Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

JAVA CÁ & LÁ
Um Middleware para Computação Paralela e Distribuída

Aluno: Antonio Carlos de Nazaré Júnior
Matricula: 08.1.4999

Orientador: Joubert de Castro Lima

Ouro Preto
15 de setembro de 2011

Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

JAVA CÁ & LÁ
Um Middleware para Computação Paralela e Distribuída

Proposta de monografia apresentada ao curso de Bacharelado em Ciência da Computação, Universidade Federal de Ouro Preto, como requisito parcial para a conclusão da disciplina Monografia II (BCC391).

Aluno: Antonio de Carlos de Nazaré Júnior
Matricula: 08.1.4999

Orientador: Joubert de Castro Lima

Ouro Preto
15 de setembro de 2011

Resumo

Resumo

Palavras-chave: Palavra1. Palavra 2. Palavra 3.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

O grande interesse por problemas cada vez mais complexos tem levado a necessidade de computadores cada vez mais potentes para resolvê-los. Entretanto, limitações físicas e econômicas têm restringido o aumento da velocidade dos computadores sequenciais, ou seja, computadores que executam instruções em série, uma após a outra pela *CPU* (*Central Processing Unit*). Por outro lado, os problemas computacionais usualmente podem ter algumas de suas partes dividida em pedaços (tarefas) que poderiam ser solucionados ao mesmo tempo, ou processados de forma simultânea. Essas tarefas podem ser executadas por meio de dois paradigmas:

Computação Paralela Único processador ou vários processadores em um único equipamento utilizando uma memória compartilhada.

Computação Distribuída Vários processadores distribuídos por uma rede utilizando memória distribuída.

Processamento concorrente é então uma forma pela qual a demanda computacional é suprida através do uso simultâneo de recursos computacionais como processadores e memória para solução de um problema [1].

1.1 Taxonomia de *Flynn*

Os modelos de arquitetura de computadores são classificados pelo fluxo de instruções e dados que se apresentam. Essa classificação é definida como taxonomia de *Flynn* [1]. Ela fica divida em quatro categorias: *SISD*, *SIMD*, *MISD* e *MIMD*.

1.1.1 *SISD - Single Instruction Single Data*

Conhecido como fluxo único de instruções sobre um único conjunto de dados é o caso das máquinas convencionais com apenas um processador *Monocore*. Essa arquitetura é conhecida também como Von Neumann [3]. A Figura 1 ilustra a arquitetura *SISD*.

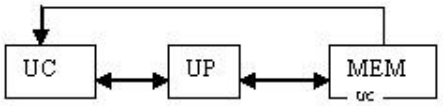

Figura 1: Arquitetura *SISD*

1.1.2 *SIMD - Single Instruction Multiple Data*

Descreve um método de operação de computadores com várias *CPU's* em computação paralela. Neste modo, a mesma instrução é aplicada simultaneamente a diversos dados para produzir mais resultados. O modelo *SIMD* é adequado para o tratamento de conjuntos regulares de dados, como as matrizes e vetores. Esse tipo de máquina opera aplicando uma única instrução a um conjunto de elementos de um vetor [3]. A Figura 2 ilustra a arquitetura *SIMD*.

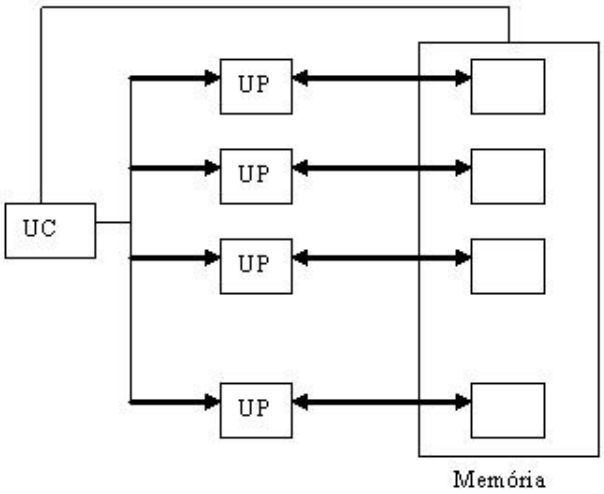

Figura 2: Arquitetura *SIMD*

1.1.3 *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 de que se pode dizer que os dados é diferente após o processamento por cada fase do pipeline. Não há muitos exemplos da existência desta arquitetura, ao contrário do *SIMD* e do *MIMD* [3]. A Figura 3 ilustra a arquitetura *MISD*.

