SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

JUMP: Uma política de escalonamento unificada com migração de processos

Juliano Ferraz Ravasi

Orientador: *Prof. Dr. Marcos José Santana*

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional.

**USP – São Carlos
Fevereiro de 2009**

JUMP: Uma política de escalonamento
unificada com migração de processos

Juliano Ferraz Ravasi

Agradecimentos

À minha mãe, Ana Maria, por todo seu amor e carinho, suas lições de vida e todo o seu incentivo à minha carreira. Seu imensurável esforço e dedicação me permitiram chegar aonde cheguei, sacrificando os seus sonhos para que eu pudesse realizar os meus. Também à minha irmã, Fabíola, pelo apoio.

Ao meu pai, José Humberto, que adoeceu e faleceu durante o período em que eu desenvolvi este trabalho. Muito difícil foi para mim aceitar e lidar com a situação. Foi uma lição de vida onde muito eu aprendi.

*“Say it loud, say it clear
You can listen as well as you hear
It's too late when we die
To admit we don't see eye to eye.”
— The Living Years, Mike & the Mechanics*

Um agradecimento muito especial à Maria Izabel, ao Guilherme, à Ana Carolina e à Ana Beatriz, minha segunda família e a qual guardo com imenso carinho e respeito. Sempre me incentivaram e me guiaram, fornecendo um apoio inestimável. Minha carreira científica e profissional foi e continua sendo construída e esculpida graças aos seus valiosos conselhos.

Ao professor Marcos José Santana e à professora Regina Helena C. Santana, por terem aberto as portas da USP para mim, pela valiosa orientação e pela paciência; pela confiança, hospitalidade, amizade, apoio e compreensão.

Ao professor Carlos Norberto Fischer, um grande amigo, que me orientou durante meu período de graduação, e traçou o meu caminho até a pós-graduação.

Ao professor Paulo Sérgio L. de Souza, pelo desenvolvimento do AMIGO, pelas valiosas sugestões no exame de qualificação e pela paciência quanto às minhas ousadas alterações em seu projeto.

Ao professor Edson dos Santos Moreira, pelas valiosas sugestões no exame de qualificação e pela supervisão durante minha segunda participação no Programa de Apoio ao Ensino. Também à professora Luciana Andreia F. Martimiano, pela supervisão durante minha

primeira participação no PAE.

Aos colegas e amigos do Grupo de Sistemas Distribuídos e Programação Concorrente, que durante os seminários ofereceram sugestões, críticas e argumentações que ajudaram a moldar o meu trabalho de pesquisa.

Aos funcionários e professores do Instituto de Ciências Matemáticas e de Computação, que de diversas formas indiretamente ajudaram o desenvolvimento deste trabalho.

À Universidade de São Paulo (USP), “*Scientia Vincet*”, como um todo, pela imensa qualidade de ensino e pela infra-estrutura que coloca à nossa disposição.

À Universidade Estadual Paulista (UNESP), pela qualidade de seu curso de graduação, que me permitiu ingressar na pós-graduação.

Finalmente, ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), pelo auxílio financeiro sem o qual este trabalho não seria possível.

Resumo

RAVASI, J. F. **JUMP: Uma política de escalonamento unificada com migração de processos.** 2009. Dissertação (Mestrado). Universidade de São Paulo, São Carlos, 2009.

Este trabalho apresenta o projeto e a implementação da política de escalonamento com suporte à migração de processos JUMP. A migração de processos é uma ferramenta importante que complementa a alocação inicial realizada pela política de escalonamento em um ambiente paralelo distribuído, permitindo um balanceamento de carga dinâmico e mais refinado, resultando em um melhor desempenho do ambiente e menor tempo de resposta das aplicações paralelas distribuídas. A nova política unifica a alocação inicial e migração de processos em um único algoritmo, de forma a compartilhar decisões para o objetivo comum de prover um melhor desempenho para aplicações de uso intensivo de processamento em *clusters* heterogêneos. A política é implementada sobre o ambiente de escalonamento flexível e dinâmico AMIGO, adaptado para o suporte à migração de processos. A avaliação de desempenho mostrou que a nova política oferece ganhos expressivos nos tempos de resposta quando comparada às outras duas políticas de escalonamento implementadas no AMIGO, em quase todos os cenários, para diversas aplicações e diversas situações de carga do ambiente.

Palavras-chave: política de escalonamento global de processos, balanceamento de carga, migração de processos.

Abstract

RAVASI, J. F. **JUMP: A unified scheduling policy with process migration.** 2009. Dissertation (Master's). Universidade de São Paulo, São Carlos, 2009.

This work presents the project and implementation of the scheduling policy with process migration support JUMP. Process migration is an important tool that complements the initial placement performed by the scheduling policy in a distributed parallel environment, allowing for dynamic and more refined load balancing, resulting in better performance of the environment and shorter response time for distributed parallel applications. The new policy unifies initial placement and process migration in a single algorithm, enabling the sharing of decisions for the common goal of providing a better performance for CPU-bound applications in heterogeneous clusters. The policy is implemented over the dynamical and flexible environment AMIGO, adapted in order to support process migration. Performance evaluation showed that the new policy offers expressive gains in response times when compared to other two scheduling policies implemented in AMIGO in almost all scenarios, for different applications and different environment load situations.

Keywords: global process scheduling policy, load balancing, process migration.

Lista de Figuras

Figura 2.1 - Taxonomia hierárquica de Casavant e Kuhl (1988).....	12
Figura 3.1 - Classificação de mecanismos de mobilidade de código.....	23
Figura 3.2 - Tipos de ligações entre processos, recursos e máquinas.....	24
Figura 3.3 - Algoritmo eager copy.....	31
Figura 3.4 - Algoritmo lazy copy.....	32
Figura 3.5 - Algoritmo pre-copy.....	32
Figura 3.6 - Algoritmo post-copy (Richmond e Hitchens, 1997).....	33
Figura 3.7 - Algoritmo flushing (Douglis e Ousterhout, 1991).....	34
Figura 4.1 - Esquema estrutural do AMIGO (Souza, 2000).....	44
Figura 4.2 - Elementos conceituais do ambiente AMIGO.....	46
Figura 4.3 - Camada inferior do AMIGO.....	47
Figura 4.4 - Cabeçalho das mensagens da camada inferior do AMIGO – projeto original (Souza, 2000).....	51
Figura 4.5 - Seqüência de mensagens para uma alocação simples de tarefas no ambiente AMIGO – projeto original.....	53
Figura 4.6 - Exemplo de seqüência de mensagens utilizando controle de seqüência.....	54
Figura 4.7 - Seqüência de mensagens para a migração de processos no ambiente AMIGO – projeto original.....	55
Figura 4.8 - Cabeçalho das mensagens da camada inferior.....	58
Figura 4.9 - Seqüência de mensagens para a migração de processos no ambiente AMIGO....	60
Figura 5.1 - Classificação da política JUMP pela taxonomia hierárquica de Casavant e Kuhl (1988).....	67
Figura 5.2 - Estrutura jumpLoadData (XDR).....	70
Figura 5.3 - Seqüência de mensagens para atualização de informações de carga.....	71
Figura 5.4 - Exemplo de alocação inicial em um cluster homogêneo desbalanceado.....	73
Figura 5.5 - Exemplo de alocação inicial em um cluster homogêneo balanceado.....	73
Figura 5.6 - Exemplo de alocação inicial em um cluster heterogêneo desbalanceado.....	74
Figura 5.7 - Exemplo de alocação inicial em um cluster heterogêneo balanceado.....	74
Figura 5.8 - Estruturas amMsgClientSched e amMsgPolicySched (XDR).....	76
Figura 5.9 - Estrutura amMsgPolicySchedReply e associadas (XDR).....	76
Figura 5.10 - Seqüência de mensagens para alocação inicial de tarefas.....	76
Figura 5.11 - Exemplo de migração de processos em um cluster desbalanceado.....	79
Figura 5.12 - Estruturas amMsgMigration e jumpMsgMigrationReply (XDR).....	81
Figura 5.13 - Seqüência de mensagens para migração de processos.....	81
Figura 6.1 - Fractal produzido pelo jrMandel, mostrando as divisões de tarefas.....	88
Figura 6.2 - Resultados da comparação do PVM-RR com o AMIGO-RR.....	92
Figura 6.3 - Resultados da comparação das políticas RR, DPWP e JUMP, sistema ocioso.....	94
Figura 6.4 - Execuções individuais para as aplicações sob controle da política JUMP.....	94
Figura 6.5 - Resultados da comparação das políticas RR, DPWP e JUMP (warm-up), sistema ocioso.....	95

Figura 6.6 - Resultados da comparação das políticas RR, DPWP e JUMP, sistema carregado.....	97
Figura 6.7 - Resultados da comparação das políticas RR, DPWP e JUMP, sistema com carga variável.....	100
Figura 6.8 - Comparação dos resultados para sistemas ocioso, carregado e com carga variável, para a política JUMP.....	101
Figura 6.9 - Tempos de resposta em relação à carga para a aplicação Integral.....	102
Figura 6.10 - Tempos de resposta em relação à carga para a aplicação Matriz.....	104
Figura 6.11 - Tempos de resposta em relação à carga para a aplicação jrMandel.....	105

Lista de Tabelas

Tabela 3.1 - Classificação de mecanismos de mobilidade de código.....	22
Tabela 3.2 - Tratamento de ligações a recursos durante a migração de processos.....	25
Tabela 3.3 - Níveis de implementação de migração de processos.....	36
Tabela 4.1 - Categorias de mensagens da camada inferior.....	59
Tabela 6.1 - Configuração do ambiente computacional utilizado na avaliação de desempenho.	84
Tabela 6.2 - Parâmetros das aplicações utilizadas no experimento.....	91
Tabela 6.3 - Resultados da comparação do PVM-RR com o AMIGO-RR.....	92
Tabela 6.4 - Testes de hipóteses para PVM-RR × AMIGO-RR.....	92
Tabela 6.5 - Diferenças dos tempos de resposta para PVM-RR × AMIGO-RR.....	93
Tabela 6.6 - Parâmetros das aplicações utilizadas no experimento.....	93
Tabela 6.7 - Resultados da comparação das políticas RR, DPWP e JUMP, sistema ocioso.....	93
Tabela 6.8 - Resultados da comparação das políticas RR, DPWP e JUMP (warm-up), sistema ocioso.....	95
Tabela 6.9 - Testes de hipóteses para RR × JUMP e DPWP × JUMP, sistema ocioso.....	96
Tabela 6.10 - Resultados da comparação das políticas RR, DPWP e JUMP, sistema carregado.....	97
Tabela 6.11 - Testes de hipóteses para RR × JUMP e DPWP × JUMP, sistema carregado.....	98
Tabela 6.12 - Resultados da comparação das políticas RR, DPWP e JUMP, sistema com carga variável.....	99
Tabela 6.13 - Testes de hipóteses para RR × JUMP e DPWP × JUMP, sistema com carga variável.....	100
Tabela 6.14 - Comparação dos resultados para sistemas ocioso, carregado e com carga variável, para a política JUMP.....	101
Tabela 6.15 - Cargas utilizadas para cada aplicação no experimento.....	102
Tabela 6.16 - Ganho em desempenho da política JUMP para a aplicação Integral.....	102
Tabela 6.17 - Testes de hipóteses para a aplicação Integral com diferentes cargas.....	103
Tabela 6.18 - Ganho em desempenho da política JUMP para a aplicação Matriz.....	103
Tabela 6.19 - Testes de hipóteses para a aplicação Matriz com diferentes cargas.....	104
Tabela 6.20 - Ganho em desempenho da política JUMP para a aplicação jrMandel.....	105
Tabela 6.21 - Testes de hipóteses para a aplicação jrMandel com diferentes cargas.....	105

Sumário

1. INTRODUÇÃO.....	1
1.1. Organização dos capítulos.....	4
2. COMPUTAÇÃO PARALELA DISTRIBUÍDA.....	7
2.1. Considerações Iniciais.....	7
2.2. Ambientes Paralelos Distribuídos.....	8
2.2.1. <i>Ambientes de passagem de mensagens</i>	9
2.3. Escalonamento de Processos.....	11
2.3.1. <i>Taxonomia de políticas de escalonamento</i>	11
2.4. Distribuição e Balanceamento de Carga.....	14
2.4.1. <i>Índices de carga</i>	15
2.4.2. <i>Organização de uma política de distribuição de carga</i>	16
2.4.3. <i>Estabilidade de uma política de distribuição de carga</i>	18
2.4.4. <i>Tipos de políticas de distribuição de carga</i>	18
2.5. Considerações Finais.....	19
3. MIGRAÇÃO DE PROCESSOS.....	21
3.1. Considerações Iniciais.....	21
3.2. Mobilidade de Código.....	21
3.2.1. <i>Gerenciamento de ligações a recursos locais</i>	23
3.3. Motivação da Migração de Processos.....	25
3.3.1. <i>Balanceamento dinâmico de carga</i>	26
3.3.2. <i>Aproximação dos processos aos recursos que acessam</i>	26
3.3.3. <i>Tolerância a falhas</i>	27
3.3.4. <i>Administração do sistema</i>	27
3.4. Políticas de Migração de Processos.....	27
3.5. Mecanismos de Migração de Processos.....	28
3.5.1. <i>Procedimento de migração de um processo</i>	29
3.5.2. <i>Projeto de mecanismos de migração de processos</i>	29
3.5.3. <i>Algoritmos de migração de processos</i>	30
3.5.4. <i>Classificação quanto ao paradigma de programação paralela</i>	35
3.6. Implementação do Suporte à Migração de Processos.....	36
3.6.1. <i>Nível de sistema</i>	37
3.6.2. <i>Biblioteca de sistema (nível de usuário)</i>	38
3.6.3. <i>Aplicação (nível de usuário)</i>	39
3.7. Checkpointing.....	39
3.8. Ambientes com Suporte à Migração de Processos.....	40
3.8.1. <i>Single System Image (SSI)</i>	40
3.8.2. <i>Checkpoint/Restart</i>	41

3.9. Considerações Finais.....	42
4. O AMBIENTE DE ESCALONAMENTO AMIGO.....	43
4.1. Considerações Iniciais.....	43
4.2. Projeto Original.....	44
4.2.1. Elementos conceituais do ambiente.....	45
4.2.2. A camada inferior.....	47
4.2.3. Comunicação na camada inferior.....	50
4.2.4. Migração de processos.....	51
4.2.5. Considerações sobre o projeto original.....	52
4.3. Alterações no Projeto.....	56
4.3.1. O AMIGO Daemon.....	56
4.3.2. Protocolo de comunicação.....	57
4.3.3. Migração de processos.....	59
4.3.4. Implementação.....	60
4.4. Considerações Finais.....	62
5. A POLÍTICA DE MIGRAÇÃO JUMP.....	65
5.1. Considerações Iniciais.....	65
5.2. Objetivos Gerais.....	65
5.3. Desenvolvimento da Política JUMP.....	66
5.4. Gerenciamento de Informações de Carga.....	68
5.4.1. Métricas e índices de carga do sistema.....	68
5.4.2. Métricas de carga de processos.....	69
5.4.3. Compartilhamento de informações de carga.....	70
5.5. Política de Alocação Inicial.....	71
5.5.1. Componentes da política da alocação inicial.....	75
5.5.2. Comunicação durante a alocação inicial.....	75
5.6. Política de Migração de Processos.....	76
5.6.1. Componentes da política de migração.....	79
5.6.2. Comunicação durante a migração de processos.....	80
5.7. Considerações Finais.....	81
6. AVALIAÇÃO DE DESEMPENHO E RESULTADOS.....	83
6.1. Considerações Iniciais.....	83
6.2. Plataforma Utilizada.....	83
6.3. Aplicações Utilizadas.....	84
6.3.1. Método dos trapézios compostos.....	85
6.3.2. Multiplicação de matrizes.....	86
6.3.3. Desenho de fractais Mandelbrot.....	86
6.4. Análise Estatística Considerada.....	88
6.5. Experimentos Realizados.....	90
6.5.1. Sobrecarga imposta pelo ambiente AMIGO.....	91
6.5.2. Desempenho das políticas em um sistema ocioso.....	93
6.5.3. Desempenho das políticas em um sistema carregado.....	96
6.5.4. Desempenho das políticas em um sistema com carga variável.....	99
6.5.5. Desempenho da política JUMP para diferentes cargas.....	101
6.6. Considerações Finais.....	106

7. CONCLUSÕES E TRABALHOS FUTUROS.....	107
7.1. Considerações Finais.....	107
7.2. Contribuições deste Trabalho.....	107
7.3. Dificuldades Encontradas.....	109
7.4. Sugestões para Trabalhos Futuros.....	110
REFERÊNCIAS.....	111

1. Introdução

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable. — Leslie Lamport

Durante as últimas décadas foi observado um crescimento expressivo da presença dos sistemas computacionais no cotidiano. A introdução do computador pessoal e o grande crescimento do número de computadores transformaram uma realidade onde originalmente diversos usuários precisavam revezar pela sua utilização, para uma nova realidade onde é comum uma única pessoa possuir diversos computadores ao seu dispor.

Não apenas o computador se tornou um objeto mais acessível e presente na sociedade, como também as redes de computadores evoluíram de forma a acompanhar tal crescimento. Inicialmente confinada aos limites físicos de um departamento ou prédio, as redes evoluíram em tecnologia de mídias de comunicação, velocidade de transmissão e alcance, permitindo a interconexão de diversas filiais distantes de corporações, crescendo e se conectando a outras redes, até finalmente a atingir a presença ubíqua da grande rede mundial – a Internet.

Um conjunto de computadores independentes, interconectados, transmitindo mensagens entre si para uma determinada finalidade forma um Sistema Computacional Distribuído¹. A área de Sistemas Distribuídos compreende desde a simples transmissão de dados entre dois computadores independentes (modelo cliente-servidor simples), até complexas malhas hierárquicas de agrupamentos de computadores (grades computacionais).

A área de Computação de Alto Desempenho, originalmente preocupada com o desenvolvimento e aplicação de arquiteturas superescalares, multiprocessadores e multicomputadores² com o propósito de resolver complexos problemas computacionais através da Computação Paralela (Quinn, 1994), viu na evolução dos sistemas distribuídos uma nova oportunidade de avanço, através do desenvolvimento de *clusters* (agrupamentos) de computadores, surgindo a Computação Paralela Distribuída.

Os sistemas distribuídos oferecem diversas vantagens técnicas e econômicas quando

¹ Daqui em diante chamado sucintamente de “sistema distribuído”.

² Daqui em diante chamadas coletivamente de “arquiteturas paralelas”.

comparados às arquiteturas paralelas. Em um sistema distribuído, a manutenção e substituição de componentes são mais simples e menos custosas, pois suas partes são geralmente construídas a partir de peças comuns de mercado; é mais simples crescer o sistema, pois sua capacidade de processamento pode ser aumentada sem fazer alterações aos componentes preexistentes; e oferece maior flexibilidade quanto à distribuição geográfica, pois suas partes podem estar tanto em uma mesma sala como em diferentes continentes.

Apesar de servirem a um propósito similar – a resolução de problemas complexos através de processamento em paralelo – os *clusters* de computadores impõem desafios diferentes à computação paralela distribuída do que aqueles encontrados na aplicação de arquiteturas paralelas, especialmente em relação a tolerância a falhas e a comunicação entre os elementos de processamento.

Dada a sua natureza, sistemas distribuídos são mais propensos a falhas parciais. A maior quantidade de processadores independentes³ resulta em maiores chances de, em dado momento, pelo menos um dos processadores estar incapacitado. A rede de comunicações entre os computadores também é uma variável importante, em dado momento a rede pode falhar de forma que uma ou mais partes do sistema estejam incapacitadas de se comunicar com as outras. Tanto a falha parcial quanto o isolamento de partes do sistema distribuído não devem impedir que o resto do sistema continue funcionando (Tanenbaum e Steen, 2002; Coulouris et al., 2005).

Não apenas a falha da rede de comunicações é uma preocupação, mas o custo de sua utilização (vazão limitada) representa mais um desafio. A interconexão entre os elementos de processamento em um sistema distribuído possui uma vazão de dados menor do que a comunicação entre processadores em uma única máquina (Tanenbaum e Steen, 2002).

A concorrência em sistemas distribuídos é uma norma, uma vez que cada elemento de processamento possui um alto nível de independência em relação ao resto do sistema. Usuários podem trabalhar individualmente em diferentes computadores, e ainda assim compartilhar recursos através da rede de computadores.

Em arquiteturas paralelas, dois paradigmas de comunicação e sincronização entre elementos de processamento são possíveis: memória compartilhada e passagem de mensagens. Em um sistema distribuído, uma vez que cada elemento de processamento possui sua própria memória local, a comunicação ocorre exclusivamente através de passagem de mensagens (Tanenbaum e Steen, 2002; Coulouris et al., 2005).

³ Um processador independente é aquele que está contido em seu próprio sistema computacional completo, com barramento, memória, dispositivos de entrada/saída e periféricos.

A execução de uma aplicação paralela distribuída sobre um sistema distribuído depende da disponibilidade de um conjunto de funcionalidades que apóiam a comunicação e o gerenciamento das tarefas através do *cluster* de computadores. Tal plataforma de *software* é chamada de ambiente paralelo distribuído, plataforma computacional distribuída, ambiente de passagem de mensagens ou máquina paralela virtual.

A comunicação custosa através da rede de computadores e a exigência da utilização de passagem de mensagens tornam o escalonamento de processos em um ambiente paralelo distribuído um componente de particular importância. Enquanto que em arquiteturas paralelas o escalonamento local realizado pelo sistema operacional pode interromper e comutar processos entre processadores rapidamente, pois todo o contexto da aplicação paralela se encontra localmente e é transmitido através de memória compartilhada, em sistemas distribuídos a escolha do elemento de processamento que irá executar a tarefa (escalonamento global) implica na transmissão do contexto da aplicação através da rede, que possui uma menor vazão de dados. O escalonamento de processos em um ambiente paralelo distribuído é controlado por uma ou mais políticas de escalonamento.

Em geral, políticas de escalonamento global procuram determinar o elemento de processamento que irá executar o processo desde a sua criação até o seu término. Tais políticas são chamadas de políticas de alocação inicial, e não realizam preempção. O objetivo principal de tais políticas é determinar quais elementos de processamento serão os receptores de novas tarefas, com o objetivo de compartilhar ou balancear a carga através do sistema distribuído (Shivaratri et al., 1992).

Contudo, nem sempre a política de escalonamento é capaz de sustentar um bom balanceamento de carga apenas através da alocação inicial, seja por falta de informações sobre o sistema distribuído ou sobre a aplicação sendo escalonada durante o processo de escalonamento, ou por ser impossível prever futuras alterações na carga dos elementos de processamento causadas por eventos externos ao ambiente sobre o qual a aplicação paralela é executada. Nessas situações, é desejável mover (ou migrar) processos durante sua execução entre diferentes elementos de processamento do sistema distribuído.

Uma política de escalonamento com migração de processos realiza a preempção de processos através do sistema distribuído, isto é, possui heurísticas que procuram detectar situações subótimas e realizar a migração de processos entre os computadores para reequilibrar a distribuição de carga. Outros objetivos que justificam a migração de processos incluem tolerância a falhas e a capacidade de administrar o sistema. A migração de processos envolve interações complexas com o sistema operacional e possui diversas limitações e

dificuldades que precisam ser tratadas para sua aplicação com sucesso em um ambiente paralelo distribuído.

Devido à importância e à complexidade do escalonador de processos em um ambiente paralelo distribuído, não é incomum a utilização de uma ferramenta de *software* dedicada a essa finalidade. Alguns exemplos de tais ferramentas são Condor (Thain et al., 2005), LSF (Platform, 2006), LoadLeveler (Kannan et al., 2001) e Sun Grid Engine (Gentzsch, 2001). O AMIGO (Souza et al., 1999; Souza, 2000) é um ambiente de escalonamento flexível e dinâmico, desenvolvido no Instituto de Ciências Matemáticas e de Computação, que fornece uma plataforma para a implementação, execução e gerenciamento de políticas de escalonamento de processos, reduzindo a lacuna entre teoria e prática nessa área de pesquisa. A característica diferencial do AMIGO em relação às outras ferramentas para essa finalidade é a sua capacidade de executar múltiplas políticas de escalonamento simultaneamente, definindo um protocolo de comunicação entre políticas e ambientes paralelos distribuídos, flexibilizando o escalonamento de processos e permitindo a comutação entre políticas ativas de forma transparente ao usuário.

O projeto original do AMIGO contemplava a migração de processos em sua especificação, definindo um conjunto de mensagens e rotinas para tal finalidade; contudo, a implementação dessa capacidade ainda não havia sido realizada.

Este trabalho aborda a migração de processos como forma de habilitar um balanceamento de carga mais eficiente em ambientes paralelos distribuídos. Entre as atividades que foram praticadas estão a adaptação do ambiente e implantação do suporte à migração de processos no AMIGO, a adaptação de um ambiente de passagem de mensagens com suporte à migração de processos para a sua utilização integrada com o AMIGO, o projeto e implementação de uma política de escalonamento unificada com suporte à alocação inicial e migração de processos e, finalmente, a avaliação e análise do desempenho dessa nova política de escalonamento.

1.1. Organização dos capítulos

Esta dissertação está organizada em uma seqüência de capítulos que percorre a contextualização, revisão bibliográfica, descrição das ferramentas utilizadas, descrição dos produtos deste trabalho, avaliação de desempenho e conclusões.

O próximo capítulo introduz os conceitos de computação paralela distribuída, escalonamento de processos e distribuição e balanceamento de carga, que são utilizados neste

trabalho.

O capítulo 3 detalha a migração de processos, desde conceitos fundamentais de mobilidade de código, motivação da migração de processos, distinção entre políticas e mecanismos de migração de processos, os diversos algoritmos que podem ser usados em mecanismos de migração de processos, implementação e ambientes com suporte à migração de processos.

O capítulo 4 apresenta o ambiente de escalonamento AMIGO, seu projeto original, considerações sobre o projeto original, e as alterações que foram necessárias para atingir o objetivo do trabalho.

O capítulo 5 apresenta a política de escalonamento unificada com suporte à migração de processos JUMP. Os objetivos e o projeto da política são apresentados, seguido da descrição dos componentes de sua implementação: o gerenciamento de informações de carga, a política de alocação inicial e a política de migração de processos.

O capítulo 6 descreve a avaliação e análise do desempenho do ambiente AMIGO e da nova política de escalonamento JUMP, incluindo a plataforma e as aplicações utilizadas, os resultados obtidos e discussão sobre os resultados.

Por fim, o capítulo 7 conclui esta dissertação, apresentando as considerações finais sobre o trabalho, suas principais contribuições, bem como as dificuldades encontradas e as sugestões para trabalhos futuros.

2. *Computação Paralela Distribuída*

2.1. *Considerações Iniciais*

Sistemas distribuídos dão suporte à computação paralela através de *clusters* de computadores, às vezes também denominados de redes de estações de trabalho (*NOW – networks of workstations*)⁴ (Anderson et al., 1995). A configuração básica de um *cluster* é uma coleção de computadores conectados através de uma rede de comunicações (Tanenbaum e Steen, 2002). Um computador que é parte de um *cluster* de computadores é também chamado de nó.

Clusters de computadores também podem servir para outras finalidades além da computação paralela, como por exemplo, os *clusters* de alta disponibilidade, que visam permitir um alto nível de tolerância a falhas através de redundância de recursos, sem que necessariamente os computadores realizem processamento em paralelo.

Algumas configurações possuem características especiais e são reconhecidas por nomes específicos, como *clusters Beowulf* (Sterling et al., 1995) e grades computacionais (Foster e Kesselman, 2004). Este trabalho não irá abordar tais configurações diretamente, mas os resultados são igualmente válidos, pois essas configurações compartilham conceitos fundamentais comuns da computação paralela distribuída.

Uma aplicação paralela é executada sobre um ambiente paralelo distribuído, que implementa funcionalidades essenciais para a comunicação entre tarefas e o gerenciamento do *cluster* de computadores.

Este capítulo introduz os conceitos de computação paralela distribuída, escalonamento de processos e distribuição de carga. Essa é uma área de pesquisa muito extensa e, assim, apenas os assuntos relevantes ao projeto apresentado neste trabalho serão abordados.

⁴ Neste trabalho, a primeira definição – *clusters* de computadores – é preferida por ser indiferente ao tipo de computador utilizado.

2.2. Ambientes Paralelos Distribuídos

Um ambiente paralelo distribuído, também chamado de plataforma computacional distribuída ou de ambiente de passagem de mensagens⁵, é um conjunto de ferramentas de *software* executadas sobre um *cluster* de computadores, que cria uma camada sobre a qual aplicações distribuídas são executadas, aumentando a transparência do sistema distribuído para a aplicação. Tal ferramenta é classificada como um *middleware* (Tanenbaum e Steen, 2002).

O ambiente paralelo distribuído possui três funções principais:

1. Alocação e gerenciamento de recursos do *cluster*;
2. alocação e gerenciamento de aplicações e processos;
3. passagem de mensagens entre processos distribuídos.

A primeira função, gerenciamento de recursos, consiste em detectar e conhecer a configuração do *cluster* de computadores, como o número de computadores disponíveis, a configuração de cada computador (tipo de arquitetura, tipo e número de processadores, quantidade de memória, sistema operacional, etc), seus endereços de rede para comunicação, etc. A adição e remoção de componentes do *cluster* em tempo de execução também é controlada pelo ambiente. O ambiente pode também monitorar a utilização (através de índices de carga) dos computadores do *cluster*. Essas informações são utilizadas tanto para atividades do próprio ambiente, quanto fornecidas para aplicações paralelas executadas sobre ele.

