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

MÁRIO HENRIQUE DE SOUZA PARDO

VVM (VIRTUAL VIRTUAL MACHINE): UMA PROPOSTA DE PORTABILIDADE EFETIVA PARA A MÁQUINA VIRTUAL JAVA

MARÍLIA
2006

MÁRIO HENRIQUE DE SOUZA PARDO

VVM (VIRTUAL VIRTUAL MACHINE): UMA PROPOSTA DE PORTABILIDADE EFETIVA PARA A MÁQUINA VIRTUAL JAVA

Dissertação apresentada ao Programa de Mestrado do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares da Rocha, para obtenção do título de Mestre em Ciência da Computação. (**Área de Concentração**: *Arquitetura de Computadores*).

Orientador:
Prof. Dr. Marcos Luiz Mucheroni

MARÍLIA
2006

MÁRIO HENRIQUE DE SOUZA PARDO

VVM (VIRTUAL VIRTUAL MACHINE): UMA PROPOSTA DE PORTABILIDADE EFETIVA PARA A MÁQUINA VIRTUAL JAVA

Banca examinadora da qualificação apresentada ao Programa de Mestrado da UNIVEM, /F.E.E.S.R., para obtenção do Título de Mestre em Ciência da Computação. Área de Concentração: Arquitetura de Computadores.

ORIENTADOR: _____
Prof. Dr. Marcos Luiz Mucheroni

1º EXAMINADOR: _____
Prof. Dr. Jorge Luiz e Silva

2º EXAMINADOR: _____
Prof. Dr. Edward David Moreno Ordonez

Marília, _____de _____de 2006

AGRADECIMENTOS

Não importa o que fizermos, nunca o realizamos sozinhos, pois, sempre se tem o auxílio de algum pensador, professor ou amigo. Assim, gostaria de agradecer a todos que foram importantes para a possível realização desse trabalho de dissertação de mestrado.

Agradeço então:

Á DEUS e à meus pais, que além de me ajudarem no custeio do curso de mestrado sempre me incentivaram a estudar e continuar, insistir, perdurar em meus ideais. Não é sempre que digo, mas, vocês sabem que os amo. Este trabalho também é uma homenagem à todo esforço que fizeram e ainda fazem por mim.

Ao meu orientador, Prof. Dr. Marcos L. Mucheroni, pelo grande incentivo, pelas palavras de apoio nos momentos de fraqueza, pelos bons conselhos, mas, principalmente pela boa vontade.

Ao amigo Thiago Luiz Parrilo Rizzo, por todo apoio durante o curso, e, na finalização na ajuda com os bugs em java que pareciam não cessar mais.

A todos os amigos do mestrado que, de alguma forma contribuíram com idéias, com palavras de incentivo, valeu!!

Aos outros professores do mestrado que ensinaram não só o conteúdo das disciplinas, mas, lições sobre a natureza do verdadeiro pesquisador e lições de vida.

À todos vocês, o meu MUITO OBRIGADO!!!!!!!!

Mário Henrique de Souza Pardo

PARDO, Mário Henrique de Souza. **VVM (VIRTUAL VIRTUAL MACHINE): UMA PROPOSTA DE PORTABILIDADE EFETIVA PARA A MÁQUINA VIRTUAL JAVA.** 2006, 120f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

RESUMO

Pesquisas recentes revelam a tendência do uso de máquinas virtuais tanto em sistemas distribuídos, quanto nas linguagens de programação atuais, como Java, Ruby e outras. A busca pelo desempenho máximo das máquinas, facilidade de implementação dos programas de aplicação paralelos e implementação de novos projetos, modelos de máquinas virtuais com recursos inovadores, tem se revelado uma constante nos artigos publicados nesse domínio de conhecimento. Este trabalho disserta sobre tecnologias de comunicação para paralelização em software, projetos e modelos de máquinas virtuais, benchmarking para máquinas virtuais paralelas e distribuídas, além disso, propõe um novo modelo de implementação de máquina virtual paralela e distribuída e um protótipo de teste chamado VVM (Virtual Virtual Machine) com características, como por exemplo, a distribuição de uma instância da própria VVM pelo ambiente distribuído. A VVM é uma máquina virtual Java paralela e distribuída programada na própria linguagem de programação Java, e, as comparações de desempenho são feitas com a Jalapeño JVM (Java Virtual Machine) que é uma máquina virtual Java Paralela.

Palavras-chave: máquina virtual paralela e distribuída, java virtual machine, projeto de máquina virtual, benchmarking paralelo para máquina virtual java, instância de máquina virtual java.

PARDO, Mário Henrique de Souza. **VVM (VIRTUAL VIRTUAL MACHINE): UMA PROPOSTA DE PORTABILIDADE EFETIVA PARA A MÁQUINA VIRTUAL JAVA.** 2006, 120f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

ABSTRACT

Recent researches reveal the tendency of the use of virtual machines so much in distributed systems as in the current programming languages, like Java, Ruby and other. The search for the maximum performance, implementation facilities to parallel application programs, implementation of new projects and models of virtual machines with innovative resources, it has revealed a constant in the papers in that knowledge domain. This work talk about communication technologies for parallelism in software, projects and models of virtual machines, benchmarking for parallel and distributed virtual machines, besides, it proposes a new implementation model of a parallel and distributed virtual machine and a test prototype called VVM (Virtual Virtual Machine) with innovative features, for example, an instance distribution of own VVM for the distributed environment. VVM is a parallel and distributed virtual machine Java, programed in Java programming language, and, the performance comparisons are made with Jalapeño JVM (Java Virtual Machine) that is a Parallel Java Virtual Machine.

Keywords: distributed and parallel virtual machine, java virtual machine, virtual machine project, parallel benchmarking for java virtual machine, java virtual machine instance.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API: Application Program Interface

CP/CMS: Central Processing / Conversational Monitor System

DARPA: Defense Advanced Research Projects Agency

E/S: Entrada/Saída

GNU/Linux: GNU is Not Unix / Linux

HPF: High Performance FORTRAN

HPJava: High Performance Java

IR: Intermediate Representation

J2EE: Java 2 Enterprise Edition

J2ME: Java 2 Micro Edition

J2SDK: Java 2 Software Development Kit

JavaNOW: Java Network of Workstations

JIT: Just In Time

JPVM: Java Parallel Virtual Machine

JVM: Java Virtual Machine

MPI: Message Passing Interface

NOWs: Network of Workstations

NPAC: Northeast Parallel Architectures Center

PCRC: Parallel Compiler Runtime Consortium

POSIX: Portable Operating System Interface UniX

PVM: Parallel Virtual Machine

RMI: Remote Method Invocation

RMI+: Remote Method Invocation Plus

SMPs: Symmetric Multi-Processing

SO: Sistema Operacional

SPCs: Scalable Parallel Computers

SPMD: Single Program, Multiple Data

TSS/360: Time Shared System 360

VM/370: Virtual Machine 370

VM: Virtual Machine

VVM: Virtual Virtual Machine

PARDO, Mário Henrique de Souza.

VVM (VIRTUAL VIRTUAL MACHINE): UMA PROPOSTA DE PORTABILIDADE EFETIVA PARA A MÁQUINA VIRTUAL JAVA / Mário Henrique de Souza Pardo; Orientador: Marcos Luiz Mucheroni. Marília, SP, 2006.

120 f.

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

1. Arquitetura de Computadores 2. Máquinas Virtuais Paralelas e Distribuídas 3. Virtual Virtual Machine

CDD: 005.82

Sumário

1 - INTRODUÇÃO

No cotidiano, a comunidade de pesquisadores em ciência da computação já percebe que a *computação em cluster* e o *grid computing* são tendências de mercado, o que mostra o número crescente de pesquisas e aplicações na área. Contudo, ainda hoje, não há linguagens de programação que ofereçam o suporte e o desempenho desejados para esta nova realidade. Por isso, as iniciativas de pesquisa intensa a respeito desta temática ganharam forças. As máquinas virtuais vem sendo cada vez mais utilizadas em linguagens de programação e sistemas com a finalidade de aumentar o grau de portabilidade e também buscando desempenho comparável aos binários compilados.

Desta dissertação resultou um *modelo de implementação* e também um *protótipo de teste de máquina virtual Java distribuída*.

No capítulo 2, são abordados da história e os conceitos fundamentais de máquina virtual. Este estudo está, também, vinculado a várias particularidades sobre sistemas operacionais, pois, a VM (Virtual Machine) executa sobre um SO (Sistema Operacional), por isso, o estudo de *processos* e *threads* será fundamental para a compreensão de como os objetos são manipulados em tempo de execução. A compilação Just In Time também é abordada para a compreensão deste novo recurso, que foi implementado nas máquinas virtuais da Sun Microsystems e que traz uma série de características que podem influenciar o desempenho da execução das classes Java.

Na seqüência, é feita uma discussão detalhada sobre os principais *mecanismos de paralelização em software* (também conhecidos como sistemas distribuídos), entre eles: MPI (Message Passing Interface), RMI (Remote Method Invocation) e PVM (Parallel Virtual Machine). As informações obtidas no capítulo dão uma noção de como são os detalhes da paralelização das aplicações.

São então vistos, no capítulo 3, alguns dos projetos e modelos de máquinas virtuais paralelas atuais mais difundidos no meio científico. Entre os que são vistos em detalhes estão: HPJava, JavaSpaces, Titanium, JavaNOW, Manta e JavaParty. No entanto, estes não são os únicos, existem inúmeros outros projetos e modelos atualmente em desenvolvimento como *T-Spaces, Java/DSM, Orca e Multipol*. É realizada, ainda no capítulo 3, uma análise das máquinas virtuais que atualmente estão sendo utilizadas pela comunidade Java. A ótica é concentrada na máquina virtual da Sun Microsystems, na Kaffe Java Virtual Machine e Jalapeño Java Virtual Machine, porém, essas não são as duas únicas VMs disponíveis atualmente, existem muitas outras disponíveis como software livre e como software proprietário. Com objetivo de compreender como a máquina virtual Java se comporta em diferentes ambientes de teste, são estudados também alguns *softwares de benchmarking* para máquinas virtuais Java.

No capítulo 4 discute-se a proposta do modelo de implementação criado neste trabalho e a programação de um protótipo. Será detalhada toda a abordagem à execução do projeto, isto é, o modelo de implementação em si, as comparações de performance, comentários sobre a programação do protótipo, dados estatísticos levantados e um resumo de idéias a respeito da implementação.

No capítulo 5 estão as conclusões finais, no capítulo 6 os apontamentos sobre os trabalhos futuros que dão continuidade a este trabalho, e, no capítulo 7 estão as referências bibliográficas.

O apêndice A mostra todos os códigos-fonte em Java utilizados para a construção do release 1.0 do protótipo da VVM discutido nessa dissertação.

1.1. Motivação

A efetiva distribuição de tarefas Java e a distribuição da própria máquina virtual em si (ou seja, a distribuição de um carregador de classe somado aos bytecodes que se deseja

executar para o ambiente distribuído) ainda é um gargalo dessa tecnologia de linguagem de programação. A máquina virtual Java faz-se um ponto central de falhas para aplicações distribuídas, sendo assim, a motivação principal para esse trabalho é a possibilidade de se obter uma máquina virtual Java distribuída, aproveitando, dessa forma, o *paralelismo de máquina virtual* num patamar de granularidade que dispensa o programador de pensar paralelamente no momento de implementar seus algoritmos. Além disso, há ainda o fato de que a máquina virtual poderá distribuir uma instância de si mesma, possibilitando, dessa forma, que um as máquinas clientes do sistema distribuído não necessitem executar uma instância da JVM, logo, recebem a instância de máquina virtual via rede junto com os bytecodes a serem executados.

1.2. Objetivo

O objetivo principal deste projeto é propor a implementação de um protótipo para uma máquina virtual Java distribuída e paralela. Além disso, será também elaborado um *modelo de implementação* para esse projeto com o contribuir com um molde para implementação de máquinas virtuais distribuídas não só para Java como também para outras linguagens que deste recurso necessitem. Um protótipo em software para testes de desempenho também foi criado usando-se da própria linguagem Java.

2 - MÁQUINAS VIRTUAIS E MECANISMOS DE PARALELIZAÇÃO

Este capítulo mostra os detalhes que envolvem a teoria essencial que dará visão crítica a respeito das máquinas virtuais. As idéias aqui ilustradas, visam salientar as necessidades de máquinas virtuais paralelas e distribuídas com inovações técnicas, a fim de não só aumentar a performance, como também ir em busca de portabilidade desse tipo de VM (Virtual Machine).

Como citado no trabalho de (BULL et al, 2001), a linguagem Java provocou impactos significantes no campo da computação científica tradicional. Neste trabalho afirma-se que há algumas razões para que Java venha a causar outras transformações em um futuro próximo.

Talvez os benefícios mais óbvios são aqueles da portabilidade e das facilidades de engenharia de software (no quesito estruturação, principalmente) que vão ser particularmente importantes quando o *Grid Computing* (ou computação em grade) estiver numa fase ainda mais madura, de forma que, um usuário pode não saber *quando* seu programa será executado e em sobre que *arquitetura de computador* irá executar. Possuir um *garbage collector* perfeito e automático, através do qual se tenha checagem de tipo e ausência de ponteiros, faz o desenvolvimento Java significantemente menos propenso a erros do que as demais linguagens tradicionais tais como C, C++ e FORTRAN. Mas, é claro, segundo (BULL et al, 2001), Java não está livre de problemas.

Há algumas formas de se propor *mecanismos de paralelização de software* em geral e algumas essencialmente específicas para a máquina virtual Java. Neste estudo de revisão são abordados: MPI tradicional, PVM tradicional, MPIJava, JPVM, RMI+.

A importância deste estudo está na *compreensão da técnica* dos arranjos feitos pelos mecanismos de paralelização e distribuição, pois é através deles é que foi possível resolver quais as melhores estratégias e idéias para os objetivos propostos neste trabalho.

O trabalho de (BULL et al, 2001) é particularmente interessante para os tópicos subseqüentes, pois, menciona e explica, de forma sucinta, a maioria dos mecanismos que este estudo tomou como abordagem essencial.

2.1. Máquinas e Mecanismos Tradicionais de Paralelização: RMI, MPI, PVM e outros

Com objetivo de cobrir as necessidades de usuários que preferiam o uso dos computadores da IBM com tempo compartilhado (ou *time sharing*), usuários do sistema operacional OS/360 escreveram rotinas de tempo compartilhado para o mesmo.

A IBM, percebendo a demanda, lançou algum tempo depois o TSS/360 (TSS significa *Time Shared System)*, que chegou tarde demais e com uma lentidão e complexidade que causaram sua rejeição no mercado.

Um grupo de cientistas de Cambridge produziu um sistema a partir do TSS/360 da IBM (TANENBAUM, 2000), mas que possuía *performance* superior que o tornou absoluto e acabou absorvendo a fatia de mercado dos mainframes, com isso, até a IBM acabou por tomá-lo como produto. O sistema comentado acima foi chamado inicialmente de *CP/CMS* (*Conversational Monitor System*) e mais tarde batizado como *VM/370* (VM significa *Virtual Machine).*

Ele tinha duas missões que cumpriu com sucesso: *executar multiprogramação* e *oferecer uma máquina estendida* com interface mais conveniente (amigável) que o hardware de base (estritamente um IBM 370).

Na figura 1, pode-se observar uma abstração em camadas de como se dava o relacionamento do VM/370 com a máquina. É importante que se repare que este SO (Sistema Operacional) constitui as três camadas mais superiores da ilustração.

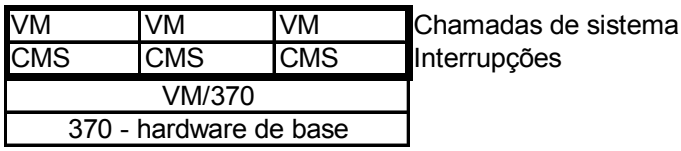


Figura 1 - O *VM/370 e seu projeto de implementação em camadas: (TANENBAUM, 2000).*

No VM/370, o coração do SO (Sistema Operacional) era o *monitor de máquina virtual* que rodava no hardware básico e executava a multiprogramação, era assim que este SO oferecia múltiplas máquinas virtuais funcionando em *pseudoparalelismo*.

Contudo, aqui, a VM não somente era uma máquina estendida (sistema de arquivos, memória e programas) ela realmente era uma cópia fiel do hardware implementado em software: incluindo subsistema de E/S, instruções, interrupções e etc.

Uma novidade estrutural lançada pelo sistema operacional VM/370 foi o chamado *exokernel* que implementa a camada de VM (primeira camada numa visão *top-down* da figura 1) para controlar e alocar recursos para as máquinas virtuais, ele monitora os recursos do hardware básico para que não haja conflitos.

(TANENBAUM, 2000) complementa suas idéias sobre VM dizendo que elas hoje são implementadas em hardware, a exemplo, o Intel Pentium, a pedido da Microsoft vem com um *modo virtual 8086* implementado em hardware para permitir retro-compatibilidade com programas antigos de MS-DOS.

2.1.1. Gerenciamento de Processos

Para que um sistema operacional possa executar seus programas geralmente eles implementam o *gerenciamento de processos*. Os processos são programas em execução sob o

controle do sistema operacional. Todavia, um processo é muito mais do que simplesmente o programa em execução: ele possui contador de programa, registradores e variáveis, e uma série de outras características importantes, além disso, os processos são sequenciais.

A organização dos processos em SO pode ser feita de duas formas: *puramente sequencial* (onde só um processo ocupa a CPU por vez até o fim de sua execução) ou usando-se de *multiprogramação*.

A multiprogramação pode ser implementada como pseudoparalelismo, paralela ou distribuída usando-se para tal um algoritmo de escalonamento.

Como o escalonamento (ou agendamento) de processos é uma tarefa implementada em algoritmos do SO, torna-se então claro que não se programam os softwares coordenados pela sua forma de execução quando tornarem-se processos, e sim, programam-se os mesmos usando as linguagens de programação em alto nível que livram o usuário da preocupação de pensar de forma paralela quanto ao tempo de execução.

Basicamente, pode-se imaginar um processo na forma de uma atividade, analogia de (TANENBAUM, 2000), assim ele tem: um programa, entrada, saída e um estado.

Para que a criação de processos não se torne uma desordem no SO e venha a desestabilizá-lo, algumas estratégias de contenção podem ser tomadas. Por exemplo, nos sistemas embarcados (ou *embedded systems*) pode-se projetar o hardware de forma que, na inicialização do conjunto, todos os processos necessários sejam criados e fiquem, em memória, esperando por algum trabalho (TANENBAUM, 2000).

Como a memória dos microcomputadores é usada para muitas atividades, é necessária uma forma de criar e destruir processos a fim de ter-se o mínimo de memória em uso a qualquer dado momento.

No GNU/Linux, tem-se a chamada de sistema *Fork* que cria um processo a partir de outro. Pensando-se que um processo-pai pode criar “N” processos-filho, então se tem no

Linux o que se chama de *hierarquia de processos*, pois, partindo-se da explicação acima é possível desenhar o relacionamento dos processos como uma árvore, por exemplo.

O primeiro processo criado no Linux para dar origem a todos os outros é o *init*, que gera os terminais de acesso para o usuário no boot do SO e quando os terminais são logados por usuários, estes por sua vez criam um processo com o *shell* este, por sua vez, cria outros processos a partir dos comandos de usuário. Por isso, quando se tenta "matar"(com comando *kill*) o processo *init,* o SO não obedece, pois ele é a essência para o funcionamento do sistema operacional Linux.

Como os processos precisam muitas vezes interagir entre si, como por exemplo, processos pai e filho, e também, quando um processo gera uma saída a ser usada por um outro, tem-se a necessidade do controle efetivo dos *estados de processo*.

Segundo (TANENBAUM, 2000), os estados básicos de um processo são: *Pronto,* quando está carregado em memória e já é executável, porém, ainda não ocupa o processador (CPU), pois estará possivelmente participando de uma fila de escalonamento. O escalonador deverá carregá-lo quando chegar a sua vez, ou conforme sua prioridade. *Bloqueado*, quando está aguardando algum evento (por exemplo, uma entrada que ainda não foi fornecida por outro processo ou uma resposta do subsistema de Entrada/Saída), logo, mesmo que exista a CPU ociosa ele não pode ocupá-la enquanto a dependência que ele espera não for resolvida. *Executando*, quando está ocupando o processador e usando de seu quantum de tempo para atingir seus objetivos.

Para implementar os processos, a maioria dos SOs mantém uma *tabela de processos* que consiste em uma *estrutura de dados* que contém um registro para cada processo, nesse registro estão armazenadas todas as informações relevantes que permitem ao processo mudar de estado e logo após continuar sua execução perfeitamente como se nunca tivesse sido interrompido.

2.1.2. Implementação de Threads

Além da implementação de processos, os sistemas operacionais modernos estão também usando *threads*.

Threads são implementados nos SOs modernos que aceitam que um mesmo processo contenha múltiplos fluxos de execução, isto é, ele tem múltiplos contadores de programa, variáveis e etc. Dessa forma, um mesmo programa poderá ser executado em múltiplas instâncias pelo mesmo processo (no mesmo espaço de endereçamento). Os *threads* também são conhecidos como *processos leves* (TANENBAUM, 2000).

No caso dos threads, quanto ao tratamento da tabela de processos, é necessária a criação de uma tabela de threads, pois os campos da tabela de processos não conseguem suportar os vários threads.

Os threads são úteis em várias ocasiões, por exemplo: para navegadores web (para estabelecerem múltiplas conexões com sites para transferência de recursos sem que isso fique "*pesado*" em relação à quantidade de processos), nos servidores de arquivo (onde os múltiplos threads podem atender as chamadas do servidor enquanto ele está realizando E/S), entre muitas outras implementações possíveis.

O SO não sabe que está processando threads, é o caso das máquinas virtuais Java, assim, quando um thread está para ser bloqueado ele passa a execução para seu sucessor dentro do quantum de CPU daquele processo. Para o SO é um processo qualquer, neste caso, diz-se que os threads foram implementados no espaço de usuário usando uma linguagem de programação com suporte aos threads. Existem muitos pacotes para implementação de threads no espaço de usuário, como por exemplo os seguintes: *POSIX-Threads e Mach C-Threads*.

Alguns SOs modernos sabem quando `threads` estão sendo executados, é o caso dos novos kernels do GNU/Linux, dessa forma, na hora de um `thread` ser sucedido é o SO quem lê a tabela de `threads` do sistema e escolhe o próximo a ocupar a CPU de acordo com os critérios do escalonador.

Uma observação importante apresentada por (TANENBAUM, 2000): a comutação de `threads` é mais rápida quando é feita no espaço de usuário do que quando necessita de chamadas de kernel de sistema operacional, por exemplo, quando um `thread` implementado no espaço de usuário (supondo um SO com suporte a `threads)` é bloqueado, o kernel bloqueia o processo inteiro.

Como surgiu muita polêmica a respeito das afirmações acima, sistemas híbridos foram projetados com objetivos de cobrir as disparidades de funcionamento em sistemas operacionais.

Apenas para uma melhor análise com um exemplo mais real, a chamada *`Fork`* (padrão para criação de um processo-filho a partir do processo-pai no Linux) precisa lidar com as situações: como tratar a possibilidade de o processo-pai tiver múltiplos threads, configurar o sistema para dar autonomia aos processos-filho para também executar múltiplos threads, decidir se um `thread` é bloqueado numa chamada *`READ`* do teclado, se quando uma linha é digitada quem tem acesso (ou uma cópia) a ela é só o processo-pai ou só o processo-filho, entre outros.

Mostrou-se até este ponto que existem vários problemas envolvendo os `threads`, contudo, esses problemas têm soluções já implementadas, e, este estudo é para mostrar a complexidade envolvendo os processos leves, pois, em linguagens como Java isso pode influenciar no comportamento das aplicações desenvolvidas (TANENBAUM, 2000).

2.1.3. Threads Java

Passando-se a temática dos `threads` para a linguagem Java, como já foi comentado, são instanciados em ambiente de programação de usuário, através de formas oferecidas pela linguagem. Na figura 2, pode-se visualizar um computador multiprocessado, a área de memória mostra dois processos em cada CPU , dentro de cada processo há um ou mais `threads` (círculos).

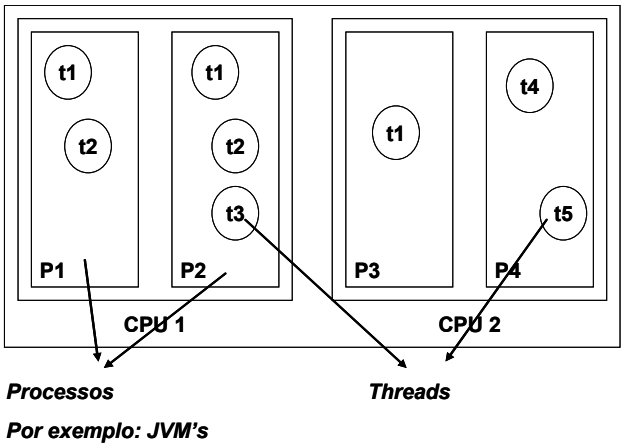


Figura 2: *Implementação de threads Java: (CARPENTER , 2002).*

Os `threads` que ocupam o mesmo processo podem apresentar comportamentos só implementáveis pela comutação de `threads` dentro de um processo comum e pela velocidade de comutação oferecida. A figura 3 , mostra como se dá o comportamento do compartilhamento de dados entre os `threads` em código Java.

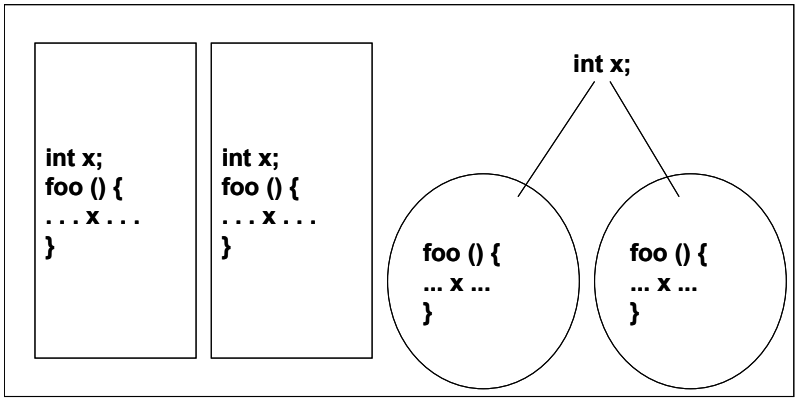


Figura 3 : *Compartilhamento de dados com Threads Java: (CARPENTER , 2002).*

A forma como os `threads` Java foram idealizados para a programação tornam possível uma espécie de computação pseudoparalela (*através da comutação dos* `threads` *Java*) que ocorre dentro de um só processo durante o tempo em que ocupa a CPU.

Se observar-se os `threads`, de modo geral, do ponto de vista de implementação, eles são como contadores de programas e pilhas. Claro que todas as linguagens realizam essa característica de certa forma, porque todas as linguagens têm pelo menos um `thread` quando da execução dos programas(thread principal ou *`main thread`*).

