UNIVERSIDADE DO ESTADO DO AMAZONAS - UEA

ESCOLA SUPERIOR DE TECNOLOGIA

ENGENHARIA DE COMPUTAÇÃO

LANIER MENEZES DOS SANTOS

COMPARAÇÃO DE DESEMPENHO DA PROGRAMAÇÃO CONCORRENTE ENTRE JAVA E ERLANG UTILIZANDO INTEL MPI BENCHMARCK

Manaus

2011

LANIER MENEZES DOS SANTOS

COMPARAÇÃO DE DESEMPENHO DA PROGRAMAÇÃO CONCORRENTE ENTRE JAVA E ERLANG UTILIZANDO INTEL MPI BENCHMARCK

Trabalho de Conclusão de Curso apresentado à banca avaliadora do Curso de Engenharia de Computação, da Escola Superior de Tecnologia, da Universidade do Estado do Amazonas, como pré-requisito para obtenção do título de Engenheiro de Computação.

Orientador: Prof. M. Sc. Jucimar Maia da Silva Júnior

Manaus

2011

Universidade do Estado do Amazonas - UEA

Escola Superior de Tecnologia - EST

Reitor:

José Aldemir de Oliveira

Vice-Reitor:

Marly Guimarães Fernandes Costa

Diretor da Escola Superior de Tecnologia:

Mário Augusto Bessa de Figueirêdo

Coordenador do Curso de Engenharia de Computação:

Daniele Gordiano Valente

Coordenador da Disciplina Projeto Final:

Raimundo Correa de Oliveira

Banca Avaliadora composta por: *Data da Defesa: 16 / 12 / 2011.*

Prof. M.Sc. Jucimar Maia da Silva Júnior *(Orientador)*

Prof. M.Sc. Raimundo Correa de Oliveira

Prof. M.Sc. Jorge Luiz Silva Barros

CIP - Catalogação na Publicação

S237c SANTOS, Lanier

Comparação de Desempenho da Programação Concorrente Entre JAVA e ERLANG Utilizando Intel MPI Benchmarck/ Lanier Santos; [orientado por] Prof. MSc. Jucimar Maia da Silva Júnior - Manaus: UEA, 2011.

83 p.: il.; 30cm

Inclui Bibliografia

Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação). Universidade do Estado do Amazonas, 2011.

CDU: 004

LANIER MENEZES DOS SANTOS

COMPARAÇÃO DE DESEMPENHO DA PROGRAMAÇÃO CONCORRENTE ENTRE JAVA E ERLANG UTILIZANDO INTEL MPI BENCHMARCK

Trabalho de Conclusão de Curso apresentado à banca avaliadora do Curso de Engenharia de Computação, da Escola Superior de Tecnologia, da Universidade do Estado do Amazonas, como pré-requisito para obtenção do título de Engenheiro de Computação.

Aprovado em: 16 / 12 / 2011

BANCA EXAMINADORA

Prof. Jucimar Maia da Silva Júnior, Mestre
UNIVERSIDADE DO ESTADO DO AMAZONAS

Prof. Raimundo Correa de Oliveira, Mestre
UNIVERSIDADE DO ESTADO DO AMAZONAS

Prof. Jorge Luiz Silva Barros, Mestre
UNIVERSIDADE DO ESTADO DO AMAZONAS

Agradecimentos

Ao finalizar este trabalho, após um ano de dedicação, tive o prazer de contar com a amizade e o incentivo de pessoas que tornaram mais suave este caminho. A elas, agradeço:

Ao meu Orientador, Prof. Msc. Jucimar Maia da Silva Júnior, que sempre esteve presente e forneceu suporte para o desenvolvimento e conclusão deste trabalho.

Minha família, que ficaram privados de minha companhia diversas vezes, mas sempre me incentivaram e apoioaram.

Aos meus amigos de curso que sempre me ajudaram, em especial ao meu amigo Rodrigo Barros Bernardino, que me ajudou na etapa mais complexa deste experimento.

A todos sou muito grato.

Resumo

Este trabalho apresenta um estudo sobre o desempenho da programação concorrente entre as linguagens *Java e Erlang*. Ambas as linguagens são largamente utilizadas para aplicações diversas e que em muitos dos casos trata frequentemente de situações de paralelismo massivo, como as aplicações *web* que recebem diversas requisições de serviços em curto espaço de tempo. Utilizou-se um conjuntos de testes disponibilizados pela Intel®, conhecido como *IMB – Intel MPI Benchmarck* para manter um caráter idôneo na execução deste trabalho, visando avaliar o desempenho das linguagens. Existe uma descrição do IMB e uma descrição específica para cada *benchmarck* utilizado. Estes *benchmarcks* foram projetados para avaliar o desempenho de máquinas, então os mesmos foram reescritos para avaliar o desempenho e o comportamento das linguagens.

Palavras-Chave: Java, Erlang, IMB, Concorrência, *Benchmarck*

Abstract

This paper presents a study on the performance of concurrent programming between the languages *Java and Erlang*. Both languages are widely used for several applications and in many of cases often deals with situations of massive parallelism, as *web* applications, receiving several service requests in a short time. We used a set of tests provided by Intel®, known as *IMB - Intel MPI Benchmarck* to maintain a suitable character in carrying out this work, to evaluate the performance of language. There is a description of the IMB and a specific description for each *benchmarck* used. These *benchmarcks* were designed to evaluate the performance of machines, then they were rewritten to evaluate the performance and behavior of both languages *Java and Erlang*.

Key-words: Java, Erlang, IMB, Concurrency, *Benchmarck*

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Códigos

Capítulo 1

Introdução

Segundo a Lei de Moore, que descreve que o número de transistores usados na construção de um microprocessador, tal número dobra a cada 18 meses, logo em alguns anos não será possível construir microprocessadores com a atual arquitetura de semi-condutores. [Patterson2008]

Pela primeira vez na história ninguém mais está tentando construir uma nova geração de microprocessadores (também conhecidos como *cores*) mono-processados [Patterson2008]. Enquanto os *cores* mono-processados forneciam *hardwares* que atendiam às necessidades computacionais (até meados de 2005), todas as empresas que investiram em pesquisa sobre *hardware* paralelo, no fim da década de 60 e início de 70, faliram (ex. Convex, Encore, Inmos (Transputer), MasPar, NCUBE, Sequent), pois introduziam conceitos computacionais mais elaborados, uma forma diferente de programar, o que tornava o paralelismo uma alternativa pouco interessante; por estes motivos as pesquisas sobre paralelismo não tinham tanto espaço como têm agora. [Patterson2008]

A comunidade de desenvolvimento de *hardware* é unânime quanto a troca de paradigma para computação paralela; além de que, todas as grandes companhias que constroem microprocessadores (AMD, Intel, IBM, Sun) já vendem uma quantidade muito mais significativa de processadores paralelos *(multicores)* do que mono-processadores *(unicores)*. [Patterson2008]

Também já existe o planejamento de que *multicores* possam ter uma melhoria de 8% por ano no *clock* (frequência de funcionamento de um processador), e os processadores já

idealizados para o futuro são todos paralelos.

Considerando que não se invista em computação paralela, os computadores chegariam ao limite físico de aumento de performance, onde então chegaríamos a um ponto onde os computadores não evoluiriam mais; isso implica em uma queda acentuada na venda de computadores em todo o mundo, já que não haveriam computadores melhores que impulsionem o desejo de troca e não haveria tecnologia ultrapassada. Isso representa um colapso no setor de produção industrial de computadores mundial. [Patterson2008]

Um fator que contribui para a migração para arquitetura paralela é escalabilidade que nos é proporcionada. Quanto mais demanda tiver em um determinado sistema, o mesmo ainda pode alocar mais recurso físico para suprir a demanda, sem penalizar o desempenho.

Outro fator, novo na computação, é a possibilidade de escalabilidade inversa, ou seja, desalocar recurso quando não for necessário, afim de aumentar a vida útil do equipamento, economizar energia e dinheiro. [Patterson2008]

[Patterson2008] também fala a respeito de reescrever as bases da computação, criando um compilador que seja escalável, ou seja, que melhore seu desempenho a medida que aumenta o número de processadores disponíveis para uso. Nessa abordagem é onde este trabalho se encaixa utilizando linguagens de programação com suporte a concorrência e como se comportam quanto a escalabilidade. Essas linguagens provêm construções para a concorrência, estas construções podem envolver multitarefa (permite repartir a utilização do processador entre várias tarefas simultaneamente), suporte para sistemas distribuídos (processo realizado por dois ou mais computadores conectados através de uma rede com o objetivo de concluir uma tarefa em comum), troca de mensagens e recursos compartilhados.

Neste aspecto as linguagens de programação Java e Erlang são relevantes pois ambas são linguagens concorrentes (Erlang oferece suporte nativo à concorrência) e são linguagens usadas em diversas aplicações conhecidas e respeitadas.

Java se tornou uma das linguagens mais populares na comunidade de desenvolvimento, é uma linguagem orientada a objetos e permite programação concorrente, fornecendo recursos para controlar fluxos de execuções concorrentes. A máquina virtual Java e seu sistema operacional subjacente (SO) fornecem mapeamentos da simultaneidade aparente para o paralelismo físico (via múltiplas CPUs), permitindo que atividades independentes possam prosseguir em paralelo quando possível e desejável ou também por tempo compartilhado.

[Lea1999]

Erlang é uma linguagem de programação de propósito múltiplo, usado primeiramente para desenvolver sistemas concorrente e distribuídos. Erlang fornece um número de funcionalidades padrão não encontrados, ou de difícil manipulação em outras linguagens. Muitas destas funcionalidades existem devido as suas raízes de telecomunicação. Inclui um modelo de concorrência simples, permitindo blocos de códigos individuais serem executados múltiplas vezes na mesma máquina com relativa facilidade. Além do modelo de concorrência, usa também um modelo de erros que permite que falhas relacionadas aos seus processos sejam identificadas e tratadas, até mesmo por um novo processo, o que permite a construção de aplicativos altamente tolerante a falhas bem simples (ex. *hot swapping*). Conta também com um processamento distribuído incluso, permitindo que componentes sejam executados em uma determinada máquina mesmo sendo requisitado por uma máquina diferente. [Brown2011]

1.1 Justificativa

A tecnologia *Java* é uma programação de alto nível e uma linguagem de plataforma independente, foi projetado para trabalhar em ambientes distribuídos na internet. Possui funcionalidades de interface gráfica (GUI) que fornece melhor "look and feel" do que *C++*, além do mais é mais fácil de usar do que *C++* e trabalha no conceito do modelo de programação Orientado-Objeto. Permite jogar jogos *online*, sistemas multimídias (audio, vídeo), conversar com pessoas ao redor do mundo, aplicações bancárias, visualizar imagens 3D, carinho de compras. O uso extensivo do *Java* encontra-se no uso de aplicações de *intranet* e outras soluções *e-business* que são as raízes da computação corporativa. [RoseIndia2008]

Considerada como a linguagem mais bem descrito e planejada desenvolver aplicações para a Web, *Java* é uma tecnologia bem conhecida que permite a escrita e projeção de *software* apenas uma vez para uma *"máquina virtual"*, que permite executar em diferentes computadores, suporta vários Sistemas Operacionais como:

- Windows
- Macintosh

- Sistemas Unix

No aspecto da *Web*, *Java* é nos servidores Web, usado em muitos dos maiores *websites* interativos. Usado para criar aplicações independentes que podem ser executadas em um único computador ou em uma rede distribuída, também é usado para criar pequenas aplicações baseadas em *applets*, que posteriormente é usado para a página Web; *applets* possibilitam e facilitam a interatividade com a página web. [RoseIndia2008]

Tabela 1.1: Vantagens do Java [RoseIndia2008]

Características Java	
Simples	Arquitetura Neutra
Orientado a Objeto	Portável
Distribuído	Alta Performance
Multithread	Robusto
Dinâmico	Seguro

Por outro lado *Erlang* foi desenvolvida pela *Ericsson* para ajudar no desenvolvimento de *softwares* para gerenciar diferentes projetos de telecomunicações, tendo sua primeira versão lançada em 1986, e o primeiro lançamento *open-source* em 1998.

Erlang usa programação funcional, as funções e operações da linguagem são projetadas de modo similar aos cálculos matemáticos, assim a linguagem opera com funções que recebem entradas e geram resultados. O paradigma de programação funcional significa que o bloco individual de código pode produzir valores de saída consistentes para os mesmo valores de entrada, isso permite prever as saídas das funções ou programas mais facilmente e consequentemente mais fácil fazer o *"debug"* e analisar. [Brown2011]

Tornou-se mais popular recentemente devido seu uso em projetos de alto perfil, como: [Brown2011]

- Facebook (Sistema de *chat*)
- CouchDB (Documentação orientada a sistemas gerenciadores de banco de dados)

1.2 Objetivos Gerais

O objetivo deste trabalho é testar e avaliar o desempenho de execução concorrente entre as linguagens de programação Java e Erlang, afim de se fornecer uma métrica, uma forma de avaliar cada linguagem, de modo que possa ser um meio de decisão qual plataforma usar para determinada aplicação.

1.2.1 Objetivos Específicos

- Reescrever o software de *Benchmarck* fornecido pela Intel em Java.
- Reescrever o software de *Benchmarck* fornecido pela Intel em Erlang.
- Executar e Avaliar o desempenho concorrente de cada uma linguagem executando sob as mesmas condições o mesmo software.

1.3 Metodologia

- Leitura e análise do IMB - *Intel MPI Benchmarck* na versão original, escrito em *C*, reescrevê-lo em cada uma das linguagens que se deseja avaliar, no caso *Java* e *Erlang*.
- Executar cada um dos módulos reescritos na mesma condição física de execução, de forma a oferecer o mínimo de interferência possível para cada teste.
- Criar *scripts* para cada uma das linguagens para executar os programas de forma automatizada para repetir 10 (dez) vezes a execução de cada módulo e salvar as saídas em arquivos de texto para análise posterior.

Capítulo 2

Concorrência

Cada vez mais os *softwares* exigem mais recurso de *hardware* para executarem normalmente. As configurações mínimas para executar alguns programas tem crescido bastante, se comparado com as da década de 2000. Quanto mais processamento é necessário para executar certa tarefa, mais nos relacionamos com questões do universo concorrente, paralelo e/ou distribuído.

Entretanto para melhor compreender os sistemas concorrentes e seus conceitos relacionados é preciso que antes sejam vistos alguns conceitos de arquiteturas paralelas.

2.1 Taxonomia de Flynn de Arquiteturas Paralelas

Computadores paralelos têm sido usados por muito anos, e muitas alternativas arquiteturais diferentes foram propostas e usadas. Em geral um comutador paralelo pode ser caracterizado como uma coleção de elementos processos que podem comunicar e cooperar para resolver grandes problemas rapidamente. Esta definição é intencionalmente vaga para capturar uma grande variedade de plataformas paralelas. Muitos detalhes importantes não são endereçados pela definição, incluindo o número e a complexidade dos processadores, a estrutura de rede de interconexão entre os processadores bem como características importantes do problema a ser resolvido.

Para uma investigação mais detalhada, é útil fazer uma classificação de acordo com

características importantes do computador paralelo. Um modelo simples para tal classificação é dada pela **Taxonomia de Flynn**. Esta taxonomia caracteriza computadores paralelos de acordo com o controle global e os dados resultantes e o controle de fluxo de instruções e dados. [Rauber and Runger2010]

São distinguidas quatro características:

1. Single-Instruction, Single-Data (SISD)
2. Multiple-Instruction, Single-Data (MISD)
3. Single-Instruction, Multiple-Data (SIMD)
4. Multiple-Instruction, Multiple-Data (MIMD)

2.1.1 Single-Instruction, Single-Data (SISD)

Instrução-Única, Dado-Único: Existe um elemento processo, que tem acesso a um único programa e armazenador de dados. Em cada etapa, o elemento processo carrega uma instrução e o dado correspondente e executa a instrução. O resultado é armazenado de volta no armazenador de dados. Assim, de acordo com o *modelo de von Neumann*, SISD é a computação sequencial convencional.

