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

RODRIGO FOGAÇA MIGUEL

Avaliação de Desempenho de Aplicações utilizando JPVM

MARÍLIA
2006

RODRIGO FOGAÇA MIGUEL

Avaliação de Desempenho de Aplicações utilizando JPVM.

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

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

MARÍLIA
2006

RODRIGO FOGAÇA MIGUEL

Avaliação de Desempenho de Aplicações utilizando JPVM

Banca examinadora do Trabalho de Conclusão de Curso apresentado a UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação. Área de concentração: Arquitetura de computadores.

Resultado: _____

_____
ORIENTADORA: Profa. Dra. Kalinka Regina Lucas Jaquie Castelo Branco.

_____
1º EXAMINADOR: Fátima L. S. Nunes Marques.

_____
2º EXAMINADOR: Antonio Carlos Sementille

Marília, 01 de dezembro de 2006.

Dedico primeiramente ao meu pai José Carlos e minha mãe Marly que sempre estiveram ao meu lado em todos os momentos, não só como pais, mas como amigos.

Ao meu irmão que mesmo longe sempre foi um exemplo de capacidade e dedicação.

E aos meus amigos que sempre me apoiaram e me deram força.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me abençoado nestes quatro anos de faculdade, por estar sempre ao meu lado e me dar força em todos os momentos.

A toda a minha família, aos meus pais e ao meu irmão, que sempre estiveram comigo, confiando na minha capacidade, e não mediram esforços para que eu concluísse a faculdade.

Agradeço a todos os meus amigos de Assis, aos meus amigos de comunidade, ao pessoal da minha banda, “ponto de partida”, a todos os meus amigos que me ajudaram e me apoiaram.

Agradeço aos meus amigos de classe, que me apoiaram sempre me ajudando nos trabalhos e nos estudos. Em especial: Adriano, Mariana, Danilo, Ana, Bárbara. Amo muito vocês.

À minha orientadora Kalinka Regina Lucas Jaquie Castelo Branco, pela paciência nesses meses de trabalho.

Enfim a todos os meus familiares e amigos que estiveram comigo sempre.

MIGUEL, Rodrigo Fogaça. Avaliação de Desempenho de Aplicações utilizando JPVM. 2006. NumFolhas 74 f. Monografia (Bacharelado em Ciências da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2006.

RESUMO

Atualmente a busca pelo alto desempenho vem ocasionando várias pesquisas e descobertas, uma delas foi a utilização de sistemas distribuídos para a realização de processamentos paralelos. Máquinas são interligadas através de uma rede possuindo mecanismos para a troca de mensagens, como o JPVM. O pacote JPVM é uma API Java que implementa funcionalidades do PVM e cria uma camada entre o usuário da aplicação distribuída e as bibliotecas tradicionais do PVM. Além disso apresenta algumas características herdadas da própria linguagem Java, como uso de *threads*. O objetivo desse trabalho é realizar aplicações que utilizem a biblioteca JPVM e analisar os resultados obtidos, verificando situações diferentes, como número de máquinas e número de tarefas, efetuando uma avaliação de desempenho entre aplicações seqüenciais e paralelas.

Palavras-chave: Processamento paralelo, JPVM, Sistemas Computacionais Distribuídos.

MIGUEL, Rodrigo Fogaça. Avaliação de Desempenho de Aplicações utilizando JPVM. 2006. NumFolhas 74 f. Monografia (Bacharelado em Ciências da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2006.

ABSTRACT

Currently the search for the high performance comes causing some research and discoveries, one of them was the use of distributed systems for the accomplishment of parallel processings. Machines are linked through net possessing mechanisms for the exchange of messages, as the JPVM. Package JPVM is an API Java that implements functionalities of the PVM and creates a layer enters the user of the distributed application and the traditional libraries of the PVM. Moreover it presents some inherited characteristics of the proper Java language, as use of threads. The objective of this work is to carry through applications that use library JPVM and to analyze the gotten results, verifying different situations as number of machines and number of tasks, effecting a performance evaluation of parallel and sequential applications.

Keywords: Parallel processings, JPVM, Distributed Systems.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

E/S: Entrada e Saída

JPVM: *Java Parallel Virtual Machine*

MISD: *Multiple Instruciton Single Data*

MIMD: *Multiple Instruction Multiple Data*

MPI: *Message Passing Interface*

MPMD: *Multiple Program Multiple Data*

ORNL: *Oak Ridge National Laboratory*

PVM: *Parallel Virtual Machine*

RPC: *Remote Procedure Call*

SPMD: *Single Program Multiple Data*

SISD: *Single Instruction Single Data*

SIMD: *Single Instruction Multiple Data*

TCP: *Transport Control Protocol*

UCP: Unidade Central de Processamento

UDP: *User Datagrama Protocol*

SÚMARIO

INTRODUÇÃO

Nas últimas décadas os computadores têm tido uma evolução muito grande, tanto na área de *hardware* quanto na área de *software*. Esta evolução intensa vem acontecendo depois da criação do "modelo de Von Neumann", que se caracteriza por um conjunto de processador, memória e dispositivos de E/S.

Esta evolução também vem da alta demanda de processamento, visando uma busca crescente por maior desempenho. Devido a esta procura por um melhor desempenho foram surgindo barreiras, uma delas a tentativa de se adequar o *software* ao *hardware*, já que o último tem tido um crescimento maior.

O modelo de Von Neumann contudo, não atendia às necessidades de alto processamento, assim, na década de 80, projetos paralelos foram criados buscando atender um nível superior de poder de processamento.

Com a evolução das redes, surgiu uma certa facilidade de permitir a comunicação de computadores a baixo e médio custo. Com o problema do custo e a evolução das redes surgiram os sistemas distribuídos, que no inicio teve como principal visão o compartilhamento de recursos.

Sistemas Distribuídos consistem em uma coleção de computadores autônomos ligados por uma rede de comunicação. As vantagens destes sistemas são a aplicação de novos computadores e linhas de comunicação, compartilhamento de dados, o que permite vários usuários acessarem uma base de dados comum, compartilhamento de recursos e a execução de programas paralelos. Mesmo surgindo de áreas divergentes, a computação paralela e os sistemas distribuídos têm sido muito utilizados de forma conjunta, visto que possuem

problemas em comum, como balanceamento de carga, tolerância a falhas, entre outras (SANTANA et al,1997).

Com o crescimento dessas áreas foi surgindo a necessidade de aplicações que auxiliavam na troca de informações entres os elementos do Sistema distribuído. Como exemplo tem-se *Parallel Virtual Machine* (PVM), *Message Passing Interface* (MPI), *Java Parallel Virtual Machine* (JPVM), entre outras bibliotecas que foram desenvolvidas.

A utilização dessas bibliotecas para geração de aplicações paralelas mostrou uma grande economia em relação à utilização de aplicações paralelas em *mainframes*, tornando-se bastante interessante seu uso e conseqüentemente mais pesquisas são feitas objetivando um desenvolvimento em potencial.

Este projeto visa implementar duas aplicações, multiplicação de matrizes e cálculo do trapézio composto utilizando o JPVM como base para analisar o desempenho de aplicações executadas em paralelo e seqüencialmente.

1. Computação Paralela Distribuída

Este capítulo apresenta conceitos básicos de computação paralela, contendo também algumas das classificações de arquiteturas paralelas e o uso da computação paralela em sistemas distribuídos.

1.1. Considerações iniciais

A computação paralela nasceu e se desenvolveu graças a grande busca pelo aumento de poder computacional (ALMASI,1994)

Um dos aspectos mais importantes que deve ser tratado quando se diz a respeito de computação paralela, é o ganho no desempenho. Atualmente devido ao grande número de dados e algoritmos complexos, as arquiteturas de Von Neuman, ainda apresentam algumas deficiências Já com o surgimento da computação paralela pode-se ter um grande avanço tanto em desempenho quanto em eficiência, conseguindo um aumento no poder computacional. Por este e por outros motivos a computação paralela vem crescendo.

1.2. Concorrência e Paralelismo

Concorrência é quando se tem a execução de vários processos, que tem por objetivo chegar a um resultado de forma mais eficiente. Pode ocorrer em sistemas com uma única unidade de processamento ou em sistemas multi-processados (ALMASI,1994)

Existem duas formas de paralelismo, o paralelismo físico e o lógico (pseudo-paralelismo). O paralelismo lógico é quando se tem várias intercalações de processos em uma única unidade de processamento. Cada processo é executado um de cada vez em um tempo determinado. Por isso o nome pseudo-paralelismo uma vez que é criada a ilusão de um paralelismo, quando na verdade os processos estão compartilhando processador.

Na Figura 1 pode-se observar este exemplo de concorrência.

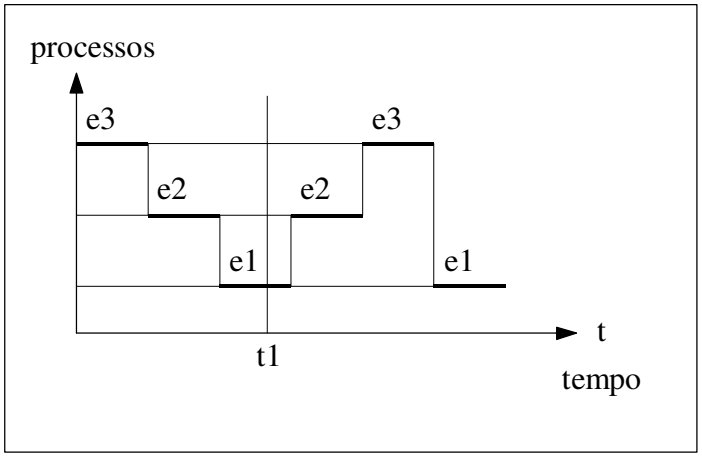


Figura 1 - Paralelismo Lógico (SANTANA et al, 1997)

Paralelismo físico é quando vários processos são executados no mesmo intervalo de tempo simultaneamente. Neste tipo de paralelismo necessita-se de mais de uma unidade de processamento. A Figura 2 ilustra o paralelismo físico.

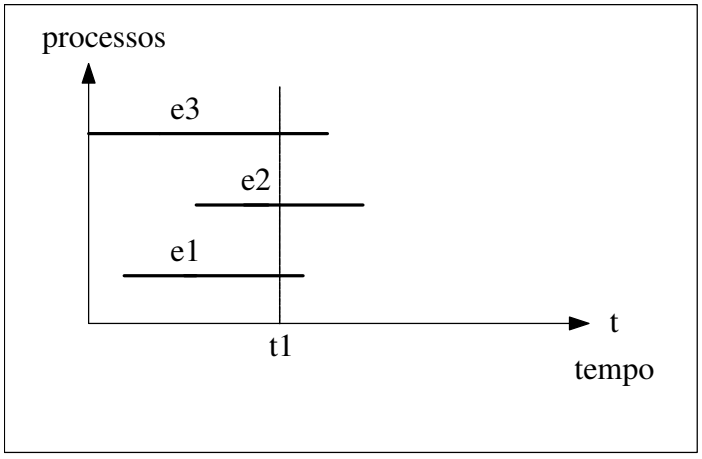


Figura 2- Paralelismo Físico (SANTANA et al,1997)

1.3. Granulação

Uma das definições que é utilizada para definir o tamanho das unidades de trabalho submetidas aos processadores é a granulação, e o número de processadores/processos e os tamanhos de cada tarefa são fatores que influenciam (NAVAUX,1998).

Granularidade está relacionada diretamente com os níveis de paralelismo. Quanto mais baixo o nível mais fino é a granularidade do processamento. A partir destes conceitos pode-se classificar a granulação em três tipos diferentes: fina, média e grossa. (ALMASI,1994) Granulação fina relaciona-se com o paralelismo de baixo nível, possui grande número de processos pequenos e simples, a granulação média já possui um número menor de processos e são mais complexos que na granulação fina, como por exemplo procedimentos, *loops*, funções, entre outros. Granulação grossa envolve uma paralelização de mais alto nível, já que são utilizados processos maiores e mais complexos.

1.4. *Speed Up*

Um dos objetivos do uso do paralelismo é o aumento do desempenho quando comparado ao seqüencial; desse modo, são utilizadas algumas técnicas para medir este desempenho. Uma delas é o *speed up*. (SANTANA et al,1997)

Speed up tem uma definição simples, é a medida da razão de um processamento seqüencial (onde existe o uso de apenas um processador) e um processamento paralelo (onde existe o uso de vários processadores).

Fórmula do *speed up*:

$$S(n) = T(1)/T(n)$$

onde:

- T(1) – Tempo da execução do aplicativo em um processamento seqüencial
- T(n) - Tempo da execução do aplicativo em um processamento paralelo(*n* processadores).

1.5. Arquiteturas Paralelas

Segundo Duncan (DUNCAN,1990) pode-se definir uma arquitetura paralela da seguinte forma:

> *"Uma arquitetura paralela fornece uma estrutura explícita e de alto nível para o desenvolvimento de soluções utilizando o processamento paralelo, através da existência de*

> *múltiplos processadores, estes simples ou complexos, que cooperam para resolver problemas através de execução concorrente".*

De certa forma as arquiteturas paralelas vieram para prover um ganho no poder computacional. Entretanto, as arquiteturas possuem alguns pontos negativos, podendo-se citar: comunicação entre os processadores, programação mais difícil. (SANTANA et al,1997)

Existem algumas classificações de arquiteturas paralelas, apenas duas de maior importância serão apresentadas: classificações segundo Flynn (FLYNN,1996) e Duncan (DUNCAN,90).

