. (1992) citam que um algoritmo de escalonamento é "tipicamente" dividido em: política de transferência, política de seleção, política de localização e política de informação.

A política de transferência indica se um processador encontra-se apto a participar de uma transferência de processos, como transmissor ou como receptor. Quando a política de transferência é baseada em limites, um processador é dito transmissor se a carga de trabalho do mesmo excede um limite T_1 e, por outro lado, é dito receptor se a carga de trabalho é menor que um limite T_2. Os valores de T_1 e T_2

podem ser iguais, dependendo da implementação feita. Quando a política de transferência é relativa, a carga de um processador é comparada com a carga de outros processadores. Com uma política de transferência relativa um processador pode, por exemplo, ser considerado receptor se a sua carga é inferior a algum outro processador em, no mínimo, um valor preestabelecido.

A política de seleção, como o próprio nome indica, é responsável por selecionar os processos que farão parte do escalonamento, uma vez que a política de transferência determinou que o processador é um transmissor. Caso a política de seleção falhe na sua busca, o processador não deve mais ser considerado como transmissor. A seleção dos processos mais novos, aqueles que estão chegando à plataforma e ainda não foram executados, é a maneira mais simples de se implementar a política de seleção. Isso implica na ausência de migração de processos. Processos ativos também podem ser selecionados, necessitando para tanto, que o escalonamento implemente a migração de processos e que avalie a relação custo x benefício dessa transferência de carga. [Souza et al. (2000)].

A política de localização é a responsável por encontrar um parceiro para a transferência dos processos, localizando um processador que tenha sido "rotulado" pela política de transferência ou como transmissor ou como receptor. A política de localização pode ser centralizada ou descentralizada. Um exemplo de descentralizada é aquela que localiza possíveis parceiros através de uma pesquisa (*polling*). Pela pesquisa, um processador "questiona" outros processadores na tentativa de encontrar algum que aceite fazer parte do escalonamento (receber/transmitir processos). Em uma política de localização centralizada há um "coordenador" responsável por localizar um processador apto a tomar parte no escalonamento.

A política de informação tem a responsabilidade de decidir quando, onde e qual informação deve ser coletada sobre a situação da carga de trabalho de cada processador pertencente à plataforma computacional. Shivaratri et al. (1992) destacam três tipos de políticas de informação: sob demanda, periódica e por mudança de estado.

As políticas de informação sob demanda (distribuídas por natureza), obtêm a carga de trabalho dos processadores remotos somente quando o processador local torna-se ou um receptor ou um transmissor. Essas políticas sob demanda podem ser iniciadas pelo transmissor, pelo receptor ou por ambos, de acordo com a política de transferência adotada.

As políticas de informação periódicas, como o próprio nome indica, fazem a atualização das informações pertinentes à carga computacional de tempos em tempos e, dependendo da implementação, podem ser centralizadas ou descentralizadas. Mesmo mais simples, as políticas de informação periódicas, a princípio, não se adaptam à variação da carga na plataforma computacional. Como exemplo, considere uma plataforma com uma alta utilização de seus recursos, onde nenhum processador encontra-se apto a receber processos. Apesar dessa situação, uma política periódica continua atualizando as cargas computacionais, contribuindo apenas para aumentar a sobrecarga e piorar a situação na plataforma.

Nas políticas de informação por mudança de estado, os processadores enviam a situação das suas cargas de trabalho para outros processadores (ou para um coordenador) quando estas mudam em, no mínimo, um valor preestabelecido. As políticas de informação por mudança de estado são diferentes das sob demanda em função de apenas enviarem as suas cargas de trabalho, não requisitando informações de outros processadores.

2.8. Índice de Carga

A qualidade de um escalonamento está relacionada diretamente com a quantidade/natureza das informações obtidas pela política de informação e disponíveis no momento em que as decisões precisam ser tomadas para a atribuição dos processos. Uma dessas informações é a quantidade de carga de trabalho existente na plataforma computacional, indicando a situação atual e/ou de um passado próximo, na tentativa de predizer como será o comportamento futuro das aplicações a serem escalonadas.

Para quantificar o conceito de carga de trabalho em um processador é utilizada uma métrica denominada na literatura por Índice de Carga, que é definida como sendo um valor não negativo, iniciando em zero se o processador está ocioso e aumentando positivamente conforme a carga de trabalho aumenta [Ferrari; Zhou (1987)]. O índice de carga é um item importante, pois está vinculado diretamente ao desempenho obtido com o uso de um algoritmo de escalonamento. Com o índice de carga é possível verificar quanto da potência computacional já está ocupada e com isso fornecer uma expectativa de desempenho futuro à aplicação escalonada.

Tanenbaum (1995) cita que quantificar a carga de trabalho não é uma atividade simples, porque não existe uma única maneira correta para determinar se um processador está carregado, moderado ou ocioso. Ferrari; Zhou (1987) citam que devido ao fato da demanda por recursos variar de aplicação para aplicação, o uso do termo genérico "a carga de um processador" pode não ser representativo. Para exemplificar, os autores citam o caso onde um processador pode estar "sobrecarregado" enquanto o acesso aos discos não está. Nessa situação, para processos *CPU-Bound* a carga do processador deve ser considerada alta, enquanto para processos *I/O-Bound* a carga do processador deve ser considerada baixa.

Para determinar a qualidade de um índice de carga, deve-se estar ciente de quais aplicações serão executadas e também quais são os objetivos esperados para o escalonamento. Ferrari; Zhou (1987) citam que, de uma maneira geral, a escolha do índice de carga deve considerar: (1) a capacidade de representar com qualidade a carga atual de um host; (2) a capacidade de predizer a carga em um futuro próximo; (3) a estabilidade do índice, o qual deve ignorar ao máximo flutuações de carga (*noise*) e (4) a relação com a métrica que avaliará o desempenho do escalonamento feito, para facilitar análises de expectativas de desempenho a ser alcançado com o uso do escalonamento.

Outra questão relacionada à qualidade de um índice de carga é a sobrecarga gerada para a sua obtenção. Um índice de carga, além de ser eficiente na representação da carga de trabalho, deve agregar a menor sobrecarga possível ao escalonamento, tendo-se em mente sempre a melhor relação custo x benefício. A maneira como o índice de carga é formado, é decisiva na sobrecarga gerada. Os índices de carga mais específicos, aqueles formados por apenas um valor obtido em cada processador, tendem a apresentar uma sobrecarga menor, além de serem mais eficientes em situações específicas. Esses devem ser utilizados corretamente em determinados tipos de aplicações e também de acordo com os objetivos esperados para o escalonamento. Exemplos de índices específicos são: tamanho da fila de execução ou tamanho da fila de acesso à memória. Os índices de carga genéricos são aqueles formados pela união de mais de um valor obtido nos processadores. Esses índices são mais indicados quando não se possui um conhecimento maior sobre as aplicações que serão executadas e nem sobre os objetivos esperados para o escalonamento. Além de apresentarem uma tendência à sobrecarga maior, os índices genéricos podem não apresentar a mesma qualidade de representação da carga de

trabalho, quando comparados com os índices específicos utilizados corretamente [Mehra; Wah (1993), Ferrari; Zhou (1987)]. Um exemplo de índice de carga genérico é a união do tamanho da fila de execução com o tamanho da fila de acesso à memória. Os índices de carga também podem ser divididos em outros grupos como: baseados no tamanho da fila de acesso ao recurso, baseados no percentual de utilização do recurso e os baseados no tempo de execução/resposta [Ferrari; Zhou (1987), Shirazi et al. (1995a)].

Os índices acima podem representar valores instantâneos (o valor exato da última obtenção) ou então representar uma média das últimas obtenções. A utilização da média ajuda a reduzir o risco do valor obtido estar representando apenas uma flutuação instantânea da carga de trabalho.

Se a plataforma utilizada é heterogênea e multiusuária, a importância do índice de carga é ainda maior. Além das alterações feitas pela própria aplicação, a presença de outros usuários e as diferenças entre os processadores influenciam decisivamente o desempenho esperado para a aplicação. As cargas de trabalho externas à aplicação são chamadas por alguns autores como *background workload* ou *external load* [Mehra; Wah (1993), Krone et al. (1998), Russ et al. (1997)] e devem ser consideradas pelo índice utilizado. As diferentes potências computacionais existentes nas plataformas distribuídas e heterogêneas, agregam ao índice de carga a tarefa de normalizar as cargas em relação às potências em cada processador. Essa normalização não é uma tarefa trivial, porque dependendo da aplicação utilizada o impacto da heterogeneidade será maior ou menor [Souza et al. (2000)].

O índice de carga utilizado pelo escalonamento de processos é um item que por si só já constitui uma linha de pesquisa. Em função disso não serão discutidos outros fatores pertencentes aos índices de cargas. Estes podem ser encontrados em: Ferrari;

Zhou (1987), Kunz (1991), Shivaratri et al.(1992), Mehra; Wah (1993), Milojicic et al. (1993), Kaplan; Nelson (1994), Muniz (1994), Shirazi et al. (1995b), Dantas (1996), Dantas; Zaluska (1998), Ferstl (1996), Schnor et al. (1996), Tanenbaum (1995), Russ et al. (1997), Krone et al. (1998), entre outros.

2.9. Desempenho Alcançado com o Algoritmo de Escalonamento

Um algoritmo de escalonamento, além de permitir a execução de processos pelos processadores (objetivo principal), também deve maximizar o desempenho esperado para a aplicação executada e/ou para a plataforma computacional como um todo. Esse desempenho que se tenta maximizar não é único e não pode ser quantificado uniformemente para todas as possíveis situações. Para determinar se o algoritmo de escalonamento está desempenhando corretamente as suas funções, deve-se saber quais são os objetivos propostos para o algoritmo e também estabelecer uma métrica capaz de avaliar o desempenho em função desses objetivos.

2.9.1. Objetivos e Métricas de Desempenho

A literatura da área demonstra que os algoritmos de escalonamento se propõem a uma grande variedade de objetivos. Exemplos destes são: compartilhamento de carga, aumento da utilização do processador, redução do tempo de resposta, redução do tempo de execução, aumento do *throughput* da plataforma, questões administrativas, imparcialidade na execução dos processos, entre outros [Wang; Morris (1985), Kunz (1991), Shivaratri et al. (1992), Mullender (1993), Kaplan; Nelson (1994), Harchol-Balter et al. (1997), Feitelson; Rudolph (1996), Franklin; Govidan (1996), Kowok; Ahmad (1996), Robinson et al. (1996), Tanenbaum (1995), Billard; Pasquale (1997), Xu; Lau (1997), Rapine et al. (1998)].

Alguns desses objetivos são conseqüência de outros, como por exemplo quando tenta-se reduzir o tempo de execução, normalmente reduz-se também o tempo de resposta. Outros objetivos podem ser contraditórios, onde a opção por um objetivo inviabiliza o alcance de outros. Um objetivo mutuamente exclusivo é o aumento da utilização do processador, onde escalonam-se processos "maiores", aumentando-se conseqüentemente o tempo de resposta das aplicações.

Alguns autores como Feitelson; Rudolph (1996) classificam os objetivos como coletivos e individuais. Os objetivos coletivos enfatizam as preferências administrativas, vendo as necessidades dos seus usuários como um todo. O aumento do *throughput* da plataforma é um objetivo coletivo. Os objetivos individuais dão ênfase às necessidades de um usuário em especial, em detrimento às necessidades coletivas. A redução do tempo de execução de uma aplicação é um objetivo individual.

As métricas de desempenho formalizam os objetivos propostos para o escalonamento e permitem avaliar se esses objetivos estão ou não sendo atingidos. Alguns objetivos são difíceis de se quantificar, por fazerem referência a questões mais subjetivas como as administrativas [Ferstl (1996)].

Alguns exemplos de métricas são: tempo de execução, *throughput*, tempo de resposta, percentual de utilização do processador, tempo normalizado ou *slowdown* (tempo carregado / tempo dedicado), tempo de espera para execução [Ferrari; Zhou (1987), Mehra; Wah (1993), Krueger et al. (1994), Feitelson;Rudolph (1996), Loh et al. (1996)].

Feitelson et al. (1997) destacam também a necessidade de se flexibilizar as métricas que avaliam o desempenho em função de determinados fatores, como por exemplo o horário de utilização e a demanda pelos recursos. Em horários "não

convencionais", como madrugadas e finais de semana, pode-se dar prioridade às aplicações que requeiram maior potência computacional e não apresentem alta interatividade com os usuários.

2.9.2. Fatores que Influenciam o Desempenho

Mesmo fixando um objetivo, um algoritmo de escalonamento pode apresentar desempenhos distintos ou até mesmo negativo. Afirmar que exista um único algoritmo de escalonamento capaz de produzir excelentes resultados em todas as possíveis combinações de hardware, software e questões administrativas é uma tarefa extremamente difícil.

Essa variação no desempenho é conseqüência principalmente de três fatores: (1) a maneira como o algoritmo de escalonamento é construído, (2) a plataforma computacional utilizada e (3) a classe (ou natureza) da aplicação executada.

O **algoritmo de escalonamento** é um fator decisivo no desempenho atingido. Além dos benefícios advindos com um escalonamento melhor elaborado, deve-se considerar também os custos necessários para a sua obtenção. Algumas vezes, um algoritmo simples ou específico possui um ganho de desempenho melhor por apresentar pouca sobrecarga, quando comparado a um algoritmo mais complexo e/ou genérico e que possui uma sobrecarga elevada [Wang; Morris (1985), Casavant; Kuhl (1988), Ferrari; Zhou (1987), Franklin; Govidan (1996), Tanenbaum (1995)].

Além da sobrecarga de um algoritmo de escalonamento, há também a questão da estabilidade e da eficiência do mesmo. Shivaratri et al. (1992), Krueger; Shivaratri (1994), Tanenbaum (1995) e Corradi et al. (1997) citam que um algoritmo de escalonamento é instável se ele piora a situação de carga de uma plataforma, realizando execuções inúteis indefinidamente que irão contribuir apenas para agravar ainda mais a demanda pelos recursos. Um algoritmo eficiente, além de ser estável,

também melhora o desempenho da plataforma, quando comparado à execução das aplicações sem o algoritmo de escalonamento [Souza et al. (2000)].

A **plataforma computacional** utilizada é decisiva no desempenho assim como na escolha do algoritmo de escalonamento. Características como organização da memória principal, topologia, velocidade de transmissão pela rede de comunicação e confiabilidade afetam principalmente as atividades que necessitam mais comunicação [Loh et al. (1996)].

Há várias plataformas propostas, com os mais variados esquemas de conexão entre processadores. Considerando como exemplo uma plataforma genérica como uma *NOW*, a simples troca da rede de conexão por outra com maior capacidade de transmissão, altera o desempenho alcançado pelos algoritmos de escalonamento. O custo para a troca de informações ou o envio do estado de algum processo em execução será menor.

A heterogeneidade das plataformas distribuídas atuais é decisiva para o escalonamento dos processos. Além de determinar a carga atual de um elemento de processamento, deve-se também normalizar essa carga em função das diferenças encontradas nesses elementos. O impacto causado pela heterogeneidade depende também da classe de aplicação que está sendo executada.

A heterogeneidade pode ser configuracional ou arquitetural. A primeira ocorre quando computadores de mesma arquitetura apresentam: diferentes velocidades de processamento, diferentes quantidades de memória disponível, entre outras. A heterogeneidade arquitetural ocorre quando as diferenças estão na própria arquitetura dos elementos de processamento utilizados, como por exemplo processadores CISC, RISC e paralelos [Mehra; Wah (1993) e Schnor et al. (1996)].

A presença de vários usuários simultâneos em uma plataforma também altera o desempenho dos algoritmos de escalonamento. Quando as plataformas distribuídas são "monopolizadas" por apenas um usuário é possível antever com mais facilidade como será o comportamento. Isso acontece porque as cargas de trabalho, conhecidas como "internas" [Krone et al. (1998)], estão relacionadas exclusivamente aos processos em execução dessa aplicação. As plataformas multiusuárias agregam ao escalonamento a preocupação de considerar as variações na carga computacional "externas" à aplicação [Krone et al. (1998)]. Assim, um escalonamento correto, realizado no início da execução de um processo pode se tornar inadequado em tempo de execução. Uma solução para esse problema é o emprego da migração de processos [Ferstl (1996)].

O comportamento de um algoritmo de escalonamento também varia em função da carga computacional que a plataforma está executando. Com altas cargas de trabalho (pior caso), alguns algoritmos de escalonamento tornam a plataforma computacional instável, contribuindo para uma degradação ainda maior do desempenho. Em outras situações, com cargas de trabalho moderada a baixa, essa situação pode não se repetir e, ao contrário, apresentar ganhos reais de desempenho [Shivaratri et al. (1992)].

A **classe da aplicação** ou **de software** é decisiva para determinar o desempenho de um escalonamento. O mesmo algoritmo, quando utilizado por um determinado tipo de aplicação, apresenta um "ganho" de desempenho que não se repete necessariamente com outro tipo de aplicação [Souza et al. (2000)]. A caracterização da carga de trabalho é uma atividade necessária para uma distribuição dos processos entre os processadores com maior qualidade [Ferstl (1996), Feitelson et al. (1997)]. O fato de diferentes usuários submeterem de maneira concorrente seus processos,

com diferentes necessidades, dificulta essa situação. Essas aplicações compartilham os mesmos recursos de processamento e afetam o desempenho delas mesmas e de outras aplicações em execução.

Para determinar a plataforma computacional usada e a natureza da aplicação que será executada, o algoritmo de escalonamento necessita de informações adicionais, além daquelas obtidas durante a execução (no caso de algoritmos de escalonamento dinâmicos). A quantidade e qualidade das informações (considerando sempre os custos para obtê-las), afeta proporcionalmente a qualidade do escalonamento realizado [Feitelson et al. (1997)].

2.10. Escalonamento: a Teoria e a Prática

Facilmente são encontrados vários trabalhos descrevendo alguma pesquisa relacionada à área de escalonamento de processos. Entretanto, o conhecimento gerado através dessas pesquisas não é empregado com a mesma facilidade nos algoritmos de escalonamento utilizados na prática [Milojicic et al. (1993), Squillante (1995), Feitelson; Rudolph (1996), Islam et al. (1996), Paindaveine; Milojicic (1996), Schnor et al. (1996), Feitelson et al. (1997), Gehring (1997), Gibbons (1997)]. Essa lacuna existente entre a teoria e a prática na área de escalonamento de processos é conseqüência de alguns fatores, discutidos a seguir.

2.10.1. Fatores que Afastam a Teoria da Prática

A **complexidade da área** de escalonamento de processos, onde a escolha pelo melhor escalonamento depende de inúmeras questões já discutidas. Não é fácil para um usuário ter que se preocupar com todas as questões pertinentes à área. A falta de transparência, não só para o escalonamento em si, mas também para a escolha do escalonamento, dificulta a utilização das melhores tecnologias.

As **divergências na nomenclatura** dificultam o entendimento dos diferentes tipos de escalonamento existentes. A literatura da área é vasta, mas imprecisa e não fornece uma visão mais abrangente. O estudo da área de escalonamento, com a intenção de escolher a política mais apropriada para uma situação, não é uma tarefa fácil, desencorajando a aplicação na prática do que está sendo descoberto nas pesquisas.

A **visão algorítmica** das propostas de escalonamento fixam no algoritmo a política de escalonamento, determinando assim a maneira como o escalonamento será realizado. Isso não permite flexibilizar as decisões tomadas, restringindo a atuação do escalonamento. Considerando a dificuldade de se desenvolver um algoritmo apto a resolver da melhor maneira possível todos os casos, os usuários ficam limitados a escalonamentos que até podem apresentar ganhos de desempenho, porém, podem não ser a melhor solução para o problema. Questões administrativas e/ou novas situações são fatos que podem surgir no decorrer do tempo e alteram os objetivos iniciais. Embora os algoritmos adaptáveis minimizem essa questão, inserindo certa flexibilidade ao escalonamento, esta continua limitada ao algoritmo.

A **complexidade dos algoritmos** de escalonamento "melhor elaborados", como por exemplo, com migração de processos ou que flexibilizam as políticas de escalonamento, muitas vezes inviabiliza a utilização dos mesmos na prática. Muitos produtos comerciais optam pela simplicidade e pelo menor custo, implementando algoritmos que apresentam algum ganho de desempenho para casos genéricos, mas não se preocupam com a qualidade dessa solução.

As **métricas de desempenho** devem ser tão flexíveis quanto o algoritmo de escalonamento que as mesmas avaliam. Métricas como a redução do tempo de execução, embora sejam muito utilizadas, não são as únicas. As métricas devem ser

determinadas em conjunto com os objetivos do escalonamento e toda vez que estes variarem, as mesmas devem ser reavaliadas.

2.10.2. Fatores que Convergem a Teoria e a Prática

Alguns autores destacam as características desejáveis de um algoritmo de escalonamento. Essas características, de certa forma, vêm ao encontro da necessidade de convergência entre a teoria e a prática na área de escalonamento.

Feitelson; Rudolph (1996) citam que a convergência é dependente dos seguintes pontos: (1) os trabalhos devem ser explícitos sobre as suas características, usando para tanto uma terminologia consistente e assim diminuindo a dificuldade em se relacionar trabalhos já desenvolvidos; (2) determinar e minimizar as deficiências de cada proposta, antes de construir novas; (3) combinar diferentes propostas, flexibilizando os algoritmos existentes; (4) usar métricas (funções de custo) que reflitam as características e objetivos da organização, permitindo um monitoramento correto sobre os ganhos obtidos com o escalonamento.

Além dos pontos destacados acima, a lacuna em questão pode ser diminuída fornecendo aos usuários transparência completa em relação à atividade de escalonamento. Por transparência completa entende-se não só a transparência em se realizar um escalonamento, mas também em se escolher o melhor escalonamento a ser utilizado. Através do auxílio de uma ferramenta de software é possível, transparentemente, automatizar o processo de escolha do melhor algoritmo de escalonamento para uma plataforma computacional e também para situações específicas com a mesma plataforma. Automatizando-se a escolha correta do algoritmo de escalonamento retira-se a visão algorítmica da área e não se sobrecarrega um único algoritmo. Dessa maneira, vários algoritmos específicos

podem ser utilizados no lugar de um genérico que apresenta maiores custos de execução.

2.11. Considerações Finais

Estudar a área de escalonamento de processos não é uma tarefa trivial. Não encontrou-se na literatura um trabalho que fornecesse uma visão abrangente da área, comparando as diferentes linhas de pesquisa descritas aqui, através das suas similaridades e diferenças. Este trabalho, além de fornecer uma revisão bibliográfica, teve por objetivo amenizar esse problema da literatura, fornecendo uma leitura abrangente e de mais alto nível, não descendo aos pormenores das implementações.

Além de fornecer um contexto mais abrangente para a área, este trabalho também destaca a importância de se organizar de maneira modular e coerente o código de um algoritmo de escalonamento, através dos seus componentes. Não foram descritos os inúmeros algoritmos de escalonamento desenvolvidos e disponíveis na literatura. Essas implementações são encontradas facilmente em vários trabalhos espalhados na literatura. O Apêndice A relaciona as referências bibliográficas encontradas de alguns desses trabalhos que descrevem novas propostas de algoritmos de escalonamento.

O desempenho de um algoritmo de escalonamento é altamente influenciável por inúmeros fatores como: objetivos da plataforma usada (computação paralela ou sistema distribuído), a própria plataforma (topologia, heterogeneidade, multiusuários, etc.), as classes de aplicação executadas, os horários de utilização, entre outros. Considerando-se todas essas possíveis variações, torna-se complexo afirmar que exista um algoritmo de escalonamento com desempenho favorável e constante para todos os casos.

A pouca utilização das pesquisas desenvolvidas nas propostas de escalonamento usadas comercialmente pode ser considerada como uma conseqüência de quatro fatores: (1) os problemas existentes na literatura; (2) a grande variação dos algoritmos de escalonamento; (3) a visão algorítmica das propostas de escalonamento; e (4) a falta de transparência para determinar qual política de escalonamento usar.

O escalonamento é tratado segundo a política e o mecanismo inseridos no algoritmo, fato que limita uma maior flexibilidade no escalonamento dos processos, mesmo que os algoritmos sejam adaptáveis. Para fornecer flexibilidade, continuar usando algoritmos específicos e com menores custos de processamento e ainda permitir transparência total para os usuários de aplicações concorrentes e/ou distribuídas, deve-se utilizar uma ferramenta de software que encapsule todas as questões discutidas aqui e, assim, viabilize a redução da lacuna existente entre a teoria e a prática na área.

Com o intuito de minimizar os problemas da área destacados aqui, esta tese tem como objetivo principal definir o AMIGO (*DynAMical FlexIble SchedulinG EnvirOnment*), descrito de maneira detalhada no próximo capítulo. O AMIGO é uma ferramenta de software aberta, responsável pela gerência do escalonamento de processos, de maneira flexível e transparente ao usuário [Souza et al. (1999a, 1999b, 1999c)]. Através dessa ferramenta de software diversos algoritmos de escalonamento podem ser reunidos sendo que cada um é específico a um conjunto de fatores. Isso permite ao usuário utilizar os algoritmos de escalonamento mais apropriados às suas necessidades, quer sejam técnicas ou administrativas.

3. AMIGO – Um Ambiente de Escalonamento Flexível e Dinâmico

3.1. Considerações Iniciais

O AMIGO (*DynAMical FlexIble SchedulinG EnvirOnment*) é uma ferramenta de software aberta, responsável pela gerência do escalonamento de processos em plataformas computacionais distribuídas. O projeto dessa ferramenta encapsula os diversos fatores discutidos no capítulo anterior, os quais influenciam o desempenho de um escalonamento, inserindo concomitantemente flexibilidade e transparência à atividade de escalonar processos [Souza et al. (1999a) (1999b) (1999c)].

A meta é reduzir a lacuna existente entre a teoria e a prática na área, permitindo que ambas sejam unidas de uma maneira viável e coerente. O AMIGO pode reunir diversos algoritmos de escalonamento, sendo cada um específico a um determinado conjunto de fatores. Isso permite ao usuário utilizar os algoritmos de escalonamento mais apropriados às suas necessidades (técnicas ou administrativas).

O AMIGO não é um algoritmo de escalonamento adaptável (com diferentes políticas implementadas). É uma ferramenta que atua na gerência do escalonamento a ser empregado estando, dessa maneira, em um nível superior ao de um algoritmo de escalonamento. Por ser uma ferramenta de software aberta, possui uma interface clara e objetiva, necessária para que quaisquer políticas de escalonamento sejam

implementadas em diferentes algoritmos e executadas futuramente sob a sua responsabilidade. Isso retira a visão algorítmica constatada anteriormente e flexibiliza a maneira como os processos são escalonados [Souza et al. (1999b)].

Considerando o grande número de fatores que influenciam na escolha das políticas e as divergências encontradas na literatura, o objetivo é flexibilizar não só o algoritmo de escalonamento em si, mas também tudo aquilo que pode influenciá-lo ou está relacionado com ele. Assim, permite-se que o usuário utilize a taxonomia que lhe é usual, as classes de aplicação (ou de software) que são necessárias, a plataforma a ser usada para execução, os *benchmarks* que determinam as heterogeneidades na plataforma, as métricas para avaliar o desempenho obtido com o escalonamento, entre outros.

A transparência oferecida ao usuário é uma das maiores preocupações do AMIGO, porque flexibilidade sem transparência atribui ao usuário final a responsabilidade pela escolha da melhor política de escalonamento a ser empregada. Conforme já discutido anteriormente, escolher corretamente as melhores políticas de escalonamento não é uma tarefa trivial.

O termo Dinâmico existente no nome do AMIGO refere-se à capacidade de mudar completamente a política e/ou também o mecanismo de escalonamento adotados, em tempo de execução da aplicação. O termo Dinâmico aqui não se refere a um "algoritmo de escalonamento dinâmico" e sim à capacidade do AMIGO de ser flexível em tempo de execução. A intenção é retirar a visão algorítmica e permitir uma completa maleabilidade na atividade de se escalonar processos [Souza et al. (1999a)].

Com o AMIGO, além de se determinar qual é a melhor política de escalonamento para uma determinada situação, também é possível utilizar várias

políticas ativas ao mesmo tempo. Isso é possível graças, principalmente, a duas características: a modularidade existente na sua estrutura e a flexibilidade nas classes de software, as quais fornecem informações sobre as aplicações que serão escalonadas. Com a existência de várias políticas ativas, diferentes aplicações podem estar executando e utilizando, simultaneamente, diferentes algoritmos de escalonamento. O relacionamento entre as classes de software e as políticas de escalonamento serão detalhadas no decorrer deste capítulo.

As mudanças feitas nas políticas são viabilizadas graças a um monitoramento realizado por métricas também inseridas no AMIGO. Devido à flexibilidade na escolha da métrica para a análise de desempenho dos algoritmos escolhidos, pode-se avaliar corretamente o escalonamento para cada objetivo e situação em que ele se encontrar.

3.2. Plataformas de Atuação do AMIGO

Inicialmente, tem-se por objetivo abranger aplicações paralelas desenvolvidas para plataformas distribuídas, heterogêneas ou não, e que utilizem o sistema operacional UNIX (genericamente, uma máquina MIMD com memória distribuída). A escolha dessa plataforma deve-se à sua grande utilização, à sua característica multiusuária (o que torna a atividade de escalonamento ainda mais importante) e pela facilidade de se encontrar os recursos de software necessários, os quais viabilizam a construção do algoritmo de escalonamento. Esses recursos de software normalmente são encontrados em linguagens de programação como a linguagem C e/ou no sistema operacional UNIX [Anderson et al. (1995), Saphir et al. (1995), Ferstl (1996)].

Para a construção das aplicações paralelas, estão sendo considerados os ambientes de passagem de mensagem como o PVM (*Parallel Virtual Machine*),

versão 3.3.11, e o MPI (*Message Passing Interface*) - implementação LAM versão 6.3 [Geist et al. (1994), Dongarra et al. (1996), Geist et al. (1997), Souza, M.A. (1996), Dongarra et al. (1998)]. A escolha por esses dois ambientes de passagem de mensagem deve-se a algumas características presentes em ambos, tais como: serem softwares de domínio público, consistentes e de grande utilização/aceitação pela comunidade da computação paralela [Geist et al. (1994), Vesseur et al. (1995)]. Outra vantagem existente na utilização do PVM e do MPI é o fato de ambos possuírem toda uma estrutura que fornece suporte à aplicação paralela. Essa estrutura compreende atividades como o agrupamento lógico de processadores que formam a máquina paralela virtual, regras simples para o escalonamento de processos para esses processadores (como *round-robin*), controle da comunicação realizada entre os processos, entre outras atividades. Dentro desse contexto, o AMIGO substitui a política de escalonamento "original" por outra, determinada antes ou até mesmo durante a execução das aplicações. [Souza et al. (1999b)].

Embora o AMIGO tenha sido projetado para ser executado inicialmente com os ambientes de passagem de mensagem, isto não impede que o mesmo venha a ser utilizado por intermédio de outros softwares, como por exemplo um compilador (Fortran) ou até mesmo um sistema operacional distribuído. Isso é possível graças à divisão da estrutura do AMIGO em duas camadas: uma superior e outra inferior. Adaptando-se a camada inferior para a plataforma e/ou software desejados, é possível utilizá-lo em outras situações que não as previstas inicialmente.

3.3. O perfil dos usuários do AMIGO

Para a construção de um ambiente de escalonamento, com os objetivos aqui propostos, deve-se considerar o perfil do usuário que irá operá-lo. Existem no

mínimo duas possibilidades: o usuário de uma aplicação, que deseja executá-la de modo mais eficiente, ou então o gerente dos recursos computacionais que supervisiona a execução de várias aplicações na plataforma sob sua responsabilidade [Souza et al. (1999b)].

Independentemente de quem seja essa pessoa, sempre haverá vantagens e desvantagens na escolha feita. Caso o usuário do AMIGO seja o mesmo de uma aplicação paralela, esse terá maior facilidade em determinar a política e o mecanismo mais apropriados para a sua aplicação, porque terá o controle total do escalonamento feito para essa aplicação. Porém, devido à 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 seja o gerente da plataforma, o escalonamento deve ser orientado por uma visão mais democrática dos recursos disponíveis, isto porque um gerente não se preocupa somente com a satisfação de um usuário, mas também com a satisfação coletiva. Essa visão pode dificultar algumas atividades como, por exemplo, medições de desempenho, onde uma aplicação necessita de um controle total da plataforma utilizada.

Para usar o AMIGO, é desejável que o usuário tenha consciência da importância de se realizar um escalonamento de processos otimizado, não precisando, obrigatoriamente, saber como essa tarefa é realizada. Graças à garantia de transparência, um usuário pode executar a sua aplicação e não ter conhecimento da existência do AMIGO. No entanto, para configurar algumas características como: hardware, classes de aplicação, algoritmos de escalonamento disponíveis e métricas para avaliação de desempenho, é necessária uma pessoa com um conhecimento maior na área. Essas configurações, embora necessárias na configuração inicial, são

aproveitadas nas execuções posteriores, tornando-as completamente transparentes para o usuário da aplicação paralela. No caso de um usuário não poder, ou não conseguir, configurar o AMIGO por falta de conhecimento, este tomará as suas decisões baseando-se em valores padrões e/ou heurísticas preestabelecidas, viabilizando, mesmo assim, o escalonamento.

O principal objetivo nesse ponto, é privilegiar os usuários que tenham um conhecimento maior sobre a área de escalonamento e ao mesmo tempo ampará-los na medida do possível, se tiverem pouco ou nenhum conhecimento.

3.4. A Estrutura do AMIGO

O AMIGO é composto por duas camadas principais: uma superior e outra inferior (Figura 3.1). Essa divisão na sua estrutura interna permite que a atividade de escalonar processos, feita na camada inferior, seja independente da configuração e do monitoramento, ambos feitos na camada superior. Essa divisão também permite uma maior modularidade ao ambiente, assim como facilita a portabilidade do mesmo para outras plataformas. Apenas a camada inferior é mais dependente da plataforma que está sendo usada. A camada superior, embora também seja dependente de alguns fatores como linguagem de programação e sistema operacional, pode ser executada facilmente em outras plataformas que não a usada inicialmente. Já a camada inferior, utiliza todas as primitivas necessárias para a comunicação, obtenção da carga computacional, execução de novos processos, entre outras. Essas atividades estão fortemente relacionadas à plataforma utilizada e precisam ser adaptadas a novas plataformas [Souza et al. (1999a) (1999c)].