2.1.2 Multiple-Instruction, Single-Data (MISD)

Múltipla-Instrução, Dado-Único: Existem múltiplos elementos processos, cada um possui uma memória de programa privado, mas existe apenas um acesso comum a uma única memória de dados global. Em cada etapa, cada elemento processo obtém o *mesmo* elemento de dado da memória de dado e carrega uma instrução da sua memória de programa privada. Estas instruções, possivelmente diferentes, são executadas então em paralelo pelo elemento processo usando o elemento de dado (idêntico) obtido previamente como operador. Este modelo de execução é muito restrito e nunca um computador paralelo comercial desse tipo foi construído.

2.1.3 Single-Instruction, Multiple-Data (SIMD)

Instrução-Única, Dado-Múltiplo: Existem múltiplos elementos processos, onde cada um possui um acesso privado a uma memória de dados (compartilhada ou distribuída), mas existe apenas uma memória de programa, do qual um processador de controle especial busca e despacha instruções. Em cada etapa, cada elemento processo obtém do processador de controle a *mesma* instrução e carrega um elemento de dado separado por seu acesso privado de dados no qual a instrução é executada. Assim, a instrução é síncronamente aplicada em paralelo por todos os elementos processos para diferentes elementos de dados.

Para aplicações com um significante grau de paralelismo de dados, a abordagem SIMD pode ser muito eficiente. Exemplos são aplicações multimídia ou algoritmos de computação gráfica para gerar uma visão tri-dimensional realista para ambientes gerados por computador.

2.1.4 Multiple-Instruction, Multiple-Data (MIMD)

Instrução-Múltipla, Dado-Múltiplo: Existem múltiplos elementos de processos, cada um possui um acesso separado a instrução e dados em uma (compartilhado ou distribuído) memória de dados e programa. Em cada etapa, cada elemento de processo carrega uma instrução separada e um elementos de dado separado, aplica a instrução no elemento de dado, e armazena um possível resultado de volta ao armazenador de dados. Os elementos processos trabalham assíncronamente entre si. Processadores *multicores* ou sistemas de *cluster* são exemplos do modelo MIMD.

Comparados aos computadores MIMD, os computadores SIMD têm a vantagem de serem fáceis de programar, desde que haja apenas um fluxo de programa e a execução síncrona não requeira sincronização a nível de programa *software*. Mas a execução síncrona é também uma restrição, desde que afirmações condicionais da forma:

```
if (b == 0)
  c = a;
else
  c = a/b;
```

devem ser executadas em dois passos. no primeiro passo, todos os elementos processos cujo valor local de b é zero executa a parte *then*. No segundo passo, todos os outros elementos processos executam a parte *else*. Computadores MIMD são mais flexíveis, como cada elemento processo pode executar seu próprio fluxo de programa. A maioria dos computadores paralelos são baseados no conceito MIMD. Apesar da taxonomia de Flynn fornecer apenas uma classificação grosseira, é útil para dar uma visão geral no espaço de projeto de computadores paralelos. [Rauber and Runger2010]

2.2 Lei de Amdahl

Existem casos de tarefas que ao serem executadas possuem porções paralelizáveis e porções que precisam ser executadas de forma sequencial. Um computador paralelo operando de forma sequencial é um grande desperdício, pois enquanto um processador trabalha no trecho serial, todos os demais ficam ociosos.

Levando em consideração essa situação de ociosidade temporária, utiliza-se a Lei de Amdahl para medir qual é o ganho de desempenho do computador durante o período de tempo em que trabalha realmente de forma paralela. Esta é a lei que governa o *speedup* (aceleração) na utilização de processadores paralelos em relação ao uso de apenas um processador; seu nome deriva do arquiteto de computadores **Gene Amdahl**.

O ganho de desempenho que pode ser obtido melhorando uma determinada parte do sistema é limitado pela fração de tempo que essa parte é utilizada pelo sistema durante a sua operação e depende do número de processadores usados para aplicar o paralelismo. [Liria Matsumoto Sato et al.1996]

- Se f é a fração de operações sequenciais de uma computação a ser resolvida por p processadores, então o ganho de aceleração *(speedup)* S é limitado de acordo com a fórmula:

$$S \leq \frac{1}{f + \frac{(1-f)}{p}} \tag{2.1}$$

2.2.1 *Speedup*

O *speedup* (S) obtido por um algoritmo paralelo executando sobre p processadores; é a razão entre o tempo levado por aquele computador executando o algoritmo serial mais rápido (T_s) e o tempo levado pelo mesmo computador executando o algoritmo paralelo usando p processadores (T_p).

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

2.2.2 Eficiência

A eficiência (E) é a razão entre o *speedup* obtido na a execução com p processadores e o número de processadores p. Esta medida mostra o quanto o paralelismo foi explorado no algoritmo.

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

2.3 Contextualização

Uma vez visto os conceitos das arquiteturas paralelas de *Flynn* e as regras que avaliam as partes paralelizáveis dos *softwares* definidas pelas leis de *Amdahl*, é possível introduzir os conceitos de Concorrência e Paralelismo.

2.3.1 Concorrência

Concorrência é a capacidade de se executar duas ou mais tarefas em um mesmo período de tempo. Estas tarefas progridem neste período de tempo e o que compartilham são os recursos do sistema (CPU, memória, disco), mas não compartilham um mesmo problema a ser resolvido, sendo logicamente independentes.

A concorrência tornou-se uma área de grande importância na computação. A capacidade de colocar um grande poder de computação em um pequeno chip fez com que os multi-processadores se tornassem um lugar comum.

A gestão da concorrência entre processos é a fonte de inúmeras dificuldades no desenvolvimento de software; se processos concorrentes não compartilham nenhum recurso, não existirá problema algum associado. O problema ocorre quando os processos necessitam acessar recursos comuns, tal como memória partilhada. As interações existentes podem levar à acessos descoordenados a um recurso *(condição de corrida)* [Tanembaum2007], situações onde a ordem de execução dos processos no tempo determina o resultado.

Condição de corrida: situações onde dois ou mais processos estão acessando dados compartilhados, e o resultado final do processo depende da ordem de execução dos processos.

A condição de corrida leva a um comportamento que não pode, em geral, ser reproduzido, chamado de comportamento não-determinístico ou não-funcional. Para evitar a condição de corrida, introduz-se o conceito de *região crítica*, ou seja, um trecho de código onde o processo está trabalhando em algum recurso compartilhado. O objetivo da região crítica é impedir que outro processo manipule a mesma entidade antes que o processo termine seu trabalho sobre ela [Tanembaum2007].

Os processos podem precisar comunicar-se com outros processos, usando para tanto primitivas para comunicação entre processos. Essas primitivas são usadas para garantir que dois processos não estejam nunca em suas regiões críticas correspondentes ao mesmo tempo, garantindo assim a exclusão mútua.

Exclusão mútua: se um processo está sendo executado em sua região crítica, deve-se impedir que todos os outros processos entrem em suas próprias regiões críticas. Reciprocamente, não pode-se permitir que um processo entre em sua região crítica se qualquer outro processo estiver em sua própria região crítica, ou seja, evita-se condições de corrida.

Segundo [Tanembaum2007], os processos precisam frequentemente se comunicar com outros processos.

Define-se sincronização de processos como sendo a sequencialização forçada de eventos executados por processos concorrentes assíncronos [Py2009]. Supõe-se que dois processos estejam executando concorrente e assíncronamente, de maneira que cada um execute um evento. Como os processos são concorrentes e assíncronos os eventos podem ocorrer a qualquer instante e em qualquer ordem, inclusive simultaneamente.

Dois processos podem ainda ser unificados a partir de um determinado ponto, chamado ponto de sincronização. Isto deu origem ao comando *join*. A sincronização possibilita a

definição de comandos que definem dois tipos de sincronização:

- Síncrona
- Assíncrona

Essa diferenciação esta vinculada ao fato do processador ficar aguardando a informação ou resposta, ou continuar processando. [Py2009]

2.3.2 Paralelismo

Computação paralela é uma forma de computação em que vários cálculos são realizados simultaneamente, operando sob o princípio de que grandes problemas geralmente podem ser divididos em problemas menores, que então são resolvidos concorrentemente (em paralelo).

Seria o tratamento de uma tarefa complexa, que é dividida em um conjunto de tarefas menores relacionadas que cooperam entre si para a realização da tarefa maior.

Tais tarefas são processadas de maneira independente e simultânea em múltiplas unidades de processamento (*cores* de CPU's).

Existem diferentes formas de computação paralela:

- Computação paralela em bit
- Computação paralela em instrução
- Computação paralela de dado
- Computação paralela de tarefa

A técnica de paralelismo já é empregada por vários anos, principalmente na computação de alto desempenho, mas recentemente o interesse no tema cresceu devido às limitações físicas que previnem o aumento de frequência de processamento.

Com o aumento da preocupação do consumo de energia dos computadores, a computação paralela se tornou o paradigma dominante nas arquiteturas de computadores sob forma de processadores multi-núcleo.

Computadores com multi-núcleos ou multi-processadores possuem múltiplos elementos de processamento em somente uma máquina, enquanto *clusters*, MPP e grades usam múltiplos computadores para trabalhar em uma única tarefa. É possível classificar as máquinas paralelas de acordo com o nível em que o hardware suporta o paralelismo.

Arquiteturas paralelas especializadas às vezes são usadas junto com processadores tradicionais, para acelerar tarefas específicas.

A programação de sistemas que façam uso do potencial disponibilizado pelo hardware paralelo é uma questão em profunda discussão na comunidade de computação. Estes sistemas são mais difíceis de programar, quando comparados ao paradigma sequencial, pois a concorrência introduz diversas novas classes de defeitos potenciais, como a condição de corrida, explicada anteriormente.

A comunicação e a sincronização entre diferentes sub-tarefas é tipicamente uma das maiores barreiras para atingir grande desempenho em programas paralelos.

A computação paralela permite um grande ganho de velocidade em relação à computação sequencial, tal ganho pode ser avaliada segundo a **Lei de Amdahl**.

2.4 Concorrência em Java

A criação de processos é a base fundamental para a existência da concorrência computacional. A plataforma Java foi projetada desde o início para suportar a programação concorrente, com suporte básico concorrente na linguagem de programação *Java* e as bibliotecas de classes *Java*. Desde a versão 5.0, a plataforma *Java* tem incluído também API's de alto-nível de concorrência, como o pacote: [Oracle Corporation]

`java.util.concurrent`

2.4.1 Processos e Threads

Na linguagem de programação *Java*, a programação concorrente é geralmente feita com o uso de *threads*, apesar de processos também serem importantes. Em um sistema, normalmente, existem muitos processos e threads ativas, mesmo que seja apenas um executado

em um dado tempo, conhecido como "fatia de tempo" ou *time slicing*, que fica a cargo do Sistema Operacional (SO). [Oracle Corporation]

Processos

Processos têm um ambiente próprio de execução integrado com recursos de execução e espaço de memória, geralmente são usados em programas ou aplicações. A maioria dos SO's suportam recursos de IPC *Internal Process Communication* (Comunicação Interna de Processos), como *pipes e sockets*.

A maioria das implementações da máquina virtual Java executa como um único processo, mas é possível criar mais processos usando um ***ProcessBuilder*** (construtor de Processos).

Threads

As *threads*, também chamadas de *lightweight processes* (processos leves), como o processo, fornece um ambiente de execução, mas exige menos recursos computacionais do que um processo. *Threads* existem agregadas a um processo compartilhando os recursos do processo para uma comunicação eficiente, porém potencialmente problemática. [Oracle Corporation]

A execução *multi-thread* é uma funcionalidade essencial na plataforma *Java*, toda aplicação possui, no mínimo, uma *thread* ou várias, mas do ponto de vista do programador, a princípio existe apenas uma *thread* chamada de *main thread* (*thread* principal) e esta pode criar novas *threads*. [Oracle Corporation]

2.4.2 Objetos Threads

Cada *thread* é associada com um instância de uma classe ***Thread***, existem duas formas de criar uma aplicação concorrente: [Oracle Corporation]

1. Gerenciar a criação de *threads* diretamente, simplesmente instanciando a classe ***Thread*** a cada vez que a aplicação precisar executar uma tarefa assíncrona

2. Abstrair o gerenciamento das *threads*, passando as tarefas da aplicação para um ***executor***

2.4.3 Iniciando uma *Thread*

Uma aplicação que crie uma instância da classe ***Thread*** deve fornecer o código que irá executar nesta *thread*. Pode ser feito de duas maneiras: [Oracle Corporation]

1. Fornecer um objeto ***Runnable***, esta interface define um único método ***run***, que contém o código que executará na *thread*, então o objeto ***Runnable*** é passado para o construtor da classe ***Thread***.

```
public class HelloRunnable implements Runnable {
     public void run() {
            System.out.println("Ola de uma thread!");
     }

     public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
     }
  }
```

2. Utilizar uma sub-classe ***Thread***; a classe ***Thread*** implementa a interface ***Runnable*** por si, por isso o método ***run*** não executa nenhum código. Assim, uma aplicação pode obter uma sub-classe ***Thread*** fornecendo sua própria implementação do método ***run***.

```
public class HelloThread extends Thread {
     public void run() {
        System.out.println("Ola de uma thread!");
     }

     public static void main(String args[]) {
        (new HelloThread()).start();
```

```
        }
    }
```

Em ambos os modos é invocado ***Thread.start*** para iniciar as *threads*. [Oracle Corporation]

2.4.4 Sincronização

Threads possuem sua própria pilha da chamadas de funções mas também podem acessar dados compartilhados, com isso existem dois problemas básicos: [Vogel2011]

1. Visibilidade
2. Acesso

O problema de visibilidade ocorre seu uma *thread* lê um dado compartilhado que posteriormente é alterado por outra *thread* e a primeira *thread* que leu o dado não é capaz de ver o novo valor assumido pelo dado compartilhado.

O problema de aceso ocorre se muitas *threads* tentam acessar e mudar o valor de um mesmo dado compartilhado ao mesmo tempo. [Vogel2011]

Estes dos problemas podem levar a aplicação às seguintes falhas: [Vogel2011]

- Falha de Existência – O programa deixa de reagir devido aos problemas no aceso concorrente dos dados (*deadlocks*)
- Falha de Segurança – O programa gera dados inconsistentes

Um *deadlock* ocorre quando duas ou mais *threads* estão bloqueadas esperando obter uma trava que uma das *threads* que também estão em *deadlock* possui. Também pode ocorrer quando múltiplas *threads* precisam de uma mesma trava, ao mesmo tempo, mas as mesmas obtém as travas em ordem diferentes. [Jenkov]

Java fornece travas que protegem cartas partes do código de serem executadas ao mesmo tempo por múltiplas threads. A forma mais simples de proteger parte do código ou uma classe Java é usar o ***sychronized*** em um método ou na declaração de uma classe.

O uso do ***sychronized*** garante: [Vogel2011]

- Apenas uma única *thread* pode executar um bloco do código ao mesmo tempo
- Toda *thread* que executar um bloco sincronizado pode ver os efeitos de todas as modificações anteriores que foram guardadas pela mesma trava.

