# VSObjects: Middleware para Gerenciamento de Objetos Virtualmente Compartilhados

Christiane V. Pousa, Dulcinéia O. da Penha, Carlos A. P. S. Martins
*Instituto de Informática/ Programa de Pós-Graduação em Engenharia Elétrica*
*Laboratório de Sistemas Digitais e Computacionais*
*Pontifícia Universidade Católica de Minas Gerais*
*Av. Dom José Gaspar 500, 30535-610, Belo Horizonte, MG, Brazil*
*Telefone / Fax: 55-31-33194305*

*christ.bh@terra.com.br, dulcineia@pucmg.br, capsm@pucminas.br*

## Resumo

*Este trabalho apresenta um software que suporta a construção de aplicações em Java baseadas no modelo de programação com variáveis compartilhadas executando em ambientes distribuídos. VSObjects é um VSM (Virtual Shared Memory) baseado em objetos, desenvolvido em Java. O nosso principal objetivo é propor, implementar e verificar o VSObjects, analisar o uso e o desempenho do VSObjects através da convolução de imagem paralela comparando com a versão seqüencial da convolução e a implementação usando somente RMI(Remote Method Invocation). A principal contribuição deste trabalho é a implementação do VSObjects. Outras contribuições são as implementações da convolução de imagens no modelo distribuído e no modelo seqüencial, e a análise dos resultados obtidos com as diferentes implementações.*

*Palavras-chaves: VSM, Arquiteturas Distribuídas, Programação com Variáveis Compartilhadas.*

## 1. Introdução

Atualmente, muitas aplicações demandam uma grande quantidade de recursos computacionais devido à necessidade de obtenção de tempos de resposta cada vez menores [1]. Uma solução muito utilizada neste caso é o uso de sistemas computacionais de alto desempenho. Sistemas de arquitetura paralela, como multiprocessadores e aglomerados de computadores, são exemplos de sistemas de alto desempenho. Atualmente, aglomerados vêm sendo muito utilizados, devido, principalmente, às suas vantagens como escalabilidade [2] e baixo custo.

Para se alcançar uma utilização efetiva de arquiteturas paralelas, torna-se imperativo a manutenção de um ambiente que permita a execução eficiente de diversas aplicações. No entanto, isto deve ser alcançado sem impor aos desenvolvedores de *software* paralelos empecilhos excessivos à produção de tais programas [3]. No caso, de arquiteturas paralelas com memória distribuída, como aglomerados, a dificuldade é ainda maior devido à necessidade do gerenciamento de comunicação entre os nodos da arquitetura. Existem algumas formas de prover a transparência de gerenciamento de paralelismo e comunicação desejada ao usuário e/ou programador. O conceito de imagem de sistema único é utilizado para definir estas formas de prover transparência. DSM (Distributed Shared Memory) [4][5][6] é um tipo de imagem de sistema único que provê transparência do gerenciamento dos recursos de memória de um aglomerado (arquitetura distribuída).

O modelo de programação com variáveis compartilhadas é mais fácil de ser utilizado do que o modelo de passagem de mensagem. Neste ultimo, o programador precisa preocupar-se com comunicação de dados entre os processos (nos diferentes nodos). Além disso, em programação com variáveis compartilhadas, estruturas de dados grandes ou complexas podem ser facilmente comunicadas sem empacotamento. Por outro lado, as arquiteturas com memória distribuída têm as vantagens do baixo custo e escalabilidade.

Esta pesquisa e a conseqüente proposta e implementação do VSObjects foi motivada pelo fato de que a linguagem Java não possui mecanismos explícitos de compartilhamentos de informações entre processos distribuídos executados paralelamente.

Desta forma, as vantagens e facilidades destes dois modelos foram unidas estendendo-se o paradigma de programação com variável compartilhada, normalmente utilizado em sistemas multiprocessadores, para arquiteturas distribuídas, através da construção de sistemas DSM e/ou VSM, que podem ser construídos na camada do hardware (DSM), do sistema operacional ou de aplicação (DSM, VSM).

DSM e VSM podem ser definidos como modelos que aplicam métodos de programação usando variáveis compartilhadas (compartilhamento lógico de dados) sobre arquiteturas de memória distribuída (compartilhamento físico de dados). Ou seja, é uma abstração de

compartilhamento de informação entre processos em sistemas que não compartilham memória fisicamente [7].

Segundo [8], a grande diferença entre um DSM e um VSM é que os VSMs são implementados através de um software e os DSM podem ser implementados tanto em software quanto em hardware.

O VSM reside fisicamente em uma única memória ou pode ser distribuído entre os vários processadores que participam da execução de um programa [8]. O principal objetivo do VSM é criar um espaço virtual de memória compartilhada, geralmente criada através de um software.