Outra vantagem da divisão feita é que para utilizar o AMIGO, uma vez configurado o escalonamento, não é necessário executar a camada superior do

ambiente. O usuário apenas executa a sua aplicação com o PVM ou com o MPI, e o escalonamento correto será empregado pela camada inferior de maneira transparente ao usuário e independente da camada superior.

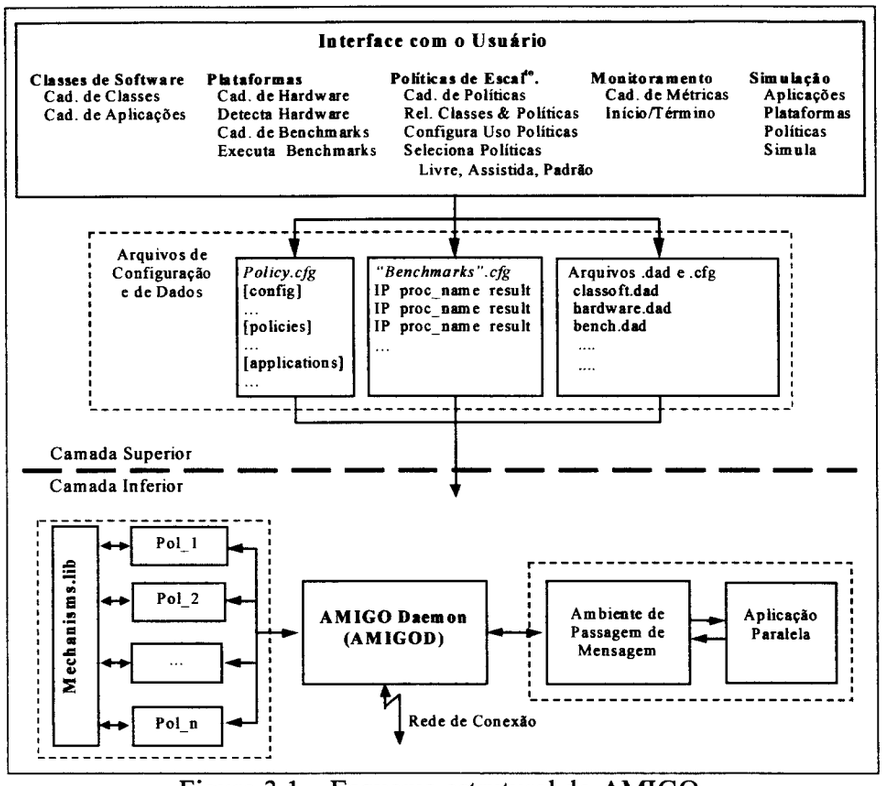


Figura 3.1 – Esquema estrutural do AMIGO.

A camada superior é composta basicamente por uma interface gráfica, pela qual o usuário tem acesso às várias opções do ambiente, como por exemplo: inserir as classes de software (grupos de aplicação a serem escalonados), inserir o hardware utilizado, inserir *benchmarks* usados para determinar a heterogeneidade existente na plataforma, inserir as métricas utilizadas para monitorar o escalonamento feito e, finalizando, inserir e determinar as políticas de escalonamento. Na camada inferior encontram-se os algoritmos disponíveis para a realização do escalonamento, assim como a interface necessária ao relacionamento com o ambiente de passagem de mensagem utilizado.

3.5. A Camada Superior do AMIGO

Antes de iniciar a discussão sobre os detalhes pertinentes à camada superior, cabe ressaltar que esta não foi implementada até a conclusão desta tese. As informações descritas nesta seção apontam diretrizes a serem utilizadas durante uma futura implementação desta porção do AMIGO.

A camada superior é responsável pela gerência da flexibilidade inserida no AMIGO [Souza et al. (1999a)]. Essa gerência é realizada em vários itens, como por exemplo: classes de software, hardware, *benchmarks*, políticas de escalonamento, e é viabilizada através de cadastros independentes. Os cadastros permitem a inserção de diferentes fatores, além de também possibilitarem o uso de diferentes terminologias, linhas de pensamentos e objetivos, conforme foi descrito no capítulo anterior.

Embora a inserção desses fatores seja fundamental aos objetivos propostos, a característica mais importante da camada superior não é a inserção, e sim o relacionamento entre esses fatores cadastrados. Através desse relacionamento, são formados conjuntos lógicos (ou virtuais) coesos, os quais direcionam as políticas de escalonamento para, por exemplo, determinadas classes de aplicação, tipos de plataformas computacionais, horários de execução, entre outros.

Para complementar a atividade de escalonar processos, a camada superior também é projetada para fornecer o suporte necessário a atividades que não aquelas pertinentes à escolha do algoritmo de escalonamento em si. Dentre essas atividades encontram-se: o monitoramento do desempenho obtido com o emprego de uma determinada política de escalonamento, a simulação do desempenho a ser obtido com o emprego de uma política de escalonamento, a análise das diferenças de potência computacional através de *benchmarks*, entre outras.

O resultado das atividades realizadas na camada superior é armazenado e posteriormente utilizado pela camada inferior através de arquivos de configuração. Esses arquivos de configuração estão denominados e detalhados nas seções seguintes, que descrevem as principais características da camada superior.

3.5.1. Classes de Software

O AMIGO permite a inclusão de classes de software (tipos de aplicações afins) e também das próprias aplicações a serem escalonadas, sendo que ambas serão posteriormente vinculadas às políticas de escalonamento. O cadastro de classes de software tem por objetivo flexibilizar os tipos de aplicação que serão escalonados. Dessa maneira, não há classes de software fixas e pré-definidas no ambiente, como por exemplo: *CPU-Bound* x *I/O-Bound* e interativa x *batch*; mas é viabilizada de maneira prática e objetiva a inclusão de qualquer classe de software que venha a ser necessária [Souza et al. (1999a) (1999b)].

A inclusão de classes de software vem ao encontro da questão do custo para a obtenção da informação sobre a aplicação a ser escalonada. Nem sempre é possível (viável) que informações detalhadas sobre a aplicação sejam obtidas antes ou durante o escalonamento. Dessa maneira, as classes de software amenizam esse custo e viabilizam a obtenção das informações. As classes não se preocupam com uma aplicação em especial e sim com um grupo afim de aplicações.

Para relacionar as aplicações escalonadas com as classes de software disponíveis, optou-se por um cadastro de aplicações, vinculadas às classes de software. Com o cadastro de aplicações, o vínculo destas com as classes pôde ser feito de maneira transparente à aplicação. Não é necessário alterar o código fonte da mesma para a inclusão de parâmetros que indiquem a qual classe essa aplicação pertence.

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

A inclusão de aplicações vinculadas diretamente às classes de software não é obrigatória. Caso uma aplicação não cadastrada venha a ser escalonada, a mesma irá utilizar o escalonamento padrão determinado para o momento da execução, independente da classe de software a que a mesma pertença. Essa política de atuação privilegia o escalonamento feito com maiores informações, permitindo que políticas de escalonamento sejam direcionadas às classes de software e, conseqüentemente, às aplicações. Porém, o AMIGO não impossibilita o escalonamento com pouca ou nenhuma informação adicional sobre as aplicações que estão sendo executadas. Em substituição às informações sobre a aplicação podem ser usadas outras, como por exemplo: horários de utilização e dias da semana. O relacionamento de classes de software com as políticas de escalonamento será abordado em maiores detalhes na seção 3.6.4. "Políticas de Escalonamento".

O acesso às informações sobre as classes de software e sobre as aplicações é feito através das opções "**Cadastro de Classes**" e "**Cadastro de Aplicações**", disponíveis em "**Classes de Software**" (Figura 3.1).

O cadastro das classes de software (armazenado no arquivo *classoft.dad*) é composto pelas seguintes informações: (1) código da classe, formado por uma numeração seqüencial e crescente de classes cadastradas (de 001 até 999) e que será usado posteriormente para o relacionamento com políticas e aplicações; (2) nome da classe, formado por uma cadeia de caracteres, a qual representa o nome destinado à classe de software cadastrada e (3) características, formadas por uma estrutura com o conteúdo variável (como a *union* da linguagem C) que armazena as características mais significativas sobre a classe, como por exemplo: interatividade com o usuário, comunicação e computação, necessidade de prazos preestabelecidos para término das aplicações (tempo real), entre outras. A inclusão das características não é obrigatória

e as inseridas não precisam ser as mesmas para todas as classes. Embora não sejam obrigatórias e possam ser formadas por diferentes valores, é através das informações contidas em "características" que a relação entre classes de software e políticas poderá ser feita automaticamente. Dessa forma, o usuário deve cadastrar características que possam ser estendidas ao maior número possível de classes, para compará-las corretamente.

As aplicações cadastradas pela opção "**Cadastro de Aplicações**" são armazenadas no arquivo *policy.cfg*, no item [*applications*] (Figura 3.3). Para inserir a aplicação a ser escalonada deve-se fornecer: (1) o nome da aplicação, (2) os seus arquivos executáveis e (3) código da classe de software a que a aplicação pertence.

A consistência das informações nos dois arquivos *classoft.dad* e *policy.cfg* é mantida toda vez que uma classe de software é excluída ou alterada.

Inicialmente, o relacionamento de uma aplicação com a classe de software, será feito diretamente pelo usuário, em função da dificuldade em se obter maiores informações sobre a aplicação a ser executada (está prevista a automação desse procedimento utilizando-se técnicas de redes neurais).

3.5.2. Plataformas Computacionais

A plataforma computacional utilizada no escalonamento é um fator decisivo na escolha da política de escalonamento. A heterogeneidade, a arquitetura dos processadores e a conexão desses processadores são apenas alguns exemplos de fatores pertencentes à plataforma que influenciam o desempenho obtido com o escalonamento [Souza et al. (1999a) (1999b)].

O item "**Plataformas**", disponível na interface da camada superior, viabiliza a inclusão desses fatores e permite que os mesmos sejam considerados durante a escolha das políticas de escalonamento ativas.

A opção "**Cadastro de Hardware**" (Figura 3.1) permite a inclusão e o acesso das informações sobre a plataforma no arquivo *hardware.dad.* As informações armazenadas nesse arquivo são: um código identificador para a máquina (como o endereço IP), nome da máquina, arquitetura, sistema operacional, número de processadores, quantia de memória principal disponível, existência ou não de acesso a disco e características especiais. O campo "número de processadores" é utilizado nas plataformas em que cada processador é formado por vários outros processadores em paralelo. O "acesso a disco" indica se o processador tem ou não acesso a um disco local. As "características especiais" têm a finalidade de informar outras características sobre o processador. Assim como no cadastro de classes de software, as "características especiais" compreendem uma estrutura de dados com conteúdo variado (como a *union* da linguagem C) e o seu uso deve ser criterioso, porque serão utilizadas posteriormente para o relacionamento com os outros itens, como as políticas e as classes de software.

A inclusão dos processadores através da opção "**Cadastro de Hardware**" é feita manualmente pelo usuário. Essa atividade também pode ser feita automaticamente através da opção "**Detecta Hardware**", existente no item "**Plataformas**" (Figura 3.1).

A detecção do hardware disponível pode ser feita através de protocolos de gerência de redes como, por exemplo, o SNMP (*Simple Network Management Protocol*) [Stallings (1993), Tanenbaum(1996)]. Ao encontrar os processadores, o usuário tem a opção de escolher quais serão realmente incluídos e/ou alterados automaticamente. A detecção "parcial" da plataforma permite que sejam incluídos apenas os processadores desejados, como por exemplo, os pertencentes a uma sub-rede em particular.

Após detectar o hardware, são utilizadas as primitivas do sistema operacional como o *rsh* (*remote shell*) para executar alguns processos nos processadores encontrados. Esses processos obtêm as demais informações necessárias ao cadastro da plataforma computacional, as quais não puderam ser obtidas diretamente durante a detecção, em função de limitações e/ou dificuldades no sistema operacional. Após coletarem as informações desejadas, esses processos são finalizados.

Além da inclusão da plataforma computacional, também são flexibilizados os *benchmarks* utilizados para medir a potência computacional de cada processador. Dependendo do *benchmark* escolhido, pode-se analisar o desempenho da plataforma em função, por exemplo, do processador, da comunicação pela rede de conexão, do acesso às memórias primárias e secundárias, entre outros.

A responsabilidade da camada superior em relação aos *benchmarks* é apenas torná-los disponíveis eficientemente para a camada inferior, permitindo que os resultados dos mesmos possam ser recuperados rapidamente e com pouca sobrecarga pelas políticas. Cabe às políticas de escalonamento determinar quais *benchmarks* serão utilizados para diferenciar os processadores.

Para incluir os *benchmarks* é utilizada a opção "**Cadastro de Benchmarks**" do item "**Plataformas**" da camada superior. As informações sobre os *benchmarks*, armazenadas no arquivo *bench.dad*, são as seguintes: código, nome, nome do arquivo executável (com o caminho completo) e principais características. As características compreendem uma estrutura com conteúdo variado (como a *union* da linguagem C) e têm a finalidade de informar os objetivos do *benchmark* usado, embora o seu preenchimento não seja obrigatório.

Tendo-se incluído os *benchmarks* disponíveis, os mesmos podem ser executados nos processadores da plataforma usada gerando, dessa maneira, as informações

necessárias para diferenciá-los em função do seu desempenho. A opção "**Executa Benchmarks**" no item "**Plataformas**" executa os *benchmarks* desejados sobre os processadores da plataforma. Para essa execução, é utilizado o "nome do arquivo executável" inserido no arquivo *bench.dad* e os resultados obtidos em cada processador são armazenados em arquivos denominados para cada *benchmark* executado, conforme ilustrado pela Figura 3.2.

NAS_kernel_EP.cfg

```
# IP / processor_name / result (time/score)
143.107.232.197 lasd02 13.21935 1.000
143.107.232.193 lasd03  1.68200 0.127
143.107.232.195 lasd04  8.77866 0.664
143.107.232.196 lasd06  3.29025 0.249
143.107.232.198 lasd07  1.69246 0.128
143.107.232.194 lasd08  1.95817 0.148
143.107.232.199 lasd13  0.88912 0.067
```

Figura 3.2 – Exemplo de arquivo com o resultado da execução de um *benchmark*.

Para exemplificar essa atividade, considere o *benchmark* NAS usando o *kernel* EP, com o conteúdo de "nome do *benchmark*" no arquivo *bench.dad* como NAS_kernel_EP. Para executá-lo em todos ou apenas em alguns processadores da plataforma computacional, o executável do *benchmark* é localizado e executado remotamente através de primitivas do sistema operacional como o *rsh* (*remote shell*). Após a execução, os tempos obtidos pelo *benchmark* em cada processador são recebidos e armazenados no arquivo *NAS_kernel_EP.cfd*, com o seguinte conteúdo: IP do processador, nome do processador e "resultado" obtido (Figura 3.2). Este "resultado" é formado por dois valores: (1) o resultado obtido com o benchmark, representado por uma métrica de desempenho, como por exemplo o tempo, e (2) o

escore variando entre 0 e 1, sendo que o 0 representa o processador com melhor desempenho e o 1 o processador com o pior desempenho. Para se obter o escore, divide-se o resultado do processador em questão pelo resultado do "pior" processador. Caso outro *benchmark* venha a ser executado sobre os mesmos processadores, outro arquivo será gerado, dessa vez com o nome do outro *benchmark*. A inclusão do resultado e do escore no arquivo de configuração (valores relacionados) permite que a própria política faça o relacionamento que desejar entre os resultados dos *benchmarks*, mesmo que isso gere uma sobrecarga maior; ou então utilize diretamente o escore gravado, diminuindo a sobrecarga para utilizá-lo.

Para desenvolver um *benchmark* para o AMIGO é necessário observar as características da plataforma utilizada (arquitetura, topologia, entre outros) e estabelecer como valor de saída (retorno) uma métrica que possa comparar efetivamente diferentes recursos computacionais (como por exemplo o tempo obtido durante a execução).

3.5.3. Políticas de Escalonamento

O item "**Políticas de Escalonamento**" merece destaque em relação aos demais, principalmente devido à importância de algumas de suas atividades, tais como: (1) viabilizar a inclusão de diferentes políticas de escalonamento e dessa forma torná-las disponíveis, (2) estabelecer o relacionamento automático destas com os demais itens (como classes de software, hardware, e outros) e (3) selecionar (ativar) as políticas que devem realmente ser utilizadas pela camada inferior [Souza et al. (1999a) (1999b)].

O acesso ao cadastro de políticas de escalonamento é feito pela opção "**Cadastro de Políticas**". Nesse cadastro são inseridas (no arquivo *policies.dad*), as seguintes informações: código, nome da política, nome do arquivo executável (com o

caminho completo) e as principais características da política. Assim como acontece nos cadastros dos outros itens, o preenchimento das características não é obrigatório mas, se feito, utiliza uma estrutura com conteúdo variável (como a *union* da linguagem C). Embora a inclusão das características não seja obrigatória, a mesma é de vital importância para que a política possa ser relacionada automaticamente com, por exemplo, as classes de software. É também através das "características" das políticas, que serão determinadas quais delas devem permanecer ativas, em função dos fatores que estão influenciando a plataforma no momento.

O item "**Relaciona Classes & Políticas**" permite o vínculo das classes de software com as políticas de escalonamento. Para cada classe de software cadastrada é possível relacionar uma única política de escalonamento, mas uma mesma política pode estar vinculada a diferentes classes de software. Para fazer o relacionamento, a interface exibe as classes de software previamente cadastradas e permite que cada uma seja vinculada a uma política apropriada.

O relacionamento entre classes de software e políticas também pode ser sugerido através da opção "Relacionamento Automático", constante dentro da opção "**Relaciona Classes & Políticas**". Para que isso seja feito, as características nos cadastros (classes, plataforma e políticas) devem ter sido previamente inseridas. A qualidade do relacionamento será diretamente proporcional à qualidade dessas características.

Após feito o relacionamento (pelo usuário ou pelo AMIGO), o mesmo é armazenado diretamente no arquivo *policy.cfg* em [*policies*] (Figura 3.3). Armazenando o relacionamento de classes com políticas dessa maneira, facilita a recuperação das informações pela camada inferior, diminuindo assim a sobrecarga da mesma.

O uso das políticas de escalonamento é configurado pela opção "**Configura Uso de Políticas**" e essa configuração é armazenada no item [*config*] do arquivo *policy.cfg* (Figura 3.3). Ao configurar o uso do escalonamento, o usuário pode determinar se é possível:

(1) a existência de mais de uma política ativa ao mesmo tempo. Essa opção (*more_policies*) autoriza a execução de políticas vinculadas às classes de software de maneira concorrente à política independente dessas mesmas classes. Impedir o uso simultâneo de várias políticas de escalonamento pode ser útil quando se deseja fixar a maneira como os escalonamentos serão feitos sobre a plataforma;

policy.cfg

```
[reserve]
#reserve policy name (binary)

[config]
more_policies=1 / 0
by_demand=1 / 0
change_policy=1 / 0
load_cache=1 / 0
direct_route=1/ 0
profile_time= <time or 0>

[policies]
#software class code, policy code, binary policy name, active

[applications]
# application name, binaries code, software class code
```

Figura 3.3 – Estrutura do arquivo de configuração policy.cfg

(2) ativar políticas sob demanda. Essa opção (*by_demand*) permite que uma política (que não está ativa) seja executada quando uma aplicação solicita o

escalonamento de processos. A ativação sob demanda só é possível se for permitida a existência de mais de uma política ativa ao mesmo tempo;

(3) mudar as políticas ativas automaticamente em tempo de execução. Essa opção (*change_policy*) permite ou não que as políticas em atividade possam ser substituídas, toda vez que for determinado que o escalonamento não está sendo profícuo (através do monitoramento), ou então, toda vez que as políticas forem alteradas pela opção "**Seleciona Políticas**" do item "**Políticas de Escalonamento**";

(4) utilizar *cache* no AMIGOD para o índice de carga obtido em cada política ativa. Essa opção (*load_index_cache*) tem por objetivo otimizar a troca de informações entre as políticas de escalonamento em cada processador;

(5) a troca de mensagens diretamente entre as políticas, sem passar pelo AMIGOD. Essa opção (*direct_route*) possibilita o aumento de desempenho na comunicação da camada inferior, por reduzir os "saltos" necessários para que cada mensagem seja enviada entre os seus módulos;

(6) o uso de um arquivo *profile* (chamado de *amigo.pfl*), o qual é responsável por armazenar informações sobre o escalonamento feito na plataforma em até um período preestabelecido. Se a opção estiver habilitada, o usuário fornece o período máximo para coleta das informações (em minutos). Se o período for de 120 minutos, por exemplo, o *amigo.pfl* irá armazenar as 2 últimas horas de execução e, após essas 2 horas de coleta, as informações mais antigas serão as primeiras a serem descartadas (*First-In-First-Out*). A responsabilidade pela coleta dessas informações é do módulo AMIGOD, pertencente à camada inferior.

O usuário utiliza a opção "**Seleciona Políticas**" para determinar quais políticas de escalonamento devem estar ativas no momento. É permitido que várias políticas de escalonamento estejam ativas em um mesmo instante, dividindo-as em três grupos: política reserva (*reserve policy*), política independente (*independent policy*) e políticas específicas (*specific policies*).

A política reserva representa a política utilizada apenas se todas as outras políticas ativas não puderam ser executadas. Como o próprio nome indica, a política reserva atua quando problemas impedirem a execução normal das outras políticas ativas. Um dos principais objetivos da política reserva é viabilizar o conceito de tolerância a falhas, de maneira transparente ao usuário. Mesmo que não haja o escalonamento mais apropriado para o momento, os processos serão escalonados sob o AMIGO.

A política independente representa a política indicada pelo ambiente para escalonar os diferentes processos, independente da classe de software a que esses pertençam. A política independente só é usada quando não é possível utilizar uma das políticas específicas.

As políticas específicas são as primeiras políticas procuradas no escalonamento. No entanto, elas serão executadas somente se: (1) a opção *more_policies* habilitar a execução de mais de uma política de escalonamento; (2) a aplicação estiver vinculada a uma política de escalonamento ativa, através da sua classe de software; e (3) através da opção *by_demand* permitir a execução de políticas que não estão ativas, quando essas foram solicitadas por uma aplicação em particular.

A opção "**Seleciona Políticas**" tem disponível três maneiras para determinar quais políticas passarão a estar ativas: a livre, a assistida e a padrão (ou automática).

Na maneira de seleção "livre", as políticas de escalonamento ativas são escolhidas diretamente pelo usuário, através de uma relação de políticas (e classes de software vinculadas) que lhe é exibida durante a seleção. Com base nessa relação o usuário determina qual política estará ativa como independente e quais estarão ativas como específicas. A maneira de seleção "livre" deve ser usada por usuários que dominam a teoria de escalonamento de processos, porque a seleção feita será de responsabilidade total do mesmo.

A seleção "assistida" auxilia o usuário na escolha das políticas ativas, através de heurísticas pré-definidas. A interação feita pelo usuário altera o grau de importância de cada questão levantada, através de pesos atribuídos às mesmas (em uma escala de 0 a 10). A heurística seguida pode, por exemplo, considerar: (1) a execução simultânea de diferentes classes de software; (2) as classes de software executadas com maior freqüência; (3) o grau de sobrecarga da plataforma computacional; (4) a variação da carga computacional em um intervalo de tempo; (5) a presença de múltiplos usuários; (6) maior qualidade nas decisões sobre o escalonamento, mesmo considerando uma sobrecarga maior gerada pelo mesmo; (7) a heterogeneidade do hardware; (8) a velocidade e confiabilidade da rede de conexão; (9) a topologia da rede de conexão, verificando a necessidade de saltos entre os processadores e/ou *gateways*.

O usuário pode, através da seleção "assistida", ponderar quais dentre essas características apresentam um impacto maior e/ou menor no desempenho obtido com o escalonamento. Tendo os pesos atribuídos pelo usuário para cada característica e a heurística a ser seguida, o AMIGO determina quais políticas devem ser ativadas como independente e específicas para os próximos escalonamentos.

Utilizando a maneira "padrão" (ou automática), o usuário não interfere nas decisões tomadas para a seleção das políticas ativas independente e específicas. A heurística usada pode, inclusive, ser a mesma empregada pela seleção "assistida", mas a importância de cada característica é ponderada automaticamente.

A política reserva não é determinada pela seleção "assistida" ou "padrão", sendo indicada exclusivamente pelo usuário. Isso é feito em virtude da política reserva ter como objetivo fundamental aumentar a confiabilidade e não a melhoria de desempenho com o uso dessa política.

Um dos principais requisitos para se obter qualidade na seleção "padrão" e "assistida" é o preenchimento adequado do campo "características" (existente em vários cadastros da camada superior). Como preenchimento adequado entende-se o uso de características que realmente destaquem o que se está cadastrando (classes de software, plataforma, políticas, ou outros), e que ao mesmo tempo sejam extensíveis aos seus pares, a fim de compará-los com exatidão. Um preenchimento desejável para uma política, por exemplo, é: se ela faz ou não migração de processos, para qual(is) classe(s) de software ela é indicada, qual índice de carga utiliza, se considera a topologia da rede de conexão, se é distribuída ou centralizada, entre outros.

Para que a heurística seja empregada, são utilizadas as informações vindas do arquivo *profile* (*amigo.pfl*), se este estiver disponível, além das características inseridas nos diversos cadastros. Com o *amigo.pfl* e as características cadastradas, há dados locais e específicos suficientes para decidir quais políticas ativar.

Independente da maneira como as políticas ativas serão selecionadas, as informações pertinentes à seleção e às configurações serão armazenadas no arquivo *policy.cfg* (Figura 3.3). O item [*reserve*] recebe o nome do arquivo executável da política reserva a ser usada pela camada inferior, caso as demais não possam ser

executadas. Se a política reserva não for especificada, o conteúdo desse item será nulo.

O item [*policies*], além de manter a relação existente entre as classes de software e as políticas, também recebe quais políticas estarão ativas para a execução. A política independente é a primeira política relacionada no item [*policies*]. No código da classe consta o termo "*independent*". As demais políticas constantes nessa relação estão vinculadas às classes de software e podem ou não estar ativas, dependendo do conteúdo do campo "*active*" (Figura 3.3).

3.5.4. Monitoramento

O monitoramento do escalonamento permite que o usuário verifique, em tempo de execução, se os objetivos propostos às políticas de escalonamento estão sendo atingidos. Embora o monitoramento não faça parte do escalonamento, ele é uma ferramenta necessária para avaliar o comportamento da plataforma computacional, principalmente se esta for multiusuária.

Com as informações vindas do monitoramento é possível afirmar se as políticas de escalonamento ativas devem ou não ser substituídas devido a fatores como por exemplo: maior ou menor utilização da plataforma por outros usuários, execução de aplicações diferentes daquelas esperadas inicialmente, alterações na plataforma computacional, entre outros.

O monitoramento pode ser feito de diversas formas. Algumas das possíveis soluções incluem: um hardware específico, alterações feitas no sistema operacional e o monitoramento feito por softwares fora do sistema operacional (em um nível mais alto). Considerando algumas características como: transparência, preocupação com a portabilidade e desenvolvimento fora do sistema operacional, optou-se inicialmente em implementar o monitoramento também com essas características. Dessa forma, o

monitoramento é feito através de ferramentas de software (sem a necessidade de um hardware específico), que obtenham informações através de primitivas do sistema operacional e de maneira externa à aplicação.

Para realizar o monitoramento, há a necessidade de se utilizar métricas que quantifiquem o quanto os objetivos foram realmente atingidos. Em função da grande quantidade de métricas que podem ser utilizadas, o usuário pode formar o seu próprio cadastro de métricas. Com esse cadastro de métricas pode-se avaliar o resultado obtido por um escalonamento sob diferentes óticas e, assim, munir o usuário com informações objetivas para realizar as alterações que por ventura sejam necessárias.

Para ter acesso às métricas cadastradas é utilizada a opção "**Cadastro de Métricas**", disponível no item "**Monitoramento**". As informações (armazenadas em *metrics.dad*) são compostas por: código, nome, nome do arquivo executável (com o caminho completo), nome do arquivo que irá armazenar os valores obtidos (com caminho completo) e características específicas da métrica. As "características" da métrica cadastrada são armazenadas através de uma estrutura com conteúdo variável, (como a *union* da linguagem C), de maneira semelhante à usada nos outros cadastros para o mesmo campo.

As métricas estão implementadas em arquivos executáveis, os quais são chamados pela rotina de monitoramento para apenas coletar as informações desejadas. Uma vez coletada a informação, a mesma é tratada pela rotina de monitoramento, para ser visualizada de maneira gráfica e/ou textual pelo usuário e em seguida armazenada nos arquivos que guardam os resultados obtidos (especificados no cadastro de métricas).

Para iniciar e controlar o monitoramento, o usuário utiliza a opção "**Início/Término**" no item "**Monitoramento**". Nessa opção, além de ativar/paralisar a atividade de monitorar a plataforma, o usuário pode: (1) escolher a métrica a ser utilizada, (2) determinar o horário para funcionamento, (3) estabelecer os intervalos para a coleta das informações e (4) visualizar o desempenho, expresso gráfica e/ou textualmente.

A rotina de monitoramento é um processo *daemon* (chamado *monitor*) pertencente à camada superior, mas com funcionamento independente da mesma. Essa separação permite-lhe autonomia de funcionamento mesmo quando a camada superior não está em execução. Ao ser iniciado pela camada superior (em cada processador), o *monitor* também recebe os parâmetros que configuram a sua atividade e lhe fornecem certa flexibilidade.

Os resultados do monitoramento, além de exibidos pela camada superior, são armazenados em um item chamado [*monitor*] existente no arquivo *profile*. Com base nessas informações, a seleção de políticas pode ser invocada pela camada superior para rever as políticas de escalonamento ativas no momento e sugerir mudanças. Dessa forma, permite-se a troca das políticas de escalonamento, de maneira transparente, dinâmica e também automatizada.

A atividade de monitorar uma plataforma computacional não é uma tarefa trivial e, por si só, constitui uma linha de pesquisa, a qual está sendo explorada em um trabalho de pesquisa à parte. Nesta seção estão especificadas as necessidades e algumas questões de implementação do monitoramento para o AMIGO sem, entretanto, ter o intuito de esgotar completamente a questão.

3.5.5. Simulação

O objetivo de incluir o item "**Simulação**" é permitir que o comportamento das políticas de escalonamento possa ser analisado, sem a necessidade de executá-las realmente.

O uso da simulação no escalonamento é extremamente atraente porque permite, entre outras coisas: (1) uma avaliação preliminar de uma política antes da sua implementação, diminuindo assim os custos dessa implementação e também com testes e avaliações posteriores; (2) avaliar o comportamento de uma política, já implementada ou não, sob outros fatores tais como: outras plataformas, outras classes de software ou variações na carga computacional; (3) analisar previamente as alterações na implementação de políticas já existentes (com o intuito de melhoria de desempenho); ou (4) o comportamento do AMIGO com o uso simultâneo de várias políticas de escalonamento.

Para que a simulação possa ser realizada há a necessidade de se modelar de maneira correta cada componente de um escalonamento (plataforma, aplicação, algoritmo de escalonamento, multiusuários, entre outros). A representação desses itens pode ser feita de diversas maneiras, cada uma agregando vantagens ou desvantagens à modelagem, dependendo do caso.

Em função da abrangência da simulação, esta tese não irá pormenorizar essa atividade, considerando apenas que ela faz parte das diretrizes iniciais do AMIGO, devendo ser especificada e implementada em trabalhos futuros. Uma opção clara é ter-se no AMIGO um ponto de chamada para um ambiente completo de simulação onde: plataforma, aplicações e políticas possam ser modeladas e simuladas.

3.6. A Camada Inferior do AMIGO

Enquanto a camada superior é a responsável pela configuração que viabiliza a flexibilidade do escalonamento dos processos, é a camada inferior que oferece todo o suporte necessário para que as características dinâmica e flexível sejam implementadas [Souza et al. (1999b) (1999c)].

A Figura 3.1 ilustra os três módulos que formam a camada inferior: o AMIGOD (*AMIGO Daemon*), as políticas de escalonamento e o ambiente de passagem de mensagem.

O AMIGOD é um processo servidor atuando como um elo de ligação entre as políticas de escalonamento e o ambiente de passagem de mensagem. O mesmo não implementa nenhuma das políticas de escalonamento, porque estas compõem um módulo à parte na camada inferior. O objetivo dessa separação é permitir que as políticas de escalonamento em atividade possam ser substituídas em tempo de execução e de maneira transparente ao ambiente de passagem de mensagem/aplicação em funcionamento. Essa troca, em tempo de execução, é a característica dinâmica do AMIGO. Na descrição do AMIGO, está sendo usado o termo "política" de escalonamento conforme definido anteriormente. Porém, uma política "continua" sendo implementada através de um algoritmo de escalonamento e necessita de mecanismos, também de acordo com as definições já apresentadas.

O ambiente de passagem de mensagem compreende uma versão modificada do PVM ou MPI. A versão modificada interage com o AMIGOD sempre que um escalonamento necessita ser realizado, modificando dessa maneira a política de escalonamento empregada originalmente. Em relação à aplicação desenvolvida pelo usuário, esta não necessita de nenhuma mudança para ser executada com o AMIGO.

Todas as características são transparentes e nenhuma linha de código da mesma precisa ser modificada [Souza et al. (1999b) (1999c)].

A camada inferior é independente do funcionamento da camada superior. Para uma funcionar a outra não precisa, necessariamente, estar funcionando. O que une as duas camadas são os arquivos de configuração gerados pela camada superior e utilizados na camada inferior.

3.6.1. A Comunicação na Camada Inferior

A comunicação realizada pela camada inferior é baseada nos protocolos TCP (*Transmission Control Protocol*) e UDP (*User Datagrama Protocol*), os quais são utilizados por intermédio da interface *sockets*. O AMIGO assume que todos os processadores que farão parte do escalonamento são capazes de se conectarem através desses mecanismos de comunicação.

A Figura 3.4 exemplifica a comunicação realizada pela camada inferior, a qual deve considerar três conexões diferentes: entre o AMIGOD e as políticas ativas (AMIGOD⇔Políticas), entre o AMIGOD e o ambiente de passagem de mensagem (AMIGOD⇔Ambiente de Passagem de Mensagem) e entre os AMIGODs situados em diferentes processadores (AMIGOD⇔AMIGOD) [Souza et al. (1999b) (1999c)].