A segunda função, gerenciamento de aplicações e processos, consiste em manter e gerenciar uma lista de processos em execução em cada computador, detectar a criação e o término de aplicações distribuídas, receber solicitações de criação de processos (tanto do usuário como de outros processos já em execução) e alocar computadores do *cluster* que receberão os novos processos para execução. Esta última função é chamada de escalonamento global de processos e será abordada em maior profundidade na Seção 2.3.

A terceira função, passagem de mensagens, consiste em habilitar a comunicação entre processos em execução sobre o ambiente. Mensagens transmitidas entre dois processos ocorrem com o auxílio do ambiente, que recebe a mensagem do primeiro processo, determina o seu destino, a retransmite ao computador apropriado e por fim a entrega ao segundo

⁵ Neste trabalho, a primeira definição – ambiente paralelo distribuído – é utilizada, pois enfatiza a natureza distribuída do ambiente e seu objetivo de servir à computação paralela. A segunda definição – plataforma computacional distribuída – não destaca o uso da computação paralela, e a terceira definição – ambiente de passagem de mensagens – é uma confusão comum entre o ambiente como um todo e um de seus componentes.

processo. Esse procedimento é transparente à aplicação distribuída: o processo que remete a mensagem não precisa saber se o processo de destino está sendo executado local ou remotamente, ou se os dados devem ser convertidos durante a transmissão.

2.2.1. Ambientes de passagem de mensagens

Existem inúmeras ferramentas para a criação de ambientes paralelos distribuídos, tanto comerciais quanto de código aberto. O componente responsável pela passagem de mensagens é, em geral, a primeira característica que diferencia os diversos ambientes, pois é o componente que interage diretamente com as aplicações paralelas, através de uma API (*Application Programming Interface*). Por esse motivo, aplicações paralelas são escritas visando um determinado ambiente de passagem de mensagens, tornando-as dependentes do mesmo.

Três ambientes de passagem de mensagens são mais comumente utilizados: *Parallel Virtual Machine* (PVM), *Message Passing Interface* (MPI) e *Common Object Request Broker Architecture* (CORBA).

a. *Parallel Virtual Machine* (PVM)

O *Parallel Virtual Machine* é um pacote de *software* de código aberto composto por uma biblioteca de funções, um *daemon* (serviço de sistema) e utilitários que permitem que um conjunto de computadores heterogêneos, utilizando sistemas operacionais Unix e Windows, ligados em rede, sejam utilizados como uma única máquina paralela virtual. O PVM gerencia de forma transparente o roteamento de mensagens, a conversão de dados e o escalonamento de processos (chamados de “tarefas”), através de arquiteturas computacionais incompatíveis (Geist et al., 1994).

O PVM utiliza um modelo computacional simples e genérico, que pode ser utilizado por diversos tipos de aplicações paralelas distribuídas. A biblioteca de funções permite iniciar e terminar tarefas em outros computadores do sistema, suspender e reiniciar outras tarefas, adicionar ou remover computadores da máquina virtual, sincronizar com outras tarefas, etc (Geist et al., 1994).

Dada a abrangência de seus recursos, o PVM não é apenas um ambiente de passagem de mensagens, mas um ambiente paralelo distribuído completo. A interface de programação do PVM é considerada simples e completa, e isso o tornou largamente aceito na área de computação de alto desempenho.

b. Message Passing Interface (MPI)

O *Message Passing Interface*, ao contrário do PVM, não é um pacote de *software*, mas, sim, uma especificação de interface de programação para comunicação e coordenação de aplicações paralelas distribuídas. Dessa forma, uma aplicação distribuída pode ser escrita utilizando a API definida pelo MPI, e essa aplicação será compatível com qualquer ambiente que implemente a especificação. A especificação MPI é coordenada pelo MPI Forum (2008a; 2008b).

O objetivo do MPI é ser um padrão largamente utilizado para o desenvolvimento de aplicações construídas sobre o modelo de passagem de mensagens. Para isso, a interface deve estabelecer um padrão prático, portátil, eficiente e flexível para passagem de mensagens.

A primeira versão do padrão – MPI-1 – especifica serviços de comunicação ponto-a-ponto, comunicação coletiva, grupos de processos, contextos de comunicação, topologias de processos, gerenciamento do ambiente e interface para monitoramento (MPI Forum, 2008a).

A segunda versão do padrão – MPI-2 – estende a especificação anterior, adicionando serviços de criação e gerenciamento de processos, comunicação unilateral, operações coletivas estendidas e entrada e saída paralelas (MPI Forum, 2008b).

Há diversas implementações da especificação MPI, comerciais e de código aberto, destacando-se as implementações MPICH (Gropp et al., 1996), LAM/MPI (Burns et al., 1994) e OpenMPI (Gabriel et al., 2003).

c. Common Object Request Broker Architecture (CORBA)

O *Common Object Request Broker Architecture* é uma especificação definida pelo *Object Management Group* para permitir a interação de componentes de software escritos em diferentes linguagens de programação e, possivelmente, executando em diferentes computadores de um sistema distribuído (OMG, 2001; Coulouris et al., 2005).

O CORBA é fundamentado sobre o paradigma de orientação a objetos. Objetos em diferentes aplicações, escritos em diferentes linguagens e executando em diferentes computadores podem trocar mensagens de forma transparente à aplicação. Para isso, objetos devem ser definidos pelo programador utilizando-se uma linguagem de definição de interfaces (*IDL – Interface Definition Language*), que é utilizada pelo CORBA para gerar as rotinas de comunicação entre objetos.

Assim como o MPI, CORBA é apenas uma especificação, a partir da qual existem diversas implementações, tais como MICO (Puder, 2004) e TAO (Schmidt et al., 1998).

Diferentemente do PVM e do MPI, a comunicação na arquitetura CORBA não ocorre diretamente sobre o paradigma de passagem de mensagens, mas sobre um paradigma de invocação remota com o suporte à orientação a objetos.

2.3. Escalonamento de Processos

O escalonamento de processos é a atividade de atribuir processos a processadores para a sua execução e pode ocorrer em dois níveis. No primeiro nível, o escalonamento local é responsável por intercalar a execução dos processos locais do sistema operacional, determinando a ordem de execução, qual processador executará cada processo (para arquiteturas multiprocessadores) e o *quantum* de tempo que cada processo será executado. A implementação desse tipo de escalonamento de processos é de interesse da área de sistemas operacionais e não será abordado neste trabalho. Tanenbaum (2003) trata desse tipo de escalonamento.

No segundo nível, o escalonamento global é responsável por determinar quais computadores do sistema distribuído são mais adequados para a execução do processo, considerando os objetivos e a plataforma como um todo (Souza, 2000).

Casavant e Kuhl (1988) definem o escalonamento global de processos como um “recurso de gerenciamento de recursos”, responsável por gerenciar de forma eficiente e efetiva o acesso e o uso de um recurso por seus vários consumidores. O escalonador, componente do sistema responsável pelo escalonamento, é controlado por uma política.

O escalonamento global de processos é definido de forma relativamente consistente entre os diversos autores, mas com grandes divergências na nomenclatura. Casavant, Kuhl (1988) e Souza (2000) usam a mesma denominação utilizada neste trabalho: escalonamento global de processos. Mullender (1993) refere-se ao escalonamento global como “gerenciamento de processos”. Tanenbaum (1995) prefere utilizar “alocação de processador”. Coulouris et al. (2005) usam simplesmente “criação de processos”.

2.3.1. Taxonomia de políticas de escalonamento

Casavant e Kuhl (1988) propuseram uma taxonomia para classificar políticas de escalonamento global, com o objetivo de fornecer um conjunto de termos e um mecanismo que permita a comparação de algoritmos. Outras taxonomias existem, mas para os efeitos deste trabalho, apenas a de Casavant e Kuhl será utilizada. Souza (2000) aborda as outras taxonomias, assim como fornece um estudo mostrando a convergência entre as mesmas.

A taxonomia de Casavant e Kuhl possui duas partes: uma classificação hierárquica e uma classificação plana. A taxonomia hierárquica é mostrada na Figura 2.1 e separa as políticas de escalonamento nos seguintes grupos:

Figura 2.1 - Taxonomia hierárquica de Casavant e Kuhl (1988).

- **Local versus global:** conforme já explicado anteriormente, o escalonamento local é aquele realizado pelo sistema operacional para escalonar os processos locais, enquanto que o global cuida da distribuição dos processos entre os computadores do sistema distribuído. Como o trabalho de Casavant e Kuhl é direcionado ao escalonamento global, não há subclassificações para o escalonamento local.
- **Estático versus dinâmico:** nesse nível, as políticas de escalonamento são classificadas de acordo com o momento em que decisões de escalonamento são tomadas. No escalonamento global estático, todas as informações necessárias para a decisão estão disponíveis em tempo de compilação e a decisão é tomada sem levar em conta o estado do sistema distribuído no momento que o escalonamento é realizado. No escalonamento global dinâmico, a política de escalonamento utiliza informações sobre o estado atual do sistema para determinar qual computador do sistema executará o processo.
- **Ótimo versus subótimo:** nesse nível, as políticas de escalonamento são classificadas de acordo com sua capacidade de determinar precisamente o melhor destino para a execução do processo, ou simplesmente um destino considerado bom. No escalonamento ótimo, todas as informações sobre o estado do sistema e dos requisitos de recursos estão disponíveis e são usadas no momento do

escalonamento e, portanto, uma atribuição ótima pode ser realizada com base em algum critério. No escalonamento subótimo, a decisão é realizada usando apenas uma parte das informações sobre o estado do sistema.

- **Aproximado versus heurístico:** as políticas de escalonamento subótimas, por sua vez, são classificadas em duas subcategorias. No escalonamento subótimo aproximado, utiliza-se um modelo computacional formal para descrever o algoritmo, que observa apenas uma parte do espaço de soluções para achar uma “boa” solução, não necessariamente a melhor. No escalonamento subótimo heurístico, o algoritmo cria hipóteses sobre as características do processo e do sistema, sem necessariamente observar todo o sistema diretamente. Nessa categoria, utilizam-se variáveis que representam o estado do sistema indiretamente, e que são mais fáceis de observar do que tentar analisar o sistema como um todo.
- **Distribuído versus não-distribuído:** sob o ramo das políticas de escalonamento dinâmicas, essas são classificadas primeiramente de acordo com a responsabilidade pela escolha do destino do processo. No escalonamento global distribuído, cada elemento do sistema distribuído tem autoridade para realizar a decisão do escalonamento. No escalonamento global não-distribuído, um único elemento do sistema centraliza as decisões de escalonamento.
- **Cooperativo versus não-cooperativo:** as políticas de escalonamento dinâmicas distribuídas são subdivididas, por sua vez, de acordo com a preocupação de cada elemento do sistema em relação ao desempenho próprio ou do sistema como um todo. No escalonamento distribuído cooperativo, os elementos trocam informações entre si de forma que todos tomem decisões de escalonamento que beneficiem o sistema como um todo, a fim de atingir um objetivo global. No escalonamento distribuído não-cooperativo, os elementos tomam decisões de escalonamento de acordo com suas próprias metas, sem considerar o impacto que causará sobre os outros elementos.

A taxonomia plana complementa a classificação das políticas de escalonamento de acordo com seus objetivos ou sua forma de funcionamento. As categorias da taxonomia plana de Casavant e Kuhl são:

- **Adaptável versus não-adaptável:** uma política de escalonamento é adaptável quando o algoritmo e os parâmetros da política mudam dinamicamente de acordo

com o comportamento do sistema, em resposta às decisões tomadas pelo sistema de escalonamento.

- **Balanceamento de carga:** uma política de escalonamento com balanceamento de carga tem como objetivo equilibrar a carga (de processamento, utilização de memória, etc.) de todos os processadores, de forma a permitir que todos os processos avancem na mesma velocidade. O balanceamento de carga será abordado na próxima seção deste capítulo.
- **Licitação (*bidding*):** uma política de escalonamento com licitação utiliza um protocolo onde computadores cooperam para a troca de tarefas. Computadores assumem papéis de gerentes, que solicitam apoio de outros computadores para o escalonamento de um processo; e de contratantes, que se oferecem para a execução do processo.
- **Probabilístico:** uma política de escalonamento probabilística simula aleatoriamente um conjunto de possíveis escalonamentos e então escolhe o melhor escalonamento entre aqueles gerados aleatoriamente. Dessa forma, reduz-se a sobrecarga de processamento gerada pelo escalonador, uma vez que não são verificadas todas as possibilidades de escalonamento.
- **Atribuição inicial versus reatribuição dinâmica:** diz-se que a política de escalonamento pertence à classe de atribuição inicial quando a decisão de onde o processo será executado é feita apenas uma vez no momento de seu escalonamento, de forma irrevogável, não preemptivo. O processo permanece executando no computador escolhido até o seu término. Se a política de escalonamento é capaz de alterar o escalonamento já realizado, ou reatribuir um destino diferente para as subtarefas do processo previamente escalonado, diz-se que a política pertence à classe de reatribuição dinâmica.

2.4. Distribuição e Balanceamento de Carga

O principal objetivo do escalonamento de processos é distribuir a carga entre os vários computadores que compõem o sistema distribuído, de forma a melhor utilizar os recursos disponíveis. Dá-se a esse processo o nome de distribuição de carga.

A distribuição de carga consiste em transferir tarefas ou processos de um computador mais carregado para um computador menos carregado. Aqui, tarefas correspondem aos processos que ainda não iniciaram a sua execução e, portanto, não possuem estado de

execução ou espaço de dados. De forma complementar, processos são as tarefas que já iniciaram sua execução⁶.

Shivaratri et al. (1992) classifica os algoritmos de distribuição de carga em duas categorias, dependendo dos objetivos de cada algoritmo. Algoritmos classificados como de compartilhamento de carga são aqueles cujo objetivo é maximizar a velocidade que o sistema distribuído realiza o trabalho, evitando a ociosidade de processadores. Algoritmos classificados como de balanceamento de carga são aqueles que, além de evitar a ociosidade de processadores, tentam igualar o nível de carga de todos os computadores.

A política de escalonamento utilizada para distribuição de carga em um sistema distribuído também é chamada de política de distribuição de carga ou política de balanceamento de carga. A política de distribuição de carga deve responder às seguintes perguntas (Eskicioğlu, 1995; Milojić et al., 2000):

- **Quando** deve ocorrer a transferência;
- **qual** tarefa deve ser transferida;
- **para onde** ela deve ser transferida.

Uma política de distribuição de carga não-preemptiva é aquela que apenas determina o computador onde o processo será criado, ou move tarefas (processos inativos) entre computadores. Uma política de distribuição de carga preemptiva é aquela capaz de executar migração de processos ativos entre os computadores do sistema. A utilização da migração de processos como forma de distribuição de carga será tratada no capítulo seguinte.

2.4.1. Índices de carga

Antes de se falar sobre políticas de distribuição de carga, é necessário definir como a carga do sistema é medida, a fim de comparar a quantidade de carga de trabalho de cada elemento do sistema. Um índice de carga é uma métrica que quantifica a utilização de um elemento do sistema (Ferrari e Zhou, 1987; Kunz, 1991; Branco, 2004).

Um índice de carga é definido como uma variável não-negativa, sendo o valor zero se o recurso está ocioso e um valor positivo crescente conforme a carga sobre esse recurso aumenta (Ferrari e Zhou, 1987; Kunz, 1991; Branco, 2004).

Vários índices de carga podem ser utilizados para representar a carga do sistema para fins de distribuição de carga, o índice mais adequado depende do tipo de aplicação e de como

⁶ Note que o ambiente PVM (e o DPVM), abordado neste trabalho, chama processos ativos de tarefas. Neste trabalho, a distinção entre tarefas e processos é mantida.

a distribuição deve ser feita. Alguns índices de carga que podem ser utilizados são: tamanho da fila de processamento, média do tamanho da fila de processamento durante certo período, quantidade de memória disponível, taxa de troca de contexto, taxa de chamadas de sistema e utilização do processador (Shivaratri et al., 1992).

É importante que o mecanismo utilizado para medir a carga do sistema e calcular o índice de carga imponha o mínimo possível de sobrecarga, a fim de não afetar o desempenho do sistema e, portanto, a qualidade do índice de carga (Ferrari e Zhou, 1987; Shivaratri et al., 1992; Souza, 2000).

Branco (2004) faz um estudo detalhado sobre índices de carga, em especial sobre sua aplicabilidade em relação à heterogeneidade de um sistema distribuído.

2.4.2. Organização de uma política de distribuição de carga

As políticas de distribuição de carga podem ser divididas em componentes, de forma a tornar a sua implementação mais simples e modular. Cada componente da política cuida de uma atividade específica do escalonamento, de forma que a combinação de todos os componentes determina o comportamento do escalonador de processos. A divisão da política de distribuição de carga em componentes permite que diferentes componentes (compatíveis entre si) sejam escolhidos para cada uma das tarefas, criando uma política de distribuição de carga específica para determinada aplicação.

Shivaratri et al. (1992) definem quatro componentes para uma política de distribuição de carga: política de transferência, política de seleção, política de localização e política de informação.

a. Política de transferência

O objetivo da política de transferência é determinar se um nó do sistema distribuído está apto a participar da transferência de uma tarefa, tanto como remetente quanto como destinatário.

Uma das formas de se implementar uma política de transferência é através de limites. Nessa técnica, definem-se dois limites T_1 e T_2 que serão utilizados para comparar a carga do nó. Se a carga do nó ultrapassa T_1 , ele é considerado um remetente; se a carga cair abaixo de T_2 , ele é considerado um destinatário. Uma alternativa ao uso de limites é o uso de uma política relativa, que compara a carga do nó com a de outros nós do sistema distribuído para classificá-lo como remetente ou destinatário (Shivaratri et al., 1992).

b. Política de seleção

A política de seleção é utilizada quando a política de transferência determina que o nó é um remetente. A política de seleção é então responsável por escolher uma tarefa ou processo que será transferido para outro nó (Shivaratri et al., 1992).

Algumas abordagens para a política de seleção são a escolha de uma tarefa aleatória, ou a escolha da tarefa mais recentemente criada (Shivaratri et al., 1992). A escolha de uma tarefa é desejável sobre a escolha de um processo ativo, pois evita a migração de processos (Souza, 2000).

Uma preocupação que a política de seleção deve ter é a de escolher uma tarefa que incorra a menor sobrecarga durante a transferência e tenha poucas dependências com o seu nó de origem, de forma a reduzir o número de chamadas de sistema remotas.

c. Política de localização

A política de localização é responsável por encontrar um nó parceiro adequado que irá participar da transferência de tarefas. Caso o nó tenha sido classificado pela política de transferência como um remetente, a política de localização deverá encontrar um nó destinatário para participar da transferência, e vice-versa.

A política de localização pode ser tanto distribuída quanto centralizada. Um exemplo de política de localização distribuída é a realização de uma pesquisa (*polling*), onde o nó consulta outros nós para encontrar um parceiro para a transferência. Outro exemplo de política de localização distribuída é através de difusão (*broadcast*), onde o nó envia um aviso a todos os outros nós a fim de encontrar um nó disponível (Shivaratri et al., 1992).

Numa política de localização centralizada, um determinado nó se torna responsável por concentrar as informações sobre o sistema (obtidos através da política de informação) e decidir qual será o nó parceiro de cada transferência (Shivaratri et al., 1992).

d. Política de informação

A política de informação é responsável por gerenciar a coleta e o armazenamento de informações sobre o estado do sistema distribuído. A política determina quando as informações são coletadas, de onde elas serão coletadas e quais informações serão coletadas (Shivaratri et al., 1992). As políticas de informação são classificadas em políticas sob demanda, periódicas ou guiadas por mudança de estado.

- **Sob demanda:** o nó passa a coletar informações de estado de outros nós quando este se torna um remetente ou um destinatário.

- **Periódicas:** o nó coleta informações do estado de outros nós periodicamente.
- **Por mudança de estado:** o nó divulga seu estado para os outros nós do sistema sempre que alguma mudança significativa ocorre.

2.4.3. Estabilidade de uma política de distribuição de carga

Shivaratri et al. (1992) definem a estabilidade de um algoritmo de distribuição de carga por duas perspectivas diferentes: através de uma perspectiva da teoria de filas e através de uma perspectiva algorítmica.

Através da perspectiva da teoria de filas, o algoritmo é considerado instável se a taxa de chegada de trabalho ao sistema é maior do que a taxa que o sistema realiza trabalho. Um algoritmo pode ser estável e ainda assim causar uma degradação ao desempenho do sistema quando utilizado. Nesse caso, o algoritmo é considerado ineficaz (Shivaratri et al., 1992).

Através da perspectiva algorítmica, o algoritmo é considerado instável se existe a possibilidade desse produzir ações inúteis, indefinidamente. Por exemplo, ao realizar uma transferência de tarefas entre dois nós, o nó destinatário se torna sobrecarregado, exigindo uma nova transferência para um terceiro nó que também se torna sobrecarregado, continuando indefinidamente (Shivaratri et al., 1992).

2.4.4. Tipos de políticas de distribuição de carga

As políticas de distribuição de carga são classificadas em três tipos, de acordo com qual participante da transferência (remetente ou destinatário) é responsável por iniciar a distribuição de carga.

a. Iniciadas pelo remetente

Nessa classe de políticas, a distribuição de carga é iniciada pelo nó que está sobrecarregado e deseja remeter trabalho a outros nós menos carregados (Shivaratri et al., 1992; Milojić et al., 2000).

As políticas dessa categoria são adequadas apenas para sistemas com cargas baixas ou médias. Para sistemas muito carregados, a probabilidade de existir um nó com pouca carga, adequado para transferência, é muito baixa. Nesses casos, a política contribui para a instabilidade do sistema, devido ao aumento do número de mensagens (Shivaratri et al., 1992).

b. Iniciadas pelo destinatário

Nessa classe de políticas, a distribuição de carga é iniciada pelo nó que está com baixa carga e deseja receber trabalho de outros nós mais carregados (Shivaratri et al., 1992; Milojević et al., 2000).

As políticas dessa categoria são mais adequadas para sistemas sobrecarregados, pois a transferência só ocorre na presença de nós com baixa carga, e o algoritmo terá alta taxa de sucesso em encontrar um nó sobrecarregado. A migração de processos é fundamental para esse tipo de política, pois é capaz de realizar a transferência de um processo em qualquer momento de sua execução (Shivaratri et al., 1992; Milojević et al., 2000).

c. Simétricas

Através da combinação de políticas iniciadas pelo remetente e pelo destinatário, pode-se ter as vantagens de ambas as classes anteriores. Na classe de políticas simétricas, tanto remetentes quanto destinatários podem iniciar a distribuição de carga (Shivaratri et al., 1992; Milojević et al., 2000).

2.5. Considerações Finais

A área de computação paralela distribuída é o resultado da intersecção das áreas de computação paralela e sistemas distribuídos. A constante pesquisa nessa área reflete a contínua demanda por maior capacidade computacional, ao mesmo tempo em que sistemas computacionais se tornam cada vez mais onipresentes e globalmente interconectados.

Contudo, a utilização eficiente de recursos computacionais distribuídos para processamento paralelo é complexa, uma vez que os processadores estão fisicamente distantes uns dos outros, exigindo coordenação e transmissão de dados através de passagem de mensagens. A sobrecarga imposta pela comunicação força a atribuição de forma quase definitiva dos processos aos processadores, o que justifica a grande importância da atribuição inicial realizada pelo escalonador de processos com balanceamento de carga.

A migração de processos oferece uma possibilidade a mais para o balanceamento de carga, permitindo o escalonamento de processos preemptivo. No próximo capítulo, a migração de processos será introduzida e explicada.

3. Migração de Processos

3.1. Considerações Iniciais

Um processo é uma abstração utilizada pelo sistema operacional, que representa uma instância de um programa em execução. Um processo é formado pelo seu espaço de endereçamento (onde está contido o seu programa executável, os dados do programa e sua pilha), além do conjunto de registradores e demais informações requeridas pelo sistema operacional (Tanenbaum, 2002).

A migração de um processo é a transferência de um subconjunto significativo de seu estado entre dois computadores de um sistema distribuído, de forma a continuar a execução no computador de destino a partir do ponto onde se encontrava no computador de origem (Smith, 1988; Miloječić et al., 2000). Migração de processos é uma forma específica de mobilidade de código (Fuggetta et al., 1998).

Apesar de ser uma área com longo histórico de pesquisa, poucos são os sistemas existentes atualmente que empregam a migração de processos (Miloječić et al., 2000). Miloječić et al. (2000) citam como possíveis razões para essa resistência à adoção da migração de processos a complexidade de implementação em sistemas originalmente desenvolvidos sem esse recurso em mente e a falta de um argumento convincente para projetistas de sistemas operacionais para adicionar esse tipo de suporte. Contudo, essa área de pesquisa ainda atrai pesquisadores e existe produção recente de sistemas operacionais distribuídos que dão suporte à migração de processos de forma transparente (Lottiaux et al., 2005).

Neste capítulo será apresentada a migração de processos, incluindo conceitos fundamentais sobre mobilidade de código, motivação, políticas e mecanismos de migração de processos e sistemas que suportam migração de processos.

3.2. Mobilidade de Código

Carzaniga et al. (1997) definem mobilidade de código como “a capacidade de reconfigurar dinamicamente, em tempo de execução, a ligação entre os componentes de software da aplicação e seus lugares físicos dentro de uma rede de computadores”, isto é,

alterar dinamicamente a ligação entre o lugar onde fragmentos de código executável estão localizados e onde são executados.

Um ambiente computacional é formado por unidades de execução e recursos. Unidades de execução compreendem fluxos seqüenciais de computação, como processos e *threads*, enquanto recursos são entidades ou objetos compartilhados por diversas unidades de execução (Fuggetta et al., 1998). Unidades de execução são compostas por um segmento de código, espaço de dados e estado de execução (Fuggetta et al., 1998).

A mobilidade de código é classificada como mobilidade forte ou mobilidade fraca, de acordo com a porção da unidade de execução que é transferida. Na mobilidade forte, o sistema permite a migração tanto do código como do estado de execução de uma unidade de execução para um computador diferente, enquanto que na mobilidade fraca, o sistema permite a migração de código entre diferentes computadores, opcionalmente acompanhado de algum dado para iniciar, mas sem transferir o estado de execução (Fuggetta et al., 1998; Tanenbaum e Steen, 2002). A Tabela 3.1 mostra as diferenças entre mobilidade forte e fraca.

Tabela 3.1 - Classificação de mecanismos de mobilidade de código.

	Mobilidade forte	Mobilidade fraca
Segmento de código	Migra	Migra
Espaço de dados	Migra	Opcional, parcial
Estado de execução	Migra	Não migra

Fuggetta et al. (1998) dividem a mobilidade forte em migração e clonagem remota. A migração ocorre quando a unidade de execução é suspensa, migrada para outro computador, e então sua execução é continuada. Na clonagem remota, a unidade de execução é copiada para outro computador, sem removê-la de seu ambiente original. A mobilidade forte ainda pode ser proativa se o momento e o destino da migração são determinados de forma autônoma pela própria unidade de execução, ou reativa se a migração é iniciada por um processo externo.

A mobilidade fraca transfere o código executável entre computadores e, em seu destino, liga o código transferido a uma unidade de execução já existente (ligação dinâmica) ou cria uma nova unidade de execução para executar o código transferido (criação de um novo processo). Na mobilidade fraca, o código pode ser remetido (iniciado pelo remetente) ou buscado (iniciado pelo destinatário) entre computadores (Fuggetta et al., 1998; Tanenbaum e Steen, 2002). A Figura 3.1 sumariza a classificação de mecanismos de mobilidade de código.

Figura 3.1 - Classificação de mecanismos de mobilidade de código.

Neste trabalho, apenas a migração de processos (mobilidade forte), reativa (o escalonador de processos comanda a migração) será considerada.

3.2.1. Gerenciamento de ligações a recursos locais

Durante sua execução, processos são ligados a recursos locais, gerenciados pelo sistema operacional. Tais recursos compreendem arquivos de dados, soquetes de comunicação, bibliotecas de funções, dispositivos de entrada e saída, etc.

Quando um processo é migrado entre diferentes computadores, as ligações entre o processo e os recursos que este utiliza precisam ser tratadas durante o processo de migração, e restabelecidas no computador de destino.

Fuggetta et al. (1998) identificam três formas de ligação entre processos e recursos: ligação por identificador, por valor e por tipo.

A ligação por identificador é a mais forte e ocorre quando o processo acessa o recurso através de um identificador único e específico, e tal recurso não pode ser substituído por outro semelhante. Exemplos de ligação por identificador são soquetes de comunicação, arquivos de dados e serviços identificados por URLs (*Uniform Resource Locator*).

A ligação por valor ocorre quando o recurso utilizado pelo processo deve ser de um determinado tipo e seu valor não pode ser alterado. Esse tipo de ligação ocorre quando o processo está interessado no conteúdo do recurso, e pode ser substituído por outro recurso semelhante no computador de destino após a migração. Exemplos de ligação por valor são bibliotecas de sistema e o próprio programa executável do processo. Nesse caso, a identificação do recurso não é importante; uma biblioteca de sistema pode, por exemplo, estar presente em um lugar diferente do sistema de arquivos do computador de destino (sua

identificação é diferente), mas é do mesmo tipo e possui o mesmo conteúdo da biblioteca equivalente no computador de origem.

