CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Análise e implementação de algoritmo paralelo distribuído para fatoração de números inteiros utilizando Java RMI

LUIS PAULO RODRIGUES PASTOR

Marília, 2013

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Análise e implementação de algoritmo paralelo distribuído para fatoração de números inteiros utilizando Java RMI

Monografia apresentada ao Centro Universitário Eurípides de Marília como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.
Orientador: Prof. Mauricio Duarte

Marília, 2013

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Luis Paulo Rodrigues Pastor

Análise e implementação de algoritmo paralelo distribuído para fatoração de números inteiros utilizando Java RMI.

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 10.0 (dez)

Orientador: Mauricio Duarte

1°. Examinador: Rodolfo Barros Chiaramonte

2°. Examinador: Emerson Alberto Marconato

Marília, 02 de dezembro de 2013.

AGRADECIMENTOS

Todo agradecimento vai para as pessoas que de alguma forma, direta ou indiretamente, contribuíram para que eu chegasse a este ponto de minha vida.

Os obstáculos que são colocados parecem que até de propósito, fazem com que a nossa vitória se torne ainda mais grandiosa. A realização vem com trabalho, dedicação e pensamento positivo. Sempre há um caminho quando se tem um obstáculo, e este caminho às vezes pode até se tornar um atalho.

Agradeço primeiramente a Deus, por ter me conduzido na realização deste objetivo, por me dar a motivação e vontade de concluir este ciclo, permitindo que chegasse até aqui, e pela mão colocada sobre mim para que todas as dificuldades fossem superadas.

Agradeço a minha família, meu pai Moacir, minha mãe Rosângela, e meu irmão João Pedro, pois sempre acreditaram que chegaria até aqui, e me deram a oportunidade para uma formação profissional. A colaboração que tiveram até mesmo em gestos simples nos infinitos dias desses quatro anos também merece um agradecimento especial.

Agradeço a meu orientador, professor Mauricio Duarte, pela atenção e prontidão em todos os momentos, principalmente nos que estive meio perdido.

Agradeço ao professor Ricardo José Sabatine, pela colaboração e conselhos dados ao final e inicio de suas aulas ao longo do desenvolvimento deste projeto.

Agradeço aos meus amigos, que estiveram comigo nesta mesma caminhada, e através dos conselhos e palavras de motivação trocadas, também são responsáveis para que chegasse até aqui.

Muito Obrigado a todos!

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE GRÁFICOS

LISTA DE SIGLAS

AES – *Advanced Encryption Standard*
API – *Application Programming Interface*
DNS – *Domain Name System*
Gbps – *Gigabits per second*
HPC – *High Performance Computing*
IP – *Internet Protocol*
LAN – *Local Area Network*
Mbps – *Megabits per second*
MIMD – *Multiple Instruction Multiple Data*
MISD – *Multiple Instruction Single Data*
MPP – *Massively Parallel Processing*
RMI – *Remote Method Invocation*
RPC – *Remote Procedure Call*
RSA – *Rivest Shamir Adleman*
SIMD – *Single Instruction Multiple Data*
SISD – *Single Instruction Single Data*
TCP – *Transmission Control Protocol*
UDP – *User Datagram Protocol*
URL – *Uniform Resource Locator*
WAN – *Wide Area Network*

RESUMO

O objetivo deste trabalho é efetuar uma análise do desempenho de uma aplicação que implemente um algoritmo com a função de realizar a fatoração de números inteiros de maneira paralela distribuída, baseando-se na metodologia de fatoração proposta por Fermat e utilizando o mecanismo da linguagem Java para programação distribuída, chamado RMI (*Remote Method Invocated*). Este algoritmo será executado de maneira sequencial, e de maneira paralela, sobre um cluster composto por máquinas de características homogêneas, utilizando o paradigma mestre-escravo, onde cada escravo participante da execução terá a responsabilidade de executar um trecho de código, cooperando com a execução. A fatoração de números inteiros é um dos métodos utilizados para que se consiga quebrar uma chave criptográfica RSA, possuindo um tempo de execução extremamente grande na fatoração de um número de um cenário real de criptografia, por isso muitos estudos são realizados para encontrar maneiras de obter ganhos no tempo de execução de uma fatoração, e por estes motivos esta abordagem foi selecionada para ser aplicada neste projeto. Após a execução e coleta dos resultados obtidos através de testes realizados em diferentes cenários, os resultados serão analisados, e então, será verificado e comparado o desempenho das diversas execuções realizadas, dando a possibilidade de se verificar em quais cenários houve ganho ou perda de desempenho, e os fatores que impactam diretamente no desempenho da aplicação.

Palavras-Chave: Sistemas Distribuídos, Java RMI, Computação Paralela, Clusters, Fatoração de Números Inteiros, Criptografia RSA.

ABSTRACT

The objective of this paper is to analyze the performance of an application that implements an algorithm to perform the integers factorization in parallel distributed, based on the methodology factorization proposed by Fermat and using the mechanism of the Java language to distributed programming, called RMI (Remote Method Invocated). This algorithm will be executed sequentially and in parallel on a cluster composed of homogeneous machines characteristics, using the master-slave paradigm, where each participant slave execution has the responsibility of executing a piece of code, cooperating with the execution. The factorization of integers is responsible for one of the methods that can break a cryptographic key RSA, having an execution time extremely large in the number factorization from a real scenario of encryption, so many studies are conducted to find ways to get gains in the runtime of a factorization, and for these reasons this approach was selected to be applied in this project. Upon execution and collection of results obtained from tests carried out in different scenarios, the results are analyzed, and then will be checked and compared the performance of several executions, giving the possibility to check on what scenarios there was a gain or loss of performance and the factors that impacted directly in the application performance.

Keywords: Distributed Systems, Java RMI, Parallel Computing, Clusters, Integer Factorization, RSA Encryption.

INTRODUÇÃO

Grandes desafios computacionais em vários segmentos da computação desafiam os profissionais desta área que é cada vez mais crescente pelo mundo todo. Sempre houve uma maneira de resolver problemas que estavam na frente de pesquisadores naquele exato momento ao longo da história, ou por necessidade daquele contexto histórico, ou pelo simples prazer de criar algo novo que fosse mudar o dia-a-dia de pessoas "normais". Com o passar do tempo e a evolução dos computadores, vem aumentando o grau dos desafios que a computação tenta solucionar, e computadores que antes eram suficientes se tornaram incapazes de ajudar.

A utilização de supercomputadores é uma alternativa para problemas mais complexos, porém o seu alto custo, e sua dificuldade em se aumentar o poder de processamento, inviabilizam o uso desta solução. Uma solução financeiramente viável e interessante é o uso de sistemas distribuídos com processamento paralelo. A combinação entre estas áreas da computação pode ajudar profissionais de vários segmentos da computação a resolverem certos problemas de maneira em que o processamento possa ser dividido e distribuído pelas máquinas que formam um sistema, desse modo podem ser diminuídos os custos com hardware, (pois interligar máquinas caseiras em uma rede de alta velocidade é mais barato que a aquisição de um supercomputador), e possivelmente obter um melhor desempenho no processamento (DANTAS, 2005).

Para implementação de um sistema distribuído é necessário que haja uma maneira em que os computadores interligados se comuniquem pela rede. Existem alguns métodos utilizados na computação como *Sockets*, RPC, entre outros, porém existe um método para a programação distribuída exclusivo da linguagem Java, o RMI *(Remote Method Invocated).*

O RMI proporciona todas as ferramentas necessárias para a implementação de um sistema distribuído utilizando a linguagem Java. Como esta API faz a função de middleware, o programador fica despreocupado com diferenças entre tecnologias de rede e de hardware, por exemplo. A comunicação então é feita pela invocação de métodos remotos, onde um cliente utiliza os serviços disponibilizados pelo servidor, esta comunicação ocorre de maneira síncrona (PEREIRA, 2003).

O obstáculo que impede o paralelismo através de invocações remotas acontece justamente pela comunicação ser síncrona. Uma maneira de contornar este obstáculo é com a utilização de threads, que podem ser criadas para efetuarem invocações remotas no mesmo instante, de modo paralelo.

Observando esta possibilidade, surgem diversas aplicações que podem ser exploradas utilizando este ambiente paralelo distribuído para execução. Problemas complexos que divididos em partes menores podem gerar desempenho melhor que um processamento sequencial. Para isso, é necessário que testes sejam realizados para verificar em que momento é benéfico o paralelismo e quais fatores impactam em uma melhora ou piora no desempenho.

