JVSObjects: Middleware Java para Compartilhamento Virtual de Objetos Concorrentes

Christiane Vilaça Pousa[1], Carlos Augusto Paiva Silva Martins[1]

[1]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, capsm@pucminas.br

Resumo. *Este trabalho apresenta um middleware que suporta o desenvolvimento de aplicações em Java baseadas no modelo de programação com variáveis compartilhadas executadas em ambientes distribuídos. JVSObjects é um VSM (Virtual Shared Memory) baseado em replicação de objetos e no modelo de consistência forte, desenvolvido em Java. O nosso principal objetivo é apresentar o JVSObjects, analisar seu uso e desempenho usando uma aplicação, distribuída que explora concorrência, comparando com a implementação usando somente RMI(Remote Method Invocation). A principal contribuição deste trabalho é o middleware JVSObjects.*

1. Introdução

Sistemas computacionais distribuídos (SCD) têm sido cada vez mais utilizados, tanto em ambientes de produção quanto de pesquisa (industrial e acadêmico). Estes sistemas apresentam grandes vantagens sobre os sistemas centralizados, como: maior disponibilidade, menor custo de implementação, maior escalabilidade, exploração de concorrência, entre outras.

Muitas aplicações demandam uma grande quantidade de recursos computacionais devido à necessidade de obtenção de tempos de resposta cada vez menores [Penha, Corrêa e Martins 2002]. Em SCD, para se alcançar uma utilização efetiva de concorrência/paralelismo, torna-se imperativo a manutenção de um ambiente que permita a execução eficiente de diversos processos/threads. Mesmo utilizando estes ambientes, o desenvolvimento de programas concorrentes/paralelos continua sendo uma tarefa difícil e não intuitiva para o programador (mapeamento dos processos/threads, particionamento do programa, comunicação entre os nodos, etc) [Morazanm e Troeger 2001].

A dificuldade, nos SCD, é ainda maior devido à necessidade do gerenciamento de comunicação entre os nodos do sistema. Este gerenciamento pode ser realizado através do modelo de passagem de mensagem, porém, este modelo torna-se muito complexo para o programador, que tem que se preocupar com a comunicação dos dados, sincronização, entre outras coisas. Como exemplos deste modelo encontramos nos meios acadêmico e industrial os Sockets [Java 2004] e bibliotecas desenvolvidas sobre eles. Um outro modelo utilizado é o RPC (Remote Procedure Call) que consiste

em permitir que um processo cliente realize uma chamada a um procedimento, como se fosse uma chamada local, mas que na verdade é executado em uma máquina remota; como exemplo deste modelo pode-se citar o RMI (Remote Method Invocation) [Java 2004] e o CORBA (Common Object Request Broker Architecture) [Corba 1997]. A transparência de gerenciamento de paralelismo e comunicação desejada pelo usuário e/ou programador pode ser obtida através do modelo de memória compartilhada distribuída. O modelo de memória compartilhada distribuída pode ser definido como sendo aquele que permite e aplica modelos de programação usando variáveis compartilhadas (compartilhamento lógico de dados) sobre sistemas computacionais de memória distribuída [Colouris , Dollimore e Kindberg 2001] [Li 1986][Li 1989].

Java [Java 2004] tem sido amplamente usada como uma linguagem de programação para processamento distribuído concorrente/paralelo. E a razão é muito simples, é orientada à objetos, portátil, e oferece suporte para aplicações distribuídas (RMI e CORBA)[Tomioka, Guimarães e Prado 2002]. As aplicações Java desenvolvidas para SCD, que utilizam concorrência/paralelismo, são desenvolvidas através do uso de pacotes (RMI, CORBA, Sockets e Threads), que oferem suporte a comunicação entre os nodos e a concorrência/paralelismo. Ou seja, o desenvolvedor destas aplicações trabalha explicitamente com os métodos destes pacotes, o que pode tornar o desenvolvimento de aplicações distribuídas concorrentes/paralelas muito trabalhoso e complexo.

Esta pesquisa e a conseqüente proposta e implementação do JVSObjects foi motivada pelo fato de não encontrarmos em Java, na literatura estudada, um único pacote que ofereça suporte implícito ao compartilhamento de objetos distribuídos em um SCD que explore concorrência/paralelismo.

Alguns ambientes para compartilhamento de objetos implementados em Java foram apresentados em trabalhos encontrados na literatura, entre eles podemos citar: MultiJav [Chen e Allan 1998], JavaSpace [Sun 1999]e JavaNOW [Thiruvathukal, Dickens e Bhatti 1998]. Estes ambientes foram implementados através de modificações na linguagem Java e na JVM (Java Virtual Machine), diferentemente do JVSObjects que foi implementado como middleware.

O JVSObjects é um middleware, que oferece suporte à construção de aplicações desenvolvidas em Java, baseadas no modelo de programação com variáveis compartilhadas em ambientes distribuídos. O middleware faz a gerência da memória compartilhada entre os nodos do SCD, criando um espaço virtual de compartilhamento de objetos. Este middleware faz parte de um projeto maior chamado VSObjects [Pousa 2003], que é independente de linguagem de programação.