A ligação por tipo é a mais fraca, e ocorre quando o processo se refere ao recurso apenas pelo seu tipo, sem se importar com seu valor ou seu identificador. O recurso pode ser substituído por outro recurso do mesmo tipo no computador de destino, sem que isso afete o funcionamento do processo. Exemplos de ligações por tipo são geradores de números aleatórios e dispositivos locais do sistema, como monitor ou impressora (não significa necessariamente que o processo não possa ter uma ligação mais forte, como por identificador, para um dispositivo específico no sistema distribuído).

Fuggetta et al. (1998) e Tanenbaum e Steen (2002) também identificam três formas de ligação do recurso ao sistema onde estão presentes em soltos, amarrados e fixos⁷.

Recursos soltos podem ser facilmente movidos entre máquinas, junto com o próprio processo, durante sua migração. Exemplos de recursos soltos são arquivos de dados utilizados pelo processo sendo migrado.

Recursos amarrados podem ser movidos ou copiados entre máquinas, mas a custos relativamente altos e, portanto, sendo preferível a criação de uma referência para acesso remoto ao recurso no computador de destino. Exemplos de recursos amarrados são bancos de dados locais.

Recursos fixos não podem ser transferidos entre máquinas e são, geralmente, dispositivos físicos locais.

A Figura 3.2 resume os tipos de ligações entre processos, recursos e máquinas.

Figura 3.2 - Tipos de ligações entre processos, recursos e máquinas.

O tratamento para cada ligação entre processo e recurso durante a migração de processos varia conforme os tipos de ligações entre processo e recurso, e entre recurso e máquina. Quatro soluções são possíveis: remoção do recurso, cópia do recurso, uso de uma referência global ou religação a outro recurso de mesmo tipo. A Tabela 3.2 mostra quais tratamentos são possíveis para cada uma das combinações. Tanenbaum e Steen divergem de

⁷ Terminologia utilizada por Tanenbaum e Steen; Fuggetta et al. utilizam livres, fixos transferíveis e fixos não transferíveis respectivamente.

Fuggetta et al. em algumas situações; a tabela mostra uma visão comum, mas outros tratamentos podem ser possíveis em casos particulares.

Tabela 3.2 - Tratamento de ligações a recursos durante a migração de processos.

<i>Ligação entre processo e recurso</i>	<i>Ligação entre recurso e máquina</i>		
	<i>Solto</i>	<i>Amarrado</i>	<i>Fixo</i>
<i>Por identificador</i>	Remoção (ou Referência)	Referência	Referência
<i>Por valor</i>	Cópia (ou Remoção, Ref.)	Referência (ou Cópia)	Referência
<i>Por tipo</i>	Religação (ou Remoção, Cópia)	Religação (ou Ref., Cópia)	Religação (ou Referência)

A remoção do recurso é possível apenas quando este é solto em relação à máquina, sua localização no ambiente distribuído (pelo menos durante a execução da aplicação) não possui relevância para o seu funcionamento e a sua utilização.

A cópia do recurso consiste em criar um recurso de mesmo tipo em outra máquina, com o mesmo valor do recurso original, mas possivelmente com um identificador diferente. A cópia é útil quando o recurso em si não pode ser movido do seu computador original, mas seu valor não possui qualquer dependência local, podendo ser replicado em outro computador.

A referência global é uma possível solução que pode ser utilizada em qualquer situação e consiste em criar um identificador universal, como uma URL, que permite que o processo acesse aquele recurso independente de em qual máquina é executado. A referência global é mais relevante para recursos fixos e amarrados.

Por fim, a religação consiste em encontrar no computador de destino um recurso de mesmo tipo (com a mesma funcionalidade) que o recurso no computador de origem, e recriar a ligação entre processo e o novo recurso. A religação é relevante apenas para a ligação entre processo e recurso por tipo.

3.3. Motivação da Migração de Processos

No contexto de sistemas computacionais distribuídos, a migração de processos é necessária quando o sistema se encontra em um estado onde a distribuição das tarefas entre os nós é ineficiente ou indesejada, em geral prejudicando o desempenho do sistema. A migração de processos permite modificar a distribuição das tarefas entre os nós, a fim de atingir um certo objetivo. Pode-se listar quatro objetivos comuns para a migração de processos: balanceamento dinâmico de carga, aproximação dos processos aos recursos que acessam, tolerância a falhas e administração do sistema.

3.3.1. Balanceamento dinâmico de carga

O balanceamento de carga é, provavelmente, o objetivo principal da migração de processos e também onde há maior volume de pesquisa. Mesmo com a utilização de políticas de escalonamento capazes de realizar o balanceamento de carga no momento da alocação inicial, com o tempo, o sistema pode atingir um estado de desequilíbrio na distribuição de carga entre os computadores. Nesse caso, processos dos computadores mais carregados são migrados para os computadores menos carregados, ajudando a equalizar a carga de trabalho do ambiente paralelo distribuído (Eskicioğlu, 1995; Richmond e Hitchens, 1997; Milojičić et al., 2000).

O balanceamento dinâmico de carga pode ser implementado através de políticas de escalonamento preemptivas, sem a necessidade de se conhecer a aplicação. A política pode simplesmente observar o estado do sistema e o comportamento do processo, e tomar a decisão baseado nessas métricas. Esse é o objetivo abordado neste trabalho.

3.3.2. Aproximação dos processos aos recursos que acessam

O segundo objetivo da migração de processos é permitir que processos migrem para computadores mais próximos dos dados ou recursos que utilizam. Isso é útil em sistemas distribuídos heterogêneos, onde alguns computadores possuem recursos físicos que não podem ser movidos (por exemplo: memória, terminal), ou cujo custo de acesso ou transferência do recurso pela rede seria maior do que a migração do processo (por exemplo: um grande banco de dados) (Eskicioğlu, 1995; Richmond e Hitchens, 1997; Milojičić et al., 2000).

Dessa forma, o tráfego de rede é reduzido, pois o processo é transferido para um computador próximo ao recurso que se deseja acessar e, após o acesso e processamento necessários, o processo pode ser movido de volta ao seu computador de origem.

Em geral, essa motivação implica na utilização de agentes móveis e migração proativa, onde o próprio processo sabe onde encontrar o recurso necessário e comanda sua própria migração para o computador onde pode acessá-lo mais facilmente. Um exemplo de tal aplicação são os D'Agents (Gray et al., 2002).

É possível também a aplicação desse objetivo através de migração reativa, envolvendo um escalonador ou gerenciador externo à aplicação. Nesse caso, ao processo é atribuído um espaço de endereçamento que inclui dados não-locais ao computador onde é executado. Quando o processo acessa regiões de memória que correspondem a dados remotos,

o sistema operacional causa a migração do processo espontaneamente (Noguchi et al., 2008).

3.3.3. Tolerância a falhas

Às vezes, a falha parcial de um dos computadores que compõe um sistema distribuído pode ser detectada, permitindo que uma ação seja tomada antes da falha total (Milojičić et al., 2000).

Quando uma falha parcial é detectada e o sistema ainda está em condições de realizar migração, os processos ativos podem ser migrados a outros computadores e, então, o computador defeituoso é retirado do conjunto de nós ativos do ambiente paralelo distribuído.

Uma falha imediata do sistema também pode ser tolerada com a utilização da migração de processos, através da técnica de *checkpointing*. Consiste basicamente de uma migração parcial do processo, realizada periodicamente durante sua execução. O estado do processo é gravado em um arquivo e enviado para um servidor de arquivos, sem interromper a execução do processo. Ao detectar uma falha de um computador, outro computador do sistema distribuído recupera o último estado salvo e reinicia a execução do processo.

A aplicação de migração de processos como forma de tolerância a falhas é abordada em maiores detalhes por Sankaran et al. (2005) e Wang et al. (2008).

3.3.4. Administração do sistema

A migração de processos também permite uma administração do sistema mais robusta, pois computadores podem ser desligados ou reiniciados para manutenção sem a perda dos processos em execução (Eskicioğlu, 1995; Richmond e Hitchens, 1997; Milojičić et al., 2000).

Estações de trabalho ociosas podem ser incorporadas em tempo de execução ao conjunto de computadores de um sistema distribuído, aumentando a potência de processamento disponível. Quando o usuário retorna à sua estação de trabalho, os processos são desalojados (*evicted*), sendo remetidos de volta aos seus nós originais, devolvendo o processamento da estação de trabalho ao seu usuário original (Douglass e Ousterhout, 1991).

3.4. Políticas de Migração de Processos

Os dois elementos que determinam a forma como a migração de processos é realizada são a sua política e o seu mecanismo. Em geral, esses algoritmos são implementados de forma independente e em níveis distintos. O mecanismo de migração de processos é

implementado em um nível inferior da plataforma, às vezes integrado ao núcleo do sistema operacional, enquanto que a política de migração é implementada em um nível superior, como um processo de usuário. Contudo, há casos, como o *Sprite* (Douglis e Ousterhout, 1991) e o *V Distributed System* (Theimer et al., 1985), em que a política e o mecanismo são implementados de forma intercalada em um único algoritmo, com a finalidade de simplificar a implementação (Eskicioğlu, 1995).

Uma política de migração de processos é basicamente uma política de distribuição de carga, com os mesmos componentes: políticas de transferência, de seleção, de localização e de informação. No entanto, Miloječić et al. (2000) divergem de Shivaratri et al. (1992) ao redefinir o escalonamento distribuído com a finalidade de tornar o escalonamento mais intimamente relacionado à migração de processos.

Para Miloječić et al. (2000), a tarefa de gerenciar as informações sobre a carga do sistema é delegada a um módulo à parte, separado da política de migração. A política de informação, a mesma definida por Shivaratri et al. (1992) como parte da política de distribuição de carga é, então, implementada nesse módulo. Dessa forma, esse módulo não participa diretamente do escalonamento dos processos, mas continua responsável por disponibilizar a informação necessária para a política de migração quando esta é solicitada.

Assim, a política de migração de processos é reduzida a apenas três componentes, que Miloječić et al. definem como política de ativação, política de seleção e política de localização. Apesar da nomenclatura diferente, a política de ativação de Miloječić et al. tem, basicamente, a mesma função da política de transferência de Shivaratri et al..

3.5. Mecanismos de Migração de Processos

Assim que é tomada a decisão de que a migração de processos deve ocorrer e já se sabe quais serão os participantes (determinados pela política de migração), o mecanismo de migração é ativado para realizar a transferência em si.

O mecanismo de migração (às vezes chamado de algoritmo de migração) é responsável por suspender a execução do processo no computador de origem, transferir o estado do processo para o computador de destino e reiniciar a execução no computador de destino. Essas tarefas básicas são divididas em um certo número de passos, cuja forma e a ordem em que são executados variam para cada mecanismo de migração.

3.5.1. Procedimento de migração de um processo

Apesar das diferenças entre os mecanismos existentes, é possível descrever o procedimento de migração de um processo através de um conjunto de tarefas comum à maioria dos algoritmos. Richmond e Hitchens (1997) e Milojević et al. (2000) listam as seguintes tarefas:

- Ocorre a decisão por realizar uma migração de processos entre dois nós;
- a execução do processo no nó de origem é suspensa;
- as comunicações com o processo são redirecionadas para o nó de destino;
- o estado do processo é transmitido para o nó de destino;
- o estado do processo é reconstruído no nó de destino;
- referências ao processo no nó de origem são modificadas para o nó de destino;
- mensagens enfileiradas no nó de origem são retransmitidas para o nó de destino;
- a execução do processo no nó de destino é reiniciada;
- as informações sobre o processo no nó de origem são descartadas.

Exceto pela decisão de migração, as outras tarefas não precisam necessariamente ser executadas na ordem em que aparecem. Algumas dessas tarefas podem ser executadas concorrentemente, dependendo do projeto de cada mecanismo. A comunicação do processo durante a migração só é tratada quando o mecanismo está integrado a um ambiente de passagem de mensagens.

Dependendo do algoritmo utilizado, a migração pode ser considerada completa assim que as informações sobre o processo no nó de origem são descartadas ou assim que a execução no nó de destino é reiniciada (mesmo que ainda existam referências ao processo no nó de origem).

3.5.2. Projeto de mecanismos de migração de processos

Entre as tarefas executadas durante o procedimento de migração de um processo, três propriedades possuem maior relevância ao diferenciar algoritmos de migração (Richmond e Hitchens, 1997):

- A quantidade do estado do processo que deve ser transferido para o nó de destino;
- o momento em que a execução do processo no nó de origem é suspensa;

- o momento em que a execução do processo no nó de destino é reiniciada.

A transferência do estado do processo durante a migração pode ser completa, ou apenas da parte necessária para iniciar a execução no nó de destino. No primeiro caso, é consumido um maior tempo durante a migração, porém evita o acúmulo de referências residuais ao processo no nó de origem. No segundo caso, o nó de origem mantém parte do estado do processo armazenado localmente até que o processo seja encerrado no nó de destino, ou toda referência residual seja transmitida.

A execução do processo no nó de origem pode ser suspensa logo que é tomada a decisão de migrar o processo, quando uma parte essencial do processo já foi transferida ao nó de destino ou somente depois que todo o estado do processo for transferido. Nos dois últimos casos, onde o processo continua em execução durante a transferência, uma nova transferência é necessária após a suspensão do processo para reenviar as partes do espaço de endereçamento que foram alteradas durante a primeira transferência.

A execução do processo no nó de destino pode ser reiniciada logo que uma parte essencial do processo for transferida ou depois que todo o estado do processo for transferido. No primeiro caso, o processo reinicia a execução enquanto que parte do estado do processo ainda se encontra no nó de origem e, portanto, quando uma região de memória ainda não transferida é acessada, o processo é suspenso e a região de memória é transferida a partir do nó de origem.

3.5.3. Algoritmos de migração de processos

a. Eager copy

O algoritmo *eager copy* é o mais trivial e simples de se implementar. Nesse algoritmo, o processo é interrompido no nó de origem, todo o estado é transferido para o nó de destino e, então, a execução é reiniciada no nó de destino. A Figura 3.3 ilustra a migração de um processo através do algoritmo *eager copy*.

As vantagens do algoritmo *eager copy* são a sua simplicidade de implementação e a remoção de referências residuais do processo no nó de origem. A desvantagem desse algoritmo é a suspensão completa do processo durante o longo tempo de transferência.

A implementação desse algoritmo de migração pode ser realizada através da técnica de *checkpointing*. Condor (Litzkow e Solomon, 1992), LSF (Zhou et al., 1994), MPVM (Casas et al., 1995), DPVM (Iskra et al., 2000a; Furquim, 2006) e LAM/MPI (Sankaran et al., 2005) utilizam esse algoritmo.

Figura 3.3 - Algoritmo *eager copy*.

Uma variação desse algoritmo consiste em transmitir ao nó de destino apenas as páginas de memória modificadas, deixando que as páginas não modificadas sejam recuperadas sob demanda, através de paginação remota (Milojičić et al., 2000). MOSIX (Barak et al., 2005) utiliza essa variação.

b. Lazy copy

O algoritmo *lazy copy*, também conhecido como *copy-on-reference*, consiste em transferir apenas uma parte essencial do estado do processo (todo o seu estado interno, exceto a memória virtual) para reiniciar a sua execução no nó de destino. O processo continua armazenado e em suspensão no nó de origem, enquanto que o nó de destino solicita a transferência das páginas de memória necessárias sob demanda, durante a execução. A Figura 3.4 ilustra a migração de um processo através do algoritmo *lazy copy*.

O algoritmo *lazy copy* permite uma transferência inicial muito mais rápida do que o *eager copy*, mas exige que o nó de origem mantenha referências sobre o processo migrado até a sua conclusão no nó de destino. Faltas de página, para páginas ainda não transferidas para o nó de destino, causam a suspensão do processo e a transmissão das páginas necessárias a partir do nó de origem.

As vantagens desse algoritmo são o reinício mais rápido da execução do processo após a transferência inicial e a possível redução do tráfego de rede caso páginas de memória não forem mais utilizadas até o final da execução do processo. As desvantagens são o aumento do tempo de execução a cada página solicitada (podendo até resultar em um desempenho pior que o *eager copy* se todas as páginas tiverem que ser transmitidas) e a dependência que deve ser mantida com o nó de origem, o que torna o algoritmo ruim para

balanceamento de carga e tolerância a falhas (Richmond e Hitchens, 1997; Milojić et al., 2000).

Figura 3.4 - Algoritmo lazy copy.

c. Pre-copy

O algoritmo *pre-copy* caracteriza-se por não suspender a execução do processo no nó de origem durante a transferência, o estado do processo é transferido concorrentemente com a execução do processo. Ao término da transferência, o processo é suspenso e as páginas de memória modificadas durante a primeira transferência são retransmitidas. A Figura 3.5 ilustra a migração de processos através do algoritmo *pre-copy*.

Figura 3.5 - Algoritmo pre-copy.

O algoritmo *pre-copy* presume que, entre o tempo do início da migração até o tempo do final da transferência inicial, apenas uma pequena região da memória será alterada pelo processo e apenas essa região precisará ser retransmitida. Com isso, o tempo durante o qual o

processo fica suspenso é menor do que o do algoritmo *eager copy*. No pior caso (todas as páginas de memória são alteradas durante a transferência inicial), o tempo de suspensão do processo será o mesmo do algoritmo *eager copy*, no entanto, o tráfego de rede será o dobro uma vez que todo o estado será transmitido duas vezes (Richmond e Hitchens, 1997; Milojić et al., 2000).

d. *Post-copy*

O algoritmo *post-copy*, proposto por Richmond e Hitchens (1997), consiste em realizar a transferência do mínimo necessário para reiniciar a execução do processo no nó destino, como no algoritmo *lazy-copy*, mas continuar transmitindo o resto do estado do processo durante a sua execução no nó destino, de forma simétrica ao algoritmo *pre-copy*. A Figura 3.6 ilustra a migração de processos através do algoritmo *post-copy*.

Figura 3.6 - Algoritmo *post-copy* (Richmond e Hitchens, 1997).

Após a transferência inicial, o processo é reiniciado no nó destino, ao mesmo tempo em que a transferência do restante do estado do processo ocorre, de forma transparente à execução do processo no nó de destino. Ao final da transferência, o nó de origem pode descartar os dados armazenados do processo.

Se, durante o segundo estágio da transferência quando o processo já está executando no nó de destino, ocorrer uma falta de página, o processo é suspenso e a página é solicitada ao nó de origem, que deve interromper temporariamente a transferência concorrente e transferir a página necessária. Ao retomar a transferência concorrente, a página já enviada em decorrência da falta de página não deve ser reenviada. Efetivamente, a cada falta de página, a ordem da transmissão das páginas de memória é alterada a fim de transmitir a página de mais alta

prioridade antes do resto do estado do processo.

e. *Flushing*

O algoritmo *flushing* (Douglass e Ousterhout, 1991) possui uma característica diferenciada em relação aos outros algoritmos, por fazer uso de um terceiro componente que participa indiretamente da migração de processos. A implementação desse algoritmo de migração depende de que a memória virtual do sistema operacional seja implementada utilizando-se arquivos comuns e, portanto, que podem ser armazenados em um servidor de arquivos.

Esse algoritmo mistura características dos algoritmos *eager copy* e *lazy copy*. Nesse algoritmo, o nó de origem transmite as páginas de memória alteradas (“sujas”) para o servidor de arquivos e transmite o estado interno do processo diretamente para o nó de destino. O nó de destino então recupera as páginas de memória necessárias a partir do servidor de arquivos e reinicia a execução do processo. A Figura 3.7 ilustra a migração de processos através do algoritmo *flushing*.

Figura 3.7 - Algoritmo *flushing* (Douglass e Ousterhout, 1991).

O algoritmo *flushing* possui as vantagens de não manter referências residuais no nó de origem após a migração (característica do algoritmo *eager copy*), evitar tráfego excessivo na rede e ter um curto tempo entre o início da migração e o reinício da execução no nó de

destino (características do algoritmo *lazy copy*). Ele possui como desvantagem a necessidade de uma implementação específica de memória virtual do sistema operacional.

O sistema operacional Sprite (Douglass e Ousterhout, 1991) implementa esse mecanismo de migração de processos, a fim de utilizar computadores ociosos em uma rede de estações de trabalho, de forma que o usuário local do computador possui prioridade sobre o uso do mesmo. Quando o usuário local volta a utilizar o seu computador, processos migrados de outros computadores são automaticamente desalojados (*evicted*) e migrados de volta ao computador de origem, ou a outro computador ocioso da rede.

3.5.4. Classificação quanto ao paradigma de programação paralela

A classificação apresentada na seção anterior separa os algoritmos de acordo com a ordem que as tarefas do mecanismo de migração de processos são realizadas, e como ocorre a transferência do estado do processo do nó de origem para o nó de destino.

Uma segunda classificação pode ser aplicada em relação ao conjunto de processos que participam do processo de migração:

a. Migração de Processos MPMD

Nessa categoria, nenhuma restrição é imposta sobre o conjunto de processos participando do ambiente paralelo distribuído. Como o nome sugere, essa categoria de algoritmos é voltada a ambientes construídos sobre o paradigma de programação MPMD (*Multiple Program, Multiple Data*). Quando um processo é migrado, toda a imagem em memória do processo em execução, incluindo segmentos de programa e de dados, precisa ser transferida para o nó de destino, onde o processo é reconstruído e reiniciado.

b. Migração de Processos SPMD

Nessa categoria, todos os processos participantes do conjunto de migração são instanciados de um único programa, previamente conhecido por todos os nós do ambiente paralelo distribuído. Como o nome sugere, essa categoria de algoritmos é voltada a ambientes construídos sobre o paradigma de programação SPMD (*Single Program, Multiple Data*). Quando um processo é migrado, apenas o segmento de dados e o estado interno do processo precisam ser transferidos para o nó de destino, onde o processo é reconstruído utilizando-se um segmento de programa já localmente disponível (de outro processo em execução ou de um arquivo de programa executável). Algoritmos implementados sobre esse modelo apresentam oportunidades de otimização do desempenho da migração de processos através da redução da

quantidade de dados que precisam ser transmitidos (Furquim, 2006).

Uma terceira categoria pode ser considerada, onde o algoritmo determina se o segmento de programa já está presente no nó de destino e, condicionalmente, o transfere se necessário (Douglass e Ousterhout, 1991; Barak et al., 2005). Essas implementações podem ser classificadas como MPMD no sentido de que nenhuma restrição é imposta sobre o conjunto de processos do ambiente paralelo distribuído, mas também são otimizadas para atuar como SPMD quando possível.

Note que esta classificação de algoritmos de migração de processos não substitui ou sobrepõe a classificação anterior. Quaisquer dos cinco algoritmos de migração de processos apresentados anteriormente podem ter implementações MPMD, SPMD ou híbridas.

3.6. Implementação do Suporte à Migração de Processos

O mecanismo de migração de processos pode ser implementado tanto em nível de núcleo do sistema (*kernel space*), quanto em nível de processo do usuário (*user space*). Implementações em nível de usuário, por sua vez, também podem ser realizadas de duas formas diferentes: através da ligação de uma biblioteca de sistema ao programa executável, ou através da implementação na própria aplicação (Paindaveine e Milošević, 1996; Milošević et al., 2000; Jankowski et al., 2005; Furquim, 2006).

Cada uma das três alternativas possui suas vantagens e desvantagens em relação à complexidade de implementação, transparência para a aplicação, independência quanto à linguagem de programação, independência quanto ao código fonte, independência quanto ao sistema operacional e independência quanto à arquitetura. O Tabela 3.3 lista as características desejáveis e sua relação com cada uma das implementações possíveis.

Tabela 3.3 - Níveis de implementação de migração de processos.

Características	Nível de sistema	Nível de usuário	
		Biblioteca	Aplicação
Complexidade de implementação	alta	média	baixa
Transparência para a aplicação	sim	restrita	não
Independência de linguagem de programação	sim	não	não
Independência do código fonte	sim	talvez	não
Independência do sistema operacional	não	não	sim
Independência da arquitetura	não	não	sim

A complexidade de implementação é considerada com relação à implementação do mecanismo de migração em si, e não da implementação de um programa que utiliza a migração de processos. A transparência para a aplicação refere-se ao quão transparente ocorre

a migração de processos do ponto-de-vista da aplicação. Uma implementação é considerada transparente se a aplicação não precisa ser adaptada ao ambiente, não sofre restrições quanto ao uso de recursos do sistema e não sofre alterações de execução que a permitam detectar a ocorrência da migração. A independência de linguagem de programação ocorre quando qualquer programa, escrito em qualquer linguagem de programação, pode se beneficiar do mecanismo de migração de processos. A independência de código fonte ocorre quando o programa pode utilizar a migração de processos sem a necessidade de adaptação de seu código fonte. A independência de sistema operacional ocorre quando a migração pode ser suportada naturalmente entre sistemas operacionais diferentes. A independência de arquitetura ocorre quando a migração pode ser suportada naturalmente entre arquiteturas de *hardware* diferentes (Paindaveine e Milojević, 1996; Milojević et al., 2000; Jankowski et al., 2005).

3.6.1. Nível de sistema

A implementação do suporte à migração de processos no nível do núcleo do sistema operacional é a de mais alta complexidade. O tratamento de detalhes como o estado de múltiplas *threads* de execução, soquetes de comunicação e referências a descritores de arquivos abertos adicionam complexidade ao núcleo do sistema, dificultando o gerenciamento. (Paindaveine e Milojević, 1996; Milojević et al., 2000; Roman, 2002)

Contudo, a implementação em nível de sistema tem um grande potencial de fornecer transparência completa para a aplicação, a ponto do processo não sofrer qualquer alteração de ambiente que o permita detectar a ocorrência de migração. A migração em baixo nível permite que aplicações possam ser migradas sem a necessidade de alterações ou recompilação de seu código fonte. Como o mecanismo é implementado no sistema operacional, a migração é limitada a ocorrer apenas entre nós que utilizem o mesmo sistema operacional e a mesma arquitetura.

Políticas de migração implementadas nesse nível podem apenas observar o comportamento do sistema (através dos índices de carga) e o comportamento do processo (como comunicação ou faltas de página). Isso pode ocasionar algumas decisões ruins de migração como, por exemplo, migrar um processo que está prestes a terminar sua execução. Em algumas implementações, é possível implementar a política de migração em um nível superior, dentro do programa, se comunicando com o mecanismo de migração através de chamadas de sistema (Douglass e Ousterhout, 1991). Dessa forma, uma solução híbrida permite obter o melhor das duas abordagens, com a desvantagem de não ser mais transparente à aplicação.

Qualquer um dos algoritmos de migração pode ser implementado nesse nível. Apenas o nível de sistema permite a paginação remota, portanto, os algoritmos baseados em paginação remota (*lazy copy*, *post-copy* e *flushing*) são exclusivos desse nível.

3.6.2. Biblioteca de sistema (nível de usuário)

A implementação do suporte à migração de processos através de bibliotecas de sistema é uma alternativa à implementação em nível de sistema, que procura fornecer certo nível de transparência para a aplicação e independência de código fonte, com menor complexidade (Paindaveine e Milojičić, 1996; Milojičić *et al.*, 2000; Roman, 2002).

Algumas vezes, a biblioteca depende de programas externos às aplicações sendo migradas para complementar o suporte à migração. A biblioteca é ligada ao programa executável da aplicação em tempo de compilação, gerando, portanto, um executável próprio preparado para a migração de processos.

Comum às soluções em nível de usuário, existe a dificuldade de se obter todo o estado do processo de dentro do próprio processo. Algumas estruturas do estado do processo só podem ser acessadas em nível do sistema, como por exemplo, o estado particular de cada *thread* de um processo com múltiplas *threads*. Também, algumas referências do processo simplesmente não podem ser migradas sem o apoio do sistema operacional, como soquetes de comunicação. Isso impõe uma série de restrições ao processo, que impedem que a migração seja completamente transparente.

Geralmente, a biblioteca também é restrita a programas de uma única linguagem de programação, ou uma única família de linguagens de programação. A biblioteca pode ou não depender de alterações no código da aplicação.

A solução através de biblioteca de sistema geralmente envolve alguma forma de *checkpointing*, de modo a extrair todo o conteúdo do espaço de endereçamento de memória da aplicação, que será enviada ao nó de destino. Isso também torna a solução dependente de sistema operacional e de arquitetura.

As políticas de escalonamento implementadas nesse nível possuem maior dificuldade para observar o comportamento do sistema e também da aplicação. A biblioteca pode permitir que a própria aplicação ou agentes externos (através de sinais assíncronos) controlem a migração de processos, de forma a contornar essa limitação.

Nesse nível, apenas os algoritmos de migração que não dependem de paginação remota podem ser implementados: *eager copy* e *pre-copy*.

3.6.3. Aplicação (nível de usuário)

Por fim, o suporte à migração de processos também pode ser implementado como parte das funções da própria aplicação. A complexidade de implementação é considerada baixa, pois a própria aplicação seleciona e exporta o conjunto mínimo de informações necessárias para reiniciar o processamento, sem a necessidade de intervir em funções do sistema ou capturar estruturas de dados internas ao sistema operacional. Contudo, a complexidade pode ser considerada alta uma vez que a reusabilidade de código é baixa, e cada aplicação tem que implementar suas próprias rotinas de migração ou exportação e importação de estado. Muitas vezes o algoritmo precisa ser completamente reescrito para dar suporte ao reinício do processamento em um ponto arbitrário. (Paindaveine e Milojičić, 1996; Milojičić et al., 2000; Roman, 2002)

Da mesma forma que a implementação através de biblioteca de sistema, a migração pela aplicação impõe restrições ao processo, mas também remove outras. Por exemplo, a utilização de arquivos de dados em disco é algo mais fácil de se tratar através da própria aplicação, uma vez que a própria rotina de migração pode ser programada para transferir ou acessar o arquivo remotamente à partir do nó de destino, ao término da migração.