OBJETIVOS

Este trabalho tem como objetivo efetuar uma análise e verificação no desempenho e custo/beneficio da paralelização de um algoritmo para fatoração de números inteiros, através da execução sequencial e paralela, utilizando o mecanismo RMI da linguagem Java para a comunicação entre as máquinas em diversos cenários de execução, com uma variação na quantidade de escravos utilizados nos mesmos. O algoritmo será implementado com base em um método para fatoração de números inteiros proposto por Fermat. A fatoração de números inteiros é uma maneira para que se consiga quebrar chaves criptográficas RSA, onde a fatoração irá resultar no encontro de dois fatores primos, que aplicados a outros procedimentos possibilitam a quebra da chave criptográfica.

Porém a fatoração deste número demanda um tempo de processamento extremamente grande e se torna inviável para execução em uma única unidade de processamento. Uma solução seria a paralelização deste problema. Com isso, espera-se por meio deste trabalho, verificar através das análises no desempenho de testes com números inteiros de menor tamanho, identificar cenários propícios para a paralelização e fatores que causam impacto no desempenho desta aplicação.

ORGANIZAÇÃO DO TRABALHO

A organização do trabalho procurou apresentar primeiramente os conceitos que envolvem a elaboração desta monografia. No primeiro capítulo será apresentada toda parte conceitual sobre sistemas distribuídos e suas características principais. Na sequencial, será apresentado um capítulo contendo informações que caracterizam toda parte conceitual da computação paralela. No terceiro capítulo, serão abordados mecanismos para comunicação remota e ambientes de troca de mensagens. Ao final da parte conceitual, serão apresentados todos os passos para elaboração e implementação do algoritmo sequencial e do algoritmo

paralelo distribuído. E por fim, os resultados obtidos com os testes realizados de maneira sequencial e paralela, seguido das conclusões finais sobre o trabalho.

1. SISTEMAS DISTRIBUÍDOS

Um grande avanço tecnológico ocorreu na década de 80 com o desenvolvimento de processadores de alto desempenho e com a invenção de redes de computadores de alta velocidade, como as LAN's e as WAN's.

As LAN's são redes pequenas de tamanho restrito que podem interligar de dezenas a centenas de computadores, que pertencem a uma mesma organização ou a um mesmo local. Essas redes locais se caracterizavam por possuir um pequeno tráfego de informações, porém o avanço da tecnologia fez com que essas redes suportem um grande trafego de informações. (TANENBAUM; STEEN, 2007).

Já as redes de longa distância, as WAN's, interligam milhões de computadores por todo o planeta a uma velocidade que pode chegar a gigabits por segundo. Esse avanço tecnológico permitiu que sistemas utilizando vários computadores conectados por uma rede de alta velocidade fossem facilmente implementados (TANENBAUM; STEEN, 2007).

O principal objetivo que leva a construção de tal sistema é a necessidade dos usuários em compartilhar recursos, sendo que o maior exemplo de um sistema distribuído é a internet, que comunica e compartilha recursos com usuários do mundo todo. A comunicação entre os componentes de um sistema distribuído para o compartilhamento de recursos é feita através da troca de mensagem, e por estes componentes serem independentes um dos outros, podem haver diferenças em questões como hardware, rede, sistemas operacionais, que dificultem ou até mesmo impeçam a comunicação entre os componentes. Para tratar este problema utilizamos uma camada de software denominada middleware (COULOURIS; DOLLIMORE; KINDBERG, 2007).

1.1. Middleware

O cenário onde vários computadores independentes estão conectados utilizando uma rede de alta velocidade, dando ao usuário a impressão de estar utilizando um sistema único e coerente, é uma das definições propostas para um sistema distribuído. Estes vários computadores que fazem parte de um sistema distribuído são componentes independentes, por isso podem possuir características diferentes em relação ao hardware, e para que cada componente desse sistema consiga relevar essas diferenças buscando a troca de informações com os outros componentes, é necessário uma camada de software denominada middleware.

Esta camada além de mascarar a heterogeneidade encontrada nos tipos de rede utilizada, no hardware, e sistema operacional, abstrai e facilita ao programador, questões da implementação do sistema distribuído (COULOURIS; DOLLIMORE; KINDBERG, 2007). A camada de middleware está ilustrada na Figura 1.

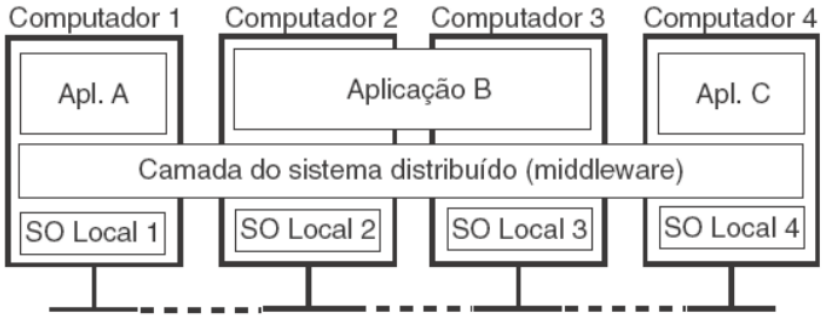

Figura 1 – Sistema Distribuído organizado como Middleware (TANENBAUM, 2007)

O middleware faz com que as aplicações e usuários possam interagir de maneira uniforme e consistente com um sistema distribuído. Alguns exemplos de middleware são: Corba, Java RMI, este último será melhor abordado no item 3.4.

1.2. Objetivos de um Sistema Distribuído

A implementação de um sistema distribuído deve traçar alguns objetivos para que este possua características que irão facilitar a vida de quem utiliza e de quem faz a manutenção no sistema (TANENBAUM; STEEN, 2007). Oferecer facilidade, segurança e eficácia aos usuários e as aplicações que se utilizam do sistema é o objetivo final na implementação de um sistema distribuído. Para atingir este objetivo, alguns pontos devem ser tratados na implementação do sistema.

Os serviços oferecidos pelo sistema devem ser padronizados, e essa questão é tratada com a utilização de interfaces, que são comumente usadas para oferecer padrões que definem a sintaxe e a semântica dos serviços oferecidos.

Para facilitar a inserção de novos componentes, ou até mesmo a modificação dos que já estão em funcionamento, o sistema distribuído deve ser extensível, ou seja, uma alteração feita em um componente do sistema não deve obrigar a uma nova alteração em um segundo componente, para isso é importante que o sistema seja bem organizado, preferencialmente divido em componentes de tamanho reduzido.

Com a rápida expansão da internet pelo mundo todo, fica cada vez mais indispensável que um sistema distribuído seja escalável. Segundo Neuman, (NEUMAN, 1994) escalabilidade pode ter três dimensões diferentes no mínimo, a primeira é em relação ao tamanho, a facilidade de se aumentar o número de usuários e recursos disponibilizados pelo sistema, a segunda é em relação à escalabilidade em termos geográficos, onde um usuário pode estar longe geograficamente de um recurso, e a terceira é a escalabilidade em relação a termos administrativos, a facilidade em se gerenciar um sistema que envolva dados e recursos de muitas organizações diferentes (COULOURIS; DOLLIMORE; KINDBERG, 2007).

A escalabilidade de um sistema pode enfrentar alguns problemas principais, que são: serviços centralizados, dados centralizados, e algoritmos centralizados. Quando um serviço oferecido é implementado por apenas um servidor, e com o passar do tempo o número de usuários e aplicações que utilizam aquele serviço vai aumentando, a capacidade de armazenamento e de processamento pode ficar limitada, o que impede o crescimento do sistema, este caso exemplifica o problema com serviços centralizados.

Dados centralizados são tão problemáticos quanto serviços centralizados, pois são utilizados apenas um banco de dados para armazenar as informações, isto sobrecarregaria os meios que o acessam.

Para impedir que se tenha uma quantidade excessiva de mensagens no sistema, algoritmos centralizados devem ser evitados a todo custo, pois o excesso de mensagens sobrecarregaria a rede. O ideal é utilizar algoritmos que sejam descentralizados, e em geral suas características são: nenhuma das máquinas que compõem o sistema possui todas as informações sobre o estado do mesmo, as decisões tomadas pelas máquinas são baseadas nas informações que as mesmas possuem, a falha de uma máquina não interfere no algoritmo, e não há dependência de um relógio global (COULOURIS; DOLLIMORE; KINDBERG, 2007).