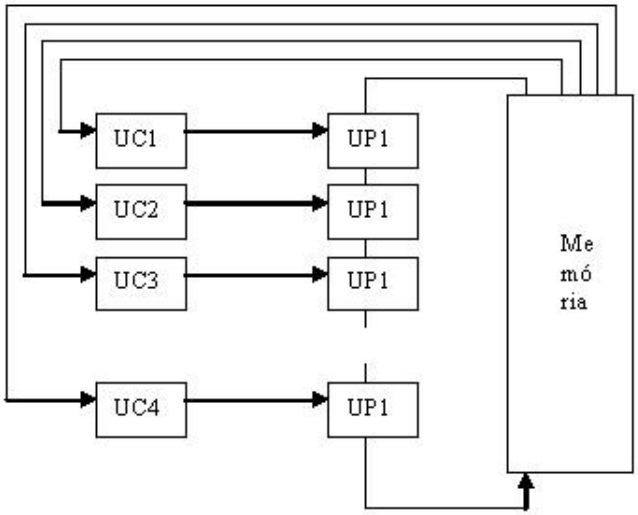

Figura 3: Arquitetura *MISD*

1.1.4 *MIMD - Multiple Instruction Multiple Data*

É um tipo de arquitetura de computação paralela, onde muitas unidades funcionais executam operações diferentes simultaneamente a diversos dados. As máquinas *MIND* são arquiteturas caracterizadas pela execução simultânea de múltiplos fluxos de ins-

truções. Essa capacidade deve-se ao fato de que são construídas a partir de vários processadores operando de forma cooperativa ou concorrente, na execução de um ou vários aplicativos. Essa definição deixa margem para que várias topologias de máquinas paralelas e de redes de computadores sejam enquadradas como *MIMD*. A Figura 4 ilustra a arquitetura *MIMD* [3]. Nas próximas seções serão detalhados os principais aspectos das arquiteturas de máquinas *MIMD*.

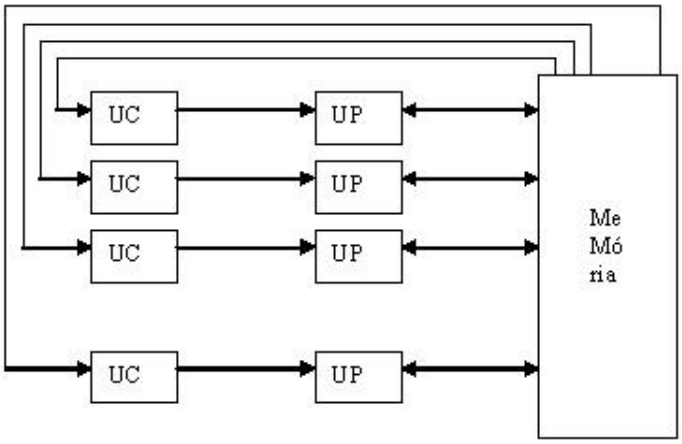

Figura 4: Arquitetura *MIMD*

1.2 Compartilhamento de Memória

1.2.1 *Shared Memory*

Neste modelo todos os processadores podem acessar toda a memória como uma área de endereçamento global. Múltiplos processadores podem independentemente, mas compartilham os mesmos recursos da memória. Outra característica importante é que as mudanças numa alocação de memória realizada por um processador são visíveis a todos os demais processadores. Essa arquitetura configura os computadores multiprocessadores [4].