A princípio, o suporte à migração de processos pela aplicação é desvantajoso devido a não transparência do procedimento à aplicação e à baixa reusabilidade. Contudo, a migração pela aplicação oferece uma grande vantagem em relação à portabilidade. Como a própria aplicação conhece os dados sobre os quais processa, a rotina de migração pode converter e exportar os dados em um formato neutro à arquitetura, como por exemplo, XML (*eXtensible Markup Language*) ou texto-puro. Isso permite que a migração seja realizada entre nós de diferentes arquiteturas e com diferentes sistemas operacionais.

3.7. Checkpointing

O termo *checkpoint* pode ser traduzido como “ponto de controle” ou “ponto de marcação”. Na área de computação, *application checkpointing* (ou simplesmente *checkpointing*) designa a técnica ou mecanismo, que permite que uma quantidade suficiente do estado de uma aplicação em determinado momento de sua execução, seja salvo em um dispositivo de armazenamento não-volátil a fim de reiniciar a execução em um momento futuro, a partir do ponto onde foi salvo (Elnozahy et al., 2002; Roman, 2002; Jankowski et al., 2005).

A principal motivação para o *checkpointing* é a tolerância a falhas. Em aplicações

que exigem um longo tempo de execução, a técnica de *checkpointing* pode ser utilizada a fim de salvar o estado da aplicação em períodos regulares de tempo, de forma que, ocorrendo uma falha de sistema que invalide a execução do processo, a aplicação pode ser reiniciada a partir do ponto salvo após a normalização da situação, evitando a perda do processamento já realizado. O *checkpointing* também pode ser utilizado para diversas outras finalidades, como *rollback* de processos, depuração de programas e migração de processos.

Como ferramenta de suporte à migração de processos, o *checkpointing* permite a implementação do algoritmo *eager copy*. A execução do processo a ser migrado é suspensa e, através do *checkpointing*, o estado do processo é salvo em um arquivo local. Esse arquivo é transmitido através da rede a um outro nó do sistema (possivelmente por intermédio de um servidor de arquivos), o processo é reconstruído e sua execução é reiniciada no nó de destino com a ajuda do arquivo gerado.

Da mesma forma que a migração de processos, o *checkpointing* pode ser implementado nos mesmos três níveis distintos: nível de sistema, biblioteca de sistema e aplicação.

3.8. Ambientes com Suporte à Migração de Processos

Existem diversos ambientes com suporte à migração de processos. Alguns ambientes apenas provêm a migração de processos entre computadores, enquanto outros permitem algum nível de integração com o ambiente de passagem de mensagens (PVM ou MPI). Vários ambientes estão obsoletos ou foram descontinuados. Esta seção listará algumas das ferramentas e ambientes com suporte à migração de processos atuais.

3.8.1. *Single System Image (SSI)*

Single System Image (SSI), traduzido como “Imagem de Sistema Única” é um mecanismo de *middleware* ou *underware*, que permite que um conjunto de computadores interligados seja visto como um único grande computador (Walker e Steel, 1999; Lottiaux et al., 2005).

Um sistema operacional SSI opera de forma a convergir a utilização dos recursos computacionais de todos os computadores do sistema distribuído, criando a visão de um computador multiprocessador com memória distribuída. Cada computador se torna um terminal para um grande computador virtual formado por todos os computadores.

Ao unificar a imagem de sistema de vários computadores em uma única, a migração

de processos surge naturalmente. Todas as chamadas de sistema executadas pelo processo são realizadas sobre essa imagem única do sistema como um todo e, portanto, a relevância sobre qual processador executa o processo diminui. Nesses sistemas, a migração de processos ocorre de forma completamente transparente.

Entre os sistemas SSI mais importantes mantidos atualmente estão o MOSIX (Barak et al., 2005), Kerrighed (Morin et al., 2003; 2004), OpenSSI (Walker, 2006) e BProc (Hendriks, 2002).

3.8.2. *Checkpoint/Restart*

A forma mais comum de dar suporte à migração de processos nos ambientes atuais é através da utilização de *checkpointing*, em geral implementada em nível de usuário. Apesar das limitações impostas pela técnica de *checkpointing* quando utilizada para migração de processos, essa solução é atrativa pela sua simplicidade de implementação e viabilidade de integração com o ambiente de passagem de mensagens.

Nesses ambientes, a ferramenta responsável pelo *checkpointing* em geral é um componente distinto do ambiente como um todo. As ferramentas mais importantes mantidas atualmente são VMADump (Hendriks, 2002), dynckpt (Iskra et al., 2000b), DMTCP (Ansel et al., 2008) e BLCR (Hargrove e Duell, 2006).

Os ambientes paralelos distribuídos e de passagem de mensagens mais importantes que suportam migração de processos são Condor (Thain et al., 2005), LSF (Platform, 2006), DPVM (Iskra et al., 2000a) e LAM/MPI (Sankaran et al., 2005).

Para o desenvolvimento deste trabalho, o ambiente DPVM (Iskra et al., 2000a; 2000b) foi escolhido. É um ambiente derivado do PVM (Geist et al., 1994), com suporte à migração de processos através da ferramenta dynckpt. Esse ambiente possui diversas limitações (Furquim, 2006), mas foi escolhido devido à falta de alternativas que atendessem a todos os requisitos necessários ao trabalho, especialmente implementação aberta e possibilidade de integração com o ambiente de escalonamento AMIGO. Dessa forma, também foi dada continuidade ao trabalho de Furquim, que foi baseado na mesma ferramenta.

A ferramenta DMTCP (Ansel et al., 2008) surgiu durante o período em que este trabalho estava sendo desenvolvido, assim como a ferramenta BLCR (Hargrove e Duell, 2006) sofreu grandes aperfeiçoamentos no mesmo período. Possivelmente, essas duas ferramentas são as que melhor atendem aos requisitos que a ferramenta dynckpt falha em cumprir. Infelizmente, o surgimento e amadurecimento dessas duas ferramentas não

ocorreram em tempo hábil para serem consideradas neste trabalho.

3.9. Considerações Finais

A migração de processos representa um vasto campo de pesquisa dentro da área de computação paralela distribuída. Neste capítulo foi apresentada uma extensa revisão bibliográfica sobre o assunto, desde seus conceitos fundamentais até as implementações atuais das ferramentas que oferecem suporte à migração de processos.

O grande interesse por essa área de pesquisa se deve pelas oportunidades de balanceamento de carga, tolerância a falhas e flexibilidade que a migração de processos oferece, que não são possíveis apenas com a utilização da alocação inicial.

Contudo, a aceitação da migração de processos fora do ambiente acadêmico ainda é muito restrita. Milošević et al. (2000) apontam como possíveis razões para a resistência de aceitação da migração de processos a falta de conhecimento por parte dos usuários e a dificuldade de adaptação de sistemas atuais para a sua utilização. A integração do suporte à migração de processos ao ambiente de escalonamento AMIGO (Souza, 2000) é um passo importante em direção a permitir que um maior número de aplicações usufrua desse mecanismo.

No próximo capítulo, o ambiente de escalonamento AMIGO é apresentado e analisado, e as alterações necessárias para a implementação do suporte à migração de processos no ambiente são descritas.

4. O Ambiente de Escalonamento AMIGO

4.1. Considerações Iniciais

O AMIGO (*dynAMical flexIble schedulinG enviroNment*) é uma ferramenta de *software* aberta, desenvolvida pelo grupo de Sistemas Distribuídos e Programação Concorrente (SDPC) do Instituto de Ciências Matemáticas e Computação (ICMC) durante a tese de doutorado de Souza (2000) e complementada por diversos outros trabalhos de pesquisa (Araújo, 1999; Figueiredo, 2000; Santos, 2001; Campos Jr., 2001; Voorsluys, 2006), orientados dentro do mesmo grupo de pesquisa.

O AMIGO é um ambiente flexível e dinâmico responsável pelo gerenciamento do escalonamento de processos em sistemas distribuídos, oferecendo uma interface através da qual ambientes de passagem de mensagem ou aplicações distribuídas podem solicitar serviços de escalonamento.

A plataforma visa a reduzir a distância entre teoria e prática no projeto, desenvolvimento e aplicação de políticas de escalonamento de processos. Através do AMIGO, diferentes políticas de escalonamento podem ser implementadas e executadas independentemente, cada uma servindo a um diferente conjunto de aplicações distribuídas, que pode ser configurado pelo usuário, pelo administrador do sistema ou determinado automaticamente através de heurísticas.

O aspecto flexível do ambiente se refere a possibilitar a adaptação e extensão dos diversos fatores que influenciam o escalonamento de processos. Para isso, a ferramenta é completamente aberta e provê interfaces claras para futuras implementações. O aspecto dinâmico do ambiente se refere a permitir a alteração das políticas de escalonamento ativas em tempo de execução. O AMIGO não é simplesmente um algoritmo de escalonamento adaptável, mas, atua em um nível superior, gerenciando o processo de escalonamento como um todo, estando um nível acima das políticas de escalonamento.

No decorrer deste trabalho, tanto a especificação, quanto a implementação originais do AMIGO precisaram ser revistas, adaptadas e aperfeiçoadas visando à implementação do suporte à migração de processos. Neste capítulo, o projeto original é resumidamente

apresentado e analisado, e as alterações no projeto e na implementação do ambiente são descritas.

4.2. Projeto Original

Esta seção descreve de forma sucinta os pontos mais relevantes da especificação original do ambiente de escalonamento AMIGO, conforme apresentado na tese de doutorado de Souza (2000), capítulo 3.

O ambiente é dividido em duas camadas: a camada superior, responsável pela configuração e monitoramento do ambiente; e a camada inferior, que compreende o gerenciamento do ambiente de escalonamento distribuído. A divisão entre as camadas pode ser vista na Figura 4.1.

Figura 4.1 - Esquema estrutural do AMIGO (Souza, 2000)

A camada superior é composta basicamente por uma interface gráfica que oferece opções de configuração do ambiente ao usuário, permitindo-o cadastrar e alterar os diversos elementos conceituais da plataforma, como políticas de escalonamento, classes de *software*, aplicações, *hardware*, *benchmarks* e métricas de monitoramento. Tais informações são

registradas em diversos arquivos de configuração que são posteriormente lidos pelos componentes da camada inferior. A interface também oferece opções de monitoramento do ambiente e simulação de políticas. Grande parte da camada superior foi implementada durante o trabalho de mestrado de Campos Jr. (2001).

A camada inferior compreende o próprio ambiente de escalonamento, sendo, portanto, a mais importante. É composta por três módulos: o núcleo do ambiente, as políticas de escalonamento e o ambiente de passagem de mensagens. A camada inferior do AMIGO será detalhada mais adiante neste capítulo.

Neste trabalho, apenas a camada inferior do AMIGO foi abordada. A camada superior não sofreu alterações, e as alterações realizadas na camada inferior não afetam a sua interoperabilidade com a camada superior.

4.2.1. Elementos conceituais do ambiente

O ambiente especifica alguns elementos conceituais que são visíveis ao usuário, e participam da atividade de escalonamento. Esses elementos são definidos pelo usuário através da camada superior ou diretamente através dos arquivos de configuração do ambiente. Tais elementos são classes de *software*, aplicações e políticas de escalonamento.

a. Classes de Software

As classes de *software* são utilizadas pelo usuário para definir características comuns a conjuntos de aplicações paralelas, de forma a flexibilizar os tipos de aplicação que serão escalonados. O ambiente não define nenhuma classe de *software* padrão, deixando ao usuário a tarefa de defini-las de acordo com a sua necessidade ou a taxonomia que lhe for mais apropriada. Cada classe de *software* contém uma ou mais aplicações que serão executadas sobre o ambiente paralelo distribuído. Exemplos de classes de software são: *CPU-bound*, *I/O-bound*, *memory-intensive*, interativa, *batch*, etc.

b. Aplicações

A fim de permitir um escalonamento adequado a cada tipo de aplicação, o usuário deve cadastrar quais aplicações serão executadas, relacionando com a classe de *software* à qual tal aplicação pertence. Essa informação é utilizada pelo ambiente para determinar a política de escalonamento que será responsável por alocar novos processos daquela aplicação.

c. Políticas de Escalonamento

Cada política de escalonamento corresponde a um programa executável instalado no

ambiente, que é executado e se comunica com o *AMIGO Daemon* para servir às solicitações de escalonamento do ambiente de passagem de mensagens. Políticas de escalonamento são relacionadas a classes de software, seja manualmente (determinado pelo usuário) ou automaticamente (utilizando características das classes de software e das políticas de escalonamento fornecidas pelo usuário para determinar qual política melhor atende a cada classe de software). Apesar de especificado no projeto original do ambiente, o relacionamento automático de classes e políticas ainda não foi implementado.

Cada política de escalonamento pode servir a uma ou mais classes de software. O relacionamento entre políticas de escalonamento, classes de software e aplicações é representado na Figura 4.2.

Figura 4.2 - Elementos conceituais do ambiente *AMIGO*.

Políticas diretamente relacionadas a classes de software são chamadas de políticas específicas. Além das políticas específicas, o ambiente ainda mantém duas associações especiais para contingência: política independente e política reserva. A política independente é utilizada para atender solicitações de escalonamento genéricas, para aplicações não cadastradas no ambiente, ou cuja classe de software não foi relacionada a uma política específica. A política reserva é utilizada como último recurso, se o ambiente falhar em determinar ou carregar ambas as políticas específica e independente.

4.2.2. A camada inferior

Como dito anteriormente, a camada inferior do AMIGO é formada por três módulos: o núcleo do ambiente, as políticas de escalonamento e o ambiente de passagem de mensagens. O núcleo do ambiente é composto pelo processo servidor AMIGOD (*AMIGO Daemon*), que atua como um elo entre o ambiente de passagem de mensagens e as políticas de escalonamento. As políticas de escalonamento são implementadas como processos, que se comunicam com o núcleo através de soquetes. O ambiente de passagem de mensagens é uma versão especialmente adaptada do PVM (Araújo, 1999), LAM/MPI (Figueiredo, 2000), TAO/CORBA (Santos, 2001) ou WARPED (Voorsluys, 2006). A Figura 4.3 mostra um esquema estrutural da camada inferior do ambiente.

Figura 4.3 - Camada inferior do AMIGO.

Dada a natureza distribuída do AMIGO, cada computador que compõe o ambiente paralelo distribuído executa uma instância de cada um dos módulos da camada inferior.

4.2.2.1. O núcleo da camada inferior – O AMIGO Daemon

O AMIGOD é um processo servidor que atua no gerenciamento das políticas de

escalonamento, e no roteamento de mensagens entre o ambiente de passagem de mensagens, as políticas de escalonamento e as outras instâncias do AMIGOD nos outros computadores do ambiente.

O processo do AMIGOD é iniciado pelo ambiente de passagem de mensagens, modificado para dar suporte ao escalonamento pelo ambiente AMIGO, durante a sua inicialização.

Como o projeto do AMIGO pretende servir a qualquer ambiente de passagem de mensagens, a comunicação entre as instâncias do AMIGOD em cada computador do ambiente é realizada através de um canal lógico de comunicação independente do ambiente de passagem de mensagens.

O AMIGOD mantém tabelas com informações sobre as outras instâncias em execução nos outros computadores do ambiente (como nome do computador e endereço de comunicação) e das políticas em execução (nome, rótulo, classe, endereço de comunicação e índice de carga fornecido pela política). Outras informações sobre o ambiente são obtidas dos arquivos de configuração, que são gerados pela camada superior ou criados manualmente pelo usuário.

O AMIGOD foi implementado parcialmente durante os trabalhos de doutorado de Souza (2000) e de mestrado de Araújo (1999).

4.2.2.2. As políticas de escalonamento

As políticas de escalonamento são implementadas como processos independentes, que são iniciados pelo AMIGOD, automaticamente ou sob demanda. As políticas de escalonamento se conectam ao AMIGOD e entram em estado de espera, aguardando por solicitações de escalonamento.

A política também é responsável por monitorar o sistema local e coletar índices de utilização e carga. Tais índices são repassados para o AMIGOD, que os distribui para os outros nós do ambiente.

Ao receber uma solicitação de escalonamento, a política deve enviar uma resposta de volta ao AMIGOD com informações sobre quais computadores do ambiente deverão receber as novas tarefas, e quantas tarefas serão carregadas em cada nó.

Diversas políticas de escalonamento foram implementadas para sua execução sobre o ambiente AMIGO:

a. DPWP – Dynamic Policy Without Preemption

Implementada durante o trabalho de mestrado de Araújo (1999), é uma política que utiliza como índice de carga o número de processos em fila no sistema operacional, classificando cada nó de acordo com a sua carga em “ocioso”, “moderado” e “sobrecarregado”. Portanto, a política visa a otimizar o escalonamento para aplicações que fazem intensa utilização de processamento.

b. I/O Best

Implementada por Voorsluys e Souza (2002), é uma política distribuída voltada a medir a eficiência de índices de carga independentes, analisando a relação entre diferentes índices e avaliando sua eficiência.

c. MinMax

Implementada por Pereira e Souza (2003), é uma política de escalonamento para aplicações *CPU-bound* que adota como diretivas de escalonamento o balanceamento de carga e o conceito de recursos mínimos.

4.2.2.3. O ambiente de passagem de mensagens

O ambiente de passagem de mensagens é o componente original do ambiente sobre o qual as aplicações distribuídas são executadas. Qualquer ferramenta que realize ou necessite de escalonamento de processos sobre um ambiente paralelo distribuído pode ser utilizada em conjunto com o AMIGO, feitas algumas adaptações.

As adaptações consistem em:

- Iniciar e encerrar o processo do AMIGOD em paralelo ao próprio ambiente de passagem de mensagens, em cada computador participante do ambiente.
- Desviar a atividade de escalonamento de processos para que estas sejam repassadas ao AMIGOD, aguardar a resposta e então distribuir as novas tarefas de acordo com as instruções retornadas pelo AMIGOD.

Os seguintes ambientes de passagem de mensagens foram adaptados para sua execução integrada ao ambiente AMIGO:

a. Parallel Virtual Machine (PVM)

O ambiente PVM 3.3.11 foi o primeiro adaptado para sua utilização integrada ao AMIGO, durante o trabalho de Araújo (1999), validando a especificação do AMIGO e

servindo como piloto para as futuras adaptações de outros ambientes. Os ganhos de desempenho para aplicações desenvolvidas sobre o modelo PVM foram evidentes, conforme documentados por Araújo et al. (1999) e Souza et al. (1999).

b. Local Area Multicomputer / Message Passing Interface (LAM/MPI)

O trabalho de Figueiredo (2000) objetivou a adaptação do ambiente LAM 6.3, permitindo que aplicações desenvolvidas sobre a especificação MPI pudessem se beneficiar do escalonamento oferecido pelo AMIGO. Assim como para o PVM, os ganhos de desempenho para as aplicações desenvolvidas sobre este modelo foram sensíveis.

c. The ACE Orb / Common Object Request Broker Architecture (TAO/CORBA)

O trabalho de Santos (2001) abordou o ambiente CORBA, através da adaptação da implementação TAO, para sua integração ao AMIGO. Os ganhos de desempenho foram notados em quase todos cenários avaliados. O trabalho também compara o desempenho obtido em cada uma das três implementações (PVM, LAM e TAO) até a data.

d. WARPED

A plataforma de simulação distribuída WARPED (Martin et al., 2003), apesar de não ser um ambiente de passagem de mensagens propriamente dito (utiliza o MPI para essa finalidade), foi adaptada durante o trabalho de Voorsluys (2006) de forma a utilizar as políticas de escalonamento do AMIGO para auxiliar no particionamento de objetos da simulação que são distribuídos para os nós trabalhadores. O trabalho demonstrou a utilização do ambiente AMIGO de uma forma inédita, onde uma aplicação distribuída utiliza o AMIGO diretamente, ao invés de usar o ambiente de passagem de mensagens como meio.

4.2.3. Comunicação na camada inferior

Os módulos da camada inferior se comunicam através de um protocolo próprio, sobre os protocolos de transporte TCP (*Transmission Control Protocol*) e UDP (*User Datagram Protocol*), utilizando a interface padrão de soquetes do sistema operacional para isso. Para a comunicação no mesmo computador, ou seja, entre o núcleo e as políticas de escalonamento e entre o núcleo e o ambiente de passagem de mensagens, o protocolo TCP é utilizado, pois fornece boa confiabilidade. A comunicação entre instâncias do AMIGOD em diferentes computadores do ambiente distribuído ocorre através do protocolo UDP, devido a sua escalabilidade.

4.2.3.1. Protocolo de comunicação

Uma vez que o AMIGO atua de forma independente ao ambiente de passagem de mensagens, o ambiente define o seu próprio protocolo de comunicação, através da troca de mensagens formatadas.

Cada mensagem é composta por um cabeçalho de 16 bytes, conforme mostrado na Figura 4.4, seguido por um conteúdo específico da mensagem. Tipos de mensagens são identificados através de um código, dividido em três categorias de acordo com os participantes da comunicação (AMIGOD $\leftrightarrow$ AMIGOD, AMIGOD $\leftrightarrow$ Política e AMIGOD $\leftrightarrow$ ambiente de passagem de mensagens).

Figura 4.4 - Cabeçalho das mensagens da camada inferior do AMIGO – projeto original (Souza, 2000).

Grande parte das mensagens trocadas entre as entidades⁸ da camada inferior seguem o modelo de solicitação e resposta, onde uma entidade faz uma solicitação (por exemplo, alocação de novas tarefas), e aguarda uma resposta da outra parte. Para essa finalidade, tais mensagens são definidas aos pares, por exemplo, *AP_SCHED* (solicitação) e *AP_SCHEDACK* (resposta). Contudo, não é definido pelo protocolo um controle de sequência de mensagens e respostas.

4.2.4. Migração de processos

O projeto original do ambiente AMIGO considerava migração de processos, especificando quatro rotinas principais e vinte mensagens para essa finalidade. No entanto, nenhuma implementação que contemplasse migração de processos havia sido feita até este trabalho.

As rotinas originalmente especificadas para esta finalidade são (Souza, 2000):

8 Aqui, “entidades” se referem às instâncias dos módulos da camada inferior do AMIGO em execução em cada nó do ambiente paralelo distribuído. Essa definição é necessária para evitar confusão entre objeto e classe, ou seja, duas instâncias do AMIGOD em execução em nós distintos são entidades diferentes, porém o mesmo módulo.

- *SendProcessPolicy()*: envia para a política de escalonamento a relação de processos em execução no computador local.
- *StopProcess()*: notifica o ambiente de passagem de mensagens que processos deverão ser suspensos e que devem ser tomadas medidas para iniciar a migração de processos, sendo: (1) criação de uma tabela de migração que mantém os endereços antigo e novo de cada processo a ser transferido, (2) remoção do processo da estrutura de dados que mantém a relação de processos no processador, (3) suspensão temporária da comunicação realizada pelo processo, e (4) confirmação de quais processos foram notificados e preparados com sucesso para a transferência.
- *AcceptNewProcess()*: informa ao ambiente de passagem de mensagens que processos estão sendo transferidos para este computador e que devem ser tomadas medidas para receber o processo migrado, sendo: (1) criar a configuração necessária para o processo dentro do ambiente de passagem de mensagens, (2) restaurar a comunicação deste com os outros processos e também com o ambiente de passagem de mensagens, (3) solicitar as mensagens que tenham chegado para o processado no computador de origem durante a migração.
- *RouteMessageForNewAddress()*: corrige o endereçamento de uma mensagem destinada a um processo já transferido para outro computador. Essa rotina também é responsável por enviar para o computador de destino as mensagens que possam ter chegado para o processo durante a migração.

As mensagens originalmente especificadas por Souza (2000) incluem mensagens para solicitar a relação de processos locais para o ambiente de passagem de mensagens, avisar a suspensão de processos, transferir processos e transferir resíduos de processos.

4.2.5. Considerações sobre o projeto original

Ao longo do trabalho, algumas características de projeto e implementação do ambiente precisaram ser revistas para a implementação adequada do suporte à migração de processos.

4.2.5.1. Protocolo de comunicação

A implementação original do ambiente exigia uma comunicação síncrona entre os módulos da camada inferior, em especial entre o ambiente de passagem de mensagens e o AMIGOD. Isso torna difícil o processamento de múltiplas mensagens de mesmo tipo, e

eventos que iniciam fora do ambiente de passagem de mensagens, como a migração de processos.

A Figura 4.5 mostra a seqüência de mensagens trocadas durante um escalonamento de processos simples solicitado pelo ambiente de passagem de mensagens. O ambiente de passagem de mensagens solicita o escalonamento ao AMIGOD através da mensagem *AM_SCHED*, que repassa a solicitação para a política através da mensagem *AP_SCHED*. A política responde ao AMIGOD com as instruções de escalonamento através da mensagem *AP_SCHEDACK*, que então repassa a resposta para o ambiente de passagem de mensagens através da mensagem *AM_SCHEDACK*. Durante esse procedimento, nenhuma outra solicitação pode ocorrer de forma concomitante, pois o ambiente de passagem de mensagens não espera receber qualquer outra mensagem além da *AM_SCHEDACK*.

Figura 4.5 - Seqüência de mensagens para uma alocação simples de tarefas no ambiente AMIGO – projeto original.

A solução para essa limitação é simples, mas exige alterações fundamentais no protocolo utilizado entre os módulos da camada inferior. A solução consiste em atribuir um número de seqüência a cada mensagem enviada por cada módulo, e usar o mesmo número para identificar cada uma das respostas. A Figura 4.6 mostra um exemplo de comunicação entre os componentes da camada inferior utilizando controle de seqüência. Uma vez que mensagens de resposta possuem características distintas das mensagens de solicitação, não se faz mais necessária a utilização de códigos especiais (como *AM_SCHEDACK* e *AP_SCHEDACK* do exemplo anterior) para respostas.

Figura 4.6 - Exemplo de seqüência de mensagens utilizando controle de seqüência.

Essa abordagem para a construção do protocolo permite uma implementação mais clara e robusta.

4.2.5.2. Migração de processos

A especificação original do AMIGO aborda o problema da migração de processos com certa profundidade, atingindo detalhes de implementação do mecanismo de migração (como suspensão do processo, transferência do processo e transferência dos resíduos), bem além do controle da política de migração. A Figura 4.7 mostra a seqüência de mensagens para migração de processos conforme o projeto original. Essa abordagem apresenta limitações e dificuldades de implementação.

Conforme visto nas seções 3.4 e 3.5, a política de migração (que determina o momento e os participantes) e o mecanismo de migração (que realiza o procedimento) são dois componentes distintos de um ambiente com suporte à migração de processos. Uma abordagem mais adequada seria manter essa separação também no ambiente AMIGO.

Diferentes ambientes com suporte à migração de processos possuem diferentes mecanismos de migração de processos, com características que diferem, como: quando a execução do processo é interrompida no nó de origem, quando a execução do processo é reiniciada no nó de destino, se o processo é transferido totalmente ou parcialmente, se o

ambiente é capaz de realizar um *rollback*⁹ na eventualidade de uma falha, etc. Mesmo se considerar as mensagens e rotinas propostas como uma interface generalizada, que pudesse ser adaptada para implementações específicas de migração de processos, isso tornaria a implementação do AMIGO e da política de escalonamento dependentes de um ambiente paralelo distribuído específico.

Figura 4.7 - Sequência de mensagens para a migração de processos no ambiente AMIGO – projeto original.

A implementação exigiu alterações substanciais nessa abordagem, que serão detalhadas em maior profundidade mais adiante neste capítulo, na Seção 4.3.3.

⁹ Um *rollback* ocorre quando por qualquer razão há uma falha na transferência ou no reinício do processo no nó de destino, e o ambiente desiste da migração e reinicia o processo novamente em seu nó original.

4.3. Alterações no Projeto

Para atingir os objetivos deste trabalho, ficou evidente que um número de alterações no projeto original do AMIGO seriam necessárias. Algumas alterações (como o protocolo de comunicação) tocaram em elementos fundamentais da camada inferior do ambiente. Essa necessidade foi vista também como uma oportunidade para se processar outras adaptações, de forma a trazer novas melhorias ao ambiente.

As alterações realizadas foram inspiradas em princípios de boa qualidade de software, entre elas: alta coesão, baixo acoplamento, concisão, consistência, reusabilidade de código, portabilidade e extensa documentação de código. Portanto, acredita-se que tais alterações agregaram um valor muito positivo ao projeto.

4.3.1. O AMIGO *Daemon*

Foi observado que ao longo dos trabalhos passados que envolveram a adaptação dos ambientes de passagem de mensagens PVM, (Araújo, 1999) LAM (Figueiredo, 2000), TAO (Santos, 2001) e WARPED (Voorsluys, 2006), todos, com exceção da própria implementação original (PVM), exigiram alterações no núcleo do ambiente, na forma como é inicializado ou na forma como é feita a primeira conexão com o ambiente de passagem de mensagens. Essas alterações foram necessárias devido à dependência do AMIGOD ser carregado durante a inicialização do ambiente de passagem de mensagens, uma vez que cada um dos ambientes possuem diferentes peculiaridades na forma como ocorrem suas inicializações. Essa complexidade de implementação é indesejável, uma vez que o projeto do AMIGO almeja fornecer uma interface padrão e estável para prover serviços de escalonamento de processos. Por essa razão, optou-se por redesenhar a forma como o AMIGOD é inicializado, utilizando-se como experiência as lições aprendidas nos trabalhos anteriores.