O VSObjects é um middleware, que dá suporte à construção de aplicações desenvolvidas em Java, baseadas no modelo de programação com variáveis compartilhadas em ambientes distribuídos (gerencia a memória dos nodos do aglomerado provendo abstração de uma memória virtualmente compartilhada). O software faz a gerência da memória compartilhada entre os nodos do aglomerado que estão fisicamente distribuídos.

Nosso principal **objetivo** é propor, implementar e verificar o VSObjects, analisar o uso e o desempenho do VSObjects através da convolução de imagem paralela comparando com a versão seqüencial da convolução e a implementação usando somente RMI.

Escolhemos a operação de convolução de imagem para realizar nossos testes porque é uma das mais importantes operações de processamento de imagem digital (PID) e representa um bom exemplo de aplicações que demandam uma grande quantidade de recursos computacionais para ser executada. Imagens são compostas por uma grande quantidade de dados e freqüentemente são armazenadas em matrizes e o custo computacional para manipulá-las é alto [9]. Além disso, algumas operações PID têm natureza paralela [10] porque realizam ações independentes sobre dados independentes (pixels da imagem, que são geralmente os elementos que compõem a matriz e representação da imagem) [11].

O VSObjects provê um ambiente para programação paralela oferecendo a ilusão de uma memória compartilhada entre os nodos da rede. Para isso, o VSObjects, gerencia automaticamente a coerência dos dados que estão em movimento, não exigindo que os programadores se preocupem explicitamente com a movimentação dos dados entre os programas que estão sendo executados nos nodos. Este artigo vem apresentar o VSObjects, enfocando a sua arquitetura e implementação.

O que distingue o VSObjects de um DSM, é que ele é um ambiente dedicado para aplicações distribuídas e paralelas desenvolvidas em Java. Além disso, a princípio, o objeto compartilhado está centralizado na máquina que executa o ambiente de execução do VSObjects.

Nas próximas seções apresentamos alguns conceitos básicos, o VSObjects, suas características principais e realizamos uma comparação entre programação com VSM, seqüencial e seqüencial distribuída, e ainda, discutimos os resultados obtidos com as três versões de programação citadas.

## 2. Formas de Compartilhamento de Informação em Programas Paralelos

### 2.1 Compartilhamento entre Processos

Em arquiteturas paralelas de memória compartilhada, variáveis compartilhadas podem ser utilizadas para comunicação e sincronização entre os processos da aplicação. A utilização efetiva de sistemas paralelos é uma tarefa difícil, pois envolve o projeto de aplicações paralelas corretas e eficientes. Isto engloba vários problemas complexos como sincronização de processos, coerência de dados e ordenamento de eventos.

O ideal para a utilização de sistemas paralelos é retirar do usuário (programador) a tarefa de gerenciar a execução paralela dos processos. Existem algumas formas de utilização do paralelismo que fornecem um certo nível de transparência ao programador. Uma das formas de se fornecer essa transparência ao usuário é através do uso de sistemas operacionais multiprocessados aplicando paralelismo SMP *(Symmetric Multiprocessors)* [12].

Do ponto de vista de softwares, muitas técnicas são conhecidas e utilizadas para programação de multiprocessadores [13]. Para comunicação, um processador escreve dados na memória, para serem lidos por todos os outros processadores. Para sincronização, podem ser usadas sessões críticas, implementando semáforos ou monitores para prover a exclusão mútua [14] necessária.

Os sistemas operacionais Unix [15] [16], Windows NT, Windows 2000 [17], entre outros, suportam sistemas multiprocessados. Nestes sistemas, a execução paralela é ativada pela geração de múltiplas threads que são processadas paralelamente. O número de threads, em princípio, é independente do número de processadores [18].

### 2.2 Compartilhamento Distribuído

A existência de um espaço único de endereçamento melhora a programabilidade em máquinas paralelas. Na programação paralela de memória distribuída, como aglomerados, o particionamento de dados e o acesso dinâmico são mais complexos do que em sistemas multiprocessados com memória compartilhada [18].

Quando a programação paralela é feita para aglomerados que possuem os nodos de processamento com memória fisicamente distribuída, é necessário que se utilize algum modelo de programação que permita a troca de informações entre os nodos que compõem o aglomerado; criando um espaço único de endereçamento ou realizando a troca de mensagens entre os nodos [19].

Os modelos de passagem de mensagem e memória compartilhada distribuída são os responsáveis, na programação paralela para aglomerados, por fazer a troca de informações entre os nodos de processamento.

Na programação com passagem de mensagem não existe a ilusão de um espaço de endereçamento único entre os nodos do aglomerado, a troca de mensagens é explicita e o programador deve se preocupar com o acesso as informações.

No modelo de memória compartilhada distribuída, o acesso às informações que movimentam entre os nodos de processamento do aglomerado é implícito. Ou seja, o programador acessa as informações remotas de outro nodo como se estivesse acessando variáveis locais ao seu programa. Neste modelo o programador não tem que se preocupar com a coerência e a consistência das informações; isto geralmente é garantido por protocolos ou por hardwares específicos [19]. Como exemplos de sistemas que provêem este modelo de programação podem ser citados DSM (Distributed Shared Memory) e VSM (Virtual Shared Memory).

