ANTONIO CARLOS DE NAZARÉ JÚNIOR

Orientador: Joubert de Castro Lima

JAVA CÁ & LÁ - UM MIDDLEWARE PARA COMPUTAÇÃO PARALELA

Ouro Preto

Dezembro de 2011

Universidade Federal de Ouro Preto
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

JAVA CÁ & LÁ - UM MIDDLEWARE PARA COMPUTAÇÃO PARALELA

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

ANTONIO CARLOS DE NAZARÉ JÚNIOR

Ouro Preto
Dezembro de 2011

UNIVERSIDADE FEDERAL DE OURO PRETO

FOLHA DE APROVAÇÃO

Java Cá & Lá - Um Middleware para Computação Paralela

ANTONIO CARLOS DE NAZARÉ JÚNIOR

Monografia defendida e aprovada pela banca examinadora constituída por:

Dr. Joubert de Castro Lima – Orientador
Universidade Federal de Ouro Preto

Dr. Fabrício Benevenuto de Souza
Universidade Federal de Ouro Preto

Dr. Ricardo Augusto Rabelo
Universidade Federal de Ouro Preto

Dr. Tiago Garcia de Senna Carneiro
Universidade Federal de Ouro Preto

Ouro Preto, Dezembro de 2011

Resumo

Com o avanço dos recursos computacionais nas últimas décadas, surgiu uma nova área: **Computação Paralela**. Nesta área vários processadores são utilizados simultaneamente para executar uma única aplicação gastando-se menos tempo. Do ponto de vista do programador, esse tipo de aplicação pode ser projetado através de dois modelos principais: **Modelo de Memória Compartilhada** (*Shared Memory*) e **Modelo de Memória Distribuída** (*Distributed Memory*).

Os benefícios do paralelismo juntamente com a demanda de aumento do poder computacional das aplicações apresentam argumentos convincentes em favor da computação paralela. Porém, programas de computador paralelos são mais difíceis de programar do que sequenciais, pois o paralelismo introduz diversas novas classes de problemas potenciais, como a comunicação e a sincronização de recursos. Assim, o desenvolvimento de aplicações paralelas não tem acompanhado a evolução dos sistemas paralelos e distribuídos, pois constitui uma tarefa árdua e pouco produtiva. Portanto, é de grande ajuda uma ferramenta para facilitar o desenvolvimento de aplicações paralelas à partir de um único conjunto de primitivas, seja em modelos de memória compartilhada ou de memória distribuída.

O presente trabalho propõe e implementa um *middleware* para dar suporte à programação paralela. Os esforços foram realizados de maneira que esse sistema fosse o mais eficiente possível, oferecendo suporte à programação para ambos modelos de memória. Tal *middleware* é chamado de **Java Cá & Lá**.

Experimentos computacionais demonstraram que o **Java Cá & Lá** apresenta um bom comportamento quando utilizado de maneira correta, assim como um bom desempenho nos testes de execuções de aplicações paralelas.

Abstract

With the progress of computing resources in the recent decades, a new area arose: Parallel Computing. In this area several processors are used simultaneously to run a single application spending less time. From a programmer standpoint, this application can be designed through two main models: Shared Memory e Distributed Memory.

The benefits of parallelism along with the increased demand for computing power of applications present convincing arguments in favor of parallel computing. However, parallel computing programs are harder to program than sequential programs, because parallelism introduces several new classes of potential problems such as communication and synchronization of resources.

Thus, the development of parallel applications has not kept pace the evolution of the parallel and distributed systems, since it is an arduous task and not very productive. Therefore, it is a helpful tool to facilitate the development of parallel applications from a single set of primitive, either in models of shared memory or distributed memory.

The present study proposes and implements a middleware to support the parallel programming. Efforts were made in manner that this system be as efficient as possible, providing support for programming models for shared memory and distributed memory. This middleware is called **Java Cá & Lá**.

Computational experiments have shown that **Java Cá & Lá** has a good performance when used correctly, as well a good performance in tests of executions of parallel applications.

Dedico este trabalho aos meus pais, Antonio Carlos de Nazaré e Inês Conceição Reis de Nazaré, os maiores exemplos de minha vida.

Agradecimentos

Primeiramente agradeço à Deus por guiar meus passos. Aos meus pais, maiores exemplos da minha vida, pelo suporte e incentivo. À Fernanda por todo carinho e atenção. Ao tio Toninho por sempre estar ao meu lado. À minha irmã que eu amo muito. Aos meus primos pelos conselhos. Às minhas avós pelas orações. À turma 08.1 e toda galera da computação, em especial Victor, Suellem, Pedrinho e Saulo. À UFOP, em especial à PRACE, pelo suporte. Aos professores do Departamento de Computação da UFOP por todo o ensinamento. Ao David Menotti, por ter sido um amigo, um irmão e um segundo pai. À eterna república FAVELA que com seus moradores e ex-alunos, que ao estarem sempre ao meu lado, me proporcionaram muitas alegrias e ensinamentos. E por fim agradeço ao Joubert, por todos os conselhos, orientações e incentivos nas horas mais difíceis.

Obrigado a todos que confiaram em mim e fico muito mais agradecido àqueles que não confiaram.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 1

Introdução

Durante as últimas décadas houve grandes avanços nos microprocessadores. Os processadores aumentaram sua velocidade de *clock* de cerca de 40 MHz (*e.g. MIPS R3000* no ano de 1988) para mais de 2.0 GHz (por exemplo, um *Pentium Extreme Edition*, do ano de 2005). Ao mesmo tempo que a velocidade aumentou, os processadores tornaram-se capazes de executar múltiplas instruções no mesmo ciclo de *clock*. Estas evoluções se traduzem no aumento do número de operações de ponto flutuante por segundo, *FLOPS* (*Floating point Operations Per Second*).

A velocidade global dos sistemas de computação é determinada não apenas pela a velocidade do processador, mas também pela capacidade do sistema de memória de enviar dados ao processador. Enquanto as taxas de *clock* dos processadores têm aumentado em cerca de 40% ao ano na última década, tempos de acesso à memória têm melhorado a uma taxa de apenas cerca de 10% por ano durante este intervalo. Juntamente com o aumento no número de *FLOPS*, este fosso entre a velocidade do processador e a memória apresenta um gargalo de desempenho tremendo (Grama et al., 2002).

Com o avanço dos recursos computacionais, surgiu uma nova área: **Computação Paralela**. Nesta área vários processadores são utilizados simultaneamente para executar uma única aplicação gastando-se menos tempo. Do ponto de vista do programador, esse tipo de aplicação pode ser projetado através de dois modelos principais: **Modelo de Memória Compartilhada** (*Shared Memory*) e **Modelo de Memória Distribuída** (*Distributed Memory*).

O modelo de memória compartilhada é direcionado para arquiteturas nas quais múltiplos processadores compartilham um único espaço de memória. A comunicação entre os processadores nesse modelo é realizada por meio de operações de leitura e escrita de dados nesse espaço de memória.

No modelo de memória distribuída, processadores não compartilham memória. Ao invés disso, eles enviam e recebem mensagens através da rede de interconexão. Todos os dados são privados e a única forma de um processador obter uma informação que não está na sua memória local é requisitando-a ao processador que a possui, por meio de uma mensagem.

Existe um terceiro modelo chamado de **Modelo de Memória Compartilhada Distribuída** (*Distributed Shared Memory*, *DSM*) que consiste em uma coleção de computadores paralelos independentes conectados por uma rede de interconexão. O *DSM* tenta combinar as vantagens dos dois modelos de memória apresentados anteriormente.

O papel da execução paralela de processos no aumento da performance computacional tem sido reconhecido por várias décadas. O seu auxílio na diminuição de tempo de processamento, maior utilização dos recursos computacionais, maior acesso aos elementos de armazenamento (memória e disco) e redução de custos reflete na grande variedade de aplicações da computação paralela.

Os computadores pessoais, as estações de trabalho e servidores com dois, quatro ou mais processadores conectados (*multicores*) estão se tornando comuns no dia a dia do usuário. Aplicações na engenharia e outras áreas, que requerem alto poder computacional, necessitam de configurações maiores de computadores paralelos, geralmente compostos por centenas de processadores. Plataforma de processamento de dados em grande escala (*e.g.* banco de dados, servidores *web* e *data mining*) utilizam, na maioria das vezes, *clusters* de estações de trabalho que oferecem maior largura de banda de disco (*disk bandwith*). Aplicações que requerem alta disponibilidade dependem de plataformas paralelas para a redundância dos recursos. Pode-se citar vários outros exemplos da utilização de computação paralela na ciência e na indústria como: Otimização de processos, Computação Gráfica, *Business Intelligence*, Processamento de Imagens Médicas, Sequenciamento do Genoma Humano, etc.

Em Grama et al. (2002) são citados três fortes argumentos à favor do uso de computação paralela, são eles:

Poder Computacional A Lei de *Moore*[1] tem sido alvo de amplo debate nos últimos anos. É possível fabricar dispositivos com o número de transistores muito grande, entretanto a questão é como converter esses transistores em poder computacional, ou seja, utilizá-los para conseguir um aumento das taxas de computação é o principal desafio arquitetônico. Um recurso para isso é confiar em paralelismo.

Velocidade do Sistema de Memória e Disco Plataformas paralelas, na maioria das ve-

[1] Em 19 de abril de 1965 Gordon Moore fez a seguinte citação (Moore, 1998):

Lei 1.1 (Moore) *A complexidade para componentes com custos mínimos tem aumentado em uma taxa de aproximadamente um fator de dois por ano... Certamente em um curto prazo pode-se esperar que esta taxa se mantenha, se não aumentar. A longo prazo, a taxa de aumento é um pouco mais incerta, embora não haja razões para se acreditar que ela não se manterá quase constante por pelo menos 10 anos. Isso significa que em torno de 1975, o número de componentes por circuito integrado para um custo mínimo será 65.000. Eu acredito que circuitos grandes como este poderão ser construídos em um único componente (pastilha).*

Em 1975, Moore revisou a sua previsão para, a cada dois anos, um aumento de 100% na quantidade de transistores dos *chips* mantendo seu custo. Originalmente, a Lei de Moore não falava nada sobre o desempenho, mas apenas sobre o número de transistores em processadores, módulos de memória e outros circuitos.

zes, produzem um melhor desempenho para os sistemas de memória por fornecerem (*i*) melhor acesso à memória cache; e (*ii*) melhor utilização da largura de banda da memória. Isso é possível, pois o paralelismo permite efetuar acessos à memória enquanto se realiza alguma computação com o processador, desde que o dado seja acessado em um momento anterior (*e.g. pipeline*). Este argumento também pode ser estendido à utilização de discos maximizando a largura de banda da transmissão de dados do disco para o processador.

Comunicação de Dados Em muitas aplicações, existem restrições sobre a localização dos dados e/ou recursos em toda a rede. Um exemplo de tal aplicação é a mineração de grandes conjuntos de dados comerciais distribuídos por uma rede de banda relativamente baixa. Em tais aplicações, mesmo se o poder de computação disponível para realizar a tarefa desejada for alto, sem recorrer à computação paralela, é inviável coletar os dados em um local centralizado. Nestes casos, a motivação para o paralelismo não vem apenas da necessidade de recursos de computação, mas também da inviabilidade ou inconveniência de acessos centralizados.

1.1 Definição do Problema

Os benefícios do paralelismo juntamente com a demanda de aumento do poder computacional das aplicações apresentam argumentos convincentes em favor da computação paralela. Porém, programas de computador paralelos são mais difíceis de programar do que sequenciais, pois o paralelismo introduz diversas novas classes de problemas potenciais, como a comunicação e a sincronização de recursos (Patterson e Hennessy, 2008). Assim, o desenvolvimento de aplicações paralelas não tem acompanhado a evolução dos sistemas paralelos e distribuídos, pois constitui uma tarefa árdua e pouco produtiva.

Segundo Buhr et al. (1996), os erros concorrentes ocorrem com frequência decrescente na seguinte ordem: erros sequenciais tradicionais, erros no projeto de algoritmos, *deadlocks*[2] e condições de corrida (*race conditions*[3]). Entretanto, a dificuldade em encontrar e corrigir estes erros cresce exponencialmente.

É necessário um conhecimento, por parte do programador, de conceitos de Linguagem de Programação, Arquitetura de Computadores, Redes de Computadores e Sistemas Distribuídos. Isso, na maioria das vezes, afugenta o programador forçando-o a desenvolver soluções sequencias ou soluções paralelas ineficientes.

[2]Situação onde um processo entra em estado de espera por um evento que deveria ser gerado por um outro processo que está neste mesmo estado.

[3]Situação onde uma mensagem inesperada chega ao seu destino antes da mensagem correta, causando um erro. O atraso pode ser causado por diversos fatores independentes do programa, como sobrecarga do processador ou da rede de comunicação.

Diante das dificuldades apresentadas, é de grande ajuda uma ferramenta para facilitar o desenvolvimento de aplicações paralelas à partir de um único conjunto de primitivas, seja em modelos de memória compartilhada ou de memória distribuída.

1.2 Objetivo

O presente trabalho propõe e implementa um *middleware* para dar suporte à programação paralela. Os esforços foram realizados de maneira que esse sistema fosse o mais eficiente possível, oferecendo suporte à programação para modelos de memória compartilhada e memória distribuída. Tal *middleware* é chamado de **Java Cá & Lá**.

Resumidamente, o papel do *middleware* **Java Cá & Lá** é fornecer dois serviços básicos. (*i*)Execução de tarefas (*Tasks*) de forma paralela; (*ii*)acesso concorrente a variáveis compartilhadas, denominadas variáveis globais. Uma *Task* nada mais é que um método, com ou sem argumentos, programado pelo usuário. Os métodos são programados de forma natural, sendo necessário atender à restrições impostas pelo *middleware*, conforme o orientado na Seção 3.4.

Um *middleware* é uma camada adicional de *software* situada entre o nível de aplicação e o nível que consiste no sistema operacional. O *middleware* tem o papel de interligar diferentes aplicações em diferentes sistemas operacionais em diferentes computadores. Ou seja, ele oculta da melhor maneira possível a heterogeneidade das plataformas das aplicações. Por ser um *software* de conectividade, consiste de um conjunto de serviços disponíveis que permite que múltiplos processos, executando em uma ou mais máquinas, interajam através de uma rede.

