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

LEANDRO CASTELLI SILVÉRIO

CONFECÇÃO E UTILIZAÇÃO DE AGENTES MÓVEIS PARA OBTENÇÃO DE ÍNDICES DE CARGA

MARÍLIA
2007

LEANDRO CASTELLI SILVÉRIO

CONFECÇÃO E UTILIZAÇÃO DE AGENTES MÓVEIS PARA OBTENÇÃO DE ÍNDICES DE CARGA

Monografia apresentada ao Curso de Graduação em Ciência da Computação, do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares da Rocha, como requisito para obtenção do grau de Bacharel em Ciência da Computação.

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

MARÍLIA
2007

SILVÉRIO, Leandro Castelli
Confecção e Utilização de Agentes Móveis para Obtenção de Índices de Carga / Leandro Castelli Silvério; orientadora: Profª. Drª. Kalinka R. L. J. C. BRANCO. Marília, SP: [s.n.], 2007.
102f.

Trabalho de Conclusão de Curso (TCC) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha.

1. Arquitetura Paralela Distribuída 2. Agentes Móveis

CDD: 005.2

"A minha princesa
Que, por motivos maiores,
Não está mais ao meu lado"

AGRADECIMENTOS

Aos meus pais e minha irmã, por terem dado todo o suporte necessário durante toda a minha vida.

Aos meus colegas de classe, principalmente ao Jhun, Teba, Rods, Boi, Marcos e Tadeu, que me ajudaram nessa grande jornada.

À Giulianna Marrega Marques e ao Ricardo José Sabatine que me auxiliaram em todo o meu projeto, sem os quais o término desse trabalho não teria sido possível.

À minha orientadora Profª Drª Kalinka Regina Lucas Jaquie Castelo BRANCO, por continuar acreditando em mim.

A todas as pessoas que, literalmente, me mandaram fazer o meu trabalho e que me deram forças para poder continuar.

E a Deus, que é o grande responsável por ter chegado até aqui.

SILVÉRIO, Leandro Castelli. Confecção e Utilização de Agentes Móveis para Obtenção de Índices de Carga. 2007. 102f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

RESUMO

A computação paralela/distribuída apresenta uma ascensão na área computacional devido à complexidade dos algoritmos e ao grande fluxo de dados que são utilizados (BRANCO, 2004). Uma área que continua sendo alvo de pesquisas são os algoritmos e as ferramentas de escalonamento de processos nessas plataformas afim de conseguir um melhor desempenho e um aproveitamento máximo do sistema. Através de estudos realizados na área da computação paralela/distribuída, mas especificamente no que diz respeito ao escalonamento de processos e na utilização de índices de carga nota-se a importância da utilização do poder computacional inerente a cada elementos de processamento componente desta plataforma. O balanceamento de carga, que visa a melhora no desempenho desses sistemas, se dá pela correta utilização da carga alocada aos elementos de processamentos da plataforma e a obtenção dessa carga não é algo trivial principalmente quando se pensa no não determinismo da mesma. Deste modo, este projeto tem como objetivo reduzir o tráfego na rede de comunicação e, conseqüentemente, obter um melhor desempenho dos elementos de processamento que compõe a plataforma através da confecção de agentes móveis que sejam responsáveis pela captura da carga de trabalho e conseqüentemente dos índices de carga de plataformas paralela/distribuída.

Palavras-Chave: Sistemas Paralelos / Distribuídos, Escalonamento de Processos, Balanceamento de Cargas, Agentes Móveis.

SILVÉRIO, Leandro Castelli. Confecção e Utilização de Agentes Móveis para Obtenção de Índices de Carga. 2007. 102f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

ABSTRACT

The parallel/distributed computing shows an ascent into computational area because of the complexity of algorithms and to large data flux that are used (BRANCO,2004). An area that continues being target of researches are scheduling processes algorithms and tools into that platform trying to improve the performance and maximize use of the system. Though researches into parallel/distributed computing area, more specifically with regard to the scheduling processes and the uses of load index was noted the importance of utilization of computational power inherent to each processing element of this platform. The Load Distributing, that aims to improve the performance of such systems, is reached for correctly utilization of allocated load into platform processing elements and the collect of this loads isn't simply especially when it's taken into consideration the not determinism of the same. Thus, this project aims to reduce traffic on the communication network and get a better performance of the processing elements that make up the platform through the manufacture of mobile agents who are responsible for the capture of the workload and therefore the load index of parallel / distributed platforms.

Keywords: *Parallel / Distributed Systems, Scheduling Processes, Load Distributing, Mobile Agents.*

LISTA DE FIGURAS

Figura 1. Classificação de Flynn (FLYNN,1972) 19
Figura 2. Modelo de computador com memória compartilhada (TANENBAUM, 2001) .. 23
Figura 3. Modelo de computador com memória privada (TANENBAUM, 2001) 23
Figura 4. Classificação das máquinas MIMD (TANENBAUM, 2002) 24
Figura 5. Multiprocessador baseado em barramento (TANENBAUM, 2002)..... 25
Figura 6. (a)Baseado em chaves (b)Omega network (TANENBAUM, 2002) 25
Figura 7. Multicomputadores baseado em barramento (TANENBAUM, 2002) 26
Figura 8. Multicomputadores com estrutura em (a) Grid e (b) Hypercube..... 26
Figura 9. Escalonador de Processos 27
Figura 10. Hierarquia da composição dos algoritmos de escalonamento (BRANCO,2004) 30
Figura 11. Taxonomia de Escalonamento Proposta por Casavant – Classificação Hierárquica (CASAVANT & KUHL,1988)..... 33
Figura 12. Sistema distribuído sem balanceamento de carga (SHIVARATRI *et al.*, 1992)36
Figura 13. Espaço ocupado pelo índice de desempenho na literatura (BRANCO, 2004)... 46
Figura 14. Exemplo com duas máquinas e dois recursos (BRANCO, 2004)..... 47
Figura 15. Mecanismos de Mobilidade (FUGGETTA; PICCO; VIGNA, 1998)..... 50
Figura 16. Funções Principais da Biblioteca *Aglets* (LANGE & OSHIMA, 1998). 59
Figura 17. Sistema Paralelo/Distribuído sem balanceamento (MARQUES, 2007)..... 63
Figura 18. Ambiente Paralelo/Distribuído balanceado (MARQUES, 2007) 63
Figura 19. Tahiti Server..... 64
Figura 20. Execução do dstat (MARQUES, 2007) 66
Figura 21. Gráfico da matriz 100x100 em sistema homogêneo..... 82
Figura 22. Gráfico da matriz 1000x1000 em sistema homogêneo..... 83
Figura 23. Gráfico da matriz 2000x2000 em sistema homogêneo..... 83
Figura 24. Gráfico da matriz 100x100 em sistema heterogêneo 84
Figura 25. Gráfico da matriz 1000x1000 em sistema heterogêneo 84
Figura 26. Gráfico da matriz 2000x2000 em sistema heterogêneo 85

LISTA DE EQUAÇÕES

Equação 1 44
Equação 2 46
Equação 3 48
Equação 4 67

LISTA DE TABELAS

Tabela 1. Paradigmas de Código Móvel 51
Tabela 2. Propriedades dos Agentes (FRANKLIN; GRAESSER, 1996) 53
Tabela 3. Eventos e *Listeners* (LANGE & OSHIMA, 1998). 60
Tabela 4. Resultado dos testes em Sistema Homogêneo 82
Tabela 5. Resultados dos testes em Sistema Heterogêneo 84

SUMÁRIO

AGRADECIMENTOS v
RESUMO vi
ABSTRACT vii
Lista de Figuras viii
Lista de Equações ix
Lista de Tabelas x
SUMÁRIO xi
CAPÍTULO 1. INTRODUÇÃO 13
1.1. Objetivos 15
1.2. Organização da Monografia 15
CAPÍTULO 2. COMPUÇÃO PARALELA/DISTRIBUÍDA 17
2.1. Computação Paralela 17
2.1.1. Definição 18
2.2. Sistemas Computacionais Distribuídos 19
2.2.1. Definição 20
2.2.2. Arquitetura para Sistemas Computacionais Distribuídos 22
2.3. Escalonamento de Processos 27
2.3.1. Componentes de um Algoritmo de Escalonamento. 28
2.3.2. Taxonomia de Escalonamento 31
2.3.3. Balanceamento de Carga 35
2.4. Considerações Finais 36
CAPÍTULO 3. ÍNDICES DE CARGAS E DESEMPENHO 38
3.1. Avaliação de Desempenho 39
3.2. Cargas de Trabalho 40
3.3. Índices de Cargas 41
3.3.1. Estado da Arte 44
3.4. Índice de Desempenho 45
3.5. Considerações Finais 48
CAPÍTULO 4. CÓDIGO MÓVEL 49

4.1. Agentes 52
4.1.1. Agentes Móveis 53
4.1.1.1. Aglets 55
4.1.1.1.1 Elementos Básicos 57
4.1.1.1.2 Modelo de Eventos de Aglets 60
4.2. Considerações Finais 61
CAPÍTULO 5. CONFECÇÃO DO AGENTE MÓVEL 62
5.1. Tahiti Server 64
5.2. Agente Monitor 65
5.3. Desenvolvimento do Agente Móvel 68
5.4. Problemas encontrados 75
5.5. Considerações Finais 76
CAPÍTULO 6. TESTE DO AMBIENTE COM AGENTE MÓVEL 78
6.1. JPVM 78
6.2. Testes 80
CAPÍTULO 7. CONCLUSÃO 86
7.1. Trabalhos Futuros 87
REFERÊNCIAS BIBLIOGRÁFICAS 88
Apêndice A – Código fonte do agente móvel para JPVM. 92
Apêndice B – Instalação da Plataforma Aglets 95
Anexo A – Código fonte mult_math (exemplo do JPVM) 100

CAPÍTULO 1. INTRODUÇÃO

Um dos principais objetivos da maioria dos sistemas computacionais é a utilização eficiente de seus recursos. Em se tratando de um sistema paralelo/distribuído, esse objetivo se torna mais importante, pois está relacionado com o desempenho de várias máquinas (BRANCO, 2004).

O escalonamento de processos em plataformas paralelas/distribuídas está diretamente ligado ao desempenho do mesmo, uma vez que uma estratégia de escalonamento eficaz garante um aproveitamento maior dos recursos do sistema, aumentando assim o desempenho do mesmo. Este escalonamento faz com que cargas ou tarefas sejam alocadas a processadores, e um dos objetivos é prover o balanceamento de cargas a fim de dividi-las de maneira adequada.

Os algoritmos de balanceamento de cargas visam um melhor desempenho do sistema, distribuindo as cargas de forma equilibrada entre as máquinas do ambiente.

Independente do tipo de escalonamento a ser considerado (estático, dinâmico, tempo-real, entre outros), um elemento essencial para a política de escalonamento é a aferição da carga.

A aferição da carga constituindo um índice de carga não é um processo trivial, particularmente considerando-se a natureza dinâmica e não determinística das aplicações que são executadas. Desta forma, bons índices de carga podem levar à obtenção de melhorias no desempenho global observado no sistema.

Em (BRANCO, 2004) é apresentada uma forma de obtenção de índices de carga e desempenho fazendo uso de rotinas executadas a partir do *software* de escalonamento e que acessam informações do sistema de tempos em tempos para captura da carga.

Entretanto, acredita-se que por meio do uso de agentes móveis é possível coletar essas cargas e fornecer o desempenho de cada máquina do ambiente para o escalonador.

Código móvel pode ser definido de maneira informal como a capacidade de mudar de forma dinâmica as ligações entre fragmentos de código e a localidade de sua execução. A habilidade de re-alocar código é um conceito interessante que originou uma série de pesquisas (KERSHENBAUM; HARRISON; CHESS,1995).

Diversas são as formas de um código móvel ter sua execução realizada, e o paradigma mais usual é o de agentes móveis. Um agente pode ser caracterizado como um programa que age de forma autônoma, em busca de um objetivo programado anteriormente e para isso pode utilizar interações com outros agentes ou ambientes móveis se necessário (KERSHENBAUM; HARRISON; CHESS,1995).

Sendo um agente móvel um elemento de *software* auto contido, responsável pela execução de uma atividade, capaz de autonomamente migrar por meio de uma rede, torna-se uma solução atrativa para a obtenção de índices de carga em ambientes paralelos/distribuídos, uma vez que pode reduzir o tempo de captura desses índices.

Agente móvel constitui uma tecnologia que promete tornar o uso de sistemas distribuídos mais fáceis tanto no seu projeto, implementação e manutenção. Agentes móveis não estão restritos aos sistemas em que iniciaram sua execução, pois eles têm a habilidade de se transportar de um sistema para outro através de uma rede. Este recurso permite um agente móvel mover-se para o sistema que possui um objeto com o qual o agente deseja interagir, obtendo a vantagem de residir na mesma máquina ou rede do objeto (ARIDOR & LANGE, 1998).

Deste modo, neste trabalho serão utilizados agentes móveis para a captura de informações necessárias para o cálculo dos índices de carga.

1.1. Objetivos

O uso de sistemas computacionais distribuídos tem se tornado nos últimos anos um atrativo principalmente com a ascensão dos *Clusters* e da computação em Grades. Conseqüentemente o escalonamento de processos nesses sistemas constitui algo motivador e desafiador para os acadêmicos.

Buscando aprimorar o escalonamento de processos e alcançar melhor desempenho nesta plataforma, este trabalho tem como objetivo principal a criação de agentes móveis para coletar as informações (informações pertinentes ao estado e capacidade dos principais recursos do sistema: cpu, memória, disco, e rede) das máquinas pertencentes à plataforma paralela/distribuída. Utiliza-se essas informações como base do cálculo dos índices de carga de cada recurso, os quais são utilizados para cálculo dos índices de desempenho de cada máquina e do ambiente como um todo.

1.2. Organização da Monografia

Tem-se como estrutura desta monografia 6 capítulos, sendo o primeiro a introdução, e os demais capítulos são detalhados a seguir.

No Capítulo 2 é feita uma revisão bibliográfica sobre sistemas paralelos/distribuídos, suas características e classificações. São descritos também conceitos de escalonamento de processos, suas taxonomias e objetivos, onde é destacado o índice de cargas.

No Capítulo 3 são apresentadas características de índices de cargas e de desempenho apresentando seu estado da arte.

No Capítulo 4 é feito um estudo sobre códigos móveis, onde são mostrados também características de agentes móveis.

No Capítulo 5 são descritos a parte de implementação do projeto e as características da plataforma *Aglets*.

No Capítulo 6 são mostrados testes realizados com o agente móvel e seus resultados.

No Capítulo 7 está a conclusão do projeto e os trabalhos futuros a ele relacionados.

CAPÍTULO 2. COMPUÇÃO PARALELA/DISTRIBUÍDA

Este Capítulo apresenta os conceitos básicos de computação paralela distribuída detalhando os dois pilares de sua formação: Sistemas Computacionais Distribuídos e a Computação Paralela

2.1. Computação Paralela

A ciência se encontra em uma constante evolução. E a cada passo que ela dá são necessários mais experimentos, e estes vão se tornando cada vez mais complexos e demorados. Alguns desses experimentos são tão complexos que necessitam de um computador muito potente a fim de conseguir um menor tempo de resposta.

Durante décadas as arquiteturas de computadores incorporaram paralelismo em vários níveis de *hardware* para conseguir acompanhar a alta velocidade que a ciência demanda. Já nos tempos atuais as arquiteturas podem incorporar o paralelismo em seus níveis mais altos. Sistemas com alto nível de paralelismo são considerados os mais rápidos e lideram na área cientifica (QUINN, 1994).

2.1.1. Definição

Existem tarefas que necessitam de um grande poder computacional, sendo complexas e lentas para serem resolvidas em computadores com arquiteturas convencionais. Executar essas tarefas como uma única tarefa é difícil. Para facilitar pode-se dividi-la em pequenas partes que podem ser executadas de uma maneira mais ágil. Processamento paralelo trata exatamente disso, dividir grandes tarefas em pequenas partes e executá-las de modo paralelo, ou seja, em mais de um elemento de processamento ao mesmo tempo.

Tarefas concorrentes consistem em dois ou mais processos disputando de maneira concorrente a utilização do processador para serem executados. A concorrência pode ocorrer tanto em sistemas paralelos quanto em sistemas com um único processador. Em sistemas com um único processador os processos recebem uma fatia de tempo (*quantum*) para simular uma execução em paralelo, essa técnica é chamada de pseudoparalelismo. Para que se caracterize um processamento paralelo é preciso que dois ou mais processos executem ao mesmo tempo (MARQUES, 2007).

Existem diversas arquiteturas e formas como um sistema ou uma plataforma pode se conectar e se organizar. Uma forma de classificar essas arquiteturas foi proposta por Flynn (FLYN, 1972).

Flynn (FLYN, 1972) classifica as máquinas por meio de duas características importantes: fluxo de instruções e fluxo de dados. O fluxo de instruções é definido pela quantidade de aplicações que o sistema executa ao mesmo tempo e o fluxo de dados está relacionado ao fluxo de operandos do sistema.

É possível classificar o sistema da seguinte maneira: SISD (*single instruction, single data*) são os computadores tradicionais com apenas um processador, SIMD (*single*

instruction, multiple data) nessa categoria se encaixam os *arrays processors*, MISD (*Multiple instruction, single data*) não existem computadores nessa categoria e MIMD (*multiple instruction, multiple data*) que são os sistemas distribuídos. A classificação de Flynn é ilustrada na Figura 1.

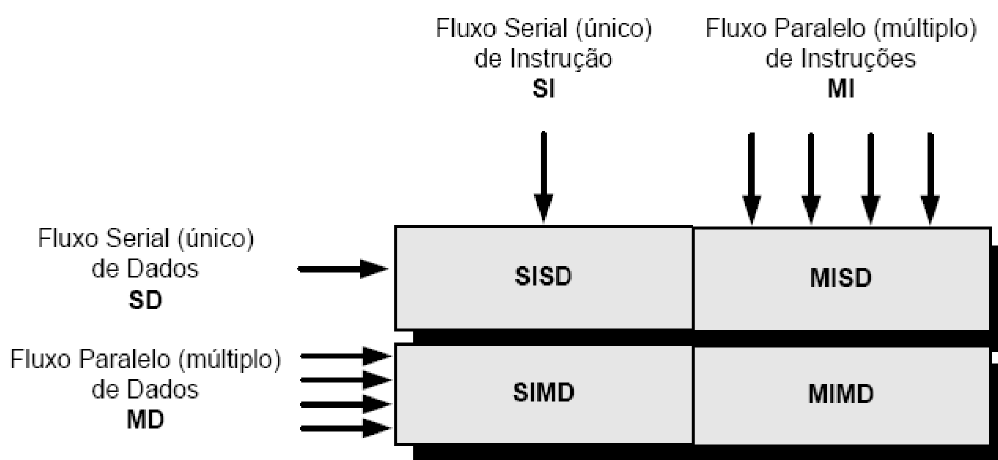

Figura 1. Classificação de Flynn (FLYNN,1972)

2.2. Sistemas Computacionais Distribuídos

Antes da década de 80 os computadores eram grandes, possuíam um baixo poder de processamento e eram extremamente caros, e eram acessíveis somente às grandes organizações. O fato de não haver um método confiável de comunicação fazia com que fossem operados de maneira independentes (TANENBAUM, 1995).