Além dos benefícios da implementação de *`threads`* apresentados, deve-se também atentar às situações onde os `threads` são realmente pertinentes. Primeiramente, a implementação de *múltiplos* `threads` é interessante para o efetivo paralelismo sobre sistemas multiprocessadores, além do mais, oferece uma concorrência de computação e dispositivos de entrada/saída. Pode também expressar facilmente alguns paradigmas como: processamento de eventos e simulações. Apesar de compartilhar algumas características, como já foi mostrado, os `threads` mantém as computações de cada unidade em separado, evitando confusões e mantendo a segurança da plataforma Java, que é umas das diretivas principais desta linguagem de programação.

Apesar dos benefícios do uso de `threads` há algumas limitações, portanto, pode-se igualmente atentar para os motivos ou circunstâncias para não se usar `threads`. Logo, a complexidade da implementação de `threads` em linguagem pode ser o primeiro ponto de partida, pois, não são de forma alguma convencionais como na programação tradicional onde, em geral, não há preocupação com o paralelismo da execução. Os `threads` também necessitam de muitos recursos e usam do hardware com singela abordagem, na qual muitas vezes causam *`overheads`* indesejáveis.

Os `threads` também não são implementadas só na linguagem Java, apenas como citação, pode-se também implementar `threads` nas linguagens: *C, C++, Objective Caml, SmallTalk*, entre outras (TANENBAUM, 2000). É importante que se entenda que em C e C++ a implementação de threads é oferecida como rotinas em bibliotecas especiais na forma de *add-on*, pois não é um recurso nativo da linguagem como acontece com Java.

Em Java, toda aplicação implementada possui pelo menos um `thread` que é lançado pela JVM, o `thread` do método principal (*`main( ) method`*). Porém, o código de *`main( )`* pode gerar outros `threads` de duas formas: *explicitamente* quando usa a classe `Thread` ou *implicitamente* quando usa-se bibliotecas que criam outros `threads` como conseqüência, por exemplo, RMI, AWT/Swing, Applets, entre outros.

Para criar um thread, após a escrita da classe (que deve ser estendida de `Thread` ou implementar a interface `Runnable`), através do método *`start( )`* o thread entra em execução. Na classe `Thread` o método *`run( )`* nativo pode ser sobrescrito.

É interessante entender como é o efeito dos `threads` em código, por isso, a seguir em um trecho de código, pode-se observar, em código Java, o uso de um método chamado *`sleep`* que solicita que o `Thread` (*usado como componente e no contexto de efeito para o próprio programa que a chamou*) entre em estado de dormência para logo após, entrar em estado de *pronto* e imprimir o tempo que foi solicitado para o mesmo ficar inativo (ou em estado de *adormecida*) e também a mensagem que foi dada como entrada (*variável `msg`*).

```
while (true)
{
            System.out.println("Alarm> ");
      // read user input
            String line = b.readLine();
            parseInput (line); //sets timeout
            //wait (in secs)
            try  {
            Thread.sleep (timeout * 1000);
            } catch (InterruptedException e) {   }
            System.out.println ("(" + timeout + ") " + msg);
}
```

No próximo código, é criada uma classe devidamente estendida da classe `Thread`, e tanto a variável *`msg`* quanto a variável *`timeout`* são atributos internos e privativos desta classe denominada *`AlarmThread`*. No método construtor, a classe utiliza os dados de tempo e mensagem para usar internamente, e na sobrescrita do método *`run`* é feita então a operação de *`sleep`* para o `thread`, e, posteriormente é impressa uma mensagem no terminal. Veja que o efeito final dos dois algoritmos até agora apresentados é o mesmo. Contudo, a implementação da classe *`AlarmThread`* fornece uma reusabilidade maior e também um ganho em escrita de código com relação à primeira forma apresentada.

```
public class AlarmThread extends Thread {
      private String msg = null;
      private int timeout = 0;

      public AlarmThread (String msg, int time) {
            this.msg = msg;
            this.timeout = time;
      }

      Public void run() {
            try {
                  Thread.sleep (timeout * 1000);
            } catch (InterruptedException e) {   }
            System.out.println ("(" + timeout + ") " + msg);
      }
}
```

Além da forma de implementação através da criação de classes estendidas de outras, pode-se usar a interface `Runnable`, que provê uma forma de se programar as particularidades de `Thread` para a classe podendo-se estendê-la de uma outra forma. Veja no código a seguir, um exemplo de como ficaria o exemplo anterior usando-se a interface `Runnable`.

```
public  class AlarmRunnable implements Runnable {
      private String msg = null;
      private int timeout = 0;
```

```
        public AlarmThread (String msg, int time) {
              this.msg = msg;
              this.timeout = time;
        }

        Public void run() {
              try {
                    Thread.sleep (timeout * 1000);
              } catch (InterruptedException e) {    }
              System.out.println ("(" + timeout + ") " + msg);
          }
}
```

Na figura 4 é mostrado como se comporta o hardware quando da alocação de dois processadores distintos para os threads. Neste primeiro caso, pode-se ver que há somente um processo por CPU.

Um processo por CPU

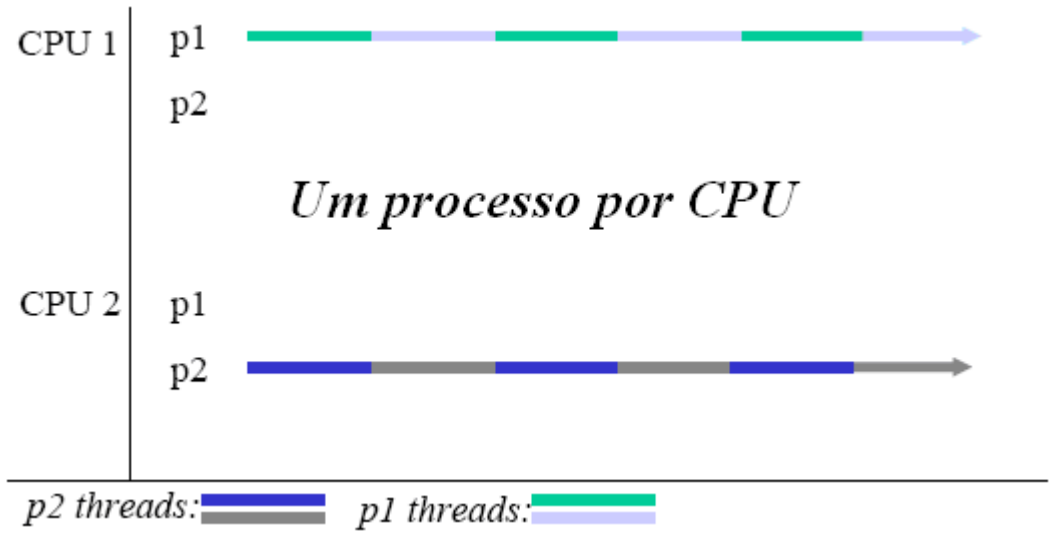


Figura 4 : *Threads rodando em CPU com um processo para cada dois threads: (CARPENTER , 2002).*

Não seria interessante mostrar a figura anterior e não citar também como fica o esquema de funcionamento e comutação de `threads` e processos quando da alocação dos processadores para mais de um processo. Sendo assim, se dispõe, na figura 5 , de uma visão detalhada desse funcionamento.

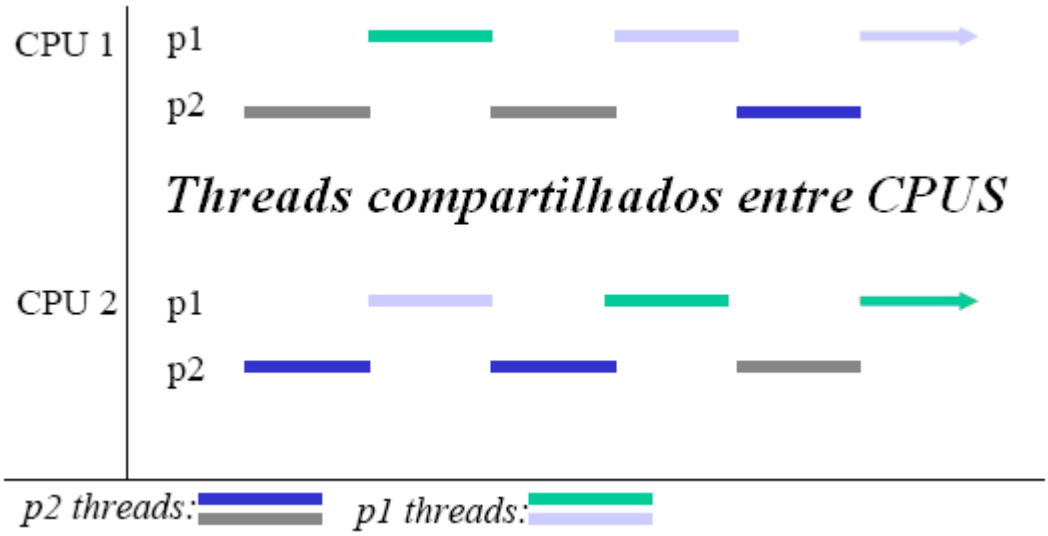


Figura 5: *Threads compartilham as CPUs estando em processos diferentes: (CARPENTER , 2002).*

Ambos os esquemas de execução e alocação das CPUs supracitados dependem do algoritmo de escalonamento (ou agendador). O modo de escalonamento dos `threads` e

processos que eles envolvem podem acarretar algumas conseqüências positivas, tais como: diferentes `threads` da mesma aplicação podem estar rodando ao mesmo tempo em processadores diferentes. `Threads` podem sofrer preempção a qualquer momento, por exemplo, para dar lugar a outros `threads` de maior prioridade.

2.1.4. Condições de Disputa

Por vezes, a comunicação interprocessos, seja ela planejada ou não, é necessária, por isso, são mostrados alguns fatores relacionados com processos no que diz respeito a: como um processo pode passar informações a outro e como evitar que dois processos não interfiram um na execução do outro quando envolvidos em atividades críticas.

As *condições de disputa* em processos ocorrem porque eles normalmente compartilham certas áreas de memória ou mesmos arquivos. As diversas formas de condição de disputa, bem como os algoritmos propostos para resolução desse problema pode-se encontrar no livro de (TANENBAUM, 2000). A maneira eficiente de se evitar as condições de corrida é fazer com que somente *um processo leia e grave os dados compartilhados ao mesmo tempo*. No vocabulário técnico, tal comportamento em SO é chamado de *Exclusão Mútua*.

Na definição mais precisa de *exclusão mútua* se coloca que: é uma maneira de se certificar de que se um processo está utilizando um arquivo ou variável compartilhados, os outros processos serão impedidos de fazer a mesma coisa. Essa definição é satisfatória para que a confiabilidade do SO seja considerada perfeita.

Para explicar as seções críticas usa-se o exemplo: imagine que um processo está realizando várias computações de ordem interna de seu algoritmo, essas computações usam áreas de memória privativas do processo, que portanto, não oferecem riscos para criar condições de disputa. A seguir, este processo executa algumas instruções que necessitam de

dados de uma área de memória compartilhada, essas instruções ou essa parte do programa são conhecidas como *região crítica* ou seção crítica.

A solução perfeita envolvendo as condições de disputa e o controle efetivo das seções críticas nos programas poderia ser modelado, segundo (TANENBAUM, 2000), de acordo com as seguintes regras:

1. Nenhum dos dois processos (ou mais processos) pode estar dentro de sua região crítica ao mesmo tempo;

2. Nenhuma suposição pode ser feito sobre as velocidades de execução ou sobre o número de CPUS;

3. Nenhum processo que está executando fora de sua região crítica pode causar o bloqueio de um outro;

4. Nenhum processo deve esperar eternamente para entrar em sua região crítica;

Obedecendo a estas quatro diretrizes principais, afirma-se que é garantida a segurança de um sistema operacional.

2.1.5. Compilação Just-In-Time

Alguns autores (CRAMER et al, 1997) afirmam que a linguagem de programação Java promete, em sua *filosofia,* portabilidade que significa que um programa Java uma vez compilado em forma de bytecodes pode ser executado em qualquer plataforma de hardware. Se essa afirmativa for observada com cautela, o que se pode concluir é que, em verdade, o que se oferece é multiplataforma através da JVM (*Java Virtual Machine*), isto é, instalando-se a JVM na máquina pode-se executar os bytecodes, caso contrário, a portabilidade não é realizada. As primeiras implementações de Java conduzidas sobre metodologia de interpretação, apresentavam uma performance insuficiente comparada a programas compilados. Compilar os programas Java para as instruções nativas da máquina provê uma performance muito maior. Como o processo de compilação tradicional em programas Java

poderia ferir os princípios de portabilidade e segurança, uma outra aproximação foi necessária.

Assim, para cobrir a carência da linguagem Java, foram desenvolvidas técnicas de compilação em tempo de execução (*ou JIT – Just In Time Compiling*). É claro que, a cada nova versão das máquinas virtuais, a performance tende a ser cada vez maior pela miscigenação de várias técnicas que revesam entre a interpretação e compilação JIT. O que se pretende neste tópico é a análise de algumas técnicas de implementação de JIT.

Segundo informações dadas por (CRAMER et al, 1997), a JVM provê um *framework* de tempo de execução bem definido, no qual os programas Java são compilados por uma arquitetura com um dado conjunto de instruções. Os programas então são distribuídos nesta forma abstrata (no caso, refere-se aos bytecodes), separada dos detalhes de qualquer outra arquitetura de computador. Executar programas Java envolve também a interpretação de instruções da JVM, compilando-as em instruções do hardware subjacente, ou diretamente executando-as na implementação de hardware da JVM.

O funcionamento da JVM, segundo (CRAMER et al, 1997), assim, é descrito como uma *máquina orientada a pilha* (`stack machine`), cada instrução executa seus operandos do topo da pilha, consumindo aqueles valores e opcionalmente substituindo-os com um resultado. As instruções codificam-se numa forma compacta de tamanho variável, com a *mais curta* das instruções ocupando 1 byte e as demais instruções ocupando entre 1 e 3 bytes. A esta forma de codificação dá-se o nome de *bytecodes***.**

Um compilador Java *fonte-para-bytecode*, tal como o `javac` do Sun J2SDK, compila as classes que constituem um programa Java. O compilador traduz os métodos em cada classe fonte para instruções de bytecode e coloca todos os bytecodes de uma classe juntos num arquivo de classe (`*.class`).

Para executar um programa Java, a JVM carrega a classe configurando o *ponto de entrada do programa*, e a execução começa. O programa pode referenciar outras classes, as quais são carregadas também. Para manter a integridade do modelo de execução Java, a JVM checa se suas variedades de restrições semânticas são satisfeitas, tanto com um arquivo de classe quanto com muitos arquivos de classe. Por exemplo, o bytecode de uma classe não pode acessar um campo definido em outra classe ao menos que seja explicitamente permitido pela especificação de acesso na definição do campo.

Como parte da execução de um programa, a JVM precisa prover vários serviços. Ela precisa gerenciar memória, aceitando programas para criar objetos e "reformar" objetos uma vez que eles não são mais requeridos (isso é feito com um processo 'thread' conhecido como *garbage collection*).

Um aspecto interessante também mencionado por (CRAMER et al, 1997) é que Java também admite interoperabilidade com código de máquina criado a partir de outras linguagens fonte (como por exemplo a linguagem C), que é encapsulado para parecer com um programa Java com métodos, inclusive. Por essa razão, a JVM precisa também mediar entre métodos Java e métodos nativos, convertendo representações de dados e gerenciando o fluxo de controle *de* e *para* os métodos nativos.

Muitas vezes, pode-se questionar quais os pros de se usar JIT em tempo de execução. Primeiramente, interpretar bytecodes é um processo demasiado lento. Além do mais, a JVM é uma arquitetura de sistema computacional implementada em software, e, como toda arquitetura realiza o ciclo de busca, decodificação, execução e despacho, dessa forma, programas compilados para uma arquitetura de hardware real não tem o *overhead* da tradução, como acontece na interpretação.

Para se confirmar a primeira afirmação, é ideal partir-se de um exemplo. No trecho de código a seguir, num exemplo de (CRAMER et al, 1997), são ilustrados os seis bytecodes que compõem a expressão *x = y + (2 * z)*.

```
 iload z
iconst 2
 iload y
  imul
  iadd
istore x
```

Na execução em pilha (já que JVM é uma *stack machine*), os bytecodes são resolvidos tendo-se por base a orientação de cima para baixo (isto é, desempilhando as instruções). Primeiro carregam-se a variável z, a constante inteira 2 e a variável y, respectivamente. A seguir é acionada uma instrução de multiplicação que serve para os dois primeiros operandos no topo da pilha. Então é ativada uma instrução de soma(a partir do topo) que usa como operandos o resultado da multiplicação anterior e o valor da variável y, o resultado da soma é armazenado então na variável x.

A partir de uma breve análise do exemplo propõe-se uma solução descrita no artigo sobre *compilação Just In Time* (CRAMER et al, 1997). Tal solução significa possuir a JVM em hardware para excluir o overhead das instruções de bytecode, contudo, tal atitude fere a diretriz de portabilidade da linguagem, pois, não resolve o problema dos processadores já existentes, que só interpretam um conjunto proprietário de instruções promovidas pelos seus fabricantes.

Em verdade, a compilação JIT é uma técnica adotada para o aumento de performance, baseada no tempo de execução insatisfatório dos primeiros interpretadores Java. Além do mais, a compilação em tempo de execução foi um alternativa encontrada para manter a segurança e portabilidade já frisados nas diretivas essenciais dos bytecodes dessa linguagem.

Há uma limitação interessante de se citar quando está se tratando da teoria conceitual de compilação JIT, é fato que o tempo despendido para compilar um método, por exemplo, é quase o mesmo gasto para interpretar seus bytecodes.

O motivo do nome da compilação JIT, é que os métodos só são compilados quando são efetivamente *chamados*, assim, não se desperdiça tempo de CPU para compilar todos os métodos de uma classe, quando nem todos poderão vir a ser executados.

A tecnologia atual das JVMs trouxe para a realidade um modo misto de execução Java, onde compilação e interpretação convivem pacificamente. A JVM é incumbida de decidir, em tempo de execução, quais partes do código do programa que deverão ser interpretados e quais deverão ser compiladas para a linguagem nativa da máquina. É importante abordar que, numa situação normal, a JVM geralmente compila os códigos mais frequentemente executados ou executados por diversas vezes, como os ninhos de laço. Os trechos de programa que possuem execução única geralmente podem ser interpretados enquanto a estratégia da compilação JIT é implementada pela JVM em background.

Criar um compilador, apesar de ser uma tarefa árdua, não é de nenhuma forma um conceito novo ou desconhecido em computação. Mas, o contexto em que esse compilador irá atuar é completamente distinto daquele em que a compilação tradicional trabalha, isto é, na chamada *compilação estática.*

O compilador JIT é denominado um *compilador dinâmico* porque entra em ação em tempo de execução e somente quando levantada uma necessidade. Olhando agora sob uma visão mais detalhada, quando o compilador dinâmico entra em ação precisa compilar para código nativo os bytecodes determinados pela JVM da forma mais rápida possível, além disso, o compilador dinâmico também tem de possuir performance suficiente, já que o usuário está esperando uma resposta aceitável da JVM, e ainda, ele não pode ficar realizando otimizações prolongadas (como é feito na compilação estática) para entregar o código

perfeitamente otimizado para aquela arquitetura, assim, considera-se que ele é sempre alvo de pressão por parte de todo o sistema.

(CRAMER et al, 1997) cita que o compilador JIT deve seguir algumas especificações especiais para executar seu trabalho sem violar as *leis de linguagem*. A seguir são abordadas algumas dessas especificações. Tomando-se por ponto de partida a minimização do `overhead` de compilação, evitar `overhead` desnecessário é fator crucial para uma compilação rápida. Em muitos casos, os compiladores geram representações intermediárias (ou *IR – intermediate representation*) como padrão para facilitar o trabalho. Os bytecodes se aproximam muito bem dos códigos intermediários, e, apesar de não serem exatamente para isso que foram criados, com pouco trabalho de implementação pode-se usá-los para tal propósito.

A tecnologia de bytecodes Java retém muitas informações do código-fonte do arquivo origem. Qualidade que é nitidamente percebida quando da *descompilação* (com programas parecidos com *disassemblers*) de arquivos de classe Java. Também é notória, por este motivo, a carência por ferramentas que consigam ocultar, cada vez mais, estes tipos de informação. A JVM contribui com alguns itens, por exemplo, como ela não possui nenhuma designação para variáveis booleanas, portanto, torna-se mais subjetivo distinguir o que é inteiro do que é booleano.

2.1.6. MPI tradicional

O MPI (*Message Passing Interface*) é um paradigma de programação usado amplamente sobre computadores paralelos, afirmam (SNIR et al, 1999), especialmente nos SPCs (*Scalable Parallel Computers*) com memória distribuída e sobre as Redes de Workstations (*NOWs – Network of Workstations*). Embora existam muitas variações, o conceito básico de processos comunicando-se através de mensagens foi muito usado por algum tempo. Quando o MPI Fórum iniciou suas atividades em 1992, bibliotecas proprietárias

de MPI estavam disponíveis em muitos sistemas SPC e estavam sendo usadas para desenvolver aplicações paralelas de grande porte. A portabilidade do código foi prejudicada pelas diferenças entre estas bibliotecas. Muitas aplicações de domínio público demonstram que o *sistema MPI* pode ser *eficiente* e *portável*.

Assim, percebeu-se que ambas, a sintaxe e semântica do *padrão Core* de rotinas de biblioteca poderia ser usado para uma ampla gama de usuários e implementações eficientes sobre uma ampla gama de computadores. Esse esforço foi empreendido de 1992 a 1994 pelo MPI Fórum, um grupo de mais de 80 pessoas de 40 organizações, representando fabricantes de sistemas paralelos, usuários industriais, indústria, laboratórios de pesquisa e universidades. Os desenvolvedores do MPI buscaram fazer uso das características mais atraentes de um dado número de sistemas de passagem de mensagens existentes, alternativa melhor do que selecionar-se um deles e adotá-lo como um padrão.

A passagem de mensagem é um método de comunicação interprocesso. Utiliza-se de duas primitivas chamadas SEND e RECEIVE, que, como os semáforos, e, ao contrário dos monitores, são chamadas de sistema ao invés de construções de linguagem.

Podem ser colocados como procedimentos de biblioteca, na seguinte formal geral:

```
send (destino, &mensagem);
e
receive (origem, &mensagem);
```

A primeira chamada envia mensagem a um destino, a segunda recebe uma mensagem de um destinatário identificado ou qualquer, caso a aplicação não se importe com a origem.

Se nenhuma mensagem está disponível, o processo pode se bloquear, ou, alternativamente, poderia retornar imediatamente um código de erro.

As mensagens *possuem questões de projeto* que *não existem com semáforos e nem monitores*. Por exemplo, elas podem se perder quando do envio via rede. Para isso, uma

técnica é solicitar ao destinatário que envie de volta ao remetente uma mensagem especial de reconhecimento. Contudo, se o remetente não receber a mensagem de reconhecimento depois de um dado intervalo de tempo, a mensagem principal é reenviada. Há também a questão da nomeação individual de cada processo para que chamadas SEND e RECEIVE não sejam enviadas a processos ambíguos em identificação. A autenticação é uma questão de segurança nas mensagens, ou seja, é importante saber se o destinatário ou remetente são *confiáveis*. O desempenho, pelo princípio de localidade, pode-se saber que, se os dois processos que estão trocando mensagens estiverem na mesma CPU, *a técnica de mensagens é muito mais lenta do que os semáforos e monitores* (TANENBAUM, 2000).

É sugerido *limitar-se o tamanho das mensagens* (também conhecido como envio de mensagens discretas) de acordo com o número de registradores da CPU para que a cópia das mensagens seja mais direta e mais rápida provendo um desempenho satisfatório.

O código em linguagem C a seguir, ilustra uma implementação do problema do produtor e do consumidor que salienta o uso das chamadas SEND e RECEIVE numa tentativa de implementação de *passagem de mensagens*.

```
#define N 100

void producer(void)
{
      int item;
      message m;

      while(TRUE) {
            produce_item(&item);
            receive(consumer, &m);
            build_message(&m, item);
            send(consumer, &m);
      }
}

void consumer(void)
{
      int item, i;
      message m;

      for (i=0 ; i < N ; i++ ) send(producer, &m);
```

```
        while (TRUE) {
              receive(producer, &m);
              extract_item(&m, &item);
              send(producer, &m);
              consume_item(item);
        }
}
```

O problema dos *produtores e consumidores* com CPUs distintas exemplifica o uso de memória não-compartilhada e de passagem de mensagem.

O consumidor gera um número N de mensagens vazias e as manda para o produtor. O produtor vai recebendo as mensagens vazias e retornando mensagens cheias, dessa forma a quantidade de mensagem entre os dois é sempre constante. As mensagens são armazenadas numa porção de memória conhecida antes de serem enviadas.

Sobre o endereçamento das mensagens: pode-se dar um endereço único a cada processo e enviar as mensagens a este endereço. Outra forma usual é o formato *mailbox* (caixa de correio) que é muito parecido com as caixas de correio que se conhece. Quando as *mailboxes* são usadas, os parâmetros de SEND e RECEIVE são as *mailboxes* em vez de endereços de processos. Como exemplo, a comunicação interprocessos do Linux é via pipes, que são, efetivamente, *mailboxes* (TANENBAUM, 2000).

2.1.7. MPIjava

Quando se aborda o *MPIjava*, segundo (BULL, 2001), cada processo executa sua cópia do código, mas o *objeto injetor* (que controla o conjunto) contém o controle do conjunto de seus *objetos filhos*. Um padrão de decomposição unidimensional é usado, com cada processo possuindo seu próprio *subconjunto de cópias de linhas* do programa que está sendo executado. O método `main` contém as chamadas `initialize` e `finalize`, assim, o MPI determina o número de processos e a ordem dos processos a serem executados. Essa forma de comunicação dá lugar a um novo método, chamado `haloexchange`, o qual faz cópias de

swap das linhas de código entre os processos vizinhos (*neighbouring processes*). Há também uma chamada para um redutor global para computação residual. Um arranjo de buffer é usado e a interface não aceita tipos primitivos de dados para serem usados em operações com MPI.

As três versões de Java Paralelo na pesquisa de (BULL, 2001) foram executadas num *Sun HPC 6500 SMP*, com processadores de 400Mhz , cada um tendo 8Mb de Cache L2. A JVM usada foi *a Sun Solaris Production JDK, na versão 1.2.1_04*, e a versão do *MPIjava* usada foi o *MPICH Version 1.1.2*. Para comparação, uma versão FORTRAN do código (usando diretivas OpenMP com o compilador KAI guidef90 na versão 3.7) também foi testada. O código-fonte da implementação usada para os testes encontra-se nos apêndices do trabalho de (BULL, 2001).

A tabela 1, é uma planilha elaborada por (BULL, 2001) em seu trabalho, mostra o tempo de execução e performance para 100 iterações sobre uma matriz 1000x1000.

Tabela 1 : *comparação de tempo de execução (em segundos) e performance (em MFlop/s) a partir de uma aplicação de demonstração no Sun HPC 6500: (BULL, 2001).*

	Fortran/OpenMP		Java/MPI		Java/Threads		Java/OpenMP	
No. Processos	Tempo	Perform.	Tempo	Perform.	Tempo	Perform.	Tempo	Perform.
1	47.5	92	97.2	45	97.9	44	98.4	44
2	24.2	181	38.2	115	48.7	90	48.7	90
4	12.6	348	19.3	227	24.8	176	24.9	176
8	6.4	684	10.3	424	12.1	362	12.3	356
12	4.3	1010	9.0	486	7.8	558	8.2	536
16	3.2	1360	8.3	528	5.8	752	6.1	716

Na figura 6, é ilustrada a performance das versões dos códigos comparados ao *speedup ideal* calculado a partir da performance do Fortran seqüencial e versões Java usadas na pesquisa de (BULL, 2001).