Definida em 1966 por Flynn (FLYNN,1996) é atualmente uma das classificações de arquiteturas mais utilizadas. A classificação de Flynn utiliza duas características essenciais: os fluxos de dados e os fluxos de instruções como ilustrado na Figura 3 :

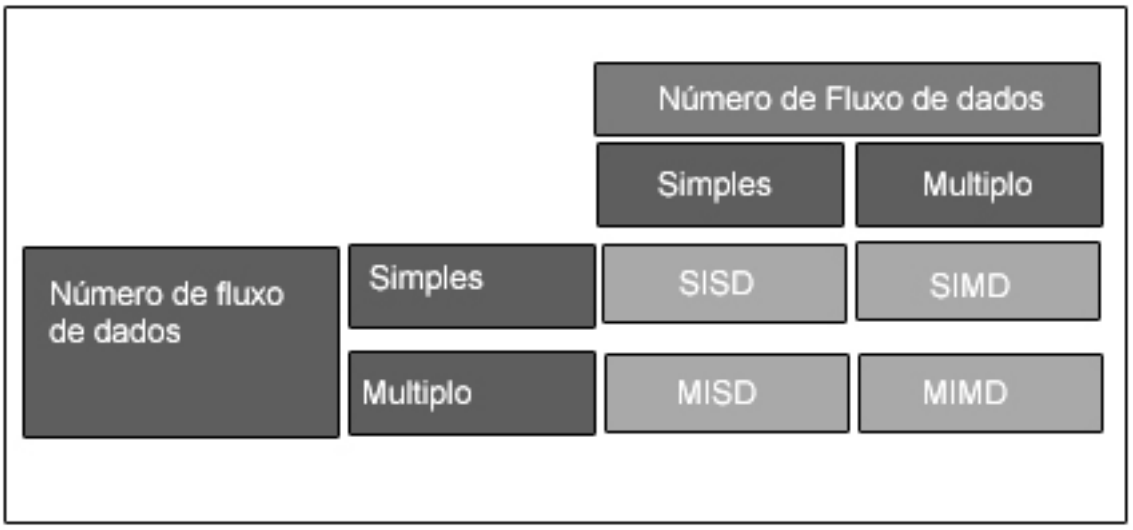


Figura 3 - Taxonomia de Flynn (FLYNN,1996)

- **SISD:** As máquinas SISD (*Single Instruction stream Single Data stream*) são as máquinas mais convencionais, com uma única unidade de processamento e existe um único fluxo de instruções atuando em um único fluxo de dados. Neste tipo de máquina as instruções são executadas uma após a outra (FLYNN,1996).

- **SIMD:** (*Single Instruction stream Multiple Data stream*) São máquinas que possuem mais de uma unidade de processamento de dados, mas contêm apenas uma unidade que supervisiona as instruções que serão feitas de forma seqüencial (*single instruction stream*). A unidade responsável por organizar as instruções é chamada de unidade de controle. Uma máquina que exemplifica esse tipo de arquitetura é a ILLIAC-IV.
- **MISD:** Nenhum modelo de sistema computacional se encaixa ao MISD (*Multiple Instruction stream Single Data stream*), este modelo de arquitetura seria a combinação de múltiplas instruções em um único fluxo de dados.
- **MIMD:** Máquinas desse modelo (*Multiple Instruction stream Multiple Data Stream*) executam várias operações usando diferentes segmentos de dados (DEPERNI,2002). É relevante mostrar que nas máquinas MIMD a memória faz a comunicação entre os processos dos diversos processadores, sem esta interação iríamos ter múltiplos SIMDs (FLYNN,1996). A maioria dos sistemas de arquitetura paralela se encaixa nessa categoria.

A classificação MIMD pode ser dividida em dois subgrupos, os multiprocessadores e multicomputadores, conforme apresentado na Figura 4 . A diferença básica entre os dois modelos é a interação com a memória, nos multiprocessadores, há apenas um endereço virtual que é compartilhado por todas as Unidades Centrais de Processamento já no multicomputador cada máquina possui sua própria memória (TANEMBAUM,1995).

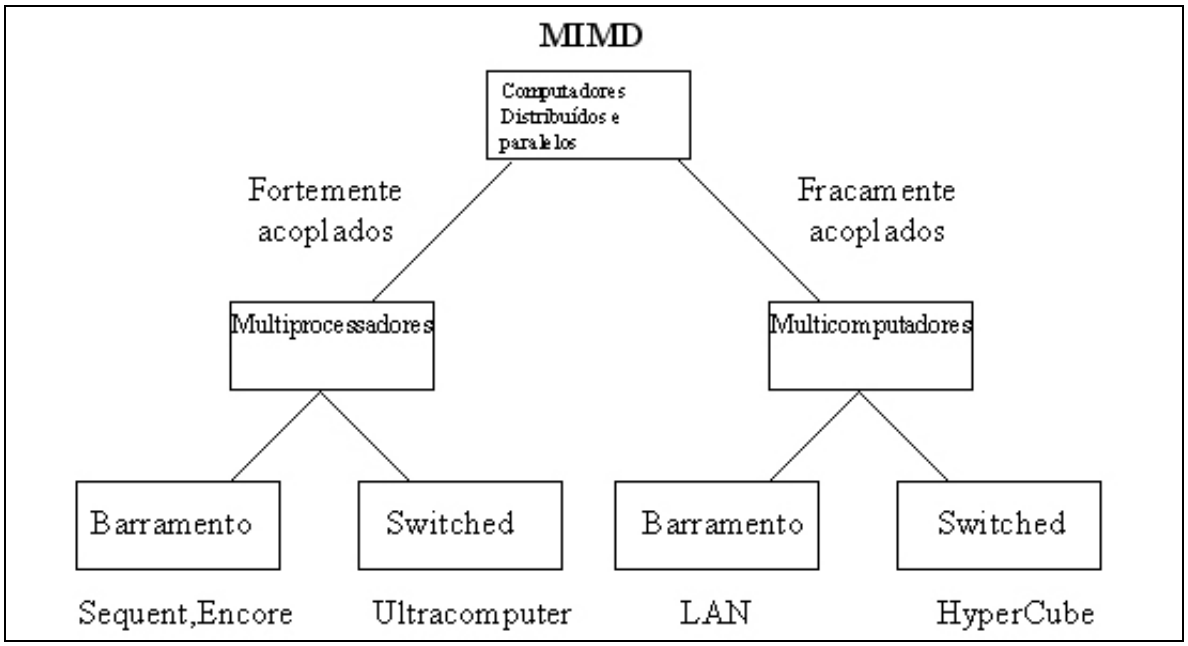


Figura 4 – Taxonomia de Sistemas Computacionais paralelos e Distribuídos (TANEMBAUM,1995)

A taxonomia de Flynn é a mais utilizada, entretanto algumas arquiteturas mais recentes ficam sem classificação. Com a tentativa de abranger estes tipos de arquiteturas, Duncan (DUNCAN, 1990) dividiu as arquiteturas em síncronas e assíncronas, como ilustrado na Figura 5, e descartou aquelas arquiteturas que possuem paralelismo baixo.

Existem dois motivos pelos quais Duncan descarta essas classificações. São elas: a falha de se adaptar ao um padrão mais rigoroso, e segundo, estas arquiteturas não apresentam uma estrutura explicita, coerente para desenvolver soluções pra problemas de paralelismo de alto nível (DUNCAN, 1990).

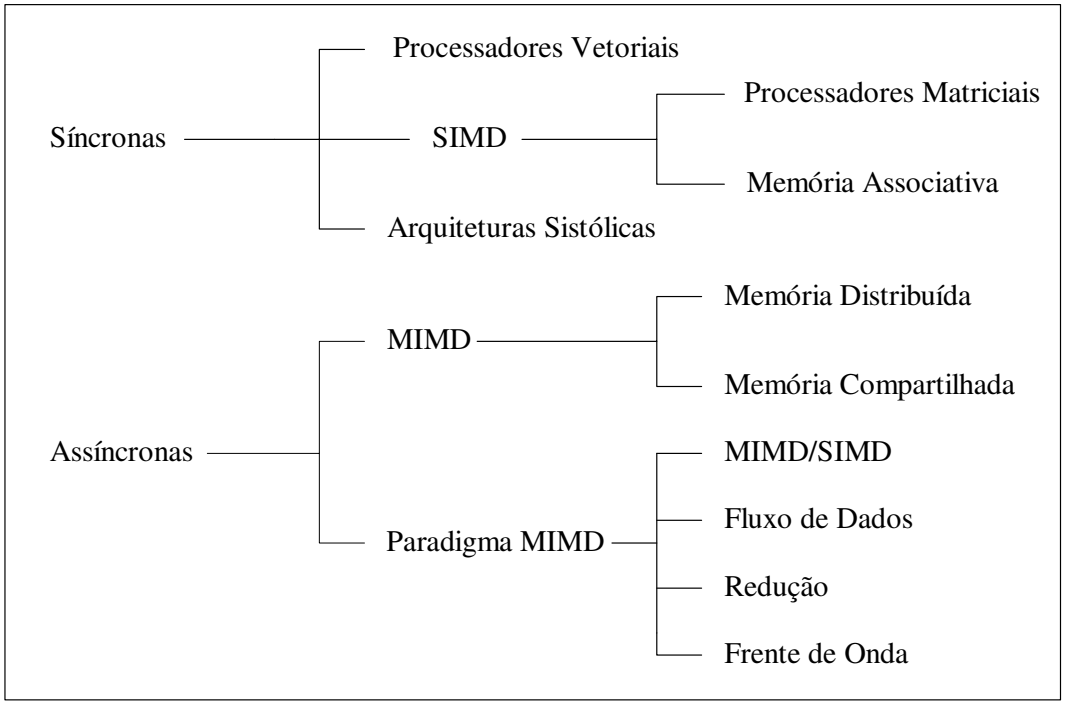


Figura 5 – Classificação Duncan (DUNCAN, 1990)

Nas arquiteturas síncronas o funcionamento dos processos é feito por *clocks* globais, unidades de controle central, ou unidades de controle vetoriais (DUNCAN, 1990). As arquiteturas síncronas são divididas em:

- **Processadores vetoriais**: São máquinas que possuem um suporte a cálculos massivos de vetores e matrizes. Possui um *hardware* específico para fazer os cálculos sobre os vetores, tendo como organização básica uma unidade de processamento vetorial, um processador de instrução e um processador escalar (DUNCAN, 1990).
- **SIMD**: São máquinas com vários processadores que operam em vários fluxos de dados. Possui uma unidade de controle a qual coordena as instruções para que estas sejam executadas de forma síncrona. São dívidas em processadores matriciais e processadores associativos. Processadores matriciais são utilizados em casos específicos para cálculos pesados sobre matrizes de dados. O acesso à memória é feito via endereço, o que diferencia dos processadores associativos

é que sua memória é acessada pelo conteúdo. Nos processadores associativos é usada uma lógica especial de comparação que determina o acesso paralelo aos dados de acordo com seu conteúdo (DUNCAN,1990).

- **Arquiteturas Sistólicas**: Usadas em ocasiões que necessitam de grande poder de processamento e várias operações de E/S. Neste tipo de arquitetura os processadores se organizam por meio de uma fila, no qual somente o processador da ponta tem acesso à memória, fazendo com que esses processadores executem um certo fluxo de dados de forma sincronizada.

As arquiteturas assíncronas são aquelas que possuem um controle descentralizado do *hardware*, e suas instruções são executadas em processadores distintos. Essas arquiteturas assíncronas podem ser subdivididas em MIMD convencionais e não convencionais.

MIMD convencionais: são caracterizadas pela organização de memória, podendo ser distribuída ou compartilhada**.** Uma vez que os processadores são independentes um do outro, um ponto particularmente importante e que deve ser levado em consideração é a comunicação que deve existir entre eles. Tendo em vista o melhor desempenho computacional, a comunicação e o sincronismo entre esses processadores são de tamanha importância e estes pontos têm uma relação com a organização da memória como citada acima.

As arquiteturas de memória distribuída (multicomputador) ilustradas na

Figura 6 são caracterizadas por fazer a comunicação de nós de processamento por troca de mensagens, nós estes conectados por uma rede de interconexão. O processador de cada nó só utiliza o seu espaço de memória (DUNCAN,1990). Esta troca de mensagens já resolve o problema da comunicação e da sincronização.

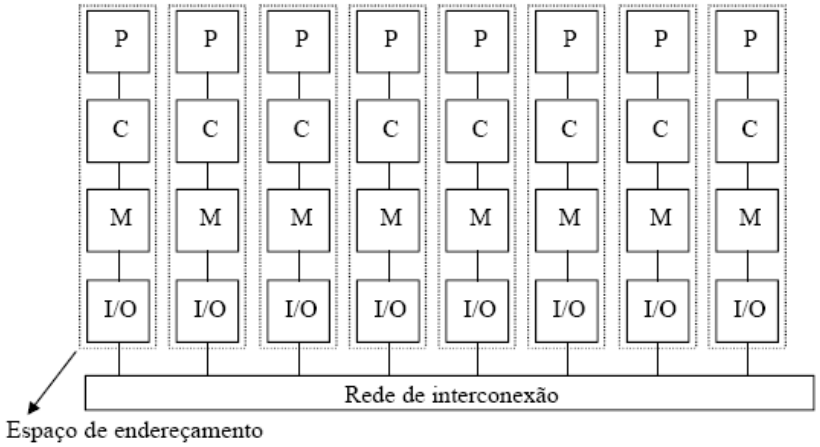


Figura 6 – Multicomputador

Já nas arquiteturas de memória compartilhada (multiprocessador), ilustrada na Figura 7 , cada processador tem acesso a todo espaço de memória da arquitetura. Com este compartilhamento, os dados na memória podem ser acessados por todos os processadores, às vezes acarretando alguns problemas. Para coordenar os processadores, existem alguns mecanismos de sincronização. Estes mecanismos podem liberar o acesso a uma parte da memória ou indicar que ela se encontra em uso.

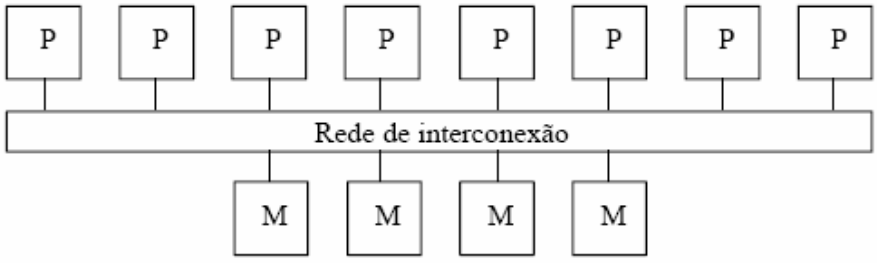


Figura 7 – Multiprocessador

MIMD não convencionais: essas arquiteturas dividem-se em MSIMD, *dataflow,* frente de onda e de redução.

MSIMD são arquiteturas híbridas (MIMD/SIMD), partes da arquitetura MIMD são controladas de modo SIMD (DUNCAN, 1990).

A principal característica das arquiteturas *Dataflow* é o modo com que as instruções são executadas. À medida que os operando vão se tornando disponíveis, as instruções vão sendo executadas dependentes dos fluxos de dados.

Arquiteturas de redução são máquinas dirigidas por demanda, ou seja, uma instrução é executada a partir de um resultado oferecido por outra instrução ou operando. Nessas arquiteturas partes do código fonte original são substituídas por seus resultados (DUNCAN, 1990).

Arquiteturas de frente de onda combinam estruturas *pipeline* sistólicas com a execução assíncrona do fluxo de dados (*dataflow*).

A classificação de arquiteturas, Flynn, mesmo possuindo falhas é ainda a taxonomia mais utilizada, entretanto a classificação proposta por Duncan incorpora um número maior de arquiteturas incluindo, inclusive, as mais recentes.