Para a comunicação no mesmo processador (AMIGOD⇔Políticas e AMIGOD⇔Ambiente de Passagem de Mensagem), o protocolo de comunicação usado é o TCP; e entre processadores diferentes, (AMIGOD⇔AMIGOD), o protocolo de comunicação é o UDP. Essas escolhas são análogas às mesmas feitas pelo PVM para a sua comunicação [Geist et al. (1994)], e referem-se principalmente à capacidade escalar (no caso da comunicação AMIGOD⇔AMIGOD) e à

confiabilidade (no caso das comunicações AMIGOD⇔Políticas e AMIGOD⇔ Ambiente de Passagem de Mensagem).

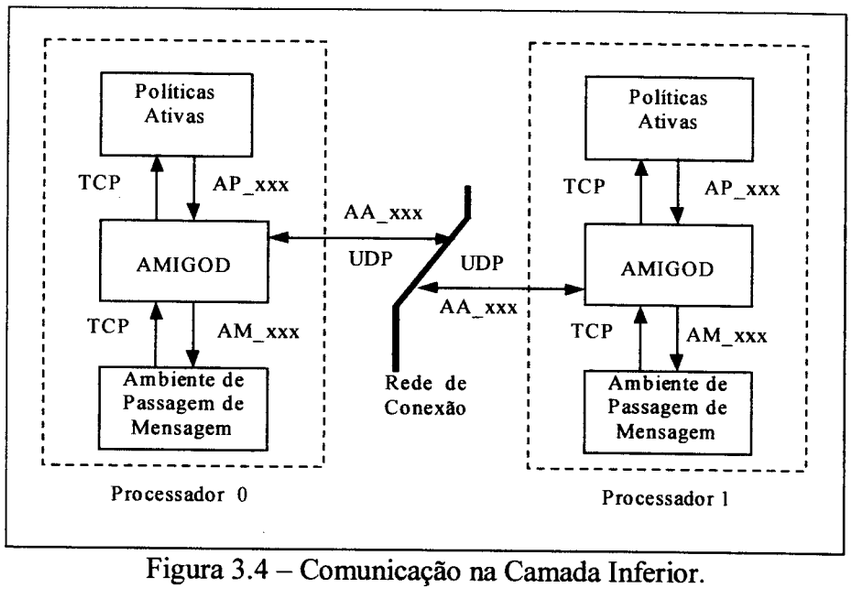


Figura 3.4 – Comunicação na Camada Inferior.

Toda mensagem trocada na camada inferior possui um cabeçalho que tem por finalidade identificá-la. Esse cabeçalho possui dezesseis *bytes* e é formado por quatro campos principais: o código da mensagem, o destino da mensagem, a origem da mensagem e o tamanho do corpo da mensagem (Figura 3.5).

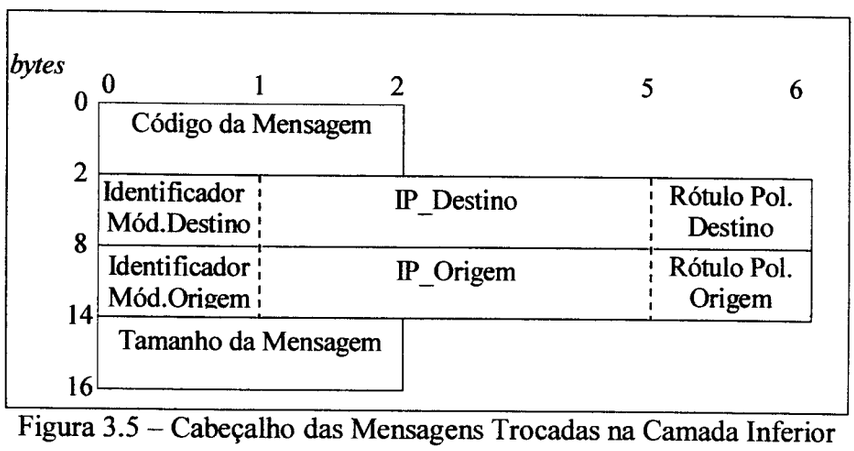


Figura 3.5 – Cabeçalho das Mensagens Trocadas na Camada Inferior

O código da mensagem utiliza dois *bytes* e determina tanto a finalidade da mensagem, quanto quais módulos estão participando da comunicação. Se a

comunicação é entre AMIGOD⇔Políticas, o código da mensagem é determinado por macros, prefixadas por AP_. Se a mensagem é entre AMIGOD⇔Ambiente de Passagem de Mensagem, o prefixo das macros é AM_. Se a mensagem é entre AMIGOD⇔AMIGOD, o prefixo das macros é AA_ (Figura 3.4). Todas as macros, com o conteúdo detalhado de todas as mensagens trocadas na camada inferior do AMIGO, estão descritas no Anexo A. O código AA_ADD é um exemplo de código de mensagem utilizado para que um AMIGOD "A" solicite a outro AMIGOD "B" a inclusão de um outro AMIGOD na *amigod_table* (do AMIGOD "B").

Com o código da mensagem, o módulo que recebê-la não precisa, necessariamente, "desempacotá-la" para determinar qual procedimento executar. Através do código pode-se saber se a mensagem recebida necessita apenas de um reenvio para outro módulo ou então se será necessário executar algo no próprio módulo.

Os campos de destino e origem da mensagem são formados por seis *bytes* cada. O primeiro *byte*, o Identificador do Módulo Destino/Origem, define exatamente quais módulos estão participando da comunicação. Se o Identificador do Módulo for o caracter "A", então o módulo é um AMIGOD. Se for o caracter "P", então o módulo é uma política. Se for o caracter "M", o módulo é o ambiente de passagem de mensagem. Os quatro próximos *bytes* representam o endereço IP do processador que está participando da comunicação como origem ou destino. No caso do cabeçalho especificar uma mensagem originada ou destinada à uma política, o último *byte*, Rótulo da Política Destino/Origem, identifica dentre as políticas ativas qual efetivamente está comunicando-se.

O campo "tamanho da mensagem" determina quantos *bytes* devem ser lidos, referentes ao corpo da mensagem. Há a possibilidade de mensagens com tamanho zero (apenas com o cabeçalho).

3.6.2. O AMIGOD

O processo servidor AMIGOD é o módulo central da camada inferior. Uma das principais características do AMIGOD é ser completamente distribuído. Não há uma relação hierárquica como mestre/escravo entre os AMIGODs. Dessa forma, estes são executados em todos os processadores da plataforma computacional que farão parte do escalonamento, tendo os mesmos direitos e obrigações em cada processador. A Figura 3.4 ilustra dois processadores, cada qual com uma cópia do AMIGOD. Todas as mensagens trocadas entre os módulos passam pelo AMIGOD, responsável pelo roteamento e gerência das mensagens que circulam pela camada inferior, assim como por manter ativas as políticas necessárias no momento. Excepcionalmente as políticas têm permissão para trocar mensagens diretamente entre elas, sem a necessidade de passá-las pelo AMIGOD (o roteamento direto é habilitado pela opção *direct_route*, existente no arquivo *policy.cfg*). Isso é adotado para melhorar o desempenho do escalonamento, com a diminuição dos saltos necessários para que uma mensagem chegue da política em um processador até outra em outro processador. No entanto, dessa forma o AMIGOD não tem controle nem responsabilidade alguma sobre as mensagens [Souza et al. (1999b) (1999c)].

O código do AMIGOD foi projetado para ser pequeno, mas eficiente na tarefa de fornecer suporte às características inseridas no ambiente. Com base nessa premissa, o AMIGOD é composto fundamentalmente por estruturas de dados, responsáveis por manter tabelas com as políticas que estão ativas no momento, tabelas de roteamento

e também os mecanismos de comunicação entre os processos (módulos da camada inferior).

As principais estruturas de dados do AMIGOD são as tabelas *amigod_table* e *policy_table*. Essas tabelas mantêm, respectivamente, uma relação de processadores e de políticas que estão ativos no momento.

A *amigod_table*, também conhecida como tabela de roteamento, relaciona os processadores que farão parte do escalonamento armazenando as seguintes informações: o nome do processador, o endereço IP e o *port number* do AMIGOD no processador (Figura 3.6). As informações dessa tabela referem-se principalmente ao AMIGOD, devido ao fato de sempre existir uma cópia deste por processador. Sem o AMIGOD em execução, o processador não fará parte do escalonamento.

```
struct amigod_table {
  char a_name[MAXNAMEHOST];      /* nome do processador          */
  unsigned long  a_ip;           /* endereço IP do AMIGOD        */
  unsigned short a_port;         /* port UDP do AMIGOD           */
  struct amigod_table *a_next;   /* próximo processador da tabela */
};
```

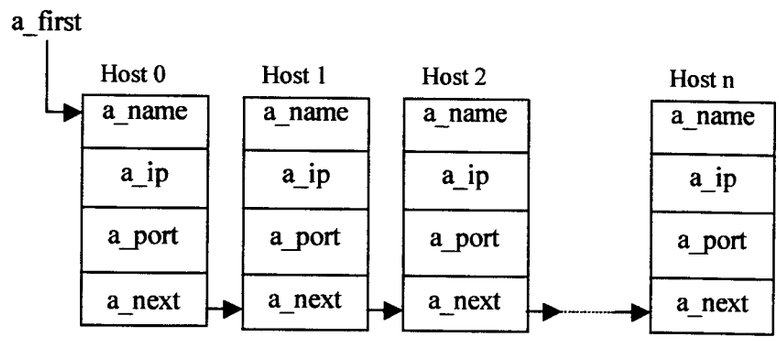


Figura 3.6 – Estrutura *amigod_table* existente no AMIGOD.

Cada AMIGOD mantém uma cópia idêntica da *amigod_table*. As atualizações são realizadas sempre que um processador ou é anexado, ou deixa o grupo de processadores.

A *policy_table* contém informações sobre as políticas que estão ativas e sob a supervisão do AMIGOD (Figura 3.7). Essas políticas são ativadas e registradas inicialmente na *policy_table* quando este (o AMIGOD) inicia a sua execução e obtém as informações vindas do arquivo de configuração *policy.cfg*. A *policy_table* também é alterada toda vez que a camada superior informa à camada inferior que houve mudanças nas diretrizes do escalonamento. A primeira política da *policy_table* (na posição 0) é sempre a política independente, indicada pela camada superior.

```
struct policy_table {
  char p_name[MAXNAME];   /* nome da política                    */
  char p_label;           /* código usado pela política          */
  char p_class;           /* classe vinculada à política         */
  int p_socket;           /* socket usado pela política          */
  int p_load;             /* cache da carga obtida pela política */
  int p_pid;              /* pid da política                     */
};
```

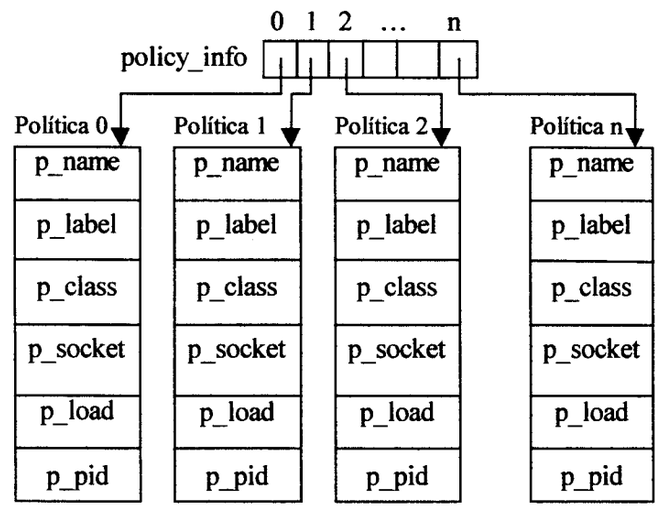


Figura 3.7 – Estrutura *policy_table* existente no AMIGOD.

A primeira atividade realizada pelo AMIGOD ao iniciar o seu funcionamento é configurar-se[1]. Na sua configuração são: (1) criadas e atualizadas as estruturas de dados, (2) obtidos os arquivos de configuração necessários, (3) iniciadas as políticas

que devem ser ativadas, (4) estabelecidas conexões TCP com essas políticas e (5) estabelecida outra conexão TCP com o ambiente de passagem de mensagem. Caso o AMIGOD não seja o primeiro a ser criado, o mesmo envia uma mensagem AA_ADD para outro preexistente, avisando a sua chegada e solicitando que as outras *amigod_table* também sejam atualizadas.

Um AMIGOD novo (recém iniciado) sabe da existência ou não de, no mínimo, outro AMIGOD, devido a um parâmetro recebido pela linha de comando que informa ambos, o endereço IP e o *port number* desse outro AMIGOD remoto. A partir do terceiro, esse AMIGOD remoto informado não é necessariamente o primeiro a ter sido criado, visto que não há uma relação mestre/escravo entre eles.

O AMIGOD remoto pode ser qualquer um pertencente ao grupo de processadores disponíveis. O escolhido terá duas responsabilidades: (1) informar ao novo a existência de todos os outros e (2) informar aos outros já existentes a presença do novo.

Feita a sua configuração e atualizadas todas as estruturas de dados necessárias (local e remotamente), o AMIGOD executa a rotina *WorkUntilDeath( )*. Essa rotina é basicamente um *loop* infinito que recebe e envia mensagens das políticas ativas, do ambiente de escalonamento local ou de algum AMIGOD remoto. Quando a *WorkUntilDeath( )* está esperando a chegada de uma mensagem, por algum dos mecanismos de comunicação criados, o AMIGOD é suspenso pelo sistema operacional do processador, sendo colocado na fila de processos que estão aguardando por *I/O*. Dessa forma, minimiza-se a concorrência ao processador.

---

[1] Todas as principais rotinas pertencentes à camada inferior estão especificadas no Anexo B.

A *WorkUntilDeath( )* também é a responsável por atualizar o arquivo *profile*, se habilitado pela camada superior. A cada troca de mensagem efetuada pelos módulos da camada inferior, são armazenadas informações como: classes de software em execução, cargas computacionais dos processadores, entre outras.

Outras questões importantes em relação ao AMIGOD são: o controle que este faz sobre as várias políticas ativas em um dado momento, e a troca de políticas ativas em tempo de execução. Para determinar quais políticas devem ser iniciadas, este obtém o item [*policies*] com a relação das políticas ativas do arquivo de configuração *policy.cfg*, (Figura 3.3). Para que mais de uma política possa estar ativa simultaneamente, a opção *more_policies* do item [*config*] no arquivo *policy.cfg* deve estar habilitada. Ao serem iniciadas pelo AMIGOD, as políticas ativas recebem um número seqüencial também chamado de "rótulo" (*label*). Esse rótulo é inserido no campo *p_label* da estrutura de dados *policy_table* e será utilizado posteriormente em todas as comunicações que envolvam as políticas (no campo R*ótulo da Política de Origem/Destino* do cabeçalho), distinguindo-as umas das outras. Uma vez que a política deixa de ser executada, esse rótulo é descartado e pode ser aproveitado mais tarde por outra política que venha a entrar em execução.

Para relacionar as classes de software com as políticas de escalonamento, o AMIGOD também utiliza o item [*policies*] existente no arquivo *policy.cfg* (Figura 3.3). A primeira política a ser ativada e também a primeira relacionada no item [*policies*], é a política independente (possui o termo "*independent*" no campo "código da classe de software").

Se uma aplicação deseja utilizar uma política específica, que não a indicada genericamente pela camada superior, é necessário que: (1) a aplicação tenha sido previamente cadastrada, relacionando dessa forma, a aplicação com uma classe de

software no item [*applications*] do arquivo *policy.cfg*; (2) a classe de software relacionada com a aplicação esteja, por sua vez, relacionada com uma política de escalonamento, conforme descrito acima; (3) que a opção *more_policies* no item [*config*] do arquivo *policy.cfg* habilite a existência de mais uma política ativa; e (4) que a política específica desejada esteja ativa durante a execução da aplicação. Caso a política específica não esteja ativa no momento do escalonamento e a opção *by_demand* no item [*config*] do arquivo *policy.cfg* esteja habilitada (com o valor um), então a política desejada será colocada em execução pelo AMIGOD, quando for solicitada pela primeira vez. Caso uma aplicação solicite uma política específica e os requisitos acima impeçam a execução dessa política, utilizar-se-á a política independente, indicada pela camada superior.

O objetivo em se relacionar políticas de escalonamento às classes de software (e como conseqüência às aplicações cadastradas) é privilegiar o conhecimento que se tem sobre as aplicações que serão escalonadas. Mesmo assim, a falta de maiores informações sobre as aplicações não deve impedir o escalonamento, nem tampouco que se almeje melhores desempenhos.

Caso o AMIGOD falhe na sua tentativa de iniciar as políticas ativas (independente e específicas), a política indicada no item [*reserve*] do arquivo *policy.cfg* será iniciada. Somente após uma eventual falha dessa política reserva é que o AMIGOD informa ao ambiente de passagem de mensagem que está impossibilitado de executar as suas políticas de escalonamento.

A troca de políticas ativas em tempo de execução do AMIGOD é iniciada pela camada superior, a qual envia um sinal assíncrono (*kill(AMIGOD_pid, SIGUSR1)*) para o AMIGOD, interrompendo a sua execução normal e chamando a rotina *ChangeActivePolicy*() que tratará a interrupção gerada. A rotina

ChangeActivePolicy() é responsável por: (1) obter novamente o arquivo *policy.cfg*, (2) verificar quais políticas devem permanecer ativas doravante, (3) executar e/ou retirar de execução as políticas necessárias, (4) enviar a mensagem AA_CHANGEPOL para todos os demais processadores, solicitando as trocas das políticas e, finalizando, (5) atualizar a tabela de políticas ativas (*policy_table*). Essa troca ocorre de maneira transparente ao usuário e se houver alguma requisição sendo atendida durante a troca, ela será concluída antes da suspensão. Se alguma requisição chegar durante o processo de troca de políticas ativas, essa irá esperar a conclusão da atividade em andamento.

3.6.3. Os Ambientes de Passagem de Mensagem

Os ambientes de passagem de mensagem compõem um dos módulos da camada inferior, porém, obviamente, não foram desenvolvidos nesta tese. De um modo geral, o ambiente de passagem de mensagem atua como um cliente do AMIGO, solicitando-lhe, por exemplo, os processadores necessários às aplicações. Para tanto, esses ambientes (PVM e MPI) sofreram pequenas, mas significativas alterações para que pudessem interagir com o ambiente [Souza et al. (1999b) (1999c)].

As mudanças necessárias foram: (1) na inclusão e remoção de novos processadores na máquina paralela virtual, onde novos AMIGODs devem ser iniciados e finalizados; (2) no escalonamento de novos processos nos processadores e (3) na migração de processos que já estão em execução. Os parágrafos abaixo descrevem as principais mudanças realizadas nesses três itens usando o PVM como exemplo.

O AMIGOD deve ser iniciado juntamente com o PVMD (o *PVM Daemon*). Dessa forma, ao iniciar a execução pela sua rotina *main( )*, o PVMD também executa a rotina *StartAMIGOD( )*, a qual cria os mecanismos de comunicação entre processos

necessários, executa o AMIGOD através das primitivas *fork( )/exec( )* e estabelece uma conexão TCP específica com o mesmo (veja a especificação da rotina *StartAMIGOD( )* no Anexo B).

Depois de iniciado, o AMIGOD envia para o PVMD o *port* UDP utilizado na comunicação AMIGOD⇔AMIGOD. O *port UDP*, em conjunto com o endereço IP do processador, será enviado posteriormente de um PVMD a um novo PVMD (em um novo processador) quando este for iniciado. O envio do endereço é feito através da passagem do parâmetro de linha de comando -a*IP:portUDP*. Essa abordagem permite que o futuro AMIGOD iniciado no novo processador receba um endereço para a comunicação com um AMIGOD já existente.

Se o AMIGOD iniciar a sua execução sem receber esse endereço IP e *port* UDP, o mesmo assume que é o primeiro a ser criado e, portanto, não há a necessidade de atualizar as suas estruturas de dados, nem tampouco comunicar a sua chegada aos outros AMIGODs.

Uma vez recebido o *port* UDP do AMIGOD local, o PVMD segue a sua execução normal de maneira independente, garantindo a ambos (PVMD e AMIGOD) autonomia nas suas atividades.

Quando um PVMD está para ser finalizado, o mesmo executa a rotina *RemoveAMIGOD( )* para que a mensagem AM_REM seja enviada para o AMIGOD e a conexão TCP PVMD⇔AMIGOD seja cancelada. Essa mensagem informa ao AMIGOD que o mesmo deve ser finalizado de maneira apropriada, informando aos outros AMIGODs a sua exclusão, finalizando as suas políticas e cancelando os mecanismos de comunicação.

O escalonamento de novos processos (no PVM) é iniciado com a execução da *pvm_spawn( )*, a qual irá solicitar a execução de novos processos sobre os

processadores disponíveis. Conforme pode-se observar na Figura 3.8, a *pvm_spawn()* na biblioteca LIBPVM apenas prepara uma mensagem a ser enviada ao PVMD local, o qual recebe a solicitação pela *tm_spawn()*. A *tm_spawn()*, com base nos parâmetros enviados pela *pvm_spawn()*, determina quais processadores farão parte do escalonamento, inserindo-os em uma estrutura de dados própria e, posteriormente, chama a rotina *assign_tasks()* ainda no PVMD local.

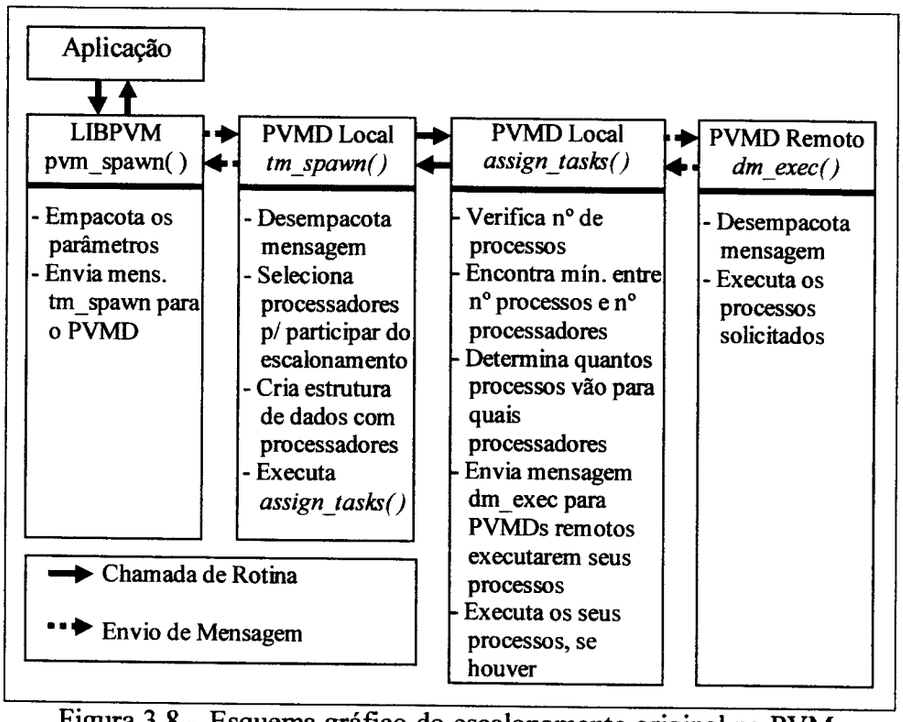


Figura 3.8 – Esquema gráfico do escalonamento original no PVM.

A rotina *assign_tasks( )* verifica o número de processos a escalonar e encontra o mínimo entre o número de processos e o número de processadores determinados anteriormente pela *tm_spawn( )*. Isso é feito para saber se será necessário atribuir mais de um processo para o mesmo processador. Determinado quantos processos vão para cada processador da relação, o PVMD local executa os processos que devem ser executados localmente (se for o caso) e envia mensagens dm_exec para os PVMDs

remotos solicitando-lhes que executem os processos que faltam, através de uma política *round-robin.*

A Figura 3.9 ilustra a alteração feita no código do PVMD com a inclusão da rotina *GetHostsFromAMIGOD( )*. Essa rotina, chamada pela *tm_spawn( )*, determina quais processadores farão parte do escalonamento em substituição à parte do código da *tm_spawn( )* responsável anteriormente por isso. A *GetHostsFromAMIGOD( )* envia uma mensagem AM_SCHED para o AMIGOD contendo o nome do processo a ser escalonado, o número de instâncias a criar deste processo e a arquitetura desejada. O AMIGOD ao receber essa mensagem determina qual política de escalonamento deve receber essa solicitação e repassa essa mensagem para a mesma através da mensagem AP_SCHED. As mensagens retornadas da política e do AMIGOD (respectivamente AP_SCHEDACK e AM_SCHEDACK) contêm uma relação com os processadores receptores dos novos processos, exatamente na ordem em que os mesmos devem ser utilizados no escalonamento.

A migração de processos é uma atividade bem mais complexa que as descritas anteriormente, por envolver algumas mudanças a mais nos ambientes de passagem de mensagem como: identificadores de processos, descritores de arquivos, mecanismos de comunicação entre processos, dependências do local de execução, entre outros. Para realizar uma migração, o ambiente de passagem de mensagem deixa de ser um simples cliente e passa a também receber solicitações para que mudanças sejam realizadas nos processos em execução. Dessa forma, o AMIGO decide quando e como a migração será realizada (através de uma de suas políticas) e o ambiente de passagem de mensagem prepara-se para a transferência do processo, controlando (principalmente) a comunicação destinada ao mesmo.

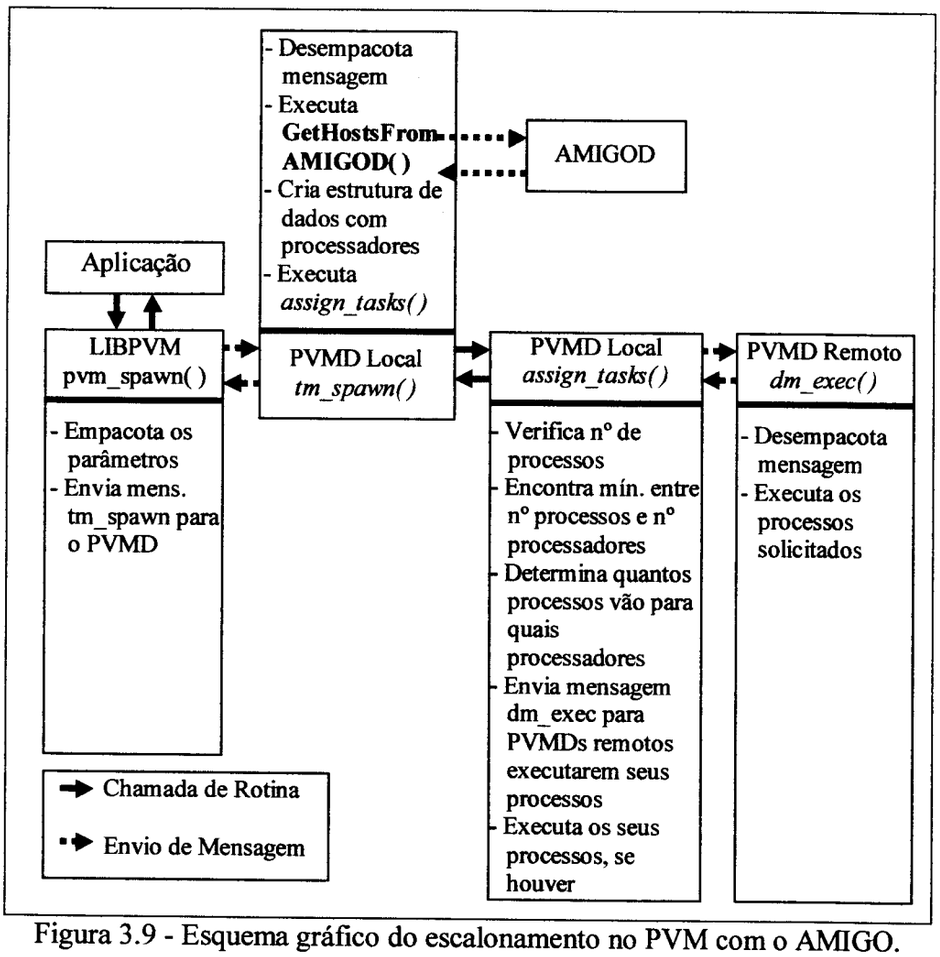


Figura 3.9 - Esquema gráfico do escalonamento no PVM com o AMIGO.

Considerando a migração de processos, há quatro rotinas principais inseridas no ambiente de passagem de mensagem: *SendProcessPolicy( )*, *StopProcess( )*, *AcceptNewProcess( )* e, finalizando, *RouteMessageForNewAddress( )*. A rotina *SendProcessPolicy( )* envia, quando solicitada, a relação de processos em execução no processador, através da mensagem AM_PUTPROCESS.

A rotina *StopProcess( )* notifica o ambiente de passagem de mensagem que alguns processos serão suspensos e solicita que o mesmo tome as medidas necessárias para que a migração possa ser realizada. Dentre essas medidas estão: (1) a criação de uma tabela de migração, responsável por manter o endereçamento antigo e novo de cada processo transferido, (2) atualização (remoção do processo) da

estrutura de dados que mantém a relação de processos no processador; (3) a suspensão temporária da comunicação realizada pelo processo, (4) confirmação de quais processos foram notificados e preparados com sucesso para a transferência, dentro do contexto do ambiente de passagem de mensagem.

A rotina *AcceptNewProcess( )* responsabiliza-se por dar ciência ao ambiente de passagem de mensagem que processos já iniciados estão sendo transferidos para este processador. A *AcceptNewProcess( )* cria novamente a configuração necessária para o processo dentro do ambiente de passagem de mensagem, restaura a comunicação deste com os outros processos e também com o ambiente de passagem de mensagem, assim como solicita as mensagens que tenham chegado para o processo no processador origem do mesmo, durante a migração.

A rotina *RouteMessageForNewAddress( )* corrige o endereçamento de uma mensagem destinada a um processo já transferido para outro processador. Essa rotina, executada no processador origem, também envia para o processador destino, quando solicitado, as mensagens que possam ter chegado durante a migração. A maneira como a atualização dos endereços, tanto nos processos quanto no ambiente de passagem de mensagem é dependente da política de atualização que se deseja utilizar. Optou-se, portanto, em deixar essa questão atrelada à política de escalonamento responsável pela migração de processos.

As alterações realizadas no ambiente de passagem de mensagem, responsáveis pela migração de processos, também independem da política de migração adotada. O objetivo aqui é viabilizar uma estrutura básica que se adapte a um número significativo de políticas com migração de processos e que através delas os processos sejam transferidos e a integridade dos mesmos seja mantida.

As alterações feitas no ambiente de passagem de mensagem para o AMIGO são sutis, mas decisivas para alterar a política de escalonamento usada. As rotinas inseridas na verdade atuam como um elo de ligação entre o ambiente de passagem de mensagem e o AMIGOD. Assim, para interagir com o AMIGOD é utilizada uma estrutura de dados própria do AMIGO e, ao interagir com o ambiente de passagem de mensagem, convertem-se os dados para a estrutura de dados adequada ao mesmo.

Considerando essas características, pode-se afirmar que adaptar o AMIGO para outro ambiente de passagem de mensagem, ou até mesmo outra ferramenta de software que necessite realizar escalonamento (como uma linguagem de programação concorrente ou até mesmo um sistema operacional), é necessário tão somente mudar as rotinas pertencentes ao AMIGO e incluídas no código do PVM. As rotinas do AMIGOD e das políticas de escalonamento se adaptarão automaticamente à mudança, porque o protocolo de comunicação entre os módulos permanecerá o mesmo.

3.6.4. As Políticas de Escalonamento

Uma das principais características do AMIGO é permitir que diferentes políticas possam ser implementadas sob o mesmo. Essas políticas compõem um módulo independente e podem apresentar características e objetivos diversos [Souza et al. (1999a) (1999b) (1999c)].

A separação das políticas de escalonamento em um módulo à parte apresenta a desvantagem de aumentar a comunicação entre processos existentes na camada inferior, mas possui a vantagem de viabilizar a flexibilidade e a característica dinâmica no AMIGO. Sendo a política um módulo à parte, a mesma pode ser substituída por outra de maneira transparente à aplicação e ao ambiente de passagem de mensagem, sendo que o AMIGOD se encarrega de encapsular essa mudança.

As duas principais características das políticas implementadas são: (1) a independência do ambiente de passagem de mensagem adotado, porque interagem somente com o AMIGOD e (2) a tendência por algoritmos específicos, uma vez que o objetivo é ter políticas pequenas, mas eficientes em determinadas situações.

Quando uma política está em funcionamento, ela é executada em cada processador onde o AMIGOD também está sendo executado. Como o AMIGOD não implementa nenhuma política de escalonamento, é necessário que em cada processador haja uma cópia da mesma para que possam ser efetivadas as decisões sobre o escalonamento. Embora o AMIGOD seja completamente distribuído, as políticas de escalonamento não necessitam ser distribuídas, podendo ser centralizadas. Nesse caso, um processador mestre (do ponto de vista da política) decide como o escalonamento será realizado e os outros processadores escravos apenas coletam informações sobre a carga computacional.

Para construir uma nova política de escalonamento para o AMIGO, o projeto da mesma deve considerar e se adaptar: (1) às regras de iniciação e finalização dos módulos da camada inferior, (2) ao protocolo de comunicação entre os módulos e (3) aos arquivos de configuração disponíveis através da camada superior e que contém informações como: hardware, *benchmarks*, entre outros.

A primeira política de escalonamento especificada e implementada para o AMIGO é a DPWP (*Dynamical Policy Without Preemption*), sendo o resultado de um mestrado desenvolvido no LASDPC (Laboratório de Sistemas Distribuídos e Programação Concorrente) do ICMC/USP. Os principais objetivos da DPWP são o balanceamento das cargas nos processadores e a redução do tempo de execução dos processos [Araújo (1999), [Souza et al. (1999c)].

Dentre as características da DPWP destacam-se: (1) ser dinâmica, utilizando informações em tempo de execução mais atualizadas e precisas, ao mesmo tempo que tenta minimizar as mensagens enviadas entre as políticas; (2) ser voltada para aplicações *CPU-Bound*, (3) não possuir migração de processos, e ser direcionada a plataformas com pouca variação das cargas computacionais em curtos intervalos de tempo, (4) ser objetiva e específica a um grupo de características, apresentando dessa forma um código pequeno e eficiente [Araújo (1999)].