Em sistemas como DSM e VSM é necessário que existam políticas que façam a consistência e a coerência das informações que são compartilhadas. Estas políticas podem ser implementadas em software e hardware e normalmente trabalham diretamente com as caches dos processadores do aglomerado [20].

Os modelos de consistência de memória são responsáveis por indicar quando uma escrita estará visível para os outros nodos [19][ 20]. Existem basicamente dois tipos de consistência de dados, a consistência seqüencial e a consistência Relaxada (*Relaxed*).

- Consistência seqüencial: todas as operações são realizadas de acordo com uma ordem, e nenhuma operação é iniciada antes do término da operação anterior.
- Consistência relaxada: Este modelo de consistência garante que as operações de leitura e de escrita sejam afrouxadas, permitindo que elas sejam realizadas sem que exista dependência, melhorando o desempenho do modelo. Existem quatro modelos de consistência relaxada: Consistência de processador, Ordem parcial de armazenamento, Consistência fraca e Consistência solta.

A consistência de processador permite que uma operação de leitura seja iniciada antes que a operação de escrita tenha sido finalizada. Ordem parcial de armazenamento permite que sejam realizadas escritas bufferizadas, ou seja, uma operação de escrita pode ser iniciada sem que outra operação de escrita tenha sido finalizada. A consistência fraca elimina as ordens que existem na realização das operações. Apenas as ordens que são importantes são garantidas, através de primitivas de sincronização.E a consistência solta, onde o segundo passo a ser tomado depois da realização do modelo de consistência fraca, é este modelo que garante a ordem as primitivas síncronas e das operações de leitura e escrita [19][ 20].

Entre os diferentes protocolos existentes para manter a coerência das informações podemos citar: protocolo múltiplos-leitores/único-escritor, e múltiplos-leitores/múltiplos-escritores. De acordo com o primeiro protoloco somente um processador escreve em um determinado dado compartilhado e os demais processadores só podem ler este dado. No segundo protocolo todos os processadores podem escrever e ler os dados compartilhados. Para manter a coerência das informações algumas políticas de invalidação de informações são utilizadas[7].

## 3. VSObjects

VSObjects, foi desenvolvido em Java, utilizando as bibliotecas de Thread e RMI (Remote Method Invocation) da linguagem Java. Esta linguagem foi escolhida por oferecer independência de plataforma (portabilidade) e uma API para documentação da implementação do VSObjects. A seguir apresentaremos a arquitetura, a implementação e o modelo de coerência do VSObjects.

### 3.1 Arquitetura

RMI é uma das tecnologias fornecidas por Java para prover as funcionalidades de uma plataforma de objetos distribuídos. Através da utilização da arquitetura RMI [21], é possível que um objeto ativo em uma máquina virtual Java possa interagir com objetos de outras máquinas virtuais Java (JVM), independentemente da localização dessas máquinas virtuais [22].

Um dos benefícios da linguagem Java é a suporte que a plataforma provê para programação multithreading como parte da linguagem. Em Java, cada thread executada em uma JVM é associada com um objeto da classe Thread. Quando uma JVM é iniciada para executar uma aplicação, uma única thread principal é criada, embora diversas outras threads possam ser iniciadas simultaneamente [22].

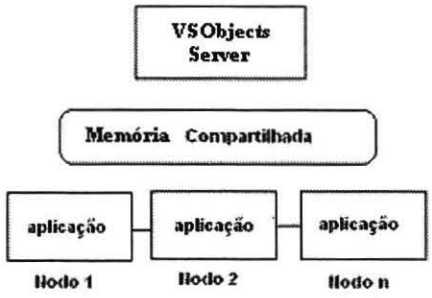



**Figura 1 – Arquitetura do VSObjects**

A finalidade do ambiente VSObjects é permitir que uma aplicação paralela em Java execute em um sistema de memória distribuída como se estivesse executando em uma máquina de memória compartilhada. Ele combina as tecnologias RMI e Thread, criando assim um espaço virtual de memória compartilhada, que permite a criação de programas paralelos em Java que sejam executados em máquinas com memória fisicamente distribuída e que

compartilhem virtualmente suas memórias umas com as outras. A figura 1 mostra a arquitetura do VSObjects.

## 3.2 Implementação

Para implementar o ambiente do VSObjects, foi desenvolvido um ambiente de execução (*run-time*) baseado em RMI e uma biblioteca baseada em MultiThread, ambas tecnologias providas pela linguagem Java.

O ambiente de execução desenvolvido é responsável por manter a transparência de acesso entre os objetos da memória virtualmente compartilhada e a coerência das informações compartilhadas entre os nodos de processamento. A figura 2 apresenta a interface gráfica do ambiente de execução do VSObjects, que roda no servidor (VSObjects server).