Neste artigo nosso principal objetivo é apresentar o JVSObjects, analisar o código, o desenvolvimento e o desempenho da aplicação de convolução de imagem paralela desenvolvida com o JVSObjects comparando com uma implementação desenvolvida com RMI.

Nas próximas seções apresentamos alguns conceitos básicos de compartilhamento de objetos concorrentes; o JVSObjects, suas características principais de arquitetura e implementação; alguns resultados obtidos com a verificação do JVSObjects e analisamos os resultados obtidos com os testes realizados. Finalmente, apresentamos as conclusões sobre os resultados alcançados e os trabalhos futuros.

2. Compartilhamento de Objetos Concorrentes

A existência de um espaço único de endereçamento pode melhorar a programabilidade em máquinas paralelas e distribuídas.

2.1 Compartilhamento centralizado

Em arquiteturas paralelas de memória compartilhada, o modelo de variáveis compartilhadas pode ser utilizado para comunicação e sincronização entre os processos/threads da aplicação. Este modelo é utilizado em sistemas operacionais multiprocessados que aplicam paralelismo SMP *(Symmetric Multiprocessing*) [Hwang e Xu 1998] de forma implícita, para criar a transparência que pode melhorar a programabilidade das aplicações.

Muitas técnicas são conhecidas e utilizadas para programação de multiprocessadores. Para comunicação, um processador escreve dados na memória, para serem lidos por todos os outros ou por alguns processadores. Para sincronização, podem ser usadas sessões críticas, implementando semáforos, monitores, entre outras, para prover a exclusão mútua [Tanenbaum e Woodhull 1997] necessária entre as Threads/Processos.

2.2 Compartilhamento Distribuído

Na programação paralela de memória distribuída, como nos SCD, onde não existe memória fisicamente compartilhada, o espaço único de endereçamento pode ser obtido usando o modelo de memória compartilhada distribuída.

No modelo de memória compartilhada distribuída, o acesso às informações compartilhadas entre os nodos de processamento do SCD é implícito para o programador. Neste modelo o programador não tem que se preocupar com a consistência e coerência das informações, entre outras coisas; isto é garantido por protocolos ou por hardwares específicos [Adve e Gharachordo 1995][Lawrence 1998].

Os modelos de consistência de memória são responsáveis por indicar quando uma escrita estará visível (atualização da memória compartilhada) para os outros nodos do SDC [Gayasen e Parashar 2002] [Lawrence 1998]. Existem basicamente dois tipos de consistência de dados, a consistência forte e a consistência fraca. Na consistência forte todas as operações são realizadas de acordo com uma ordem, e nenhuma operação é iniciada antes do término da operação anterior. Enquanto que na consistência fraca as operações de leitura e de escrita são relaxadas, permitindo que elas sejam realizadas sem que exista uma ordem entre elas, melhorando o desempenho do modelo.

Para manter a coerência, são usados protocolos que determinam como a memória compartilhada será tratada pelos nodos, em operações de escrita. Para manter a coerência das informações os protocolos podem ser de invalidação ou de atualização de informações [Colouris, Dollimore e Kindberg 2001]. Esses garantem que objetos compartilhados vão ser invalidados ou atualizados através de mensagens enviadas para os nodos do SCD.

3. JVSObjetcs

O JVSObjects é um middleware que oferece suporte ao compartilhamento virtual de objetos distribuídos concorrentes, em aplicações Java concorrentes/paralelas desenvolvidas para SCD; que faz parte de um projeto maior chamado VSObjects.

JVSObjects, foi implementado em Java, utilizando as bibliotecas de Thread (concorrência) e RMI (Comunicação entre os nodos do SCD) desta linguagem. Esta linguagem foi escolhida por oferecer independência de plataforma (portabilidade) e conjunto de APIs para documentação da implementação do JVSObjects. Com o pacote RMI a complexidade de implementação do ambiente de execução do middleware foi amenizada, pois, este oferece uma interface que facilita a programação de sistemas distribuídos. Além disso, a linguagem Java tornou-se motivação para o desenvolvimento deste trabalho, uma vez que não encontramos na literatura estudada, um pacote em Java que disponibilizasse o compartilhamento virtual e implícito de objetos distribuídos. A seguir apresentaremos a arquitetura, à implementação, o modelo de consistência e coerência, as funções da biblioteca e o desenvolvimento de aplicações com JVSObjetcs.

3.1 Arquitetura

No JVSObjects algumas características arquiteturais do VSObjects foram implementadas. Nesta primeira versão do middleware, entre estas características arquiteturais podemos citar: controle de consistência e coerência centralizados, modelo de consistência forte, política de coerência que permite acessos de leitura concorrentes, réplica de objetos, biblioteca com métodos para compartilhamento de objetos distribuídos concorrentes e um ambiente de execução que permite ao programador/usuário gerenciar as execuções das aplicações.