Das versões de Java utilizadas, as versões com `threads` tiveram a melhor escalabilidade, e é só um fator de duas vezes mais lento do que Fortran com a versão OpenMP. A versão *MPIjava* mostrou alguma *superlinearidade* sobre pequeno número de processadores (presumivelmente provocado pelo efeito do cache) mas sua performance diminuiu com mais de oito processadores.

A versão JOMP exibe uma baixa performance comparado ao código na versão com `threads`. É importante notar que a versão de JOMP requer significativamente menos esforço do programador do que as outras versões de MPI, sendo, o que os dados da tabela e os gráficos das figuras refletem é que quanto maior a simplicidade da linguagem para o programador, há o custo da perda de performance, ao passo que, quanto maior a complexidade de programação ao programador, maiores os ganhos nas aplicações (pois, controla-se de forma mais efetiva o paralelismo).

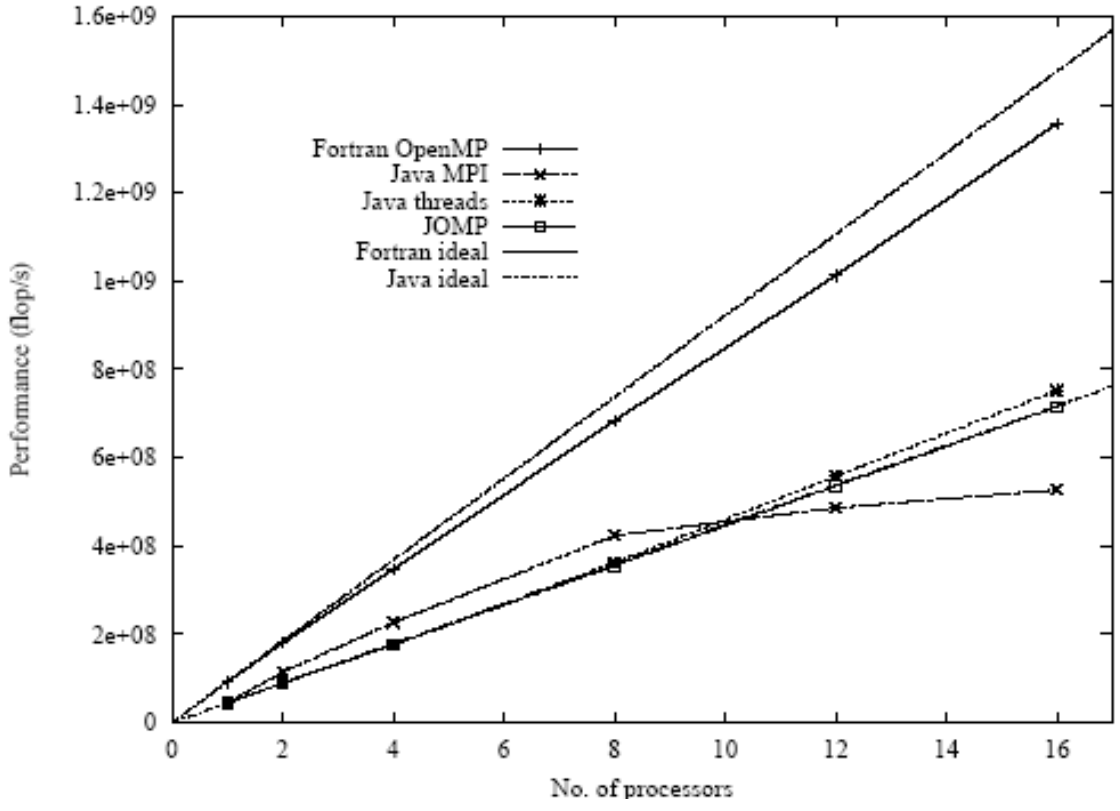


Figura 6 : *performance dos diferentes mecanismos de paralelização: (BULL, 2001).*

2.1.8. PVM – Parallel Virtual Machine tradicional

O sistema PVM (*Parallel Virtual Machine*) tradicional consiste de um *daemon* (*ou* `pvmd`), o *console de processo* e a interface das rotinas de biblioteca. Um processo *daemon* reside em cada máquina que constitui a VM. *Daemons* são iniciados quando o usuário inicia a PVM especificando um *hostfile*, ou adicionando hosts usando o *PVM console*.

O *PVM console* é a interface para os usuários interagirem com o ambiente PVM. O console pode ser iniciado sobre qualquer máquina da VM. Usando-se o console, um usuário pode monitorar o status do ambiente PVM ou reconfigurá-lo caso assim deseje. O componente final para a estrutura, a biblioteca de interface PVM, tem rotinas para passagem de mensagem, gerando processos de aplicação e a coordenação destes processos.

(LEE et al, 1999) faz uma discussão mais trabalhada a respeito de PVM e JPVM que será abordada no próximo tópico quando for detalhada a teoria que envolve a *Java Parallel Virtual Machine*.

2.1.9. JPVM – Java Parallel Virtual Machine

O ***PVM for Java*** ou ***JPVM*** é um ambiente de programação paralelo que provê um conjunto de uma *Máquina Virtual Paralela* com bibliotecas de classe desenvolvidas usando Java. O JPVM herda os atributos de Java, especialmente o suporte Java de sistemas heterogêneos. Porém, também há a desvantagem da baixa performance quando comparada a códigos nativos. A performance de Java tem sido consideravelmente aperfeiçoada com a introdução da compilação JIT, verificada através do uso de alguns algoritmos de processamento paralelo como benchmark. Tal prática também destaca algumas das áreas que limitam o uso de Java em processamento paralelo sobre sistemas distribuídos.

Segundo (LEE et al, 1999), atualmente há muitos tipos de bibliotecas para *implementação da passagem de mensagem*. Entre elas estão: *Message Passing Interface (MPI), Parallel Virtual Machine (PVM), Express, P4 e PARMACS*. Estes ambientes habilitam uma coleção de computadores heterogêneos a serem usados como um coerente e flexível recurso de computação paralela.

Apesar da diversidade de ambientes de programação baseados em Java, o autor original do JPVM foi o pesquisador (FERRARI, 2005), logo, o JPVM para Java (ou seja, o sistema JPVM) é uma implementação de uma biblioteca de classes de objeto do tipo PVM na linguagem de programação Java a ser usada com uma JVM comum instalada em cada host.

O objetivo principal de (LEE et al, 1999) em seu trabalho foi o de traçar um comparativo, usando-se de *benchmarks* para aferir a performance de JPVM e comparar os resultados com uma implementação nativa em C da PVM.

Para entender melhor a relevância para (LEE et al, 1999) a respeito da comparação entre PVM e JPVM, é interessante observar algumas idéias a respeito da *arquitetura de PVM* e *JPVM*.

(LEE et al, 1999) parte do ponto inicial que ambas PVM e JPVM são baseadas na idéia de uma *Virtual Machine (VM)*, pela qual uma rede de computadores é configurada como um ambiente de computação paralelo. *A arquitetura física subjacente é transparente ao usuário*. *Tarefas* são geradas como processos residindo sobre diferentes máquinas, mas, trabalham coletivamente como uma simples aplicação. A natureza distribuída do ambiente da VM requer que estes processos transfiram dados e mensagens de sincronização via rede. As *comunicações entre os processos são explicitamente especificadas no código de usuário*, mas o usuário não precisa, usualmente, saber onde estes processos concebidos irão residir quando da sua execução. Similarmente, o usuário não precisa se preocupar com os diferentes tipos de

representação de dados *inteiros* e *ponto flutuante* usados pelas distintas máquinas, pois a VM irá automaticamente executar as conversões de dados quando necessárias.

A JPVM tem uma arquitetura similar à da PVM, tal modelo conceitual de arquitetura pode ser visto na figura 7.

De modo simples, pode-se explicá-la como sendo composta por um trabalho de ***daemons*** (pode-se entendê-los como *processos* que oferecem *serviços*), um console da JPVM que interage diretamente com estes *daemons* e uma interface de funções de biblioteca para o usuário.As comunicações, conforme a representação, são feitas *daemon-a-daemon*, e, de forma direta também de *task-para-task* (*tasks são tarefas)*. Ainda assim, tais características não diminuem a importância da JPVM representada como um console. As *tasks* são implementadas como processos no SO de cada host, mas, a comunicação entre elas é feita através de *TCP sockets* e *threads.*

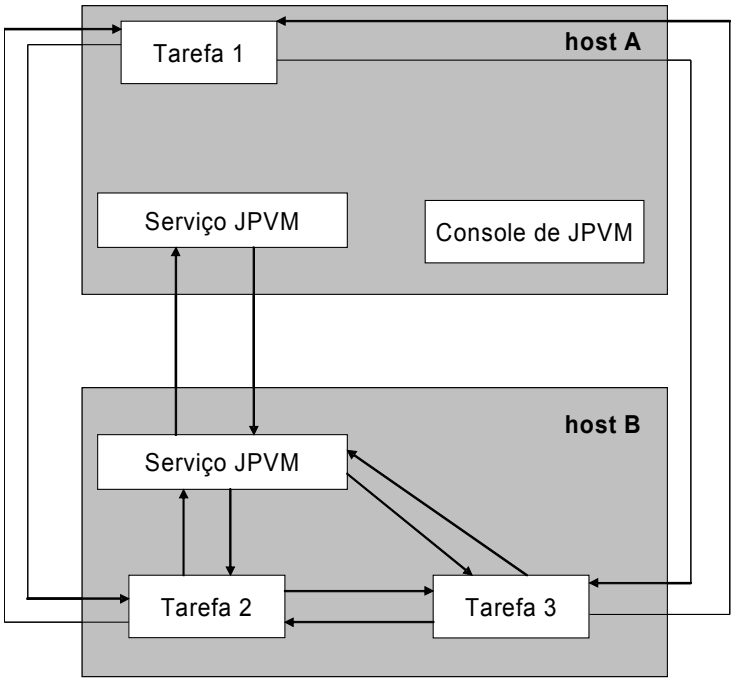


Figura 7: *Arquitetura JPVM: (LEE et al, 1999).*

Existem *diferenças nas formas de comunicação* de PVM e JPVM. No primeiro ambiente, usam-se ambos os protocolos TCP e UDP, já a segunda, é completamente baseada no protocolo TCP. A JPVM também adota um *procedimento de três passos* para uma task

enviar (ou receber) uma mensagem, não há opção para enviar-se os dados usando uma simples chamada.

Alguns experimentos de (LEE et al, 1999), usaram-se os seguintes benchmarks para avaliar a JPVM de FERRARI: *PingPong, EP e Heat.* Cada um dos benchmarks possui uma forma diferente de avaliação porque estimula características diferentes da JPVM. Maiores detalhes sobre o funcionamento metodológico desta pesquisa podem ser obtidos com a leitura aprofundada dos trabalhos de performance do JPVM (LEE et al, 1999).

Para ilustrar-se a pesquisa e demonstrar os resultados atingidos, (LEE et al,1999) criaram alguns gráficos que são aqui ilustrados nas figuras 8 e 9. O primeiro gráfico, representado pela figura 8, mostra os resultados das performances de tempo de execução da JPVM com e sem JIT, também comparada com a PVM(que é uma espécie de linha guia para demonstrar as potencialidades de JPVM) usando-se o *EP Benchmark*.

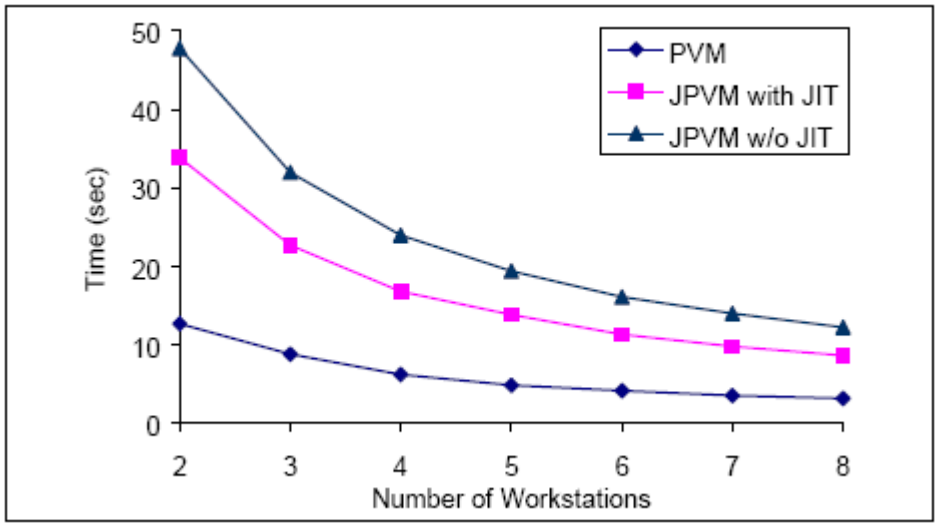


Figura 8 : *Resultados com o EP Benchmark: (LEE et al, 1999).*

O gráfico da figura 8, mostra que a JPVM obteve performance inferior a PVM com e sem JIT para este benchmark, cujo algoritmo gera *números Gaussianos randômicos*.

Nos experimentos realizados com o *Heat Benchmark*, cujo domínio de problema de algoritmo é separar e formar uma malha 2D de pontos discretos com a localização de cada ponto sendo calculada iterativamente, a JPVM com JIT praticamente equiparou-se em performance à PVM, conforme pode-se observar pela figura 9.

Ainda assim, a PVM mostrou performance superior à JPVM com JIT. Tal fato revela que a compilação Just In Time em um ambiente distribuído, cuja comunicação massiva é feita via sockets não produz o desempenho desejado como quando da execução de um programa com JIT em uma máquina *stand alone*.

(LEE et al, 1999) afirmam que, as aferições realizadas com o conjunto de benchmarks definidos na pesquisa permitiram a mensuração e a comparação de características comportamentais em *velocidade de comunicação, latência de comunicação e velocidade de computação* entre JPVM e PVM.

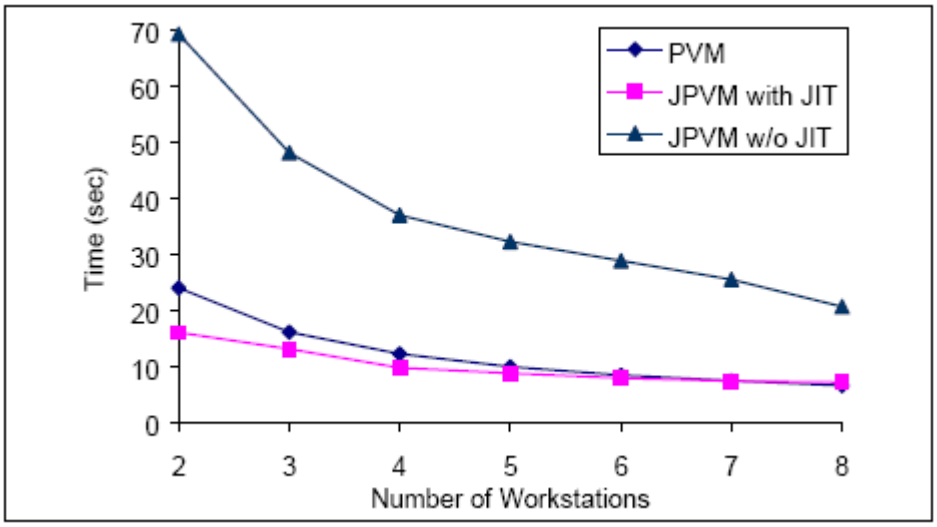


Figura 9 : *Resultados com o Heat Benchmark: (LEE et al, 1999).*

E como conclusão final, apesar dos experimentos terem mostrado, em quase todos os casos, que a PVM ainda possui uma performance superior.

*2.1.10. RMI+ (*Remote Method Invocation Plus*)*

O principal objeto de *RMI+* é permitir que uma aplicação baseada em RMI seja programada de forma desinteressada sobre qualquer condição relacionada sobre *mobilidade específica em tempo de execução* e simplificar as suas customizações. A fim de alcançar este objetivo, (WELLING, 1999) construiu várias aplicações com RMI+ e *testou-as com diversas políticas distintas* (as quais são descritas em detalhes em seu trabalho).

Contudo, para que o entendimento fique ainda mais completo, (WELLING, 1999), durante seu trabalho mostra vários códigos com as políticas para otimizar o tempo de execução da aplicação. A seguir é então mostrada uma espécie de "esqueleto" do que seria o código real do `applet` Hangman implementado no trabalho de (WELLING, 1999).

```
public class Hangman extends Applet {
      GameServer server;
      Game currentGame;
      public void init() {
            server = lookup("HangmanServer");
            newGame();
            // Initialize the user interface.
      }
      public void newGame() {
            // Get a new game from the server.
            currentGame = server.newGame();
            // Start the new game.
            secretWordLength = currentGame.wordLength();
            // Set up the user interface for the new game.
      }
      public void keyPressed(KeyEvent e) {
            if (end of round)
                  newGame();
            Integer[]positions=
currentGame.charPositions(e.getKeyChar());
            // Update the user interface.
      }
```

```
        // Other methods . . .
    }
```

A classe Hangman tem duas implementações distintas, pois a estratégia de projeto montada por (WELLING, 1999) funciona, na essência, pelo modelo Cliente/Servidor.

Com relação às políticas utilizadas para os testes, é bom que se tenha a ciência de que elas são muito importantes no projeto de (WELLING, 1999), pois, inclusive são implementadas como construções de código na linguagem estendida que este pesquisador projetou. A seguir, mostra-se um exemplo de uma classe implementando políticas através da interface *Policy*.

```
public class Cache implements Policy
{
      Control object, cache;
      // Initialization of the policy
      public void init(Control object) {
            this.object = object;
            // Enable use of policy
            object.usePolicy(true);
            // Cache the object
            cache = object.copy();
      }
      // Gets a remote call object
      public RemoteCall newCall(String signature) {
            // Return a special remote call that directs
            // method calls to the cache
            return new LocalCacheCall(cache, signature);
      }
      // Called when no more references
     public void detach(Control object) {
          object.synchronize(cache);
      }
}
```

Alguns trabalhos, (WELLING, 1999) apresentam o RMI+ como sendo um *framework* para introdução de adaptações em aplicações Java baseadas em RMI. Diversas políticas de comportamento adaptativo podem ser definidas com o *framework RMI+*, provendo recursos para a construção de blocos para implementação de aplicações móveis. As

políticas podem ser dinamicamente acopladas a uma referência de objeto remoto a fim de incorporar funcionalidades adaptativas apropriadas.

Uma aplicação pode, consequentemente, ser implementada independente de qualquer técnica relacionada com RMI, além disso, a técnica pode ser adiada até a distribuição ou a execução da aplicação. (WELLING, 1999) executou políticas no estudo de RMI+ e *comportamento adaptável*, entre essas políticas, garantias diferentes para a consistência do cache e uma política para as chamadas a métodos não utilizados. Utilizando estas políticas em aplicações de RMI+, pode-se demonstrar a praticabilidade da arquitetura proposta por (WELLING, 1999) para o *middleware adaptável*, é mostrado também que a flexibilidade adicional que RMI+ fornece foi obtida com pouca perda em desempenho. Pode-se, conseqüentemente, concluir que o desacoplamento do comportamento adaptável em relação à funcionalidade da aplicação e do *middleware* subjacente simplifica não somente aplicações de computação móveis adaptáveis no que diz respeito à sua construção, mas também permite ajustes conforme as *circunstâncias ambientais de execução*.

O trabalho de (WELLING, 1999) é relevante, pois, através da estratégia de uma linguagem Java estendida o autor se propôs a determinar uma nova metodologia de trabalho para o RMI com o intuito de prover melhor performance para aplicações móveis em ambientes voláteis. Tal missão exigiu que a pesquisa implementasse características de *design adaptável*. Tornar a comunicação via RMI mais eficiente com certeza é caminho certo para a melhora do desempenho dos softwares que, como os applets de (WELLING, 1999), trabalhem em ambientes de rede de escopo incerto e com uma metodologia de funcionamento baseada no modelo Cliente/Servidor.

2.1.11. JavaRMI

(BULL, 2001) classifica JavaRMI (*ou Java Remote Method Invocation*) como sendo uma parte da API Java a qual suporta invocação remota de métodos sobre objetos em

máquinas virtuais Java diferentes, isto é, cada uma estando num *host* distinto em uma rede de computadores.

Isto permite a execução de aplicações distribuídas usando o modelo de objeto Java. O RMI é primeiramente intencionado para uso em um paradigma Cliente/Servidor. Uma *aplicação servidor* cria um número de objetos; uma *aplicação cliente* invoca então métodos nestes objetos remotamente. Este padrão não é comumente usado em aplicações científicas de alta performance (exceto talvez nos casos onde há um paralelismo trivial de tarefa). A maioria de técnicas de decomposição requer a passagem dos dados entre processos, que são processados então localmente, melhor que invocar a computação sobre dados remotos.

Além disso, (BULL, 2001) afirma que o RMI sofre penalidades de desempenho significativas, tendo por resultado as latências que são certamente inaceitáveis em um contexto de alta performance. No projeto interno de Java RMI é intencional ter um protocolo independente abstrato subjacente à camada de transporte com código específico de protocolo nos sub-pacotes que facilitam assim a implementação de transportes alternativos. Tal como no release da Sun Microsystems do Java 2 SDK 1.2 RMI , *somente* o transporte baseado em protocolo TCP é implementado.

Entretanto, a provisão para o uso de outros protocolos baseados em *socket* é incluída na API do RMI. Devido a esta dependência sobre o TCP/IP, a implementação atual (Sun Microsystems Java 2 SDK 1.5.0) do Java RMI ainda é considerada desapropriada para o uso em um *sistema paralelo acoplado e fechado*. Contudo, em seu trabalho, (BULL, 2001) revela que na teoria; a estrutura interna do Java RMI deveria aceitar camadas de transporte alternativas e subjacentes, baseadas em relações mais apropriadas tais como MPI ou VIA. (BULL, 2001) menciona que as pesquisas atuais têm investigado a praticabilidade de mover o RMI para um transporte baseado em MPI usando o *MPIjava*.

No código-fonte do RMI release 1.2, há muitas suposições sobre o TCP/IP feitas nos pacotes chamados de “transporte independente” que fazem tal adaptação impraticável. Numa comunicação com os colaboradores do RMI, (BULL, 2001) alega que se compreende que os *releases futuros do JavaRMI teriam estas limitações removidas*. Isto deve fazer implementações de transporte alternativo em RMI mais interessantes. Entretanto, o desempenho do *Sun Microsystems RMI* não fornece um transporte de alta performance, isso é evidente pelo *overhead* que se pode constatar quando usa-se o mesmo numa dada camada qualquer de software.

Ambos *JavaRMI e a Serialização de objetos em Java* (o processo requerido para transformar objeto, e, recursivamente, todos os objetos que se referem em um *byte stream* apropriado para comunicação) como implementados atualmente pela Sun Mycrosystems, são significativamente e computacionalmente lentos. As implementações alternativas de terceiros de ambas as técnicas, tais como a universidade de Karlsruhe com o *KaRMI e a UKA-Serialization* mostraram melhorias consideráveis na eficiência, mas no uso do RMI é sempre provável acarretar custos extras significativos comparados ao uso de um mecanismo de transporte mais direto. Dessa forma, (BULL, 2001) conclui que usar o RMI simplesmente para passar dados entre *VMs múltiplas* requer também a criação de *threads adicionais* para controlar pedidos remotos para dados, ocorrendo *switching de contexto* e os *overheads de sincronização*.

O natural bloqueio implícito de métodos remotos torna as *técnicas de ocultação da latência* mais difíceis de programar. Parece, conseqüentemente, improvável que o RMI provará ser uma *escolha popular* para aplicações paralelas no futuro. Dessa forma BULL (2001) acha importante notar-se alguns outros trabalhos realizados como na Universidade de Karlsruhe, que para fornecer um alto nível de interface às aplicações distribuídas ou paralelas usaram um mecanismo de RMI especial denominado *JavaParty*. Este, por sua vez, consiste

em uma *extensão da linguagem Java* (*na verdade, um qualificador remoto para objetos*), um *compilador* para traduzir a *linguagem aumentada* para Java com as chamadas de RMI, e um *Runtime System* que controle a distribuição dos objetos remotos.

2.1.12. Conclusões sobre Máquinas Virtuais e Mecanismos de Paralelização

De acordo com a própria história, a tecnologia das máquinas virtuais, ao contrário do que se pensa, não é uma tecnologia nova, em verdade, surgiu na época do sistema operacional VM/370. Dessa forma, fica claro que a intenção de emular o hardware em software, com objetivo de obter alta performance, respostas mais rápidas, e ainda, diminuir a complexidade para a programação de algoritmos, é uma técnica há muito tempo criada. Atualmente é que a tecnologia de máquinas virtuais tornou-se novamente o ápice da tecnologia de linguagens de programação, mas o objetivo já é outro, a busca da multiplataforma (mais divulgada com o conceito de portabilidade).

Gerir a execução dos programas da forma que é tradicionalmente feito, isto é, através da gerência de processos foi uma alternativa interessante e organizada que os projetistas de sistemas operacionais encontraram, todavia, tal alternativa torna difícil a implementação de um paralelismo viável, portável e com desempenho aceitável, pois, a comunicação interprocessos funciona de forma mais efetiva quando o sistema operacional executa em uma mesma máquina. Já num ambiente paralelo, e, também distribuído, as problemáticas são outras, mesmo assim, pode-se contornar os problemas que cercam a gerência de processos e procurar-se uma forma de tratar o paralelismo e distribuição das tarefas em nível de processos, evitando, assim, a necessidade do programador em alto nível ter de se preocupar em projetar algoritmos pensando na forma mais correta de paralelizar e distribuir seu programa.

Os threads são um recurso que já existe há muito tempo, mas, que os sistemas operacionais ainda não absorveram por completo, como pode-se ver, somente o Linux possui atualmente uma implementação de kernel que já trata processos e threads em igual teor, contudo, mesmo no Linux os recursos para a manipulação dos threads ainda são limitados. Outra limitação encontrada no estudo foi a dos threads Java ficarem explicitamente vinculados ao processo do programa Java que os criou, impossibilitando separá-los do processo e aproveitar para aumentar a paralelização que poderia ser oferecida por esses threads. Contudo, há uma alternativa que pode ser vinculada a tecnologia de linguagem Java que são os POSIX Threads. Java pode trabalhar com POSIX Threads a partir de pacotes separados e fazer com que os programas Java sejam manipulados como threads no padrão POSIX. Tal recurso, pode permitir que sistemas operacionais como o Linux (que possui bibliotecas e chamadas de sistema que podem suportar os threads POSIX) consigam aumentar a paralelização de execução de aplicações Java.