A sincronização é necessária para o acesso exclusivo-mútuo do blocos de códigos e para uma comunicação confiável entre as *threads*. [Vogel2011] É possível sincronizar um método ou um bloco de código, como mostrado a seguir: [Jenkov]

- Sincronização de Método

```
public synchronized void add(int value) {
     this.count += value;
     }
```

- Sincronização de Bloco de Código

```
public void add(int value){
     synchronized(this){
       this.count += value;
     }
  }
```

2.5 Concorrência em Erlang

2.5.1 Criação de Processos

Um processo é auto-suficiente, unidade de computação separada que existe concorrentemente com outros processos no sistema. Não existe hierarquia entra os processos, o projetista da aplicação deve explicitar a criação de tal hierarquia. Uma *Built-In-Function – BIF* "Função Integrada" *spawn/3* cria e inicializa a execução de um novo processo. [Armstrong et al.1996]

A *BIF – spawn/3* cria um processo concorrente para avaliar a função e retorna o *Process Identifier - Pid* do processo criado. Os *Pid's* são usados para todas as formas de comunicação entre processos. No caso de mais de um processo criado pela *BIF – spawn/3* para executarem concorrentemente, apenas os processos criados conhecem os *Pid's* de cada um simultaneamente, como estes são indispensáveis para realizar a comunicação entre os processos, a segurança nos sistemas Erlang é baseada na restrição da distribuição dos *Pid's* dos processos. [Armstrong et al.1996]

```
Pid = spawn(Modulo, NomeFuncao, ListaArgumento)
```

2.5.2 Comunicação Entre Processos

Em Erlang a única forma possível de comunicar processos é através de *passagem de mensagem*, ou seja, uma mensagem é enviada de um processo para outro usando a primitiva *"!" (send)*. Uma mensagem pode ser caracterizada por qualquer termo Erlang válido, a primitiva ***send*** avalia seus argumentos e retorna o valor ***sent***. Enviar uma mensagem é uma tarefa assíncrona, dessa forma a chamada ***send*** não espera pela chegada da mensagem ao destino ou ser recebida. Mesmo que o processo alvo de um envio de mensagem não exista quando a mesma chegar, o sistema não notificará o processo remetente da mensagem, pois faz parte da natureza assíncrona de passagem de mensagem, a aplicação deve implementar uma forma de verificar tal situação. [Armstrong et al.1996]

```
Pid !  Mensagem
```

Cada processo Erlang possui uma caixa de mensagem *"mailbox"* e todas as mensagens enviadas à um processo são armazenada nessa caixa de mensagens respeitando a ordem de recebimento de cada mensagem. Quando uma mensagem é encontrada e existe um ***Guard*** correspondente, esta mensagem é selecionada, retirada da *mailbox* e sua ***Action*** é avaliada, então o ***receive*** retorna o valor da última expressão avaliada na ***Action***, nenhuma mensagem bloqueia uma outra mesmo que sejam mensagens inesperadas para um processo. Existem mensagens que podem não serem selecionadas pelo ***receive***, devido não existir ***Guard*** correspondente, e permanecem no *mailbox*; é responsabilidade do desenvolvedor garantir que o sistema não se preencha com este tipo de mensagem. [Armstrong et al.1996]

```
receive
  Message [when Guard] -> 
    Action;
end
```

2.5.3 Escalonamento de Processos, Tempo Real e Prioridades

O escalonamento de processos no Erlang depende da sua implementação, mas existem alguns critérios que precisam ser satisfeitos: [Armstrong et al.1996]

- O algoritmo de escalonamento escalonamento deve ser justo, ou seja, qualquer processo que possa ser executado será executado, se possível ma mesma ordem em que tornou-se executável.
- Não será permitido à nenhum processo bloquear a máquina por um longo período de tempo, o processo pode executar por um curto período de tempo, chamado de *time slice* (fatia de tempo), antes de ser re-escalado e permitir a execução de outro processo.

As fatias de tempo são configuradas para permitir que um processo em execução execute aproximadamente 500 chamadas de função, depois disso o mesmo é re-escalado.

Uma outra funcionalidade importante para sistemas Erlang empregados em aplicações de *Tempo Real* é o gerenciamento de memória, pois é feita de forma implícita, transparente do ponto de vista do usuário. [Armstrong et al.1996]

Recursos de memória são alocados automaticamente no momento em que se faz necessário para uma nova estrutura de dados e é desalocado posteriormente quando a estrutura não é mais necessária, esse gerenciamento de memória deve ser feito de modo a não bloquear o sistema por um período de tempo, se possível, maior do que a fatia de tempo de um processo, pois assim a natureza de *tempo real* da aplicação não é afetada.

A princípio, todos os processos criados são executados com a mesma prioridade, porém, algumas vezes é necessário que alguns processos sejam executados mais frequentemente que

outros. Para mudar a prioridade de execução de um processo a BIF *process_flag* é usada. Os processos podem possuir dois valores de prioridade: Normal e Baixa [Armstrong et al.1996]

Processos marcados com prioridade *Baixa* executam com menor frequência que os marcados com prioridade *Normal*, entretanto, o valor padrão de prioridade dos processos Erlang é *Normal*. [Armstrong et al.1996]

```
process_flag(priority, Normal)
```

Capítulo 3

Intel MPI Benchmarck

Um *Benchmark* é um programa de teste de desempenho que analisa as características de processamento e de movimentação de dados de um sistema de computação, com o objetivo de medir ou prever seu desempenho e revelar os pontos fortes e fracos de sua arquitetura. Podem ser classificados de acordo com a classe de aplicação para a qual são voltados como, por exemplo, computação científica, serviços de rede, aplicações multimídia, processamento de sinais entre outros.

A comunicação entre os processos pode ser realizada, mediante troca de mensagens ou usando memória compartilhada. Na forma de memória compartilhada permite que diversos processos executem em uma mesma arquitetura de hardware concorrendo ao uso de seus recursos. Para que isso funcione adequadamente, o escalonador de processos do sistema operacional deve ser capaz de bloquear e desbloquear processos, afim de realizar a troca de contexto, como é mostrado na Figura 3.1. [Tanembaum2007]

Em se tratando de troca de mensagens, utiliza-se protocolo de comunicação assíncrona, de forma que o remetente e o destinatário da mensagem não precisam interagir ao mesmo tempo, as mensagens são enfileiradas e armazenadas até que o destinatário as processe. A maioria das filas de mensagens definem limites ao tamanho dos dados que podem ser transmitidos numa única mensagem, as que não possuem tal limite são chamadas caixas de mensagens, como ilustrado na Figura 3.2.

Neste trabalho será utilizado o *benchmark* conhecido por *IMB*; originalmente usado para medir a performance de grandes servidores e/ou *clusters* de computadores.

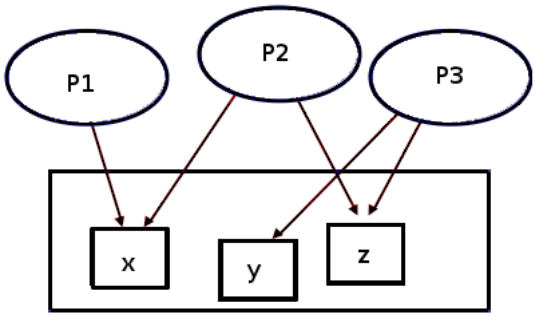

Figura 3.1: Abstração de Compartilhamento de Memória (ex. Java e C#)

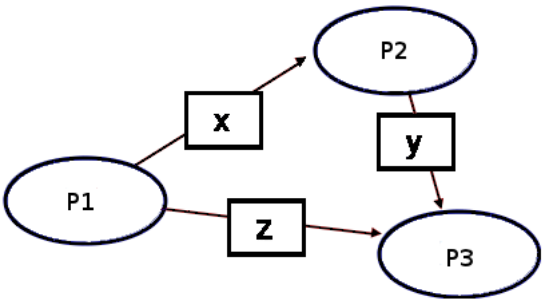

Figura 3.2: Abstração de Troca de Mensagens (ex. Erlang e Occam)

3.1 MPI - Message Passing Interface

O MPI é uma biblioteca padrão de passagem de mensagem baseado no consenso do Fórum de MPI (MPIF), o qual contou com cerca de 175 pessoas de aproximadamente 40 organizações participantes, entre estes haviam vendedores, pesquisadores, desenvolvedores de bibliotecas de *softwares* e usuários. O objetivo do MPI é estabelecer uma biblioteca padrão de passagem de mensagem que seja:

- Prática
- Portável
- Eficiente
- Flexível

Esta biblioteca seria largamente utilizada para escrever programas de passagem de mensagem. O MPI não é um padrão IEEE ou ISO, tornou-se o "padrão industrial" para escrever estes tipos de *softwares*. O MPI em si não é uma biblioteca, mas é uma especificação do que a biblioteca deveria ser, estas especificações foram projetadas para desenvolvimento em C/C++ e Fortran.

O rascunho final do MPI foi divulgado em 1994 [MPI2009a], ainda houve uma melhora no padrão disponibilizada em 1996, o MPI-2 e a primeira versão do MPI ficou conhecida como MPI-1, o MPIF agora discute uma nova versão MPI-3 [MPI2009b], mas até agora as implementações de MPI são uma combinação do MPI-1 e MPI-2.

3.2 Testes - IMB

IMB (*Intel MPI Benchmark*) é um conjunto de *benchmarks* desenvolvido pela Intel para avaliar a eficiência das mais importantes funções do MPI (*Message Passing Interface*), bem como o desempenho de um conjunto de processadores executando algoritmos concorrentes. Os testes são divididos em três categorias:

- Transferência Simples
  - Uma única mensagem é trocada entre dois processos
- Transferência Paralela
  - Uma única mensagem é trocada entre dois processos, porém existem vários pares de processos executando simultaneamente
- Transferência Coletiva
  - Vários processos trabalham em conjunto para realizar uma tarefa

Elas indicam as formas com que se deve interpretar os resultados e como deve ser a estruturação do código. A Tabela 3.1 mostra todos os testes IMB. [Intel Corporation Document Number: 320714-0022010]

Tabela 3.1: Tabela de classe dos testes IMB

Transferência Simples	Transferência Paralela	Coletiva
PingPong	Sendrecv	Bcast
PingPongSpecificSource	Exchange	Allgather
PingPing	Multi-PingPong	Allgatherv
PingPingSpecificSource	Multi-PingPing	Alltoall
	Multi-Sendrecv	Alltoallv
	Multi-Exchange	Scatter
		Scatterv
		Gather
		Gatherv
		Reduce
		Reduce_scatter
		Allreduce
		Barrier

3.3 PingPing

O principal objetivo é medir a eficiência no tratamento de obstruções. Uma obstrução ocorre quando um processo recebe uma mensagem no momento em que envia uma outra.

Registra, também, o tempo para se iniciar processos e a vazão de mensagens que o sistema fornece, (processamento de cada mensagem).

3.3.1 Descrição

Neste teste, um processo A envia uma mensagem de tamanho x *bytes* para o outro processo B e, simultaneamente, B envia a mesma mensagem para o processo A, como mostrado na Figura 3.3.

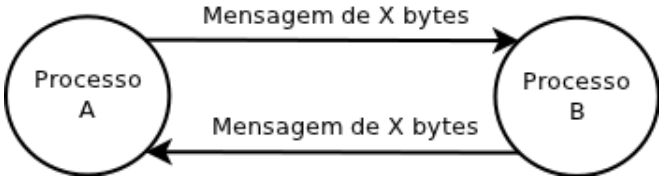

Figura 3.3: Teste PingPing

3.3.2 Medições

São medidos os tempos para se iniciar os processos e o tempo total desde o envio até que todos recebam suas respectivas mensagens.

3.3.3 Detalhes Técnicos

Cada *benchmark* é executado com tamanhos de mensagens variantes de *x bytes*, e as medidas de tempo são uma média de múltiplas amostras. Está é a visão de uma única amostra, com uma mensagem fixa de tamanho *x bytes*.

Valores de Processamento são definidos em:

$$MBytes/seg = 2^{20} bytes/seg \tag{3.1}$$

$$Processamentos = X/2^{20} * 10^6 \tag{3.2}$$

$$tempo = X/1,048576/tempo \tag{3.3}$$

3.4 PingPong

Juntamente com o *PingPing*, são formas clássicas de medir o processamento de uma única mensagem enviada entre dois processos.

3.4.1 Descrição

O *benchmark PingPong* funciona de maneira análoga ao *benchmark PingPing*. A diferença é que, um *processo A* envia uma mensagem de tamanho *x bytes* para um *processo B* e este, por sua vez, recebe a mensagem de *A* e então responde para o *processo A* com outra mensagem de tamanho *x bytes* e vice-versa. A Figura 3.4 representa este comportamento.

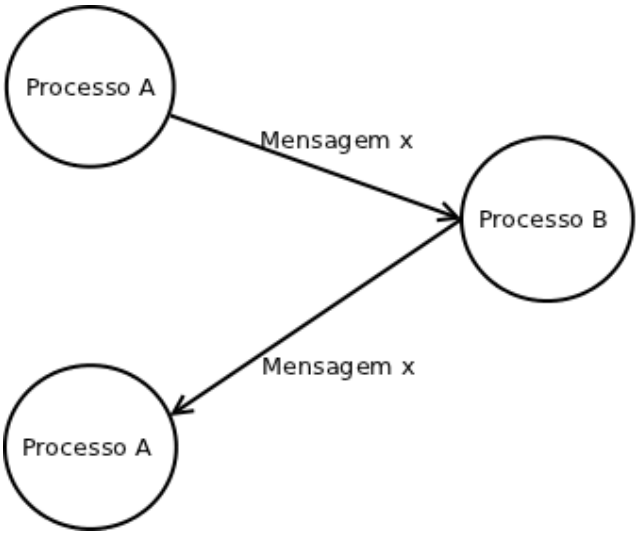

Figura 3.4: Teste PingPong

3.4.2 Medições

As medições realizadas são as mesmas que são feitas no *benchmark PingPing*.

3.4.3 Detalhes Técnicos

O *PingPong* é um *benchmark* da categoria de Transferência Simples, assim como o *PingPing*. Dessa forma estes dois testes compartilham dos mesmos detalhes técnicos.

3.5 Sendrecv

Este é um *benchmark* transferência paralela, ou seja, a atividade em um certo processo está em concorrência com outros processos, e deste modo o *benchmark* mede a eficiência do transcurso da mensagem sob a carga global.

3.5.1 Descrição

Vários processos formam uma corrente de comunicação periódica. Cada processo envia uma mensagem de tamanho x *bytes* para o processo à direita e recebe uma mensagem de tamanho x *bytes* do proceso à esquerda na corrente, como apresentado na Figura 3.5.

A contagem de ações são duas mensagens por amostra (uma entra e uma sai) para cada processo.

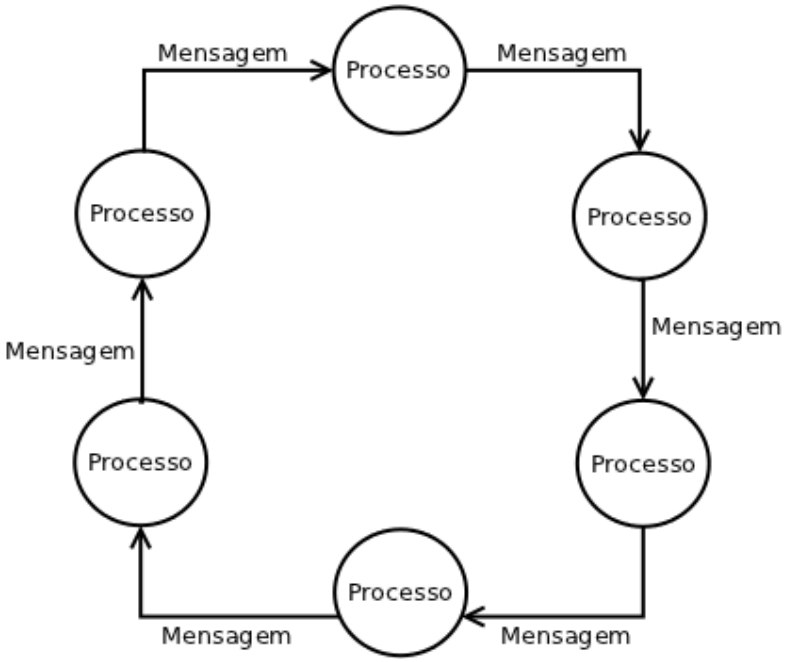

Figura 3.5: Teste Sendrecv ou Anel de Processos

3.5.2 Medições

O cálculo do processamento do *benchmark* descrito aqui leva em consideração a (por amostra) multiplicidade *nmsg* da mensagem que enviou de entrada de um processo em particular.

No *benchmark Sendrecv*, um processo em particular envia e recebe *x bytes*, esta ação é *2x bytes*, *nmsg = 2*.