A figura 1 apresenta a arquitetura do JVSObjects, que pode ser dividida em três unidades: Unidade de Controle e Gerenciamento e Unidade de Comunicação, que ficam localizadas no nodo servidor e Unidade de Objetos que fica localizado nos demais nodos do SCD, onde as aplicações são executadas. A Unidade de Controle e Gerenciamento esta dividida em três módulos: Módulo de Controle responsável por manter consistência, coerência e as réplicas dos objetos compartilhados; Módulo de Gerenciamento que gerencia as mensagens que trafegam na rede e Módulo de Estatística responsável por gerar *logs* com informações sobre a execução das aplicações. A Unidade de Comunicação é dividida em dois módulos: Módulo de Comunicação responsável por manter a comunicação no Servidor e Módulo de Segurança responsável por manter segurança na comunicação e nas mensagens que são trocadas na rede. A Unidade de Objetos está dividida em dois módulos: Módulo de Comunicação que é responsável por manter a comunicação nos nodos do SCD e Módulo de Bibliotecas que possui as bibliotecas com métodos para manipulação dos objetos compartilhados.

As três unidades que compõem a arquitetura do middleware JVSObjects, trocam mensagens entre si; fazendo com que os módulos mantenham comunicação um com o outro como pode ser observado na figura1. Apenas o Módulo de Estatística na Unidade de Controle e Gerenciamento não envia mensagens para os outros módulos, este módulo apenas recebe mensagens dos outros módulos para gerar os *logs*.

A principal diferença entre o projeto VSObjects e o middleware JVSObjects é que o primeiro é um projeto mais completo e independente de linguagem de programação, enquanto que o segundo foi desenvolvido apenas para aplicações na linguagem Java e possui apenas algumas das características arquiteturais do VSObjetcs.

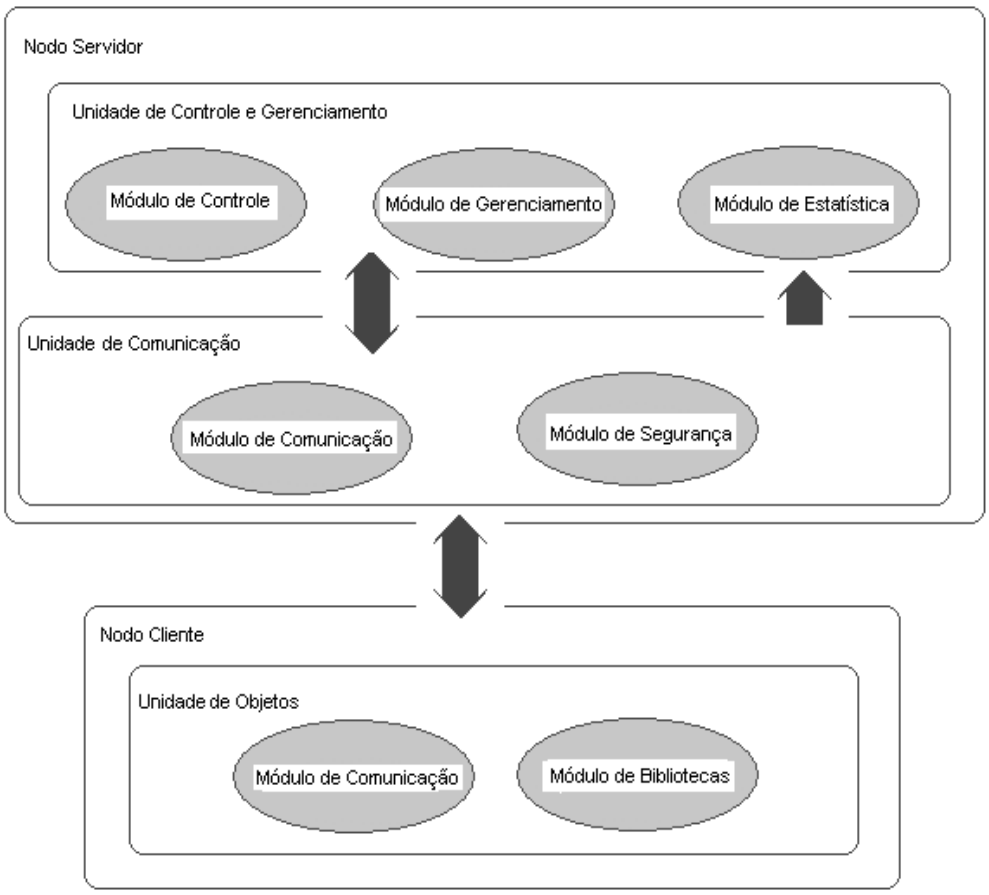

Figura 1. Arquitetura do JVSObjects

3.2. Implementação

Para implementar o ambiente JVSObjects, foi desenvolvido um ambiente de execução, localizado no nodo servidor, (*JVSObjects Server*) que utiliza o pacote RMI para realizar a comunicação com os demais nodos e o pacote Thread que oferece suporte a concorrência/paralelismo, ambos providos pela linguagem Java e um conjunto de bibliotecas com métodos para manipulação dos objetos .