No contexto das condições de disputa, a conclusão que se chega é ainda mais complexa: num ambiente paralelo e distribuído garantir a exclusão mútua passa a ser uma tarefa de alta complexidade, devido ao fato de que os programas não estarão disputando recursos compartilhados (memória ou arquivos) que estão num só computador, mas sim, em vários computadores espalhados por uma rede. Sendo assim, sistemas operacionais que já possuem um suporte nativo aos ambientes paralelos, como por exemplo um cluster de computadores passam a ser aliados. O sistema operacional Linux possui o mérito de possuir em suas primitivas de kernel os recursos necessários para trabalhar com o mesmo configurado como um ambiente de cluster, basta apenas ativar as opções certas e realizar uma recompilação de kernel.

Característica indispensável de qualquer máquina virtual, a técnica de compilação Just In Time torna-se um item obrigatório em qualquer VM que queira manifestar um

desempenho apropriado e próximo de um binário de compilação estática. As versões das JVMs recentemente liberadas pela Sun Microsystems possuem o recurso da compilação JIT desejado para se obter performances mais arrojadas na execução dos bytecodes Java.

O estudo dos mecanismos de paralelização em software auxilia na compreensão de questões de projeto importantes quando está se pensando na elaboração de um modelo de implementação para um protótipo de máquina virtual. O MPI tradicional (livre de quaisquer plataforma, e, apresentado como uma tecnologia pura e sem dogmas de "marcas" para as suas bibliotecas) mostra-se uma opção muito interessante pelos aspectos da portabilidade que esta tecnologia de comunicação paralela (também considerada técnica de comunicação interprocessos) possui. Além disso, a eficiência e a performance do MPI tradicional agradam aos projetistas de sistemas paralelos e distribuídos. Apesar de todos os prós, MPI tradicional não funciona bem quando implementado para ambientes monoprocessados ou ainda, para sistemas operacionais que funcionam em *stand alone*. Ainda assim, MPI é apontada pelos pesquisadores, como uma forte opção atual para a implementação de novos recursos para os ambientes paralelos e distribuídos.

A tecnologia de linguagem Java em sua constante busca pelo aprimoramento da *versatilidade*, possui uma implementação de MPI específica à linguagem, a qual foi vista neste estudo: trata-se de *MPIjava*. Com poucos arranjos em código-fonte, e, usando instruções específicas para esta tecnologia, MPIjava fornece uma forma de programação em harmonia com o código Java normal. Apesar de se mostrar ainda em dificuldades de competir com outras bibliotecas MPI como a *Fortran/OpenMP*, MPIjava mostra-se com um desempenho linear até certo número de processadores. A pesquisas mostram que MPIjava ainda é uma tecnologia jovem que pode ser melhorada em muito na busca de performances ainda maiores.

A implementação de ambientes paralelos completos nos quesitos: administração de recursos, distribuição de tarefas e gerenciamento da comunicação entre os nós; é a técnica

empregada pela PVM (Parallel Virtual Machine). O uso de *daemons*, e da sistemática de serviços torna o ambiente mais fácil de configurar, e, como o gerenciamento fica por conta do ambiente, o programador de aplicações torna-se livre para implementar seus programas sem a preocupação de inserir linhas de código que coordenem a paralelização da execução de seu programa. No entanto, PVM apresenta-se como um recurso ideal para projetos de sistemas paralelos ou distribuídos em que se queira uma montagem rápida, pois, a performance máxima do sistema dependerá de muitas customizações posteriores, que são necessárias a partir das muitas variáveis que um sistema paralelo e distribuído pode prover, como: rede, número de nós, arquitetura dos computadores, compatibilidade entre os sistemas operacionais, etc.

A JPVM (Java Parallel Virtual Machine), projeto que possui muitas semelhanças estruturais com a PVM (Parallel Virtual Machine), mostra-se com uma performance superior em algumas ocasiões (JPVM ainda é um projeto em fase de evolução), isto se dá pelo fato da linguagem de programação Java possuir facilidades de estruturação não disponíveis em outras linguagens, threads nativas da linguagem e outros pequenos recursos que auxiliam na paralelização, o ambiente JPVM mostra-se estável, escalável e com desempenho aceitável. Partindo-se da afirmativa anterior, pode-se observar que o uso de ambientes paralelos e distribuídos vinculados a tecnologia de uma linguagem de programação podem prover maior poder de processamento.

Descontente com o desempenho do JavaRMI (Java Remote Method Invocation), o pesquisador (WELLING, 1999) implementou a tecnologia denominada RMI+ (Remote Method Invocation Plus), que trouxe como principal recurso adicional, a possibilidade da aplicação reorganizar-se (isto é, ser adaptável) a ambientes paralelos instáveis, onde podem ocorrer situações em que seja necessária a *mobilidade específica em tempo de execução*. Através de políticas que são implementadas pelo programador, a aplicação pode se adaptar a

novas situações e continuar provendo comunicação e processamento paralelos sem perder performance. Apesar de ser uma iniciativa muito interessante, RMI+ deixa por conta da programação em espaço de usuário todo o controle que será feito em tempo de execução para que a aplicação torne-se flexível em ambientes paralelos e distribuídos. O fato revelado no parágrafo anterior, infelizmente, é visto como um agente limitador ao uso do RMI+ que só poderia ser utilizado de forma efetiva por experientes programadores de aplicações paralelas.

É interessante reforçar que, RMI+ não é uma tecnologia nativa da linguagem Java e que foi desenvolvida à parte na pesquisa de (WELLING, 1999). Todavia, a linguagem de programação Java possui a sua implementação nativa de RMI, que já faz parte da API padrão que vem com uma máquina virtual Java. Por limitações no transporte de dados e falta de recursos adicionais para tornar JavaRMI mais versátil, autores das pesquisas afirmam que JavaRMI ainda não é a tecnologia de paralelização mais indicada para uso em sistemas que solicitam grande desempenho das aplicações.

3 - PROJETOS E IMPLEMENTAÇÕES DE MÁQUINAS VIRTUAIS

O estudo dos projetos e modelos de máquinas virtuais paralelas em Java é de grande importância para a compreensão do *estado da arte* e também das pesquisas que estão sendo feitas pela iniciativa de pesquisadores, empresas, centros de pesquisa e outras instituições de incentivo na área da computação paralela de alta performance.

Os projetos e modelos a seguir possuem muitas facetas que os levam a objetivos distintos, mas, a essência de todos que é a paralelização de aplicações Java visando a melhor performance possível, é notoriamente um objetivo comum a todos.

As máquinas virtuais Java atualmente disponíveis são muitas. É evidente que a Sun Microsystems seja a principal interessada no progresso dessa linguagem (pois foi a Sun que a lançou no mercado, através de seu funcionário James Gosling), e as VMs da Sun para os seus *pacotes prontos* como o ***J2SDK, J2EE e o J2ME*** são muito bem aceitas academicamente e comercialmente. Contudo, isso não é uma máxima para o assunto, pois, a VM da Sun tem seu código fechado e proprietário de forma que não se sabe, por exemplo, o quão bem ela está usando um hardware. Além do mais, pela própria diversidade de arquiteturas de computadores existentes, com certeza nenhuma VM há de se comportar de forma otimizada e homogênea sobre todos os ambientes onde for instalada.

Iniciativas do Software Livre (ou Free Software, como é mais comumente citado em literatura) deram início a uma VM para Java, projetada inicialmente para Unix e agora também disponível para GNU/Linux, chamada ***Kaffe Java Virtual Machine*** (WILKINSON, 2005). Atualmente esta VM está em sua versão 1.1.4 e uma versão nova está em desenvolvimento pela equipe de colaboradores. A *Kaffe Java Virtual Machine*, (WILKINSON, 2005) possui código-fonte aberto e é protegida sob as diretrizes da GPL (*General Public License*). Por fim, é comentado a respeito da ***Jalapeño Java Virtual Machine***, projeto acadêmico, apoiado pela *IBM Corporation*, é também um dos mais tradicionais projetos de software livre na área de máquinas virtuais. Muito difundido no meio acadêmico e usada em muitas comparações com outras VMs, será usada como parâmetro de comparação nos testes com a VVM (Virtual Virtual Machine), máquina virtual Java distribuída que foi proposta por essa dissertação através de um modelo de implementação e um protótipo de teste em software.

O estudo a respeito da *Sun Microsystems J2SDK, Kaffe VM* e *Jalapeño VM* visa descobrir os detalhes nas características de cada uma dessas VMs.

3.1. HPJava

Subsidiado pelo *Northeast Parallel Architectures Center* (NPAC), um centro de pesquisa e desenvolvimento focado em computação moderna para informação em larga escala e simulação de aplicações, o projeto *HPJava* surgiu dentro do núcleo do *Parallel Compiler Runtime Consortium* (PCRC), a partir de pesquisas dos membros deste grupo.

Esse projeto surgiu durante o andamento do projeto HPF (*High Performance Fortran*) que era patrocinado pelo DARPA (*Defense Advanced Research Projects Agency*) e executado pelo grupo de pesquisa do PCRC. O objetivo do HPF foi produzir uma versão do FORTRAN que servisse para computação de alta performance.

O projeto também possuía uma idéia para esses sistemas de arranjos de dados a serem computados em ambiente distribuído baseada no SPMD (*Simple Program Multiple Data*), ou seja, usar uma metodologia de programação simples para trabalhar com múltiplos dados que, de preferência, estivessem distribuídos e pudessem ser processados através de computação distribuída. Os trabalhos caminharam de uma forma para o projeto HPF que os programas HPF foram traduzidos para uma linguagem baseada em SPMD que usa MPI como interface, para tanto, foi necessário trabalhar em um descritor de arranjo distribuído em tempo de execução.

Um pesquisador deste projeto (CARPENTER, 1997) afirma que um dos problemas principais encontrados no HPF e já detectado em muitos outros projetos de cunho semelhante é o *multiprocessamento de dados distribuídos*. É fato que um processador pode em velocidades altas (e de modo geral aceitáveis) acessar dados em sua memória local. Contudo, o problema maior não está neste ponto, quando trata-se de computação de alta performance com dados distribuídos o empecilho é recuperar com eficiência e rapidez os dados que estão

alojados em uma outra memória (comumente não-local), pois, se esta CPU não está na mesma placa de plataforma (*ou motherboard*), com certeza será necessário uma comunicação com uma mídia de rede, que poderá complicar a performance da resposta.

Outro problema levantado no projeto HPF foi o do *balanço de carga*. Se um processador está recebendo maior quantidade de trabalho do que os outros no ambiente paralelo em que ele está instalado, então o balanço de carga deve ser revisto, pois, não está havendo sinergia no processamento dos dados.

O HPJava descende do HPF. É uma linguagem de programação com recursos adquiridos (por que não dizer legados) do projeto HPF e também tem o objetivo de convergir para o horizonte da computação de alta performance.

Para se ter uma breve noção, a seguir é ilustrado um trecho de código de um programa em *High Performance FORTRAN*, que distribui algum serviço para quatro processadores, é interessante que se repare na natureza procedural do código.

```
!HPF$ PROCESSORS P(4)

      REAL A(50)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P

      FORALL (I=1:50) A(I)=1.0*I
```

O algoritmo supracitado parece simples, porém, isso se deve ao teor de seus objetivos, quando da colocação de problemas mais complexos e de tarefas que exigem mais do ambiente distribuído, a programação vai ficando cada vez mais difícil. Observe-se agora um algoritmo usando-se HPF cujo problema em código alto nível expressa um problema bem mais complexo para o tempo de execução, devido à dificuldade de distribuição e reagrupamento das informações para a finalização dos resultados do algoritmo.

```
!HPF$ PROCESSORS P(4)

      REAL A(50), B(50)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
!HPF$ DISTRIBUTE B(BLOCK) ONTO P

      A(1:49) = B(2:50)
```

Na figura 10, é mostrada a intenção de se fazer uma atribuição usando-se dados distribuídos, o HPF faz uma operação deste tipo conforme a ilustração.

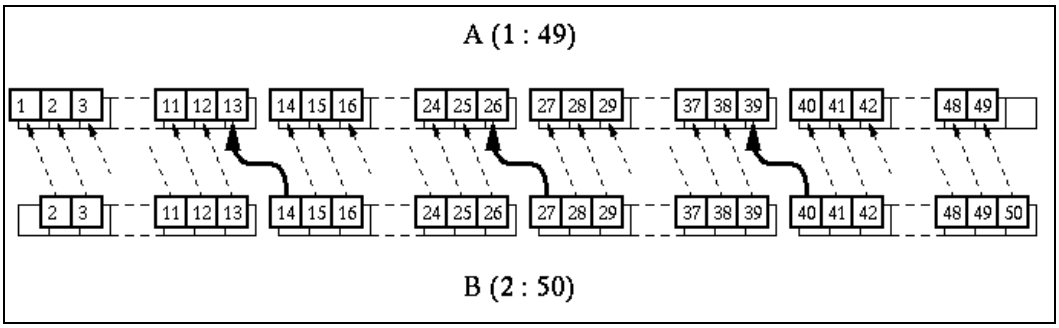


Figura 10 : *procedimento de atribuição nos dados distribuídos*: *(CARPENTER, 1997).*

O relacionamento entre HPF e HPJava é a ênfase no trabalho com dados distribuídos, principalmente na forma de arranjos que é uma constante em ambos os projetos.

Entre as características da linguagem HPJava estão: foi projetada para programação paralela, ela estende a linguagem Java provendo uma sintaxe definida para manipulação de arranjos distribuídos, implementa o modelo *HPspmd* (isto é, processos independentes rodando o mesmo programa, compartilhando elementos de arranjos distribuídos), processos operam diretamente sobre os seus próprios elementos, comunicação explícita é necessária num programa para permitir acesso a elementos pertencentes a outros processos.

Suponha-se, como exemplo, que um programa em HPJava queira criar uma grade de processos (também citada como grade de objetos na literatura de CARPENTER(1997)) de ordem 2x3 para alocar pelo menos 1 processador para cada elemento desse "arranjo". O processamento poderá ser realizado por, no mínimo, seis ou mais processadores.

Os processos são instanciados a partir do conceito de grade de objetos, vide a seguinte construção em HPJava:

```
Procs p = new  Procs2(2, 3);
```

Assume-se que o programa estará executando em seis ou mais processadores, como já mencionado. Dessa forma, um programa completo que fornece uma visão essencial do uso de grades de processos no HPJava seria:

```
Procs p = new Procs2(2,3);
On (p) {
      Dimension d = p.dim(0), e = p.dim(1);

      System.out.println("My coordinates are (" + d.crd()
            + "," + e.crd() + ")");
}
```

Uma saída *possível* enviada para a saída-padrão (terminal de vídeo) pelo programa acima citado seria algo parecido com o texto seguinte:

```
My coordinates are (0,2)
My coordinates are (1,2)
My coordinates are (0,0)
My coordinates are (1,0)
My coordinates are (1,1)
My coordinates are (0,1)
```

Apesar das semelhanças entre o HPF e o HPJava, as diferenças de desempenho mostradas no trabalho de (CARPENTER, 1997) e do paradigma de linguagem (no cas, a orientação a objetos presentes em Java) mostrados pelo HPJava atraíram grande número de pesquisadores em computação paralela.

3.2. JavaSpaces

A tecnologia de rede *Jini*, da Sun Microsystems, a qual inclui a tecnologia *JavaSpaces e Jini Extensible Remote Invocation* (ou Jini ERI), é uma arquitetura aberta que habilita desenvolvedores a criarem *serviços de rede cêntricos*, que são implementados em hardware ou software e são altamente adaptáveis a mudanças. Além disso, a tecnologia Jini pode ser usada para construir *redes adaptativas escaláveis e flexíveis* como tipicamente é requisitado em ambientes de computação dinâmica (SUN, 2005).

Na documentação de *especificação de serviços JavaSpaces*, é informado que sua principal vocação é prover uma *persistência distribuída* e um *mecanismo de troca de objeto para códigos escritos na linguagem Java*. Objetos são escritos em entradas que provém um agrupamento tipado de campos relevantes. *Clientes* podem realizar operações simples sobre um serviço JavaSpace para escrever uma nova entrada, procurar entradas existentes e remover entradas do espaço. Usando estas ferramentas, pode-se escrever *sistemas de armazenamento* e também escrever *sistemas que usam fluxo de dados* para implementar algoritmos distribuídos e permitir que os serviços JavaSpaces implementem a persistência distribuída para a aplicação em tempo de execução.

Segundo (SUN, 2005), sistemas distribuídos são difíceis de implementar, eles requerem reflexões cuidadosas sobre os problemas que normalmente não ocorrem em computações locais. Os problemas primários mais conhecidos são os de *falha parcial*, grande aumento de latência e compatibilidade de linguagem. A linguagem de programação Java tem um sistema de invocação remota de método (JavaRMI) que permite aproximações para computações distribuídas usando Java com técnicas de programação naturais da linguagem no ambiente de programação de usuário. Isto é estendido em camadas sobre o mecanismo de *serialização* de objetos da plataforma Java para ordenar parâmetros de métodos remotos em

uma forma que pode ser embarcada através de uma linha desordenada num servidor remoto da máquina virtual Java (JVM).

É interessante a ilustração de algum código para que se tenha noção de como os objetos são construídos na tecnologia JavaSpaces baseada em serviço. O código a seguir, feito com uso de JavaSpaces, mostra um exemplo típico de um *sample* que emula um programa do tipo "Hello World".

```
package jsbook.chapter1.helloWorld;
import jsbook.util.SpaceAccessor;
import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;

public class HelloWorld {
    public static void main(String[] args) {
        try {
            Message msg = new Message();
            msg.content = "Hello World";

            JavaSpace space = SpaceAccessor.getSpace();
            space.write(msg, null, Lease.FOREVER);

            Message template = new Message();
            Message result =
                (Message)space.read(template,null,
Long.MAX_VALUE);
            System.out.println(result.content);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Como mencionado na especificação do JavaSpaces, a implementação possui entradas de serviço, e uma entrada é, explanando-se de forma simples, um tipo de grupo de objetos, expressados numa classe para a plataforma Java que implementa a interface *net.jini.core.entry.Entry*. Existem muitos detalhes acerca das entradas JavaSpace que podem ser esclarecidos através da leitura integral do *JavaSpaces Service Specification* da *Sun Microsystems*.

Uma entrada pode ser escrita num serviço JavaSpaces, o qual cria uma cópia da entrada no *space* (termo usado na literatura sobre JavaSpaces para referenciar implementações

de serviço JavaSpaces) que pode ser usada em futuras operações de procura. Também é possível se procurar por entradas no serviço JavaSpaces usando templates, os quais são entradas de objetos que tem alguns ou todos de seu conjunto de campos configurados para especificar valores que precisam ser exatamente encontrados. Os campos restantes são deixados como *curingas*, ou seja, estes campos não são usados na procura. Ilustrando o conceito de entradas em JavaSpaces, o código seguinte então é um algoritmo da criação de uma entrada.

```
package jsbook.chapter2.fieldTest;
import net.jini.core.entry.Entry;

public class TestEntry implements Entry {
    public Integer obj1;
    public Integer obj2;

    public TestEntry() {
    }
}
```

Há dois tipos de operações de procura: *leitura e captura*. A requisição de leitura para um espaço retorna qualquer entrada que case o template no qual a leitura foi feita, se o template for encontrado, a entrada de casamento é removida do espaço. Pode-se requisitar a um serviço JavaSpaces para, notificar quando uma entrada casar, e um template especificado seja escrito. Isto é feito usando o modelo de evento distribuído contido no pacote `net.jini.core.event` e descrito na tecnologia *Jini Core Platform Specification*. É válido lembrar que todas as operações que modificam um serviço JavaSpaces são executadas de uma maneira transacionalmente segura no que diz respeito ao espaço.

Se uma operação de escrita retorna sucesso, então a entrada foi escrita no espaço (embora intervenções possam removê-la do espaço antes de uma nova procura subseqüente). E se uma operação retorna uma entrada, essa entrada pode ter sido removida do espaço, e nenhuma operação futura lerá ou fará exame dessa entrada. Ou seja, cada entrada no espaço

pode ser examinada, na maioria dos casos, uma vez. Note-se, entretanto, que duas ou mais entradas em um espaço podem ter exatamente o mesmo valor.

Um exemplo do uso do pacote *net.jini.core.event* pode ser visto no código seguinte, nele implementa-se a classe `Listener` ("ouvidor") que notifica eventos remotos.

```
package jsbook.chapter8.helloWorld;
import jsbook.chapter1.helloWorld.Message;
import java.rmi.server.*;
import java.rmi.RemoteException;

import net.jini.core.event.*;
import net.jini.space.JavaSpace;

public class Listener implements RemoteEventListener  {
    private JavaSpace space;

    public Listener(JavaSpace space) throws RemoteException {
        this.space = space;
        UnicastRemoteObject.exportObject(this);
    }

    public void notify(RemoteEvent ev) {
        Message template = new Message();

        try {
            Message result =
                (Message)space.read(template, null,
Long.MAX_VALUE);
            System.out.println(result.content);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

A arquitetura da tecnologia de JavaSpaces suporta um *mecanismo de transação simples* que aceita atualizações de multi-operações e/ou multi-espaços para atomicidade completa. Realiza-se esta tarefa através do uso de um modelo de `commit` de duas fases sobre as semânticas de transação padrão, tal como é definido no pacote *net.jini.core.transaction* e é descrito na Especificação da *Jini Technology Core Plataform*, assim como, as entradas escritas num serviço JavaSpaces são governadas por um

mecanismo especial, como definido pelo pacote *net.jini.core.lease*. O próximo código ilustra um algoritmo que usa JavaSpaces e mecanismo de transação.

```
package jsbook.util;
import java.rmi.*;
import net.jini.core.transaction.server.TransactionManager;
import com.sun.jini.mahout.Locator;
import com.sun.jini.outrigger.Finder;

public class TransactionManagerAccessor {
    public static TransactionManager getManager(String name) {
            Locator locator = null;
            Finder finder = null;

            if (System.getSecurityManager() == null) {
            System.setSecurityManager(
                new RMISecurityManager());
        }

            if
(System.getProperty("com.sun.jini.use.registry") != null) {
                locator = new
com.sun.jini.mahout.RegistryLocator();
                finder = new
com.sun.jini.outrigger.RegistryFinder();
            } else {
                locator = new
com.sun.jini.outrigger.DiscoveryLocator();
                finder = new
com.sun.jini.outrigger.LookupFinder();
            }

        return (TransactionManager)finder.find(locator, name);
    }

    public static TransactionManager getManager() {
        return
getManager(com.sun.jini.mahalo.TxnManagerImpl.DEFAULT_NAME);
    }
}
```

Apesar das muitas características interessantes de JavaSpaces, pode-se perceber que não foram realizadas alterações na máquina virtual Java para a obtenção de quaisquer características de paralelismo de JavaSpaces, ou seja, os autores (SUN, 2005) implementaram bibliotecas de classe Java para obter o paralelismo desejado para as classes, transações e implementação de serviços com JavaSpaces (SUN, 2005).

3.3. Titanium

O Titanium é uma versão paralelo explícito de Java, desenvolvido na Universidade de Berkeley para dar suporte a computação científica de alta performance sobre multiprocessadores em larga escala, isso inclui supercomputadores maciçamente paralelos e clusters de memória distribuída com um ou mais processadores por nó. Outro objetivo da linguagem inclui segurança, portabilidade e suporte a construção de estruturas de dados complexas.

Apesar do pouco material de referência encontrado sobre o Titanium, que é um interessante projeto para a linguagem de programação Java em ambientes paralelos, pode-se citar, a partir do website dos autores, o seguinte: as principais modificações que o Titanium proporciona ao Java são: classes imutáveis definidas pelo usuário (frequentemente chamadas "peso leve" ou classes de "valor"), Matrizes multidimensionais flexíveis e eficientes, tipos embutidos para representação de pontos multidimensionais, retângulos e domínios gerais. Iteração de *loops* desordenados para proporcionar otimização agressiva, modelo de controle SPMD (*Single Program Multiple Data*) explicitamente paralelo, gerenciamento de memória baseado em zonas (em adição ao *garbage collection* padrão do Java), sistema de tipos da linguagem estendido para expressar ou inferir localmente e propriedades compartilhadas de estruturas de dados distribuídas, prevenção (ainda em tempo de compilação) de *deadlocks* sobre barreiras de sincronização. Além disso, oferece uma biblioteca útil de primitivas de paralelização (barrier, broadcast, reductions, etc), sobrecarga (overloading) de operadores e templates (ou classes parametrizadas).

O Titanium provê uma abstração de espaço de memória global (similar a outras linguagens tais como *Split-C e UPC*) pela qual todos os dados têm uma afinidade com um processador controlável pelo usuário, mas, processos paralelos podem diretamente referenciar outras memórias para ler ou escrever valores ou para aumentar as transferências de dados. A

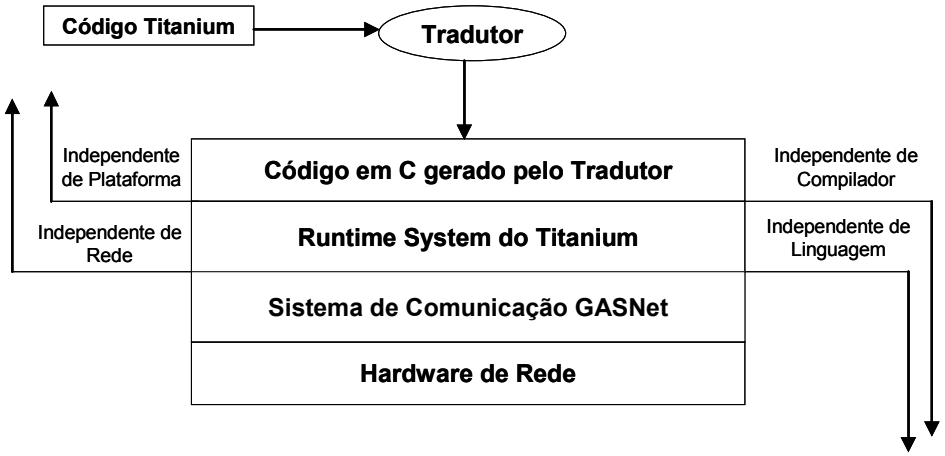
portabilidade resulta em *programas Titanium* que podem executar inalterados em máquinas monoprocessadas, máquinas com memória compartilhada e máquinas com memória distribuída. A refinação do desempenho pode ser necessária para melhorar, por exemplo, uma aplicação de estrutura de dados para ambientes em que se tem memória distribuída, mas a portabilidade funcional dos programas Titanium aceita o desenvolvimento sobre máquinas de memória compartilhada ou monoprocessadas.

Pelas características até aqui apresentadas, pode-se então compreender o Titanium como um *super conjunto* de Java 1.4 que herda todas as propriedades de expressividade, usabilidade e segurança que a linguagem oferece. Na verdade, o Titanium incrementa as características de segurança de Java por prover sincronização testada que prevê um certo conjunto de bugs de sincronização nas classes. Para suportar essas estruturas de dados complexas, o Titanium usa o mecanismo de classes orientadas a objeto de Java por meio do qual o espaço de endereço global aceita grandes estruturas compartilhadas. As matrizes multidimensionais do Titanium adicionam facilidades ao suporte para computações de alta performance hierárquicas e adaptativas baseadas em grid (HILFINGER et al, 2004).