3.5.3 Detalhes Técnicos

Cada *benchmark* é executado com mensagens variantes de tamanho *x bytes*, apesar de demonstrarmos aqui apenas um exemplo com um tamanho fixo de *x bytes*.

$$MBytes/seg = 2^{20} bytes/seg \tag{3.4}$$

$$Processamento = nmsg * x/2^{20} * 10^6 \tag{3.5}$$

$$tempo = nmsg * x/1.048576/tempo,$$

(quando o tempo está em μseg) (3.6)

3.6 AlltoAll

Este é um *benchmark* classificado como transferência coletiva, onde cada processo tem uma ligação com os outros processos existentes e a ação de cada um deles interfere de alguma forma em um outro processo.

3.6.1 Descrição

Muitos processos são criados ao início do *benchmark*; cada um destes processos envia uma mensagem de tamanho x *bytes* pra todos os outros processos criados, e recebe uma mensagem de tamanho x *bytes* de cada um dos outros processos. Ilustrado na Figura 3.6

3.6.2 Medições

Neste *benchmark* são medidos apenas os tempos de execução, nenhum processamento é levado em consideração.

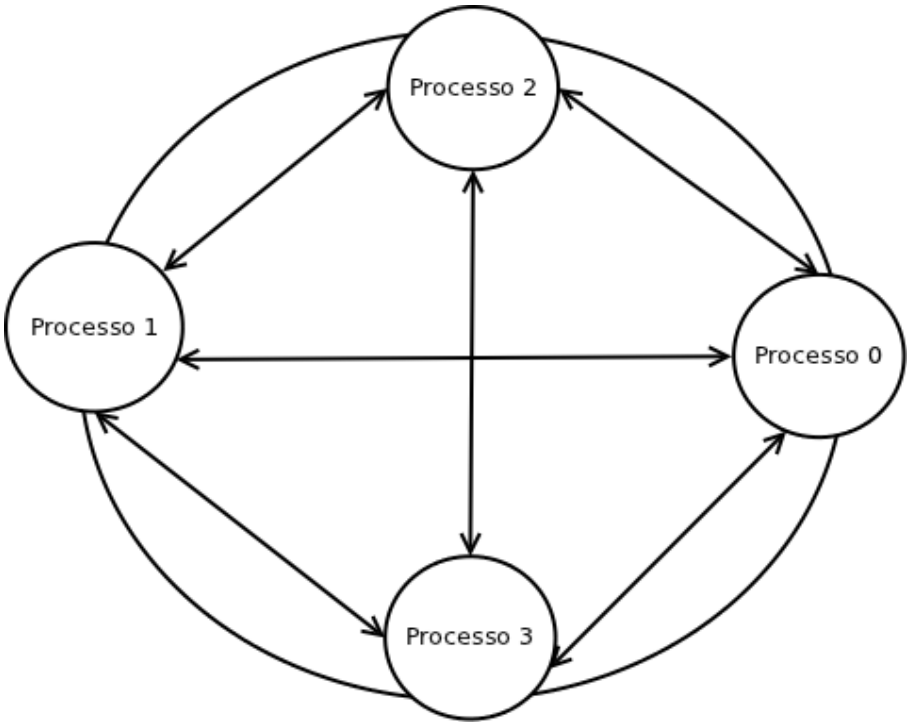

Figura 3.6: Teste AlltoAll

Capítulo 4

Ambiente de Execução

4.1 Configurações

Cada conjunto de testes de cada *benchmarck* foi executado em uma mesma máquina com a seguinte configuração:

4.1.1 Sistema Operacional

- Ubuntu – 11.04 (natty)
- GNU/Linux – 2.6.38-8-server x86_64

4.1.2 Hardware

- Memória – 4 Gb - DDR2
- Disco Rígido – 250 Gb - Sata 2
- Processador – Intel® Core™2 Duo CPU E7400 - 2.8 GHz

4.1.3 Linguagens

- Java

  - Versão - 1.6.0_25
  - Java HotSpot (TM) 64-Bit Server - Build 20.1-b02

- Erlang
  - Erlang R14B03 - (64-bits)
  - Eshell Versão - 5.8.4

4.2 PingPing

Este *benchmarck* foi realizado para valores diferentes de:

- Tamanho de Mensagem
- Número de Repetições

Cada conjunto de resultados foi classificado, por Tamanho de Mensagens Enviadas como é possível ver no Capítulo 5.

4.3 PingPong

Para este *benchmarck*, como visto no capítulo anterior, assim como o *PingPing* também é um *benchmarck* de Transferência Simples [Intel Corporation Document Number: 320714-0022010], logo o a estrutura destes testes é análoga.

- Tamanho de Mensagem
- Número de Repetições

Cada conjunto de resultados foi classificado por Tamanho de Mensagens Enviadas como é possível ver no Capítulo 6.

4.4 SendRecv

Como este *benchmarck* foi realizado para valores diferentes de:

- Tamanho de Mensagem
- Número de Repetições
- Número de Processos

Os resultados foram classificados em Tamanho de Mensagens Enviadas e Número de Processos Criados, como é mostrado no Capítulo 7

4.5 AlltoAll

Apesar deste *benchmarck* não estar na mesma classificação do *SendRecv*, pois respectivamente são classificados como Transferência Coletiva e Transferência Paralela [Intel Corporation Document Number: 320714-0022010], a estrutura dos mesmos são iguais.

- Tamanho de Mensagem
- Número de Repetições
- Número de Processos

Da mesma forma como *benchmarck SendRecv*, os resultados foram classificados em Tamanho de Mensagens Enviadas e Número de Processos Criados, como é mostrado no Capítulo 8.

Capítulo 5

Resultados PingPing

Segundo a análise dos testes para este *benchmarck* foi observado que ambas linguagens realizaram todos os testes, porém, a linguagem Java se comportou de uma forma diferente se comparado ao Erlang.

Tabela 5.1: Tabela de Resultados do *Benchamarck PingPing*

Testes	**Médias de Tempo de Execução**	
Bytes x Repetições	**Erl_Exec(sec)**	**Java_Exec(sec)**
5B x 5r	0,080767	0,2157208
5B x 10r	0,1560906	0,3082723
5B x 50r	0,7662869	0,8320841
5B x 100r	1,5286092	1,4214889
5B x 500r	7,61966	12,3238589
5B x 1000r	15,2457484	39,6962131
10B x 5r	0,1509212	0,2287589
10B x 10r	0,2976758	0,3722199
10B x 50r	1,4839679	1,3632561
10B x 100r	2,960617	2,17026
10B x 500r	14,7892995	53,6620604
10B x 1000r	29,5680771	120,4303839
50B x 5r	0,7257385	0,8774377
50B x 10r	1,4475493	1,2260585
50B x 50r	7,2325575	17,4878319
50B x 100r	14,4608183	46,0189199
50B x 500r	72,2860941	353,3512706
50B x 1000r	144,564611	762,1553427
Média	**17,5201177444**	**78,5634132556**

Observou-se características comuns em outros *benchmarcks* pois a linguagem Erlang se mostrou mais estável que a linguagem Java, representados por grandes picos de tempo de execução aleatórios, em todos os casos de testes.

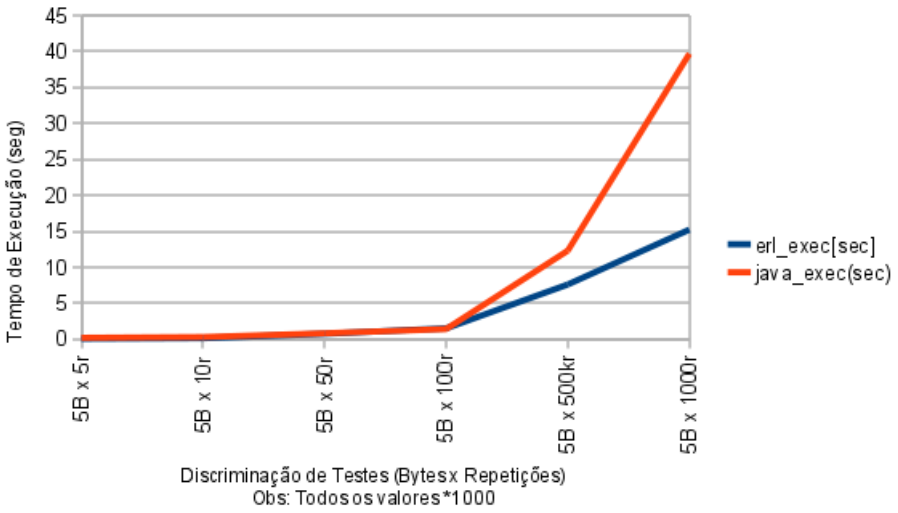

Figura 5.1: *Benchmarck PingPing* para Mensagens de 5 Kbytes

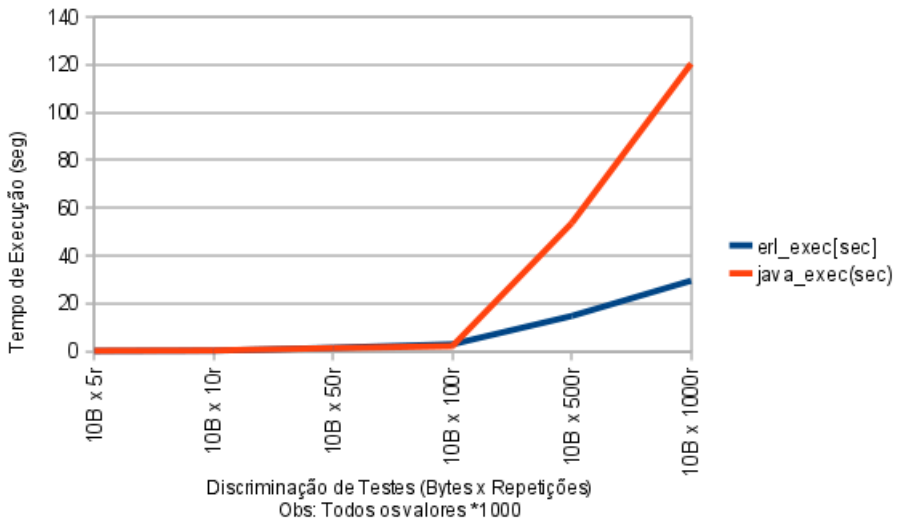

Figura 5.2: *Benchmarck PingPing* para Mensagens de 10 Kbytes

A linguagem Erlang teve um tempo de resposta ao *benchmarck* inferior a linguagem Java em todos os casos de testes conforme mostra a Tabela 5.2.

De forma geral, a linguagem Erlang respondeu aos testes mais rapidamente em 77,77% dos casos de teste, enquanto a linguagem Java respondeu mais rápido em 22,22% dos casos.

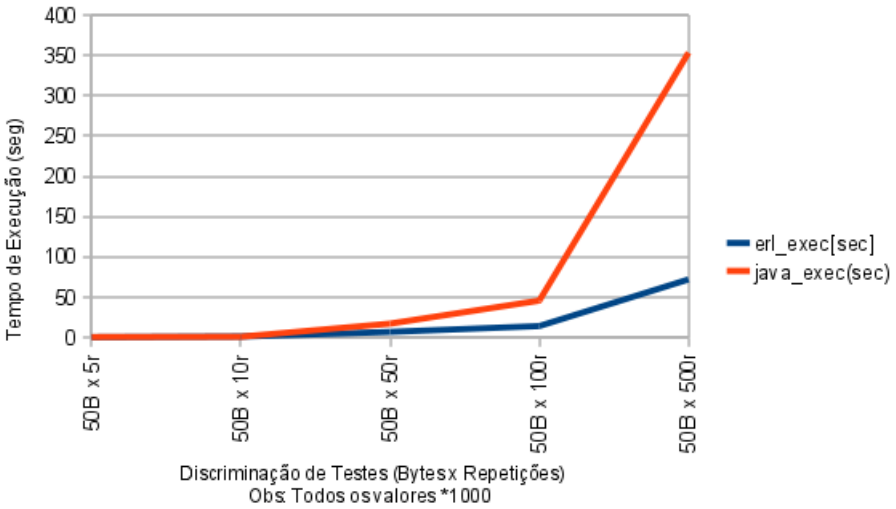

Figura 5.3: *Benchmarck PingPing* para Mensagens de 50 Kbytes

Tabela 5.2: Frequência de Menor Tempo de Resposta

Tamanho da Mensagem	**Erlang**	**Java**
5 Kbytes	83,33%	16,66%
10 Kbytes	66,66%	33,33%
50 Kbytes	83,33%	16,66%

A medida em que o número de repetições aumenta, durante a execução dos testes as duas linguagens variam os tempos de resposta como é possível ver na Tabela 5.1, porém, quando o número de repetições de envio de mensagens se aproximam de 100.000, o tempo de resposta de ambas linguagens cresce, mas em especial, a linguagem Java apresenta um crescimento superior no tempo de resposta quando comparado ao aumento do tempo de resposta a linguagem Erlang, como é possível ser visto na Figura 5.1 e Figura 5.2. No pior caso mapeado pelos testes realizados, envio de mensagem de 50 Kbytes, a linguagem Java começa a apresentar um aumento significativo do tempo de resposta em relação ao Erlang com 50.000 repetições de envio e não mais com 100.000 como dito para os outros tamanhos de mensagens, como é possível ver no Figura 5.3.

Capítulo 6

Resultados PingPong

Para este *benchmarck*, segundo [Intel Corporation Document Number: 320714-0022010], que é classificado como *benchmarck* de transferência simples, assim como o *PingPing*, os resultados foram análogos aos observados no capítulo anterior.

Tabela 6.1: Tabela de Resultados do *Benchamarck PingPong*

Testes	**Médias de Tempo de Execução**	
Bytes x Repetições	**Erl_Exec(sec)**	**Java_Exec(sec)**
5B x 5r	0,0801032	0,3053983
5B x 10r	0,1578888	0,3251125
5B x 50r	0,769948	1,0904576
5B x 100r	1,5312113	1,3157443
5B x 500r	7,6235237	17,3235074
5B x 1000r	15,2257889	38,0917687
10B x 5r	0,152707	0,2736428
10B x 10r	0,2994445	0,4375563
10B x 50r	1,48333513	1,3031145
10B x 100r	2,9587884	2,362707
10B x 500r	14,7799315	36,7402369
10B x 1000r	29,5498582	82,4653145
50B x 5r	0,7245493	0,6309346
50B x 10r	1,4490356	1,5373947
50B x 50r	7,2322102	15,3243676
50B x 100r	14,4517292	40,6628583
50B x 500r	72,2322904	258,7443386
50B x 1000r	144,4708575	519,9133507
Média	**17,509623166**	**56,5878225167**

As características comuns visíveis na Tabela 6.1 entre os *benchmarcks PingPing e Ping-Pong* são a maior estabilidade do Erlang comparado ao Java e o menor tempo de resposta.

