FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” – UNIVEM
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

GIULIANNA MAREGA MARQUES

AVALIAÇÃO DE ÍNDICES DE CARGA E DE DESEMPENHO EM AMBIENTES PARALELOS DISTRIBUÍDOS COM AGENTES MÓVEIS

MARÍLIA
2008

GIULIANNA MAREGA MARQUES

AVALIAÇÃO DE ÍNDICES DE CARGA E DE DESEMPENHO EM AMBIENTES PARALELOS DISTRIBUÍDOS COM AGENTES MÓVEIS

Dissertação apresentada ao Centro Universitário Eurípides de Marília – UNIVEM, mantido pela Fundação de Ensino Eurípides Soares da Rocha, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação (Área de Concentração: Arquitetura de Sistemas Computacionais).

Orientadora:
Profª. Drª. Kalinka R. L. J. C. Branco.

MARÍLIA
2008

MARQUES, Giulianna Marega

Avaliação de Índices de Carga e de Desempenho em Ambientes Paralelos Distribuídos com Agentes Móveis / Giulianna Marega Marques; orientadora: Profª. Drª. Kalinka R. L. J. C. Branco. Marília, SP: [s.n.], 2008.

124 f.

Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha.

1. Desempenho 2. Escalonamento 3. Agente Móvel

CDD: 005.2

GIULIANNA MAREGA MARQUES

AVALIAÇÃO DE ÍNDICES DE CARGA E DE DESEMPENHO EM AMBIENTES PARALELOS DISTRIBUÍDOS COM AGENTES MÓVEIS

Banca examinadora da dissertação apresentada ao Programa de Mestrado do Centro Universitário Eurípides de Marília – UNIVEM, para obtenção do Título de Mestre em Ciência da Computação (Área de Concentração: Arquitetura de Sistemas Computacionais).

Resultado: Aprovada

ORIENTADORA: Profª. Drª. Kalinka R. L. J. C. Branco

1º EXAMINADOR: Prof. Dr. Márcio Eduardo Delamaro

2º EXAMINADOR: Prof. Dr. Marcos José Santana

Marília, 10 de março de 2008.

A meus pais,
irmãos e sobrinho.
A meu namorado.

AGRADECIMENTOS

Vencer a si próprio consiste em uma das mais relevantes vitórias. Destarte, creio ter superado grandes dificuldades na consecução de meus ideais... algumas vezes seguindo sozinha; em muitas, acompanhada e assistida de outras pessoas. Injusto seria não agradecer a TODOS que, direta ou indiretamente, cooperaram no sentido da conquista deste ideal: concluir o presente trabalho – itinerário decisivo para o título de Mestre.

A Deus, pelo existir de TUDO e TODOS.

À minha família, agradeço o crescimento pessoal e profissional traduzido em ensinamentos, educação, motivação, apoio e confiança em mim depositada.

Ao meu namorado, pelo carinho, paciência, compreensão e colaboração. Sua presença cotidiana favorece minha felicidade e serenidade.

À Profª. Drª. Kalinka R. L. J. Castelo Branco, minha orientadora, que a mim devotou dedicação, ensinamentos, confiança.

Sou grata a todos os Professores – dispensadores de conhecimento e experiências.

Ao pessoal do Mestrado e dos Laboratórios, em especial ao Rodrigo (LOST), Silvio (LRV), Franciene (LES), Marçal (LAS), Lima (LAS), professor Raul (LAS) e a Michele (LAPIS), por agraciarem-me o cotidiano de companheirismo e de auxílio. Em particular, ao Sabatine (LAPIS/LAS), por suas inúmeras contribuições. Que sua sabedoria seja utilizada com discernimento.

A amigos e colegas, por compreenderem minha ausência e os momentos de fragilidade emocional.

À CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), pelo amparo financeiro.

OBRIGADA a todos os colaboradores que, de alguma forma caminhando a meu lado, beneficiaram o desenvolvimento deste trabalho e possibilitaram-me conquistar um de meus ideais.

"Faz da tua alma um diamante.
Por cada novo golpe uma nova face,
para que um dia ela seja toda luminosa."
(Rogelio Stela Bonilla)

"Não encontre apenas problemas,
também encontre as soluções."
(Giulianna Marega Marques)

MARQUES, Giulianna Marega. Avaliação de Índices de Carga e de Desempenho em Ambientes Paralelos Distribuídos com Agentes Móveis. 2008. 124 f. Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2008.

RESUMO

Os ambientes paralelos distribuídos tornaram-se de grande importância devido às vantagens que podem prover: bom desempenho, flexibilidade, baixo custo de aquisição e de manutenção, entre outras. Nesses ambientes, o bom desempenho é proporcional à eficiência do escalonamento de processos, e, caso este efetue o balanceamento de cargas, o desempenho pode ser ainda maior. Não obstante, o balanceamento de cargas não é uma tarefa trivial, sobremaneira quando se tratar de ambientes heterogêneos (tanto em sua forma configuracional quanto na arquitetural), porquanto há necessidade de utilização coerente da potência computacional dos recursos de todos os *hosts* que compõem o ambiente, sem que se sobrecarreguem alguns, deixando outros ociosos. A utilização de índices de desempenho no balanceamento de cargas pode trazer grandes benefícios, reduzindo o impacto da heterogeneidade; no entanto, a freqüência em que são coletados pode gerar tráfego na rede, ou mesmo um balanceamento de cargas incorreto. Este trabalho apresenta uma contribuição no tocante à otimização do balanceamento de cargas a partir da instrumentação da biblioteca de passagem de mensagem JPVM e do desenvolvimento de uma estratégia de escalonamento que possua um tratamento de heterogeneidade do ambiente por meio da utilização de índices de desempenho. Para a coleta dos índices de desempenho dos *hosts* do ambiente paralelo distribuído, utilizou-se um agente móvel. Os resultados foram satisfatórios quando efetuada análise de desempenho de aplicações paralelas, escalonadas com a estratégia de escalonamento desenvolvida utilizando-se agente móvel.

Palavras-Chave: Escalonamento de Processos, Balanceamento de Carga, Índice de Carga, Índice de Desempenho, e Agente Móvel.

MARQUES, Giulianna Marega. Avaliação de Índices de Carga e de Desempenho em Ambientes Paralelos Distribuídos com Agentes Móveis. 2008. 124 f. Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2008.

ABSTRACT

Owing largely to advantages they can provide, such as a good performance, flexibility, low cost of acquisition and maintenance among others, distributed parallel environments have become very important nowadays. In these environments, a good performance is proportional to efficient scheduling of the processes, and if a load balancing is held, the performance can be improved for the better. However, load balancing is not a trivial task, moreover when configurational or architectural heterogeneous environments. In order to achieve a good performance, it is necessary to use coherently the computational power of resources of all hosts which make up the environment, without overloading some and lying idle the others. Thus the whole procedure must be done in a balanced way, that is, concerned about the intercommunication network. Applying performance indices to load balancing can be useful to reduce the impact of heterogeneity, but the frequency in which they are collected may generate network traffic or an incorrect load balancing. As a contributory means of achieving a good load balancing, it must be necessary to use both an instrumentation of a JPVM *Message* passage library and the development of a scheduling strategy which has treatment of heterogeneity in the environment by applying performance indices. With the main purpose of gathering all hosts performance indices from a distributed parallel environment, a mobile agent was applied. Promising favorable results were achieved when a performance analysis of parallel applications was made, scheduled with scheduling techniques developed by using mobile agent.

Keywords: Schedule, Load Balancing, Load Index, Performance Index, and Mobile Agent.

LISTA DE ILUSTRAÇÕES

Figura 1. Taxonomia de Flynn 22

Figura 2. Arquitetura Multiprocessador 24

Figura 3. Arquitetura Multicomputador 24

Figura 4. Arquitetura da Rede Baseada em Barramento 25

Figura 5. Arquitetura da Rede Baseada em Comutação 26

Figura 6. Processos em um Sistema Fisicamente Paralelo 27

Figura 7. Processos em um Sistema Logicamente Paralelo (Pseudoparalelismo) 27

Figura 8. Relacionamento Existente entre as Taxonomias. 28

Figura 9. Sistema Computacional Paralelo Distribuído. 33

Figura 10. Classificação Hierárquica da Composição dos Algoritmos de Escalonamento 36

Figura 11. Taxonomia de Escalonamento - Classificação Hierárquica 38

Figura 12. Sistema Distribuído sem Balanceamento de Carga 42

Figura 13. Sistema Computacional Paralelo Distribuído com carga desbalanceada. 42

Figura 14. Lacunas Existentes na Literatura Quando Levado em Consideração os Níveis Arquiteturais e Configuracionais 53

Figura 15. ***M***odelo ***D***e ***Í***ndice de ***D***esempenho em ***A***mbientes ***h***eterogêneos (MEDIDA*h*). 54

Figura 16. Estratégia para Obtenção do Índice de Desempenho 54

Figura 17. Espaço bidimensional formado pelos recursos 1 e 2, e duas máquinas com cargas iguais (processo limitado por um recurso) 56

Figura 18. Mecanismos de Mobilidade 59

Figura 19. Arquitetura do *CoordAgent* 65

Figura 20. Configurações Iniciais de um *Organic Grid* ....

Figura 21. Organização Resultante após alguns Ciclos de Execução do Al
Escalonamento do *Organic Grid* 70

Figura 22. Execução do jpvmDaemon 75

Figura 23. Funcionalidades do jpvmConsole. 76

Figura 24. Execução do *Dstat*.. 79

Figura 25. Execução do jpvmDaemon com Argumento para se Utilizar o P*Agent*. 82

Figura 26. Estrutura Básica do JPVM-PRM P*Agent*. 83

Figura 27. Execução do jpvmDaemon com Argumento para se Utilizar o P*Message*...............84
Figura 28. Estrutura Básica do JPVM-PRM P*Message*. 85
Figura 29. Execução do jpvmDaemon com Argumento de Filtro de *Hosts*..............................87
Figura 30. Multiplicação de Matriz - 9 Processos - Mestre Las08...93
Figura 31. Multiplicação de Matriz - 9 Processos - Mestre Les05...94
Figura 32. Multiplicação de Matriz - 9 Processos - Mestre Las08 - Ambiente Carregado........95
Figura 33. Multiplicação de Matriz - 16 Processos - Mestre Las08..96
Figura 34. Multiplicação de Matriz - 16 Processos - Mestre Les05..97
Figura 35. Multiplicação de Matriz - 16 Processos - Mestre Las08 - Ambiente Carregado......97
Figura 36. SOR - 9 Processos - Mestre Las08...99
Figura 37. SOR - 9 Processos - Mestre Les05...100
Figura 38. SOR - 9 Processos - Mestre Las08 - Ambiente Carregado...................................101
Figura 39. SOR - 16 Processos - Mestre Las08...102
Figura 40. SOR - 16 Processos - Mestre Les05...103
Figura 41. SOR - 16 Processos - Mestre Las08 - Ambiente Carregado.................................103

LISTA DE TABELAS

Tabela 1. Comparações entre Paradigmas de Código Móvel 60
Tabela 2. Principais Propriedades de Agentes 62
Tabela 3. Cenários Elaborados para os Estudos de Caso. 91
Tabela 4. Multiplicação de Matriz - 9 Processos - Mestre Las08. 93
Tabela 5. Multiplicação de Matriz - 9 Processos - Mestre Les05. 94
Tabela 6. Multiplicação de Matriz - 9 Processos - Mestre Las08 - Ambiente Carregado..... 94
Tabela 7. Multiplicação de Matriz - 16 Processos - Mestre Las08. 95
Tabela 8. Multiplicação de Matriz - 16 Processos - Mestre Les05. 96
Tabela 9. Multiplicação de Matriz - 16 Processos - Mestre Las08 - Ambiente Carregado..... 97
Tabela 10. SOR - 9 Processos - Mestre Las08. 99
Tabela 11. SOR - 9 Processos - Mestre Les05. 100
Tabela 12. SOR - 9 Processos - Mestre Las08 - Ambiente Carregado. 100
Tabela 13. SOR - 16 Processos - Mestre Las08. 101
Tabela 14. SOR - 16 Processos - Mestre Les05. 102
Tabela 15. SOR - 16 Processos - Mestre Las08 - Ambiente Carregado. 103

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Program Interface*
CPU	*Central Processing Unit*
EP	Elemento de Processamento
HTTP	*HyperText Transfer Protocol*
JPVM	*Java Parallel Virtual Machine*
JPVM-PRM	*Java Parallel Virtual Machine with Performance Resource Monitor*
LAN	*Local Area Network*
MAN	*Metropolitan Area Network*
MCS	*Mobile Code Systems*
MDS	*Globus Metacomputing Directory Service*
MEDIDA*h*	***M**od**e**lo **D**e **Í**ndice de **D**esempenho em **A**mbientes **h**eterogêneos*
MIMD	*Multiple Instruction Stream/ Multiple Data Stream*
MISD	*Multiple Instruction Stream/ Multiple Data Stream*
MPI	*Message Passage Interface*
MPP	*Massively Parallel Processing*
NORMA	***NO** **R**emote **M**emory **A**ccess*
PAgent	*Performance Agent*
P*Message*	*Performance Message*
PRM	*Performance Resource Monitor*
PRS	*Performance Resource Scheduling*
PVIP	*Ponderated Vector for Index of Performance*
PVM	*Parallel Virtual Machine*
RAD	*Rapid Application Development*
SIMD	*Single Instruction Stream/ Multiple Data Stream*
SISD	*Single Instruction Stream/ Single Data Stream*
UCP	Unidade Central de Processamento
VIP	*Vector for Index of Performance*
WAN	*Wide Area Network*

SUMÁRIO

CAPÍTULO 1.INTRODUÇÃO ... 16

1.1.Organização do Trabalho ... 18

CAPÍTULO 2.COMPUTACÃO PARALELA DISTRIBUÍDA ... 20

2.1.Computação Paralela ... 21

2.2.Sistemas Distribuídos ... 29

2.3.Computação Paralela Distribuída ... 32

2.3.1. Escalonamento de Processos ... 34

2.3.1.1. Componentes de um Algoritmo de Escalonamento ... 34

2.3.1.2. Taxonomia de Escalonamento ... 37

2.3.1.3. Balanceamento de Carga ... 41

2.4.Considerações Finais ... 44

CAPÍTULO 3.ÍNDICES DE CARGA E DE DESEMPENHO ... 45

3.1.Avaliação de Desempenho ... 46

3.2.Carga de Trabalho ... 47

3.3.Índices de Carga ... 49

3.3.1. Estado da Arte ... 51

3.4.Índice de Desempenho ... 52

3.5.Considerações Finais ... 57

CAPÍTULO 4.CÓDIGOS MÓVEIS ... 58

4.1.Agentes ... 61

4.1.1. Agentes Móveis ... 63

4.2.Considerações Finais ... 70

CAPÍTULO 5.UTILIZAÇÃO DE ÍNDICE DESEMPENHO EM AMBIENTES PARALELOS DISTRIBUÍDOS....................72

5.1.JPVM – *Java Parallel Virtual Machine*....................74

5.2.JPVM-PRM – *Java Parallel Virtual Machine with Performance Resource Monitor*...77

5.2.1. PRM – *Performance Resource Monitor*....................77

5.2.2. PRS – *Performance Resource Scheduling*....................81

5.3.Considerações Finais....................87

CAPÍTULO 6.AVALIAÇÃO DE DESEMPENHO E RESULTADOS....................89

6.1.Análise dos Resultados Obtidos....................90

6.1.1. Multiplicação de Matriz....................92

6.1.2. SOR - *Successive Over Relaxation*....................98

6.2.Considerações Finais....................104

CAPÍTULO 7.CONCLUSÃO....................105

7.1.Sugestões de Trabalhos Futuros....................107

7.2.Publicações Mais Relevantes....................108

REFERÊNCIAS BIBLIOGRÁFICAS....................109

APÊNDICE A.PROCEDIMENTOS UTILIZADOS PARA INSTALAÇÃO E CONFIGURAÇÃO DO AMBIENTE PARALELO DISTRÍBUIDO....................116

CAPÍTULO 1. INTRODUÇÃO

No início de sua era, os computadores eram grandes, não flexíveis e com alto custo de aquisição e de manutenção. Com o passar do tempo, os computadores tiveram reduzidos seu custo e seu tamanho, tornando-se cada vez mais rápidos e foram capacitados para se comunicar por meio de redes de intercomunicação (TANENBAUM, 1999).

Com o desenvolvimento tecnológico, surgiram as arquiteturas paralelas, os sistemas distribuídos e, posteriormente, a convergência dessas tecnologias, a nomeada computação paralela distribuída. Junto a esse novo paradigma, surgiram diversas possibilidades de pesquisa, dentre as quais, as taxonomias para classificação da arquitetura de computadores. O crescente avanço deu-se pela busca por maior desempenho, confiabilidade, flexibilidade quanto ao crescimento da potência computacional, custo acessível para aquisição e manutenção.

Dentre as taxonomias existentes na literatura, uma das que se destaca, é a proposta por Flynn (FLYNN, 1972), que se baseia em fluxos de instruções e fluxos de dados e possui quatro combinações possíveis. Neste estudo, enfatiza-se apenas uma delas, visto que se enquadra à topologia utilizada na pesquisa.

A convergência de sistemas paralelos, que podem prover um bom desempenho, com sistemas distribuídos, que visam a fornecer confiabilidade, flexibilidade, transparência, custo acessível, entre outros, trouxe bons resultados.

Para que um sistema computacional paralelo distribuído tenha alto desempenho não basta um grande número de *hosts* (elementos de processamento ou nós). Faz-se necessário

também que todos tenham um alto grau de participação na execução da aplicação. Deste modo, divide-se a aplicação em processos, os quais são distribuídos entre esses *hosts*. O responsável por efetuar a distribuição de processos e determinar em que ordem serão executados é o escalonador de processos (REWINI *et al.,* 1995).

O escalonador de processos pode atingir diversos objetivos a partir das políticas e mecanismos implementados em seu algoritmo. Um de seus objetivos é efetuar o balanceamento de cargas entre os vários *hosts*, que consiste em selecionar uma tarefa e definir o local onde será executada (ZALUSKA, 1991). Utiliza-se este objetivo para que todos os *hosts* tenham a capacidade de seus recursos explorada de forma coerente e equilibrada entre si.

A tarefa atribuída a um elemento de processamento gera uma carga de trabalho e, para quantificar essa carga e distinguir o estado do recurso analisado, podem-se utilizar os índices de cargas (FERRARI e ZHOU, 1987)(KUNZ, 1991).

O processo de obtenção desses índices de carga se dá nas políticas de escalonamento e, dependendo da freqüência com que é feita a coleta das informações das cargas, pode-se gerar um tráfego maior do que o esperado, prejudicando o desempenho do sistema, ao invés de melhorar.

A utilização de políticas, mecanismos e índices de carga adequados para um determinado sistema pode proporcionar um desempenho elevado e satisfatório, mas há um paradigma importante a se ressaltar: é mais simples obter essa resultante em ambientes homogêneos do que em ambientes heterogêneos (FERRARI e ZHOU, 1987)(KUNZ, 1991)(SOUZA, 2004).

Normalmente, um sistema computacional paralelo distribuído compõe-se de *hosts* heterogêneos, tanto referentes à arquitetura quanto às configurações. Essa característica afeta nas resultantes dos índices de cargas e de seus dependentes, como o escalonamento com

balanceamento de cargas.

Branco (BRANCO, 2004) propôs um índice de desempenho que pode fornecer informações da carga de trabalho e da situação de operação de cada um dos *hosts* do sistema envolvido no processo, considerando todos os tipos de heterogeneidades.

O propósito deste trabalho é oferecer alternativas para coleta dos índices de carga e de desempenho de um ambiente paralelo distribuído, objetivando um escalonamento balanceado e confiável para se obter um bom desempenho referente ao tempo de execução de aplicações apropriadas para essa caracterização de ambiente.

1.1. Organização do Trabalho

Estruturalmente, o trabalho compõe-se de sete capítulos, sendo o primeiro esta introdução, e os demais detalhados a seguir.

No CAPÍTULO 2 faz-se uma revisão bibliográfica a respeito da computação paralela distribuída. Descrevem-se ainda os conceitos de escalonamento de processos bem como seus objetivos, onde se enfatiza o balanceamento de carga.

Apresentam-se no CAPÍTULO 3, os fatores que influenciam na avaliação de desempenho, como os índices de carga e de desempenho.

No CAPÍTULO 4 foi estabelecido o histórico teórico dos códigos móveis, em que se colocam em evidência as características dos agentes móveis.

Apresentam-se no CAPÍTULO 5, motivadores para o desenvolvimento deste trabalho, detalhes do projeto implementado, incluindo as contribuições que pode prover.

No CAPÍTULO 6 aborda-se o método utilizado de avaliação de desempenho do projeto desenvolvido e os resultados a que se chegou, com o intuito de demonstrar a

viabilidade do mesmo.

Finalmente, no CAPÍTULO 7, tece-se a conclusão do presente trabalho, sugerem-se possíveis trabalhos futuros e apresentam-se as principais publicações relacionadas a este projeto.

CAPÍTULO 2. COMPUTACÃO PARALELA DISTRIBUÍDA

No início da era dos computadores, as máquinas eram muito grandes e caras. Em função disso, apenas grandes organizações possuíam computadores e os mesmos operavam de forma independente por falta de uma forma confiável de interligá-los (TANENBAUM, 1999).

Com o passar do tempo, cresceu a busca por menor custo, maior processamento e maior desempenho, e o fator alto custo com supercomputadores representou um obstáculo para sua disseminação.

Por algum tempo, utilizaram-se somente máquinas multiprocessadoras paralelas de alto custo (supercomputadores), que oferecem alto desempenho. No entanto, não eram flexíveis quanto à escalabilidade, ampliação da potência computacional e de configuração. Assim, devido ao crescente avanço tecnológico e a busca por maior desempenho computacional, surgiram novos conceitos e novas idéias.

A partir dos anos 80 dois avanços na tecnologia mudaram essa situação. O primeiro foi o desenvolvimento de microprocessadores com a potência de processamento cada vez maior e de custo menor. O segundo foi a comunicação de computadores em redes locais de alta velocidade chamadas de LANs (*Local Area Network*). A partir desses avanços surgiram os conceitos de sistemas distribuídos (TANENBAUM, 1999), e a classificação das arquiteturas paralelas.

2.1. Computação Paralela

A computação paralela explora a existência de tarefas que podem ser executadas em paralelo, para que se obtenha um tempo de resposta menor. Para isso, utilizam-se múltiplos *hosts* capazes de se comunicar e cooperar para resolver grandes problemas de forma ágil (ALMASI e GOTTLIEB, 1994).

Utiliza-se o termo tarefas concorrentes para definir a rotina de dois ou mais processos inicializados disputando entre si para utilizar o processador para sua execução. A concorrência pode ocorrer tanto em sistemas paralelos quanto em sistemas com um único processador. Em sistemas uniprocessados tem-se um pseudoparalelismo, que é a execução de cada processo em pequenos intervalos de tempo, dando a impressão de que os processos executam simultaneamente. Para que haja paralelismo real é preciso que dois ou mais processos executem no mesmo intervalo de tempo (ALMASI e GOTTLIEB, 1994).

Com o intuito de reduzir os custos com a aquisição e manutenção de supercomputadores e aumentar a flexibilidade, realizaram-se estudos para o desenvolvimento de tecnologias com grande potência computacional a custos razoáveis. Propuseram-se várias maneiras para se conectarem os recursos computacionais, originando diferentes arquiteturas.

Cada arquitetura apresenta características visando ao melhor desempenho sob um dado enfoque. Para acompanhar o desenvolvimento das arquiteturas paralelas e agrupar os equipamentos com características comuns, propuseram-se algumas taxonomias, dentre as quais a de Flynn (FLYNN, 1972)(FLYNN e RUDD, 1996), e a de Duncan (DUNCAN, 1990).

A taxonomia proposta por Flynn baseia-se em fluxos de instruções e fluxos de dados. O fluxo de instruções está relacionado com o programa que o processador executa, enquanto o fluxo de dados se relaciona com os operandos manipulados por essas instruções (QUINN, 1994)(TANENBAUM, 1999). Como se pode observar na Figura 1, o fluxo de instruções e o

fluxo de dados são independentes, e, por isso, existem quatro combinações possíveis: SISD (*Single Instruction Stream/ Single Data Stream*), SIMD (*Single Instruction Stream/ Multiple Data Stream*), MISD (*Multiple Instruction Stream/ Multiple Data Stream*) e MIMD (*Multiple Instruction Stream/ Multiple Data Stream*) (FLYNN, 1972)(FLYNN e RUDD, 1996).

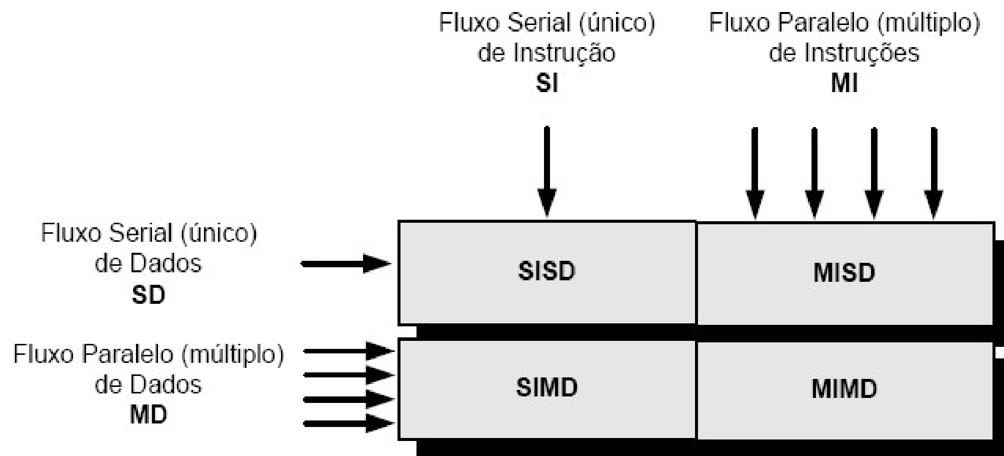


Figura 1. Taxonomia de Flynn (FLYNN, 1972).

Essas definições permitem que várias topologias de máquinas paralelas e de redes de computadores sejam enquadradas como arquitetura MIMD. Faz-se a diferenciação entre as diversas topologias MIMD pelo tipo de organização da memória principal, da memória *cache* e da rede de interconexão. Quando vários *hosts* são interconectados, a forma como cada um visualiza a memória é decisiva para a definição dos modelos de comunicação. A questão que se destaca entre as variantes desse modelo é o espaço de endereçamento de memória.

Há possibilidade de os processadores compartilharem a mesma memória (*shared memory* ou memória compartilhada) e denominarem-se de fortemente acoplados (*tightly coupled*), ou dos processadores possuírem sua própria memória [*multiple private address space* ou memória distribuída (privada)] e serem chamados de fracamente acoplados (*loosely coupled*).

Em decorrência de ser uma extensão natural do modelo de programação seqüencial, a

memória compartilhada popularizou-se. Caracteriza-se por possuir um único espaço de endereçamento virtual compartilhado entre todos os processadores que se comunicam por meio de *load* e *store* (carrega e armazena) nos endereços de memória (CULLER *et al.,* 1999)(STALLINGS, 2003)(TANENBAUM e VAN STEEN, 2002).

Um espaço de endereçamento distinto para cada processador caracteriza a memória distribuída ou privada, ou seja, ela possui seu próprio espaço de endereçamento, além de expor a plataforma paralela como uma coleção de *hosts* conectados a uma rede de alta velocidade, que se comunicam por meio de troca de mensagens com as primitivas *send* e *receive* (envia e recebe) (CULLER *et al.,* 1999)(STALLINGS, 2003)(TANENBAUM e VAN STEEN, 2002).

Os dois modelos são bem aceitos e possuem vantagens e desvantagens. Ao mesmo tempo em que o modelo compartilhado não precisa lidar com o problema da comunicação, este possui um limite em sua potência de processamento, além dos problemas de compartilhamento de variáveis entre os processadores. Em contrapartida, o modelo distribuído não possui um limite para sua potência de processamento – bastando que mais uma *host* seja conectado – mas exige uma atenção especial com a comunicação e o sincronismo (SKILLICORN e TALIA, 1998).

Essa primeira distinção refere-se aos aspectos lógicos da memória e, dependendo desse aspecto, a arquitetura MIMD pode dividir-se em dois grupos (em se tratando dos aspectos físicos): memória distribuída (*distributed memory*) – memória composta por vários módulos, cada um dos quais está próximo de um processador; e memória centralizada (*centralized memory*) – memória que se encontra à mesma distância de todos os processadores e pode ser implementada com um ou vários módulos.

Em estudos mais avançados da taxonomia de Flynn (FLYNN, 1972)(FLYNN e RUDD, 1996), chegou-se ao consenso de que a categoria MIMD pode ser dividida em dois

grupos: multiprocessadores e multicomputadores (STALLINGS, 2003)(TANENBAUM e VAN STEEN, 2002).