O compilador do Titanium realiza pesquisas e análises nos programas, e assim, trabalha otimizando transformações para programas Titanium. Por ser uma linguagem explicitamente paralela, novas análises são necessárias até mesmo para transformações de código padrão. A análise do compilador envolve construções de sincronização e variáveis de acesso compartilhado. As transformações também incluem otimizações de cache, comunicação, identificação de referências a objetos sobre o processador local, substituição do *overhead* de gerenciamento de memória em tempo de execução com checagens estáticas. A versão atual traduz programas em Titanium para a linguagem C, que são compilados para binários nativos pelo compilador de C, e, então, são *editados* para o *Titanium Runtime Libraries* (dessa forma, não há JVM!). Uma ilustração simples, que pode orientar como se dá

o processo explicado acima, é representada na figura 11 em camadas de relacionamento do código dos programas Titanium até a transformação dele em código C ou da transmissão para um ambiente de rede.

Figura 11 : *visão do funcionamento da versão Titanium com tradutor em camadas:*



(HILFINGER et al, 2004).

O Titanium na versão atual executa sobre uma ampla gama de plataformas, incluindo: *sistemas monoprocessados, multiprocessados de memória compartilhada, cluster com memória distribuída de monoprocessadores ou SMPs (CLUMPS)*, e um número específico de arquiteturas de supercomputadores (*Cray, T3E, IBM SP, Origin 2000*). Por trás da memória distribuída, pode-se utilizar uma ampla variedade de interconexões de rede de alta performance, incluindo *Active Messages, MPI, IBM LAPI, shmem e UDP*. Logo, o Titanium é recomendado para escrita de aplicações paralelas científicas baseadas em Grid, e muitas das principais aplicações têm sido escritas com Titanium e continuam a avançar seu desenvolvimento (HILFINGER et al, 2004).

*3.4 JavaNOW (*Java on Network Of Workstations)

No trabalho de (THIRUVATHUKAL et al, 2000) afirma-se que Java pode conservar o estado de um objeto e recriar esse objeto em uma outra máquina, suportando ambos os objetos e a migração persistente do objeto. Recentemente, um número de estruturas

distribuídas foram desenvolvidas pela Sun Microsystems, tal como a Invocação Remota de Método (ou RMI, um procedimento remoto client/server), por *JavaSpaces* imaginado como um framework de espaço de tupla, já mencionado nesta revisão, e Jini (um framework de diretório e serviços). Talvez, o benefício maior de Java seja ainda a sua característica multiplataforma, isto é, uma aplicação Java pode ser executada em toda máquina com a JVM instalada, sem recompilação. THIRUVATHUKAL et al(2000) faz uma ressalva quando cita que, em paralelo à emergência de Java como linguagem preferida para a programação distribuída vieram as redes de workstations como plataforma preferida para a computação distribuída. A razão primária pela qual os clusters de workstations se tornaram tão importantes é a excelente relação do custo/performance de tais sistemas quando comparada aos tradicionais multi-computadores maciçamente paralelos. É natural explorar as aproximações pelas quais Java pode ser usado para formar uma máquina virtual paralela usando-se de redes de workstations. JavaNOW é uma tentativa.

Usar *redes de workstations* como um *computador virtual paralelo* não é uma idéia completamente nova, há muitas características de JavaNOW que a tornam original entre os sistemas de passagem de mensagem com características similares. Segundo (THIRUVATHUKAL et al, 2000), JavaNOW é um sistema baseado em Java de forma pura, que pode executar em toda arquitetura com sustentação para máquina virtual Java (JVM), sem recompilação. Isto está em contraste com outros *frameworks* baseados em rede mais tradicionais, tais como MPI, PVM, Linda e o Memo, que requerem bibliotecas binárias com arquiteturas específicas e os executáveis a serem instalados em cada máquina na qual o computador virtual paralelo executará.

Adicionalmente, a maioria dos frameworks baseados em rede está embasado em *processos pesados*, enquanto JavaNOW suporta ambas computações baseada em threads e baseada em processos. Além disso, outros sistemas baseados em rede são sistemas restritos a

passagem de mensagem, enquanto JavaNOW provém o modelo de passagem de mensagem e o modelo de Linda de memória associativa compartilhada distribuída para comunicação inter-processos e para o acesso exclusivo aos objetos compartilhados distribuídos. JavaNOW estende o modelo Linda (de modo significativo) fornecendo um conjunto rico de coleções de comunicação e primitivas de computação similares àqueles encontrados em MPI.

JavaNOW é uma tecnologia com performance superior (THIRUVATHUKAL et al, 2000) a *Linda* e *MPI,* pois, suporta o modelo de computação de fluxo de dados (*data flow model*).

Para ilustrar como o JavaNOW é projetado em código, existem algumas abstrações fundamentais citadas por (THIRUVATHUKAL et al, 2000). A primeira a ser considerada é a abstração de entidades ativas no cluster de workstations, esta é representada pela classe `ActiveEntity`. Observe o código a seguir:

```
public class Task extends ActiveEntity {
    public Object execute(Object arg, JavaNOWAPI api) {
        Object o;
        int myid = ((Integer)arg).intValue();
        ...
        return o;
    }
}
```

Contudo, para projetar uma aplicação JavaNOW mais complexa há outros detalhes de implementação a serem considerados, como a *serialização de objetos* via interface `java.io.Serializable`. Pode-se ver um exemplo simples de implementação de aplicação no código a seguir:

```
public class AnApp extends JavaNOWApplication implements
java.io.Serializable {
    public static void main(String args[]) {
        AnApp app = new AnApp(args[0]);
    }
    public Hello(String propertyFile) {
        super(PropertyFile);
        // local initialization
        applicationIsReady();
    }
    public void master() {
```

```
            // . .
        }
        public void slave(int myid) {
            //. . .
        }
}
```

É importante que se repare nos métodos *master* e *slave*, pois, eles implementam quem é o mestre e quem é o escravo no modelo de implementação de comunicação interprocessos de JavaNOW. Este modelo é, em alguns aspectos, uma aproximação ao modelo Client/Server.

3.5. Manta

Segundo (BAL et al, 1998), *Manta* é um compilador Java nativo. Ele compila os códigos-fonte de Java para executáveis de plataforma *x86*. Seus objetivos são *vencer em desempenho* todas as implementações de Java. Atualmente possui uma eficiente versão do RMI (código-fonte compatível com o RMI atual). É aproximadamente 30 vezes mais rápido do que as implementações atuais (BAL et al, 1998). As bibliotecas de classe são retiradas de *kaffe*, *classpath* e parcialmente de *homebrew*.

Nas experiências com *Java RMI* e *JavaParty*, (BAL et al, 1998) encontrou o modelo de programação do RMI e do JavaParty convenientes, mas acharam também que o desempenho da invocação remota de métodos para programação de clusters paralelos de workstations lenta demais. A latência em ambos os sentidos do RMI da Sun Microsystems está na ordem de 1200 microssegundos sobre a *Myrinet*. (BAL et al, 1998), alega que com a biblioteca Panda, projetada pela equipe na Universidade de Vrije, consegue 30 microssegundos sobre o mesmo hardware de rede. Baseado em sua experiência com o Orca, (BAL et al, 1998) escreveram *um sistema Java* chamado *Manta*, caracterizado por um *compilador nativo* e um *Runtime* para RMI, que faz um "RMI nulo" no tempo de 34 microssegundos. Manta suporta a linguagem completa de Java 1.1.X em diante, incluindo exceções, *garbage collector* e o carregamento dinâmico da classes. Manta suporta também

algumas extensões de Java, tais como o modelo de programação de JavaParty (isto é, a palavra reservada *remote*), os objetos replicados descritos em (JAVAGRANDE, 2001), e ainda, a eficiente divisão e conquista do paralelismo (as palavras reservadas *spawn* e *sync* provenientes do *cilk*). Um sistema chamado *Satin* foi descrito nos eventos *(*EUROPAR, 2000) e (PPoPP, 2001). Além disso, (BAL et al, 2005) construíram um sistema de memória compartilhada distribuída (*DSM*) no topo das camadas de Manta, chamado *Jackal*, apresentado e descrito em eventos como *(*JAVAGRANDE, 2006) e (PPoPP, 2001*)*.

Em (MAASSEN et al, 1998) disponibilizam em seu trabalho trechos de código, dessa forma, a partir do projeto Manta, sugiram as primeiras idéias para melhorar os sistemas de paralelização de aplicações usando Java. O trecho de código a seguir representa o uso da palavra chave adicional *remote* na implementação de objetos remotos no ambiente *Runtime* do Manta Java.

```
remote class Hello {
    public String SayHello( ) {
        return "Hello, World!!!";
    }
    public static void main (String arg[ ]) {
        //create new remote object on CPU 4...
        RunTimeSystem.setTarget(4);
        new Hello( ).SayHello( );
    }
}
```

O próximo trecho de código, também usando a palavra-chave *remote* tenta ilustrar o emprego dos recursos de paralelismo com variáveis locais.

```
remote classe A {
    int var;
    void foo ( ) {
        var = 1;
    }
}
class test {
    remote int i; //não permitido!!! Só classes podem ser
    remotas...
    public static void main (String arg[ ]) {

        A a = new A ( );
```

```
            a.var = 1 //não permitido pois var está numa classe
            remota...
        }
    }
```

Analisando-se os códigos presentes no trabalho de (MAASSEN et al, 1998), fica evidente que as possibilidades de construções úteis com o Manta são muitas, contudo, há também a necessidade de um cuidado especial para com as construções errôneas, como as do exemplo com variáveis locais remotas, pois, as palavras reservadas adicionais desse "dialeto" de Java não podem ser usadas em qualquer contexto e nem para quaisquer construções para implementação de paralelismo. Algumas primitivas que mantêm a atomicidade de certos elementos da linguagem, como as classes não podem ser "quebrados" em partes num ambiente distribuído como o programador pode *erroneamente* pensar.

3.6. JavaParty

Com melhoria da tecnologia de interconexão de clusters e a crescente confiabilidade e disponibilidade dos clusters, os mecanismos atuais de Java tornaram-se insuficientes, a comunicação explícita de `sockets` é de muito *baixo nível* para que a programação paralela seja confortável. O RMI é *restritivo* e seu *overhead* para tratar dos problemas de rede é demasiado lento com clusters. Os *`sockets`* e o *RMI* resultam num programa de tamanho maior e, dessa forma, reduzem a produtividade do programador e a manutenibilidade do código (PHILIPPSEN et al, 1997).

JavaParty preenche este *gap,* pois estende Java tão minimamente e tão transparentemente como seria possível com um pré-processador e um sistema *run-time* para programação paralela distribuída em clusters de workstations heterogêneos.

Embora, a princípio, o JavaParty possa incluir qualquer número e tipo de workstations, latências elevadas de rede e as velocidades baixas de rede restringem o uso de redes tradicionais. Os melhores tempos de execução podem ser conseguidos com uma

interconexão especial em hardware. A implementação atual de JavaParty em uma rede ParaStation, com *sockets TCP/IP-like* trafega-se a *15 MByte/s* para pacotes de tamanho discreto numa comunicação ponto-a-ponto e com uma latência máxima de 3,4 microssegundos. Assim, (PHILIPPSEN et al, 1997) o JavaParty é uma plataforma com dois propósitos. Serve como um ambiente para aplicações com clusters, e, é uma base para a pesquisa em ciência da computação na otimização de técnicas para prover localidade e reduzir o tempo de uma comunicação. Um programa Java `multi-thread` pode facilmente ser convertido para um programa distribuído de JavaParty através da identificação daquelas *classes* e *threads* que devem ser *espalhadas* através do ambiente distribuído. O programador indica isto por um novo modificador de classe introduzido que se chama `remote`.

O novo modificador é a única extensão de Java. Desde que os `threads` de Java são os objetos de uma classe `thread`, então, `threads` remotos podem ser criados como objetos de uma classe remota de `threads`. Por fim, fica explícito que não são necessárias poucas mudanças no que diz respeito à reescrita e reorganização do programa Java. O tamanho do código-fonte fica, portanto, quase do mesmo tamanho comparado à uma aplicação comum (não paralela.)

JavaParty fornece um espaço de endereço compartilhado, isto é, embora os objetos de classes remotas possam residir em máquinas diferentes, seus métodos e as variáveis (ambas estáticas e não-estáticas) podem ser acessadas da mesma maneira que em Java puro. Pede atenção o fato de que o JavaParty esconde os mecanismos de comunicação e endereçamento do usuário e controla as exceções da rede internamente, nenhum protocolo de comunicação explícito necessita ser projetado e implementado pelo programador.

O modificador `remote` é a única extensão da linguagem. Tentou-se evitar esta extensão, mas não há nenhuma maneira de transformar as classes básicas da biblioteca do

JDK em classes remotas pelas seguintes razões: não se tem o código-fonte do JDK completo; e mesmo que se tivesse, diversas classes têm muito código nativo que é especialmente projetado para implementações de linguagem que só podem ser usadas com um único processador, por exemplo, `Thread` , `I/O`, `Runtime`, ou `System`. Fazendo um *upgrading* nestas classes para executar transparentemente, a semântica remota requereria um *redesign* das bibliotecas da classe e do *runtime system* de Java. Além disso, afirma (PHILIPPSEN et al, 1997), se o informado no parágrafo anterior ocorresse, JavaParty não estaria mais então disponível em todas as plataformas, pois a partir daí seria completamente distinto da implementação *standard* do JDK.

Uma visão do funcionamento interno de JavaParty de acordo com o supracitado pode ser visto de forma ilustrativa na figura 12.

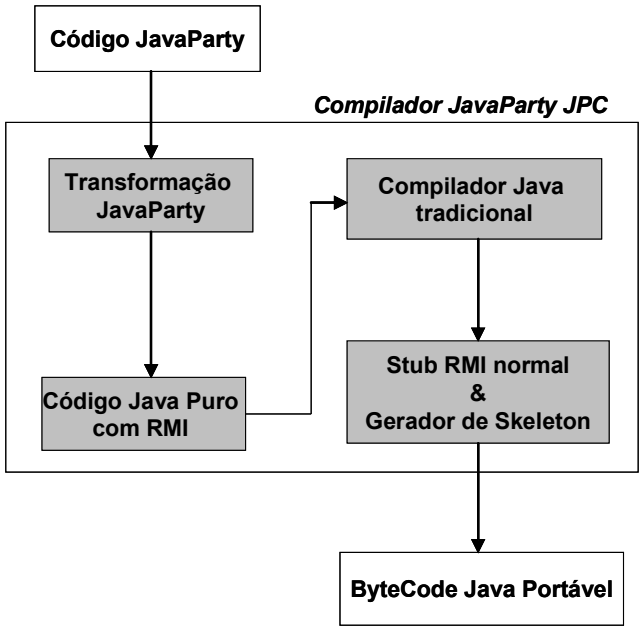


Figura 12: *transformações de códigos realizadas pelo compilador JavaParty JPC:* (PHILIPPSEN et al, 1997)

O código de JavaParty também é considerado “limpo” por não preocupar o programador com detalhes do paralelismo da aplicação nos baixos níveis de execução. Um

exemplo de classe implementada com JavaParty e usando-se da palavra reservada `remote` é mostrado a seguir.

```
remote class B extends A implements C {
    T x = I; // instance variable
    static U y = J; // static variable
    T foo(V z) { P } // method
    static void foo2() { Q } // static method
    static { R } // static block
    B(T z) { S } // constructor
}
```

A *classe B* pode ser, portanto, executada tanto local quanto remotamente no ambiente distribuído, conforme forem levantadas as necessidades, de forma dinâmica. A palavra-reservada `remote` é a diretriz que permite a essa classe migrar para outros nós do ambiente distribuído.

O JavaParty é um projeto destaque entre todos aqueles pesquisados até este ponto. Referenciado em muitos estudos e usado por uma comunidade singular de usuários experientes e interessados por alta performance em aplicações paralelas.

3.7. Sun Microsystems Java Virtual Machine

A plataforma de Java 2 edição *Standard* (J2SE), segundo (Sun Microsystems, 2005), fornece um ambiente completo para o desenvolvimento de aplicações *desktop* e *server*. Serve também como a fundação para a plataforma de Java 2 edição *Enterprise* (J2EE) e para *Java Web Services*. Há dois produtos principais na família da plataforma de J2SE: o ambiente *runtime* do J2SE (JRE) e o de desenvolvimento de J2SE (JDK). O JRE fornece a máquina virtual de Java, as APIs e outros, além disso, JRE não contém ferramentas e utilitários tais como *compiladores* ou *o depurador* para applet para desenvolvimento de *aplicações* e *applets*. O SDK é um super conjunto do JRE, e contém tudo que está no JRE, mais ferramentas adicionais como os compiladores e os debuggers necessários para *applets* e *aplicações*. A figura 13 ilustra as tecnologias e componentes usados na plataforma de J2SE .

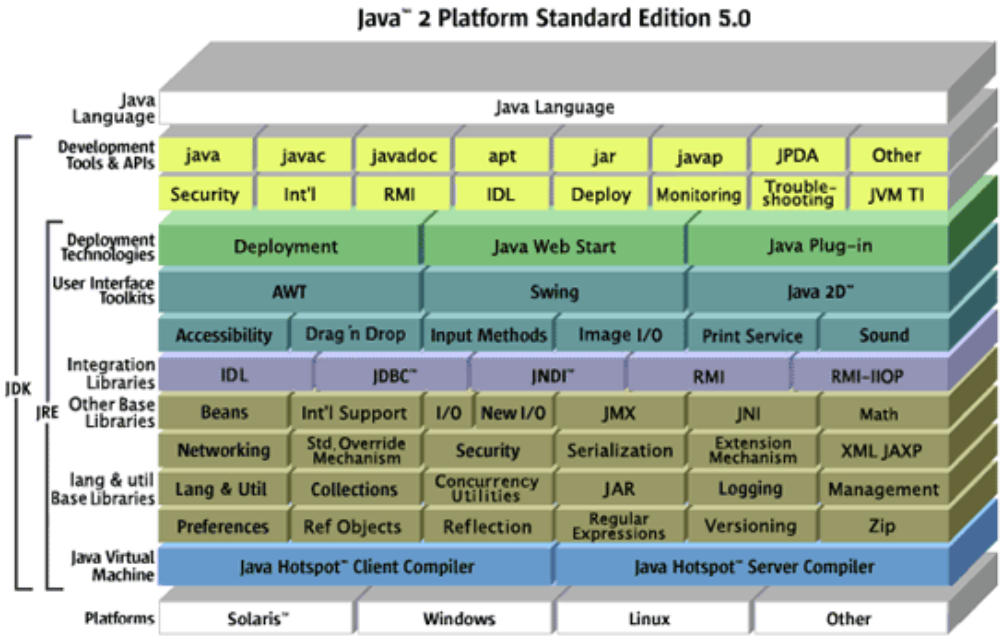


Figura 13: *Plataforma Standard do J2SE na versão 5.0: (SUN, 2005).*

Ao analisar a figura 13, não fica difícil perceber como esta plataforma, que é mantida pela Sun Microsystems, é robusta. Ela é muito completa e ainda fornece implementação da VM da Sun para múltiplos de sistemas operacionais.

(LINDHOLM et al, 1996) criaram um livro, que está disponível em HTML no site da Sun Microsystems que descreve o funcionamento e detalhes de implementação da máquina virtual Java. No site, pode-se obter desde os conceitos básicos da VM até o conjunto de instruções que a VM dispõe para a sua linguagem *Assembly-like*.

A API (*Application Program Interface*) do J2SE define a maneira prescrita pela qual um *applet* ou uma *aplicação* podem fazer requisições e usar a funcionalidade disponível nas bibliotecas de classe compiladas de J2SE. Na verdade, para (Sun Microsystems, 2005) as bibliotecas de classe de J2SE são também parte da plataforma de J2SE. A API de J2SE consiste em tecnologias que organizam-se em dois grupos: *Core Java* e *Desktop Java*. O Core Java fornece a funcionalidades de auxílio na elaboração de programas empresariais poderosos em áreas chaves tais como o acesso a bancos de dados, segurança, invocação remota de métodos (RMI) e comunicações, entre outros. Já o *Java Desktop* fornece uma série de características para auxiliar na construção de *aplicações desktop* que provém ao usuário experiências ricas. O *Java desktop* consiste em produtos da distribuição tais como o *Java Plugin*, APIs de modelagem de componente tal como o *JavaBeans*, APIs de *interface gráfica de usuário (GUI)* tal como as *classes de fundação de Java (JFC*) *e Swing*, e APIs de *multimídia* tal como *Java3D*.

O desenvolvedor oficial de Java (Sun Microsystems, 2005) informa que há ainda no site um guia completo em HTML a respeito da API de J2SE. Avaliando-se todos os pontos positivos em relação à VM da Sun, pode-se dizer que o suporte e a documentação são fatores que a tornam forte candidata para uso no projeto desta dissertação.

3.8. Kaffe Java Virtual Machine

Kaffe é uma boa escolha como base para cursos e/ou pesquisas a respeito de máquinas virtuais, ou, para o caso de se necessitar uma máquina virtual como um componente integral de um *software open source* ou uma *distribuição livre de software Java.*

Segundo (WILKINSON, 2005), o projeto Kaffe cresceu e se expandiu rapidamente na comunidade dos usuários de software livre, assim, hoje é distribuído na forma de pacotes em muitas distribuições do GNU/Linux como DEB, Red Hat, Mandrake, SuSE, Debian, Gentoo, Conectiva, PLD, Ark Linux, FreeBSD, NetBSD, OpenBSD, e o muitos outros.

A iniciativa de (WILKINSON, 2005) não é isolada. Existem outros grupos de pesquisa acadêmica realizando esforços para produzirem VMs Java cada vez mais eficientes e que retirem do hardware o maior potencial possível dando para as aplicações da linguagem um desempenho singular. Alguns destes projetos relacionados são: ***JanosVM*** (LEPREAU et al, 2006), ***Latte*** (YANG et al, 2006), , ***Gilgul*** (COSTANZA, 2006), ***Alta*** (TULLMANN, 2006), ***JC*** (COBBS, 2006), entre outros.

Atualmente a equipe de colaboradores do projeto Kaffe está se esforçando para melhorar esta VM incorporarando novos recursos para ganho de desempenho.

A documentação disponível no site a respeito desta VM melhorou muito nos últimos tempos e vêm sendo acrescentados vários tópicos que descrevem desde a estrutura aberta desta VM até a sua API completa. Kaffe (WILKINSON, 2005) incorpora a maioria das características da VM da Sun sem, contudo, copiar uma linha sequer do código proprietário da Sun Microsystems. Mesmo assim, os bytecodes gerados pelo compilador da KaffeVM tem a mesma segurança, compatibilidade e performance por aqueles gerados pelo compilador do J2SE (WILKINSON, 2005), por exemplo.

Além de todos os pontos positivos, Kaffe já foi testada na maioria dos sistemas operacionais existentes e obteve êxito de execução na grande maioria dos sistemas operacionais (WILKINSON, 2005).

A comunidade de pesquisa em ciência da computação também tem respondido positivamente aos desenvolvedores da KaffeVM usando-a como máquina virtual em muitos ambientes de teste, como por exemplo no trabalho de (RADHAKRISHNAN, 1999) que utilizou-a em parte dos testes numa máquina com processador de arquitetura Sun SPARC e com sistema operacional Solaris.

3.9. Jalapeño Java Virtual Machine (Jikes RVM)

A Jalapeño Java Virtual Machine possui um objetivo essencial: foi desenvolvida para ser uma máquina virtual Java para ***servidores*** escrita em linguagem de programação Java. Jalapeño busca *alta performance* e *escalabilidade* como características particulares em tempo de execução. Além disso, e, segundo a pesquisa desses autores, Jalapeño implementa um modelo de objeto e layout de memória ***único***, que, entre outras qualidades, oferece em seu *Runtime* **acesso rápido** a recursos como: array de elementos, campos e métodos. Como projeto de máquina virtual paralela, Jalapeño executa a chamada *multiplexação de threads Java* através de **processadores virtuais** (estes, por sua vez, e segundo (ALPERN et al, 2000) são implementados como *threads de sistema operacional*). Jalapeño também possui uma implementação própria de sistema de compilação Just In Time, mais conhecida como ***compilador de otimização dinâmica,*** para os métodos que são mais frequentemente usados e que geram computação intensiva, dessa forma, podem operar com desempenho mais satisfatório quando traduzidos para código nativo da máquina.

Como máquina virtual destinada a servidores, a Jalapeño VM cobre muitos requisitos (na verdade, são objetivos do projeto Jalapeño para que a Virtual Machine de (ALPERN et al, 2000) fosse diferenciada de outros projetos) que não necessariamente fazem parte de sistemas clientes, pessoais ou ainda JVMs para sistemas embarcados. Esses requisitos são: *exploração de processadores de alto desempenho*, *escalabilidade em SMPs, trabalho diferenciado com limites de threads, disponibilidade contínua, respostas rápidas, uso de bibliotecas, degradação amenizada*.

Cada item dos requisitos corresponde a implementações particulares das classes de objeto contidas no pacote Jikes RVM, atualmente em sua versão 2.4.4, que, em verdade, corresponde à Jalapeño JVM.

Partindo-se do princípio dos requisitos do projeto Jalapeño JVM, pode-se entender, então, como esta VM funciona sem, todavia, entrar em detalhes mais profundos de sua implementação.

Explorar processadores de alto desempenho não é uma tarefa simples (ALPERN et al, 2000), pois, mesmo com o uso dos compiladores JIT das VMs atuais, as otimizações propostas por estes acabam sendo aquém do nível de otimização para um processador de alto desempenho. Para tanto, Jalapeño implementa um compilador de otimização dinâmico (espécie de JIT avançado e adaptado à realidade dos processadores de alta performance) que atende a este requisito.

A *escalabilidade em SMPs*, isto é, em computadores multiprocessados de memória compartilhada é importante devido à popularidade desse tipo de máquina para ambientes paralelos na função de servidores. As JVMs atuais mapeiam as threads de aplicação Java diretamente como threads de sistema operacional (normalmente agrupadas num só processo, isto é, numa só entrada na tabela de processos do SO), dessa forma, o recurso de multithreading da linguagem Java acaba provendo baixo desempenho e uma escalabilidade pobre, que não aproveita bem os recursos paralelos do hardware. Jalapeño executa uma melhor paralelização dos threads Java, através de uma implementação própria de sistema de gerenciamento de threads, que executa a efetiva distribuição dos threads e provê a performance esperada de aplicações Java paralelas que se apóiam na utilização do *multithreading*.

A rotina de *servidores de alta disponibilidade* (ou também chamada *disponibilidade contínua*) exige que o atendimento a muitas milhares de requisições simultâneas seja feito. (ALPERN et al, 2000) afirma que devido à restrições de sistemas operacionais, algumas JVMs não conseguem criar um grande número de threads para atender à todas as *requisições de entrada*. Essa limitação de sistemas operacionais que, por sua vez, acaba por afetar as

JVMs limita em muito o uso de JVMs como máquinas virtuais para operarem como servidoras de aplicação. Jalapeño resolve também esse requisito através do já citado sistema de gerenciamento de threads próprio dessa VM em particular, e, acaba por aumentar a responsividade do servidor baseado em JVM no quesito requisições de entrada.