O ambiente de execução desenvolvido é responsável por manter a transparência de acesso entre os objetos da memória virtualmente compartilhada e a coerência e consistência das informações compartilhadas entre os nodos cliente. Nesta primeira versão do JVSObjects optamos por implementar o controle centralizado, realizado pelo ambiente de execução. O modelo de consistência implementado foi o modelo forte centrado em dados, onde todos os nodos observam as mesmas seqüências de operações nos objetos compartilhados. Além disso, implementamos a política de coerência de múltiplos leitores/único escritor. Na interface gráfica do ambiente de execução do JVSObjects, além do gerenciamento dos objetos compartilhados, é permitido ao usuário/programador configurar as aplicações a serem executadas no SCD e obter

informações sobre a execução (Tempo de resposta, overhead de comunicação, tempo de processamento, mensagens trocadas entre os nodos e o ambiente de execução, etc).

Além do ambiente de execução o middleware disponibiliza um conjunto de bibliotecas para criação e manipulação dos objetos que representam a memória virtualmente compartilhada em cada nodo cliente do SCD. A biblioteca Shared_Object contém as definições do objeto compartilhado e é executada em todas as máquinas do SCD que executam partes paralelas da aplicação. Os objeto definidos como Shared_Object são uma informação compartilhada pelos nodos durante a execução das aplicações. O conjunto de objetos Shared_Object compõe a memória virtualmente compartilhada no JVSObjects.

O JVSObjects permite o compartilhamento de objetos remotos concorrentes, sem que, para isso, o programador tenha que se preocupar com a criação de *interfaces* e classes remotas, como é necessário no RMI. Além disso, o programador não tem que implementar as *thread*, pois, o ambiente de execução cria e dispara, de acordo com as configurações realizadas nele, que são os objetos concorrentes remotos. O programador, apenas, tem que desenvolver uma aplicação em Java, utilizando os métodos das bibliotecas do JVSObjects, como se fosse uma aplicação centralizada. Nas próximas seções apresentaremos o modelo de consistência e coerência implementado e os métodos da biblioteca para manipulação dos objetos compartilhados.

3.2.1. Consistência e Coerência dos Objetos

O middleware cria réplicas dos objetos durante a execução e o ambiente de execução do JVSObjects é o responsável por manter a consistência e coerência destas réplicas nos vários nodos do SCD que executam a aplicação.

Nesta primeira versão do JVSObjects, a política implementada é a consistência seqüencial, onde todos os processos têm as mesmas intercalações das operações de escrita e leitura. Além disso, neste modelo não existe relação ao tempo, mas sim a ordem na qual as operações ocorrem na aplicação. Este modelo é simples de implementar, porém, ele acrescenta à aplicação um grande *overhead* na comunicação entre os nodos, pois, neste modelo a ordem das operações na aplicação é mantida. Nas próximas versões do JVSObjects outras políticas que geram menor *overhead* serão implementadas.

A coerência dos objetos compartilhados no JVSObjects 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 pode ser lida e escrita 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 no objeto em questão.

3.2.2. Métodos do JVSObjects

Como já mencionado anteriormente, o JVSObjects é composto de um conjunto de bibliotecas e um ambiente de execução de aplicações. Na biblioteca Shared_Object foram definidos alguns métodos, que permitem ao programador acessar os objetos compartilhados e assim realizar leitura e escrita nos objetos compartilhados entre os nodos. Além de métodos de acesso, foram criados métodos de inicialização e de finalização dos objetos compartilhados. Estes métodos, além de gerenciar o conteúdo inicial e final dos objetos compartilhados, enviam ao servidor do JVSObjects informações sobre o nodo onde este objeto está sendo inicializado ou finalizado.

Nesta primeira versão do JVSObjects, os objetos compartilhados só podem ser escalar primitivos, vetores de inteiros e matrizes de inteiros. Nas próximas versões serão implementados métodos que dêem suporte para outros tipos de objetos. Os métodos são:

write() – dependendo do tipo de objeto compartilhado (escalar, vetor ou matriz) este método poderá receber até no máximo três parâmetros (linha, coluna, novo valor), que são inteiros. Este método retorna um inteiro, que sendo negativo, expressa um erro na escrita do objeto compartilhado e sendo positivo expressa que o objeto foi atualizado.

read() – retorna um valor inteiro do objeto. Assim como o write, este método possui alguns parâmetros, que dependendo do tipo de objeto este método poderá receber até dois parâmetros (linha e coluna), que também são inteiros. Este método retorna um inteiro, que sendo negativo, expressa um erro na leitura do objeto compartilhado. Caso contrário, o valor do objeto compartilhado é fornecido à aplicação.

Start() – fornece ao ambiente de execução informações (tipo de objeto compartilhado, nome do objeto, etc) sobre o nodo e inicializa os objetos compartilhados naquele nodo. Este método não possui parâmetros. Este método retorna um inteiro, que sendo negativo, expressa um erro na inicialização do objeto compartilhado e sendo positivo expressa que o objeto foi inicializado.

Stop() – este método finaliza o objeto compartilhado no nodo e libera o nodo do processamento. Este método não possui parâmetros. Este método retorna um inteiro, que sendo negativo, expressa um erro na liberação do objeto e sendo positivo expressa que o objeto foi finalizado.