Os multiprocessadores são as arquiteturas que utilizam memória compartilhada. Essa estrutura é semelhante à colocação de múltiplos processadores em uma máquina Von Neumann tradicional. Os múltiplos processadores são conectados à memória por meio de uma rede de interconexão, como é possível visualizar no exemplo da Figura 2.

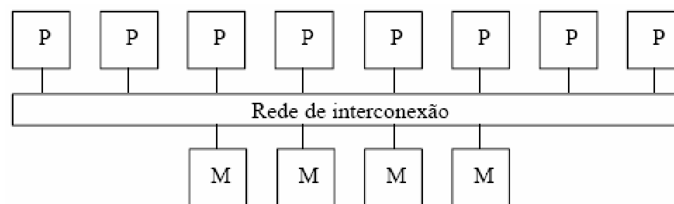


Figura 2. Arquitetura Multiprocessador (TANENBAUM, 2001).

Diferente dos multiprocessadores, os multicomputadores caracterizam-se por utilizar memória distribuída e, em decorrência disso, também podem ser chamados de sistemas de troca de mensagens (*message passing systems*). Por conseguinte, essas máquinas paralelas podem ser implementadas por meio de um conjunto de máquinas autônomas, ou seja, computadores tradicionais fazendo o uso de uma biblioteca de passagem de mensagem para se comunicarem. Exibe-se um exemplo da arquitetura de multicomputador na Figura 3.

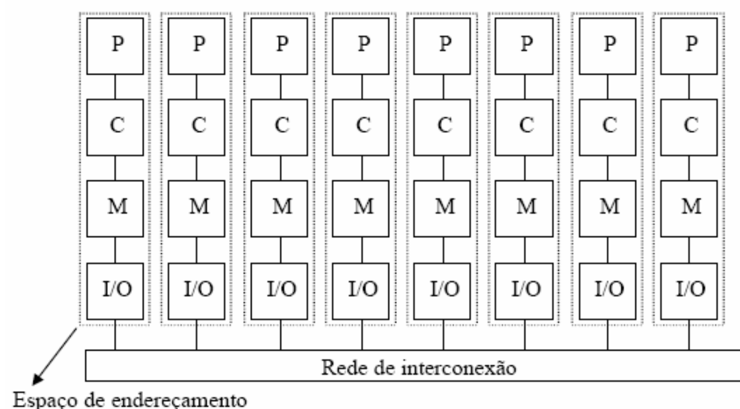


Figura 3. Arquitetura Multicomputador (TANENBAUM, 2001).

Além da subdivisão com relação à organização da memória, existe uma divisão que toma por base a arquitetura da rede de interconexão, podendo se basear em barramento (*bus-based*), ou em comutação (*switch-based*).

Interpreta-se a arquitetura da rede baseada em barramento como uma rede principal que apenas possui ramificações desta para interconectar os nós. Este ambiente pode ser visualizado na Figura 4.

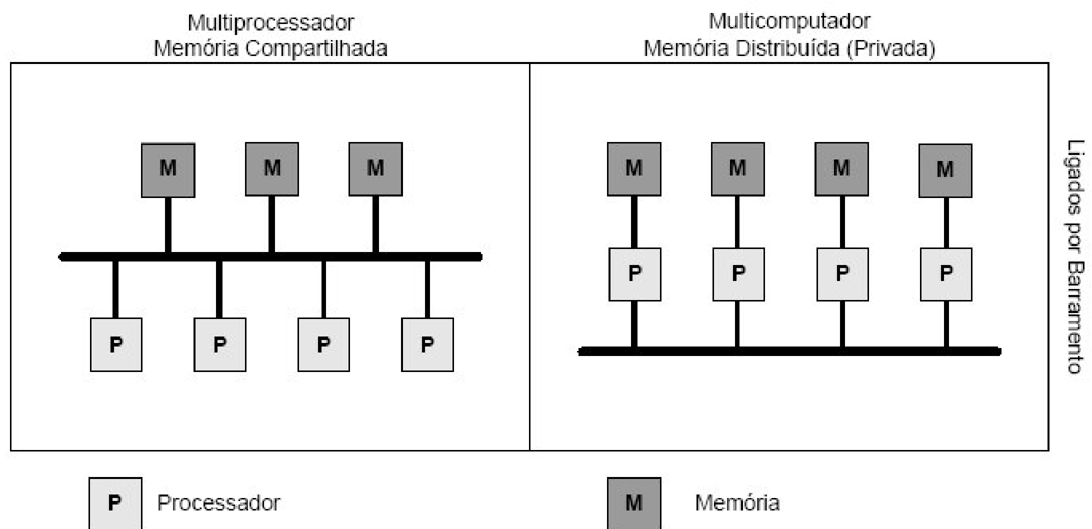


Figura 4. Arquitetura da Rede Baseada em Barramento (TANENBAUM e VAN STEEN, 2002).

Na arquitetura da rede baseada em comutação, todos os nós são interconectados entre si de forma individual, como uma rede pública de telefonia. As mensagens caminham pela rede carregando a informação do trajeto que devem seguir para chegar a seu destino. Para melhor entendimento essa arquitetura é ilustrada na Figura 5.

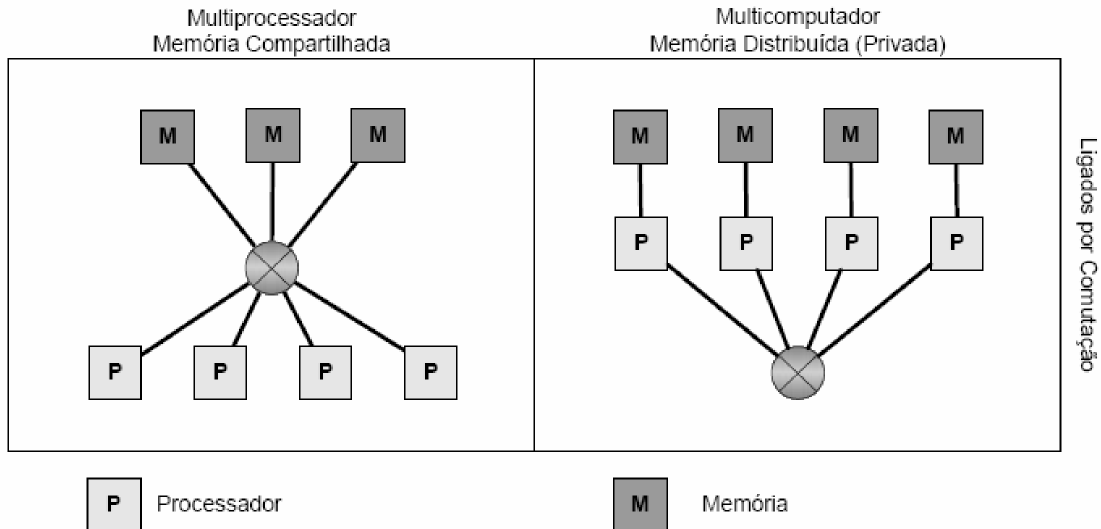


Figura 5. Arquitetura da Rede Baseada em Comutação (TANENBAUM e VAN STEEN, 2002).

A taxonomia de Flynn é muito útil e bastante difundida, mas se limita às quatro categorias citadas, que não são suficientes para acomodar de forma adequada vários computadores modernos. Assim, mantiveram-se os elementos dessa classificação e incluíram-se outros em taxonomias mais abrangentes como, por exemplo, a taxonomia de Duncan (DUNCAN, 1990).

Segundo Duncan (DUNCAN, 1990), uma arquitetura paralela fornece uma estrutura explícita e de alto nível para o desenvolvimento de soluções utilizando o processamento paralelo, por meio de múltiplos processadores (Figura 6) – simples ou complexos – que cooperam para resolver problemas por meio de execução concorrente (Figura 7). Com o objetivo de classificar e incluir arquiteturas que a taxonomia de Flynn não aborda com tanta clareza (por exemplo, processadores vetoriais com *pipeline*), Duncan apresentou uma taxonomia mais abrangente, que consiste em dividir as arquiteturas em síncronas e assíncronas (DUNCAN, 1990).

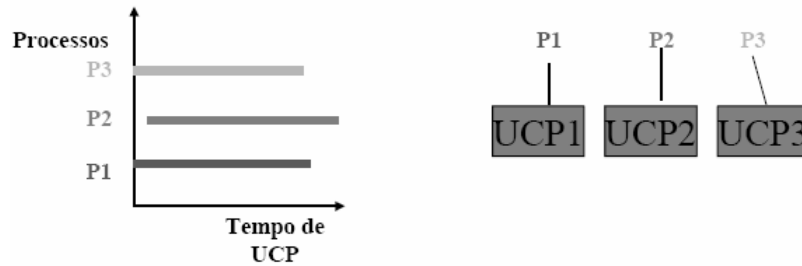


Figura 6. Processos em um Sistema Fisicamente Paralelo (TANENBAUM, 2001).

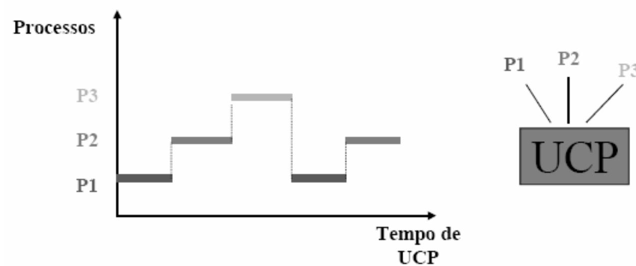


Figura 7. Processos em um Sistema Logicamente Paralelo (Pseudoparalelismo) (TANENBAUM, 2001).

As arquiteturas síncronas coordenam suas operações concorrentes sincronamente em todos os processadores, ou seja, utilizam-se relógios globais, unidades únicas de controle ou controladores de unidade vetorial. Apresentam pouca flexibilidade para a expressão de algoritmos paralelos. Esse grupo é composto pelas arquiteturas SIMD da taxonomia de Flynn, pelas arquiteturas vetoriais e sistólicas (DUNCAN, 1990).

Nas arquiteturas assíncronas, não há controle centralizado mantido por *hardware*, isto é, os processadores podem operar de maneira autônoma. Esse grupo é composto, basicamente, pelas arquiteturas MIMD da taxonomia de Flynn, sejam elas convencionais (MIMD com memória distribuída ou compartilhada) ou não-convencionais (MIMD/SIMD, Fluxo de Dados, Redução ou Dirigidas à demanda e Frente de Onda) (DUNCAN, 1990).

Apesar de a taxonomia de Duncan ser bastante abrangente, grande parte dos autores (TANENBAUM, 1999)(STALLINGS, 2003)(GRAMA *et al.,* 2003) prefere utilizar a taxonomia de Flynn, porquanto enfatizar apenas as arquiteturas paralelas mais utilizadas atualmente. Na Figura 8, baseando-se em todas as informações citadas anteriormente, estão

relacionadas as taxonomias elaboradas por Duncan, Tanenbaum e Flynn.

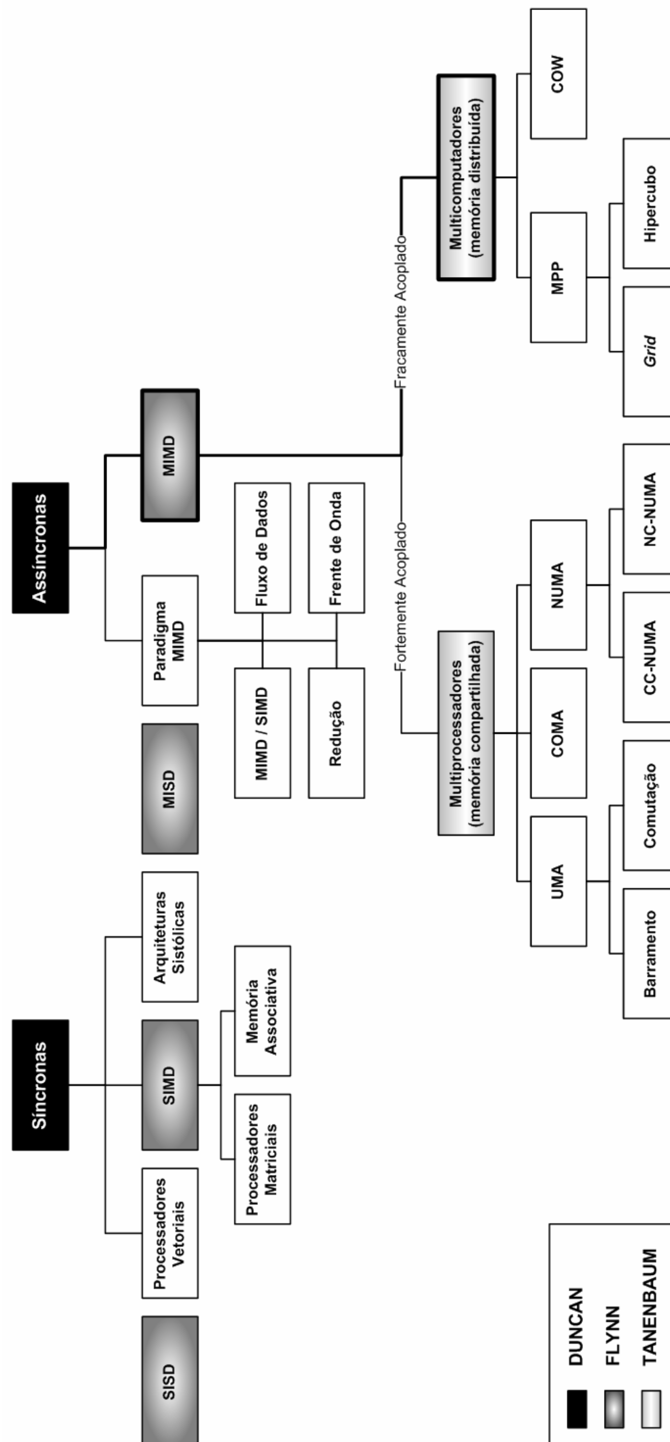


Figura 8. Relacionamento Existente entre as Taxonomias.

As redes de comunicação entre os *hosts* podem variar tanto relacionadas ao tipo (barramento ou comutação) quanto à dimensão (redes locais: LAN – *Local Area Network*; redes metropolitanas: MAN – *Metropolitan Area Network*; ou redes muito amplas: WAN –

Wide Area Network). A interconexão dessas redes pode formar um sistema computacional distribuído se apresentar algumas características, a serem detalhadas na seção 2.2.

2.2. Sistemas Distribuídos

Encontram-se muitas definições de sistemas distribuídos na literatura, como a de Tanenbaum, que define sistemas distribuídos como um conjunto de computadores independentes que aparecem para os usuários do sistema como um único computador (TANENBAUM, 1995); e de Coulouris *et. al.*, que define como um sistema em que componentes de *hardware* e *software* localizados em computadores em rede se comunicam e coordenam suas ações por meio de passagem de mensagens (COULOURIS *et al.*, 2001).

Ao interpretar essas definições, conclui-se que os sistemas distribuídos permitem agrupar a potência computacional de diversos hosts interligados por uma rede de comunicação, para processar colaborativamente determinada tarefa de forma coerente e transparente, ou seja, como se apenas um único e centralizado computador estivesse executando a tarefa.

Essas definições implicam o *hardware* formado por máquinas autônomas e o *software*, fornecendo a abstração de uma única máquina. Dessa maneira, podem trazer algumas vantagens (TANENBAUM, 1999):

- econômica: aproveita máquinas potencialmente ociosas com microprocessadores, pois oferecem melhor relação preço/desempenho que os supercomputadores;
- velocidade: um sistema distribuído pode ter uma potência de processamento maior que um supercomputador;

- distribuição inerente: algumas aplicações são distribuídas por natureza;
- confiabilidade: em caso de falha de uma máquina, o sistema como um todo pode continuar executando a aplicação, apresentando possivelmente uma degradação de desempenho;
- crescimento incremental: há possibilidade da potência computacional aumentar com a inclusão de novos equipamentos.

Embora os sistemas distribuídos tenham muitas vantagens em relação aos centralizados, existem alguns aspectos a serem analisados com cuidado antes de empregá-los. São eles (TANENBAUM, 1999):

- *software*: pouca disponibilidade de *software* para sistemas distribuídos;
- segurança: dificuldades para evitar acesso indevido;
- rede de intercomunicação: pode não dar vazão à demanda.

Segundo alguns autores (MULLENDER, 1993)(TANENBAUM, 1995) (COULOURIS *et al.*, 2001), em aspectos de projeto, os sistemas distribuídos devem apresentar as seguintes características:

- transparência: figura-se como se apenas um único e centralizado computador estivesse executando a tarefa;
- flexibilidade: representa a facilidade de fazer reconfigurações (inter-operabilidade);
- confiabilidade: que caracterizam-na a disponibilidade, tolerância a falhas e a segurança;
- *performance*: é a avaliação do paralelismo e da comunicação para obter o nível de granularidade;

- escalabilidade: refere-se à capacidade do sistema expandir-se com o mínimo de degradação de desempenho.

Os sistemas distribuídos podem ser homogêneos ou heterogêneos. Sistemas homogêneos são compostos de multicomputadores interconectados por uma rede, onde todos utilizam a mesma tecnologia, ou seja, todos os elementos de processamento possuem a mesma potência de processamento, de memória, de disco, entre outros. Por outro lado, os sistemas heterogêneos são mais comuns e possuem uma grande diversidade de tecnologia em relação à rede de comunicação, *hardware*, *software*, linguagens de programação, entre outros (COULOURIS *et al.*, 2007).

Os processos em um sistema distribuído, assim como nos sistemas centralizados, precisam comunicar e sincronizar suas ações. Devido à ausência de memória compartilhada, mecanismos como semáforos e monitores não podem ser utilizados. Nos sistemas distribuídos a comunicação entre processos (IPC - *InterProcess Communication*) é realizada por meio de passagem de mensagens, utilizando o suporte provido pela rede (TANENBAUM e VAN STEEN, 2002).

Embora sistemas para computação paralela e distribuída tenham sido tradicionalmente desenvolvidos com objetivos diferentes, o aumento na velocidade das redes de comunicação de uso geral tem possibilitado o uso de sistemas distribuídos para o processamento paralelo de diversos problemas (KUNG *et al.,* 1991)(GRAMA *et al.,* 2003). Da mesma forma, técnicas para o processamento paralelo podem muitas vezes oferecer benefícios para sistemas distribuídos. Na seção 2.3, descrevem-se as características, vantagens e desvantagens da convergência da programação paralela com os sistemas distribuídos, e a origem da computação paralela distribuída.

2.3. Computação Paralela Distribuída

A utilização dos microcomputadores com um só processador – onde o modelo de Von Neumann prevaleceu desde o princípio –, aliada à disponibilidade de tecnologias de redes de computadores de alto desempenho (sistemas distribuídos), motivou a origem da computação paralela distribuída.

A utilização da computação paralela distribuída aumenta diariamente e pode ser aplicada às mais diversas áreas, na solução de problemas simples ou complexos, com o propósito de obter um retorno mais rápido do que se estivessem sendo solucionados de forma seqüencial.

Para que um sistema computacional possa ser considerado um sistema paralelo distribuído, faz-se necessária a utilização de bibliotecas (APIs - *Application Program Interface*) com rotinas para passagem de mensagens (exemplo: PVM - *Parallel Virtual Machine* e MPI - *Message Passage Inteface*). Utilizando-se dessas bibliotecas, tem-se a chamada máquina paralela virtual que provê transparência para o usuário (BEGUELIN, 1994)(PITANGA, 2003).

Os ambientes de passagem de mensagens não foram desenvolvidos especificamente com o intuito de utilizar os sistemas distribuídos para o desenvolvimento de aplicações paralelas. Desenvolveram-se inicialmente para máquinas com processamento maciçamente paralelo (*Massively Parallel Processing* - MPP), onde, devido à ausência de um padrão, cada fabricante desenvolvia seu próprio ambiente sem se preocupar com portabilidade.

Com o passar dos anos, muita experiência foi adquirida, porquanto os diversos projetos de interfaces de passagem de mensagens enfatizavam aspectos diferentes para o seu sistema. E essas bibliotecas passaram a ser aplicadas com o propósito de utilizar

computadores pessoais para a composição da máquina paralela virtual (MCBRYAN, 1994).

Na Figura 9, é possível observar que não existe endereçamento global: cada processador tem sua memória local (fracamente acoplado). Dessa forma, a memória não é acessada diretamente por todos os processadores do sistema, o que acarreta um custo mais elevado de comunicação (trocas de mensagens) e de sincronização.

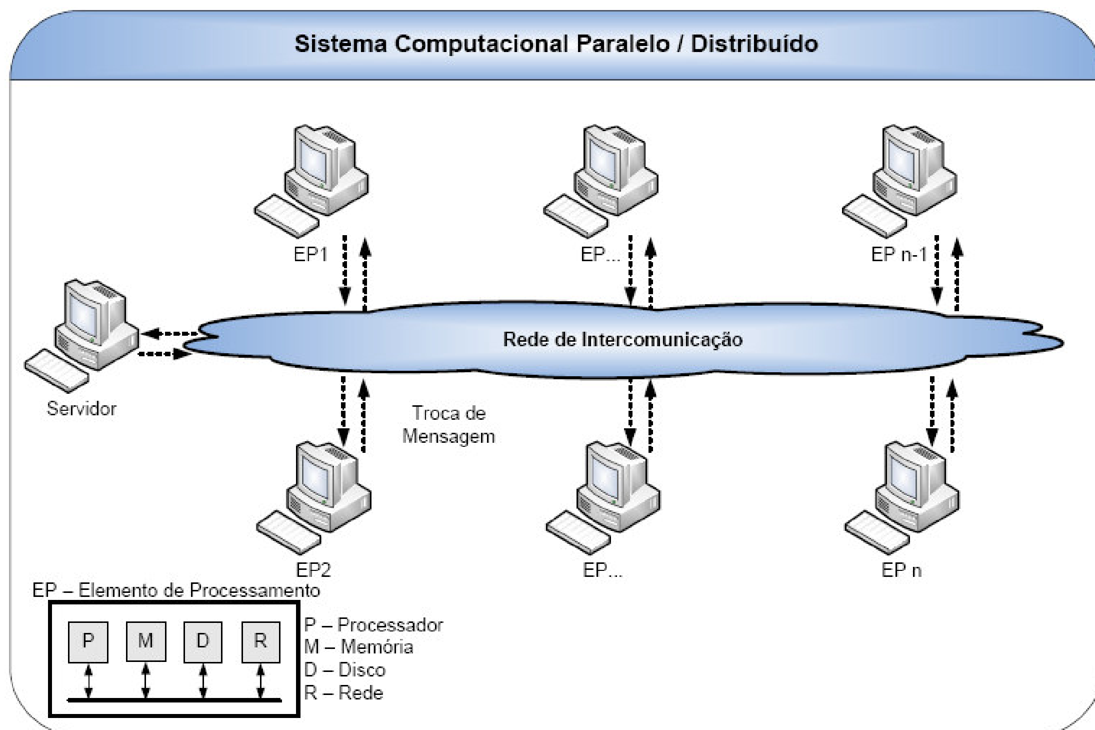


Figura 9. Sistema Computacional Paralelo Distribuído.

Sabendo-se que os sistemas computacionais distribuídos têm convergido para a busca de um melhor desempenho e, para que esse objetivo seja alcançado, não basta apenas existirem sistemas com grande número de processadores ou de estações de trabalho, fica clara a necessidade de garantir que todos os *hosts* pertinentes ao sistema sejam bem aproveitados (BRANCO, 2004). Para isso, a aplicação é dividida em processos (tarefas ou *tasks*), e estes, por sua vez, são distribuídos entre os *hosts* que compõem a máquina paralela virtual. Este procedimento é chamado de escalonamento de processos e é detalhado na seção 2.3.1.

2.3.1. Escalonamento de Processos

Quando mais de um processo precisa ser executado em um uniprocessador, o sistema operacional precisa decidir qual deles deve ser executado primeiro. Denomina-se escalonamento de processos a escolha de qual processo deve executar naquele determinado instante (TANENBAUM, 2001).

Em ambientes paralelos distribuídos, o escalonamento consiste em atribuir processos aos *hosts* e determinar em que ordem esses processos serão executados (REWINI *et al.,* 1995). O escalonador é de vital importância para sistemas paralelos distribuídos e pode ser considerado um dos problemas mais desafiantes nessa área (SHIVARATRI *et al.,* 1992).

Segundo Casavant e Kuhl (CASAVANT e KUHL, 1988) o escalonamento refere-se à atividade de alocar os recursos disponíveis entre as tarefas que compõem cada aplicação. Por conseguinte, o escalonador é responsável pelo gerenciamento dos recursos (processador, memória, disco, rede, entre outros) e, ao utilizar as regras de uma política de escalonamento, o escalonador atribui aos recursos disponíveis os consumidores (processos ou tarefas que compõem uma aplicação).

2.3.1.1. Componentes de um Algoritmo de Escalonamento

Tanto sistemas paralelos distribuídos quanto sistemas maciçamente paralelos necessitam de técnicas para escalonamento de processos e, em ambos, pode ocorrer degradação de desempenho pelo congestionamento da comunicação, caso ocorra

escalonamento inadequado (ZALUSKA, 1991).

O escalonador de processos faz sua decisão baseado em uma política de escolha, utilizando os algoritmos de escalonamento e implementando mecanismos, ou seja, as políticas determinam quais, quando e como os mecanismos serão empregados para que o escalonamento seja efetuado (SHIVARATRI *et al.,* 1992)(BRANCO, 2004).

Nas políticas e nos mecanismos são definidos os objetivos do escalonamento. Dentre os objetivos, podem-se destacar a diminuição do tempo médio de resposta, a diminuição dos atrasos na comunicação, a maximização da utilização dos recursos disponíveis, e o balanceamento das cargas entre os elementos de processamento.

As políticas de escalonamento definem critérios e regras para a ordenação das tarefas a serem realizadas para que ocorra o escalonamento. Dividem-se da seguinte forma: política de transferência, política de seleção, política de localização, e política de informação (SHIVARATRI *et al.,* 1992).

A política de transferência determina se um *host* está capacitado a participar de uma transferência como transmissor ou como receptor, conforme a carga do mesmo.

A política de seleção está relacionada à escolha de qual tarefa será transferida após a definição de qual *host* será o transmissor (geralmente é a tarefa iniciada mais recentemente).

Assim que tenha decidido qual *host* é emissor ou receptor, a política de localização se responsabiliza por definir quais *hosts* serão parceiros de transferência.

A política de informação é responsável por decidir em que momento as informações a respeito do estado dos *hosts* devem ser coletadas, de onde serão coletadas, e quais informações serão coletadas (utilizadas). Existem três tipos de políticas de informação:

- política orientada à demanda: onde uma máquina coleta o estado das demais somente quando ela se torna emissora ou receptora;
- política periódica: as informações são coletadas de tempos em tempos;

- política orientada à mudança de estado: as informações das máquinas são coletadas de acordo com a mudança de seu estado.

Os mecanismos respondem pela definição de como o escalonamento deverá ser efetuado. Dividem-se em três categorias: mecanismo de métrica da carga, mecanismo de comunicação da carga e o mecanismo de migração.

O mecanismo de métrica da carga define o método utilizado para medir a carga de cada elemento de processamento. O mecanismo de comunicação da carga determina o método por meio do qual será efetuada a comunicação das informações de cargas entre as máquinas disponíveis. E o mecanismo de migração delibera o protocolo que deverá ser utilizado quando ocorrer migração de processos entre máquinas (SHIVARATRI *et al.*, 1992)(BRANCO, 2004).

A classificação hierárquica da composição dos algoritmos de escalonamento é representada na Figura 10.

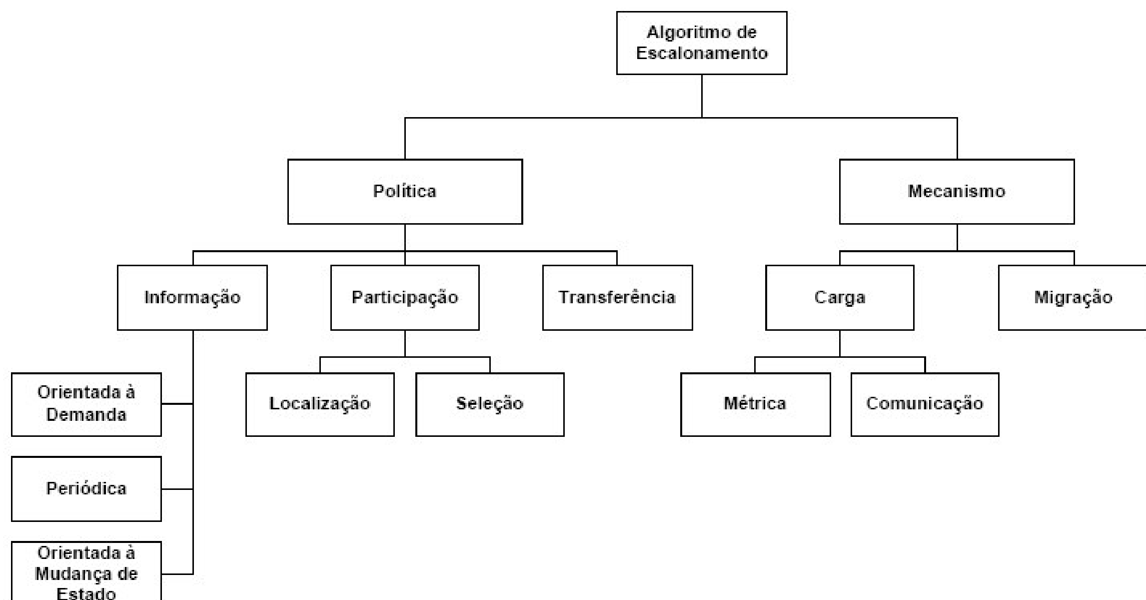


Figura 10. Classificação Hierárquica da Composição dos Algoritmos de Escalonamento (BRANCO, 2004).