A solução proposta apresenta as seguintes características:

- Execução em ambientes de memória compartilhada, memória distribuída ou memória compartilhada distribuída, à partir de um único modelo de programação simples e eficiente;
- Acesso transparente à variáveis compartilhadas;
- Mecanismos de sincronização;
- Estratégias para minimizar a troca de mensagens;
- Políticas de escalonamento das tarefas, de maneira que seja explorado todo o potencial da estrutura computacional disponível;
- Transparência de concorrência, localização, migração e à falhas;
- Auditoria por meio de *logging*;

1.3 Organização da Monografia

O Capítulo 1 realiza uma breve descrição do cenário da computação paralela, além de apresentar o problema e a motivação necessária para alcançar os objetivos pretendidos.

O Capítulo 2 faz uma revisão de conceitos sobre computação paralela, abordando os modelos de arquitetura de computadores e memórias, comunicação, desempenho e programação de aplicações paralelas.

Os detalhes da arquitetura e o desenvolvimento do *middleware* são explicados no Capítulo 3.

No Capítulo 4 são apresentados a elaboração e os resultados dos testes e os mesmos são analisados para se caracterizar o comportamento do *middleware* e verificar a validade das estratégias propostas e implementadas.

Por fim, as conclusões são apresentadas no Capítulo 5.

Capítulo 2

Referencial Teórico

2.1 Computação Paralela

Segundo Cerqueira (2002) os sistemas de computação paralela e distribuída são compostos por vários processadores que operam concorrentemente, cooperando na execução de uma determinada tarefa. Estes sistemas são necessários quando as tarefas exigem um alto poder computacional e um melhor desempenho na solução dos problemas.

O uso de sistemas de computação paralela e distribuída se faz necessário quando os problemas excedem os limites físicos e algorítmicos da computação sequencial. Com o uso das arquiteturas paralelas objetiva-se aumentar a capacidade de processamento de dados, utilizando o potencial oferecido por um grande número de computadores e processadores.

O Paralelismo pode ser classificados em três níveis distintos (Silva, 2006), apresentados a seguir.

2.1.1 Paralelismo de Serviços (*Jobs*)

É o nível mais alto de paralelismo e é de interesse maior para administradores de sistema do que de usuários. Nesse tipo de paralelismo, o mais importante é que um laboratório ou centro de comunicação execute uma maior quantidade de *jobs* possíveis em um período de tempo específico. Isso pode ser alcançado, adquirindo-se mais máquinas de forma que uma maior quantidade de *jobs* seja executada ao mesmo tempo, apesar de para o usuário um *job* particular não executar mais rápido.

2.1.2 Paralelismo da Aplicação

Ocorre quando um programa simples é dividido em partes que são executadas ao mesmo tempo em múltiplos processadores ou múltiplas unidades funcionais. O Paralelismo em nível de programa geralmente se manifesta de duas formas: sobre seções independentes de um

mesmo programa; ou sobre iterações individuais de um laço onde não há dependência entre dados.

2.1.3 Paralelismo de Instruções

É invisível para os usuários e está no nível de organização de computadores. *Pipelines* são a forma mais comum de alcançar esse tipo de paralelismo (Patterson e Hennessy, 2008). Nesse caso, instruções podem ser sobrepostas ou uma determinada instrução pode ser decomposta em sub-operações e essas sub-operações serem sobrepostas. Em geral, programadores não precisam se preocupar com esse nível de paralelismo já que compiladores são capazes de organizar programas para explorá-lo.

2.2 Tipos de Paralelismo

No nível da aplicação existem dois tipos de paralelismo: o paralelismo de dados e o paralelismo de controle (Cerqueira, 2002).

Paralelismo de Dados É o aproveitamento da concorrência que deriva da aplicação da mesma operação a múltiplos elementos de uma estrutura de dados. Teoricamente k unidades processadoras produzem um aumento de vazão (número de resultados produzidos por unidade de tempo)de k vezes no sistema. Neste tipo de paralelismo os processadores executam as mesmas instruções sobre dados diferentes. Um exemplo de aplicação deste tipo de paralelismo é a resolução de sistemas lineares e multiplicação de matrizes.

Paralelismo de Controle Diferentes operações são aplicadas a diferentes porções de dados simultaneamente. O fluxo de dados sobre estes processos pode ser arbitrariamente complexo.

No desenvolvimento de algoritmos paralelos, deve-se procurar escolher o tipo de paralelismo que mais se adapte à natureza do problema. Muitos problemas reais também podem explorar ambos os tipos de paralelismo.

2.3 Modelos de Arquitetura Paralela e Distribuída

Com o aparecimento de várias arquiteturas de computação sequencias e paralelas surgiu a necessidade de classificá-las. Em Flynn (1972) é apresentado uma classificação de tais arqui-

teturas baseando se nos conceitos de **fluxo de instrução**[1] e **fluxo de dados**[2]. Segunda a classificação, denominada **Taxonomia de Flynn**, existem quatro modelos de arquiteturas: *SISD*, *SIMD*, *MISD* e *MIMD*. A seguir são apresentados, de forma resumida, as principais características de cada modelo.

2.3.1 *SISD - Single Instruction Single Data*

Este modelo de arquitetura é baseado nos computadores sequenciais convencionais, com o programa armazenado em memória. Nesta classe, estão incluídas todas as arquiteturas que, em cada instante, estão executando apenas um fluxo de instruções e um fluxo de dados, realizando uma operação de cada vez. É o modelo de arquitetura mais utilizado. A Figura 2.1 ilustra o modelo *SISD*.

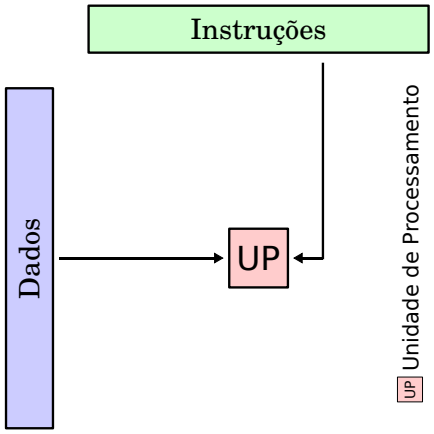

Figura 2.1: Ilustração do Modelo de Arquitetura *SISD*

2.3.2 *SIMD - Single Instruction Multiple Data*

Descreve um método de operação de computadores com várias unidades de processamento em computação paralela. Neste modo, a mesma instrução é aplicada simultaneamente a diversos dados para produzir mais resultados. O modelo *SIMD*, ilustrado pela Figura 2.2 é adequado para o tratamento de conjuntos regulares de dados, como as matrizes e vetores.

[1] De acordo com Tanenbaum e Goodman (1998), um fluxo de instruções corresponde a um contador de programa, um sistema com n CPU's possui n contadores de programa, e então n fluxos de instruções.

[2] Um fluxo de dados consiste em um conjunto de operandos, por exemplo, um programa que calcula a média das notas de uma lista de alunos possui um fluxo de dados, um programa que calcula a média das temperaturas de n termômetros espalhados pelo mundo possui n fluxos de dados (Tanenbaum e Goodman, 1998).

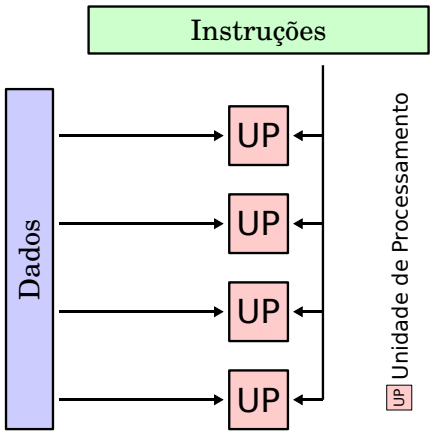

Figura 2.2: Ilustração do Modelo de Arquitetura *SIMD*

2.3.3 *MISD - Multiple Instruction Single Data*

É um tipo de arquitetura de computação paralela, onde muitas unidades funcionais executam operações diferentes sobre os mesmos dados. Arquiteturas *pipeline* pertencem a este tipo, apesar que pode-se considerar que os dados são diferentes após o processamento por cada fase do *pipeline*. Não há muitos exemplos da existência deste modelo de arquitetura, ao contrário do *SIMD* e do *MIMD*. O modelo é ilustrado pela Figura 2.3

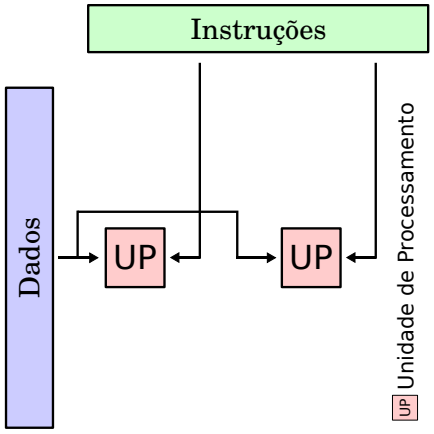

Figura 2.3: Ilustração do Modelo de Arquitetura *MISD*

2.3.4 *MIMD - Multiple Instruction Multiple Data*

Neste modelo de arquitetura estão incluídas todas as máquinas que, tendo várias unidades processadoras, permitem, em cada instante, a execução de múltiplas instruções diferentes sobre

outros múltiplos conjunto de dados. A maioria das arquiteturas paralelas estão classificados na categoria *MIMD*, como por exemplo os supercomputadores atuais e computadores pessoais com processadores *multicores*. A seguir o modelo é ilustrado na Figura 2.4.

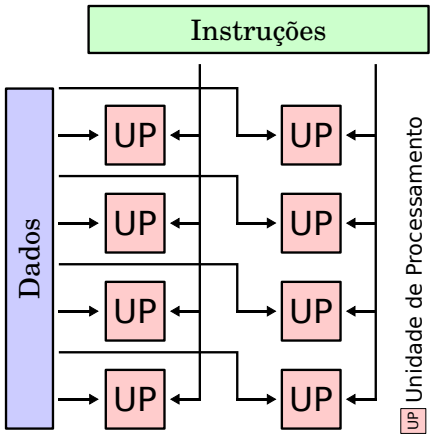

Figura 2.4: Ilustração do Modelo de Arquitetura *MIMD*

Existe uma grande variedade de arquiteturas *MIMD* e especialmente nessa categoria, a classificação de *Flynn* se mostra inadequada, pois inclui arquiteturas totalmente distintas. A categoria das arquiteturas *MIMD* pode ser subdividida em dois outros grupos: Sistemas de Memória Compartilhada e Sistemas de Memória Distribuída, detalhadas na Seção 2.4.

2.4 Modelos de Arquitetura de Memória

2.4.1 Memória Compartilhada (*Shared Memory*)

Neste modelo, também conhecido como *Multiprocessadores*, os processadores compartilham fisicamente uma única memória centralizada e eles são interconectados por um barramento. Com caches grandes, o barramento e a memória única podem satisfazer as demandas de memória de um número pequeno de processadores. Pelo fato de existir uma única memória principal que tem um relacionamento simétrico com todos os processadores e um tempo de acesso uniforme a partir de qualquer processador, esses multiprocessadores frequentemente são chamados de multiprocessadores simétricos. Neste modelo todos os processadores podem acessar toda a memória como uma área de endereçamento global. Eles pode operar independentemente, mas compartilham os mesmos recursos da memória. Assim mudanças numa alocação de memória realizada por um processador são visíveis a todos os demais processadores (Quinn, 1994).

As vantagens e desvantagens deste modelo são apresentados na Tabela 2.1.

As máquinas com memoria compartilhada podem ser classificadas quanto a distância dos processadores à memória, e quanto aos esquemas de coerência de cache, separando-se assim

Vantagens
• Endereçamento global permite uma perspectiva de programação mais amigável; • O compartilhamento de dados entre tarefas paralelas é mais rápido e uniforme devido à proximidade entre memória e *CPU's*;
Desvantagens
• Falta de escalabilidade entre memória e *CPU's*. Adicionar mais *CPU's* pode aumentar a contenção entre *CPU* e memória. Considerando coerência de cache, aumentamos também a contenção entre cache e memória; • O programador é responsável pela sincronização que garante o acesso correto à memória; • À medida que o número de processadores aumenta, torna-se difícil e de alto custo o projeto de máquinas com memória compartilhada;

Tabela 2.1: Vantagens e desvantagens do modelo de memória compartilhada.

em duas classes: *UMA* e *NUMA*, apresentadas a seguir:

2.4.1.1 UMA - Uniform Memory Access

Neste tipo de máquina, o tempo para o acesso aos dados na memória é o mesmo para todos os processadores a para todas as posições da memória. Essas arquiteturas também são chamadas de *SMP* (*Symmetric MultiProcessor*). A forma de interconexão mais comum neste tipo de máquina é o barramento e a memória geralmente é implementada com um único módulo. O principal problema com este tipo de arranjo é que o barramento e a memória tornam-se gargalos para o sistema, que fica limitado a uma única transferência por vez. A seguir a Figura 2.5

Na Figura 2.5 são ilustradas as memórias cache de cada processador. Tais memórias são utilizadas para esconder a latência no acesso à memória principal e para diminuir o tráfego no barramento. Como várias cópias de um mesmo dado podem ser manipuladas simultaneamente nas caches de vários processadores, é necessário que se garanta que os processadores sempre acessem a cópia mais recente. Esta garantia é chamada de coerência de cache (*cache coherence*), e máquinas *UMA* geralmente lidam com este problema diretamente em hardware. Um dos protocolos de coerência de cache mais populares é chamado de *snooping*, ou 'bisbilhoteiro'. Neste caso, quando um dado compartilhado por várias caches é alterado por algum processador, todas as demais cópias são invalidadas ou então atualizadas.

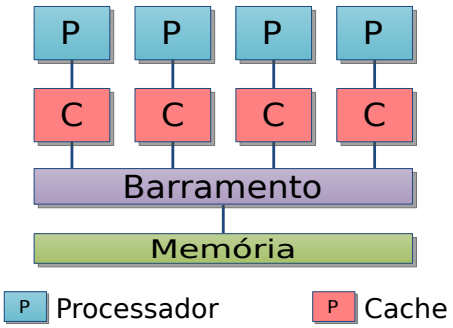

Figura 2.5: Arquitetura de memória compartilhada do tipo *UMA*

2.4.1.2 NUMA - Non-Uniform Memory Access

Neste tipo de arquitetura de memória compartilhada, a memória geralmente é dividida e portanto implementada com múltiplos módulos. Cada processador está associado a um módulo, mas o acesso aos módulos ligados a outro processador é possível. O espaço de endereçamento é comum a todos os processadores e a latência para ler ou escrever na memória pertencente a um outro processador é maior que a latência para o acesso à memória local. É constituído de duas ou mais *SMP's*, conforme o exemplo da figura 2.6. As máquinas *NUMA* também estão sujeitas aos problemas de coerência de cache, e conforme a solução implementada existem variações deste tipo de arquitetura.