Assim, a comunicação entre processos é muito simples, bastando para tanto, utilizar operações do tipo *load* e *store*. Essa estrutura é semelhante à colocação de múltiplos processadores em uma máquina von Neumann tradicional. Os múltiplos processadores são conectados à memória através de uma rede de interconexão. A Figura 5 apresenta um modelo de arquitetura de uma arquitetura *Shared Memory*.

Os multiprocessadores ainda podem ser classificados quanto a distância dos processadores à memória, e quanto aos esquemas de coerência de cache.

1.2.2 *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

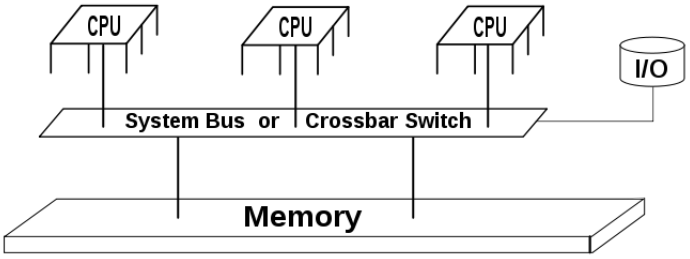

Figura 5: Arquitetura *Shared Memory*

o barramento e a memória tornam-se gargalos para o sistema, que fica limitado a uma única transferência por vez [4].

Na Figura 6 estão mostradas as memórias cache de cada processador. Essas 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.

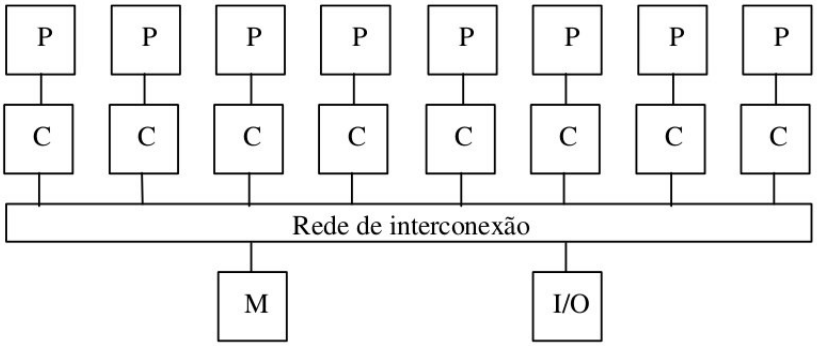

Figura 6: Arquitetura *UMA*

1.2.3 *NUMA Non-Uniform Memory Access*

Neste tipo de multiprocessadores, a memória geralmente é distribuída 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. A Figura 7 mostra a arquitetura de uma máquina NUMA [4].

1.2.4 *Distributed Memory*

Essas máquinas caracterizam-se pelo fato de que cada processador enxerga somente a sua própria memória e são também chamados de multicomputadores. Para a troca de mensagens e dados é preciso o envio de requisições através da rede de interconexão. Os

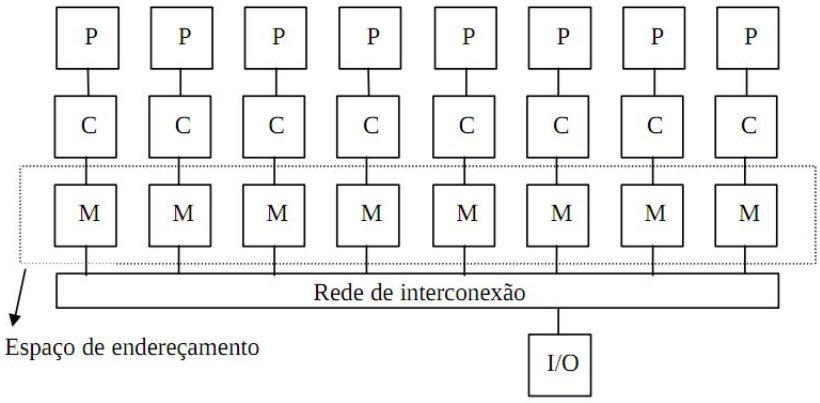

Figura 7: Arquitetura *NUMA*

multicomputadores também são chamados de sistemas de troca de mensagens (*message passing systems*). Com estas características, tais máquinas paralelas podem ser implementadas através de um conjunto de máquinas autônomas, ou seja, computadores tradicionais. A Figura. 8 mostra uma arquitetura de um multicomputador [4].