Grande parte das políticas de escalonamento tem levado em consideração as características da aplicação executada, procurando tratar adequadamente aplicações com diferentes requisitos. As aplicações normalmente consideradas no escalonamento de aplicações paralelas distribuídas são (FERRARI e ZHOU, 1987)(KUNZ, 1991)(ZHOU *et al.,* 1993):

- *CPU-Bound:* aplicações que possuem alta demanda por processamento e pouca atividade de entrada e saída (I/O);
- *Memory-Bound:* aplicações que possuem alta demanda por memória;
- *I/O-Bound:* aplicações que possuem alta demanda por entrada e saída. Essas aplicações podem ser chamadas também de aplicações *Disk-Bound*, que possuem alta demanda por disco, ou seja, necessitam muito acesso ao disco, tanto para leitura quanto para escrita;
- *Network-Bound:* aplicações que possuem alta demanda por comunicação entre processos, o que implica grande tráfego na rede.

2.3.1.2. Taxonomia de Escalonamento

As classificações e taxonomia dos escalonadores de processos não possuem um padrão a ser seguido, portanto os algoritmos de escalonamento podem ser classificados de várias maneiras. Para unir o maior número possível de características existentes dos algoritmos de escalonamento, diversos autores têm sugerido taxonomias para a área de escalonamento de processos (CASAVANT e KUHL, 1988)(SOUZA, 2004).

A estratégia de escalonamento baseia-se no tipo de informações utilizadas para que

as tarefas sejam escalonadas, no local onde as tarefas serão alocadas quando efetuado o re-escalonamento, na forma em que serão efetuadas as tomadas de decisão (centralizadas e distribuídas), e na obtenção das informações (CASAVANT e KUHL, 1988)(BRANCO, 2004).

A taxonomia proposta por Casavant e Kuhl (CASAVANT e KUHL, 1988) se destaca por ser mais abrangente e de grande aceitação. Divide-se em uma classificação hierárquica, conforme apresentado na Figura 11, e em classificação plana, em que os termos se relacionam de modo independente.

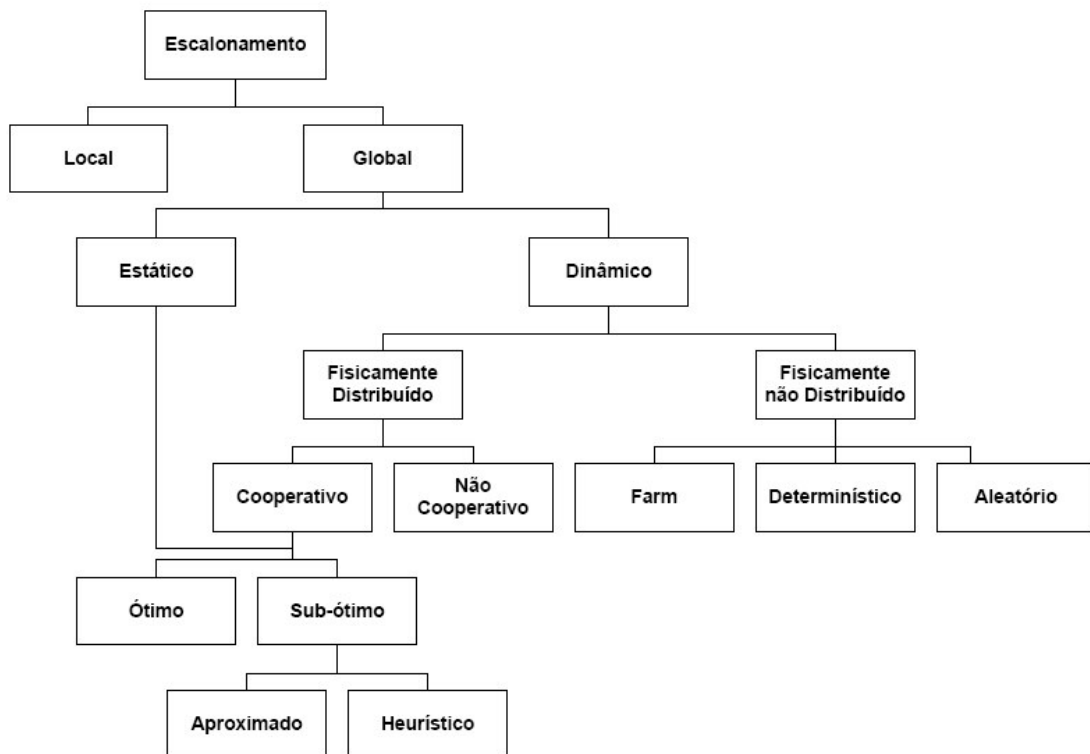


Figura 11. Taxonomia de Escalonamento - Classificação Hierárquica (CASAVANT e KUHL, 1988).

A classificação hierárquica é dividida em escalonamento local e global.

No escalonamento local, os períodos de tempo de utilização de um único *host* são atribuídos aos processos em execução por meio da política de compartilhamento de tempo

(*time-sharing*). Portanto, o tempo é segmentado em pequenas fatias chamadas de *time-slices* (ou *quantum*) (TANENBAUM, 2001).

Em um sistema com vários elementos de processamento (multiprocessador ou multicomputador), o escalonamento global é a atividade que determina para qual elemento o processo será alocado e executado. Essa atividade pode ser feita de forma centralizada, em que as tarefas e as responsabilidades da distribuição são feitas por apenas um processador, ou de forma distribuída, em que as tarefas e responsabilidades são feitas por diversos *hosts*.

A estratégia de distribuição fisicamente distribuída é a alternativa preferida quando as tarefas podem ser iniciadas em qualquer elemento do sistema (BRANCO, 2004).

O escalonamento global divide-se em estático e dinâmico, isso de acordo com o momento em que as decisões do escalonador são tomadas.

No escalonamento estático, tomam-se as decisões no momento da compilação da aplicação (antes que se inicie a execução). Para que isso seja possível, faz-se necessário que as informações acerca do sistema estejam disponíveis, ou seja, há um conhecimento prévio do tempo de execução das tarefas e dos recursos de processamento.

Esta técnica de escalonamento não permite preempção. A modelagem da aplicação pode utilizar grafos, programação matemática e teoria das filas. Para resolver um modelo são empregados algoritmos que buscam uma solução ótima ou, quando algumas informações não estão disponíveis, sub-ótima. As soluções sub-ótimas empregam o uso de heurísticas ou resultados aproximados na busca de uma solução aceitável para o problema (CASAVANT e KUHL, 1988).

No escalonamento dinâmico, há pouca informação a respeito dos recursos necessários, desconhece-se o ambiente em que a aplicação será executada, ou as características do ambiente podem mudar dinamicamente. As informações sobre o estado do sistema são coletadas em tempo de execução para auxiliar nas decisões do escalonamento,

dando maior flexibilidade e aumentando o desempenho global. Neste caso, o escalonamento dinâmico é preemptivo, pois permite que, por exemplo, haja re-alocação de cargas entre os elementos de processamento durante a execução da aplicação.

Quando as decisões do escalonamento dinâmico são fisicamente distribuídas entre diversos elementos de processamento, o escalonamento pode ser cooperativo ou não cooperativo. No cooperativo, os escalonadores interagem durante a alocação de recursos na busca da melhor alocação dos recursos do sistema. Também podem ser classificadas como soluções ótimas ou sub-ótimas. As soluções sub-ótimas empregam o uso de heurísticas ou resultados aproximados na busca de uma solução aceitável para o problema. No escalonamento não cooperativo os mesmos trabalham isoladamente, tendo autonomia para decidir como alocar seus próprios recursos (CASAVANT e KUHL, 1988).

A classificação plana engloba características do escalonamento que não têm relação de hierarquia (são independentes) como: a atribuição de tarefas (atribuição inicial e re-atribuição dinâmica), o compartilhamento de carga, o balanceamento de carga, e o escalonamento adaptativo (CASAVANT e KUHL, 1988)(SHIVARATRI *et al.*, 1992).

Na atribuição inicial, um processo inicia e termina sua execução no mesmo elemento de processamento para o qual foi atribuído (mesmo sob altas variações de carga). Na re-atribuição dinâmica, um processo que já foi atribuído a um elemento de processamento e, possivelmente, já iniciou sua execução, pode ser migrado para outro elemento de processamento.

O compartilhamento de carga (*load sharing*) e o balanceamento de carga (*load balancing*) são mecanismos utilizados para a distribuição de carga. O compartilhamento de carga busca evitar que os elementos de processamento fiquem sobrecarregados, enquanto outros elementos de processamento possam estar disponíveis. O balanceamento de carga estende o mecanismo de compartilhamento de carga, buscando manter a mesma carga em

cada elemento de processamento (SHIVARATRI *et al.*, 1992). Pode-se obtê-lo por meio do escalonamento com re-atribuição dinâmica (CASAVANT e KUHL, 1988). Na seção 2.3.1.3 abordam-se mais detalhes a respeito do balanceamento de cargas.

Os algoritmos de escalonamento adaptativos estão capacitados a se adaptarem dinamicamente às mudanças no contexto do ambiente, em função dos estados anteriores e atuais do sistema, para que exista uma referência à tomada de decisões. Em outros termos, utiliza o histórico dos estados para a tomada de decisão, e suas políticas de escalonamento são alteradas também em função desse histórico.

O termo adaptativo pode-se confundir com o termo dinâmico, mas denotam conceitos distintos. O escalonamento dinâmico utiliza o estado atual do sistema, ao passo que o adaptativo utiliza, por meio da avaliação do histórico de estados das cargas, os parâmetros que julga serem consistentes e relevantes para a tomada de decisão (PITANGA, 2003).

2.3.1.3. Balanceamento de Carga

Em um ambiente paralelo distribuído, na maioria das vezes, pode acontecer de alguns *hosts* completarem suas tarefas antes de outros, tornando-se ociosos, por conseguinte. A causa geradora dessa ocorrência pode-se relacionar ao fato de alguns serem mais rápidos que os demais, e/ou a carga não ter sido distribuída de forma balanceada e coerente com o potencial do *host*.

O escalonamento de processos tem como um de seus objetivos propostos o balanceamento de carga – o que consiste em selecionar uma tarefa e definir o local onde esta será executada (ZALUSKA, 1991), buscando manter a mesma carga em cada *host* (SHIVARATRI *et al., 1992).*

Quando se almeja o balanceamento de cargas, é extremamente indesejado, que existam alguns processadores com carga de processamento muito elevada e outros em estado quase ocioso, assim como nos exemplos da Figura 12 e da Figura 13.

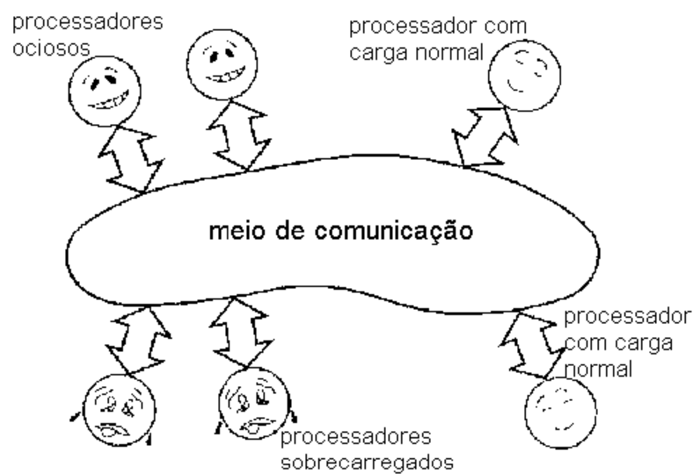


Figura 12. Sistema Distribuído sem Balanceamento de Carga (SHIVARATRI *et al., 1992).*

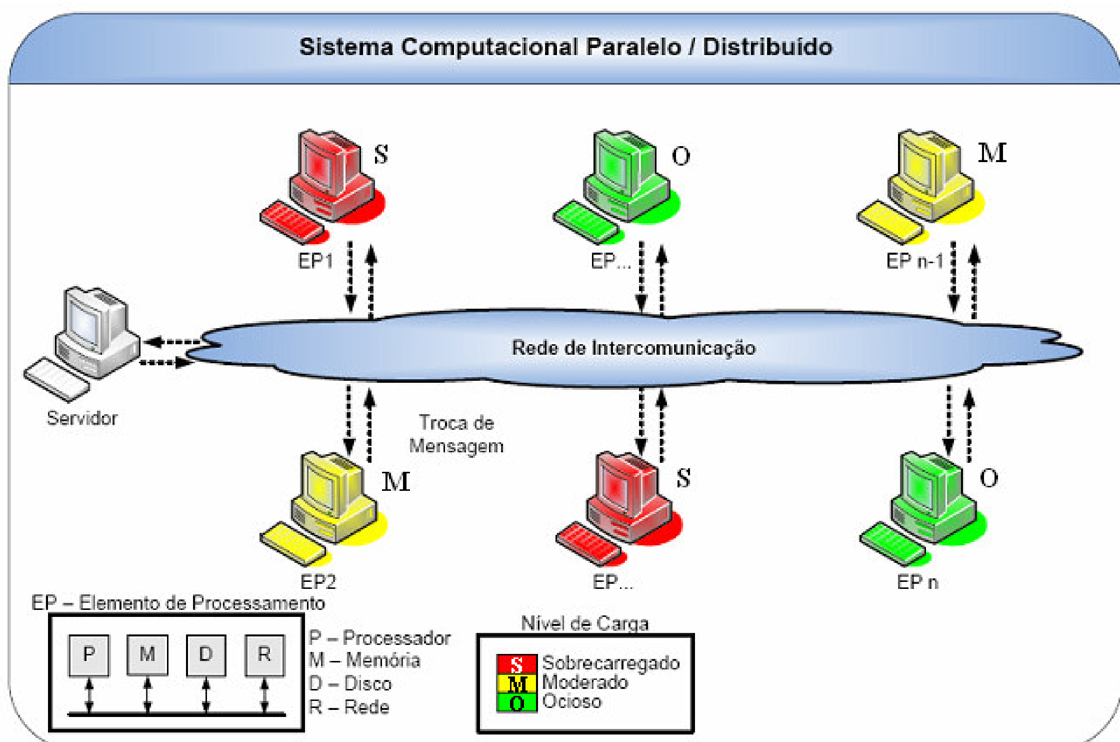


Figura 13. Sistema Computacional Paralelo Distribuído com carga desbalanceada.

Apresentam-se como fator que pode dificultar a execução de um mesmo código em

diferentes máquinas tanto a heterogeneidade arquitetural quanto a heterogeneidade configuracional, pois as características das máquinas são diferentes, fato que pode torná-las incompatíveis, ou até mesmo gerar inconsistências.

Deve-se considerar a heterogeneidade ao efetuar uma alocação de tarefa, uma vez que existem *hosts* com diferentes potências computacionais. É imprescindível escolher o que melhor se aplica às restrições de determinada tarefa (BRANCO, 2004).

Para obter o melhor aproveitamento em ambientes heterogêneos a distribuição de carga deve ser suficiente e compatível com a capacidade total do sistema de maneira que todos os *hosts* tenham uma carga equilibrada entre si (nem sobrecarregados, nem ociosos). Para isso, o escalonador de processos faz a distribuição de processos (tarefas) entre os *hosts* e utiliza o balanceamento de carga para diminuir o efeito das diferenças de velocidade e da capacidade dos *hosts* heterogêneos. O resultado desta administração e redistribuição é uma possível melhora do desempenho e da eficiência de execução da aplicação (SHIVARATRI *et al.,* 1992).

No escalonamento dinâmico realizam-se (re)distribuições de processos nos *hosts* segundo o que este elemento pode oferecer de *hardware* e de *software*. Adota-se como objetivo do escalonamento de processos o balanceamento de carga entre os elementos, com o objetivo de melhorar o desempenho e a eficiência da aplicação. Esses métodos realizam a distribuição de processos entre os *hosts* quando possui sobrecarga, ou mesmo ociosidade, retirando o trabalho dos que estão sobrecarregados e transferindo para os que possuem menos tarefas. Decorre daí uma rede balanceada. Esse escalonamento pode ser classificado como centralizado e descentralizado.

Conseguir essa distribuição ideal não é uma tarefa simples e caracteriza o problema de balanceamento de carga (PLASTINO, 2000). Várias são as causas da falta de balanceamento de carga entre os *hosts* quando se executa uma aplicação paralela em um

sistema computacional distribuído. Entre elas, podem-se citar: a falta de conhecimento sobre a carga de trabalho que envolve cada tarefa, a criação dinâmica de novas tarefas, a variação da carga externa à aplicação em um ambiente não dedicado, além da própria heterogeneidade da arquitetura e do sistema operacional (CASAVANT e KUHL, 1988)(BRANCO, 2004).

2.4. Considerações Finais

Neste capítulo fez-se uma revisão bibliográfica sobre sistemas paralelos, sistemas distribuídos e uma síntese da convergência de ambos – denominada computação paralela distribuída.

Citaram-se e classificaram-se características desses ambientes. Além disso, enfatizou-se o escalonamento de processos, seus componentes, suas taxonomias e seus objetivos, em especial ao balanceamento de carga e suas dificuldades.

Importa destacar que, independentemente do tipo de escalonamento a ser considerado (estático, dinâmico, entre outros), um elemento essencial para a política de escalonamento é a aferição da carga, o que é efetuado por meio dos índices de carga e de desempenho, a serem abordados no próximo capítulo.

CAPÍTULO 3. ÍNDICES DE CARGA E DE DESEMPENHO

Os sistemas computacionais, em geral, surgiram para auxiliar e agilizar as execuções e as soluções de tarefas. Para que não haja gastos desnecessários e tempo perdido, visa-se, sobremaneira, ao desempenho de cada máquina que compõe o sistema, bem como o desempenho do sistema como um todo.

A obtenção de desempenho de sistemas computacionais, simples ou complexos, é uma área que oferece grande abertura para exploração por não ser trivial. Uma avaliação feita sem critérios pode causar efeito contrário ao esperado (JAIN, 1991).

Para avaliar o desempenho de um sistema, podem-se utilizar métricas para aferir e quantificar as cargas de trabalho submetidas a cada recurso dos elementos de processamento (se está sobrecarregado, moderado, ou ocioso). A essa métrica dá-se o nome de índice de carga (FERRARI e ZHOU, 1987)(BRANCO, 2004).

Outra métrica utilizada para obter desempenho, diretamente relacionada ao índice de carga, é o índice de desempenho. Em algumas literaturas, são abordadas como se fossem sinônimos. No entanto, o índice de desempenho analisa, além da heterogeneidade das configurações, a heterogeneidade arquitetural e temporal do ambiente avaliado (BRANCO, 2004).

3.1. Avaliação de Desempenho

A avaliação de desempenho depende do sistema em estudo, da carga de trabalho que lhe é imposta, do algoritmo de escalonamento e da métrica de desempenho (FEITELSON, 2002a)(BRANCO, 2004). Assim, a escolha da métrica adequada tem grande relevância na obtenção de resultados conclusivos de uma avaliação de algoritmos de escalonamento de processos para aplicações paralelas distribuídas (FEITELSON, 2003)(SOUZA, 2004).

Existem dois grandes grupos de técnicas para avaliação de desempenho: as técnicas de modelagem e as técnicas de aferição.

As técnicas de modelagem são utilizadas nos casos em que não se deseja interferir no sistema existente ou, em sistemas que ainda não foram desenvolvidos. Essas técnicas baseiam-se na representação do sistema computacional por meio de modelos, que podem ser resolvidos utilizando soluções analíticas ou soluções por simulação. Os modelos possuem abstrações das características mais relevantes do sistema desejado, e quanto mais objetivo o modelo, mais fácil torna-se sua avaliação de desempenho. As técnicas mais utilizadas são: Redes de Filas, Redes de Petri, *Statecharts* e Cadeia de Markov (ORLANDI, 1995).

As técnicas de aferição são aplicadas a sistemas computacionais reais, sejam completos, sejam seus protótipos. Essas técnicas proporcionam maior precisão, mas podem influenciar no comportamento do sistema, uma vez que disputam por recursos do sistema avaliado. As técnicas de aferição mais utilizadas são: construção de protótipo, coleta de dados e *benchmarking*.

Indica-se a construção de protótipo para obter informações precisas a respeito de um sistema computacional que virá a existir, ou que já existe. Considera-se protótipo uma simplificação do sistema computacional, que constitui-se de suas mesmas funcionalidades e características, podendo possuir dimensões bem menores, ter menor custo e possibilitar maior

facilidade de alteração quando comparado à implementação do sistema computacional real. Entretanto, se comparado com as demais técnicas de aferição, o protótipo pode apresentar maior custo (ORLANDI, 1995).

A coleta de dados é indicada para sistemas computacionais reais e para validar modelos, devendo ser executada de forma muito criteriosa. Devido a isso, é considerada a técnica mais precisa dentre as demais. A coleta pode ser realizada com a utilização de monitores de *hardware* e de *software*.

O *benchmarking*, por sua vez, é indicado para avaliar e comparar as informações obtidas por meio de análises das cargas de trabalho impostas ao *hardware*, *software*, ou mesmo ao sistema computacional completo. Quando utilizado para análise de desempenho do escalonamento de processos, o *benchmarking* impõe uma carga de trabalho. Ao final deste processo é possível saber qual estratégia obteve melhor desempenho sobre determinada carga de trabalho (SOUZA, 2004).

3.2. Carga de Trabalho

Caracteriza-se carga de trabalho (*workload*) pela demanda de recursos dos elementos de processamento, e pode ser classificada como real (natural) ou sintética (artificial) (KUNZ, 1991).

A carga de trabalho real é gerada pelo sistema, e não é possível controlá-la, razão pela qual ela não é frequentemente utilizada para efetuar a caracterização da carga (KUNZ, 1991) apud (BRANCO, 2004).

A carga de trabalho sintética possui características similares à carga de trabalho real, mas pode ser utilizada repetidamente e de maneira controlada, além de poder ser facilmente

modificada sem afetar a operação do sistema, ser facilmente reproduzida e possuir maior flexibilidade (KUNZ, 1991).

Outra vantagem apresentada pela carga de trabalho sintética é a possibilidade de ser elaborada especificamente para o experimento a ser executado. No entanto, para isso, exige que o ambiente seja dedicado ao experimento (FERRARI e ZHOU, 1987) apud (BRANCO, 2004). Isso quer dizer que o objetivo é avaliar o desempenho de uma política de escalonamento voltada às aplicações com grande utilização de CPU. Essa avaliação torna-se inviável para caracterizar outros tipos de carga, como as aplicações voltadas à memória (FEITELSON, 2002b)(FEITELSON, 2007).

Os recursos analisados com maior relevância na literatura são: CPU, memória, disco e a rede de intercomunicação (ZHOU *et al.*, 1993)(BRANCO, 2004)(FEITELSON, 2007).

As cargas do sistema, dependendo da aplicação, alteram-se rapidamente. Portanto, as informações obtidas destas cargas tendem a tornar-se obsoletas. Desse modo, a freqüência com que às informações das cargas são verificadas é outro fator importante no desempenho do sistema.

Analisando-se a carga atribuída ao recurso, observa-se que o recurso é definido ocioso, quando sua carga de trabalho é inexistente, ou possui um valor muito pequeno. Por conseguinte, pode estar apto a receber carga. Quando o recurso é considerado moderado, a carga de trabalho é regular. Decorre daí a possibilidade de o recurso ainda receber alguma carga de trabalho. E, por fim, um recurso é tido como sobrecarregado quando a carga de trabalho a ele imposta ultrapassa um determinado limite e, dessa maneira, ele deve transferir, e não mais estar apto a receber carga.

3.3. Índices de Carga

A informação da carga de trabalho de um recurso é fator importante e decisivo para melhoria do desempenho de um sistema computacional.

O índice de carga é uma métrica que quantifica a carga de trabalho submetida a um recurso do sistema (FERRARI e ZHOU, 1987)(KUNZ, 1991), e tem por objetivo indicar se o recurso analisado está ocioso, moderado, ou sobrecarregado (BRANCO, 2004).

Alguns autores definem índice de carga como uma variável numérica, inteira e não negativa que possui valor zero quando o recurso está ocioso e, à medida que a carga deste recurso aumenta seu valor é acrescido (FERRARI e ZHOU, 1987)(KUNZ, 1991)(BRANCO, 2004).

A qualidade do índice de carga está diretamente relacionada com o desempenho do escalonador de processos, porquanto se volta ao objetivo proposto pelo algoritmo de escalonamento. Dessa forma, o modo de coleta e de utilização das informações das cargas do sistema e a periodicidade com as quais são coletadas influenciam na eficiência do balanceamento de cargas.

Sobremaneira em sistemas paralelos distribuídos, faz-se necessária uma grande cautela com a comunicação e, se a freqüência de verificação das cargas nos elementos de processamento não for coerente com a real necessidade, provavelmente serão obtidos resultados indesejáveis.

Caso a carga seja atualizada com grande freqüência, ocorre uma sobrecarga na rede de interconexão e o desempenho geral do sistema diminui; caso seja pouco atualizada, o balanceamento de cargas não terá as informações corretas e será efetuado de maneira incorreta, prejudicando mais uma vez seu desempenho (BRANCO, 2004).

Por conseguinte, a finalidade do índice de carga é prever o comportamento futuro das

cargas, com base no comportamento atual/passado, e fornecer essa informação para o escalonador, para que o balanceamento de carga seja efetuado.

É possível citar uma grande variedade de índices de carga, entre os quais comprimento da fila da CPU (instantâneo), comprimento médio da fila da CPU em um determinado tempo, utilização da CPU, tempo de resposta, tempo de resposta normalizado, quantidade de memória disponível, taxa de mudança de contexto, entre outros (FERRARI e ZHOU, 1987)(ZHOU *et al.*, 1993)(BRANCO, 2004). De modo geral, esses índices podem-se dividir em grupos baseados no tamanho da fila de acesso ao recurso, no percentual de utilização do recurso e no tempo de execução/resposta (FERRARI e ZHOU, 1987)(ZHOU *et al.*, 1993).

A utilização de um índice de carga pode gerar carga extra no sistema e, em conseqüência, a obtenção de resultados imprecisos. Para que a carga gerada seja a mínima possível, devem-se utilizar índices adequados para cada sistema individualmente (SHIRAZI *et al.*, 1995)

A maioria dos índices de carga existentes na literatura destina-se a ambientes com configurações e arquiteturas homogêneas. Encontra-se pouco na literatura sobre índices de carga para ambientes com heterogeneidade configuracional e homogeneidade arquitetural, e até mesmo para ambientes totalmente heterogêneos.

A heterogeneidade configuracional e arquitetural são fatores que exigem tratamento diferenciado na utilização de índices de carga. Para compreender essa afirmação, utiliza-se o seguinte exemplo na literatura (BRANCO, 2004): um ambiente composto de dois computadores com potências computacionais diferentes e que possuam cargas semelhantes. Em dado instante, um dos computadores apresenta dois processos de baixa ocupação em sua fila de CPU, e o outro, um processo que ocupa cerca de 99% da CPU. Na técnica de índice de carga baseado no comprimento da fila de processos, o resultado no primeiro computador

citado é um índice de valor 2, e no segundo é um índice de valor 1. Por ter obtido um menor índice, o segundo computador recebe os novos processos mesmo não possuindo recursos disponíveis; enquanto isso, o primeiro computador encontra-se semi-ocioso, o que degradará o desempenho do sistema como um todo.

3.3.1. Estado da Arte

Na literatura, há modelos para cálculo de índices de carga consagrados, tanto para ambientes homogêneos quanto para ambientes heterogêneos. Para ambientes configuracionalmente homogêneos e arquiteturalmente heterogêneos, não são encontradas referências na literatura dessas métricas (BRANCO, 2004).