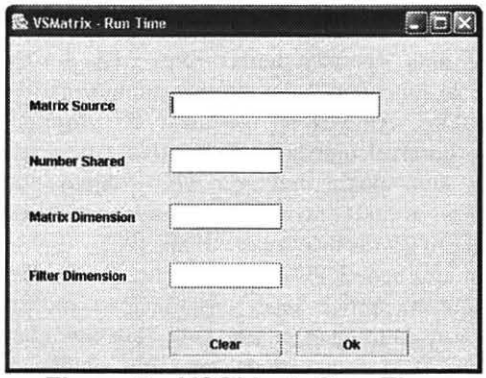



**Figura 2 – VSObjects Run-Time**

Além do ambiente de execução o VSObjects utiliza uma biblioteca para criação do objeto que representa a memória virtualmente compartilhada em cada nodo de processamento do aglomerado. Esta biblioteca contém as definições do objeto Shared_Object e roda em todas as máquinas do aglomerado que executam partes paralelas da aplicação. O objeto Shared_Object é a informação compartilhada pelos nodos durante a execução das aplicações. O conjunto de objetos Shared_Object compõe a memória virtualmente compartilhada no VSObjects.

Para desenvolver as aplicações paralelas, usando o VSObjects, o programador deverá utilizar o objeto Shared_Object. É importante ressaltar que os programas paralelos que utilizam o VSObjects precisam ser desenvolvidos em Java. Para executar as aplicações no ambiente VSObjects, é necessário rodar o VSObjects em uma máquina.

### 3.3 Coerência de Memória

O sistema cria réplicas dos dados durante a execução e o ambiente de execução do VSObjects é o responsável por manter a coerência destas réplicas nos vários nodos do aglomerado que executam a aplicação.

A coerência de dados compartilhados no VSObjects é implementada através da política de coerência múltiplos-leitores/único-escritor, esta política faz com que o sistema tenha um compartilhamento da forma *múltiplos-leitores/único-escritor*. Ou seja, ao mesmo tempo, uma informação pode ser acessada em modo somente-leitura por um ou mais processos, ou ele pode ser lido e escrito por um único processo. Um objeto que está sendo acessado atualmente está no modo somente-leitura e pode ser replicado indefinidamente para outros processos. Quando um processo tenta realizar alguma operação de escrita neste objeto (que está em modo somente-leitura) uma mensagem *multicast* é primeiramente enviada para todas as outras cópias para invalidá-las e esta mensagem é respondida antes da operação de escrita ser realizada naquela posição (do objeto em questão). Esse procedimento faz com que os outros processos não leiam dados antigos (isto é, dados que não estão atualizados). O processo que requisitou a operação de escrita passa a ser o escritor daquele objeto. Qualquer processo é bloqueado quando tentar acessar um objeto que está sendo escrito (ou que pertence a um processo escritor). Eventualmente, o controle é transferido do processo escritor, e outros acessos podem ocorrer uma vez que a atualização foi enviada.

No VSObjects quando um nodo deseja fazer a leitura de um dado que não está na sua memória local, o servidor obtém uma cópia deste objeto sem que os outros nodos do aglomerado tenham que ser avisados. Se mais de um nodo requisitar ao VSObjects server uma cópia do dado que ele deseja ler, ocorrerá exclusão mútua, provida pela implementação de mecanismos de semáforo. As Threads em Java utilizam este mecanismo para tratar o problema de acesso aos dados compartilhados.

No VSObjects, dados replicados não podem ser escritos por mais de um nodo. Para garantir que isso nunca aconteça, o VSObjects oferece a cada nodo a informação sobre qual porção de dados ele poderá escrever. Assim, um nodo de processamento pode distinguir qual objeto ele tem acesso de leitura e acesso de escrita.

## 4. Método de Verificação

A verificação e validação do VSObjects foi realizada em quatro estágios: (1) implementação, verificação e execução da versão seqüencial da convolução em Java; (2) implementação do ambiente de execução e da biblioteca do VSObjects e, posteriormente, implementação da convolução para o ambiente VSM construído; (3) implementação da versão seqüencial distribuída da convolução utilizando conceitos de programação distribuída em Java RMI; e (4) realização de testes e análise do desempenho do VSObjects em relação ao programa seqüencial e ao programa desenvolvido em

RMI, e comparação dos resultados obtidos com as diferentes implementações da convolução.