Com o passar do tempo foram desenvolvidos microprocessadores mais potentes e com um custo muito reduzido, aumentando a relação custo/desempenho. Um outro avanço tecnológico foi a melhoria nas redes de computadores, permitindo interligar um grande número de computadores de forma que pudessem trocar informações entre si de uma maneira rápida e confiável.

A convergência dos sistemas distribuídos com a computação paralela permitiu que vários computadores trabalhassem juntos de uma maneira mais rápida com fácil implementação, confiável e mais barata do que um computador de grande porte (*mainframe*). A essa junção de tecnologias dá-se o nome de Sistemas Computacionais Distribuídos.

2.2.1. Definição

"*Um sistema distribuído é uma coleção de computadores independentes que são vistos pelos usuários como um único computador*" (TANENBAUM,1995).

Um sistema distribuído nada mais é do que uma série de computadores autônomos interligados entre si através de uma rede de alta velocidade que cooperam de alguma maneira entre si, fazendo isso de forma transparente ao usuário do sistema, ou seja, para o usuário será como utilizar um sistema centralizado.

Comparado aos sistemas centralizados, os sistemas distribuídos apresentam algumas vantagens (TANENBAUM,1995):

- Econômicas: microprocessadores possuem uma melhor relação custo/desempenho do que computadores de grande porte.
- Velocidade: um sistema distribuído pode ter poder total de processamento maior que de um computador de grande porte.
- Distribuição inerente: algumas aplicações envolvem máquinas espacialmente dispersas.
- Confiabilidade: caso uma das máquinas falhe, o sistema continuará executando, reduzindo apenas seu desempenho.

- Crescimento incremental: pode-se aumentar o poder computacional adicionando pequenos componentes ou novas máquinas.

Mas os sistemas distribuídos ainda possuem alguns problemas que devem ser considerados, são eles:

- Rede de comunicação: a rede pode causar problemas ocasionando a perda de pacotes de dados;
- Segurança: devido o compartilhamento de dados, o acesso a dados secretos pode ser facilitado.

Além disso, os sistemas distribuídos apresentam alguns aspectos de extrema importância que devem ser observados, são eles:

- Transparência: propriedade que garante que o fato dos processos e recursos estarem fisicamente distribuídos entre vários computadores não seja visto pelo usuário, dando a falsa impressão de ser um sistema centralizado.
- Flexibilidade: facilita o trabalho de sistemas com implementações diferentes.
- Confiabilidade: capacidade de o sistema continuar operando mesmo diante de falhas. Essa característica se divide em três categorias:
- Disponibilidade: fração de tempo em que o sistema está em condições de ser usado.
- Segurança: garantia de proteção e integridade dos dados.
- Tolerância à faltas: usuário não percebe a reorganização do sistema diante de uma falha.

- Desempenho: um sistema distribuído deve apresentar um desempenho no mínimo igual a um sistema centralizado. São utilizadas várias formas de medir esse desempenho (*throughput*, utilização do sistema, tempo de resposta, entre outros).

2.2.2. Arquitetura para Sistemas Computacionais Distribuídos

A grande maioria dos sistemas paralelos distribuídos utiliza arquitetura MIMD. As máquinas MIMD são classificadas em dois grupos: multiprocessadores e multicomputadores. A diferença entre os dois está na maneira que a memória está dividida. Nos multiprocessadores a memória é a mesma para todos, ou seja, todos compartilham a mesma memória (*shared memory)*, *s*e duas máquinas lerem o endereço X as duas obterão a mesma resposta, pois estão usando a mesma memória (FLYNN, 1972) (FLYNN & RUDD, 1996).

Já os multicomputadores possuem memória particular (*private memory)*, ou seja, cada um tem sua própria memória. Se duas máquinas lerem o endereço X, não se pode garantir que elas irão obter a mesma informação, pois estão lendo de lugares diferentes.

Nos sistemas que possuem memória compartilhada, também chamados de fortemente acoplados (*tightly coupled)*, os processadores utilizam uma mesma memória principal, permitindo que os processos se comuniquem através dessa memória. A utilização dessa memória compartilhada facilita a programação, pois para um processo se comunicar com outro basta escrever na memória que qualquer outro processo poderá ler, mas sempre tem que se tomar cuidado com as regiões críticas sendo necessário o uso de semáforos e monitores para garantir a exclusão mútua.

Esse tipo de modelo é ilustrado na Figura 2.

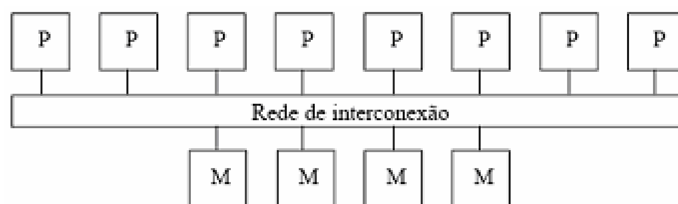

Figura 2. Modelo de computador com memória compartilhada (TANENBAUM, 2001)

Já os sistemas que possuem memória privada, também chamados de fracamente acoplados (*loosely coupled)*, têm uma memória principal independente para cada processador. A comunicação entre os processos é feita por meio da troca de mensagens. O projeto de uma máquina com memória privada é mais simples do que uma máquina com memória compartilhada, porém o sistema de troca de mensagens apresenta vários problemas como controle de fluxo, mensagens perdidas, buferização e bloqueamento (TANENBAUM, 2001). O modelo computacional de memória distribuída é ilustrado pela Figura 3.

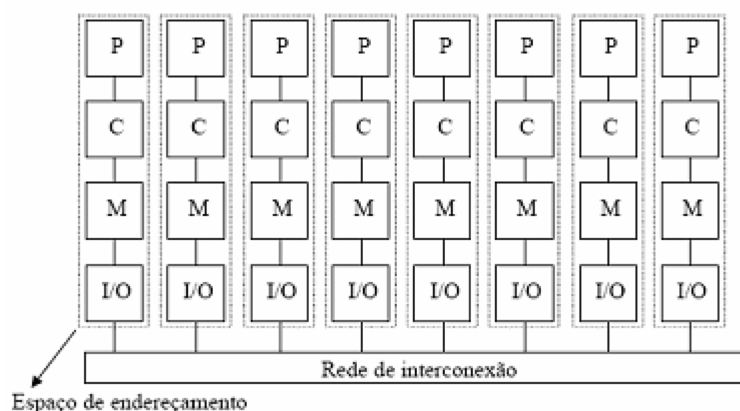

Figura 3. Modelo de computador com memória privada (TANENBAUM, 2001)

Em resumo os multiprocessadores são mais simples de serem programados, porém mais complexos de serem construídos. Entretanto os multicomputadores são mais fáceis de serem construídos e mais complexos de serem programados.

Além da classificação de acordo com o tipo de memória utilizada, cada uma dessas categorias ainda pode ser dividida levando em consideração a sua rede de interconexão. Elas são divididas em arquitetura de interconexão através de barramento (*bus*) e arquitetura de interconexão chaveada (*switched*). Na Figura 4 são mostradas essas classificações.

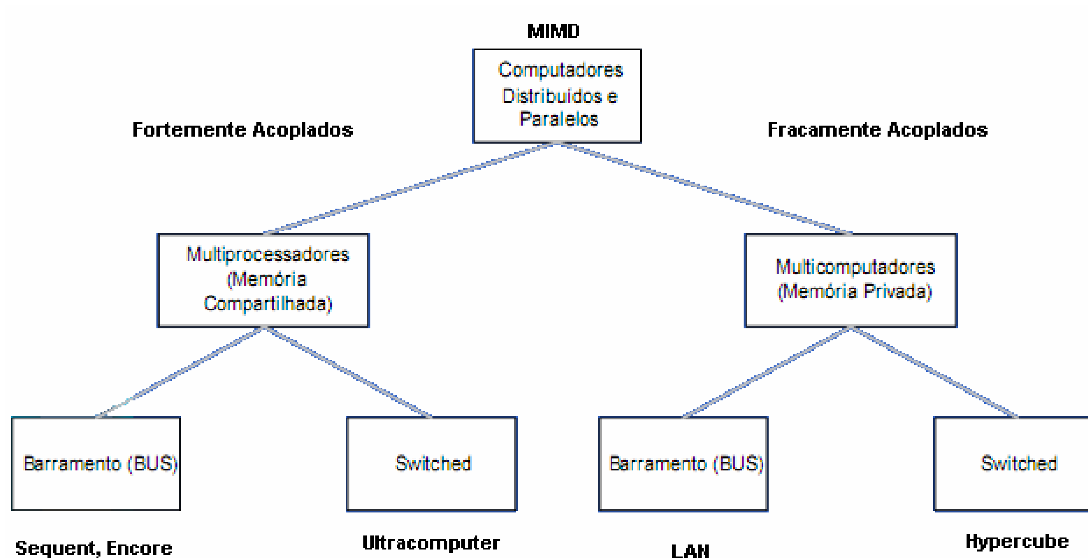

Figura 4. Classificação das máquinas MIMD (TANENBAUM, 2002)

Os multiprocessadores baseados em barramento consistem em um único barramento ligando todos os processadores entre si e na memória principal. Como existe apenas uma memória se um processador escrever nela qualquer processador que ler esse endereço irá conseguir essa informação. Mas quando utilizado com um grande número de processadores esse barramento é facilmente sobrecarregado e o desempenho diminui. Esse modelo é ilustrado na Figura 5.

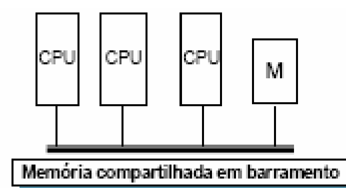

Figura 5. Multiprocessador baseado em barramento (TANENBAUM, 2002)

Nos multiprocessadores baseados em chaves as memórias são divididas em módulos e todos os módulos são ligados a todos os processadores através de chaves. Todos os processadores podem acessar todas as memórias, mas caso dois ou mais queiram acessar o mesmo módulo de memória somente um deles poderá acessar enquanto os outros esperam. Para um sistema com n processadores e n memórias são necessário n^2 chaves, o que pode ser muito grande dependendo do tamanho do sistema. Um dos modos de diminuir esse número de chaves é a utilização da *Omega network* que é mostrada na Figura 6.

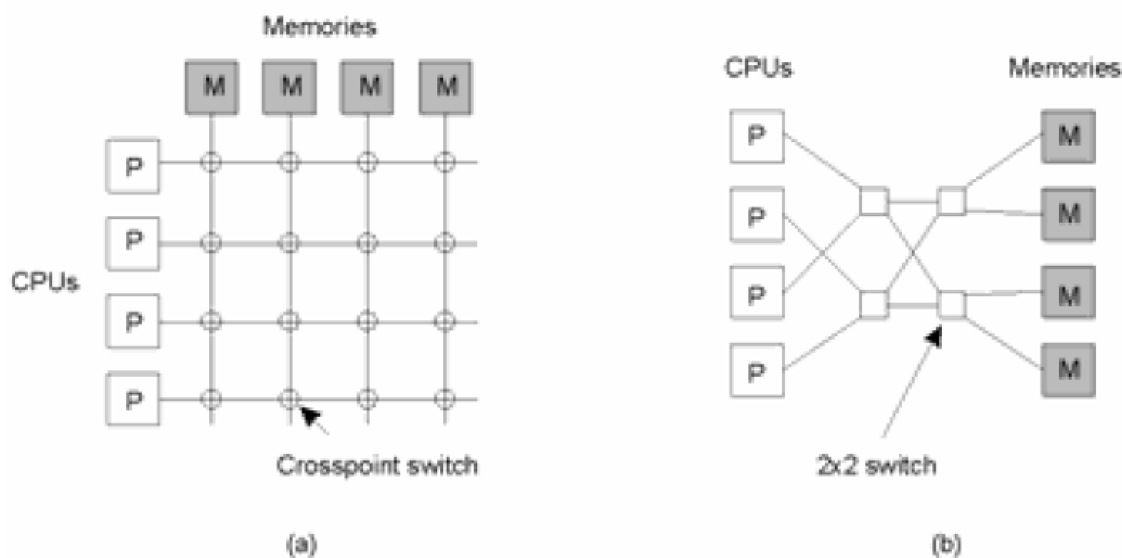

Figura 6. (a)Baseado em chaves (b)Omega network (TANENBAUM, 2002)

Em sistemas multicomputadores baseados em barramento cada processador acessa somente sua própria memória privada, sendo necessário somente um modo de

comunicação entre as CPUs. Sendo assim esse barramento terá um tráfego bem menor se comparado ao dos multiprocessadores baseados em barramento. Esse modelo é apresentado na Figura 7.

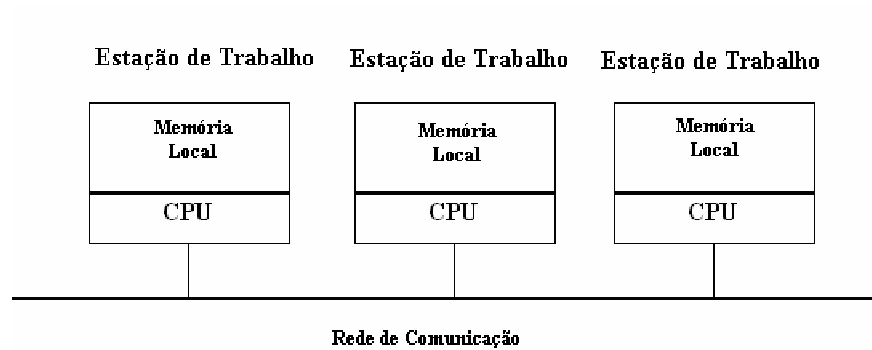

Figura 7. Multicomputadores baseado em barramento (TANENBAUM, 2002)

Os multicomputadores baseados em chaves possuem a mesma característica de que cada processador acessa somente sua memória privada. As duas arquiteturas mais utilizadas são os *grids* e os *hypercubes.*

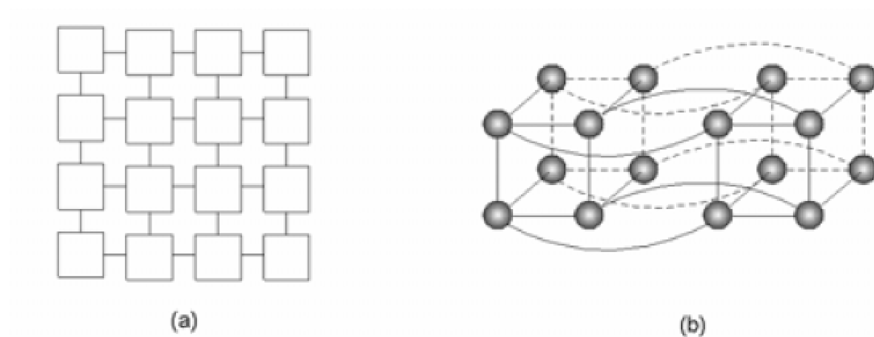

Figura 8. Multicomputadores com estrutura em (a) Grid e (b) Hypercube

2.3. Escalonamento de Processos

Quando mais de um processo está pronto para ser executado, o sistema operacional deve decidir qual processo será executado primeiro, por quanto tempo continuará executando esse processo e qual será o próximo processo a ser executado. Essa tarefa é feita pelo escalonador, e é chamada de escalonamento de processos (TANENBAUM, 1999). Esse processo pode ser visualizado na Figura 9.

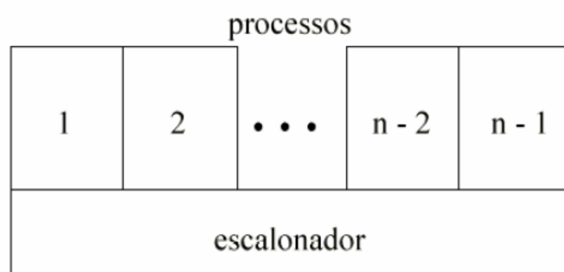

Figura 9. Escalonador de Processos

O escalonador atribui os processos às unidades de processamento e decide quem deve ser executado e quem deve sair. E essa escolha se torna mais difícil se tratando de um sistema paralelo/distribuído uma vez que o escalonamento não será simplesmente efetuado pelo sistema operacional, e sim por um escalonador de processos. Neste caso, o escalonamento está diretamente ligado ao aproveitamento máximo do sistema.

É através do escalonamento de processos que se pode utilizar cada recurso de uma maneira mais adequada. Uma boa política de escalonamento faz com que os recursos sejam alocados de maneira eficiente fazendo com que a plataforma envolvida obtenha um melhor desempenho.

2.3.1. Componentes de um Algoritmo de Escalonamento.

O escalonador se baseia em vários critérios para realizar o escalonamento, os principais são (TANENBAUM, 1999):

- justiça: garantir que todos os processos do sistema terão chances iguais de uso do processado;
- eficiência: manter os recursos ocupados 100% do tempo;
- tempo de resposta: minimizar o tempo de resposta para os usuários interativos;
- *turnaround time*: minimizar o tempo que os usuários batch devem esperar pela saída;
- *throughput*: maximizar o número de tarefas processadas na unidade de tempo.

Os escalonamentos de processos são baseados em uma política de escolha e em mecanismos. As políticas são responsáveis pela definição do modo que o escalonamento será efetuado, gerenciando os mecanismos de maneira que consigam realizar o escalonamento da maneira esperada.

São essas políticas que definem os objetivos do escalonamento. Dentre eles é possível citar a diminuição do tempo médio de resposta, diminuição dos atrasos na comunicação, maximização da utilização dos recursos disponíveis, e o balanceamento das cargas entre outros.