A DPWP também contempla a heterogeneidade da plataforma computacional, possibilitando que as decisões possam considerar as diferenças existentes entre os processadores disponíveis. Dessa forma, além de considerar o índice de carga escolhido (número de processos prontos na fila), a DPWP também normaliza os valores obtidos com a potência computacional de cada processador. Informações mais detalhadas sobre as características e funcionamento da DPWP podem ser encontradas em Araújo (1999), Araújo et al. (1999a) e em Araújo et al. (1999b).

3.7. Utilização do AMIGO

A utilização do AMIGO está relacionada com as duas camadas: superior e inferior. Para utilizar a interface gráfica da camada superior e assim ter acesso às opções de configuração do escalonamento, o usuário executa o arquivo ***amigo*** existente no diretório AMIGOPATH/interface (o Apêndice B contém a estrutura de diretórios usada no AMIGO).

A atividade de escalonamento em si é realizada na camada inferior. Toda a camada inferior é iniciada através do ambiente de passagem de mensagem, o qual tão logo inicia a sua execução, também inicia o AMIGOD. Este, ao ser executado, deve iniciar as políticas de escalonamento determinadas na camada superior.

Todo o funcionamento da camada inferior é transparente à aplicação e também ao usuário que utilizar diretamente o ambiente de passagem de mensagem (como, por exemplo, para iniciar processadores). Embora o ambiente de passagem de mensagem tenha sido alterado para a interação, as mudanças não afetam a interface do ambiente de passagem de mensagem com seus usuários.

Para que os módulos da camada inferior localizem-se (por exemplo, o ambiente de passagem de mensagem consiga iniciar a execução do AMIGOD), é configurada a variável de ambiente AMIGOPATH, que indica a rota para o início da estrutura de diretórios do AMIGO. A partir dessa variável de ambiente os módulos conseguem encontrar, dentro da estrutura de diretórios padronizada, os arquivos executáveis, de configuração e todos os demais que sejam necessários.

3.8. Situação Atual do Projeto

O AMIGO é um projeto de pesquisa concebido neste trabalho de doutorado e que conta com a colaboração de pesquisadores do Grupo de Sistemas Distribuídos e Programação Concorrente do ICMC/USP, através de outros projetos de mestrado e de doutorado.

Nesta tese estão sendo especificadas as diversas características e funcionalidades tanto para a camada superior, quanto para a inferior. Para que resultados práticos pudessem ser viabilizados de maneira mais rápida, optou-se em dar prioridade (em um primeiro momento), à implementação da camada inferior [Souza et al. (1999c)]. Dessa forma pôde-se colocar o AMIGO em funcionamento sem o auxílio da camada superior, a qual ainda está por ser implementada.

A implementação da camada inferior definida nesta tese foi realizada no trabalho de mestrado que especificou e implementou a DPWP. Assim, o trabalho de mestrado

já concluído e desenvolvido por Araújo (1999) implementou: o AMIGOD, as alterações necessárias no PVM (para que o mesmo pudesse interagir como o AMIGO) e, finalmente, a própria DPWP, de acordo com os protocolos de comunicação do AMIGO. A implementação do AMIGOD feita por Araújo (1999), está pronta na sua quase totalidade. Ficaram faltando apenas itens alterados na especificação durante o desenvolvimento deste trabalho e que foram posteriores à conclusão do trabalho de mestrado de Araújo (1999).

O mestrado de Figueiredo (2000), encontra-se em fase adiantada de desenvolvimento e é responsável por desenvolver outra política de escalonamento para o AMIGO, agora para aplicações *I/O-Bound*. Além da nova política de escalonamento, o mestrado de Figueiredo (2000) também implementará as alterações necessárias no MPI (versão LAM). Dessa forma, as políticas que estiverem implementadas e disponíveis poderão ser executadas através de qualquer um dos dois ambientes de passagem de mensagem, PVM ou MPI. A implementação do AMIGO para o MPI (já em fase de conclusão) está comprovando a flexibilidade esperada em relação à sua implementação para diferentes ambientes de passagem de mensagem. Não houve a necessidade de alterar o código do AMIGO (inicialmente implementado para o PVM), sendo que as adaptações limitaram-se apenas ao código inserido no MPI e que viabiliza a interação entre ambos.

O mestrado de Campos (2000), em fase inicial de desenvolvimento, é responsável por desenvolver a interface gráfica da camada superior e também por verificar as facilidades e/ou dificuldades encontradas com os usuários do ambiente para utilizá-lo e/ou explorar o seu potencial. Esta análise, frente aos usuários, está vinculada à questão da lacuna existente entre a teoria e a prática do escalonamento.

Um dos objetivos deste projeto é justamente contribuir para a redução dessa distância.

O doutorado de Branco (2000), em fase inicial de desenvolvimento, é responsável por pesquisar o impacto do índice de carga no escalonamento de processos, principalmente em plataformas heterogêneas e multiusuárias. Embora o doutorado em questão não esteja vinculado diretamente a este projeto (por exemplo, através de alguma especificação e/ou implementação para o mesmo), os seus resultados serão completamente utilizados pelas políticas de escalonamento desenvolvidas.

O doutorado de Souza, M.A. (2000), em fase inicial de desenvolvimento, é responsável por pesquisar o monitoramento do escalonamento já realizado, fornecendo maneiras concretas e viáveis para que a atividade de distribuir processos aos processadores seja controlada e melhorada em tempo de execução. Além do estudo das métricas de desempenho, objetivos das políticas e técnicas de monitoramento, o trabalho de doutorado de Souza, M.A. (2000) fornecerá os subsídios necessários para que se decida sobre a substituição correta das políticas de escalonamento ativas.

3.9. Considerações Finais

O AMIGO apresentado neste capítulo é um ambiente de escalonamento que flexibiliza, de maneira dinâmica, as decisões tomadas na atribuição de processos aos processadores, em uma plataforma computacional distribuída.

Uma das suas principais características é a transparência. Espera-se fornecer transparência total aos usuários, tanto na camada inferior quanto na camada superior. Na camada inferior a transparência já existe, em função da sua utilização não ser

percebida pelos usuários, tanto sob o ponto de vista da aplicação, quanto sob o ponto de vista do ambiente de passagem de mensagem.

Neste primeiro instante do projeto, a camada superior não apresenta o mesmo grau de transparência. Alguns pontos como o cadastro de classes de software e de características das políticas de escalonamento podem ser automatizados através de técnicas de redes neurais que absorvam conhecimento com a utilização normal da plataforma. Usando técnicas como as de redes neurais é possível aumentar a qualidade das informações cadastradas e ao mesmo tempo desobrigar o usuário do preenchimento dessas informações.

O uso da camada superior requer que o usuário tenha conhecimento sobre a atividade de escalonar processos, o que de certa forma não contribui para diminuir a lacuna existente hoje entre a teoria e a prática na área. Não obstante, a atividade de configurar o escalonamento pode ser feita por um número bem reduzido de pessoas (provavelmente os administradores da plataforma) e os demais usuários podem usufruir, de maneira transparente, dos benefícios obtidos com o uso do AMIGO. Espera-se, em um futuro bem próximo, reduzir essa dependência utilizando para tanto técnicas automatizadas e inteligentes que captem as informações espalhadas pela plataforma.

Outra característica presente neste projeto é a portabilidade. Estão sendo utilizados recursos considerados padrões, como: o Unix, o protocolo TCP/IP, a linguagem C e os ambientes de passagem de mensagem, justamente para facilitar a execução em diversas plataformas. As adaptações necessárias para que o AMIGO possa ser executado em mais de uma plataforma simultaneamente, estão relacionadas mais diretamente à conversão de tipos de dados.

Considerando o quesito flexibilidade, o AMIGO possibilita que sejam inseridos os principais fatores que alteram o desempenho do escalonamento. O usuário pode inserir, com os cadastros, as informações que mais lhe convier. Por exemplo, no cadastro de classes de software, é possível inserir classes próprias, ao invés de utilizar classes genéricas e que podem não representar a realidade das execuções.

O fato do código fonte do AMIGO estar disponível, facilita a flexibilidade das políticas de escalonamento. Através da especificação das rotinas e das mensagens, diferentes políticas de escalonamento podem ser desenvolvidas e incluídas. Essa possibilidade não se limita apenas às políticas de escalonamento. A camada superior, bem como os outros módulos da camada inferior, têm os seus códigos fonte disponíveis para que alterações e/ou melhorias possam ser realizadas.

Além das atividades inerentes ao escalonamento de processos, permite-se o uso de outras atividades que melhoram a qualidade do escalonamento, mas não fazem parte da atividade de escalonar em si. O monitoramento e a simulação do escalonamento são exemplos dessas atividades no AMIGO. O monitoramento permite verificar de maneira *on-line* o escalonamento que está sendo realizado, fornecendo as informações necessárias para eventuais trocas de políticas de escalonamento. A simulação, por sua vez, possibilita analisar o comportamento das políticas de escalonamento antes das mesmas serem executadas. Analisando-se os inúmeros fatores que afetam o escalonamento, a simulação é uma ferramenta eficiente para a redução dos custos de desenvolvimento e/ou adaptação das políticas.

Considerando o exposto neste capítulo, conclui-se que o AMIGO é uma ferramenta de software inovadora, porque agrupa de maneira lógica e factível as grandes divergências encontradas hoje na área de escalonamento de processos. O uso de várias políticas de escalonamento agrupa duas características que normalmente

não caminham juntas: o desempenho e a abrangência. Não é uma tarefa trivial fornecer abrangência a um algoritmo de escalonamento (ser mais genérico) e ao mesmo tempo incumbi-lo de produzir bons resultados para todos os casos. Quanto mais abrangente um algoritmo de escalonamento é, mais sobrecarga ele produz e, conseqüentemente, menos desempenho ele terá. Com o AMIGO, vários algoritmos específicos fazem a mesma atividade de um algoritmo maior e genérico, mas com uma sobrecarga bem menor.

4. Avaliação de Desempenho com o AMIGO

4.1. Considerações Iniciais

O escalonamento altera o desempenho de aplicações executadas em plataformas computacionais distribuídas, tanto positivamente quanto negativamente. Considerando os conceitos envolvidos nessa atividade de escalonar processos e os objetivos propostos para o AMIGO, faz-se necessário avaliar o desempenho obtido por aplicações escalonadas sob a supervisão deste.

O objetivo deste capítulo, no entanto, não é a validação do AMIGO, e sim a demonstração de que é possível se obter ganhos de desempenho quando o mesmo é utilizado. Considerando que a qualidade do escalonamento é determinada pela política utilizada sob o AMIGO, e não por ele, os resultados descritos aqui são dependentes da política de escalonamento usada, no caso a DPWP. O AMIGO é o instrumento que viabiliza, além do uso de várias políticas de escalonamento, todas as demais características já discutidas.

As avaliações expostas neste capítulo demonstram o comportamento de aplicações paralelas executadas com e sem a utilização do AMIGO-DPWP, citando as melhorias de desempenho obtidas. Além da simples verificação de melhorias no

desempenho, também é analisado o impacto de fatores que afetam o escalonamento, tais como: diferentes classes de aplicação, carga de trabalho da plataforma, entre outros. No entanto, as avaliações limitam-se às porções do AMIGO atualmente implementadas. O enfoque dessas avaliações foi direcionado à camada inferior (utilizando-se a DPWP). Não foram verificadas questões como, por exemplo, o impacto da troca de políticas de escalonamento em tempo de execução, visto a falta de uma segunda política disponível e também devido à ausência da implementação da camada superior do AMIGO.

As execuções foram realizadas no Laboratório de Sistemas Distribuídos e Programação Concorrente (LASDPC) do Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (ICMC/USP). A plataforma utilizada é composta por computadores pessoais (PCs) (Tabela 4.1), conectados por uma rede padrão *ethernet* e utilizando o sistema operacional Linux (*kernel* 2.2.6).

Tabela 4.1 – Hardware utilizado para a avaliação de desempenho.

Nome	Processador	Velocidade	Memória
LASD03	Intel Pentium-MMX	200 MHz	32Mb
LASD04	Intel 486DX4	100 MHz	12Mb
LASD06	Intel Pentium	100 MHz	32Mb
LASD07	Intel Pentium-MMX	200 MHz	32Mb
LASD08	Intel Pentium	166 MHz	32Mb
LASD13	Intel Pentium II	400 MHz	128 Mb

Durante a avaliação de desempenho impediu-se o acesso externo à plataforma computacional através da sua rede de comunicação, eliminando assim a presença de multiusuários. Dessa maneira, as medições de tempo foram realizadas de forma controlada, variando-se desde uma plataforma ociosa até uma plataforma sobrecarregada. Para tanto, foram acrescentadas sistematicamente cargas de trabalho externas às aplicações executadas, simulando-se a presença de outros usuários.

4.2. Avaliações Realizadas

Considerando que a política de escalonamento utilizada é voltada para aplicações *CPU-Bound*, optou-se por três aplicações distintas: uma *CPU-Bound*, uma *I/O-Bound* e outra com ambas as características (*CPU* e *I/O-Bound*). A escolha por essas três classes permite que o desempenho obtido com o escalonamento seja avaliado em função da aplicação que o utiliza.

Como *CPU-Bound* foi utilizada a aplicação "Método dos Trapézios Compostos" definida em Cortés (1999). O método dos trapézios compostos é um típico algoritmo numérico que necessita de muito processamento e de pouca comunicação (vide Apêndice C). Nessa aplicação, uma integral é resolvida pela soma de outras ***n*** integrais, estas resolvidas em paralelo. A comunicação acontece quando um processo mestre envia para cada escravo a faixa de integrais a ser calculada e quando os escravos enviam para o mestre as integrais já calculadas. O método dos trapézios compostos foi executado com 1.200.000, 12.000.000 e 120.000.000 integrais.

A aplicação *I/O-Bound* foi representada por uma versão paralela do algoritmo de ordenação *quicksort*, desenvolvido em Souza (1996). Nessa versão paralela do *quicksort*, um vetor com ***n*** elementos é dividido em ***p*** processos escravos que ordenam em paralelo partes desse vetor (vide Apêndice C). Após ordenados, os subvetores são enviados para o processo mestre que os une novamente, gerando um único vetor ordenado. Em função de alguns fatores como: tamanho do vetor utilizado (306.000 elementos inteiros), rede de conexão utilizada (*ethernet* de 10Mbits) e o próprio algoritmo de ordenação, o tempo utilizado para o envio de cada subvetor é maior que o tempo gasto pelos escravos para ordenar cada um desses subvetores [Souza (1996)].

Para a execução da aplicação com características *CPU-Bound* e *I/O-Bound*, desenvolveu-se uma aplicação "sintética", onde foi possível variar o percentual do tempo total que a mesma passa realizando processamento e comunicação. O objetivo da aplicação é permitir a variação entre o tempo de comunicação e de processamento feitos durante a execução. A aplicação em questão define um vetor com ***n*** elementos inteiros os quais deverão ser enviados para processos escravos, e define também um número ***m*** de iterações que deverão ser realizadas tanto pelo processo mestre quanto pelos escravos. Variando-se os valores de ***n*** e ***m*** tem-se uma aplicação que pode ser tanto *CPU-Bound* quanto *I/O-Bound* (vide Apêndice C).

Na avaliação feita com as aplicações Método dos Trapézios Compostos e *Quicksort* Paralelo, além de se comparar o escalonamento feito pelo PVM com o escalonamento feito pelo AMIGO-DPWP, foram feitas outras execuções. Essas execuções estabeleceram nas aplicações o escalonamento a ser realizado (estaticamente, portanto), com o intuito de verificar a qualidade do escalonamento sugerido pelo AMIGO-DPWP. Nesse sentido, foram realizados sete escalonamentos estáticos diferentes para cada aplicação e os resultados comparados tanto com o escalonamento sugerido pelo PVM quanto pelo AMIGO-DPWP.

4.3. Técnica para Análise Estatística

O objetivo da análise estatística é verificar se as variações de desempenho observadas com os diferentes escalonamentos são estatisticamente significativas [Achcar; Rodrigues (1995)].

Dessa forma, para cada comparação entre dois valores são elaboradas hipóteses que devem ser comprovadas ou rejeitadas, isto é, essas hipóteses são passíveis de aceitação ou rejeição. A hipótese (H_1) utilizada nas comparações pode ser

exemplificada da seguinte maneira: "o desempenho da aplicação usando o AMIGO-DPWP é melhor que o desempenho da mesma aplicação usando PVM". Essa hipótese H_1 é utilizada quando os valores obtidos com o AMIGO-DPWP são menores que os obtidos com o PVM.

Sobre a hipótese H_1, também conhecida como hipótese alternativa, é anexada uma hipótese complementar, chamada de hipótese nulidade (H_0). A hipótese H_0 é a negação da hipótese H_1 [Achcar; Rodrigues (1995)].

Para se determinar estatisticamente qual hipótese pode ser aceita, tenta-se provar a hipótese nulidade. Caso não se consiga prová-la, a hipótese alternativa é aceita.

Para se realizar a análise estatística dos tempos coletados fazem-se, portanto, os seguintes testes de hipóteses:

- para amostras onde o tempo do AMIGO-DPWP < tempo do PVM:

  H_0: μAMIGO-DPWP ≥ μPVM

  H_1: μAMIGO-DPWP < μPVM

- para amostras onde o tempo do AMIGO-DPWP > tempo do PVM:

  H_0: μAMIGO-DPWP ≤ μPVM

  H_1: μAMIGO-DPWP > μPVM

onde: μAMIGO-DPWP e μPVM são as médias dos tempos de execução com o AMIGO-DPWP e com o PVM, respectivamente.

Considerando que os 50 tempos obtidos para cada média comparada possuem uma distribuição normal, a estatística dos testes de hipóteses acima é dada por:

$$Z = \frac{\bar{X}_{AMIGO-DPWP} - \bar{X}_{PVM}}{\sqrt{\frac{s^2_{AMIGO-DPWP}}{n_{AMIGO-DPWP}} + \frac{s^2_{PVM}}{n_{PVM}}}}$$

IFSC-USP SERVIÇO DE BIBLIOTECA E INFORMAÇÃO

onde: $\overline{X}_{AMIGO-DPWP}$ e $\overline{X}_{PVM}$ são as médias amostrais dos tempos obtidos;

$s^2_{AMIGO-DPWP}$ e s^2_{PVM} representam o desvio padrão amostral;

$n_{AMIGO-DPWP}$ e n_{PVM} representam o tamanho das amostras (no caso 50).

Para um nível de significância (α) igual a 0.01 (probabilidade de estar correto 99% das vezes que a análise estatística for feita), rejeita-se a hipótese nulidade quando Z ultrapassar o limite fornecido por $z_{0,01}$, o qual é 2.33. O valor de $z_{0,01}$= 2.33 é fornecido pela Tabela de Distribuição Normalizada [Achcar; Rodrigues (1995)]. Rejeita-se a hipótese nulidade H_0 caso $Z \leq -2.33$, ou então, $Z \geq 2.33$.

4.4. Análise dos Resultados Obtidos

A métrica utilizada nas avaliações de desempenho é a redução do tempo de execução da aplicação e a unidade de medida utilizada é o segundo. Cada valor expresso nos gráficos e tabelas desta seção representa a média de 50 execuções. Os valores inseridos nas tabelas em ***itálico*** e em ***negrito***, representam valores estatisticamente iguais, segundo a análise estatística descrita anteriormente.

Para todas as avaliações feitas, a ordem de inclusão dos processadores na máquina paralela virtual foi do processador com menor potência computacional para o processador com maior potência, a saber: lasd04, lasd06, lasd08, lasd07, lasd03 e lasd13 (Tabela 4.1). Considerando que as aplicações executadas possuem organização mestre x escravo, os processos "mestre" de cada aplicação sempre foram executados no processador lasd04 e os escravos (nove para cada aplicação) distribuídos pelos processadores segundo a política: (1) *round-robin* no PVM e (2) DPWP no AMIGO.

4.4.1. Desempenho com o AMIGO em uma aplicação *CPU-Bound*

A aplicação Método dos Trapézios Compostos foi executada inicialmente em uma plataforma sem nenhuma carga de trabalho externa (plataforma ociosa). Nessa execução variou-se apenas o peso da própria aplicação, realizando-se 1.200.000, 12.000.000 e 120.000.000 divisões (Tabela 4.2 e Gráfico 4.1). Analisando-se os valores obtidos com essa execução percebe-se que o AMIGO-DPWP conseguiu reduzir o tempo de execução da mesma aplicação paralela em aproximadamente 45% para cada uma das três variações.

Tabela 4.2 – Execução da aplicação Método dos Trapézios Compostos em uma plataforma ociosa, com 10 processos (1 mestre e 9 escravos) e 6 processadores.

Número de divisões =>		1.200.000	12.000.000	120.000.000
PVM	Média (50)	2,180	21,405	213,545
	Variância	0,000	0,001	0,003
	Desvio Padrão	0,009	0,027	0,059
AMIGO-DPWP	Média (50)	1,207	11,573	115,980
	Variância	0,008	0,005	0,111
	Desvio Padrão	0,087	0,069	0,333
Ganho de desempenho (%)		44,63%	45,94%	45,69%

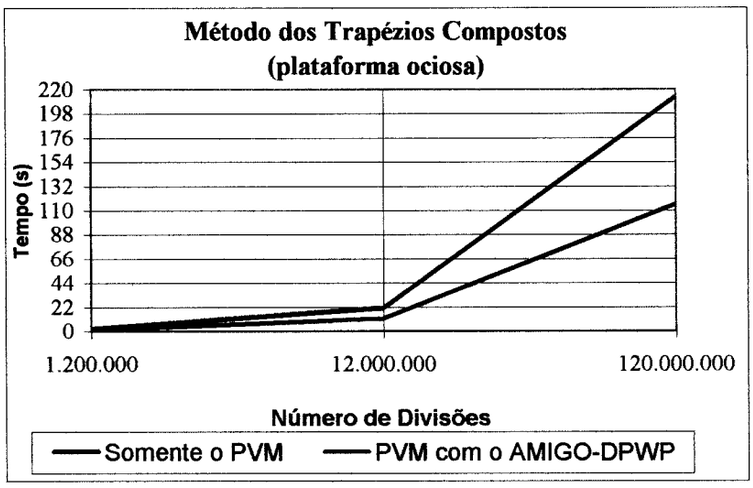


Gráfico 4.1- Tempo médio com a execução do Método dos Trapézios Compostos em uma plataforma ociosa, com 10 processos (1 mestre e 9 escravos) e 6 processadores.

Para analisar o comportamento do AMIGO-DPWP em relação às cargas externas a uma aplicação *CPU-Bound*, acrescentou-se até oito processos seqüenciais

(seriais) em cada um dos seis processadores disponíveis. Esses processos seriais realizam uma quantia predeterminada de operações matemáticas, consumindo para tanto um período de tempo variável em função do processador usado. Antes de reiniciarem as mesmas operações, os processos são colocados para "dormir" por um período igual ao do processamento realizado. Dessa forma, a quantia de tempo que o processo ocupa o processador e o período de tempo que ele é suspenso por estar "dormindo" é variável em função da potência do processador que o está executando. Outra característica desses processos seriais, é que os mesmos sobrecarregam apenas o processador em que estão executando e não a rede de comunicação.

Analisando-se os valores da Tabela 4.3 e do Gráfico 4.2, percebe-se que o AMIGO-DPWP continuou proporcionando ganho de desempenho em relação à mesma execução utilizando apenas o escalonamento do PVM. Esse ganho variou de no mínimo 8,01% para a execução com seis processos seriais concorrentes e de no máximo 28,88% para a execução com um processo serial concorrente (sem considerar o valor com a plataforma ociosa).

O ganho de desempenho foi menor (%) que o obtido com a plataforma ociosa. Acredita-se que isso tenha ocorrido em função da plataforma ter sido sobrecarregada de maneira uniforme, pois todos os processadores disponíveis receberam sistematicamente processos seriais para execução concorrente. Como a potência disponível na plataforma está dividida por vários processos (caminhando para a saturação da mesma), os benefícios alcançados com o AMIGO-DPWP também são menores, quando analisados apenas os percentuais de ganho de desempenho. No entanto, se considerado que: (1) é principalmente em plataformas sobrecarregadas que o escalonamento deve ser mais empregado e (2) mesmo com alta sobrecarga, o

AMIGO-DPWP encontrou uma distribuição dos processos que fornecesse melhoria de desempenho; esse ganho, embora percentualmente menor, é ainda significativo.

Analisando-se os ganhos percentuais que o AMIGO-DPWP proporcionou à aplicação, percebe-se uma queda nesses ganhos até a execução com seis processos seriais, respectivamente 45,94%; 28,88%; 11,60% e 8,01%. Com oito seriais houve novamente um aumento no ganho de desempenho, indo para 10,33%. Esse aumento percentual pode ser atribuído à maneira como a DPWP age em situações onde a plataforma está sobrecarregada. Considerando que cada uma das 50 execuções feitas pode apresentar um escalonamento diferente, acredita-se que a partir de oito processos seriais concorrentes houve uma mudança significativa no escalonamento, que proporcionou um aumento no ganho de desempenho para aplicações *CPU-Bound*. Se essas ponderações forem confirmadas, visto que os dados aqui não podem ser considerados conclusivos por não terem sido feitas execuções com mais processos seriais (10, 12, ou mais), essa característica será uma das grandes vantagens da DPWP.

Tabela 4.3 – Variação da carga da plataforma durante a execução da aplicação Método dos Trapézios Compostos com 12.000.000 de divisões, com 10 processos (1 mestre e 9 escravos) e 6 processadores.

Carga na Plataforma		ociosa	2 seriais	4 seriais	6 seriais	8 seriais
PVM	Média (50)	21,405	40,204	59,764	82,532	116,815
	Variância	0,001	7,242	27,962	9,935	0,765
	D. Padrão	0,027	2,691	5,288	3,152	0,874
AMIGO DPWP	Média (50)	11,573	28,592	52,831	75,922	104,744
	Variância	0,005	47,350	6,383	17,779	5,711
	D. Padrão	0,069	6,881	2,527	4,216	2,390
Ganho de desempenho (s)		9,832	11,612	6,933	6,610	12,071
Ganho de desempenho (%)		45,94%	28,88%	11,60%	8,01%	10,33%

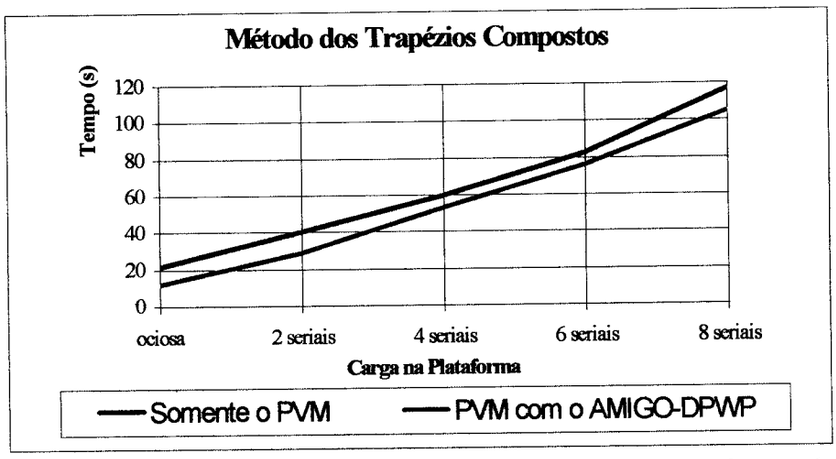


Gráfico 4.2 – Tempo médio com a variação da carga na plataforma durante a execução da aplicação Método dos Trapézios Compostos com 12.000.000 de divisões, com 10 processos (1 mestre e 9 escravos) e 6 processadores.

Para comparar a qualidade do escalonamento proposto pelo AMIGO-DPWP foram realizados outros sete escalonamentos diferentes, todos estáticos (determinados pela aplicação antes da sua execução). Nessas sete variações de escalonamento (Tabela 4.4) as atribuições foram feitas diretamente pelo PVM, sem utilizar a política *round-robin* do mesmo e também sem a execução do AMIGO-DPWP. A aplicação estava configurada para executar 12.000.000 de divisões em uma plataforma ociosa.

A 1ª variação do escalonamento teve por objetivo colocar um processo em cada processador e os 4 processos escravos excedentes no processador com maior potência computacional (lasd13). A 2ª variação distribuiu todos os processos escravos sobre dois dos processadores com maior potência computacional (lasd03 e lasd13). A 3ª variação mudou o local de execução do processo mestre, onde o mesmo deixou de ser executado no processador com menor potência computacional (lasd04) e passou para um dos processadores com a segunda maior potência computacional (lasd07). Nessa 3ª variação novamente distribuíram-se todos os processos escravos sobre dois dos processadores com maior potência computacional (lasd03 e lasd13). A

4ª variação fixou o processo mestre no processador lasd07 e atribuiu todos os processos escravos para o processador com maior potência computacional (lasd13). As variações de número 5, 6 e 7 distribuíram todos os processos (mestre e escravos), respectivamente: em um dos processadores com a segunda maior potência computacional (lasd03), no processador com a maior potência computacional (lasd13) e no processador com a menor potência (lasd04).

Tabela 4.4 – Sete variações de escalonamento realizadas nas aplicações para comparar a qualidade do escalonamento proposto pelo AMIGO-DPWP, usando 10 processos (1 mestre e 9 escravos) e um número variado de processadores (de 1 a 6 processadores).

Variações no Escalonamento	Processadores					
	lasd04	lasd06	lasd08	lasd07	lasd03	lasd13
1ª variação	1 mestre	1 escravo	1 escravo	1 escravo	1 escravo	5 escravos
2ª variação	1 mestre				4 escravos	5 escravos
3ª variação				1 mestre	4 escravos	5 escravos
4ª variação				1 mestre		9 escravos
5ª variação					1 mestre 9 escravos	
6ª variação						1 mestre 9 escravos
7ª variação	1 mestre 9 escravos					

Os valores da Tabela 4.5 e do Gráfico 4.3 demonstram que cinco das sete variações de escalonamento apresentam um desempenho melhor, quando comparadas à proposta de escalonamento feita pelo AMIGO-DPWP para a mesma aplicação em plataforma ociosa (conforme resultados da Tabela 4.3). Essa melhoria de desempenho pode estar associada a dois fatores. O primeiro refere-se à maior quantidade de informação que se tem quando o escalonamento é feito diretamente pela aplicação. Como a DPWP não utiliza nenhuma informação complementar à aplicação, a não ser que a mesma deva ser *CPU-Bound*, o seu escalonamento proposto não apresentou o melhor desempenho.

O segundo fator refere-se à presença na plataforma do processador lasd04, muito mais lento que os demais (Tabela 4.1), o qual executa o processo "mestre". Como a DPWP não implementa migração de processos e o escalonamento desse processo mestre é determinado pelo usuário, iniciando-o através da linha de comando do sistema operacional no processador lasd04, esse processador acaba por piorar o desempenho final da aplicação. As maiores melhorias de desempenho (3ª, 4ª e 6ª variações) não utilizam o processador lasd04. A 5ª variação, embora também não utilize o processador lasd04, apresentou uma perda de desempenho em relação ao escalonamento feito pelo AMIGO-DPWP, em virtude de executar todos os processos de uma aplicação *CPU-Bound* em um único processador e sem a potência computacional necessária para oferecer melhoria de desempenho.

Mesmo com as 3ª, 4ª e 6ª variações apresentando ganhos significativos de desempenho, deve-se ponderar que o uso do processador lasd04 nas execuções feitas com o AMIGO-DPWP não pôde ser alterado, pois foi fixado diretamente pelo usuário. Desse modo, as três variações que não utilizam o processador lasd04 para o escalonamento tiveram uma vantagem extra, a qual não pôde ser usada pelo AMIGO-DPWP.

Tabela 4.5 – Sete variações de escalonamento para a aplicação Método dos Trapézios Compostos, determinadas na aplicação de maneira estática, utilizando 10 processos (1 mestre e 9 escravos), 12.000.000 de divisões e processadores sem carga extra (plataforma ociosa). Os ganhos percentuais em relação ao PVM e em relação ao AMIGO-DPWP foram calculados com base no tempo médio para plataforma ociosa e com 12.000.000 de divisões, respectivamente 21,405s e 11,573s (vide Tabela 4.3).

	1ª var.	2ª var.	3ª var.	4ª var.	5ª var.	6ª var.	7ª var.
Média (50)	10,973	10,923	7,733	9,612	20,050	10,647	103,278
Variância	0,000	0,000	0,000	0,001	0,001	0,001	0,001
Desvio Padrão	0,013	0,014	0,005	0,024	0,023	0,025	0,038
Ganho % em relação ao PVM	48,74%	48,97%	63,87%	55,10%	6,33%	50,26%	-382,49%
Ganho % em relação ao AMIGO-DPWP	5,18%	5,61%	33,18%	16,94%	-73,25%	8,00%	-792,44%

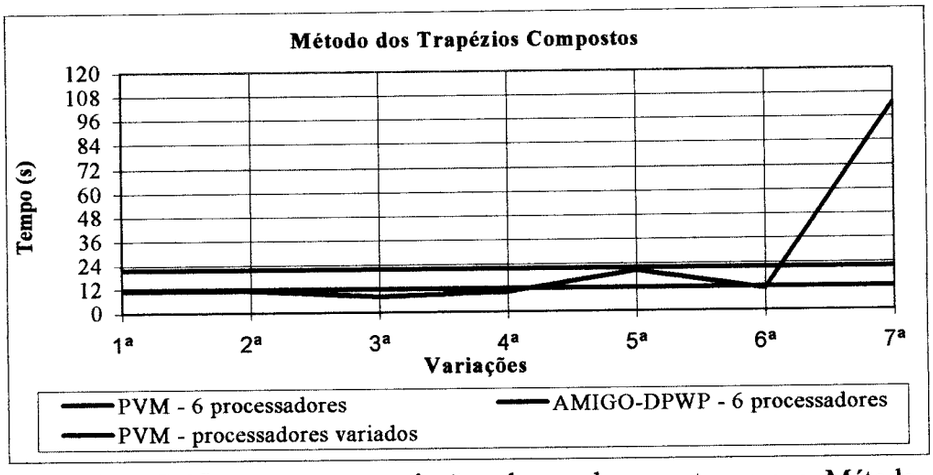


Gráfico 4.3 – Gráfico com sete variações de escalonamento para o Método dos Trapézios Compostos, determinadas de maneira estática, utilizando 10 processos (1 mestre e 9 escravos), 12.000.000 de divisões e plataforma ociosa. Os valores para "PVM – 6 processadores" e "AMIGO-DPWP – 6 processadores" representam o tempo para plataforma ociosa e com 12.000.000 de divisões, respectivamente 21,405s e 11,573s (vide Tabela 4.3).