1.6 Programação Concorrente

Para o entendimento da programação concorrente faz-se necessária a explicação do funcionamento de um programa seqüencial. O programa seqüencial possui instruções definidas pelo programador. Estas instruções são executadas de forma seqüencial, portanto a execução de uma programação concorrente é gerada através de dois ou mais programas seqüências, estes sendo executados produzindo um certo paralelismo (pseudoparalelismo) (SANTANA et al, 1997).

O controle desta concorrência é necessário, em meio a estas execuções é preciso determinar: qual processo deve ser finalizado, qual deve ser inicializado entre outros. Notações são utilizadas para representar esta execução concorrente, algumas são apresentadas a seguir.

Fork/Join: Basicamente é gerado um processo filho que executa em paralelo com o processo pai. No momento do disparo do processo filho (*Fork*), o processo pai continua sua execução, neste momento haverá o paralelismo desses dois processos, até que haja a sincronização (*Join*) como pode ser visto na Figura 8.

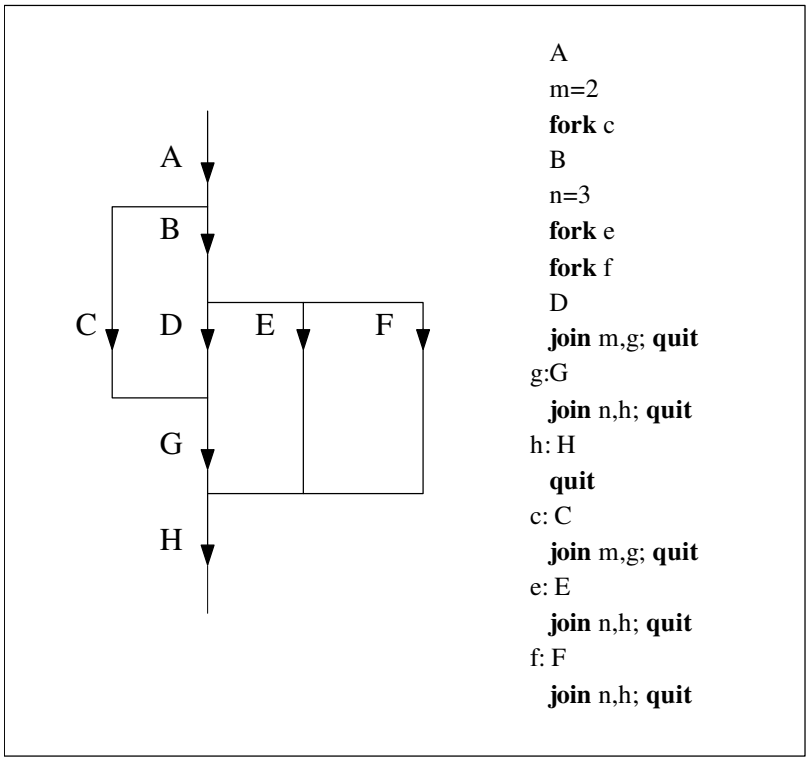


Figura 8 – Funcionamento do fork/join (SANTANA et al, 1997)

Na Figura 8 pode-se observar que no primeiro momento o processo pai espera o processo filho terminar para continuar a execução, já no segundo momento o processo filho é finalizado primeiro.

Cobegin/Coend: É uma forma estruturada na qual se podem ativar vários processos concorrentes. A seguir a sintaxe:

COBEGIN P_1//P_2//P_3//..//P_n COEND

No exemplo acima se pode dizer que P_1,P_2,P_3,..P_n sejam chamadas de procedimento, enquanto estes procedimentos não terminam o processo pai fica bloqueado.

Corotinas: São subrotinas que não possuem hierarquia na transferência de controle. *Call* inicia uma corotina, *Resume* transfere para outra corotina, *return* volta para corotina que a chamou.

Deve-se saber distinguir a ativação de processos e a sincronização dos mesmos. Algumas aplicações necessitam de mais de um processo, fazendo com que estes tenham que se comunicar. Pode-se dividir estas primitivas de sincronização dependo da organização da memória.

Na memória compartilhada tem-se a implementação de exclusão mútua, que pode ser definida, como o acesso de processos de forma exclusiva a regiões criticas, pode-se chegar a exclusão mútua de várias maneiras (TANEUMBAUM,1995):

- ***Busy waiting*** **(espera ocupada):** A espera ocupada funciona com uma variável compartilhada que pode ser modificada. Basicamente o processo entra em um *loop* testando a variável, quando a região crítica estiver desocupada a variável é modificada, podendo assim, o processo que esta em *loop* entrar na região crítica. Possui algumas desvantagens como tempo gasto da UCP e sua implementação é difícil, geralmente sua programação é de mais baixo nível (TANEUMBAUM,1995).
- **Semáforo:** São mais sofisticados que o *busy waiting*, possui a mesma função que é a implementação da exclusão mútua, porém descarta a desvantagem de

perda de tempo da UCP. Funciona com duas operações sobre a variável compartilhada, p e v (*down* e *up*), sendo essas operações indivisíveis.

- **Monitores:** ao contrário dos semáforos os monitores apresentam uma forma mais estruturada para implementar a exclusão mútua. Para a implementação são utilizadas variáveis que representam o estado de algum recurso compartilhado e procedimentos que implementam operações sobre esses recursos. Essas variáveis podem ser obtidas somente pelos procedimentos internos a cada monitor, e a execução desses procedimentos é feita de maneira mutuamente exclusiva (TANEUMBAUM,1995).

Uma vez que a forma mais utilizada de arquitetura paralela é a MIMD de memória distribuída, e sendo que essas arquiteturas possuem memórias locais, tem-se que fazer a comunicação desses processos via troca de mensagens. Para obter-se este tipo de comunicação são utilizadas duas primitivas básicas que são *send*, que envia até um destino, e *receive*, recebe da máquina que esta enviando a mensagem.

Essas primitivas *send/receive* podem ser bloqueantes ou não-bloqueantes. Na bloqueante o processo que envia a mensagem fica bloqueado esperando a confirmação do destino. Já o não bloqueante continua com a execução, não se bloqueia.

Partindo destas primitivas tem-se três tipos de bases para a implementação da troca de mensagens:

- **Ponto a ponto:** Comunicação unidirecional com o uso das primitivas *send/receive* bloqueante (ilustrada na Figura 9 a)**.**
- **Rendezvouz:** Utiliza-se também das primitivas *send/receive* bloqueantes, entretanto de forma bidirecional, de modo que um processo executa um trecho de código em outro processo (ilustrada na Figura 9 b).

- **RPC** (*remote procedure call*): funciona como a execução de procedimentos não locais, usando o mesmo funcionamento de chamadas locais e dando uma idéia de transparência. O processo que esta requisitando um determinado serviço fica bloqueado até que se obtenha um resultado como apresentado na Figura 10.

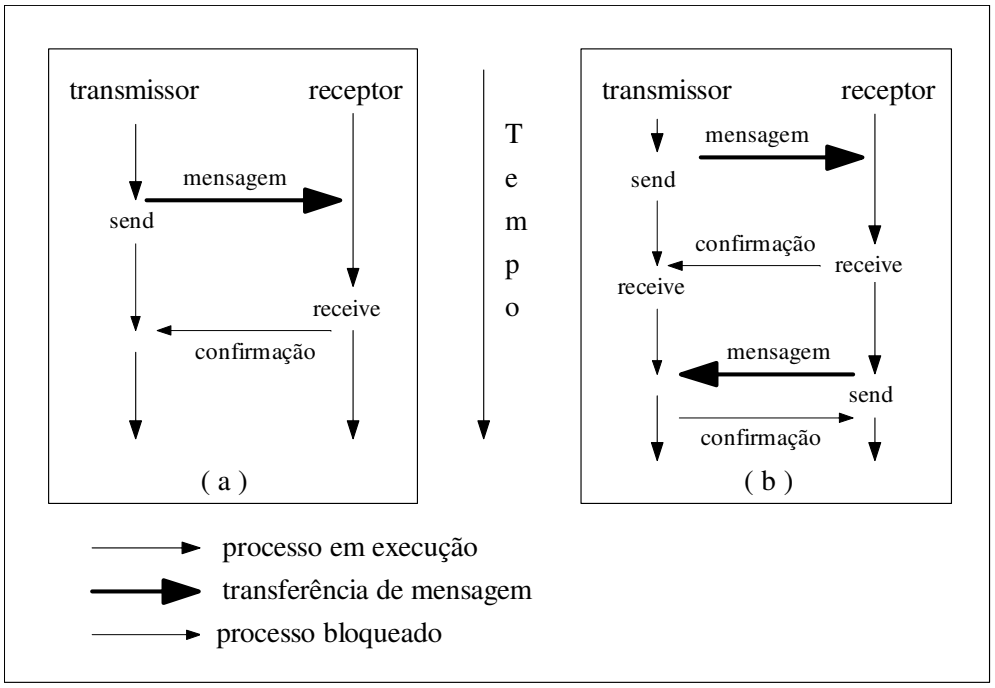


Figura 9 – Primitivas de sincronização (a) Ponto-a-Ponto (b) *Rendezvouz* (SANTANA et al, 1997)

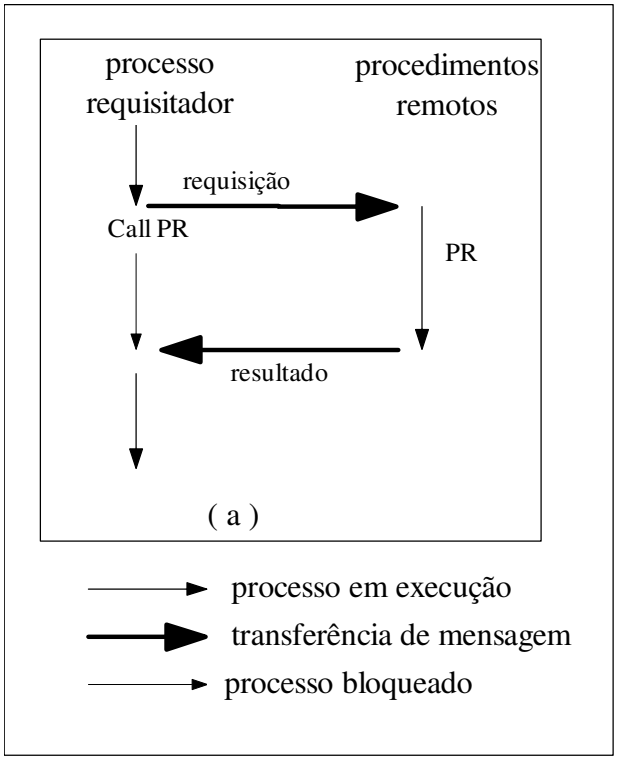


Figura 10 – RPC (SANTANA et al,1997)

1.7 Sistemas Computacionais Distribuídos

Com a evolução e o desenvolvimento dos computadores, das redes de comunicação e da necessidade de compartilhar recursos, os sistemas distribuídos surgiram e apresentaram uma grande evolução.

Segundo (TANEUMBAUM,1995) pode-se definir os sistemas distribuídos como "uma coleção de computadores autônomos, interligados por uma rede de comunicação e equipados com um sistema operacional distribuído, que permite o compartilhamento transparente de recursos existentes no sistema". Mostrado dessa maneira, transparente ao usuário, esse sistema passa a ser visto como se fosse apenas uma máquina centralizada.

Os Sistemas Distribuídos apresentam algumas vantagens em relação aos sistemas centralizados, podendo-se apontar a economia, uma vez que microprocessadores têm um custo bem menor que nos computadores de grande porte (*mainframes*), vantagem que auxilia a utilização dos Sistemas Distribuídos em vez de sistemas centralizados.

Tem-se também como vantagem a velocidade, aliando toda a potência computacional dos sistemas distribuídos pode-se ultrapassar a potência computacional de um *mainframe*.

Pode-se citar ainda várias outras vantagens, como confiabilidade, crescimento incremental, compartilhamento de dados e de dispositivos. Entretanto esses sistemas não apresentam somente vantagens, sendo que as principais desvantagens estão relacionadas ao *software*, à rede de comunicação e à segurança dos dados.

Para que um sistema seja considerado distribuído ele deve apresentar as seguintes características:

- **Compartilhamento de recursos**: Característica fundamental, compartilhamento de dispositivos e dados são essências, não somente em Sistemas Distribuídos como em sistemas centralizados.
- **Abertura**: Capacidade de expandir o sistema tanto em *software* quanto em *hardware* sem mudança no funcionamento do sistema distribuído (TANEUMBAUM,1995) (JAQUIE, 1999).
- **Concorrência**: Ocorre pelo fato de existiram varias execuções de processos, independência de recurso e indefinição de localização de tarefas, tornando mais complexo o gerenciamento (COLOURIS,1994)
- **Escalabilidade**: Refere-se a capacidade do sistema expandir-se com o mínimo de degradação de desempenho.
- **Tolerância a Falhas**: Por possuir vários processos em diferentes máquinas os sistemas distribuídos têm que possuir um suporte a tolerância a falhas. Pode-se obter a

tolerância a falhas por redundância de *hardware*, replicação de componentes e de redundância de *software*.

- **Transparência**: Capacidade de mostrar ao usuário que o sistema é único. Tem vários aspectos que podem ser citados da transparência, alguns deles: transparência de localização é quando os usuários não têm conhecimento onde estão os recursos. Transparência de paralelismo é quando atividades podem acontecer em paralelo sem que o usuário saiba. A transparência pode ser aplicada em outros aspectos como transparência de migração, os recursos podem mudar de localização sem mudança de nome, transparência de acesso, o usuário pode acessar algum recurso em qualquer lugar do sistema.

Outro aspecto pertinente em sistemas distribuídos é a comunicação. A comunicação é muito importante, uma vez que os recursos estão dispersos fisicamente.

A comunicação pode ser feita de duas maneiras, por troca de mensagens ou RPC. Troca de mensagens pode ser feita por *send/receive* bloqueantes ou não bloqueantes como foi mencionado anteriormente. O RPC por sua vez da uma certa transparência, por fazer chamadas em máquinas diferentes se parecer com chamadas locais.

1.8 Computação paralela e Sistemas Distribuídos

Mesmo aparecendo por razões diferentes, os sistemas distribuídos e a computação paralela acabaram por convergir, levando à computação paralela distribuída.

Antes a computação paralela era utilizada somente em arquiteturas SIMD, apresentando um alto custo e desvantagens, com a evolução das arquiteturas MIMD com

memória distribuída, a computação paralela pode fazer uso das vantagens apresentadas por essas.

Muitas pesquisas são ainda efetuadas, principalmente no que diz respeito à comunicação, uma vez que esta ainda constitui os "gargalos" quando se pretende obter maior desempenho utilizando máquinas paralelas ao invés de máquinas seqüenciais ou *mainframes*.