Assim, o AMIGOD foi alterado de forma que a sua inicialização seja independente por padrão, removendo sua subordinação em relação ao ambiente de passagem de mensagens. Dessa forma, o AMIGOD é instalado e executado como um serviço do sistema em cada computador que compõe o *cluster*, as instâncias em cada computador se comunicam e fazem a auto-descoberta das características do ambiente distribuído, montando sua própria tabela de nós. Ao invés de iniciar a conexão com o ambiente de passagem de mensagens, o AMIGOD simplesmente aguarda por conexões em uma porta padrão (atendendo a requisitos encontrados ao adaptar os ambientes LAM e WARPED) e também é capaz de atender a múltiplas conexões (atendendo a requisitos encontrados ao adaptar o ambiente LAM). Automaticamente, o AMIGOD passa a ser capaz de, em uma única instância, atender a

diversos ambientes de passagem de mensagens em execução simultaneamente no mesmo *cluster* de computadores.

De forma a reconhecer o fato de que o AMIGOD pode atender a aplicações como o WARPED, que não é um ambiente de passagem de mensagens propriamente dito, a implementação passa a chamar simplesmente de “clientes” as aplicações que solicitam serviços de escalonamento.

Como o AMIGOD realiza a auto-descoberta da configuração do *cluster*, a tabela de nós pode ser compartilhada com os clientes, de forma a evitar configurações adicionais e individuais para cada ambiente de passagem de mensagens.

4.3.2. Protocolo de comunicação

O protocolo de comunicação entre os módulos da camada inferior foi completamente reestruturado, de forma a atender aos requisitos de uma comunicação totalmente assíncrona entre as entidades, baseado em solicitação e resposta com controle de seqüência.

A comunicação entre os módulos da camada inferior é gerenciada por um conjunto de objetos e funções que cria uma camada de abstração entre o protocolo do AMIGO e detalhes de sua implementação sobre os protocolos da camada de transporte (TCP e UDP). Esse conjunto de objetos e funções (chamado de “camada de protocolo”) é compartilhado por todos os módulos através de uma biblioteca de sistema, de forma que futuras alterações no protocolo causem o mínimo de impacto na implementação dos diversos módulos, bastando no máximo uma recompilação.

A camada de protocolo cuida de forma transparente da codificação e decodificação dos cabeçalhos das mensagens, controle de seqüência, *buffers* de entrada e saída, monitoramento de soquetes e controle de resposta. Cada mensagem enviada é numerada seqüencialmente e caso uma resposta seja esperada da outra parte, uma função de *callback* é registrada junto à camada de protocolo. A função de *callback* é chamada automaticamente pela camada de protocolo ao receber a resposta da outra parte, ou no evento de uma falha (queda de conexão, tempo de espera esgotado).

O conteúdo das mensagens é representado por variáveis estruturadas, e codificados através do formato XDR (Srinivasan, 1995), que é um padrão para descrição e codificação de dados que garante interoperabilidade para a comunicação entre diferentes arquiteturas.

As entidades do ambiente são identificadas através de um Identificador Universalmente Único (*UUID – Universally Unique Identifier*). *UUID* é um número de 128

bits usado para identificar objetos em um ambiente distribuído. UUID garante unicidade através do tempo e do espaço, e dispensa a existência de uma entidade central de registro (ITU, 2004; Leach et al., 2005).

O cabeçalho das mensagens trocadas pelos componentes do ambiente AMIGO é mostrado na Figura 4.8. O primeiro campo do cabeçalho, “*Magic*” (4 bytes), é um “número mágico” único e igual para todas as mensagens trocadas. Esse campo serve para diferentes propósitos dependendo do protocolo da camada de transporte. Para comunicação sobre soquetes TCP, a leitura correta do número mágico no fluxo de dados indica que ambas as partes estão corretamente sincronizadas em sua comunicação, a leitura de um número diferente do número mágico sinaliza um provável defeito de implementação. Para comunicação sobre soquetes UDP, a leitura de um número diferente do número mágico indica que certamente o datagrama recebido não faz parte do protocolo do AMIGO, possivelmente alguma outra aplicação enviando pacotes para a mesma porta usada pelo AMIGOD. O número mágico também pode ser usado para evitar e sinalizar a comunicação entre versões incompatíveis dos componentes do ambiente.

Figura 4.8 - Cabeçalho das mensagens da camada inferior.

O próximo campo, “*Code*” (2 bytes), contém o código da mensagem. “*Length*” (2 bytes) é o comprimento da mensagem que segue o cabeçalho. Os dois próximos campos, “*Seq*” e “*Ack*” (2 bytes cada), fazem parte do controle de sequência e resposta; o primeiro contém um número seqüencial para cada mensagem enviada, enquanto o segundo indica a qual mensagem recebida pela outra parte tal mensagem é resposta. O campo “*Flags*” (2 bytes) contém sinalizadores diversos, como se a mensagem espera uma resposta da outra parte ou se é uma resposta para a outra parte. Os dois últimos campos, “*Destination UUID*” e “*Origin UUID*” (16 bytes cada) identificam as entidades de destino e origem da mensagem.

Os tipos de mensagens são divididos em cinco categorias (em contraste a três categorias no projeto original), dependendo do módulo que deve receber e interpretar a mensagem. As categorias são listadas na Tabela 4.1. Além das cinco categorias padrão, uma sexta categoria é reservada para mensagens privadas entre componentes, que não são interpretadas pelo AMIGOD (por exemplo, entre duas políticas).

Tabela 4.1 - *Categorias de mensagens da camada inferior.*

Categoria	Orig.	Descrição
AM_MSG_PEER	AA	Mensagens trocadas entre instâncias do AMIGOD rodando em diferentes nós.
AM_MSG_CLIENT	AM	Mensagens enviadas pelo AMIGOD para clientes.
AM_MSG_DAEMON	AM	Mensagens enviadas por clientes para o AMIGOD.
AM_MSG_POLICY	AP	Mensagens enviadas pelo AMIGOD para políticas.
AM_MSG_SCHED	AP	Mensagens enviadas por políticas para o AMIGOD.

Finalmente, o AMIGOD é capaz de utilizar *multicast* de datagramas em nível de protocolo de rede (IPv4 ou IPv6) quando possível. Isso é particularmente benéfico para mensagens de atualização de índices de carga e configuração do *cluster*, que devem ser enviadas para todos os nós, reduzindo assim o tráfego na rede. A configuração da comunicação *multicast* ocorre automaticamente quando possível, de acordo com os procedimentos descritos nos RFCs 3306 (Haberman e Thaler, 2002) e 4489 (Park et al., 2006). Quando não é possível a utilização de *multicast*, o ambiente utiliza automaticamente o envio de mensagens individuais para cada nó.

4.3.3. Migração de processos

O suporte à migração de processos foi implementado visando a uma separação bem distinta entre a política de migração e o mecanismo de migração, onde a primeira deve ser implementada no ambiente AMIGO, enquanto que a segunda é considerada como detalhe específico do cliente, que o ambiente de escalonamento não deverá interferir.

A migração de processos é iniciada de forma assíncrona pela política de escalonamento. O ambiente disponibiliza à política as informações sobre os clientes conectados, os processos gerenciados por cada cliente e os índices de carga dos outros nós do *cluster*. Com essas informações, a política de ativação é capaz de decidir o momento de realizar a migração de processos, a política de seleção pode escolher quais processos serão migrados, e a política de localização determina o destino dos processos escolhidos.

A migração de processos inicia quando a política envia a mensagem *AM_MSG_SCHED_MIGRATE_TASKS* para o AMIGOD, contendo a identificação do cliente que será comandado a migrar processos, uma lista de receptores para os processos migrados e, para cada receptor, a lista de processos locais que serão transmitidos para cada receptor. As instruções são repassadas para o cliente através da mensagem *AM_MSG_CLIENT_MIGRATE_TASKS*.

Durante a execução do mecanismo de migração de processos pelo cliente, o AMIGOD recebe mensagens *AM_MSG_DAEMON_TASK_STARTED* e *AM_MSG_DAEMON_TASK_FINISHED* do cliente, conforme os processos são migrados entre os computadores do *cluster*. Ao término do procedimento, o cliente responde à solicitação original com informações sobre o êxito da migração, quantos receptores aceitaram receber os processos transferidos e quantos processos foram totalmente transferidos (sem *rollback*). O AMIGOD, por sua vez, retorna a mesma informação à política de escalonamento. A Figura 4.9 mostra a seqüência de mensagens de uma migração de processos.

Figura 4.9 - Seqüência de mensagens para a migração de processos no ambiente AMIGO.

Essa abordagem é muito semelhante à mesma abordagem já utilizada no projeto original para a alocação inicial: o AMIGO apenas fornece instruções sobre quantos processos devem ser criados em cada receptor, independentemente da maneira como o ambiente de passagem de mensagens irá executar os novos processos.

Essa abordagem foi verificada através da adaptação com sucesso do ambiente de passagem de mensagens DPVM 2.0, e da implementação da política de escalonamento com suporte à migração de processos JUMP, que será vista no capítulo 5.

4.3.4. Implementação

Este trabalho resultou na adaptação de três componentes do ambiente AMIGO, e na implementação de outros quatro novos componentes.

O núcleo da camada inferior do ambiente, o AMIGOD, foi profundamente alterado

de forma a refletir as alterações no projeto descritas nas seções anteriores. Todos os recursos encontrados na implementação original continuam presentes na nova versão, com diversas melhorias. O suporte à migração de processos foi implementado e seu funcionamento correto foi validado, cumprindo este objetivo do trabalho.

O ambiente de passagem de mensagens PVM 3.4.5 (Geist et al., 1994) foi adaptado para seu funcionamento integrado à nova versão do AMIGO. A adaptação difere do trabalho original de Araújo (1999) em que a comunicação com o AMIGOD é integrada ao monitoramento de soquetes nativamente implementado no PVM, permitindo receber solicitações assíncronas do AMIGOD, da mesma forma como ocorre com processos do ambiente. A nova adaptação também permite o PVM inicializar seu ambiente paralelo distribuído automaticamente através de informações de nós obtidas do AMIGOD.

O ambiente de passagem de mensagens DPVM 2.0 (Iskra et al, 2000a; 2000b) também foi adaptado para seu funcionamento integrado à nova versão do AMIGO. A adaptação realizada possui grandes semelhanças à mesma adaptação realizada para o PVM 3.4.5, apenas com a adição do suporte à migração de processos que o DPVM oferece.

Um novo cliente, chamado AMIGOC (*AMIGO Client*), foi implementado. Esse cliente se conecta ao AMIGOD como qualquer outro cliente ou ambiente de passagem de mensagens e permite um controle em linha de comando de algumas funções do AMIGOD, como simular escalonamentos de processos, mostrar a tabela de nós (com os índices de carga das políticas), e encerrar a execução do AMIGOD. O principal objetivo desse cliente é permitir testes rápidos do ambiente e das políticas, assim como ter um exemplo simples e claro de como é feita a comunicação com o AMIGOD, que possa servir como guia para próximas implementações.

A política de escalonamento DPWP (Araújo, 1999), descrita na Seção 4.2.2.2, foi portada para sua execução sobre a nova versão do AMIGO. Grande parte da implementação original, que constituía originalmente quase 1.500 linhas de código em linguagem C, era responsável por gerenciar a conexão com o núcleo do ambiente e a codificação de mensagens. Na nova implementação, utilizando a camada de protocolo já pronta e compartilhada entre todos os módulos do ambiente, o código foi reduzido para cerca de 400 linhas, que é exatamente o que compõe o algoritmo real da política. Dessa forma, é mostrado que alguns objetivos da alteração do projeto foram atingidos: concisão e reusabilidade de código, permitindo um desenvolvimento mais simples e maior qualidade de *software*.

Uma nova política de escalonamento simples *round-robin*, visando a servir para duas

finalidades. Primeiramente, permitir a comparação do escalonamento padrão do ambiente de passagem de mensagens, que geralmente é uma política *round-robin*, com essa mesma política implementada sobre o ambiente AMIGO, de forma a determinar e analisar o *overhead* imposto pela utilização do AMIGO sobre o escalonamento. Tal aferição não foi feita desde o projeto original do AMIGO. Em segundo lugar, a simplicidade da política (apenas 120 linhas de código) deve servir como referência e ajudar como um ponto-de-partida para a implementação de novas políticas de escalonamento.

Finalmente, a nova política de escalonamento unificada com suporte à migração de processos JUMP, foi projetada e implementada para sua execução sobre a nova versão do ambiente AMIGO e foi testada com a utilização do ambiente de passagem de mensagens DPVM 2.0. O projeto e implementação da política serão descritos no capítulo 5.

4.4. Considerações Finais

O AMIGO é um ambiente que fornece suporte às decisões de escalonamento de processos para ambientes paralelos distribuídos, oferecendo flexibilidade e transparência ao usuário. A viabilidade do ambiente para a alocação inicial de processos foi demonstrada através de vários trabalhos desde a sua concepção até a data.

O trabalho original de Souza (2000) merece destaque pela sua inovação e o sucesso dos objetivos alcançados: reduzir a lacuna entre teoria e prática na área de pesquisa de políticas de escalonamento global de processos.

Para a implementação do suporte à migração de processos, alterações foram necessárias no projeto e na implementação do ambiente. Este capítulo apresentou um resumo das características principais do projeto original, assim como uma descrição geral das alterações que foram feitas. Dessa forma, este capítulo visou a complementar o trabalho de Souza (2000), de forma a servir como guia para futuros pesquisadores que optarem por utilizar a nova versão do ambiente.

Nem todos os componentes do AMIGO produzidos até a presente data foram portados para a nova versão do ambiente. Em especial, ainda falta a adaptação dos outros ambientes de passagem de mensagens além do PVM, e das outras políticas de escalonamento além da DPWP, que por enquanto só possuem compatibilidade com a versão anterior do AMIGO. A conversão desses componentes não foi realizada por limitações de tempo e do escopo deste trabalho. Cuidado deve ser tomado ao adaptar tais componentes, de forma a garantir que os mesmos resultados sejam mantidos em ambas as versões do ambiente. As

melhorias feitas no ambiente, o modelo utilizado que enfatiza a reutilização de código e simplicidade de implementação e a extensa documentação do código fonte devem facilitar bastante o trabalho de adaptação dos componentes restantes. A utilização apropriada das novas bibliotecas de sistema do AMIGO deve garantir a compatibilidade continuada dos módulos do ambiente, mesmo se forem necessárias novas alterações fundamentais na implementação.

5. A Política de Migração JUMP

5.1. Considerações Iniciais

O trabalho de Furquim (2006), entre outros resultados, demonstrou que a utilização de políticas de escalonamento de alocação inicial e de migração de processos em conjunto oferece um melhor balanceamento de carga do que a utilização de apenas uma delas, pois a migração de processos é capaz de equilibrar a carga do ambiente paralelo distribuído de forma mais dinâmica, através da preempção.

O trabalho de Araújo (1999) apresentou uma política de escalonamento distribuída (DPWP) para aplicações de uso intensivo de processamento, em que a carga de cada computador do ambiente paralelo distribuído é classificada em três níveis, e a alocação de novos processos ocorre prioritariamente nos computadores menos carregados.

Neste trabalho, é apresentada uma nova política de escalonamento preemptiva, denominada JUMP, que estende as idéias apresentadas nos trabalhos anteriores, unindo a alocação inicial e a migração de processos em uma única política de escalonamento.

O nome JUMP é um acrônimo recursivo que significa *Jump Unified Migration Policy*, ou política de migração unificada *Jump*. A característica unificada da política, presente em seu nome, se refere a unificar em um mesmo algoritmo as decisões de alocação inicial e migração de processos.

Neste capítulo serão mostrados os objetivos, o projeto e as características da implementação da política. A avaliação de desempenho da política será apresentada no capítulo 6.

5.2. Objetivos Gerais

A política de escalonamento JUMP visa a obter um melhor balanceamento de carga no ambiente paralelo distribuído, para aplicações de processamento intensivo (*CPU-bound*) através da unificação das decisões de alocação inicial e de migração de processos na mesma política.

A política leva em consideração que a migração de processos é um procedimento custoso para o ambiente distribuído, e deve ser evitada quando possível. A relevância dessa consideração está ligada ao desenvolvimento piloto da política, que visava a funcionar em conjunto com o ambiente DPVM, cujo algoritmo do mecanismo de migração é o *eager-copy*, o menos eficiente de todos.

Ao atender a uma solicitação de alocação inicial de processos, a política visa a tomar decisões que irão beneficiar também futuras migrações de processos. O mesmo ocorre quando a política ativa a migração de processos: a decisão sobre quais serão os receptores dos processos migrados visa a beneficiar futuras alocações iniciais.

Para atender a esse objetivo, a política periodicamente distribui e coleta informações de carga de todos os computadores do ambiente e utiliza uma heurística para prever as alterações de carga para cada outro computador, até o próximo ciclo de distribuição e coleta de informações de carga. A cada ciclo de distribuição e coleta de informações de carga, a política observa o reflexo das decisões previamente tomadas através da média do número de processos em execução e o índice de carga de processamento.

Através da observação do comportamento do índice de carga de processamento relativo ao número de processos em execução em cada computador do ambiente, a política determina aproximadamente qual o peso médio que cada processo representa em cada computador do ambiente. Com isso, a política detecta a heterogeneidade do *cluster* durante a sua execução, e é, portanto, mais apropriada para ambientes heterogêneos.

Diferentemente da política DPWP, os computadores do ambiente são ordenados de forma contínua em relação ao seu índice de carga e a distribuição de tarefas prioriza os nós que possuem a carga esperada mais baixa.

5.3. Desenvolvimento da Política JUMP

JUMP é uma política de escalonamento distribuída, ou seja, cada nó do *cluster* executa uma instância da política, que atende às requisições originadas localmente. As instâncias da política trocam informações de carga entre si periodicamente, de forma que cada instância conhece a carga coletada por todas as outras instâncias.

A política é executada em duas *threads*, sendo a primeira para atender as mensagens enviadas pelo AMIGOD, como solicitações de alocação inicial e ordens de migração e a segunda para cuidar de eventos assíncronos, tais como o monitoramento dos índices de carga, distribuição das informações de carga e a ativação da migração de processos quando um

desequilíbrio é observado.

As políticas de alocação inicial e de migração de processos, apesar de compartilharem informações e decisões, possuem certa independência que permite que as políticas sejam ativadas ou desativadas independentemente uma da outra. Quando a política de migração de processos é desativada, JUMP se comporta como uma política não-preemptiva, similar à DPWP, mas ainda mantendo suas heurísticas de distribuição e predição de carga dos nós. Quando a política de alocação inicial é desativada, solicitações de escalonamento resultam na alocação de todos os processos no próprio nó local.

Também é possível sobrepor a política de alocação inicial com outra política do ambiente, através de uma configuração específica do AMIGOD, tornando JUMP uma política usada exclusivamente para migração de processos. Contudo, essa configuração pode resultar em uma eficiência menor que a esperada, pois a política JUMP não terá conhecimento das decisões tomadas pela outra política e, portanto, não será capaz de atualizar a predição da carga do ambiente distribuído até o próximo ciclo de distribuição e coleta de informações de carga.

A união da alocação inicial e migração de processos em uma única política permite a reutilização de alguns componentes comuns, como a política de localização e a política de informação.

Figura 5.1 - Classificação da política JUMP pela taxonomia hierárquica de Casavant e Kuhl (1988).

De acordo com a taxonomia de políticas de escalonamento proposta por Casavant e Kuhl (1988), descrita na Seção 2.3.1, a política JUMP é classificada como uma política de

escalonamento global, dinâmica, fisicamente distribuída, cooperativa, subótima e heurística. A Figura 5.1 mostra a classificação da política JUMP de acordo com a taxonomia hierárquica. A política JUMP é também classificada como não-adaptável, com balanceamento de carga e com reatribuição dinâmica, de acordo com a taxonomia plana de Casavant e Kuhl.

5.4. Gerenciamento de Informações de Carga

O módulo de gerenciamento de informações de carga da política possui dois ciclos, um ciclo de medição e um ciclo de informação. O ciclo de medição ocorre com um período muito curto ($T_m = 1s$, na configuração padrão da política), e é responsável por medir e acumular três métricas do sistema: número de tarefas em execução, carga de processamento e utilização dos processadores. O ciclo de informação ocorre com um período maior ($T_i = 15s$), e é responsável por calcular índices de carga através das métricas coletadas e distribuir as informações de carga para o resto do ambiente.

Apesar da frequência relativamente alta dos ciclos do módulo de gerenciamento de informações de carga, cuidados foram tomados para que tais medições impusessem o mínimo de carga adicional no sistema.

Além da aferição dos índices de carga do sistema, o módulo de gerenciamento de informações também coleta informações de carga de processos para o balanceamento de carga. Essa coleta só ocorre no momento em que a política realiza uma migração de processos, por solicitação da política de seleção, para determinar quais processos serão migrados.

5.4.1. Métricas e índices de carga do sistema

De forma a determinar a carga do sistema, as seguintes métricas e índices são utilizados.

a. Número de tarefas em execução

Representa o número de processos ativos gerenciados pelo ambiente de passagem de mensagens. Todos os processos ativos são considerados, independente se estão realizando processamento, em estado de espera ou bloqueados. Processos externos ao ambiente de passagem de mensagens não são contados.

O índice de carga é obtido através da média aritmética dos valores medidos durante um ciclo de informação. É uma variável real positiva, representada em número de processos.

b. Carga de processamento

A carga de processamento é medida através do número de processos na fila de processamento do sistema. Essa métrica mede efetivamente quantas *threads* disputam pela utilização dos processadores. Esse valor é medido diretamente do sistema operacional e, portanto, inclui todos os processos em execução no sistema, independente se são ou não gerenciados pelo ambiente de passagem de mensagens.

A política leva em consideração que a própria *thread* que realiza a medição é contada no valor obtido no exato momento da medição, e subtrai em um o valor da métrica.

O índice de carga é obtido através da média aritmética dos valores medidos durante um ciclo de informação, normalizado através da divisão pela potência computacional do nó. É uma variável real positiva, representada em número de processos sobre potência computacional.

Como esse índice é normalizado através da potência computacional do nó, ele é também chamado de índice de desempenho, conforme definido por Santana (1990) e Branco (2004).

c. Utilização dos processadores

Essa métrica indica a utilização dos processadores do sistema, sendo cada processador classificado em ativo ou inativo, independente do número de *threads* disputando pela sua utilização.

O índice de carga é obtido através da média aritmética dos valores medidos durante um ciclo de observação, dividido pelo número de processadores do nó. É uma variável real entre 0,0 e 1,0 (ou 0% e 100%), representando a razão de tempo útil de processamento.

5.4.2. Métricas de carga de processos

Quando a migração de processos precisa selecionar processos que serão transferidos, as seguintes métricas de processos são utilizadas.

a. Tempos de usuário e de sistema

Essas duas métricas representam o tempo de processamento do processo, em espaço de usuário ou em chamadas de sistema respectivamente. A soma dessas duas métricas corresponde ao tempo total que o processo esteve em execução. São variáveis inteiras positivas, representadas em número de interrupções do relógio do sistema (*ticks*).

b. Tempo desde a criação do processo

Essa métrica representa o tempo real desde que o processo foi criado. É uma variável inteira positiva, representada em número de interrupções do relógio do sistema (*ticks*).

5.4.3. Compartilhamento de informações de carga

A política repassa as informações de carga coletadas para o AMIGOD através da mensagem *AM_MSG_SCHED_UPDATE_LOAD*, definida pela biblioteca de sistema comum aos componentes do AMIGO. Essa mensagem transporta apenas um campo de tipo ponto-flutuante para o índice de carga, que não é utilizado pela política. Em vez disso, a política utiliza um campo de informações de carga estendida, onde é possível transportar uma estrutura de dados arbitrária, que carrega todos os índices necessários. A estrutura *jumpLoadData* é descrita no formato XDR (Srinivasan, 1995) na Figura 5.2.

```
struct jumpLoadData {  
    int serial;  
    float tasks;  
    float load;  
    float usage;  
    bool receiving;  
};
```

Figura 5.2 - Estrutura *jumpLoadData* (XDR).

Os campos “*tasks*”, “*load*” e “*usage*” carregam os índices de carga descritos anteriormente. O campo “*serial*” é um número de série incrementado a cada nova geração das informações transmitidas, usado pela política para determinar quando é o primeiro conjunto de informações de carga enviado por outra instância da política. O campo “*receiving*” informa se a instância da política está apta a receber processos migrados, ou seja, há um ambiente de passagem de mensagens com suporte à migração de processos conectado localmente.

As informações de carga são transmitidas entre instâncias do AMIGOD através da mensagem *AM_MSG_PEER_POLICY_ACTION*, e essas por sua vez informam suas instâncias locais da política através da mensagem *AM_MSG_POLICY_REMOTE_LOAD*. A Figura 5.3 mostra a seqüência de mensagens da atualização de carga. Neste gráfico, “*Dest*” representa cada outro nó do ambiente.

Ao receber a mensagem *AM_MSG_POLICY_REMOTE_LOAD* de uma instância remota da política, as informações de carga recebidas são utilizadas para atualizar a informação local sobre o peso de processos criados no nó remoto.

Figura 5.3 - Seqüência de mensagens para atualização de informações de carga.

O peso de um nó (W) é calculado através de uma média móvel exponencial sobre a razão entre a carga de processamento (L) e o número de tarefas em execução (T):

$$W_x = \alpha \cdot \frac{L_x}{T_x} + (1 - \alpha) \cdot W_{x-1}$$

O fator de amaciamento (α) da média móvel exponencial é ajustável na configuração da política, sendo 0,1 seu valor padrão. Esse valor foi determinado empiricamente durante o desenvolvimento e teste da política, para os valores padrões dos períodos dos ciclos de medição e informação ($T_m = 1s$ e $T_i = 15s$). Valores mais altos tendem a variar excessivamente o peso dos nós, tornando a política de migração instável e, conseqüentemente, prejudicando o tempo de resposta das aplicações distribuídas.

Na situação em que o número de tarefas em execução é igual a zero, ou muito próximo de zero, a política mantém o peso do nó registrado anteriormente. Por enquanto, essa situação está mantida em aberto. Futuras versões da política podem usar essa situação para calcular a interferência de processos não gerenciados pelo ambiente sobre a carga de processamento do nó.

5.5. Política de Alocação Inicial

A política de alocação inicial é acionada em resposta a uma solicitação de escalonamento de processos recebida do AMIGOD. A política recebe o número de tarefas que devem ser criadas e identificadores da arquitetura e do sistema operacional dos computadores receptores que devem ser selecionados para o escalonamento. A política deve determinar a melhor distribuição de tarefas que satisfaça os requisitos e retornar como resposta as instruções sobre quantas tarefas devem ser criadas em cada computador do ambiente.

O primeiro passo da política de alocação inicial é determinar o conjunto dos computadores que serão receptores das novas tarefas, baseado nas restrições de arquitetura e sistema operacional. Na maioria das situações, a aplicação não impõe tais restrições, o que implica que a política recebe os valores *AM_ARCH_ANY* e *AM_OS_ANY*, que causam a

seleção de todo o conjunto de computadores do ambiente.

Para cada computador do ambiente, a política mantém informações em uma estrutura privada sobre a carga real e a carga esperada do nó. Esses índices são de fundamental importância para o algoritmo de distribuição de carga.

A carga real do nó representa a carga computacional recebida pela última vez da instância remota da política. A carga esperada é ajustada inicialmente para a carga real, mas varia conforme a instância local da política toma decisões de escalonamento, de forma a permitir um escalonamento mais preciso até a próxima coleta de informações de carga da política remota.

As tarefas são distribuídas entre o conjunto de computadores selecionados de forma a equilibrar o balanceamento das cargas, transferindo tarefas prioritariamente para os computadores com a menor carga, de forma a reduzir a variância entre as cargas dos nós. Com isso, em um *cluster* desbalanceado, a carga dos computadores menos carregados tende a se aproximar da média de carga do conjunto de computadores selecionados.

A distribuição das tarefas é realizada com base na carga esperada e o peso calculado de cada computador. O receptor de cada tarefa solicitada é determinado iterativamente, selecionando o computador com a menor carga esperada e atualizando a mesma através da soma do peso calculado daquele computador.

A Figura 5.4 mostra um exemplo de alocação inicial de nove tarefas em um *cluster* desbalanceado de cinco nós. Em (a), o *cluster* é representado em seu estado inicial. Cada barra representa a carga de cada um dos cinco nós, seguido pela representação da média e do desvio padrão das cargas do *cluster*. Os losangos sobre cada nó representam o peso calculado de uma tarefa para cada nó. Nesse exemplo, todos nós possuem pesos iguais a 1, que é característica de um *cluster* homogêneo. As tarefas são distribuídas pela política conforme mostrado em (b). A média de carga do *cluster* se eleva, em resposta à criação de novas tarefas, mas o desvio padrão das cargas (e por consequência a variância) diminui, mostrando o balanceamento de carga.

Figura 5.4 - Exemplo de alocação inicial em um *cluster* homogêneo desbalanceado.

A Figura 5.5 mostra um exemplo de alocação de nove tarefas no mesmo *cluster* homogêneo da figura anterior, porém balanceado. Nesse exemplo, a política possui um comportamento muito semelhante a uma política *round-robin*, que distribui a carga de forma cíclica entre os computadores participantes da distribuição de carga. O balanceamento do *cluster* diminui ligeiramente uma vez que o estado anterior já se encontrava em uma situação quase-ótima, e a alocação das novas tarefas não coincide precisamente de forma a manter o mesmo balanceamento.