4.4.2. Desempenho com o AMIGO em uma aplicação *I/O-Bound*

A aplicação "*Quicksort* Paralelo" foi utilizada para a ordenação de 306.000 elementos inteiros, os quais foram divididos em nove subvetores (um subvetor com 34.000 elementos para cada processo escravo).

A Tabela 4.6 e o Gráfico 4.4 demonstram os resultados obtidos com a plataforma ociosa e carregada com um número variado de processos seriais, de forma semelhante à execução da aplicação Método dos Trapézios Compostos. Os valores em ***negrito*** e em ***itálico*** em uma mesma coluna da Tabela 4.6 não apresentam diferenças sob o ponto de vista estatístico. Os valores mostram que o AMIGO-DPWP não proporcionou o mesmo ganho de desempenho alcançado anteriormente com a aplicação *CPU-Bound*. Para a plataforma ociosa o AMIGO-DPWP apresentou uma perda de desempenho de 6,72%. Para a plataforma carregada com 2 e 4 processos seriais concorrentes não houve diferença significativa entre os valores comparados e somente com a plataforma carregada com 6 e 8

processos seriais é que o AMIGO-DPWP apresentou um ganho de desempenho estatisticamente superior, respectivamente de 8,38% e 9,46%.

Tabela 4.6 - Execução da aplicação *Quicksort* Paralelo em um vetor de 306.000 elementos inteiros, variando-se da carga da plataforma computacional, utilizando 10 processos (1 mestre e 9 escravos) e 6 processadores.

Carga na Plataforma		ociosa	2 seriais	4 seriais	6 seriais	8 seriais
PVM	Média (50)	4,249	***10,003***	***14,174***	19,165	24,945
	Variância	0,023	3,119	4,588	4,774	3,540
	D. Padrão	0,151	1,766	2,142	2,185	1,881
AMIGO DPWP	Média (50)	4,534	***9,666***	***14,271***	17,559	22,585
	Variância	0,039	1,672	5,088	16,291	2,872
	D. Padrão	0,197	1,293	2,256	4,036	1,695
Ganho de desempenho (s)		-0,285	0,337	0,097	1,606	2,360
Ganho de desempenho (%)		-6,72%	3,37%	-0,68%	8,38%	9,46%

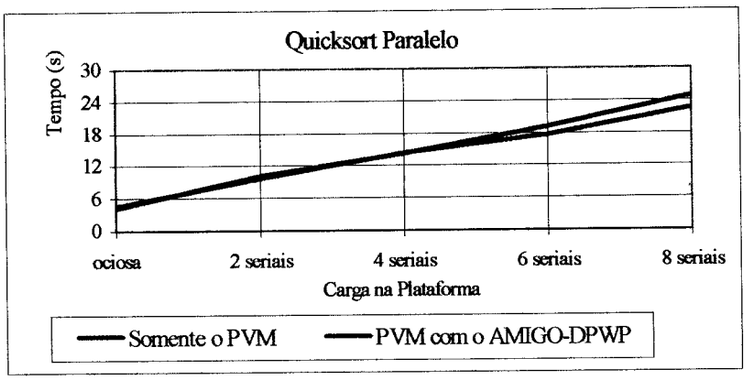


Gráfico 4.4 - Gráfico com a execução da aplicação *Quicksort* Paralelo em um vetor de 306.000 elementos inteiros, variando-se da carga da plataforma computacional, utilizando 10 processos (1 mestre e 9 escravos) e 6 processadores.

Através desse exemplo percebe-se que a DPWP não apresenta as características necessárias para realizar o escalonamento de processos para aplicações *I/O-Bound*, exemplificando a afirmação feita anteriormente neste trabalho, que uma única política de escalonamento não consegue proporcionar bons desempenhos para quaisquer tipos de aplicação.

O ganho de desempenho verificado com seis e com oito processos seriais é atribuído à grande sobrecarga que os processadores estão submetidos, aumentando dessa forma, apenas o tempo de processamento na relação processamento x

comunicação. Aumentando-se o tempo de processamento e não o de comunicação, os processos podem até passar a ser *CPU-Bound*, tornando-os mais apropriados à DPWP.

Considerando que o escalonamento proposto pelo AMIGO-DPWP não apresentou melhorias significativas no desempenho em relação à política *round-robin* do PVM, executou-se para o *Quicksort* Paralelo as mesmas sete variações de escalonamento discutidas anteriormente para o Método dos Trapézios Compostos (Tabela 4.4).

Os resultados expostos na Tabela 4.7 e no Gráfico 4.5 demonstram que uma política de escalonamento apropriada para aplicações *I/O-Bound* pode facilmente melhorar o desempenho alcançado com a DPWP. Variando-se apenas sete vezes o escalonamento, encontrou-se um ganho de desempenho de até 81,25% em relação ao escalonamento feito por ela (DPWP). Dessas execuções em particular, o melhor desempenho alcançado foi na sexta variação, onde todos os processos foram executados em um único processador (aquele com maior potência computacional). Considerando a grande heterogeneidade configuracional existente na plataforma e, principalmente, a característica *I/O-Bound* da aplicação, esse resultado pode ser considerado normal e demonstra mais uma vez que o escalonamento deve ser direcionado à aplicação a ser executada ou a um grupo destas. Outro fator a ser destacado é, novamente, a perda de desempenho obtida com a utilização do processador lasd04. Assim como verificou-se na aplicação Método dos Trapézios Compostos (porém aqui com um percentual maior), os maiores ganhos de desempenho foram nas execuções sem o uso desse processador (3ª, 4ª, 5ª e 6ª variações).

Tabela 4.7 – Sete variações de escalonamento para o *Quicksort* Paralelo, determinadas de maneira estática, utilizando 10 processos (1 mestre e 9 escravos), vetor com 306.000 elementos inteiros e plataforma ociosa. Os ganhos percentuais em relação ao PVM e em relação ao AMIGO-DPWP foram calculados com base na média para plataforma ociosa, respectivamente 4,249s e 4,534s (vide Tabela 4.6).

	1ª var.	2ª var.	3ª var.	4ª var.	5ª var.	6ª var.	7ª var.
Média (50)	4,364	4,279	2,912	2,941	1,800	0,850	4,858
Variância	0,004	0,002	0,005	0,006	0,003	0,000	0,016
Desvio Padrão	0,064	0,047	0,068	0,077	0,054	0,017	0,127
Ganho % em relação ao PVM	-2,71%	-0,73%	31,45%	30,77%	57,64%	79,99%	-14,36%
Ganho % em relação ao AMIGO-DPWP	3,76%	5,61%	35,76%	35,13%	60,31%	81,25%	-7,16%

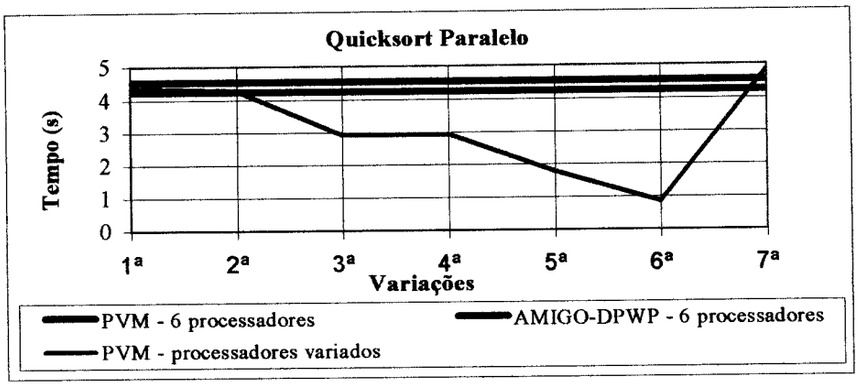


Gráfico 4.5 – Gráfico com as sete variações de escalonamento para o *Quicksort* Paralelo, determinadas de maneira estática, utilizando 10 processos (1 mestre e 9 escravos), vetor com 306.000 elementos inteiros e plataforma ociosa. Os valores para "PVM – 6 processadores" e "AMIGO-DPWP – 6 processadores" representam a média para plataforma ociosa, respectivamente 4,249s e 4,534s (vide Tabela 4.6).

4.4.3. Desempenho em uma aplicação *I/O* e *CPU-Bound*

As duas aplicações anteriores exemplificam os ganhos e perdas de desempenho de um escalonamento frente a dois extremos: uma aplicação com muito processamento e pouca comunicação e outra com pouco processamento e muita comunicação.

Esta seção demonstra o impacto do escalonamento sobre diferentes componentes de uma mesma aplicação, configurada para apresentar ambas as características de *I/O-Bound* e/ou de *CPU-Bound*. Para tanto, acertou-se o tempo total de execução em relação ao PVM, dividindo esse tempo em tempo de comunicação e em tempo de

processamento. Foram criados nove grupos de execuções, variando-se desde 90% para comunicação e 10% para processamento, até 10% de comunicação e 90% para processamento (Tabela 4.8 e Gráfico 4.6).

Tabela 4.8 –Resultados obtidos com a execução de uma aplicação com ambas as características *I/O-Bound* e *CPU-Bound*. As execuções tem como base os tempos obtidos para o PVM e assim são divididas em grupos desde 90% comunicação e 10% processamento até grupos com 10% comunicação e 90% processamento.

% Comunicação % Processamento		90% 10%	80% 20%	70% 30%	60% 40%	50% 50%	40% 60%	30% 70%	20% 80%	10% 90%
PVM	Comunicação	9,014	8,198	7,331	6,555	5,318	4,180	3,098	2,133	1,134
	Processamento	1,065	1,962	2,717	3,619	4,963	6,042	7,154	8,352	8,964
	Tempo Total	10,079	10,160	10,047	10,174	10,282	10,222	10,252	10,485	10,098
AMIGO DPWP	Comunicação	8,697	7,796	7,002	6,054	4,850	3,629	2,470	1,607	0,682
	Processamento	0,043	0,019	0,015	0,016	0,010	0,007	0,007	0,013	0,009
	Tempo Total	8,741	7,816	7,018	6,070	4,860	3,635	2,477	1,620	0,691
Ganho desempenho (%) Tempo de Processamento		95,94%	99,01%	99,44%	99,57%	99,79%	99,89%	99,90%	99,84%	99,90%
Ganho desempenho (%) Tempo Total		13,28%	23,07%	30,15%	40,34%	52,73%	64,43%	75,84%	84,55%	93,16%

Os valores expressos na Tabela 4.8 e no Gráfico 4.7 demonstram o comportamento do tempo de processamento quando as mesmas execuções são feitas com o escalonamento produzido pelo AMIGO-DPWP. A redução é expressiva sendo que o ganho de desempenho (considerando o tempo de processamento) é superior a 99% para quase todos os casos. Embora o tempo de processamento tenha sido reduzido, o tempo de comunicação permaneceu praticamente inalterado. Com essa separação observa-se o impacto do escalonamento em diferentes itens de uma mesma aplicação (como processamento e comunicação) e os benefícios que podem ser alcançados em cada um deles.

O Gráfico 4.8 e a Tabela 4.8 ilustram o comportamento do tempo total de execução da aplicação no PVM e no AMIGO-DPWP. Considerando que o tempo de comunicação no AMIGO-DPWP passou a ser o (maior) responsável pelo tempo total

de execução, quanto menos comunicação a aplicação apresenta, menor fica o tempo total da mesma.

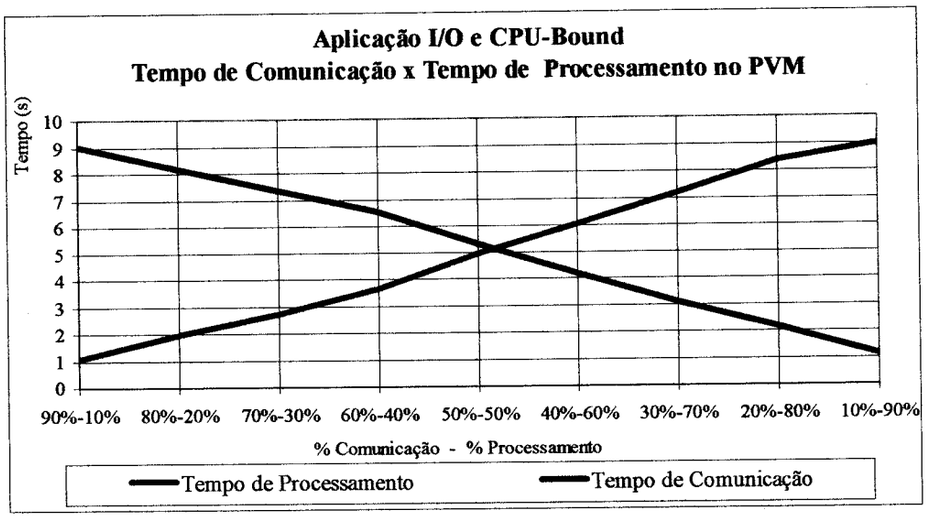


Gráfico 4.6 – Gráfico de uma aplicação *I/O* e *CPU-Bound* com a relação do tempo de comunicação x tempo de processamento para as execuções no PVM sem o AMIGO-DPWP, com 10 processos (1 mestre e 9 escravos) e 6 processadores.

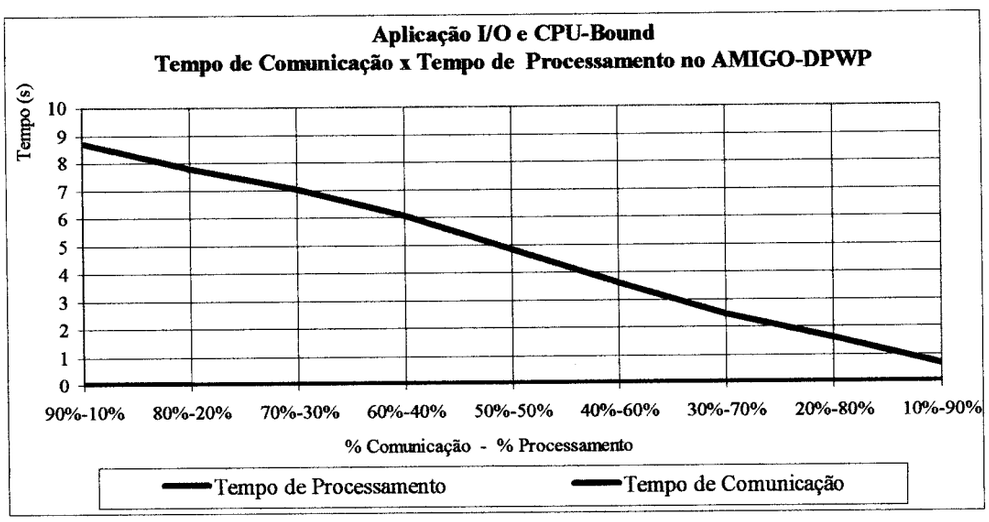


Gráfico 4.7 – Gráfico com o tempo de comunicação x tempo de processamento de uma aplicação *I/O* e *CPU-Bound* para as execuções com o AMIGO-DPWP, com 10 processos (1 mestre e 9 escravos) e 6 processadores.

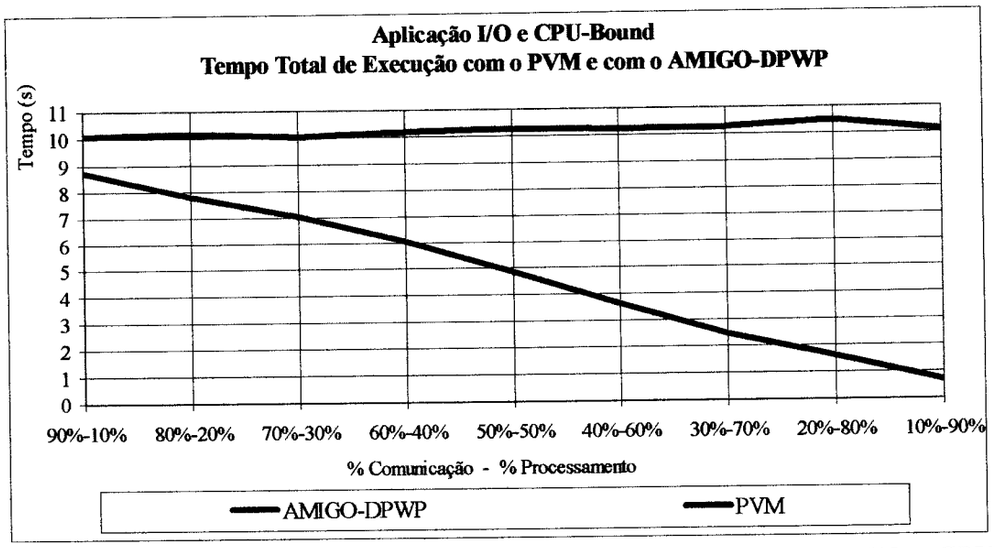


Gráfico 4.8 - Gráfico com o tempo total de execução de uma aplicação *I/O* e *CPU-Bound* para as execuções com o PVM e com o AMIGO-DPWP, com 10 processos (1 mestre e 9 escravos) e 6 processadores.

4.5. Considerações Finais

Os valores descritos aqui demonstram claramente que o AMIGO, em conjunto com a DPWP, não impedem ganhos reais de desempenho, ao contrário, permitem ganhos superiores a 99%, com a mesma aplicação e plataforma.

Considerando a carga de trabalho existente nos recursos utilizados, os resultados demonstram que o AMIGO-DPWP apresenta um comportamento estável mesmo em situações de alta carga, contribuindo para melhorar o desempenho, segundo a métrica escolhida: tempo total de execução.

Exemplificou-se também o impacto no desempenho com a utilização de diferentes classes de aplicação sob o mesmo escalonamento. Em aplicações *CPU-Bound*, o AMIGO-DPWP possibilitou melhorias significativas, porém, em aplicações *I/O-Bound* houve inclusive perda de desempenho.

A avaliação de desempenho com a aplicação *I/O* e *CPU-Bound* demonstrou o comportamento não apenas do resultado final, mas também em dois dos seus

componentes: a porção responsável pelo processamento e a responsável pela comunicação. Em virtude da DPWP ser específica para aplicações *CPU-Bound*, os ganhos de desempenho ficaram restritos à parte de processamento e a parte de comunicação permaneceu praticamente inalterada.

As avaliações de desempenho realizadas, embora parciais, demonstram o potencial existente em uma ferramenta de escalonamento de processos como o AMIGO. Empregando-se a política de escalonamento correta para os diversos fatores a serem considerados no momento da execução, pode-se melhorar o desempenho de uma aplicação de maneira significativa. Esses resultados, por serem parciais, não são conclusivos. À medida que novas porções do AMIGO estiverem disponíveis para execução, novos testes devem ser realizados. Dentre esses testes destacam-se aqueles relacionados: (1) à troca de políticas de escalonamento em tempo de execução; (2) à utilização concorrente de várias políticas de escalonamento; (3) à escolha transparente da política de escalonamento, entre outros.

5. Conclusões

5.1. Ponderações Finais desta Tese

Esta tese propôs uma maneira viável de equacionar alguns problemas existentes na atividade de se escalonar processos em plataformas distribuídas. O AMIGO, uma nova ferramenta de software para o escalonamento de processos foi proposta para inserir flexibilidade ao algoritmo de escalonamento de uma maneira dinâmica (em tempo de execução da aplicação) e transparente ao usuário.

A análise da bibliografia apresentada no Capítulo 2 demonstra que, apesar da importância e grande interesse pela área, sérios problemas de divergências são encontrados, principalmente na terminologia e nas taxonomias utilizadas. Essas divergências dificultam a comparação dos algoritmos de escalonamento, tornando árdua a tarefa de escolher o algoritmo de escalonamento correto a ser empregado. Os poucos trabalhos que visam organizar a área têm abrangência limitada e omitem características importantes para o escalonamento.

Ao apresentar as divergências da literatura, também são propostos pontos de convergência, destacando que algumas atividades semelhantes possuem termos distintos e que outras atividades distintas possuem termos semelhantes. Com o intuito de não aumentar as divergências existentes na literatura, nenhuma taxonomia

para a área foi proposta, porém, destacou-se como analisar as taxonomias já existentes e que não estão relacionadas diretamente.

O projeto do AMIGO (descrito no Capítulo 3) teve como princípios básicos a flexibilidade dinâmica (em tempo de execução) e a transparência. A flexibilidade dinâmica foi motivada pela necessidade de se incluir diferentes características ao escalonamento, em função deste ser influenciado por diversos fatores específicos ao momento da utilização (como a aplicação que será escalonada, a plataforma utilizada, o horário de utilização, entre outros). A transparência citada retira do usuário final a responsabilidade por aplicar e também por configurar as questões inerentes ao escalonamento. Dessa forma, nenhuma linha de código da aplicação necessita ser alterada para que o AMIGO passe a auxiliar no escalonamento dos processos.

Outra característica importante do AMIGO é a sua modularidade. A divisão em duas camadas separa a porção responsável por configurar o escalonamento (camada superior) da porção responsável por aplicar o escalonamento efetivamente (camada inferior). Com essa separação viabilizaram-se (de uma maneira simples) várias das características do AMIGO, como por exemplo: (1) a própria flexibilidade, pois enquanto a camada inferior aplica o escalonamento, a camada superior pode estar monitorando as atividades desenvolvidas e efetuar mudanças nas políticas de escalonamento que são empregadas, (2) a portabilidade, pois as duas camadas podem ser modificadas independentemente para atuarem em outras plataformas. No caso da camada inferior, a mesma pode ser adaptada para, por exemplo: outros ambientes de passagem de mensagem, linguagens de programação concorrente ou sistemas operacionais distribuídos.

O AMIGO ainda está em desenvolvimento. Outros trabalhos de mestrado e doutorado originados à partir da definição do projeto estabelecido nesta tese estão em andamento, cada qual estudando diferentes características do escalonamento e contribuindo de alguma forma para a implementação global do AMIGO. A DPWP é fruto de um desses trabalhos que, além de disponibilizar a primeira política de escalonamento, também desenvolveu o trabalho de implementação da camada inferior.

A avaliação de desempenho realizada (com as porções do AMIGO já implementadas) foi discutida no Capítulo 4 e demonstrou que, em conjunto com a política utilizada (a DPWP), o AMIGO pode fornecer ganhos de desempenho superiores a 99%, em relação às mesmas aplicações escalonadas pelo PVM. Em função da DPWP ser específica para aplicações *CPU-Bound*, foram executadas aplicações pertencentes a essa classe e também aplicações *I/O-Bound*. Os resultados obtidos demonstram que uma ferramenta de software como o AMIGO pode melhorar em muito o desempenho esperado para uma aplicação e/ou plataforma computacional. Além da questão desempenho, a transparência na aplicação do escalonamento feito pelo AMIGO pôde ser comprovada, pois as execuções das aplicações foram feitas exatamente da mesma maneira que seriam feitas se o mesmo não fosse utilizado.

Ao final desta tese conclui-se que aplicar corretamente o escalonamento de processos não é uma tarefa fácil. Essa dificuldade deve-se aos vários fatores que influenciam a escolha do escalonamento, às inúmeras propostas e à literatura divergente da área. Automatizando-se a atividade de escalonamento com uma ferramenta como o AMIGO, essa dificuldade é eliminada, pois é permitido utilizar a

terminologia e taxonomia desejadas, determinar o melhor escalonamento a ser utilizado e encapsular todas essas atividades, tornando-as transparentes aos usuários.

5.2. Contribuições desta Tese

Os trabalhos descritos nesta tese abordaram vários aspectos do escalonamento de processos em plataformas computacionais distribuídas, enfatizando uma maneira viável para que trabalhos teóricos fossem utilizados na prática. As principais contribuições apresentadas são:

- análise crítica da bibliografia existente, fornecendo um contexto geral para o escalonamento de processos em plataformas computacionais distribuídas. O texto também permite que os pesquisadores iniciantes previnam-se das divergências existentes e aprofundem posteriormente seus estudos de uma maneira correta e mais fácil;
- levantamento das divergências existentes com a terminologia da área, exemplificando-as e discutindo os motivos que levam a elas;
- descrição das principais taxonomias existentes. Mais do que simplesmente apresentar algumas taxonomias, foram escolhidas as mais representativas para diferentes segmentos dentro da área de escalonamento de processos, demonstrando como elas divergem em função dos objetivos e como elas podem convergir desde que analisadas detalhadamente;
- projeto do AMIGO, uma ferramenta de software de apoio ao escalonamento de processos em plataformas computacionais distribuídas que: (1) vem ao encontro dos problemas da área destacados nesta tese, (2) não é apenas mais um algoritmo de escalonamento, (3) viabiliza a união das mais diferentes propostas, maximizando as vantagens de cada uma e minimizando as suas desvantagens, (4)

torna a atividade de escalonar processos transparente e ao mesmo tempo eficiente ao usuário, (5) minimiza a dificuldade de se tratar todas as divergências encontradas na área, por permitir a inclusão da terminologia, taxonomia, e características que se deseja (e quando se deseja), (6) permite uma constante avaliação dos benefícios obtidos com o escalonamento e (7) possibilita mudanças das políticas de escalonamento em tempo de execução, adaptando-se dinamicamente às mudanças ocorridas na plataforma computacional, quer sejam de utilização ou de configuração;

- criação de uma nova linha de pesquisa no grupo de Sistemas Distribuídos e Programação Concorrente do ICMC/USP, sendo que novos trabalhos de mestrado e doutorado foram e estão sendo desenvolvidos a partir deste;
- convergência entre a teoria e a prática na área de escalonamento de processos, em função: (1) da utilização da análise crítica da bibliografia desenvolvida, a qual facilita a comparação entre diferentes propostas e (2) da utilização do AMIGO, com as características mencionadas acima.

5.3. Sugestões para Trabalhos Futuros

Devido à grande flexibilidade que o AMIGO proporciona ao escalonamento de processos, diferentes características e/ou situações podem ser investigadas em conjunto com a implementação do restante da ferramenta. Alguns dos principais projetos relacionados a este e que contribuem para a expansão dos resultados aqui obtidos são:

- implementação de outras políticas de escalonamento além da DPWP, para que seja possível: (1) investigar o impacto da flexibilidade no escalonamento, com o uso de diferentes políticas de escalonamento de maneira simultânea, (2) investigar

com maiores detalhes o uso do escalonamento em diferentes classes de software, plataformas, horários de utilização e outros. O trabalho em andamento de Figueiredo (2000) tem como um dos objetivos a implementação de uma nova política de escalonamento para aplicações *I/O-Bound*;

- estudo de políticas de escalonamento que contemplem a migração de processos, devido à importância dessas políticas para a área;
- estudo da portabilidade do AMIGO para outros ambientes de passagem de mensagem, como o MPI (em andamento no trabalho de Figueiredo (2000)). Além da utilização de outros ambientes de passagem de mensagem, também é possível estudar a utilização do AMIGO com outras "ferramentas" de software como sistemas operacionais distribuídos, CORBA (este último em andamento no trabalho de Santos (2000)) e linguagens de programação concorrente;
- estudo da portabilidade do AMIGO para plataformas que apresentem heterogeneidade na sua arquitetura (e não apenas na sua configuração), como por exemplo, unir as plataformas LINUX e SunOS;
- investigação do grau de satisfação (e de dificuldade) do usuário com o uso do AMIGO, que melhore o escalonamento de uma maneira transparente, viável e eficiente. O trabalho em andamento de Campos (2000) atua nessa linha e, para tanto, tem como um dos seus objetivos a implementação da camada superior;
- investigação de técnicas de monitoramento apropriadas para o escalonamento de processos e estudo da flexibilidade das métricas de desempenho aplicadas (em andamento no trabalho de Souza, M.A. (2000));
- inclusão no AMIGO de técnicas de simulação, para que o escalonamento possa, por exemplo, ser avaliado antes da sua utilização;

- investigação de diferentes índices de carga sobre diferentes políticas de escalonamento, aplicações e plataformas (em andamento no trabalho de Branco (2000));
- implementação de técnicas de redes neurais para a escolha da política de escalonamento apropriada, permitindo que a mesma seja feita de maneira inteligente e específica para os seus usuários;

Anexo A – Especificação das Mensagens da Camada Inferior do AMIGO

Neste Anexo estão relacionadas as mensagens trocadas entre os módulos na camada inferior do AMIGO. Para descrever as mensagens são detalhados: (1) a macro usada no campo do cabeçalho chamado "Código da Mensagem" e que estão inseridas em *message.h*, (2) os seus objetivos e (3) o seu conteúdo.

As mensagens estão separadas em três grupos principais, tomando-se como módulo central sempre o AMIGOD. Dessa forma, os grupos de mensagens criados são: (1) AMIGOD ⇔ Ambiente de Passagem de Mensagem, (2) AMIGOD ⇔ AMIGOD e (3) AMIGOD ⇔ Políticas de Escalonamento.

A.1. Mensagens entre o AMIGOD e o Ambiente de Passagem de Mensagem

A.1.1. Mudanças no número de processadores

AM_REM

objetivo: remoção de um AMIGOD

conteúdo: NULO

A.1.2. Escalonamento inicial de processos

AM_SCHED

objetivo: ambiente de passagem de mensagem local pede relação de possíveis receptores

conteúdo: nome do processo executável, instâncias a criar e a arquitetura

AM_SCHEDACK

objetivo: informa ao ambiente de passagem de mensagem local a relação de processadores receptores solicitada

conteúdo: número de processadores receptores, relação de processadores receptores para as tarefas

A.1.3. Migração de Processos

AM_GETPROCESS

objetivo: AMIGOD repassa pedido de relação de processos locais feito da política para o ambiente de passagem de mensagem.

conteúdo: nulo.

AM_PUTPROCESS

objetivo: AMIGOD recebe relação de processos locais vinda do ambiente de passagem de mensagem para a política.

conteúdo: número de processos existentes, relação de processos existentes (nome, pid, ...).

AM_STOPPROCESS

objetivo: AMIGOD repassa o pedido de aviso de suspensão de processos locais vindo da política para o ambiente de passagem de mensagem.

conteúdo: número de processos a suspender, relação com o pid dos processos a suspender.

AM_STOPPROCESSACK

objetivo: AMIGOD repassa a confirmação do aviso de suspensão de processos locais, vinda do ambiente de passagem de mensagem para a política.

conteúdo: número de processos notificados, relação com o pid dos processos notificados.

AM_PUTNEWPROCESS

objetivo: AMIGOD avisa o ambiente de passagem de mensagem da chegada de processos suspensos transferidos.

conteúdo: número de processos transferidos, relação de processos transferidos.

A.2. Mensagens entre os AMIGODs

A.2.1. Mudanças no número de processadores

AA_ADD

objetivo: inclusão de um AMIGOD

conteúdo: *fwd* para indicar se o conteúdo é para ser repassado para os outros processadores, relação de AMIGODs a incluir;

AA_REM

objetivo: remoção de um AMIGOD

conteúdo: número de processadores a excluir, relação de processadores a excluir

A.2.2. Atualização da política de informação

AA_GETLOAD

objetivo: pedido para obter cargas remotas

conteúdo: carga local para atualização remota

AA_PUTLOAD

objetivo: informa carga local para AMIGODs;

conteúdo: carga local para atualização remota

A.2.3. Mudança da política de escalonamento

AA_CHANGEPOL

objetivo: 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

objetivo: AMIGOD avisa demais AMIGODs depois de detectar "morte" de uma política.

conteúdo: política a excluir.

A.2.4. Política de Localização

AA_GETPART

objetivo: AMIGOD repassa pedido de parceiros

conteúdo: responsabilidade da política de localização

AA_PUTPART

objetivo: AMIGOD repassa relação de parceiros

conteúdo: responsabilidade da política de localização

A.2.5. Migração de Processos

AA_TRANSFPROCESS

objetivo: AMIGOD repassa o estado do processo suspenso para outro processador.

conteúdo: responsabilidade da política.

AA_TRANSFPROCESSACK

objetivo: AMIGOD repassa para outro processador remoto a confirmação vinda da sua política local que o processo ora transferido já está em execução neste processador.

conteúdo: responsabilidade da política.

AA_GETPROCESSRESIDUE

objetivo: AMIGOD repassa para outro AMIGOD o pedido dos resíduos do estado dos processos já transferidos.

conteúdo: responsabilidade da política.

AA_PUTPROCESSRESIDUE

objetivo: AMIGOD envia para outro AMIGOD, os resíduos que permaneceram neste processador de processos já transferidos.

conteúdo: responsabilidade da política.

A.3. Mensagens entre o AMIGOD e as Políticas

A.3.1. Política de informação

AP_GETLOADREMOTE

objetivo: política pede cargas remotas

conteúdo: carga local para atualização remota, número de processadores solicitados, relação de processadores

AP_PUTLOADLOCAL

objetivo: política informa sua carga

conteúdo: carga local, número de processadores, relação de processadores para informar a carga

AP_GETLOADLOCAL

objetivo: AMIGOD pede carga local

conteúdo: carga remota (do processador que solicitou a carga local)

AP_PUTLOADREMOTE

objetivo: AMIGOD informa carga remota

conteúdo: carga remota

A.3.2. Política de localização

AP_POLGETPART

objetivo: política de localização pede parceiros

conteúdo: nome do AMIGOD com a política coordenadora, número de parceiros desejados, tipo de parceiros (transmissores ou receptores). Responsabilidade da política de escalonamento.

AP_SCHEGETPART

objetivo: AMIGOD solicita parceiros

conteúdo: número de parceiros desejados, tipo de parceiros (transmissores/receptores)

AP_POLPUTPART

objetivo: política de localização informa parceiros

conteúdo: número de parceiros, relação de parceiros solicitados

AP_SCHEPUTPART

objetivo: AMIGOD informa parceiros

conteúdo: número de parceiros, relação de parceiros solicitados

A.3.3. Pedido de escalonamento inicial

AP_SCHED

objetivo: AMIGOD pede relação de receptores

conteúdo: nome executável, número de instâncias a criar e arquitetura

AP_SCHEDACK

objetivo: política informa relação de receptores

conteúdo: número de receptores, relação de possíveis receptores

A.3.4. Inclusão de uma política

AP_GETTABLE

objetivo: AMIGOD recebe pedido para enviar *amigod_table* para política.

conteúdo: nulo

AP_PUTTABLE

objetivo: AMIGOD envia a *amigod_table* à política.

conteúdo: número de processadores da relação, relação de processadores.