Essas políticas definem regras e procedimentos que devem ser seguidos para que o escalonamento de processos ocorra, elas se dividem da seguinte forma segundo Shivaratri (SHIVARATRI *et al.*, 1992):

- política de transferência (*transfer policy*): define se o nó participará do processo como emissor ou receptor. Um nó é definido como emissor se uma nova tarefa originada do nó faça o comprimento da fila da CPU exceder o ponto inicial T. E um nó é definido como receptor se aceitando uma tarefa ele faz com que o comprimento de fila exceda o ponto inicial T (quando levado em consideração o comprimento de fila de Cpu para tomada de decisões);
- política de seleção: escolha da tarefa a ser transferida (na maioria das vezes a mais recente);
- política de localização: encontrar um parceiro de transferência (emissor ou receptor), uma vez que o nó já foi identificado como emissor ou receptor pela política de transferência;
- política de informação: decide quando, onde e quais informações sobre os estados dos outros nós do sistema serão coletadas. Existem três políticas de informação:
    - política orientada a demanda: as informações são coletadas somente quando o nó é definido como emissor ou receptor, sendo assim um candidato a iniciar o compartilhamento da carga;
    - política periódica: as informações são coletadas de tempos em tempos;
    - política orientada a mudanças de estados: as informações sobres os nós são enviadas conforme mude seu grau de estado;

Os mecanismos são os responsáveis pela definição de como o escalonamento de processos será feito. São divididos em: (BRANCO,2004)

- mecanismo de métrica de carga: define o modo de métrica de carga em cada um dos nós do sistema;
- mecanismo de comunicação de carga: define o método utilizado para efetuar a troca de informações de carga entre os nós do sistema;
- mecanismo de migração: define o protocolo utilizado quando um processo migrar entre os nós do sistema.

A Figura 10 ilustra a composição dos algoritmos de escalonamento de processos de forma hierárquica.

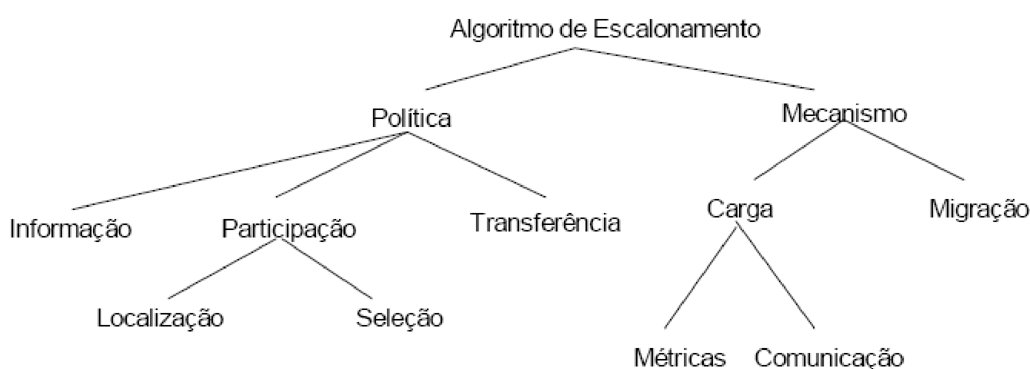

Figura 10. Hierarquia da composição dos algoritmos de escalonamento (BRANCO,2004)

Cada algoritmo de escalonamento possui seus objetivos, e junto desses objetivos são necessárias às métricas para que possa haver uma avaliação do desempenho do algoritmo em determinada situação (BRANCO, 2004).

Na maioria das vezes, mesmo apresentando o mesmo objetivo, os algoritmos tendem a obter resultados diferentes, sendo difícil apontar um determinado algoritmo que ofereça um desempenho excelente em todos os casos, pois há um grande número de combinações entre *hardware* e *software* diferentes.

Essas diferenças entre os resultados podem ser atribuídas principalmente a três fatores: a plataforma computacional, o algoritmo de escalonamento e o tipo de aplicações utilizadas. Sendo assim, cada algoritmo de escalonamento possui um melhor desempenho dependendo das condições em que será aplicado, podendo ser escolhidos de acordo com previsões de tipo e quantidade de trabalho que irá ocorrer.

As políticas de escalonamento geralmente são voltadas para certos tipos de aplicações, moldando suas características de acordo com a necessidade impostas pelos requisitos dessas aplicações (BRANCO, 2004). Essas aplicações são divididas em:

- *CPU-Bound*: aplicações que exigem uma alta demanda de processamento e pouca demanda por entradas/saídas;
- *Memory-Bound*: aplicações que utilizam muita memória;
- *I/O-Bound:* aplicações que possuem alta demanda para entrada/saída;
- *Network-Bound:* aplicações que exigem muita comunicação.

2.3.2. Taxonomia de Escalonamento

Muitas taxonomias foram propostas para classificar os algoritmos de escalonamento de processos. Através dessa classificação é possível mostrar grande parte das características dos algoritmos de escalonamento.

Algumas características são esperadas de um algoritmo de escalonamento, são elas(WANG;MORRIS, 1985) apud (BRANCO,2004):

- desempenho ideal do sistema como um todo: utilizar as unidades de processamento de maneira a obter o aproveitamento máximo do sistema.
- eqüidade de serviço: o desempenho deve ser uniforme, independente da origem das tarefas.
- tolerância a falhas: o sistema deve continuar operando caso algum nó se torne inoperante.

Quanto a classificação dos escalonadores, não existe uma taxonomia padrão que deva ser seguida. Uma taxonomia muito utilizada é a proposta por Casavant e Kuhl (1988). Ela classifica os tipos de escalonamento de forma hierárquica. Essa taxonomia se baseia em alguns pontos:

- tipos de informações usadas para que as tarefas sejam escalonadas;
- lugar em que as tarefas serão alocadas quando ocorrer o re-escalonamento;
- forma através da qual são feitas as tomadas de decisão e a obtenção de informações.

A Figura 11 ilustra a classificação hierárquica proposta por Casavant e Kuhl (1988).

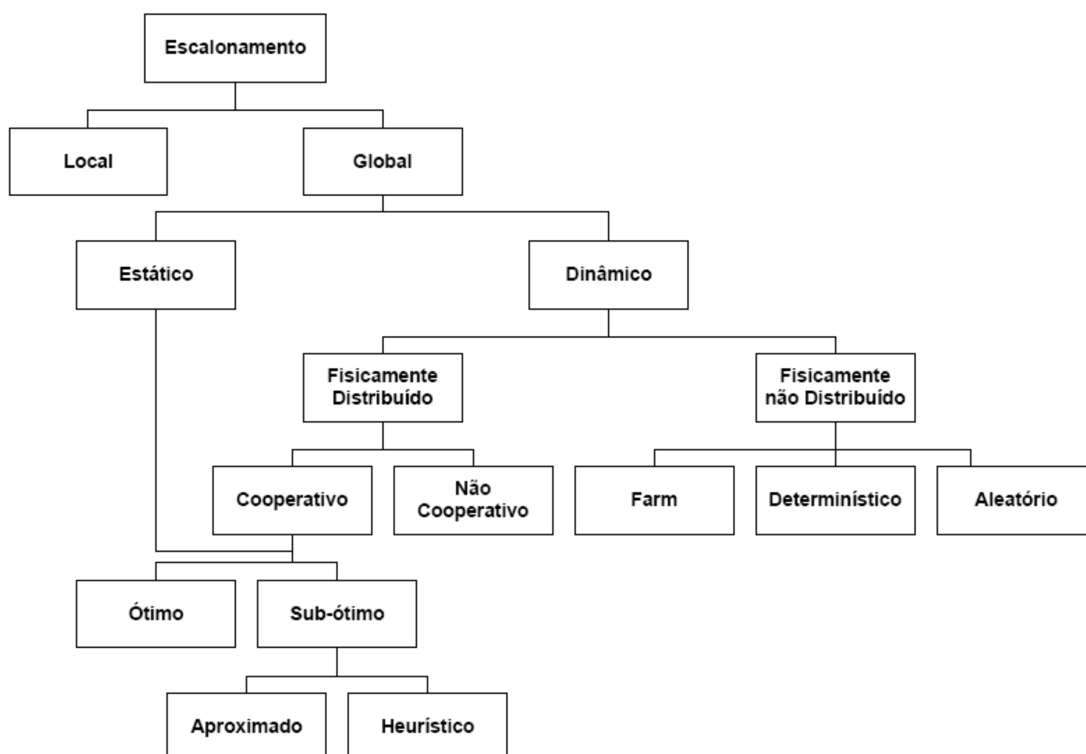

Figura 11. Taxonomia de Escalonamento Proposta por Casavant – Classificação Hierárquica (CASAVANT & KUHL,1988).

No topo da hierarquia se encontram os tipos de escalonamento local e global. O escalonamento local é composto pela disputa dos processos pela atenção do processador. A cada processo é atribuída uma fatia de tempo (*quantum*) e, esses processos dividem o tempo de um único processador.

O escalonamento global trata do escalonamento em sistemas com várias unidades de processamento. Ele decide a divisão das tarefas nas unidades de processamento. O escalonamento global pode ser tratado de duas maneiras: centralizada e distribuída.

Na centralizada existe um processador responsável pelo escalonamento de todo o sistema. Na distribuída a decisão é tomada por cada unidade de processamento, ou seja, cada processador decide o seu escalonamento. Essa forma de escalonamento se divide em

duas ramificações baseadas no momento que as decisões do escalonamento são tomadas: estático e dinâmico.

Na estratégia estática as decisões são tomadas antes que os processos sejam executados, necessitando um conhecimento prévio dos processos a serem executados no sistema (conhecimento do seu tempo de execução e dos recursos de processamento). Esse modo de escalonamento não permite preempção.

Na estratégia dinâmica não se tem um conhecimento dos processos a serem executados no sistema. Dessa forma o escalonamento é feito durante a execução. Enquanto os processos são executados as informações são coletadas, podendo assim reajustar as cargas de cada processador conforme sua necessidade. Para diferenciar máquinas ociosas e sobrecarregadas são utilizados índices de cargas.

Se um escalonamento dinâmico permite que os processos migrem enquanto estão sendo executados, é chamado de preemptivo, mas se não permitir é chamado de não-preemptivo.

Nos sistemas em que o escalonamento está fisicamente distribuído entre os processadores do sistema existe uma classificação de acordo com a forma de tratar os processos, podendo ter um escalonamento cooperativo ou não-cooperativo.

No cooperativo os processadores trabalham de maneira conjunta fazendo com que as cargas sejam alocadas de maneira a aproveitar os recursos do sistema de uma maneira mais adequada. Esse modo pode ter soluções classificadas em ótimas e sub-ótimas. As soluções sub-ótimas utilizam heurísticas ou resultados aproximados para conseguir uma solução aceitável para o problema. Já no caso do escalonamento não-cooperativo, cada processador trabalha de maneira isolada, decidindo como alocar seus próprios recursos.

Uma outra taxonomia foi proposta por Shivarati (SHIVARATI *et al.*, 1992), ele propõe a utilização de algoritmos chamados adaptativos. Esses algoritmos estariam em

uma classe especial dos algoritmos dinâmicos. Eles se adaptam dinamicamente a sua atividade, mudando seus parâmetros e até a sua política. Essas decisões são baseadas em estados anteriores e atuais do sistema.

O termo escalonamento adaptativo muitas vezes é confundido com escalonamento dinâmico, mas possuem conceitos diferentes. Enquanto os algoritmos dinâmicos se baseiam somente nas informações atuais do sistema, os algoritmos adaptativos tomam suas decisões com base em um histórico do sistema (PITANGA, 2003; MARQUES, 2007).

2.3.3. Balanceamento de Carga

O escalonamento de processos consiste em decidir onde cada tarefa será executada, sendo essa tarefa comum tanto aos sistemas distribuídos quanto aos multiprocessadores.

Muitas vezes, os elementos de processamento terminam suas tarefas em tempos diferentes. Isso faz com que enquanto alguns processadores estão sobrecarregados outros estão ociosos. É função do escalonador manter um equilíbrio entre os processadores, e a esse equilíbrio é dado o nome de balanceamento de cargas. Um exemplo de um sistema distribuído sem provimento de balanceamento de cargas pode ser observado na Figura 12.

Figura 12. Sistema distribuído sem balanceamento de carga (SHIVARATRI *et al.*, 1992)

O escalonador tem que levar em consideração a heterogeneidade tanto arquitetural quanto configuracional da plataforma em questão, pois máquinas diferentes tendem a executar a mesma tarefa em tempos diferentes (BRANCO, 2004). Definir uma carga de acordo com o poder de processamento de cada nó é um fator muito importante para o balanceamento de cargas.

2.4. Considerações Finais

Um dos objetivos de um sistema distribuído é o alto desempenho computacional que ele procura. Entretanto, somente um grande número de processadores não garante esse alto desempenho desejado. Faz-se necessário que cada elemento seja bem aproveitado para que esse alto desempenho seja alcançado (BRANCO,2004).

Para se obter um bom aproveitamento do sistema é necessário uma distribuição dos processos entre os processadores existentes. Ao fazer a alocação dos processos é

necessário distribuir as cargas de acordo com as características e capacidade de cada processador, garantindo que a carga seja suficiente e não ultrapasse os limites do processador.

Dessa forma o trabalho será distribuído de maneira “uniforme” (considerando as características de cada máquina), evitando que existam processadores ociosos enquanto outros estão sobrecarregados. A distribuição das cargas de trabalho é essencial para o bom desempenho do sistema (BRANCO,2004).

Uma vez que o balanceamento de cargas é objetivo importante para o escalonamento de processos, principalmente em ambientes heterogêneos, e que para se conseguir alcançar esse escalonamento fazem-se necessários índices de carga.

O próximo capítulo tem como objetivo apresentar os conceitos básicos sobre índices de carga e desempenho.

CAPÍTULO 3. ÍNDICES DE CARGAS E DESEMPENHO

Para que um sistema obtenha um bom desempenho é necessário aproveitar os recursos nele disponíveis de maneira correta. Em se tratando de um sistema distribuído isso se torna ainda mais importante porque trata do desempenho global obtido e da heterogeneidade presente no mesmo (BRANCO, 2004).

Um bom algoritmo de escalonamento tem que utilizar os recursos da melhor maneira possível, distribuindo a carga de trabalho entre os processadores de maneira equilibrada de acordo com a capacidade de cada um deles. Desse modo o desempenho do sistema inteiro poderá ser melhorado.

Para que esse desempenho seja obtido é necessário introduzir índices de cargas para poder analisar a quantidade de trabalho submetida a um determinado elemento de processamento para descobrir se o mesmo se encontra ocioso, sobrecarregado ou com uma carga normal de trabalho (MARQUES, 2007).

Outra métrica que deve ser levada em consideração é o índice de desempenho, que muitas vezes é confundido com o índice de cargas. Apesar de estarem diretamente ligados o índice de desempenho analisa tanto a heterogeneidade configuracional quanto arquitetural da plataforma (BRANCO, 2004).

3.1. Avaliação de Desempenho

Para avaliar um sistema devem ser levados em consideração as características, o escalonamento e as métricas utilizadas. Uma escolha adequada das métricas é muito importante para a avaliação do mesmo.

A definição de uma métrica é tão importante quanto as medidas que ela traz. Uma métrica mal definida pode atrapalhar a avaliação do desempenho do sistema. Essa definição tem que ser feita da maneira mais clara possível, evitando ambigüidades (JAIN, 1991).

Existem dois tipos de técnicas para avaliação de desempenho, são eles: técnicas de modelagem e técnicas de aferição (ORLANDI, 1995).

As técnicas de modelagem são geralmente utilizadas quando não se pode interferir no sistema ou quando o sistema ainda não foi desenvolvido. Estas técnicas baseiam-se na representação do sistema computacional em forma de modelos, onde os mesmos 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 a avaliação de desempenho do mesmo. (ORLANDI, 1995 apud MARQUES, 2007).

Já as técnicas de aferição são utilizadas em sistemas reais, tanto nos completos quanto nos protótipos. Apesar de serem mais precisas essas técnicas disputam os recursos do sistema, podendo atrasá-lo. As técnicas mais utilizadas são (ORLANDI,1995):

- construção de protótipo: utilizada em sistemas reais ou que serão desenvolvidos. Um protótipo é considerado um sistema simplificado, com as mesmas características e funções do sistema original. Essa técnica apresenta um maior custo;

- 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;
- *benchmarking:* baseia-se em comparações obtidas por meio de cargas de trabalhos. O mesmo impõe uma carga de trabalho à política e a gerencia, no final deste processo é possível saber qual política obteve melhor desempenho sobre determinada carga de trabalho.

3.2. Cargas de Trabalho

Cargas de trabalho (*workload*) são as tarefas alocadas a cada processador (KUNZ, 1991). Estas devem ser distribuídas por meio do escalonador de uma maneira equilibrada, não deixando que alguns processadores se tornem ociosos enquanto outros estejam sobrecarregados. Caso alguma máquina fique sobrecarregada sua carga tem que ser redistribuída para outras máquinas, dando prioridade as mais ociosas.

Essas cargas se dividem em duas categorias: real e sintéticas.

A carga de trabalho real é a gerada pelo sistema, essa carga não pode ser repetida, ou seja, não é freqüentemente usada para a caracterização da carga (KUNZ, 1991 apud BRANCO, 2004).

Diferentemente das cargas reais, as cargas de trabalho sintéticas podem ser repetidas e de maneira controlada. Ela é uma representação da carga real, mas tem como vantagens a facilidade para ser modificada, transferida e reproduzida, trabalhando assim de uma maneira mais flexível do que as reais (KUNZ, 1991 apud BRANCO, 2004).

Outra vantagem muito importante das cargas sintéticas é que elas permitem que sejam elaboradas para um experimento específico. Cada sistema tem uma característica diferente que deve ser levada em consideração. Como exemplo pode-se citar uma situação em que o processador está sobrecarregado e o acesso a disco não. Dessa forma a carga de trabalho do processador é considerada alta enquanto a do acesso ao disco é considerada baixa. Isso mostra que são necessárias cargas especificas para cada tipo de recurso (BRANCO, 2004).

Os recursos que são analisados com maior freqüência na literatura são: CPU, memória, disco e rede de intercomunicação (BRANCO, 2004).

3.3. Índices de Cargas

Os índices de cargas são métricas utilizadas para avaliar o seu desempenho, através desses índices pode-se avaliar se um recurso encontra-se ocioso, moderado ou sobrecarregado.

Esse índice é composto por uma variável numérica, inteira e não negativa. Quando é atingido o valor nulo, o recurso é considerado ocioso e conforme é aumentada a carga o valor desse índice é incrementado. (FERRARI & ZHOU, 1987; KUNZ, 1991; BRANCO, 2004).

A qualidade dos índices está diretamente ligada ao desempenho do escalonador de processos, pois bons índices irão contribuir para um bom escalonamento. Portanto, deve-se ter cuidado ao escolher a forma de coleta e utilização de informações assim como a periodicidade em que elas são coletadas.

Uma vez que as cargas alteram-se uma maneira muito rápida, tornam as informações que foram coletadas antes da mudança obsoletas. Então outro fator a ser levado em conta é o intervalo de tempo entre uma coleta de informações dessas informações.