Figura 5.5 - Exemplo de alocação inicial em um *cluster* homogêneo balanceado.

A Figura 5.6 mostra um exemplo de alocação inicial de nove tarefas, agora em um *cluster* heterogêneo desbalanceado de cinco computadores. O primeiro nó é o mais potente e possui o menor peso por tarefa, enquanto o quinto nó é o menos potente e possui o maior peso por tarefa. Nesse exemplo é possível observar com clareza o efeito que o peso do nó exerce

sobre o número de tarefas alocadas sobre o mesmo.

Figura 5.6 - Exemplo de alocação inicial em um cluster heterogêneo desbalanceado.

Finalmente, a Figura 5.7 mostra um exemplo de alocação inicial de nove tarefas no mesmo *cluster* heterogêneo da figura anterior, porém balanceado. Assim como na Figura 5.5, o balanceamento é ligeiramente prejudicado, pelas mesmas razões.

Figura 5.7 - Exemplo de alocação inicial em um cluster heterogêneo balanceado.

O índice de carga esperada permite que a política de escalonamento continue oferecendo bons resultados entre os ciclos de informação do gerenciamento de informações de carga. Caso o escalonamento de processos retornado pela política cause uma alteração diferente da esperada nos índices de carga dos nós, o índice de peso será reajustado na próxima distribuição de informações de carga, permitindo que as próximas alocações iniciais façam previsões mais precisas.

O índice de peso dos computadores, em especial a forma como é calculado e utilizado durante o escalonamento, traz um grande benefício à política, pois reflete tanto a característica de heterogeneidade do ambiente paralelo distribuído quanto o comportamento da aplicação paralela sendo executada.

5.5.1. Componentes da política da alocação inicial

Os componentes de uma política de distribuição de carga, conforme definidos por Shivaratri et al. (1992) e Milojić et al. (2000) são aplicados separadamente para as políticas de alocação inicial e de migração de processos. Exceção é feita para a política de informação, que é implementada no gerenciamento de informações de carga, explicado anteriormente.

A política de transferência, durante a alocação inicial pela política JUMP, determina que um nó é transmissor sempre que o nó local não for o menos carregado do ambiente, ou quando o número de tarefas a serem criadas fará a carga prevista do nó local ultrapassar a carga do segundo nó menos carregado do ambiente. Todos os nós são sempre potenciais receptores de tarefas transmitidas por outros nós.

A política de seleção da alocação inicial é fixa, e sempre escolhe as próprias tarefas cuja alocação está sendo solicitada no momento do escalonamento. Essa propriedade é comum às políticas de escalonamento que realizam a alocação inicial, pois sem preempção apenas as tarefas ainda a serem criadas podem ser transferidas para outros nós.

A política de localização da alocação inicial determina os nós receptores para cada tarefa iterativamente, selecionando sempre o nó menos carregado, computando a nova carga prevista daquele nó baseado em seu índice de peso, e repetindo a para a próxima tarefa até que o destino de todas as tarefas tenha sido determinado.

5.5.2. Comunicação durante a alocação inicial

A alocação inicial, em geral, é iniciada pela aplicação distribuída ou pelo ambiente de passagem de mensagens. O cliente envia a mensagem *AM_MSG_DAEMON_SCHEDULE* para o AMIGOD, carregando a estrutura *amMsgClientSched*. As informações enviadas pelo cliente são: o nome identificador do programa executável (binário) das tarefas que serão criadas, o número de instâncias a serem criadas e identificadores de arquitetura e sistema operacional, usados para restringir o conjunto de potenciais receptores das novas tarefas. O AMIGOD determina a política através do identificador do programa executável e a invoca através da mensagem *AM_MSG_POLICY_SCHEDULE*, carregando a estrutura *amMsgPolicySched*. As estruturas são descritas na Figura 5.8.

<pre>enum amPeerArch { AM_ARCH_ANY = -1, AM_ARCH_UNKNOWN = 0, AM_ARCH_ALPHA = 1, AM_ARCH_SPARC = 2, ... }; enum amPeerOS { AM_OS_ANY = -1, AM_OS_UNKNOWN = 0, AM_OS_NETBSD = 1, AM_OS_OPENBSD = 2, ... };</pre>	<pre>struct amMsgClientSched { string binary; unsigned int n_inst; amPeerArch arch; amPeerOS os; }; struct amMsgPolicySched { unsigned int n_inst; amPeerArch arch; amPeerOS os; };</pre>
--	--

Figura 5.8 - Estruturas *amMsgClientSched* e *amMsgPolicySched* (XDR).

A política retorna como resposta a lista de nós que serão receptores de tarefas, e o número de tarefas que devem ser alocadas em cada nó. As estruturas utilizadas na resposta são descritas na Figura 5.9. A sequência de mensagens é mostrada na Figura 5.10.

<pre>struct amPeer { opaque uuid[16]; string hostname; ... }; struct amMsgSchedPeerEntry { amPeer peer; unsigned int n_inst; };</pre>	<pre>struct amMsgPolicySchedReply { amMsgSchedPeerEntry peers<>; }; struct amMsgClientSchedReply { amMsgSchedPeerEntry peers<>; };</pre>
---	--

Figura 5.9 - Estrutura *amMsgPolicySchedReply* e associadas (XDR).

Figura 5.10 - Sequência de mensagens para alocação inicial de tarefas.

5.6. Política de Migração de Processos

A política de migração de processos é responsável por detectar o desbalanceamento de carga no ambiente paralelo distribuído durante a execução da aplicação paralela e comandar a migração de processos entre computadores de forma a rebalancear a carga.

Para a migração de processos, é necessária uma decisão global que determine a

ativação da política. A política deve ser coordenada de forma que durante a migração de processos, apenas um nó seja transmissor. Essa é tanto uma limitação do ambiente de passagem de mensagens utilizado durante o desenvolvimento da política, que não é capaz de gerenciar mais de uma migração de processos simultaneamente, como também é uma boa decisão para evitar queda de desempenho do ambiente devido a picos de transmissão durante transferências de estados de processos.

Por esse motivo, a política de migração possui um componente centralizado, que determina qual nó do ambiente será transmissor da migração de processos. As outras decisões relacionadas à política de migração de processos são distribuídas.

A política de migração é ativada periodicamente no nó mestre do *cluster*, a cada minuto. Quando ativada, a política obtém uma lista com todos os computadores do ambiente, com suas informações de carga, e seleciona os potenciais transmissores de acordo com um número de condições. Potenciais transmissores devem possuir o índice de utilização dos processadores próximo ou igual a 100%, índice de carga de processamento maior que um, número de tarefas em execução maior ou igual a um, e devem ser aptos a migrar processos.

Mesmo computadores com um índice de carga de processamento relativamente alto podem não ser selecionados como transmissores se a utilização dos processadores não for próxima ou igual a 100%. Isso ocorre em situações onde um grande número de processos de processamento intensivo são bloqueados periodicamente para sincronização com outros processos em outros computadores menos potentes em um *cluster* heterogêneo. Em tais situações, em geral esses computadores menos potentes se encontram com 100% de utilização dos processadores, e são mais aptos a serem transmissores.

Tendo determinado os potenciais transmissores, a política comanda a ativação da migração de processos a partir do computador mais carregado. A partir desse momento, a política de ativação continua sua execução no nó selecionado e não mais no nó mestre.

No potencial transmissor escolhido pelo nó mestre, algumas verificações são realizadas, ainda como parte da política de ativação: se o computador realmente está apto a participar da migração, se há pelo menos um outro computador no ambiente compatível para a troca de processos e se há tarefas locais que podem ser migradas. Se alguma condição não for satisfeita, a política informa ao nó mestre que não é possível realizar a migração.

O próximo passo é determinar o número de processos que deverão ser migrados, e selecioná-los. O número de processos é determinado pela diferença do índice de carga do transmissor para a média de carga do ambiente. Através do índice de peso do nó, tantas

tarefas são selecionadas quantas forem necessárias para fazer o índice carga esperada do nó diminuir até a carga média do ambiente. Os processos selecionados para a migração são aqueles com maior tempo de processamento e menor tempo desde sua criação, ou seja, os processos criados mais recentemente (que causaram o desequilíbrio de carga) e que tiveram processamento mais intensivo (de forma a evitar selecionar processos ociosos, que não alteram a carga do nó).

Por fim, determina-se os receptores dos processos que serão migrados. Nesse momento, o mesmo procedimento de seleção de receptores da política de alocação inicial é utilizado, apenas com um conjunto específico de potenciais receptores, que são aqueles compatíveis para trocar processos com o nó transmissor (possuem mesma arquitetura e mesmo sistema operacional).

Da mesma forma como ocorre com a alocação inicial, a política de migração de processos utiliza os índices de peso do nó e de carga esperada para determinar os receptores dos processos migrados, e manter uma predição das alterações das cargas dos nós.

Tendo determinado os processos que serão migrados e os computadores receptores de cada processo, a política instrui o ambiente de passagem de mensagens a realizar a migração.

A Figura 5.11 mostra um exemplo de migração de processos em um *cluster* heterogêneo desbalanceado de cinco nós. Em (a), o *cluster* é representado em seu estado inicial, onde pode ser observada a carga do quinto nó muito superior às dos outros nós, sendo selecionado como transmissor. Em (b), a política de seleção no quinto nó determina que dois processos podem ser transferidos, de forma a reduzir a carga do nó para a média de carga do *cluster*. Em (c), os dois processos selecionados são transferidos para os nós com menores índices de carga esperada. O índice de carga esperada dos três nós envolvidos na transferência são corrigidos com base no índice de peso de cada nó. O estado final do *cluster* após a migração apresenta uma pequena redução na média de carga, e uma grande redução no desvio padrão, mostrando o efeito do balanceamento de carga.

Figura 5.11 - Exemplo de migração de processos em um cluster desbalanceado.

5.6.1. Componentes da política de migração

Como na política de migração JUMP, o nó com maior carga é selecionado para distribuir sua carga para outros nós do ambiente, a política é classificada como iniciada pelo transmissor.

A política de transferência ou ativação, durante a migração de processos pela política JUMP, determina que um nó é transmissor se estiver entre os mais carregados do ambiente, possuir índice de utilização de processadores próximo ou igual a 100% e pelo menos uma tarefa em execução. A política de ativação é dividida em duas partes; a primeira, centralizada, é executada no nó mestre do *cluster* para determinar os potenciais transmissores; e a segunda, distribuída, é executada no nó do *cluster* determinado como transmissor.

A política de seleção da migração de processos escolhe os processos que serão transferidos com base em seu tempo desde a criação e o tempo de processamento utilizado

pelo processo, priorizando processos criados há menos tempo e com maior tempo de processamento.

A política de localização da migração de processos é a mesma da alocação inicial, determinando os nós receptores de cada processo a ser migrado iterativamente, selecionando sempre o nó menos carregado e computando a carga prevista daquele nó baseado em seu peso.

5.6.2. Comunicação durante a migração de processos

A migração de processos é iniciada periodicamente pela instância da política JUMP em execução no nó mestre do *cluster*. Essa instância determina o primeiro provável transmissor e envia a mensagem *JUMP_MSG_TRY_MIGRATION*, com conteúdo nulo. A instância da política que recebe a mensagem (pode ser tanto o próprio mestre, quanto outra instância em execução em outro nó) verifica a viabilidade da migração de processos, determina os processos a serem transferidos e os receptores e envia uma mensagem *AM_MSG_SCHED_MIGRATE_TASKS* para o AMIGOD local, carregando a estrutura *amMsgPolicyMigration*. As informações enviadas pela política incluem o cliente que deve executar a migração, os receptores e quais processos deverão ser transmitidos para cada receptor. O AMIGOD envia a mensagem *AM_MSG_CLIENT_MIGRATE_TASKS* para o ambiente de passagem de mensagens, carregando a estrutura *amMsgMigration*, contendo os receptores e os processos que deverão ser transmitidos.

O cliente deve responder com a estrutura *amMsgMigrationReply*, contendo o resultado da migração de processos, o número de processos migrados e total, e o número de receptores aceitos e total. O AMIGOD repassa a mesma mensagem para a instância local da política JUMP no nó transmissor. A instância local responde à mensagem original enviada pelo nó mestre, carregando a estrutura *jumpMsgMigrationReply* que contém o resultado local específico da política mais os detalhes passados pelo ambiente de passagem de mensagens. As estruturas *amMsgMigration*, *amMsgMigrationReply* e *jumpMsgMigrationReply* são descritas na Figura 5.12.

<pre>enum amMsgMigrationResult { AM_MIG_SUCCESS = 0, AM_MIG_PARTIAL = 1, AM_MIG_FAILURE = 2 }; enum jumpMigrationResult { JUMP_ACCEPTED = 0, JUMP_NOT_READY = 1, JUMP_NO_MOVABLE_TASKS = 2, JUMP_NO_RECEIVER_FOUND = 3 }; struct amMsgMigrationDest { amPeer peer; unsigned int tasks<>; };</pre>	<pre>struct amMsgMigration { amMsgMigrationDest dests<>; }; struct amMsgMigrationReply { amMsgMigrationResult result; unsigned int tasks_total; unsigned int tasks_moved; unsigned int dests_total; unsigned int dests_accepted; }; struct jumpMsgMigrationReply { jumpMigrationResult result; amMsgMigrationReply detail; };</pre>
---	---

Figura 5.12 - Estruturas *amMsgMigration* e *jumpMsgMigrationReply* (XDR).

A sequência de mensagens é mostrada na Figura 5.13. Outros detalhes da comunicação durante a migração de processos são mostrados na Figura 4.9.

Figura 5.13 - Sequência de mensagens para migração de processos.

5.7. Considerações Finais

Este capítulo apresentou o desenvolvimento e a implementação da nova política de escalonamento preemptiva JUMP, sendo a primeira política deste tipo desenvolvida e implementada sobre o ambiente AMIGO.

A política não apenas demonstra a capacidade do ambiente AMIGO de dar suporte ao escalonamento preemptivo de processos, mas também implementa um algoritmo que demonstra os benefícios de unificar as políticas de alocação inicial e migração de processos em uma única política de escalonamento.

Apesar do desenvolvimento piloto da política ter sido limitado ao único ambiente de passagem de mensagens com suporte à migração de processos adaptado para ao AMIGO, o

DPVM (Iskra et al., 2000a), a sua utilização deve ser possível em outros ambientes com suporte à migração de processos, uma vez que o AMIGO foi adaptado para fornecer uma interface bastante genérica para a descrição da migração de processos.

A política apresenta características fortes na forma como trata a heterogeneidade de potência computacional dos computadores do ambiente, e na sua capacidade de se adaptar a alterações do ambiente e da aplicação distribuída.

No próximo capítulo, a avaliação e análise do desempenho da política serão apresentadas.

6. Avaliação de Desempenho e Resultados

6.1. Considerações Iniciais

Este trabalho contribuiu com uma grande adaptação do ambiente de escalonamento de processos AMIGO e a implementação de uma nova política de escalonamento com suporte à migração de processos. O maior objetivo que motiva a pesquisa em políticas de escalonamento de processos em ambientes paralelos distribuídos é permitir uma melhor utilização dos recursos disponíveis e, por consequência, um melhor desempenho das aplicações distribuídas.

Existem diversas formas de realizar o escalonamento de processos em ambientes distribuídos, utilizando diferentes ferramentas e diferentes políticas de escalonamento. Cada política ou ferramenta pode ser mais adequada para algumas situações específicas.

A avaliação de desempenho é um requisito para permitir determinar quais os benefícios práticos da utilização de determinadas ferramentas ou políticas, verificando os objetivos alcançados pelo trabalho e também servindo como guia para o usuário final da plataforma, no momento de determinar a política mais adequada.

Uma forma de determinar a boa satisfação do usuário final é através de um menor tempo de resposta da aplicação distribuída sendo executada sobre o ambiente. O menor tempo de resposta é obtido indiretamente ao otimizar a utilização dos recursos do ambiente, que é consequência direta do balanceamento da carga de trabalho, objetivo almejado pela nova política de escalonamento JUMP.

Neste capítulo serão descritos os experimentos realizados para avaliar e analisar o desempenho da nova versão do ambiente AMIGO (descrita no capítulo 4), e da nova política de escalonamento JUMP (descrita no capítulo 5).

6.2. Plataforma Utilizada

Todos os experimentos foram executados sobre um *cluster* de quatro computadores pessoais de arquitetura IBM/PC x86 32-bits. A configuração dos computadores não é

uniforme, possuindo diferenças na potência dos processadores e quantidade de memória. Portanto, trata-se de um *cluster* heterogêneo, escolhido por ser o ambiente que melhor evidencia os benefícios da migração de processos, que é o objetivo deste trabalho.

A Tabela 6.1 apresenta a configuração de cada um dos computadores utilizados. Os nós escravos não possuem disco rígido, sendo classificados como *diskless* ou *stateless*. (Malcom, 2004). *Bench Integral* e *Bench Threads* são *benchmarks* que representam aproximadamente a potência computacional de cada computador. *Bench Integral* faz parte da implementação original do AMIGO, enquanto que *Bench Threads* foi reimplementado durante este trabalho baseado no original, de forma a melhor refletir o desempenho de processadores mais capazes de gerenciar múltiplas *threads* em disputa por processamento simultaneamente.

Tabela 6.1 - Configuração do ambiente computacional utilizado na avaliação de desempenho.

Nó	Processador	Memória	Disco Rígido	Bench Integral	Bench Threads
master	Intel Pentium 4 3,00 GHz	512 MB	40 GB	3,525	4,384
n01	Intel Pentium 4 1,40 GHz	256 MB	-	2,015	2,329
n02	Intel Pentium II 350 MHz	256 MB	-	1,045	1,044
n03	Intel Pentium II 333 MHz	128 MB	-	1,000	1,000

Todos os computadores executam o sistema operacional Linux 2.4.26, com a biblioteca de sistema *glibc 2.3.2-dynckpt*, que é uma versão alterada da *glibc 2.3.2* com suporte a *checkpointing*, como parte dos requisitos do ambiente de passagem de mensagens.

O ambiente não compartilha usuários e foi especialmente preparado de forma a cada instância do sistema operacional executar apenas os serviços fundamentais e, assim, minimizar a influência de processos externos sobre a avaliação. No ambiente foram instalados a nova versão do AMIGO, o ambiente de passagem de mensagens DPVM 2.0 (Iskra et al., 2000a), as políticas e as aplicações utilizadas para a avaliação de desempenho.

Os computadores foram interconectados através de uma rede *FastEthernet* (100Mbps), com uma *switch* de mesma tecnologia, com comunicação *full-duplex*. O sistema operacional opera em pilha de protocolos dupla para a rede, suportando IPv4 e IPv6.

6.3. Aplicações Utilizadas

Como a política de escalonamento desenvolvida neste trabalho visa a otimizar o desempenho de aplicações que demandam processamento intensivo, foram escolhidas aplicações que tivessem essa mesma característica para a avaliação de desempenho.

Mesmo entre as aplicações de processamento intensivo (*CPU-bound*), cada aplicação pode apresentar perfil de execução e comportamento que afeta a qualidade do escalonamento de processos. Por essa razão, foram escolhidas três aplicações paralelas distribuídas, com diferentes perfis de execução.

6.3.1. Método dos trapézios compostos

Essa é uma aplicação de cálculo numérico, que utiliza a propriedade das integrais onde dada uma função $f(x)$ qualquer, integrável entre os pontos de a até b , a integral dessa função pode ser aproximada para a área de um trapézio, com vértices sobre o eixo de integração e nos extremos de integração.

$$\int_a^b f(x)dx \approx (b-a) \frac{f(a)+f(b)}{2}$$

Outra propriedade utilizada é que dado um ponto c , tal que $a < c < b$, a função é integrável entre os pontos a e c , e entre c e b ; e a soma dessas integrais é igual à integral da função entre os pontos de a até b (Táboas, 2002).

$$\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx$$

Dessa forma, dividindo-se o intervalo de integração em um grande número de subintervalos, calculando-se as integrais desses subintervalos através da área do trapézio e somando-se esses resultados, obtemos uma aproximação muito boa para a integral da função entre os pontos a e b .

$$\int_a^b f(x)dx \approx \frac{b-a}{n} \left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f\left(a+k \frac{b-a}{n}\right) \right)$$

A aplicação usada para o teste foi desenvolvida durante o trabalho de Cortés (1999), funciona através do modelo de mestre e escravo, onde o mestre divide o intervalo de integração em partições e distribui para os escravos, que calculam o valor de cada partição. O mestre soma os resultados parciais recebidos de cada escravo e retorna o valor final.

Essa aplicação exige processamento intensivo com cálculos em ponto flutuante, e uma quantidade muito baixa de comunicações. Uma descrição mais detalhada da aplicação pode ser encontrada no trabalho de Cortés (1999).

6.3.2. Multiplicação de matrizes

Essa é também uma aplicação de cálculo numérico, que realiza o cálculo paralelo do produto de duas matrizes de alta ordem. O algoritmo trivial para o cálculo de matrizes possui ordem de complexidade $O(n^3)$, o que o torna um algoritmo de processamento intensivo e particularmente demorado para matrizes de alta ordem.

A paralelização do algoritmo utiliza a redução de uma das matrizes em um conjunto de vetores, que podem ser calculados independentemente. A definição formal da multiplicação de matrizes é:

$$A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times p} \Rightarrow AB \in \mathbb{R}^{m \times p}$$

$$(AB)_{i,j} = \sum_{k=0}^n A_{i,k} B_{k,j}$$

Cada coluna da matriz B é tratada como um vetor, que é multiplicado pela matriz A, resultando em um novo vetor. A matriz resultante consiste na concatenação dos vetores resultantes.

$$B'_i = [B_{1,i} \ B_{2,i} \ \cdots \ B_{p,i}]$$

$$AB = [AB'_1 \ AB'_2 \ \cdots \ AB'_n]$$

A aplicação usada para o teste é uma adaptação da aplicação original desenvolvida por Cortés (1999), funciona através do modelo de mestre e escravo, onde o mestre carrega as matrizes, divide a matriz B em conjuntos de colunas, e para cada escravo envia a matriz A e um conjunto de colunas da matriz B, que realiza o cálculo. O mestre concatena os resultados de recebidos de cada escravo em uma nova matriz e retorna a matriz final.

Assim como o método dos trapézios compostos, essa aplicação exige processamento intensivo com cálculos em ponto flutuante, porém possui uma quantidade maior de comunicações.

6.3.3. Desenho de fractais Mandelbrot

O *jrMandel* (Ravasi, 2008) é uma aplicação de código fonte aberto, escrita em linguagem C++, para o desenho de fractais do conjunto de Mandelbrot em alta resolução, utilizando processamento em paralelo. Foi desenvolvida pelo autor desta dissertação durante o curso de mestrado e sua utilização foi oportunamente aproveitada neste trabalho.

O conjunto de Mandelbrot é um conjunto de pontos do plano complexo onde a sequência definida abaixo é limitada e não tende ao infinito:

$$\begin{aligned}Z_0 &= 0 \\ Z_{i+1} &= Z_i^2 - c\end{aligned}$$

O desenho de um fractal do conjunto de Mandelbrot é obtido iterando sobre cada ponto do plano até que um limite seja atingido ou uma condição de escape seja satisfeita. Caso o limite de iterações seja atingido, o ponto é considerado como tendendo ao infinito; caso a condição de escape seja satisfeita, a sequência é considerada limitada, e o número de iterações necessárias é utilizado para determinar o valor (cor) daquele ponto.

O desenho de fractais do conjunto de Mandelbrot é altamente paralelizável, pois o cálculo do valor de cada ponto não possui dependência de nenhum outro cálculo ou informação da tarefa.

A aplicação jrMandel possui três modos de operação: independente, paralelizado através de *threads*, e paralelizado através de PVM. O terceiro modo foi utilizado. Seu modelo de programação paralela é bastante diferente das outras duas aplicações, o que motivou sua inclusão entre as aplicações avaliadas. Ao invés de um processo mestre que escalona o trabalho para diversos escravos, cada processo tem autonomia para determinar a necessidade e viabilidade de dividir a sua parte do cálculo para outros processos.

Cada processo (a iniciar pelo próprio processo mestre) analisa a região do plano que deve ser desenhada e faz uma estimativa do número de cálculos que serão necessários. Se um determinado limiar for atingido, o processo subdivide a sua região do desenho e escalona novos processos para atender a cada uma das regiões. Assim, cada processo é ao mesmo tempo mestre e escravo, formando uma árvore. Regiões mais complexas do fractal causam a utilização de um maior número de processos para a sua produção.

Ao contrário das outras duas aplicações, onde número de processos pode ser determinado no momento de sua execução e o escalonamento ocorre apenas no nó onde a aplicação foi iniciada, no jrMandel o número de processos é determinado em tempo de execução, e todos os nós do *cluster* tem a oportunidade de realizar o escalonamento. A imagem final gerada pela aplicação é a concatenação das diversas regiões calculadas por diferentes processos. A Figura 6.1 é um exemplo de imagem gerada pela aplicação. Cada retângulo demarca a região da figura desenhada por um processo distribuído.

Figura 6.1 - Fractal produzido pelo jrMandel, mostrando as divisões de tarefas.

6.4. Análise Estatística Considerada

Os resultados obtidos através dos experimentos realizados devem ser analisados através de uma técnica de inferência estatística, de forma a determinar se as diferenças nos tempos de resposta observadas para diferentes níveis de cada fator são estatisticamente significantes. Uma técnica de inferência estatística é o Teste de Hipóteses (Magalhães e Lima, 2002), que foi utilizado neste trabalho.

Para cada resultado a ser verificado, duas hipóteses são formuladas: uma hipótese H_0 , chamada de hipótese nula (ou nulidade) e uma hipótese H_1 , chamada de hipótese alternativa. A hipótese nula geralmente tenta contrariar os valores obtidos pelo experimento. Quando a hipótese nula é rejeitada, a hipótese alternativa (que concorda com os valores obtidos) é então aceita.

Ao realizar um teste de hipóteses, dois tipos de erros podem ser cometidos: rejeitar a hipótese nula quando ela é verdadeira (erro do tipo I) e aceitar a hipótese nula quando ela é falsa (erro do tipo II). A probabilidade de rejeitar a hipótese nula quando ela é verdadeira é definida pelo nível de significância (α) do teste. Como esse erro é o mais importante a ser evitado, define-se um valor pequeno para o nível de significância.

O nível de significância utilizado neste trabalho é de 0,05, que significa que a

hipótese nula será corretamente aceita quando for verdadeira com uma probabilidade de 95%.

O teste de hipótese utilizado para verificar os resultados dos experimentos é a comparação de médias para duas populações, para amostras independentes. O teste verifica se a diferença da média dos tempos de resposta de duas amostras (com diferentes níveis para um ou mais fatores) pode ser considerada significativa, permitindo concluir que há de fato uma diferença no tempo de resposta. O teste utilizado é classificado como bilateral, e as hipóteses formuladas são:

$$\begin{aligned} H_0: \mu_X &= \mu_Y \\ H_1: \mu_X &\neq \mu_Y \end{aligned}$$

Para experimentos com amostras de tamanho suficientemente grande (30 execuções), a variância amostral pode ser utilizada como um bom estimador para a variância populacional (Magalhães e Lima, 2002; Achcar e Rodrigues, 2000), permitindo a utilização do teste para variâncias conhecidas. Considerando-se que os tempos de resposta seguem o modelo Normal, a estatística do teste é dada por:

$$Z = \frac{\bar{X}_X - \bar{X}_Y}{\sqrt{\frac{S_X^2}{n_X} + \frac{S_Y^2}{n_Y}}} \sim N(0, 1)$$

Onde $\bar{X}_X$ e $\bar{X}_Y$ representam as médias amostrais dos tempos de resposta obtidos em cada cenário, S_X^2 e S_Y^2 representam as variâncias amostrais e n_X e n_Y representam os tamanhos de cada amostra.

Para um nível de significância $\alpha = 0,05$, delimita-se a região de aceitação de H_0 utilizando $z_{(1-\alpha)/2} = z_{0,475} = 1,96$, fornecido pela tabela da distribuição Normal padrão. Ou seja, a hipótese nula é aceita se Z calculado estiver entre $-1,96$ e $1,96$; caso contrário ($Z \leq -1,96$ ou $Z \geq 1,96$) a hipótese nula é rejeitada e a hipótese alternativa é aceita.

Para experimentos com amostras de tamanho pequeno (10 execuções), o *t-test* de *Student* para amostras independentes com variâncias desconhecidas e diferentes é utilizado. A estatística desse teste é dada por:

$$T = \frac{\bar{X}_X - \bar{X}_Y}{\sqrt{\frac{S_X^2}{n_X} + \frac{S_Y^2}{n_Y}}} \sim t_{(v)}$$

A região de aceitação de H_0 para T depende do número de graus de liberdade para a distribuição *t* de *Student* (representado por v) que por sua vez depende das variâncias

amostrais e portanto não é fixo para todos os testes. Ao invés de definir a região de aceitação de T , é calculado e apresentado para cada teste o resultado do t -test, que representa o nível de significância necessário para que a hipótese nula seja aceita. Portanto, H_0 é aceita apenas se o resultado do t -test for maior ou igual a 0,05.