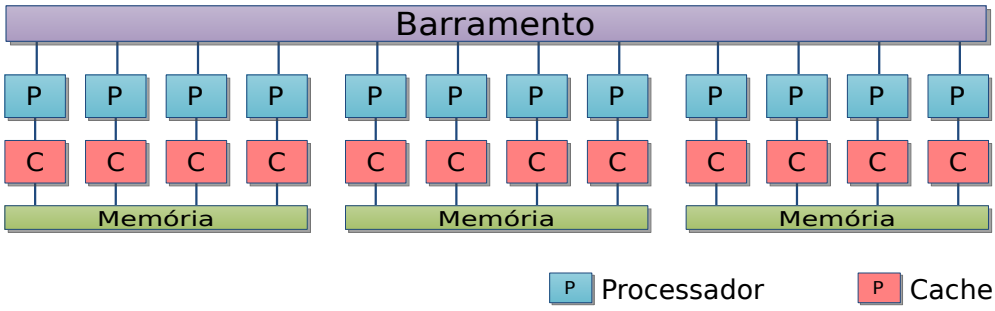

Figura 2.6: Arquitetura de memória compartilhada do tipo *NUMA*

2.4.2 Memória Distribuída - *Distributed Memory*

Uma arquitetura de memória distribuída, denominada também como **multicomputadores**, consiste em múltiplos nodos de processamento independentes com módulos de memória privativos, conectados por uma rede de interconexão. Esses nodos de processamento caracterizam-se pelo fato de que cada processador enxerga somente a sua própria memória. Para a troca de

mensagens e dados é preciso o envio de requisições através da rede de interconexão. Os multicomputadores também são chamados de sistemas de troca de mensagens (*message passing systems*). Com estas características, este modelo pode ser implementado por meio de um conjunto de máquinas autônomas, ou seja, computadores tradicionais. A escalabilidade natural destes sistemas permitem o desenvolvimento de aplicações com um poder de computação muito alto. A seguir a figura 2.7 ilustra o modelo de arquitetura de memória distribuída.

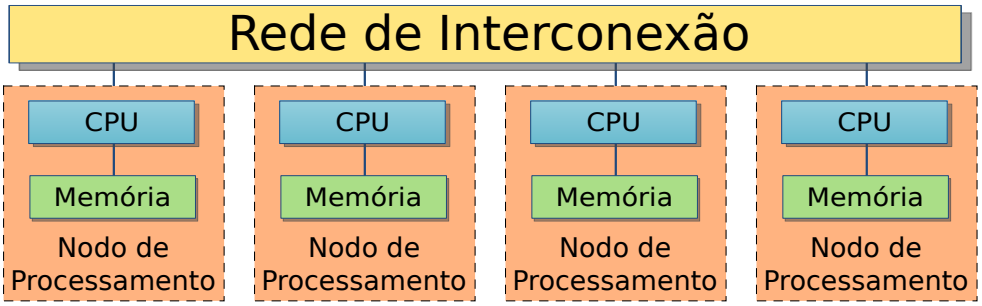

Figura 2.7: Modelo de arquitetura de memória distribuída

As vantagens e desvantagens deste modelo são apresentados na Tabela 2.2.

Vantagens
• Baixo custo, pois pode ser implementado por meio de computadores convencionais. • Permite uma boa escalabilidade; • Cada nodo processador pode acessar rapidamente sua memória local (sem os problemas causados pela coerência de cache);
Desvantagens
• Cabe ao programador detalhes como comunicação entre os nodos de processamento e sincronização do acesso aos dados; • A sobrecarga da rede de interconexão (*overhead de rede*) pode causar a degradação do desempenho do sistema a ser executado; • A interrupção da comunicação entre os nodos de processamento, decorrente de uma eventual falha no barramento, pode causar a paralisação parcial ou total do sistema.

Tabela 2.2: Vantagens e desvantagens do modelo de memória distribuída.

2.5 Métricas de Desempenho em Computação Paralela

Segundo Grama et al. (2002) existem duas classes distintas de métricas de desempenho:

Métricas de Desempenho para Processadores métricas que permitem avaliar a performance de um processador tendo por base a velocidade/número de operações que este consegue realizar num determinado espaço temporal.

Métricas de Desempenho para Aplicações Paralelas métricas que permitem avaliar a performance de uma aplicação paralela tendo por base a comparação entre a execução com múltiplos processadores e a execução com um só processador.

Seguindo o foco deste trabalho, a seguir são apresentadas métricas de desempenho para aplicações paralelas.

2.5.1 Tempo de Execução

O tempo de execução é a medida de desempenho mais confiável e que melhor traduz o que se busca em termos de velocidade de processamento, tanto do hardware quanto do software (Silva, 2006). Nesse sentido uma relação do desempenho (D) em relação ao tempo de execução (T) pode ser formulada conforme a Equação 2.1

$$D = \frac{1}{T} \tag{2.1}$$

O tempo de execução pode ser visto de duas maneiras: tempo decorrido desde o início até o final da execução do programa ou o tempo de *CPU*, isto é, o tempo que efetivamente foi utilizado pela *CPU* para executar o programa, excluindo-se o tempo consumido pelo próprio sistema operacional durante a execução do programa. O normal é a utilização do tempo decorrido ou do programa como um todo, ou parte dele. Pois o tempo de *CPU* também não contabiliza o tempo de comunicação, que é adicionado para se utilizar as máquinas paralelas.

O tempo total de execução pode ser decomposto em tempo de serviço, tempo de comunicação e tempo de espera (Culler et al., 1997).

Tempo de Serviço É o tempo consumido realizando algum trabalho. O tempo de serviço geralmente vai depender de alguma forma, do tamanho do problema. Se o algoritmo paralelo replica computação, então o tempo de computação também dependerá do número de tarefas ou processadores.

Tempo de Comunicação É o tempo que suas tarefas gastam enviando e recebendo mensagens. Existem dois tipos básicos de comunicação: interprocessos e intraprocessos. Na comunicação interprocessos, as duas tarefas comunicantes estão localizadas em processos diferentes. Na comunicação intraprocessos, duas tarefas comunicantes estão localizadas no mesmo processo.

Tempo de Espera É o tempo no qual o serviço (*job*) está em espera devida à falta de recursos computacionais ou a falta de dados.

Ambos, tempo de serviço e tempo de comunicação são especificados explicitamente em um algoritmo paralelo. Assim, é mais fácil determinar suas contribuições para o tempo de execução. O tempo de espera pode ser mais difícil de se determinar, pois na maioria das vezes depende da ordem na qual as operações são apresentadas.

2.5.2 *Speedup*

O *Speedup* (S) é definido como a razão entre o tempo (T_s) em que um computador paralelo executa o algoritmo sequencial mais eficiente (com apenas um processador) e o tempo (T_p) em que o mesmo computador paralelo executa o correspondente algoritmo paralelo usando p processadores, ambos sobre a mesma computação conforme a equação 2.2. O *Speedup* evidencia o ganho de tempo obtido na execução paralela para um dado número de tarefas concorrentes.

$$S = \frac{T_s}{T_p} \tag{2.2}$$

O *Speedup* ideal a ser alcançado, isto é, o máximo ganho obtido com a paralelização, deveria assintotar ao número de processadores utilizados,apresentando-se assim um comportamento linear. Entretanto na prática ocorre um comportamento típico semelhante ao apresentado na Figura 2.8. O *Speedup* não é diretamente proporcional ao número de processadores utilizados. Contudo, Quinn (1994) afirma que um *Speedup* superlinear é possível desde que a escolha do algoritmo seja feita antes da instância do problema e que o algoritmo paralelo trate alguns casos específicos que o sequencial trata de modo genérico. A seguir são apresentados alguns dos fatores que podem tornar o *Speedup* em superlinear:

- Custos de comunicação/sincronização/iniciação praticamente inexistentes;
- Tolerância à latência da comunicação;
- Aumento da capacidade de memória (o problema passa a caber todo em memória);
- Subdivisão do problema;
- Aleatoriedade da computação em problemas de otimização ou com múltiplas soluções;

2.5.3 Eficiência

A eficiência (E) é a razão entre o valor do Speedup (S) e o número de processadores (p) utilizados na execução paralela, conforme a Equação 2.3. Assim, a eficiência mostra se os ganhos obtidos com a adição de máquinas estão sendo relevantes de forma a determinar a

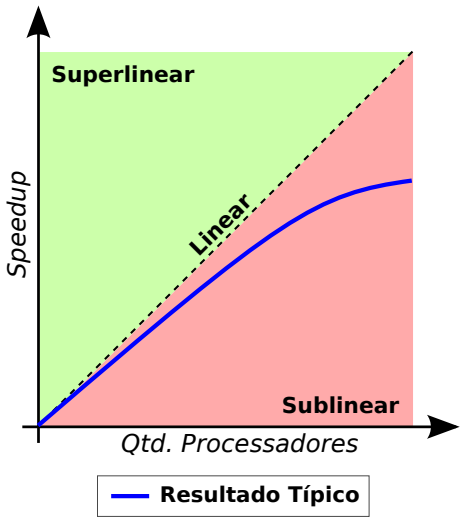

Figura 2.8: Comportamento do *Speedup*

quantidade ótima de máquinas necessárias para a execução paralela de um dado tipo de problema e volume de dados (Foster, 1995).

$$E = \frac{S}{p} \tag{2.3}$$

2.5.4 Lei de Amdahl e Lei de Gustafson-Barsis

Em Amdahl (1967) é formulado o que hoje é chamado de a **Lei de Amdahl** para caracterizar a maneira como uma aplicação poderia utilizar de forma eficiente processadores paralelos.

Lei 2.1 (Amdahl) *Seja f a fração de uma computação que deve ser executada sequencialmente, onde $f \leq f \leq 1$. O Speedup máximo (S) alcançado por um computador executando uma computação com p processadores é dado por:*

$$S \leq \frac{1}{f + \frac{(1-f)}{p}}$$

Por meio da Lei de Amdahl pode-se determinar dois limites de *Speedup*:

1. O *Speedup* máximo que pode-se obter ao resolver um determinado problema com p processadores.
2. O *Speedup* que uma determinada aplicação poderá alcançar independentemente do número de processadores a serem utilizados. Bastando para isso resolver a Equação 2.4:

$$\lim_{p \to \infty} \frac{1}{f + \frac{(1-f)}{p}} \tag{2.4}$$

Portanto a Lei de Amdahl impõe que uma fração de operações sequenciais, mesmo que em números pequenos, pode significativamente limitar o *Speedup* alcançado por um computador paralelo.

Por exemplo, suponha que pretende-se determinar se é vantajoso desenvolver uma versão paralela de uma determinada aplicação sequencial em um computador com 8 processadores. Empiricamente, verificou-se que 90% do tempo de execução é passado em procedimentos que se julga ser possível paralelizar, ou seja a fração (f) de computação sequencial será de $0,1$. Assim o cálculo do *Speedup*(S_1) máximo para $p = 8$ é dada pela Equação 2.5.

$$S_1 = \frac{1}{f + \frac{(1-f)}{p}} \Longrightarrow S_1 = \frac{1}{0,1 + \frac{(1-0,1)}{8}} \approx 4,71 \tag{2.5}$$

Também é possível calcular o *Speedup* (S_2) máximo, para o exemplo anterior, independente do número de processadores, conforme a Equação 2.6.

$$S_2 = \lim_{p \to \infty} \frac{1}{0,1 + \frac{(1-0,1)}{p}} = 10 \tag{2.6}$$

Entretanto, essa lei não contempla certas aplicações paralelas, onde a fração sequencial do algoritmo é reduzida à medida que o tamanho do problema aumenta. Pois a lei assume que a fração sequencial é fixa, o que não é verdade. E como pode ser visto na Figura 2.9, em geral, quando o tamanho (n) do problema cresce a fração (f) de operações sequenciais decresce, tornando o problema mais viável à paralelização.

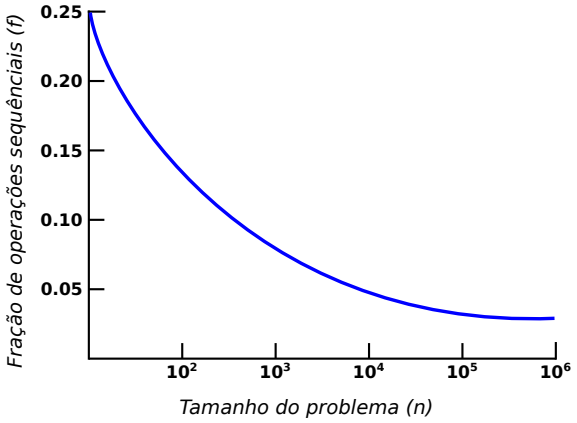

Figura 2.9: Fração de operações sequenciais em função do tamanho do problema (Adaptado de (Quinn, 1994))

Este fenômeno é chamado de Efeito Amdahl (Cerqueira, 2002). Tal efeito pode ser confirmado na Figura 2.10, que exibe o *Speedup*(S) obtido com o aumento do tamanho (n) de um problema em função do número de processadores (p).

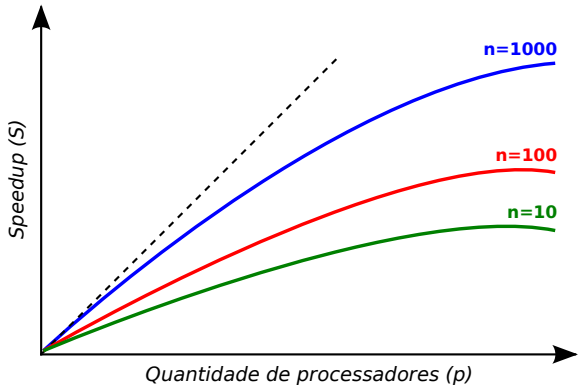

Figura 2.10: *Speedup* alcançado para diferentes tamanhos de um problema (Adaptado de (Quinn, 1994))

Devido ao fato do *Speedup* potencial ser tão influenciado pelo tamanho do problema, surgiram algumas novas leis que foram criadas para capturar esse efeito. Pode-se citar como a principal delas a **Lei de Gustafson-Barsis** (Gustafson, 1988) que prova que a **Lei de Andahl** não pode ser sempre aplicada, pois ela considerava que as partes sequenciais e paralelas eram independentes entre si, o que não é verdade.