Uma outra cautela que deve ser tomada em relação aos índices de carga ocorre em sistemas paralelos/distribuídos. Se as informações forem coletadas com uma freqüência muito alta, acarretará em uma sobrecarga na rede de intercomunicação, fazendo assim que o desempenho do sistema diminua. Mas se essas informações forem pouco atualizadas os índices ficarão desatualizados, fazendo com que o balanceamento de cargas seja realizado de maneira incorreta, prejudicando assim o desempenho do sistema.

Os índices de cargas se baseiam em informações atuais e/ou passadas para tentar prever o comportamento futuro dessas cargas e passam essas informações para o escalonador podendo assim realizar o balanceamento de cargas.

Os tipos de índices que são atualmente utilizados e estudados são compostos por: tamanho da fila de CPU, média do tamanho da fila de CPU em um intervalo de tempo determinado, utilização de CPU, taxa de chamada do sistema, quantidade de memória disponível, tempo de resposta e funções hibridas que utilizam junções e combinações dos índices citados (FERRARI; ZHOU, 1987; KUNZ, 1991; XU; LAU, 1997; DANTAS & ZALUSKA, 1998; BRANCO, 2004).

Esses índices podem ser divididos em grupos, são eles: baseados na fila de espera do recurso, utilização do recurso e tempo de execução/resposta (BRANCO, 2004).

Um índice de carga pode gerar cargas no sistema, trazendo assim valores imprecisos. São necessários índices que geram um mínimo de sobrecarga no sistema e para isso é necessário um conhecimento prévio do sistema para utilizar um índice adequado ao

mesmo. Pode-se dividir esses índices em duas categorias: índices genéricos e índices específicos.

Os índices genéricos são utilizados quando não se tem conhecimento da utilização do sistema nem dos objetivos do escalonador, sendo compostos pela união de um ou mais valores obtidos.

Já os índices específicos são compostos por apenas um valor obtido e tendem a gerar uma menor sobrecarga. Como o próprio nome sugere, são utilizados em casos específicos.

Na literatura, geralmente, são abordados índices de cargas em sistemas que possuem homogeneidade tanto na arquitetura quanto nas configurações. São encontradas poucas referencias sobre utilização de índices de cargas em sistemas com heterogeneidade arquitetural e configuracional ou totalmente heterogêneos.

Sistemas completamente heterogêneos necessitam de uma atenção especial. Um dos índices muito usado em sistemas homogêneos é o comprimento da fila de CPU. Já em sistemas heterogêneos esse índice não funciona corretamente. Tome-se por exemplo um sistema composto por duas máquinas heterogêneas. Na primeira delas são encontrados dois processos que exigem baixa ocupação na fila de processos, já na segunda máquina se encontra um processo que utiliza 99% da CPU. Se for levado em conta o comprimento da fila de CPU a primeira máquina receberá um índice 2 enquanto a segunda receberá um índice 1. Esses índices farão que o escalonador atribua mais processos a segunda máquina, fazendo que ela fique mais sobrecarregada enquanto a primeira está semi-ociosa. Dessa forma o desempenho do sistema cai drasticamente e nos mostra que a escolha do índice de carga é altamente importante (BRANCO, 2004).

3.3.1. Estado da Arte

Na literatura são observados alguns modelos para cálculo de índices de carga tanto para os modelos completamente homogêneos quanto para os completamente heterogêneos. A maioria dos modelos é referente a sistemas que possuem arquiteturas e configurações homogêneas, apenas uma pequena parte trata da heterogeneidade dos sistemas.

Dentre esses modelos alguns podem ser citados (BRANCO & ORDONEZ, 2007):

(FERRARI & ZHOU, 1987) propõem a utilização de um índice de carga obtido através da combinação linear do tempo de serviço $\mathbf{s_j}$ requerido por uma tarefa para execução em um determinado recurso $\mathbf{r_j}$, sendo $\mathbf{q_j}$ o comprimento da fila de $\mathbf{r_j}$, de tal modo que o índice de carga (li = *load index*) seja obtido através de:

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

Equação 1

Onde: **N** o número de recursos que possuem filas. Esse modelo foi proposto para máquinas com arquitetura e configuração homogêneas.

(KUNZ,1991) em seu estudo foi apresentado que qualquer escalonamento utilizando índices de carga é muito melhor do que um escalonamento que não efetua o balanceamento de carga. Foi proposta também a utilização de índices agregados ao invés dos índices lineares. Obteve-se como resultado que os índices agregados não apresentavam melhorias no sistema se comparados aos índices lineares, muito pelo contrário, sobrecarregavam o sistema na obtenção dos vários índices lineares que compunham os índices agregados. Esses estudos também foram realizados em modelos homogêneos.

(ZHOU et. al., 1993) propõe que os índices de carga variem de acordo com a natureza do recurso que se está avaliando, ou seja, um índice para cada tipo de recurso. Os índices utilizados são obtidos através da média do comprimento de fila de 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,2004) propõe um índice de desempenho que leva em consideração a heterogeneidade do sistema e tem como base o índice de carga e uma métrica Euclidiana. Os resultados obtidos mostraram que os ambientes paralelos/distribuídos que utilizaram esse índice de desempenho obtiveram um desempenho melhor do que os que não usaram.

3.4. Índice de Desempenho

Em (BRANCO,2004), é proposto um índice de desempenho que leva em consideração a heterogeneidade arquitetural, configuracional e temporal das máquinas que compõem sistema, ocupando assim uma lacuna existente na literatura que aborda sistemas homogêneos ou com somente a arquitetura ou configuração heterogênea. A Figura 14 mostra essa lacuna mais detalhadamente.

Figura 13. Espaço ocupado pelo índice de desempenho na literatura (BRANCO, 2004)

Um índice de desempenho mostra a capacidade de trabalho de um recurso, e um índice de desempenho, assim como um índice de carga, tem que prever os valores futuros utilizando valores atuais e passados, ou seja, bons índices de desempenho são baseados em índices de cargas (BRANCO, 2004).

O índice de desempenho proposto por (BRANCO, 2004) é baseado na distância euclidiana entre a máquina ociosa (ponto de origem) 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 que possuir a menor distância euclidiana é considerada a mais adequada para receber aquela aplicação.

Com base nos recursos básicos que serão analisados (cpu, disco, memória e rede), o índice de desempenho pode ser calculado como mostrado na equação a seguir.

$$ID = f(I_{CPU}, I_{Memoria}, I_{Disco}, I_{Rede})$$

Equação 2

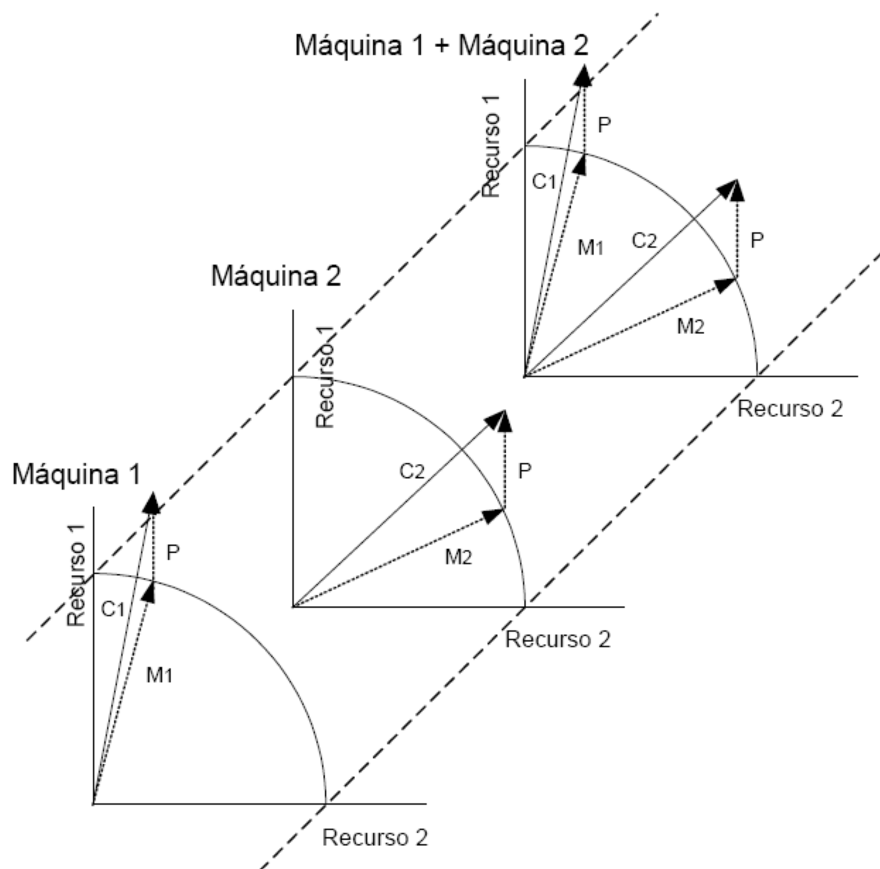

Figura 14. Exemplo com duas máquinas e dois recursos (BRANCO, 2004)

Na Figura 14 é mostrado uma situação de um sistema com duas máquinas. Na primeira (M1) o recurso 1 está sendo mais utilizado do que o recurso 2. Já na segunda máquina (M2) ocorre o inverso, o recurso 2 está sendo mais utilizado do que o recurso 1. O processo P pode ser alocado em qualquer uma das máquinas. Para escolher em qual máquina ele será alocado é utilizado o vetor resultante de cada máquina. O vetor resultante que apresentar um menor comprimento indicará a máquina em que o processo P será alocado. O vetor C1 é resultante da Máquina 1 com a alocação de P, enquanto C2 é o vetor resultante da Máquina 2 com a alocação do mesmo processo. Neste exemplo o menor vetor resultante é o da máquina 2 (BRANCO, 2004).

Dessa forma, uma métrica que pode ser adotada para o cálculo do índice de Desempenho é a distância Euclidiana entre o ponto e a origem, ou seja, o comprimento do

vetor resultante da origem até o ponto (BRANCO,2004). Então, a Equação 2 pode ser reescrita como mostrado na Equação 3.

$$ID = \sqrt{I_{CPU}^2 + I_{Rede}^2 + I_{Memória}^2 + I_{Disco}^2}$$

Equação 3

3.5. Considerações Finais

Um bom desempenho do sistema é essencial para qualquer aplicação. Para que esse desempenho seja alcançado, é necessário considerar uma série de fatores como escalonamento dos processos, cargas de trabalho de cada máquina, os índices de cargas utilizados, maneira de se calcular e obter as informações necessárias para esses índices, configuração e arquitetura das máquinas, e entre outros.

Os índices de carga são fatores muito importantes para esse desempenho esperado. É requerida uma atenção especial para as formas de coleta de informações, a periodicidade em que elas são coletadas, o modo que elas são coletadas. Neste trabalho será proposto o uso de agentes móveis como uma forma de prover esses índices.

No capítulo a seguir são apresentados conceitos sobre código móvel, agentes e agentes móveis.

CAPÍTULO 4. CÓDIGO MÓVEL

Como discutido nos capítulos anteriores, os sistemas paralelos/distribuídos se tornaram um grande alvo de pesquisas. Uma abordagem interessante para trabalhar com esses sistemas é a utilização de código móvel.

Apesar de não haver um consenso na literatura sobre código móvel, é possível defini-lo como a capacidade de mudar dinamicamente as ligações entre os fragmentos de códigos e seu lugar de execução. A habilidade de realocar código é um conceito poderoso que oferece uma série de opções para pesquisa e desenvolvimento. Apesar do grande interesse e vantagens do código móvel, sua utilização ainda está em fase de crescimento (FUGGETTA; PICCO; VIGNA, 1998).

O conceito de código móvel não é novo. Diversos mecanismos e facilidades foram implementados para migrar código entre os nós da rede. Uma área de grande interesse em pesquisas foi a utilização de código móvel para fazer a migração de processos entre os nós de um sistema distribuído, tornando possível a migração de um processo para outra máquina enquanto este estava sendo executado. Essa migração era feita de modo transparente ao usuário, não permitindo a ele um controle nem uma visibilidade dessa migração (FUGGETTA; PICCO; VIGNA, 1998).

Em (FUGGETTA; PICCO; VIGNA, 1998) foi proposta uma classificação dos mecanismos móveis de acordo com a sua mobilidade. Na figura 15 é retratada essa classificação.

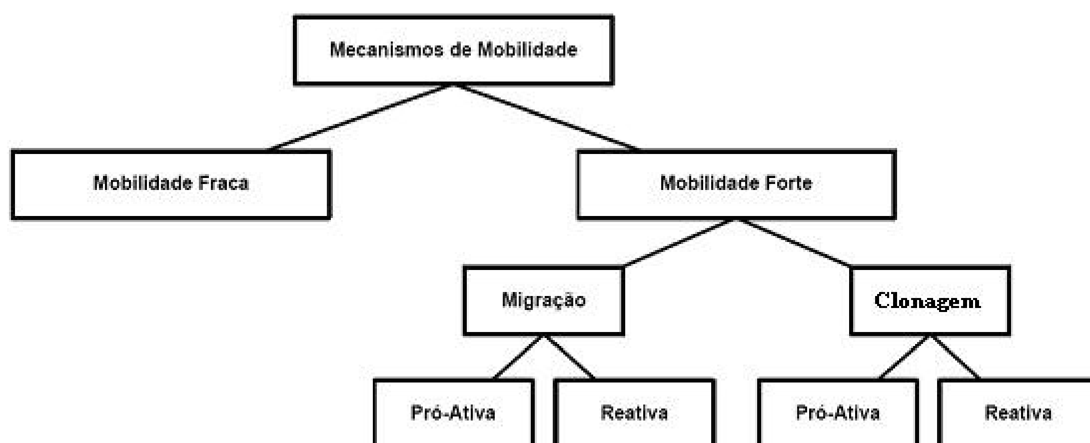

Figura 15. Mecanismos de Mobilidade (FUGGETTA; PICCO; VIGNA, 1998)

Em mecanismos que apresentam mobilidade fraca (*weak mobility*) o código pode ser transferido entre ambientes computacionais sendo ligado dinamicamente em uma unidade de execução ou seu código é utilizado em uma nova unidade de execução. Esse código pode carregar consigo apenas alguns dados para iniciar-se, sem levar o estado de execução.

Já em mecanismos com mobilidade forte (*strong mobility*) tanto o código quanto o estado de execução são transferidos de um ambiente computacional para uma unidade de execução.

A mobilidade forte é dividida em dois seguimentos: migração e clonagem. Na migração a unidade de execução é suspensa, transferida para outro ambiente computacional e então retoma sua execução do ponto em que parou. Na clonagem é feita uma cópia da unidade de execução em outro ambiente computacional. Ela difere do processo de migração porque a unidade de execução original não é desvinculada do ambiente computacional original. Ambas as técnicas se dividem em:

- reativas: a migração é invocada por outra unidade de execução;

- pró-ativas: a unidade de execução decide de maneira autônoma tanto o momento quanto o destino da migração.

Os paradigmas de código móvel se baseiam em diferentes fatores ao antes e depois da execução do código, estabelecendo os elementos que compõem o sistema e a interação entre eles. Os principais paradigmas são: Cliente/Servidor, execução remota, código sob demanda e agentes móveis. Na Tabela 1 é mostrado um cenário onde um componente computacional *A*, localizado em *LA*, que necessita do resultado da execução de uma tarefa, e assumisse um componente computacional *B*, localizado em *LB*, que estará envolvido na realização do serviço. Para cada paradigma é mostrada a localização dos componentes antes e depois da execução do serviço. Os recursos movidos se encontram em itálico (FUGGETTA; PICCO; VIGNA, 1998).

Tabela 1. Paradigmas de Código Móvel

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

No paradigma Cliente/Servidor, tanto as instruções quanto os recursos estão localizados em *B* (servidor), sendo que este está em *LB*. O cliente componente *A*, que está

localizado em *LA* requer um serviço de *B*. Este recebe a requisição e a executa utilizando as instruções e os recursos, podendo assim mandar uma resposta ao *A*.

Na execução remota *A* possui as instruções, mas os recursos estão localizados em *LB*. Então *A* envia suas instruções para que *B* possa executá-las e enviar uma resposta para *A*.

No código sob demanda, *A* possui acesso aos recursos necessários para a execução os quais se encontram em *LA*, entretanto não existem instruções em *LA* para a execução da tarefa. Sendo assim, *A* interage com *B* para que este lhe mande as instruções necessárias que se encontram em *LB* para que assim possa realizar a execução.

O agente móvel situado em *A* possui as instruções necessárias em *LA*, mas alguns recursos necessários se encontram em *LB*. Então o agente móvel migra de *A* para *B*, levando consigo não somente as instruções, mas também alguns resultados intermediários. Chegando em *B* o agente pode completar a execução da tarefa.

O paradigma de agente móvel se difere dos outros paradigmas de código móvel, pois não leva consigo somente o código para o outro componente, ele leva também o seu estado de execução e alguns recursos que ele já dispõe.

4.1. Agentes

O conceito de agente é proposto por vários autores, mas cada um o define de acordo com a área de pesquisa que ele atua, sendo assim difícil de criar um conceito propriamente dito. Em (FRANKLIN; GRAESSER, 1996) é proposta uma definição, a partir da definição de vários autores, que se mostra mais 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.

Os agentes podem ser classificados de acordo com o subconjunto de características que possuem. Na Tabela 2 estão classificadas as principais propriedades.

Tabela 2. Propriedades dos Agentes (FRANKLIN; GRAESSER, 1996)

Propriedade	Significado
Reativo	Responde as mudanças do seu ambiente.
Autônomo	Exerce controle sobre suas próprias ações.
Pró-Ativo	Orientado a metas. Não age somente em resposta ao ambiente
Contínuo	Executa continuamente.
Comunicativo	Pode se comunicar com outros agentes ou com pessoas.
Inteligente	Muda seu comportamento de acordo com suas experiências anteriores.
Móvel	Pode mudar de uma máquina para outra.
Flexível	Ações não definidas através de *scripts*.

Pela definição, todo agente possui pelo menos as 4 primeiras propriedades.Os agentes podem ser classificados de outras maneiras, alguns exemplos de classificação são: tarefas realizadas, arquitetura de controle, grau de sensibilidade, ambiente etc.

4.1.1. Agentes Móveis

Um agente móvel herda as características de um agente comum. O que o difere dos demais é a sua habilidade de migrar entre os *host* (ou elementos de processamento) de uma rede, levando com ele o seu código e o seu estado de execução, o que permite que ele retome suas atividades.

O uso de agentes móveis em um sistema distribuído traz algumas vantagens que devem ser consideradas, essas vantagens estão definidas a seguir. (LANGE; OSHIMA, 1998).