Algumas técnicas podem ser aplicadas para minimizar ou até mesmo resolver estes problemas de escalabilidade apresentados anteriormente.

Ocultar as latências de comunicação: quando se tenta evitar ao máximo esperar por respostas remotas de serviços requisitados. A ideia é utilizar uma comunicação assíncrona que ao invés do processo do lado do cliente ficar ocioso esperando uma resposta do processo do lado do servidor, ele executa outra atividade enquanto espera a resposta.

Ocultar as latências de distribuição consiste em dividir um componente em partes menores e distribuí-lo pelo sistema, como no caso do Sistema de Nomes de Domínio da Internet (DNS), que distribui o serviço por vários servidores que tratam certa quantidade de

requisições, essa técnica evita de certo modo sobrecarregar os servidores e possibilita o aumento da Internet pelo mundo todo.

Considerando que estes problemas de escalabilidade aparecem frequentemente sob forma de perda de desempenho, é interessante replicar componentes pelo sistema distribuído, a replicação vai aumentar a disponibilidade, ajudar no balanceamento entre os componentes, e pode ainda ocultar alguns problemas de comunicação (TANENBAUM; STEEN, 2007).

A questão da transparência é sem dúvida a meta mais discutida que um sistema distribuído deve seguir, sua definição consiste na ocultação de aspectos que fazem com que um sistema distribuído aparente ser um sistema único e consistente.

Diferentes tipos de transparência podem estar presentes em um sistema distribuído, como a transparência de acesso, de localização, migração, relocação, replicação, concorrência, e de falha.

A transparência de acesso oculta as diferenças existentes na representação dos dados e no modo como um recurso é acessado. A transparência de localização oculta para os usuários a real localização do dado ou serviço requisitado, que é o caso da URL, que mascara em qual servidor seu serviço está situado.

Quando um recurso pode ser movido para outro local sem afetar o modo como este recurso é acessado, o sistema possui uma transparência de migração. Já a transparência de relocação trata de ocultar a movimentação de um recurso em uso sem que o usuário perceba essa mudança. Na transparência de replicação o usuário não consegue identificar se o recurso que esta utilizando é na verdade uma réplica do recurso original.

Transparência de concorrência acontece quando dados podem ser acessados por mais de um usuário ao mesmo tempo sem que este perceba. Por fim, a transparência a falha tem como objetivo mascarar as falhas que um sistema distribuído pode apresentar, ou mascarar que o sistema se recuperou de uma falha, este tipo de transparência é difícil ou até mesmo impossível de tratar, pois um recurso pode apresentar vários estados, e para o usuário, no momento do acesso o recurso pode estar apenas disponível ou indisponível, não mostrando seu real estado, como exemplo, um recurso pode estar apenas ocupado (indisponível no momento), onde uma tentativa de acesso após um tempo traria o recurso como disponível, ou estar danificado onde qualquer tentativa de acesso a qualquer momento traria o recurso como indisponível (COULOURIS; DOLLIMORE; KINDBERG, 2007).

Contudo, aplicar ao máximo as técnicas de transparência em um sistema distribuído nem sempre é o melhor a se fazer, por não ser viável financeiramente e por às

vezes o melhor a se fazer é mostrar o problema ao usuário para que ele decida pelo que fazer. (TANENBAUM; STEEN, 2007).

1.3. Sistemas de Computação Distribuídos

Existem diferentes tipos de sistemas distribuídos referentes à sua utilização, alguns deles são: sistemas de computação distribuídos, sistemas de informação distribuídos e sistemas embutidos distribuídos (TANENBAUM; STEEN, 2007). O tipo de sistema distribuído que será utilizado neste projeto são os sistemas de computação distribuídos, que será discutido na sequencia.

Dentro da classe dos sistemas de computação distribuídos pode-se fazer uma separação entre um grupo onde são utilizados um conjunto de computadores com hardware semelhante, com o mesmo sistema operacional e interligados por meio de uma rede local de alta velocidade, este grupo pode ser denominado de computação de cluster. No outro grupo as características são totalmente diferentes da computação de cluster, com computadores que podem apresentar uma grande diferença de hardware, sistema operacional e tecnologia de rede e ainda podem estar em redes de domínios administrativos diferentes, é a chamada computação em grade (TANENBAUM; STEEN, 2007).

Os sistemas de computação de cluster são na maioria das vezes designados a resolver algum problema complexo, que é dividido entre os nós que compõem o cluster para quebrar o problema em tarefas de menor complexidade, deste modo, estas partes do problema são executadas em paralelo nos nós, utilizando também a técnica da computação paralela, que será mais bem abordada no capitulo 2.

Processamento de dados, computação cientifica, análises financeiras, previsões meteorológicas são algumas das aplicações que comumente utilizam computação de cluster, todas elas necessitam de um alto grau computacional. Este tipo de computação ficou muito mais acessível com o passar do tempo, aonde o preço de máquinas com alto desempenho não chega a ser um absurdo (BACELLAR, 2010). As vantagens apresentadas pelos sistemas de cluster são: expansibilidade, baixo custo, alta disponibilidade, tolerância a falhas, e balanceamento de carga, porém também apresentam desvantagens como: a manutenção dos nós, monitoração dos nós, e o gargalo que se forma com a troca de informações (BACELLAR, 2010).

1.3.1. Clusters

Quando dois ou mais computadores estão interligados, normalmente próximos geograficamente e estão sendo utilizados para resolver um mesmo problema, temos a definição de um cluster. Os computadores que formam um cluster são denominados nós, nodes ou nodos. Cada nó do cluster deve passar ao usuário a imagem de que o sistema é único e coerente, para isso é necessário que haja transparência do uso da computação paralela distribuída em questões como: métodos de sincronização, comunicação (troca de mensagens), distribuição de tarefas, e outros fatores que podem afetar a transparência do sistema. Lembrando que o conceito de transparência deve estar quase que totalmente aplicado em um sistema distribuído, pois nem sempre é interessante mascarar a distribuição do sistema para o usuário (TOULOUEI, 2010).

A Figura 2 apresenta um exemplo genérico de um cluster, formado por alguns nós denominados nós escravos, que estão interligados por uma rede de alta velocidade e são responsáveis pela execução das instruções, e um nó controlador que fica responsável pela distribuição e controle das instruções, além do monitoramento dos nós escravos.