A.3.5. Remoção de uma política ativa

AP_REMPOL

objetivo: AMIGOD solicita que a política pare a sua execução.

conteúdo: nulo.

A.3.6. Remoção de um processador remoto

AP_REMHOST

objetivo: AMIGOD solicita que os processadores remotos inseridos na mensagem sejam eliminados das tabelas.

conteúdo: número de processadores, relação de processadores remotos a excluir.

A.3.7. Remoção de uma política remota

AP_POLICYDEATH

objetivo: AMIGOD remoto comunica perda de uma política.

conteúdo: nome (Pip:label) da política remota a eliminar das tabelas.

A.3.8. Migração de Processos

AP_GETPROCESS

objetivo: Política pede relação de processos locais para o Ambiente de Passagem de Mensagem através do AMIGOD.

conteúdo: nulo.

AP_PUTPROCESS

objetivo: AMIGOD repassa relação de processos locais para a política.

conteúdo: número de processos existentes, relação dos processos existentes (nome, pid, ...).

AP_STOPPROCESS

objetivo: Política avisa suspensão de processos locais para o Ambiente de Passagem de Mensagem.

conteúdo: número de processos a suspender, relação com o pid dos processos a suspender.

AP_STOPPROCESSACK

objetivo: AMIGOD repassa confirmação do aviso de suspensão dos processos locais, do ambiente de passagem de mensagem para a política.

conteúdo: número de processos notificados, relação com o pid dos processos notificados.

AP_TRANSFPROCESS

objetivo: Política transfere o estado dos processos (locais) suspensos.

conteúdo: Responsabilidade da política.

AP_TRANSFPROCESSACK

objetivo: Essa mensagem é usada em duas situações diferentes: (1) quando a política que recebeu o estado dos processos transferidos, confirma que os mesmos já estão em execução (aqui a mensagem é enviada da política para o AMIGOD) e (2) quando o AMIGOD, no processador que inicialmente enviou o estado do processo, repassa para a sua política que o processo em questão já está executando remotamente (aqui a mensagem é enviada do AMIGOD para a política).

conteúdo: Responsabilidade da política.

AP_PUTNEWPROCESS

objetivo: AMIGOD repassa o estado dos processos suspensos (remotos) para a política.

conteúdo: Responsabilidade da política.

AP_GETPROCESSRESIDUE

objetivo: Política pede resíduos remotos do estado dos processos já transferidos.

conteúdo: Responsabilidade da política.

AP_GETLOCALRESIDUE

objetivo: AMIGOD pede à política os resíduos que permaneceram neste processador, de um processo já transferido para outro processador.

conteúdo: Responsabilidade da política.

AP_PUTLOCALRESIDUE

objetivo: Política envia para o AMIGOD os resíduos de processos suspensos. Esses resíduos são repassados posteriormente para outro processador.

conteúdo: Responsabilidade da política.

AP_PUTPROCESSRESIDUE

objetivo: AMIGOD repassa para a política que solicitou, os resíduos de processos já transferidos anteriormente de um outro processador remoto para este.

conteúdo: Responsabilidade da política.

Anexo B – Especificação das Rotinas da Camada Inferior do AMIGO

Neste Anexo estão relacionadas as rotinas especificadas para o AMIGOD e as rotinas incluídas no código fonte do PVM, necessárias para adaptá-lo ao AMIGO. As rotinas estão separadas, portanto, em duas seções: "Rotinas do AMIGOD" e "Rotinas Incluídas no PVM".

B.1. Rotinas do AMIGOD

B.1.1. Iniciação e Roteamento de Mensagens do AMIGOD

Protótipo: *int main(int Argc, char **Argv)*

Objetivo: iniciar a execução do AMIGOD.

Descrição:

- recebe no máximo 2 argumentos pela linha de comando: uma *string* com o *IP:PORT TCP* para conexão com ambiente de passagem de mensagem local e pode receber também outra *string* com o *IP:PORT UDP* para conexão com um AMIGOD remoto, se este existir;
- inicia variáveis necessárias;

- cria IPCs para: ambiente de passagem de mensagem local – *TCP*;

  outros AMIGODs – *UDP*;

  políticas ativas – *TCP*;
- estabelece conexão *TCP* com ambiente de passagem de mensagem local;
- torna disponível o seu *PORT UDP* ao ambiente de passagem de mensagem local (atividade feita pelo *File System*);
- obtém o arquivo de configuração *policy.cfg* que contém a lista de políticas ativas (tarefa feita pela rotina *GetActivePolicy( )*);
- caso possua o *IP:PORT UDP* de algum outro AMIGOD (se não for o primeiro a ser executado), envia mensagem AA_ADD para esse AMIGOD para atualizar as tabelas de roteamento existentes nesses AMIGODs. Isso é feito através das rotinas *PutHost( )* e *GetHost( )*;
- executa rotina para iniciar a(s) política(s) ativa(s): *int StartPolicy( )*;
- executa rotina para troca de mensagens: *WorkUntilDeath( )*;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *void WorkUntilDeath (void)*

Objetivo: controla a entrada e saída de mensagens.

Descrição:

- recebe e envia mensagens para: ambiente de passagem de mensagem local ou política local ou AMIGOD remoto;
- desempacota e empacota mensagens;
- retira código da mensagem do corpo da mesma e chama rotina correta para tratar o conteúdo da mensagem;
- retransmissão de mensagens *UDP* perdidas;

- considera transmissão entre plataformas heterogêneas;
- armazena informações sobre o escalonamento no arquivo profile. (amigo.pfl)

B.1.2. Protocolo com o ambiente de passagem de mensagem local.

Na inclusão de um AMIGOD:

Torna disponível o seu *PORT UDP* (utilizado por outros AMIGOD) para o ambiente de passagem de mensagem local. Esse *port* poderá ser enviado a todo AMIGOD que for criado, através dos argumentos da linha de comando. Atividade feita através do *file system*, na rotina *main( )* do AMIGOD.

Na remoção de um AMIGOD:

Protótipo: *void RemoveLocalAMIGOD (void)*

Objetivo: suspender a execução do AMIGOD local e avisar todos os interessados. Depois de receber a mensagem AM_REM, chama a rotina *RemoveLocalAMIGOD( )* para cancelar o IPC com ambiente de passagem de mensagem.

Descrição:

- cancela IPC com o ambiente de passagem de mensagem;
- envia mensagem AA_REM para todos os AMIGODs existentes na sua tabela de roteamento;
- envia mensagem AP_REMPOL para todas as políticas ativas através da rotina RemovePolicies( );
- elimina variáveis existentes;
- finaliza execução.

No Escalonamento Inicial de Processos:

Protótipo: *int RequestReceivers(char *Message)*

Objetivo: receber pedido do ambiente de passagem de mensagem, solicitando receptores e repassar às políticas.

Descrição:

- repassa o pedido da mensagem AM_SCHED à política através da mensagem AP_SCHED;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int ReplyReceivers(char *Message)*

Objetivo: receber relação de receptores da política (mensagem AP_SCHEDACK) e repassar ao ambiente de passagem de mensagem.

Descrição:

- repassa relação de receptores ao ambiente de passagem de mensagem através da mensagem AM_SCHEDACK;
- retorna 0 se Ok e < 0 se houve algum problema.

Na migração de processos:

Protótipo: *int PutProcessTable (char *Message)*

Objetivo: AMIGOD recebe do ambiente de passagem de mensagem local a tabela de processos solicitada pela política e repassa a mensagem para a mesma. Depois de receber a mensagem AM_PUTPROCESS do ambiente de passagem de mensagem, o AMIGOD envia a mensagem AP_PUTPROCESS à política, contendo a relação dos processos existentes neste processador, solicitada pela própria política.

Descrição:

- repassa a relação de processos em execução através da mensagem AP_PUTPROCESS;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int StopProcessAck (char *Message)*

Objetivo: AMIGOD recebe do ambiente de passagem de mensagem local a confirmação de que a suspensão dos processos solicitada anteriormente pela política foi realmente realizada. Depois de receber a mensagem AM_STOPPROCESSACK do ambiente de passagem e mensagem, o AMIGOD envia a mensagem AP_STOPPROCESSACK à política, contendo a relação dos processos que realmente foram notificados da suspensão.

Descrição:

- repassa a relação de processos notificados no ambiente de passagem de mensagem através da mensagem AP_STOPPROCESSACK;
- retorna 0 se Ok e < 0 se houve algum problema.

B.1.3. Protocolo com os outros AMIGODs.

Na inclusão de um AMIGOD

Protótipo: *int PutHost(int NumElemDestHost, unsigned long *DestHost, char Fwd, int NumHostsInsert, unsigned long *HostsInsert)*

Objetivo: informa a existência (*PORT UDP*) dos processadores.

Descrição:

- empacota a mensagem AA_ADD contendo relação de processadores a inserir na tabela de roteamento dos processadores destinos e se os valores informados devem ser repassados (parâmetro *Fwd*) aos processadores existentes na tabela do processador destino;
- envia mensagem;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int GetHost(char * Message)*

Objetivo: receber relação informando a existência e o *PORT UDP* de novos processadores;

Descrição:

- desempacota a mensagem AA_ADD contendo relação de novos processadores que devem ser inseridos na tabela roteamento do processador local;
- insere na tabela;
- se *fwd* indicar, envia conteúdo recebido para todos os processadores previamente existentes na tabela, através da rotina *PutHost( )* e envia para o processador que solicitou (provavelmente um processador recém inserido no ambiente), a tabela completa, excluindo a indicação local e a do novo processador;
- se estiver incluindo o segundo processador da tabela (há apenas 2 processadores ativos no momento), retorna mensagem nula para o processador origem;
- retorna 0 se Ok e < 0 se houve algum problema.

Na remoção de um AMIGOD

Protótipo: *int RemoveRemoteAMIGOD(char *Message)*

Objetivo: AMIGOD é informado da perda de outro AMIGOD. Remove da tabela de roteamento local os processadores inseridos na mensagem.

Descrição:

- retira da mensagem AA_REM o número de processadores a excluir da tabela;
- retira da mensagem a relação de processadores;
- elimina da tabela de roteamento os processadores;
- envia mensagem para as políticas ativas avisando e pedindo a remoção dos processadores (mensagem AP_REMHOST);
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int DetectRemoteAMIGODDeath(char *HostName)*

Objetivo: AMIGOD detecta a perda de um AMIGOD Remoto. Elimina o processador remoto da tabela de roteamento e avisar os interessados.

Descrição:

- elimina processador remoto da tabela de roteamento;
- envia mensagem AA_REM para todos AMIGODs existentes, informando a perda do processador;
- retorna 0 se Ok e < 0 se houve algum problema.

Na atualização da Política de Informação.

Protótipo: *int AMIGODGetLoadLocal(char *Message)*

Objetivo: AMIGOD solicita a carga local para a sua política. Depois de receber a mensagem AA_GETLOAD de um AMIGOD remoto (política remota solicita carga local através do AMIGOD), envia mensagem para a política local solicitando a carga atual do processador. Nessa mensagem pode estar a carga remota do processador que

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

solicitou a carga local. Código da mensagem para a política: AP_GETLOADLOCAL.

Descrição:

- retira da mensagem AA_GETLOAD a carga remota;
- envia mensagem AP_GETLOADLOCAL para a política e informa, se for válida, a carga remota;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int AMIGODPutLoadRemote(char *Message)*

Objetivo: AMIGOD informa carga remota. Depois de receber a mensagem AA_PUTLOAD (política remota informa sua carga através do AMIGOD remoto), envia mensagem para a política local informando a(s) carga(s) remota(s) recebida(s). Código da mensagem para a política: AP_PUTLOADREMOTE.

Descrição:

- retira da mensagem AA_PUTLOAD, a carga remota para ser atualizada;
- envia mensagem AP_PUTLOADREMOTE contendo carga remota para a política local;
- retorna 0 se Ok e < 0 se houve algum problema.

Na mudança da Política de Escalonamento.

Protótipo: *int ChangePolicy(char *Message)*

Objetivo: AMIGO solicita mudança nas políticas de escalonamento ativas

Descrição:

- retira da mensagem AA_CHANGEPOL o número de políticas a excluir e a relação de políticas a excluir;

- retira da mensagem AA_CHANGEPOL o número de políticas a incluir e a relação de políticas a incluir;
- executa a rotina *RemovePolicies ()* para paralisar as políticas necessárias;
- inicia a execução das novas políticas. Atividade feita através da rotina *StartPolicy()*;
- atualiza tabela de políticas;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int DetectRemotePolicyDeath(char *Pip:label)*

Objetivo: AMIGOD remoto detecta "morte" de uma política ativa e informa aos outros AMIGODs para que estes avisem suas políticas locais. Depois de receber a mensagem AA_POLICYDEATH, informando que uma política remota parou de funcionar, o AMIGOD local informa o ocorrido para a política local igual à paralisada remotamente.

Descrição:

- envia mensagem AP_POLICYDEATH para a política local, solicitando a exclusão da política remota das tabelas existentes;
- retorna 0 se Ok e < 0 se houve algum problema.

Na política de localização

Protótipo: *int AMIGODGetPartner(char *Message)*

Objetivo: AMIGOD remoto repassa pedido de parceiros da política remota. Depois de receber a mensagem AA_GETPART de um AMIGOD remoto pedindo uma relação de parceiros. O pedido é repassado para a sua política de localização pela mensagem AP_SCHEGETPART.

Descrição:

- envia mensagem AP_SCHEGETPART à política solicitando parceiros;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int AMIGODPutPartner(char *Message)*

Objetivo: repassar relação de parceiros solicitados ao AMIGOD remoto. Depois de receber a mensagem AA_PUTPART de um AMIGOD remoto, envia mensagem para a política local informando a relação de parceiros solicitados anteriormente. Código da mensagem: AP_SCHEPUTPART.

Descrição:

- envia a mensagem AP_SCHEPUTPART com os parceiros;
- retorna 0 se Ok e < 0 se houve algum problema.

Na migração de processos

Protótipo: *int AMIGODTransfProcess(char *Message)*

Objetivo: AMIGOD transfere o estado de processos suspensos. Depois de receber a mensagem AA_TRANSFPROCESS de outro AMIGOD com o estado de processos suspensos remotamente, o AMIGOD local repassa a mensagem recebida para a política através da mensagem AP_PUTNEWPROCESS. O ambiente de passagem de mensagem local também é avisado da chegada do estado dos processos através da mensagem AM_PUTNEWPROCESS.

Descrição:

- envia mensagem AP_PUTNEWPROCESS à política;
- envia mensagem AM_PUTNEWPROCESS ao ambiente de passagem de mensagem;

- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int AMIGODTransfProcessAck(char *Message)*

Objetivo: AMIGOD repassa a confirmação da transferência do estado de processos suspensos. Depois de receber a mensagem AA_TRANSFPROCESSACK de outro AMIGOD confirmando a transferência anterior do estado de processos, o AMIGOD local repassa a mensagem recebida para a política através da mensagem AP_TRANSFPROCESSACK.

Descrição:

- envia mensagem AP_TRANSFPROCESSACK à política;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int AMIGODGetProcessResidue(char *Message)*

Objetivo: AMIGOD repassa pedido dos resíduos do estado de processos suspensos anteriormente e que estão neste processador. Depois de receber a mensagem AA_GETPROCESSRESIDUE de outro AMIGOD, pedindo os resíduos do estado de processos suspensos anteriormente neste processador, o AMIGOD local repassa a mensagem para a política através da mensagem AP_GETLOCALRESIDUE.

Descrição:

- envia mensagem AP_ GETLOCALRESIDUE à política;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int AMIGODPutProcessResidue (char *Message)*

Objetivo: AMIGOD repassa os resíduos do estado de processos suspensos em outro processador, os quais foram solicitados por este processador. Depois de receber a

mensagem AA_PUTPROCESSRESIDUE de outro AMIGOD, com os resíduos do estado de processos suspensos anteriormente e solicitados por este processador, o AMIGOD local repassa a mensagem para a política local através da mensagem AP_PUTPROCESSRESIDUE.

Descrição:

- envia mensagem AP_ PUTPROCESSRESIDUE à política;
- retorna 0 se Ok e < 0 se houve algum problema.

B.1.4. Protocolo com as políticas ativas.

Na Inclusão de Políticas

Protótipo: *int StartPolicy (int NumberPolExec, char ** PolName, char * PORT TCP)*

Objetivo: iniciação da política ativa

Descrição:

- executa política(s) solicitada(s) nos parâmetros;
- aguarda pedido de conexão *TCP* vindo da política local;
- retorna 0 se Ok e < 0 se houve algum problema.

Na Remoção de Políticas

Protótipo: *int RemovePolicies(int NumberPol, char *Policies)*

Objetivo: enviar mensagens para as políticas locais inseridas na relação para que elas parem a sua execução.

Descrição:

- verifica se todas as políticas locais relacionadas são válidas (estão executando);
- envia mensagem AP_REMPOL para cada uma das políticas;

- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int DetectPolicyDeath (void)*

Objetivo: enviar mensagens para as políticas remotas que constam na relação de AMIGODs remotos, solicitando a remoção da política local das suas tabelas. Depois do AMIGOD detectar a "morte" de uma política local, envia mensagem AA_POLICYDEATH para os outros AMIGODs, informando a perda da política e solicitando a exclusão dessa política local das tabelas no processador remoto. O AMIGOD local continua funcionando, pois podem existir outras políticas ativas. As políticas remotas semelhantes à política que parou de executar, continuam a sua execução, excluindo, entretanto, as suas referências sobre a política paralisada.

Descrição:

- remove o IPC existente com a política morta;
- envia mensagem AA_POLICYDEATH para os AMIGODs existentes, comunicando a "morte" da política neste processador;
- retorna 0 se Ok e < 0 se houve algum problema.

Na Política de Informação

Protótipo: *int PolGetLoadRemote(char *Message)*

Objetivo: atualizar política local com cargas remotas. Recebe mensagem da política local pedindo a carga de outros processadores (todos ou os especificados no vetor). Essa mensagem também pode conter a carga local atual para ser informada aos outros AMIGODs e/ou ficar no cache do AMIGOD. Código da mensagem: AP_GETLOADREMOTE.

Descrição:

- retira a carga local da mensagem AP_GETLOADREMOTE;
- retira da mensagem o número de processadores solicitados;
- retira da mensagem a relação de processadores;
- se a carga local é válida, insere-a no cache;
- envia mensagem AA_GETLOAD para cada processador relacionado, solicitando a carga destes e informando, se houver, a carga local;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int PolPutLoadLocal(char *Message)*

Objetivo: receber carga local e passá-la aos AMIGODs desejados. Receber uma mensagem da política pedindo que a carga local seja enviada para outros processadores (todos ou os especificados no vetor) e/ou fique no cache do AMIGOD. Código da mensagem vinda da política: AP_PUTLOADLOCAL. Código da mensagem a ser enviada aos AMIGODs : AA_PUTLOAD.

Descrição:

- retira da mensagem AP_PUTLOADLOCAL: (carga local, número de processadores a informar e relação de processadores);
- insere a carga local no cache do AMIGOD local;
- envia a carga local através da mensagem AA_PUTLOAD para os processadores relacionados;
- retorna 0 se Ok e < 0 se houve algum problema.

Na Política de Localização;

Protótipo: *int PolGetPartner(char *Message)*

Objetivo: repassa pedido de parceiros ao AMIGOD correto. Recebe mensagem da política local pedindo uma relação de parceiros. Código da mensagem: AP_POLGETPART. O AMIGOD envia a solicitação para outro AMIGOD remoto (mensagem AA_GETPART).

Descrição:

- repassa o pedido ao AMIGOD coordenador, através da mensagem AA_GETPART;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: int *PolPutPartner(char *Message)*

Objetivo: informar à outra política a relação de parceiros. Recebe mensagem da política local informando a relação de parceiros solicitados. Código da mensagem: AP_POLPUTPART. O AMIGOD repassa a relação para outro AMIGOD remoto (mensagem AA_PUTPART).

Descrição:

- repassa mensagem com código AA_PUTPART para a política remota, através do AMIGOD remoto;
- retorna 0 se Ok e < 0 se houve algum problema.

Nas Configurações feitas pelas políticas

Protótipo: *int PutTable(char *Message)*

Objetivo: política local pede para o AMIGOD a tabela com os processadores disponíveis. AMIGOD recebe a mensagem AP_GETTABLE vinda da política pedindo a tabela de roteamento (com os processadores disponíveis). O AMIGOD envia a tabela para a política através da mensagem AP_PUTTABLE

Descrição:

- prepara a mensagem AP_PUTTABLE com a quantidade de *IPs* enviados e a relação de *IPs*, excluindo-se o *IP* local;
- envia a mensagem AP_PUTTABLE;
- retorna 0 se Ok e < 0 se houve algum problema.

Na Migração de Processos

Protótipo: *int PolGetProcessTab(char *Message)*

Objetivo: política pede processos em execução. AMIGOD recebe a mensagem AP_GETPROCESS vinda da política pedindo ao ambiente de passagem de mensagem a relação de processos em execução no processador. O AMIGOD repassa a solicitação ao ambiente de passagem de mensagem através da mensagem AM_GETPROCESS.

Descrição:

- recebe a mensagem AP_GETPROCESS;
- envia a mensagem AM_GETPROCESS;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int PolStopProcess(char *Message)*

Objetivo: política avisa o ambiente de passagem de mensagem a suspensão de processos em execução. AMIGOD recebe a mensagem AP_STOPPROCESS vinda da política avisando o ambiente de passagem de mensagem que os processos constantes na relação serão suspensos para migração. O AMIGOD repassa a solicitação ao ambiente de passagem de mensagem através da mensagem AM_STOPPROCESS.

Descrição:

- recebe a mensagem AP_STOPPROCESS;
- envia a mensagem AM_ STOPPROCESS;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int PolTransfProcess(char *Message)*

Objetivo: política envia o estado dos processos suspensos para outro processador. AMIGOD recebe a mensagem AP_TRANSFPROCESS vinda da política contendo o estado dos processos suspensos. O AMIGOD repassa a mensagem AA_TRANSFPROCESS para o AMIGOD no processador destino dos processos.

Descrição:

- recebe a mensagem AP _TRANSFPROCESS;
- envia a mensagem AA_TRANSFPROCESS;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int PolTransfProcessAck(char *Message)*

Objetivo: política que recebeu o estado dos processos suspensos envia a confirmação que a transferência do estado foi feita com sucesso. AMIGOD recebe a mensagem AP_TRANSFPROCESSACK vinda da política contendo a confirmação que a transferência do estado dos processos suspensos foi realizada com sucesso. O AMIGOD repassa a mensagem AA_TRANSFPROCESSACK para o AMIGOD no processador destino dos processos.

Descrição:

- recebe a mensagem AP _TRANSFPROCESSACK;
- envia a mensagem AA_TRANSFPROCESSACK;

- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int PolGetProcessResidue(char *Message)*

Objetivo: política que recebeu os processos suspensos solicita que sejam transmitidos os resíduos do estado desses processos. AMIGOD recebe a mensagem AP_GETPROCESSRESIDUE vinda da política, solicitando que lhe sejam enviados os resíduos do estado dos processos transferidos. O AMIGOD repassa a mensagem AA_GETPROCESSRESIDUE para o AMIGOD no processador onde os processos estavam antes da migração.

Descrição:

- recebe a mensagem AP_GETPROCESSRESIDUE;
- envia a mensagem AA_ GETPROCESSRESIDUE;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int PolPutLocalResidue(char *Message)*

Objetivo: política informa os resíduos existentes neste processador do estado dos processos já transferidos. AMIGOD recebe a mensagem AP_PUTLOCALRESIDUE vinda da política, solicitando que sejam enviados os resíduos remanescentes neste processador, pertencentes ao estado de processos já transferidos. O AMIGOD repassa a mensagem AA_PUTPROCESSRESIDUE para o AMIGOD no processador onde os processos estão atualmente.

Descrição:

- recebe a mensagem AP_PUTLOCALRESIDUE;
- envia a mensagem AA_PUTPROCESSRESIDUE;
- retorna 0 se Ok e < 0 se houve algum problema.

B.1.5. Interação com a camada superior do AMIGO

Na Obtenção do Arquivo de Configuração:

Protótipo: *char **GetActivePolicy(void)*

Objetivo: encontrar e ler o arquivo contendo relação de políticas ativas.

Descrição:

- verifica a existência do arquivo de configuração com o nome: policy.cfg;
- obtém relação de políticas ativas e demais informações;
- retorna estruturas de dados com as informações.

Nas Mudanças de Políticas Ativas:

Protótipo: *int ChangeActivePolicy(void)*

Objetivo: trata recebimento de sinal assíncrono vindo do AMIGO, configurando quais políticas devem estar ativas.

Descrição:

- obtém novamente o arquivo de configuração através da rotina *GetActivePolicy( )*;
- verifica quais políticas atualmente ativas devem deixar de ser executadas e executa a rotina *RemovePolicies ( )* para paralisar as políticas necessárias;
- inicia a execução das novas políticas (aquelas que não estavam executando e que agora estão no arquivo de configuração). Atividade feita pela rotina *StartPolicy()*;
- envia mensagem AA_CHANGEPOL para os AMIGODs remotos informando quais políticas devem ser paralisadas e quais políticas devem ser iniciadas;
- atualiza tabela de políticas;
- retorna 0 se Ok e < 0 se houve algum problema.

Observação: a *ChangeActivePolicy( )* é executada quando é gerado um sinal assíncrono pela camada superior do AMIGO com a primitiva *kill(SIGUSR1, pid_AMIGOD)*, e devido à presença do comando: *signal(SIGUSR1, ChangeActive Policy)* no código fonte do AMIGOD.

B.2. Mudanças no feitas no PVM para a adaptação com o AMIGO

O PVMD pode receber mais um argumento pela linha de comando: o *IP:PORT UDP* de um AMIGOD remoto. Só receberá se já houver outro AMIGOD executando. O parâmetro será recebido com o prefixo -a (Ex:-*aIP:PORT*)

Rotinas alteradas: ***main( )*** do pvmd.c.

Protótipos das rotinas: amigod_pvm33.h

B.2.1. Configuração inicial

Protótipo: int *StartAMIGOD(void)*

Objetivo: criar o ambiente necessário para executar o AMIGOD.

Descrição:

- cria e inicia as variáveis necessárias ao AMIGOD;
- cria *IPC TCP* para AMIGOD;
- atribui *IP* para a variável *ip_local_AMIGOD* para ser usado nas futuras mensagens ambiente de passagem de mensagem ⇔ AMIGOD;
- executa AMIGOD, com estes parâmetros de entrada:

PORT TCP para conexão AMIGOD/ambiente de passagem de mensagem local;

se possuir o valor do *IP:PORT UDP* de um AMIGOD remoto, envia também este conteúdo;

- aguarda conexão *TCP*;
- recebe *PORT UDP* do AMIGOD local para comunicação AMIGOD/AMIGOD, através do *File System*. O *PORT UDP* recebido será armazenado e utilizado posteriormente em uma nova inclusão de um AMIGOD;
- retorna 0 se Ok e < 0 se houve algum problema.

Localização: no início da execução, na rotina main() do pvmd.c.

B.2.2. Inclusão de mais um processador

Quando for executar o novo processo *daemon* passa como parâmetro de linha de comando o *IP:PORT UDP* do AMIGOD local.

Rotinas alteradas: PVM => phase1( ) (startup.c) – acrescentar parâmetro - *aIP:PORT* quando estiver preparando os argumentos da linha de comando;

PVM => main( ) do pvmd.c - acrescentar uma opção ao case (case -a).

B.2.3. Exclusão de um processador

Protótipo: *int RemoveAMIGOD(void)*

Objetivo: desfazer o ambiente necessário para executar o AMIGOD

Descrição:

- envia mensagem com o código AM_REM para que o AMIGOD local possa se auto eliminar;

- cancela *IPC TPC* usado para comunicação ambiente de passagem de mensagem/AMIGOD;
- elimina as variáveis utilizadas pelo AMIGOD;
- retorna 0 se Ok e < 0 se houve algum problema.

Localização: início da rotina pvmbailout( ).

B.2.4. Escalonamento inicial de processos

Protótipo: *int GetHostsFromAMIGOD (Relação de processadores compatível com o ambiente de passagem de mensagem *Hosts, char *FileNameExec, int Copies, char *Arch)*

Objetivo: obter da política uma relação de processadores receptores para os processos

Descrição:

- prepara a mensagem AM_SCHED com o cabeçalho e com o seguinte conteúdo: nome do executável (FileNameExec), número de cópias para executar desse executável (*Copies*) e a arquitetura desejada (*Arch*);
- envia a mensagem AM_SCHED ao AMIGOD local;
- aguarda recebimento da mensagem AM_SCHEDACK com a relação de receptores (os *IPs* dos receptores);
- executada a rotina *iptohost( )* para localizar cada *IP* recebido na estrutura de dados que contém os processadores no ambiente de passagem de mensagem (no caso do PVM a *host_table*) e, a cada *IP* encontrado, copia este para a variável *hosts*;
- finaliza contendo em *hosts* os processadores desejados;
- retorna 0 se Ok e < 0 se houve algum problema.

Localização: executada em *tm_spawn( )* que está no arquivo tdproc.c e faz parte do PVMD

Protótipo: *Ponteiro para a localização do processador na tabela* iptohost(tabela de processadores do ambiente de passagem de mensagem* processors, int IP);*

Objetivo: encontrar na relação de processadores do ambiente de passagem de mensagem a localização do processador desejado pelo seu *IP*.

Descrição:

- recebe como parâmetros o endereço inicial da relação de processadores e o *IP* do processador desejado;
- localiza a posição do processador na relação pelo *IP*;
- se encontrar o processador pelo seu *IP*, retorna a posição do mesmo; caso contrário retorna *NULL*.

Localização: executada em *GetHostsFromAMIGOD( )* que está no arquivo tdproc.c e faz parte do PVMD.

B.2.5. Migração de processos

Protótipo: int *SendProcessPolicy(char *Message)*

Objetivo: depois de receber a mensagem AM_GETPROCESS (quando a política solicita os processos em execução), o ambiente de passagem de mensagem envia para o AMIGOD a relação de processos existentes neste processador pela mensagem AM_PUTPROCESS.

Descrição:

- obtém as informações disponíveis/necessárias para a relação de processos existentes, através da estrutura de dados específica do ambiente de passagem de mensagem (no caso do PVM a *task_table*);
- prepara a mensagem AM_PUTPROCESS com a relação;
- envia a mensagem para o AMIGOD;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: int *StopProcess (char *Message)*

Objetivo: quando a política avisa que processos serão suspensos, recebe do AMIGOD a relação de processos que serão suspensos para a futura migração. A suspensão dos processos é de responsabilidade da política e não do ambiente de passagem de mensagem. Cabe a este apenas ser notificado e preparar-se no que for necessário para a migração, criando toda a estrutura necessária para que o estado atual do processo possa ser interrompido no ambiente de passagem de mensagem e roteado para o novo processador.

Descrição:

- desempacota a mensagem AM_STOPPROCESS contendo o *pid* dos processos a serem suspensos e o endereço *IP* do novo processador que irá executar cada um dos processos;
- verifica se a estrutura de dados *migration_table* (responsável por registrar as migrações) já está criada. Se não estiver a mesma é iniciada;
- insere na *migration_table* as informações referentes aos processos que estão sendo suspensos;
- suspende a comunicação (direta entre os processos ou através do ambiente de passagem de mensagem) de cada processo a ser transferido até que o mesmo

esteja novamente em execução no novo processador. A maneira como é feita essa suspensão, depende da política usada;

- prepara a mensagem AM_STOPPROCESSACK com a relação dos processos que foram notificados e preparados com sucesso para a transferência;
- envia a mensagem para o AMIGOD;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: int *AcceptNewProcess (char *Message)*

Objetivo: quando o AMIGOD avisa a chegada de novos processos para execução neste processador, recebe do mesmo a relação dos processos que serão executados, solicitando que estes sejam incluídos nas estruturas de dados. Os estados desses processos foram enviados para a política responsável pela migração. Restabelece toda a configuração necessária para o processo, incluindo a comunicação via ambiente de passagem de mensagem.

Descrição:

- desempacota a mensagem AM_PUTNEWPROCESS contendo o número de processos transferidos e a relação dos mesmos;
- restabelece toda a configuração pertencente ao processo, inclusive a comunicação com outros processos (se necessário) e com o ambiente de passagem de mensagem;
- recupera as mensagens que possam ter chegado no processador de origem do processo enquanto a transferência estava em andamento;
- retorna 0 se Ok e < 0 se houve algum problema.

Protótipo: *int RouteMessageForNewAddress(void)*

Objetivo: faz a conversão de endereços nas mensagens enviadas para processos migrados anteriormente, quando posteriormente chegam mensagens para estes. A maneira como a mensagem será tratada é dependente da política que se deseja para o caso. A *RouteMessageForNewAddress( )* pode: (1) retransmitir a mensagem para o destino correto, (2) comunicar a troca de endereços para o processador origem da mensagem, solicitando dessa maneira que o mesmo envie a mensagem para o endereço correto ou (3) ambos, enviando a mensagem para o endereço correto e avisando a mudança ao processador transmissor da mensagem.

Descrição:

- encontra na estrutura de dados *process_migration* o endereçamento correto do processo;
- caso seja solicitado, retransmite a mensagem para o novo endereço;
- caso seja solicitado, transmite uma mensagem ao ambiente de passagem de mensagem no processador transmissor, solicitando que o endereço antigo seja substituído por outro;
- retorna 0 se Ok e < 0 se houve algum problema.

Referências Bibliográficas

ACHCAR, J.A.; RODRIGUES, J. *Introdução à Estatística para Ciências e Tecnologia.* Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo (ICMC/USP), 1995.

AGUILAR, J.; JIMENEZ, T. A Processors Management System for PVM. *Euro-Par' 97 Parallel Processing - Third International Euro-Par Conference, Lecture Notes in Computer Science v.1300*, Passau, Germany, agosto, 1997.

ALANYALI, M.; HAJEK, B. On Simple Algorithms for Dynamic Load Balancing. In: *INFOCOM'95*, Boston, Massachusetts, abril, 1995.

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

AL-SAQABI, K.; PROUTY, R.M.; MCNAMEE, D.; OTTO, S.W.; WALPOLE, J. Dynamic Load Distribution in MIST. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDTA 97)*, 1997.

ANDERSON, T.E.; CULLER, D.E.; PATTERSON, D.A. A Case for Now (Networks of Workstations). *IEEE Micro*, p.54-64, fevereiro, 1995.