Então, pode-se observar que essas duas áreas são de grande importância quando as unimos, trazendo inúmeras vantagens como redução de custo, aumento computacional e aumento do desempenho.

A utilização da computação paralela sobre sistemas distribuídos é feita através de ambientes de troca de mensagens.

1.9 Considerações Finais

Como foi apresentada, a computação paralela surgiu devido à necessidade de obter um desempenho computacional maior do que o que se conseguia com arquiteturas seqüenciais. As arquiteturas foram aprimoradas e desenvolvidas a fim de se ter um aumento computacional. Inúmeras classificações foram propostas, mas as mais difundidas foram de Flynn (FLYNN,96) e de Duncan (DUNCAN, 1990).

Com o grande avanço dos computadores e das redes de comunicação foram criados os sistemas distribuídos que tinham por objetivo realizar o compartilhamento de recursos. Mesmo tendo sido originadas por razões distintas, computação paralela e sistemas distribuídos convergiram. Várias soluções, vantagens e melhorias foram sendo criadas após a união dessas duas áreas. Aumento computacional, economia, desempenho, são algumas delas.

A utilização da computação paralela sobre sistemas distribuídos dá origem à computação paralela distribuída, que é viabilizada pelo uso de ambientes de passagem de mensagem. Deste modo, o próximo capítulo apresenta os conceitos de ambientes de passagem de mensagem além de apresentar alguns dos ambientes mais utilizados na literatura.

CAPÍTULO 2 – AMBIENTES DE PASSAGEM DE MENSAGENS

Neste capítulo será versado sobre ambientes paralelos virtuais, mais especificamente sobre PVM, MPI e JPVM. Será feita uma breve descrição sobre eles, mostrando as principais características e funcionalidades. Maiores detalhes serão dados sobre JPVM uma vez que se trata do ambiente foco deste trabalho.

2.1 Considerações iniciais

Muitos ambientes foram criados, não só com o intuito de criar uma portabilidade, mas também executar em sistemas heterogêneos, alguns deles: P4 , Parmacs , Express, Linda , PVM (*Parallel Virtual Machine*), MPI (*Message Passing Interface*) entre outros.

O aparecimento desses sistemas se deu devido ao destaque do modelo computacional MIMD dentro da computação paralela, conseguido pela sua flexibilidade e seu potencial para execução de programas paralelos de média e alta complexidade. As plataformas de memória distribuída têm maior destaque sobre as de memória compartilhada, uma vez que os são baseados em computadores de arquiteturas paralelas ou em máquinas paralelas virtuais, essas construídas a partir de sistemas distribuídos.

Uma vez que os ambientes de passagem de mensagem possuem várias vantagens, os fabricantes criaram seus próprios ambientes ocasionando uma falta de padrão. Ao longo do tempo deu-se a preocupação com a padronização e alguns ambientes foram desenvolvidos para que houvesse maior facilidade de portabilidade.

Os ambientes de passagem de mensagem têm tido grande destaque na literatura, não só pela flexibilidade, mas também pelo fato de constituírem um tipo de solução para o problema da portabilidade de programas paralelos entre sistemas diferentes (JAQUIE, 1999). Diferente das máquinas maciçamente paralelas estes ambientes são utilizados para compor uma máquina paralela virtual, ou seja, fazer uso de computadores interconectados sejam eles homogêneos ou heterogêneos, executando uma aplicação de forma paralela.

2.2 PVM

O PVM (*Paralel Virtual Machine*) é um conjunto de bibliotecas, que por meio de troca de mensagens, pode emular um sistema paralelo. Foi uns dos primeiros ambientes a funcionar em máquinas heterogêneas e no qual pesquisadores demonstraram interesse.

Teve início em 1989 na ORNL *(Oak Ridge National Laboratory)* onde foi usado somente em laboratório. Após algumas modificações foi disponibilizada a versão 2 em 1992. Utilizada em várias aplicações científicas, esta versão foi muito importante para o desenvolvimento do PVM.

O PVM possui várias rotinas que permitem que sejam efetuados a sincronização e o comando das tarefas na máquina paralela virtual. Essa máquina paralela virtual lida de forma transparente com as tarefas através de passagem de mensagens, que é o meio de comunicação entre os vários computadores interligados (OLIVEIRA,2005).

Como foi dito o PVM executa em máquinas heterogêneas, abaixo uma tabela em quais sistemas o PVM é executado:

Tabela 1 – Principais equipamentos que executam o PVM (SUNDERAM,1994)

Alliant FX/8
DEC Alpha
Sequent Balance
Bbn Butterfly TC2000
80386/486/Pentium com UNIX (Linux ou BSD)
Thinking Machines CM2 CM5
Convex C-series
C-90, Ymp, Cray-2, Cray S-MP
HP-9000 modelo 300, Hp-9000 PA-RISC
Intel iPSC/860, Intel iPSC/2 386 host
Intel Paragon
DECstation 3100, 5100
IBM/RS6000, IBM RT
Silicon Graphics
Sun 3, Sun 4, SPARCstation, Sparc multiprocessor
DEV Micro VAX

O PVM permite que sejam escritas aplicações nas seguintes linguagens: Fortran, C e C++. Ele foi projetado para se hospedar em tais linguagens uma vez que a maioria das aplicações que são paralelizáveis estão escritas nas mesmas. Está disponível em *Unix*, seus derivados e *Windows* NT (OLIVEIRA,2005).

Seu funcionamento é feito por meio de execução de tarefas (*tasks*), onde cada tarefa equivale a uma parte da aplicação, caracterizando um paralelismo de granulação de média a grossa. Permite SPMD (*Single Program Multiple Data*), MPMD (*Multiple Program Multiple Data*) e métodos híbridos de programação. SPMD segundo (SANTANA et al,1997) é um método em que consiste um único programa sendo executado em todos os processadores de

uma máquina MIMD, sendo o mesmo código para todos os processadores. O MPMD, conhecido também como paradigma mestre-escravo, funciona da seguinte maneira: Um processo controla todas as tarefas (mestre), estas tarefas são executadas pelos outros processos (escravos), estes podendo executar códigos distintos ou até o mesmo código do mestre.

O PVM possui dois componentes, o PVM *daemon* (*pvmd*), que é executado em todas as máquinas que compõem a máquina paralela virtual, e o outro componente é a biblioteca de rotinas com interface PVM (LibPVM), esta biblioteca possui um conjunto de primitivas que são utilizadas nas comunicações entres as máquinas.

O *pvmd* pode ser instalado por qualquer usuário com *login* válido, sendo este executado em todos os nós da máquina paralela virtual, sendo assim, o *pvmd*.deve ser instalado em todas as máquinas pois é responsável pela troca de mensagens e a coordenação das tarefas da aplicação.

O primeiro *pvmd* configurado é chamado de mestre, responsável por chamar os outros nós, chamados de escravos. Deste modo é feita a sincronização e comunicação entre os nós para que a aplicação seja executada de forma paralela.

A *Libpvm* apresenta rotinas que são utilizadas: para fazer a comunicação dos nós da máquina paralela virtual, para gerenciar processos, para coordenar as tarefas, e para fazer a verificação e manutenção da máquina virtual (OLIVEIRA, 2005).

O PVM apresenta duas interfaces de linguagem (FORTRAN, C/C++), por ser mais utilizada serão abordados a seguir alguns comandos básicos da interface C (SUNDERAM,1994):

- pvm_mytid(): Geralmente é a primeira função chamada no programa, tem a funcionalidade de registrar o processo na máquina virtual e informar ao processo seu identificador na máquina virtual

- pvm_spawn(): Usada no programa mestre tem como função lançar os demais escravos nos nós da maquina virtual.
- pvm_parent(): Retorna o identificador do processo mestre que gerou o processo atual. Necessário para o programa escravo trocar informações com o processo mestre.
- pvm_initsend(): Função relacionada com envio de mensagens. Tem como função definir a codificação que será usada para troca de dados. Deve ser usada pelo menos uma vez no programa.
- pvm_pk??(): Usada para empacotar um conjunto de dados em um *buffer* ativo. Dependendo do tipo de dado é usada uma determinada função. Usando pvm_pkint() estão sendo empacotados dados do tipo *int*, usando pvm_pkdouble(), do tipo *double* e assim por diante.
- pvm_send(): envia dados a uma determinada tarefa. Envia a mensagem previamente empacotada. Quando se deseja enviar para mais de um processo *(multicasting*) é usada a função pvm_msend().
- Pvm_upk(): tem como função desempacotar os dados e tirar do *buffer* a mensagem. Funciona da mesma maneira que o pvm_pk, em relação aos tipos de dados.

A Figura 11 apresenta um exemplo de um programa utilizando a biblioteca PVM com hospedeiro C:

```
#include "pvm3.h"
main()
{
  int cc, tid, msgtag;
  char buf[100];
  printf("i'm %x", pvm_mytid());
  cc = pvm_spawn("hello_other",
        (char **)0,0,"", 1, &tid);
  if (cc==1){
    msgtag=1;
    pvm_recv (tid, msgtag);
    pvm_upkstr(buf);
    printf("from %x: %s\n", tid, buf);
  }else
   printf("can't start hello_other\n");
  pvm_exit();
}
```

```
#include "pvm3.h"
main()
{
  int ptid, msgtag;
  char buf[100];
  ptid = pvm_parent();
  strcpy(buf, "hello, world from");
  gethostname(buf+strlen(buf), 64);
  msgtag=1;
  pvm_initsend (PvmDataDefault);
  pvm_pkstr (buf);
  pvm_send (ptid, msgtag);
  pvm_exit();
}
```

Figura 11 – Exemplo de um programa PVM em linguagem C

2.3 MPI

Como uma tentativa de padronizar os ambientes de passagem de mensagem surgiu o MPI (*Message Passing Interface*) (SANTANA et al,1997). Essa nova especificação teve como objetivos principais a padronização para fabricantes de *hardware* e plataformas portáteis, e uma melhor eficiência. A padronização era necessária, visto que com o crescimento dos ambientes a portabilidade acaba sendo um fator importante.

Os desenvolvedores do MPI se preocuparam com as características e vantagens dos demais ambientes, buscando incorporar novas funcionalidades mas mantendo algumas existentes no P4, PVM, Express, entre outros.

Enquanto o PVM é uma implementação o MPI é apenas uma especificação sintática e semântica de rotinas constituintes da biblioteca de comunicação, podendo ser implementado de maneiras distintas, como exemplo pode ser observado que em 1996 existiam pelo menos

15 implementações do MPI (LINHALIS, 1998). Alguns exemplos *free*, ou seja, exemplos de sua implementação e distribuição livre são:

- CHIMP/MPI: *Edinbugh Parallel Computing Centre* (EPCC);
- LAM: *Ohio Supercomputer Center*;
- MPIAP: *Australian National University*;
- MPICH: *Argonne National Laboratory & Mississipi State University*;
- MPI-FM: *University of Illinois*;
- W32MPI: *Universidade de Coimbra – Portugal*;

O MPI é constituído por 129 rotinas. Essas rotinas oferecem alguns serviços, são eles (JAQUIE,1999):

- Comunicação ponto-a-ponto;
- Comunicação coletiva;
- Suporte para grupo de processos;
- Suporte para contextos de comunicação
- Suporte para topologia de processos.

Muitas são as questões que envolvem a padronização desses ambientes, pode-se ver através do MPI alguns pontos interessantes, que motivam a pesquisa e continuidade de aperfeiçoamento do MPI. Alguns desses pontos são (LINHALIS, 98):

- **Portabilidade e facilidade de uso**: À medida que a utilização do MPI aumentar será possível portar transparentemente aplicações entre um grande número de plataformas paralelas;
- **Fornecer uma especificação precisa**: Fabricantes de *hardware* podem implementar eficientemente em suas máquinas um conjunto bem definido de rotinas;

- **Crescimento da indústria de *software* paralelo**: A existência de um padrão torna a criação de *software* paralelo por empresas uma opção comercialmente viável, o que também implica em maior difusão do uso de computadores paralelos.

Seu funcionamento é baseado em rotinas, rotinas essas que enviam e recebem dados. Na comunicação de mensagens o MPI funciona da seguinte maneira: é definido um tipo de dado que deseja enviar, estes dados são guardados em um vetor e guarda-se o endereço do primeiro elemento e o número de elementos que compõem o vetor. No MPI é possível empacotar dados de diferentes tipos, no caso uma *struct*.

Algumas rotinas básicas do MPI utilizadas para a troca de mensagens são apresentadas a seguir:

- MPI_SEND (): rotina usada para envio de mensagens. Nela faz-se necessário especificar o tipo de dado, para onde está enviando e o número de elementos. Esta rotina está dividida em 4 modos: padrão, *bufferizada,* síncrono e pronto. Podendo ser bloqueantes e não-bloqueantes.
- MPI_Init(): registra a aplicação junto à sessão MPI.
- MPI_Recv(): recebimento bloqueante de mensagens, ou seja espera até que seja confirmado o recebimento da mensagem no *buffer*.
- MPI_IRecv(): recebimento não-bloqueante, não aguarda a confirmação de recebimento da mensagem.
- MPI_Finalize(): Informa que um processo esta saindo da sessão do MPI.
- MPI_Abort(): Termina todos os processos em um mesmo comunicador.

A Figura 12 apresenta um exemplo de programa usando o MPI com hospedeiro C:

```
#include <mpi.h>
#define BUFSIZE 64
int buf[BUFSIZE];
main(argc, argv)
int argc;
char *argv[];
{
int size, rank;
MPI_Status status;
MPI_Init (&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (size!=2){
MPI_finalize();
return (1);
}
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
if (rank==0)
MPI_Send(buf, sizeof(buf),MPI_INT,
1,11, MPI_COMM_WORLD);
else
MPI_Recv(buf,sizeof(buf),MPI_INT,
0,11,MPI_COMM_WORLD,&status);
MPI_Finalize();
return (0);
}
```

Figura 12 – Programa em MPI implementação em linguagem C

2.4 JPVM

Como visto anteriormente o PVM foi uma das primeiras bibliotecas a ser implementada para sistemas heterogêneos. Porém esta característica pode apresentar algumas falhas quando se executa aplicações paralelas em sistemas operacionais distintos.

Com a tentativa de minimizar estas falhas o JPVM (OLIVEIRA,2005) foi criado, com o mesmo princípio do PVM, sua comunicação é feita através de passagem de mensagem. Ele é constituído de uma API pequena implementada em Java usada para ser rodado em uma arquitetura sistemas MIMD de memória distribuída (OLIVEIRA, 2005).

O JPVM apresenta uma portabilidade maior que o PVM, por apresentar algumas vantagens em relação à implementação de programas concorrentes, devido a características próprias da linguagem Java (FERRARI, 98).