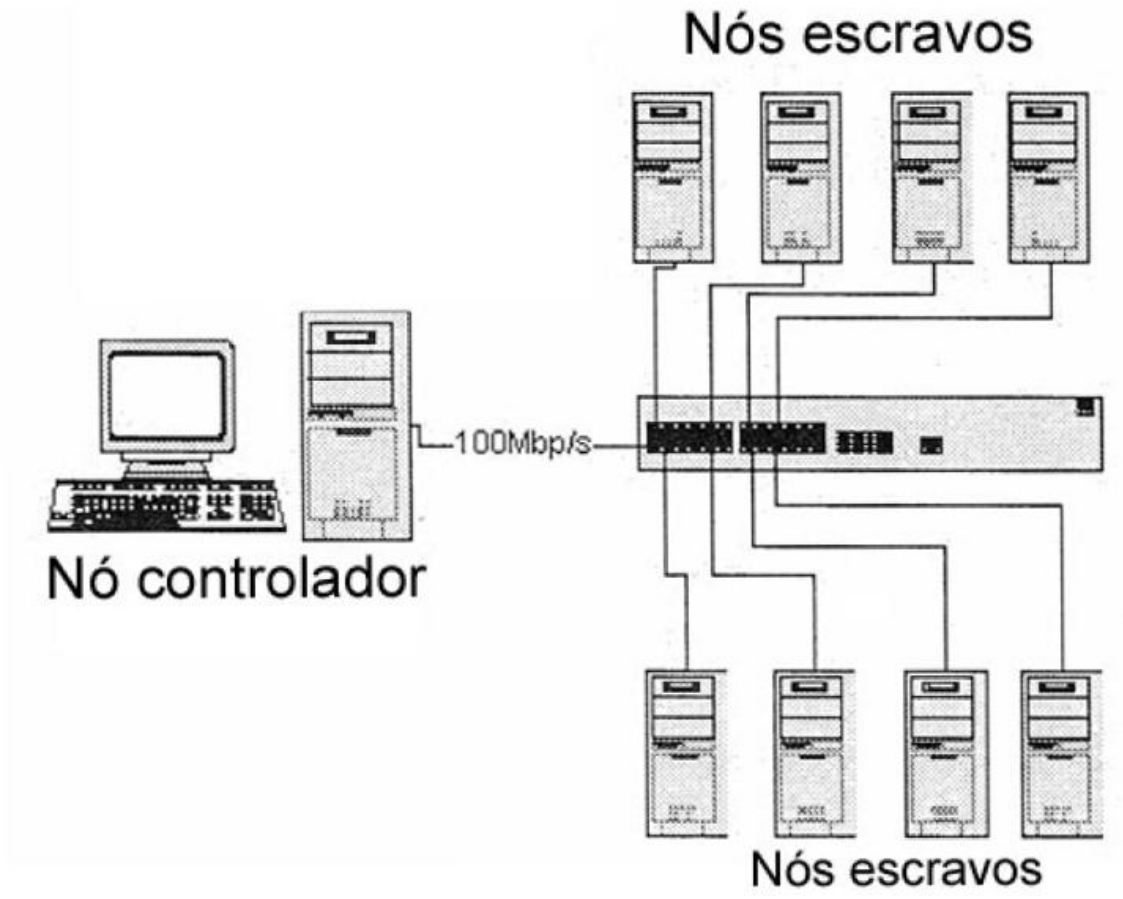

Figura 2 – Exemplo genérico de um cluster (TOULOUEI, 2010).

Os clusters estão sendo muito utilizados devido a ser mais viável financeiramente que a aquisição de um supercomputador, além de que um cluster proporciona flexibilidade no momento em que há a necessidade de se aumentar o poder de processamento, ou então diminuir o poder de processamento para problemas menos complexos, a remoção ou adição de nós ao cluster, implica apenas em um balanceamento da carga de processamento (DANTAS, 2005).

Os componentes necessários para que seja possível projetar um cluster, basicamente são:

- Os computadores que representam os nós, e são responsáveis pela parte da execução dos algoritmos ou trechos de um algoritmo. Alguns clusters utilizam um nó para ser o responsável pela distribuição de tarefas aos nós escravos, este nó que distribuí tarefas é denominado nó mestre.
- A parte de transmissão de dados fica por conta das redes de internet, que é composta pelo meio físico, mecanismos de controle de transmissão de dados, e pela política de sincronização. A tecnologia de rede mais utilizada é a Ethernet,

sendo composta pela Fast Ethernet e pela Gigabit Ethernet. A Fast Ethernet é muito utilizado em clusters, podendo chegar a uma velocidade que varia entre 10 Mbps a 100 Mbps, já a Gigabit Ethernet pode chegar a 1 Gbps (BACELLAR, 2010).

- Um sistema operacional que tem a função de controlar o funcionamento do computador, gerenciando a utilização e o compartilhamento de todos os recursos disponíveis. O sistema operacional utilizado pode ser especifico para clusters, ou então sistemas operacionais comuns para computadores pessoais.
- Bibliotecas de passagem de mensagem, para a comunicação entre processos, estas bibliotecas serão melhores abordadas no capitulo 3.
- Aplicações paralelas, para se utilizarem do serviço final que o cluster disponibiliza, a execução em paralelo, lembrando que se for extraído ao máximo o paralelismo das aplicações, melhor irá ser o desempenho na execução.

As inúmeras vantagens, dentre elas, a facilidade de se agrupar computadores pessoais, e a viabilidade financeira, fez com que os clusters tivessem várias implementações, projetadas por empresas, faculdades, dentre outras organizações. E essas diferentes implementações foram separadas e nomeadas de acordo com suas características semelhantes.

Não há uma taxonomia aceita para se classificar ambientes de clusters, porém, com base em algumas características presentes nos clusters, é possível efetuar uma classificação, tomando como base alguns aspectos como: limite geográfico, utilização dos nós, tipo de topologia, aplicações alvo, e tipos de nós (DANTAS, 2005).

1.3.2. Tipos de Clusters

Em um primeiro momento os clusters estavam relacionados apenas à computação de alto desempenho, conhecida como *High Performance Computing* (HPC), porém surgiram mais duas ramificações com o intuito de refinar o propósito da implementação de um cluster. Essas duas novas ramificações se referem ao cluster de alta disponibilidade, e o cluster de balanceamento de carga. Lembrando que esses dois tipos de cluster podem estar combinados dentre de apenas uma implementação.

Em grande parte das residências, ou até mesmo de empresas, quando um computador de um usuário comum apresenta algum problema e fica inoperante, não há uma

urgência ou certo desespero em estar colocando o computador em funcionamento novamente, pelo fato de não haver perdas significativas. Agora, se o computador em questão é um servidor, que é responsável por disponibilizar todos os dados e serviços informatizados de uma empresa, ou este computador seja uma peça fundamental na logística de uma organização, há neste caso uma urgência para que o computador volte a operar normalmente, pois podem ocorrer perdas significativas em diversos pontos, como o financeiro. Este caso exemplifica o tipo de cluster de alta disponibilidade, que têm a função de manter os dados e serviços de uma organização sempre disponíveis, utilizando a redundância de software e de hardware. Os mesmos dados e serviços são replicados pelos nós do cluster, onde estão sempre sincronizados. Cada nó monitora os outros nós do cluster, sendo que um nó apenas é o principal e quando este nó fica inoperante, outro logo assume seu lugar e passa a oferecer os serviços que estavam sendo disponibilizados naquele momento, esta troca acontece sem que o usuário perceba qualquer tipo de modificação (TOULOUEI, 2010).

Para evitar a sobrecarga sobre apenas um nó do cluster, e consequentemente evitar que o nó fique indisponível, as requisições não devem ser direcionadas para um mesmo nó. Para isso é necessário que haja um balanceamento da carga de processamento, onde todos os nós sejam os nós principais, com o objetivo de melhorar o desempenho, distribuindo trabalhos e balanceando o processamento pelos nós do cluster. Este tipo de cluster, denominado de cluster de balanceamento de carga é muito utilizado para minimizar os graves problemas de serviços online que recebem muitos acessos e podem ficar indisponíveis por conta disso. Um cluster de balanceamento de carga bem projetado permite que sejam mantidas a qualidade de acesso ao serviço e a velocidade da resposta de requisição, além de que todo este balanceamento fica transparente ao usuário (DANTAS, 2005).

Como dito no inicio deste subcapítulo, os clusters de alto desempenho foram o inicio e o primeiro tipo de cluster projetado. O objetivo destes clusters é realizar um processamento paralelo onde se consiga ganhar desempenho. Este alto poder de processamento e os ganhos obtidos com o paralelismo é importantíssimo para serviços como: processamento de dados, previsão do tempo, análises financeiras, modelagem astronômica, entre outros.

O cluster de alto desempenho possui características similares a de um supercomputador, pelo fato dos nós que formam o cluster trabalharem juntos para resolução de um problema, transmitindo a imagem de ser uma única unidade de processamento. Uma diferença que é importante e muitas vezes influenciam na escolha pela utilização do cluster ao invés de um supercomputador, é o valor financeiro, pois o cluster é formado pelo

agrupamento de computadores populares, que com o passar do tempo adquiriram um valor acessível a todos.

1.4. Sistemas Distribuídos Baseados em Objetos.

Este tipo de sistema distribuído tem como base a utilização do modelo cliente-servidor e do paradigma de orientação a objetos, que servirá para que se tenham objetos distribuídos no sistema, onde os recursos e serviços só estarão disponíveis após ser feito um vinculo do processo cliente a um objeto que oferece recursos e serviços, e que se encontra com seu estado localizado no servidor (PEREIRA, 2003).

Quando acontece esta vinculação entre um cliente e um objeto remoto, um proxy é carregado como um objeto local no lado do cliente, implementando a interface do objeto remoto, com a função de montar chamadas a métodos e desmontar chamadas a métodos por meio de mensagens, e passar o resultado desta chamada para o cliente. O objeto em si está localizado no lado do servidor, e disponibiliza a mesma interface que o lado do cliente implementou. Quando são requisitados os serviços oferecidos pelo objeto remoto, um proxy do lado do servidor recebe as requisições, as desmonta e então repassa ao objeto para que o método mais apropriado seja executado (COLOURIS, 2007).