Lei 2.2 (Gustafson-Barsis) *Seja f a fração de uma computação que deve ser executada sequencialmente, onde $f \leq f \leq 1$. O Speedup máximo (S) alcançado por um computador executando uma computação com p processadores é dado por:*

$$S \leq p - f \times (p - 1)$$

Essa lei alterou completamente o panorama das pesquisas com sistemas paralelos e distribuídos, impulsionando o seu estudo, pois provou que estes sistemas poderiam apresentar ganhos maiores do que o imaginado. Por exemplo, para $f = 0,05$ e $p = 10$, o *Speedup* máximo de acordo com a **Lei de Gustafson-Barsis** é de $9,55$ e para a **Lei de Andahl** o valor de *Speedup* máximo é $6,89$. Ou seja, pela fórmula de Gustafson (1988) obtêm-se um resultado bem mais próximo do valor ideal (linear) que é 10.

No entanto, a **Lei de Amdahl** pode ser relevante quando os programas sequenciais são paralelizados incrementalmente. Nesta abordagem, durante o desenvolvimento de aplicações paralelas, um algoritmo sequencial é inicialmente analisado para identificar os componentes com requisitos computacionais. Esses componentes são depois adaptados para execução paralela, um após o outro, até atingir um rendimento aceitável.

2.6 Desenvolvimento de Aplicações Paralelas

O desenvolvimento de programas capazes de realizar execuções em paralelo pode ser obtido de duas maneiras.

Uma delas, **paralelismo explícito**, ocorre quando o paralelismo fica a cargo do programador, que sabe construir programas paralelos e faz uso de linguagens e ferramentas de programação que lhe oferecem suporte. Nessa proposta o programador é responsável por especificar o que pode e/ou deve ser executado em paralelo, exigindo que, além de dominar o algoritmo, o programador conheça as características operacionais da arquitetura paralela. Além disso, nesse caso, o projeto do compilador paralelizador tem sua complexidade reduzida, porém ainda engloba aspectos mais sofisticados que o projeto de compiladores para códigos sequenciais.

Outra forma, **paralelismo implícito**, faz uso de compiladores que detectam o paralelismo existente em um código sequencial gerando código paralelo automaticamente. Fica deste modo oculto ao programador o acréscimo de complexidade introduzido pelo paralelismo, mantendo-se a sintaxe da codificação sequencial, ganhando, porém grande complexidade a tarefa de elaboração do compilador paralelizador. Entretanto, atualmente, os compiladores capazes de paralelizar programas de forma automática não conseguem bons resultados em tarefas mais sofisticadas e com um certo grau de dependência entre elas (Krishnamurthy e Yelick, 1995).

Capítulo 3

Desenvolvimento

O presente capítulo trata dos detalhes de implementação do **Java Cá & Lá**, o *middleware* para computação paralela em memória compartilhada e distribuída proposto neste trabalho de Monografia. Especificamente, são apresentados, nas Seções seguintes, as principais decisões de projeto necessárias à este trabalho, e os motivos que compeliram a tais escolhas.

Primeiramente na Seção 3.1 são descritos, de forma sucinta, os componentes do *middleware* e os serviços que cada um provê. A Seção 3.2 descreve a arquitetura de cada componente. Na Seção 3.3 são apresentadas as características do *middleware*. E por último a utilização do *middleware* **Java Cá & Lá** é exemplificada na Seção 3.4.

3.1 Descrição

Para atingir o seu objetivo, o *middleware* fornece um conjunto de serviços para o usuário desenvolver aplicações paralelas.

A arquitetura de *software* adotada na solução, foi a **Arquitetura de Componentes** (descrita na Seção 3.2). O *middleware* para o modelo de memória compartilhada é baseado em apenas um componente chamado de **Client**, enquanto que para o modelo de memória distribuída são necessários, além do componente **Client**, os componentes **MasterServer**, **RunnerServer** e **GlobalVarServer**. Para ambos modelos, também é necessário a utilização do componente **GlobalVarAccess** que fornece os serviços para manipulação das variáveis globais. As funcionalidades de tais componentes são descritos a seguir:

3.1.1 Componente Client

É o componente utilizado pelo usuário para acesso aos serviços do *middleware*. É o principal componente, sendo utilizado pelos dois modelos de memória (*shared* e *distributed*). O componente **Client** fornece três serviços básicos para o desenvolvimento de aplicações paralelas:

Execução de tarefas Principal serviço do *middleware*. É responsável pela execução das

tarefas desenvolvidas pelo usuário, de forma transparente, ocultando todas as rotinas necessárias ao modelo de programação paralelo, como por exemplo, sincronização, comunicação, balanceamento de carga, controle de acesso aos processos, dentre outras rotinas. Assim, o usuário pode executar de maneira concorrente qualquer método que atenda aos padrões do *middleware*. Ao invocar o serviço, deve-se especificar o nome do método a ser executado juntamente com os seus argumentos, caso o método necessite de argumentos. A cada nova execução de tarefa é retornado um índice (*idx*) utilizado para identificar a chamada do serviço.

Recuperação de resultados Após a execução da tarefa, o seu resultado estará disponível para consulta. O resultado de uma tarefa é o valor de retorno do método no qual foi requisitado a sua execução. Para se recuperar um resultado é necessário especificar o índice, *idx*, da chamada do serviço de execução. Existem duas maneiras de invocar o serviço de recuperação de resultado:

- ***Esperando o término da execução*** Desta maneira, ao se invocar o serviço de recuperação, o mesmo verifica se a execução em questão já foi concluída. Caso contrário ele bloqueia o programa principal até que a execução termine, retornado o resultado.
- ***Imediata:*** O resultado é retornado no momento da invocação do serviço, independentemente se a execução da tarefa já foi concluída ou não. Ao se requisitar o resultado de uma execução não concluída, é retornado um valor nulo, pré-definido para o *middleware*.

Aguardar execução de uma tarefa Ao ser invocado este serviço bloqueia o programa principal até que a tarefa, indicada pelo índice (*idx*), termine. Tal serviço funciona de maneira análoga ao comando `join()` das bibliotecas de manipulação de *threads*.

3.1.2 Componente MasterServer

Utilizado apenas quando o *middleware* opera no modelo de memória distribuída, este componente é responsável pelo gerenciamento de toda a infraestrutura do **Java Cá & Lá**. São funções deste componente:

- Gerenciar a localização dos outros componentes do *middleware*;
- Receber as requisições de execução de tarefas do componente **Client** e distribuí-las, de forma balanceada, entre os componentes **RunnerServers** para a execução;
- Verificar a integridade de cada componente da solução, ou seja, verificar se algum componente apresentou alguma falha ou foram removidos da infraestrutura;

- Registrar a inserção de novos componentes na infraestrutura durante o ciclo de vida do *middleware*;

Existirá apenas um componente **MasterServer** na infraestrutura do *middleware*.

3.1.3 Componente RunnerServer

Também utilizado apenas no modelo de memória distribuída, o componente **RunnerServer** é um tipo de *slave* que tem como função, apenas a execução das tarefas enviadas pelo **MasterServer** e o envio dos resultados para o **Client**. Pode-se utilizar quantos componentes **RunnerServer** deseja-se no **Java Cá & Lá**.

3.1.4 Componente GlobalVarAccess

O componente **GlobalVarAccess** provê os serviços necessários à manipulação de variáveis globais. São estes os serviços:

Criar variável global Cria uma nova variável global. Para utilizar este serviço é necessário fornecer o nome e o valor da variável.

Atualizar variável global Atualiza o valor de uma variável, fornecendo para este fim o nome e o novo valor da variável global.

Recuperar valor da variável global Ao receber o nome da variável este serviço retorna o seu atual valor.

Bloquear variável global Bloqueia uma variável para uso exclusivo do processo que solicitou o bloqueio. Caso a variável já esteja em estado de bloqueio, a requisição é empilhada para ser atendida posteriormente.

Desbloquear variável global Desbloqueia uma variável na qual o processo requisitante efetuou o seu bloqueio.

Qualquer processo pode fazer uso deste componente, ou seja, pode ser utilizado tanto por um processo do cliente que esteja executando separado do *middleware*, como em uma *Task* dentro do **Java Cá & Lá**.

3.1.5 Componente GlobalVarServer

Quando o *middleware* opera no modelo de memória distribuída, se faz necessário a utilização de um componente central para controle do acesso a variáveis compartilhadas (variáveis globais) de maneira distribuída. Pois, tais variáveis estarão disponíveis para acesso por diferentes componentes distribuídos na rede. Portanto este é o papel do **GlobalVarServer**.

Ao ser utilizado no modelo de memória compartilhada, o componente **GlobalVarAccess** é embutido como um subcomponente do **Client**, como é descrito na Seção 3.2.1.2.

3.2 Arquitetura do Middleware

Para o desenvolvimento da solução foi utilizado a Arquitetura de Componentes. A escolha desta arquitetura é justificada pelo fato dos componentes serem grupos de código coesos, na forma de código fonte ou executável, com interfaces bem definidas e comportamentos que fornecem forte encapsulamento do conteúdo e são, portanto, substituíveis. As arquiteturas baseadas em componentes tendem a reduzir o tamanho efetivo e a complexidade da solução e, portanto, são mais robustas e flexíveis.

A seguir é apresentada, na Figura 3.1 a arquitetura do middleware (para o modelo *Shared* e *Distributed Memory*) e em seguida cada componente é detalhado de forma individual. Neste detalhamento são apresentados as características dos subcomponentes que compõe os quatro principais componentes que foram apresentados na Seção 3.1. É válido ressaltar que vários subcomponentes são comuns aos componentes principais, desta maneira, para fins didáticos, os subcomponentes serão descritos por ordem de aparição neste texto.

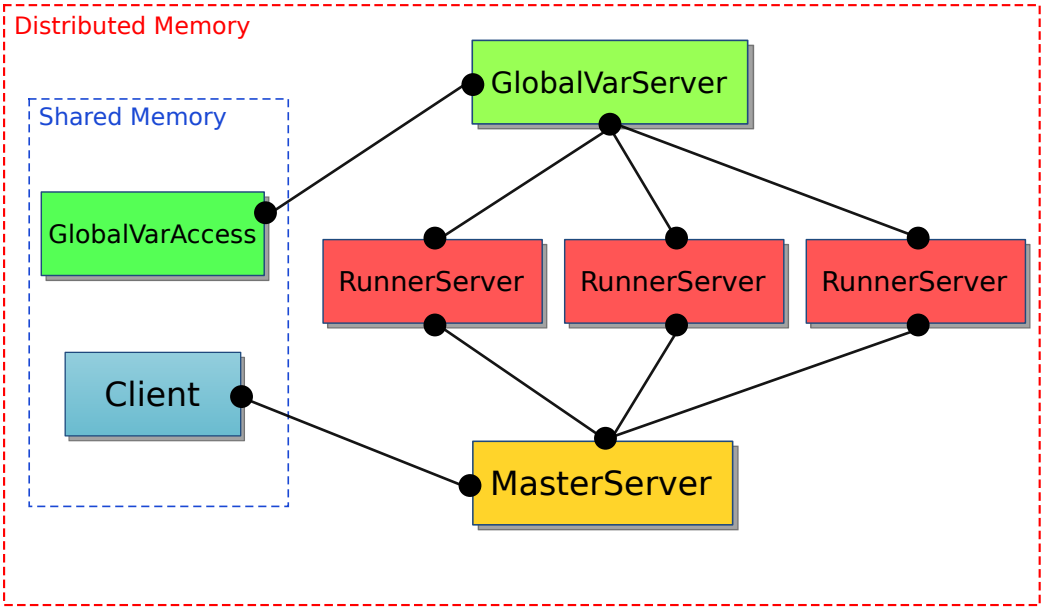

Figura 3.1: Arquitetura do *middleware* **Java Cá & Lá**.

3.2.1 Arquitetura do Componente Client

Como descrito na Seção 3.1, existem duas variações do componente **Client**. Um para o modelo de memória compartilhada e o outro para o modelo de memória distribuída. A seguir, a arquitetura de cada variação do componente **Client** é descrita:

3.2.1.1 Componente Client para Memória Compartilhada

A Figura 3.2 apresenta a arquitetura para o componente **Client** no modelo de memória compartilhada. Neste componente, que é executado em uma máquina com multiprocessadores, não é necessária comunicação com outros componentes, assim todo o fluxo de dados e instruções é realizado internamente.

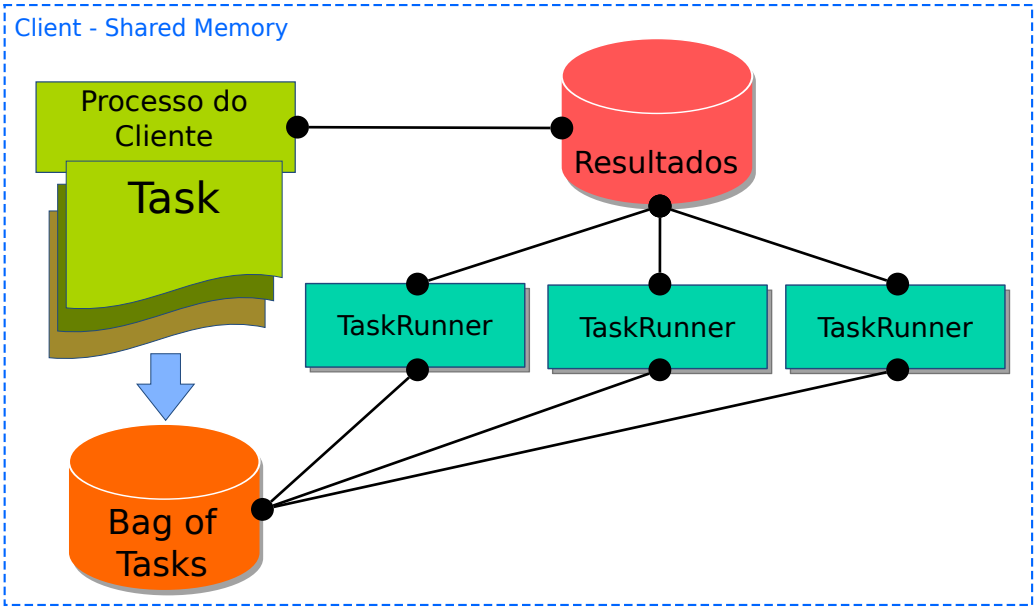

Figura 3.2: Arquitetura do componente **Client** para o modelo de memória compartilhada.