3.3 Desenvolvendo aplicações com JVSObjects

3.3.1 Implementando uma aplicação

Apresentamos o exemplo da implementação da aplicação, convolução de imagens, que será utilizada neste artigo para realização da verificação do JVSObjects. No tópico 4 apresentaremos as principais características desta aplicação. Na figura 2 apresentamos um exemplo de aplicação desenvolvida com a biblioteca do JVSObjects, onde os métodos desta bibliotecas estão circulados.

No início do código o programador deverá adicionar a declaração de um objeto do Tipo Shared_Objetc, neste caso como é um objeto matriz, o tipo é Shared_Matrix.
Realizada a declaração do objeto, é necessário inicializar o objeto compartilhado com as informações que são compartilhadas no espaço de endereçamento virtual gerenciado pelo ambiente de execução, o método para realizar a inicialização do objeto é o start().

Todos os acessos de leitura a este objeto devem ser feitos com o método read(), no caso desta aplicação este método recebe alguns parâmetros (linha e coluna da imagem). Os acessos de escrita são feitos com o método write(), que além da linha e coluna, necessários apenas em aplicações com matrizes, o valor a ser escrito deve ser um parâmetro deste método. O último método utilizado nas aplicações deve ser o stop(), ele informa ao ambiente de execução que aquele objeto não pertence mais a memória virtualmente compartilhada.

```
public class Convolucao_Java
{           ........
            public Shared_Matrix SharedImage; // Objeto compartilhado

    public Convolucao_Java(......){   ......  }

    public void convolucao(){
        try {
              .........

         SharedImage.start();

         for(i = startWindowT; i < endWindowT; i++)
           for(j = 0; j < imageTam; j++) Tmp[i][j] = SharedImage.read(i,j);

            for(i = startWindow; i < endWindow; i++){
              for(j = bound; j < (imageTam - bound); j++) {  sum = 0;
                  for(m = 0; m < filterDim; m++)
                    for(n = 0; n < filterDim; n++){
                      sum += Filter[m][n] * Tmp[m + i - bound][n + j - bound]; }
                sum = sum/FatDiv;
                temp = ((sum >= 0) && (sum < 256) ? sum : 255);
                SharedImage.write(i,j,temp); }}

         SharedImage.stop();}
        catch (Exception e) {  System.out.println("Erro!!");}}

    public static void main(String[] args)
    {    ........
        Convolucao_Java test = new Convolucao_Java();
        test.convolucao();
        .........}}
```

Figura 2. Exemplo de programa desenvolvido usando o JVSObjects

3.3.2 Executando uma aplicação

Para executar uma aplicação no JVSObjects, é necessário que o *rmiregistry* do Java seja inicializado em todos os nodos clientes que farão parte do processamento e no nodo servidor que irá executar o ambiente de execução. Uma cópia da aplicação deve estar nos nodos do SCD e então basta inicializar o ambiente de execução do JVSObjetcs executando a sua interface gráfica, através do comando: *Java Server*. Após inicializado, é necessário que se informe o número de objetos compartilhados e a aplicação a ser executada, na sua interface gráfica.

4. Verificação do JVSObjects

Para verificar o JVSObjects, vamos apresentar dois tipos de testes: análise do código e análise de desempenho da aplicação com a biblioteca do JVSObjects. Nas próximas seções apresentaremos o ambiente de experimentação, as características da aplicação de convolução de imagens, análise do código e os resultados de desempenho.

4.1. Ambiente de Experimentação

O ambiente de experimentação foi um SCD composto de quatro computadores Athlon XP 2000+ 1.667 GHz, interconectados através de rede Fast Ethernet com Switch/Hub. O sistema operacional utilizado é o Windows XP.

Para os testes com as aplicações do JVSObjects, uma quinta máquina foi acrescentada ao sistema, além das quatro que compõem o SCD. Esta quinta máquina é um Dual Pentium 933 MHz, também com sistema operacional Windows XP, e foi utilizada para executar o ambiente de execução do JVSObjects, nodo servidor. Idealmente, o ambiente de execução não deve estar localizado nos nodos onde o processamento acontece, para obter um melhor balanceamento de cargas; mas se for necessário, o ambiente de execução poderá ser executado no mesmo nodo que executa a aplicação. Todos os nodos usaram o compilador/interpretador J2SDK1.4.1_01, da Sun.

Para realizar os testes com o JVSObjetcs, implementamos a aplicação de convolução de imagens em duas versões: versão 1 - distribuída com RMI e versão 2 - distribuída com os métodos da biblioteca do JVSObjects.

Os testes com a convolução de imagens foram realizados com uma variação das resoluções das imagens (512x512, 1024x1024 e 2048x2048) e com um filtro de convolução com dimensão fixa (11x11). Os testes foram executados com quatro processos, por causa do número de nodos clientes disponíveis e pela escolha da eficiência (um processo/Thread em cada nodo do SCD).