A interface de programação do JPVM é similar a do PVM. Como foi visto o PVM funciona através da alocação de tarefas (*Tasks*) que executam de forma seqüencial. No JPVM as implementações das tarefas são codificadas em Java e o suporte para criação de tarefas e da passagem de mensagem provém das bibliotecas do JPVM.

Faz-se importante ressaltar que o JPVM é totalmente funcional sem a instalação do PVM, o contrário de algumas implementações de PVM em outras linguagens que necessitavam primeiramente do PVM para o funcionamento.

2.4.1 Funcionamento do JPVM

Bem similar ao PVM o JPVM é constituído de dois componentes, o *jpvmDaemon* e o *jpvmEnvironment*.

O *jpvmDaemon* tem a função de coordenar às tarefas e fazer a comunicação dos processos e sua execução ocorre em segundo plano nos nós da máquina virtual (OLIVEIRA, 2005).

O *jpvmEnvironment* é uma biblioteca que implementa um conjunto completo de primitivas que permitem codificar, enviar e receber mensagens, criar e eliminar processos, sincronizar tarefas e modificar a máquina virtual. As aplicações desenvolvidas pelo usuário utilizam o ambiente paralelo criado pelo JPVM através dessa biblioteca

A Figura 13 apresenta a interface provida pela biblioteca *jpvmEnvironment* ilustrando as principais funções existentes.

```
class jpvmEnvironment {
   public jpvmEnvironment(); // Constructor registers task with the JPVM system
   public void pvm_exit();

   // Identity:
   public jpvmTaskId pvm_mytid();
   public jpvmTaskId pvm_parent();

   // Task creation:
   public int pvm_spawn(String task_name, int num, jpvmTaskId tids[]);

   // Send messages:
   public void pvm_send(jpvmBuffer buf, jpvmTaskId tid, int tag);
   public void pvm_mcast(jpvmBuffer buf, jpvmTaskId tids[], int ntids, int tag);

   // Receive messages, blocking (non-blocking versions not depicted):
   public jpvmMessage pvm_recv(jpvmTaskId tid, int tag);
   public jpvmMessage pvm_recv(jpvmTaskId tid);
   public jpvmMessage pvm_recv(int tag);
   public jpvmMessage pvm_recv();

   //Probe for message availability:
   public boolean pvm_probe(jpvmTaskId tid, int tag);
   public boolean pvm_probe(jpvmTaskId tid);
   public boolean pvm_probe(int tag);
   public boolean pvm_probe();

   // System configuration and control:
   public jpvmConfiguration pvm_config();
   public jpvmTaskStatus pvm_tasks(jpvmConfiguration conf, int which);
   public void pvm_halt();
};
```

Figura 13 – Interface do *jpvmEnvironmet* (FERRARI,1997)

Diferente do PVM, no qual o protocolo de comunicação é o UDP, a implementação de passagem de mensagem do JPVM é baseada na comunicação sobre *TCP sockets.* Cada instância do *jpvmEnvironment* cria um *server socket* e disponibiliza o nome do *host* e o número da porta de conexão. Internamente, o JPVM usa *threads* para gerenciar as conexões e a troca de mensagens. Quando uma tarefa X deseja se comunicar com uma tarefa Y, ela simplesmente se conecta com Y usando o nome do *host* e a porta que está contido no identificador da tarefa Y. Quando uma nova conexão é aceita, cria-se uma nova *thread* dedicada à gerência da mesma. (FERRARI, 98).

As tarefas são identificadas por um número inteiro e estes identificadores são os *jpvmTaskId.*

A Figura 14 ilustra a comunicação do JPVM:

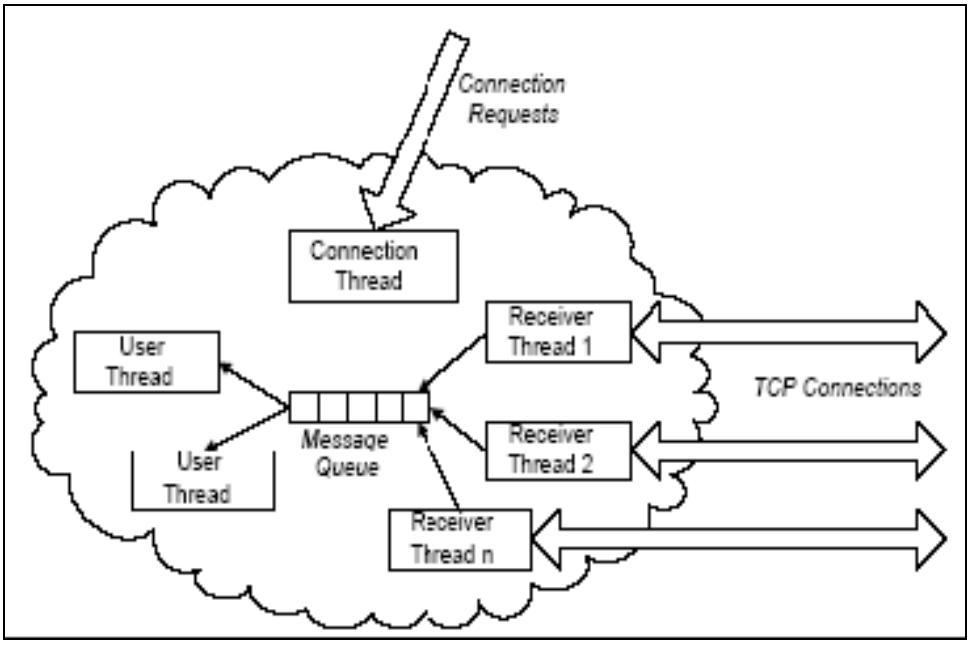


Figura 14 – Comunicação do jpvm (FERRARI,1997)

As tarefas são criadas usando-se do método *pvm_spawn*, onde este possui 3 parâmetros: o nome da classe a ser instanciada, a quantidade de instâncias a serem criadas e um vetor que recebe os identificadores. Cada tarefa criada pelo *pvm_spawn()* executa na sua própria instância da máquina virtual, evitando conflitos entre as tarefas.

Pvm_send() e *Pvm_recv()* são métodos da classe *jpvmEnvironment* que são usados para executar a passagem de mensagem no JPVM. Este método de comunicação pode ser, síncrono, assíncrono e bloqueante, como acontece no PVM.

Pvm_send() possui 3 parâmetros: um *buffer* onde a mensagem é guardada, o *TaskId* para onde a mensagem será enviada e um valor inteiro que identifica a mensagem.

Como no PVM tradicional, pode-se ter um envio *multicast* usando o método *pvm_mcast()*. No caso do *pvm_mcast* necessita de outros parâmetros, como o vetor dos *TaskId*s e a quantidade de destinatários.

Para receber mensagens é utilizado o método *pvm_recv()*, que é bloqueante e bloqueia a tarefa até que ela receba a mensagem.

Existem 3 tipos de recebimento de mensagem sendo elas:

- *public jpvmMessage pvm_recv(jpvmTaskId tid, int tag):* permite especificar o *TaskId* e o *tag;*
- *public jpvmMessage pvm_recv(jpvmTaskId tid):* permite especificar o *Task Id*;
- *public jpvmMessage pvm_recv(int tag)*: permite especificar o *tag*.

Os métodos de controle são usados para desenvolver novos métodos e geralmente não são usados pelo programador que está desenvolvendo a aplicação por ser transparente.

- *public jpvmConfiguration pvm_config()*: retorna o *status* da máquina virtual. Informa o número de *hosts*, os identificadores das máquinas e dos *daemons* que estão executando em cada nó.
- *public jpvmTaskStatus pvm_tasks(jpvmConfiguration conf, int which)*: retorna o *status* das tarefas que estão em execução em um determinado nó. Tem como parâmetro a configuração atual da máquina paralela virtual e um valor inteiro que identifica a posição a ser lida no vetor que contém os identificadores dos *daemons*
- *public void pvm_halt()*: método que tem por funcionalidade encerrar o JPVM. Apaga o registro das tarefas da máquina virtual.
- *public void pvm_addhosts(int nhosts, String hostnames[], jpvmTaskId daemonTids[])*: tem como função adicionar novas máquinas. Os parâmetros são: quantidade de máquinas a serem adicionadas, vetor com as identificações de cada máquina e as identificações dos *daemons* que estão executando em cada nó.

Com esses e outros métodos é possível que o programador desenvolva seu programa de forma paralela, com muita eficiência. Figura 15 ilustra um exemplo de um programa utilizando a biblioteca JPVM:

```
1  import jpvm.*;
2  class example {
3     public static void main(String args[]) {
4        try {
5           jpvmEnvironment jpvm = new jpvmEnvironment();
6
7           // Spawn N worker tasks
8           jpvmTaskId tids[] = new jpvmTaskId[N];
9           jpvm.pvm_spawn("worker",N,tids);
10
11          for (int i=0;i<N; i++) { // Farm out work
12             jpvmBuffer buf = new jpvmBuffer();
13             int work = getWork(i);
14             buf.pack(data);
15             jpvm.pvm_send(buf, tids[i], 123);
16          }
17          for (int i=0;i<N; i++) { // Receive results
18             jpvmMessage message = jpvm.pvm_recv(tids[i]);
19             int result = message.buffer.upkint();
20             processResult(result);
21          }
22          jpvm.pvm_exit();
23       }
24       catch (jpvmException jpe) {
25          System.out.println("Error - jpvm exception");
26       }
27    }
28 };
```

Figura 15– Exemplo de um programa utilizando JPVM linguagem hospedeira Java (FERRARI,1997)

2.4 Considerações Finais

Os ambientes de troca de mensagem tornaram-se muito úteis para serem aplicados na computação paralela distribuída, viabilizando-a.

O PVM foi um dos primeiros a executar aplicações em ambientes heterogêneos e em máquinas MIMD com memória distribuída sendo muito utilizado atualmente por permitir grande portabilidade.

Almejando melhorar ainda mais a portabilidade, a especificação MPI foi criada, visando maior eficiência e padronização dos ambientes existentes. O MPI manteve características de alguns ambientes existentes, mantendo assim todo o esforço dos desenvolvedores.

Entretanto, os ambientes paralelos virtuais existentes possuem algumas falhas quando aplicados a sistemas heterogêneos principalmente com diferentes sistemas operacionais.

Objetivando minimizar algumas falhas o JPVM (OLIVEIRA,2005) foi criado e tem apresentado uma maior portabilidade e principalmente interoperabilidade em relação ao PVM padrão.

Uma vez que essa biblioteca apresenta novas funcionalidades e provê uma portabilidade e uma interoperabilidade maior que os demais ambientes, o próximo capítulo apresenta o plano de trabalho que tem por objetivo efetuar a análise de desempenho de algumas aplicações paralelas em C utilizando a biblioteca de passagem de mensagem JPVM.

CAPÍTULO – 4 METODOLOGIA E RESULTADOS

Nesse capítulo é apresentado o uso da máquina virtual, mostrando os comandos e configurações para executar as aplicações em paralelo, os resultados da implementação, e a análise das duas aplicações (Multiplicação de Matrizes e Trapézio Composto) que foram utilizadas pra testar o uso da biblioteca de troca de mensagens JPVM.

4.1. PLATAFORMA DE DESENVOLVIMENTO

Para a realização do projeto foi utilizado um *cluster* composto por máquinas comuns, (quatro máquinas todas interligadas por uma rede). A tabela abaixo mostra a configuração das maquinas:

Tabela 2 – Configuração das máquinas do cluster

Processador	Pentium 4
Memória	512 mb
HD	40 gb

As máquinas possuem o sistema operacional linux – fedora core 5, e foi instalada a máquina virtual Java (JVM) – JDK 1.4.2 e 1.5.07 – e em todos os nós foi instalada a biblioteca JPVM para a comunicação entre os nós.

4.2. CONFIGURAÇÃO

Após a configuração dos nós do *cluster* foi feita a instalação da JVM (*Java virtual machine*) e do pacote JPVM. A Instalação do JPVM é simples, primeiramente o pacote foi encontrado no *site* http://www.cs.virginia.edu/~ajf2j/jpvm.html e descompactado para o diretório /root/jpvm, podendo ser descompactado em qualquer pasta desde que as variáveis de ambiente sejam configuradas corretamente posteriormente, como na figura abaixo:

```
export CLASSPATH="/usr/java/jdk1.5.0_07/lib":"/root/jpvm"
```

É necessário que as aplicações a serem executadas em paralelo estejam corretamente localizadas na pasta onde foi configurada a variável *classpath*, neste caso "/root/jpvm"

Na configuração do ambiente com o JPVM os nós são identificados pelos nomes, por isso foi utilizado o arquivo *hosts*, que se encontra na pasta /etc no *linux*. Este arquivo contém os nós com seus respectivos IPs, nomes e apelidos.

Após essas configurações do *cluster* o ambiente está pronto para a implementação e execução de aplicações paralelas. Quando necessária a inserção de novos nós escravos, basta repetir as configurações realizadas nos nós escravos que já estão em funcionamento. Além disso, a única alteração nos demais nós, incluindo o mestre, é inclusão do endereço IP e do nome da nova máquina no arquivo *hosts*. Na Figura 16 pode-se observar o ambiente utilizando JPVM executado em 4 *hosts*.