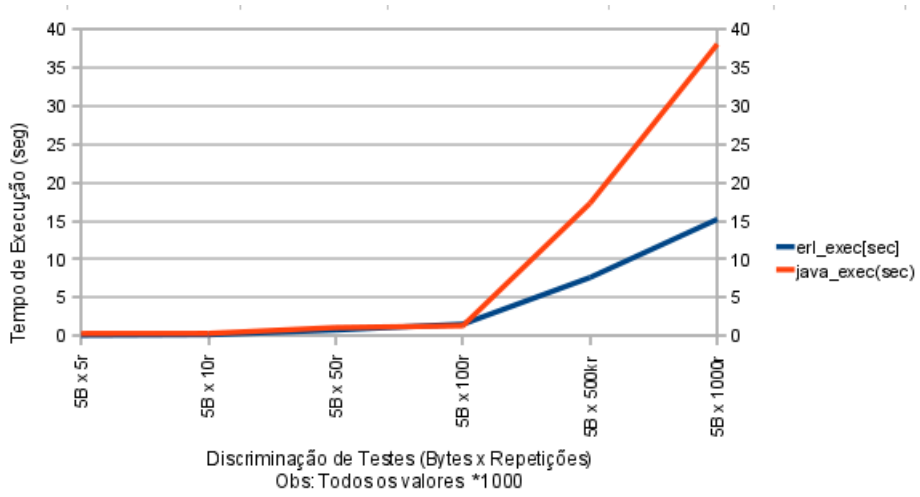

Figura 6.1: *Benchmarck PingPong* para Mensagens de 5 Kbytes

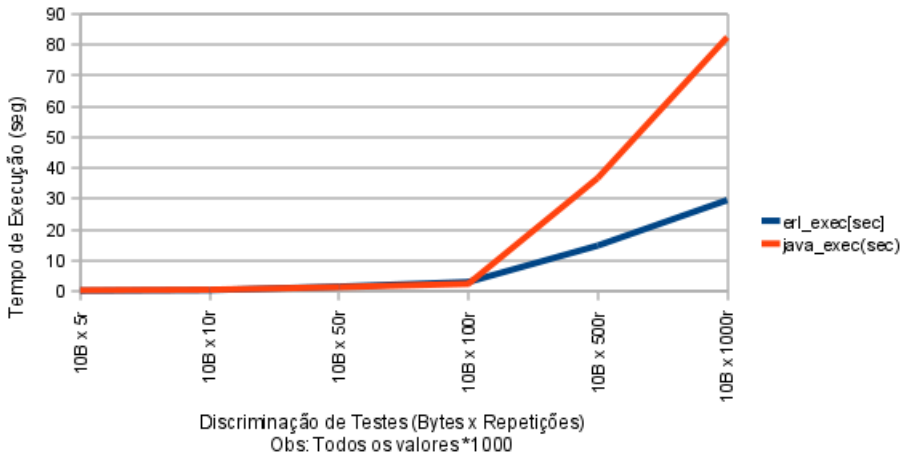

Figura 6.2: *Benchmarck PingPong* para Mensagens de 10 Kbytes

O tempo de resposta do Erlang é inferior ao Java como mostra a Tabela 6.2 e, como no *PingPing*, Erlang se mostra com melhor desempenho em 77,77% dos casos de testes de forma geral.

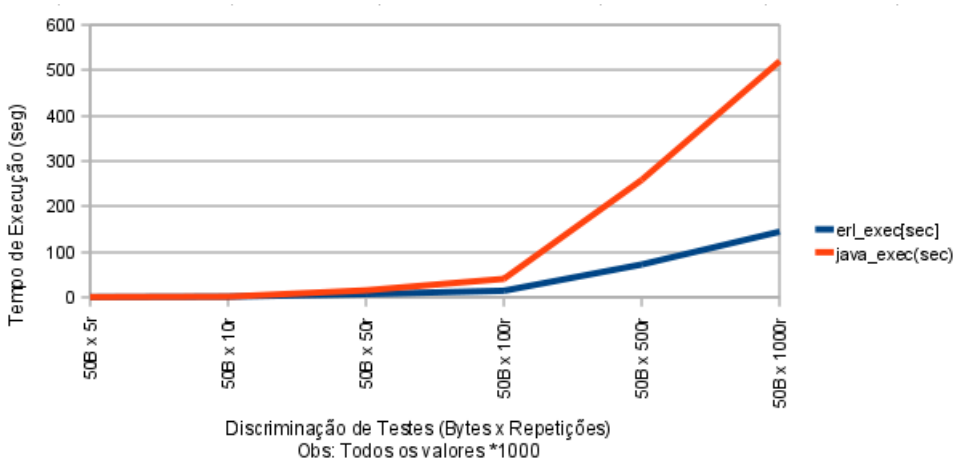

Figura 6.3: *Benchmarck PingPong* para Mensagens de 50 Kbytes

Tabela 6.2: Frequência de Menor Tempo de Resposta

Tamanho da Mensagem	**Erlang**	**Java**
5 Kbytes	83,33%	16,66%
10 Kbytes	66,66%	33,33%
50 Kbytes	83,33%	16,66%

Assim como ocorre no benchmarck PingPing é visível na Figura 6.1 e na Figura 6.2 o aumento do tempo de resposta da linguagem Java com 100.000 repetições, na Figura 6.3 representa-se o aumento significativo do tempo de resposta do Java com mensagens de 50 Kbytes

Capítulo 7

Resultados SendRecv

Este *benchmarck* é definido segundo [Intel Corporation Document Number: 320714-0022010] como um *benchmarck* de transferência paralela, onde vários processos formam uma corrente e cada processo envia uma mensagem para o processo imediatamente à sua direita e recebe uma mensagem do processo imediatamente à sua esquerda. A contagem de ações é feita para a medida que cada processo finaliza sua execução (independentemente). [Intel Corporation Document Number: 320714-0022010]

Neste trabalho foi modificado a contagem do tempo de execução para este *benchmarck*. Ao invés de medir a execução independente de cada processo e contabilizar como uma ação o envio de mensagem de um processo para outro na corrente de processos, mede-se o tempo total usado para enviar uma mensagem de tamanho x *bytes* para todos os processos, e contabiliza-se como término da ação quando a mensagem realiza uma volta completa na corrente de processos.

Para este *benchmarck* os casos de testes foram definidos da seguinte maneira:

- Tamanho de Mensagens – 5 Kbytes, 10 Kbytes, 50 Kbytes
- Número de Processos – 1.000, 10.000, 20.000, 30.000, 50.000, 100.000
- Número de Repetições – 5.000, 10.000, 50.000, 100.000, 500.000, 1.000.000

Neste *benchamrck* foi observado a princípio que ambas as linguagens falharam ao executar todos os casos de testes, entretanto a linguagem Erlang executou um número maior

de testes comparado a linguagem Java.

Tabela 7.1: Tabela de Resultados do *Benchamarck SendRecv*

Testes	**Médias de Tempo de Execução**	
Bytes x Repetições	**Erl_Exec(sec)**	**Java_Exec(sec)**
5B x 5r x 1p	3,1823864	42,4540341
5B x 10r x 1p	6,4215896	84,3864334
5B x 50r x 1p	32,34443075	421,8924501
5B x 100r x 1p	65,0742073	835,6209643
5B x 500r x 1p	325,9612866	4165,6786009
5B x 1000r x 1p	685,7565915	4982,037213
5B x 5r x 10p	56,2396087	
5B x 10r x 10p	112,364835	
5B x 50r x 10p	564,2407549	
5B x 100r x 10p	1119,0465663	
5B x 500r x 10p	5642,9084965	
5B x 1000r x 10p	11331,5365532	
5B x 5r x 20p	112,5124355	
5B x 10r x 20p	223,820632	
5B x 50r x 20p	1123,4387507	
5B x 100r x 20p	2283,40927	
10B x 5r x 1p	3,1824651	42,3381821
10B x 10r x 1p	6,2597909	84,1613471
10B x 50r x 1p	30,5975867	421,3482128
10B x 100r x 1p	61,6072029	835,4450724
10B x 500r x 1p	313,6770049	421,3482128
10B x 1000r x 1p	627,7633861	835,4450724
10B x 5r x 10p	55,8749572	
10B x 10r x 10p	111,26609	
10B x 50r x 10p	112,2963786	
50B x 5r x 1p	3,6600469	47,4757988
50B x 10r x 1p	7,332387	88,3795737
50B x 50r x 1p	36,5025565	421,6092752
50B x 50r x 1p	72,349375	833,438064
50B x 500r x 1p	364,9815657	4144,7872849
50B x 1000r x 1p	732,0653479	1655,4409392
50B x 5r x 10p	60,8595058	
50B x 10r x 10p	121,8079763	
50B x 50r x 10p	608,404974	
50B x 100r x 10p	1216,6501597	
50B x 500r x 10p	6083,876974	
Média	**980,5506988971**	**1131,29370729**

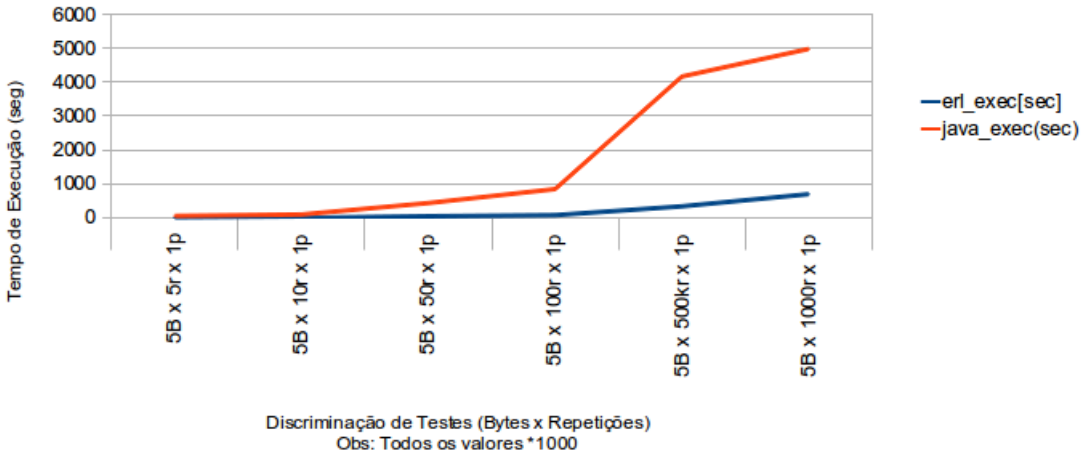

Figura 7.1: *Benchmarck SendRecv* para Mensagens de 5 Kbytes e 1.000 Processos

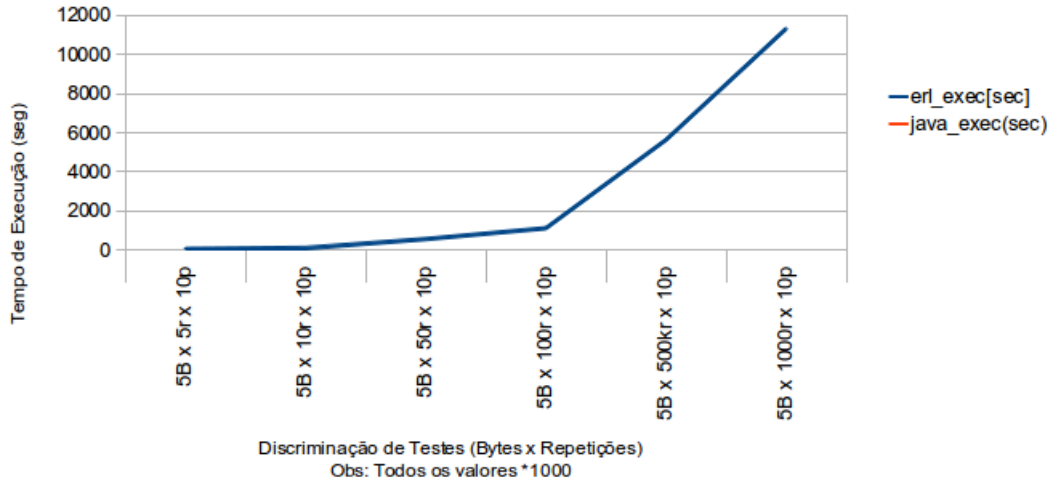

Figura 7.2: *Benchmarck SendRecv* para Mensagens de 5 Kbytes e 10.000 Processos

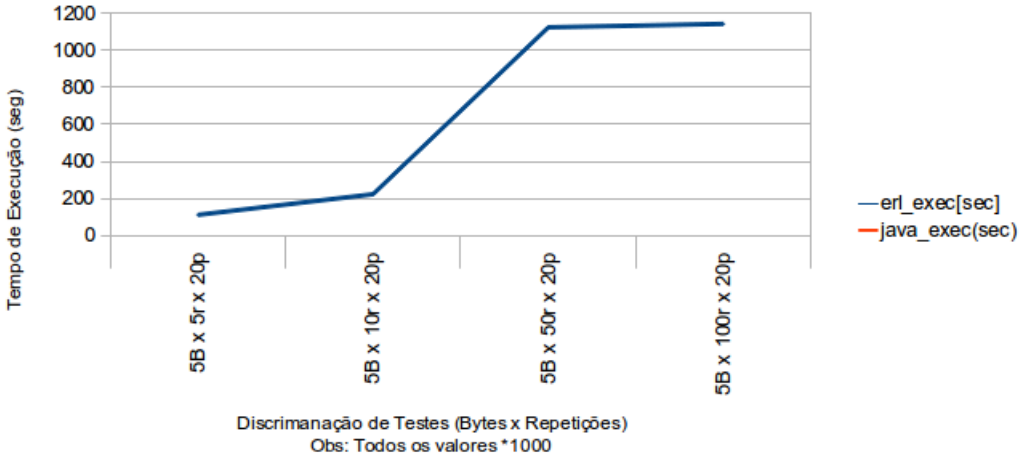

Figura 7.3: *Benchmarck SendRecv* para Mensagens de 5 Kbytes e 20.000 Processos

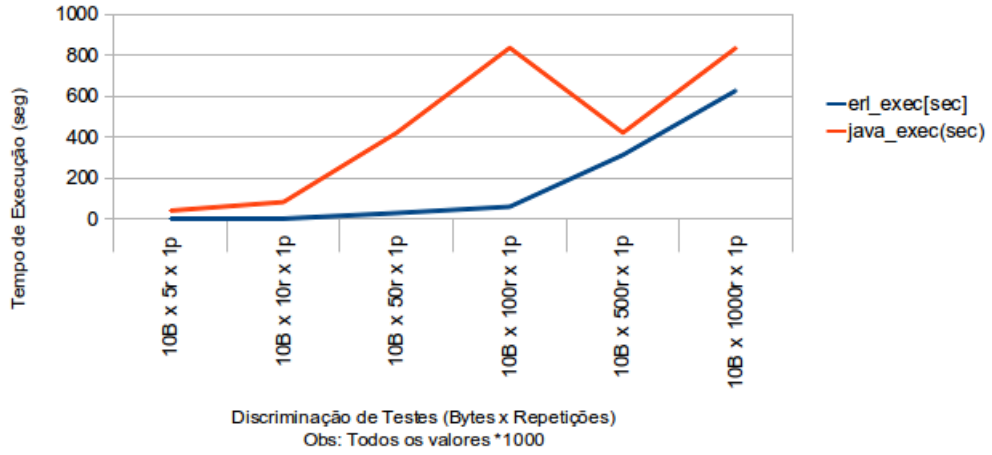

Figura 7.4: *Benchmarck SendRecv* para Mensagens de 10 Kbytes e 1.000 Processos

Nos casos de testes em que as duas linguagens executaram, sendo que Java executou 51,43% dos casos de teste, Erlang apresentou tempo de resposta menor que Java em 100% dos casos, como mostrado na Tabela 7.2. Nos outros 48,57% de casos de teste, o tempo de resposta de Erlang cresce, se comparado com os tempos anteriores do próprio Erlang, mas a linguagem Java não gerou resultados para realizar a comparação, como visto na Tabela 7.1.