ANTONIS, K.; GAROFALAKIS, J.; SPIRAKIS, P. A competitive symetrical transfer policy for load sharing. In: *Euro-Par' 98 Parallel Processing - Fourth International Euro-Par Conference, Lecture Notes in Computer Science v.1470*, Southampton, UK, p.352-355, setembro, 1998.

ARABE, J.; BEGUELIN, A.; LOWEKAMP, B.; SELIGMAN, E.; STARKEY, M.; STEPHAN, P. Dome: Parallel programming in a heterogeneous multi-user environment. *Technical Report CMU-CS-95-137*, School of Computer Science, Carnegie Mellon University, abril, 1995.

ARAÚJO, A.P.F. *DPWP - Uma Nova Abordagem para o Escalonamento Dinâmico em Computação Paralela Virtual.* Dissertação de Mestrado, Instituto de Ciências

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

Matemáticas e de Computação, Universidade de São Paulo (ICMC/USP), maio, 1999.

ARAÚJO(a), A.P.F.; SANTANA, M.J.; SANTANA, R.H.C.; SOUZA, P.S.L. A New Dynamical Scheduling Algorithm. In: *International Conference on Parallel and Distributed Processing Techniques and Applications – PDPTA'99*, Las Vegas, Nevada, U.S.A., junho, 1999.

ARAÚJO(b), A.P.F.; SANTANA, M.J.; SANTANA, R.H.C.; SOUZA, P.S.L. DPWP - A New Load Balancing Algorithm. In: *5th International Conference on Information Systems Analysis and Synthesis – ISAS'99*, Orlando, U.S.A., julho, 1999.

ARTSY, Y.; FINKEL, R. Designing a Process Migration Facility: The Charlotte Experience. *Computer*, v.22 n 9, p.47-56, setembro, 1989.

BARAK, A.; BRAVERMAN, A.; GILDERMAN, I.; LA'ADAN, O. Performance of PVM with the MOSIX Preemptive Process Migration. In: *Proceedings of the 7th Israeli Conf. on Computer Systems and Software Engineering*, Herzliya, p.38-45, junho, 1996.

BAUMGARTNER, K.M.; WAH, B. W. Computer Scheduling Algorithms: Past, Present and Future. *Information Sciences, Elsevier Science Pub. Co., Inc.*, New York, NY, v.57 & 58, p.319-345, setembro/dezembro, 1991.

BECKER, W. Dynamic balancing complex workload in workstation networks - challenge,concepts and experience. In: *High-Performance Computing and Networking, Lecture Notes in Computer Science v.919*, Milan, Italy, p.407-412, maio, 1995.

BEGUELIN, A.; SELIGMAN, E.; STEPHAN, P. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. *Technical Report CMU-CS-96-157*, School of Computer Science, Carnegie Mellon University, agosto, 1996.

BEMMERL, T. The TOPSYS Architecture. *In: Proceedings of CONPAR'90 VAp.IV, Lecture Notes in Computer Science v.457*, 1990.

BERMAN, P.; CHARIKAR, M.; KARPINSKI, M. On-line load balancing for related machines. In: *Algorithms and data structures, Lecture Notes in Computer Science v.1272*, p.116-125, 1997.

BERMAN, F.; STRAMM, B. Mapping Function-Parallel Programs with the Prep.-P Automatic Mapping Preprocessed. *Technical Report Number CS 94-397*, p.1-28, dezembro, 1994.

BESTAVROS, A. Load Profiling in distributed real-time systems. *Information Sciences*, v.101, Iss 1-2, p.1-27, 1997.

BILLARD, E.A.; PASQUALE, J.C. Load Balancing to adjust for proximity in some network topologies. *Parallel Computing*, 22, p.2007-2023, 1997.

BLUMOFE, R.D.; JOERG, C.F.; KUSZMAUL, B.C.; LEISERSON, C.E.; RANDALL, K.H.; ZHOU, Y.L. Cilk: na efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, v.37, Iss 1, p.55-69, 1996.

BOERES, C.; REBELLO, V.E.F.; SKILLICORN, D.B. Static scheduling using task replication for LogP and BSP models. In: *Euro-Par' 98 Parallel Processing - Fourth International Euro-Par Conference, Lecture Notes in Computer Science v.1470*, Southampton, UK, p.337-346, setembro, 1998.

BOUKERCHE, A.; DAS, S.K. Experimental studies in load balancing. In: *Euro-Par' 98 Parallel Processing - Fourth International Euro-Par Conference, Lecture Notes in Computer Science v.1470*, Southampton, UK, p.318-321, setembro, 1998.

BRANCO, K.R.L.J.C. *Índice de Carga e Desempenho em Ambientes Paralelos/Distribuídos - Modelos e Métricas.* Trabalho de doutorado em andamento no Instituto de Ciências Matemáticas de São Carlos, Universidade de São Paulo (ICMC/USP), 2000.

BREST, J.; ZUMER, V.; OJSTERSEK, M. Dynamic Scheduling on a Network Heterogeneous Computer System (Extend Abstract). In: *4th International ACPC Conference, Lecture Notes in Computer Science v.1557*, Salzburg, Austria, p.584-585, fevereiro, 1999.

BUBAK, M.; MOSCINSKI, J.; POGODA, M.; SLOTA, R. Load balancing for lattice gas and molecular dynamics simulations on networked workstations. In: *High-Performance Computing and Networking*, Lecture Notes in Computer Science v.919, Milan, Italy, p.329-334, maio, 1995.

CAMPOS, A. JR. *Interface Gráfica da Camada Superior do Ambiente AMIGO.* Trabalho de mestrado em andamento no Instituto de Ciências Matemáticas de São Carlos, Universidade de São Paulo (ICMC/USP), previsão de término para dezembro de 2000.

CANDEA, G.M.; JONES, M.B. Vassal: Loadable Scheduler Support for Multi-Policy Scheduling. *Technical Report MSR-TR-98-30.* Microsoft Corporation, Advanced Technology Division. Redmond, WA, USA, agosto, 1998.

CASAS(a), J.; CLARK, D.; KONURU, R.; OTTO, S.; PROUTY, R.; WALPOLE, J. MPVM: A Migration Transparent Version of PVM. *Technical Report CSE-95-002*, fevereiro, 1995.

CASAS(b), J.; CLARK, D.; GALBIATI, P.; KONURU, R. ; OTTO, S.; PROUTY, R.; WALPOLE, J. MIST: PVM with Transparent Migration and Checkpointing. In: *3rd Annual PVM Users' Group Meeting*, Pittsburgh, PA, maio, 1995.

CASAVANT; T.L.; KUHL, J.G. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, p.141-154, fevereiro, 1988.

CHANDRA, R.; GUPTA, A.; HENNESSY, J.L. Data locality and load balancing in cool. *Sigplan notices*, v.28, Iss 7, p.249-259, 1993.

CHEN, H-L.; KING, C-T. Eager Scheduling with Lazy Retry for Dynamic Task Scheduling. In: *Second International Euro-Par Conference - Euro-Par'96, Lecture Notes in Computer Science v.1124*, Lyon, France, p.615-620, agosto, 1996.

CHERKASOVA, L. Scheduling strategy to improve response time for web applications. In: *Proceedings of High-Performance Computing and Networking, Lecture Notes in Computer Science v.1401*, Amsterdan, The Netherlands, p.305-314, abril, 1998.

CHICLAYO, P.W.C.; HENRIQUES, M.A.A. Alocação de Recursos Computacionais em Modelo Econômico para Sistemas de Processamento Massivamente Paralelo Virtuais. In: *XXVI Seminário Integrado de Software e Hardware-SEMISH99, Anais do XIX Congresso Nacional da Sociedade Brasileira de Computação-SBC99*, Rio de Janeiro, p.211-220, julho, 1999.

CHIEN, Y.P.; ECER, A.; AKAY, H.U.; CARPENTER, F.; BLECH, R.A. Dynamic load balancing on a network of workstations for solving computational fluid-dynamics problems. *Computer Methods in Applied Mechanics and Engineering*, v.119, Iss 1-2, pp17-33, 1994.

CHRISTEN, P. Dynamic Load Balancing within a Parallel Iterative Linear System Solver. In: *Proceedings of High-Performance Computing and Networking, Lecture Notes in Computer Science v.1401*, Amsterdan, The Netherlands, p.857-859, abril, 1998.

CLARK, D.L.; CASAS, J.; OTTO, S.W.; PROUTY, R.M.; WALPOLE, J. Scheduling of Parallel Jobs on Dynamic, Heterogenous Networks. janeiro, 1995. *Disponível em: http://www.cse.ogi.edu/DISC/projects/cpe/*

COLAJANNI, M.; YU, P.S.; DIAS, D.M. Analysis of task assignment policies in scalable distributed web-server systems. *IEEE Transactions on Parallel and Distributed Systems*, v.9, Iss 6, p.585-600, 1998.

CORRADI, A.; LEONARDI, L.; ZAMBONELLI, F. Performance Comparison of Load Balancing Policies based on a Diffusion Scheme. In: *Euro-Par' 97 Parallel Processing - Third International Euro-Par Conference, Lecture Notes in Computer Science v.1300*, Passau, Germany, agosto, 1997.

CORRADI, A.; LEONARDI, L.; ZAMBONELLI, F. On the effectiveness of different diffuse load balancing policies in dynamic applications. In:

Proceedings of High-Performance Computing and Networking, Lecture Notes in Computer Science v.1401, Amsterdan, The Netherlands, p.274-283, abril, 1998.

CORTÉS, O.A.C. *Desenvolvimento e Avaliação de Algoritmos Numéricos Paralelos.* Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo (ICMC/USP), março, 1999.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. Distributed Systems: Concepts and Design. 2.ed., Wokingham, England, Addison-Wesley Publishing Company, 1994.

CZAJKOWSKI, K.; FOSTER, I.; KARONIS, N.; KESSELMAN, C.; MARTIN, S.; SMITH, W.; TUECKE, S. A Resource Management Architecture for Metacomputing Systems. In: *IPPS/SPDP' 98 Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science v.1459*, Orlando, Florida, USA, p.62-82, março, 1998.

DANTAS, M.A.R. *Efficient Scheduling of Parallel Applications on Workstation Clusters.* Tese de Doutorado apresentada à Faculdade de Engenharia da Universidade de Southampton, Reino Unido, setembro, 1996.

DANTAS, M.A.R.; ZALUSKA, E.J. Efficient Scheduling of MPI applications on Networks of Workstations. *Future Generation Computer Systems - FGCS*, v.13, p.489-499, 1998

DAVOLI, R.; BABAOGLU, Ö. Parallel Computing in Networks of Workstations with Paralex. *IEEE Transactions on Parallel and Distributed Systems*, v.7, n.4, abril, 1996.

DECKER, T. Virtual data space - A universal load balancing scheme. In: *Solving irregularly structured problems in parallel, Lecture Notes in Computer Science v.1253*, p.159-166, 1997.

DEVARAKONDA, M.V.; IYER, R.K. Predictability of Process Resource Usage: A Measurement-Based Study on UNIX. *IEEE Transactions on Software Engineering*, v.15, n.12, p.1579-1586, dezembro, 1989.

DIKENELLI, O.; OZKASAP, O.; OZKARAHAN, E. Scheduling Parallel Programs Involving Parallel Database Interactions. *In: 4th International Conference PaCT'97 Yaroslaul, Parallel Computing Tecnologies, Lecture Notes in Computer Science v.1277*, Rússia, setembro, 1997.

DONGARRA, J.J.; OTTO, S.W.; SNIR, M.; WALKER, D. A message Passing Standard for MPP and Workstations Communications of the ACM, v.39, n.7, julho, 1996.

DONGARRA, J.J.; GEIST, A.; KOHL, J.A.; PAPADOPOULOS, P.M.; SUNDERAM, V. HARNESS: Heterogeneous Adaptable Reconfigurable

Netorked SystemS. Relatório técnico *disponível em http://www.epm.ornl.gov/harness/*, março, 1998.

DOUGLIS, F. Experience with Process Migration in Sprite. In: *Distributed and Multiprocessor Systems Workshop Proceedings*, Fort Lauderdale, FL, p.59-72, outubro, 1989.

DOUGLIS, F.; OUSTERHOUT, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, v.21, n.8, p.757-785, agosto, 1991.

DOWNEY, A.B. Lachesis: A Job Scheduler for the Cray T3E. In: *IPPS/SPDP' 98 Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science v.1459*, Orlando, Florida, USA, p.47-61, março, 1998.

DOWNEY, A.; HARCHOL-BALTER, M. A note on 'The Limited Performance Benefits of Migrating Active Processes for Load Sharing. *Technical Report UCB/CSD-95-888*, University of California at Berkeley, 1995.

DURAND, D.; MONTAUT, T.; KERVELLA, L.; JALBY, W. Impact of memory contention on dynamic scheduling on NUMA multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, v.7, Iss 11, p.1201-1214, 1996.

EPEMA, D.H.J.; LIVNY, M.; DANTZIG, R.; EVERS, X.; PRUYNE J. A worldwide flock of Condors:Load sharing among workstation clusters. *Future Generation Computer Systems - FGCS*, 12, p.53-65, 1996.

ESKICIOGLU, M.R. Design Issues of Process Migration Facilities in Distributed Systems. *IEEE Technical Committee on Operating System Newsletter*, p.3-13, 1989.

FAGG, G.E.; LONDON, K.S.; DONGARRA, J.J. Taskers and General Resource Managers: PVM Supporting DCE Process Management. In: *Third European PVM Conference - EuroPVM'96; Lecture Notes in Computer Science v.1156*, Munich, Alemanha, outubro, 1996.

FEITELSON, D.G.; RUDOLPH, L.; SCHWIEGELSHOHN, U.; SEVCIK, K.C.; WONG, P. Theory and Practice in Parallel Job Scheduling. In: *IPPS' 97 Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science v.1291*, Santa Barbara, Geneva, Switzerland, abril, 1997.

FEITELSON, D.G.; RUDOLPH, L. Parallel Job Scheduling: Issues and Approaches. In: *IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science v.949*, Santa Barbara, CA, USA, abril, 1995.

FEITELSON, D.G.; RUDOLPH, L. Toward Convergence in Job Schedulers for Parallel Supercomputers. In: *IPPS' 96 Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science v.1162*, Santa Barbara, Honolulu, Hawaii, abril, 1996.

FERRARI, D.; ZHOU, S. An Empirical Investigation of Load Indices for Load Balancing Applications. In: *Proceedings of Performance'87, the 12th Int'l Symposium on Computer Performance Modeling, Measurement, and Evaluation*, p.515-528, 1987.

FERREIRA, A.B.H. *Novo Dicionário Aurélio da Língua Portuguesa.* 2.ed., Rio de Janeiro, Nova Fronteira S.A., 1986.

FERSTL, F. Job-and resource-management systems in heterogeneous clusters. *Future Generation Computer Systems - FGCS*, 12, p.39-51, 1996.

FIGUEIREDO, T.C. *Balanceamento de Carga com PVM, MPI e AMIGO.* Trabalho de mestrado em andamento no Instituto de Ciências Matemáticas de São Carlos, Universidade de São Paulo (ICMC/USP), previsão de término para 2000.

FLYNN, M.J. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, v.C-21, p.948-960, 1972.

FONLUPT, C.; MARQUET, P.; DEKEYSER, J.L. Data -parallel load balancing strategies. *Parallel Computing*, v.24, p.1665-1684, 1998.

FRANKLIN, M.A.; GOVINDAN, V. A general matrix iterative model for dinamic load balancing. *Parallel Computing*, v.22, p.969-989, 1996.

GEHRING, J. Dynamic Program Description as a Bassis for Runtime Optimization. Euro-Par' 97 Parallel Processing – In: *Third International Euro-Par Conference, Lecture Notes in Computer Science v.1300*, Passau, Germany, agosto, 1997.

GEIST, G.A.; BEGUELIN, A.; DONGARRA J.J.; JIANG, W; MANCHEK, R.; SUNDERAM, R. *PVM 3 Users Guide and Reference Manual.* Oak National Laboratory, setembro, 1994.

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. *Fourth European PVM-MPI Conference - Euro PVM/MPI'97, Lecture Notes in Computer Science v.1332*, Cracow, Polônia, p.116-126, novembro, 1997.

GENCO, A.; LORE, G. The egoistic approach to parallel process migration into heterogeneous workstation network. *Journal of Systems Architecture*, v.42, Iss 4, p.267-278, 1996.

GIBBONS, R. A Historical Application Profiler for Use by Parallel Schedulers. In: *IPPS' 97 Workshop on Job Scheduling Strategies for Parallel Processing,*

Lecture Notes in Computer Science v.1291, Santa Barbara, Geneva, Switzerland, abril, 1997.

GIL, J.; MATIAS, Y. An effective load balancing policy for geometric-decaying algorithms. *Journal of Parallel and Distributed Computing*, v.36, Iss 2, p.185-188, 1996.

GLAZEK, W. How To Share a Divisible Load in a Hypercube (Extend Abstract). *In: 4th International ACPC Conference, Lecture Notes in Computer Science v.1557*, Salzburg, Austria, p.588-589, fevereiro, 1999.

GRAHAM, P.C.J.; BARKER, K.E. Service Migration to Improve Load Balancing in Distributed Systems. *Technical Report TR94-04*, Dept. of Computer Science, University of Manitoba , abril 1994.

HA, S.H.; LEE, E.A. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, v.40, Iss 11, p.1225-1238, 1991.

HALANG, W.A.; JUNG, S.K.; KRÄMER, B. Enforcing Management Policies in Distributed Systems. In: *Proceedings of the Fith IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Cheju Island, Korea, p.474-480, agosto, 1995.

HARCHOL-BALTER, M.; CROVELLA, M.; MURTA, C. To Queue or Not to Queue: When Queueing is Better Than Timesharing in a Distributed System. *Technical Report BUCS-TR-1997-017*, outubro, 1997.

HARCHOL-BALTER, M.; DOWNEY, A. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Transactions on Computer Systems* v.15, n.3, agosto, 1997.

HENDERSON, R.L. Job Scheduling Under the Portable Batch System. In: *IPPS' 95 Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science v.949*, Santa Barbara, CA, USA, abril, 1995.

HILL, J.M.D.; DONALDSON, S. R.; LANFEAR, T. Process migration and fault tolerance of BSPlib programs running on networks of workstations. In: *Euro-Par' 98 Parallel Processing - Fourth International Euro-Par Conference, Lecture Notes in Computer Science v.1470*, Southampton, UK, p.80-91, setembro, 1998.

HU, Y.F.; BLAKE, R.J.; EMERSON, D.R. An Optimal Migration algorithm for Dynamic load balancing. *Concurrency: Practice and Experience*, v.10(6), p.467-483, 1998.

HUANG, G.; ONGSAKUL, W. An efficient load-balancing processor scheduling algorithm for parallelization of Gauss-Seidel Type Algorithms. *Journal of Parallel and Distributed Computing*, v.22, Iss 2, p.350-358, 1994.

HUI, C.C.; CHANSON, S.T. Flexible and extensible load balancing. *Software-practice & experience*, v.27, Iss 11, p.1283-1306, 1997.

HWANG, K.; XU, Z. *Scalable Parallel Computing*: Technology, Architecture, Programming. New York, McGraw-Hill Inc., janeiro, 1998.

ISLAM, N.; PRODROMIDIS, A.; SQUILLANTE, M.S. Dynamic Partitioning in Diferent Distributed-Memory Environments. In: *IPPS' 96 Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science v.1162*, Santa Barbara, Honolulu, Hawaii, abril, 1996.

JACKSON, D.J.; HUMPHRES, C.W. A simple yet effective load balancing extension to the PVM software system. *Parallel Computing*, v.22, p.1647-1660, 1997.

JONES, M.B.; RO| U, D.; RO| U, M. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. *Technical Report MSR-TR-97-19*. Microsoft Corporation, Advanced Technology Division. Redmond, WA, USA, outubro, 1997.

JOOSEN, W.; BERBERS, Y.; VERBAETEN, P. Dynamic Load Balancing in Transputer Applications with Geometric Parallelism. *Microprocessing and Microprogramming*, v.30(1-5), p.77-84, agosto, 1990.

JOOSEN, W.; BIJNENS, S.; VERBAETEN, P. A Reusable Load Balancer for Parallel Search Problems. *Microprocessing and Micropragramming*, v.38(1-5), p.205-212, setembro, 1993.

JOOSEN, W.; BIJNENS, S.; ROBBEN, B.; OEYEN, J.V.; VERBAETEN, P. Flexible load balancing software for parallel applications in a time-sharing environments. In: *High-Performance Computing and Networking, Lecture Notes in Computer Science v.919*, Milan, Italy, p.398-406, maio, 1995.

KANAMORI, A.; WEISE, D. Worklist Management Strategies for Dataflow Analysis. *Technical Report MSR-TR-94-12. Microsoft Corporation*, Advanced Technology Division. Redmond, WA, USA, maio, 1994.

KAPLAN, J.A.; NELSON, M.L. A comparison of Queueing, Cluster and Distributed Computing Systems. *Nasa Langley Research Center*, junho, 1994.

KEMELMAKHER, M.; KREMIEN, O. Scalable and Adaptive Resource Sharing in PVM. In: *5th European PVM/MPI User's Group Meeting, Lecture Notes in Computer Science v.1497*, Liverpool, UK, p.196-205, setembro, 1998.

KONURU, R.; CASAS, J.; OTTO, S.; PROUTY, R.; WALPOLE, J. A User-Level Process Package for Concurrent Computing. *Technical Report CSE-93-016*, agosto, 1993.

KRONE, O.; RAAB, M.; HIRSBRUNNER, B. Load Balancing for Network Based Multi-threaded Applications. In: *5th European PVM/MPI User's Group Meeting, Lecture Notes in Computer Science v.1497*, Liverpool, UK, p.206-214, setembro, 1998.

KRUEGER, P.; LAI, T.; DIXIT-RADIYA, V.A. Job Scheduling is More Important than Processor Allocation for Hypercube Computers. *IEEE Transactions on Parallel and Distributed Systems*, v.5, n.5, maio, 1994.

KRUGER, P.; SHIVARATRI, N.G. Adaptive location policies for global scheduling. *IEEE Transactions on Parallel and Distributed System*, v.20, Iss 6, p.432-444, 1994.

KUNZ, T. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Transactions on Software Engineering*, v.17, n.7, p.725-730, julho, 1991.

KWOK, Y-K.; AHMAD, I. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, v.7, n.5, maio, 1996.

LEWIS, T.; EL-REWINI, H. Parallax: A Tool for Parallel Program Scheduling. *IEEE Parallel & Distributed Technology*, pp-62-72, maio, 1993.

LIM, K.; KIM, C. Dynamic Load Balancing in Distributed Computer Systems with Star Topology. In: *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Cheju Island, Korea, p.514-519, agosto, 1995.

LIN, F.C.H.; KELLER, R.M. The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engineering*, v.SE-13, n.1, janeiro, 1987.

LITZKOW, M.; TANNENBAUM, T.; BASNEY, J.; LIVNY, M. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. *Computer Sciences Technical Report #1346 of the University of Wisconsin-Madison*, abril, 1997.

LO, V.; RAJOPADHYE, S.; GUPTA, S.; KELDSEN, D.; MOHAMED, M.A.; NITZBERG, B.; TELLE, J.A.; ZHONG, X. OREGAMI: Tools for Mapping Parallel Computations to Parallel Architectures. *International Journal of Parallel Programming*, p.237-270, 1991.

LOH, P.K.K.; HSU, W.J.; WENTONG, C.; SRISKANTHAN, N. How Network Topology Affects Dynamic Load Balancing. *IEEE Parallel & Distributed Technology*, v.04 n.03, p.25-35, 1996.

LÜLING, R.; MONIEN, B.; RAMME, F. Load Balancing in Large Networks: A Comparative Study. *Relatório Técnico, Departamento de Matemática e Ciência de Computação*, Universidade de Paderborn, Alemanha, 1993.

LÜLING, R.; MONIEN, B. A Dynamic Distributed Load Balancing Algorithm with Provable Good Performance. In: *Proceedings of ACM Symposium on Parallel Algorithms and Architectures (SPAA-93)*, 1993.

MASCARENHAS, E.; REGO, V. Migrant threads on process farms: parallel programming with Ariadne. *Concurrency-practice and experience*, v.10, Iss 9, p.673-698, 1998.

MEHRA, P.; WAH, B.W. Automated Learning of Load-Balancing Strategies for a Distributed Computer System. University of Illinois at Urbana-Champaign, 1993. *Disponível em ftp.ibr.cs.tu-bs.de*

MILOJICIC, D.S.; GIESE, P.; ZINT, W. Load Distribution on Microkernels. In: *IEEE Workshop "Future Trends in Distributed Computing Systems"*, setembro, 1993.

MULLENDER, S. *Distributed Systems.* 2a. ed., New York, Addison-Wesley Publishing Company, 1993.

MUNIZ, F.J. *Parallel Load-Balancing on Message Passing Architectures.* Tese de Doutorado apresentada à Faculdade de Engenharia da Universidade de Southampton, março, 1994.

NASTEA, S.G.; EL-GHAZAWI, T.; FRIEDER, O. Performance optimization of combined variable-cost computations and I/O. In: *Solving irregularly structured problems in parallel, Lecture Notes in Computer Science v.1253*, p.198-206, 1997.

NEUMAN, B.C.; RAO, S. Resource Management for Distributed Parallel Systems. In: *Proceedings of the 2nd Internationsl Syposium on High Performance Distributed Computing*, Spokane, p.316-323, julho, 1993.

NEUMAN, B.C.; RAO, S. The Prospero Resource Manager: A Scalable Framework for Processor Allocations in Distributed Systems. *Concurrency: Practice and Experience*, v.6(4), p.339-355, junho, 1994.

NIKOLOPOULOS D.S.; POLYCHRONOPOULOS, E.D.; PAPATHEODOROU, T.S. Enhancing the performance of autoscheduling in distributed shared memory multiprocessors. In: *Euro-Par' 98 Parallel Processing - Fourth International Euro-Par Conference, Lecture Notes in Computer Science v.1470*, Southampton, UK, p.491-501, setembro, 1998.

NOSSAL, R. An evolutionary approach to multiprocessor scheduling of dependent tasks. In: *Proceedings of Parallel and Distributed Processing, Lecture Notes in Computer Science v.1388*, Orlando, Florida, USA, p.279-287, março/abril, 1998.

OLIKER, L.; BISWAS, R.; GABOW, H.N. Performance analysis and portability of the PLUM load balancing system. In: *Euro-Par' 98 Parallel Processing -*

Fourth International Euro-Par Conference, Lecture Notes in Computer Science v.1470, Southampton, UK, p.307-317, setembro, 1998.

ORLANDO, S.; PEREGO, R. Scheduling data-parallel computations on heterogeneous and time-shared environments. In: *Euro-Par' 98 Parallel Processing - Fourth International Euro-Par Conference, Lecture Notes in Computer Science v.1470*, Southampton, UK, p.356-366, setembro, 1998.

OUSTERHOUT, J.K. Scheduling Techiniques for Concurrent Systems. In: *Proceedings Third International Conference on Distributed Computing Systems*, IEEE, p22-30, 1982.

OVEREINDER, B.J.; SLOOT, P.M.A.; HEEDERIK, R.N.; HERTZBERGER, L.O. A dynamic load balancing system for parallel cluster computing. *Future Generation Computer Systems - FGCS*, 12, p.101-115, 1996.

PAINDAVEINE, Y.; MILOJICIC, D.S. Process vs. Task Migration. In: *29th Annual Hawaii International Conference on System Sciences*, janeiro, 1996.

POLYCHRONOPOULOS, C.D.; GIRKAR, M.; HAGHIGHAT, M.R.; LEE, C.L.; LEUNG, B.; SCHOUTEN, D. Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors. In: *Proceedings of International Conference on Parallel Processing*, v.II, p.II-39-II48, 1989.

PROUTY, R.; OTTO, S.; WALPOLE, J. Adaptive Execution of Data Parallel Computations on Networks of Heterogeneous Workstations. *Technical Report CSE-94-012 Departament of Computer Science and Engineering*, Oregon Graduate Institute of Science & Technology, março, 1994.

PRUYNE, J.; LIVNY, M. Managing Checkpoints for Parallel Programs. In: *IPPS' 96 Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science v.1162*, Santa Barbara, Honolulu, Hawaii, abril, 1996.

QUINN, M.J. *Parallel Computing*: Theory and Practice. 2.ed., New York, McGraw-Hill Inc., 1994.

RAPINE, C.; SCHERSON, I.D.; TRYSTRAM, D. On-line sheduling of parallelizable jobs. In: *Euro-Par' 98 Parallel Processing - Fourth International Euro-Par Conference, Lecture Notes in Computer Science v.1470*, Southampton, UK, p.322-327, setembro, 1998.

RENESSE, R.V.; STAVEREN, H.V.; TANENBAUM, A.S. The Performance of the Amoeba Distributed Operating System. *Software - Practice and Experience*, v.19(3), p.223-234, março, 1989.

ROBINSON, J.; RUSS, S.H.; FLACHS, B.; HECKEL, B. A Task Migration Implementation of the Message-Passing Interface. In: *IEEE conference*

proceedings of the 5 th High Performance Distributed Computing Conference (HPDC-5), OnCenter, Syracuse, New York, agosto, 1996.

RONDE, J.F.; SCHONEVELD, A.; SLOOT, P.M.A. Load Balancing by redundant decomposition and mapping. *Future Generation Computer Systems - FGCS*, 12, p.391-406, 1997.

RUSS, S.H.; FLACHS, B.; ROBINSON, J.; HECKEL, B. Hector: Automated Task Allocation for MPI. In: *10th International Parallel Processing Symposium in Honolulu, (IEEE conference proceedings)*, Hawaii, abril, 1996.

RUSS, S.H.; MEYERS, B.; ROBINSON, J.; GLEESON, M.; RAJAGOPALAN, L.; TAN, C-H.; HECKEL, B. User-Transparent Run-Time Performance Optimization. In: *EHPC'97 - the 2nd International Workshop on Embedded HPC Systems and Applications at the 11th IEEE International Parallel Processing Symposium*, 1997.

SANTOS, R.R. *Escalonamento de Aplicações Paralelas: Uma interface entre o AMIGO e ORB-CORBA.* Trabalho de mestrado em andamento no Instituto de Ciências Matemáticas de São Carlos, Universidade de São Paulo (ICMC/USP), 2000.

SAPHIR, W.; TANNER, L.A.; TRAVERSAT, B. Job Management Requirements for NAS PArallel Systems and Clusters. In: *IPPS' 95 Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science v.949*, Santa Barbara, CA, USA, abril, 1995.

SCHNOR, B. Dynamic Scheduling of Parallel Applications. In: *Third International conference PaCT-95, Lecture Notes in Computer Science v.964, St.Petersburg, Rússia,* p.109-116, setembro, 1995.

SCHNOR, B.; PETRI, S.; LANGENDÖRFER, H. Load Management for Load Balancing on Heterogeneous Plataforms: A Comparison of Traditional and Neural Network Based Approaches. In: *Second International Euro-Par Conference - Euro-Par'96, Lecture Notes in Computer Science v.1124*, Lyon, France, p.611-614, agosto, 1996.

SCHWIEGELSHOHN, U. Preemptive Weighted Completion Time Scheduling of Parallel Jobs. In: *Fourth Annual European Symposium - Algorithms - ESA' 96; Lecture Notes in Computer Science v.1136*, Barcelona, Espanha, setembro, 1996.

SEKHARAN, C.N.; GOEL, V.; SRIDHAR, R. Load balancing methods for ray tracing and binary tree computing using PVM. *Parallel Computing*, v.21, Iss 12, p.1963-1978, 1995.

SHIRAZI(a), B.A.; HURSON, A.R.; KAVI, K.M. Introduction to Scheduling and Load Balancing. *Introdução do Primeiro Capítulo do Livro Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press, Los Alamitos, CA, p.2-6, USA, 1995.

SHIRAZI(b), B.A.; HURSON, A.R.; KAVI, K. M. Mechanisms for Process Migration. *Introdução do Sexto Capítulo do Livro Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press, Los Alamitos, CA, USA, p.411-413, 1995.

SHIRAZI, B.; HURSON, A.R. Special Issue on Scheduling and Load Balancing: guest editor's introduction. *Journal of Parallel and Distributed Computing*, v.16, Iss 4, p.271-275, 1992.

SHIVARATRI, N.G.; KRUEGER, P.; SINGHAL, M. Load Distributing for Locally Distributed Systems. *IEEE Computer*, dezembro, 1992.

SHMOYS, D.B.; WEIN, J.; WILLIAMSON, D.P. Scheduling Parallel Machines On-Line. *SIAM Journal on Computing*, v.24, n.6, p.1313-1331, dezembro, 1995.

SHU, W.; WU, M-Y. Runtime Incremental Parallel Scheduling (RIPS) on Distributed Memory Computers. *IEEE Transactions on Parallel and Distributed Systems*, v.7, n.6, junho, 1996.

SMITH, P.; HUTCHINSON, N.C. Heterogenous Process Migration: The tui System. *Software - Practice and Experience*, v.28, n.6, p.611-639, 1998

SNIR, M.; OTTO, S.; STEVEN, H.; WALKER, D.; DONGARRA, J.J. *MPI: The Complete Reference*. The MIT Press, Massachusetts, 1996.

SOBALVARRO, P.G.; PAKIN, S.; WEIHL, W.E.; CHIEN, A.A. Dynamic Coscheduling on Workstation Clusters. In: *IPPS/SPDP' 98 Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science v.1459*, Orlando, Florida, USA, p.231-256, março, 1998.

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(1), p.1-19, janeiro, 1997.

SOUZA, M.A. *Avaliação das Rotinas de Comunicação Ponto-a-Ponto do MPI*. Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo (ICMC/USP), dezembro, 1996

SOUZA, M.A. *Monitoramento de Aplicações Paralelas para Escolha de Políticas de Escalonamento de Processos*. Trabalho de doutorado em andamento no Instituto de Ciências Matemáticas de São Carlos, Universidade de São Paulo (ICMC/USP), 2000.

SOUZA, P.S.L. *Máquina Paralela Virtual em Ambiente Windows*. Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo (ICMC/USP), maio, 1996.

SOUZA(a), P.S.L.; SANTANA, M.J.; SANTANA, R.H.C. A New Scheduling Environment for Near-Optimal Performance. In: *International Conference on Parallel and Distributed Processing Techniques and Applications – PDPTA'99*, Las Vegas, Nevada, U.S.A., junho, 1999.