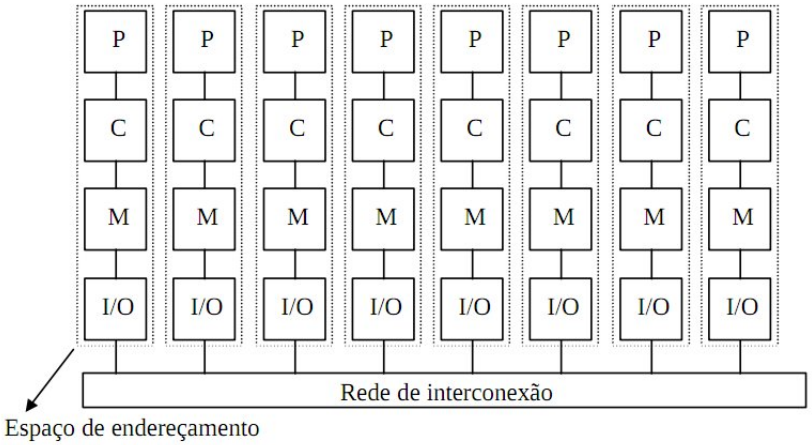

Figura 8: Arquitetura *Distribuited Memory*

1.3 Visão Geral da Computação Paralela

Tradicionalmente, o software tem sido escrito para ser executado sequencialmente. Para resolver um problema, um algoritmo é construído e implementado como um fluxo serial de instruções. Tais instruções são então executadas por uma unidade central de processamento de um computador. Somente uma instrução pode ser executada por vez; após sua execução, a próxima então é executada [2].

Por outro lado, a computação paralela faz uso de múltiplos elementos de processamento simultaneamente para resolver um problema. Isso é possível ao quebrar um problema em partes independentes de forma que cada elemento de processamento pode executar sua parte do algoritmo simultaneamente com outros. Os elementos de processamento podem ser diversos e incluir recursos como um único computador com

múltiplos processadores, diversos computadores em rede, hardware especializado ou qualquer combinação dos anteriores [2].

O aumento da frequência de processamento foi o principal motivo para melhorar o desempenho dos computadores de meados da década de 1980 a 2004. Em termos gerais, o tempo de execução de um programa corresponde ao número de instruções multiplicado pelo tempo médio de execução por instrução. Mantendo todo o resto constante, aumentar a frequência de processamento de um computador reduz o tempo médio para executar uma instrução, reduzindo então o tempo de execução para todos os programas que exigem alta taxa de processamento (em oposição à operações em memória). Aumentar a frequência significa aumentar a quantidade de energia usada em um processador. Por isso a indústria de processadores optou por aumentar o número de núcleos (unidades de processamento) de seus produtos [2].

Programas de computador paralelos são mais difíceis de programar que sequenciais, pois a concorrência introduz diversas novas classes de defeitos potenciais, como a condição de corrida. A comunicação e a sincronização entre diferentes subtarefas é tipicamente uma das maiores barreiras para atingir grande desempenho em programas paralelos. E os problemas são maiores quando é utilizado arquitetura de memória compartilhada (Shared Memory) [2].

1.4 Linguagens de Programação Paralelas

A execução simultânea de instruções possibilita ganhos no tempo final de execução e o melhor aproveitamento das potencialidades das arquiteturas em que executam. Em particular, de modo geral, apenas parte do conjunto de instruções de um programa merece atenção quanto à possibilidade de paralelização [4].