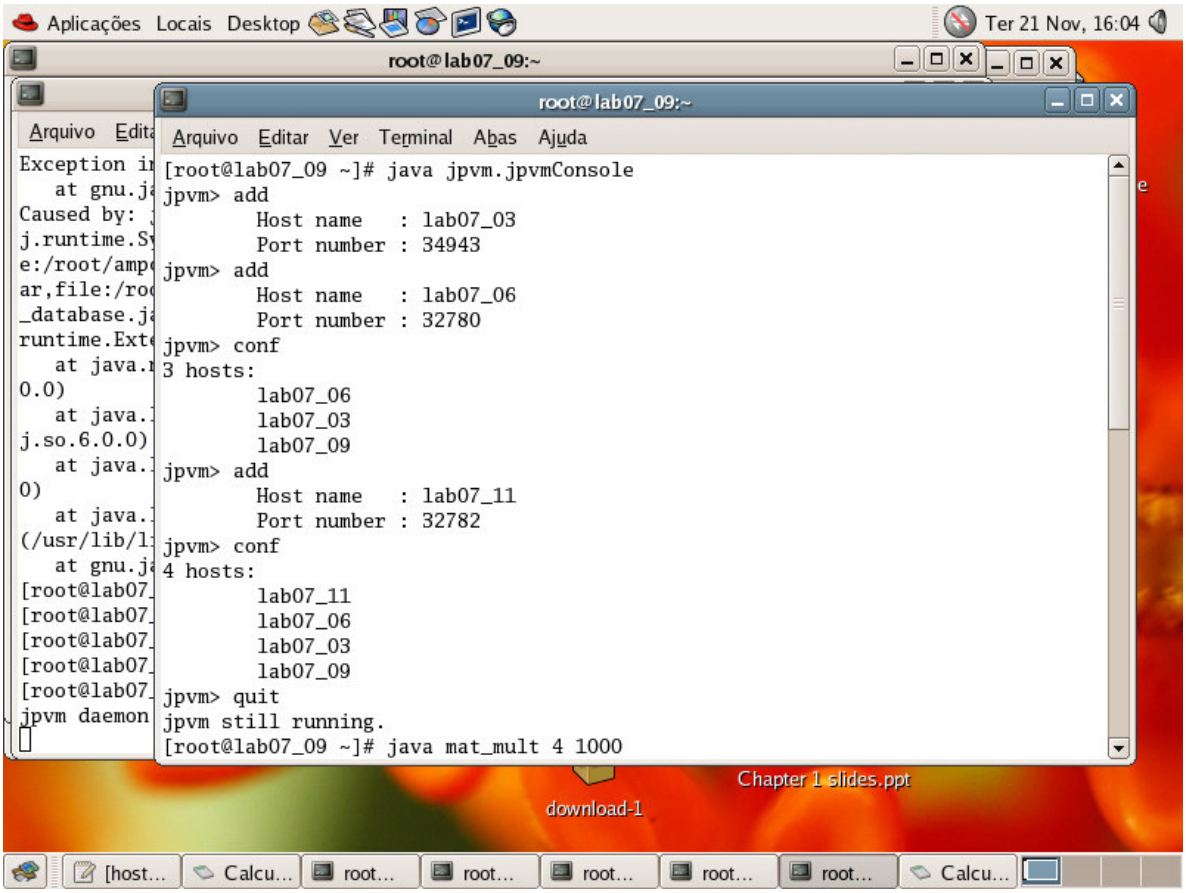


Figura 16 – *Daemons* ativos do JPVM sendo executados em 4 máquinas.

4.3. PROGRAMAS PARALELOS IMPLEMENTADOS

Para a realização dos testes foram utilizados dois programas de cálculos matemáticos baseado em algoritmos seqüenciais que foram paralelizados utilizando a interface JPVM. O primeiro a ser analisado é a aplicação de cálculo de matrizes (Multiplicação de Matrizes), e a segunda aplicação é o cálculo da integral (Método do Trapézio Composto).

4.3.1 Multiplicação de matriz

Para fazer o cálculo de matriz foi utilizado um programa que vem como exemplo da própria biblioteca JPVM (FERRARI,1997). A aplicação foi desenvolvida por Adam J.Ferrari e se encontra à disposição na pasta *examples* do JPVM

Antes de apresentar os cálculos será feita uma análise do código de multiplicação de matrizes.

```
...
jpvm = new jpvmEnvironment();
myTaskId = jpvm.pvm_mytid();
masterTaskId = jpvm.pvm_parent();
...
```

Figura 17 – Interface jpvm

A Figura 17 ilustra o código da definição do inicio da interface JPVM:

- *JpvmEnvironment()*: registro da classe atual no ambiente JPVM;
- *Jpvm.pvm_mytid()*: retorna o número de identificação dentro do ambiente;
- *jpvm.pvm_parent()*: retorna a identificação do processo pai que instanciou a aplicação atual.

```
...
start = msecond();
   taskMeshDim = getTaskMeshDim(numTasks);
   tids = new jpvmTaskId[numTasks];
   if(numTasks>1)
 jpvm.pvm_spawn("mat_mult",numTasks-1,tids);
   tids[numTasks-1] = tids[0];
   tids[0] = myTaskId;
   localTaskIndex = 0;

   /*Broadcast parameters to all tasks */
   mmstart = msecond();
   jpvmBuffer buf = new jpvmBuffer();
   buf.pack(taskMeshDim);
  buf.pack(numTasks);
   buf.pack(tids,numTasks,1);
   buf.pack(matDim);
  jpvm.pvm_mcast(buf,tids,numTasks,ParamTag);
....
```

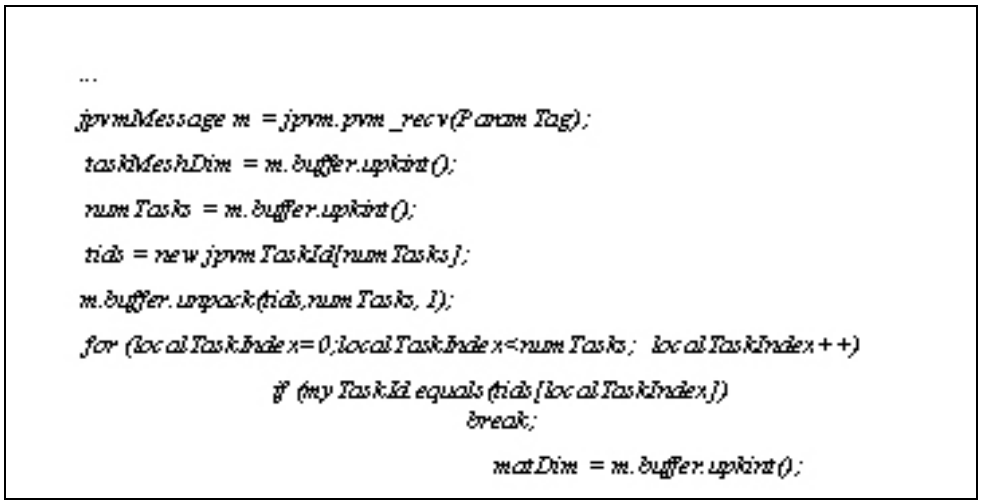
Figura 18 – Parte do código da multiplicação de matrizes.

Na Figura 18 pode-se observar o início da paralelização, é criado um vetor de *tids* (identificadores de processos) para receber o *tid* de uma das tarefas a serem instanciadas.

O método *jpvm.jpvm_spawn* é responsável por instanciar as classes que executam as tarefas, neste caso *"mat_mult"*, recebe como parâmetro o número de tarefas e o vetor que receberá a *tid* de cada uma das instâncias.

Depois de feito o *spawn* é feito um *broadcast* para todos os nós enviando os parâmetros da execução da aplicação.

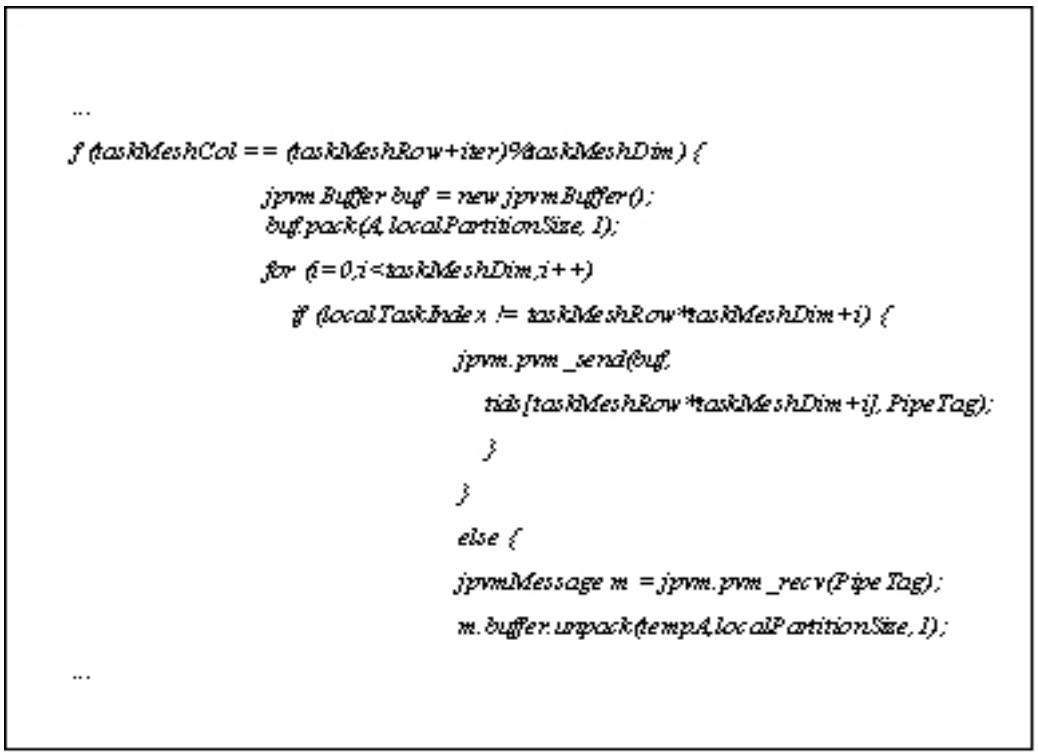
Essas são as instruções que o processo mestre vai efetuar, se for o processo escravo este executará comandos como mostra na Figura 19.



```
...
jpvmMessage m = jpvm.pvm_recv(ParamTag);
taskMeshDim = m.buffer.upkint();
numTasks = m.buffer.upkint();
tids = new jpvmTaskId[numTasks];
m.buffer.unpack(tids,numTasks, 1);
for (localTaskIndex=0;localTaskIndex<numTasks; localTaskIndex++)
        if (myTaskId.equals(tids[localTaskIndex])
                break;
                matDim = m.buffer.upkint();
```

Figura 19 – Parte do código do processo escravo

Nota-se que uma vez que o processo é escravo ele precisa receber os parâmetros do processo mestre, então é feito o desempacotamento dos parâmetros enviados. Este desempacotamento é efetuado pela função *upkint* e *unpack.* Além disso, todos os nós recebem também a dimensão da matriz a ser multiplicada.



```
...
if (taskMeshCol == (taskMeshRow+iter)%taskMeshDim) {
        jpvmBuffer buf = new jpvmBuffer();
        buf.pack(A, localPartitionSize, 1);
        for (i=0;i<taskMeshDim;i++)
            if (localTaskIndex != taskMeshRow*taskMeshDim+i) {
                jpvm.pvm_send(buf,
                    tids[taskMeshRow*taskMeshDim+i], PipeTag);
                }
            }
            else {
            jpvmMessage m = jpvm.pvm_recv(PipeTag);
            m.buffer.unpack(tempA,localPartitionSize, 1);
...
```

Figura 20 – Código da Divisão da Matriz

Como apresentado na Figura 20, as partes da matriz são enviadas para os nós escravos. Após esta tarefa cada nó efetua os cálculos necessários para a composição total da matriz resultante.

4.3.2 RESULTADOS DOS TESTES

Nesta seção serão mostrados os resultados dos testes da multiplicação de matrizes. Foi efetuado um cálculo de matrizes 1000x1000 e testado o desempenho de forma seqüencial, com 2 *hosts*, 3 *hosts* e 4 *hosts.*

Tabela 3 –Tempo médio de execução em milisegundos do cálculo da multiplicação da matriz

Tarefas	1 *host*	2 *hosts*	3 *hosts*	4 *hosts*
1	21850	x	x	x
4	18000	9698	8878	5366
16	11839	8192	5094	4466
64	15210	8519	6052	5255

Os resultados foram obtidos executando a mesma aplicação diversas vezes, com isso pode-se calcular o tempo médio de execução em milisegundos.

Analisando os resultados apresentados na Tabela 3, pode-se perceber que houve um ganho de desempenho com a utilização da plataforma paralela. É importante ressaltar que com o aumento no número de tarefas ocorreu um aumento no tempo de cálculo, no caso 64 tarefas significa um aumentou significativo acarretando um maior volume de dados a serem processados e trocados entre os nós.

Pode-se perceber que quando adicionado um *host* na plataforma o tempo de processamento da multiplicação cai aproximadamente 56%. Já com 3 *hosts* o tempo não cai

tão drasticamente devido à máquina virtual distribuir 4 processos para 3 nós o que acontece neste caso é que um nó fica com 2 processos.

Desta forma, percebe-se que a utilização de *clusters* realmente proporciona uma alternativa de maior poder computacional e baixo custo, pois conforme adicionados mais nós ao *cluster* o tempo de processamento tende a reduzir até um limite de saturação onde começa a haver uma degradação no desempenho.

Isso pode ser observado na Figura 21:

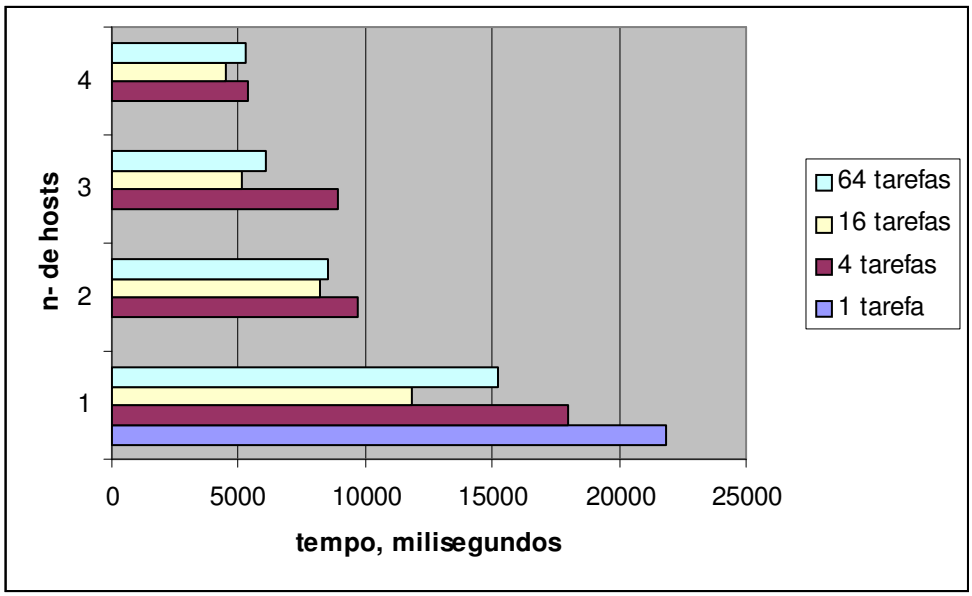


Figura 21 – Tempo médio de resposta do cálculo da multiplicação de matrizes

4.3.2 CÁLCULO INTEGRAL

O cálculo de integral foi implementado para executar no ambiente JPVM. Foram utilizadas duas classes para realização dos cálculos, a classe escravo e a classe integraljpvm. A Figura 22 ilustra o funcionamento da aplicação.