Outra técnica de inferência estatística utilizada é a estimação por intervalo de confiança. O intervalo de confiança é representado graficamente nas figuras por barras sobre e sob cada valor médio, e é dado por:

$$IC(\mu, \gamma) = \left[\bar{X} - t_{\gamma/2} \frac{S}{\sqrt{n}} ; \bar{X} + t_{\gamma/2} \frac{S}{\sqrt{n}} \right]$$

Onde γ representa o coeficiente de confiança e S representa o desvio-padrão amostral. O coeficiente de confiança utilizado é $\gamma = (1 - \alpha) = 0,95$, que equivale à significância escolhida para o teste de hipóteses. $t_{\gamma/2}$ é fornecido pela tabela da distribuição t de Student com $n - 1$ graus de liberdade. Para os experimentos com amostras de tamanho igual a 30, $t_{0,475} = 2,045$; para amostras de tamanho igual a 10, $t_{0,475} = 2,262$.

6.5. Experimentos Realizados

Para a avaliação de desempenho do ambiente AMIGO e da política JUMP, há diversos conjuntos de fatores que podem ser utilizados na comparação, como: a utilização ou não do ambiente AMIGO, a política de escalonamento utilizada, a distribuição prévia de carga do ambiente, as aplicações executadas, a variação de parâmetros de cada aplicação, etc.

Como a abordagem de todas as combinações de conjuntos de fatores resultaria em um problema multidimensional, os experimentos realizados procuraram cobrir algumas dessas combinações. A aplicação distribuída é um fator comum a todos os experimentos.

O primeiro fator considerado foi a presença do AMIGO como escalonador de processos para o ambiente paralelo distribuído. Analisando o tempo de resposta das aplicações distribuídas com e sem a utilização do AMIGO, procurou-se determinar qual a sobrecarga imposta pelo ambiente sobre o desempenho das aplicações.

Em seguida, foram considerados como fatores as políticas de escalonamento utilizadas e o nível de carga do sistema. Nesses experimentos, o desempenho da política JUMP é comparado com as outras duas políticas implementadas no AMIGO, para três diferentes situações de carga do sistema.

Por fim, o fator da carga de trabalho foi considerado. Quatro diferentes cargas de

trabalho foram utilizadas para cada uma das aplicações, comparando a eficiência da política JUMP quando utilizada para diferentes cargas de trabalho.

6.5.1. Sobrecarga imposta pelo ambiente AMIGO

Em trabalhos passados, pela falta da implementação de uma política de escalonamento *round-robin* no ambiente AMIGO, as avaliações de desempenho se limitaram a comparar o desempenho dos ambientes de passagem de mensagens com seus escalonamentos padrões (em geral baseado na política *round-robin*) com a política DPWP (Araújo, 1999). Tais trabalhos mostram o melhor desempenho da política DPWP em diversas situações.

Nesses trabalhos, as avaliações de desempenho na verdade comparam dois fatores que são alterados conjuntamente em todos os experimentos: a presença do ambiente AMIGO, e a utilização da política DPWP. Não era possível determinar o efeito isolado da política DPWP, uma vez que ela fora construída para executar sobre o ambiente AMIGO; tão pouco o efeito isolado do ambiente AMIGO, uma vez que este não possuía uma política semelhante à do ambiente de passagem de mensagens.

Neste trabalho, a política de escalonamento *round-robin*, a mesma utilizada nos ambientes PVM e DPVM entre outros, foi implementada sobre a nova versão do ambiente AMIGO. Com isso, agora é possível determinar qual o efeito do ambiente AMIGO sobre o desempenho de aplicações distribuídas.

Nesse experimento, o desempenho do PVM 3.4.5 sozinho (PVM-RR) é comparado com o desempenho do mesmo PVM 3.4.5 unido ao AMIGO, executando a política de escalonamento *round-robin* (AMIGO-RR) sobre a plataforma ociosa. As configurações utilizadas para cada aplicação são mostradas na Tabela 6.2.

Tabela 6.2 - Parâmetros das aplicações utilizadas no experimento.

Aplicação	Parâmetros
<i>Integral</i>	processos: 12; iterações: 10.000.000; repetições: 4
<i>Matriz</i>	processos: 12; matrizes: 768×768
<i>jrMandel</i>	centro: -0,251786 +0,639560i; aproximação: 10; resolução: 4096x3072; iterações: 128

As médias dos tempos de resposta de 30 execuções de cada aplicação, sobre cada ambiente, são mostradas na Tabela 6.3 e na Figura 6.2 com IC(μ , 95%).

Tabela 6.3 - Resultados da comparação do PVM-RR com o AMIGO-RR.

Aplicação	PVM-RR		AMIGO-RR	
	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}
Integral	11,266	0,008	11,340	0,039
Matriz	20,973	0,887	20,994	0,830
jrMandel	28,700	0,081	28,747	0,059

Figura 6.2 - Resultados da comparação do PVM-RR com o AMIGO-RR.

Os testes de hipóteses para os resultados obtidos, apresentados na Tabela 6.4, mostram que a utilização do AMIGO impõe uma sobrecarga para duas das três aplicações verificadas (Integral e jrMandel). Contudo, a diferença entre as médias dos tempos de resposta para cada uma das aplicações é bem pequena, como pode ser observado na Tabela 6.5. Essa diferença é esperada, uma vez que o ambiente precisa se comunicar com o AMIGO antes de criar novos processos. Considera-se que os benefícios da utilização da plataforma AMIGO, em especial a possibilidade de utilização de outras políticas de escalonamento mais eficientes, compensam pela sobrecarga causada pela sua utilização.

Tabela 6.4 - Testes de hipóteses para PVM-RR × AMIGO-RR.

	Integral		Matriz		jrMandel	
	PVM-RR	AMIGO-RR	PVM-RR	AMIGO-RR	PVM-RR	AMIGO-RR
Média	11,266	11,341	20,973	20,994	28,700	28,747
Variância	0,000	0,002	0,787	0,689	0,007	0,004
Tamanho	30	30	30	30	30	30
Teste (Z)	10,174		0,094		2,566	
Conclusão	$H_1: \mu_{PVM-RR} \neq \mu_{AMIGO-RR}$		$H_0: \mu_{PVM-RR} = \mu_{AMIGO-RR}$		$H_1: \mu_{PVM-RR} \neq \mu_{AMIGO-RR}$	

Tabela 6.5 - Diferenças dos tempos de resposta para PVM-RR × AMIGO-RR.

	Integral		Matriz		jrMandel	
	PVM-RR	AMIGO-RR	PVM-RR	AMIGO-RR	PVM-RR	AMIGO-RR
Média	11,266	11,341	20,973	20,994	28,700	28,747
Dif. Absol.	0,075		0,021		0,047	
Dif. Relat.	0,66%		0,10%		0,16%	

6.5.2. Desempenho das políticas em um sistema ocioso

Nesse experimento, a política de escalonamento JUMP é comparada com outras políticas em um sistema ocioso, ou seja, a única aplicação em execução no sistema é a própria aplicação sendo avaliada. O objetivo desse experimento não é comparar o desempenho das diferentes políticas com diferentes cargas de trabalho, que será visto na Seção 6.5.5 para a política JUMP, e já foi visto em trabalhos anteriores (Araújo, 1999; Figueiredo, 2000) para as outras políticas.

Os parâmetros utilizados neste experimento, mostrados na Tabela 6.6, foram escolhidos de forma que cada execução de cada aplicação tivesse um tempo de resposta de aproximadamente 60 segundos quando a política de escalonamento *round-robin* era utilizada, servindo como referência para determinar o ganho de desempenho obtido pela utilização das outras políticas de escalonamento.

Tabela 6.6 - Parâmetros das aplicações utilizadas no experimento.

Aplicação	Parâmetros
Integral	processos: 16; iterações: 50.000.000; repetições: 4
Matriz	processos: 16; matrizes: 768×768; repetições: 4
jrMandel	centro: -0,251786 +0,639560i; aproximação: 10; resolução: 2560×1920; iterações: 1024

Cada uma das três aplicações foram executadas 30 vezes para cada uma das políticas de escalonamento. As médias dos tempos de resposta das aplicações são mostradas na Tabela 6.7 e na Figura 6.3 com IC(μ , 95%).

Tabela 6.7 - Resultados da comparação das políticas RR, DPWP e JUMP, sistema ocioso.

Aplicação	AMIGO-RR		AMIGO-DPWP		AMIGO-JUMP	
	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}
Integral	55,804	0,071	55,849	0,073	39,210	7,524
Matriz	73,692	4,030	62,523	4,286	42,159	10,842
jrMandel	54,659	0,017	51,350	1,274	32,767	8,963

Figura 6.3 - Resultados da comparação das políticas RR, DPWP e JUMP, sistema ocioso.

Observa-se uma grande variância (representada pelo desvio padrão na tabela e no gráfico) nos tempos de resposta da política JUMP. Dois fatores são refletidos nessa variação. Primeiro, a grande dinamicidade da política de escalonamento, onde cada distribuição de carga realizada é um reflexo de diversas informações colhidas anteriormente e, segundo, a capacidade da política de “aprender” a heterogeneidade do sistema durante a sua execução. A Figura 6.4 mostra os tempos de resposta de cada execução individual para cada aplicação ao longo das 10 primeiras execuções do experimento.

Figura 6.4 - Execuções individuais para as aplicações sob controle da política JUMP.

As primeiras execuções da aplicação sobre o ambiente controlado pela política JUMP tendem a apresentar resultados não tão bons, pois a política ainda não possui

informações suficientes para determinar o reflexo dos processos criados sobre a carga de processamento do sistema. Execuções subsequentes possuem um tempo de resposta melhor.

Como grande parte da variância observada é devida às primeiras execuções da aplicação, optou-se por reavaliar o desempenho da política utilizando a técnica de *warm-up*, ou seja, efetuar algumas execuções adicionais de cada aplicação no começo de cada teste antes das execuções efetivamente medidas. Tal técnica não é necessária para as outras duas políticas por terem um comportamento estático (no caso da política *round-robin*) ou determinístico (no caso da política DPWP), que é evidenciado pela baixa variância observada. A Tabela 6.8 e a Figura 6.5 mostram os resultados obtidos.

Tabela 6.8 - Resultados da comparação das políticas RR, DPWP e JUMP (*warm-up*), sistema ocioso.

	AMIGO-RR		AMIGO-DPWP		AMIGO-JUMP (com <i>warm-up</i>)	
	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}
Integral	55,804	0,071	55,849	0,073	37,696	5,041
Matriz	73,692	4,030	62,523	4,286	30,876	3,307
jrMandel	54,659	0,017	51,350	1,274	31,546	5,629

Figura 6.5 - Resultados da comparação das políticas RR, DPWP e JUMP (*warm-up*), sistema ocioso.

Nota-se que tanto a média quanto a variância dos resultados da política JUMP são menores do que sem a utilização da técnica de *warm-up*. Apesar do resultado mais favorável, o usuário da política deve estar ciente de que a política JUMP depende desse período de aquecimento, onde a heterogeneidade do ambiente é detectada, para oferecer melhores resultados.

Ainda há uma variância observável no tempo de resposta das aplicações para a política JUMP, que é atribuído ao sincronismo das aplicações com a migração de processos. A migração de processos é ativada periodicamente com uma frequência de uma vez a cada minuto. Quando a migração de processos ocorre, a aplicação pode estar próximo do começo de sua execução (e ser muito beneficiada com o balanceamento de carga em seu tempo de resposta) ou próximo do final (e ser virtualmente irrelevante, ou até prejudicial para o tempo de resposta).

Observa-se que nessa situação a política de escalonamento DPWP oferece um pequeno ganho de desempenho apenas para as aplicações Matriz e jrMandel, que possui um perfil de execução mais dinâmico. Esse resultado é esperado, uma vez que a política DPWP distribui a carga durante a alocação inicial, onde o *cluster* se encontra totalmente ou quase totalmente ocioso, e portando distribuindo os processos igualmente para todos os computadores.

A política JUMP demonstra um significativo ganho de desempenho comparado às duas outras políticas, conforme é verificado através dos testes de hipóteses apresentados na Tabela 6.9.

Tabela 6.9 - Testes de hipóteses para $RR \times JUMP$ e $DPWP \times JUMP$, sistema ocioso.

	<i>Integral</i>			<i>Matriz</i>			<i>jrMandel</i>		
	<i>RR</i>	<i>DPWP</i>	<i>JUMP</i>	<i>RR</i>	<i>DPWP</i>	<i>JUMP</i>	<i>RR</i>	<i>DPWP</i>	<i>JUMP</i>
<i>Média</i>	55,804	55,849	37,696	73,692	62,523	30,876	54,659	51,350	31,546
<i>Variância</i>	0,005	0,005	25,516	16,244	18,371	10,935	0,000	1,623	31,682
<i>Tamanho</i>	30	30	30	30	30	30	30	30	30
	<i>RR JUMP</i>	<i>DPWP JUMP</i>			<i>RR JUMP</i>	<i>DPWP JUMP</i>	<i>RR JUMP</i>	<i>DPWP JUMP</i>	
<i>Teste (Z)</i>	19,671	19,720	44,983		32,020		22,491		18,796
<i>Conclusão</i>	$H_1: \mu_{RR} \neq \mu_{JUMP}$	$H_1: \mu_{DPWP} \neq \mu_{JUMP}$	$H_1: \mu_{RR} \neq \mu_{JUMP}$		$H_1: \mu_{DPWP} \neq \mu_{JUMP}$		$H_1: \mu_{RR} \neq \mu_{JUMP}$		$H_1: \mu_{DPWP} \neq \mu_{JUMP}$

Todos os próximos experimentos consideram a utilização da técnica de *warm-up* para a política JUMP de forma a reduzir a influência da detecção da heterogeneidade do *cluster* nos resultados da aplicação.

6.5.3. Desempenho das políticas em um sistema carregado

Nesse experimento, a política de escalonamento JUMP é comparada com outras políticas em um sistema previamente carregado. Os mesmos testes realizados no experimento anterior, com os mesmos parâmetros para cada aplicação, são realizados sobre esse novo

cenário. Para a política JUMP, os resultados com a utilização da técnica de *warm-up* são considerados.

Antes da execução de cada teste desse experimento, cada computador é carregado com três processos de “peso-morto”, não gerenciados pelo ambiente de passagem de mensagens. Cada processo permanece constantemente ocupando o processador até o final do teste.

A Tabela 6.10 e a Error: Reference source not found mostram as médias dos tempos de resposta das aplicações executadas sobre este cenário.

Tabela 6.10 - Resultados da comparação das políticas RR, DPWP e JUMP, sistema carregado.

	AMIGO-RR		AMIGO-DPWP		AMIGO-JUMP	
	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}
Integral	84,365	0,106	62,844	0,603	48,408	6,793
Matriz	110,031	5,650	45,256	0,706	49,939	7,962
jrMandel	77,823	0,280	47,850	0,483	42,479	5,445

Figura 6.6 - Resultados da comparação das políticas RR, DPWP e JUMP, sistema carregado.

Os testes de hipóteses apresentados na Tabela 6.11 confirmam que política de escalonamento JUMP fornece tempos de resposta diferentes das outras duas políticas em todos os casos.

Tabela 6.11 - Testes de hipóteses para $RR \times JUMP$ e $DPWP \times JUMP$, sistema carregado.

	Integral			Matriz			jrMandel		
	RR	DPWP	JUMP	RR	DPWP	JUMP	RR	DPWP	JUMP
Média	84,365	62,844	48,408	110,031	45,256	49,939	77,823	47,850	42,479
Variância	0,011	0,364	46,144	31,821	0,498	63,399	0,078	0,233	29,652
Tamanho	30	30	30	30	30	30	30	30	30
	RR JUMP		DPWP JUMP		RR JUMP		DPWP JUMP		
Teste (Z)	28,989		11,594		33,712		-3,209		5,381
Conclusão	$H_1: \mu_{RR} \neq \mu_{JUMP}$		$H_1: \mu_{DPWP} \neq \mu_{JUMP}$		$H_1: \mu_{RR} \neq \mu_{JUMP}$		$H_1: \mu_{DPWP} \neq \mu_{JUMP}$		$H_1: \mu_{DPWP} \neq \mu_{JUMP}$

Os resultados desse experimento são particularmente interessantes. Em primeiro lugar, o melhor desempenho da política de escalonamento DPWP comparada à *round-robin*, quando utilizada em sistemas carregados, já verificado em trabalhos anteriores (Araújo, 1999; Figueiredo, 2000) é mais uma vez confirmada. A política JUMP também mostrou um desempenho melhor que a política *round-robin* em todos os casos, e um desempenho melhor que a política DPWP para as aplicações Integral e jrMandel. Curiosamente, para a aplicação Matriz, o desempenho da política JUMP não reflete os mesmos ganhos vistos para as outras duas aplicações, tendo um desempenho inferior ao da política DPWP. Mesmo repetindo o experimento, o resultado se confirma.

Observando-se os tempos de resposta da aplicação Matriz no experimento anterior (Tabela 6.8, Figura 6.5), vê-se que sua variância é sempre maior do que das outras duas aplicações. Nesse experimento, exceto para a política DPWP, sua variância também é maior do que para as outras aplicações. Essa variação parece estar relacionada à característica de comunicação da aplicação, uma vez que ela é observada em ambos os experimentos. O número de cálculos realizados por cada processo escravo é determinístico e fixo, quando os processos escravos terminam a multiplicação de suas porções das matrizes, estes enviam uma grande quantidade de dados de volta para o processo mestre. Processos que terminam seus cálculos aproximadamente ao mesmo tempo irão competir pelo canal de comunicações para enviar as informações de volta ao processo mestre. Portanto, suspeita-se que a variação seja devida a variáveis desconhecidas e não controladas dos equipamentos de rede utilizados.

Observou-se durante o experimento que a política DPWP distribuiu grande parte dos processos da aplicação Matriz para o computador com maior potência computacional do *cluster*, que também é o computador onde o processo mestre foi executado, reduzindo drasticamente a utilização da rede e encurtando a comunicação entre processos. Já a política JUMP esforçou-se para manter a carga da plataforma equilibrada, o que faz com que todos os

processos da aplicação Matriz a terminarem sua execução aproximadamente ao mesmo tempo, causando um pico de utilização da rede e conseqüentemente elevando o tempo de resposta da aplicação.

Essa é uma clara evidência de que há situações onde um grande balanceamento de carga resulta em um tempo de resposta pior. A política JUMP foi desenvolvida visando o balanceamento de carga de processamento, sendo mais apropriada para aplicações de uso intensivo de processamento (*CPU-bound*). Apesar da aplicação Matriz ter essa característica, ela também possui uma característica de uso intensivo de comunicação (*network-bound*) em determinados momentos de sua execução.

6.5.4. Desempenho das políticas em um sistema com carga variável

Nesse experimento, a política de escalonamento JUMP é comparada com outras políticas em um sistema cuja carga varia durante a execução da aplicação paralela. Os mesmos testes realizados nos experimentos anteriores, com os mesmos parâmetros para cada aplicação, são realizados sobre esse novo cenário.

Durante a execução de cada teste deste experimento, dois computadores da plataforma são carregados com processos de “peso-morto”, não gerenciados pelo ambiente de passagem de mensagens.

A Tabela 6.12 e a Figura 6.8 mostram as médias dos tempos de resposta das aplicações executadas sobre esse cenário.

Tabela 6.12 - Resultados da comparação das políticas RR, DPWP e JUMP, sistema com carga variável.

	AMIGO-RR		AMIGO-DPWP		AMIGO-JUMP	
	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}
Integral	86,479	0,172	86,521	0,170	47,789	4,789
Matriz	101,534	10,467	91,747	4,082	50,468	7,621
jrMandel	69,264	1,289	63,743	1,417	36,533	5,628

Figura 6.7 - Resultados da comparação das políticas RR, DPWP e JUMP, sistema com carga variável.

Os testes de hipóteses apresentados na Tabela 6.13 confirmam que a política de escalonamento JUMP fornece tempos de resposta diferentes das outras duas políticas em todos os casos.

Tabela 6.13 - Testes de hipóteses para $RR \times JUMP$ e $DPWP \times JUMP$, sistema com carga variável.

	Integral			Matriz			jrMandel		
	RR	DPWP	JUMP	RR	DPWP	JUMP	RR	DPWP	JUMP
Média	86,479	86,521	47,789	101,534	91,747	50,468	69,264	63,743	36,533
Variância	0,030	0,029	22,936	109,559	16,663	58,078	1,662	2,008	31,679
Tamanho	30	30	30	30	30	30	30	30	30
	RR JUMP		DPWP JUMP	RR JUMP		DPWP JUMP	RR JUMP		DPWP JUMP
Teste (Z)	44,221		44,269	21,603		26,152	31,048		25,678
Conclusão	H1: $\mu_{RR} \neq \mu_{JUMP}$		H1: $\mu_{DPWP} \neq \mu_{JUMP}$	H1: $\mu_{RR} \neq \mu_{JUMP}$		H1: $\mu_{DPWP} \neq \mu_{JUMP}$	H1: $\mu_{RR} \neq \mu_{JUMP}$		H1: $\mu_{DPWP} \neq \mu_{JUMP}$

Os resultados deste experimento evidenciam exatamente a melhor situação onde uma política de escalonamento com suporte à migração de processos é mais eficiente. Os resultados obtidos concordam com os resultados apresentados no trabalho de Furquim (2006). A política DPWP apresenta um resultado próximo ao da política *round-robin*, uma vez que a política não suporta preempção, decisões de alocação inicial são baseadas no estado inicial do *cluster* (ocioso), e perduram até o final da execução da aplicação. Já a política JUMP apresenta um excelente desempenho em todos os casos, uma vez que a política é capaz de detectar o desbalanceamento de carga e reequilibrar a carga do ambiente dinamicamente.

Nos últimos três experimentos, um problema com três dimensões foi abordado: a

aplicação, a política de escalonamento e o estado de carga do ambiente. Cada experimento mostrou os resultados no plano perpendicular ao estado de carga do ambiente. Para conveniência, a Tabela 6.14 e a Figura 6.8 comparam os resultados obtidos pela política de escalonamento JUMP nos últimos três experimentos, mostrando os resultados no plano perpendicular à política de escalonamento.

Tabela 6.14 - Comparação dos resultados para sistemas ocioso, carregado e com carga variável, para a política JUMP.

AMIGO JUMP	Ocioso		Carregado		Variável	
	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}
Integral	37,696	5,041	48,408	6,793	47,789	4,789
Matriz	30,876	3,307	49,939	7,962	50,468	7,621
jrMandel	31,546	5,629	42,479	5,445	36,533	5,628

Figura 6.8 - Comparação dos resultados para sistemas ocioso, carregado e com carga variável, para a política JUMP.

6.5.5. Desempenho da política JUMP para diferentes cargas

Os experimentos anteriores mostraram o desempenho da política JUMP quando comparada às outras políticas em sistemas com diferentes perfis de carga, para diferentes aplicações. Nesse experimento, a política JUMP é comparada apenas com a política *round-robin*, para diferentes cargas de trabalho de cada aplicação. O objetivo é medir o ganho em desempenho obtido pelo balanceamento de carga oferecido pela política JUMP em relação ao sistema sem balanceamento de carga.

Cada aplicação foi executada com quatro diferentes cargas de trabalho, relacionadas na Tabela 6.15. Os testes foram realizados sobre o sistema ocioso, sem influência de cargas

externas. Em cada teste, a aplicação foi executada 10 vezes com cada carga de trabalho e com cada política de escalonamento, e a média dos tempos de resposta é apresentada.

Tabela 6.15 - Cargas utilizadas para cada aplicação no experimento.

Aplicação	Variável	Valores			
Integral	iterações	25M	50M	100M	200M
Matriz	dimensões	384×384	512×512	768×768	1024×1024
jrMandel	iterações	128	256	512	1024

6.5.5.1. Método dos trapézios compostos (aplicação Integral)

A Tabela 6.16 e a Figura 6.9 mostram as médias dos tempos de resposta da aplicação Integral com diferentes cargas de trabalho e políticas de escalonamento, e o ganho de desempenho quando a política JUMP é utilizada. Os parâmetros utilizados foram: 16 processos e número de iterações variável.

Tabela 6.16 - Ganho em desempenho da política JUMP para a aplicação Integral.

Integral	AMIGO-RR		AMIGO-JUMP		Ganho (%)
Iterações	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}	
25M	7,195	0,054	5,369	0,778	25,38
50M	14,172	0,038	8,422	1,158	40,57
100M	28,023	0,038	17,320	2,283	38,19
200M	55,795	0,062	30,475	3,417	45,38

Figura 6.9 - Tempos de resposta em relação à carga para a aplicação Integral.

Os testes de hipóteses para as diferenças dos tempos de resposta para a aplicação Integral são apresentados na Tabela 6.17. O *t-test* de *Student* é utilizado devido ao pequeno tamanho das amostras.

Tabela 6.17 - Testes de hipóteses para a aplicação Integral com diferentes cargas.

	Integral							
	25M		50M		100M		200M	
	RR	JUMP	RR	JUMP	RR	JUMP	RR	JUMP
Média	7,196	5,370	14,172	8,423	28,024	17,320	55,795	30,475
Variância	0,003	0,606	0,001	1,343	0,001	5,216	0,004	11,679
Tamanho	10	10	10	10	10	10	10	10
Teste (T)	7,401		15,679		14,818		23,425	
t-test	0,000		0,000		0,000		0,000	
Conclusão	$H_1: \mu_{RR} \neq \mu_{JUMP}$		$H_1: \mu_{DPWP} \neq \mu_{JUMP}$		$H_1: \mu_{RR} \neq \mu_{JUMP}$		$H_1: \mu_{DPWP} \neq \mu_{JUMP}$	

6.5.5.2. Multiplicação de matrizes (aplicação Matriz)

A Tabela 6.18 e a Figura 6.10 mostram as médias dos tempos de resposta da aplicação Matriz com diferentes cargas de trabalho e políticas de escalonamento, e o ganho de desempenho quando a política JUMP é utilizada. Os parâmetros utilizados foram: 16 processos e dimensões das matrizes variáveis.

Tabela 6.18 - Ganho em desempenho da política JUMP para a aplicação Matriz.

Matriz	AMIGO-RR		AMIGO-JUMP		Ganho (%)
Dimensões	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}	
384×384	3,288	0,042	3,074	0,176	6,51
512×512	6,583	0,329	6,157	0,278	6,47
768×768	19,761	0,635	15,598	1,047	21,07
1024×1024	57,553	0,667	44,341	3,603	22,96

Figura 6.10 - Tempos de resposta em relação à carga para a aplicação Matriz.

Os testes de hipóteses para as diferenças dos tempos de resposta para a aplicação Matriz são apresentados na Tabela 6.19. O *t-test* de Student é utilizado devido ao pequeno tamanho das amostras.

Tabela 6.19 - Testes de hipóteses para a aplicação Matriz com diferentes cargas.

	Matriz							
	384x384		512x512		768x768		1024x1024	
	RR	JUMP	RR	JUMP	RR	JUMP	RR	JUMP
Média	3,289	3,075	6,584	6,158	19,761	15,598	57,553	44,342
Variância	0,002	0,031	0,108	0,078	0,404	1,096	0,446	12,989
Tamanho	10	10	10	10	10	10	10	10
Teste (T)	3,726		3,120		10,746		11,398	
t-test	0,004		0,006		0,000		0,000	
Conclusão	$H_1: \mu_{RR} \neq \mu_{JUMP}$		$H_1: \mu_{DPWP} \neq \mu_{JUMP}$		$H_1: \mu_{RR} \neq \mu_{JUMP}$		$H_1: \mu_{DPWP} \neq \mu_{JUMP}$	

6.5.5.3. Desenho de fractais Mandelbrot (aplicação jrMandel)

A Tabela 6.20 e a Figura 6.11 mostram as médias dos tempos de resposta da aplicação jrMandel com diferentes cargas de trabalho e políticas de escalonamento, e o ganho de desempenho quando a política JUMP é utilizada. Os parâmetros utilizados foram: centro em $-0,251786 + 0,639560i$, aproximação 10, resolução 2560×1920 e número de iterações variável.

Tabela 6.20 - Ganho em desempenho da política JUMP para a aplicação jrMandel.

<i>jrMandel</i>	AMIGO-RR		AMIGO-JUMP		Ganho (%)
Iterações	$\bar{x}_{obs}$	dp_{obs}	$\bar{x}_{obs}$	dp_{obs}	
128	11,316	0,051	8,667	1,838	23,41
256	19,174	0,070	13,405	2,838	30,09
512	32,298	0,038	17,280	2,950	46,50
1024	54,649	0,037	30,302	3,446	44,55

Figura 6.11 - Tempos de resposta em relação à carga para a aplicação jrMandel.

Os testes de hipóteses para as diferenças dos tempos de resposta para a aplicação jrMandel são apresentados na Tabela 6.21. O *t-test* de Student é utilizado devido ao pequeno tamanho das amostras.

Tabela 6.21 - Testes de hipóteses para a aplicação jrMandel com diferentes cargas.

	jrMandel							
	128		256		512		1024	
	RR	JUMP	RR	JUMP	RR	JUMP	RR	JUMP
Média	11,316	8,668	19,174	13,406	32,298	17,280	54,650	30,302
Variância	0,003	3,381	0,005	8,056	0,001	0,895	0,001	11,877
Tamanho	10	10	10	10	10	10	10	10
Teste (T)	4,553		6,425		50,164		22,340	
t-test	0,001		0,000		0,000		0,000	
Conclusão	$H_1: \mu_{RR} \neq \mu_{JUMP}$		$H_1: \mu_{DPWP} \neq \mu_{JUMP}$		$H_1: \mu_{RR} \neq \mu_{JUMP}$		$H_1: \mu_{DPWP} \neq \mu_{JUMP}$	

6.6. Considerações Finais

Neste capítulo foram apresentados os experimentos realizados para avaliar e analisar o desempenho do ambiente AMIGO e da nova política de escalonamento JUMP, os resultados obtidos em cada experimento e a discussão sobre os diversos resultados apresentados.