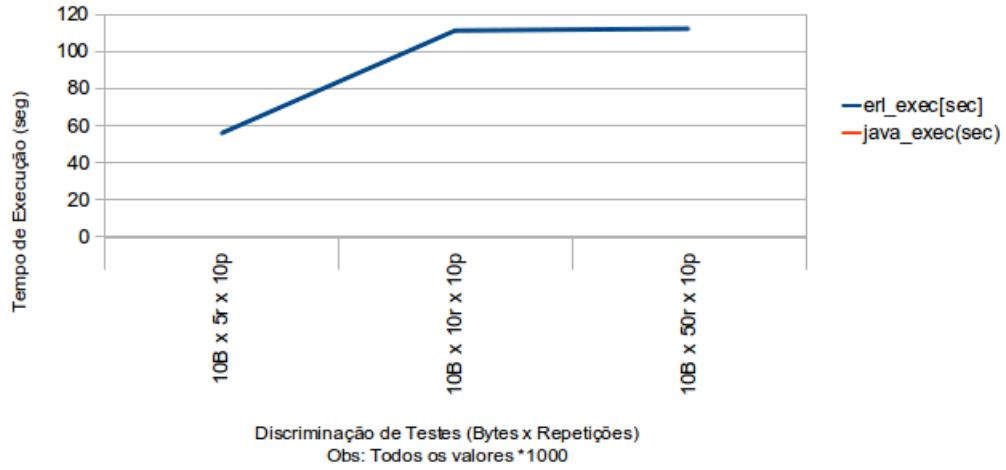

Figura 7.5: *Benchmarck SendRecv* para Mensagens de 10 Kbytes e 10.000 Processos

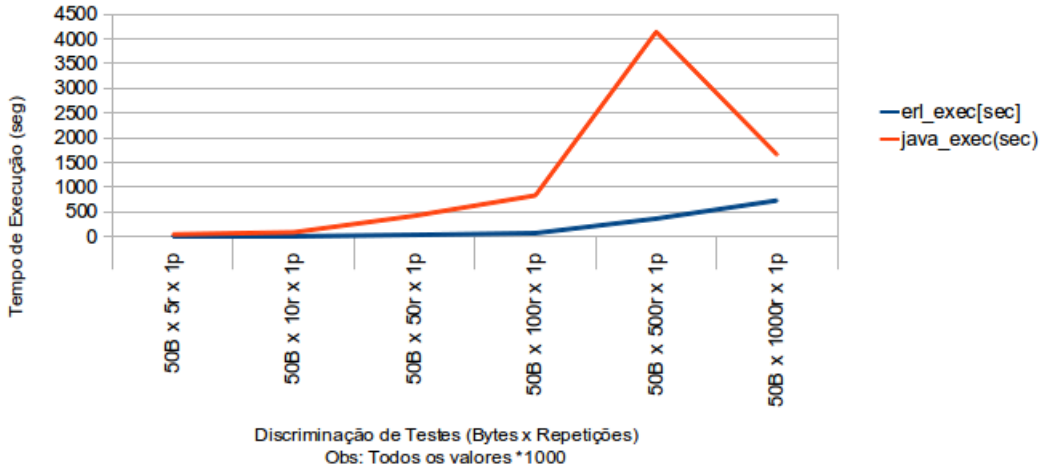

Figura 7.6: *Benchmark SendRecv* para Mensagens de 50 Kbytes e 1.000 Processos

Apesar da linguagem Erlang apresentar um tempo de resposta relativamente grande, comparados aos resultados das duas seções anteriores, continua executando para os parâmetros de Número de Repetições e Número de Processos diferente da linguagem Java que não executou os testes com Número de Processos igual a 10.000 como é possível notar nos Gráficos 7.2, 7.3, 7.5 e 7.7.

De forma similar ao *benchmarck PingPing e PingPong*, a linguagem Java se mostra pouco estável comparado ao Erlang, como é possível ver nos Gráficos 7.1, 7.4 e 7.6.

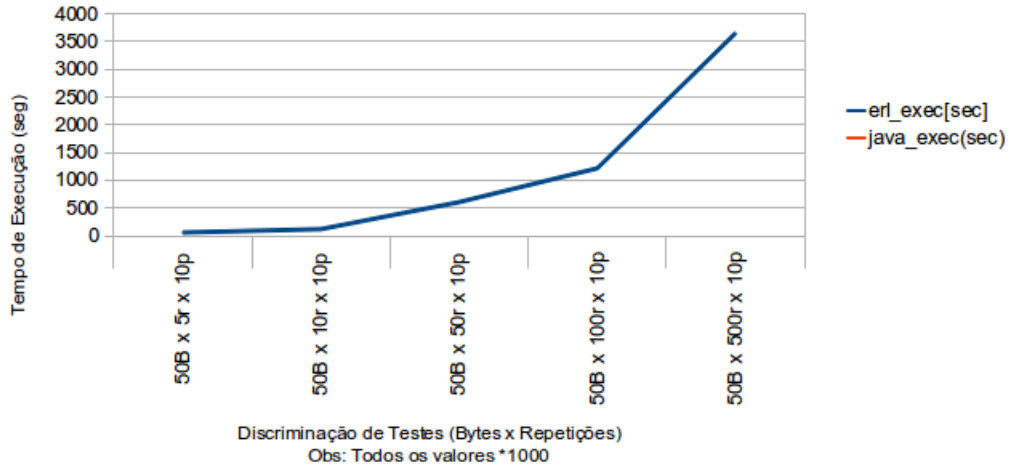

Figura 7.7: *Benchmarck SendRecv* para Mensagens de 50 Kbytes e 10.000 Processos

Tabela 7.2: Frequência de Menor Tempo de Resposta

Tamanho da Mensagem	Número de Processos	Erlang	Java
5 Kbytes	1.000	100%	0%
5 Kbytes	10.000	100%	0%
5 Kbytes	20.000	100%	0%
10 Kbytes	1.000	100%	0%
10 Kbytes	10.000	100%	0%
50 Kbytes	1.000	100%	0%
50 Kbytes	10.000	100%	0%

Capítulo 8

Resultados AlltoAll

Segundo a Documentação do MPI, este teste é classificado como *"coletivo"*, para este tipo de teste é definido que cada *benchmarck* é executado com tamanhos de mensagens variados e a medição de tempo é feita com a média de várias amostras, como é definido na metodologia adotada [Intel Corporation Document Number: 320714-0022010].

É importante ressaltar a diferença nos tempos medidos entre os diferentes testes. Nos testes de transferência simples e paralela todos os processos participam de uma mesma tarefa, que é repetida sucessivamente por um número determinado de vezes, então mede-se o tempo total. [Intel Corporation Document Number: 320714-0022010] Mesmo no caso do anel, onde a mesma mensagem é repassada por todos os processos em cada volta, cada participante é ativado um por vez e a tarefa só é concluía após uma volta completa, como explicado na seção anterior.

Especificamente no *benchmarck AlltoAll*, de transferência coletiva, cada participante é ativado simultaneamente com os outros e conclui sua tarefa de forma independente, o que se deseja aferir com este teste é como o tempo de resposta varia para cada processo quando se aumenta a quantidade de participantes ativos simultâneos. Sendo assim, cada processo fornece uma amostra diferente e é calculado a média de todas as amostras, bem como o tempo máximo e mínimo.

Para este *benchmarck* os casos de testes foram definidos da seguinte maneira:

- Tamanho de Mensagens – 5 Kbytes, 10 Kbytes, 50 Kbytes

- Número de Processos – 1.000, 2.000, 3.000, 4.000, 5.000, 10.000
- Número de Repetições – 5.000, 10.000, 50.000, 100.000, 500.000, 1.000.000

Tabela 8.1: Tabela de Resultados do *Benchamarck AlltoAll*

Testes	**Médias de Tempo de Execução**	
Bytes x Repetições	**Erl_Media(sec)**	**Java_Media(sec)**
5B x 5r x 1p	6228,987337	2318,852663
5B x 10r x 1p	12667,0834	4653,658752
5B x 50r x 1p	63197,73975	23358,17155
5B x 100r x 1p	121880,2713	47122,76883
5B x 5r x 2p	34689,4557312	7529,3951096
5B x 10r x 2p	76250,4876035	15012,094483875
10B x 5r x 1p	6588,894787	2304,9061162222
10B x 10r x 1p	12960,82521	4656,1376023333
10B x 50r x 1p	62786,78425	23381,1006506667
10B x 100r x 1p	120444,6996	
50B x 5r x 1p	6317,5517821	
50B x 10r x 1p	112706,54821	
50B x 50r x 1p	63475,79479	
Média	**46169,17526732**	**14481,9119514849**

De maneira análoga ao *SendRecv*, as duas linguagens não executaram todos os casos de testes pré-definidos, entretanto, Erlang executou mais parâmetros dos casos de testes se comparado a linguagem Java.

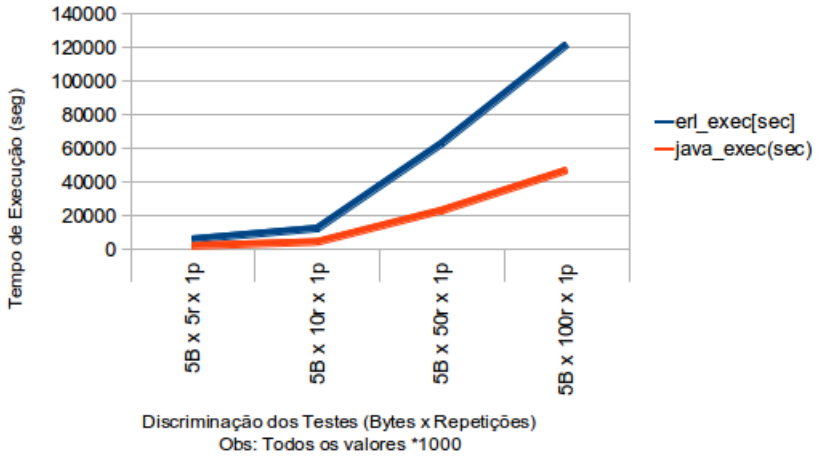

Figura 8.1: *Benchmarck AlltoAll* para Mensagens de 5 Kbytes e 1.000 Processos

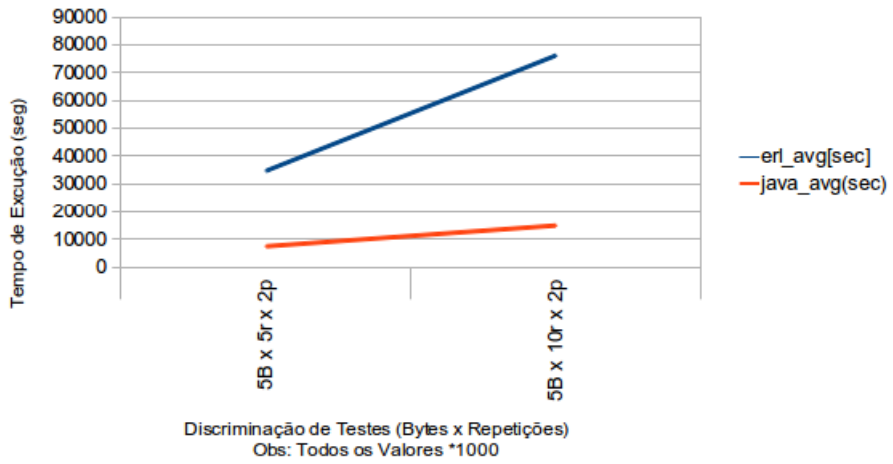

Figura 8.2: *Benchmarck AlltoAll* para Mensagens de 5 Kbytes e 2.000 Processos

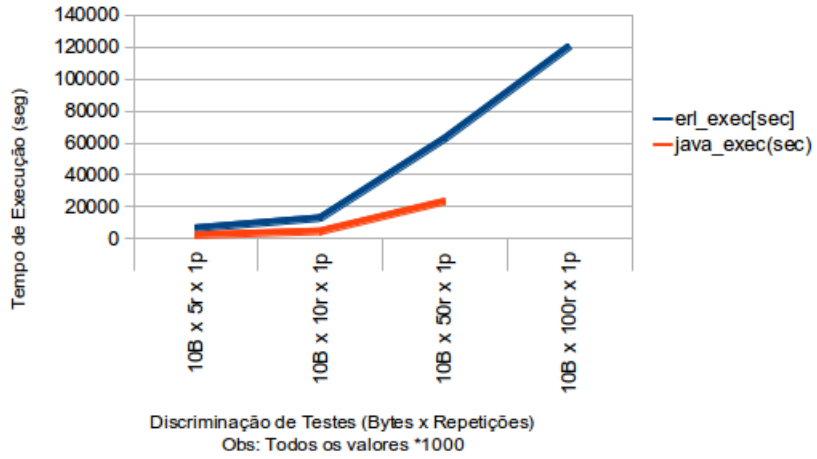

Figura 8.3: *Benchmarck AlltoAll* para Mensagens de 10 Kbytes e 1.000 Processos

A linguagem Java executou 69,23% dos casos de testes, dentre estes casos, Java apresentou tempo de resposta menor que Erlang em 100% dos testes como mostra a Tabela 8.2, apesar de que nos outros 30,76% dos casos de testes apenas Erlang gerou resultados.

A linguagem Java foi mais eficiente para este *benchmarck* uma vez que os tempos de resposta obtidos foram mais baixos que os de Erlang, como representado nas Figuras 8.1, 8.2 e 8.3. Nas Figuras 8.3 e 8.4 é possível ver os testes em que Java não foi capaz de executar, entretanto Erlang os executou, mesmo com tempos altos de resposta.

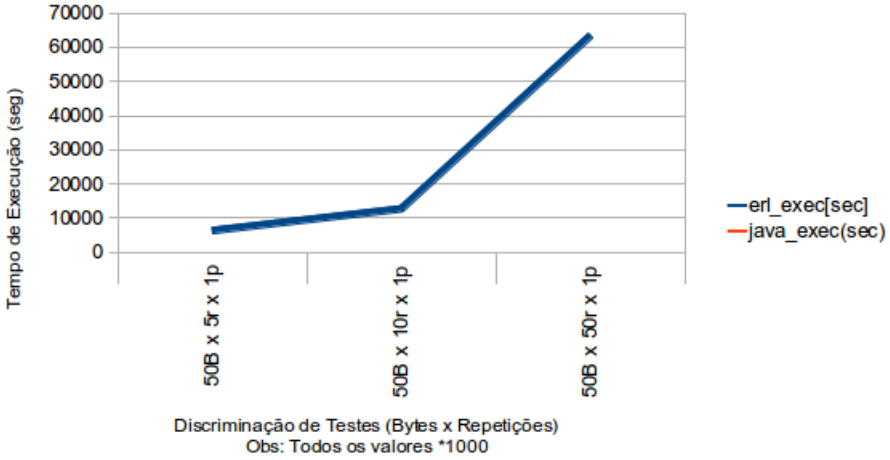

Figura 8.4: *Benchmarck AlltoAll* para Mensagens de 50 Kbytes e 1.000 Processos

Tabela 8.2: Frequência de Menor Tempo de Resposta

Tamanho da Mensagem	Número de Processos	Erlang	Java
5 Kbytes	1.000	0%	100%
5 Kbytes	2.000	0%	100%
10 Kbytes	1.000	0%	100%
50 Kbytes	1.000	0%	100%

Capítulo 9

Conclusão

De acordo com os resultados obtidos durante a execução deste trabalho conforme demonstrados nas Seções 5, 6, 7 e 8 pode-se notar que as duas linguagens sumbetidas aos testes, *Java e Erlang*, realmente suportam a programação concorrente de modo a conseguir executar os *benchmarks* usados do IMB [Intel Corporation Document Number: 320714-0022010]. Entretanto, cada linguagem apresentou um comportamento distinto entre si.

Ambas as linguagens conseguiram executar todos os casos de teste dos *benchmarks Ping-Ping e PingPong*, que segundo a [Intel Corporation Document Number: 320714-0022010] são de transferência simples, porém a linguagem Erlang apresentou um desempenho superior comparado a linguagem Java, respondendo em menor tempo de forma geral em 77,77% dos casos de teste tanto do *benchmarck PingPing* quanto do *benchmarck PingPong*.

No caso específico do *benchmarck SendRecv*, Java e Erlang não executaram todos os casos de testes pré definidos, porém, Java se mostrou ineficiente quando não conseguiu executar os mesmos parâmetros de teste que Erlang executou e nos 51,43% dos casos de testes que foi capaz de executar, em 100% dos mesmos Java apresentou um tempo de resposta superior aos tempos obtidos com a linguagem Erlang.

No *benchmarck AlltoAll* nota-se um comportamento semelhante ao do *SendRecv* de ambas as linguagens, pois não conseguem executar todos os casos de testes, e Erlang executa mais casos de teste quando comparado ao Java, que executou 69,23% dos testes que a linguagem Erlang executou. Para este *benchmark* a linguagem Java mostrou-se superior a linguagem Erlang no seguinte aspecto, em 100% dos casos de testes que Java

executou apresentou tempo de resposta menor que Erlang.