SOUZA(b), P.S.L.; SANTANA, M.J.; SANTANA, R.H.C. AMIGO – A Dynamical Flexible Scheduling Environment. In: *5th International Conference on Information Systems Analysis and Synthesis – ISAS'99*, Orlando, U.S.A., julho, 1999.

SOUZA(c), P.S.L.; SANTANA, M.J.; SANTANA, R.H.C.; ARAÚJO, A.P.F. PVM and a Viable and Flexible Scheduling. In: *Eleventh IASTED International Conference on Parallel and Distributed Computing and Systems - PDCS'99*, Cambridge, MA, USA, novembro, 1999.

SOUZA, P.S.L.; SANTANA, M.J.; SANTANA, R.H.C. Escalonamento de Processos: uma contribuição para a Convergência da Área. Notas do ICMC, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, maio, 2000.

SQUILLANTE, M.S. On the Benefits and Limitations of Dynamic Partitioning in Parallel Computer Systems. In: *IPPS' 95 Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science v.949,* Santa Barbara, CA, USA, abril, 1995.

STALLINGS, W. *SNMP, SNMPv2 and CMIP.* MA, Addison-Wesley Publishing Company, 1993.

STATHOPOULOS, A.; YNNERMAN, A. Dynamic load balancing of atomic structure programs on a PVM cluster. *In: High-Performance Computing and Networking*, Milan, Italy, Lecture Notes in Computer Science v.919, p.384-391, maio, 1995.

STEENSGAARD, B.; JUL, E. Object and Native Code Process Mobility Among Heterogenous Computers. *Technical Report MSR-TR-95-17.* Microsoft Corporation, Advanced Technology Division. Redmond, WA, USA, março, 1995.

STELLNER, G.; TRINITIS, J. Load Balancing Based on Process Migration for MPI. *Euro-Par' 97 Parallel Processing - Third International Euro-Par Conference, Lecture Notes in Computer Science v.1300*, Passau, Germany, agosto, 1997.

TANENBAUM, A.S. *Modern Operating Systems.* New Jersey, Prentice Hall International, Inc., 1992.

TANENBAUM, A.S. *Distributed Operating Systems.* New Jersey, Prentice Hall International, Inc., 1995.

TANENBAUM, A.S. *Computer Networks*. 3. ed. New Jersey, Prentice-Hall Inc., 1996.

THEIMER, M.M.; HAYES, B. Heterogeneous Process Migration by Recompilation. In: *Proceedings IEEE 11th Int'l Conf. on Distributed Computing System, IEEE CS Press*, p.18-25, Los Alamitos, California, 1991.

TRIPATHI, A.R.; KARNIK, N.M. Trends in multiprocessor and distributed operating-systems designs. *Journal of supercomputing*, v.9, Iss 1-2, p.23-49, 1995.

VESSEUR, J.J.J.; HEEDERIK, R.N.; OVEREINDER, B.J.; SLOOT, P.M.A. Experiments in Dynamic Load Balancing for Parallel Cluster Computing. In: *Proceedings of the Workshop on Parallel Programming and Computation*, Amsterdam, p.189-194, junho , 1995.

WANG, J.; LI, J.; KAMEDA, H. Scheduling Algorithms for Parallel Transaction Processing Systems. In: 4th International Conference, PaCT'97 Yaroslaul, Parallel Computing Tecnologies, Lecture Notes in Computer Science v.1277, Rússia, setembro, 1997.

WANG, Y.T.; MORRIS, R.J.T. Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, p.204-217, março, 1985.

WU, M.Y. On runtime parallel scheduling for processor load balancing. *IEEE Transactions on Parallel and Distributed Systems*, v.8, Iss 2, p.173-186, 1997.

XIROGIANNIS, G.; TAYLOR, H. A dynamic task distribution and engine allocation strategy for distributed execution of logic programs. In: *Proceedings of High-Performance Computing and Networking, Lecture Notes in Computer Science v.1401*, Amsterdan, The Netherlands, p.294-304, abril, 1998.

XU, C.; LAU, F.C.M. Load Balancing in Parallel Computers: Theory and Practice. *Kluwer Academic Publishers*, Boston, USA, 1997.

YAN, Y.; JIN, C.M.; ZHANG, X.D. Adaptively scheduling parallel loops in distributed shared-memory systems. *IEEE Transactions on Parallel and Distributed Systems*, v.8, Iss.1, p.70-81, 1997.

ZALUSKA, E.J. Research lines in distributed computing system and concurrent computation In: Anais do Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software, p.132-155, 1991.

ZHOU, S.; ZHENG, X.; WANG, J.; DELISLE, P. Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software: Practice and Experience*, v.23(12), p.1305-1336, dezembro, 1993.

ZHOU, B.B.; BRENT, R.P.; WALSH, D.; SUZAKI, K. Job Scheduling Strategies for Networks of Workstations. In: *IPPS/SPDP' 98 Workshop on Job Scheduling*

Strategies for Parallel Processing, Lecture Notes in Computer Science v.1459, Orlando, Florida, USA, p.143-157, março, 1998.

Apêndice A – Relação de Novas Propostas de Algoritmos de Escalonamento

Este apêndice relaciona trabalhos que descrevem novas propostas de escalonamento. Mesmo não sendo uma listagem completa, devido à dificuldade em se encontrar "todas" as propostas feitas, a relação oferece um vasto material à pesquisa e demonstra a grande quantidade de trabalhos que descrevem novas maneiras de se escalonar processos.

Os trabalhos abaixo estão organizados em ordem alfabética do nome do algoritmo de escalonamento proposto. Quando o trabalho não cita um nome específico, é apresentada apenas a referência que o descreve.

- ADM - Prouty et al. (1994), Song et al. (1997);
- AMOEBA - Renesse et al. (1989), Tanenbaum (1995);
- ARIADNE SYSTEM - Mascarenhas; Rego (1998);
- AUTOESCHEDULING - Nikolopoulos et al. (1998);
- BALANCE SYSTEM - Hui; Chanson (1997);
- CAMAS - Ronde et al. (1997);
- CHARLOTTE - Artsy; Finkel (1989), Song et al. (1997);
- CILK - Blumofe et al. (1996);
- COCHECK - Pruyne; Livny (1996);
- CODINE (Computing in DIstributed Networked Environments) - Kaplan;Nelson (1994), Ferstl (1996);
- CONDOR - Kaplan; Nelson (1994), Epema et al. (1996), Litzkow et al. (1997);
- CONNECT - Kaplan; Nelson (1994);
- COOL (Concurrent Object-Oriented Language) - Chandra et al. (1993);
- DCP (Dynamic Critical-Path) - Kwok; Ahmad (1996);
- DCS (Dynamic CoScheduling) - Sobalvarro et al. (1998);
- DI (Difference-Initiated) - Antonis et al. (1998);
- DOME - Beguelin et al. (1996), Arabe et al. (1995);
- DJM (Distributed Job Manager) - Kaplan; Nelson (1994);
- DQS (Distributed Queueing System) - Kaplan; Nelson (1994);
- DYNAMICPVM - Overeinder et al. (1996), Vesseur et al. (1995);
- ESLR (Eager Scheduling with Lazy Retry) - Chen; King (1996);

- FCFSP (FCFS with Priority) - Wang et al. (1997);
- GA (Genetic Algorithms) - Nossal (1998);
- GRADIENTE - Lin; Keller (1987);
- GRM (General Resource Manager) - Fagg et al. (1996);
- HARNNES (Heterogeneous Adaptable RecoNfigurable NEtwork Systems) - Dongarra et al. (1998);
- HECTOR - Russ et al. (1996), Robinson et al. (1996), Russ et al. (1997),
- HiCon - Becker (1995);
- LACHESIS - Downey (1998);
- LBS - Krone et al. (1998);
- LOAD BALANCER - Kaplan; Nelson (1994);
- LOAD LEVELER - Kaplan; Nelson (1994);
- LRF (Least Recently Fired) - Kanamori; Weise (1994);
- LSF (Load Sharing Facility) - Kaplan; Nelson (1994);
- MACH - Milojicic et al. (1993), Paindaveine; Milojicic (1996), Tanenbaum (1995);
- MIST (Migration And Integrated Scheduling Tools) - Al-Saqabi et al. (1997), Casas et al. (1995b);
- MOSIX - Barak et al. (1996);
- MPVM - Song et al. (1997);
- MODELO ECONÔMICO - Chiclayo; Henriques (1999);
- MPVM (Migratable PVM) - Casas et al. (1995a), Clark et al. (1995);
- MSA (Multi-stage Sheduling Approach) - Boeres et al. (1998);
- NEAREST LAYER FIRST - Glazek (1999);
- NC TOOLSET - Kaplan; Nelson (1994);
- NQE (Network Queueing Environment) - Kaplan; Nelson (1994);
- OREGAMI - Lo et al. (1991);
- PAISS (PArallel Iterative linear System Solver) - Christen (1998);
- PAN (Prolog Area Network) - Xirogiannis; Taylor (1998);
- PARAFRASE-2 - Polychronopoulos et al. (1989);
- PARALLAX - Lewis; El-Rewini (1993);
- PARALEX - Davoli; Babaoglu (1996);
- PBS (Portable Batch System) - Henderson (1995);
- PIRANHA - Song et al. (1997);
- PLUM - Oliker et al. (1998);
- PMS (Processors Management System) - Aguilar; Jimenez (1997);
- PREP-P - Berman; Stramm (1994);
- PRM (Prospero Resource Manager) - Neuman; Rao (1993), Neuman; Rao (1994);
- RIALTO - Jones et al. (1997);
- RIPS (Runtime Incremental Parallel Scheduling) - Shu; Wu (1996);
- SCAN - Krueger et al. (1994);
- S-MPI (Selective MPI) - Dantas; Zaluska (1998);
- SITA-E - Harchol-Balter et al. (1997), Harchol-Balter; Downey (1997);
- SPAWN - Schnor (1995);
- SPRITE - Douglis (1989), Douglis; Ousterhout (1991);
- SUPLE - Orlando; Perego (1998);

- TASK BROKER - Kaplan; Nelson (1994);
- TOPSYS (TOols for Parallel SYStems) - Bemmerl (1990);
- TPM (Task-to-Processor Mapping) - Dikenelli et al. (1997);
- TUMPI - Stellner; Trinitis (1997);
- TUI SYSTEM - Smith; Hutchinson (1998);
- UPVM - Konuru et al. (1993);
- UTOPIA - Zhou et al. (1993);
- VASSAL - Candea; Jones (1998);
- V-SYSTEM - Tanenbaum (1995);
- VDS (Virtual Data Space) - Decker (1997);
- XENOOPS - Joosen et al. (1995);

- Berman et al. (1997);
- Boukerche; Das (1998);
- Brest et al. (1999);
- Bubak et al. (1995);
- Cherkasova (1998);
- Chiclayo; Henriques (1999);
- Chien et al. (1994);
- Colajanni et al. (1998);
- Corradi et al. (1998);
- Czajkowski et al. (1998);
- Durand et al. (1996);
- Fonlupt et al. (1998);
- Genco; Lore (1996);
- Gil; Matias (1996);
- Ha; Lee (1991);
- Halang et al. (1995);
- Hill et al. (1998);
- Hu et al. (1998);
- Huang; Ongsakul (1994);
- Jackson; Humphres (1997);
- Joosen et al. (1990);
- Joosen et al. (1993);
- Kemelmakher; Kremien (1998);
- Krueger; Shivaratri (1994);
- Lim; Kim (1995);
- Lüling; Monien (1993);
- Nastea et al. (1997);
- Sekharan et al. (1995);
- Shivaratri et al. (1992);
- Shmoys et al. (1995);
- Stathopoulos; Ynnerman (1995);
- Steensgaard; Jul (1995);
- Theimer; Hayes (1991);
- Wu (1997);

- Yan et al. (1997);
- Zhou et al. (1998);

Apêndice B – Estrutura de Diretórios do AMIGO

O AMIGO está sendo implementado na estrutura de diretórios ilustrada pela Figura B.1. O diretório "amigod" contém o código fonte e os arquivos executáveis do AMIGOD. O diretório "bench" contém os *benchmarks* utilizados para avaliar a plataforma computacional. Cada *benchmark* cadastrado possui um diretório específico para ele, subordinado ao diretório "bench".

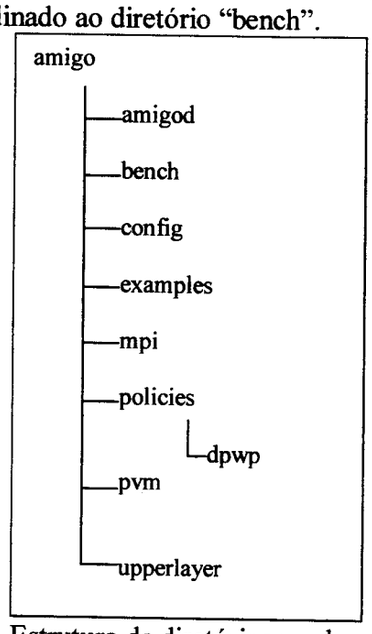


Figura B.1 – Estrutura de diretórios usada pelo AMIGO.

O diretório "config" possui os arquivos de configuração gerados pela camada superior, como por exemplo o *policy.cfg*. O diretório "examples" possui exemplos de aplicações que foram escalonadas sob a supervisão do AMIGO. O diretório "mpi" contém a implementação do MPI utilizada pelo AMIGO. Nesse diretório encontram-se também os arquivos com as alterações necessárias feitas no MPI para que este interaja com o AMIGO.

O diretório "policies" contém as políticas desenvolvidas para o AMIGO, como a DPWP.

O diretório "pvm" contém a versão do PVM utilizada pelo AMIGO. Assim como no MPI, nesse diretório também estão os arquivos com as alterações necessárias feitas no PVM, para que o mesmo interaja com o AMIGO.

O diretório "upperlayer" contém a camada superior do AMIGO. No "upperlayer" encontram-se outros diretórios como o "interface" que armazena a interface gráfica da camada superior, "simul" que armazena a simulação utilizada pelo AMIGO, entre outros.

Apêndice C – Códigos Fonte das Aplicações Usadas nas Avaliações de Desempenho

Neste Apêndice encontram-se os códigos fonte das aplicações utilizadas nesta tese para fazer as avaliações de desempenho com o AMIGO. As listagens representam, respectivamente, os códigos de: integralpvm.c e escravo.c (método dos trapézios compostos), ordena.c e quick.c (*quicksort* paralelo) e synthetic.c (aplicação "sintética" *CPU e I/O-Bound*).

```
1   /*****************************************************
2   ARQUIVO integralpvm.c
3   processo mestre aplicação método dos trapézios compostos
4   *****************************************************/
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include "time.h"
8   #include <math.h>
9   #include "pvm3.h"
10  #include<sys/time.h>
11  #include<unistd.h>
12
13  #define ESCRAVO "/home/amigo/examples/integral/escravo"
14  #define NOME_ARQ_TEMPO "tempo_amigo_12000_.txt"
15
16  #define N 12000000  /*quantidade de divisoes*/
17  /* a quantidade de divisoes deve ser um multiplo de (NPROCS+1)*/
18  #define NPROCS 9    /*quantidade de escravos*/
19
20  double   s = 0,sum = 0;
21  struct timeval tempo1, tempo2;
22  struct timezone tzp;
23  double tempo;
24  double a,b;
25  double limite_inferior,limite_superior;
26  double x;
27  double resultado,res,valor_retorno;
28  int info, infos;
29
30  float integral(double a, double b, int n)
31  {
32     double h,var,r,s = 0, aux;
33     int i;                // numero de termos
```

```
34      h = (double)(b-a)/n;
35      s = s + sin(a) + exp(a);
36      s = s + sin(b) + exp(b);
37      aux = a;
38      for(i=1;i<n-1;i++){
39         var = aux + h;
40         r = 2 * (sin(var) + exp(var));
41         s = s + r;
42         aux = var;
43      }
44      return((h/2) * s);
45   }
46
47   main(int argc, char **argv)
48   {
49   int mytid;            /* Identificacao do processo mestre */
50   int tids[NPROCS];     /* Identificacao dos processos escravos */
51   int cc;               /* Numero de processos criados */
52   int tid;              /* Identificacao do processo escravo */
53   int i;                /* Contador */
54   FILE *out;            /* Arquivo p/ o tempo de Processamento */
55   float timef;          /* Tempo para a execucao da integral */
56
57
58   mytid = pvm_mytid ();
59   printf("Processo mestre: %x\n", mytid);
60   cc=pvm_spawn(ESCRAVO, (char**)0, PvmTaskDefault, NULL, NPROCS,tids);
61   if (cc < NPROCS){
62   printf("Nao consegui gerar algum escravo...\n");
63   exit(0);
64   }
65      printf("Iniciando calculos...\n");
66      gettimeofday(&tempo1,&tzp);
67      a = 0.0;
68      b = 1.2;
69
70      x = (double)(b - a)/(NPROCS+1); //numtasks;
71
72      limite_inferior = a;
73      limite_superior = a + x;
74      for(i=0;i<NPROCS;i++){
75         pvm_initsend(PvmDataDefault);
76         pvm_pkdouble(&limite_inferior,1,1);
77         pvm_pkdouble(&limite_superior,1,1);
78         pvm_send (tids[i],0);
79         limite_inferior = limite_superior;
80         limite_superior = limite_superior + x;
81      }
82
83      res = integral(limite_inferior,limite_superior,N/(NPROCS+1));
84
85      for(i=0;i<NPROCS;i++){
86         pvm_recv(-1, -1);
87         pvm_upkdouble(&valor_retorno, 1, 1);
88         sum = sum + valor_retorno;
89      }
90
91      res = res + sum;
92
93      gettimeofday(&tempo2,&tzp);
```

```
94
95      tempo = (double)(tempo2.tv_sec - tempo1.tv_sec) +
96   (((double)(tempo2.tv_usec-tempo1.tv_usec))/1000000);
97
98      /* Colocando o resultado no arquivo*/
99
100      /* Abrindo arquivo timeresults para escrita...*/
101      out=fopen(NOME_ARQ_TEMPO,"a");
102      if(out==NULL) //se arquivo ainda nao existe
103          out=fopen(NOME_ARQ_TEMPO,"w"); //criando o arquivo
104
105      if (out==NULL){ //nao abriu o arquivo
106          printf("Problemas para criar o arquivo");
107      }
108      else { // gravando o tempo no arquivo...
109          fprintf(out, "%f\n",tempo);
110      }//end if
111
112      fclose(out);
113
114      printf("Tempo em segundos = %f\n",tempo);
115      printf("Quantidade de divisoes = %d\n",N);
116      printf("-----------------------------------------------\n");
117      fflush(stdout);
118      pvm_exit();
119   }
```

```
1   /*************************************************************
2   ARQUIVO escravo.c
3      processo escravo da aplicação método dos trapézios compostos
4   *************************************************************/
5   #include <stdio.h>
6   #include "pvm3.h"
7   #include <stdlib.h>
8   #include "time.h"
9   #include <math.h>
10
11  #define N 12000000  /*quantidade de divisoes*/
12  /* a quantidade de divisoes deve ser um multiplo de (NPROCS+1). */
13  #define NPROCS 9   /*numero total de escravos*/
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+1) ));
46
47     pvm_initsend(PvmDataDefault);
48     pvm_pkdouble(&res,1,1);
49     pvm_send(master,4);
50
51     pvm_exit ();
52  }
```

```
1   /*************************************************************
2   ARQUIVO ordena.c
3      processo mestre da aplicação quicksort paralelo
4   *************************************************************/
5   #include <stdio.h>
6   #include <limits.h>
7   #include <sys/time.h>
8   #include "pvm3.h"
9
10  #define MaxVet 306000  // numero de elementos para ordenar
11  #define NUM_PROC 9     // numero de escravos (quick.c)
12
13  // aquivo onde serao gravados os tempos coletados
14  #define ARQ_TEMPO "/home/amigo/examples/quicksort/quick_amigo.txt"
15
16  // arquivos escravos que irao ordenar partes do vetor de elementos
17  #define ESCRAVO "/home/amigo/examples/quicksort/quick"
18
19
20  void grava(double dif_tot, double dif_spawn, double dif_sr, double
21  dif_seq)   {
22
23      FILE *fd;
24
25      fd = fopen(ARQ_TEMPO, "a+");
26
27      if(fd == NULL)    {
28          fprintf(stderr, "Cannot open input file.\n");
29          return;
30      }
31
32  fprintf(fd, "tot:%f spawn:%f send/rcv:%f seq:%f \n", dif_tot,
33  dif_spawn, dif_sr, dif_seq);
34
35      fflush(fd);
36      fclose(fd);
37  }
38
39
40  int main() {
41      int i, t, ind;
42      int tid_sort[NUM_PROC], indices[NUM_PROC];
43      int tot_subvet, pos_min;
44      int *vetor = (int *)malloc( MaxVet * sizeof(int) );
45      int *vet_ordenado = (int *) malloc( MaxVet * sizeof(int) );
46      unsigned long int val_min;
47      int block = 1, cc, my_tid;
48
49       struct timeval tmp1, tmp2, tmp_mandar, tmp_receber;
50       struct timezone tzp;
51
52       double dif_tot, dif_spawn, dif_sr, dif_seq;
53
54       my_tid = pvm_mytid();
55       printf("Ordena tid=[%x]\n", my_tid);
56
57       tot_subvet = MaxVet/NUM_PROC;
58
59       for (t = 0; t < MaxVet; t++) {  // gera vetor
60           vetor[t] = rand() % 30000;
```

```
61          }
62
63          // determinar tempo inicial da execucao
64         gettimeofday(&tmp1,&tzp);
65
66      cc = pvm_spawn(ESCRAVO, (char **)0, PvmTaskDefault, NULL, NUM_PROC,
67      tid_sort);
68         if( cc < NUM_PROC) {
69            printf ("ERRO:pvm_spawn nao gerou todas as tarefas\n");
70         exit (-1);
71         }
72
73          // determinar tempo para enviar subvetores
74         gettimeofday(&tmp_mandar,&tzp);
75
76          // envia subvetores para os processos
77          for (t = 0; t < NUM_PROC; t++)  {
78             pvm_initsend (PvmDataRaw);
79             pvm_pkint (&vetor[(tot_subvet * t)], tot_subvet, 1);
80             pvm_send (tid_sort[t], 1);
81          }
82
83          // recebe os subvetores ordenados
84          for (t = 0; t < NUM_PROC; t++)  {
85             pvm_recv (-1, 2);
86             pvm_upkint (&vetor[(tot_subvet * t)], tot_subvet, 1);
87             indices[t] = (tot_subvet * t);
88          }
89
90          // determinar tempo do recebimento dos subvetores
91          gettimeofday(&tmp_receber,&tzp);
92
93          // merge - busca o menor elemento dos subvetores
94          for (t = 0; t < MaxVet; t++) {
95             val_min = ULONG_MAX;
96             pos_min = -1;
97             for (ind = 0; ind < NUM_PROC; ind++)    {
98      if((indices[ind]<(tot_subvet*(ind+1))) &&
99      (vetor[indices[ind]]<=val_min))
100            {
101               val_min = vetor[indices[ind]];
102               pos_min = ind;
103            }
104         }
105         if (pos_min > -1)     {
106            vet_ordenado[t] = val_min;
107            (indices[pos_min])++;
108         }
109         else
110            printf ("Nao conseguiu achar o menor...\n");
111         }
112          // determina o tempo final da execucao
113          gettimeofday(&tmp2,&tzp);
114
115     dif_tot = (double)(tmp2.tv_sec - tmp1.tv_sec) +
116     (((double)(tmp2.tv_usec-tmp1.tv_usec))/1000000);
117         printf("Tempo total: %f\n", dif_tot);
118
119     dif_spawn = (double)(tmp_mandar.tv_sec - tmp1.tv_sec) +
120     (((double)(tmp_mandar.tv_usec-tmp1.tv_usec))/1000000);
```

```
121
122  dif_sr = (double)(tmp_receber.tv_sec - tmp_mandar.tv_sec) +
123  (((double)(tmp_receber.tv_usec-tmp_mandar.tv_usec))/1000000);
124
125  dif_seq = (double)(tmp2.tv_sec - tmp_receber.tv_sec) +
126  (((double)(tmp2.tv_usec-tmp_receber.tv_usec))/1000000);
127
128      grava(dif_tot, dif_spawn, dif_sr, dif_seq);
129
130      // mostra os elementos ordenados
131      for (t = (MaxVet-1); t < MaxVet; t++)
132         printf ("vet_ordenado [%d] = %d \n", t, vet_ordenado[t]);
133
134      free(vetor);
135      free(vet_ordenado);
136
137      pvm_exit();
138      return(0);
139  }
```

```
1   /*************************************************************
2   ARQUIVO quick.c
3       processo escravo da aplicação quicksort paralelo
4   *************************************************************/
5   #include "pvm3.h"
6
7   #define MaxVet 306000  // numero de elementos para ordenar
8   #define NUM_PROC 9     // numero de escravos (quick.c)
9
10  typedef int indice;
11
12  void particao (int *a, indice esq, indice dir, indice *i, indice *j)
13      {
14      int x, w;
15
16      *i = esq;
17      *j = dir;
18      x = a[(*i + *j)/2];
19
20      do
21      {
22          while (a[*i] < x)    (*i)++;
23          while (a[*j] > x)    (*j)--;
24          if ( *i <= *j)
25          {
26              w = a[*i];
27              a[*i] = a[*j];
28              a[*j] = w;
29              (*i)++;
30              (*j)--;
31          }
32      } while (*i <= *j);
33  }
34
35
36  void ordena (int *a, indice esq, indice dir)  {
37
38      indice i, j;
39
40      particao (a, esq, dir, &i, &j);
41      if (esq < j)
42          ordena (a, esq, j);
43      if (i < dir)
44          ordena (a, i, dir);
45  }
46
47
48  int main(void)    {
49
50      int tid_parent;
51      int tot_subvet;
52      int *a = (int *)malloc( (MaxVet/NUM_PROC) * sizeof(int) );
53      int buf_rcv;
54
55      tid_parent = pvm_parent();
56
57      tot_subvet = (MaxVet/NUM_PROC);
58
59      buf_rcv = pvm_recv (tid_parent, 1);
60      pvm_upkint (a, tot_subvet, 1);
```

```
61      pvm_freebuf(buf_rcv);
62
63      ordena (a, 0, (tot_subvet - 1));
64
65      pvm_initsend(PvmDataRaw);
66      pvm_pkint(a, tot_subvet, 1);
67      pvm_send(tid_parent, 2);
68
69      pvm_exit();
70      free(a);
71
72      return(0);
73  }
74
```

```
1   /*************************************************************
2   ARQUIVO  synthetic.c
3      aplicação "sintética" CPU e I/O-Bound
4   *************************************************************/
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include <sys/time.h>
8   #include <math.h>
9   #include "pvm3.h"
10
11  #define TAM_VETOR 30000       // vetor para a comunicacao
12  #define NUM_ITERACOES 110000 // num. iteracoes para processamento
13  #define NUM_PROC 9                // numero de escravos
14  #define ARQUIVO "tempo19-10.txt" // nome do arquivo que sera gerado
15
16  // Gravar os resultados em Arquivo
17  void Grava(double Total, double CPUBound, double IOBound)
18  {
19    FILE *fd;
20
21    fd = fopen(ARQUIVO, "a+");
22    if (fd == NULL)
23    {
24      fprintf(stderr, "Nao e possivel abrir o arquivo\n");
25      return;
26    }
27    fprintf(fd,"Total: %f Processamento: %f Comunicacao: %f\n",
28            Total, CPUBound, IOBound);
29    fflush(fd);
30    fclose(fd);
31  }
32
33  // Funcao para aumentar o processamento
34  double Funcao(int Iteracoes)
35  {
36    int i;
37    double j, X[Iteracoes], Y[Iteracoes];
38    for (i=0; i<Iteracoes; ++i)
39    {
40       X[i]=rand()%3000;
41       Y[i]=rand()%3000;
42    }
43
44    for (i=0; i<Iteracoes; ++i)
45        j += sqrt(X[i]*Y[i]);
46
47    return 1;
48  }
49
50  //Funcao Principal
51  main()
52  {
53    int i, count, bufid, Iteracoes;        // Contadores Auxiliares
54    int Resto, nb;                         // Contadores do Vetor
55    int Ack;                               // Confirmacao
56    int tids[NUM_PROC], MyTid, MyParent;   // Identidicadores
57    double x[TAM_VETOR];                   // Tamanho do Vetor
58    double AuxRes, TempoCPU, TempoIO;      // Resultados
59    struct timeval Tempo1, Tempo2, Tempo3; // tempo
60    struct timezone TZone;                 // tempo
```

```
61
62      MyTid = pvm_mytid();
63      if (MyTid < 0)
64      {
65        pvm_perror("Bench");
66        return -1;
67      }
68
69      MyParent = pvm_parent();
70      if ((MyParent < 0) && (MyParent != PvmNoParent))
71      {
72        pvm_perror("Bench");
73        pvm_exit();
74        return -1;
75      }
76
77      if (MyParent == PvmNoParent) //Mestre
78      {
79         //Calcula a distribuicao das tarefas
80         Resto = TAM_VETOR%NUM_PROC;
81         nb = (TAM_VETOR-Resto)/NUM_PROC;
82
83         // Inicializa os vetores
84         for (i=0; i<TAM_VETOR; ++i)
85            x[i] = i;
86
87         // Inicializa o Resultado
88         Iteracoes = NUM_ITERACOES;
89
90         // Espalha os processos
91         //pvm_catchout(stdout);
92         if (pvm_spawn("/home/amigo/examples/conc/bench", (char**)0,
93           PvmTaskDefault, (char*)0, NUM_PROC, tids) < NUM_PROC)
94         {
95            fprintf(stderr, "pvm_spawn() error:\ntids: ");
96            exit(-1);
97         }
98
99         //Funcao que pega o Primeiro Tempo
100        gettimeofday(&Tempo1, &TZone);
101
102        // Envia os Dados para os Escravos
103        for (count=0,i=1; i<=NUM_PROC; ++i)  {
104          pvm_initsend(PvmDataDefault);
105          pvm_pkint(&Iteracoes, 1, 1);
106          pvm_pkdouble(&x[count], nb, 1);
107          pvm_send(tids[i-1], 1);
108          count += nb;
109        }
110
111        // Recebe o ACK de envio
112        for (i=1; i<=NUM_PROC; ++i) {
113          bufid=pvm_recv(-1, 1);
114          pvm_upkint(&Ack, 1, 1);
115        }
116
117        // Pega o segundo tempo
118        gettimeofday(&Tempo2, &TZone);
119
120        // Executa o Resto
```

```
121         AuxRes = Funcao(Iteracoes);
122
123         // Pega o Resultado
124         for (i=1; i<=NUM_PROC; ++i) {
125           bufid=pvm_recv(-1, 2);
126           pvm_upkdouble(&AuxRes, 1, 1);
127         }
128
129         // Pega o Terceiro tempo
130         gettimeofday(&Tempo3, &TZone);
131
132         // Calcula o Tempo de Comunicacao
133         TempoIO = (double)(Tempo2.tv_sec - Tempo1.tv_sec)+
134                   (((double)(Tempo2.tv_usec -
135   Tempo1.tv_usec))/1000000);
136
137         // Calcula o Tepo de Processamento
138         TempoCPU = (double)(Tempo3.tv_sec - Tempo2.tv_sec)+
139                   (((double)(Tempo3.tv_usec -
140   Tempo2.tv_usec))/1000000);
141
142         // Mostra o Resultado
143         printf("\nTempo Total: %f\n", TempoIO+TempoCPU);
144         printf("Tempo Comunicacao: %f\n", TempoIO);
145         printf("Tempo Processamento: %f\n\n", TempoCPU);
146       }
147       else { //Escravo
148           // Recebe os dados do Mestre
149         bufid=pvm_recv(-1, 1);
150         pvm_upkint(&Iteracoes, 1, 1);
151         pvm_upkdouble(x, nb, 1);
152
153         // Envio do ACK
154         pvm_initsend(PvmDataDefault);
155         pvm_pkint(&Ack, 1, 1);
156         pvm_send(MyParent, 1);
157
158         // Funcao para Processamento
159         AuxRes = Funcao(Iteracoes);
160         pvm_initsend(PvmDataDefault);
161         pvm_pkdouble(&AuxRes, 1, 1);
162         pvm_send(MyParent, 2);
163         pvm_exit ();
164         exit(0);
165       }
166       // soh o mestre passa por aqui
167       pvm_exit();
168
169       Grava(TempoCPU+TempoIO, TempoCPU, TempoIO);
170     }
```

Apêndice D – Publicações Realizadas com Base nos Trabalhos Descritos nesta Tese

Neste apêndice estão relacionadas as principais publicações realizadas, elaboradas com base nos trabalhos descritos nesta tese. São elas:

SOUZA, P.S.L.; SANTANA, M.J.; SANTANA, R.H.C. A New Scheduling Environment for Near-Optimal Performance. In: *International Conference on Parallel and Distributed Processing Techniques and Applications – PDPTA'99*, Las Vegas, Nevada, U.S.A., junho, 1999.

ARAÚJO, A.P.F.; SOUZA, P.S.L.; SANTANA, M.J.; SANTANA, R.H.C. A New Dynamical Scheduling Algorithm. In: *International Conference on Parallel and Distributed Processing Techniques and Applications – PDPTA'99*, Las Vegas, Nevada, U.S.A., junho, 1999.

SOUZA, P.S.L.; SANTANA, M.J.; SANTANA, R.H.C. AMIGO – A Dynamical Flexible Scheduling Environment. In: *5th International Conference on Information Systems Analysis and Synthesis – ISAS'99*, Orlando, U.S.A., julho, 1999.

ARAÚJO, A.P.F.; SOUZA, P.S.L.; SANTANA, M.J.; SANTANA, R.H.C. DPWP - A New Load Balancing Algorithm. In: *5th International Conference on Information Systems Analysis and Synthesis – ISAS'99*, Orlando, U.S.A., julho, 1999.

SOUZA, P.S.L.; SANTANA, M.J.; SANTANA, R.H.C.; ARAÚJO, A.P.F. PVM and a Viable and Flexible Scheduling. In: *Eleventh IASTED International Conference on Parallel and Distributed Computing and Systems - PDCS'99*, Cambridge, MA, USA, novembro, 1999.

SOUZA, P.S.L.; SANTANA, M.J.; SANTANA, R.H.C. Escalonamento de Processos: uma contribuição para a Convergência da Área. *Notas do ICMC, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo*, maio, 2000.