Para ambientes compostos por máquinas configuracionalmente e arquiteturalmente homogêneas, Ferrai e Zhou (FERRARI e ZHOU, 1987) propõem a utilização de um índice de carga, obtido por meio da combinação linear do tempo de serviço $\mathbf{s_j}$ requerido para a execução de uma tarefa em um determinado recurso $\mathbf{r_j}$, em que o comprimento da fila do recurso $\mathbf{r_j}$ é dado por $\mathbf{q_j}$, de tal modo que se obtém o índice de carga por meio da Equação 1, onde **N** representa o número total de recursos que possuem filas.

$$li = \sum_{j=1}^{N} s_j \times q_j$$

Equação 1

Segundo Kunz (KUNZ, 1991), o desempenho obtido pelo escalonador, ao efetuar o balanceamento de carga com qualquer índice de carga linear, é superior ao que não efetua o balanceamento. Por meio de experimentos em ambientes homogêneos, avaliou-se que índices

de carga agregados não acrescentam melhoria no desempenho do sistema quando comparados com índices de carga lineares, e que, por outro lado, ocasionavam sobrecarga na obtenção dos diversos índices lineares que comporiam os índices agregados.

Zhou *et al.* (ZHOU *et al.*, 1993) propõem que os índices de carga variem conforme a natureza do recurso a ser avaliado, isso é, que sejam específicos para cada recurso. Obtêm-se índices por meio da média do comprimento de fila da CPU, quantidade de memória livre, média da taxa de transferência de um disco para todos os outros discos quando efetuado um I/O, quantidade de espaço disponível em disco para troca de páginas e o número de usuários existentes no sistema.

Branco (BRANCO, 2004) propôs um índice de desempenho que leva em consideração a heterogeneidade do sistema, fundado no índice de carga e em uma medida Euclidiana. Por meio de modelagem e simulação foi aplicado e testado esse novo índice na implementação do escalonamento de processos. Os resultados obtidos, analisados estatisticamente, comprovaram que, ao utilizar o novo índice de desempenho em ambientes computacionais paralelos distribuídos heterogêneos pode-se atingir um desempenho superior a ambientes que não o utilizam. Por ser foco de estudo desse trabalho esse índice será melhor explicado na seção 3.4

3.4. Índice de Desempenho

Para suprir a lacuna encontrada na literatura no tocante a índice de carga para ambientes heterogêneos, Branco (BRANCO, 2004) propôs um índice de desempenho que fornece informações da carga de trabalho e da situação de operação de cada um dos *hosts* do sistema, levando em consideração as heterogeneidades configuracional, arquitetural e

temporal (BRANCO, 2004), como pode ser observado na Figura 14.

Figura 14. Lacunas Existentes na Literatura Quando Levado em Consideração os Níveis Arquiteturais e Configuracionais (BRANCO, 2004).

Um bom índice de desempenho, assim como os índices de carga, deve possuir meios de estimar o futuro por meio de valores atuais e fatores do passado. Portanto, para que se possa obter um bom índice de desempenho, suas bases devem estar fundadas nos índices de carga.

Outra característica importante do índice de desempenho é seu tempo de uso, como já observado na seção 3.2. As cargas são voláteis; conseqüentemente, os índices de cargas e de desempenho também o são.

Na Figura 15, apresenta-se uma visão macroscópica do ***M***odelo ***D***e ***Í***ndice de ***D***esempenho em ***A***mbientes ***h***eterogêneos (MEDIDA*h*) proposto por Branco (BRANCO, 2004). Nota-se que, para a compreensão e a obtenção do MEDIDA*h,* faz-se necessário um conhecimento prévio a respeito dos diferentes tipos de aplicações e do *hardware* envolvido (seção 2.3.1.1), bem como dos índices de cargas existentes (seção 3.3).

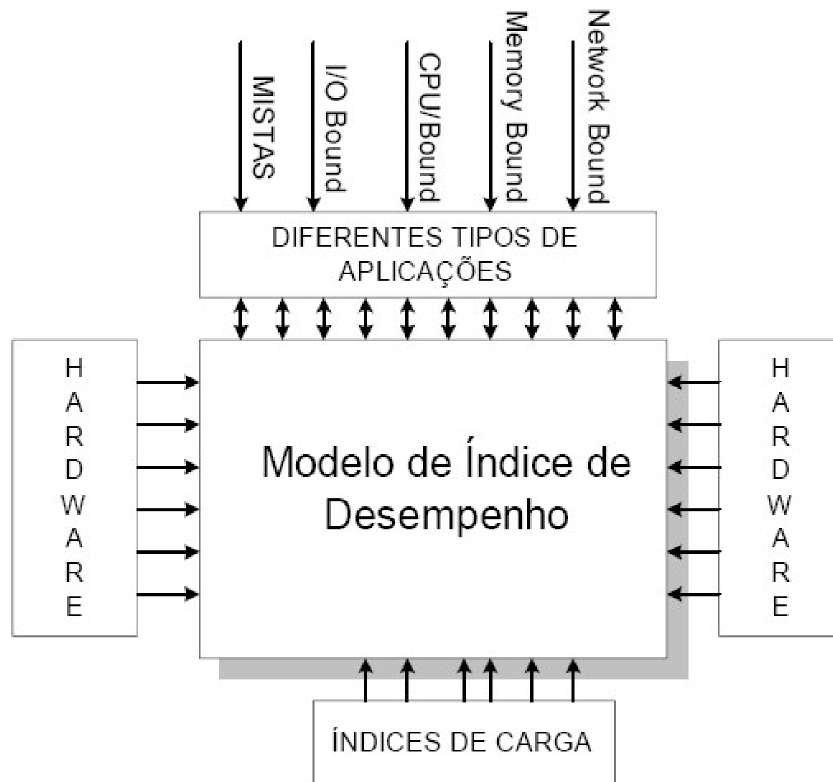


Figura 15. ***M***odelo ***D***e ***Í***ndice de ***D***esempenho em ***A***mbientes ***h***eterogêneos (MEDIDA*h*).

Exibe-se na Figura 16 a estratégia para obtenção do índice de desempenho.

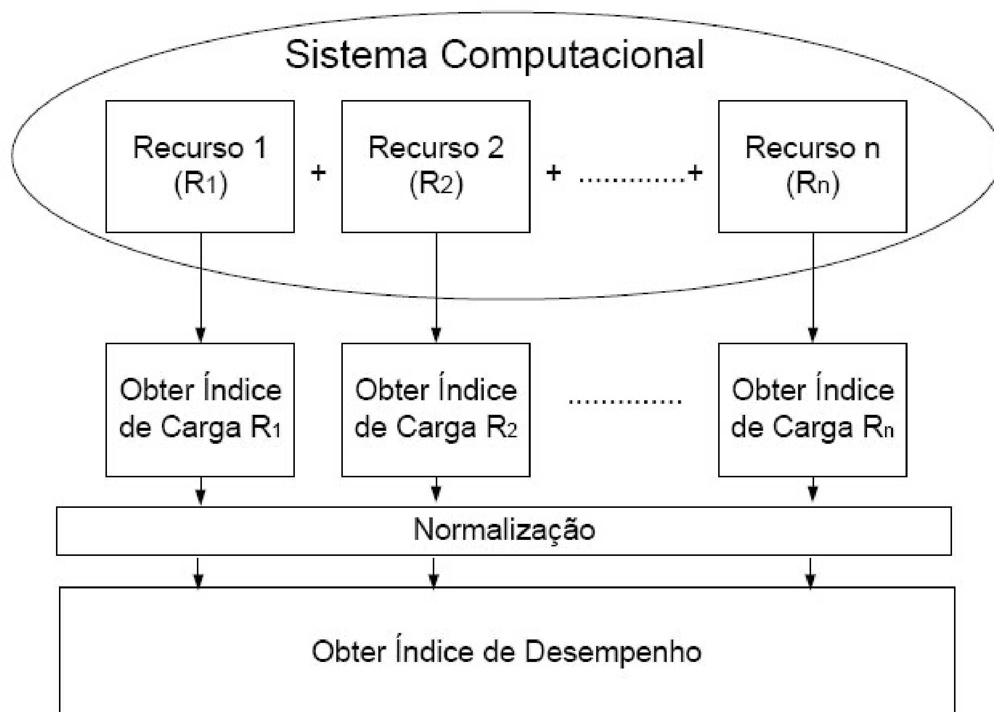


Figura 16. Estratégia para Obtenção do Índice de Desempenho (BRANCO, 2004).

O índice de desempenho baseia-se na distância euclidiana[1] entre o ponto de origem (onde a máquina está ociosa) e o ponto resultante entre os valores de carga da máquina antes de receber uma determinada aplicação e o vetor da carga imposta por essa aplicação. A máquina mais adequada para receber a aplicação é a que obtém a menor distância euclidiana (BRANCO, 2004)(BRANCO *et al.*, 2006). Denomina-se VIP (*Vector for Index of Performance*) o índice baseado em vetor de carga.

Considerando a relação existente entre os diferentes recursos que compõem uma máquina e permitindo que a alocação dos processos se efetue de maneira equilibrada, é possível obter o índice de desempenho (*ID*) como visto na Equação 2.

$$ID = \sqrt{I_{CPU}^2 + I_{Disco}^2 + I_{Memória}^2 + I_{\mathrm{Re}de}^2}$$

Equação 2

onde: *ID* representa o valor do índice de desempenho; o *I* refere-se ao índice de carga do recurso.

No exemplo utilizado por Branco (BRANCO, 2004), ilustrado na Figura 17, adotam-se duas máquinas distintas, mas que estão igualmente carregadas. O recurso 1 está mais carregado na máquina M_1, enquanto o recurso 2 está mais carregado na máquina M_2 em termos de utilização. Um processo P, que utiliza apenas o recurso 1 (R_1 *bound*), pode ser alocado para M_1 e M_2. Para determinar em qual situação se obtém melhor resultado, pode ser adotada a distância Euclidiana entre o ponto e a origem, isto é, o comprimento do vetor da

---

[1] É um espaço vetorial real de dimensão finita munido de um produto interno.

origem ao ponto. Dessa forma, obtêm-se os vetores C_1 e C_2, e o resultado do comprimento C2 é menor que o de C_1. Por conseguinte, o processo P deve se alocado em M_2. Embora as máquinas estejam igualmente carregadas, a distinção de carga quanto aos recursos que estão sendo utilizado e o tipo de tarefa que será alocada permite uma melhor alocação da tarefa (BRANCO *et al.*, 2006).

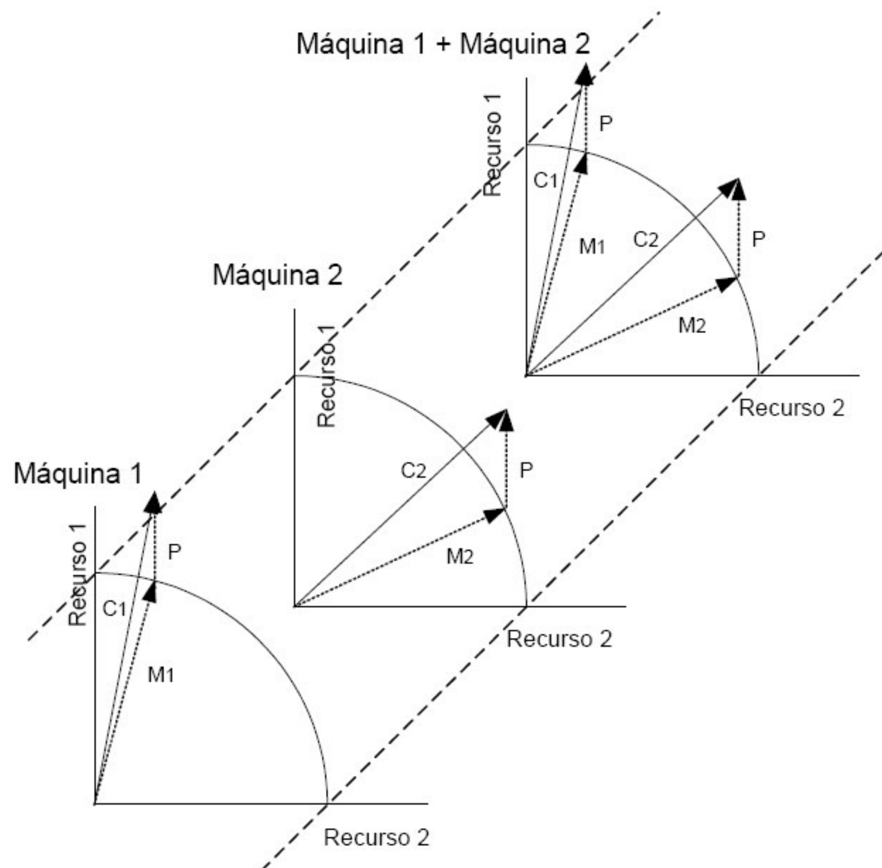


Figura 17. Espaço bidimensional formado pelos recursos 1 e 2, e duas máquinas com cargas iguais (processo limitado por um recurso) (BRANCO, 2004).

Considera-se, na Equação 2, que os pesos de todos os recursos são iguais. Em uma variante do VIP, estabelece-se um peso para cada recurso, denominado PVIP (*Ponderated Vector for Index of Performance*). Deste modo, o PVIP pode apresentar melhores resultados em casos que se tenha conhecimento prévio do tipo de aplicação a ser considerada.

3.5. Considerações Finais

Obter bom desempenho em um sistema computacional é essencial, mas não trivial, uma vez que há dependência de uma série de fatores. Dentre esses fatores, apresentam-se o algoritmo de escalonamento, o modo de coleta e utilização das informações das cargas do sistema, os índices de carga, a periodicidade com que são coletadas, as métricas para quantificar e determinar as cargas, características dos recursos (homogêneos ou heterogêneos), entre outros.

A complexidade desses fatores muda para cada ambiente e, caso ocorram falhas na definição destes fatores, um desempenho não aceitável pode ser obtido.

Dentre os fatores que auxiliam na obtenção de um bom desempenho computacional destaca-se o índice de carga. Caso a utilização desses índices não seja bem estruturada, pode piorar o desempenho ao invés de contribuir para sua melhora.

Uma vez que a coleta de carga e a periodicidade desta coleta afetam o uso dos índices de carga, a utilização de agentes móveis pode prover uma melhora na etapa de coleta e de cômputo desses índices. O estudo dessa utilização é objetivo deste trabalho, destarte, o próximo capítulo apresenta informações relevantes sobre agentes.

CAPÍTULO 4. CÓDIGOS MÓVEIS

Os sistemas distribuídos – em particular os sistemas computacionais paralelos distribuídos – tornaram-se foco de pesquisa, como abordado no CAPÍTULO 2. Em decorrência da evolução, surgiu uma nova abordagem para esses ambientes, denominada sistemas de código móvel (*Mobile Code Systems* - MCS) (CARZANIGA *et al.*, 1997)(FUGGETTA *et al.*, 1998).

Uma das motivações para o surgimento de código móvel baseou-se no escalonamento de processos efetuado em sistemas distribuídos, sobremaneira quando ocorre a migração de processos durante sua execução. Os sistemas computacionais paralelos distribuídos exigem preocupações relacionadas ao tempo de execução, utilização de recursos e comunicação na rede de intercomunicação para essa comunicação não influencie de modo indesejado no desempenho. Como a migração de processos faz o uso da rede de intercomunicação, a utilização de agentes torna-se algo motivador (FUGGETTA *et al.*, 1998)(TANENBAUM e VAN STEEN, 2002).

Na literatura não há consenso em relação a código móvel, metodologia terminológica para sua estrutura, ou mesmo um padrão. Todavia, código móvel é definido informalmente como a capacidade de mudar dinamicamente as ligações entre fragmentos de código e a localidade de sua execução (FUGGETTA *et al., 1998).*

Fuggetta *et al.* propuseram um arcabouço teórico que identifica os diversos mecanismos de mobilidade, que podem ser diferenciados por meio das unidades em execução e pelos recursos (FUGGETTA *et al.*, 1998). Na Figura 18 apresenta-se um esquema resumido

dos mecanismos de mobilidade.

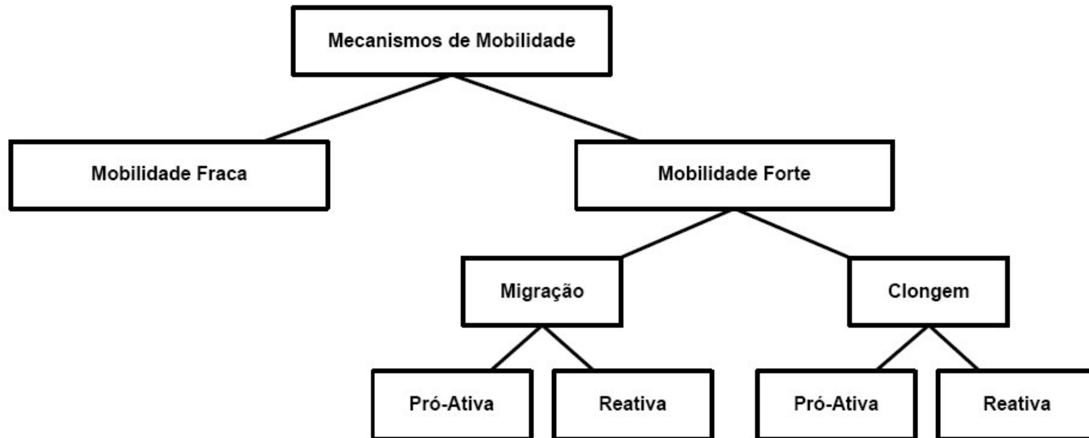


Figura 18. Mecanismos de Mobilidade (FUGGETTA *et al.*, 1998).

O mecanismo de mobilidade mais simples é denominado de mobilidade fraca, em que apenas o código é transferido a diferentes ambientes computacionais. Possivelmente, o código faz-se acompanhar de alguns dados de inicialização, entretanto não há migração de estado de execução.

Denomina-se mobilidade forte ao mecanismo mais complexo, e é indicado para sistemas distribuídos heterogêneos, pois o código e o estado de execução de uma unidade de execução são transferidos para diferentes ambientes computacionais, isto é, o código pode ser interrompido em um *host* e continuar sua execução em outro, a partir do ponto em que foi interrompido (TANENBAUM e VAN STEEN, 2002).

A mobilidade forte é dividida em outros dois mecanismos: migração e clonagem remota. O mecanismo de migração suspende a unidade de execução e transmite-a para o ambiente computacional destino, onde retoma sua execução exatamente do ponto em que foi interrompida. O mecanismo de clonagem remota cria uma cópia exata da unidade de execução em um outro ambiente computacional, não suspendendo a original de seu ambiente, ou seja, o clone executa em paralelo com a unidade de execução original.

Os mecanismos de migração e de clonagem remota podem ser do tipo pró-ativa ou

reativa. Quando a unidade de execução decide de maneira autônoma o momento e o destino da migração, considera-se mobilidade do tipo pró-ativa. Por outro lado, quando a transferência é requerida por outra unidade de execução, diz-se que ocorre a mobilidade reativa.

Existem diversos fatores relacionados ao antes e depois da execução do código móvel, os quais determinam os elementos que compõem o sistema e a forma pela qual se interagem. As interações exigem cuidados com latência, desempenho, tolerância à falhas, entre outras. Os principais paradigmas são: execução remota, código sob demanda, e agentes móveis, sendo este último o de maior relevância para este trabalho. Para compreendê-los, utiliza-se na literatura um exemplo que os compara com o paradigma cliente/servidor de um sistema distribuído. No exemplo, apresentado na Tabela 1, utiliza-se um componente computacional ***A***, localizado no local L_A , que precisa do resultado da execução de uma tarefa, e um componente computacional ***B***, localizado no local L_B , para assistência. Para cada paradigma exibe-se a localização dos componentes antes e depois da realização da tarefa, além de demonstrar qual componente é responsável pela execução das instruções, e o local onde a tarefa de fato se realiza.

Tabela 1. Comparações entre Paradigmas de Código Móvel (FUGGETTA *et al.,* 1998).

Paradigma	**Antes**		**Depois**	
	L_A	**L_B**	**L_A**	**L_B**
Cliente/Servidor	A	instruções	A	instruções
		recursos		recursos
		B		B
Execução Remota	instruções	recursos	A	instruções
	A	B		recursos
				B
Código sob Demanda	recursos	instruções	recursos	B
	A	B	instruções	
			A	
Agentes Móveis	instruções	recursos		instruções
	A		-	recursos
				A

No paradigma cliente/servidor, o componente computacional ***A*** (cliente) faz requisição de serviço ao componente computacional ***B*** (servidor), que possui um conjunto de serviços, ***B*** realiza o serviço utilizando suas instruções e recursos e envia a resposta para ***A***.

Na execução remota, o componente computacional ***A*** detém as instruções necessárias para realizar o serviço, mas não possui recursos; então, envia suas instruções para L_B ; ***B*** executa-as com seus recursos e devolve o resultado ao ***A***.

O código sob demanda, o componente ***A*** possui acesso aos recursos em L_A , mas as instruções necessárias para manipular os recursos se encontram em L_B . ***A*** solicita ao ***B*** que lhe envie suas instruções. Após o envio, ***A*** as executa com seus recursos.

Um agente móvel é representado pelo componente ***A***, inicialmente em L_A . ***A*** possui as instruções, mas é ***B*** que possui o recurso para executá-las. ***A*** migra para ***B***, levando consigo suas instruções e possivelmente o resultado de alguma computação realizada anteriormente. Estando em L_B , ***A*** utiliza o recurso de ***B*** para poder completar ou continuar o serviço.

O paradigma agente móvel mostrou-se eficiente no desenvolvimento de sistemas distribuídos (GHEZZI e VIGNA, 1997). Ao passo que os paradigmas código sob demanda e execução remota têm como foco a transferência de instruções (código) entre os componentes, o agente móvel efetua a transferência de instruções e o estado da execução.

4.1. Agentes

A tecnologia de agentes pode ser aplicada comercial e academicamente nos mais diversos campos, em que se destacam inteligência artificial, engenharia de *software*, e sistemas distribuídos (SELKER, 1994)(HAYES, 1999)(JENNINGS e WOOLDRIDGE, 1998)(WOOLDRIDGE, 1998)(RUSSELL; NORVIG, 2003).

Pesquisadores não chegaram a um consenso a respeito da exata definição de agentes. Após comparações feitas nas definições, Franklin e Grasser elaboraram uma definição extremamente abrangente: um agente é um sistema limitado por um ambiente, que nele percebe e atua continuamente em busca de sua própria agenda, a fim de aplicar o que percebeu em um momento futuro, isto é, capaz de monitorar e executar o controle sobre suas próprias ações a partir de sua própria experiência (FRANKLIN e GRAESSER, 1996).

Os agentes podem ser classificados de acordo com o subconjunto de propriedades que possuem, ou seja, de acordo com as tarefas que cumprem, pelo alcance, sensibilidade, e efetividade de suas ações, pela quantidade de estados internos que possui, ou ainda pelo ambiente onde o agente atua e se localiza.

A Tabela 2 apresenta as principais propriedades dos diferentes tipos de agentes. Maiores detalhes dessas propriedades podem ser encontradas em (FRANKLIN e GRAESSER, 1996) e em (TANENBAUM e VAN STEEN, 2002).

Tabela 2. Principais Propriedades de Agentes (FRANKLIN e GRAESSER, 1996).

Propriedade	**Descrição**
Autônomo	Exerce controle sobre suas próprias ações.
Reativo	Percepção e Ação. Responde conforme as mudanças no ambiente.
Pró-Ativo	Orientado a metas. Não age simplesmente em função do ambiente.
Comunicativo	Pode se comunicar com os usuários ou outros agentes.
Contínuo	Executa continuamente.
Inteligente	Muda seu comportamento com base na sua experiência anterior.
Móvel	Pode se mover de um lugar para outro.
Flexível	Adaptativo. Ações não são definidas através de *scripts*.

4.1.1. Agentes Móveis

Um agente móvel pode ser caracterizado como um programa que age de forma autônoma, em busca de um objetivo programado anteriormente. Para isso, ele pode utilizar interações com outros agentes ou ambientes móveis, se necessário (JANSEN, 2000). Em outras palavras, dispõe de liberdade para transitar entre os *hosts* de uma rede de intercomunicação transportando seu código para outro ambiente, onde retoma sua execução.

A motivação e o interesse pelos agentes móveis devem-se aos benefícios alcançados no desenvolvimento de sistemas distribuídos ao utilizá-los. Em estudos empreendidos por (BIESZCZAD *et al.*, 1998)(LANGE e OSHIMA, 1998)(LANGE e OSHIMA, 1999) encontram-se os benefícios sintetizados a seguir:

- redução no tráfego de rede: os sistemas distribuídos utilizam protocolos de comunicação que geralmente envolvem trocas de mensagens, o que resulta em tráfego indesejado na rede. Com a utilização de agentes móveis pode ser feito um empacotamento das instruções e seu estado, e em seguida sua migração para o *host* destino, onde, dessa forma, as instruções são realizadas localmente. Portanto, ao utilizar agentes móveis, o tráfego pode ser reduzido;
- superação à latência: o fato da redução de tráfego gerada na utilização de agentes móveis tem como conseqüência a superação à latência, por efetuar interações localmente;
- execução autônoma e assíncrona: ao migrar para outro *host,* um agente móvel pode se tornar independente da aplicação que o criou, podendo executar uma tarefa de forma autônoma e assíncrona. Não é necessário manter conexões abertas entre os *hosts*;
- suporte à heterogeneidade: as redes de computadores são, normalmente,

heterogêneas, tanto em perspectiva de *hardware* quanto de *software*. Os agentes móveis são projetados de forma a lidar com essa heterogeneidade, fazendo que o agente seja dependente apenas de um ambiente de execução que é criado em cada combinação de *hardware* e *software*. Portanto, ao utilizar agentes móveis um sistema distribuído ganha uma forma de integrar sistemas heterogêneos;

- robustez e tolerância a falha: as características de execução autônoma e assíncrona e de adaptação dinâmica, favorecem a construção de sistemas tolerantes a falhas, pois permitem que um agente tome uma atitude quando se encontra em situações desfavoráveis. Exemplificando: caso uma máquina precisar ser desligada, seus agentes são avisados para migrarem para outra máquina da rede (espera-se que seja em tempo hábil), para que possam continuar o seu trabalho;
- extensão instantânea de serviços: os agentes móveis podem ser utilizados para estender as capacidades de aplicativos; por exemplo, fornecer serviços. Essa característica permite o desenvolvimento de sistemas extremamente flexíveis.

As desvantagens de agentes móveis listadas na literatura referem-se à segurança e complexidade. Os problemas com segurança podem ser de três tipos: exposição de informação, negação de serviço e corrupção de informação. Exemplo: um agente móvel pode atacar um sistema consumindo uma quantidade excessiva dos recursos da máquina. Esse ataque pode ocorrer intencionalmente explorando vulnerabilidades do sistema, mas também pode ocorrer sem intenção, causado por um erro de programação do agente (JANSEN e KARYGIANNIS, 1999).

Na literatura encontram-se investigações baseadas na utilização de agentes móveis para implementação de diversos mecanismos em ambientes de grande escala, como por exemplo, para o gerenciamento de recursos (CAO *et al., 2002)*(OVEREINDER *et al., 2002)*(MOBACH *et al., 2004)*; descoberta de recursos e serviços (FUKUDA *et al., 2003)*(AVERSA *et al., 2004)*, e escalonamento de tarefas (CHAKRAVARTI *et al.,* 2004).

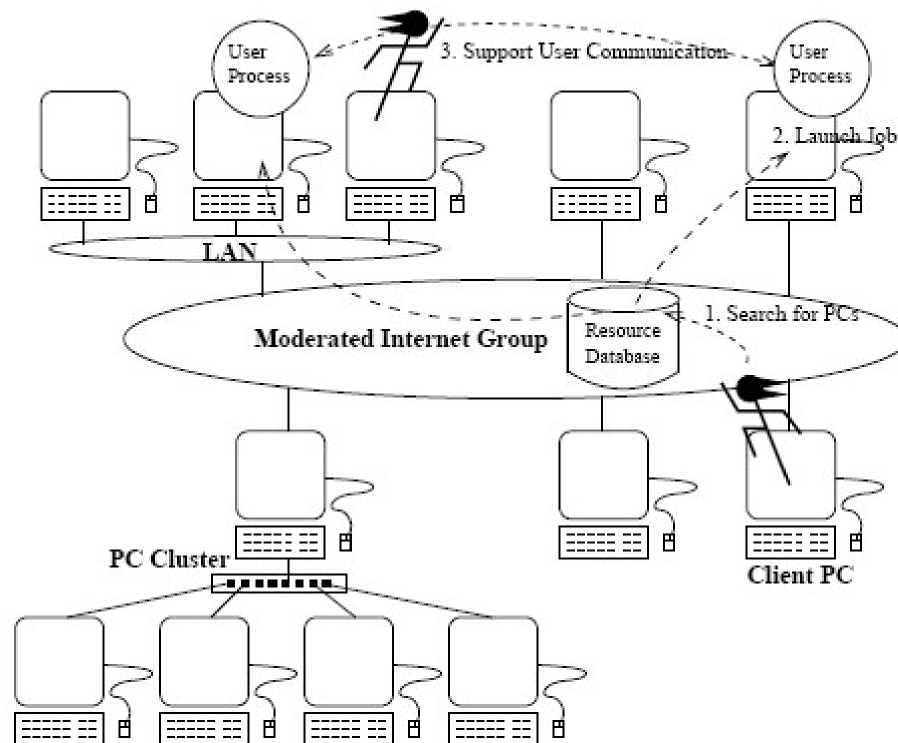


Figura 19. Arquitetura do *CoordAgent* (FUKUDA *et al.*, 2003).