A proposta de exploração do paralelismo implícito procura manter a sintaxe da codificação sequencial. Nesta primeira proposta o compilador traduz a aplicação escrita em linguagem de alto nível sequencial para sua forma paralela. Na segunda proposta, exploração da distribuição/paralelismo explícito, as linguagens existentes são ampliadas com construtores específicos para o paralelismo, ou são criadas novas que disponibilizem os mesmos [4].

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/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 [4].

1.4.1 Extensões Paralelas

São bibliotecas cujo conjunto de instruções, aliado com as primitivas da linguagem hospedeira, permite o desenvolvimento de aplicações paralelas. Possuem como vantagem o fato de não exigirem do usuário o aprendizado de uma nova linguagem. Normalmente apresentam um desempenho melhor quando comparadas com os compiladores paralelizadores. Como exemplo tem-se *PVM* (*Parallel Virtual Machine*), *MPI* (*Message Passing Interface*, para arquiteturas de memória distribuída) e *Parmacs* [4];

1.5 Message Passing Interface

Message Passing Interface (*MPI*) é um padrão para comunicação de dados em computação paralela. Existem várias modalidades de computação paralela, e dependendo do problema que se está tentando resolver, pode ser necessário passar informações entre os vários processadores ou nodos de um *cluster*, e o *MPI* oferece uma infraestrutura para essa tarefa.

No padrão *MPI*, uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos. Inicialmente, na maioria das implementações, um conjunto fixo de processos é criado. Porém, esses processos podem executar diferentes programas. Por isso, o padrão *MPI* é algumas vezes referido como *MPMD* (*multiple program multiple data*). Elementos importantes em implementações paralelas são a comunicação de dados entre processos paralelos e o balanceamento da carga. Dado o fato do número de processos no MPI ser normalmente fixo, neste texto é enfocado o mecanismo usado para comunicação de dados entre processos. Os processos podem usar mecanismos de comunicação ponto a ponto (operações para enviar mensagens de um determinado processo a outro). Um grupo de processos pode invocar operações coletivas (collective) de comunicação para executar operações globais. O MPI é capaz de suportar comunicação assíncrona e programação modular, através de mecanismos de comunicadores (communicator) que permitem ao usuário MPI definir módulos que encapsulem estruturas de comunicação interna.

O objetivo de MPI é prover um amplo padrão para escrever programas com passagem de mensagens de forma prática, portátil, eficiente e flexível. MPI não é um IEEE ou um padrão ISO, mas chega a ser um padrão industrial para o desenvolvimento de programas com troca de mensagens.

1.5.1 Vantagens

- Facilita a comunicação entre processos, utilizando uma comunicação de mais alto nível;
- O MPI permite que você coloque sua aplicação em uma plataforma diferente, mas que suporte o padrão MPI sem necessitar alterar o código de forma que processos em diferentes linguagens podem rodar em plataformas diferentes;
- Mais de 300 rotinas são definidas em MPI;

1.5.2 Desvantagens

- Código se torna confuso e complicado;
- Necessário o conhecimento de Sistemas Distribuídos como balanceamento de carga, particionamento de Dados e tolerância à falhas;
- Necessário o uso de ferramentas auxiliares, como o DEINO-MPI;
- Exige recompilação do código a cada modificação e distribuição do executável;
- Na prática significa aprender uma nova linguagem de programação. É um padrão da indústria com várias implementações individuais.

1.6 Dificuldades na Programação Paralela

Abaixo são listadas as dificuldades envolvidas ao se desenvolver aplicações utilizando o modelo de Computação Paralela [4]:

1.6.1 Sincronização de Processos

- A sincronização é necessária para que exista um controle de sequência e um controle de acesso durante a execução de processos concorrentes.
- O controle de sequência é utilizado quando existe a necessidade de estabelecer uma determinada ordem na execução dos processos.
- O controle de acesso é utilizado quando é importante o acesso organizado aos recursos do sistema que são compartilhados pelos processos concorrentes.
- Determinar a sequência de eventos para garantir a sincronização entre processos concorrentes não é um trabalho trivial.
- Uma tarefa fora de ordem pode desencadear uma sequência errada de eventos, comprometendo o desempenho final da aplicação paralela.