4.2. Convolução de Imagens

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 aplicação que demandam uma grande concorrência. Imagens são compostas por uma grande quantidade de dados e freqüentemente são armazenadas em matrizes de *pixels* e geralmente o custo computacional para manipulá-las é alto [Martins 1998]. Além disso, esta operação tem natureza paralela porque realiza ações independentes sobre dados independentes (*pixels* da imagem, que são geralmente os elementos que compõem a matriz e representação da imagem) [Gonzalez e Woods 2000].

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 [Gonzalez e Woods 2000]. Neste trabalho realizamos a convolução de imagens para a implementação de filtros passa-baixa.

4.3. Resultados

Para analisar os resultados obtidos com o JVSObjects, implementamos a versão distribuída concorrente da convolução de imagem utilizando a sua biblioteca e verificamos, comparando seu código e execução com a versão implementada somente com RMI. Nas próximas seções apresentaremos as análises do código e de desempenho das duas versões.

4.3.1. Análise do Código

A nossa análise de código será realizada, baseada na quantidade de pacotes, classes e métodos necessários para implementar a aplicação e no grau de dificuldade para o uso destes métodos e pacotes.

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;

/** Interface do objeto remoto */
public interface NodoInterfaceMestre extends Remote {

  public boolean vizinhoDesconexao(......)  throws RemoteException ;

  public boolean vizinhoConexao(.......)  throws RemoteException ;

  public void adicionarResultados(......) throws RemoteException;
}
```

Figura 3. Interface do servidor

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;


/** Interface do objeto remoto */
public interface NodoInterfaceEscravo extends Remote {

   public void convolucaoRemota (.......) throws RemoteException;

}
```

Figura 4. Interface do cliente

```
class NodoThread extends  Thread{
    .......
   NodoThread(.......)
    {  super(); ....... }

    public void run()
    { try{Destino.convolucaoRemota(.......);}
        catch(Exception e){ ... })
}
public class JNodoMestre extends UnicastRemoteObject
                    implements NodoInterfaceMestre
{
 .......
  private String caminhoImagem;
  private int tamanhoImagem,dimensaoFiltro,ImgFinal[][];

  NodoInterfaceMestre nodoPesquisa;

  public JNodoMestre (String meuNome) throws RemoteException {
    super(); .....  }

  public boolean vizinhoConexao (....)
               throws RemoteException {
   ........
 }


  public boolean vizinhoDesconexao (....)
               throws RemoteException {
      ....  }

  public void convolucaoDispara (.....) {
    try {
          ........
        for(int i=0; i < nodosVizinhos.size(); i++) {
          ........
          //dispara as Threads que vao realizar a convolucao
          NodoThread pesquisa = new NodoThread(this,.....);
          pesquisa.start(); }}
    catch (Exception e) { .....} }

  public synchronized void adicionarResultados(int NLinhas,
             int IniJan,int Img[][]) throws RemoteException {
    try { ..... }
    catch (Exception e) { ...  }  }
}
```

Figura 5. Implementação da Interface do servidor

```
public class JNodoEscravo extends UnicastRemoteObject
                          implements NodoInterfaceEscravo
{
 ......

 private int Filtro[][];
 private int Tmp[][];
 private int Img[][];

 .......

 public JNodoEscravo (....) throws RemoteException {
   super();
   ......
   Filtro = new int[2048][2048];
   Tmp = new int[2048][2048];
   Img = new int[20][20]; }
public void LerArquivoImagem(.....){
  try{
    ......)   catch(Exception e){}  }

 public void conectar (String nomeVizinho) throws RemoteException {
   try
   {
     NodoVizinho meuVizinho = new NodoVizinho
      (nomeVizinho,(NodoInterfaceMestre)Naming.lookup(nomeVizinho)).
      ......        }
    catch (Exception e) {.... }  }

 public void desconectar (String nomeVizinho)
                          throws RemoteException {
 try { ....
     meuVizinho.Referencia.vizinhoDesconexao(meuNome);
     ......  }}
 catch (Exception e) {....} }

public void convolucaoRemota (.......) throws RemoteException {
  try {
   int i, j, m, n,Soma,FatDiv,Borda,FimJan
   .......
   LerArquivoImagem(.....);

   for(i = (IniJan + Borda); i < (FimJan - Borda); i++)
   { for(j = Borda; j < (tamImagem - Borda); j++){ Soma = 0;
       for(m = 0; m < DimFil; m++)
         for(n = 0; n < DimFil; n++)
           Soma += Filtro[m][n] * Tmp[m + i - Borda][n + j - Borda].
       Soma = Soma/FatDiv;
       Img[i][j] = ((Soma >= 0) && (Soma < 256) ? Soma : 255);
     } }
     origem.adicionarResultados(NLinhas,IniJan,Img); }
  catch (Exception e) {....}
}
```

Figura 6. Implementação da Interface do cliente

A implementação com RMI obriga o programador a criar *interfaces remotas* que serão responsáveis por manter a comunicação entre os clientes e o servidor. Após criar as interfaces remotas o desenvolvedor deverá implementá-las gerando assim um maior número de classes. Através destas interfaces e de suas implementações os stubs e skeletons serão gerados pelo compilador e, assim, os objetos remotos realização a sua comunicação. Nas figuras 3, 4, 5 e 6 apresentamos a implementação com RMI. Nas figuras 3 e 4 apresentamos as interfaces remotas do servidor e do cliente. Nas figuras 5 e 6 apresentamos a implementação destas interfaces. Pela simplicidade dos códigos, não serão apresentados os códigos dos programas principais do cliente e do servidor, que são responsáveis pela chamada dos métodos que disparam a operação de convolução.