De forma geral, Erlang além de apresentar menor tempo de resposta em alguns dos testes, como mencionado anteriormente, também foi o mais estável, mesmo apresentando elevados tempos de resposta.

Com isso, pode-se notar que a linguagem Erlang mostrou ser mais eficiente em 75% dos *benchmarcks*, em utilizações de transferências simples e paralela de mensagens com tamanhos variados e repetições sucessivas comparado ao Java e a linguagem Java apresentou-se ser eficiente em aplicações de *broadcast*, tratando melhor a recepção de múltiplas mensagens e envio de múltiplas mensagens "simultaneamente".

9.1 Trabalhos Futuros

Como continuação de pesquisa baseado no desenvolvimento deste trabalho é possível prosseguir com a comparação do desempenho entre linguagens diferentes linguagens concorrentes, como:

- Ruby
- Python
- Scala

Referências Bibliográficas

[Armstrong et al.1996] Armstrong, J., Virding, R., Wikström, C., and Williams, M. (1996). *Concurrent Programming in ERLANG*. Prentice Hall, second edition. ISBN: 0-13-508301-X.

[Brown2011] Brown, M. (2011). Introduction to Programming in Erlang. http://www.ibm.com/developerworks/opensource/library/os-erlang1/index.html.

[Intel Corporation Document Number: 320714-0022010] Intel Corporation Document Number: 320714-002 (2010). Intel® MPI Benchmarks. User Guide and Methodology Description. http://www.intel.com.

[Jenkov] Jenkov, J. Java Concurrency. Acessado em Novembro de 2011, disponívem em: http://tutorials.jenkov.com/java-concurrency/deadlock.html.

[Lea1999] Lea, D. (1999). *Concurrent Programming in Java^{TM}: Design Principles and Patterns*. Addison Wesley Longman, 2nd edition. ISBN: 0-201-31009-0.

[Liria Matsumoto Sato et al.1996] Liria Matsumoto Sato, Edson Toshimi Midorikawa, and Hermes Senger (1996). Introdução a Programação Paralela e Distribuída.

[MPI2009a] MPI, F. (2009a). Mpi - Documents. http://www.mpi-forum.org/docs/docs.html.

[MPI2009b] MPI, F. (2009b). Mpi 3.0 standardization effort. http://meetings.mpi-forum.org/MPI_3.0_main_page.php.

[Oracle Corporation] Oracle Corporation. The Java^{TM} – Concurrency.

[Patterson2008] Patterson, D. (2008). The Parallel Revolution Has Started Are You Part of the Solution or Part of the Problem? Disponível em: http://www.youtube.com/watch?v=A2H_SrpAPZU.

[Py2009] Py, M. X. (2009). Máquina de Turing Paralela com Memória Compartilhada. Disponível em: http://www.inf.ufrgs.br/gppd/disc/cmp134/trabs/T1/001/mtp/concorrencia.

[Rauber and Runger2010] Rauber, T. and Runger, G. (2010). *Parallel Programming for Multicore and Cluster Systems*. Springer Heidelberg Dordrecht London New York, 2nd edition.

[RoseIndia2008] RoseIndia (2008). What is the use of java? http://www.roseindia.net/java/use-java/uses-of-java.html.

[Tanembaum2007] Tanembaum, A. S. (2007). *Sistemas Operacionais Modernos*. Prentice Hall do Brasil, Rio de Janeiro.

[Vogel2011] Vogel, L. (2011). Java Concurrency/Multithreading. http://www.vogella.de/articles/JavaConcurrency/article.htmlconcurrency.

Apêndice 1 – Codigos *Benchmarck*

PingPing

```
1  package pingping;
2
3  import persistencia.Salvar;
4
5  public class PingPingPrincipal {
6
7          public static void main(String[] args) {
8
9                  // arg[0] -> tam bytes!
10                 // arg[1] -> quantidade de mensagens!
11
12                 int tamMsg = Integer.parseInt(args[0]);
13                 int qtdMsg = Integer.parseInt(args[1]);
14
15                 //String localSaida = "/home/pesquisador/temp/out_java_pingping.txt";
16                 String localSaida = Salvar.OUT_PATH + "pingping.txt";
17
18                 PingPing p = new PingPing(tamMsg, qtdMsg, localSaida);
19                 p.start();
20         }
21
22 }
```

Código 9.1.1: Código Java da classe *PingPingPrincipal*

```
1  package pingping;
2  import persistencia.Salvar;
3
4  public class PingPing extends Thread{
5          private final int QTD_PROC_TOTAL = 2;
6          // a medida que os processos finalizarem qtdTerminados é incrementados
7          private int qtdProcTerminados = 0, tamDados, qtdRept;
8          private long timeSpawn, timeExec, timeStart, timeEnd;
9
10         private String outLocation;
11
12         public PingPing(int tamDados, int qtdMsg, String outLocation) {
13                 this.tamDados = tamDados;
14                 this.qtdRept = qtdMsg;
15                 this.outLocation = outLocation;}
16
17         public void run() {
18                 byte[] dado = generateData(tamDados);
19
20                 // cria os processos
21                 timeStart = timeMicroSeg();
22                 ProcPing p1 = new ProcPing("1", dado, this, qtdRept);
23                 ProcPing p2 = new ProcPing("2", dado, this, qtdRept);
24                 timeEnd = timeMicroSeg();
25
26                 // armazena tempo de criação
27                 timeSpawn = timeEnd - timeStart;
28
29                 // inicia execução dos processos
30                 timeStart = timeMicroSeg();
31                 p1.setPeer(p2); // seta o par
32                 p1.start();
33                 p2.setPeer(p1); // seta o par
34                 p2.start();
35
36                 dormirAteTerminar();
37
38                 // após todos terminarem, executa a prox linha
39                 timeEnd = timeMicroSeg(); // captura tempo atual
40
41                 //FINALIZA TESTE
42                 timeExec = timeEnd - timeStart; // armazena tempo de execução
43
44                 // escreve a saída
45                 Salvar.writeResultPeer(outLocation, tamDados, qtdRept, timeExec, timeSpawn);}
46
47         private synchronized void dormirAteTerminar() {
48                 while(qtdProcTerminados!= QTD_PROC_TOTAL){
49                         try {
50                                 wait();
51                         } catch (InterruptedException e) {
52                                 e.printStackTrace();
53         }}}
54
55         public synchronized void acordar() {
56                 qtdProcTerminados++;
57                 notifyAll();}
58
59         private byte[] generateData(int tamDados) {
60                 byte[] dado = new byte[tamDados];
61                 return dado;}
62
63         private long timeMicroSeg() {
64                 return System.nanoTime()/1000;}
65 }
```

Código 9.1.2: Código Java da classe *PingPing*

```
1  package pingping;
2
3  public class ProcPing extends Thread {
4          // public String mailbox = new String();
5          public byte[] mailbox;
6          private ProcPing peer;
7          private PingPing parent;
8          private int qtdMsg;
9          private byte[] dado;
10
11         public ProcPing(String name, byte[] dado, PingPing parent, int qtdMsg) {
12                 this.setName(name);
13                 this.dado = dado;
14                 this.parent = parent;
15                 this.qtdMsg = qtdMsg;
16         }
17
18         public void setPeer(ProcPing peer) {
19                 this.peer = peer;
20         }
21
22         private synchronized void send(byte[] msg) {
23                         peer.mailbox = msg.clone();
24         }
25
26         private void recv() {
27                 // verifica mailbox até ter mensagem
28                 while (true) {
29                         synchronized (this) {
30                                 if (mailbox != null) {
31                                         mailbox = null;
32                                         break;
33                                 }
34                         }
35                 }
36         }
37
38         public void run() {
39                 for (int i = 1; i <= qtdMsg; i++) {
40                         send(dado);
41
42                         if(i!=qtdMsg){
43                                 recv();
44                         }
45                 }parent.acordar();}
46 }
```

Código 9.1.3: Código Java da classe *ProcPing*

```
1  -module(pingping).
2  -export([run/2]).
3
4  -include("conf.hrl").
5  -import(persist, [write_result/3]).
6  -import(medicoes, [generate_data/1, time_microseg/0]).
7
8  run(DataSize, R) ->
9      Data = generate_data(DataSize),
10
11     Self = self(),
12
13     SpawnStart = time_microseg(),
14     P1 = spawn(fun() -> pingping(Data, Self, R) end),
15     P2 = spawn(fun() -> pingping(Data, Self, R) end),
16     SpawnEnd = time_microseg(),
17
18     SpawnTime = SpawnEnd - SpawnStart,
19
20     TimeStart = time_microseg(),
21     P1 ! {init, self(), P2},
22     P2 ! {init, self(), P1},
23     finalize(P1),
24     finalize(P2),
25     TimeEnd = time_microseg(),
26
27     TotalTime = TimeEnd - TimeStart,
28
29     write_result(peer, ?OUT_PATH, [Data, R, TotalTime, SpawnTime]).
30
31 pingping(_,Parent, 0) ->
32     Parent ! {finish, self()};
33
34 pingping(Data, Parent, R) ->
35     receive
36         {init, Parent, Peer} ->
37             Peer ! {self(), Data},
38             pingping(Data, Parent, R-1);
39
40         {Peer, Data} ->
41             Peer ! {self(), Data},
42             pingping(Data, Parent, R-1)
43     end.
44
45 finalize(P1) ->
46     receive
47         {finish, P1} ->
48             ok
49     end.
```

Código 9.1.4: Código Erlang do módulo *PingPing*

Apêndice 2 – Codigos *Benchmarck PingPong*

```
1  package pingpong;
2
3  import persistencia.Salvar;
4
5  public class PingPongPrincipal {
6
7          public static void main(String[] args) {
8                  int tamMsg = Integer.parseInt(args[0]);
9                  int qtdMsg = Integer.parseInt(args[1]);
10
11                 //String localSaida = "/home/pesquisador/temp/out_java_pingpong.txt";
12                 String localSaida = Salvar.OUT_PATH + "pingpong.txt";
13
14                 PingPong p = new PingPong(tamMsg, qtdMsg, localSaida);
15                 p.start();
16         }
17
18 }
```

Código 9.1.5: Código Java da classe *PingPongPrincipal*

```
1  package pingpong;
2  import persistencia.Salvar;
3
4  public class PingPong extends Thread{
5
6          private boolean espera = true;
7          private long timeSpawn,timeExec, timeStart, timeEnd;
8          private int tamDados, qtdRept;
9          private String outLocation;
10
11         public PingPong(int tamDados, int qtdMsg, String outLocation) {
12                 this.tamDados = tamDados;
13                 this.qtdRept = qtdMsg;
14                 this.outLocation = outLocation;}
15
16         public void run() {
17                 byte[] dado = generateData(tamDados);
18
19                 // cria os processos
20                 timeStart = timeMicroSeg();
21                 ProcPing ping = new ProcPing("1", dado, this, qtdRept);
22                 ProcPong pong = new ProcPong("2", dado, qtdRept);
23                 timeEnd = timeMicroSeg();
24
25                 // armazena tempo de criação
26                 timeSpawn = timeEnd - timeStart;
27
28                 // inicia execução dos processos
29                 timeStart = timeMicroSeg();
30
31                 ping.setPeer(pong); // seta o par
32                 ping.start();
33                 pong.setPeer(ping); // seta o par
34                 pong.start();
35
36                 dormirAteTerminar();
37
38                 // após todos terminarem, executa a prox linha
39                 timeEnd = timeMicroSeg(); // captura tempo atual
40
41                 //FINALIZA TESTE
42                 timeExec = timeEnd - timeStart; // armazena tempo de execução
43
44                 // escreve a saída
45                 Salvar.writeResultPeer(outLocation, tamDados, qtdRept, timeExec, timeSpawn);}
46
47         private synchronized void dormirAteTerminar() {
48                 while(espera){
49                         try {
50                                 wait();
51                         } catch (InterruptedException e) {
52                                 e.printStackTrace();
53         }}}
54
55         public synchronized void acordar() {
56                 espera = false;
57                 notifyAll();}
58
59         private byte[] generateData(int tamDados) {
60                 byte[] dado = new byte[tamDados];
61                 return dado;}
62
63         private long timeMicroSeg() {
64                 return System.nanoTime()/1000;}
65 }
```

Código 9.1.6: Código Java da classe *PingPongPrincipal*

```
1  package pingpong;
2
3  public class ProcPing extends Thread {
4          public boolean espera = false;
5          public byte[] mailbox;
6          private ProcPong peer;
7          private PingPong parent;
8          private int qtdMsg;
9          private byte[] dado;
10
11         public ProcPing(String name, byte[] dado, PingPong parent, int qtdMsg) {
12                 this.setName(name);
13                 this.dado = dado;
14                 this.parent = parent;
15                 this.qtdMsg = qtdMsg;
16         }
17
18         public void setPeer(ProcPong peer) {
19                 this.peer = peer;
20         }
21
22         private synchronized void send(byte[] msg) {
23                 peer.mailbox = msg.clone();
24         }
25
26         private void recv() {
27                 // verifica mailbox até ter mensagem
28                 while (true) {
29                         synchronized (this) {
30                                 if (mailbox != null) {
31                                         mailbox = null;
32                                         break;
33                                 }
34                         }
35                 }
36         }
37
38         public void run() {
39                 for (int i = 1; i <= qtdMsg; i++) {
40
41                         send(dado);
42
43                         recv();
44                 }
45                 parent.acordar();
46         }
47 }
```

Código 9.1.7: Código Java da classe *ProcPing*

```
1  package pingpong;
2
3  public class ProcPong extends Thread {
4          // public String mailbox = new String();
5          public boolean espera = false;
6          public byte[] mailbox;
7          private ProcPing peer;
8          private int qtdMsg;
9          private byte[] dado;
10
11         public ProcPong(String name, byte[] dado, int qtdMsg) {
12                 this.setName(name);
13                 this.dado = dado;
14                 this.qtdMsg = qtdMsg;
15         }
16
17         public void setPeer(ProcPing peer) {
18                 this.peer = peer;
19         }
20
21         private synchronized void send(byte[] msg) {
22                 peer.mailbox = msg.clone();
23         }
24
25         private void recv() {
26                 // verifica mailbox até ter mensagem
27                 while (true) {
28                         synchronized (this) {
29                                 if (mailbox != null) {
30                                         mailbox = null;
31                                         break;
32                                 }
33                         }
34                 }
35         }
36
37         public void run() {
38                 for (int i = 1; i <= qtdMsg; i++) {
39                         recv();
40                         send(dado);
41                 }
42         }
43 }
```

Código 9.1.8: Código Java da classe *ProcPong*

```
1  -module(pingpong).
2  -export([run/2]).
3
4  -include("conf.hrl").
5  -import(persist, [write_result/3]).
6  -import(medicoes, [generate_data/1, time_microseg/0]).
7
8  run(DataSize, R) ->
9      Data = generate_data(DataSize),
10
11     Self = self(),
12
13     SpawnStart = time_microseg(),
14     P1 = spawn(fun() -> pingping(Data, Self, R) end),
15     P2 = spawn(fun() -> pingping(Data, Self, R) end),
16     SpawnEnd = time_microseg(),
17
18     SpawnTime = SpawnEnd - SpawnStart,
19
20     ExecStart = time_microseg(),
21     P1 ! {init, self(), P2},
22     P2 ! {init, self(), P1},
23     finalize(P1),
24     finalize(P2),
25     ExecEnd = time_microseg(),
26
27     TotalTime = ExecEnd - ExecStart,
28
29     write_result(peer, ?OUT_PATH, [Data, R, TotalTime, SpawnTime]).
30
31 pingping(_,Parent, 0) ->
32     Parent ! {finish, self()};
33
34 pingping(Data, Parent, R) ->
35     receive
36         {init, Parent, Peer} ->
37             Peer ! {self(), Data},
38             pingping(Data, Parent, R-1);
39
40         {Peer, Data} ->
41             Peer ! {self(), Data},
42             pingping(Data, Parent, R-1)
43     end.
44
45 finalize(P1) ->
46     receive
47         {finish, P1} ->
48             ok
49     end.
```