Uma característica de grande parte dos objetos distribuídos é que eles não possuem um estado distribuído, mas sim, um estado que reside em apenas uma máquina. O que é distribuído em outras máquinas são somente as interfaces implementadas pelo objeto (TANENBAUM; STEEN, 2007).

Invocações a métodos de objetos distribuídos podem utilizar definições predefinidas de interfaces, onde essas interfaces devem estar sob o conhecimento dos clientes no momento do seu desenvolvimento, e caso ocorram mudanças na interface, a aplicação cliente deverá ser recompilada para que possa utilizar a interface modificada, esta abordagem de invocação é denominada de invocação estática. Outra abordagem de invocação acontece quando somente há a necessidade de uma aplicação selecionar qual método irá invocar em tempo de execução, esta abordagem é denominada de invocação dinâmica (TANENBAUM; STEEN, 2007).

2. COMPUTAÇÃO PARALELA

Cada vez mais a computação exige um alto grau de complexidade e desempenho para resolver problemas que estão em torno de diversas áreas de atuação, como: áreas científicas, médicas, militares, entre outras, e necessitam de um alto poder computacional para fazer o processamento de algoritmos complexos com grande quantidade de dados. O surgimento da computação paralela se deu pela necessidade de se obter um desempenho que uma única unidade de processamento não conseguia obter (DETOMINI, 2010).

Uma definição para computação paralela apresentada pela literatura, por meio de Quinn (QUINN, 1987) é de que a computação paralela é o processamento de informações que enfatiza a manipulação concorrente dos dados. Esta manipulação pode pertencer a um ou mais processos que objetivam resolver um único problema.

O entendimento de computação paralela envolve o conhecimento de três essências, são elas: concorrência, paralelismo e granularidade (SAITO, 2007). A concorrência é a disputa entre dois ou mais processos para execução. O paralelismo consiste na divisão de uma aplicação em partes, onde essas partes são executadas por vários elementos de processamento no mesmo intervalo de tempo, buscando o melhor desempenho (DETOMINI, 2010). A granularidade é definida pela razão entre o tempo necessário para o cálculo de determinada operação e os custos envolvidos nas trocas de mensagens. A granularidade pode ser fina (conjunto de instruções simples), média ou grossa (conjunto de instruções complexas) (DETOMINI, 2010).

No entanto uma máquina ou um sistema que trabalhe em paralelo não será de muita utilidade se não for extraído ao máximo seu poder de processamento, para que isto ocorra é preciso que se tenham também excelentes programas paralelos, onde se paralelize tudo o que é possível, o que se torna uma tarefa complexa para os programadores (DETOMINI, 2010).

2.1. Taxonomias de Arquiteturas Paralelas

As arquiteturas paralelas procuram agrupar unidades que possuem características semelhantes. Dentre as diversas classificações propostas, a taxonomia de Flynn é muito citada, esta taxonomia teve origem na década de 70, nela são formados quatro grupos para classificar as arquiteturas.

Levando em conta que um computador executa uma sequencia de instruções sobre uma sequencia de dados, são analisados o fluxo de instruções e o fluxo de dados, que podem ser definidos como simples ou múltiplos. As classes formadas foram: SISD (Single Instruction Single Data), MISD (Multiple Instruction Single Data), SIMD (Single Instruction Multiple Data), MIMD (Multiple Instruction Multiple Data (DETOMINI, 2010).

A taxonomia de Tanenbaum subdividiu a classe MIMD em outras duas novas classes, a de multiprocessadores, que são processadores independentes que utilizam apenas um único espaço de endereçamento (fortemente acoplados) efetuando loads e stores (carrega e armazena) para a comunicação entre os processadores. E a de multicomputadores que são processadores que possuem um espaço de endereçamento distinto e que se comunicam através da troca de mensagem utilizando sends e receives (envia e recebe) através de uma rede de comunicação (DETOMINI, 2010).

A Figura 3 representa as várias classes de máquinas com arquitetura paralela que podem existir segundo a taxonomia de Tanenbaum.

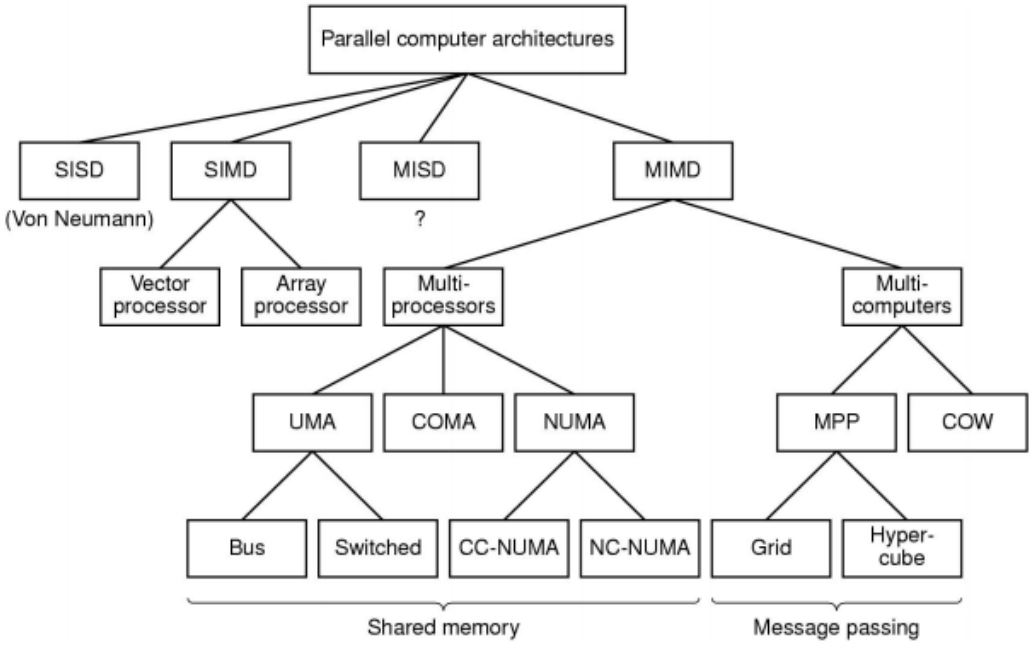

Figura 3 – Classificação das arquiteturas paralelas segundo Taxonomia de Tanenbaum (SAITO, 2007).

Na classe SISD somente um único fluxo de instruções age sobre um único fluxo de dados. Na classe MISD alguns fluxos de instruções agem sobre apenas um único fluxo de dados, onde cada unidade de processamento recebe uma diferente instrução que será executada em cima de um mesmo fluxo de dados. As diversas instruções são executadas ao

mesmo tempo sobre um mesmo espaço de memória, o que torna esta técnica impraticável e a classe sem nenhum aproveitamento prático. Na classe SIMD apenas um único fluxo de instrução é executado sobre diversos fluxos de dados, onde uma mesma instrução é enviada aos processadores que estão envolvidos no trabalho e estes executam em paralelo sobre diferentes fluxos de dados, neste caso é necessário que a memória não esteja implementada como um único modulo de memória. Na classe MIMD diversas instrução são executadas sobre diversos fluxos de instruções, onde um processador recebe um fluxo de instrução diferente das demais instruções enviadas ao outros processadores envolvidos e estas instruções são executadas sobre diferentes fluxos de dados para cada unidade de processamento, assim como no caso da classe SIMD, é necessário que a memória não esteja implementada como um único modulo de memória (DE ROSE, 2001).

2.2. Computação Paralela Distribuída

Computação paralela e sistemas distribuídos são duas ramificações da computação que surgiram por razões diferentes mais que juntas proporcionaram uma maneira viável economicamente de ganhar desempenho. A utilização de sistemas distribuídos trouxe as vantagens de transparência de acesso aos recursos, tolerância a falhas, e confiabilidade. As melhorias obtidas em relação à tecnologia de rede, capacidade de processamento e ferramentas de software tornou os sistemas distribuídos um cenário adequado para a aplicação da computação paralela, que também levou algumas de suas vantagens, como a eficácia e o ganho de desempenho (SAITO, 2007).

O que diferencia essas duas áreas, é que na computação paralela os computadores têm seu processamento dedicado para um fim especifico, e normalmente possuem características semelhantes quanto ao hardware e sistema operacional. Já em sistemas distribuídos, os computadores podem estar distantes geograficamente, na maioria das vezes não dedicando seu processamento para apenas um propósito e podendo ter características totalmente diferentes nas questões de hardware e software (TOLOUEI, 2010).