Na figura 2 apresentamos a implementação com o JVSObjects, os métodos da biblioteca do JVSObjects estão circulados na figura. A implementação da aplicação com o JVSObjects é mais simplificada que a com RMI, pois na biblioteca do JVSObjects utilizamos o modelo de programação de variável compartilhada, enquanto no RMI o modelo utilizado para implementação de aplicações é o RPC (Remote Procedure Call). Além disso, com a biblioteca do JVSObjects não é necessário criar as interfaces e as classes remotas. Apenas é necessário que o programador utilize os métodos da biblioteca e os objetos do tipo Shared, que foram definidos nesta biblioteca.

Podemos verificar que a implementação com RMI é mais complexa, e que utiliza muitas classes e os métodos dos pacotes RMI e Threads. É importante lembrar, que a implementação com RMI, também foi realizada utilizando Threads.

4.3.2. Análise de desempenho

Na nossa análise de desempenho realizamos os testes, com as duas implementações, no SCD apresentado no ambiente de experimentação.

As nossas análises quantitativas e qualitativas, realizados com um software comercial de processamento de imagens, das imagens convoluídas (filtradas) mostraram que os resultados obtidos com as duas implementações foram iguais e estavam corretos.

Para realizar os testes, utilizamos as mesmas configurações para todas as implementações, o que nos permitiu fazer a análise comparativa das versões implementadas.

Os testes consistiram em executar todas as versões da operação de convolução de imagem implementadas, com as resoluções de imagens citadas anteriormente. Cada implementação foi executada 10 vezes para cada tamanho de imagem, para aumentar a confiança nos valores encontrados e reduzir erros (rede, inicialização de objetos na cache, etc). A média geométrica dos tempos de resposta obtidos foi utilizada como métrica para análise e comparação de desempenho das implementações.

Durante os testes, a principal métrica utilizada foi o tempo de resposta que cada implementação da convolução obteve para executar a operação para cada um dos três tamanhos de imagem e o filtro escolhido. 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 os processos/threads distribuídos.

A tabela 1 apresenta a média geométrica dos tempos de resposta obtidos com a execução das duas implementações, para as três resoluções de imagem escolhida.

Tabela 1 – Média Geométrica dos Tempos de Respostas

Modelos / Dimensão da Imagem	Versão 1 RMI	Versão 2 JVSObjetcs
512x512	67393	34164
1024x1024	263426	134623
2048x2048	1048307	528984

Tabela 2 – SpeedUp JVSObjects x RMI

Image Dimensions	512x512	1024x1024	2048x2048
SpeedUp	1,97	1,95	1,98

Comparando com a versão utilizando RMI, a versão com JVSObjects obteve um desempenho bom (tempos de resposta menores) em relação aos tempos obtidos com a versão RMI. Em nenhuma das implementações foi considerado o tempo gasto com leitura do arquivo da imagem original, pois, este tempo não fazia parte do processamento da operação; a medição dos tempos foi iniciada após o carregamento da imagem na memória principal.

Observamos que o tempo de resposta obtido com a implementação utilizando JVSObjects é menor do que o tempo de resposta obtido com a implementação usando RMI, para todas as resoluções de imagens processadas. Na tabela 2 podemos verificar que o speedup foi sempre superior a 1.9 para esta execução. O *JVSObjects Server* cria uma *thread* para cada requisição de leitura a um objeto compartilhado, realizado por um nodo cliente, por isso o tempo de resposta obtido com ele foi menor que a versão 1; leituras concorrentes. O JVSObjects foi implementado sobre o RMI, e por isto o *speedup* foi menor que o esperado; uma vez que o RMI foi utilizado somente para realizar comunicação entre os nodos e o ambiente de execução. O RMI possui serialização lenta e as threads em Java utilizam o método *sincronized* que bloqueia o objeto compartilhado até o final da operação do método sincronizado. Para este teste, o speedup esperado era de 4, pois tínhamos uma thread sendo executada em cada nodo do SCD, mas foi de 1,9 em média, isso pelo fato de o RMI ter sido usado na implementação do JVSObjects apenas para comunicação entre os nodos e o ambiente de execução e pelo overhead de comunicação encontrados nos protocolos de comunicação.

Com a utilização de threads para realizar os acessos de leitura e escrita na matriz de imagem, cada nodo possui um canal único de comunicação com o ambiente de execução, não ficando bloqueado aguardando pelo acesso. Com esta característica o JVSObjects 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.

O JVSObjects foi implementado utilizando os pacotes RMI e Threads, 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 JVSObjects.