A Figura 3.2 mostra o fluxo de funcionamento do componente que utiliza o modelo de programação paralela conhecido na literatura como **Produtor e Consumidor**[1]. Primeiramente o processo do cliente requisita a execução de uma tarefa "produzindo" assim uma **Task**. Esta **Task** por sua vez é armazenada em um "recurso" (**Bag of Tasks**). Por último os subcomponentes **TaskRunners**, que desempenham o papel de "consumidores", recuperam uma **Task** do recurso e efetuam a sua execução, produzindo um valor de resultado que é armazenado em uma estrutura de dados específica, chamada de **Resultados**. A partir deste momento,

[1]Segundo Grama et al. (2002), o modelo de programação paralela **Produtor e Consumidor** descreve dois processos, o produtor e o consumidor, que partilham um *buffer* limitado utilizado como uma fila. A função do produtor é gerar algum trabalho (*work*) e colocá-lo no buffer. Ao mesmo tempo o consumidor está consumindo estes *works* (ou seja, removendo-os do *buffer*) e realizando as tarefas necessária para concluir o *work*

o resultado da **Task** em questão estará disponível para consulta pelo processo do cliente. A quantidade de **TaskRunners** que o componente possuíra é parametrizado.

O subcomponente **TaskRunner** é descrito na Seção 3.2.2.

3.2.1.2 Componente Client para Memória Distribuída

O componente **Client** para o modelo de memória distribuída difere-se da sua versão para o modelo de memória compartilhada, no sentido de que há necessidade de comunicação com outros componentes. Essa comunicação é realizada pelo subcomponente **Connector**, descrito na Seção 3.2.3. A seguir a Figura 3.3 apresenta a arquitetura deste componente.

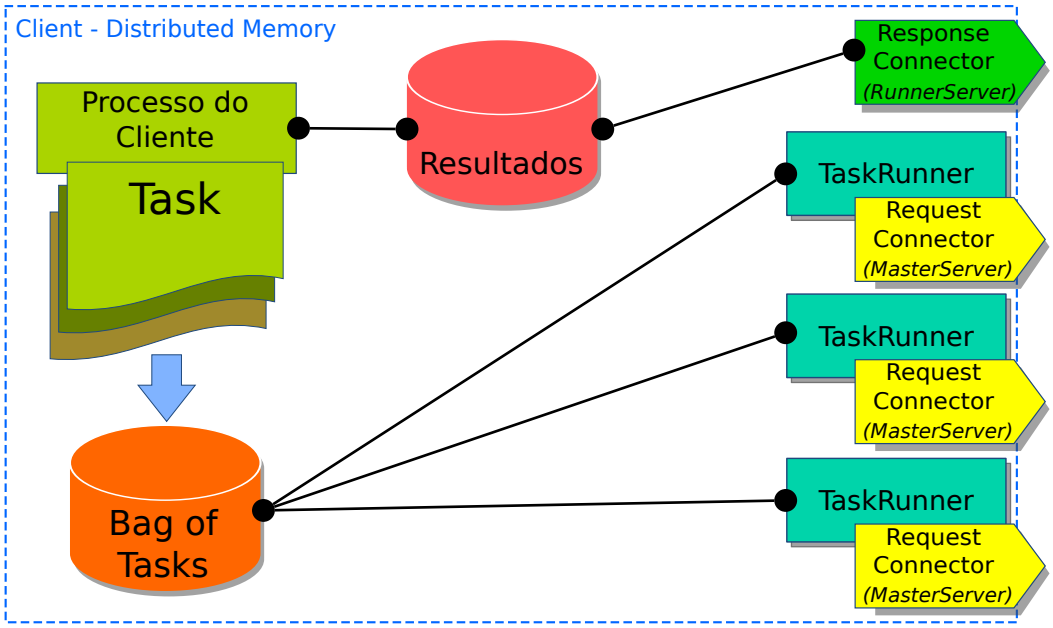

Figura 3.3: Arquitetura do componente **Client** para o modelo de memória distribuída.

O funcionamento é análogo ao da versão para memória compartilhada, diferindo-se apenas na maneira de execução das tarefas. Nesta versão, um **TaskRunner** não executa a **Task** localmente ao recuperá-la do **Bag of Tasks**, ao invés disso ele requisita a execução remota da **Task** por meio de um **RequestConnector**, que contém uma conexão direta com o **MasterServer**. Ao mesmo tempo, os resultados das execuções de tarefas remotas são recebidos por um **ResponseConnector** que fica disponível para receber o resultado a qualquer instante. Após receber um resultado de uma execução, o mesmo é armazenado na estrutura **Resultados**. A quantidade de **TaskRunners** que o componente possuíra é parametrizado.

3.2.2 Subcomponente TaskRunner

Este subcomponente é responsável pela execução de uma **Task**. Quando utilizado em um **Client** no modelo de memória compartilhada ou em um **RunnerServer** (Seção 3.2.5) ele apenas executa a tarefa e retorna o seu valor. Entretanto ao ser utilizado em um **Client** no modelo de memória distribuída, a função do **TaskRunner** é apenas enviar a **Task** para ser executada remotamente.

3.2.3 Subcomponente Connector

A função deste subcomponente é a comunicação, ou seja, a transmissão de mensagens entre os componentes do *middleware* **Java Cá & Lá** quando utilizado no modelo de memória distribuída. Existem dois tipos de **Connector**:

RequestConnector É um tipo de conector simples que estabelece uma conexão única com outro conector. Sua função é enviar mensagens de requisição e esperar por uma mensagem de retorno. A mensagem de retorno pode ser apenas uma confirmação que a mensagem de requisição foi recebida ou pode ser o resultado da requisição.

ResponseConnector Este tipo de conector é capaz de atender múltiplas conexões e está sempre disponível para uma nova conexão. Ele recebe mensagens de requisição e retorna uma outra mensagem confirmando o recebimento ou retornando o resultado da requisição.

3.2.4 Arquitetura do Componente MasterServer

Quando o *middleware* é utilizado no modelo de memória distribuída o componente **MasterServer** exerce um importante papel no gerenciamento de toda a infraestrutura do **Java Cá & Lá**. A seguir a figura 3.4 apresenta a sua arquitetura.

O **MasterServer** também adota o modelo de programação paralela **Produtor e Consumidor**. O componente é capaz de atender várias requisições de execução de tarefas. Ao receber uma **Task**, através de um **ResponseConnector** que está sempre disponível, ela é armazenada em um **Bag of Tasks**. Posteriormente esta **Task** é recuperada, ou seja consumida, pelo subcomponente **RunnerManager** para ser enviada a algum dos **RunnerServers** disponíveis. A escolha de qual **RunnerServer** irá executar a **Task** é feita de uma maneira que o balanceamento de carga entre eles seja homogêneo. A seguir é descrito o subcomponente **RunnerManager**.

3.2.4.1 Subcomponente RunnerManager

Exerce um papel fundamental dentro do **MasterServer**. Ele é responsável por todo o gerenciamento dos componentes **RunnerServers** conectados ao *middleware*. Suas funções são:

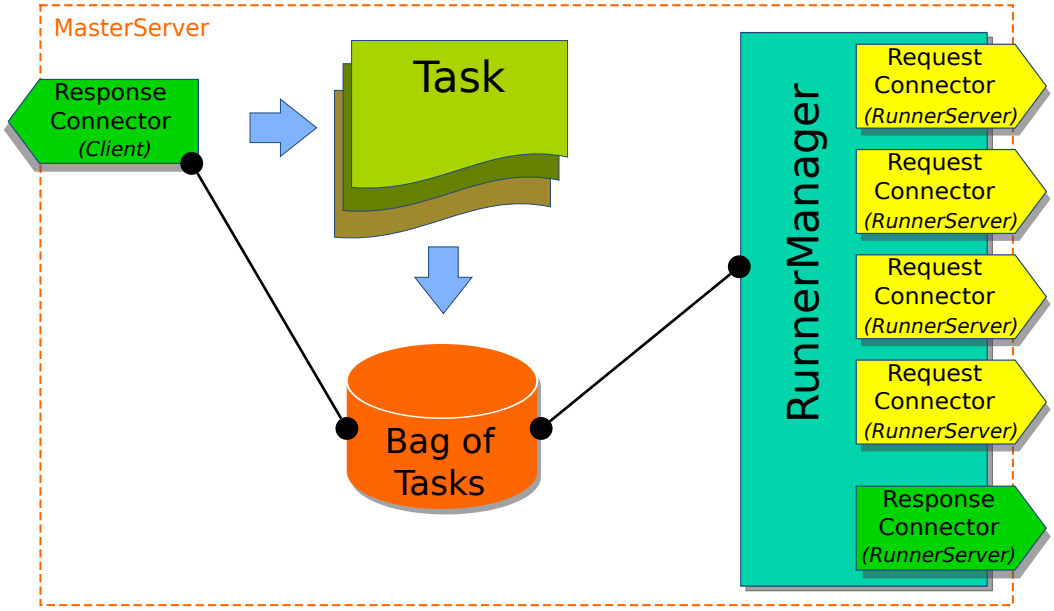

Figura 3.4: Arquitetura do componente **MasterServer**.

Registrar um novo RunnerServer Ao ser ativado um novo **RunnerServer** ele solicita a sua inserção no *middleware* ao **MasterServer**. Assim, é registrado o novo **RunnerServer** armazenando a sua localização. Por este motivo, o **RunnerManager** possui um **ResponseConnector** que está sempre disponível para novas conexões.

Envio de tarefas a um RunnerServer Seleciona o **RunnerServer** que possui o menor número de tarefas pendentes e lhe envia uma **Task** para execução. Para manter atualizada a informação de quantas tarefas pendentes cada **RunnerServer** possui, ao enviar uma **Task** é solicitado o número de **Tasks** não concluídas que o **RunnerServer** em questão possuí.

Remoção de um RunnerServer Ao se paralisar um **RunnerServer** o mesmo envia uma mensagem ao **RunnerManager** solicitando a sua remoção da infraestrutura do *middleware*. Entretanto pode acontecer o caso de um **RunnerServer** cair e não enviar a solicitação de remoção, este caso é detalhado na Seção 3.3.2.

3.2.5 Arquitetura do Componente RunnerServer

O componente **RunnerServer** que desempenha o papel de um *slave*, tem como função apenas a execução de **Tasks**. Seu funcionamento, como pode ser observado pela Figura 3.5, é semelhante ao componente **Client** para memória compartilhada.

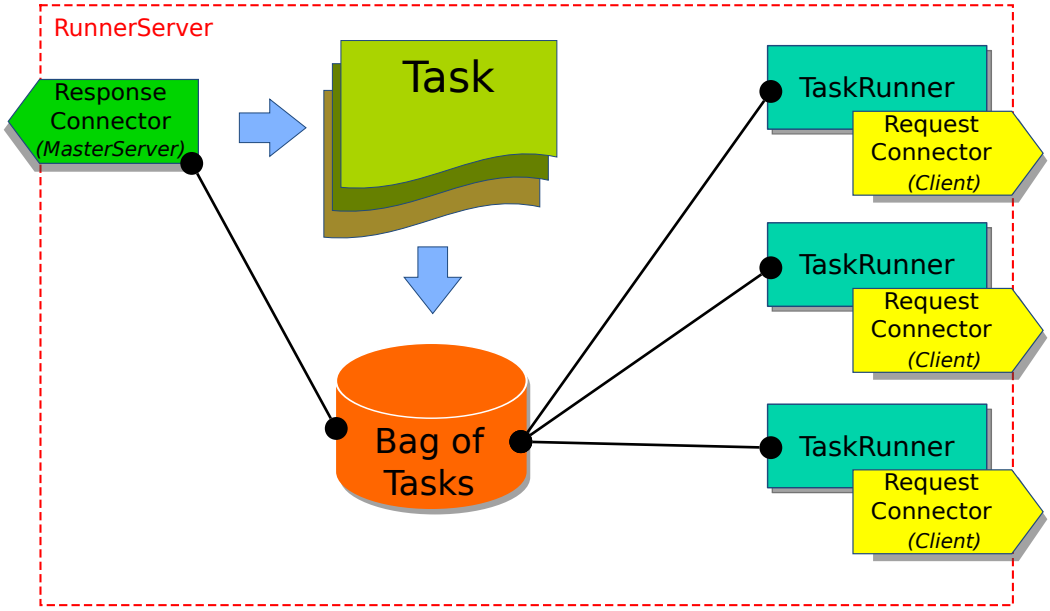

Figura 3.5: Arquitetura do componente **RunnerServer**.

Adotando também o modelo de programação paralela **Produtor e Consumidor**, o componente recebe as tarefas para execução vindas do **MasterServer** e armazena-as em um **Bag of Tasks**. Posteriormente os **TaskRunners** recuperam essas tarefas, executa-as e envia o resultado para o cliente que solicitou a execução da **Task**. As informações (localização) de qual cliente solicitou a execução é recebida juntamente com a **Task**.

3.2.6 Arquitetura dos Componentes GlobalVarAccess e GlobalVarServer

Estes dois componentes são utilizados na manipulação de variáveis compartilhadas. O primeiro, **GlobalVarAccess** é utilizado para o acesso à essas variáveis. Enquanto o segundo, **GlobalVarServer**, é responsável pelo gerenciamento das variáveis compartilhadas quando o *middleware* é executado utilizando o modelo de memória distribuída. A Figura 3.6 apresenta a arquitetura dos componentes envolvidos na manipulação de variáveis globais.

O subcomponente **GlobalVarControl**, quando utilizado em memória compartilhada faz o gerenciamento de todas as operações sobre as variáveis globais. Entretanto, quando utilizado em memória distribuída ele fornece apenas uma conexão para o componente **GlobalVarServer**, que por sua vez é quem desempenha o papel de gerenciamento das variáveis globais.