### 4.1 Convolução de Imagens

A operação de filtragem no domínio do espaço é chamada convolução. O termo domínio do espaço refere-se à agregação de pixels que compõem uma imagem, e operações no domínio do espaço são procedimentos aplicados diretamente sobre esses pixels [11]. Neste trabalho realizamos a convolução de imagens com filtros passa-baixa. A operação de convolução de imagem com filtro passa-baixa produz um efeito de borramento da imagem (suavizar). O efeito suavizado é produzido atenuando os componentes de alta freqüência da imagem. Os componentes de alta freqüência são filtrados e as informações nas freqüências baixas mudam de escala sem atenuação[11].

$$C[i,j] = \sum_{u=-\frac{M}{2}}^{\frac{M}{2}} \sum_{v=-\frac{M}{2}}^{\frac{M}{2}} I[(i+u),(j+v)] \times T[u,v]$$

**Figura 3 – Equação da Convolução de Imagem**

Uma operação de convolução 2D é definida pela Equação da figura 3, onde I[0..N-1, 0..N-1] é uma matriz NxN contendo a imagem que será processada, C[0..N-1, 0..N-1] é a imagem processada e T[0..M-1, 0..M-1] o filtro utilizado na convolução. Este problema foi escolhido por causa da sua simplicidade, seu alto grau de paralelismo e comunicação [23][24].

### 4.2 Ambiente de testes

O ambiente de testes utilizado foi um aglomerado composto de quatro computadores Athlon XP 2000+ 1.667 GHz, interconectados através de rede Fast Ethernet. O sistema operacional utilizado é o Windows XP.

Para os testes com as aplicações do VSObjects, uma quinta máquina foi acrescentada ao sistema, além das quatro que compõem o aglomerado. Esta quinta máquina é um Dual Pentium 933 MHz, também rodando o sistema operacional Windows XP, e foi utilizada para executar o ambiente de execução do VSObjects. Todas as máquinas usaram o compilador/interpretador J2SDK1.4.1_01, da Sun.

Os testes com a convolução de imagens foram realizados com uma variação das dimensões das imagens (512x512, 1024x1024 e 2048x2048) e com um filtro de convolução com dimensão fixa (11x11). No caso das implementações distribuídas, incluindo a do VSObjects, os testes foram executados com quatro processos, por causa do número de máquinas do aglomerado disponíveis e pela escolha do paralelismo puro.

### 4.3 Implementações

A versão seqüencial da convolução de imagens foi implementada em Java e validada através de algumas análises qualitativas e quantitativas, comparando os resultados com algumas imagens obtidas com softwares comerciais de processamento de imagens.

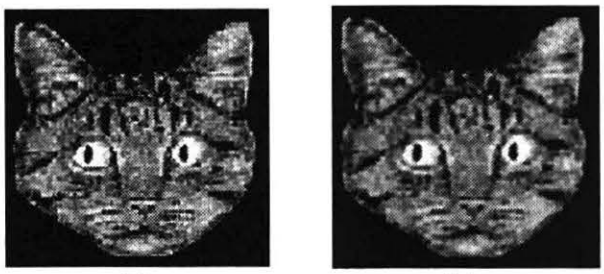
O programa de convolução de imagens para o VSObjects foi implementado em Java utilizando a biblioteca do VSObjects. Para executar o ambiente de execução do VSObjects, é necessário que sejam informados alguns parâmetros: localização da matriz da imagem (servidor), número de máquinas que irão compor o espaço virtual de memória compartilhada, atributos do objeto(dimensão da matriz de representação da imagem, no caso da convolução) e dimensão da máscara para convolução (também para o caso da convolução). Para a execução das aplicações desenvolvidas para o VSObjects, é necessário informar o IP da máquina onde a aplicação será executada através do VSObjects Server.

A implementação distribuída seqüencial da convolução de imagens foi realizada utilizando a biblioteca RMI do Java. A matriz de representação da imagem é armazenada em uma máquina separada fisicamente da máquina que roda o código da convolução da imagem. Os resultados da convolução foram verificados através da sua comparação com os resultados obtidos com a implementação seqüencial em Java.

## 5. Resultados

Para verificar o VSObjects, implementamos a versão paralela da convolução de imagem utilizando a sua biblioteca e verificamos os resultados obtidos com a sua execução comparando-os com os resultados obtidos com a versão seqüencial da convolução de imagens.

Da mesma forma, os resultados obtidos com a execução da versão seqüencial distribuída da convolução foram verificados através da comparação com os resultados obtidos com a versão seqüencial.



(a) (b)

**Figura 4 – (a) Imagem original, (b) Imagem convoluída com filtro passa-baixa**

As nossas análises quantitativas e qualitativas das imagens convoluídas (filtradas) mostraram que os resultados obtidos com todas as implementações eram

iguais e estavam corretos. Ou seja, as imagens foram convoluídas corretamente por cada uma das versões de implementação da convolução. A figura 4 (a) apresenta uma imagem original que foi convoluída por cada uma das implementações, resultando na imagem apresentada na figura 4 (b). Neste caso, a convolução foi realizada utilizando um filtro passa-baixa com dimensão 11x11.

Para realizar os testes descritos anteriormente no quarto e último estágio, utilizamos as mesmas configurações para todas as implementações, o que nos permitiu fazer a análise comparativa das três versões implementadas.