Como outro requisito muito procurado em JVMs paralelas, *a disponibilidade contínua,* é um aperfeiçoamento que a Jalapeño também cobre. Tal requisito é notável, pois, um servidor de aplicações deve operar com um grande número de requisições de entrada (conforme explicação anteriormente citada) e manter-se executando em modo on-line por longos períodos. Apesar disso não ser uma prioridade na implementação de VMs, atentou-se para esta necessidade e procurarou-se resolvê-la com a Jalapeño VM através dos tipos de dados auxiliares Java, e, com a chamada *prevenção e gerenciamento automático de armazenamento em memória* visando a ocorrência de ponteiros **nulos** ou **oscilantes**, assim, evitando os chamados *vazamentos em armazenamento de memória.*

Repostas rápidas são esperadas por qualquer software que se diga digno da apreciação dos usuários de computação em geral, pois, não é comum usar-se um software que *resolve um problema da forma mais lenta possível*. Para ser considerada rápida uma VM deve, normalmente, atender à noventa por cento das requisições em menos de um segundo. Segundo os autores (ALPERN et al, 2000), devido à sistemática usada atualmente nos Garbage Collectors das JVMs, o que ocorre são grandes atrasos e tempos de resposta falhos, gerando muitos ***timeouts*** desnecessários, por falta de eficiência da JVM. No projeto de Jalapeño VM, procurou-se resolver tal requisito através de *algoritmos de gerenciamento de memória concorrente e incremental*.

O *uso das bibliotecas Java* nativas das JVMs comuns, como por exemplo a VM da Sun Microsystems implica na problemática de que as mesmas foram escritas para propósitos genéricos que não incluem os da execução paralela, sendo assim, ao se forçar o uso das

mesma nesse estilo diferente de execução torna o desempenho baixo. Para cobrir este requisito, Jalapeño implementa transformações de especialização em seu *compilador otimizado e dinâmico* (em verdade, um compilador JIT avançado) que aperfeiçoa o compilador, de modo que o mesmo emenda o código dinamicamente compilado com uma biblioteca para o contexto de chamada do servidor de aplicação (ALPERN et al, 2000).

Evitar a degradação do servidor de aplicação controlado por uma JVM também é um requisito desafiador. ALPERN et al(2000) afirmam que apesar da degradação ser um fato normal para um servidor que é amplamente solicitado, no projeto da Jalapeño VM não encontrou-se um meio de programar a VM que provesse a garantia da não degradação do servidor de aplicação, assim, é por isso que eles optaram por concluir que Jalapeño apenas fornece *degradação amenizada*.

3.10. Comparativo entre as propostas e projetos de máquinas virtuais

Estabelecer parâmetros de comparação, faz-se uma prática necessária para uma melhor análise do estudo praticado a respeito das máquinas virtuais paralelas Java. A tabela 2 apresenta vários aspectos básicos bastante procurados em cada modelo ou projeto de máquina virtual. Tais aspectos influenciam na performance de cada máquina virtual, bem como podem ser, na verdade, aspectos específicos procurados por certos programadores para as suas aplicações.

Tabela 2: *Tabela comparativa entre algumas máquinas virtuais paralelas Java.*

Máquina Virtual	Sist. De Comunicação	Trabalha com processos	Trabalha com threads	Particularidades
HPJava	MPI, com descritor de arrays	Sim	Não	Performance parecida com o HPFortran. Usa primitivas de SPMD. Usa descritor de arrays automático para contornar as complexidades de certas computações.
JavaSpaces	jini e RMI	Sim	Sim	Usa o conceito de espaços de nomes. Mescla o uso de tecnologias de rede: jini e RMI. Permite implementação de serviços de rede no ambiente distribuído.
Titanium	MPI	Sim	Não	Apresenta alta performance através do uso de primitivas SPMD. Gerenciamento de memória distribuído (distinto do padrão Java, é baseado em zonas).
JavaNOW	RMI	Sim	Sim	Suporte a computação baseada em threads ou processos. Modelo de memória associativa compartilhada.
ManTA	ManTA RMI	Sim	Sim	Possui compilador próprio para os códigos ManTA paralelos. Possui implementação própria do RMI para uma performance mais alta. Possui um runtime próprio e distinto daqueles distribuídos nos pacotes Sun JSDK.
JavaParty	RMI ou JP-RMI	Sim	Sim	Possui compilador próprio para os códigos JavaParty. Usa de técnicas de otimização de RMI para elaborar os bytecodes finais, após a compilação. Isola o programador da complexidade do ambiente paralelo oferecendo a palavra reservada remote como agente facilitador.
Sun Microsystems J2SDK	RMI e RMI-IIOP	Sim	Sim	Máquina Virtual Java de uso mais difundido em todo o mundo. Muito usada para ambiente acadêmico e para aplicações comerciais simples. Possui a maioria dos recursos da API Java da Sun Microsystems.
Kaffe JVM	RMI	Sim	Sim	Máquina Virtual Java de código aberto (Software Livre), programada em C e C++, implementa quase todos os recursos da API Java da Sun Microsystems e possui boa compatibilidade com a maioria dos bytecodes gerados por outras JVMs.
Jalapeño JVM	RMI	Sim	Sim	Possui como recursos inovadores: array de elementos, campos e métodos. Compilador de otimização dinâmica, multiplexação de threads Java através de processadores virtuais, esquema de gerência de memória compartilhada avançado, e, provê recursos melhorar a escalabilidade em máquinas SMPs.
Virtual Virtual Machine	RMI ou MPI	Sim	Sim	Distribui, além de bytecodes de classes, instâncias da própria máquina virtual pelo ambiente paralelo e distribuído. Implementa facilidades para acoplamento da JVM a um kernel de sistema operacional.

Ainda tratando das informações contidas na tabela 2, na última linha aparece a VVM (Virtual Virtual Machine), máquina virtual Java distribuída que será apresentada nesta dissertação, no capítulo que trata da implementação. A VVM é uma tentativa de criar mais do que uma máquina virtual distribuída, em verdade, o que se deseja é que a máquina virtua Java possa ser distribuída de forma efetiva até um outro host. Dessa forma, pode-se usar um paradigma de máquina virtual diferente do atualmente praticado pelas JVMs Java, pois, estarão sendo distribuídas ***instâncias de máquina virtual*** (considerou-se como instância o

conjunto formado por um carregador de classe (*class loader*) e mais os bytecodes do programa que se deseja executar), possibilitando que um host cliente do ambiente distribuído receba a instância e execute os bytecodes. Concluindo, se tal realidade for materializada, no futuro os sistemas operacionais poderão executar bytecodes sem contudo possuírem instalada em seu disco rígido a JVM.

3.11 Conclusões sobre Modelos e Projetos de Máquinas Virtuais

HPJava é uma tecnologia arrojada no que se diz respeito a controle da aplicação paralela em código-fonte Java, contudo, é fato que a complexidade de implementação aumenta. Essa tecnologia usa de primitivas SPMD com comunicação em MPI. Para alocação de vários processadores simultaneamente, bem como para a criação rápida de um Grid Computing a partir de poucas linhas de código.

JavaSpaces usa-se da tecnologia JINI da Sun Microsystems para se beneficiar dos recursos de criação de serviços de rede, e, do uso estratégico de espaços de nomes usados como mídia na busca e uso dos recursos paralelos e distribuídos. Para a comunicação, JavaSpaces ainda usa do RMI Java nativo para as operações remotas. JavaSpaces possui uma formalidade de implementação muito bem construída, e, não requer muita habilidade com a paralelização da aplicação, desde que o programador saiba operar corretamente com os espaços de nomes, serviços, entidades e outros detalhes particulares que devem ser implementados em adendo ao código normal Java.

Como a própria definição de seus autores, o Titanium é um versão paralelo da linguagem de programação Java que traz consigo uma série de inovações. Tal tecnologia para paralelização de aplicações Java usa o modelo SPMD (*Single Program Multiple Data*) como premissa de implementação de todos os seus recursos. Pelas informações constantes do website dos autores do Titanium, pode-se desenvolver vários tipos de aplicações paralelas com o uso das bibliotecas de classes de funcionalidades fornecidas com o Titanium. Além do

modo compilado/interpretado há também a versão que converte o código para binários em C visando uma otimização e um tempo de execução mais rápido.

JavaNOW vem com muitas características que distinguem este projeto dos demais estudados neste capítulo. Trabalha tanto com processos quanto com threads, visando aumento do paralelismo da aplicação, na questão da comunicação pode funcionar com passagem de mensagem ou com modelo de memória associativa. Oferece suporte ao modelo de fluxo de dados (*data flow model*), característica não encontrada na maioria dos outros projetos de máquinas virtuais paralelas.

O projeto Manta vem com uma arquitetura de desenvolvimento própria e completa: oferece um compilador próprio para as instruções adicionais que implementa na linguagem Java, e também, oferece um ambiente de Runtime específico para os bytecodes produzidos para serem rodados em ambientes paralelos e distribuídos. Possui uma implementação singular do RMI que desempenha em rede uma comunicação bem mais efetiva do que as realizadas em outros projetos como JavaParty e RMI Java tradicional.

O mais notável dos projetos encontrados neste estudo, JavaParty é a tecnologia de máquina virtual Java que mais incorpora os elementos essenciais para alto desempenho em execução e para facilitar a codificação por parte dos programadores de aplicações paralelas. Com recursos avançados em compilação, e, sem a necessidade de envolver o programador na organização da paralelização (isto é, no sincronismo), JavaParty com a palavra-reservada `remote` consegue realizar a maioria das expectativas que os programadores esperam para a interação com um ambiente paralelo e distribuído.

A partir do estudo de modelos e projetos de máquinas virtuais, pode-se verificar que a construção de máquinas virtuais Java tornou-se uma grande área em pesquisas, as JVMs de código fechado, como, por exemplo, a pertecente à Sun Microsystems, ainda possuem maior credibilidade, desempenho, suporte e API maior. As JVMs de código aberto tornaram-se

muito difundidas no meio científico, contudo, em sua maioria, são implementações em fase de amadurecimento ainda não completas.

A comunidade de Software Livre tem se mobilizado para mudar essa realidade. Isso se faz necessário porque, muitas vezes, procura-se um recurso ainda não presente na JVM que, se for de código fechado, não há como ser providenciado, e, ficará dependente da empresa fabricante colocar ou não o recurso nas futuras versões.

Kaffe VM parece ser uma JVM de futuro, devido à grande adesão que tem conseguido na comunidade de usuários GNU/Linux. A JVM da Sun Microsystems continuará a ser a mais preferida entre as JVMs, contudo, com o tempo será mais focada em aplicações comerciais e para Web, assim, ficando as aplicações mais simples, desktop, acadêmicas e outras para a fatia de mercado das VMs opensource.

A Jalapeño VM, a Virtual Machine do projeto Jikes RVM, apoiada pela IBM Corporation é um exemplo de grande êxito na busca do alto desempenho em aplicações paralelas. Implementada totalmente em Java, a Jalapeño VM é uma JVM paralela que é executada sobre uma JVM comum. Seus modelos de gerenciamento de memória e layout de objeto são avançados, assim, escolheu-se a Jalapeño VM para ser a Virtual Machine de base para as comparações e benchmarking com a VVM (*Virtual Virtual Machine*).

3.12. Medidas de Benchmarking em Projetos de Máquinas Virtuais Paralelas

Para testar as distintas máquinas virtuais em relação ao desempenho na execução de aplicações paralelas em ambientes distribuídos é interessante que se tenham bons *softwares de benchmark*.

Performance não significa somente o menor tempo de execução, existem muitos aspectos envolvendo os softwares de *benchmarking*, por isso, muitas vezes usa-se mais de um benchmark para um mesmo teste para verificar as diferenças de desempenho. Estudar os

vários benchmarks disponíveis no mercado auxiliou na escolha de um benchmark para estabelecer os parâmetros de comparação entre a VVM (Virtual Virtual Machine) e a Jalapeño Virtual Machine.

Enquanto as técnicas existentes de execução são melhoradas, novas técnicas estão sendo desenvolvidas, os *benchmarks* servem como um veículo para fornecer a consistência entre avaliações para finalidades de comparação.

Para (KAZI et al, 1999), *aplicações de benchmark* são usadas para avaliar performances de sistemas globais. *Microbenchmarks*, por outro lado, são usados para avaliar a performance de sistemas individuais ou de características de linguagens, tais como o armazenamento de um inteiro numa variável local, incrementação de um byte, ou a criação de um objeto. No trabalho destes autores, fala-se com muita propriedade sobre os diversos benchmarks disponíveis no mercado para testes com Java bem como uma série de outros tópicos sobre esta linguagem de programação emergente.

O *Open Systems Group* (*OSG*) do *Standard Performance Evaluation Corporation* (SPEC) desenvolveram uma série de Java benchmarks denominada *SPEC JVM98*, que mensura a performance da JVM. O *SPEC JVM98* é formado por 8 classes Java distintas que reproduzem o comportamento de 8 aplicações: mtrt, jess, compress, db, mpegaudio, jack, javac, Geometric Mean. Os tipos dos algoritmos supracitados são: algoritmo de compressão, simulação de aplicação de banco de dados, descompressor de áudio (*mpegaudio*), multiplicação de matrizes, . (RADHAKRISHNAN, 1999), usou o *SPEC JVM98* em seu trabalho, e, revela que, na prática, dependendo dos tipos distintos de aplicações usadas, elas estimulam as VMs de formas diferentes que podem levar a comportamentos inesperados.

Além deste, há também o *SYSmark,* um outro benchmark que consiste em quatro classes Java: *JPhotoWorks, JNotePad, JSpreadSheet e MPEG.*

O *CaffeineMark* (PENDRAGON, 2006), é também um *software de benchmarking* desenvolvido para testar aspectos específicos da máquina virtual Java. Ele testa a performance da JVM nos aspectos de: performance com algoritmos de *looping*, execução de instruções do tipo *decision-making* (testes lógicos), teste com chamadas de funções recursivas (ou teste de métodos). Para sistemas embarcados há ainda o *Embedded CaffeineMark* (PENDRAGON, 2006) que não contempla os testes gráficos.

Para aplicações que seguem um modelo Cliente/Servidor há o *VolanoMark* (VOLANO, 2006), desenvolvido para testes de alta performance da JVM frente a aplicações multithread e distribuídas em ambiente de rede.

(KAZI et al, 1999) afirma que há muitos outros benchmarks que podem ser encontrados para os testes com JVM como: *Java Grande Fórum Benchmark, Linpack Benchmark, JavaLex (gerador de analisador léxico), JavaCup (combina múltiplos arquivos em um único arquivo sem usar compressão), JHLUnzip (usado em par com o JHLUZip para descomprimir arquivos), JByte, Symantec Benchmark, Dhrystone CPU Benchmark, Jell, Jax, EspressoGrinder, JMark, entre outros.*

Enfim, conclui-se que a escala de medição para cada um dos benchmarks é diferente, e, normalmente seus desenvolvedores mantém tabelas de comparação em que usuários da Internet colaboram colocando seus dados de teste. Estes dados são então os parâmetros para as pesquisas futuras de outros usuários dos benchmarks.

3.13 Conclusões sobre Benchmarking para Máquinas Virtuais

Os softwares de benchkmarking normalmente estão presentes em pesquisas que envolvam a produção de software que vise, em suma, atingir alta performance. Além disso, através de benchmarkings, pode-se mostrar a viabilidade ou não do uso de um novo software através da comparação das marcas de benchmark do mesmo com as marcas de benchmark de outros softwares de igual categoria já existentes.

Com o estudo dos benchmarks pode-se chegar a algumas conclusões: muitas vezes não compensa produzir o seu próprio software de benchmark, pois, já existem muitos que estão sendo usados como padrão pela comunidade de pesquisadores. Para cada tipo de aplicação existe sempre mais de um benchmark sendo usado, logo, a escolha pode ser por fatores como: popularidade, software proprietário (pago) ou software livre, tipo de benchmark desejado (multiplicação de matrizes, polinômios, gráficos, compressão de dados, transação e muitos outros), instituto ou grupo que mantém o projeto do benchmark ativo, entre outros.

4 - IMPLEMENTAÇÃO DE UM PROTÓTIPO DE MÁQUINA VIRTUAL JAVA DISTRIBUÍDA

Vários pesquisadores tentaram soluções para o problema do paralelismo de objetos em sistemas distribuídos, dentre estas implementações estão as classes de objeto para paralelismo, compiladores Java estendidos (com instruções adicionais no código-fonte Java), o uso de novas implementações da Invocação Remota de Métodos (RMI), passagem de

mensagem tradicional ou com implementações proprietárias, criação de ambientes paralelos especiais e máquinas virtuais paralelas

O porém da questão destes softwares e pesquisas é que todas as soluções propostas possuem limitações, isto é, resolvem alguns problemas, porém possuem seus problemas intrínsecos do ambiente, por exemplo, os problemas de comunicação.

Esta dissertação utiliza da análise feita nos projetos mais populares a respeito de: máquinas virtuais, mecanismos de paralelização em software, ambientes e classes de objeto para a paralelização de aplicações Java voltadas para computação de alto desempenho. O objetivo da análise foi fornecer os parâmetros para a formalização das idéias para a implementação desta dissertação.

Não se optou por implementar uma JVM completa pelos seguintes fatores: tempo de implementação e pelas *oportunidades* que os atuais projetos e tecnologias em andamento oferecem.

O que se intenciona ao final é oferecer uma forma amigável e de fácil programação para que os objetos das aplicações paralelas tenham um dinamismo maior para paralelização da aplicação, isolando o programador da preocupação de ter de pensar como o paralelismo agirá com efeito sobre seu código de programa. A Virtual Virtual Machine, uma máquina virtual Java distribuída, vem com a proposta de paralelizar e distribuir não só as aplicações como também a si mesma. A idéia da VVM foi implementada porque verificou-se, pelos estudos anteriores, que não há uma JVM que distribua instâncias de si mesma pela rede, normalmente, em cada máquina do ambiente distribuído há uma *cópia do software da JVM instalada*. Distribuir a JVM significa enviar para os outros hosts uma instância de carregador de classe mais os bytecodes que se deseja executar, em anexo.

4.1 Process VMs e System VMs

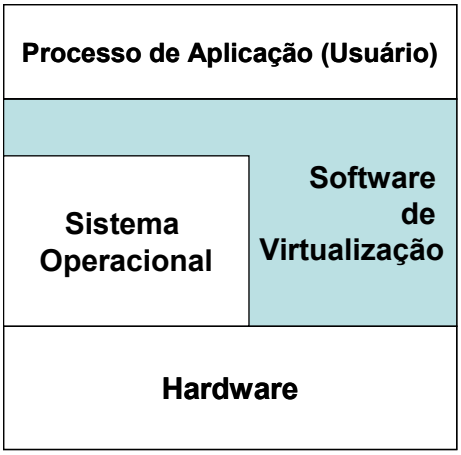
Publicações recentes (SMITH & NAIR, 2005) mostram novas perspectivas de entendimento e desenvolvimento das máquinas virtuais (VMs). As máquinas virtuais podem ser de dois tipos, cujas características são bem distintas e definidas: *Process VMs* (ou máquinas virtuais orientadas a processos) e *System VMs* (ou máquinas virtuais orientadas a sistema).

Uma *Process Virtual Machine* é constituída por uma *software de virtualização*, que traduz um conjunto de instruções de sistema operacional e instruções em nível de usuário, compondo, dessa forma, uma conversão completa de uma plataforma para outra.

Logo, uma Process VM é capaz de executar programas desenvolvidos para diferentes sistemas operacionais e diferentes Conjunto de Instruções de Arquitetura (ISA - *Instruction Set Architeture*).

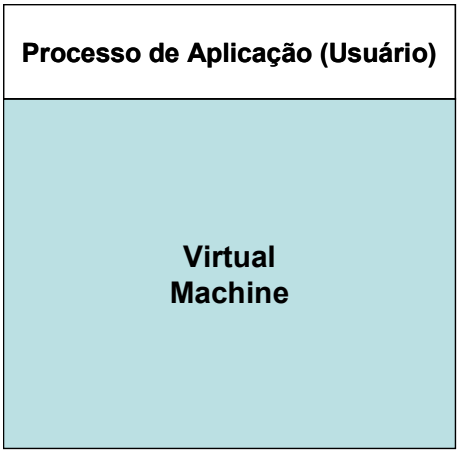
A figura 14 ilustra uma Process VM: o processo de aplicação ou processo de usuário fica na camada mais exterior e é também conhecido como visitante (Guest), o ambiente de Execução (Runtime Environment) é composto pelo software de virtualização (que consiste na implementação da VM propriamente dita). O conjunto formado pelo Hardware (e seu conjunto de instruções ISA) e o sistema operacional são chamados de *Host*.

Figura 14: *Um Process VM e suas camadas: (SMITH & NAIR, 2005).*



Para o processo de aplicação, que, por sua vez, fica em contato com a superfície do software de virtualização para toda e qualquer interação com a máquina em questão, as camadas inferiores formam uma grande máquina virtual, como mostra a figura 15.

Figura 15: *Process VM vista como um processo de aplicação:* (SMITH & NAIR, 2005).

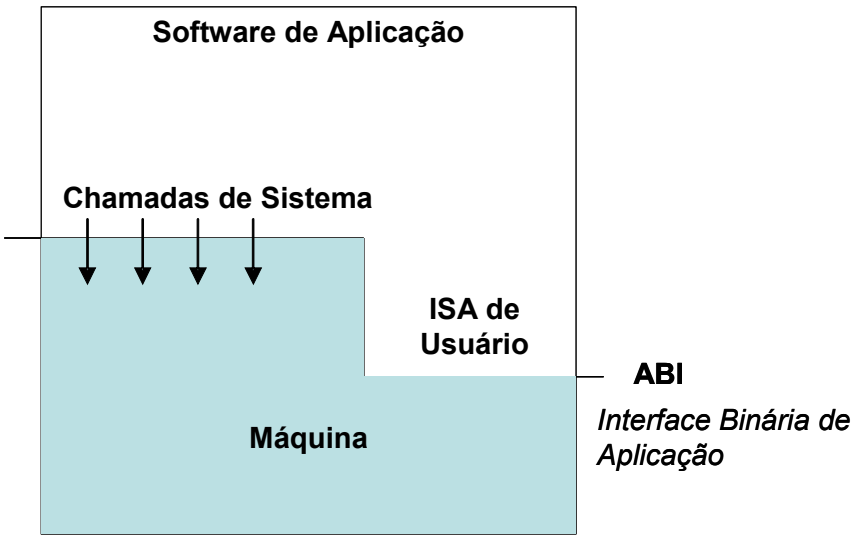


Para (SMITH & NAIR, 2005), há também duas perspectivas de visão da máquina: uma do *processo* e outra do *sistema operacional*.

Do ponto de vista de processo, ilustrado na figura 16, a máquina consiste em espaço de alocação de memória (que é fundamental para um processo), registradores de instruções e de dados, o subsistema de Entrada/Saída é visível ao processo apenas através de chamadas de sistema, logo, e, concluindo, a perspectiva da máquina para um processo é composta do sistema operacional e do hardware a nível de usuário que está subjacente ao sistema operacional. Seguindo a linha de raciocínio, a perspectiva da máquina a partir do sistema operacional é um tanto quanto diferente: a máquina precisa suportar um sistema completo de forma organizada, executando muitos processos de vários usuários diferentes de forma quase simultânea. Além disso, deve dividir entre os diferentes processos todos os recursos existentes no subsistema de Entrada/Saída sem deixar que deadlocks ocorram. O interfaceamento da máquina (hardware própriamente dito) e o sistema (ou sistema operacional) é realizado

através do chamado ISA (Instruction Set Architeture) ou Conjunto de Instruções da Arquitetura de Computador em questão.

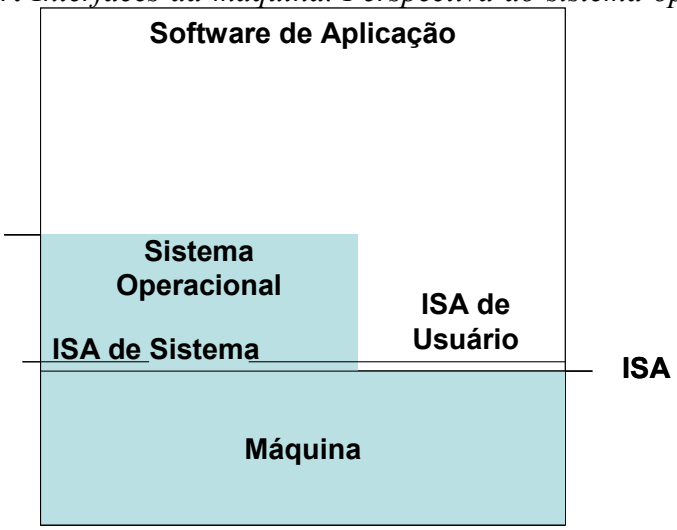
Figura 16: *Interfaces de uma máquina. Perspectiva do processo de aplicação:*



(SMITH & NAIR, 2005).

O trabalho de SMITH & NAIR (2005), mostra que a visão do sistema operacional é de intermédio entre as camadas mais superiores de software (mais alto nível) e as camadas mais inferiores (mais baixo nível), como pode-se visualizar na figura 17. Contudo, além de mediar as camadas também precisa realizar, em parceria com a máquina (hardware), uma administração e performance adequada às expectativas dos usuários. O sistema operacional é considerado como a máquina virtual existente entre a máquina (hardware / ISA) e os programas de usuário.

Figura 17: *Interfaces da máquina. Perspectiva do sistema operacional: (SMITH & NAIR, 2005).*



As visões e interfaces mostradas a respeito das Process VMs são particularmente importantes, pois, mostram as camadas em que o software de virtualização irá atuar. Assim, pode-se visualizar o como foi idealizada a implementação que se desenvolveu para esta dissertação, ou seja, pensou-se em camadas *de virtualização* principalmente.

A máquina virtual distribuída a qual é proposta nesta dissertação funciona como uma *Process VM*, com a camada de software de virtualização sobrepondo o sistema operacional. O protótipo que foi implementado funciona como uma nova camada acima da máquina virtual Java, ou seja, tal protótipo é uma VM sobre outra VM, logo, a VVM (Virtual Virtual Machine) é uma máquina virtual Java distribuída que executa sobre uma máquina virtual Java comum da Sun Microsystems.

4.2 Arquitetura de uma Máquina Virtual de Alto Nível

A teoria de (SMITH & NAIR, 2005) a respeito da criação de arquiteturas de máquinas virtuais de alto nível é ampla e envolve todos os aspectos estruturais, de segurança, conjunto de instruções, organização de memória e também um tópico que aborda aspectos particulares da máquina virtual Java. Para este estudo em particular, o que nos interessa focar são conceitos elementares desse tipo de arquitetura, visando mostrar que a Virtual Virtual

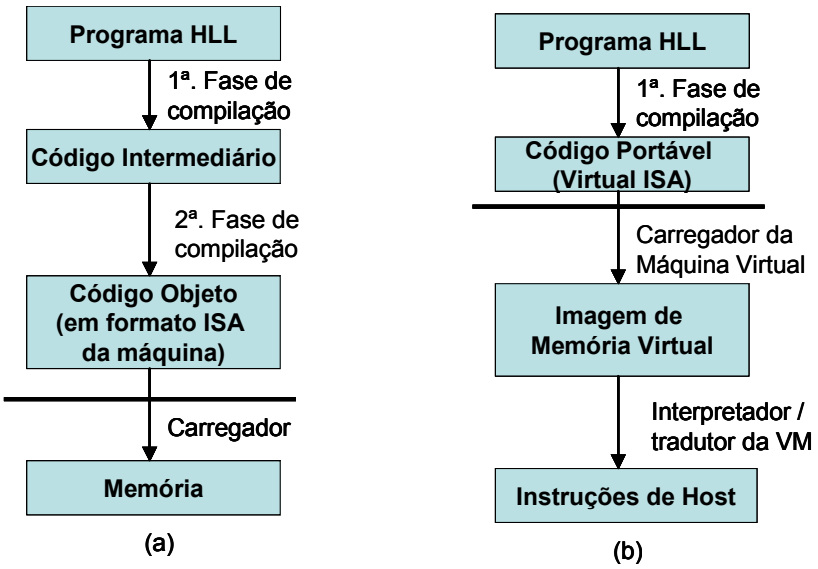
Machine (VVM) obedecerá a estes critérios, de forma a validar o modelo de implementação de máquina virtual Java paralela e distribuída que está se propondo.