5. Conclusão

Baseado na verificação do JVSObjects e nos resultados obtidos, concluímos que o principal objetivo do trabalho foi alcançado: apresentação do middleware de gerenciamento de memória virtualmente compartilhada em ambientes de memória fisicamente distribuída, que executa aplicações distribuídas concorrentes desenvolvidas em Java. Além disso, avaliamos a facilidade de programação (implementação de classes remotas e interfaces) e o desempenho do JVSObjects em relação a outro modelo de programação (programação distribuída com Java RMI).

O desenvolvimento de aplicações utilizando a biblioteca do JVSObjects é mais fácil do que o desenvolvimento das mesmas utilizando Java RMI. Isto acontece porque o programador pode utilizar o modelo de programação de variável compartilhada e não o modelo RPC, onde o gerenciamento dos objetos é explícito.

Na versão atual, o JVSObjects 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. Esta limitação será eliminada em versões futuras do middleware.

Com os testes realizados podemos concluir que a versão concorrente da convolução, implementada com a biblioteca do JVSObjects e executada com o ambiente de execução, obteve melhor desempenho (menor tempo de resposta) que a versão distribuída desenvolvida com o pacote RMI da linguagem Java.

A principal contribuição deste trabalho é o JVSObjects, que provê um ambiente de programação concorrente em SCD com memória virtualmente compartilhada, mas fisicamente distribuída. O JVSObjects libera o programador da preocupação com a gerência das informações compartilhadas. Outras contribuições são as implementações da convolução de imagens no modelo distribuído e a análise dos resultados obtidos com os testes das implementações.

Dentre os possíveis trabalhos futuros, já estamos desenvolvendo uma nova versão do JVSObjects com o acréscimo de outras características arquiteturais do VSObjects. Esta versão está sendo implementada utilizando *sockets*, nela, o foco é a melhoria do acesso às informações compartilhadas, uma vez que este ponto foi menos trabalhado na primeira versão do JVSObjects. Além disso, realizaremos comparações dos resultados obtidos com o JVSObjects com outras implementações usando bibliotecas de passagem de mensagem, CORBA e outros VSM desenvolvidos em Java que também foram construídos sobre *sockets*.

6. Referências

Adve S. V. and Gharachorloo K. “Shared Memory Consistency Models: a Tutorial”. Technical Report Distribution, DEC Western Research Laboratory, September 1995.

Chen X. and Allan V. H. “MultiJav: A distributed Shared Memory System Based on Multiple Java Virtual Machine”. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp 91–98, 1998.

Colouris G.; Dollimore J.; Kindberg T. “Distributed Systems: Concepts and Design”. 3rd Edition. Addison-Wesley. 2001.

CORBA services: Common Object Services Specification, Revised Edition, July 1997. OMG Document formal/97-07-04, or successor.

Especificação Java Standard 2 SDK: http://java.sun.com/j2se/1.4.2/docs/index.html, Versão 1.4.2, Abril 2004.

Gonzalez R.C. and Woods R.E., *Digital Image Processing* 3rd Edition, Ed. Edgard Blucher , New York, 2000.

Hwang K. and Xu Z., “Scalable Parallel Computing: Technology, Architecture, Programming”, McGraw-Hill, 1998.

Lawrence R., “A Survey of Cache Coherence Mechanisms in Shared Memory”, Department of Computer Science, University of Manitoba. May, 1998.

Li, K.; “Shared Virtual Memory on Loosely Coupled Multiprocessors”, Ph.D. Thesis, Department of Computer Science, Yale University, 1986.

Li, K.; Hudak, P. “Memory Coherence in Shared Virtual Memory Systems”, ACM Trans. On Computer Systems, vol. 7, pp. 321—359, 1989.

Martins C. A. P. S., "Subsistema de exibição de imagens digitais com desacoplamento de resolução - SEID-DR", Tese de Doutorado, Universidade de São Paulo, SP, 1998.

Martins, C. A. P. S., Zuffo J., and Kofuji S., “Two Dimensional Normalized Sampled Finite Sinc Reconstructor”, inAeroSense‘97, Proc. SPIE-3074, SPIE, Orlando, pp. 240-250, 1997.

Morazanm T. M. and Troeger D. R., “A Case Study of List Memory Paging in a Distributed Virtual Memory System for Functional Languages”. V Brazilian Symposium on Programming Languages, SBLP 2001.

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 em Sistemas Computacionais de Alto Desempenho (WSCAD), pp 118-125. Vitória, Brazil, 2002.

Pousa, C. V, Penha, D. O., Martins, C. A. P. S. “VSObjects: Middleware para Gerenciamento de Objetos Virtualmente Compartilhados”. IV Workshop em Sistemas Computacionais de Alto Desempenho, pp. 41-48, WSCAD – 2003.

Sun Microsystems. JavaSpace *specification*. available at: http://java.sun.com/, 1999.

Tanenbaum A. and Woodhull A., “Operating Systems Design and Implementation”, 2.ed., Editora Prentice Hall, Upper Saddle River, New Jersey, 1997.

Thiruvathukal G. K., Dickens P. M., Bhatti S., “Java on Networks of Workstations (JavaNOW): A Parallel Computing Framework Inspired by Linda and the Message Passing Interface (MPI)”, Concurrency : Practice and Experience Vol. 12, pp. 1093-1116 ,1998.