Os testes consistiram em executar todas as versões da operação de convolução de imagem implementadas, com imagens de dimensão 512x512, 1024x1024 e 2048x2048 e filtro passa-baixa de dimensão 11x11. Cada implementação foi executada 10 vezes para cada tamanho de imagem, e a média geométrica dos tempos de resposta obtidos foi utilizada para análise e comparação das implementações.

**Tabela 1 – Média Geométrica dos Tempos de Respostas Mínimos**

| Modelos / Dimensão da Imagem | Sequential | RMI | VSObjects |
|---|---|---|---|
| | 1129 | 67393 | 34164 |
| 1024x1024 | 4629 | 263426 | 134623 |
| 2048x2048 | 18608 | 1048307 | 528984 |

Durante os testes, a principal métrica utilizada foi o tempo de resposta que cada implementação da convolução gastou para executar a operação para os três tamanhos de imagem e o filtro escolhidos. O tempo de resposta foi medido considerando todo o tempo gasto para executar a operação, incluindo o tempo de processamento das operações e o tempo de comunicação entre as aplicações distribuídas.

A tabela 1 apresenta a média geométrica dos tempos de resposta obtidos com a execução de todas as implementações, para os três tamanhos de imagem escolhidos. Podemos verificar na tabela 1 que o menor tempo de resposta obtido para todas as dimensões de imagem foi o da implementação seqüencial centralizado. Este resultado era esperado e pode ser explicado pelo fato de que esta implementação não tem o gasto com o tempo de comunicação necessário para a execução das implementações com o VSObjects e com o RMI.

Comparamos e analisamos os tempos de resposta obtidos com a execução da implementação usando a biblioteca e o ambiente VSObjects e os tempos obtidos com a implementação seqüencial. A partir da análise, verificamos que a primeira obteve um tempo de resposta cerca de 30 vezes menor do que o obtido com a ultima, apesar de o VSObjects permitir uma implementação paralela da operação de convolução. Assim como a versão utilizando RMI, a versão com VSObjects obteve um desempenho baixo (tempos de resposta altos) em relação aos tempos obtidos com a versão seqüencial centralizada devido ao tempo gasto com a comunicação entre os processos e o ambiente de execução. A maior parte do tempo obtido com a aplicação do VSObjects foi gasta na inicialização da matriz de imagens, ou seja, quando o processo requisita o pedaço da imagem que ele necessita para a memória virtualmente compartilhada. Neste momento, existe comunicação de dados entre os nodos, e o protocolo de comunicação do RMI é bastante pesado. Isto torna o tempo de comunicação alto, degradando o desempenho. Na versão seqüencial, este tempo de inicialização da imagem não é necessário, devido ao fato de que imagem está armazenada na própria máquina (localmente). Em nenhuma das implementações foi considerado o tempo gasto com leitura do arquivo da imagem original, a medição dos tempos foi iniciada após o carregamento da imagem na memória principal.

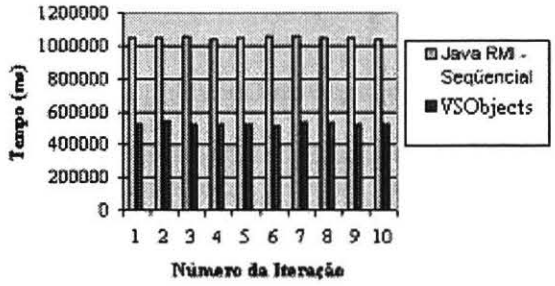



**Figura 5 – Tempo de Resposta Imagem 2048**

A figura 5 apresenta o gráfico dos tempos de resposta obtidos com as 10 iterações da execução das implementações utilizando RMI e VSObjects para a imagem com dimensão 2048x2048. Podemos verificar, a partir deste gráfico, que o tempo de resposta do VSObjects é menor que o tempo de resposta da implementação com RMI, em todas as iterações.

**Tabela 2 – SpeedUp RMI x VSObjects**

| Image Dimensions | | 1024x1024 | 2048x2048 |
|---|---|---|---|
| SpeedUp | 1,973 | 1,956 | 1,987 |

Observamos que o tempo de resposta obtido com a implementação utilizando VSObjects é 1.9 vezes menor do que o tempo de resposta obtido com a implementação usando RMI, para todos as dimensões de imagens executadas. Este resultado pode ser explicado pelo fato de que a execução da convolução utilizando a biblioteca e o ambiente VSObjects foi realizada em paralelo, utilizando quatro processos. O VSObjects foi implementado sobre o

RMI e por este motivo o speeUp foi menor que o esperado.

A tabela 2 apresenta o speedup do VSObjects em relação a implementação RMI. Em média, o speedup do VSObjects foi de 1.9 em relação a implementação seqüencial com RMI. Com a utilização de threads para realizar os acessos de leitura e escrita na matriz de imagem, o VSObjects conseguiu melhorar seu tempo de resposta alcançando assim uma melhora significativa nos tempos de resposta, em relação à implementação distribuída com RMI, em todas as suas execuções.