Um sistema computacional paralelo distribuído pode ser observado na Figura 4, onde fica claro que o trabalho de um nó do sistema é executado utilizando uma memória exclusiva para aquele nó, dando ao sistema a característica de ser fracamente acoplado. Por possuir esta característica, a troca de mensagens é a solução para a comunicação entre os nós do

sistema, levando um alto custo de comunicação e sincronização para sistemas paralelos distribuídos (SAITO, 2007).

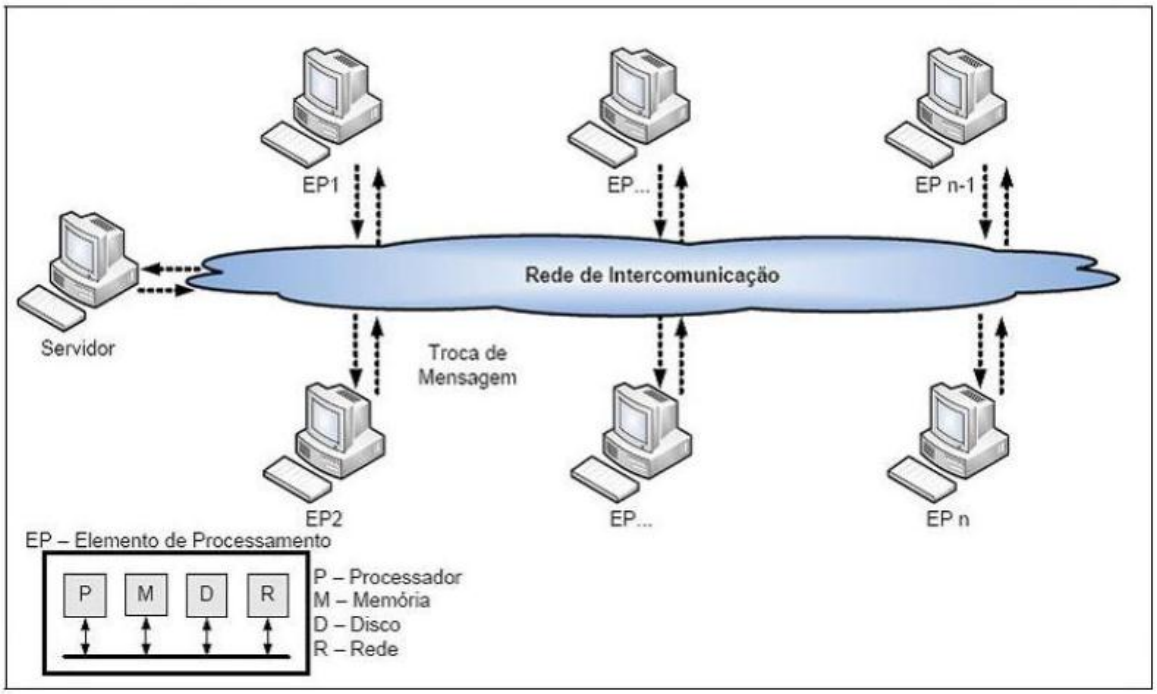

Figura 4 – Sistema computacional paralelo distribuído (SAITO, 2007).

Com o aperfeiçoamento das redes de comunicação e as máquinas populares se tornando máquinas com alto poder de processamento e com um valor acessível a qualquer individuo, os sistemas distribuídos se difundiram com a computação paralela proporcionando compartilhar recursos e ter um alto desempenho (DETOMINI, 2010).

3. MECANISMOS DE COMUNICAÇÃO REMOTA E AMBIENTES DE TROCA DE MENSAGENS

Se não existisse a comunicação entre computadores não existiriam sistemas distribuídos, a comunicação permite que ampliemos para inúmeros, os campos em que a computação pode ser aplicada. E graças a mecanismos que permitem essa comunicação pela troca de mensagens, foi possível aplicar a computação paralela sobre sistemas distribuídos, estes mecanismos estão muitas vezes incorporados com a linguagem que será utilizada.

O modelo de comunicação mais utilizado em aplicações distribuídas é o modelo cliente/servidor. Este modelo funciona basicamente através da requisição-resposta, no qual o cliente faz uma solicitação ao servidor por meio de uma mensagem, requisitando um serviço que o mesmo oferece, após receber esta solicitação o servidor executa a tarefa e retorna ao cliente o resultado da solicitação. Sabendo disto, existem diferentes maneiras de fazer uma comunicação em um sistema distribuído utilizando este modelo, mecanismos de mais baixo nível como sockets, e mecanismos de níveis mais altos, como o RPC, e no caso da utilização da linguagem de programação Java, o RMI. Para paralelização de aplicações existem também algumas bibliotecas que auxiliam nesta questão, como é o caso do MPI.

3.1. Sockets

Este mecanismo de mais baixo nível possibilita a comunicação entre duas máquinas, onde uma máquina espera por conexões em uma determinada porta.

Quando um processo necessita estabelecer um canal de comunicação com outra máquina, uma porta local é selecionada para formar um canal de comunicação com uma outra porta de um processo remoto, estas portas são denominadas sockets e possuem a função de estabelecer este canal de comunicação, efetuando o envio e recebimento de informações entre os dois processos.

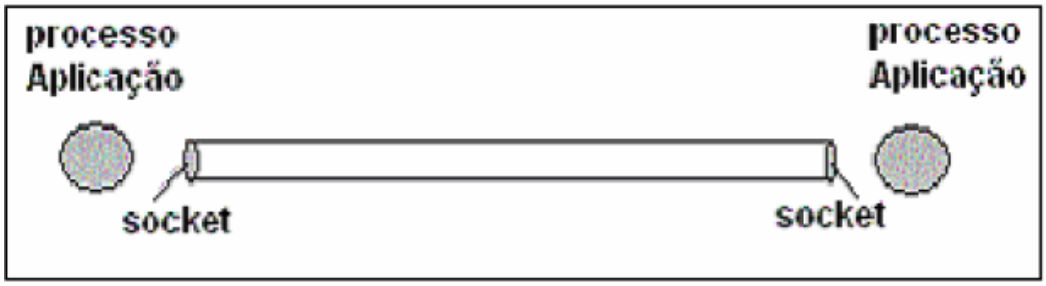

Figura 5 – Sockets (DA SILVA; FRANCO; AVELINO, 2006)

A comunicação entre processos utilizando sockets pode ser feita baseando-se em dois modos diferentes. Um dos modos é orientado a conexão, que utiliza o protocolo TCP *(Transmission Control Protocol)* e outro modo que é orientado a datagrama e utiliza o protocolo UDP *(User Datagrama Protocol).* Os dois modos de comunicação têm em comum a atuação sobre o protocolo IP (*Internet Protocol*) (DA SILVA; FRANCO; AVELINO, 2006).

O modo orientado a conexão, que utiliza o protocolo TCP, funciona da seguinte maneira: o servidor fica aguardando por conexões em uma determinada porta, e para o cliente conseguir estabelecer uma comunicação com o servidor, este deve saber previamente algumas informações, como: saber qual *host* é o servidor, e qual o número da porta em que ele está atendendo. Após o servidor aceitar a comunicação com o cliente, um socket é criado na porta do servidor e então o canal de comunicação entre o cliente e o servidor está devidamente estabelecido (DA SILVA; FRANCO; AVELINO, 2006).

No modo que não é orientado a conexão, e que utiliza o protocolo UDP, o funcionamento é totalmente diferente do modo orientado a conexão. O protocolo de transporte UDP não oferece garantia de entrega dos dados, portanto as aplicações que utilizam este modo devem de alguma maneira utilizar métodos que ofereçam controle de fluxo, recuperação dos dados perdidos, eliminação de dados duplicados. Por possuir estas características, o protocolo UDP é bem simples e rápido, porém não oferece a mesma confiabilidade que o protocolo TCP. Neste modo não é estabelecida uma conexão entre cliente e servidor, o cliente apenas envia um datagrama (mensagem) a outro *host* da rede, e caso o host receptor não esteja aguardando por uma mensagem, a mesma é perdida.