Fato importante que ajudou a impulsionar a tecnologia das máquinas virtuais de alto nível foi a portabilidade (SMITH & NAIR, 2005) desse tipo de VM, pois, como o conjunto de instruções das VMs é o mesmo para várias as plataformas, as diferentes implementações possuem uma interoperabilidade harmoniosa.

Esse tipo de máquina virtual que está sendo enfatizado nesse trabalho, ou seja, as *Process VMs*, são também chamadas na literatura de **HLL VMs** (*High-Level Language Virtual Machines*), tendo na *Java Virtual Machine* a mais popular das suas representantes. Para se entender melhor como é a arquitetura desse tipo de máquina virtual, (SMITH & NAIR, 2005) propõem uma figura ilustrativa, que é reproduzida pela figura 18.

O processo de compilação e execução representado pela *alternativa A* da figura 18 ilustra o formato transformação de um código-fonte alto nível para um formato executável (binário) que usa o conjunto de instruções da arquitetura da máquina. Apesar de haver uma primeira fase de compilação onde gera-se um código intermediário (que as vezes aproxima-se do formato dos bytecodes), esse tipo de compilação tem por meta a criação de um código objeto completamente legível àquela arquitetura de máquina (formato ISA da máquina).

Figura 18: *Comparação entre uma arquitetura convencional de execução de programas (a) e a arquitetura de uma máquina virtual de alto nível (b): (SMITH & NAIR, 2005).*



Já o processo demonstrado pela *alternativa B* da figura 18 ilustra o novo panorama adotada para a maioria das tecnologias de linguagens e sistemas que se utilizam de máquinas virtuais de arquitetura de alto nível. É importante que se perceba que, já na primeira fase de compilação, o resultado obtido são bytecodes (formato de código intermediário entendido pela JVM), chamados genericamente como ***código-portável*** (ou ainda, Virtual ISA), dessa forma, esse código-portável, desprendido de quaisquer características proprietárias das arquiteturas de computadores existentes, torna-se neutro o suficiente para ser executado por qualquer Virtual Machine que consiga interpretar os bytecodes do programa. Assim, a terceira e quarta etapas (terceiro e quarto quadrados) da *alternativa B* mostram as tarefas executadas pela VM para proceder a efetiva execução do programa. Essa estrutura, apesar de simples e trivial, é fundamental para que qualquer projetista possa se basear, quando da intenção de implementação de uma máquina virtual que queira oferecer portabilidade e uma organização padrão de compilação e execução que é atualmente adotada.

A Virtual Virtual Machine, irá se utilizar, de forma efetiva, da arquitetura representada pela alternativa B da figura 18, pois, em verdade, a VVM será uma VM sobre

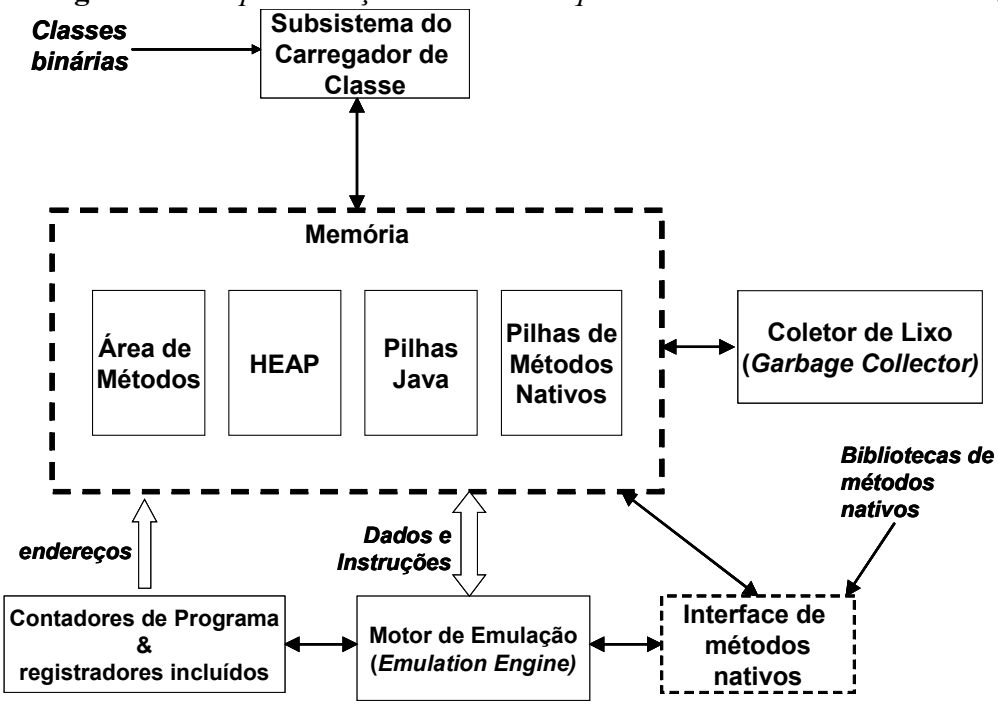
outra VM, isto é, uma máquina virtual Java distribuída suportada por uma máquina virtual Java comum,isto é, uma Sun Microsystems J2SDK Virtual Machine. Logo, como a VM da Sun Microsystems já está adequada aos moldes de compilação/execução demonstrados na *alternativa B* da figura 18, a Virtual Virtual Machine usa a mesma arquitetura. Com essa abordagem a VVM demonstra atualidade em sua organização estrutural e funcional.

4.3 Implementação de Máquinas Virtuais de Alto Nível

Atualmente há um padrão moderno para implementação de máquinas virtuais de alto nível. A JVM é também implementada (SMITH & NAIR, 2005) através do estudo e da compreensão dos componentes que formam esse padrão, pode-se entender, como funciona a Java Virtual Machine ou qualquer outra implementação através do estudo de *HLL VMs*.

É mostrada a seguir uma figura descritiva de como é a implementação tradicional de uma máquina virtual Java, conforme convenção dos projetistas de todo o mundo. Para se entender os detalhes da figura 19, (SMITH & NAIR, 2005) fornecem, em seu trabalho, uma discussão detalhada como segue: *a área de memória* (quadrado pontilhado maior) é onde se alocam o código de programa, uma área global de memória e as pilhas para código Java e código nativo das bibliotecas. É importante citar que a máquina virtual Java de acordo com esse *modelo de implementação* possui a sua própria forma de gerenciamento e alocação de memória. Os *contadores de programa e registradores*, ficam alocados em diferentes locais de memória que não aqueles administrados pela JVM (Java Virtual Machine). Contudo, tais contadores e registradores só podem ser acessados, para fornecerem dados e atualizarem seus valores, pelos seus respectivos programas em execução. A *pilha de execução Java* já foi comentada nesta dissertação no *segundo capítulo*. É interessante que se repare que as pilhas de execução de código Java são separadas das *pilhas de execução dos métodos nativos*, não só por questão de organização, mas também, por questão de eficiência.

Figura 19: *Implementação de uma máquina virtual Java tradicional. (SMITH & NAIR, 2005).*



Ainda na figura 19, a memória global mencionada é alocada dinamicamente num espaço de memória específico denominado *HEAP*. Para salientar melhor o uso do HEAP, por exemplo, quando um novo objeto é instanciado, sua memória é alocada a partir do espaço de memória do HEAP, lembrando que o HEAP aloca memória dinamicamente e não possui um tamanho pré-definido. Objetivando um desempenho ainda melhor da JVM, o Coletor de Lixo (*Garbage Collector*) entra em ação para limpar todas as áreas da memória administrada pela JVM, onde existam dados e informações que já não estão mais sendo usados por programa algum. A injeção de métodos nativos nas pilhas de memória se dá a partir da leitura de tais métodos nas chamadas *bibliotecas de métodos nativos*. Após essa leitura, através da *interface de métodos nativos* é realizado o carregamento dos métodos nativos para a memória da JVM. Entre as unidades dos *Contadores de Programa (PCs) e registrados,* e também, da *Interface de Métodos Nativos*, encontra-se o chamado **Motor de Emulação**, que, na realidade usa-se da memória e de todos os demais elementos da JVM para desempenhar a efetiva execução dos programas, pode-se dizer que esta unidade seria o equivalente ao processador da JVM. É no

motor de emulação que ocorre a interpretação ou tradução para código nativo (compilação JIT) dos bytecodes Java (SMITH & NAIR, 2005).

Pela *popularidade* e pelo *alto desempenho* que este modelo de implementação de máquinas virtuais de alto nível possui com a tecnologia de linguagem de programação Java, no projeto da VVM (Virtual Virtual Machine) a implementação escolhida foi a de um HLL VM, pelos benefícios oferecidos e comentados a respeito dessa metodologia de implementação para máquinas virtuais.

4.4 Arquitetura da VVM

A arquitetura da VVM (*Virtual Virtual Machine*) difere um pouco da arquitetura demonstrada por (SMITH & NAIR, 2005) na figura 18, contudo, é interessante ilustrar alguns detalhes da arquitetura VVM.

Do ponto de vista arquitetural, a VVM é uma *Process VM*, sendo também, portanto, classificada como *Máquina Virtual de Alto Nível*. Possui a característica particular de ser uma VM distribuída, além disso, é uma VM que intenciona a distribuição de instâncias de si mesma no ambiente distribuído. Esse é um ponto chave do projeto arquitetural da VVM, pois, a maioria das máquinas virtuais Java paralelas e distribuídas, não implementam o recurso de enviar uma instância da própria VM até uma máquina remota (que, de preferência, não possua a JVM instalada em seu disco rígido). É interessante que fique claro que há vários obstáculos a serem transpostos para esse envio de instância da VVM para uma outra máquina.

A figura 20 ilustra as diferenças entre a arquitetura de máquina virtual em alto nível comum (*alternativa A*) que é usada, por exemplo, nas máquinas virtuais da Sun Microsystems e Kaffe JVM, e, a arquitetura de máquina virtual alto nível para ambientes paralelos e distribuídos (*alternativa B*).

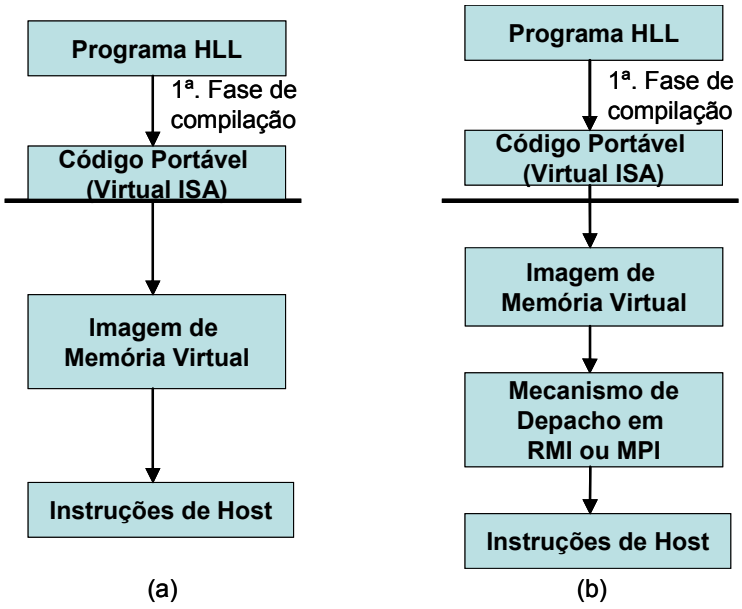


Figura 20: *Arquitetura de máquina virtual alto nível e arquitetura VVM.*

Observando-se, à primeira vista, a única modificação executada na arquitetura da *alternativa A* para a arquitetura da *alternativa B* é a camada de ***Mecanismo de Despacho (Dispatching Mechanism) em RMI ou MPI***, contudo, essa camada representa muito mais do que somente o despacho (ou seja, a distribuição de programas de bytecodes por uma rede). Em verdade, está implícito nessa camada todas as novas funcionalidades que o modelo de implementação da VVM irá empregar, objetivando, diferenciar a forma como a máquina virtual distribui, processa e devolve os resultados da execução de aplicações. Além disso, há também a intenção de trabalhar, neste nível, de forma mais próxima do sistema operacional.

4.5 Modelo de Implementação da VVM

O modelo de implementação que será detalhado a seguir, trata-se de uma explicação em torno das tecnologias e do funcionamento do protótipo de software desenvolvido. Essa explanação está detalhada o suficiente para que se possa reproduzir o modelo de implementação em outras situações.

O modelo de implementação da VVM (Virtual Virtual Machine) será explicado em etapas: o nível de paralelismo em que atua a VVM (figura 21), posteriormente são explicadas

as tecnologias que foram selecionadas para comporem a VVM, e, na etapa final, explica-se como foi implementado o protótipo da VVM usando a linguagem de programação Java.

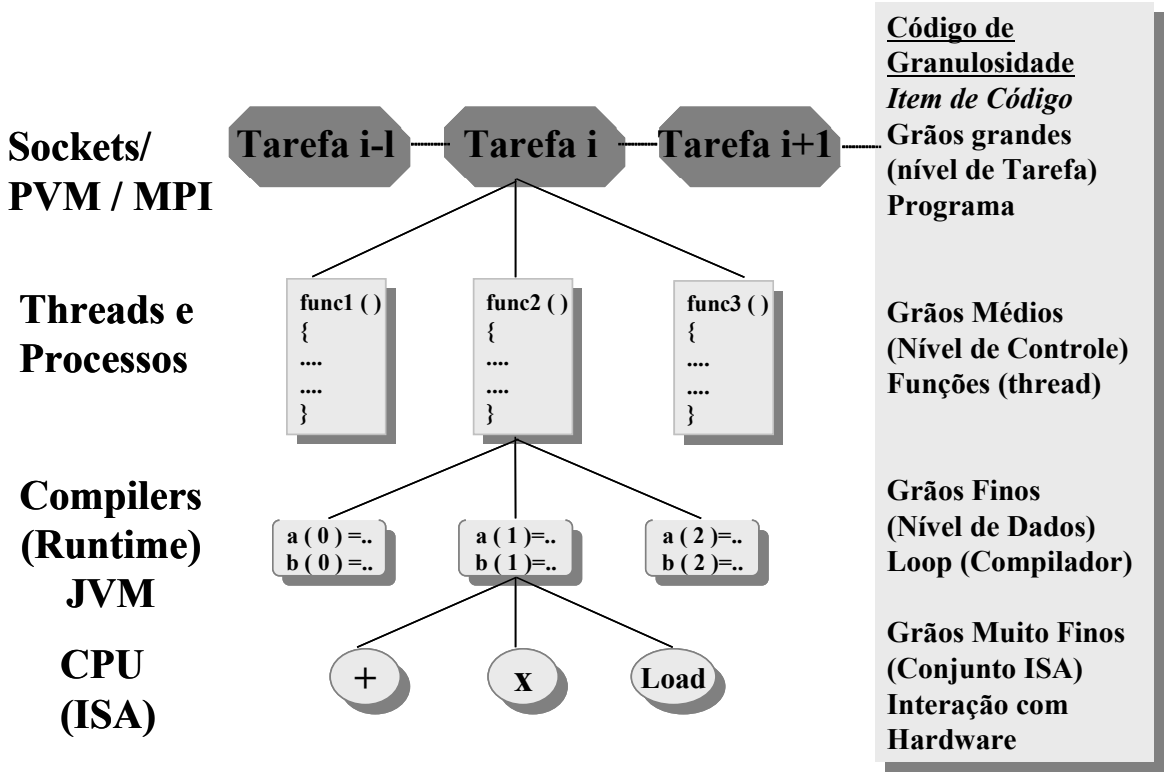


Figura 21: *Diferentes níveis de paralelismo e a VVM.*

De acordo com a figura 21, pode-se visualizar quatro diferentes níveis de paralelismo, citados como ***Código de Granularidade***. Será realizada uma explicação *bottom-up* da figura 21: o nível de paralelismo mais inferior é o de *grãos muito finos*, que representa o paralelismo de conjunto de instruções de arquitetura de máquina (ISA), assim, esse nível acaba sendo o mais complexo de se implementar paralelismo do ponto de vista de programação, pois, exige um controle absoluto sobre o hardware que se está manipulando. O segundo nível, denominado de *grãos finos* já se refere ao paralelismo com relação aos **dados manipulados**, este tipo de paralelismo deve ser implementado com versões de compiladores, interpretadores e ambientes de *Runtime* especiais. Neste caso de paralelismo, em Java, teria-se que programar uma versão especial da JVM orientada ao paralelismo de dados. O nível de paralelismo denominado *grãos médios*, implica num controle executado sobre *threads e*

processos, usando-se da manipulação desses recursos como forma de se obter um paralelismo de maior performance. No último nível, *grãos grandes*, usa-se de cada **tarefa** (que pode ser composta por um ou mais threads ou processos) para implementar o paralelismo a nível de programa. Geralmente as tecnologias usadas no nível de *grãos grandes* são aquelas de comunicação interprocesso e comunicação de rede, tais como ***Sockets, MPI, PVM*** e outros.

A VVM, atua no nível de *grãos médios* podendo estender sua atuação até o nível de *grãos grandes*, pois, de qualquer forma, a VVM estará também se utilizando de tecnologias de comunicação como MPI e RMI para executar a distribuição de tarefas no ambiente paralelo e distribuído.

Após situar o nível de paralelismo de atuação da VVM, são ilustradas as tecnologias que fazem parte do modelo de implementação da VVM e que a tornam diferenciada dos demais modelos de máquinas virtuais demonstradas na revisão bibliográfica deste trabalho.

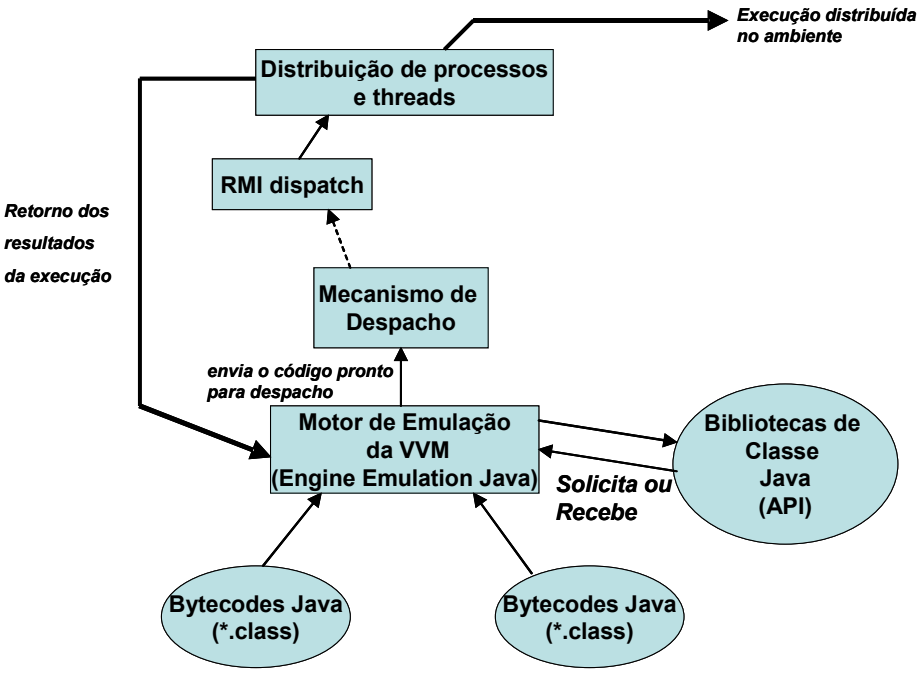


Figura 22: *Modelo de implementação da VVM (Virtual Virtual Machine).*

A figura 22 ilustra como é a organização das tecnologias que envolvem a VVM. Iniciando-se a explicação a partir do *motor de emulação da VVM*, é importante deixar claro que na realidade usa-se o mesmo Motor de Emulação da JVM (*Java Virtual Machine)* Java

instalada na máquina, pois, a princípio, a VVM está executando sobre uma JVM comum, por exemplo, a *Sun Microsystems J2SDK 1.5.X*. Esse motor de emulação da VVM carrega os bytecodes usando um carregador de classes da própria VVM (este detalhe será explicado a seguir) para criar os objetos do programa e *solicita e/ou recebe* das Bibliotecas de Classe Java (API Java) os demais objetos que serão executados de forma paralela e distribuída. Após o carregamento inicial, o código a ser executado já está previamente pronto para ser distribuído pelo ***Mecanismo de Despacho***, que pode escolher entre tecnologias diversas de comunicação (o release atual da VVM opera somente com RMI). Após essa seleção de tecnologia de comunicação, suponha-se que RMI seja escolhido, os bytecodes são despachados para a *máquina de destino* (escolhida de forma dinâmica) onde serão efetivamente executados. Vale lembrar, que, neste ponto, os bytecodes estão sendo distribuídos na forma de processos e threads no ambiente paralelo e distribuído, sabe-se que as diferenças de plataformas de sistemas operacionais podem se tornar incompatíveis, gerando vários conflitos e rejeições neste momento, todavia, como solução a este problema pensou-se no uso dos POSIX Threads para a distribuição dos bytecodes, pois, todos os sitemas operacionais modernos possuem aceitação dessa tecnologia em forma de API do sistema operacional ou mesmo na forma de bibliotecas que acompanham o SO. Continuando a explanação, junto com os bytecodes de programa é também enviada uma instância da VVM para a máquina destino. Este é o ponto onde o projeto da VVM torna-se ímpar em relação aos demais projetos. As distribuições de instâncias da VVM em conjunto com bytecodes de programa a serem executados, tornam a VVM uma tecnologia de máquina virtual de portabilidade maior do que a JVM comum, pois, pode-se enviar uma instância da VM para uma máquina que possuam somente um software *cliente* esperando pela chegada da VM e dos bytecodes a serem executados. Para concluir a explicação, após a execução paralela e distribuída no ambiente (via mecanismos de redes de computadores), é retornado ao host de onde partiu a solicitação de execução dos bytecodes o

resultado da execução, que poderá ser agrupado com outros resultados fragmentados para compor o *resultado final da execução.* Esse resultado final é gerado também pelo *Motor de Emulação da VVM* e então é mostrada a ***Saída dos Resultados da Execução*** conforme ilustrado na figura 22.

Tendo em vista como é o funcionamento da VVM, e, quais as tecnologias implementadas para que ela possa ser efetivamente uma máquina virtual Java distribuída **que pode ter instâncias de si distribuída**s, daí a inspiração do nome *Virtual Virtual Machine.*

O problema inicial foi conseguir um carregador de classes diferente do carregador de classes padrão das JVMs comuns. Presente na API 1.5 da Sun Microsystems Java Virtual Machine, a classe ***ClassLoader***, permitiu ao protótipo manipular sua própria instância de carregador de classes. Isto significa que a VVM pode sozinha efetuar o carregamento de um arquivo de bytecodes e iniciar sua execução sem a necessidade de usar-se o objeto carregador de classes padrão. Concluindo, a VVM pode, dessa forma, transmitir instâncias desse carregador de classes juntamente com os bytecodes a serem executados para outros hosts do rede.

Partindo dessa premissa, a VVM foi programada inicialmente, em seu primeiro release, sobre uma JVM comum, isto é, a VVM é uma Java Virtual Machine distribuída programada em linguagem Java.

Foi implementada uma classe denominada **CLVVM.java** (*Class Loader Of Virtual Virtual Machine*), cujo conteúdo básico é criar o objeto Carregador de Classe distinto do carregador de classes tradicional, distribuindo no ambiente qualquer classe Java que lhe for passado como argumento. Há também a classe **VVM.java** que é a classe na qual se estruturou a aplicação da *Virtual Virtual Machine* em si, ou seja, é com ela que o usuário interage a fim de passar, como parâmetro, as classes Java em bytecodes que desejar executar.

Na distribuição das tarefas, atualmente a VVM utiliza-se somente da tecnologia RMI (*Remote Method Invocation*) para executar o envio e recebimento de informações no ambiente paralelo e distribuído. Para o uso de RMI, nos códigos-fonte da VVM importa-se o pacote **java.rmi.***, objetivando utilizar-se de todos os recursos necessários a partir da *biblioteca de classes RMI Java.*

4.6 Viabilidade da VVM

A VVM é viável porque torna a tecnologia de linguagem Java ainda mais portável, partindo-se do ponto que está se tratando do projeto de uma máquina virtual distribuída que pode te instâncias de si distribuídas num ambiente de *Grid Computing* ou *Cluster* de computadores, por exemplo. O aumento de portabilidade é notável quando verifica-se a possibilidade de envio de carregadores de classe para outras máquinas que estarão rodando uma simples aplicação cliente que fica esperando pelas tarefas a serem executadas. Isso pode ser chamado de *novo paradigma de distribuição da própria VM por ambientes distribuídos*.

Além disso, VVM está se utilizando de tecnologias já existentes e consolidadas em Ciência da Computação, como a linguagem de programação Java, RMI, um carregador de classes estendido para as funcionalidades distribuídas, o uso de *Threads* e *Serialização* como incremento na distribuição e portabilidade da VVM, entre outras características.

Há ainda a intenção de se realizar o acoplamento da VVM a um kernel de sistema operacional, o que pode ser chamado também de ***novo paradigma de máquina virtual***, distinto de todos os revisados e demonstrados teoricamente pelas diversas pesquisas explanadas neste trabalho**.** Espera-se que, com estas premissas a VVM consiga se como projeto de *software livre*, para que ganhe novas contribuições e melhorias pela comunidade de programadores e usuários de máquinas virtuais distribuídas.

4.7 Comparativo entre as características da VVM e da Jalapeño VM

Comparar a VVM (*Virtual Virtual Machine*) com a Jalapeño VM é uma tarefa um tanto quanto delicada, tendo em vista que, cada projeto foca diferentes objetivos.

Discutindo individualmente o objetivo de cada projeto, a VVM objetiva, na verdade, realizar efetivamente a *distribuição de instâncias da VM* por um ambiente distribuído composto por qualquer arquitetura e topologia de rede, conforme explicações anteriores, é considerada instância de máquina virtual distribuída para esta dissertação o conjunto formado por um objeto carregado de classe somado aos bytecodes a serem executados que são efetivamente enviados para outros hosts por RMI e Serialização. Os benefícios de tal técnica é que o host só irá processar as informações usando-se, para isso, de uma aplicação cliente simples. Além disso, também se intenciona melhorar a comunicação paralela realizada no ambiente distribuído através do uso de RMI.