**Tabela 3 – Média Geométrica dos Tempos de Processamento da Convolução**

| Modelos / Dimensão da Imagem | Sequential | RMI | VSObjects |
|---|---|---|---|
| 512x512 | 1129 | 1137 | 171 |
| 1024x1024 | 4629 | 4642 | 714 |
| 2048x2048 | 18608 | 18654 | 2920 |

A tabela 3 apresenta a média geométrica dos tempos de processamento da operação convolução de imagens. Nestes tempos de resposta, desconsideramos os tempos de comunicação entre os processos e o ambiente de execução do VSObjects e também desconsideramos os tempos de comunicação da implementação RMI. Podemos verificar através da tabela 3 que, da mesma forma, o menor tempo de resposta obtido foi o do VSObjects. Isto já era esperado, pois, o VSObjects utiliza processamento paralelo e nestes testes foram utilizados quatro processos para executar a operação de convolução de imagens no VSObjects, enquanto que, nas implementações seqüencial e usando RMI somente um processo realizava toda a operação.

**Tabela 4 – SpeedUp Seqüencial x VSObjects**

| Image Dimensions | 512x512 | 1024x1024 | 2048x2048 |
|---|---|---|---|
| SpeedUp | 6,573 | 6,483 | 6,372 |

A tabela 4 apresenta o speedup (considerando somente o tempo de processamento) do VSObjects em relação a implementação seqüencial. Em média, o speedup do VSObjects foi de 6.5 em relação a implementação seqüencial. Com a utilização de processamento paralelo (quatro processos e um run-time) para realizar a operação de convolução, o VSObjects conseguiu melhorar seu tempo de processamento alcançando assim uma melhora significativa nos tempos de processamento, em relação à implementação distribuída com RMI e a implementação seqüencial, em todas as suas execuções. Além disso o VSObjects server cria uma thread para cada requisição de leitura do objeto compartilhado, realizado por um nodo.

O VSObjects foi implementado utilizando a tecnologia Java RMI, como já citado anteriormente, por isso os tempos de resposta obtidos, considerando os tempos de comunicação, foram tão altos. A tecnologia Java RMI não possui bom desempenho com relação à comunicação e este fator degradou o desempenho obtido com a implementação usando VSObjects, principalmente quando comparado com o obtido com a versão seqüencial. Porém, quando consideramos somente o tempo de processamento o VSObjects obteve melhor desempenho do que as demais implementações.

## 6. Conclusões

Baseado na verificação do VSObjects e nos resultados obtidos, concluímos os objetivos propostos no trabalho foram alcançados, com a verificação da implementação do software de gerenciamento de memória virtualmente compartilhada em ambientes de memória fisicamente distribuída entre os nodos do aglomerado, que executa aplicações paralelas desenvolvidas em Java. Além disso, avaliamos o desempenho do VSObjects em relação a outros modelos de programação (programação distribuída e programação centralizada).

Todos os testes foram realizados utilizando a aplicação de convolução de imagem. A avaliação dos resultados mostrou que a versão seqüencial da convolução obteve os menores tempos de resposta. A versão paralela da convolução, implementada com a biblioteca do VSObjects e executada através do ambiente de execução obteve melhor desempenho (menor tempo de resposta) que a versão distribuída desenvolvida com a biblioteca RMI da linguagem Java.

O desenvolvimento de aplicações que manipulam matrizes utilizando a biblioteca do VSObjects é mais fácil do que o desenvolvimento das mesmas usando Java RMI. Isto acontece porque o programador não precisa preocupar-se com as diversas diretivas como no RMI. A versão distribuída desenvolvida com Java RMI obteve o maior tempo de resposta. Esse resultado era esperado devido ao tempo gasto para a comunicação entre os processos distribuídos.

Na versão atual do VSObjects possui uma limitação com relação ao espaço virtualmente compartilhado. Uma vez declarado o número de objetos que irão compor o espaço compartilhado este não deverá ser alterado até o final da execução da aplicação.

A principal contribuição deste trabalho é a proposta e implementação do VSObjects, que provê um ambiente de programação paralela em aglomerados com memória virtualmente compartilhada, mas fisicamente distribuída. O VSObjects libera o programador da preocupação com a movimentação das informações compartilhadas .Outras contribuições são as implementações da convolução de

imagens no modelo distribuído e no modelo seqüencial, e a análise dos resultados obtidos com as diferentes implementações.

Os resultados obtidos podem ser melhorados a partir da implementação do VSObjects usando outros protocolos, como TCP, para comunicação (sem RMI). Além disso, a distribuição dos dados compartilhados entre os nodos, descentralizando o servidor (de objetos) pode reduzir o impacto que o gargalo da centralização de dados acarreta na execução.

## 7. Trabalhos Futuros

Como trabalhos futuros, já iniciados, estamos desenvolvendo uma nova versão do VSObjects, nesta versão o foco é a melhoria do acesso às informações compartilhadas, uma vez que este ponto foi menos trabalhado na primeira versão do VSObjects.