3.2. Message Passing Interface (MPI)

Devido à falta de padronização entre as bibliotecas de troca de mensagem para

computação paralela e a incompatibilidade entre diferentes plataformas de hardwares e sistemas operacionais, alguns problemas foram aparecendo. Para se padronizar, oferecer uma interoperabilidade aos desenvolvedores e tentar minimizar estes problemas, foi criado uma biblioteca de troca de mensagens para computação paralela, chamada MPI, esta biblioteca oferece um padrão, onde este se limita apenas a nível de biblioteca de troca de mensagens, não criando uma camada particular para tratar esta questão (TOLOUEI, 2010).

Projetado em um fórum aberto constituído por diversas organizações na década de 90, o projeto MPI foi ganhando diferentes implementações com o passar dos anos, algumas proprietárias, outras de código aberto, porém todas seguindo o mesmo padrão de usabilidade e funcionamento (JÚNIOR, 2010).

O MPI é um ambiente de passagem de mensagens comumente utilizado, neste ambiente um processo de uma determinada aplicação utiliza funções para fazer a comunicação com outros processos (SABATINE, 2007). A comunicação acontece basicamente na troca de mensagens (troca de dados e sincronização) por uma rede de alta velocidade que interliga os nós que compõem um cluster. As mensagens trocadas são compostas por informações referentes ao endereço de alocação dos dados, a identificação do processo emissor e do processo receptor, e a identificação dos processos que podem receber a mensagem (TOLOUEI, 2010).

O principal objetivo do MPI é se tornar o padrão de interface mais utilizado, assumindo o lugar de bibliotecas de interfaces específicas, levando um melhor desempenho e um aperfeiçoamento na comunicação dos nós que fazem parte do sistema. Este ambiente de troca de mensagens não possui gerenciamento dinâmico de processos, por isso tendem a ser mais eficientes, pois não haverá um excesso de processos em tempo de execução (JÚNIOR, 2010).

Como mostra a Figura 5, no momento da execução de uma aplicação que utiliza MPI, os processos são divididos em processos menores e então são distribuídos pelos nós do cluster. Cada nó executa o processo que lhe foi enviado e retorna o resultado ao nó mestre. Os processos trabalham com memória fracamente acoplada, ou seja, em um sistema de memória distribuída.

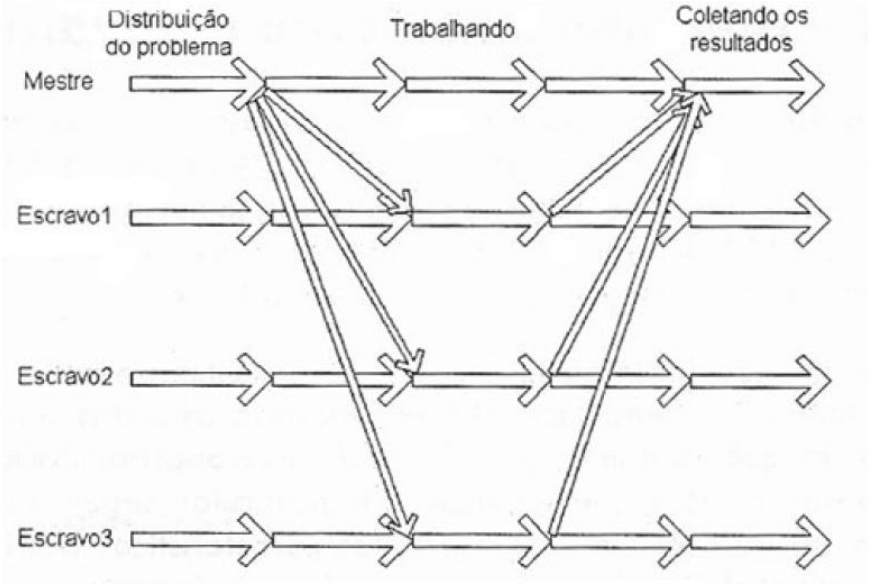

Figura 6 – Modelo de execução do MPI (PITANGA, 2008).

Uma vantagem encontrada no MPI é a sua portabilidade e sua independência de plataforma. Um código MPI desenvolvido para uma arquitetura A com sistema operacional B, pode ser portado para uma arquitetura C com sistema operacional D, com poucas ou nenhuma alteração no código (PITANGA, 2008).

3.3. RPC

Entre os mecanismos de comunicação que podem ser utilizados em sistemas distribuídos, vale ressaltar o RPC (*Remote Procedure Call).*

O RPC foi criado com base em observações no funcionamento do mecanismo de transferência de dados dentro de um programa em um único computador. A partir daí, surgiu a ideia de levar o mesmo conceito desta funcionalidade para um ambiente distribuído, onde seria possível efetuar chamadas a procedimentos remotos, ou seja, localizados em outros computadores. Em geral, o objetivo do RPC é tornar a comunicação distribuída transparente ao desenvolvedor, abstraindo-o de questões de diferenças de hardware, fazendo com que chamadas a procedimentos remotos sejam o mais parecido possível com chamadas a procedimentos locais. A chamada de procedimento remoto é feita de maneira síncrona, onde o processo que efetuou a chamada é suspenso até o procedimento chamado terminar sua execução (NODA, 2005).

Este mecanismo é muito utilizado em muitos sistemas distribuídos, porém não oferece algumas vantagens que o RMI (*Remote Method Invocated*), um mecanismo de

comunicação mais recente consegue oferecer e por isso vem sendo muito utilizado em sistemas distribuídos atuais.

3.4. RMI Java

De acordo com (SARAMAGO, 2012), RMI é essencialmente o RPC, porém, por RMI ser implementado utilizando a linguagem Java algumas vantagens são apontadas para o uso de RMI, como: a segurança, a portabilidade, o *garbage collection* distribuído, a facilidade de desenvolvimento e utilização. Outro fator que diferencia a utilização do RPC para a utilização do RMI é que o RPC não consegue criar instâncias de objetos, e assim não dá suporte à referência remota, portanto em sistemas orientado a objetos o RMI substituiu as chamadas a procedimentos remotos (BARCELLOS, 2002).

O RMI consiste em um mecanismo da linguagem de programação Java que disponibiliza um conjunto de funcionalidades para possibilitar ao programador o desenvolvimento de aplicações Java que utilizem objetos distribuídos, mantendo uma semântica muito semelhante à programação de uma aplicação que envolva apenas objetos locais, essa semântica só não exatamente igual, pois quando se trata de objetos distribuídos, há questões de rede que devem ter tratamentos de erros e execuções particulares (PEREIRA, 2003). A restrição do RMI em apenas poder ser utilizado em aplicações escritas em Java é uma característica importante a ser constada e que pode ser considerada como uma desvantagem.

A camada que está abaixo da aplicação e acima dos protocolos de comunicação, sistema operacional e hardware é a camada onde está posicionado o RMI, como ilustrado na Figura 7, esta camada tem a função de dar transparência nas diferenças das camadas inferiores e possibilitar que um objeto local possa invocar métodos de um objeto remoto que está disponibilizando seus serviços através de uma interface (COLOURIS, 2007).

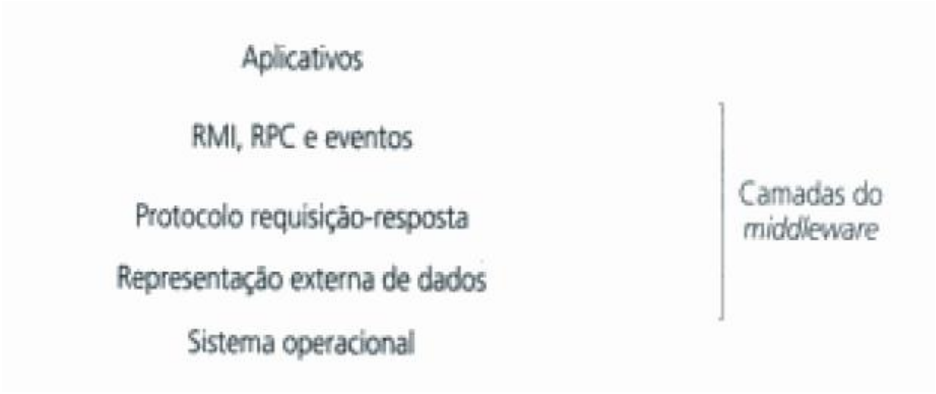

Figura 7 – Camadas do Middleware (COLOURIS, 2007)

Como um sistema distribuído que é implementado utilizando a linguagem Java trabalhará com o conceito de orientação a objetos, uma classe irá conter a implementação de todos os serviços disponibilizados pelo sistema, que ficarão disponíveis para invocações de clientes por meio de objetos distribuídos.