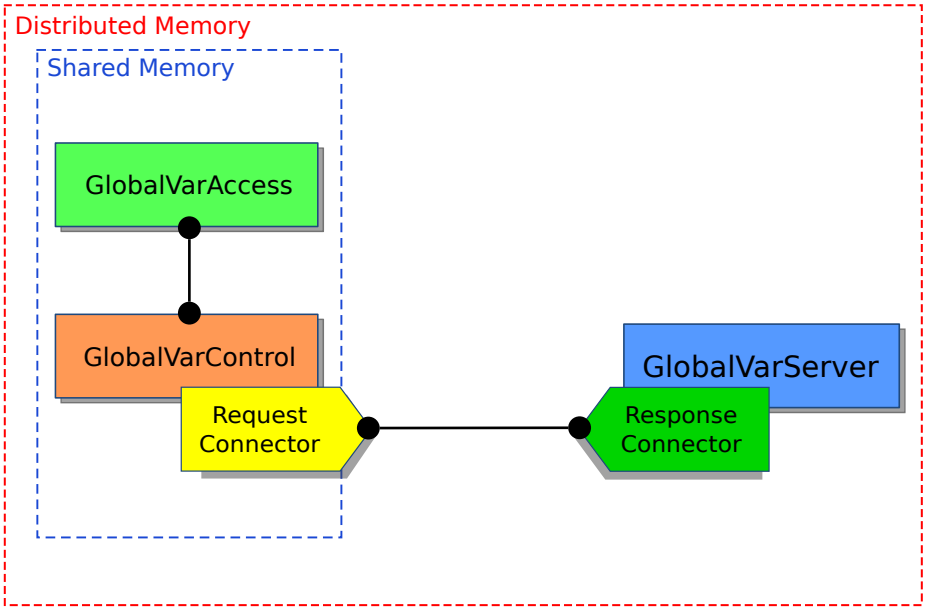

Figura 3.6: Arquitetura dos componentes **GlobalVarAccess** e **GlobalVarServer**.

3.3 Características do *middleware*

Segundo Tanenbaum e Steen (2001) um sistema distribuído tem como principal objetivo facilitar o acesso do usuário a recursos remotos e compartilhar estes recursos com outros usuários de uma forma controlada. Também é de suma importância esconder o fato de que seus processos e recursos estão fisicamente distribuídos através de vários computadores. Essa meta é obtida aplicando-se níveis de transparências ao sistema.

As próximas Seções descrevem os níveis de transparências alcançados no *middleware* **Java Cá & Lá**. Outra característica descrita é a sua capacidade de auditoria.

3.3.1 Transparência de Concorrência

Na transparência de concorrência o usuário não está ciente da existência de acesso simultâneo à recursos remotos. O mesmo ocorre no *middleware* **Java Cá & Lá**, o usuário submete a tarefa para execução e não tem conhecimento de quantas outras tarefas estão sendo executadas simultaneamente.

Outro tipo de concorrência que foi ocultado do usuário é o acesso às variáveis compartilhadas. Pois é possível acessar e efetuar bloqueios nas variáveis globais sem a preocupação de quantos outros processos estão tentando acessar a mesma variável.

3.3.2 Transparência a Falhas

Esse é um fator de extrema importância, uma vez que recursos são compartilhados, não é nada agradável que os problemas também sejam compartilhados, por isso, em caso de falhas é essencial que o sistema seja capaz o suficiente para tratar uma possível falha, sem que isso propague para o usuário. O *middleware* **Java Cá & Lá** apresenta vários tratamentos de falhas, descritos a seguir:

Falha de Comunicação Os conectores são capazes de detectar quando uma mensagem enviada não foi entregue. Para isso ele espera por uma confirmação durante um tempo (t) pré-determinado. Caso não haja uma confirmação de entrega, a mensagem é enviada novamente. Este processo é repetido durante n vezes, onde n é um parâmetro. Somente depois de n tentativas, é lançada uma exceção para o cliente tratar.

Falhas durante a execução das tarefas Durante a execução de uma **Task** podem ocorrer falhas derivadas do método a ser executado (erros de programação do usuário). Tais falhas são repassadas ao **Client** em forma de exceções.

Falha de integridade de componentes Com exceção do componente **MasterServer** qualquer outro componente pode falhar sem afetar o funcionamento do *middleware*. Isso é possível, pois o **MasterServer** realiza uma consulta, a cada intervalo de tempo (t) pré-determinado, a todos os componentes conectados a ele. Assim, existem duas situações possíveis de um componente falhar:

1. ***Falha do componente Client*:** Se um dos clientes parar de responder, o **MasterServer** detecta a sua falha e cancela todas as tarefas que ele solicitou execução. Em um segundo momento, ele destrói todas as conexões estabelecidas com este cliente.
2. ***Falha do componente RunnerServer:*** O **MasterServer** mantém um registro de quais tarefas foram enviadas a quais **RunnerServers**. Assim, caso um componente **RunnerServer** falhar, é possível enviar as tarefas não concluídas por este componente a outros que estejam disponíveis. Também é tratada a falha de todos os componentes *RunnerServers*. Nesta ocasião, o **MasterServer** cria um novo **RunnerServer** local para executar as tarefas.

3.3.3 Transparência de Localização

Neste nível de transparência o usuário não tem ciência de onde os processos estão sendo executados e nem da localização dos dados. O *middleware* **Java Cá & Lá** oferece este tipo de transparência pois o usuário não necessita conhecer a localização dos componentes, com exceção do **MasterServer**. Por exemplo, um componente **RunnerServer** pode estar alocado em uma máquina da rede local ou em um máquina fora de sua rede.

3.3.4 Transparência de Migração

Este nível de transparência é atingido, pois os dados do usuário, quando repassados sob forma de argumentos em uma tarefa ou na criação de variáveis globais, são transportados sem o seu conhecimento.

3.3.5 Auditoria

A auditoria do *middleware* é realizada por *Logging*[2]. São oferecidos quatro níveis hierárquicos de *logs*:

1. ***ERROR*:** São registrados apenas os erros que ocorrem no *middleware*.
2. ***WARNING*:** São registrados os erros e alertas que ocorrem no *middleware*.
3. ***INFO*:** É o nível padrão, onde são registrados informações relevantes, assim como os erros e alertas.
4. ***DEBUG*:** Semelhante ao nível ***INFO***, as informações registradas são mais detalhes. Este nível auxilia o usuário a detectar problemas durante o desenvolvimento com o *middleware*.

Os *logs* podem sem registrados na saída padrão do usuário ou gravados em arquivos.

3.4 Utilização do *middleware* Java Cá & Lá

As próximas Seções detalham a forma como o **Java Cá & Lá** é utilizado. São apresentados os detalhes que o usuário deve estar atento durante o desenvolvimento de aplicações com o *middleware*, a maneira como os componentes são configurados e quais são as primitivas (chamadas de métodos) estão disponíveis. Por fim é apresentado um conjunto de exemplos demonstrando a utilização do **Java Cá & Lá**.

3.4.1 Pré-requisitos

Existem algumas regras impostas pelo *middleware* que o usuário deve seguir durante o desenvolvimento. São elas:

Objetos serializáveis Todas os objetos repassados ao middleware devem ser passiveis de serialização. Em *Java* isto é possível declarando que a classe deve implementar a interface `Serializable`.

[2] *Logging* é a técnica que registra os eventos relevantes em um sistema computacional. Esse registro é importante para a realização de auditoria e diagnóstico de problemas no sistema

Bloqueio/desbloqueio de variáveis Ao bloquear uma variável global, o usuário deve sempre desbloqueá-la.

Enceramento do componente Client Ao inicializar um componente **Client**, o usuário deve encerrá-lo. Caso contrário o processo do cliente não terminará a sua execução.

3.4.2 Configuração e Inicialização dos Componentes

Os componentes do *middleware* **Java Cá & Lá** requerem uma pré-configuração por meio de passagem de parâmetros. Essa configuração é realizada por meio de um arquivo de propriedades, conforme exemplo do Código 3.1, que é indicado na inicialização do componente. Alguns parâmetros são opcionais, assumindo assim um valor padrão. A seguir são apresentados os parâmetros necessário para a configuração de cada componente e a forma de inicializá-los.

3.4.2.1 Configuração e Inicialização do Componente Client

O componente **Client** é o único que o usuário utiliza por meio de uma biblioteca. É necessário que ele adicione a biblioteca **Java Cá & Lá** ao seu projeto de desenvolvimento. Os parâmetros necessários para a configuração são apresentados no Código 3.1 e descritos a seguir.

Código 3.1: Exemplo de um arquivo de configuração do componente **Client**

```
1  TYPE = distributed
2  QTY_TASKRUNNERS = 4
3  LOG_LEVEL = info
4  HOST_MASTER_SERVER = 200.131.208.16
5  PORT_MASTER_SERVER = 5000
6  HOST_GLOBALVAR_SERVER = 200.131.208.20
7  PORT_GLOBALVAR_SERVER = 5005
```

`TYPE` Tipo de cliente, ou seja, deve-se especificar um dos dois modelos de memória a ser utilizado: (*i*)`shared` - memória compartilhada; (*ii*)`distributed` - memória distribuída.

`QTY_TASKRUNNERS`(opcional) Quantidade de **TaskRunners** que serão utilizados no cliente. **Valor Padrão:** número de processadores disponíveis na máquina hospedeira.

`LOG_LEVEL`(opcional) Nível de *log* a ser utilizado. As opções são `ERROR`, `WARNING`, `INFO` e `DEBUG`, conforme apresentado na Seção 3.3.5. **Valor Padrão:** `INFO`.

`HOST_MASTER_SERVER`* Endereço da máquina que hospeda o componente **MasterServer**.

`PORT_MASTER_SERVER`* Porta na qual o componente **MasterServer** está disponível.

`HOST_GLOBALVAR_SERVER`* Endereço da máquina que hospeda o componente **GlobalVarServer**.

PORT_GLOBALVAR_SERVER* Porta na qual o componente **GlobalVarServer** está disponível.

Os parâmetros marcados com um (*) são necessários apenas quando é utilizado o modelo de memória distribuída, ou seja, quando `TYPE = distributed`.

A inicialização do componente é realizada por meio do método `initialize(String fileProperties)`, conforme exemplo do Código 3.2, onde `fileProperties` é o nome do arquivo de configuração.

Código 3.2: Exemplo da chamada do método de inicialização do componente **Client**

```
1  //código do usuário
2  JCL.initialize("configuracoes.properties");
3  //código do usuário
```

3.4.2.2 Configuração e Inicialização dos Componentes do Tipo Server

Os componentes do tipo **Server** (**MasterServer**, **RunnerServer** e **GlobalVarServer**) são utilizados na forma de arquivos executáveis do tipo **JAR**[3]. Também é necessária a passagem de parâmetros de configuração na inicialização do componente. A seguir são apresentados os parâmetros necessários para cada tipo de **Server**.

- **MasterServer**

  PORT_MASTER_SERVER Porta na qual o componente **MasterServer** ficará disponível na máquina hospedeira.

  QTY_WORKERMASTERS(opcional) Quantidade de *threads* que serão utilizadas para atender as requisições ao **MasterServer**. **Valor Padrão:** número de processadores disponíveis na máquina hospedeira.

  LOG_LEVEL(opcional) Nível de *log* a ser utilizado. As opções são `ERROR`, `WARNING`, `INFO` e `DEBUG`, conforme apresentado na Seção 3.3.5. **Valor Padrão:** `INFO`.

- **RunnerServer**

  PORT_RUNNER_SERVER Porta na qual o componente **RunnerServer** ficará disponível na máquina hospedeira.

  QTY_TASKRUNNERS(opcional) Quantidade de **TaskRunners** que serão utilizados para execução de tarefas no **RunnerServer**. **Valor Padrão:** número de processadores disponíveis na máquina hospedeira.

[3] *Java Archive* (JAR) é um arquivo compactado usado para distribuir um conjunto de classes *Java* ou um aplicativo *Java*. É usado para armazenar classes compiladas e metadados associados que podem constituir um programa.

LOG_LEVEL(opcional) Nível de *log* a ser utilizado. As opções são `ERROR`, `WARNING`, `INFO` e `DEBUG`, conforme apresentado na Seção 3.3.5. **Valor Padrão:** `INFO`.

HOST_MASTER_SERVER Endereço da máquina que hospeda o componente **MasterServer**.

PORT_MASTER_SERVER Porta na qual o componente **MasterServer** está disponível.

HOST_GLOBALVAR_SERVER Endereço da máquina que hospeda o componente **GlobalVarServer**.

PORT_GLOBALVAR_SERVER Porta na qual o componente **GlobalVarServer** está disponível.

- **GlobalVarServer**

PORT_GLOBALVAR_SERVER Porta na qual o componente **GlobalVarServer** ficará disponível na máquina hospedeira.

LOG_LEVEL(opcional) Nível de *log* a ser utilizado. As opções são `ERROR`, `WARNING`, `INFO` e `DEBUG`, conforme apresentado na Seção 3.3.5. **Valor Padrão:** `INFO`.

A inicialização de um componente do tipo **Server** é realizada por meio de uma chamada ao seu arquivo executável, passando como parâmetro a localização do arquivo de configuração, conforme o exemplo do Código 3.3.

Código 3.3: Exemplo de inicialização de um **RunnerServer**

```
java -jar runnerserver.jar configuration.properties
```

No exemplo do Código 3.3 é inicializado um **RunnerServer** passando o arquivo **configuration.properties**, que está localizado no mesmo diretório do executável, como argumento.

3.4.3 Primitivas Disponíveis

Ao desenvolver aplicações com o *middleware* **Java Cá & Lá** o usuário contará com duas classes que oferecem os serviços descritos na Seção 3.1.

3.4.3.1 Primitivas da Classe `JCL`

É uma classe estática[4] que provê os serviços necessários para a execução de tarefas no *middleware*. A seguir são apresentadas as primitivas (métodos) disponíveis na classe `JCL`.

`void initialize(String fileProperties)`

Inicializa o componente cliente.

[4] Uma classe estática não necessita de uma instância para acessar seus métodos.

String fileProperties: Nome do arquivo de configuração;

int execute(Object object, String nameMethod, Object... args)
Executa um método de um objeto instanciado.

Object object: Objeto da classe que contém o método a ser executado;

String nameMethod: Nome do método a ser executado;

Object... args: Argumentos para o método no qual foi requisitado a execução. Pode-se repassar quantos argumentos seja necessário;

Retorno: Id (*idx*) da requisição de execução da tarefa.

int executeStatic(Class classe, String nameMethod, Object... args)
Semelhante a primitiva `execute()`, o método `executeStatic()` executa um método estático de uma classe. Assim não é necessário criar uma instância da classe que contém o método.

Object object: Classe que contém o método a ser executado;

String nameMethod: Nome do método a ser executado;

Object... args: Argumentos para o método no qual foi requisitado a execução. Pode-se repassar quantos argumentos seja necessário;

Retorno: Índice (*idx*) da requisição de execução da tarefa.