Três aplicações distribuídas foram utilizadas na avaliação de desempenho, que foram escolhidas por apresentarem a característica de processamento intensivo, que encaixa com o objetivo almejado no desenvolvimento da política JUMP; ao mesmo tempo em que possuem outras características comportamentais diferentes entre si, que foram interessantes para a avaliação.

O primeiro passo foi averiguar a influência do AMIGO sobre o ambiente de passagem de mensagens, utilizando uma política compatível com a mesma política nativa do ambiente. Foi atestado que a sobrecarga imposta sobre o ambiente pela utilização do AMIGO é desprezível, e vastamente compensada pelo benefício da utilização do escalonamento dinâmico do AMIGO e suas políticas de escalonamento.

Em seguida, o desempenho das políticas *round-robin*, DPWP (Araújo, 1999) e JUMP foram comparados para cada uma das aplicações distribuídas, em cada um dos cenários de sistema ocioso, carregado, e com carga variável. A política apresentou melhores tempos de resposta do que as outras políticas para todas as situações e aplicações, exceto uma, onde a política DPWP teve um melhor desempenho.

Por fim, o ganho de desempenho da política de escalonamento JUMP em relação a um ambiente sem balanceamento dinâmico de carga (utilizando a política *round-robin*) foi medido para diferentes cargas de trabalho para cada uma das aplicações distribuídas escolhidas.

Os resultados apresentados são extremamente favoráveis à nova política de escalonamento apresentada neste trabalho, validando a viabilidade de sua utilização em ambientes paralelos distribuídos, para aplicações de uso intensivo de processamento.

O capítulo seguinte encerra este trabalho, discutindo os objetivos almejados e atingidos, as dificuldades encontradas e oferecendo sugestões de pesquisa para trabalhos futuros.

7. Conclusões e Trabalhos Futuros

7.1. Considerações Finais

Este trabalho apresentou uma nova política de escalonamento de processos para balanceamento dinâmico de carga em ambientes paralelos distribuídos, com suporte à migração de processos. A política é voltada para a utilização em plataformas heterogêneas, para aplicações de uso intensivo de processamento. É implementada visando o funcionamento sobre o ambiente de escalonamento flexível e dinâmico AMIGO (Souza, 2000), que foi adaptado para dar suporte à migração de processos.

Durante o desenvolvimento deste trabalho, diversos outros objetivos tiveram que ser cumpridos de forma a atingir o objetivo final. Esses objetivos incluíram alterações no projeto original do ambiente AMIGO, tanto como parte da implementação do suporte à migração de processos, como de outras funcionalidades derivadas que eram necessárias pela primeira; e a adaptação do ambiente de passagem de mensagens DPVM 2.0 (Iskra et al., 2000a) para seu funcionamento integrado ao AMIGO.

Diversos experimentos foram realizados a fim de avaliar e analisar o desempenho do ambiente AMIGO com as alterações realizadas durante este trabalho e da nova política de escalonamento apresentada. As avaliações de desempenho cobrem um grande conjunto de fatores, incluindo diversas aplicações distribuídas, outras políticas de escalonamento, sistemas com diferentes situações de cargas externas e diferentes quantidades de carga de trabalho. Os resultados apresentados enfatizam a qualidade do ambiente AMIGO, evidenciam a eficiência da política de escalonamento JUMP e os benefícios da utilização de um escalonamento de processos dinâmico e preemptivo.

7.2. Contribuições deste Trabalho

Este trabalho traz várias contribuições práticas para a área de pesquisa de escalonamento de processos em ambientes paralelos distribuídos, dentre as quais destacam-se:

- Melhorias no projeto e na implementação do ambiente AMIGO, com um protocolo de comunicações mais robusto, uma implementação modularizada que

ênfatisa a reutilização de código e simplifica a implementação de políticas de escalonamento, extensiva documentação de código e novos recursos. As melhorias no ambiente trazem um benefício para futuros pesquisadores e desenvolvedores, que poderão contar com uma plataforma mais flexível e confiável, permitindo uma maior facilidade de desenvolvimento, simulação e execução de políticas de escalonamento.

- Implementação do suporte à migração de processos no AMIGO. A implementação expande a flexibilidade do ambiente, permitindo agora a sua utilização para o desenvolvimento de políticas de escalonamento com suporte a migração de processos.
- Adaptação do ambiente de passagem de mensagens DPVM 2.0 para a sua execução integrada ao AMIGO, oferecendo aplicação para o suporte à migração de processos implementado.
- O projeto e a implementação da nova política de escalonamento preemptiva JUMP. A política oferece um balanceamento de cargas mais dinâmico para aplicações paralelas distribuídas, por meio da utilização da migração de processos, e da união das decisões de alocação inicial e migração de processos em um algoritmo comum, compartilhando índices de carga e critérios de escalonamento. A implementação traz benefícios práticos e também para a área de pesquisa, uma vez que o algoritmo tem uma implementação aberta, pode ser adaptado e melhorado de diversas formas, atendendo a novos cenários.
- Avaliação e análise do desempenho dos produtos deste trabalho, tanto do AMIGO como da nova política JUMP, considerando diversas aplicações com características distintas. A avaliação cobre um número de fatores aplicados à utilização do ambiente AMIGO e da política JUMP, servindo tanto para evidenciar os resultados práticos dos produtos deste trabalho, quanto para guiar pesquisadores e utilizadores do ambiente na escolha e utilização de políticas de escalonamento para aplicações paralelas distribuídas.
- Outras ferramentas e recursos menores, que acompanham os outros produtos deste trabalho, como: readaptação do ambiente de passagem de mensagens PVM 3.4.5 e da política DPWP, originários do trabalho de Araújo (1999), para a nova versão do AMIGO; uma política *round-robin* para o AMIGO; uma nova ferramenta de *benchmark* para o AMIGO para processadores com suporte a múltiplas *threads*,

console em linha de comando para o AMIGO, correções de defeitos encontrados na implementação do DPVM 2.0, etc. Apesar de serem contribuições de menor significância, todas agregam algum valor aos outros produtos principais deste trabalho.

7.3. Dificuldades Encontradas

O desenvolvimento deste trabalho apresentou diversas dificuldades. A primeira delas foi determinar qual ferramenta de migração de processos seria utilizada no trabalho. O trabalho de Furquim (2006) havia utilizado a ferramenta *dynckpt*/DPVM 2.0 (Iskra et al., 2000a; 2000b), que apresenta diversas limitações, como a necessidade de uma versão específica (e hoje obsoleta) da biblioteca *glibc* alterada para suportar *checkpointing*, falta de suporte a programas que executam múltiplas *threads*, falta de suporte à versão corrente do Linux (2.6), etc. Infelizmente, todas as alternativas disponíveis apresentavam outros problemas diversos como: não preencher os requisitos necessários para este trabalho (poder ser utilizado com um ambiente de passagem de mensagens, integrável ao AMIGO), não estar disponível para uso público, ter sido abandonado há anos, ou ser apenas uma “prova-de-conceito” com funcionalidades ainda mais limitadas do que o *dynckpt*. Para contornar os problemas encontrados no *dynckpt*/DPVM 2.0 seria necessário adaptar alguma outra ferramenta abandonada a um ambiente de passagem de mensagens, adaptar algum outro ambiente para essa finalidade ou construir um novo ambiente com suporte à migração de processos. Todas as opções divergiam do foco do trabalho e inviabilizavam sua conclusão em tempo hábil. Dada a falta de alternativas, optou-se por continuar utilizando o mesmo *dynckpt*/DPVM 2.0 utilizado por Furquim, e construir um *cluster* para atender a este trabalho, uma vez que todos os outros *clusters* à disposição possuíam versões mais recentes e modernas do sistema operacional, incompatíveis com o *dynckpt*.

A segunda grande dificuldade deste trabalho está relacionada a ajustar o ambiente para permitir a migração de processos. A migração de processos em si é uma operação delicada, e o *dynckpt* não ajuda neste sentido. A primeira migração de processos com sucesso foi um marco no desenvolvimento do trabalho. Se a versão de alguma biblioteca, ou alguma característica do sistema operacional for diferente entre o transmissor e o receptor, ao reiniciar o processo, este causa uma “falha de segmentação” de uma forma muito difícil de depurar.

Outra dificuldade foi ocasionada por defeitos e limitações na implementação do DPVM 2.0. Havia diversas regiões do código específico da migração de processos que sofriam com condições de disputa ou erros de comunicação, que foram identificadas e

corrigidas durante o desenvolvimento deste trabalho. Mesmo assim, ainda há ocasiões em que a migração falha, e mesmo depurando o DPVM não foi possível determinar a causa. Essa falha ocorre com uma pequena frequência, mas o suficiente para atrapalhar a execução de experimentos mais longos do que 10 ou 15 execuções contínuas das aplicações.

7.4. Sugestões para Trabalhos Futuros

Diversas outras idéias podem ser utilizadas para futuros projetos de pesquisa utilizando os resultados obtidos neste trabalho:

- A falta de uma outra opção para ferramenta de migração de processos além do *dynckpt*/DPVM é um grande empecilho para a continuidade da pesquisa na área de migração de processos. É necessária uma avaliação mais detalhada das alternativas ao *dynckpt*/DPVM, incluindo um estudo sobre a viabilidade de sua adaptação ao ambiente AMIGO e, se possível, a sua adaptação propriamente dita.
- A política de escalonamento JUMP visa a aplicações de uso intensivo de processamento. Novas políticas de escalonamento com suporte a migração de processos podem ser desenvolvidas visando a outros tipos de balanceamento de carga, como por exemplo, uso de memória. O trabalho de Alves (2008) aborda políticas de escalonamento para aplicações de uso intensivo de memória.
- Observou-se durante o desenvolvimento da política JUMP que diversas políticas de escalonamento em execução simultaneamente tendem a duplicar esforços para a coleta e distribuição de índices de carga. O AMIGO se beneficiaria com a implementação de um módulo dedicado para gerenciamento de informações de carga, que determinaria os índices de carga que devem ser coletados e distribuídos junto às políticas em execução.
- Desenvolvimento de políticas de migração de processos com objetivos diferentes do balanceamento de carga, como tolerância a falhas ou a administração do sistema.
- Adaptar e testar os módulos da camada inferior do AMIGO para outros sistemas operacionais além do Linux, que vem sendo utilizado desde sua implementação original. Os usuários do ambiente se beneficiariam com uma maior variedade de escolha de sistemas onde o AMIGO possa ser utilizado.

Referências

- (Achcar e Rodrigues, 2000) ACHCAR, J. A.; RODRIGUES, J. **Introdução à Estatística para Ciências e Tecnologia**. Apostila de Consulta. Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2000.
- (Alves, 2008) ALVES, L. C. D. **Políticas de escalonamento memory-intensive para aplicações distribuídas**. Dissertação (Mestrado). Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2008.
- (Anderson et al., 1995) ANDERSON, T. E.; CULLER, D. E.; PATTERSON, D. A.; AND THE NOW TEAM. A Case for NOW (Networks of Workstations). **IEEE Micro**, v. 15, n. 1, pp. 54-64. Fev 1995.
- (Ansel et al., 2008) ANSEL, J. COOPERMAN, G.; ARYA, K. **DMTCP: Scalable User-Level Transparent Checkpointing for Cluster Computations**. Technical Report arXiv:cs/0701037. Abr 2008.
- (Araújo, 1999) ARAÚJO, A. P. F. **DPWP – Uma nova Abordagem ao Escalonamento Dinâmico em Computação Paralela Virtual**. Dissertação (Mestrado). Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo, São Carlos, 1999.
- (Araújo et al., 1999) ARAÚJO, 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 and Synthesis – ISAS'99**, Orlando, U.S.A. Jul 1999.
- (Barak et al., 2005) BARAK, A.; SHILOH, A.; AMAR, L. An Organizational Grid of Federated MOSIX Clusters. **IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)**, v. 1, pp. 350-357. 2005.
- (Branco, 2004) BRANCO, K. R. L. J. C. **Índices de carga e desempenho em ambientes paralelos/distribuídos – modelagem e métricas**. Tese (Doutorado). Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2004.
- (Burns et al., 1994) BURNS, G., DAOUD, R., VAIGL, J. LAM: An Open Cluster Environment for MPI. **Proceedings of Supercomputing Symposium**. pp. 379-386. 1994.
- (Campos Jr., 2001) CAMPOS JUNIOR., A. **Desenvolvimento de uma interface gráfica para um ambiente de escalonamento**. Dissertação (Mestrado). Instituto de

Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2001.

- (Carzaniga et al., 1997) CARZANIGA, A.; PICCO, G. P.; VIGNA, G. Designing Distributed Applications with Mobile Code Paradigms. In: **Proceedings of the 19th International Conference on Software Engineering – ICSE'97**, pp. 22-31, ACM Presss, 1997.
- (Casas et al., 1995) CASAS, J.; CLARK, D. L.; CONURU, R.; OTTO, S. W.; PROUTY, R. M.; WALPOLE, J. MPVM: A Migration Transparent Version of PVM. **Computing Systems**, v. 8, n. 2, pp. 171-216. 2005.
- (Casavant e Kuhl, 1994) CASAVANT, T. L.; KUHL, J. G. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. **IEEE Transactions on Software Engineering**, v. 14, n. 2, pp. 141-154. Fev 1988.
- (Cortés, 1999) CORTÉS, O. A. C. **Desenvolvimento de Algoritmos Numéricos Paralelos**. Dissertação (Mestrado). Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos. 1999.
- (Coulouris et al., 2005) COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Distributed Systems: Concepts and Design**. 4th ed. Addison Wesley, 2005. 927 p.
- (Douglass e Ousterhout, 1991) DOUGLIS, F.; OUSTERHOUT, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. **Software - Practice and Experience**, v. 21, n. 8, pp. 757-785. Ago 1991.
- (Elnozahy et al., 2000) ELNOZAHY, E. N.; ALVISI, L.; WANG, Y.; JOHNSON, D. B. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. **ACM Computing Surveys (CSUR)**, v. 34, n. 3, pp. 375-408. Set 2002.
- (Eskicioğlu, 1995) ESKICIOĞLU, M. R. Design Issues of Process Migration Facilities in Distributed Systems. In: SHIRAZI, B. A. et al. Editors. **Scheduling Load Balancing in Parallel and Distributed Systems**. IEEE Computer Society Press, p. 414-424. 1995.
- (Ferrari e Zhou, 1987) FERRARI, D.; ZHOU, S. An Empirical Investigation of Load Indices for Load Balancing Applications. In: **Proceedings of Performance '87, the 12th International Symposium on Computer Performance Modeling, Measurement, and Evaluation**, pp. 515-528, 1987.
- (Figueiredo, 2000) FIGUEIREDO, T. C. **Interface AMIGO-MPI: Uma Abordagem Flexível e Dinâmica para o Escalonamento de Processos**. Dissertação (Mestrado). Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2000.
- (Foster e Kesselman, 2004) FOSTER, I.; KESSELMAN, C. **The Grid: Blueprint for a New Computing Infrastructure**. 2nd ed. Morgan Kaufmann, 2004. ISBN 1558609334. 748 p.

- (Fuggetta et al., 1998) FUGGETTA, A.; PICCO, G. P.; VIGNA, G. Understanding Code Mobility. In: **IEEE Transactions on Software Engineering**, v. 24, n. 5, pp. 342-361, Mai 1998.
- (Furquim, 2006) FURQUIM, G. A. **Balanceamento de cargas de aplicações SPMD em sistemas computacionais distribuídos**. 2006. Dissertação (Mestrado). Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo. São Carlos, 2006.
- (Gabriel et al., 2004) GABRIEL, E. et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: **Proceedings, 11th European PVM/MPI Users' Group Meeting**, Budapest, Hungary, Set 2004.
- (Geist et al., 1994) GEIST, A.; BEGUELIN, A.; DONGARRA, J. J.; JIANG, W.; MANCHEK, R.; SUNDERAM, V. **PVM 3 Users Guide and Reference Manual**. Oak National Laboratory. 1994.
- (Gentzsch, 2001) GENTZSCH, W. Sun Grid Engine: Towards Creating a Compute Power Grid. **Proceedings of the 1st International Symposium on Cluster Computing and the Grid – CCGRID'01**. IEEE, 2001.
- (Gray et al., 2002) GRAY, R. S.; CYBENKO, G.; KOTZ, D.; PETERSON, R. A.; RUS, DANIELA. D'Agents: Applications and Performance of a Mobile-Agent System. **Software - Practice and Experience**, v. 32, n. 6, pp. 543-573, Mai 2002.
- (Gropp et al., 1996) GROPP W.; LUSK, E.; DOSS, N.; SKJELLUM, A. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. **Parallel Computing**, v. 22, n. 6, pp. 782-828. 1996.
- (Haberman e Thaler, 2002) HABERMAN, B.; THALER, D. **Unicast-Prefix-based IPv6 Multicast Addresses**. RFC 3306. Internet Engineering Task Force, Network Working Group. Ago 2002.
- (Hargrove e Duell, 2006) HARGROVE, P. H.; DUELL, J. C. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. **Journal of Physics: Conference Series**, v. 46, pp. 494-499. 2006.
- (Hendriks, 2002) HENDRIKS, E. BProc: The Beowulf Distributed Process Space. **Proceedings of the 16th international conference on Supercomputing**. pp. 129-136. 2002.
- (Iskra et al., 2000a) ISKRA, K. A.; HENDRIKSE, Z. W.; VAN ALBADA, G. D.; OVEREINDER, B. J.; SLOOT, P. M. A. Dynamic Migration of PVM Tasks. In: VAN VLIET, L. J.; HEIJNSDIJK, J. W. J.; KIELMANN, T.; KNIJNENBURG, P. M. W (Org.). **Proceedings of the sixth annual conference of the Advanced School for Computing and Imaging**, ASCI 2000. p. 206-212. Jun 2000.
- (Iskra et al., 2000b) ISKRA, K. A.; VAN DER LINDEN, F.; HENDRIKSE, Z. W.; OVEREINDER, B. J.; VAN ALABADA, G. D.; SLOOT, P. M. A. The implementation of Dynamite - an environment for migrating PVM tasks,

- Operating Systems Review**, v. 34, n. 3, pp. 40-55. Association for Computing Machinery, Special Interest Group on Operating Systems, Jul 2000.
- (ITU, 2004) INTERNATIONAL TELECOMMUNICATION UNION. **Information technology – Open Systems Interconnection – Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 object identifier components**. ITU-T Recommendation X.667. Set 2004.
- (Jankowski et al., 2005) JANKOWSKI, G.; JANUSZEWSKI, R.; KOVACS, J.; MIKOLAJCZAK, R.; MAYER, N. Towards Grid Checkpoint Architecture. **PPAM 2005 Poznan**. 2005.
- (Kannan et al, 2001) KANNAN, S.; ROBERTS, M.; MAYES, P.; BRELSFORD, D.; SKOVIRA, J. F. **Workload Management with LoadLeveler**. IBM, 1st ed. Nov. 2001. Disponível em <<http://ibm.com/redbooks>>
- (Kunz, 1991) KUNZ, T. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. **IEEE Transactions on Software Engineering**, v. 17, n. 7, pp. 725-730. Jul 1991.
- (Leach et al., 2005) LEACH, P.; MEALLING, M.; SALZ, R. **A Universally Unique Identifier (UUID) URN Namespace**. RFC 4122. Internet Engineering Task Force, Network Working Group. Jul 2005.
- (Litzkow e Solomon, 1992) LITZKOW, M.; SOLOMON, M. Supporting Checkpointing and Process Migration outside the UNIX Kernel. **Proceedings of the USENIX Winter Conference**, p. 283-290. 1992.
- (Lottiaux et al., 2005) LOTTIAUX, R.; GALLARD, P.; VALLÉE, G.; MORIN, C.; BOISSINOT, B. OpenMosix, OpenSSI and Kerrighed: A Comparative Study. **Proceedings of 2005 IEEE International Symposium on Cluster Computing and the Grid**. pp. 1016-1023. 2005.
- (Magalhães e Lima, 2002) MAGALHÃES, M. N.; LIMA, A. C. P. **Noções de Probabilidade e Estatística**. 5^a ed. São Paulo: Editora da Universidade de São Paulo, 2002.
- (Malcom, 2004) MALCOM, D. **Stateless Linux Tutorial**. Red Hat, Inc. 2004. Disponível em <<http://people.redhat.com/dmalcolm/stateless/stateless-linux-HOWTO-en/>>
- (Martin et al., 2003) MARTIN, D. E.; WILSEY, P. A.; HOEKSTRA, R. J.; KEITER, E. R.; HUTCHINSON, T. V. R.; WATERS, L. J. Redesigning the WARPED Simulation Kernel for Analysis and Application Development. **Proceedings of the 36th Annual Simulation Symposium (ANSS'03)**, 2003.
- (Morin et al., 2003) MORIN, C.; LOTTIAUX, R.; VALLÉE, G.; GALLARD, P.; UTARD, G.; BADRINATH, R.; RILLING, L. Kerrighed: A Single System Image Cluster Operating System for High Performance Computing. **Proceedings of Euro-Par 2003, LNCS 2790**, pp. 1291-1294. 2003.

- (Morin et al., 2004) MORIN, C.; LOTTIAUX, R.; VALLÉE, G.; GALLARD, P.; MARGERY, D.; BERTHOU, J.; SCHERSON, D. Kerrighed and Data Parallelism: Cluster Computing on Single System Image Operating Systems. **Proceedings of the 2004 IEEE International Conference on Cluster Computing**. 2004.
- (Milojičić et al., 2000) MILOJIČIĆ, D. S.; DOUGLIS, F.; PAINDAVEINE, Y.; WHEELER, R.; ZHOU, S. Process Migration. **ACM Computing Surveys (CSUR)**, v. 32, n. 3, p. 241-299. Set 2000.
- (MPI Forum, 2008a) MPI FORUM. **MPI: A Message-Passing Interface Standard**. Version 1.3. Message Passing Interface Forum. Mai 2008. Disponível em <<http://www.mpi-forum.org/>>
- (MPI Forum, 2008b) MPI FORUM. **MPI: A Message-Passing Interface Standard**. Version 2.1. Message Passing Interface Forum. Jun 2008. Disponível em <<http://www.mpi-forum.org/>>
- (Mullender, 1993) MULLENDER, S. J. Kernel Support for Distributed Systems. In: _____ (Org.). **Distributed Systems**. 2nd ed. Addison-Wesley, 1993. 601 p.
- (Noguchi et al., 2008) NOGUCHI, K.; DILLEN COURT, M. B.; BIC, L. F. Efficient Global Pointers With Spontaneous Process Migration. **16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)**. pp. 87-94. 2008
- (OMG, 2001) OMG. **The Common Object Request Broker: Architecture and Specification**. 2001. Disponível em <<http://www.omg.org/cgi-bin/doc?formal/01-12-01>>
- (Quinn, 1994) QUINN, M. J. **Parallel Computing: Theory and Practice**. 2nd ed. McGraw-Hill, 1994. 446 p.
- (Paindaveine e Milojičić, 1996) PAINDAVEINE, Y.; MILOJIČIĆ, D. S. Process vs. Task Migration. **Proceedings of the 29th Annual Hawaii International Conference on System Sciences**. 1996.
- (Park et al., 2006) PARK, J-S.; SHIN, M-K.; KIM, H-J. **A Method for Generating Link-Scoped IPv6 Multicast Addresses**. Internet Engineering Task Force, Network Working Group. Abr 2006.
- (Pereira e Souza, 2003) PEREIRA, D. C., SOUZA, P. S. L. Uma Nova Política de Escalonamento para Clusters Beowulf Heterogêneos e Multiusuários. In: **XII Encontro Anual de Iniciação Científica – XII EAIC**, Anais..., v. 1, p. 45. Unioeste, Foz do Iguaçu, 2003.
- (Platform, 2006) PLATFORM Computing Inc. **Platform LSF Family of Products**. Platform Computing Inc. 2006. Disponível em <<http://www.platform.com/Products/platform-lsf/resources>>
- (Puder, 2004) PUDER, A. MICO: an open source CORBA implementation. **Software**, IEEE, v. 21, n. 4, pp. 17-19. 2004.

- (Ravasi, 2008) RAVASI, J. F. **jrMandel**, parallel Mandelbrot set fractal renderer. 2008. Disponível em <<http://juliano.info/en/Projects/jrMandel>>
- (Richmond e Hitchens, 1997) RICHMOND, M.; HITCHENS, M. A New Process Migration Algorithm. **ACM SIGOPS Operating Systems Review**, v. 31, p. 31-42. 1997.
- (Roman, 2002) ROMAN, E. A Survey of Checkpoint/Restart Implementations. **LBNL Technical Report**. Lawrence Berkeley National Laboratory, Berkeley, CA. Jul 2002.
- (Sankaran et al., 2005) SANKARAN, S.; SQUYRES, J. M.; BARRETT, B.; LUMSDAINE, A.; DUELL, J.; HARGROVE, P.; ROMAN, E. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. **International Journal of High Performance Computing Applications**. v. 19, n. 4, pp. 479-493. 2005.
- (Santana, 1990) SANTANA, M. J. **An Advanced Filestore Achitecture for a Multiple-LAN Distributed Computing System**. Tese (Doutorado). Department of Electronics and Computer Science, University of Southampton. 1990.
- (Santos, 2001) SANTOS, R. R. **Escalonamento de Aplicações Paralelas: Interface AMIGO-CORBA**. Dissertação (Mestrado). Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2001.
- (Schmidt et al., 1998) SCHMIDT, D. C.; LEVINE, D. L.; MUNGEE, S. The Design of the TAO Real-Time Object Request Broker. **Computer Communications**, Elsevier Science, v. 21, n. 4. 1998.
- (Shivaratri et al., 1992) SHIVARATRI, N. G.; KRUEGER, P.; SINGHAL, M. Load Distributing for Locally Distributed Systems. **IEEE Computer**, v. 25, n. 12, p. 33-44. Dez 1992.
- (Smith, 1988) SMITH, J. M. A Survey of Process Migration Mechanisms. **Operating Systems Review**. v. 22, n. 3, p. 28-40. 1988.
- (Souza, 2000) SOUZA, P. S. L. **AMIGO: Uma Contribuição para a Convergência na Área de Escalonamento de Processos**. 2000. 237 p. Tese (Doutorado). Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 2000.
- (Souza et al., 1999) SOUZA, P. S. L.; SANTANA, M. J.; SANTANA, R. H. C.; ARAÚJO, A. P. F. PVM and a Viable and Flexible Scheduling. **Eleventh IASTED International Conference on Parallel and Distributed Computing and Systems – PDCS' 99**. Cambridge, MA, USA. Nov 1999.
- (Srinivasan, 1995) SRINIVASAN, R. **XDR: External Data Representation Standard**. RFC 1832. Internet Engineering Task Force, Network Working Group. Ago 1995.
- (Sterling et al., 1995) STERLING, T.; BECKER, D. J.; SAVARESE, D.; DORBAND, J. E.; RANAWAKE, U. A.; PACKER, C. V. BEOWULF: A parallel workstation for scientific computation. **Proceedings of the 24th International Conference on Parallel Processing**, pp 11-14. 1995.

- (Táboas, 2002) TÁBOAS, P. **Cálculo em uma variável real**. Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos. 2002.
- (Tanenbaum, 1995) TANENBAUM, A. S. **Distributed Operating Systems**. 1st ed. Prentice-Hall, 1995. 614 p.
- (Tanenbaum, 2003) TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2^a ed. São Paulo: Pearson Prentice Hall, 2003.
- (Tanenbaum e Steen, 2002) TANENBAUM, A. S.; STEEN, M. V. **Distributed Systems: Principles and Paradigms**. 1st ed. Prentice-Hall, 2002.
- (Thain et al., 2005) THAIN, D.; TANNENBAUM, T.; LIVNY, M. Distributed Computing in Practice: the Condor Experience. **Concurrency and Computation: Practice and Experience**, v. 17, n. 2-4, pp 323-356. 2005.
- (Theimer et al., 1985) THEIMER, M. M.; LANTZ, K. A.; CHERITON, D. R. Preemptive Facilities for the V System. In: **Proceedings of the 10th ACM Symposium on Operating Systems Principles**, p. 1-12, 1985.
- (Voorsluys, 2006) VOORSLUYS, B. L. **Influências de políticas de escalonamento no desempenho de simulações distribuídas**. Dissertação (Mestrado). Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2006.
- (Voorsluys e Souza, 2002) VOORSLUYS, W.; SOUZA, P. S. L. Uma Política de Escalonamento de Processos como uma Ferramenta de Testes para Índices de Carga. In: **XI Encontro Anual de Iniciação Científica – XI EAIC**, Anais... Universidade Estadual de Maringá, Maringá, 2002.
- (Walker, 2006) WALKER, B. J. **OpenSSI (Single System Image) Clusters for Linux**. 2006. Disponível em: <<http://www.openssi.org/>> Acesso em: 5 fev 2007.
- (Walker e Steel, 1999) WALKER, B. J.; STEEL, D. Implementing a Full Single System Image UnixWare Cluster: Middleware vs Underware. In: **Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'99**, 1999.
- (Wang et al., 2008) WANG, C.; MUELLER, F; ENGELMANN, C.; SCOTT, S. L. Proactive Process-Level Live Migration in HPC Environments. In: **Proceedings of the 2008 ACM/IEEE conference on Supercomputing**, 2008.
- (Zhou et al. 1994) ZHOU, S.; ZHENG, X.; WANG, J.; DELISLE, P. Utopia: A Load Sharing Facility for Large Heterogeneous Distributed Computer Systems. **Software - Practice and Experience**. 1994.