Além disso, realizaremos comparações dos resultados obtidos com o VSObjects com outras implementações de convolução de imagem usando bibliotecas de passagem de mensagem como PVM [25] e MPI [26], de programação multithread (winthread e pthread) [27], e usando a API de programação paralela OpenMp [18].

## 8. Agradecimentos

Nossos agradecimentos à Pro-reitoria de Pesquisa e de Pós-graduação (ProPPG) da PUC-Minas e ao CNPq pelo suporte e financiamento do projeto FIP-2002/44P. E um agradecimento especial, pelo apoio/suporte, ao Laboratório de Sistemas Digitais e Computacionais, ao Grupo de Sistemas Digitais e Computacionais e aos colegas e amigos do PPGEE (Programa de Pós-Graduação em Engenharia Elétrica).

## 9. Referências

[1] Penha, D. O., Corrêa, J. B. T., Martins, C. A. P. S. Análise Comparativa do Uso de Multi-Thread e OpenMp Aplicados a Operações de Convolução de Imagem. *III Workshop de Sistemas Computacionais de Alto Desempenho (WSCAD)*, pages 118-125. Vitória, Brazil, 2002.

[2] Bennett, J. K., Carter, J. B., Zwaenepoel, W., "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence", 1990.

[3] Rossetti, R. J. F., "Dash: Uma aplicação DSM".

[4] Li, K.; "Shared Virtual Memory on Loosely Coupled Multiprocessors", Ph.D. Thesis, Yale Univ., 1986.

[5] Li, K.; Hudak, P. "Memory Coherence in Shared Virtual Memory Systems", ACM Trans. On Computer Systems, vol. 7, 1989.

[6] URL: http://web.mit.edu/dsm/Tutorial/tutorial.htm

[7] Colouris G.; Dollimore J.; Kindberg T. "Distributed Systems: Concepts and Design". 3rd Edition. Addison-Wesley. 2001.

[8] Scientific Computing Associates , 2003 http://www.lindaspaces.com/products/vsm_mp.html

[9] C. A. P. S. Martins, "*Subsistema de exibição de imagens digitais com desacoplamento de resolução - SEID-DR*", tese de doutorado, Universidade de São Paulo, SP, 1998.

[10] Almasi G.S. and Gottlieb A., Highly Parallel Computing, 2.ed, Benjamin/Cummings, 1994.

[11] Gonzalez R.C. and Woods R.E., *Digital Image Processong* 2nd Edition, Addison Wesley Publishing Co., Massachusetts, 1987.

[12] Hwang K. and Xu Z., "Scalable Parallel Computing: Technology, Architecture, Programming", McGraw-Hill, 1998.

[13] Yue K.K., Lilja D.J., "An Effective Processor Allocation Strategy for Multiprogrammed Shared-Memory Multiprocessors", IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 12, Dezembro 1997.

[14] Tanenbaum A. and Woodhull A., "Operating Systems Design and Implementation", 2.ed., Editora Prentice Hall, Upper Saddle River, New Jersey, 1997.

[15] Schimmel C., "UNIX Systems Architectures Symetric Multiprocessing and Caching for Kernel Programmers", Addison-Wesley Professional Computing Series, 1994.

[16] URL: http://www.linux.org

[17] URL: http://www.microsoft.com/windows/default.asp

[18] "Introduction to OpenMP", Advanced Computational Research Laboratory, Faculty of Computer Science, UNB Fredericton, New Brunswick.

[19] Gayasen A., Parashar A., "Cache Coherence and Consistency Issues in Distributed Systems". December, 2002

[20] Lawrence R., "A Survey of Cache Coherence Mechanisms in Shared Memory", Department of Computer Science, University of Manitoba. May, 1998.

[21] Grosso, W., Java RMI - Java remote method invocation. Sebastopol: O'Reilly, 2002.

[22] www.java.sun.com.

[23] Martins, C. A. P. S., J. Zuffo, S. Kofuji, "Two Dimensional Normalized Sampled Finite Sinc Reconstructor", inAeroSense'97, Proc. SPIE-3074, SPIE, Orlando, 1997.

[24] Ratha N.K., A.K. Janin, and D.T. Rover, "Convolution on Splash 2 ", Michigan State University, *IEEE*, 1995.

[25] "PVM: Parallel Virtual Machine", URL: www.epm.ornl.gov/pvm/pvm_home.html

[26]Gropp, W., Lusk E., "An Introduction to MPI Parallel Programming with the Message Passing Interface", Argonne National Laboratory URL: http://www-unix.mcs.anl.gov/mpi/tutorial/mpiintro/index.htm

[27] Penha, D. O., Corrêa, J. B. T., Góes L. F. W., Ramos L. E. S., Pousa C. V., Martins, C. A. P. S., "Comparative analysis of multi-threading on different operating systems applied on digital image processing". International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA'03) . Rio de Janeiro, Brasil.