1.6.2 Conversão de Algoritmo Sequencial para Paralelo

Há um considerável tempo consumido pelo programador em analisar o código serial para depois recodificá-lo de forma paralela. São passos necessários:

- Rever o programa sequencial;
- Avaliar como será particionado;
- Definir seus pontos de sincronização;
- Determinar quais partições poderão ser executadas em paralelo;
- Decidir qual a forma de comunicação será empregada de acordo com a arquitetura utilizada;

Se o programador não tiver uma boa experiência, ele poderá tentar paralelizar problemas que tenham uma alta taxa de dependências. Por isso, é necessário realizar uma análise do problema ou do algoritmo sequencial, verificando-se quais pontos podem ser executados em paralelo, assim como descobrir quais partes não são paralelizáveis.

2 Justificativa

O uso de computação paralela nos últimos anos tem se tornado mais presente no dia dia do desenvolvedor de software. Entretanto a tarefa de desenvolver aplicações que utilizam a arquitetura paralela não é trivial, demandando conhecimento, prática e paciência por parte do desenvolvedor. Diante das dificuldades apresentadas, é necessária uma ferramenta para facilitar o desenvolvimento de aplicações paralelas.

3 Objetivos

3.1 Objetivo geral

Desenvolver um Middleware para computação paralela (*Shared e Distributed Memory*) capaz de abstrair todo o conceito de tarefas concorrentes e recursos distribuídos do usuário.

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.

Objetiva-se assim conseguir as seguintes transparências:

Transparência de Acesso oculta diferenças na representação de dados e no modo como os recursos podem ser acessados por usuários;

Transparência de Localização oculta o lugar em que um recurso está localizado;

Transparência de Migração oculta que um recurso pode ser movido para outra localização;

Transparência de Relocação oculta que um recurso pode ser movido para outra localização enquanto em uso;

Transparência de Replicação oculta o fato de um recurso ter várias cópias;

Transparência de Concorrência oculta que o mesmo recurso esta sendo usado por vários usuários ao mesmo tempo;

Transparência de Falha oculta a falha e a recuperação de um recurso sem a percepção do usuário.

3.2 Objetivos específicos

- Desenvolver um middleware de computação paralela para abstração do usuário;
- Oferecer as transparências de Acesso, Localização, Migração, Relocação, Replicação, Concorrência e Falha de um SIstema Distribuído;
- Oferecer uma API juntamente com uma especificação para expansão do middleware;
- Utilizar o middleware para desenvolvimento de uma aplicação exemplo;
- Disponibilizar o middleware gratuitamente para utilização;
- Elaborar a sua documentação;

4 Metodologia

Inicialmente foi estudado técnicas de computação paralela e sistemas distribuídos. Depois foi realizada uma revisão bibliográfica de problemas e soluções semelhantes às apresentadas. A partir disso será implementado o middleware seguindo uma especificação que será totalmente documentada. O último passo será a disponibilização do middleware para utilização e construção de uma aplicação exemplo.

5 Cronograma de atividades

Na Tabela 1, é apresentado um cronograma tentativo para o desenvolvimento das atividades.

Atividades	**Ago**	**Set**	**Out**	**Nov**	**Dez**
Estudo de Técnicas	X	X			
Revisão Bibliográfica	X	X			
Implementação	X	X	X	X	X
Documentação			X	X	X
Aplicação Exemplo				X	X
Redigir a Monografia			X	X	X
Apresentação do Trabalho					X

Tabela 1: Cronograma de Atividades.

Referências

[1] Review of "highly parallel computing"by g. s. almasi and a. gottlieb, benjamin-cummings publishers, redwood city, ca, 1989. *IBM Syst. J.*, 29:165–166, January 1990. Reviewer-Lorin, Harold R.

[2] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *Int. J. Parallel Program.*, 30:65–98, April 2002.

[3] Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[4] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.