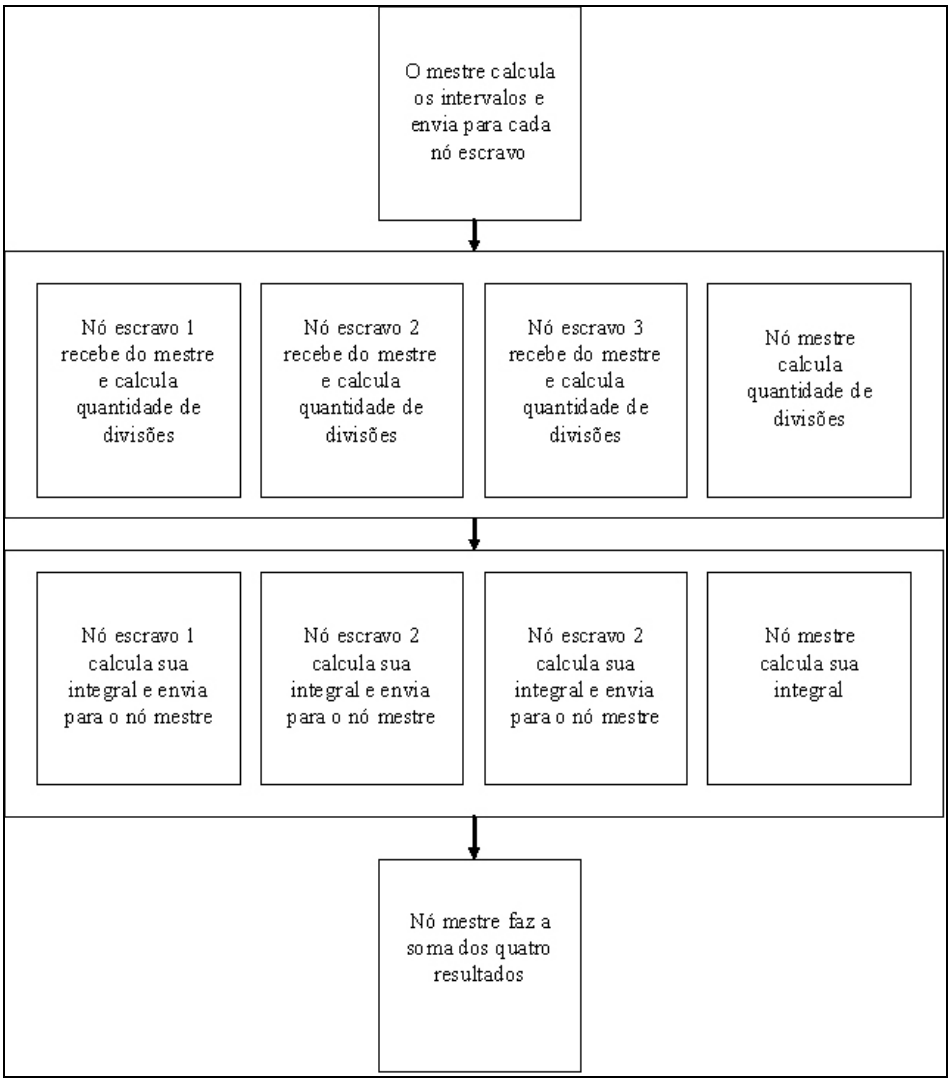
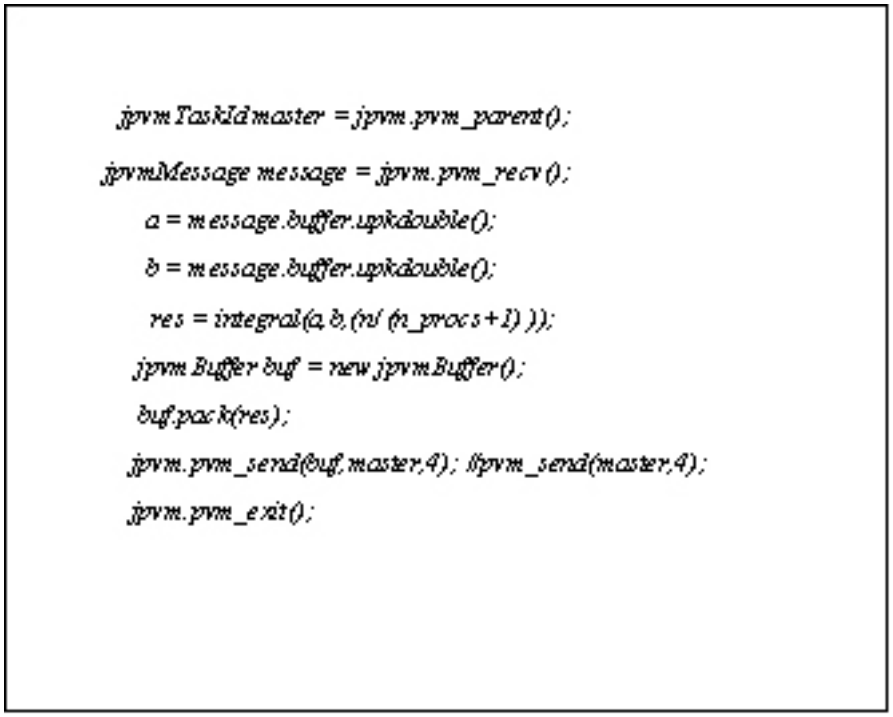


Figura 22 – Algoritmo paralelo da aplicação trapézio composto



```
jpvmTaskId master = jpvm.pvm_parent();
jpvmMessage message = jpvm.pvm_recv();
    a = message.buffer.upkdouble();
    b = message.buffer.upkdouble();
    res = integral(a,b,(n/(n_procs+1)));
jpvmBuffer buf = new jpvmBuffer();
buf.pack(res);
jpvm.pvm_send(buf,master,4); //pvm_send(master,4);
jpvm.pvm_exit();
```

Figura 23 – Classe escravo do cálculo da integral - recebimento

Na Figura 23 observa-se que são desempacotados o limite inferior e o limite superior que são mandados para o método integral onde serão efetuados os cálculos. Depois o resultado é empacotado e enviado para o mestre como o *jpvm.pvm_send()*, e logo depois é feita a finalização do jpvm.

Já na classe integraljpvm tem-se o envio da aplicação “escravo” para todos os nós da máquina virtual utilizando o método *jpvm.pvm_spawn("escravo",NPROCS,tids)*, posteriormente tem-se o envio do limite inferior e superior para todos os nós da máquina virtual e por fim é feito o desempacotamento dos resultados e a soma dos cálculos. A Figura 24 mostra o código desse processo.

```
...
jpvm.pvm_spawn("escravo",NPROCS,tids);
...

for(i=0;i<NPROCS;i++){
    jpvm.Buffer buf = new jpvm.Buffer();
                      buf.pack(limite_inferior);
     buf.pack(limite_superior);
     jpvm.pvm_send(buf,tids[i],0);
     limite_inferior = limite_superior;
     limite_superior = limite_superior + x;
...

for(i=0;i<NPROCS;i++){
    jpvm.Message message =jpvm.pvm_recv();
     valor_retorno = message.buffer.upkdouble();
     sum = sum + valor_retorno;
  }
...
```

Figura 24 – Classe escravo do cálculo da integral – envio.

4.3.2 RESULTADOS DOS TESTES

Os testes foram realizados igualmente aos testes da multiplicação de matrizes. Foram realizados os cálculos de forma seqüencial com 1 *host*, 2 *hosts*,3 *hosts* e por fim 4 *hosts*.

Tabela 4 – Tempos médios de execução em milisegundos do cálculo da integral

Tarefas	1 *host*	2 *hosts*	3 *hosts*	4 *hosts*
1	40220	27915	40204	44914
4	27915	18263	14914	14791
16	40204	21322	14214	10095
64	44914	24678	15043	12560

Como foi visto no cálculo de matrizes o desempenho da aplicação em paralelo apresenta resultados satisfatórios, resultados observados depois das várias execuções da mesma

aplicação. Mais uma vez, o tempo médio de resposta reduz drasticamente ao ser inseridos nós para efetuar o cálculo. Pegando como exemplo o cálculo de 64 tarefas nota-se que ao se inserir um nó o tempo cai aproximadamente 45%, já com 4 *hosts* o tempo em relação ao seqüencial é 72% menor.

A Figura 25 a seguir mostra o que desempenho da aplicação executada em paralelo é melhor que o desempenho da mesma aplicação executada sequencialmente.

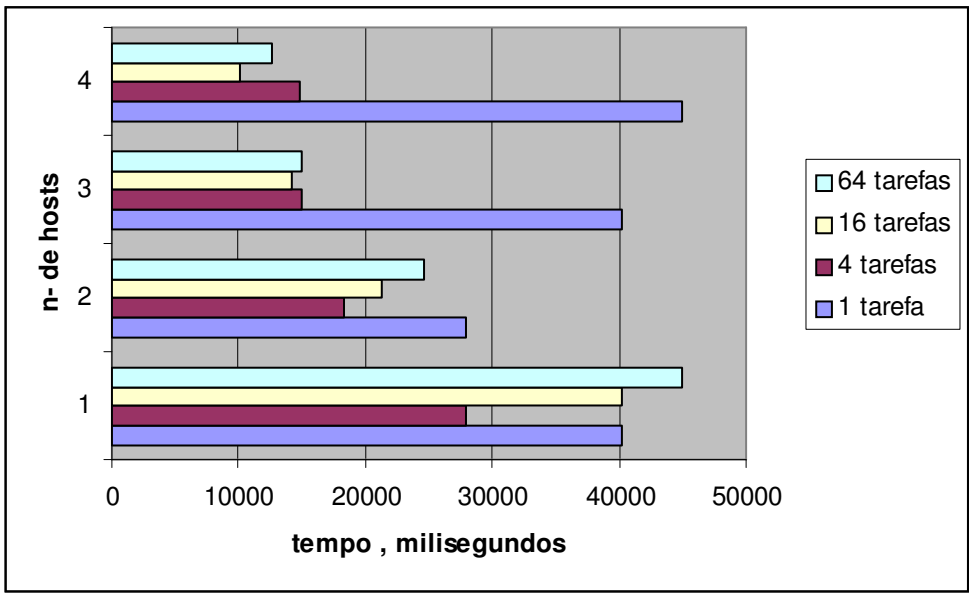


Figura 25 – Tempo médio de resposta do cálculo do trapézio composto

4. 2 CONSIDERAÇÕES FINAIS

Para a realização dos testes foi utilizado um *cluster* homogêneo, ou seja, uma plataforma com 4 máquinas todas com o mesmo poder computacional. Para a comunicação dos mesmos foi utilizada a biblioteca de troca de mensagens JPVM e a partir dos resultados foi feita uma análise comparativa entre o processamento seqüencial e o processamento paralelo.

Como mostraram os testes práticos houve uma melhora no tempo médio de resposta das aplicações em paralelo em relação à mesma executada seqüencialmente. Essa melhora no tempo de resposta é observada devido à execução real das aplicações do cálculo da multiplicação de matrizes e da integral.

5. CONCLUSÃO

A arquitetura na qual foi implementado este trabalho serviu para demonstrar que a computação paralela distribuída não só pode ser usada como apresenta, de modo geral, resultados satisfatórios.

O objetivo principal deste trabalho era a análise e descrição do JPVM, seu funcionamento, sua portabilidade e interoperabilidade. Foi observado que o JPVM veio reforçar algumas características do PVM padrão, uma delas a própria portabilidade e interoperabilidade.

O JPVM em relação ao PVM padrão apresenta algumas vantagens como criação de *threads* e independência de plataforma, já que o JPVM pode ser instalado e executado separadamente, sem a necessidade do PVM. Outra vantagem observada é que seus métodos podem ser modificados e com isso podem-se efetuar melhorias e modificações de acordo com a necessidade do usuário.

Uma observação importante a ser feita, é que no caso do JPVM as mensagens são trocadas através do protocolo TCP, diferentemente do PVM em que se pode escolher entre os protocolos TCP e UDP. É importante salientar estas características pois para este tipo de aplicação efetuada neste projeto, o UDP se torna mais apropriado, pois no UDP não possui nenhum controle de segurança o que torna mais rápido a comunicação das mensagens, e velocidade que neste caso é importante.

Apesar das vantagens apresentadas o JPVM apresentou um console muito resumido o que o enfraquece em relação ao controle de nós, apresentando poucos comandos ao usuário.

A construção de máquinas paralelas virtuais utilizando o JPVM é bastante simples, além de rodar em várias arquiteturas o JPVM é um pacote pequeno podendo ser obtido facilmente.

Os resultados obtidos neste trabalho foram satisfatórios demonstrando que se pode utilizar a biblioteca JPVM para aplicações que exigem um alto desempenho. Pode-se realizar outros tipo de aplicações, não só aplicações matemáticas mas também processamento de imagens e outras aplicações que fazem uso de um poder computacional maior provido. Esse poder computacional é provido pelo processamento paralelo que permite um ganho em tempo médio de execução das aplicações em paralelo quando comparada a seqüencial.

Como trabalhos futuros pode-se efetuar um comparativo científico em relação ao uso do PVM e do JPVM em um mesmo *cluster*. Outro trabalho bastante interessante constitui em efetuar uma análise comparativa, tanto em um *cluster* homogêneo quanto em um *cluster* heterogêneo, das bibliotecas PVM e JPVM efetuando não somente um comparativo entre os tempos médios de execução de aplicações em JPVM e PVM, mas também um comparativo quanto aos protocolos utilizados por essas bibliotecas (TCP e UDP).

REFERÊNCIAS BIBLIOGRÁFICAS

(ALMASI,1994) Almasi, G. S, Gottlieb A., ***High Parallel Computing***, 2a ed., The Benjamin Communings Publishing Company, Inc., 1994.

(COLOURIS,1994) COLOURIS, G., Dolimore, J., Kindberg, T., ***Distributed Systems Concepts and Design***, 2a ed, Addison Wesley Publishing Company, s.1, 1994.

(DEPERNI,2002) DEPERNI. Paulo H. R,.**Um Estudo sobre Métodos Iterativos na Solução de Sistemas de Equações Provenientes do Método dos Elementos de Contorno**. Disponível em http://www.coc.ufrj.br/teses/doutorado/estruturas/2002/teses/DEPERNI_PHR_02_t_D_est.pdf acessado em 10 de maio de 2006.

(DUNCAN,1990) DUNCAN, R., **A Survey of Parallel Computer Architectures**, *IEEE Computer*, pp.5-16, Fevereiro, 1990.

(FERRARI,1997) Ferrari, Adam J. JPVM: Network Parallel Computing in

Java. Disponível em http://www. cs. virginia.edu/jpvm/doc/jpvm-97-29.pdf acesso em 9 de maio de 2006.

(FLYNN,1996) FLYNN, Michael J.; RUDD, Kevin W., Parallel Architectures, ACM Computing Surveys, v.28, pp 67-70, Março, 1996.