O Laboratório de Sistemas Distribuídos da Universidade de Washington, Bothell desenvolveu um *middleware* de *Grade* que agrega a potência computacional de computadores pessoais conectados à Internet, denominado *CoordAgent* (FUKUDA *et al., 2003).* Sua infra-estrutura é composta por agentes móveis que realizam todas as funções pertinentes à *Grade*, como, a busca por recursos disponíveis e a execução de aplicações. Exibe-se na Figura 19 a arquitetura do *CoordAgent*.

O administrador da infra-estrutura da Grade – chamado de moderador – criou um

Internet Group, que é responsável por manter as informações das máquinas conectadas à *Grade* por meio de uma base de dados. Múltiplos *Internet Group* podem ser criados e conectados, utilizando o MDS (*Globus Metacomputing Directory Service*) (FOSTER e KESSELMAN, 1998). Para participar é necessário criar uma conta de usuário, executar um servidor *web* na máquina que será integrada, registrar as informações no *Internet Group* e baixar o mecanismo de execução de aplicações (desenvolvido com agente móvel).

Depois de entrar no *Internet Group* o usuário requer o agente móvel, que exibe a janela que permite solicitar a execução de aplicações. Estas aplicações podem ter sido desenvolvidas utilizando as linguagens C/C++ e Java. Ademais, o *CoordAgent* permite a execução de aplicações paralelas desenvolvidas com as linguagens de programação C/C++ utilizando MPI ou PVM, e Java utilizando JPVM (*Java Parallel Virtual Machine*).

Ao receber a requisição de execução, o agente móvel migra até a base de dados do *Internet Group*, procura computadores que possam satisfazer a requisição do usuário (cliente). Caso não encontre máquinas com requisitos adequados à execução da aplicação no grupo local, o agente visita bases de dados de outros *Internet Group* para atender à solicitação. Na migração, o agente utiliza a técnica de tunelamento HTTP (*HyperText Transfer Protocol*), em que o agente é entregue como uma mensagem HTTP e, depois, é recriado por um *servlet* no destino (FUKUDA *et al.*, 2003).

Quando o agente móvel encontra a máquina que satisfaz a solicitação do usuário, este copia todos os arquivos necessários à execução da aplicação da máquina cliente para a máquina destino. Caso seja uma aplicação que requeira mais de uma máquina para sua execução, como o caso de uma aplicação paralela, o agente cria novos agentes filhos e os envia às outras máquinas. Assim, todos os agentes lançam as aplicações nas máquinas de destino e passam a monitorar suas execuções.

Utiliza-se um sistema de ponto de verificação (*check-pointing*) que armazena as

máquinas da *Grade* que participaram da execução juntamente com o estado dessa execução. Por conseguinte, caso uma máquina se torne indisponível durante a execução de uma aplicação, o agente deve migrar sua aplicação correspondente para uma outra máquina disponível. Esse processo requer que o estado de execução da aplicação seja capturado, migrado e recuperado no destino. O mecanismo também é utilizado para prover tolerância à falhas, em que o estado de execução é periodicamente salvo em um repositório que resiste a falhas do sistema (FUKUDA *et al.*, 2003).

A Universidade de Ohio desenvolveu um projeto de *Grade* oportunista, chamado de *Organic Grid*, em que propõe uma abordagem completamente descentralizada, sem a presença de componentes que mantenham informações a respeito de todos os recursos presentes no sistema. Além disso, utiliza um esquema de escalonamento autônomo, inspirado na organização de complexos sistemas biológicos. Por meio desse esquema de escalonamento é possível atingir um alto grau de escalabilidade, agregando ao sistema milhões de computadores, o que não é possível quando utilizada uma abordagem centralizada (CHAKRAVARTI *et al.*, 2004)(CHAKRAVARTI *et al.*, 2005a)(CHAKRAVARTI *et al.*, 2005b)(CHAKRAVARTI *et al., 2005c).*

O esquema de escalonamento descentralizado baseou-se em um trabalho anterior proposto por Kreaseck *et al.* (KREASECK *et al.*, 2003). Ambos os trabalhos organizam as computações em redes *overlay* (redes sobrepostas) (DOVAL e O'MAHONY, 2003) estruturadas como árvores.

Redes *overlay* são estruturas lógicas que representam topologias virtuais – não representam a rede como ela está organizada fisicamente. Estruturadas sobre o protocolo de transporte utilizado, facilitam, por exemplo, buscas determinísticas na rede. As redes *overlay* representam, em suas estruturas, os *hosts* da rede, seus vizinhos e as ligações lógicas existentes entre eles. A presença de uma ligação lógica entre dois *hosts* em uma rede *overlay*

indica que eles podem comunicar-se diretamente um como o outro. Essas redes são consideradas uma alternativa escalável às atuais redes *peer-to-peer* (CHAKRAVARTI *et al.*, 2005a).

O *Organic Grid* encapsula as aplicações submetidas à *Grade* em agentes móveis, ou seja, todas as aplicações na *Grade* têm um agente responsável por efetuar sua execução. Os agentes móveis, além de executar as aplicações, permitem que os *hosts* da *Grade* sejam liberados, caso os usuários reclamem o seu uso, por meio da transmissão da aplicação encapsulada juntamente com o estado de execução da mesma para outra máquina da *Grade*. Esta capacidade é fornecida por meio de arcabouço para migração forte de aplicações *multi-threading* (CHAKRAVARTI *et al.*, 2003).

Os agentes móveis do *Organic Grid* também são responsáveis por implementar um esquema de escalonamento autônomo. A árvore *overlay*, que constitui a base do escalonamento no *Organic Grid*, é mantida por estes agentes sendo cada um deles mapeado como um nó da árvore.

O algoritmo de escalonamento funciona do seguinte modo: uma aplicação é submetida a um nó ***A***; este divide a tarefa em sub-tarefas menores e passa a executar uma delas. As outras sub-tarefas são enviadas aos *hosts* amigos do *host* **A** (*host* com endereços previamente conhecidos) para a execução das mesmas. Portanto, formam-se uma hierarquia, pois o nó ***A*** inicia a montagem da árvore *overlay*, sendo ele mesmo sua raiz, e os nós amigos tornam-se seus filhos. A cadeia prossegue com os filhos de ***A*** até que toda a árvore seja montada. Deste modo, os nós que recebem solicitações de execução tornam-se filhos dos solicitantes. Essa estrutura pode ser visualizada na Figura 20.

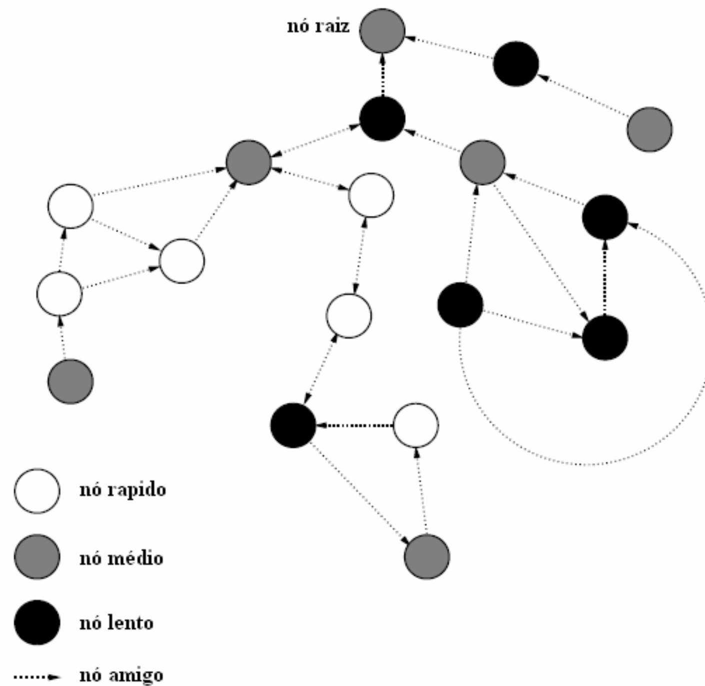


Figura 20. Configurações Iniciais de um *Organic Grid* (CHAKRAVARTI *et al.*, 2005c).

As resultantes das execuções das sub-tarefas são enviadas para o respectivo nó pai, e os nós filhos ainda podem solicitar aos seus pais mais sub-tarefas. Após obter os resultados de todos os filhos, o nó pai deverá escolher o filho que obteve o maior *throughput* – obteve o resultado da execução mais rapidamente – para movê-lo a um nível acima na árvore, juntamente com suas sub-arvores, se existirem. Ao assumir uma nova posição na árvore, este nó ganha de seus novos pais o estado de “filho em potencial” e permanece neste estado até ser avaliado pelo seu novo pai. De maneira análoga, o filho com menor *throughput* será removido da árvore, e poderá, no futuro, tentar voltar a compor a árvore.

Como se pode notar na Figura 21, o algoritmo de escalonamento do *Organic Grid* tende a fazer que os nós mais rápidos fiquem mais próximos ao nó que solicitou a execução da aplicação; e que os nós lentos sejam removidos da estrutura, melhorando assim o tempo de resposta da grade às solicitações de execução de aplicações. Esse esquema é inspirado em sistemas biológicos compostos por milhões de organismos encontrados na natureza, que, de maneira autônoma, produzem complexos padrões de formação, organizando-se da melhor

maneira para executarem seu trabalho.

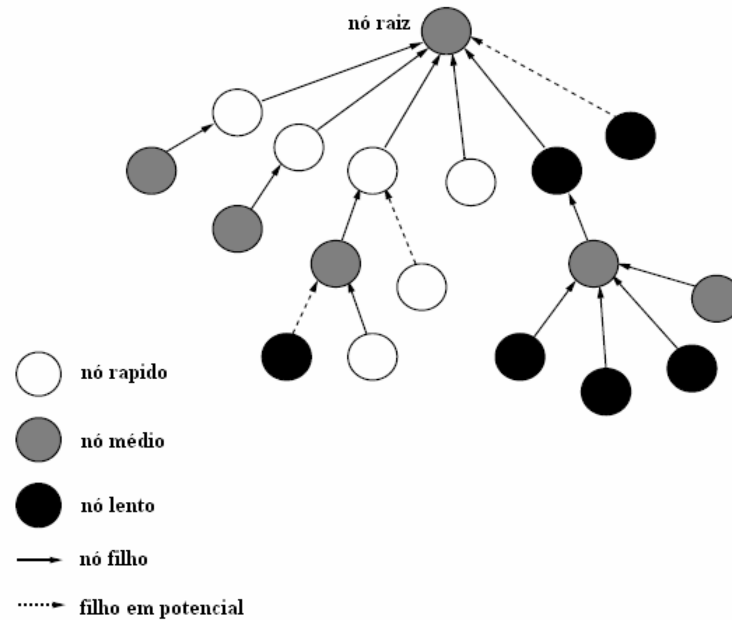


Figura 21. Organização Resultante após alguns Ciclos de Execução do Algoritmo de Escalonamento do *Organic Grid* (CHAKRAVARTI *et al.*, 2005c).

O *Organic Grid* propõe um esquema de escalonamento descentralizado em que as decisões de escalonamento independem da existência de conhecimento global sobre os recursos disponíveis na grade, mas seu desempenho depende diretamente da formação inicial da árvore. Esta abordagem cria um alto grau de escalabilidade ao sistema, uma vez que não há presença de elementos centralizados (CHAKRAVARTI *et al.*, 2004)(CHAKRAVARTI *et al.*, 2005c).

4.2. Considerações Finais

O código móvel tem-se mostrado uma solução promissora para o desenvolvimento e, sobremaneira, para o aprimoramento de resultados do desempenho de sistemas distribuídos. O

agente móvel, como se apresenta na literatura, possui muitas vantagens para ambientes de larga escala.

Os sistemas distribuídos devem prover uma melhoria de desempenho, principalmente no tocante ao tempo de execução, utilização de recursos, comunicação na rede de intercomunicação, heterogeneidade de *hardware* e de *software*. Analisando as vantagens do uso de agentes móveis em ambientes distribuídos, vislumbrou-se que sua utilização na coleta de índice de desempenho nesses ambientes acarretasse a redução de tráfego na rede quando comparado ao uso de troca de mensagens e, conseqüentemente, uma melhoria no desempenho geral do sistema em questão.

O próximo capítulo apresenta a utilização de índices de carga e desempenho em ambientes paralelos distribuídos fazendo uso de agente móvel.

CAPÍTULO 5. UTILIZAÇÃO DE ÍNDICE DESEMPENHO EM AMBIENTES PARALELOS DISTRIBUÍDOS

Os sistemas distribuídos devem prover uma melhoria de desempenho sobremaneira no tocante ao tempo de execução, utilização de recursos, comunicação na rede, e suporte à heterogeneidade de *hardware* e de *software*. Ademais, oferecer confiabilidade, escalabilidade, flexibilidade e custo acessível para aquisição e manutenção quando comparados a computadores intrinsecamente paralelos.

Embora possua muitas qualidades e benefícios, a computação massiçamente paralela apresenta um alto custo de aquisição e manutenção quando comparada aos sistemas computacionais distribuídos, o que dificultou muito sua disseminação. Por meio de pesquisas a união da computação paralela com os sistemas distribuídos resultou em um bom desempenho a um custo acessível.

Com a convergência de ambos não só as vantagens foram adicionadas a essa nova linha de pesquisa, mas também, suas desvantagens e problemáticas. Um dos problemas inerentes à junção dessas áreas é o escalonamento de processos, que foi, e ainda o é, um fator desafiador.

O escalonador de processos é de vital importância para os sistemas paralelos distribuídos (SHIVARATRI *et al.,* 1992), e sua função é distribuir e determinar a ordem em que os processos serão executados. Dentre dos diversos objetivos do escalonamento de processos, destaca-se o balanceamento de cargas, que consiste em definir o local onde o processo deve ser executado, buscando manter a igualdade de carga em cada um dos recursos,

de acordo com sua capacidade.

Quando atribuída a um *host*, a tarefa gera uma carga de trabalho e a métrica que pode ser utilizada para quantificar essa carga e distinguir o estado do recurso analisado é o índice de carga (FERRARI e ZHOU, 1987), e, por extensão, o índice de desempenho (BRANCO, 2004). A utilização adequada dessas métricas no balanceamento de cargas pode fazer que o escalonamento de processos propicie uma melhoria de desempenho ainda maior no sistema como um todo.

Em sistemas de grande escala, como sistema paralelo distribuído, a necessidade do escalonamento de processos é ainda maior. Deste modo, para se obter um escalonamento eficiente, faz-se uso de índices de carga e índices de desempenho. Entretanto, a freqüência com que é feita a coleta e a atualização dos índices de carga de cada recurso dos *hosts* (processador, memória, disco e rede) interfere de forma considerável no desempenho do sistema. Se as informações são frequentemente atualizadas, há um aumento no tráfego da rede. Caso contrário, o balanceamento de cargas pode ser efetuado de forma incorreta por utilizar cargas antigas para tomar decisão.

O propósito deste trabalho é oferecer alternativas para coleta dos índices de carga e de desempenho para ambientes paralelos distribuídos, objetivando a um escalonamento balanceado e confiável para se obter um bom desempenho referente ao tempo de execução de aplicações.

Com base nos estudos efetuados (FERRARI e ZHOU, 1987)(BRANCO *et al.*, 2006) observa-se que manter as informações das cargas de trabalho (refletidas nos índices de carga e de desempenho) sempre atualizadas para o escalonador, geralmente, faz com que os *hosts* permaneçam com uma carga coerente a sua potência computacional, sem que o tráfego gerado na rede influencie significativamente no desempenho do sistema como um todo.

5.1. JPVM – *Java Parallel Virtual Machine*

Como abordado na seção 2.3, para se obter um ambiente paralelo distribuído faz-se necessária a utilização de APIs com rotinas para passagem de mensagem. Para prover passagem de mensagem, as APIs mais utilizadas são o MPI (*Message Passing Interface*) e o PVM (*Parallel Virtual Machine*), podendo, inclusive, ser utilizadas na implementação de aplicações desenvolvidas em linguagens como Fortran, C e C++. Propostas foram apresentadas para a utilização de bibliotecas a partir do surgimento da linguagem de programação Java. Dentre elas, citam-se o mpiJava (BAKER *et al.*, 1998) e o JPVM (*Java Parallel Virtual Machine*) (FERRARI, 1998).

O JPVM é uma API implementada totalmente em Java que permite a passagem de mensagens explícitas baseadas em memória distribuída MIMD. Essa API combina as vantagens da linguagem Java – como portabilidade e interoperabilidade –, com as técnicas de troca de mensagem entre processos paralelos em ambientes distribuídos (FERRARI, 1998).

Com o propósito de os programadores habituados ao PVM migrarem com maior facilidade para o JPVM, este apresenta uma interface de programação muito próxima, mas independente, do PVM. Contudo, adicionaram-se ao JPVM novas características, como segurança de *threads,* múltiplos pontos de comunicação por tarefa e roteamento padrão para mensagens diretas (FERRARI, 1998).

A principal classe da biblioteca JVPM é a *jpvmEnvironment,* por meio da qual uma aplicação tem acesso às funções básicas, como a criação e eliminação de processos; a sincronização de tarefas; o envio e recebimento de mensagens.

A comunicação é feita de forma direta, tarefa-para-tarefa, implementada sobre TCP *sockets*. A cada instância do *jpvmEnvironment,* cria-se um *server socket* que atribui o nome do *host* e um número de porta de conexão internamente para realizar passagem de mensagens

e para gerenciar os *threads*. Para identificar as tarefas nas passagens de mensagens, utiliza-se um único número inteiro denominado *jpvmTaskId*. Quando uma tarefa X deseja se comunicar com uma tarefa Y ela simplesmente se conecta com Y, usando o nome do *host* e a porta contidos no identificador da tarefa Y. Caso a conexão seja aceita, cria-se um *thread* dedicada à sua gerência (FERRARI, 1998).

O pacote JPVM pode ser utilizado em praticamente todas as plataformas que têm suporte à máquina virtual Java, como *Unix* e seus derivados, *Windows* e *Macintosh*. Desde que a rede de comunicação esteja devidamente instalada, não se apresenta dificuldade em integrar as máquinas de plataformas diferentes na máquina virtual.

Após as configurações necessárias em cada *host* (APÊNDICE A), a criação da máquina virtual paralela com o JPVM é dividida em duas etapas. A primeira etapa consiste em iniciar o *jpvmDaemon* em cada *host* que fará parte do ambiente. Ilustra-se na Figura 22 a execução do *jpvmDaemon.*

```
# java jpvm.jpvmDaemon
jpvm daemon: las08, port #59203
```

Figura 22. Execução do jpvmDaemon

Após a execução dos *daemons* nos *hosts*, deve-se cadastrá-los à máquina virtual paralela. Portanto, a segunda etapa consiste em executar um console interativo do JPVM para cadastrar os *hosts* (utilizando o comando *add*). No console, também é possível listar os *hosts* já cadastrados (comando *conf*) e ter conhecimento de quais e quantas tarefas estão em execução em cada *host* (comando *ps*). Na Figura 23 é possível observar uma execução do *jpvmConsole*.

```
# java jpvm.jpvmConsole
jpvm> add
        Host name   : las07
        Port number : 42892
jpvm> add
        Host name   : gmm
        Port number : 33065
jpvm> conf
3 hosts:
        gmm
        las07
        las08
jpvm> ps
gmm,   0 tasks:
las07, 0 tasks:
las08, 1 tasks:
jpvm console
jpvm>
```

Figura 23. Funcionalidades do jpvmConsole.

Os processos *daemons* são responsáveis por favorecer a criação de tarefas e por coordená-las. Para distribuir e determinar a ordem em que os processos serão executados, o JPVM utiliza como padrão o escalonamento *Round Robin.*

O escalonamento de processos *Round Robin* tem como função atribuir, de maneira cíclica, os processos aos *hosts*, até que não haja mais processo algum a ser escalonado (TANENBAUM, 2001). No *Round Robin* não é necessária a coleta de informações do sistema. No entanto, em Sistemas Distribuídos, a utilização da política de escalonamento *Round Robin* pode gerar um desbalanceamento de carga entre os *hosts*, principalmente se a plataforma for heterogênea, uma vez que *hosts* com baixa potência computacional podem receber a mesma quantidade de carga que *hosts* com maior potência computacional.

5.2. JPVM-PRM – *Java Parallel Virtual Machine with Performance Resource Monitor*

Objetivando um escalonamento balanceado e confiável para se obter um bom desempenho em ambientes paralelos distribuídos, instrumentou-se a biblioteca de passagem de mensagem JPVM, a qual possui como padrão o escalonamento *Round Robin*, e adicionou-se a estratégia de escalonamento PRS (*Performance Resource Scheduling*). Por meio do índice de desempenho dos *hosts,* teve-se com produto deste trabalho a estratégia PRS, que determina que processo pode ser executado em cada um dos *hosts*, ou seja, efetua-se o balanceamento de cargas. Para se obter a carga dos recursos, efetuar o cálculo do índice de carga e o de desempenho, desenvolveu-se o PRM (*Performance Resource Monitor*).

O nome designado ao JPVM instrumentado foi JPVM-PRM (*Java Parallel Virtual Machine with Performance Resource Monitor*).

Analisadas as vantagens do uso de agentes móveis em ambientes distribuídos, deles se utilizou para efetuar a coleta do índice de desempenho em ambientes paralelos distribuídos, e comparou-se com o uso de passagem de mensagem exercendo a mesma função, tendo em vista um baixo tráfego na rede e, conseqüentemente, uma melhora no desempenho geral do sistema.

5.2.1. PRM – *Performance Resource Monitor*

O desempenho de um sistema é determinado pela eficiência com que os recursos são alocados ou compartilhados. Diversos são os recursos que podem ser considerados em um

sistema. Não obstante, é possível dividi-los em quatro grupos que têm grande influência no desempenho: CPU, Memória, I/O de disco e de Rede.

Com o intuito de avaliar a carga imposta a cada um desses recursos, desenvolveu-se um *thread* nomeado de PRM (*Performance Resource Monitor*).

O PRM tem como incumbência obter a carga crua[2] do processador, da memória, do disco e da rede; efetuar o cálculo dos índices de carga com o auxílio de *benchmarks* normalizados para tratar a heterogeneidade do ambiente, caso exista; calcular o índice de desempenho e disponibilizá-lo em um arquivo com o nome do *host* e porta de conexão utilizada pelo JPVM-PRM.

Para obter a carga crua dos recursos, utiliza-se o pacote *Dstat*, que é oferecido pela maioria das distribuições Linux. Esse pacote tem como objetivo tratar as informações contidas no */proc* para se obter de forma clara a carga dos recursos de uma máquina.

O projeto de *Dstat* foi desenvolvido com a linguagem de programação Python, o que proporciona velocidade e precisão. É uma substituição versátil do *iostat, vmstat e ifstat* visando superar algumas de suas limitações e possui algumas características extras, como: organizar os resultados dos recursos em colunas; exibir apenas os recursos desejados; e, salvar os resultados em formato CVS (*Comma Separated Value*).

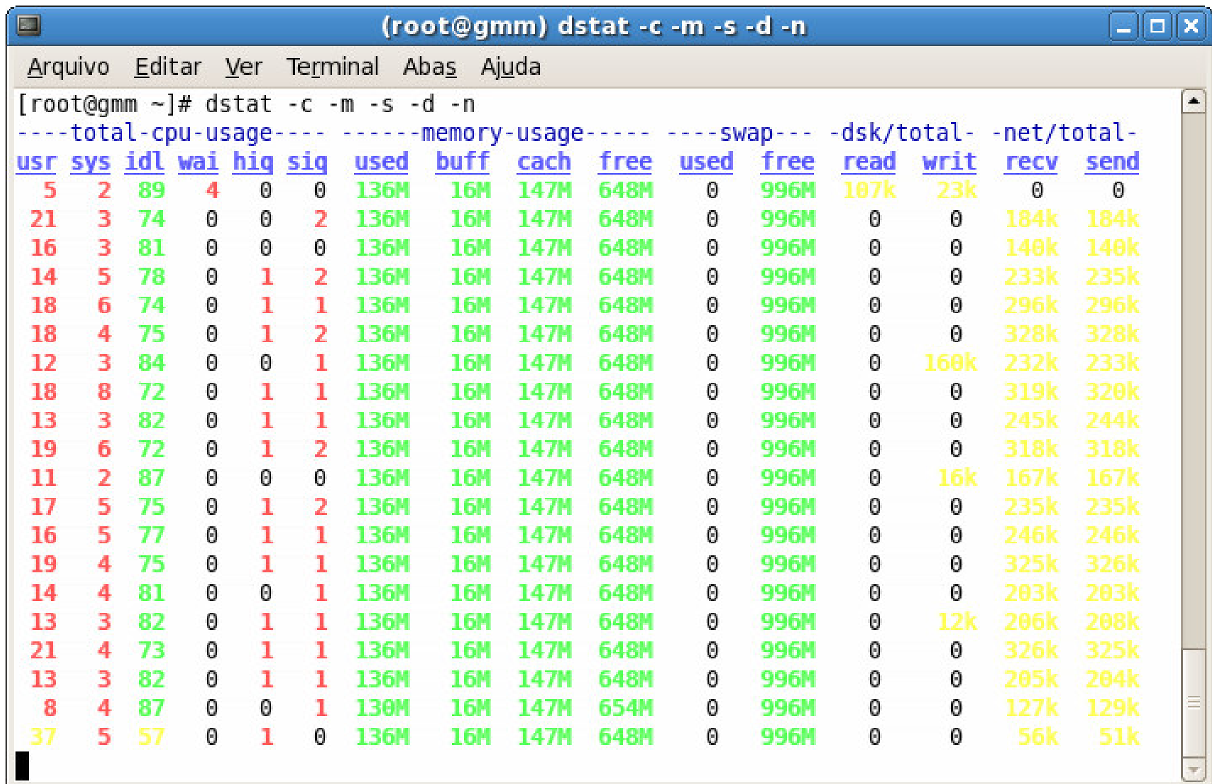
O PRM executa o *Dstat*, salva as informações lidas em um vetor, e, depois extrai do vetor adquirido as informações de interesse com o intuito obter a carga de cada recurso.

Na Figura 24 ilustra-se a execução do *Dstat*. As informações utilizadas para calcular a carga de CPU são: porcentagem de CPU livre (*idl*) e números de processos na fila de CPU (*wai*). Relativo à Memória são: memória propriamente livre (*used, buff, cach, free*) e *swap* (*used e free*). Referente à Disco são: número de leituras (*read*) e número de escritas (*writ*). E

---

[2] Carga crua são informações de carga de trabalho obtidas diretamente do sistema, em termos numéricos, sem tratamento algum.

respectivo à Rede são: número de pacotes que entram (*recv*) e número de pacotes que saem (*send*).




```
[root@gmm ~]# dstat -c -m -s -d -n
----total-cpu-usage---- ------memory-usage----- ----swap--- -dsk/total- -net/total-
usr sys idl wai hiq siq| used  buff  cach  free| used  free| read  writ| recv  send
  5   2  89   4   0   0| 136M   16M  147M  648M|   0   996M| 107k   23k|   0     0
 21   3  74   0   0   2| 136M   16M  147M  648M|   0   996M|   0     0 | 184k  184k
 16   3  81   0   0   0| 136M   16M  147M  648M|   0   996M|   0     0 | 140k  140k
 14   5  78   0   1   2| 136M   16M  147M  648M|   0   996M|   0     0 | 233k  235k
 18   6  74   0   1   1| 136M   16M  147M  648M|   0   996M|   0     0 | 296k  296k
 18   4  75   0   1   2| 136M   16M  147M  648M|   0   996M|   0     0 | 328k  328k
 12   3  84   0   0   1| 136M   16M  147M  648M|   0   996M|   0   160k| 232k  233k
 18   8  72   0   1   1| 136M   16M  147M  648M|   0   996M|   0     0 | 319k  320k
 13   3  82   0   1   1| 136M   16M  147M  648M|   0   996M|   0     0 | 245k  244k
 19   6  72   0   1   2| 136M   16M  147M  648M|   0   996M|   0     0 | 318k  318k
 11   2  87   0   0   0| 136M   16M  147M  648M|   0   996M|   0    16k| 167k  167k
 17   5  75   0   1   2| 136M   16M  147M  648M|   0   996M|   0     0 | 235k  235k
 16   5  77   0   1   1| 136M   16M  147M  648M|   0   996M|   0     0 | 246k  246k
 19   4  75   0   1   1| 136M   16M  147M  648M|   0   996M|   0     0 | 325k  326k
 14   4  81   0   0   1| 136M   16M  147M  648M|   0   996M|   0     0 | 203k  203k
 13   3  82   0   1   1| 136M   16M  147M  648M|   0   996M|   0    12k| 206k  208k
 21   4  73   0   1   1| 136M   16M  147M  648M|   0   996M|   0     0 | 326k  325k
 13   3  82   0   1   1| 136M   16M  147M  648M|   0   996M|   0     0 | 205k  204k
  8   4  87   0   0   1| 130M   16M  147M  654M|   0   996M|   0     0 | 127k  129k
 37   5  57   0   1   0| 136M   16M  147M  648M|   0   996M|   0     0 |  56k   51k
```

Figura 24. Execução do *Dstat*.