- **Redução do tráfego de rede**: Os sistemas distribuídos exigem um grande volume de comunicação para realizar uma tarefa, ainda mais quando utilizados mecanismos de segurança. Os agentes móveis reduzem esse tráfego, pois permitem o empacotamento das instruções e a migração para o *host* onde se encontram os dados, realizando as instruções localmente. Utilizando o seguinte princípio: mover o processamento para os dados ao invés de mover os dados até o processamento.
- **Superação da latência da rede**: Os sistemas críticos necessitam de respostas em tempo real para lidar com as mudanças no ambiente. O controle desses sistemas gera uma latência alta na rede, o que na maioria dos casos é inaceitável. O uso dos agentes móveis reduz essa latência, pois podem ser despachados pelo controlador central para realizar suas tarefas localmente.
- **Encapsulamento de protocolos:** Cada máquina possui seu próprio código para implementar a transferência de dados. Novos requerimentos de segurança e eficiência podem exigir uma mudança no protocolo, o que pode causar problemas. Os agentes móveis podem trafegar entre os *hosts* afim de estabelecer um canal de comunicação utilizando um protocolo próprio.
- **Execução assíncrona e autônoma**: Os agentes móveis podem executar de maneira assíncrona e autônoma, sem depender da aplicação que o criou, sem necessidade de manter uma conexão aberta entre os *hosts*.

- **Adaptação dinâmica**: Agentes móveis possuem a habilidade de perceber mudanças no ambiente de execução e reagir de forma autônoma.
- **Independência de plataforma**: Grande parte das redes de computadores são heterogêneas, tanto na parte de hardware quanto de software. Os agentes móveis são independentes da máquina e da rede, sendo somente dependentes do seu ambiente de execução. Sendo assim uma ótima alternativa para integrar sistemas heterogêneos.
- **Robustez e tolerância à falhas:** A habilidade dos agentes móveis de reagir dinamicamente às situações e eventos desfavoráveis facilita a construção de um sistema distribuído robusto e tolerante a falhas. Por exemplo: se uma máquina precisa ser desligada, os agentes nessa máquina podem ser avisados, em tempo hábil, para migrarem para outra máquina da rede 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.

4.1.1.1. *Aglets*

A palavra *Aglet* vem da junção das palavras *agent* com a palavra *applet.* Criado pela IBM, é uma forma de se trabalhar com agentes móveis de uma forma semelhante ao uso dos *applets* em Java. Por conseqüência herda as vantagens da linguagem Java que são

muito úteis para desenvolver agentes móveis, algumas delas estão listadas a seguir. (LANGE; OSHIMA, 1998).

- **Independência de plataforma**: A linguagem Java foi designada para operar em redes heterogêneas. Somente é necessária a presença do *Java Runtime System* para a execução de qualquer aplicação Java.
- **Execução Segura**: Como é uma linguagem para uso de Internet, a demanda por segurança influenciou o projeto de várias maneiras. Por exemplo o bloqueio de informações privadas para programas que não possuem acesso a elas, garantindo que agentes não-confiáveis obtenham acesso a informações privadas dos *hosts*.
- **Carregamento de classes dinamicamente**: Permite que classes sejam carregadas e definidas pela *Java Virtual Machine* em tempo de execução. Provendo um espaço para os agentes, além de permitir que eles sejam executados de maneira independente e segura dos outros agentes.
- **Programação *Multithreading*:** O uso de *threads* permite que os agentes sejam executados em sua própria *thread* e isso é uma forma de permitir que os agentes sejam autônomos.
- **Serialização de objetos:** Uma das características chaves de agentes móveis é que podem ser serializados e reconstruídos. A linguagem Java fornece um mecanismo de serialização que faz esse processo de forma eficiente.

A biblioteca de classes *Aglets* foi concebida com os seguintes objetivos (OYAMADA; ITO, 1998):

- Fornecer um modelo compreensivo e simples de programação utilizando agentes móveis, sem implicar em modificações na máquina virtual Java ou em código nativo;
- Disponibilizar mecanismos de comunicação poderosos e dinâmicos que permitissem os agentes se comunicarem uns com os outros, fossem eles conhecidos ou não;
- Projetar uma arquitetura de agentes móveis que permitisse extensibilidade e reusabilidade;
- Obter uma arquitetura altamente coerente com o modelo tecnológico Web/Java.

4.1.1.1.1 Elementos Básicos

A biblioteca *Aglets* além de herdar as características da linguagem *Java* possui alguns elementos básicos na sua composição. É por meio desses elementos que é possível suprir algumas das deficiências da linguagem Java. Os elementos básicos da biblioteca *Aglets* estão listados a seguir (LANGE & OSHIMA, 1998):

- ***Aglet*:** um *aglet* é um objeto *Java* móvel que pode visitar outros *hosts*. Ele é um agente autônomo, pois executa em sua própria *thread* quando chega a um *host* e reativo pela sua capacidade de responder mensagens externas.
- ***Proxy:*** Um *Proxy* é uma representação de um *aglet*. Ele funciona como uma espécie de escudo, protegendo o acesso direto aos métodos públicos do *aglet*. Além de prover uma transparência ao *aglet* escondendo sua real localização.

- **Contexto (*Context*):** O contexto é um objeto estacionário que prove um ambiente para manutenção e gerenciamento dos *Aglets* em estado de execução. Cada nó de uma rede pode apresentar vários servidores e todos os servidores podem conter vários contextos.
- **Mensagem (*Message*):** Uma mensagem é um objeto trocado entre um par de *Aglets*. A troca de mensagens pode ser feita de um modo síncrono ou de um modo assíncrono. Essas mensagens podem ser usadas para colaboração entre os *Aglets* ou para trocarem informações.
- ***Future reply*:** É um manipulador usado para tratar resultados de envio assíncrono de mensagens.
- **Identificador (*Identifier*):** É uma identificação única que cada *aglet* possui. Ele recebe essa identificação no momento da sua criação e não muda durante toda a sua vida.

O comportamento de um objeto *aglet* é baseado na sua "vida e morte". A biblioteca *Aglets* possui algumas operações fundamentais que são usadas para criação e gerenciamento dos agentes móveis (LANGE & OSHIMA, 1998). Essas operações são:

- ***Creation* (Criação):** é a criação de um novo *aglet*. O novo *aglet* recebe a sua identificação, é inserido em um contexto e é iniciado. Assim que iniciado com sucesso, o agente começa a executar.
- ***Cloning* (Clonagem):** a clonagem de um *aglet* faz uma cópia quase idêntica do agente original. As únicas diferenças são o seu identificador e o fato do novo agente reiniciar sua execução. A *thread* do *aglet* original não é clonada.

- ***Dispatching* (Despacho)**: quando um *aglet* é despachado ele é removido do contexto atual e inserido no contexto destino.
- ***Retraction* (Retração):** o *aglet* é "puchado" (removido) do contexto que se encontra para o contexto que a retração foi requisitada.
- ***Activation / Deactivation* (Ativação / Desativação):** A desativação de um *aglet* consiste em parar sua execução e salvá-la. A ativação retorna o *aglet* no contexto.
- ***Disposal* (Eliminação):** A execução do agente é interrompida e depois ele é removido do contexto.
- ***Messaging(*Troca de Mensagens):** A troca de mensagens entre *Aglets* pode ser feita tanto de maneira síncrona quanto assíncrona.

A figura a seguir exemplifica as principais funções providas pela biblioteca Aglets.

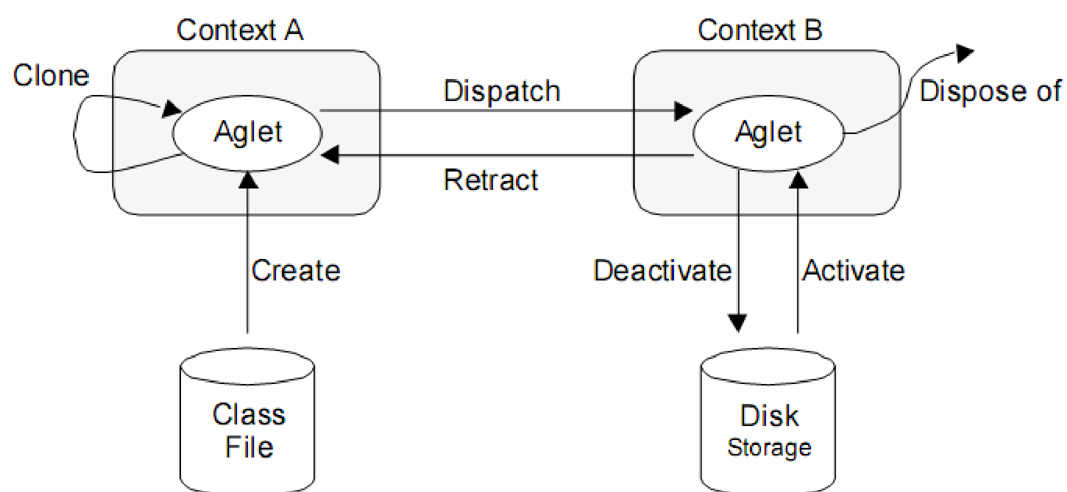

Figura 16. Funções Principais da Biblioteca *Aglets* (LANGE & OSHIMA, 1998).

4.1.1.1.2 Modelo de Eventos de Aglets

O modelo de programação em *Aglet* é baseado em eventos. São definidos eventos que ocorrem durante a vida do agente. Para reconhecer esses eventos são utilizados *listeners* específicos para cada tipo de evento. Esses *listeners* captam o evento e proporcionam ao programador uma possibilidade de ação durante esses eventos. (LANGE & OSHIMA, 1998).

Na plataforma *Aglet* existem três tipos *listeners* que permitem esse tipo de tratamento, são eles:

- ***Clone Listener:*** Capta eventos de clonagem de agentes. Permite que se façam ações especificas antes da clonagem, no momento que o clone é criado e depois da criação do clone.
- ***Mobility Listener:*** Esse *listener* capta eventos de mobilidade. As ações podem ser tomadas quando o agente é despachado para outro contexto, quando ele é retraído de um outro contexto e quando chega de um novo contexto.
- ***Persistence Listener:*** Utilizado quando um *aglet* é desativado ou ativado novamente.

É demonstrado na Tabela 3 os tipos de eventos e seus respectivos *listeners*.

Tabela 3. Eventos e *Listeners* (LANGE & OSHIMA, 1998).

Quando	*Event Object*	*Listener*	*Method called*
Antes da clonagem.	*CloneEvent*	*CloneListener*	*onCloning*
Quando o clone é criado.	*CloneEvent*	*CloneListener*	*onClone*
Depois da criação do clone.	*CloneEvent*	*CloneListener*	*onCloned*
Antes de ser despachado.	*MobilityEvent*	*MobilityListener*	*onDispatching*
Antes de ser retraído.	*MobilityEvent*	*MobilityListener*	*onReverting*
Depois de chegar no destino.	*MobilityEvent*	*MobilityListener*	*onArrival*
Antes da desativação.	*PersistencyEvent*	*PersistencyListener*	*onDeactivating*
Depois da ativação.	*PersistencyEvent*	*PersistencyListener*	*onActivation*

4.2. Considerações Finais

Os agentes móveis são uma forma eficiente de comunicação entre computadores ligados em uma rede, trazendo alguns benefícios relevantes ao sistema.

Como mencionado nos capítulos anteriores, os sistemas distribuídos provêem um melhor desempenho em relação aos sistemas centralizados, sendo que um das principais razões dessa melhoria se deve ao melhor aproveitamento dos recursos disponíveis. Com a utilização dos agentes móveis nesse tipo de sistema na coleta de índices de cargas e de desempenho, devem ser consideradas as vantagens trazidas pelos agentes, melhorando ainda mais o desempenho do sistema, sendo esse o objetivo principal de um sistema distribuído.

O Capitulo seguinte contém a confecção de um agente móvel utilizando a biblioteca *Aglets*, sendo este destinado a coleta de índices de carga em um sistema distribuído.

CAPÍTULO 5. CONFECÇÃO DO AGENTE MÓVEL

O Capítulo anterior mostrou as características de um agente móvel e da biblioteca *Aglets*. Devida as vantagens da utilização do agente móvel em um sistema distribuído, este capítulo irá mostrar detalhadamente a criação de um agente móvel para coleta de cargas em um sistema distribuído a fim de realizar o balanceamento de cargas em todo o sistema.

Para a criação do agente móvel foi utilizada a biblioteca *Aglets* mostrada no capítulo anterior.

O trabalho do agente móvel desenvolvido neste trabalho consiste em trafegar entre as máquinas do sistema coletando os índices de desempenho de cada uma das máquinas e, após terminar a coleta de todas os índices, informar uma lista ordenada com os *hosts* que obtiveram os melhores índices para o escalonador de processos. Para obter esse índice, é necessário o uso de um agente monitor que colete os índices de carga dos recursos mais relevantes para o desempenho do sistema e por meio desses índices, é possível calcular o índice de desempenho. Um modelo desse agente monitor foi proposto por (MARQUES, 2007) que será mostrado nesse mesmo Capítulo. Nas Figuras 17 e 18 é ilustrado o sistema no qual o agente móvel atua e o seu funcionamento.

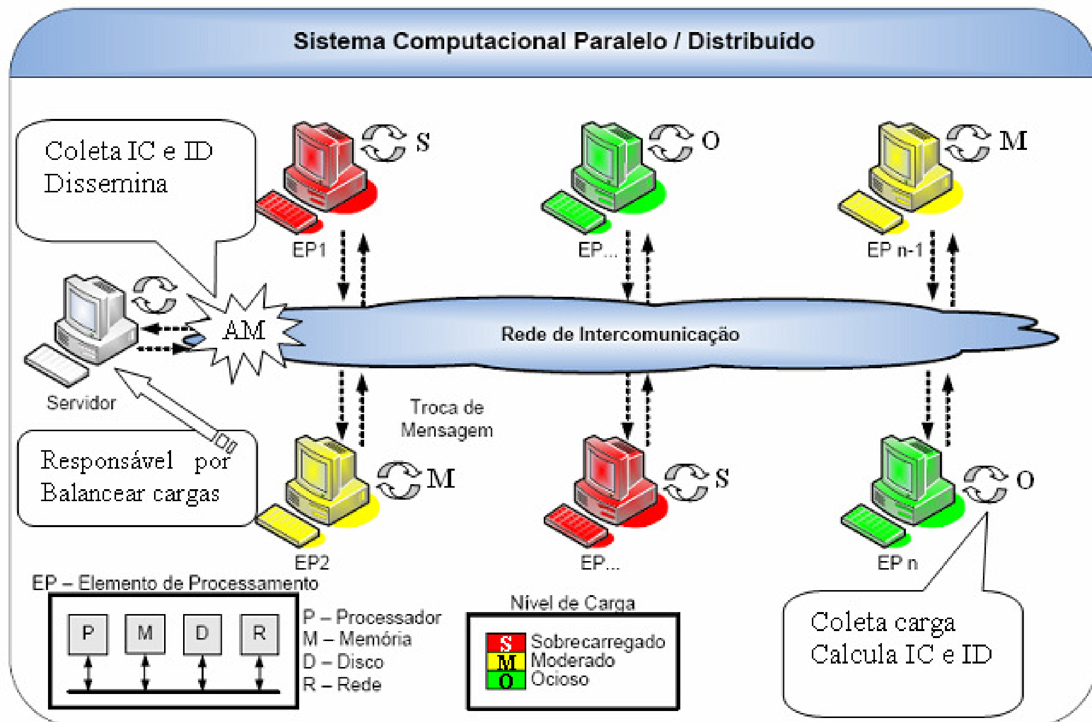

Figura 17. Sistema Paralelo/Distribuído sem balanceamento (MARQUES, 2007)

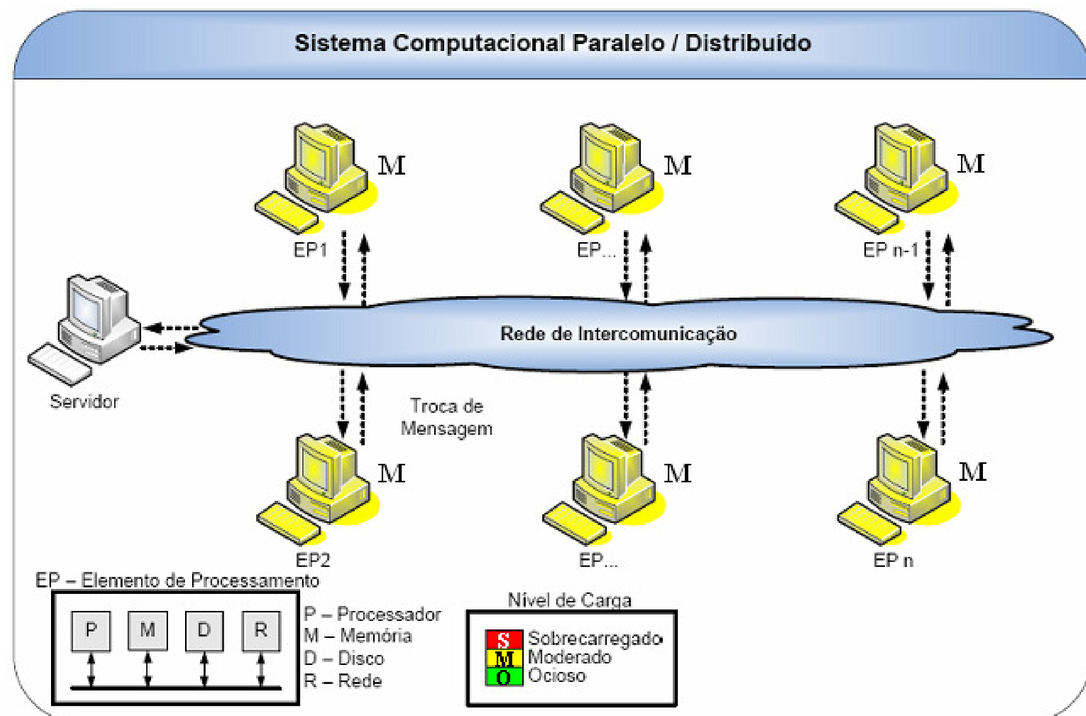

Figura 18. Ambiente Paralelo/Distribuído balanceado (MARQUES, 2007)

5.1. Tahiti Server

Para que o agente móvel consiga trafegar entre os *hosts*, é necessário que cada máquina destino esteja executando um servidor *Aglets*. Incluso na versão 2.0.2 do *Aglets* está o servidor Tahiti.