Código 9.1.9: Código Erlang do módulo *PingPong*

Apêndice 3 – Codigos *Benchmarck SendRecv*

```
1  package sendrecv;
2
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  import persistencia.Salvar;
7
8
9  public class SendRecvPrincipal {
10         public static void main(String[] args) {
11                 int tamMsg = Integer.parseInt(args[0]);
12                 int num_proc = Integer.parseInt(args[1]);
13                 int num_rep = Integer.parseInt(args[2]);
14
15                 ExecutorService executor = Executors.newFixedThreadPool(num_proc);
16
17                 String localSaida = Salvar.OUT_PATH + "sendrecv.txt";
18
19                 SendRecv ring = new SendRecv (localSaida, num_proc, num_proc, num_rep,
20                 tamMsg, executor);
21
22                 executor.execute(ring);
23         }
24 }
```

Código 9.1.10: Código Java da classe *SendRecvPrincipal*

```
1  package sendrecv;
2  import java.util.concurrent.ExecutorService;
3  import persistencia.Salvar;
4
5  public class SendRecv extends Node{
6          private final String OUT_LOCATION;
7          private final int NUM_PROC,NUM_REP,TAM_MSG;
8          private volatile byte[] msg;
9          private ExecutorService executor;
10
11 public SendRecv(String outLocation,int nos,int threads,int num_rep,int tamMsg,
12 ExecutorService executor){
13                 super(null);
14                 OUT_LOCATION = outLocation;
15                 NUM_PROC = nos;
16                 NUM_REP = num_rep;
17                 TAM_MSG = tamMsg;
18                 this.executor = executor;}
19
20         public long getTimeMicro() {
21                 return System.nanoTime() / 1000;}
22
23         public byte[] generateData(int tamDados) {
24                 byte[] dado = new byte[tamDados];
25                 return dado;}
26
27         public void spawnAndConnectNodes() {
28                 int n = NUM_PROC;
29                 Node[] nodes = new Node[n];
30                 nodes[0] = this; //o próprio "ring" é o primeiro
31                 nodes[n-1] = new Node(nodes[0]);
32                 executor.execute(nodes[n-1]);
33
34         //são criados n menos um processos, pois o proprio ring é o primeiro!
35         //conexao feita de forma inversa para evitar ter que percorrer o vetor duas vezes
36                         nodes[i] = new Node(nodes[i+1]);
37                         executor.execute(nodes[i]);}
38         //conecta o segundo (que foi gerado por último) com primeiro
39                 nodes[0].connect(nodes[1]);}
40
41         private void senderNodeMode(){
42                 try {
43                         for (int i = 1; i <= NUM_REP; i++) {
44                                 this.getNextNode().send(msg);
45                                 @SuppressWarnings("unused")
46                                 byte[] receivedMsg = recv();}
47                 } catch (InterruptedException e) {
48                         System.out.println("Thread interrompida!!!");}}
49
50         public void run(){
51                 long timeStart, timeEnd, timeSpawn, timeExec;
52
53                 msg = generateData(TAM_MSG);
54                 timeStart = getTimeMicro();
55                 spawnAndConnectNodes();
56                 timeEnd = getTimeMicro();
57                 timeSpawn = timeEnd - timeStart;
58
59         // Inicia-se o teste!
60                 timeStart = getTimeMicro();
61                 senderNodeMode();
62                 timeEnd = getTimeMicro();
63                 timeExec = timeEnd - timeStart;
64                 Salvar.writeResultMulti(OUT_LOCATION, TAM_MSG, NUM_PROC, NUM_REP, timeExec,
65                 timeSpawn);
66
67         // 0 significa sair com status normal, 1 seria com erro
68                 System.exit(0);
69 }}
```

Código 9.1.11: Código Java da classe *SendRecv*

```
1  package sendrecv;
2
3  import java.util.concurrent.BlockingQueue;
4  import java.util.concurrent.LinkedBlockingQueue;
5
6  public class Node implements Runnable {
7
8          private Node nextNode;
9
10         private BlockingQueue<byte[]> mailbox = new LinkedBlockingQueue<byte[]>();
11
12
13         public Node(Node nextNode) {
14                 this.nextNode = nextNode;
15         }
16
17         public Node getNextNode(){
18                 return this.nextNode;
19         }
20
21         public void connect(Node node) {
22                 this.nextNode = node;
23         }
24
25
26         public void send(byte[] m) {
27                 mailbox.add(m);
28
29         }
30
31         public byte[] recv() throws InterruptedException {
32                 byte[] msg = mailbox.take();
33                 return msg;
34         }
35
36         public void run() {
37                 try {
38                         while (true) {
39
40                                 byte[] msg = recv();
41
42                                 nextNode.send(msg);
43                         }
44                 } catch (InterruptedException e) {
45
46                         System.out.println("Thread Interrompida!");
47                 }
48         }
49 }
```

Código 9.1.12: Código Java da classe *Node*

```
1  -module(sendrecv).
2  -export([run/3, ring_node/1]).
3
4  -include("conf.hrl").
5  -import(persist, [write_result/3]).
6  -import(medicoes, [generate_data/1, time_microseg/0]).
7
8  run(DataSize, Rep, QtdProcs) ->
9      Data = generate_data(DataSize),
10
11     SpawnStart = time_microseg(),
12     Second = create_procs(QtdProcs),
13     SpawnEnd = time_microseg(),
14
15     SpawnTime = SpawnEnd - SpawnStart,
16
17     ExecStart = time_microseg(),
18     sender_ring_node(Data, Rep, Second),
19     ExecEnd = time_microseg(),
20
21     TotalTime = ExecEnd - ExecStart,
22
23     write_result(sendrecv, ?OUT_PATH, [Data, Rep, QtdProcs, TotalTime, SpawnTime]),
24     erlang:halt().
25
26 create_procs(QtdProcs) ->
27     lists:foldl(
28       fun(_Id, RightPeer) -> spawn(sendrecv, ring_node, [RightPeer]) end,
29       self(),
30       lists:seq(QtdProcs, 2, -1)
31      ).
32
33 sender_ring_node(_,0,_) -> ok;
34 sender_ring_node(Data, Rep, Second) ->
35     Second ! Data,
36     receive
37         Data ->
38             sender_ring_node(Data, Rep-1, Second)
39     end.
40
41 ring_node(RightPeer) ->
42     receive
43         Data ->
44             RightPeer ! Data,
45             ring_node(RightPeer)
46     end.
```

Código 9.1.13: Código Erlang do módulo *SendRecv*

Apêndice 4 – Codigos *Benchmarck AlltoAll*

```
1  package alltoall;
2
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  import persistencia.Salvar;
7
8
9  public class AllToAllPrincipal {
10         public static void main(String[] args) {
11                 int tamMsg = Integer.parseInt(args[0]);
12                 int num_proc = Integer.parseInt(args[1]);
13                 int num_rep = Integer.parseInt(args[2]);
14
15                 ExecutorService executor = Executors.newFixedThreadPool(num_proc);
16
17                 String localSaida = Salvar.OUT_PATH + "alltoall.txt";
18
19                 AllToAll alltoall = new AllToAll(localSaida,num_proc,num_proc,num_rep,
20                 tamMsg,executor);
21
22                 alltoall.run();
23         }
24 }
```

Código 9.1.14: Código Java da classe *AllToAllPrincipal*

```
1  package alltoall;
2
3  public class Message {
4
5          public volatile Object value;
6          public int source;
7
8          public Message(int source, Object value) {
9                  this.source = source;
10                 this.value = value;
11         }
12 }
```

Código 9.1.15: Código Java da classe *Message*

```
1  package alltoall;
2  import java.util.Arrays, java.util.concurrent.BlockingQueue, java.util.concurrent.ExecutorService,
3  java.util.concurrent.LinkedBlockingQueue;
4  import persistencia.Salvar;
5
6  public class AllToAll{
7          private final String NOME = "Anel",String OUT_LOCATION;
8          private final int NUM_PROC,NUM_REP,TAM_MSG;
9  private BlockingQueue<Message> mailbox = new LinkedBlockingQueue<Message>(),Message msg,
10 ExecutorService executor;
11
12         public AllToAll(String outLocation,int num_proc,int threads,int num_rep,int tamMsg,
13         ExecutorService executor){
14                 OUT_LOCATION = outLocation;
15                 NUM_PROC = num_proc;
16                 NUM_REP = num_rep;
17                 TAM_MSG = tamMsg;
18                 this.executor = executor;}
19
20         public void send(Message m) {
21                 mailbox.add(m);}
22         public long getTimeMicro() {
23                 return System.nanoTime() / 1000;}
24         public byte[] generateData(int tamDados) {
25                 byte[] dado = new byte[tamDados];
26                 return dado;}
27         public Proc[] spawnProcs(int n) {
28                 Proc[] procList = new Proc[n];
29                 for (int i = 0; i < n; i++) {
30                         procList[i] = new Proc(i+1);}
31                 return procList;}
32
33         private long[] finalize(int n){
34                 try {   long[] endTimeList = new long[n];
35                         for(int i=0; i < n; i++){
36                                 Message recvMsg = mailbox.take();
37                                 Object[] tuple = (Object[]) recvMsg.value;
38                                 long endTime = (Long) tuple[1];
39                                 endTimeList[i] = endTime;}
40                         return endTimeList;
41                 } catch (InterruptedException e) {
42                         System.out.println("Thread AllToAll Interrmopida!");
43                         return null;}}
44         private long sum(long[] list){
45                 long result = 0;
46                 for(long elem : list){
47                         result += elem;}
48                 return result;}
49         public void run(){
50                 long spawnStart, spawnEnd, timeSpawn, execStart, timeMin, timeMax, timeAvg;
51                 long[] endTimeList, timeList;
52
53                 msg = new Message(0, generateData(TAM_MSG));
54                 spawnStart = getTimeMicro();
55                 Proc[] procList = spawnProcs(NUM_PROC);
56                 spawnEnd = getTimeMicro();
57                 timeSpawn = spawnEnd - spawnStart;
58
59                 execStart = getTimeMicro();
60                 for (Proc proc : procList) {
61                         proc.setRepetitions(NUM_REP);
62                         proc.setData(((byte[]) msg.value).clone());
63                         proc.setParent(this);
64                         proc.setProcList(procList.clone());
65                         executor.execute(proc);}
66                 endTimeList = finalize(NUM_PROC);
67                 timeList = new long[NUM_PROC];
68                 for (int i=0; i < NUM_PROC; i++){
69                         timeList[i] = endTimeList[i] - execStart;}
70                 Arrays.sort(timeList);
71                 timeMin = timeList[0];
72                 timeMax = timeList[NUM_PROC-1];
73                 timeAvg = sum(timeList)/timeList.length;
74         Salvar.writeResultAlltoall(OUT_LOCATION,TAM_MSG,NUM_REP,NUM_PROC,timeMin,timeMax,timeAvg,
75         timeSpawn);
76                 System.exit(0);}}
```

Código 9.1.16: Código Java da classe *AllToAll*

```
1  package alltoall;
2
3  import java.util.concurrent.BlockingQueue;
4  import java.util.concurrent.LinkedBlockingQueue;
5
6  public class Proc implements Runnable {
7          private int nodeId;
8          private BlockingQueue<Message> mailbox = new LinkedBlockingQueue<Message>();
9
10         private AllToAll parent;
11
12         private int repetitions;
13         private Message msg;
14         private Proc[] procList;
15
16         public Proc(int id) {
17                 this.nodeId = id;}
18
19         public void setRepetitions(int repetitions){
20                 this.repetitions = repetitions;}
21
22         public void setData(byte[] data){
23                 this.msg = new Message(nodeId, data);}
24
25         public void setParent(AllToAll parent){
26                 this.parent = parent;}
27
28         public void setProcList(Proc[] procList){
29                 this.procList = procList;}
30
31         public int getNodeId(){
32                 return nodeId;}
33
34         public void send(Message m) {
35                 mailbox.add(m);}
36
37         public Message recv() throws InterruptedException {
38                 Message msg = mailbox.take();
39                 return msg;}
40
41         private void alertEnd(long endTime){
42                 Object[] tuple = new Object[2];
43                 tuple[0] = this;
44                 tuple[1] = endTime;
45                 Message alert = new Message(nodeId, tuple);
46                 parent.send(alert);}
47
48         public long getTimeMicro() {
49                 return System.nanoTime() / 1000;}
50
51         private void scatter(Proc[] procs, Message msg){
52                 for(Proc p : procs){
53                         p.send(msg);}}
54
55         private Message[] gather(int n) throws InterruptedException{
56                 Message[] msgs = new Message[n];
57                 for(int i=0; i<n; i++){
58                         msgs[i] = recv();}
59                 return msgs;}
60
61         public void run() {
62                 System.out.println("COMECOU: P" + nodeId);
63                 try {
64                         for(int i=1; i <= repetitions; i++){
65                                 scatter(procList, msg);
66                                 @SuppressWarnings("unused")
67                                 Message[] recvMsgs = gather(procList.length);}
68                         long endTime = getTimeMicro();
69                         alertEnd(endTime);}
70                         catch (InterruptedException e) {
71                         System.out.println("Thread " + nodeId + " Interrmopida!");}
72 }}
```

```
1  -module(alltoall).
2  -export([run/3]).
3
4  -include("conf.hrl").
5  -import(persist, [write_result/3]).
6  -import(medicoes, [generate_data/1, time_microseg/0]).
7
8  run(DataSize, Rep, QtdProcs) ->
9      Data = generate_data(DataSize),
10
11     SpawnStart = time_microseg(),
12     Procs_list = create_procs(QtdProcs, Data),
13     SpawnEnd = time_microseg(),
14
15     SpawnTime = SpawnEnd - SpawnStart,
16
17     ExecStart = time_microseg(),
18     lists:foreach(fun(X) -> X ! {init, Rep, Procs_list, self()} end, Procs_list),
19     EndTime_list = finalize(QtdProcs),
20
21     Time_list = [ EndTime - ExecStart || EndTime <- EndTime_list ],
22
23     TimeMin = lists:min(Time_list),
24     TimeMax = lists:max(Time_list),
25     TimeAvg = lists:sum(Time_list) / length(Time_list),
26
27     write_result(alltoall, ?OUT_PATH, [Data, Rep, QtdProcs, TimeMin, TimeMax, TimeAvg, SpawnTime]),
28     erlang:halt().
29
30 create_procs(QtdProcs, Data) -> create_procs(QtdProcs, Data, []).
31 create_procs(0, _, Procs_list) -> Procs_list;
32 create_procs(QtdProcs, D, L) ->
33     create_procs(QtdProcs-1, D, [spawn(fun() -> wait_init(D) end) | L]).
34
35 finalize(QtdProcs) -> finalize(QtdProcs, []).
36 finalize(0, EndTime_list) -> EndTime_list;
37 finalize(QtdProcs, EndTime_list) ->
38     receive
39         {done, _From, ExecEnd} ->
40             finalize(QtdProcs-1, [ExecEnd | EndTime_list])
41     end.
42
43 wait_init(Data) ->
44     receive
45         {init, Rep, Procs_list, Parent} ->
46             alltoall(Rep, Procs_list, Data),
47             ExecEnd = time_microseg(),
48             Parent ! {done, self(), ExecEnd}
49     end.
50
51 alltoall(0, _, _) -> done;
52 alltoall(Rep, Plist, Data) ->
53     scatter(Plist, Data),
54     _RecvData_list = gather(Plist),
55     alltoall(Rep-1, Plist, Data).
56
57 scatter(Plist, Data) ->
58     lists:foreach(fun(P) -> P ! {msg, self(), Data} end, Plist).
59
60 gather(Plist) -> gather(Plist, []).
61
62 gather([], RecvData_list) -> RecvData_list;
63 gather([From| Plist], Dlist) ->
64     receive
65         {msg, From, Data} ->
66             gather(Plist, [{From, Data}| Dlist])
67     end.
```

Código 9.1.18: Código Erlang do módulo *AlltoAll*