(HWANG,1984) HWANG, K., Briggs, F. A., Computer Architecture and Parallel Processing, McGraw-Hill International Editions, 1984.

(JAQUIE,1999) JAQUES, Kalinka R.L., Extensão da ferramenta de apoio à programação paralela (F.A .A .P) para ambientes virtuais, Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo- ICMC/USP, como parte dos requisitos para a obtenção do título de Mestre em Ciências – Área de Ciências de Computação e Matemática Computacional, 1999.

(LINHALIS,98) LINHALIS, Flávia; FIATS, Mayb I.. Disponível em http://black.rc.unesp.br/gpacp/bibliografia/D-01 - PVM e MPI - ICMC - USP - São Carlos.doc acesso em 20 de maio de 2006.

(LOURENÇO, 2004) Lourenço, João. Introdução ao PVM. Disponível em http://www-asc. di. fct. unl.pt/~jml/LEI/SCPD/Docs/Introducao_ao_PVM. pdf acesso em maio de 2006.

(NAVAUX,1989) NAVAUX, P.O.A, Introdução ao processamento paralelo, Revista Brasileira de Computação, v.5, n-2,pp 33-43, Outubro, 1989.

(OLIVEIRA,2005) OLIVEIRA, Daniel C. Implementação de Aplicações Paralelas com o pacote JPVM. Disponivel em http://www.ulbra-to.br/ensino/43020/artigos/relatorios2005-1/Arquivos/Daniel C O - Trabalho de Conclusao de Curso.pdf acesso em 8 de maio de 2006.

(SANTANA et al,1997) SANTANA, Regina H.Carlucci; SANTANA, Marcos J.; SOUZA, Márcio Augusto; SOUZA, Paulo Sérgio L.; PIEKARSKI, Ana Elisa T., Disponivel na internet em http://black.rc.unesp.br/gpacp/Bibliografia/A-01 - Computação Paralela - ICMC - USP - São Carlos.doc acesso em 10 de fevereiro de 2006.

(SOUZA et al, 2006) SOUZA, Sérgio H.G; SHIOZER, Denis J.; MONTICELLI, Alcir J.; Uma Intodução ao PVM – Como a paralelização pode ajudar a resolver problemas complexos de otimização. Disponível em http://black.rc.unesp.br/gpacp/bibliografia/C-02 - Introdução ao PVM - FEM - UNICAMP.pdf acesso em 15 de maio de 2006.

(SUNDERAM,1994) SUNDERAM, V. S., Geist, A., Dongarra, J., Manchek, R., *The PVM concurrent computing system:evolution and trends*, Parallel Computing, v. 20, pp. 531-545,1994

(TANEUMBAUM,1995) Tanenbaum, Andrew S. Sistemas Operacionais Modernos.Editora Prentice-Hall do Brasil Ltda. Rio de Janeiro, 1995.

APENDICE A – Código fonte integraljpvm.java.

```
import jpvm.*;
import java.lang.*;
import java.util.*;
class integraljpvm {
static int N = 120000000;  /*quantidade de divisoes*/
/* a quantidade de divisoes deve ser um multiplo de (NPROCS+1), pois,
   serao NPROCS escravos mais o mestre. */
static int NPROCS = 8;    /*quantidade de escravos*/
static double   s = 0,sum = 0;
static double a,b;
static double limite_inferior,limite_superior;
static double x;
static double resultado,res,valor_retorno;
static int info, infos;
public  static  double  integral(double  a,  double  b,  int  n)  throws
jpvmException
{
     double h,var,r,s = 0, aux;
   int i;                // numero de termos
   h = (double)(b-a)/n;
   s = s + Math.sin(a) + Math.exp(a);
   s = s + Math.sin(b) + Math.exp(b);
   aux = a;
   for(i=1;i<n-1;i++){
      var = aux + h;
      r = 2 * (Math.sin(var) + Math.exp(var));
      s = s + r;
      aux = var;
   }
   return((h/2) * s);
}
public static double msecond() {
            Date d = new Date();
            double msec = (double) d.getTime();
            return msec;
      }

public static void main(String args[])
{
 try {
int cc;              /* Numero de processos criados */
int tid;             /* Identificacao do processo escravo */
int i;               /* Contador */
float timef;         /* Tempo para a execucao da integral */
double mmstart,start,end,tempo_final;
  jpvmEnvironment jpvm = new jpvmEnvironment();
jpvmTaskId mytid = jpvm.pvm_mytid();
  System.out.println("Processo mestre: "+ mytid.toString());
    jpvmTaskId tids[] = new jpvmTaskId[NPROCS];
   jpvm.pvm_spawn("escravo",NPROCS,tids);
  System.out.println("Iniciando calculos...\n");
  start = msecond();
   b = 1.2;
   x = (double)(b - a)/(NPROCS+1); //numtasks;
    limite_inferior = a;
   limite_superior = a + x;
  for(i=0;i<NPROCS;i++){
   jpvmBuffer buf = new jpvmBuffer();
```

```
   buf.pack(limite_inferior);
    buf.pack(limite_superior);
     jpvm.pvm_send(buf,tids[i],0);
      limite_inferior = limite_superior;
      limite_superior = limite_superior + x;
   }
mmstart=msecond();
   res       =       integral(limite_inferior,limite_superior,N/(NPROCS+1));
//numtasks);
   for(i=0;i<NPROCS;i++){
     jpvmMessage message = jpvm.pvm_recv();
      valor_retorno = message.buffer.upkdouble();
      sum = sum + valor_retorno;
   }

   res = res + sum;
  end = msecond();
System.out.println("Total time: "+(end-start)+
                        " (mult: "+(end-mmstart)+")");

  jpvm.pvm_exit();// pvm_exit ();
}
catch (jpvmException jpe) {
            System.out.println("Error - jpvm exception");
          }
}
}
```

APENDICE B – Código fonte escravo.java.

```
import jpvm.*;
import java.lang.*;
class escravo {
      static int n_procs = 8;
      static int n = 120000000;
 public   static   double   integral(double   a,double   b,int   n)   throws
jpvmException
{
   double h,vari,r,s = 0, aux;
   int i;               // numero de termos
   h = (double)(b-a)/n;
   s = s + Math.sin(a) + Math.exp(a);
   s = s + Math.sin(b) + Math.exp(b);
   aux = a;
   for(i=1;i<n-1;i++){
      vari = aux + h;
      r = 2 * (Math.sin(vari) + Math.exp(vari));
      s = s + r;
      aux = vari;
   }
   return((h/2) * s);
}


public static void main(String args[])
{
 try {
   double a,b,res;
   jpvmEnvironment jpvm = new jpvmEnvironment();
      jpvmTaskId mytid = jpvm.pvm_mytid(); // mytid = pvm_mytid ();
      System.out.println("Task Id: "+mytid.toString());
            // Spawn some workers...
            jpvmTaskId tids[] = new jpvmTaskId[n_procs];
            jpvmTaskId master = jpvm.pvm_parent(); // master = pvm_parent
();
      jpvmMessage message = jpvm.pvm_recv(); // pvm_recv(-1,-1);
      a = message.buffer.upkdouble();//pvm_upkdouble(&a,1,1);
      b = message.buffer.upkdouble();  //pvm_upkdouble(&b,1,1);
      res = integral(a,b,(n/ (n_procs+1) ));
   jpvmBuffer buf = new jpvmBuffer();
   buf.pack(res);
   jpvm.pvm_send(buf,master,4);
   jpvm.pvm_exit();

   }
          catch (jpvmException jpe) {
            System.out.println("Error - jpvm exception");
          }
}}
```

ANEXO A – Código fonte mat_mult.java.

```
/* mat_mult.java
 *
 * A simple parallel matrix-matrix multiply example using jpvm.
 *
 * Adam J Ferrari
 * Tue Jun  4 10:31:02 EDT 1996
 *
 * Copyright (C) 1996  Adam J Ferrari
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Library General Public
 * License as published by the Free Software Foundation; either
 * version 2 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Library General Public License for more details.
 *
 * You should have received a copy of the GNU Library General Public
 * License along with this library; if not, write to the
 * Free Software Foundation, Inc., 675 Mass Ave, Cambridge,
 * MA 02139, USA.
 */

import jpvm.*;
import java.io.*;
import java.util.*;

class mat_mult {
      static final int ParamTag     = 11;
      static final int DoneTag      = 22;
      static final int PipeTag      = 33;
      static final int RollTag      = 44;
      static int numTasks           = 0;
      static int matDim             = 0;
      static int taskMeshDim        = 0;
      static int localPartitionDim  = 0;
      static int localPartitionSize = 0;
      static int taskMeshRow        = 0;
      static int taskMeshCol        = 0;
      static int localTaskIndex     = 0;
      static boolean debug          = false;
      static jpvmTaskId myTaskId    = null;
      static jpvmTaskId masterTaskId  = null;
      static jpvmEnvironment jpvm     = null;
      static jpvmTaskId tids[];
      static float C[], A[], B[], tempA[];

      public static void main(String args[]) {
            double mmstart = 0.0;
            double start   = 0.0;
            double end     = 0.0;
            boolean cmd = false;

            try {
                  jpvm = new jpvmEnvironment();
```

```
        myTaskId = jpvm.pvm_mytid();
        masterTaskId = jpvm.pvm_parent();
        if(masterTaskId==jpvm.PvmNoParent) {
            if (args.length != 2) usage();
            cmd = true;
            try {
              numTasks = Integer.parseInt(args[0]);
              matDim = Integer.parseInt(args[1]);
            }
            catch (NumberFormatException e) {
              usage();
            }
            System.out.println(""+matDim+"x"+matDim+
              " matrix multiply, "+numTasks+" tasks");

            start = msecond();
            taskMeshDim = getTaskMeshDim(numTasks);

            tids = new jpvmTaskId[numTasks];
            if(numTasks>1)
                jpvm.pvm_spawn("mat_mult",numTasks-1,tids);
            tids[numTasks-1] = tids[0];
            tids[0] = myTaskId;
            localTaskIndex = 0;

            /* Broadcast parameters to all tasks */
            mmstart = msecond();
            jpvmBuffer buf = new jpvmBuffer();
            buf.pack(taskMeshDim);
            buf.pack(numTasks);
            buf.pack(tids,numTasks,1);
            buf.pack(matDim);
            jpvm.pvm_mcast(buf,tids,numTasks,ParamTag);
        }
        else {
            // Normal worker task - get parameters...
            jpvmMessage m = jpvm.pvm_recv(ParamTag);
            taskMeshDim = m.buffer.upkint();
            numTasks = m.buffer.upkint();
            tids = new jpvmTaskId[numTasks];
            m.buffer.unpack(tids,numTasks,1);
            for (localTaskIndex=0;localTaskIndex<numTasks;
              localTaskIndex++)
              if (myTaskId.equals(tids[localTaskIndex]))
                    break;
            matDim = m.buffer.upkint();
        }

        /* Do the matrix multiplication */
        matmul();

        if(cmd) {
            end = msecond();
            System.out.println("Total time: "+(end-start)+
              " (mult: "+(end-mmstart)+")");
        }
        jpvm.pvm_exit();
    }
    catch (jpvmException jpe) {
        error("jpvm Exception - "+jpe.toString(),true);
    }
```

```
}

public static void matmul() throws jpvmException {
      int i,j,k;

      localPartitionDim = matDim/taskMeshDim;
      localPartitionSize = localPartitionDim*localPartitionDim;
      taskMeshRow = localTaskIndex/taskMeshDim;
      taskMeshCol = localTaskIndex%taskMeshDim;
      A = new float[localPartitionSize];
      B = new float[localPartitionSize];
      C = new float[localPartitionSize];
      tempA = new float[localPartitionSize];

      if(debug) {
          System.out.println("localTaskIndex\t= "+localTaskIndex);
          System.out.println("matDim\t= "+matDim);
          System.out.println("taskMeshDim\t= "+taskMeshDim);
          System.out.println("localPartitionDim\t= "+
            localPartitionDim);
          System.out.println("localPartitionSize\t= "+
            localPartitionSize);
          System.out.println("taskMeshRow\t= "+taskMeshRow);
          System.out.println("taskMeshCol\t= "+taskMeshCol);
      }

      for (i=0;i<localPartitionSize;i++) {
            C[i] = (float) 0.0;
            A[i] = (float)(i+localTaskIndex*(taskMeshRow+1));
            B[i] = (float)(i-localTaskIndex*(taskMeshRow+1));
      }

      for (i=0;i<taskMeshDim;i++) {
            Pipe(i);
            Multiply();
                  if(i<(taskMeshDim-1)) Roll();
      }
}

public static void Pipe(int iter) throws jpvmException {
      int i;
      if (taskMeshCol == (taskMeshRow+iter)%taskMeshDim) {
            jpvmBuffer buf = new jpvmBuffer();
            buf.pack(A,localPartitionSize,1);
            for (i=0;i<taskMeshDim;i++)
                if (localTaskIndex != taskMeshRow*taskMeshDim+i) {
                  jpvm.pvm_send(buf,
                      tids[taskMeshRow*taskMeshDim+i],PipeTag);
                }
      }
      else {
            jpvmMessage m = jpvm.pvm_recv(PipeTag);
            m.buffer.unpack(tempA,localPartitionSize,1);
      }
}

public static void Multiply() throws jpvmException {
      int i,j,k;
      float temp;
      for (i = 0; i < localPartitionDim; i ++)
            for (j = 0; j < localPartitionDim; j++) {
```

```
                        temp = 0;
                        for (k = 0; k < localPartitionDim; k++)
                              temp += A[i*localPartitionDim+k] *
                                    B[k*localPartitionDim+j];
                        C[i*localPartitionDim+j] += temp;
                  }
      }

      public static void Roll() throws jpvmException {
            int who = ( (taskMeshRow!=0) ?
                  (taskMeshRow-1)*taskMeshDim+taskMeshCol:
                  (taskMeshDim-1)*taskMeshDim+taskMeshCol);
            jpvmBuffer buf = new jpvmBuffer();
            buf.pack(B,localPartitionSize,1);
            jpvm.pvm_send(buf,tids[who],RollTag);
            jpvmMessage m = jpvm.pvm_recv(RollTag);
            m.buffer.unpack(B,localPartitionSize,1);
      }

      public static double msecond() {
            Date d = new Date();
            double msec = (double) d.getTime();
            return msec;
      }

      public static void error(String message, boolean die) {
            System.err.println("mat_mult: "+message);
            if(die) {
                  if(jpvm!=null) {
                        try {
                              jpvm.pvm_exit();
                        }
                        catch (jpvmException jpe) {
                        }
                        System.exit(1);
                  }
            }
      }

      public static void usage() {
            error("usage -  java mat_mult <tasks> <mat dim>", true);
      }

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