Os *benchmarks* são aplicações padronizadas que se empregam para medir o desempenho de diferentes recursos (SOUZA, 2004). Normalizam-se os valores neles obtidos, respectivamente a cada recurso analisado, com o propósito de mantê-los no intervalo de 0 a 1, dado que os valores mais próximos de 0 representam os melhores recursos, ao passo que os valores mais próximos de 1 representam os recursos com menor potência computacional. Uma vez obtidos os resultados referentes à execução dos *benchmarks* e efetuadas suas normalizações, podem-se comparar de igual para igual as máquinas do sistema paralelo distribuído heterogêneo.

Utiliza-se a Equação 3 para normalizar os *benchmarks*.

$$BN_{\mathrm{Re}curso} = 1{,}1 - \frac{P}{maiorP}$$

Equação 3

onde: *BN* representa o valor do *benchmark* normalizado; *P* refere-se ao potencial do recurso analisado; e *maiorP* é a melhor potência computacional encontrado.

Calcular o índice de carga com base no valor obtido por meio da normalização dos *benchmarks* permite classificar o índice de desempenho das máquinas de modo relativo, ou seja, conforme a potência computacional dos recursos de cada máquina do ambiente. Assim, o escalonador que tomar como base o índice de desempenho para efetuar o balanceamento de cargas, atribuirá cargas de trabalho aos recursos de modo coerente, sem que fiquem ociosos ou sobrecarregados.

Para se obter um índice de carga, utiliza-se a Equação 4.

$$IC_{\mathrm{Re}curso} = CR \times BN$$

Equação 4

onde: *IC* representa o valor do índice de carga; *CR* refere-se à carga do recurso; e o *BN* é o valor do *benchmark* normalizado do recurso analisado.

Para o cálculo do índice de desempenho, utiliza-se a Equação 5, já abordada na seção 3.4.

$$ID = \sqrt{I_{CPU}^2 + I_{Disco}^2 + I_{Mem\acute{o}ria}^2 + I_{\mathrm{Re}de}^2}$$

Equação 5

onde: *ID* representa o valor do índice de desempenho; *I* refere-se ao índice de carga do recurso.

Após o cálculo do índice de desempenho, o PRM salva o resultado no arquivo *id.txt* localizado no diretório */tmp* juntamente com o *tid* – nome do *host* e porta de conexão utilizada pelo JPVMD. Este procedimento é utilizado para disponibilizar o índice de desempenho para a estratégia de escalonamento PRS (*Performance Resource Scheduling*).

5.2.2. PRS – *Performance Resource Scheduling*

Quando a aplicação do usuário lhe solicita a criação de um processo, o JPVMD faz uma requisição ao PRS para saber em que *host* este processo pode ser executado. Em suma, o PRS consiste em um *thread* incorporado no JPVM (denominado JPVM-PRM). O JPVM-PRM interage com o PRS sempre que houver necessidade de se realizar algum escalonamento.

O PRS é instanciado pelo método *init()* da classe *jpvmDaemon* em todos os *hosts* que participam do ambiente paralelo distribuído. No entanto, o *thread* só será disparado por meio do método *start* no JPVMD mestre a partir do momento em que o segundo *host* for adicionado ao ambiente, o que cria uma relação hierárquica mestre/escravo.

Por conseguinte, ao iniciar o JPVMD no *host* mestre com o propósito de se utilizar a estratégia de escalonamento com base no índice de desempenho, é necessário informar o modo de obtenção do índice de desempenho e ter no mínimo um *host* escravo. Caso contrário, o escalonamento padrão *Round Robin* será utilizado.

Coleta do Índice de Desempenho

A obtenção do índice de desempenho, necessária para o PRS, pode ser efetuada por meio de um agente móvel (P*Agent* - *Performance Agent*), ou por passagem de mensagem (P*Message* - *Performance Message*). O PRM é iniciado pelos JPVMDs quando se opta por utilizar o PRS, independentemente do modo de obtenção do índice.

A mobilidade de código de um agente móvel deve ser provida por uma API. Duas merecem destaque: a API Aglets (LANGE e OSHIMA, 1998) e a µCode (muCode) (PICCO, 2007).

A mobilidade de código do P*Agent* provém da API muCode, que permite a mobilidade de código entre os MuServers remotos. O P*Agent* utiliza a classe MuAgent dessa API, fornecendo uma abstração natural para a implementação de agentes móveis.

Para se utilizar o JPVM-PRM com o P*Agent*, executa-se o comando ilustrado na Figura 25. O primeiro argumento informa o JPVMD do modo de coleta das informações dos *hosts* (no caso de se empregar o P*Agent*, utiliza-se ***A***), e o segundo argumento determina o período em que se efetuará a coleta..

```
# java jpvm.jpvmDaemon A 3000
jpvm daemon: las08, port #59203
```

Figura 25. Execução do jpvmDaemon com Argumento para se Utilizar o P*Agent*.

Ao adicionar um *host* escravo no *host* mestre do ambiente paralelo distribuído, quando se determina o modo de obtenção de índice P*Agent*, o JPVMD mestre dispara uma mensagem destinada ao escravo, solicitando o início do servidor muCode em uma determinada porta de conexão.

Dispara-se o P*Agent* para realizar a coleta das informações a partir do momento em

que o segundo *host* (primeiro escravo) for adicionado ao ambiente, e seu percurso de coleta define-se por um vetor que contém o nome do *host* e a porta de conexão do muCode.

Após o segundo *host* ser adicionado, ao acréscimo de um novo *host*, o JPVMD mestre gera o arquivo *PAgent_conf.txt* no diretório */tmp* com a palavra *new,* o nome do novo *host* e a porta de conexão do servidor muCode, para que P*Agent* adicione o novo *host* ao vetor de percurso.

Cabe ao P*Agent* a coleta periódica das informações disponibilizadas pelos PRMs no arquivo *id.txt* (contendo o *ID* e *tid*) dos *hosts* escravos, e o armazenamento posterior dessas informações no arquivo *idHost.txt* do diretório */tmp* no *host* mestre.

O PRS lê esse arquivo (*idHost.txt*) e cria uma lista ordenada – do menor índice de desempenho para o maior – com apenas os *tids* dos *hosts*. Dessa forma, os *hosts* com menor carga (melhor índice de desempenho) recebem processos antes daqueles com maior carga, o que implica em um ambiente balanceado.

Na Figura 26 ilustra-se a estrutura básica do JPVM-PRM P*Agent*.

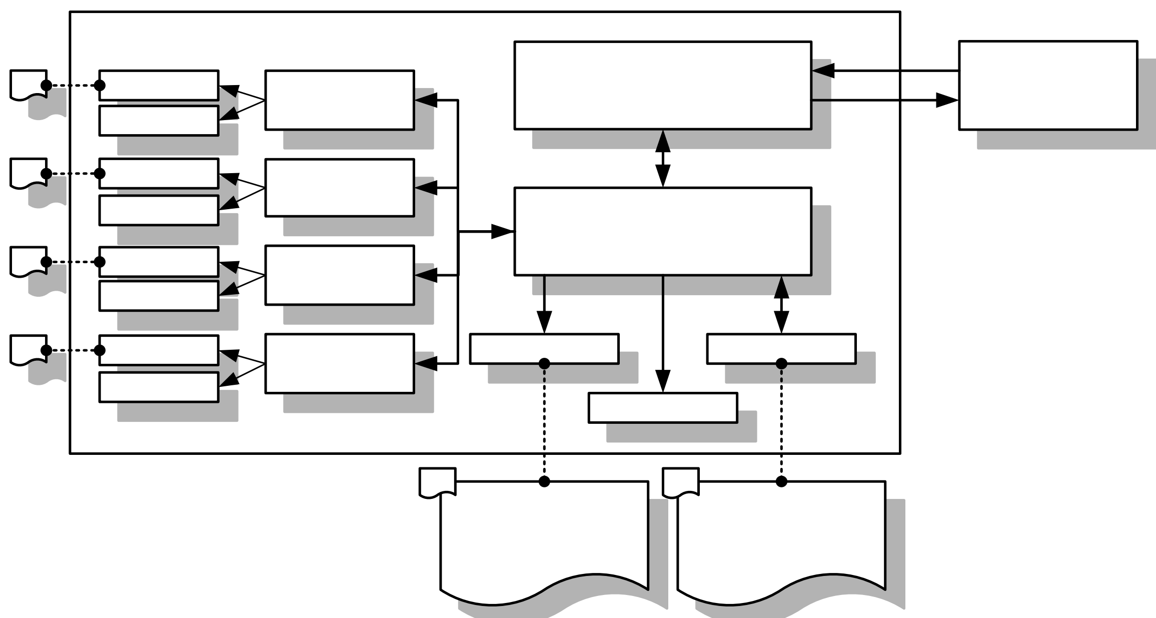


Figura 26. Estrutura Básica do JPVM-PRM P*Agent*.

Como uma alternativa ao P*Agent,* desenvolveu-se o P*Message*. Ao adotar o P*Message* como modo de coleta do índice de desempenho, periodicamente, ocorrem os seguintes passos:

1º. PRS solicita ao JPVMD mestre que envie uma mensagem para os *hosts* do ambiente requisitando o *ID* e o *tid* local desses *hosts*;

2º. JPVMD mestre envia a mensagem por *broadcast*;

3º. JPVMDs locais dos *hosts* escravos invocam seus PRMs;

4º. PRM retornam o *ID* do *host* ao seu respectivo JPVMD;

5º. JPVMDs escravos encapsulam as informações (ID e o seu *tid*) e as enviam ao JPVMD mestre;

6º. conforme o recebimento das informações, o JPVMD mestre as adiciona ordenadamente em uma lista – do menor ID para o maior;

7º. quando concluído o recebimento das informações de todos os *hosts*, o JPVMD mestre repassa a lista para o PRS, para que exerça sua função;

Conforme ilustrado na Figura 27, utiliza-se o argumento *M* para empregar o P*Message*. O intervalo entre as requisições de informação pode ser determinado pelo usuário.

```
# java jpvm.jpvmDaemon M 3000
jpvm daemon: las08, port #59203
```

Figura 27. Execução do jpvmDaemon com Argumento para se Utilizar o P*Message*.

Na Figura 28 demonstra-se a estrutura básica do JPVM-PRM P*Message*.

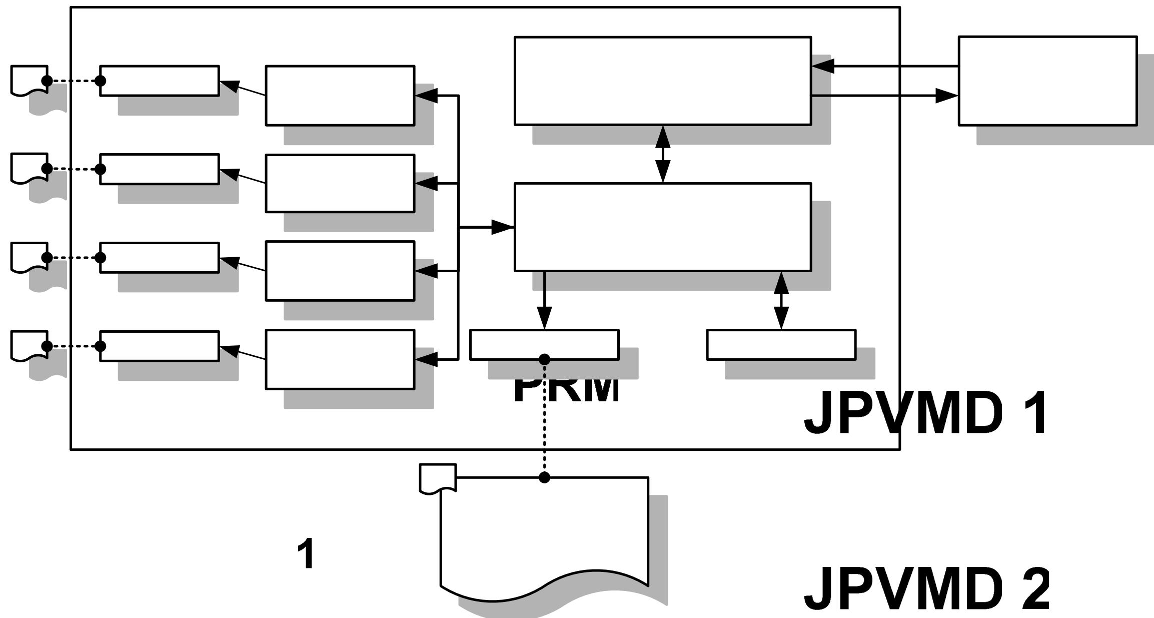


Figura 28. Estrutura Básica do JPVM-PRM P*Message*.

Existem algumas diferenças entre P*Agent* e P*Message*, dentre as quais é possível citar: na coleta P*Agent,* há necessidade de utilizar arquivos, o que não ocorre no P*Message*; na utilização do P*Message* é necessário enviar N mensagens (N é igual à quantidade de *hosts*) por *broadcast* e mais N mensagens de retorno, o que resulta em 2N mensagens, e, conseqüentemente, em um tráfego significante na rede, necessário, no entanto, para manter um sincronismo na coleta das informações. Contrastando com o P*Message*, o P*Agent* utiliza o equivalente a N mensagens, que representa o percurso do agente móvel de um *host* para outro.

Estratégia de Escalonamento PRS

No JPVM original o escalonamento de novos processos inicia-se quando a aplicação paralela do usuário invoca o método *pvm_spawn( )* da classe *jpvmEnviroment*, que, por sua vez, solicita a execução de seus novos processos nos *hosts* disponíveis no ambiente paralelo distribuído. O método *pvm_spawn( )* envia uma mensagem com o número de processos que

devem ser criados para o JPVMD mestre, que recebe a solicitação por meio do método *spawnTask()*. Baseando-se nos parâmetros enviados pelo *pvm_spawn()*, o método *spawnTask()* entra em um laço de repetição (*loop*) que, a cada ciclo, determina o *host* destino de um determinado processo com fundamentos em uma lista encadeada representada pela variável "*hosts*" (contém o *tid* de todos os *hosts* que estão aptos para receber processos).

Caso o processo tenha sido atribuído ao JPVMD mestre, este invoca o método *CreateTask()* localmente; caso seja atribuído a um JPVMD remoto, envia-se uma mensagem para o *host* remoto para ele invoque seu método *CreateTask()*.

Para realizar o escalonamento de processos fundamentados no índice de desempenho fez-se necessário realizar alterações no método *spawnTask()*. Em vez de consultar a lista "*hosts"*, solicita-se ao PRS, a cada laço de repetição, que informe um *tid* de um *host* para que seja atribuído um determinado processo. De forma ordenada, o PRS informa os *tids* por meio do índice de desempenho, partindo do menor índice para o maior índice, de modo que os *hosts* com menos carga (melhor índice de desempenho) recebam processos anteriormente àqueles com mais carga. Resulta daí um ambiente balanceado.

Na estratégia PRS, consideram-se carregados os *hosts* com o índice de desempenho superior à média de todos os índices. Utiliza-se a média como parâmetro para determinar se o *host* está apto para participar do escalonamento ou não. Assim, enquanto o índice de desempenho do *host* fica acima da média aceitável (*threshold*), ele não participa do escalonamento. Denomina-se este procedimento como filtro de *hosts*. Atualiza-se a média de índices com a mesma freqüência em que se efetua a coleta dos índices de desempenho.

Pode-se aumentar a tolerância do filtro, ou seja, acrescer uma porcentagem à média para permitir que *hosts* com índices de desempenho um pouco mais altos participem do escalonamento. No comando ilustrado na Figura 29, o primeiro argumento informa o JPVMD o modo de coleta das informações dos *hosts* (A ou M). O segundo argumento determina o

período em que se efetuará a coleta, e o terceiro diz respeito à porcentagem a ser acrescida na média do filtro de *hosts*. Neste exemplo são acrescidos 50% no valor da média dos índices de desempenho.

```
# java jpvm.jpvmDaemon A 3000 0.5
jpvm daemon: las08, port #59203
```

Figura 29. Execução do jpvmDaemon com Argumento de Filtro de *Hosts*.

5.3. Considerações Finais

O JPVM-PRM tem como função oferecer alternativas para a coleta dos índices de carga e de desempenho objetivando um escalonamento balanceado e confiável, de maneira a se obter um bom desempenho referente ao tempo de execução de aplicações apropriadas para ambientes paralelos distribuídos.

Compõem a estrutura do JPVM-PRM:

- PRM: tem a incumbência de obter as cargas cruas dos recursos; efetuar o cálculo dos índices de carga com o auxílio de *benchmarks* normalizados; calcular o índice de desempenho e disponibilizá-lo em um arquivo. O PRM permite gerar o índice de desempenho com base no tipo da aplicação do usuário;
- PRS: responsável por informar ao JPVMD mestre, o *host* apto a receber determinado processo, para que ocorra o balanceamento de cargas. Auxiliam o PRS:
  - P*Agent*: coleta de índice de carga por meio de agente móvel;

- P*Message*: coleta de índice de carga por meio de passagem de mensagem.

Para utilizar o JPVM-PRM, não há necessidade de re-compilar a aplicação paralela do usuário, pois todas as opções podem ser selecionadas na execução do JPVMD no *host* mestre.

O próximo capítulo apresenta a avaliação de desempenho e os resultados do escalonamento de processos executado com e sem a utilização do agente móvel, uma vez efetuadas as instrumentações necessárias na API JPVM e implementados o agente móvel e o monitor de recursos.

CAPÍTULO 6. AVALIAÇÃO DE DESEMPENHO E RESULTADOS

Com o intuito de se obter subsídios para avaliar o impacto da utilização de agentes móveis no escalonamento de processos em ambientes paralelos distribuídos, foram analisados diferentes estudos de caso.

Realizaram-se execuções de experimentos no Laboratório de Arquitetura de Sistemas (LAS) do Centro Universitário "Eurípides Soares da Rocha" de Marília - UNIVEM.

Os experimentos foram realizados em uma rede padrão *ethernet* 100 Mbps interligada por um *switch,* e composta por 9 computadores pessoais: dois Pentium 3 800MHz com 128MB de memória (Las05 e Las06); um Pentium 3 800MHz com 256MB de memória (Les05); um Pentinum 3 730MHz com 256MB (Las04); um Pentium 4 1.60GHz com 128MB de memória (Les02); um Pentium 4 1.70GHz com 256MB (Les04); um Pentium 4 2.66GHz com 1024MB de memória (GMM); dois AMD Athlon MP 2400+ 2GHz com 1024MB (Las07 e Las08).

O ambiente possibilitou a formação de uma arquitetura MIMD com memória distribuída por meio da API JPVM-PRM, tornando-se factível a execução de aplicações paralelas que aceitam esse tipo de plataforma.

Para avaliar o desempenho do ambiente, realizaram-se diferentes testes reais em que se deu a execução de algoritmos paralelos, foram comparados os tempos de execução e efetuada a análise estatística dos resultados obtidos.

6.1. Análise dos Resultados Obtidos

Para executar os estudos de caso, selecionaram-se dois algoritmos que exigem uma grande potência computacional: o algoritmo paralelo de multiplicação de matriz e o SOR *Red/Black* (SOR - *Successive Over-Relaxation*) (FERRARI, 1998). Para seu emprego, consideraram-se os seguintes argumentos:

- Multiplicação de Matriz 2000x2000 com 9 processos;
- Multiplicação de Matriz 2000x2000 com 16 processos;
- SOR *Red/Black* para uma Grade de 48 com 9 processos;
- SOR *Red/Black* para uma Grade de 48 com 16 processos.

Realizaram-se 30 execuções de cada algoritmo (com 9 e 16 processos) nos cenários elaborados em ambientes dedicados e também em ambientes não dedicados.

Para avaliar o comportamento do JPVM-PRM em ambientes não dedicados, desenvolveram-se cargas sintéticas com o objetivo de gerar cargas externas às oriundas da aplicação paralela sobre os recursos. A carga de CPU consiste em um *thread* que realiza operações matemáticas pré-definidas; a de Memória é um *thread* cuja função é carregar um vetor de cem mil posições e aplicar o método de ordenação *bubble-sort*; e a de Disco é um *thread* que carrega um arquivo de 20Mbs e, posteriormente, efetua a transferência deste para outro diretório. As cargas são iniciadas ao mesmo tempo em uma freqüência de execução aleatória. Ao fim de cada período de execução, os *threads* são colocados em pausa (*sleep*) por um período definido – duas vezes o período em que ficaram em execução.

Descrevem-se, na Tabela 3, os cenários elaborados. É possível observar que, nos cenários onde se fez o uso da estratégia de escalonamento PRS, utilizou-se o tempo de coleta de índice de desempenho de 3 segundos (3000ms) e de 6 segundos (6000ms), para avaliar a

viabilidade de ambos.

No primeiro cenário, aplicou-se o escalonamento padrão *Round Robin*; do segundo até o quinto cenário, fez-se o uso da estratégia de escalonamento PRS padrão, ou seja, sem indicar os pesos dos índices de carga (emprega-se o VIP) (BRANCO, 2004). Para o tratamento específico de cada aplicação, além dos cenários básicos, criaram-se mais oito cenários com o uso de índices de carga com pesos distintos para cálculo do índice de desempenho (emprega-se PVIP).

Tabela 3. Cenários Elaborados para os Estudos de Caso.

Nº	Cenário	Descrição
1	***Round Robin***	Escalonamento *Round Robin.*
2	**P*Message* 3000**	Escalonamento PRS P*Message* atualizado a cada 3000ms.
3	**PAgent 3000**	Escalonamento PRS P*Agent* atualizado a cada 3000ms.
4	**P*Message* 6000**	Escalonamento PRS P*Message* atualizado a cada 6000ms.
5	**PAgent 6000**	Escalonamento PRS P*Agent* atualizado a cada 6000ms.
6	**P*Message* MULT 3000**	Escalonamento PRS P*Message* com base no tipo da aplicação, sendo atualizado a cada 3000ms.
7	**PAgent MULT 3000**	Escalonamento PRS P*Agent* com base no tipo da aplicação, sendo atualizado a cada 3000ms.
8	**P*Message* MULT 6000**	Escalonamento PRS P*Message* com base no tipo da aplicação, sendo atualizado a cada 6000ms.
9	**PAgent MULT 6000**	Escalonamento PRS P*Agent* com base no tipo da aplicação, sendo atualizado a cada 6000ms.
10	**P*Message* SOR 3000**	Escalonamento PRS P*Message* com base no tipo da aplicação, sendo atualizado a cada 3000ms.
11	**PAgent SOR 3000**	Escalonamento PRS P*Agent* com base no tipo da aplicação, sendo atualizado a cada 3000ms.
12	**P*Message* SOR 6000**	Escalonamento PRS P*Message* com base no tipo da aplicação, sendo atualizado a cada 6000ms.
13	**PAgent SOR 6000**	Escalonamento PRS P*Agent* com base no tipo da aplicação, sendo atualizado a cada 6000ms.

A ordem em que são adicionados os *hosts* no ambiente paralelo distribuído pode influenciar no desempenho quando utilizado o escalonamento *Round Robin*. Portanto, independente do cenário, fez-se o uso das seguintes seqüências de *hosts*:

- las08 (mestre com grande potência computacional); las04, las06, las07, les05, las05, gmm, les04, les02; e

- las05 (mestre com pouco potência computacional); las04, las06, las07, les08, las05, gmm, les04, les02.

6.1.1. Multiplicação de Matriz

Os estudos de caso efetuados com o algoritmo de multiplicação de matriz com 9 e 16 processos demonstraram que o P*Agent* proporciona um desempenho superior (porém próximo) ao P*Message*, e que ambos possuem um desempenho consideravelmente maior que a política de escalonamento *Round Robin*.

O tempo de atualização do índice de desempenho utilizado no PRS (P*Message* e P*Agent*) de 6 segundos (6000ms) apresentou um ganho superior ao de 3 segundos (6000ms) em todos os casos avaliados, pois há um menor tráfego na rede e um consumo menor de recurso uma vez que as atualizações são executadas em uma freqüência maior.

Quando utilizado o índice de desempenho específico para o algoritmo – PVIP – obtêm-se um ganho superior ao que não utiliza pesos para seu cálculo – VIP. Foram estes os pesos utilizados para cálculo do índice de desempenho em específico para o algoritmo de multiplicação de matriz: 60% *CPU-Bound*, 30% *Memory-Bound*, 0% *Disk-Bound* e 10% *Network-Bound*. Dimensionaram-se esses pesos pelo prévio conhecimento da aplicação.

Os resultados obtidos com o algoritmo de multiplicação de matriz com 9 processos quando executado em um ambiente dedicado são apresentados com detalhes na Tabela 4 (resultados obtidos com o mestre com maior potência computacional) e na Tabela 5 (resultados obtidos com o mestre com menor potência computacional). Na Figura 30 e na Figura 31 observaram-se as diferenças de desempenho entre os cenários avaliados.

Tabela 4. Multiplicação de Matriz - 9 Processos - Mestre Las08.

Cenário	Multiplicação de Matriz - 9 Processos - Mestre Las08	
	Média (ms)	Ganho %
Round Robin	53.308,53	
PMessage3000	44.975,20	15,63%
PAgent3000	42.623,73	20,04%
PMessageMULT3000	41.406,93	22,33%
PAgentMULT3000	39.678,47	25,57%
PMessage6000	42.221,60	20,80%
PAgent6000	41.916,87	21,37%
PMessageMULT6000	39.698,47	25,53%
PAgentMULT6000	38.418,73	27,93%

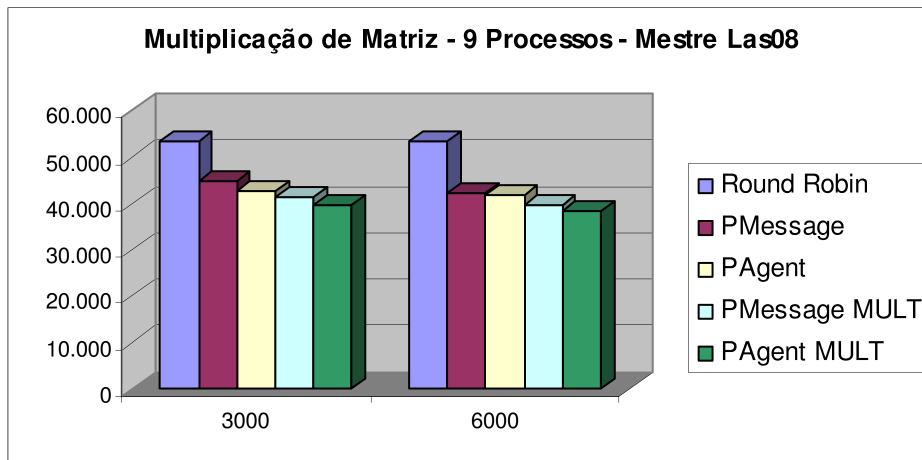


Figura 30. Multiplicação de Matriz - 9 Processos - Mestre Las08.

Pode-se observar na Tabela 5 e na Figura 31 que, quando executados os estudos de casos do algoritmo de multiplicação de matriz com 9 processos com um mestre de pouca potência computacional, o ganho de desempenho obtido na utilização do JPVM-PRM foi em média de 42,97% com P*Message*, e de 43,48% com P*Agent*, se comparados com o *Round Robin*.

Tabela 5. Multiplicação de Matriz - 9 Processos - Mestre Les05.