O servidor Tahiti provê um ambiente para execução dos agentes móveis, além de permitir a utilização das funcionalidades principais do *Aglets* (*create, dispatch, dispose* etc.). Além disso, possui uma interface gráfica com todas as suas opções, desde as funcionalidades principais até configurações de rede e segurança. A interface do Tahiti pode ser vista na Figura 19.

Figura 19. Tahiti Server

5.2. Agente Monitor

O trabalho do agente móvel é percorrer os *hosts* do sistema coletando o índice de desempenho de cada um, fazendo assim uma lista com as melhores máquinas aptas a receber tarefas do escalonador. Para que essa coleta seja possível é necessário que sejam calculados esses índices de desempenho em cada uma das máquinas que o agente percorrer.

Em (MARQUES, 2007) foi proposto um agente monitor chamado PRM (*Performance Resource Monitor)*, a sua funcionalidade é coletar as cargas dos recursos e calcular o índice de desempenho de cada uma das máquinas.

Em um sistema distribuído, muitos recursos podem ser levados em consideração para o calculo do índice de desempenho. O agente monitor utiliza os quatro recursos que possuem mais influência no desempenho do sistema, são eles: CPU, Memória, *I/O* de disco e Rede (MARQUES, 2007).

Para conseguir coletar a carga dos recursos é utilizado um pacote chamado *dstat*. Esse pacote é oferecido para quase todas as distribuições Linux e trata as informações contidas no diretório */proc* para obter os índices dos recursos. Na Figura 20 é mostrado o *dstat* em execução.

```
(root@gmm) dstat -c -s -m -d -n
Arquivo  Editar  Ver  Terminal  Abas  Ajuda
[root@gmm ~]# dstat -c -s -m -d -n
----total-cpu-usage---- ----swap--- ------memory-usage----- -dsk/total- -net/total-
usr sys idl wai hiq siq  used  free  used  buff  cach  free  read  writ  recv  send
 21   4  67   7   0   0     0  996M  173M   20M  215M  539M  191k   86k     0     0
 26   5  68   1   0   0     0  996M  173M   20M  215M  539M     0  428k   33k 2136B
 61   6  24   7   2   1     0  996M  173M   20M  215M  539M   64k     0   47k 3424B
 45   3  51   0   1   0     0  996M  174M   20M  215M  538M     0 8192B   46k 4293B
 74   7  17   0   2   0     0  996M  174M   20M  215M  538M     0     0   42k 1497B
 69   8  22   0   0   1     0  996M  174M   20M  215M  538M     0     0   46k 2613B
 67   9  24   0   0   0     0  996M  174M   20M  215M  538M     0     0   34k 2257B
 23   5  71   1   0   0     0  996M  174M   20M  215M  537M     0  292k   47k 3680B
 25   5  69   0   0   1     0  996M  174M   20M  215M  538M     0  404k   45k 3228B
 25   6  69   0   0   0     0  996M  174M   20M  215M  537M     0     0   43k 2990B
 24   5  70   0   0   0     0  996M  174M   20M  215M  537M     0     0   46k 1428B
 24   5  70   0   1   0     0  996M  174M   20M  215M  537M     0     0   47k 1428B
 66   8  26   0   0   0     0  996M  171M   20M  214M  542M     0  324k   46k 1776B
  5   1  94   0   0   0     0  996M  171M   20M  214M  542M     0     0   44k 2160B
 21   1  78   0   0   0     0  996M  171M   20M  214M  542M     0     0   40k 3130B
 37   3  60   0   0   0     0  996M  172M   20M  213M  542M     0     0   27k 2146B
 78   6  15   0   0   1     0  996M  173M   20M  214M  540M   36k     0   13k 5324B
```

Figura 20. Execução do dstat (MARQUES, 2007)

O agente monitor, além de coletar o uso dos recursos, tem como função o cálculo dos índices de carga de cada recurso e o índice de desempenho do sistema. Para o cálculo desses índices são utilizados *benchmarks* normalizados, podendo assim tratar a heterogeneidade do sistema, caso exista (BRANCO, 2004)(MARQUES, 2007).

Os *benchmarks* são testes padronizados que são utilizados para medir o desempenho de diferentes recursos em tipos específicos de aplicação. E o objetivo de normalizar os valores obtidos nos mesmos referente a cada recurso analisado é a mantê-los no intervalo de 0 a 1, sendo os valores mais próximos de 1 representam os melhores recursos, e os valores mais próximos de 0 representam os recursos com menor poder computacional. Tomando por base a avaliação feita pelo *benchmark* normalizado, foi possível classificar as máquinas que compõem o sistema paralelo distribuído heterogêneo com valores variando em 0 e 1.(BRANCO, 2004)(MARQUES, 2007)

Portanto, o cálculo do índice de carga de cada recurso pode ser obtido com base na Equação 3.

$$\text{Índice de Carga} = \text{Carga do Recurso x Benchmark Normalizado}$$

Equação 4

Após obter o índice de carga de todos os recursos que serão considerados, o agente móvel calcula o índice de desempenho da máquina como proposto em (BRANCO, 2004). Após o calculo do índice, ele gera um arquivo contendo o índice de cada máquina.

5.3. Desenvolvimento do Agente Móvel

O primeiro passo na construção do agente é a criação da sua classe herdando as características da classe Aglet. O seguinte código mostra a criação da classe.

```
import java.net.URL;
import java.util.Scanner;
import java.io.*;
import com.ibm.aglet.*;
import com.ibm.aglet.event.*;

public class MobileAgent extends Aglet {
     int num_host=0;
     int step = 0,aux=0;
     int lastHost = 0;
           double media=0;
     URL nexthost;
    long tempoE1, tempoE2, tempoER, tempoP1, tempoPR;

     String nome="",carga="",host="";
}
```

O agente móvel desenvolvido nesse trabalho tem a necessidade de trafegar entre os *hosts* para poder obter os índices de desempenho. Esses índices são calculados pelo Agente Monitor (MARQUES, 2007) através dos índices de carga de cada recurso coletados pelo mesmo. Sendo assim a primeira coisa a ser definida é a rota a ser seguida pelo Agente Móvel. Para definir esse caminho é utilizado o arquivo /tmp/hosts.txt, nesse arquivo é inserido os *hosts* na sua devida ordem.

Os *hosts* devem seguir o seguinte padrão: protocolo utilizado, nome do *host* e porta utilizada. O primeiro espaço é reservado para o Mestre, que é onde o agente é iniciado e onde ele grava as informações coletadas das outras máquinas durante o seu percurso. O seguinte código mostra um exemplo de arquivo de *hosts*:

atp://lab1101:4434

atp://lab1102:4434

atp://lab1103:4434

atp://lab1104:4434

Esse exemplo mostra um arquivo com quatro *hosts* destino, sendo o primeiro o lugar de onde o agente será iniciado. O protocolo utilizado como padrão no *Aglets* é o ATP (*Agent Transfer Protocol)* (LANGE;OSHIMA,1998) e a porta padrão é a 4434.

A leitura desse arquivo é feita logo na criação do agente, e para isso a biblioteca do *Aglets* possui um método chamado onCreation. Esse método é invocado logo que é dado o comando Create no servidor Tahiti. Esse método possibilita a leitura do arquivo de *hosts* somente na criação do agente. O código a seguir mostra o método onCreation.

```
public void onCreation (Object o){

       Scanner sx;
       try{
               File arq = new File( "/tmp/hosts.txt");
               FileReader reader = new FileReader(arq);
               BufferedReader buff = new
BufferedReader(reader);
               sx = new Scanner (buff);
               while (sx.hasNext()){
                         host+=sx.next()+" ";
                         num_host++;
                    }
                    System.out.println(num_host);
               }
               catch(Exception e){
                    System.out.println("erro host");
               }

     addMobilityListener(
                    new MobilityAdapter(){

public void onDispatching(MobilityEvent e){  }
```

```
     public void onArrival(MobilityEvent e){ }
                    }
          );
     }
```

Para a leitura do arquivo é utilizada a classe Scanner para percorrer o arquivo e salvar as informações em uma variável do tipo String.

Para que o agente móvel colete o índice de desempenho do *host* em que ele se encontra, ele utiliza uma função chamada Coleta(). Essa função retorna uma String na qual é convertida para o tipo *double* para ser usada como índice de desempenho. A chamada da função Coleta() e a sua implementação estão demonstradas no código a seguir.

```
carga+=" "+Coleta();
public String Coleta(){
          String coletaID="";
            try{
                String fileName = ("/tmp/id.txt");
                File f = new File(fileName);
                FileReader ler = new FileReader(f);
                BufferedReader buff = new
BufferedReader(ler);
                String line;
                while ((line = buff.readLine()) != null){
                       Scanner sc = new Scanner(line);
                          while (sc.hasNext()){
                             coletaID = sc.next();
                       }
                    }
                   ler.close();
                   buff.close();
            }
            catch (Exception e){System.out.println("erro
leitura");}
            return coletaID;
     }
```

Novamente é utilizada a classe Scanner para percorrer o arquivo. O Agente Monitor (MARQUES,2007) calcula o índice de desempenho do *host* através dos índices de

carga e grava o resultado no arquivo /tmp/id.txt . O agente móvel lê esse arquivo e salva o resultado em uma String, que depois será convertida em um vetor do tipo *double*.

Outro tipo de coleta feita pelo agente é o nome do *host* que ele se encontra. Como saída final do agente móvel, é gerado um arquivo com o número de *hosts* e uma lista com o nome dos *hosts* ordenada de acordo com o seu índice de desempenho. É através desse arquivo que o escalonador escolhe para qual *host* será enviada a próxima tarefa. No código a seguir é mostrada a função NomeMaquina() e a sua chamada.

```
nome+=" "+ NomeMaquina();
public String NomeMaquina(){
          String x=null;

          try{
               Scanner s = new
               Scanner(
Runtime.getRuntime().exec("hostname").getInputStream() );
                         x = s.nextLine();
          }catch(Exception e){
               System.out.println("erro nome maquina");
          }

          return x;
     }
```

Através do comando *hostname* é obtido o nome da máquina em que o agente móvel se encontra. Esse nome é armazenado em uma String onde, posteriormente, será passado para um vetor.

O agente móvel precisa trabalhar dentro de um *loop* infinito, porque sua rota funciona como uma lista circular, quando o agente chega no último *host*, ele volta ao *host* de origem e salva as informações coletadas durante o seu percurso.

Ao chegar ao *host* de origem (mestre), o agente móvel calcula o tempo que foi gasto no percurso. Além disso, ele passa as variáveis de coleta de índices de desempenho e a variável de nomes para vetores, para que seja feita a ordenação dos mesmos de acordo

com os índices de desempenho de cada máquina. Para fazer a ordenação, é utilizado o método *Bubble*.

Após a ordenação do vetor, o passo seguinte do agente móvel é gravar essas informações no arquivo /tmp/id_h.txt, localizado na máquina mestre. É através desse arquivo que o escalonador faz a escolha de envio para a próxima tarefa.

Nesse arquivo o agente móvel escreve o número de *hosts* que ele passou, e uma lista ordenada de maneira crescente dos *hosts*. Para que seja feita essa gravação, o agente móvel utiliza o método SalvaResultado(), que tem como parâmetro o vetor com os nomes das máquinas já ordenado.

Para melhorar o desempenho do sistema, uma máquina cujo índice demonstre que está sobrecarregada é retirada da lista de *hosts*, pois não é interessante para o desempenho do sistema que essa máquina receba mais tarefas.

Para isso, é calculada a média dos índices do vetor de cargas coletado pelo agente móvel. Caso o índice da máquina ultrapasse essa média em 30%, ela é retirada da lista de *hosts* passada para a máquina mestre.

Caso a máquina visitada não seja a máquina mestre, o agente móvel apenas lê a carga e o nome da máquina e passa para o próximo *host*. O código a seguir mostra o método run(), onde se encontram as funções de coleta, tempo, ordenação, salva e despacho para o *host* seguinte e mostra também o método SalvaResultado().

```
public void run(){
     Scanner sx;
     double auxVet;
        String auxVet2,teste;
        int i=0;
     for(;;){
          double vetor[] = new double[num_host];
          String hosts[] = new String[num_host];
          String hosts2[] = new String[num_host];

//********definição dos hosts*********
```

```
          sx = new Scanner(host);
          while(sx.hasNext()){
                    hosts[i]=(sx.next());
                    i++;
          }

//*************COleta da carga e do nome da maquina*******
          carga+=" "+Coleta();
          nome+=" "+ NomeMaquina();

//******** Chegada ao mestre**********
          if(lastHost==0){
          tempoE1 = System.currentTimeMillis();

          if(aux != step){ tempoPR =
System.currentTimeMillis() - tempoP1; }
//*********Ordenação do VEtor*********
          Scanner sc = new Scanner(nome);
          i=0;
          while(sc.hasNext()){
               teste=(sc.next());
               hosts2[i]=teste;
               i++;
          }
          i=0;
          sc = new Scanner(carga);
          while(sc.hasNext()){
               vetor[i]=Double.parseDouble(sc.next());
               i++;
          }
          boolean changed = true;
          try{
               while (changed) {
                      changed = false;
                    //Metodo de Ordenacao: Bubble
                      for (i = 0; i < vetor.length -1; i++) {
                           if (vetor[i] > vetor[i + 1]) {
                              auxVet = vetor[i];
                              auxVet2 = hosts2[i];
                              vetor[i] = vetor[i+1];
                              hosts2[i] = hosts2[i+1];
                              vetor[i+1] = auxVet;
                              hosts2[i+1] = auxVet2;
                              changed = true;
                           }
                       }
               }
          }
          catch(Exception e){System.out.println("erro
ordenação");}
```

```
//************FIm da ordenação*********

//*******Calculo da média dos índices*********
               for (i=0;i<vetor.length;i++){
                    media+=vetor[i];
                    media=media/num_host;
                    media+=media*0.3;

//*********Salva resultado e zera variaveis*********
                    salvaResultado(hosts2,vetor);
                    carga="";
                    nome="";
                    tempoP1 = System.currentTimeMillis();

                    aux = step;
                    step++;
                    System.out.println("  Tempo de percurso:
" + tempoPR + " milisegundos \n" + "passo: "+step );


               }

//********Proximo host**********
          if (lastHost == (num_host - 1) ) { lastHost = 0;}
                       else { lastHost++; }

          try{
               System.out.println("Proximo host... \n");
//             Thread.sleep(300);
               dispatch(new URL(hosts[lastHost]));

           }
               catch (Exception e) {
                         System.out.print(e.getMessage());
               }
     }

    }

public void salvaResultado(String[] hosts2, double[] vetor){

          try{
               BufferedWriter salva = new BufferedWriter(new
FileWriter("/tmp/id_h.txt"));
          salva.write(String.valueOf(hosts2.length)+"\n");
               for(int i=0;i<hosts2.length;i++)
                    if (vetor[i]<media)
                    salva.write(hosts2[i]+"\n");
```

```
          salva.close();
          media=0;

     } catch   (Exception e) {System.out.println("erro ao
salvar dados");}
}
```

Como mostrado no código anterior, o despacho para o *host* seguinte é feito através da função dispatch() provida pela classe Aglet. Se a máquina atual for a última do percurso, o agente móvel volta a máquina mestre, caso contrário ela apenas avança para o próximo *host*.

5.4. Problemas encontrados

Durante a fase de desenvolvimento do agente móvel, foram encontrados alguns problemas que dificultaram a implementação do agente.

O primeiro obstáculo encontrado foi a falta de material disponível sobre o projeto *Aglets*, dificultando assim tanto a parte de instalação da plataforma quanto ao aprendizado de suas principais características e funcionalidades.

Outro problema encontrado está relacionado com a permissão de leitura e escrita do agente móvel a arquivos na máquina. Como mencionado anteriormente, o agente móvel lê o índice de desempenho gerado pelo agente monitor em cada máquina, e, ao retornar ao computador mestre, ele escreve a lista dos *hosts* que ele passou em um arquivo, que será lido pelo escalonador para poder realizar o escalonamento dos processos entre as máquinas do sistema e atingir o balanceamento de cargas.

Como discutido anteriormente, as permissões referente a manipulação de arquivos estão no arquivo *java.policy*, mas devido a falta de material sobre o assunto, foi difícil conseguir superar esse problema.

Um outro problema foi a utilização de vetores nos agentes. A biblioteca *Aglets* não permite que alguns tipos de variáveis sejam transportadas para outros *hosts*, variáveis estáticas e transientes não são levadas junto com o agente móvel.

Como não é possível o uso de uma variável estática, um vetor criado dentro da classe do agente móvel precisa ser iniciado. No inicio da implementação ocorreu um problema com o agente móvel, ao se transferir para outra máquina não passava para o próximo *host*. Os únicos vetores que foram usados sem problemas foram criados dentro dos métodos da classe do agente. Mas, por serem variáveis locais, suas informações eram perdidas ao passar para o *host* seguinte, o que impossibilitava o agente a carregar as cargas coletadas e o caminho que ele deveria seguir.

Para suprir esse problema foram usadas variáveis do tipo String para levar as cargas, os nomes dos *hosts* e o caminho a ser seguido pelo agente móvel. Uma variável do tipo Scanner foi usada para ler essas informações nas Strings e passá-las para vetores nos métodos em que elas seriam utilizadas.

5.5. Considerações Finais

Apesar dos problemas que surgiram na fase de implementação do agente móvel, o mesmo se mostrou eficiente no trabalho para o qual foi desenvolvido.

As funções providas pelo *Aglets* se mostraram eficientes para fazer o transporte do agente móvel nos nós do sistema. Tornando assim possível coletar o índice de desempenho gerado pelo monitor de recurso.

Ao fazer a coleta periodicamente entre as máquinas, o agente móvel cria uma lista atualizada das máquinas do sistema. Dessa forma, o escalonador pode fazer o escalonamento de processos baseado nessa lista, tendo assim a classificação das máquinas conforme seu *status*, sendo da máquina que possui menos recursos disponíveis para a que possui mais recursos disponíveis.

Devido ao êxito do agente móvel em coletar as cargas, no capítulo a seguir mostra testes de desempenho utilizando o agente móvel e a plataforma JPVM.

CAPÍTULO 6. TESTE DO AMBIENTE COM AGENTE MÓVEL

Concluída a fase de implementação do agente móvel, o mesmo se mostrou capaz de realizar a coleta dos índices de desempenho gerados pelo agente monitor em cada um dos *hosts* do sistema. Sendo assim o próximo passo do projeto está relacionado com o desempenho do agente móvel.

A coleta dos índices de carga deve ser feita com uma freqüência correta para manter esses índices atualizados sem sobrecarregar a rede de comunicação. Se o agente móvel fizer a coleta com uma freqüência muito alta, o sistema poderá perder desempenho, pois o tráfego gerado na rede será muito alto, dificultando assim a comunicação entre as máquinas. Mas, considerando que os índices de carga dos requisitos mudam muito rapidamente, uma demora da coleta irá fazer com que os índices informados ao escalonador fiquem desatualizados, não podendo assim realizar o escalonamento de processos de maneira correta.