Object getResult(int idx)
Retorna o resultado da execução da tarefa de índice `idx`, independentemente se a execução da tarefa já foi concluída ou não.

int idx: Índice da tarefa na qual deseja-se recuperar o seu resultado;

Retorno: Resultado da execução da tarefa `idx`. Ao se requisitar o resultado de uma execução não concluída, é retornado um valor nulo, pré-definido para o *middleware*.

Object getForceResult(int idx)
Retorna o resultado da execução da tarefa de índice `idx` após a sua conclusão. Caso a tarefa ainda não esteja concluída, o método bloqueia o processo que solicitou o serviço.

int idx: Índice da tarefa na qual deseja-se recuperar o seu resultado;

Retorno: Resultado da execução da tarefa `idx`.

void joinTask(int idx)
Bloqueia a execução do processo principal até que a **Task**, de índice `idx`, seja concluída.

int idx: Índice da tarefa na qual deseja-se esperar a sua conclusão;

void joinAllTasks()

Bloqueia a execução do processo principal até que todas as **Tasks** sejam concluídas.

void stopClient()

Finaliza o cliente, independentemente se as **Tasks** para as quais foram solicitadas a execução já foram concluídas ou não.

void stopClientAfterJoinAllTasks()

Finaliza o cliente, porém aguarda a conclusão de todas as **Tasks** que estão em execução.

3.4.3.2 Primitivas da Classe **AccessGlobalVar**

Classe disponível ao usuário para manipulação de variáveis compartilhadas durante o desenvolvimento. Necessita de uma instância para acesso aos seus métodos. A instanciação de objetos da classe **AccessGlobalVar** é realizada por meio do *factory* `FactoryAccessGlobaVar`. A seguir são apresentadas as primitivas que ela provê para manipulação de variáveis compartilhadas.

void setGlobalVar(String nameGlobalVar, Object valueGlobalVar)

Cria uma nova variável global.

String nameGlobalVar: Nome da variável global a ser criada;

Object valueGlobalVar: Valor da variável global a ser criada;

void updateGlobalVar(String nameGlobalVar, Object valueGlobalVar)

Atualiza o valor de uma variável global.

String nameGlobalVar: Nome da variável global a ser atualizada;

Object valueGlobalVar: Valor da variável global a ser atualizada;

Object getGlobalVar(String nameGlobalVar)

Retorna o atual valor de uma variável global.

String nameGlobalVar: Nome da variável global na qual deseja-se recuperar o seu valor;

Retorno: Valor da variável global de nome `nameGlobalVar`.

void lockGlobalVar(String nameGlobalVar)

Bloqueia uma variável global para uso exclusivo do processo que solicitou o bloqueio.

String nameGlobalVar: Nome da variável global na qual deseja-se efetuar o bloqueio;

`void unlockGlobalVar(String nameGlobalVar)`

Desbloqueia uma variável na qual o processo requisitante efetuou o seu bloqueio.

`String nameGlobalVar`: Nome da variável global na qual deseja-se efetuar o desbloqueio;

3.4.4 Exemplos

A seguir são apresentados exemplos de programação utilizando o *middleware* **Java Cá & Lá**. O exemplos serão baseados em operações da classe exemplo `Calculadora`, apresentada no Código 3.4.

A classe `Calculadora` oferece três operações básicas (adição, subtração e fatorial). Cada operação, compartilha uma variável global que é utilizada para armazenar o valor da operação. Este compartilhamento ilustra o uso de variáveis globais no desenvolvimento de aplicações com o *middleware*. Primeiramente ao ser invocado o construtor da classe cria uma nova variável global chamada `resultado`. As operações por sua vez, realizam o cálculo no novo resultado e atualizam a variável global.

Código 3.4: Classe `Calculadora` que contém funções para operações matemáticas

```
1  import jcl.globalvar.JCLAccessGlobalVarFactory;
2  import jcl.globalvar.access.AccessGlobalVar;
3
4  public class Calculadora {
5
6    public Calculadora() throws Exception {
7      AccessGlobalVar globalVarAccess = JCLAccessGlobalVarFactory
8      .getAccessGlobalVar();
9      globalVarAccess.setGlobalVar("resultado", 0);
10   }
11
12
13   public int soma(int a, int b) throws Exception {
14     AccessGlobalVar globalVarAccess = JCLAccessGlobalVarFactory
15     .getAccessGlobalVar();
16     int r = a + b;
17     globalVarAccess.updateGlobalVar("resultado", r);
18     return r;
19   }
20
21   public static int fatorial(int n) throws Exception {
22     AccessGlobalVar globalVarAccess = JCLAccessGlobalVarFactory
23     .getAccessGlobalVar();
24     int r = fat(n);
```

```
25        globalVarAccess.updateGlobalVar("resultado", r);
26        return r;
27      }
28
29      private static int fat(int n) {
30        if (n <= 1)
31        return 1;
32        else
33        return n * fat(n-1);
34      }
35
36      public static int subtracao(int a, int b) throws Exception{
37        AccessGlobalVar globalVarAccess = JCLAccessGlobalVarFactory
38        .getAccessGlobalVar();
39        int r = a - b;
40        globalVarAccess.updateGlobalVar("resultado", r);
41        return r;
42      }
```

O Código 3.5 apresenta o programa principal que utiliza as operações da classe `Calculadora`. No exemplo do código 3.4 pode-se observar que existem métodos estáticos e não-estáticos. Assim a execução de tais métodos utilizam primitivas diferentes.

Primeiramente foi solicitado a execução do método `fatorial()`. Este método por sua vez solicita o bloqueio de uma variável global e mantém este bloqueio até o fim da sua execução. Em seguida é solicitada a execução do método `soma()`, que também solicita o bloqueio da variável global. Entretanto, calcular o fatorial de um número demanda mais tempo que calcular uma soma. Assim a execução de `soma()` ficará em estado de espera até que a variável global seja desbloqueada pelo método `fatorial()`.

Código 3.5: Programa utilizando as primitivas do **Java Cá & Lá** para execução.

```
1   public class Main {
2
3     public static void main(String[] args) throws Exception {
4
5       JCL.initialize("config.properties");
6
7       AccessGlobalVar globalVarAccess = JCLAccessGlobalVarFactory
8       .getAccessGlobalVar();
9
10      Calculadora c = new Calculadora();
11
12      //Método fatorial(30), deve retornar 5040
13      int idTaskFatorial = JCL.execute(c, "fatorial", 30);
```

```
14          //Método soma(2,5), deve retornar 7
15          int idTaskSoma = JCL.execute(c, "soma", 2, 5);
16          //Espera a execução de todas as tasks
17          JCL.joinAllTasks();
18          //recupera o valor da variável global "resultado"
19          int resultadoFinal = (Integer)globalVarAccess.
20              getGlobalVar("resultado");
21          System.out.println(resultadoFinal);
22          //Método subtração(5, 2), deve retornar 3
23          int idTaskSubtracao = JCL.executeStatic(Calculadora.class,
24              "subtracao", 5, 2);
25          //recupera novamente o valor da variável global "resultado"
26          resultadoFinal = (Integer)globalVarAccess.
27               getGlobalVar("resultado");
28          System.out.println(resultadoFinal);
29          //Interrompe o cliente após a execução de todas as tasks
30          JCL.stopClientAfterJoinAllTasks();
31
32      }
33
34  }
```

No Código 3.6 é ilustrado um exemplo onde o bloqueio e desbloqueio de variáveis compartilhadas é explícito no código do usuário. O programa principal cria a variável global `varGlobal` e executa três tarefas, que contém o método `executar()` da classe `FazAlgumaCoisa` (Código 3.7). O método `executar()` realiza o bloqueio da variável global `varGlobal` para uso privado.

Código 3.6: Exemplo de uso das operações de bloqueio e desbloqueio de variáveis globais.

```
1
2   public static void main(String[] args) throws Exception {
3
4     JCL.initialize("config.properties");
5
6     AccessGlobalVar globalVarAccess = JCLAccessGlobalVarFactory
7     .getAccessGlobalVar();
8
9     Integer valorVariavel = 10;
10    globalVarAccess.setGlobalVar("varGlobal", valor);
11
12    JCL.executeStatic(FazAlgumaCoisa, "executar");
13    JCL.executeStatic(FazAlgumaCoisa, "executar");
14    JCL.executeStatic(FazAlgumaCoisa, "executar");
```

```
15
16      JCL.stopClientAfterJoinAllTasks();
17
18    }
```

Código 3.7: Classe estática *FazAlgumaCoisa* com métodos que utilizam o bloqueio de uma variável global.

```
1
2   public class FazAlgumaCoisa{
3
4     public static void executar(){
5
6       AccessGlobalVar globalVarAccess = JCLAccessGlobalVarFactory
7       .getAccessGlobalVar();
8
9       globalVarAccess.lockGlobalVar("varGlobal");
10      Integer varLocal = (Integer)globalVarAccess.getGlobalVar("↩
            varGlobal");
11      varLocal = fazAlgoPrivado(varLocal);
12      globalVarAcess.updateGlobalVar("varGlobal", varLocal);
13      globalVarAccess.unlockGlobalVar("varGlobal");
14
15    }
16
17  }
```

Capítulo 4

Experimentos

Este capítulo apresenta os resultados obtidos a partir dos experimentos realizados com o *middleware* **Java Cá & Lá**.

Foi escolhido o algoritmo do **Filtro da Mediana** para a execução de experimentos computacionais. Este algoritmo é uma técnica utilizada no processamento digital de imagens. O filtro da mediana é normalmente usado para reduzir o ruído em uma imagem. Uma de suas vantagens é manter os principais detalhes da imagem, ao contrário de outros filtros. A desvantagem é que o seu algoritmo é mais complexo, visto que utiliza ordenamento de valores para obter o resultado.

O funcionamento do **Filtro da mediana** é basicamente a verificação dos valores do *pixels* vizinhos em comparação com o pixel a ser calculado. Essa verificação consiste em ordenar os valores vizinhos de forma crescente e então escolher a mediana dessa ordenação como o novo valor do *pixel* considerado (Gonzalez e Woods, 2001). O algoritmo utilizado considera que a imagem é formada por apenas uma matriz de *pixels*, que assume valores de 0 a 255. Ou seja foram utilizadas imagens em escala de cinza.

Foram implementadas duas versões do algoritmo:

Sequencial Implementação simples que realiza a operação para todos os *pixels* da imagem.

Paralela Esta versão foi implementada utilizando o *middleware* proposto. A decomposição de tarefas foi feita de maneira simples, onde cada `Task` é responsável pela aplicação do filtro a uma porção da imagem.

A escolha deste algoritmo para realização dos experimentos é justificada pelo fato do **Filtro da Mediana** ser um algoritmo com baixa dependabilidade entre as *tasks*.

O objetivo dos experimentos aqui apresentados é demonstrar o desempenho do *middleware* e não o desempenho da implementação do algoritmo selecionado. Pois as boas práticas de programação paralela devem ser seguidas pelo usuário programador ao utilizar o **Java Cá & Lá**.

4.1 Ambiente Computacional dos Experimentos

Os experimentos foram executados em dois ambientes distintos.

- O primeiro, representando o modelo de memória compartilhada, é uma máquina com processador *Core 2Quad* de quatro núcleos e 4 *GB* de memória *RAM* utilizando Sistema Operacional **Linux Ubuntu**.
- O segundo ambiente é um *cluster* composto de 13 máquinas com processador *Phenom* de dois núcleos e 4 *GB* de memória *RAM*. As máquinas do *cluster* são conectadas por um *Switch Ethernet* 100 *Mb/s*.

O primeiro ambiente foi utilizado para testar as implementações sequencial e paralela (para *Shared Memory*). Enquanto no segundo ambiente foi testado a versão paralela para o modelo de memória distribuída.

4.2 Metodologia dos Experimentos

Foram utilizadas diversas imagens, geradas de forma aleatória, com tamanhos distintos. Cada imagem foi construída sob a forma de uma matriz de inteiros (tipo `Integer` do *Java*) de 4 *bytes*. A adoção do tipo `Integer` foi proposicional para aumentar o tamanho em disco das imagens, pois poderia ter sido utilizado o tipo `Byte` que ocupa apenas 1 *byte*. A tabela 4.1 relaciona a dimensão das imagens utilizadas com o seu tamanho em disco.

2500×2500	$\approx 23,8$ *MB*
5000×5000	$\approx 95,3$ *MB*
7500×7500	$\approx 214,5$ *MB*
10000×10000	$\approx 381,4$ *MB*
12500×12500	$\approx 596,1$ *MB*
15000×15000	$\approx 858,30$ *MB*
17500×17500	$\approx 1,1$ *GB*
20000×20000	$\approx 1,49$ *GB*
22500×22500	$\approx 1,88$ *GB*
25000×25000	$\approx 2,32$ *GB*
30000×30000	$\approx 3,35$ *GB*
35000×35000	$\approx 4,56$ *GB*
40000×40000	$\approx 5,96$ *GB*

Tabela 4.1: Relação dos tamanhos de imagens utilizados nos experimentos.

A justificativa de se aumentar o tamanho em disco das imagens é demonstrar o limite do algoritmo quando utilizado em *Shared Memory*, como o apresentado na Seções 4.3 e 4.4.

Nos experimentos realizados sobre o modelo de memória distribuída, cada máquina continha uma cópia das imagens em seus discos. Assim, não foi contabilizado o tempo de transferência das imagens. Durante a execução, as *tasks* carregaram para a memória apenas a porção da imagem que iriam utilizar no processamento do **Filtro da Mediana**.

4.3 Resultados da Implementação Sequencial

O gráfico da Figura 4.1 apresenta o tempo de execução do algoritmo em relação à dimensão da imagem.

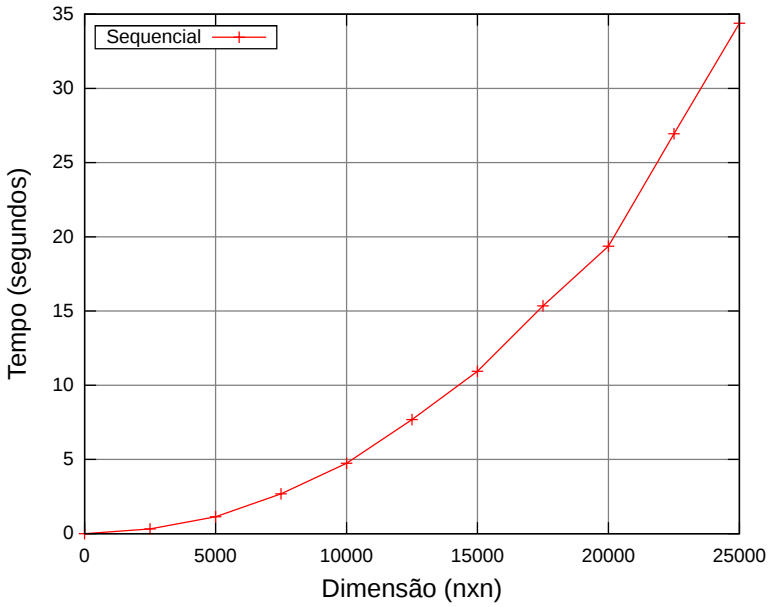

Figura 4.1: Tempo de Execução em relação à dimensão da imagem para a implementação sequencial.