A Jalapeño Java Virtual Machine é uma máquina virtual paralela com objetivos específicos dentro do universo da computação paralela, logo, o que difere Jalapeño da VVM são, essencialmente, os requisitos focados pela implementação do Jalapeño, que já foram citados e, agora, são reforçados, isto é: *exploração de processadores de alto desempenho*, *escalabilidade em SMPs, trabalho diferenciado com limites de threads, disponibilidade contínua, respostas rápidas, uso de bibliotecas, degradação amenizada*.

Neste tópico, visou-se somente a comparação pura e simples das características de implementação de cada modelo de máquina virtual, colocando em contraste VVM e Jalapeño. Os projetos miram objetivos diferentes, contudo, por serem implementações de máquinas virtuais distribuídas, achou-se interessante colocá-las em modo comparativo para visualizar os alguns contrastes.

4.8 Testes com a VVM

Os testes com a VVM (Virtual Virtual Machine) foram feitos e comparados com testes realizados com o Jalapeño. O ambiente de testes distribuído é composto por dois microcomputadores: o primeiro consiste num Intel Pentium 4 de 3.0 Ghz de clock, 200 Gb de disco rígido e 1 Gb de memória dinâmica. O segundo consiste num Intel Pentium 233 Mhz, 40 Gb de disco rígido e 128 Mb de memória dinâmica. Ambas as máquinas possuem interfaces de rede de 100 Mbits/s e estão operando numa rede Fast Ethernet através de um hub Encore ENH908-NWY. Nessa rede também há tráfego de Internet através de um link Speedy Business de 256 kbits/s.

Para estabelecer as comparações iniciais dentre cada um dos computadores do ambiente de testes foi executado em cada um dos mesmos o *LINPACK Benchmark for Java*. Os dados de desempenho fornecidos pelo *LINPACK for Java* para os dois computadores encontram-se na tabela 3. Os testes com o *LINPACK for Java* foram executados usando uma JVM comum da Sun Microsystems na versão 1.5.

Tabela 3: *Comparação de desempenho entre os computadores Pentium 4 3.0 Ghz e Pentiu 233 Mhz do ambiente de testes distribuído* .

Máquina / Dados	MFlops/s	Tempo (segundos)
Pentium 4 3.0 Ghz	42.92	0.02
Pentium 233 Mhz	6.24	0.11

É notável, de acordo com os dados expressos na tabela 3, que o computador Pentium 4 de 3.0 Ghz é cerca de 7 vezes mais rápido em *MFlops/s* do que a outra estação de rede com um Pentium 233 Mhz, o *tempo de execução* para o benchmark também aumenta consideravelmente, porém, os valores de *Norm Res* e *precisão* são pouco diferentes.

Após os testes iniciais com o *LINPACK for Java Benchmark*, os esforços foram então concentrados em testar a VVM, comparando-a com benchmarks já realizados para a

Jalapeño JVM. A seguir, na tabela 4, estão dispostos os *tempos de execução* em **segundos** de vários aplicativos de benchmarking do pacote Symantec Microbenchmark. Infelizmente, como tal pacote é proprietário, e, não foi possível a aquisição dos benchmarkings, foi realizada uma aproximação de alguns dos benchmarkings com algoritmos em Java alternativos, que foram carregados na VVM e executados no ambiente distribuído composto pelos dois microcomputadores já citados no início deste tópico.

Tabela 4: *Tempos de execução da Jalapeño JVM usando o Symantec MicroBenchmarking:* (ALPERN et al, 2000).

Sist. Compilador / Executor	**Algoritmos**	
	Bubble Sort	**Fibonacci**
IBM DK Interpreter	**77.10**	**20.20**
Jalapeño baseline compiler	**36.70**	**11.70**
IBM DK JIT compiler	**3.20**	**1.10**
Jalapeño optimizing compiler	**4.00**	**1.00**

Usou-se para fins de teste da VVM dois algoritmos similares aos que constam do conjunto do *Symantec MicroBenchmarking* (Symantec, 2006): um primeiro algoritmo, chamado **ConcatString1** que realiza concatenações sucessivas numa variável String Java por *3.000 vezes* e depois oferece o tempo em milissegundos da execução. Um segundo benchmarking chamado **ConcaString2** também realiza concatenações sucessivas numa variável String Java agora por *30.000 vezes* e depois oferece o tempo em milissegundos da execução. Esse dois algoritmos são similares ao *Bubble Sort* da Symantec.

Os tempos individuais dos microcomputadores do ambiente distribuído de testes usando os softwares benchmarking ConcatString1.java e ConcatString2.java (usados para comparar o desempenho da VVM com o Jalapeño através do benchmark *Bubble Sort*), são demonstrados na tabela 5.

Tabela 5: *Tempos de execução (em segundos) diversos dos Benchmarkings ConcatString1 e ConcatString2 executando sobre a VVM operando em RMI.*

Máquina	ConcatString1 VVM	ConcatString2 VVM
Pentium IV 3.0 Ghz	**12,42**	**1.527,00**
Pentium 233 Mhz	**1.873,00**	**5.470,30**

Observando os dados da tabela 5, pode-se verificar, primeiramente a grande disparidade dos tempos de execução da máquina Pentium 233 Mhz, revelando nestes testes a importância dos processadores de alto desempenho para o funcionamento de um sistema distribuído.

Para incrementar os testes com a VVM, e, salientar ainda mais a comparação com os dados já dispostos na tabela 4 sobre os testes de benchmarking com a Jalapeño JVM, usou-se um algoritmo de *Fibonacci* que repete-se *3.000.000 vezes* e executa um looping de cerca de 40 iterações para cada chamada ao método, totalizando cerca de 120.000.000 iterações ao final, a tabela 6 possui os resultados dos tempos de execução, em segundos, deste segundo teste de benchmarking envolvendo a VVM.

Tabela 6: *Tempos de execução (em segundos) diversos do Benchmarking Fibonacci, com a VVM operando em RMI.*

Máquina	Fibonacci VVM
Pentium 4 3.0 Ghz	**1.285,80**
Pentium 233 Mhz	**9.480,50**

É importante observar como se comportam as máquinas quando do desempenho no ambiente distribuído: na execução da VVM instalada no Pentium IV (ela utiliza-se da máquina Pentium 233 Mhz para a execução dos bytecodes e instância da VM) acaba-se perdendo muito desempenho. Na execução da VVM instalada no Pentium 233 Mhz (que se utiliza da da máquina Pentium IV 3.0 Ghz) a execução acaba atingindo um desempenho mais alto.

4.9 Conclusões sobre a implementação da VVM

As medidas tomadas como parâmetro para comparação entra a VVM e a Jalapeño JVM são as aferições realizadas na última linha da tabela 4, sendo comparados os resultados com os alcançados pela VVM nas tabelas 5 e 6.

Por exemplo, em relação ao tempo de execução do benchmark *Bubble Sort* com Jalapeño JVM o tempo foi de 4.00 segundos aproxidamente, já a execução equivalente usando-se do aplicativo *ConcatString1* para a VVM foi de 12.42 segundos e com o *ConcatString2* foi de 1.527.00 segundos tomando como base o CPU Pentium IV 3.0 Ghz, o melhor dos equipamentos que havia disponível no ambiente de testes distribuído proposto.

Em relação ao Fibonacci Benchmark, a Jalapeño JVM possui a excelente marca de tempo de execução de 1.0 segundo , contra 1.285,80 segundos de tempo de execução da VVM tomando como base novamente o máquina equipada com o processador Pentium IV 3.0 Ghz. Apesar da grande disparidade nessa aferição de benchmarking, é válido lembrar que todos os algoritmos de aplicação (ConcatString1, ConcatString2 e Fibonacci) testados para a VVM eram aproximações do *Symantec Microbenchmark*, contudo realizam os mesmos tipos de operação nos algoritmos. Contudo, e, ainda assim, foram de grande valia para a avaliação final, pois, mostraram que a VVM realiza o seu propósito apesar do desempenho inferior à Jalapeño JVM.

5 - CONCLUSÕES

Baseando-se nas informações técnicas obtidas dos softwares analisados pode-se dizer que, a maior dificuldade de todos os pesquisadores envolvidos nos trabalhos sobre máquinas virtuais distribuídas foi a de atingir alta performance mantendo a boa legibilidade dos algoritmos, logo, concentrar-se na legibilidade dos algoritmos de usuário, isto é, na facilidade com que podem ser escritos e compreendidos por qualquer programador, foi uma

preocupação constante. Um algoritmo paralelo deveria ser de tão boa leitura quanto um algoritmo procedural, deixando clara a semântica do código a qualquer programador que leia o código.

As ditas *máquinas virtuais paralelas* são muitas vezes, na verdade, classes de objeto com programação paralela usando os mecanismos nativos de Java(como o RMI ou Jini) para criar as primitivas de paralelismo na aplicação, por vezes também usaram-se da estratégia de usar um compilador Java estendido que compila as palavras reservadas especiais criadas pelos pesquisadores (como `remote` e outras) e as transformava em códigos que se usavam de chamadas RMI para supostamente depois gerar os bytecodes cem por cento compatíveis com todas as VMs Java. Alguns projetos mais audaciosos chegaram a converter os códigos Java para a linguagem C e gerar binários executáveis com as bibliotecas de paralelização em C aproveitando vantagem da velocidade de execução dos binários em C.

Concluiu-se a partir das análises anteriores que a passagem de mensagem é um método de comunicação interprocessos que fornece altas performances quando usada especialmente com aplicações paralelas. Uma demonstração da aposta que os atuais desenvolvedores fazem em tal tecnologia está, por exemplo, no fato de que as versões mais recentes dos sistemas operacionais modernos, vêm habilitados com suporte nativo a MPI.

O protótipo de testes da VVM, de acordo com o modelo de implementação que foi apresentado nessa dissertação, poderá, em versões futuras estar trabalhando, tanto com a tecnologia de RMI (já implementada nesta primeira versão do protótipo) bem como com a tecnologia de MPI, poderá ser uma máquina virtual Java distribuída de potencial maior, pois ainda faltam melhorias no software para atingir marcas melhores nos benchmarkings. Além disso, a VVM vem com a nova proposta de distribuir instâncias da própria Virtual Machine pelo ambiente distribuído, fato inovador do ponto de vista do paradigma atual de funcionamento das VMs.

Os resultados obtidos usando os aplicativos ConcatString1, ConcatString2 e Fibonacci, revelam que a VVM ainda precisa desempenhar melhor performance quando do uso do ambiente distribuído, além disso, ficaram claras as disparidades causadas nas aferições devido às diferenças de capacidade entre as máquinas que compunham o ambiente de testes distribuído.

A Jalapeño JVM ainda é superior e seu tempo foi recorde quando da comparação da execução da VVM com os algoritmos *ConcatString* (comparado com o benchmark Bubble Sort) e em comparação da execução da VVM com o algoritmo de Fibonacci. Porém, os tempos de execução (mensurados em segundos) nas comparações de benchmarking, no tópico de testes da VVM revelam que ela pode se tornar uma candidata a fazer parte das máquinas virtuais Java distribuídas, principalmente pelas características, conceitos e novos recursos que sua arquitetura e modelo de implementação trazem consigo.

6 – TRABALHOS FUTUROS

Há muito o que realizar ainda neste projeto da Virtual Virtual Machine, alguns dos objetivos iniciais foram cumpridos parcialmente devido a dificuldades com a programação da comunicação da VVM via RMI, e, também há complementos que serão necessários.

Desenvolver na VVM o *módulo de dispatch em MPI* é um trabalho futuro que será necessário para este trabalho, pois, se tornará um mecanismo de comuincação distribuída em adendo ao RMI atualmente implementado no protótipo. Anexar tal funcionalidade à VVM

significa aumentar a versatilidade desta máquina virtual, e ainda, oferecer mais uma opção de tecnologia de comunicação em ambiente distribuído.

A Virtual Virtual Machine, pode vir ser ainda mais útil se puder interoperar com o sistema operacional e também conseguir *transmitir POSIX threads* através do ambiente de rede para os demais hosts. Vale lembrar que os POSIX threads geralmente são suportados pela grande maioria dos SOs através de bibliotecas, APIs ou ainda implementações dos mesmos ainda em primitivas de kernel, logo, desenvolver para a VVM um módulo de gerenciamento e envio das tarefas como POSIX threads é um trabalho futuro. Além disso, há também o fato de que ainda se devem levantar as primitivas de acoplamento da VVM a um kernel de sistema operacional (tratam-se de chamadas de sistema de kernel para a VVM), para que ela possa funcionar de forma mais diretamente ligada com a infra-estrutura de hardware sob o SO, e, para que suas requisições ao SO sejam atendidas com uma maior prioridade e desempenho.

A implementação da VVM também necessita de uma interface para transformação de *threads comuns de Java* para ***POSIX-Threads***, contudo, no primeiro release, tal recurso não está presente, pois, exige uma programação à parte de interação da VVM com o sistema operacional, que seriam as primitivas ou chamadas de sistemas que realizariam o efetivo acoplamento da VVM ao kernel do SO.

O mecanismo de *dispatch em RMI* da VVM pode ser melhorado, ele funciona no modelo Cliente/Servidor, com otimizações no código-fonte de modo que a programação da versão Cliente e Servidor com RMI utilizem-se de sockets em adendo à técnica de serialização de objetos (nem todo objeto Java pode ser serializado), pode-se melhorar as estatísticas de desempenho demonstradas no capítulo de implementação. O release atual da VVM ainda é instável (por vezes, não carrega os bytecodes de forma adequada, lançando erros de exceção).

Para aumentar o grau de portabilidade da VVM em si, e, visando relacionar-se com os diversos sistemas operacionais modernos, implementar-se uma interface em JNI (Java Native Interface) seria também um de trabalho futuro, pois, através da conversão das funcionalidades de base da VVM para código C na linguagem nativa de uma dada arquitetura de máquina, aumenta-se também o poder de desempenho da Virtual Machine, indo de encontro aos objetivos dos projetos mais importantes de máquinas virtuais paralelas, como, por exemplo, a Jalapeño Java Virtual Machine.

Como todo projeto de software, é interessante que se desenvolvam também recursos adicionais para a VVM, como templates de programas de demonstração, documentação em JavaDOC e outros recursos adicionais para que novos usuários consigam usufruir da mesma, e, também, colaborar através da proposição de novos recursos.

7 - REFERÊNCIAS

ALPERN B.; ATTANASIO, C. R.; BARTON, J. J.; BURKE, M. G.; CHENG, P.; CHOI, J. D.; COCCHI, A.; FINK, S. J.; GROVE, D.; HIND, M.; HUMMEL, S. F.; LIEBER, D.; LITVINOV, V.; MERGEN, M. F.; NGO, T.; RUSSELL, J. R.; SARKAR, V.; Serrano , M. J.; SHEPHERD, J. C.; SMITH, S. E.; SREEDHAR, V. C., SRINIVASAN, H.; WHALEY J. **The Jalapeño Virtual Machine.** *IBM System Journal*, Vol 39, No. 1, págs. 211-237. Fevereiro de 2000. Disponível em: <http://www.research.ibm.com/journal/sj/391/alpern.pdf >. Acesso em: 26 de outubro de 2006.

BAL, Henri E.; KIELMANN, Thilo. **Manta Fast Parallel Java.** *Technical Report IR-450*, Vrije Universiteit. Amsterdã, 1998. Disponível em: <http://www.cs.vu.nl/~robn/manta/>. Acesso em: 26 de outubro de 2006.

BAPCO Benchmarks. **SYSmark 2004 - Benchmarking.** Disponível em: <http://www.bapco.com/products/sysmark2004/ >. Acesso em: 27 de outubro de 2006.

BULL, M.; SMITH, L. **Development of mixed mode MPI / OpenMP applications**. *Scientific Programming*, vol. 9, no. 2-3, pp. 83-98, 2001. Disponível em: <http://www.epcc.ed.ac.uk/~markb/pubs.html>. Acesso em: 30 de outubro de 2006.

CARPENTER, Bryan; CHANG, Yuh-Jye; FOX, Geoffrey; LESKIW, Donald; LI, Xiaoming. **Experiments with HPJava. Concurrency: Practice and Experience**, 9(6):633, 1997. Disponível em: <http://www.npac.syr.edu/projects/pcrc/HPJava/>. Acesso em: 30 de outubro de 2006.

CARPENTER, Bryan. **Development of Data-Parallel Programming.** *NPAC at Syracuse University***,** Syracuse, NY13244, 2002. Disponível em: <http://www.hpjava.org/talks/beijing/hpf/introduction/introduction.html> . Acesso em 30 de outubro de 2006.

COBBS, Archie L. **JC Virtual Machine Official Documentation.** Disponível em: <http://jcvm.sourceforge.net/share/jc/doc/jc.html >. Acesso em: 26 de outubro de 2006.

COSTANZA, Pascal. **Gilgul JVM extended.** Disponível em: <http://javalab.cs.uni-bonn.de/research/gilgul/ >. Acesso em: 26 de outubro de 2006.

CRAMER, T. et al. **Compiling Java Just in Time**. *IEEE Micro*, pp. 36-43, Vol. 17, No. 2, Maio/Junho de 1997. Disponível em: <http://citeseer.ist.psu.edu/cramer97compiling.html>. Acesso em 28 de Janeiro de 2006.

EURO-PAR CLUSTER COMPUTING WORKSHOP 2000. Disponível em <http://www.buyya.com/EuroParCluster2000/>. Acesso em 26 de outubro de 2006.

FERRARI, A. **JPVM: The Java Parallel Virtual Machine**. Disponível em: <http://www.cs.virginia.edu/~ajf2j/jpvm/>. Acesso em: 10 de fevereiro de 2006.

HILFINGER, P. N.; BONACHEA, Dan; GAY, David; GRAHAM, Susan; LIBLIT, Bem; PIKE, Geoff; YELICK, Katherine. **Titanium Language Reference Manual Version 1.16.8.** Disponível em: <http://titanium.cs.berkeley.edu/doc/lang-ref.pdf>. Acesso em 30 de outubro de 2006.

JAVAGRANDE FORUM 2000. Disponível em: <http://www.javagrande.org/>. Acesso em 25 de outubro de 2006.

KAZI, Iffat H.; CHEN, Howard H.; STANLEY, Berdenia, LILJA, David L. **Techniques for Obtaining High Performance in Java Programs.** *High-Performance Parallel Computing Research Group Technical Report No. HPPC-99-01*, 1999. Disponível em: <http://citeseer.ist.psu.edu/article/kazi99techniques.html>. Acesso em: 30 de outubro de 2006.

LEE, B. S.; GU, Yan; CAI, Wentong; HENG, Alfred. **Performance Evaluation of JPVM**. *Parallel Processing Letters*, further-coming. Disponível em: <http://citeseer.ist.psu.edu/500144.html>. Acesso em: 30 de outubro de 2006.

LEPREAU, Jay; HSIEH, Wilson; CARTER, John; FLATT, Matthew; ZACHARY, Joe. **Janos Virtual Machine.** Disponível em: <http://www.cs.utah.edu/flux/janos/ >. Acesso em 26 de outubro de 2006.

LINDHOLM, Tim; YELLIN, Frank. **The Java Virtual Machine Specification**. 2 ed *Addison-Wesley*, 1996. Disponível em: <http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html >. Acesso em 30 de outubro de 2006.

MAASSEN, J.; NIEUWPOORT , R. Van. **Fast Parallel Java**. *Master's thesis Vrije Universiteit,* Amsterdam, 1998. Disponível em: <http://www.cs.vu.nl/albatross/> ou < http://citeseer.ist.psu.edu/maassen98fast.html>. Acesso em: 30 de outubro de 2006.

MICROSYSTEMS, Sun. **Java 2 Platform Standard API Specifications**. Disponível em: <http://java.sun.com/reference/api/index.html> . Acesso em 30 de março de 2005.

MICROSYSTEMS, Sun. **Java 2 Platform Standard Edition Overview**. Disponível em: <http://java.sun.com/j2se/overview.html> . Acesso em 30 de março de 2005.

MICROSYSTEMS, Sun. **JavaSpaces Service Specification version 1.2.1.** Disponível em: < http://www.jini.org/nonav/standards/porter/doc/specs/html/js-spec.html>. Acesso em 28 de Janeiro de 2006.

MICROSYSTEMS, Sun. **Jini Network Technology.** Disponível em: <http://www.sun.com/software/jini/>. Acesso em: 09 de março de 2005.

PENDRAGON Software Corporation. **CaffeineMark and Embedded CaffeineMark Benchmarks - version 3.0.** Disponível em: <http://www.benchmarkhq.ru/cm30/info.html>. Acesso em: 27 de outubro de 2006.

PHILIPPSEN, Michael and ZENGER, Matthias. **JavaParty - Transparent Remote Objects in Java**. *Journal: Concurrency - Practice and Experience*, 1997. Número 9, Volume 11, Páginas 1125-1242. Disponível em: <http://citeseer.ist.psu.edu/philippsen97javaparty.html>. Acesso em: 28 de Janeiro de 2006.

PPoPP' 2001 - SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING. Disponível em < http://www.ppopp.org/>. Acessado em 26 de outubro de 2006.

RADHAKRISHNAN, R; VIJAYKRISHNAN, N; JOHN, L. K.; SIVASUBRAMANIAM, A. **Architetural Issues in Java Runtime Systems**. *Technical Report*, *TR-990719*. University of Texas, Austin, 1999. Disponível em: <http://citeseer.ist.psu.edu/radhakrishnan99architectural.html> ou <http://www.ece.utexas.edu/projects/ece/lca/ps/tr990719.pdf>. Acesso em: 30 de outubro de 2006.

SMITH, James E; NAIR, Ravi. **VIRTUAL MACHINES: Versatile Platforms for Systems and Processes.** 1 ed. San Francisco: Ed. Morgan Kaufmann / Elsevier, 2005.

SNIR, Marc; OTTO, Steve; HUSS-LEDERMAN, Steven; WALKER, David; DONGARRA Jack. **MPI - The Complete Reference - Volume 1, The MPI Core Second Edition**. Massachusetts Institute of Technology. Second printing, 1999. Disponível em: <http://netlib2.cs.utk.edu/utk/papers/mpi-book/mpi-book.html>. Acesso em: 30 de outubro de 2006.

SYMANTEC Corporation. **Symantec Microbenchmark for Java.** Disponível em: < http://www.symantec.com/domain/cafe/jit.html >. Acesso em: 12 de setembro de 2006.

TANENBAUM, Andrew S. **Sistemas Operacionais – Projeto e Implementação.** 2 ed. Porto Alegre: Editora Bookman, 2000.

THIRUVATHUKAL, George K; DICKENS, Phil M; BHATTI, Shahzad**. Java on Networks of Workstations (JavaNOW): A Parallel Computing Framework Inspired by Linda and the Message Passing Interface (MPI).** Concurrency: Practice and Experience, Volume 12, Edição 11, Páginas 1093-1116, 2000. Disponível em: <http://citeseer.ist.psu.edu/308190.html>. Acesso em: 30 de outubro de 2006.

THURMAN, Dave. **JavaPVM.** Disponível em: <http://www.chmsr.gatech.edu/people/dave/>. Acesso em: 15 de março de 2005.

TULLMANN, Patrick. **Alta Virtual Machine: The Nested Process Model In Java.** Disponível em: <http://www.cs.utah.edu/flux/java/alta/index.html >. Acesso em 26 de outubro de 2006.

VOLANO Software. **Volano Chat Benchmark 2.6.4.** Disponível em: <http://www.volano.com/>. Acesso em: 27 de outubro de 2006.

YANG, Byung-Sun; PARK, Jinpyo; LEE, Junpyo; PARK, Seongbae; LEE, SeungIl; KIM , Suhyun; CHUNG, Yoo C. **Latte Virtual Machine.** Disponível em: <http://latte.snu.ac.kr/>. Acesso em: 10 de outubro de 2006.

WELLING, Girish Sharad. **Designing Adaptive Environment-Aware Applications for Mobile Computing.** *Doctor's thesis, New Brunswick, New Jersey. October, 1999* Disponível em: < http://www.sigmobile.org/phd/1999/theses/welling.pdf>. Acesso em 30 de outubro de 2006.

WILKINSON, Tim. **Kaffe Java Virtual Machine.** Disponível em: <http://www.kaffe.org>. Acesso em 25 de outubro de 2006.

8 – APÊNDICE A – Código-Fonte

CLVVM.java

```
import java.lang.Object;
import java.net.URL;
import java.net.URLClassLoader;
import java.net.MalformedURLException;
import java.io.*;

public class CLVVM implements java.io.Serializable {
         String host;
         int port;
       private URLClassLoader urlClassLoader = null;

       public CLVVM() {
```

```
          try {
            URL[] url = new URL[1];
            url[0] = new URL("file", null, System.getProperty("user.dir") +
File.separator);
           } catch (MalformedURLException e) {
                System.out.println(e);
           }
       }

       public URLClassLoader getURLClassLoader() {
             return urlClassLoader;
       }

       public Class loadClassCLVVM(String className) {
          Class loadedClass = null;

           try {
                 loadedClass = urlClassLoader.loadClass(className);
          } catch(ClassNotFoundException e) {
                 System.out.println(e);
            }

          return loadedClass;
       }
}
```

RMIClient.java

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.NotBoundException;
import java.net.MalformedURLException;

 public class RMIClient {

    public static void main( String args[] ) {
         try {
             InterfaceRMI interfaceRMI = (InterfaceRMI)
Naming.lookup("rmi://192.168.0.3/CLVVMService");
           CLVVM loaderCLVVM = (CLVVM) interfaceRMI.getCLVVM();
             System.out.println("object loaderCLVVM: " + loaderCLVVM);
            interfaceRMI.sendObject("HelloWorld");
         }
         catch( MalformedURLException e ) {
             System.out.println("MalformedURLException: " + e.toString() );
         }
         catch( RemoteException e ) {
             System.out.println("RemoteException: " + e.toString() );
         }
         catch( NotBoundException e ) {
             System.out.println("NotBoundException: " + e.toString() );
         }
         catch( Exception e ) {
             System.out.println("Exception: " + e.toString() );
         }
     }
 }
```

CLVVMImpl.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

 public class CLVVMImpl extends UnicastRemoteObject implements InterfaceRMI
{

     private CLVVM loaderCLVVM;

     public CLVVMImpl() throws RemoteException {
         super();

       loaderCLVVM = new CLVVM();
     }

     public void sendObject(String className) throws RemoteException {
       Class classToLoad = loaderCLVVM.loadClassCLVVM(className);


       } catch (InstantiationException e) {
            System.out.println(e);
       } catch (IllegalAccessException e) {
            System.out.println(e);
       }
    }

    public Object getCLVVM() throws RemoteException {
      return loaderCLVVM;
    }
 }
```

InterfaceRMI.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface InterfaceRMI extends Remote {

    public void sendObject(String className) throws RemoteException;
    public Object getCLVVM() throws RemoteException;
}
```

RMIServer.java

```
import java.rmi.Naming;
```

```
public class RMIServer {

     public RMIServer() {
         try {
             InterfaceRMI interfaceRMI = new CLVVMImpl();
             Naming.rebind("rmi://192.168.0.3:1099/CLVVMService",
interfaceRMI);
         }
         catch(Exception e) {
             System.out.println(e);
         }
     }

     public static void main(String[] args) {
         new RMIServer();
     }
 }
```

HelloWorld.java

```
public class HelloWorld implements java.io.Serializable {

      public HelloWorld() {
            System.out.println("\n\n\n \t\t Hello World of Java
Programming!!!!! \n\n\n");
      }
}
```