Neste Capítulo serão mostrados testes feitos com o agente móvel desenvolvido neste trabalho junto com a biblioteca JPVM.

6.1. JPVM

O JPVM (*Java Parallel Virtual Machine*) é uma biblioteca semelhante ao PVM (*Parallel Virtual Machine*) que abriga classes de objetos para implementação que permite a troca de mensagens explícitas baseado em memória distribuída MIMD (FERRARI,1997).

O PVM, por sua vez, é um conjunto de bibliotecas, que por meio de troca de mensagens, pode emular um sistema paralelo. Considerado um dos primeiros ambientes a funcionar em máquinas heterogêneas e que os pesquisadores demonstraram interesse (MIGUEL, 2006).

O PVM possui várias rotinas que permitem que sejam efetuados a sincronização e o comando das tarefas na máquina paralela virtual. Essa máquina paralela virtual lida de forma transparente com as tarefas através de passagem de mensagens, que é o meio de comunicação entre os vários computadores interligados (MIGUEL, 2006).

A interface de programação do JPVM é semelhante a dos sistemas PVM, facilitando assim a migração dos programadores que estão habituados a usar o PVM para o JPVM.

As principais vantagens do JPVM sobre o PVM são as características herdadas da linguagem Java. Dentre elas é possível destacar a portabilidade e interoperabilidade provenientes pela linguagem Java. (FERRARI,1997).

A portabilidade é referida a facilidade de execução em praticamente todas as plataformas que suportam a máquina virtual Java, rodando em plataformas *Windows*, *Unix* e seus derivados e *Macintosh.*

A interoperabilidade refere-se ao ponto alto do JPVM, desde que a rede de comunicação esteja devidamente instalada o JPVM praticamente não oferece dificuldades na união de máquinas de plataformas diferentes (FERRARI,1997).

O JPVM é composto por dois componentes, o *jpvmDaemon* e o *jpvmEnvironment*.

O *jpvmDaemon* tem a função de coordenar às tarefas e fazer a comunicação dos processos e sua execução ocorre em segundo plano nos nós da máquina virtual (FERRARI,1997).

Já o *jpvmEnvironment* é uma biblioteca que contem as primitivas que permitem troca de mensagens, criação e eliminação de processos, sincronização de tarefas e modificação da máquina virtual, envio e recebimento de mensagens. As aplicações desenvolvidas pelo usuário utilizam o ambiente paralelo criado pelo JPVM através dessa biblioteca (FERRARI,1997)(MIGUEL, 2006)(SABATINE, 2007).

6.2. Testes

Para verificar se o uso do agente móvel diminui o tráfego da rede, aumentando assim o desempenho do sistema, foram feitos alguns testes comparando o JPVM tradicional e o JPVM com o agente móvel integrado.

O JPVM já possui um escalonador padrão, esse escalonador utiliza o modelo *round-robin*. Para inserir o agente móvel no JPVM foi acoplado ao *jpvmDaemon* o agente monitor (MARQUES, 2007) e um novo escalonador de processos. Esse escalonador, através da lista passada pelo agente móvel, utiliza a melhor máquina para fazer a distribuição das tarefas.

Após adicionar a quantidade de máquinas passada como argumento, o *jpvmDaemon* da máquina mestre inicia o agente monitor em todas as máquinas. Iniciar os agentes monitores ao mesmo tempo é muito importante, pois caso não ocorra, o agente móvel coletará cargas desatualizadas nas máquinas, comprometendo assim o escalonamento.

Logo após iniciar os agentes monitores, foi iniciado o agente móvel de maneira manual. O agente móvel precisou ser iniciado manualmente porque quando era iniciado

pelo *jpvmDaemon* após algum tempo ele parava de percorrer os *hosts*, não atualizando mais a lista de máquinas do escalonador.

Algumas modificações foram feitas também no agente móvel e no agente monitor. O agente monitor, além de salvar o índice de desempenho, recebeu a função de gravar o nome da máquina que está executando e a porta utilizada pelo JPVM. No caso do agente móvel, foi tirada função NomeMaquina(), pois tanto o nome quanto a porta do JPVM já eram lidos do arquivo onde o monitor grava essas informações. Além disso, na lista gerada pelo agente móvel que era passada para o escalonador, foi adicionada a porta usada pelo JPVM.

Depois que o agente móvel começa a percorrer as máquinas, o escalonador é iniciado na máquina mestre. Desse modo a máquina paralela virtual é criada, estando apta a receber tarefas.

Os primeiros testes foram realizados em um sistema homogêneo, sendo realizados nos laboratórios do Centro Universitário "Eurípides Soares da Rocha" de Marília – UNIVEM. Os experimentos foram realizados em uma rede de computadores pessoais conectados por uma rede padrão *ethernet* 100 Mbits interligada por um *switch*, em que todos utilizavam o sistema operacional Linux (*kernel* 2.6). A rede foi composta por 4 máquinas homogêneas (Pentium IV de 2.7GHz com 512Mbytes de RAM), a qual possibilitou a formação de uma arquitetura MIMD com memória distribuída por meio da API JPVM.

Foram feitos testes utilizando um programa de multiplicação de matrizes criado por (FERRARI, 1997). Esse programa está disponível no diretório de exemplos do JPVM.

Os resultados dos testes estão apresentados na Tabela 4.

Tabela 4. Resultado dos testes em Sistema Homogêneo

	JPVM Padrão			JPVM com *Aglets*		
Tarefas	100x100	1000x1000	2000x2000	100x100	1000x1000	2000x2000
4	222.06ms	4049.0ms	32969.4ms	283.16ms	5815.75ms	35284.0ms
9	388.13ms	4070.35ms	33693.0ms	628.53ms	4431.21ms	48361.8ms
16	671.46ms	4118.15ms	39040.0ms	1077.0ms	5310.66ms	50326.33ms

Nos testes realizados em um sistema homogêneo, não houve ganho em relação ao tempo de execução da aplicação. Isso se deve ao fato das máquinas possuírem os mesmos recursos, tornando assim o cálculo do índice de desempenho dispensável, pois o número de tarefas atribuídas a cada máquina já é o suficiente para realizar o escalonamento. Mas este teste mostra que, mesmo tendo um desempenho pior do que o padrão, o JPVM com o agente móvel funciona corretamente. Na Figuras 21, 22 e 23 são mostrados gráficos representando o desempenho do sistema.

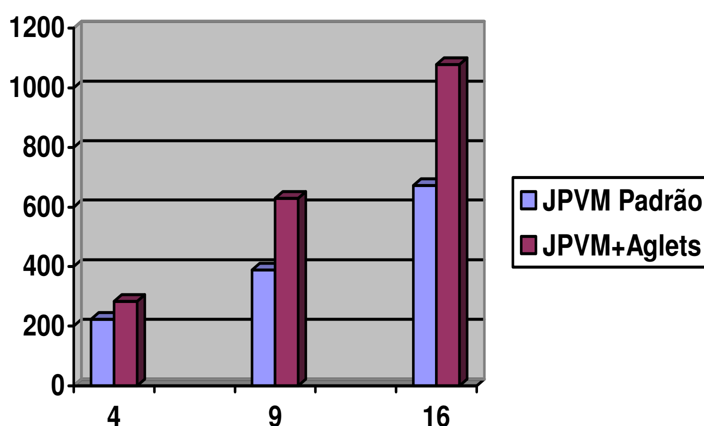

Figura 21. Gráfico da matriz 100x100 em sistema homogêneo

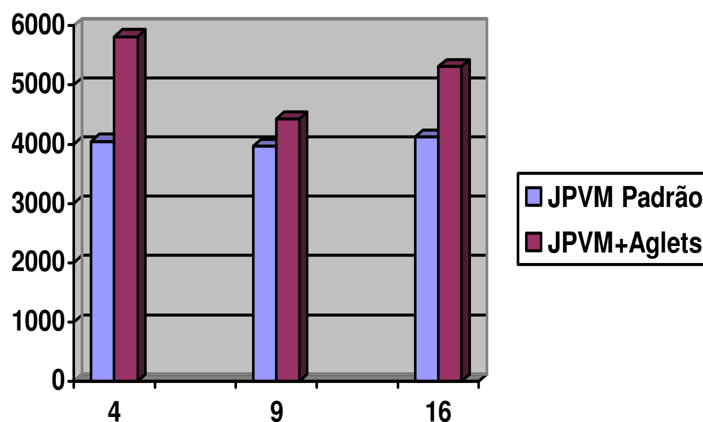

Figura 22. Gráfico da matriz 1000x1000 em sistema homogêneo

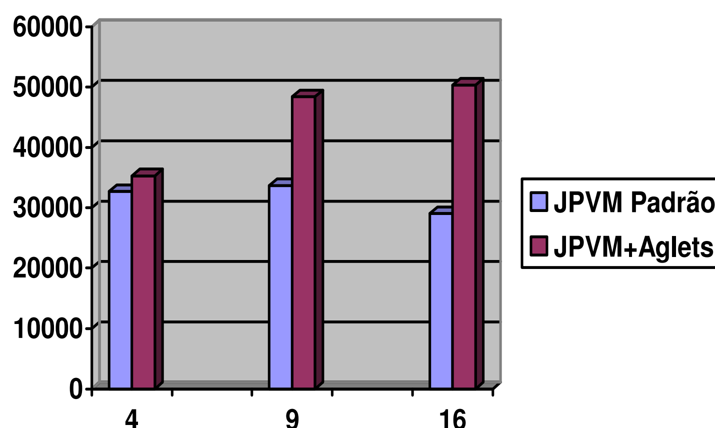

Figura 23. Gráfico da matriz 2000x2000 em sistema homogêneo

Alguns testes em sistemas heterogêneos foram realizados. Para isso foi utilizado o laboratório LAS do Centro Universitário "Eurípides Soares da Rocha" de Marília – UNIVEM. Para a análise de desempenho do ambiente proposto foi comparado o JPVM padrão com o JPVM modificado, sendo o ambiente composto por máquinas heterogêneas: um AMD Athlon MP 2000+ com 1024MB de memória (mestre), um Pentium 4 1600 MHz com 256MB de memória e um Pentium 4 1600 MHz com 128MB de memória, todas interligadas por uma rede ethernet de 10Mb/s.

Na Tabela 5 são mostrados os resultados dos testes.

Tabela 5. Resultados dos testes em Sistema Heterogêneo

	JPVM Padrão			**JPVM com *Aglets***		
Tarefas	100x100	1000x1000	2000x2000	100x100	1000x1000	2000x2000
4	473.9ms	12772.6ms	125648.5ms	423.81ms	11982.83ms	115841.0ms

Os resultados dos testes utilizando um sistema heterogêneo, o uso do agente móvel mostrou um pequeno ganho em relação ao JPVM padrão. Esse pequeno ganho mostra que o escalonador está conseguindo distribuir as tarefas de acordo com a quantidade de recursos disponíveis em cada máquina e o agente móvel está atualizando a lista de máquinas de maneira correta. Nas Figuras 24, 25 e 26 são mostrados gráficos com os resultados dos testes.

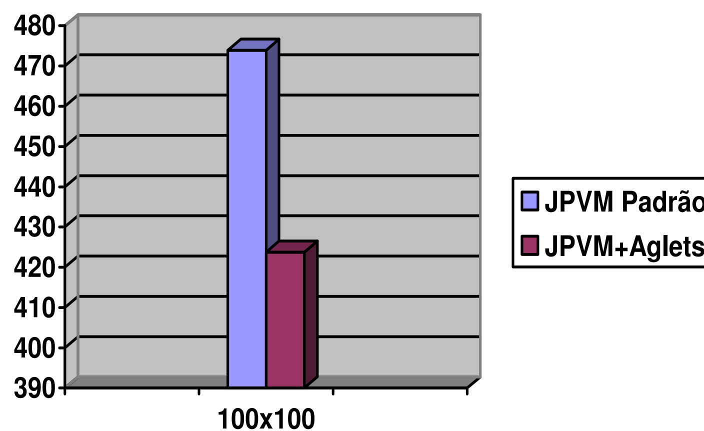

Figura 24. Gráfico da matriz 100x100 em sistema heterogêneo

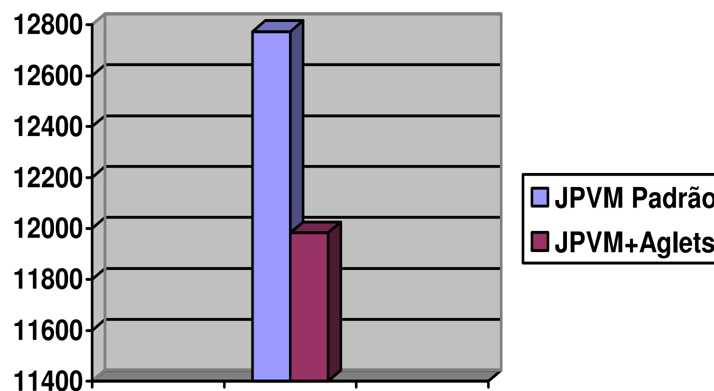

Figura 25. Gráfico da matriz 1000x1000 em sistema heterogêneo

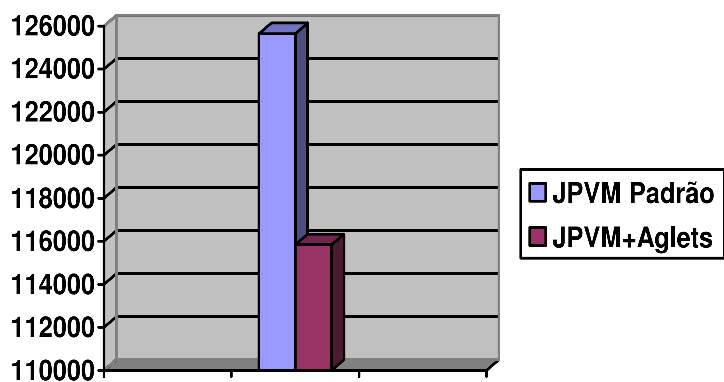

Figura 26. Gráfico da matriz 2000x2000 em sistema heterogêneo

Para se obter um resultado exato, será necessário realizar mais testes em sistemas heterogêneos. Para esses testes serão aumentados os números de tarefas para a multiplicação de matrizes e o número de máquinas no sistema.

É previsto que ao aumentar o número de máquinas do sistema seja obtido um desempenho melhor do agente móvel em relação ao JPVM padrão.

CAPÍTULO 7. CONCLUSÃO

Para um sistema Paralelo/Distribuído a preocupação em fazer o balanceamento de cargas é um fator essencial para se obter um melhor desempenho do sistema. Neste trabalho foram utilizados índices de carga e de desempenho, dando ao escalonador um parâmetro para realizar o balanceamento, mantendo assim os elementos de processamento com uma carga coerente aos seus recursos, obtendo assim em alguns casos um desempenho superior.

O uso de agentes móveis para a coleta dos índices se mostrou uma alternativa interessante. O agente se mostrou eficiente ao efetuar a coleta dos índices nas máquinas sem aumentar gerar um tráfego prejudicial ao sistema, o que era um dos objetivos do projeto. O ganho obtido em sistemas heterogêneos mostra que a lista criada pelo agente móvel serviu como parâmetro para o escalonador escolher as máquinas que receberiam cada tarefa, eliminando assim a sobrecarga em determinadas máquinas.

As funções providas pela plataforma *Aglets* foram suficientes para realizar a criação e o tráfego do agente móvel, mostrando um bom desempenho ao realizar as suas respectivas funções.

Os resultados mostraram que houve uma compatibilidade entre o agente móvel e a API JPVM, tornando o uso do agente móvel para coleta de cargas em sistemas Paralelos/Distribuídos uma boa alternativa na busca por um melhor desempenho.

7.1. Trabalhos Futuros

Para dar continuidade ao trabalho realizado serão realizados testes em sistemas Paralelos/Distribuídos heterogêneos. As multiplicações das matrizes serão divididas em um maior número de tarefas, além de aumentar o número de máquinas do sistema, onde se espera obter um ganho maior.

Após o termino da fase de testes e coletas de resultado, tem-se como trabalho futuro o refinamento do código do agente móvel, em busca de um melhor desempenho do mesmo, aumentando assim o desempenho do sistema em geral.

Obtido todos os resultados dos testes, o próximo trabalho a ser realizado é um estudo comparativo entre o agente móvel proposto neste trabalho, que utiliza a biblioteca *Aglets* com o proposto por (MARQUES,2007) que utiliza a biblioteca muCode.

REFERÊNCIAS BIBLIOGRÁFICAS

(ARIADOR & LANGE, 1998) Aridor, Y., Lange, D. B. (1998) Agent Design Patterns: Elements of Agents Application Design. Proceedings of the second international conference on Autonomous agents, Minneapolis, Minnesota, United States, Pages: 108 - 115

(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.

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

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

(FERRARI,1997) FERRARI, A. J. JPVM: Network Parallel Computing in Java. Department of Computer Science, University of Virginia, Charlottesville,USA, 1997.

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

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

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

(KERSHENBAUM; HARRISON; CHESS.1995) D. Chess, C. Harrison, A. Kershenbaum. Mobile Agents: Are they a good idea? Lecture Notes in Computer Science, 1995.

(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.

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

(LANGE;OSHIMA,1998)Lange, D. B. and M. Oshima (1998), Programming and Deploying Mobile Agents with Java. Forthcoming book. Addison-Wesley, Reading, MA.

(MARQUES,2007) MARQUES G. M. Avaliação de Índices de Carga e de Desempenho em Ambientes Paralelos/Distribuídos com Agentens Móveis. Dissertação (Mestrado) UNIVEM, Marília 2007.

(MIGUEL, 2006) MIGUEL,R.F. Avaliação de Desempenho de Aplicações utilizando JPVM, Centro Universitário “EURÍPIDES DE MARÍLIA” – UNIVEM, Bacharelado em Ciência da Computação, 2006.

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

(OYAMADA; ITO , 1998): OYAMADA, M. S.; ITO, S. A. Aglets: Agentes Móveis em Java. UFRGS – CPGCC,1998.

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

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

(SABATINE, 2007) SABATINE,R. J. Implementação de Algoritmo Paralelo para Apoiar o Processamento de Imagens Utilizando JPVM, Centro Universitário "EURÍPIDES DE MARÍLIA" – UNIVEM, Bacharelado em Ciência da Computação, 2007.

(SHIRAZI; HURSON; KAVIN, 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; KRUEGER; SINGHAL, 1992) SHIVARATRI, N. G.; KRUEGER, P.; SINGHAL, M. Load Distribution for Locally Distributed Systems. IEEE Computer. 1992.