Este experimento foi realizado para demonstrar o comportamento polinomial da implementação e será utilizado para comparações com o algoritmo paralelo. Não foi possível executar o algoritmo para imagens de dimensão superior à 25000 por motivos de limitações físicas da memória.

4.4 Resultados da Implementação Paralela no Modelo *Shared Memory*

Para testar o algoritmo utilizando o *middleware* em memória compartilhada, foi utilizada a implementação paralela. As imagens foram submetidas ao **Filtro da Mediana** utilizando-se 1, 2 e 4 **TaskRunners**. A Figura 4.2 apresenta o gráfico do tempo de execução em relação

à dimensão da imagem. Neste experimento também não foi possível utilizar imagens de dimensão superior à 25000, pelos mesmos motivos de limitação física da memória.

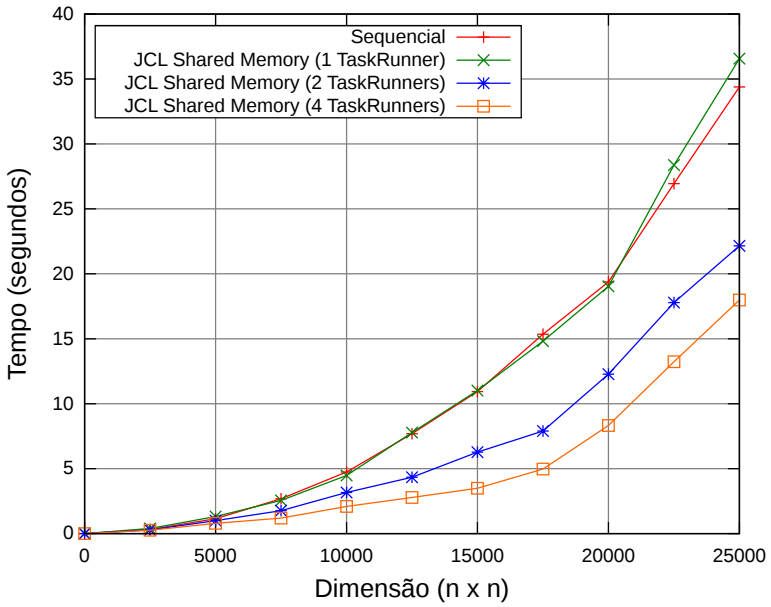

Figura 4.2: Tempo de Execução em relação à dimensão da imagem para a implementação paralela no modelo *Shared Memory*.

É possível observar uma diminuição no tempo de execução proporcional ao número de **TaskRunners** utilizados no *middleware*, o que demonstra a vantagem na sua utilização em modelos de memória compartilhada. Outro ponto importante é a grande semelhança no comportamento do tempo entre a implementação sequencial e a implementação paralela utilizando apenas um **TaskRunner**, pois isto demonstra que a utilização do *middleware* influencia de maneira tênua o desempenho da aplicação.

Também é apresentado o *Speedup* relativo ao número de **TaskRunners** utilizados no experimento, para três tamanhos distintos de imagens (5000×5000, 15000×15000 e 25000×25000). O gráfico da Figura 4.3 demonstra que o *Speedup* obtido para instâncias pequenas do problema é inferior ao obtido quando o algoritmo é utilizado em imagens maiores. Isto é explicado pela **Lei de Gustafson-Barsis**, que demonstra que o *Speedup* cresce de forma proporcional ao tamanho do problema.

4.5 Resultados da Implementação Paralela no Modelo *Distributed Memory*

Por último são apresentados os resultados dos experimentos envolvendo o modelo de memória distribuída. A implementação paralela foi testada para todos os tamanhos de imagem dispo-

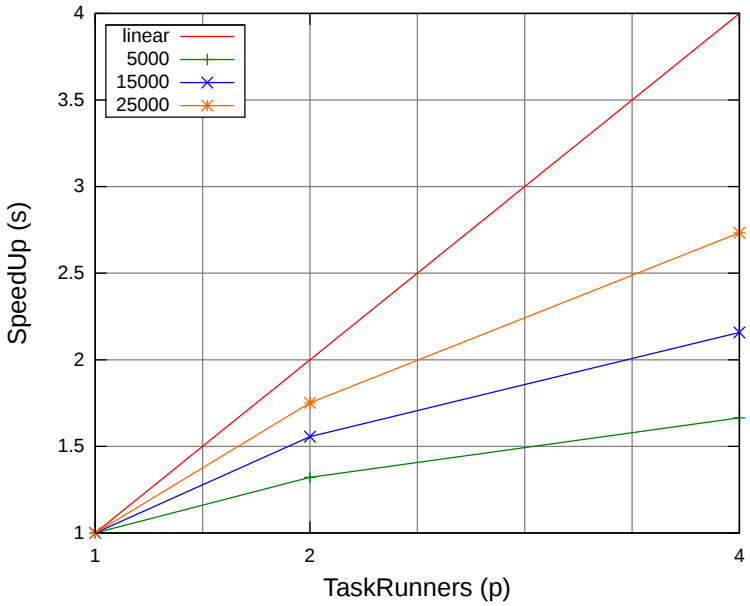

Figura 4.3: Speedup obtido pelo *middleware* no modelo *Shared Memory* para imagens de dimensões distintas.

níveis. Foram realizados testes utilizando 1, 2, 4, 8 e 12 máquinas do *cluster*, cada uma hospedando um componente **RunnerServer** e sempre uma máquina foi reservada para hospedar o componente **MasterServer**. Não foi necessário o uso do componente **GlobalVarServer**, uma vez que o algoritmo não compartilhava variáveis entre diferentes processos.

A Figura 4.4 apresenta o gráfico do tempo de execução em relação à dimensão da imagem.

Neste experimento notou-se uma redução significativa do tempo de execução do algoritmo ao aumentar o número de máquinas, o que demonstra a vantagem da utilização do *middleware* em memória distribuída. Para instâncias pequenas a redução do tempo não foi tão significativa, pois o tempo de comunicação foi maior que o tempo de computação.

A seguir a Figura 4.5 apresenta o *Speedup* relativo ao número de máquinas, para quatro tamanhos distintos de imagens (5000×5000, 15000×15000, 25000×25000 e 35000×35000).

O *Speedup* apresentado no modelo de memória distribuída aproximou-se mais do linear para grandes instâncias e manteve um comportamento similar ao *Speedup* do modelo de memória compartilhada para problemas de tamanho pequeno.

No modelo de memória distribuída foi possível executar o algoritmo para todos os tamanhos de imagens, pois cada **Task** carregava na memória apenas a porção da imagem que iria utilizar. Esta propriedade demonstra a escalabilidade do *middleware* **Java Cá & Lá**.

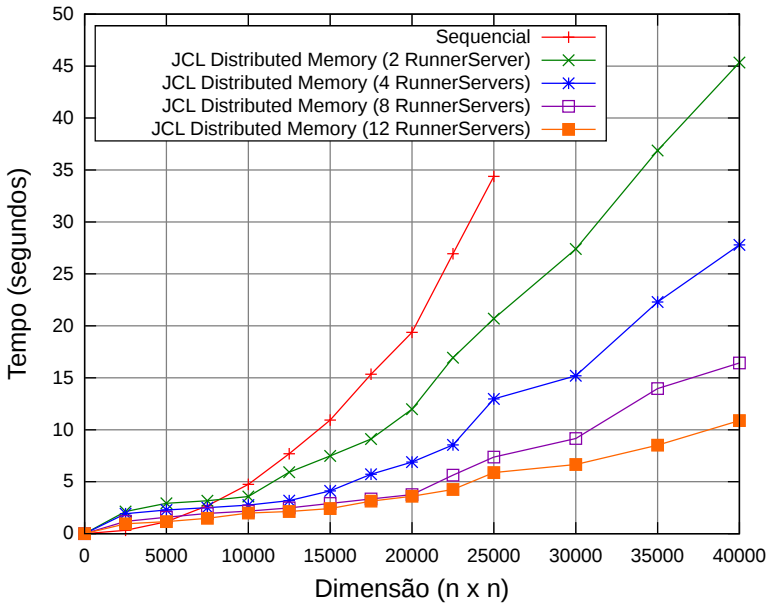

Figura 4.4: Tempo de Execução em relação à dimensão da imagem para a implementação paralela no modelo *Distributed Memory*.

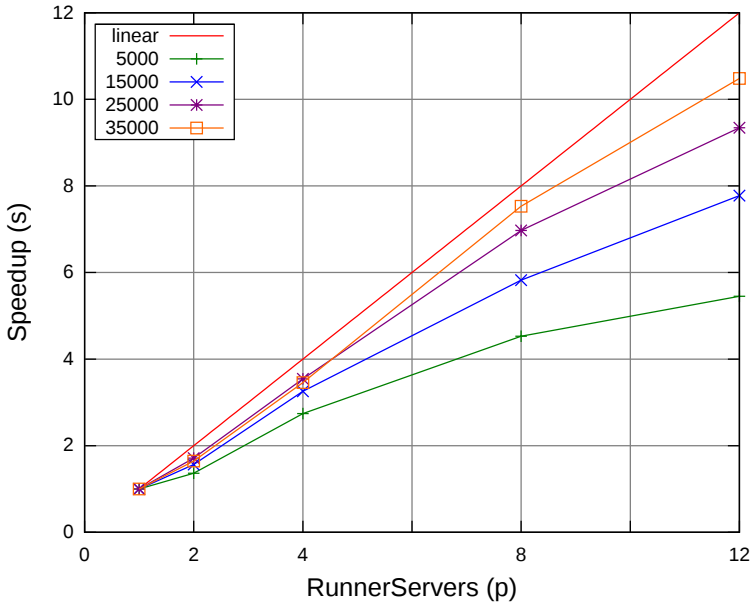

Figura 4.5: Speedup obtido pelo *middleware* no modelo *Distributed Memory* para imagens de dimensões distintas.

Capítulo 5

Conclusão

Este trabalho apresentou uma proposta de *middleware* para computação paralela em modelos de memória compartilhada e distribuída.

Foram alcançados os objetivos do trabalho, que é o desenvolvimento de um ferramenta capaz de executar tarefas de maneira paralela e que ofereça suporte a manipulação de variáveis compartilhadas. Esta ferramenta tem como principais características a transparência de concorrência, localização, migração e de falhas.

Quando programado de maneira correta, o *middleware* apresentou bons resultados, demonstrados por meio de experimentos computacionais.

Entretanto, o **Java Cá & Lá** pode ser a semente para uma grande ferramenta no futuro. Para tanto, lista-se propostas de trabalhos futuros:

- Incorporar ao *middleware* um modelo de de programação para suporte de computações paralelas semelhante ao *MapReduce*[1].
- Comparação do *middleware* proposto com trabalhos correlatos, como: *MPI*, *Hadoop*, *PVM*, etc. . . ;
- Oferecer um conjunto de primitivas para áreas específicas como: *Business Inteligence*, Inteligência Artificial, Processamento Digital de Imagens, etc. . . ;
- Aperfeiçoar a manipulação de variáveis globais, oferecendo suporte a bloqueio a porções de uma coleção. Exemplo, permitir o bloqueio de uma posição em uma *array*, ao invés de ter que bloqueá-lo totalmente;

[1] MapReduce é um modelo de programação, e *framework* introduzido pelo *Google* para suportar computações paralelas em grandes coleções de dados em *clusters* de computadores. Atualmente o *MapReduce* é considerado um novo modelo computacional distribuído, inspirado pelas funções *map* e *reduce* usadas comumente em programação funcional. Hoje existem diversas implementações de *MapReduce*, como : *Hadoop*, *Disco*, *Skynet*, *FileMap* e *Greenplum*. O *Hadoop* é a implementação mais famosa implementada em *Java* como um projeto open source (White, 2009).

- Realização experimentos mais elaborados com algoritmos que possuam uma grande dependabilidade entre as *tasks* e que exploram de maneira exaustiva o uso de variáveis compartilhadas e também realizar testes em cenários que avaliem cenários de falhas;
- Criação de uma interface gráfica para o componentes do tipo **Server** afim de facilitar a configuração e inicialização destes componentes.

Por fim, pretende-se disponibilizar o código fonte da implementação do *middleware* **Java Cá & Lá** em repositórios de *softwares open source* como o *Source Forge*.

Referências Bibliográficas

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pp. 483–485, New York, NY, USA. ACM.

Buhr, P. A.; Karsten, M. e Shih, J. (1996). Kdb: a multi-threaded debugger for multi-threaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '96, pp. 80–87, New York, NY, USA. ACM.

Cerqueira, A. G. (2002). Geração de arquivo invertido utilizando programação paralela - mpi. Master's thesis, Universidade Federal de Lavras.

Culler, D. E.; Gupta, A. e Singh, J. P. (1997). *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edição.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21:948–960.

Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gonzalez, R. C. e Woods, R. E. (2001). *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edição.

Grama, A.; Gupta, A.; Karypis, G. e Kumar, V. (2002). *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edição.

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Commun. ACM*, 31:532–533.

Krishnamurthy, A. e Yelick, K. (1995). Optimizing parallel programs with explicit synchronization. *SIGPLAN Not.*, 30:196–204.

Moore, G. (1998). Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85.

Patterson, D. A. e Hennessy, J. L. (2008). *Computer Organization and Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edição.

Quinn, M. J. (1994). *Parallel computing (2nd ed.): theory and practice*. McGraw-Hill, Inc., New York, NY, USA.

Silva, L. N. (2006). Modelagem de desempenho de sistemas com paralelismo pipeline. Master's thesis, Universidade de Brasília.

Tanenbaum, A. S. e Goodman, J. R. (1998). *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edição.

Tanenbaum, A. S. e Steen, M. V. (2001). *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edição.

White, T. (2009). *Hadoop: The Definitive Guide*. O'Reilly, 1 edição.