Cenário	Multiplicação de Matriz - 9 Processos - Mestre Les05	
	Média (ms)	Ganho %
Round Robin	101.964,40	
PMessage3000	61.884,60	39,31%
PAgent3000	60.553,20	40,61%
PMessageMULT3000	59.564,80	41,58%
PAgentMULT3000	59.090,20	42,05%
PMessage6000	56.397,67	44,69%
PAgent6000	56.325,93	44,76%
PMessageMULT6000	54.737,53	46,32%
PAgentMULT6000	54.539,80	46,51%

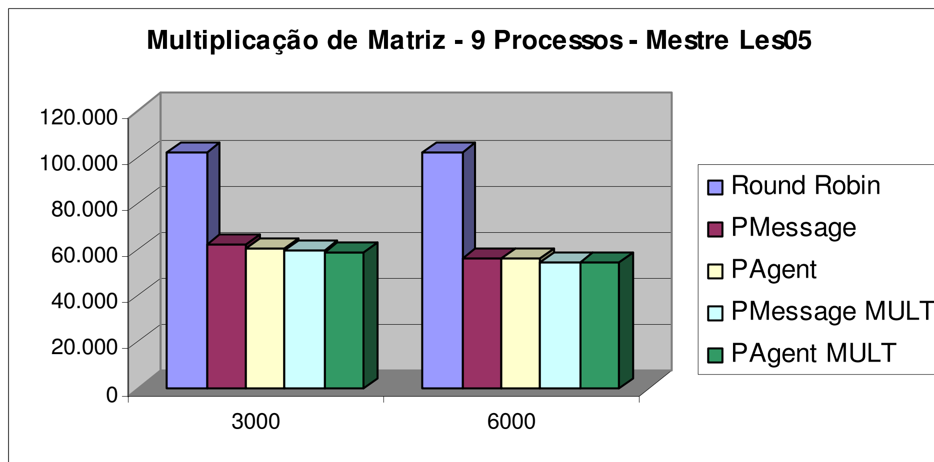


Figura 31. Multiplicação de Matriz - 9 Processos - Mestre Les05.

Apresentam-se, na Tabela 6, os resultados obtidos em ambiente carregado, isto é, os processos da aplicação concorrendo aos recursos juntamente com as cargas sintéticas desenvolvidas.

Tabela 6. Multiplicação de Matriz - 9 Processos - Mestre Las08 - Ambiente Carregado.

Cenário Ambiente Carregado	Multiplicação de Matriz - 9 Processos - Mestre Las08	
	Média (ms)	Ganho %
Round Robin	85.284,07	
PMessage3000	66.618,87	21,89%
PAgent3000	65.401,87	23,31%
PMessage6000	62.686,27	26,50%
PAgent6000	61.482,13	27,91%

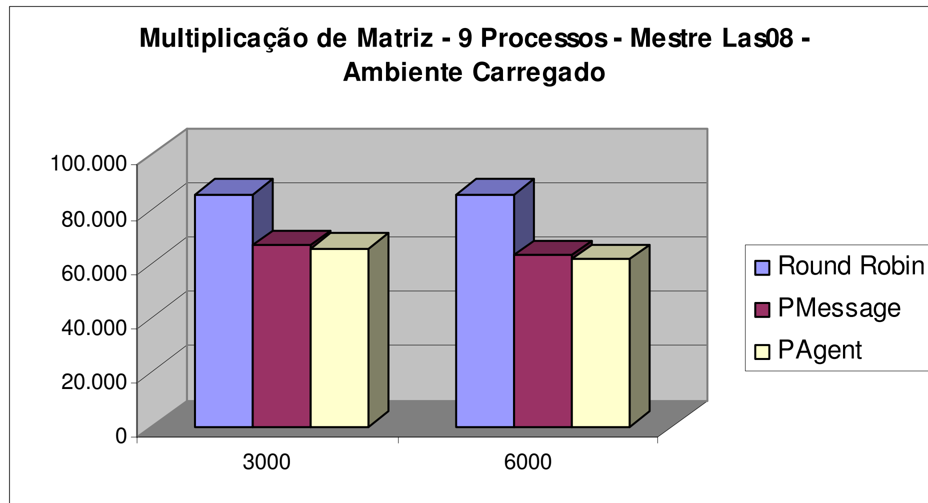


Figura 32. Multiplicação de Matriz - 9 Processos - Mestre Las08 - Ambiente Carregado.

Os resultados obtidos com o algoritmo de multiplicação de matriz com 16 processos quando executado em um ambiente dedicado são apresentados com detalhes na Tabela 7 (resultados obtidos com o mestre com maior potência computacional) e na Tabela 8 (resultados obtidos com o mestre com menor potência computacional). Na Figura 33 e na Figura 34 observaram-se as diferenças de desempenho entre os cenários avaliados.

Tabela 7. Multiplicação de Matriz - 16 Processos - Mestre Las08.

Cenário	Multiplicação de Matriz - 16 Processos - Mestre Las08	
	Média (ms)	Ganho %
Round Robin	53.391,07	
PMessage3000	43.436,73	18,64%
PAgent3000	43.242,80	19,01%
PMessageMULT3000	42.233,60	20,90%
PAgentMULT3000	41.273,80	22,70%
PMessage6000	41.370,67	22,51%
PAgent6000	40.974,40	23,26%
PMessageMULT6000	40.262,40	24,59%
PAgentMULT6000	39.605,60	25,82%

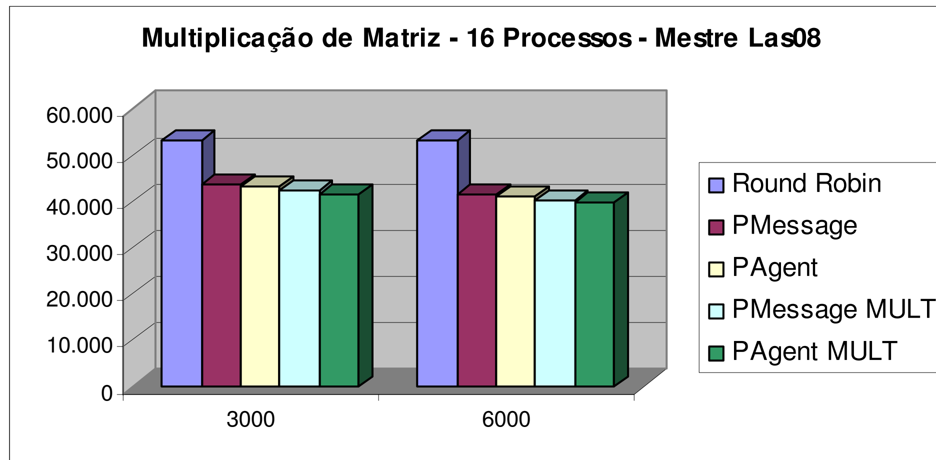


Figura 33. Multiplicação de Matriz - 16 Processos - Mestre Las08.

Pode-se observar na Tabela 8 e na Figura 34 que, quando executados os estudos de casos do algoritmo de multiplicação de matriz com 16 processos com um mestre de pouca potência computacional, o ganho de desempenho obtido na utilização do JPVM-PRM foi em média de 29,75% com P*Message*, e de 30,70% com P*Agent*, se comparados com o *Round Robin*.

Tabela 8. Multiplicação de Matriz - 16 Processos - Mestre Les05.

Cenário	Multiplicação de Matriz - 16 Processos - Mestre Les05	
	Média (ms)	Ganho %
Round Robin	80.253,93	
PMessage3000	61.729,07	23,08%
PAgent3000	60.603,07	24,49%
PMessageMULT3000	57.729,47	28,07%
PAgentMULT3000	57.026,33	28,94%
PMessage6000	56.247,60	29,91%
PAgent6000	55.783,87	30,49%
PMessageMULT6000	49.795,60	37,95%
PAgentMULT6000	49.061,60	38,87%

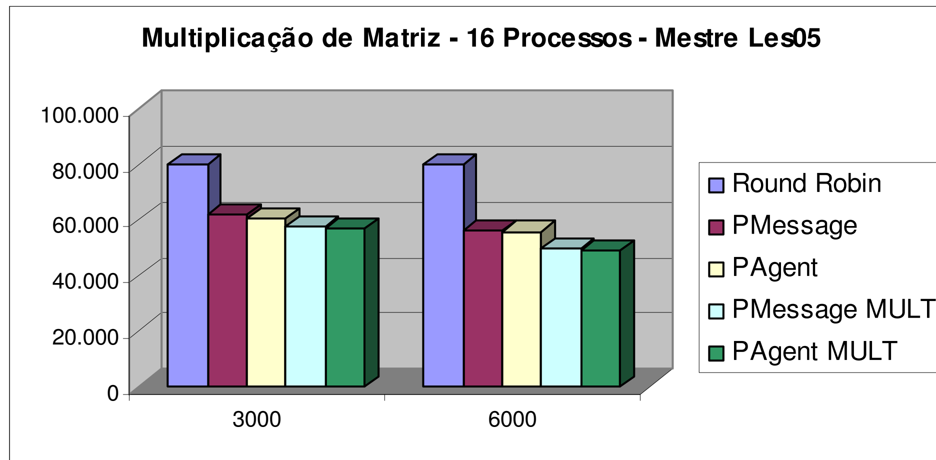


Figura 34. Multiplicação de Matriz - 16 Processos - Mestre Les05.

Apresentam-se, na Tabela 9, os resultados obtidos em ambiente carregado, isto é, os processos da aplicação concorrendo aos recursos juntamente com as cargas sintéticas desenvolvidas.

Tabela 9. Multiplicação de Matriz - 16 Processos - Mestre Las08 - Ambiente Carregado.

Cenário Ambiente Carregado	Multiplicação de Matriz - 16 Processos - Mestre Las08	
	Média (ms)	Ganho %
Round Robin	77.911,87	
PMessage3000	64.146,87	17,67%
PAgent3000	62.911,33	19,25%
PMessage6000	57.464,07	26,24%
PAgent6000	57.199,40	26,58%

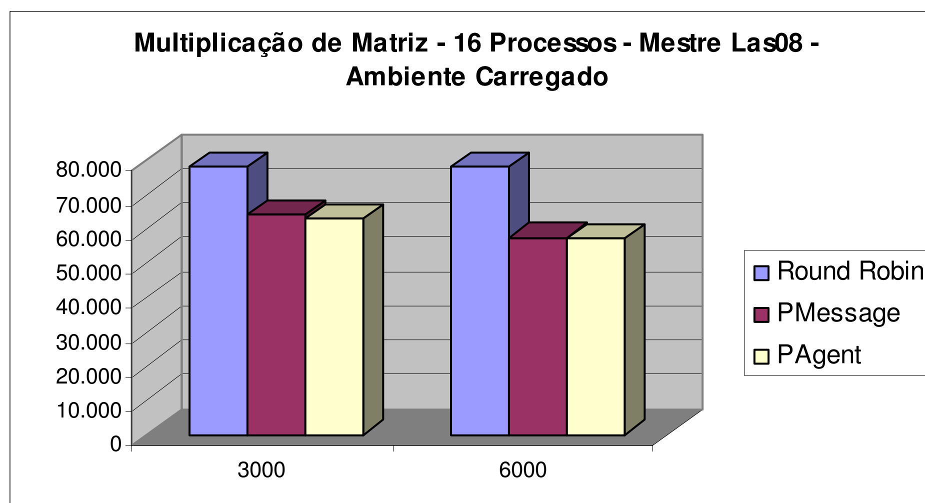


Figura 35. Multiplicação de Matriz - 16 Processos - Mestre Las08 - Ambiente Carregado.

6.1.2. SOR - *Successive Over Relaxation*

O método das sobre-relaxações sucessivas (SOR - *Successive Over-Relaxation*) representa uma variação do método de *Gauss-Seidel* pela introdução de um fator de relaxação (ω). Os estudos de caso efetuados com o algoritmo SOR com 9 e 16 processos demonstraram que o P*Agent* apresenta um desempenho superior (porém próximo) ao P*Message*, e que ambos possuem um desempenho consideravelmente maior que o escalonamento *Round Robin*.

O tempo de atualização do índice de desempenho utilizado no PRS (P*Message* e P*Agent*) de 6 segundos (6000ms) apresentou um ganho superior ao de 3 segundos (6000ms) em todos os casos de teste, pois gera um menor tráfego na rede e um consumo menor de recurso uma vez que efetua um número menor de atualizações da carga.

Quando aplicado o índice de desempenho específico para o algoritmo – PVIP – obtêm-se um ganho superior ao que não utiliza pesos para seu cálculo – VIP. Foram estes os pesos utilizados para cálculo do índice de desempenho em específico para o algoritmo SOR: 80% *CPU-Bound*, 0% *Memory-Bound*, 0% *Disk-Bound* e 20% *Network-Bound*. Dimensionaram-se esses pesos pelo prévio conhecimento da aplicação.

Os resultados obtidos com o algoritmo SOR com 9 processos quando executado em um ambiente dedicado são apresentados com detalhes na Tabela 10 (resultados obtidos com o mestre com maior potência computacional) e na Tabela 11 (resultados obtidos com o mestre com menor potência computacional). Na Figura 36 e na Figura 37 observaram-se as diferenças de desempenho entre os cenários avaliados.

Tabela 10. SOR - 9 Processos - Mestre Las08.

Cenário	SOR - 9 Processos - Mestre Las08	
	Média (ms)	Ganho %
Round Robin	38.768,20	
PMessage3000	32.810,73	15,37%
PAgent3000	31.956,60	17,57%
PMessageMULT3000	31.110,13	19,75%
PAgentMULT3000	30.957,60	20,15%
PMessage6000	30.498,73	21,33%
PAgent6000	29.893,20	22,89%
PMessageMULT6000	28.853,73	25,57%
PAgentMULT6000	28.589,60	26,26%

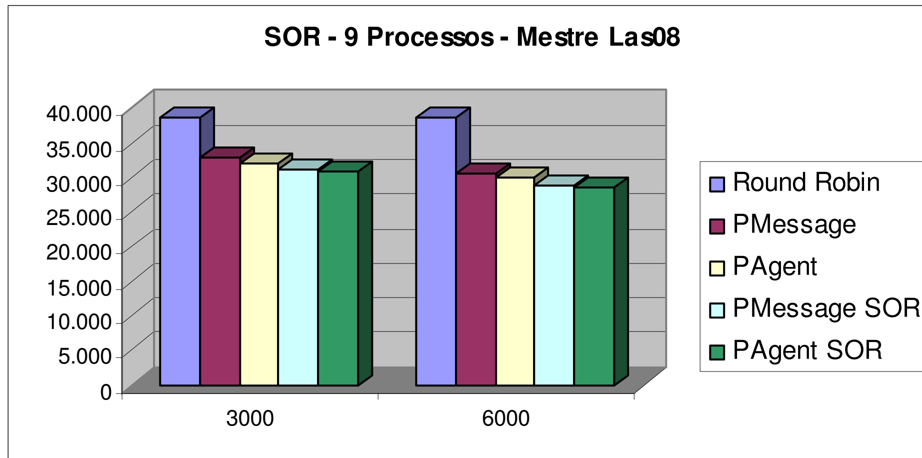


Figura 36. SOR - 9 Processos - Mestre Las08.

Pode-se observar na Tabela 11 e na Figura 37 que, quando executados os estudos de caso do algoritmo SOR com 9 processos com um mestre de pouca potência computacional, o ganho de desempenho obtido na utilização do JPVM-PRM foi em média de 17,14% com o uso do P*Message*, e de 18,29% com o P*Agent*, se comparados com o *Round Robin*.

Tabela 11. SOR - 9 Processos - Mestre Les05.

Cenário	SOR - 9 Processos - Mestre Les05	
	Média (ms)	Ganho %
Round Robin	42.937,73	
PMessage3000	36.979,80	13,88%
PAgent3000	36.688,53	14,55%
PMessageMULT3000	36.144,40	15,82%
PAgentMULT3000	35.879,73	16,44%
PMessage6000	35.255,13	17,89%
PAgent6000	34.996,07	18,50%
PMessageMULT6000	33.941,67	20,95%
PAgentMULT6000	32.771,33	23,68%

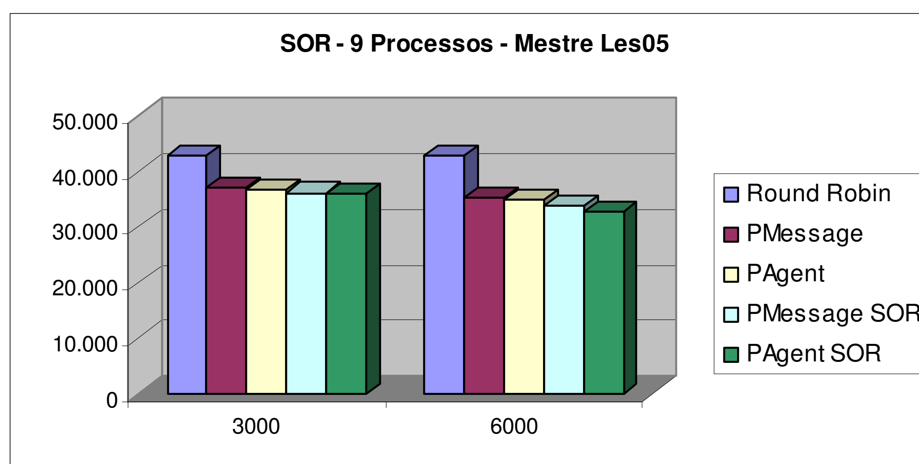


Figura 37. SOR - 9 Processos - Mestre Les05.

Apresentam-se, na Tabela 12, os resultados obtidos em ambiente carregado, isto é, os processos da aplicação concorrendo aos recursos juntamente com as cargas sintéticas desenvolvidas.

Tabela 12. SOR - 9 Processos - Mestre Las08 - Ambiente Carregado.

Cenário Ambiente Carregado	SOR - 9 Processos - Mestre Las08	
	Média (ms)	Ganho %
Round Robin	266.638,00	
PMessage3000	231.401,40	13,22%
PAgent3000	224.052,33	15,97%
PMessage6000	204.178,13	23,42%
PAgent6000	193.468,60	27,44%

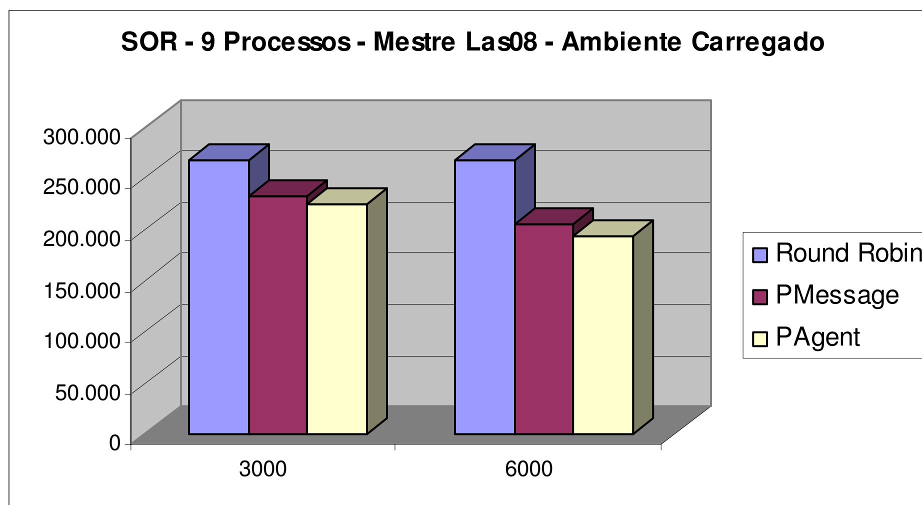


Figura 38. SOR - 9 Processos - Mestre Las08 - Ambiente Carregado.

Os resultados obtidos com o algoritmo SOR com 16 processos quando executado em um ambiente dedicado são apresentados com detalhes na Tabela 13 (resultados obtidos com o mestre com maior potência computacional) e na Tabela 14 (resultados obtidos com o mestre com menor potência computacional). Na Figura 39 e na Figura 40 observaram-se as diferenças de desempenho entre os cenários avaliados.

Tabela 13. SOR - 16 Processos - Mestre Las08.

Cenário	SOR - 16 Processos - Mestre Las08	
	Média (ms)	Ganho %
Round Robin	61.461,13	
PMessage3000	57.549,93	6,36%
PAgent3000	57.083,73	7,12%
PMessageMULT3000	56.446,40	8,16%
PAgentMULT3000	56.184,00	8,59%
PMessage6000	53.455,80	13,03%
PAgent6000	52.636,13	14,36%
PMessageMULT6000	51.828,20	15,67%
PAgentMULT6000	50.453,80	17,91%

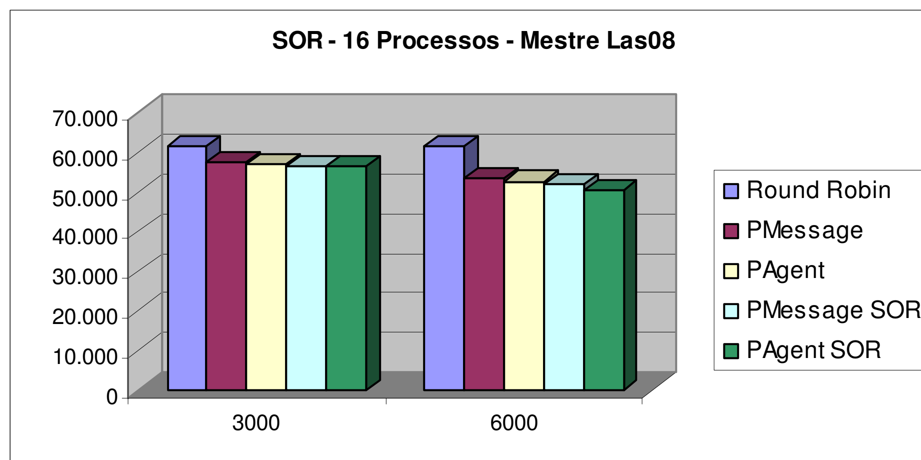


Figura 39. SOR - 16 Processos - Mestre Las08.

Pode-se observar na Tabela 14 e na Figura 40 que, quando executados os estudos de caso do algoritmo SOR com 16 processos com um mestre de pouca potência computacional o ganho de desempenho obtido na utilização do JPVM-PRM foi em média de 15,53% com P*Message*, e de 16,28% com P*Agent*, se comparados com o *Round Robin*.

Tabela 14. SOR - 16 Processos - Mestre Les05.

Cenário	SOR - 16 Processos - Mestre Les05	
	Média (ms)	Ganho %
Round Robin	74.566,80	
PMessage3000	66.458,00	10,87%
PAgent3000	65.974,20	11,52%
PMessageMULT3000	65.418,80	12,27%
PAgentMULT3000	65.036,13	12,78%
PMessage6000	61.823,40	17,09%
PAgent6000	60.487,67	18,88%
PMessageMULT6000	58.248,87	21,88%
PAgentMULT6000	58.219,27	21,92%

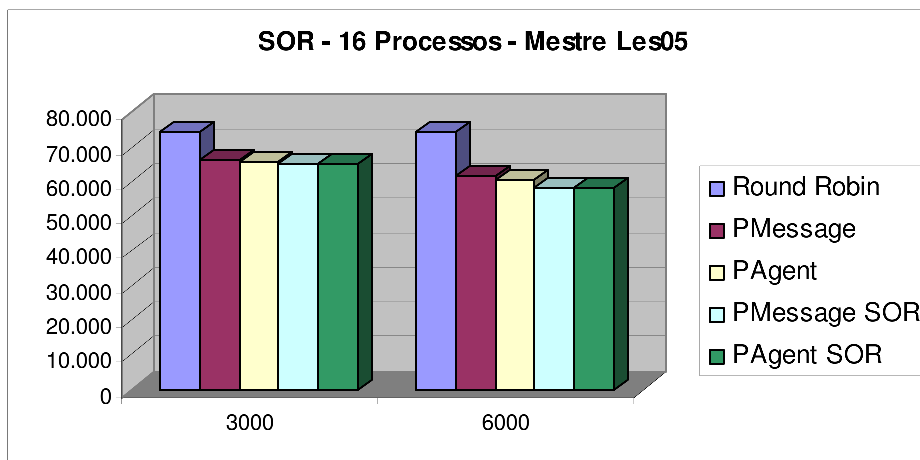


Figura 40. SOR - 16 Processos - Mestre Les05.

Apresentam-se, na Tabela 15, os resultados obtidos em ambiente carregado, isto é, os processos da aplicação concorrendo aos recursos juntamente com as cargas sintéticas desenvolvidas.

Tabela 15. SOR - 16 Processos - Mestre Las08 - Ambiente Carregado.

Cenário Ambiente Carregado	SOR - 16 Processos - Mestre Las08	
	Média (ms)	Ganho %
Round Robin	273.116,53	
PMessage3000	239.348,80	12,36%
PAgent3000	231.401,40	15,27%
PMessage6000	221.279,83	18,98%
PAgent6000	218.523,83	19,99%

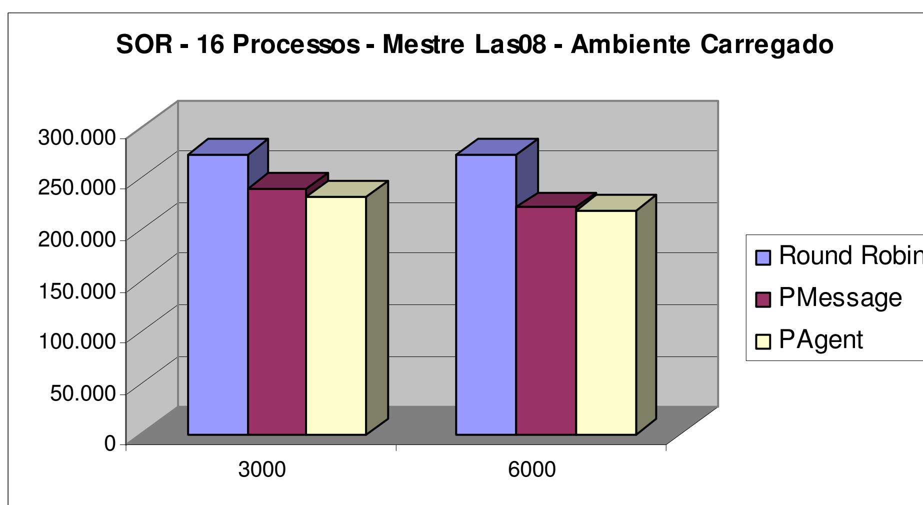


Figura 41. SOR - 16 Processos - Mestre Las08 - Ambiente Carregado.

6.2. Considerações Finais

Baseando-se nos resultados obtidos, verifica-se que, independentemente do modo de coleta do índice de carga, existe um ganho considerável em utilizar-se dele para efetuar o balanceamento de cargas no escalonamento.

Foi possível obter um ganho satisfatório no tempo de execução do algoritmo de multiplicação de matriz e do SOR em um ambiente paralelo distribuído heterogêneo utilizando o JPVM-PRM (P*Message* e P*Agent*), devido ao tratamento de heterogeneidade implantado em seu código e a agilidade do coletor do índice de desempenho. Verificou-se a viabilidade do uso de agente móvel na coleta do índice de desempenho em ambientes paralelos distribuídos, se comparado com passagem de mensagem.

Por meio das avaliações de desempenho, pode-se observar que o algoritmo SOR obteve uma diferença considerável entre o tempo de atualização de 3 segundos e à de 6 segundos, o que não ocorreu com tanto impacto no algoritmo de multiplicação de matriz. O motivo dessa diferença reside em que o algoritmo SOR consome muita CPU e rede. E, quando se faz o uso da atualização de 6 segundos, estes recursos são menos utilizados pelo JPVM-PRM (P*Message* e P*Agent*), obtendo, assim, um melhor desempenho. Os pesos aplicados nos índices de carga para calcular um índice de desempenho específico para o algoritmo também contribuíram para se obter um desempenho ainda maior.

Ao efetuar o balanceamento de cargas com base no índice de desempenho, o escalonador colaborou para que os *hosts* mantivessem seus recursos com uma carga coerente a sua potência computacional, o que permitiu um desempenho final superior ao obtido no escalonamento *Round Robin*.

CAPÍTULO 7. CONCLUSÃO

Objetivando o bom desempenho referente ao tempo de execução de aplicações paralelas em ambientes paralelos distribuídos, instrumentou-se a biblioteca de passagem de mensagem JPVM para efetuar o escalonamento de processos com base no índice de desempenho dos *hosts* constituintes do ambiente. Proveu-se, portanto, um escalonamento balanceado e confiável. Ao JPVM assim instrumentado, designou-se JPVM-PRM.

Distinto do JPVM – que utiliza o escalonamento *Round Robin* – o JPVM-PRM dispõe de uma estratégia de escalonamento que possui tratamento da heterogeneidade do ambiente, por meio do PRM. Esta estratégia, denominada PRS, determina por meio do índice de desempenho do *host* qual processo pode ser nele executado, permitindo que a carga a ele atribuída seja relativa à sua potência computacional.

Como abordado, o JPVM-PRM disponibiliza duas técnicas para coleta de índice de desempenho: a P*Agent*, que aplica agente móvel (foco do projeto), e a P*Message*, que utiliza passagem de mensagem. A técnica P*Message* foi implementada com o objetivo de se efetuarem comparações com a P*Agent* para verificar o impacto causado no ambiente pelo agente móvel e para demonstrar sua viabilidade.

Com base nos resultados obtidos, pode-se verificar que existe bom ganho em se fazer o escalonamento de processos tomando como base o índice de desempenho no processamento de algoritmos paralelos. Ou seja, há uma considerável redução no tempo de resposta ao se utilizar o índice de desempenho, quando comparada ao escalonamento padrão *Round Robin*. Os resultados obtidos com índices de desempenho adequados à aplicação (PVIP)

proporcionaram um desempenho ainda melhor.