(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.

(TANENBAUM; 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.

(WANG;MORRIS,1985) WANG, Y-T; MORRIS, R. J. T. (1985). Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, v. c-34, n. 3, p. 204-217, March.

(Xu; Lau) Xu, C.; Lau, F.C.M. (1997). Load Balancing in Parallel Computers: Theory and Practice. Kluwer Academic Publishers, Boston, USA, 1997.

(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 – Código fonte do agente móvel para JPVM.

```
import java.net.URL;
import java.util.Scanner;
import java.io.*;
import com.ibm.aglet.*;
import com.ibm.aglet.event.*;


public class jpvmMobileAgent extends Aglet {
      int num_host=0;
      int step = 0,aux=0;
      int lastHost = 0;
      URL nexthost;
        double media=0;
    long tempoE1, tempoE2, tempoER, tempoP1, tempoPR;
      String nome="",carga="",host="";

  public void onCreation (Object o){

        Scanner sx;
        try{
                  File arq = new File( "/tmp/hosts.txt");
                  FileReader reader = new FileReader(arq);
                  BufferedReader buff = new BufferedReader(reader);
                  sx = new Scanner (buff);
                  while (sx.hasNext()){
                              host+=sx.next()+" ";
                              num_host++;

                        }
                        System.out.println(num_host);
                  }
                  catch(Exception e){
                        System.out.println("erro host");
                  }

      addMobilityListener(
                        new MobilityAdapter(){
                              public void onDispatching(MobilityEvent e){
                              }
                              public void onArrival(MobilityEvent e){


                              }
                        }
            );
    }


    public void run(){

      Scanner sx;
      double auxVet;
        String auxVet2,teste;
        int i=0;
      for(;;){

                              double vetor[] = new double[num_host];
```

```
                        String hosts[] = new String[num_host];
                        String hosts2[] = new String[num_host];
                        //String porta[] = new String[num_host];
//                            ********definiÃ§Ã£o dos hosts*********
                        sx = new Scanner(host);
                        while(sx.hasNext()){
                                  hosts[i]=(sx.next());
                                  i++;
                            }
//                      *****COleta da carga e do nome da maquina***
                        carga+=" "+Coleta();

//                      ******** Chegada ao mestre**********
                            if(lastHost==0){
                        try{Thread.sleep(400);}
                        catch(Exception e){}
                        tempoE1 = System.currentTimeMillis();

                            if(aux != step){ tempoPR =
System.currentTimeMillis() - tempoP1; }
//                      *********OrdenaÃ§Ã£o do VEtor***************
                            Scanner sc = new Scanner(carga);
                            i=0;
                            while(sc.hasNext()){

      vetor[i]=Double.parseDouble(sc.next());
                                  hosts2[i]=sc.next();
                                  hosts2[i]+=" "+sc.next();
                                  i++;
                            }
                            boolean changed = true;
                               try{
                               while (changed) {
                                    changed = false;
                                    //Metodo de Ordenacao: Bubble

                                    for (i = 0; i < vetor.length -1;
i++) {
                                        if (vetor[i] < vetor[i + 1]) {

                                        auxVet = vetor[i];
                                               auxVet2 = hosts2[i];
                                          vetor[i] = vetor[i+1];
                                        hosts2[i] = hosts2[i+1];
                                        vetor[i+1] = auxVet;
                                        hosts2[i+1] = auxVet2;

                                            changed = true;
                                        }
                                    }
                                }
                               }
                               catch(Exception
e){System.out.println("erro
      ordenaÃ§Ã£o");}
//                              *********FIm da ordenaÃ§Ã£o************
//                        *******Calculo da média dos indices**********
                            for (i=0;i<vetor.length;i++)
                                  media+=vetor[i];
                            media=media/num_host;
                            media+=media*0.3;
//                          *******Salva resultado e zera variaveis****
                          salvaResultado(hosts2,vetor);
                          carga="";
                          nome="";
                 tempoP1 = System.currentTimeMillis();
                      aux = step;
                      step++;
```

```
                        System.out.println("  Tempo de percurso: " +
tempoPR + " milisegundos \n" + "passo: "+step );

                          }
//                              *********Proximo host**********
                          if (lastHost == (num_host - 1) ) { lastHost = 0;}

                           else { lastHost++; }


                          try
                    {
                          System.out.println("Proximo host... \n");

                    dispatch(new URL(hosts[lastHost]));

                    }
                    catch (Exception e) {
                                System.out.print(e.getMessage());
                    }

      }

    }
      public String Coleta(){
            String coletaID="";
              try{
                   String fileName = ("/tmp/id.txt");
                   File f = new File(fileName);
                   FileReader ler = new FileReader(f);
                   BufferedReader buff = new BufferedReader(ler);
                   String line;
                   while ((line = buff.readLine()) != null){
                           Scanner sc = new Scanner(line);
                             while (sc.hasNext()){
                                 coletaID = sc.next();
                                 coletaID +=" "+sc.next();
                                 coletaID +=" "+sc.next();
                           }
                        }
                       ler.close();
                       buff.close();
              }
              catch (Exception e){System.out.println("erro leitura");}
              return coletaID;
      }
public void salvaResultado(String[] hosts2, double[] vetor){
            try
            {
                  BufferedWriter salva = new BufferedWriter(new
FileWriter("/tmp/id_h.txt"));

                  salva.write(String.valueOf(hosts2.length)+"\n");
                  for(int i=0;i<hosts2.length;i++)
                        if (vetor[i]<media)
                        salva.write(hosts2[i]+"\n");

            salva.close();
            media=0;
      } catch     (Exception e) {System.out.println("erro ao salvar
dados");}
}
}
```

Apêndice B – Instalação da Plataforma Aglets

O primeiro passo para criação de um agente móvel em *Aglets* é a instalação do ambiente de desenvolvimento (ASDK) e da biblioteca *Aglets* cuja versão atual é a 2.0.2, a qual pode ser baixado em http://sourceforge.net/projects/*Aglets*/. É necessária a instalação da plataforma Java, e a versão utilizada é a 1.6 (http://java.sun.com). A instalação foi realizada em um ambiente Linux (*kernel* 2.6).

Após o *download* do *Aglets*-2.0.2.jar, é necessário fazer a descompressão do mesmo. O diretório utilizado para extração/instalação foi ~/java/*Aglets*. Para realizar a extração foi executado o seguinte comando.

```
# jar xvf Aglets-2.0.2.jar
```

Terminada a descompressão dos arquivos é necessária a instalação da plataforma, para isso é necessário o uso do Apache Ant, que é um software feito para compilar e instalar aplicativos Java. Essa versão do *Aglets* já possui uma versão do Ant compatível para a instalação da plataforma.

Na instalação foi utilizada a versão do Ant contida no *Aglets*, para isso foi necessário acessar a pasta ~/java/*Aglets*/bin, que é onde se encontra o arquivo de construção do Ant *build.xml*, e dentro do diretório executar os seguinte comandos:

```
# chmod 755 ant
# ./ant
```

Políticas de Segurança

Assim como outras aplicações Java, o *Aglets* precisa de um arquivo *java.policy* para ter acesso a *sockets*, arquivos, etc. É necessária a criação do arquivo *Aglets.policy*, o qual é criado a partir do Ant com o comando:

```
# ant install-home
```

Esse comando além de criar o arquivo *Aglets.policy*, também cria um *keystore* que contém chaves para migrações seguras dos agentes.

Variáveis de Ambiente

O passo seguinte foi configurar as variáveis de ambiente necessárias. Além das variáveis padrões do Java, é necessária a criação de novas variáveis para o *Aglets*. As variáveis podem ser definidas da seguinte maneira:

```
export AGLETS_HOME=/java/Aglets
export AGLETS_PATH=$AGLETS_HOME
export PATH=$AGLETS_HOME/bin
export classpath=$classpath:/java/Aglets/lib/Aglets-2.0.2.jar
```

Configurações Finais

Após a instalação da plataforma, algumas configurações são necessárias para que o *aglet* consiga trafegar entre os *hosts*, e ler e escrever em arquivos.

A configuração para o tráfego entre os *hosts* é feita no arquivo ~/java/*Aglets*/cnf/*Aglets*.props. As seguintes configurações devem ser feitas:

atp.resolve=true

Aglets.secure=false

Para que o agente móvel tenha permissão de leitura e escrita em um determinado arquivo ou diretório são necessárias alterações no arquivo ~/.*Aglets*/security/*Aglets*.policy, e a seguinte alteração deve ser feita:

permission java.io.FilePermission "/tmp" , "read,write"

Essa alteração permite que o *aglet* leia e escreva em arquivos. O diretório utilizado para criar arquivos foi o /tmp.

Configurações Tahiti

Para que o servidor Tahiti funcione de maneira correta e eficiente, são necessárias algumas alterações em seus arquivos de configuração. O primeiro arquivo de configuração a ser editado é o *Aglets*.props localizado no diretório ~/java/*Aglets*/cnf. Nesse arquivo são encontradas configurações de protocolo de transferência dos agentes, de segurança, de usuários e outras configurações do próprio Tahiti.

No *Aglets*.props foram necessárias alterações em algumas das opções nele presente. A primeira alteração foi na opção de segurança do agente. A utilização da segurança do servidor é utilizada como padrão, para que todas as atividades do servidor fossem permitidas a opção por segurança foi retirada, através da alteração do seguinte campo:

Aglets.secure=false

Outra opção importante que deve ser alterada é a atp.resolve. Essa opção, por padrão, é desabilitada. Foi necessário habilitá-la para que os *hosts* fossem habilitados para transferir agentes entre eles. Portanto a opção foi alterada da seguinte forma:

atp.resolve=true

Para tornar a execução do agente móvel mais rápida, foi desabilitada a opção atp.secureseed. Essa opção desabilitada diminui a segurança do agente, mas torna o servidor mais rápido.

atp.secureseed=false

As duas últimas alterações feitas no *Aglets*.props foi relacionado ao modo de iniciar o servidor. Quando o servidor é iniciado, são pedidos um nome de usuário e uma senha para prosseguir a execução do Tahiti. Ao colocar esse nome de usuário e sua respectiva senha diretamente no arquivo de configurações, não é necessário informá-los

novamente quando o servidor for iniciado. O nome de usuário e sua senha são colocados nas seguintes opções:

Aglets.owner.name=nome_do_usuário

Aglets.owner.password=senha_do_usuário

O agente móvel desenvolvido nesse trabalho tem a necessidade de manipular arquivos que se encontram em diretórios específicos do *host*. Para que o *aglet* tenha permissão de ler e escrever nesses arquivos, outro arquivo deve ser editado. O arquivo *Aglets*.policy pode ser encontrado no diretório ~/.*Aglets*/security, e é nesse arquivo que são estabelecidas as permissões dos agentes. Sendo assim, para permitir que o agente móvel manipule os arquivos /tmp/hosts.txt, /tmp/id.txt e /tmp/id_h.txt é necessária a inserção das seguintes linhas:

permission java.io.FilePermission "/tmp/hosts.txt", "write";

permission java.io.FilePermission "/tmp/id.txt", "read";

permission java.io.FilePermission "/tmp/id_h.txt", "write";

Anexo A – Código fonte mult_math (exemplo do JPVM)

```
/* mat_mult.java
*
* A simple parallel matrix-matrix multiply example using jpvm.
*
* Adam J Ferrari
* Tue Jun 4 10:31:02 EDT 1996
*
* Copyright (C) 1996 Adam J Ferrari
*
* This library is free software; you can redistribute it and/or
* modify it under the terms of the GNU Library General Public
* License as published by the Free Software Foundation; either
* version 2 of the License, or (at your option) any later version.
*
* This library is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
* Library General Public License for more details.
*
* You should have received a copy of the GNU Library General Public
* License along with this library; if not, write to the
* Free Software Foundation, Inc., 675 Mass Ave, Cambridge,
* MA 02139, USA.
*/
import jpvm.*;
import java.io.*;
import java.util.*;
class mat_mult {
static final int ParamTag = 11;
static final int DoneTag = 22;
static final int PipeTag = 33;
static final int RollTag = 44;
static int numTasks = 0;
static int matDim = 0;
static int taskMeshDim = 0;
static int localPartitionDim = 0;
static int localPartitionSize = 0;
static int taskMeshRow = 0;
static int taskMeshCol = 0;
static int localTaskIndex = 0;
static boolean debug = false;
static jpvmTaskId myTaskId = null;
static jpvmTaskId masterTaskId = null;
static jpvmEnvironment jpvm = null;
static jpvmTaskId tids[];
static float C[], A[], B[], tempA[];
public static void main(String args[]) {
      double mmstart = 0.0;
      double start = 0.0;
      double end = 0.0;
      boolean cmd = false;

      try {
            jpvm = new jpvmEnvironment();
            myTaskId = jpvm.pvm_mytid();
            masterTaskId = jpvm.pvm_parent();
            if(masterTaskId==jpvm.PvmNoParent) {
                  if (args.length != 2) usage();
                  cmd = true;
                  try {
                        numTasks = Integer.parseInt(args[0]);
                        matDim = Integer.parseInt(args[1]);
```

```
                    }
                    catch (NumberFormatException e) {
                          usage();
                    }
                    System.out.println(""+matDim+"x"+matDim+
                    " matrix multiply, "+numTasks+" tasks");
                    start = msecond();
                    taskMeshDim = getTaskMeshDim(numTasks);
                    tids = new jpvmTaskId[numTasks];
                    if(numTasks>1)
                    jpvm.pvm_spawn("mat_mult",numTasks-1,tids);
                    tids[numTasks-1] = tids[0];
                    tids[0] = myTaskId;
                    localTaskIndex = 0;
                    /* Broadcast parameters to all tasks */
                    mmstart = msecond();
                    jpvmBuffer buf = new jpvmBuffer();
                    buf.pack(taskMeshDim);
                    buf.pack(numTasks);
                    buf.pack(tids,numTasks,1);
                    buf.pack(matDim);
                    jpvm.pvm_mcast(buf,tids,numTasks,ParamTag);
              }
              else {
                    // Normal worker task - get parameters...
                    jpvmMessage m = jpvm.pvm_recv(ParamTag);
                    taskMeshDim = m.buffer.upkint();
                    numTasks = m.buffer.upkint();
                    tids = new jpvmTaskId[numTasks];
                    m.buffer.unpack(tids,numTasks,1);
                    for (localTaskIndex=0;localTaskIndex<numTasks;
                          localTaskIndex++)
                          if (myTaskId.equals(tids[localTaskIndex]))
                                break;
                          matDim = m.buffer.upkint();
              }
              /* Do the matrix multiplication */
              matmul();
              if(cmd) {
                    end = msecond();
                    System.out.println("Total time: "+(end-start)+
                    " (mult: "+(end-mmstart)+")");
              }
              jpvm.pvm_exit();
        }
        catch (jpvmException jpe) {
        error("jpvm Exception - "+jpe.toString(),true);
        }
  }
  public static void matmul() throws jpvmException {
        int i,j,k;
        localPartitionDim = matDim/taskMeshDim;
        localPartitionSize = localPartitionDim*localPartitionDim;
        taskMeshRow = localTaskIndex/taskMeshDim;
        taskMeshCol = localTaskIndex%taskMeshDim;
        A = new float[localPartitionSize];
        B = new float[localPartitionSize];
        C = new float[localPartitionSize];
        tempA = new float[localPartitionSize];
        if(debug) {
              System.out.println("localTaskIndex\t= "+localTaskIndex);
              System.out.println("matDim\t= "+matDim);
              System.out.println("taskMeshDim\t= "+taskMeshDim);
              System.out.println("localPartitionDim\t= "+
                    localPartitionDim);
              System.out.println("localPartitionSize\t= "+
                    localPartitionSize);
              System.out.println("taskMeshRow\t= "+taskMeshRow);
```

```
            System.out.println("taskMeshCol\t= "+taskMeshCol);
        }
        for (i=0;i<localPartitionSize;i++) {
            C[i] = (float) 0.0;
            A[i] = (float)(i+localTaskIndex*(taskMeshRow+1));
            B[i] = (float)(i-localTaskIndex*(taskMeshRow+1));
        }
        for (i=0;i<taskMeshDim;i++) {
            Pipe(i);
            Multiply();
            if(i<(taskMeshDim-1)) Roll();
        }
    }
    public static void Pipe(int iter) throws jpvmException {
        int i;
        if (taskMeshCol == (taskMeshRow+iter)%taskMeshDim) {
            jpvmBuffer buf = new jpvmBuffer();
            buf.pack(A,localPartitionSize,1);
            for (i=0;i<taskMeshDim;i++)
                if (localTaskIndex != taskMeshRow*taskMeshDim+i) {
                    jpvm.pvm_send(buf,
                        tids[taskMeshRow*taskMeshDim+i],PipeTag);
                }
        }
        else {
            jpvmMessage m = jpvm.pvm_recv(PipeTag);
            m.buffer.unpack(tempA,localPartitionSize,1);
        }
    }
    public static void Multiply() throws jpvmException {
        int i,j,k;
        float temp;
        for (i = 0; i < localPartitionDim; i ++)
            for (j = 0; j < localPartitionDim; j++) {
                temp = 0;
                for (k = 0; k < localPartitionDim; k++)
                    temp += A[i*localPartitionDim+k] *
                        B[k*localPartitionDim+j];
                C[i*localPartitionDim+j] += temp;
            }
    }
    public static void Roll() throws jpvmException {
        int who = ( (taskMeshRow!=0) ?
            (taskMeshRow-1)*taskMeshDim+taskMeshCol:
            (taskMeshDim-1)*taskMeshDim+taskMeshCol);
        jpvmBuffer buf = new jpvmBuffer();
        buf.pack(B,localPartitionSize,1);
        jpvm.pvm_send(buf,tids[who],RollTag);
        jpvmMessage m = jpvm.pvm_recv(RollTag);
        m.buffer.unpack(B,localPartitionSize,1);
    }
    public static double msecond() {
        Date d = new Date();
        double msec = (double) d.getTime();
        return msec;
    }
    public static void error(String message, boolean die) {
        System.err.println("mat_mult: "+message);
        if(die) {
            if(jpvm!=null) {
                try {
                    jpvm.pvm_exit();
                }
                catch (jpvmException jpe) {
                }
                System.exit(1);
            }
        }
```

```
}
public static void usage() {
      error("usage - java mat_mult <tasks> <mat dim>", true);
}
public static int getTaskMeshDim(int n) {
      int dim;
      for(dim=1;dim<=n && n>0;dim++) {
            if(dim*dim == n) return dim;
      }
      error("Number of tasks must be an even square",true);
      return -1;
}
};
```