Constatou-se que, quando empregado um tempo de atualização de índices de desempenho muito pequeno, o escalonamento é mais preciso. Não obstante, o desempenho resultante pode não ser satisfatório por carregar o ambiente. E um tempo muito longo faz que o escalonador utilize índices de desempenho desatualizados e efetue o escalonamento baseando-se neles, podendo causar uma maior sobrecarga no ambiente, em conseqüência, um desempenho insatisfatório.

O agente móvel apresentou um impacto inferior no ambiente àquele decorrente da passagem de mensagem, o que favoreceu um melhor desempenho. No entanto, ambos se mostraram adequados para manter as informações das cargas de trabalho – refletidas pelos índices de carga e de desempenho – convenientemente atualizadas, o que contribuiu para que o escalonamento fosse efetuado de forma balanceada, mantendo os recursos dos *hosts* com uma carga equilibrada a sua potência computacional.

O fato da utilização de agente móvel não degradar o desempenho da plataforma vem ao encontro do que se pretende demonstrar, a viabilidade do uso do agente móvel. Uma vez demonstrada essa viabilidade pode-se implementar e avaliar a utilização de agentes mais robustos que venham a permitir ganhos ainda maiores de desempenho.

Decorrentes dos experimentos, observações apuradas apontam que, comparado com o JPVM padrão, o JPVM-PRM contribuiu sobremaneira para a consecução de um bom desempenho.

7.1. Sugestões de Trabalhos Futuros

Como trabalhos futuros, sugerem-se avaliações de desempenho do ambiente proposto com outros tipos de aplicações paralelas, além do confronto entre os agentes móveis desenvolvidos com a API muCode e os agentes desenvolvidos com a API Aglets.

Ademais, sugere-se:

- avaliar o desempenho do JPVM-PRM em um ambiente homogêneo;
- otimizar a estratégia de escalonamento do PRS por meio do uso de técnicas de inteligência artificial;
- desenvolver um método para predizer o impacto a ser gerado em um determinado *host*, para que seja possível julgar se este suportará ou não a carga que lhe será atribuída;
- desenvolver novas técnicas para a consecução de carga dos recursos, com o intuito de reduzir, no ambiente, a carga gerada pelo PRM;
- implementar novas políticas de escalonamento;
- implementar um modo de adaptar o tempo de atualização das cargas conforme a necessidade, sem que sobrecarregue o ambiente, ou deixe o escalonador desatualizado;
- implementar e avaliar agentes mais robustos que possam inclusive colaborar para a escolha da política de escalonamento a ser utilizada provendo assim um ambiente inteligente.

7.2. Publicações Mais Relevantes

MARQUES, Giulianna Marega ; SABATINE, R. J.; BRANCO, K. R. L. J. C. . Performance Analysis of Scheduling Techniques in Distributed Parallel Environments. In: IEEE International Conference on Industrial Technology, 2008, Chengdu. IEEE International Conference on Industrial Technology, 2008. p. 1-6.

MARQUES, Giulianna Marega ; SABATINE, R. J.; BRANCO, K. R. L. J. C. . Load Balancing using Mobile Agents. In: Symposium on Agents and Multi-Agent Systems, 2008, Korea. The 2nd KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications, 2008. p. 1-10. (Aceito Para Publicação)

MARQUES, Giulianna Marega ; SABATINE, R. J.; BRANCO, K. R. L. J. C. . Analysis of Scheduling Techniques in Distributed Parallel Environments Using Mobile Agents. In: International Conference on Networking and Services, 2008, Gosier – Guadeloupe. The Fourth International Conference on Networking and Services, 2008. p. 1-6.

REFERÊNCIAS BIBLIOGRÁFICAS

(ACHCAR e RODRIGUES, 1995) ACHCAR, J.A., RODRIGUES, J. Introdução à Estatística para Ciências e Tecnologia. ICMSC-USP, São Carlos - Apostila de Consulta, 1995.

(ALMASI e GOTTLIEB, 1994) ALMASI, G. S.; GOTTLIEB, A. Highly Parallel Computing. 2ed. Redwood City: The Benjamin/Cummings Publishing Company, Inc, 1994.

(AVERSA *et al.*, 2004) AVERSA, R.; DI MARTINO, B.; MAZZOCCA, N. VENTICINQUE, S.. Terminal-Aware Grid Resource and Service Discovery and Access Based on Mobile Agents Technology. 2004.

(BEGUELIN, 1994) BEGUELIN, A. PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing, The MIT Press, 1994.

(BAKER *et al.*, 1998) BAKER, M. et al. mpiJava: A Java Interface to MPI. Submitted to First UKWorkshop on Java for High Performance Network Computing, Europar

(BIESZCZAD *et al.*, 1998) BIESZCZAD, A.; WHITE, T.; PAGUREK, B. Mobile Agents for Network Management. In IEEE Communications Surveys, September, 1998.

(BOOCH *et al.*, 2000) BOOCH, G; RUMBAUGH, J; JACOBSON, I. UML: Guia do Usuário,1ed. Rio de Janeiro: Campus, 2000.

(BRANCO, 2004) BRANCO, K. R. L. J. C. Índice de Carga e Desempenho em Ambientes Paralelos/ Distribuídos – Modelagem e Métricas. Tese de mestrado. ICMC-USP. 2004.

(BRANCO *et al.*, 2006) BRANCO, K. R. L. J. C. ; SANTANA, Marcos José ; SANTANA, Regina Helena Carlucci ; CAGNIN, Maria Istela ; ORDONEZ, Edward David Moreno ; BRUSCHI, Sarita Mazzini . Utilização de Simulação para Avaliação de Desempenho de Escalonamento de Processos. In: Ildeberto Aparecido Rodello, José Remo Ferreira Brega, Kalinka Regina Lucas Jaquie Castelo Branco. (Org.). Escola Regional de Informática São Paulo/Oeste 2006. 1 ed. Marília: UNIVEM, 2006, v. , p. 182-214.

(CAO *et al.*, 2002) CAO, J.; JARVIS, S. A.; KERBYSON, D. J.; NUDD, G. R.. ARMS: An

Agent-Based Resource Management System for Grid Computing. Scientific Programming 10, 2002.

(CARZANIGA *et al.*, 1997) CARZANIGA, A.; PICCO, G.; VIGNA, G. Designing Distributed Applications With Mobile Code Paradigms. Proceedings of the 19th International Conference on Software Engineering (ICSE´97), Boston, MA, p. 22-32, Maio 1997.

(CASAVANT e KUHL, 1988) CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Transactions on Software Engineering. 1988.

(CHAKRAVARTI *et al.*, 2003) CHAKRAVARTI, A. J.; WANG, X.; HALLSTROM, J. O.; BAUMGARTNER, G.. Implementation of Strong Mobility for Multi-Threaded Agents in Java. In 2003 International Conference on Parallel Processing (ICPP '03). IEEE Computer Society Press., pages 321-330, Koahsiung, Taiwan, 6-9 Outubro, 2003.

(CHAKRAVARTI *et al.*, 2004) CHAKRAVARTI, A. J.; BAUMGARTNER, G.; LAURIA, M.. Application-Specific Scheduling for the Organic Grid. In Proceedings of 5th IEEE/ACM International Workshop on Grid Computing, p. 146-155, Nov. 2004.

(CHAKRAVARTI *et al.*, 2005a) CHAKRAVARTI, A. J.; BAUMGARTNER, G.; LAURIA, M.. The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network. To Appear in IEEE Transactions on Systems, Man, and Cybernetics, Vol. 35 (No. 3), Maio 2005.

(CHAKRAVARTI *et al.*, 2005b) CHAKRAVARTI, A. J.; BAUMGARTNER, G.; LAURIA, M.. The Organic Grid: Self-Organizing Computational Biology on Desktop Grids. To Appear in A. Zomaya, Parallel Computing for Bioinformatics, 2005.

(CHAKRAVARTI *et al.*, 2005c) CHAKRAVARTI, A. J.; BAUMGARTNER, G.; LAURIA, M.. Self-Organizing Scheduling on the Organic Grid. To appear in International Journal on High-Performance Computing Applications, 2005.

(COULOURIS *et al.*, 2001) COULOURIS, G. *et al.* Distributed Systems: Concelemento de processamentots and Design (Third Edition), Addison-Wesley, 2001.

(COULOURIS *et al.*, 2007) COULOURIS, G.; Dollimore, J.; Kindberg, T. Sistemas Distribuídos Conceitos e Projetos, 4nd Edition, Bookman 2007.

(CULLER *et al.*, 1999) CULLER, David E.; GUPTA, Anoop; SINGH, Jaswinder Pal. Parallel computer architecture: a hardware/ software approach. San Francisco, Caliornia: Morgan Kaufmann Publishers, 1999.

(DOVAL e O'MAHONY, 2003) DOVAL, D.; O'MAHONY, D.. Overlay networks: A Scalable Alternative for P2P. IEEE Internet Computing, 7:79-82, Julho/Agosto 2003.

(DUNCAN, 1990) DUNCAN, R. A Survey of Parallel Computer Architectures. IEEE Computer, 1990.

(FEITELSON, 2007) FEITELSON, D. G. Workload Modeling for Computer Systems Performance Evaluation. Book Draft. Disponível em: <http://www.cs.huji.ac.il/%7Efeit/wlmod/wlmod.pdf>. Acesso em Janeiro de 2007.

(FEITELSON, 2003) FEITELSON D. G. Metric and Workload Effects on Computer Systems Evaluation. IEEE Computer Society Press, Los Alamitos, CA, USA, vol. 36, n. 9, p. 18-25, 2003.

(FEITELSON, 2002a) FEITELSON, D. G. The Forgotten Factor: Facts on Performance Evaluation and Its Dependence on Workloads. In: Euro-Par ’02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing. London, UK: Springer-Verlog, 2002.

(FEITELSON, 2002b) FEITELSON, D. G. Workload modeling for performance evaluation. In: CALZAROSSA, M. C.; TUCCI, S. (Ed.). Performance Evaluation of Complex Systems: Techniques and tools. Berlim: Springer-Verlag, 2002.

(FERRARI, 1998) FERRARI, A. J. JPVM: Network parallel computing in Java, In ACM 1998 Workshop on Java for High-Performance Network Computing, Palo Alto, February 1998. Concurrency: Pactice and Experience, 1998.

(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 Int'l Symposium on Computer Performance Modeling, Measurement, and Evaluation. 1987.

(FLYNN, 1972) FLYNN, M. J. Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers, 1972.

(FLYNN e RUDD, 1996) FLYNN, M. J.; RUDD, K. W. Parallel Architectures. ACM Computing Surveys. 1996.

(FOSTER e KESSELMAN, 1998) FOSTER, I.; KESSELMAN, C. The Globus Project: A Status Report. In Proc. IPPS/SPDP’98 Heterogeneous Computing Workshop, p. 4–18, 1998.

(FRANKLIN e GRAESSER, 1996) FRANKLIN, S.; GRAESSER, A. Is it an Agent, or Just a Program? A Taxonomy for Autonomous Agents. In Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages, pages 21-35. Springer-Verlag.1996

(FUGGETTA *et al.*, 1998) FUGGETTA, A.; PICCO, G.; VIGNA, G. Understanding Code Mobility. IEEE Transactions on Software Engineering, vol. 24, p. 343-353, Maio 1998.

(FUKUDA *et al.*, 2003) FUKUDA, M.; TANAKA, Y.; BIC, L. F.; KOBAYASHI, S. A Mobile-Agent-Based PC Grid. IEEE Computer, 2003.

(GHEZZI e VIGNA, 1997) GHEZZI, C.; VIGNA, G. Mobile code paradigms and technologies: a case study. In: Mobile agents: 1st international workshop MA '97. LNCS, v. 1219. Germany : Springer-Verlag, p. 39-49. Abril, 1997.

(GRAMA *et al.*, 2003) GRAMA, A.; KARYPIS, G.; KUMAR, V.; GUPTA, A. Introduction to Parallel Computing. 2. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

(HAYES, 1999) HAYES, C. C. Agents in a Nutshell - A Very Brief Introduction. Knowledge and Data Engineering, 11(1):127-132. 1999.

(JAIN, 1991) JAIN, R. The art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. John Wiley & Sons, Inc, 1991.

(JANSEN, 2000) JANSEN, Wayne. Countermeasures for Mobile Agent Security. Computer Communications, vol. 23, p. 1667-1676, Novembro 2000.

(JANSEN e KARYGIANNIS, 1999) JANSEN, W.; KARYGIANNIS, T. Mobile Agent Security. National Institute of Standards and Technology - Computer Security Division, NIST Special Publication 800-19,1999.

(JENNINGS e WOOLDRIDGE, 1998) JENNINGS, N. R.; WOOLDRIDGE, M. J. Applications of Intelligent Agents. Agent Technology: Foundations, Applications, and Markets, pages 3-28. Springer-Verlag: Heidelberg, Germany. 1998.

(KUNG *et al.*, 1991) KUNG, H. T. *et al.* Network-Based Multicomputers: an emerging parallel architecture. In: Proceedings of the 1991 ACM/IEEE conference on Supercomputing – Albuquerque, New Mexico, United States. New York, NY, USA: ACM Press, 1991.

(KREASECK *et al.*, 2003) KREASECK, B.; CARTER, L.; CASANOVA, H; FERRANTE, J. Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-Task Applications. In Proceedings of the International Parallel and Distributed Processing Symposium, p. 23-25, Abril, 2003.

(KUNZ, 1991) KUNZ, T. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. IEEE Transactions on Software Engineering. 1991.

(LANGE e OSHIMA, 1999) LANGE, D. B.; OSHIMA, M. Seven Good Reasons for Mobile Agents. Communications of the ACM, 42(3):88-89. 1999.

(LANGE e OSHIMA, 1998) LANGE, D. B.; OSHIMA, M. Mobile Agents with Java: The Aglet API. World Wide Web, 1(3). 1998.

(MCBRYAN, 1994) MCBRYAN, O. A. "An overview of *Message* passing environments", Parallel Computing, v. 20.1994.

(MOBACH *et al.*, 2004) D.G.A. MOBACH, B.J. OVEREINDER, O. MARIN, F.M.T. BRAZIER. Lease-based Decentralized Resource Management in Open Multi-Agent Systems. In Proceedings of the Second European Workshop on Multi-Agent Systems (EUMAS'04), p. 459-464, 2004.

(MONTGOMERY e RUNGER, 2003) MONTGOMERY, Douglas C. e RUNGER, George C.. Applied statistics and probability for engineers. 3rd ed. John Wiley & Sons, Inc 2003.

(MULLENDER, 1993) MULLENDER, S. J. Distributed Systems. *2nd Edition*. ACM-Press. Addison-Wesley. 1993.

(ORLANDI, 1995) ORLANDI, R. C. G. S. Ferramenta para Analise de Desempenho de Sistemas Computacionais Distribuídos. Dissertação (Mestrado). ICMC-USP, São Carlos, 1995.

(OVEREINDER *et al.*, 2002) OVEREINDER, B.J.; WIJNGAARDS, N. J. E.; VAN STEEN, M.; BRAZIER, F. M. T.. Multi-Agent Support for Internet-Scale Grid Management. In Proceedings of the AISB'02 Symposium on AI and Grid Computing, p. 18-22, Abril 2002.

(PICCO, 2007) PICCO, Gian. A Mobile Code Toolkit. Disponível em: <http://mucode.sourceforge.net>. Acesso em Fevereiro de 2007.

(PITANGA, 2003) PITANGA M. Computação em Cluster: o estado da arte da computação.

Brasport, 2003.

(PLASTINO, 2000) Plastino, A. Balanceamento de Carga de Aplicações Paralelas SPMD. Tese (Doutorado). Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, 2000.

(QUINN, 1994) QUINN, M. J. Parallel Computing: Theory and practice. 2. ed. New York: McGraw Hill, 1994.

(REWINI *et al.*, 1995) REWINI, H. E.; ALI, H. H.; LEWIS, T. Task Scheduling in Multiprocessing Systems. IEEE Computer. 1995.

(RUSSELL; NORVIG, 2003) RUSSELL, S.; NORVIG, P. Artificial Intelligence - A Modern Approach. Prentice-Hall (2nd Edition). 2003.

(SCHOINAS, 1997) SCHOINAS, I. Fine-Grain Distributed Shared Memory on a Cluster of Workstations. PhD thesis, Computer Sciences Department, University of Wisconsin, 1997.

(SELKER, 1994) SELKER, T. Coach: A Teaching Agent that Learns. Communications of the ACM, 37(7): 92-99. 1994.

(SHIRAZI *et al.*, 1995) SHIRAZI, B.A.; HURSON, A. R.; KAVIN, K. M. Scheduling and Load Balancing in Parallel and Distributed Systems. IEEE Computer Society Press. 1995.

(SHIVARATRI *et al.*, 1992) SHIVARATRI, N. G.; KRUEGER, P.; SINGHAL, M. Load Distribution for Locally Distributed Systems. IEEE Computer. 1992.

(SKILLICORN e TALIA, 1998) SKILLICORN D.B.; TALIA D. Models and Languages for Parallel Computation, ACM Computing Surveys, vol. 30, no. 2, pp. 123-169, June 1998.

(SOUZA, 2004) SOUZA, M. A. Uma Abordagem para a Avaliação do Escalonamento de Processos em Sistemas Distribuídos Baseada em Monitoração. Tese (Doutorado). ICMC-USP, São Carlos, 2004.

(STALLINGS, 2003) STALLINGS, W. Arquitetura e Organização de Computadores: Projeto para o desempenho. 5. ed. São Paulo: Prentice Hall, 2003. Tradução: Carlos Camarão de Figueiredo e Lucília Camarão de Figueiredo.

(TANENBAUM e VAN STEEN, 2002) TANENBAUM, Andrew S.; VAN STEEN, Maarten.

Distributed systems: principles and paradigms. Upper Saddle River, New Jersey: Prentice Hall, 2002.

(TANENBAUM, 2001) TANENBAUM, A. S. Modern Operating Systems. 2. ed. Upper Saddle River, New Jersey: Prentice-Hall, Inc., 2001.

(TANENBAUM, 1999) TANENBAUM, Andrew S. Sistemas operacionais modernos. Rio de Janeiro: Livros Técnicos e Científicos, 1999.

(TANENBAUM, 1995) TANENBAUM, Andrew S. Distributed operating systems. New Jersey: Prentice Hall, 1995.

(TANENBAUM, 1992) TANENBAUM, Andrew S. Modern Operating Systems. New Jersey, Prentice Hall International, Inc. 1992.

(WOOLDRIDGE, 1998) WOOLDRIDGE, M. Agent-based Computing. Interoperable Communication Networks, 1(1):71-97.

(ZALUSKA, 1991) ZALUSKA, E. J. Research Lines in Distributed Computing Systems and Concurrent Computation. Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software, ICMC/USP, São Carlos/SP. 1991.

(ZHOU *et al.*, 1993) Zhou, S.; Zheng, X.; Wang, J.; Delisle, P. Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. Software: Practice and Experience, v.23, 1993.

APÊNDICE A. PROCEDIMENTOS UTILIZADOS PARA INSTALAÇÃO E CONFIGURAÇÃO DO AMBIENTE PARALELO DISTRÍBUIDO

Para instalar e configurar o ambiente paralelo distribuído, tal como feito neste trabalho, faz-se necessário efetuar os procedimentos em todos os *hosts* que nele participarão. Indica-se que seja utilizado o usuário *root* para sua execução.

Os procedimentos seguintes foram efetuados nos sistemas operacionais Fedora Core 6.0 "Zod" e Fedora 7.0 "Moonshine", ambos Linux *kernel* 2.6. Fizeram-se testes em outras distribuições Linux, e também eles não apresentaram problemas.

A.1. Instalações dos Pacotes

O JPVM utiliza o interpretador Java para sua execução; portanto, faz-se imprescindível sua instalação.

Download: http://java.sun.com/javase/downloads/index.jsp

Comando para instalação:

```
# ./jdk-6u2-linux-i586.rpm
```

Como abordado na seção 5.2.1, para a coleta da carga dos recursos, aplica-se o pacote *dstat*, que é oferecido para a maioria das distribuições Linux.

Download: *http://dag.wieers.com/home-made/dstat/*

Comando para instalação:

```
# ./dstat-0.6.6-1.rh9.rf.noarch.rpm
```

Para que a comunicação entre os *hosts* seja livre e rápida, faz-se necessário instalar o pacote *rsh-server* (*remote shell*) e configurá-lo, de modo que não seja solicitado senha a cada tentativa de comunicação. Para apoiar as configurações do *rsh* sem senha, procede-se à instalação do pacote *xinetd.*

Download: *http://rpmfind.net/linux/rpm2html/search.php?query=rsh-server*

Comando para instalação:

```
# ./rsh-server-0.17-24.1.i386
```

Download: *http://rpmfind.net/linux/rpm2html/search.php?query=xinetd*

Comando para instalação:

```
# ./xinetd-2.3.14-11.i386.rpm
```

A pasta ***jpvm*** deve ser salva no diretório ***/root*** assim como a pasta ***mucode*** contendo o ***mucode.jar***. Qualquer alteração no nome de pasta, no nome de arquivos, ou até mesmo na sua localização, pode interferir na definição das variáveis de ambiente.

A.2. Configurações

Definir as variáveis de ambiente no arquivo *~/.bash_profile*.

```
# vi .bash_profile
```

```
#JAVA
JAVA_HOME=/usr/java/jdk1.6.0_02
export JAVA_HOME
JAVA_BIN=$JAVA_HOME/bin
CLASSPATH=$CLASSPATH:$JAVA_HOME:$JAVA_HOME/lib
PATH=$JAVA_BIN:$PATH
export JAVA_BIN CLASSPATH PATH

#JPVM
export CLASSPATH=/root/jpvm:$CLASSPATH
export CLASSPATH=/root/jpvm/Monitor:$CLASSPATH

#muCode
export CLASSPATH=/root/mucode/mucode.jar:.:$CLASSPATH
```

Adicionar ao */etc/hosts* os *IPs* (*internet protocol*) seu respectivo *hostname* (nome do anfitrião).

```
# vi /etc/hosts
```

```
192.168.42.240    gmm          gmm
192.168.42.241    las01 las01
192.168.42.242    las02 las02
192.168.42.243    las03 las03
192.168.42.244    las04 las04
192.168.42.245    las05 las05
192.168.42.246    las06 las06
192.168.42.247    las07 las07
192.168.42.248    las08 las08

192.168.42.231  les01   les01
192.168.42.232  les02   les02
192.168.42.233  les03   les03
192.168.42.234  les04   les04
192.168.42.235  les05   les05
```

Para que não haja erro de leitura das cargas dos recursos, no arquivo */usr/bin/dstat,* substituir o pipe ' | ' da linha 1366, posição 18 (aproximadamente), por um espaço em branco ' '. Ao finalizar a alteração, salvar e fechar o arquivo.

```
# vi /usr/bin/dstat
```

```
      'pipe': ' ',
```

A.2.1. RSH sem senha

A configuração do *rsh* sem senha deve ser efetuada com mesmo usuário, id e grupo em todos os *hosts*, conforme as etapas a seguir. Como já abordado, utilizou-se o usuário *root.*

Editar os arquivos abaixo e substituir *disable* = *yes* por *disable* = *no.*

```
# vi /etc/xinetd.d/rexec
# vi /etc/xinetd.d/rsh
# vi /etc/xinetd.d/rlogin
```

```
disable     =     no
```

Adicionar os comandos *rsh, rexec,*e *rlogin* no final do arquivo */etc/securetty*.

```
# vi /etc/securetty
```

```
rsh
rexec
rlogin
```

Editar o arquivo */etc/pam.d/login* e comentar com (#) a linha *auth required pam_securetty.so*.

```
# vi /etc/pam.d/login
```

```
# auth required pam_securetty.so
```

No arquivo /etc/pam.d/rsh:

- adicionar: auth required pam_rootok.so;
- comentar: # auth required pam_securetty.so;
- alterar: auth required pam_r*hosts*_auth.so para auth sufficient pam_r*hosts*_auth.so;

```
# vi /etc/pam.d/rsh
```

```
auth        required          pam_rootok.so
# auth     required           pam_securetty.so
auth      sufficient    pam_rhosts_auth.so
```

Criar um arquivo chamado *.rhost* no ambiente do usuário que terá autorização para executar o *rsh*. Importa que este arquivo tenha permissões de escrita e leitura para o dono, para o grupo e para todos. O conteúdo deste arquivo deve ser o *hostname* dos prováveis participantes da máquina virtual paralela. O mesmo conteúdo deve ser incluído nos arquivos: */etc/hosts.allow* e */etc/hosts.equiv*.

```
# vi ~/.rhosts
# vi /etc/hosts.allow
# vi /etc/hosts.equiv
```

```
localhost
gmm
las01
```

```
las02
las03
las04
las05
las06
las07
las08
les01
les02
les03
les04
les05
```

Para definir as permissões do arquivo *.rhosts,* conforme necessário, executar o comando:

```
# chmod 666 .rhosts
```

Em algumas redes, é necessário configurar (se assim o desejar, desabilitar) o *iptables*, para que a comunicação entre os *hosts* não seja bloqueada.

Todas as configurações tornam-se validadas, quando reiniciado *xinetd*:

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

Parando o xinetd:                                    [  OK  ]
Iniciando xinetd:                                    [  OK  ]
```

A.3. Testes das Configurações

Finalizadas todas as configurações, indica-se reiniciar os computadores para validá-las. Para testá-las, executar os comandos seguintes e verificar se o seu retorno aproxima-se dos demonstrados.

```
# java -version

java version "1.6.0_02"
Java(TM) SE Runtime Environment (build 1.6.0_02-b05)
Java HotSpot(TM) Client VM (build 1.6.0_02-b05, mixed mode, sharing)
```

```
# dstat -c -m -s -d -n

----total-cpu-usage---- ------memory-usage----- ----swap--- -dsk/total- -net/total-
usr sys idl wai hiq siq  used  buff  cach  free  used  free read  writ  recv  send
  1   0  99   0   0   0  197M  179M  268M  366M     0  1984M 564B 5233B    0     0
 21   5  16  58   0   0  219M  180M  339M  272M     0  1984M 8080k    0  3133B  132B
 25   5   8  63   0   0  220M  180M  347M  263M     0  1984M 7608k 1064k  158B  132B
 19   3  26  52   0   0  211M  180M  356M  264M     0  1984M 9556k  240k  152B    0
```

```
# java mucode.util.Launcher -port 2000

-- listing properties --
compression=false
debug=false
port=2000
ubiclasses=
Messages=true
timeout=30000
ubipackages=java.* javax.* mucode.*
errors=true
mucode: MuServer activated on port 2000
```

Iniciar o *jpvmDaemon* em cada host que fará parte do ambiente. Exibe-se o *hostname* e a porta de conexão, que deverão ser informados para o *console* do JPVM.

```
# java jpvm.jpvmDaemon
jpvm daemon: las08, port #59203
```

Após a execução dos *daemons* nos *hosts*, a máquina virtual paralela precisa ser informada de sua existência, portanto, executar um *console* interativo do JPVM para nela cadastrá-los. O *jpvmConsole* é dependente do *jpvmDaemon,* e só será possível executá-lo se o *jpvmDaemon* já estiver em execução.

```
# java jpvm.jpvmConsole

jpvm> add
        Host name   : las04
        Port number : 56794
jpvm> add
        Host name   : las06
        Port number : 2261
jpvm> add
        Host name   : las07
        Port number : 42892
jpvm> add
        Host name   : les05
        Port number : 41255
jpvm> add
        Host name   : las05
        Port number : 2286
jpvm> add
        Host name   : gmm
        Port number : 33065
jpvm> add
        Host name   : les04
        Port number : 40180
jpvm> add
        Host name   : les02
        Port number : 4287
jpvm> quit
```

Além de ser utilizado para cadastrar os *hosts*, pode-se aplicá-lo para listar os *hosts* já cadastrados, utilizando o comando *conf,* e para informar quais e quantas tarefas estão em execução nos *hosts*, utilizando o comando *ps*.

```
# java jpvm.jpvmConsole

jpvm> conf
9 hosts:
        les02
        les04
        gmm
        las05
        les05
        las07
        las06
        las04
        las08

jpvm> ps
les02, 0 tasks:
les04, 0 tasks:
gmm, 0 tasks:
las05, 0 tasks:
les05, 0 tasks:
las07, 0 tasks:
las06, 0 tasks:
las04, 0 tasks:
las08, 1 tasks:
        jpvm console

jpvm>
```

Caso todos os *hosts* adicionados no *console* forem aí listados, é possível testar a aplicação paralela no ambiente. Há um exemplo simples fornecido no diretório *~/jpvm/examples* chamado *hello*. Ao executar a aplicação *hello*, disparam-se objetos da classe *hello_other*, que devem estar no diretório *~/jpvm* dos *hosts* escravos, caso não tenha sido configurada a variável de ambiente apontando para o *~/jpvm/examples.*