No momento em que o cliente se vincula a um objeto distribuído acontece a criação de duas entidades. No lado do cliente é criado um proxy denominado stub, que fica situado no espaço de endereçamento do cliente. O stub é um representante local que implementa a mesma interface que o objeto que está representando, sua responsabilidade é fazer a serialização dos parâmetros de uma invocação e enviá-la ao servidor. No lado do servidor um objeto denominado skeleton faz a comunicação direta com o cliente por meio do stub, o skeleton tem a responsabilidade de decodificar os parâmetros recebidos na invocação e repassar ao servidor para que o método mais apropriado para aquela invocação seja executado, e mais tarde, serializar a resposta que será encaminhada ao stub para ser repassada á aplicação. Objetos em RMI são passados por referência nas invocações de chamadas e nas mensagens de resposta, enquanto tipos primitivos são passados por valor (PEREIRA, 2003).

A organização e localização dos componentes de um sistema distribuído baseado em objetos e que utiliza invocação de métodos remotos está sendo representada na Figura 8, na imagem podemos visualizar o stub localizado no lado do cliente, o skeleton que está localizado no lado do servidor, e também números que indicam a ordem dos passos do momento em que a aplicação invoca um método, até o momento em que esta mesma aplicação utiliza os resultados recebidos do serviço invocado.

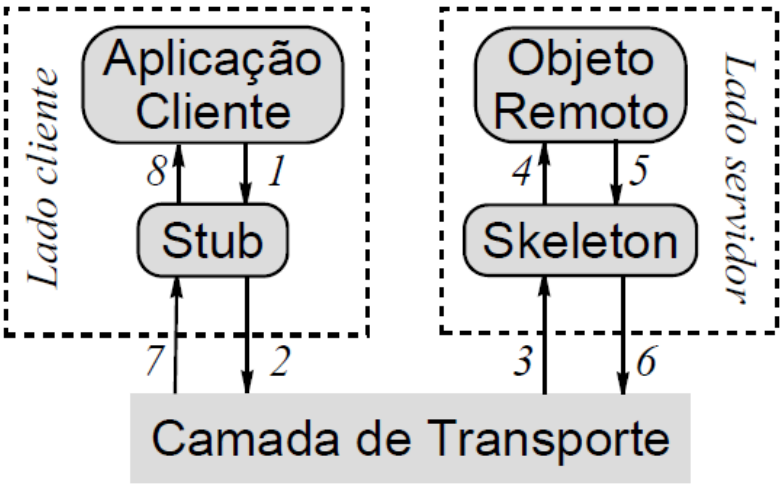

Figura 8 – Organização dos elementos de um sistema distribuído usando Java RMI (PEREIRA, 2003).

Para um melhor entendimento, estão descritos em ordem os passos que acontecem na comunicação, do inicio da invocação de um método por um cliente, até o momento em que a aplicação recebe os valores de retorno. São estes os passos:

- Surge a necessidade da aplicação cliente utilizar um serviço disponibilizado pelo servidor. Esta necessidade é passada ao stub que ficará responsável por toda a comunicação e decodificação dos parâmetros envolvidos.
- O stub inicia uma conexão com o servidor em que o objeto remoto está localizado. As informações que precisam ser enviadas ao servidor são colocadas no parâmetro da invocação e serializadas pelo stub para posterior envio. A invocação está pronta e então é transmitida ao servidor pela rede.
- O servidor cria um elemento denominado skeleton que ficará responsável por receber as invocações que chegam do stub.
- Logo após receber a invocação, o skeleton decodifica os parâmetros e repassa ao servidor para que este execute o método que irá atender a necessidade da aplicação cliente.
- O método é executado e o resultado é passado ao skeleton para que este faça a comunicação corresponde à mensagem de retorno.
- Antes de enviar a resposta ao cliente, o skeleton serializa os parâmetros da resposta e então a transmite pela rede.
- O stub recebe a resposta enviada pelo skeleton.

- Após receber, o stub decodifica os parâmetros da mensagem de reposta e repassa a aplicação cliente para que esta utilize o resultado do serviço utilizado.

Todos os serviços oferecidos ficarão disponíveis em um objeto remoto como dito anteriormente, porém para que o cliente possa efetuar consultas aos serviços oferecidos é preciso que o objeto remoto se registre junto a um servidor de nomes. A utilização de um serviço que atribui nomes a diversos recursos, como computadores, usuários, arquivos, e objetos remotos, por exemplo, é muito útil no projeto de um sistema distribuído. A execução de um serviço oferecido por um objeto remoto, no caso do Java RMI, acontece após o cliente efetuar uma solicitação sobre determinado nome que estará representando o recurso procurado.

O objeto se registra em uma porta especifica do host por meio do método *bind* disponibilizado pela classe *java.rmi.Naming*, passando uma referência do objeto e o nome do mesmo. Este servidor registra os objetos remotos associando-os a um nome, é por meio deste nome que o cliente irá procurar o objeto, através do método *lookup* que também é disponibilizado pela classe *java.rmi.Naming*. Ao encontrar, o servidor de nomes retorna ao cliente a localização do objeto remoto, através de uma referência do objeto remoto (NODA, 2005).

Este serviço de localização de nomes é denominado RMIRegistry e é uma ferramenta que acompanha a máquina virtual Java. Por padrão os objetos remotos são registrados na porta 1099 do host, mas nada impede que esta porta seja alterada ou omitida.

A escolha do RMI como mecanismo de comunicação no desenvolvimento deste projeto se deu pelo fato do RMI ser um mecanismo pertencente à linguagem de programação Java, permitindo a utilização do paradigma de orientação a objetos, sendo que toda implementação dos algoritmos de fatoração, tanto o algoritmo sequencial, como o algoritmo paralelo distribuído, serem feitas utilizando a linguagem. Além de que o RMI é um mecanismo mais recente em comparação a outros mecanismos, e por isso pode apresentar características ainda não apresentadas em outros estudos ou pesquisas.

4. IMPLEMENTAÇÃO DO ALGORITMO

Este capítulo visa apresentar as características do projeto desenvolvido, com questões que envolvem o embasamento da implementação do algoritmo, como: fatoração de números inteiros e criptografia RSA, a implementação do algoritmo em si, o modo como o algoritmo foi paralelizado, e os diferentes modos de execução.

4.1. Fatoração de números inteiros

Tema muito discutido ao longo da história da humanidade, a fatoração de números inteiros é também muito discutida até na atualidade, ainda mais quando é utilizada para implementação de métodos de criptografia.

Grande parte dos métodos modernos de criptografia assume que a fatoração de números inteiros é um problema difícil de ser solucionado. Um método de criptografia que utiliza este conceito é a criptografia RSA, muito utilizada na computação. Neste caso da criptografia RSA, é levado em consideração que é fácil gerar um número inteiro extremamente grande que seja produto de dois números primos inteiros, porém é muito difícil encontrarmos os dois números primos (fatores) que geraram este número inteiro extremamente grande.

Existem algumas maneiras de se fazer a fatoração de números inteiros, como o método de fatoração por tentativas, método de Pollard, métodos de curvas elípticas e o método que iremos abordar na sequencia, o método de Fermat (ANTUNES, 2002).

4.1.1. Método de Fermat

O método de fatoração desenvolvido por Fermat baseia-se que um determinado número *n* composto, pode ter seus fatores encontrados, assumindo que $n = x^2 - y^2 = (x - y)(x + y)$ temos que $(x + y)(x - y)$ são fatores de *n*, onde *n* por suposição é um número ímpar e *x* e *y* são números inteiros positivos tal que $n = (x + y)(x - y) = n$. Este método de fatoração é considerado eficiente, e muito mais eficiente quando o valor de *n* possui um fator primo perto da raiz quadrada de *n*.

O método de Fermat consiste inicialmente em *x* contendo a parte inteira da raiz quadrada de *n*, $x = [\sqrt{n}]$, caso $x^2 = n$, então acontece o caso mais simples onde *n* seria um

quadrado perfeito e *x* seria um dos fatores de *n*. Caso não seja um quadrado perfeito, o *x* é incrementado de um em um até que *y* assuma um número inteiro na equação $\sqrt{x^2 - n}$·. Depois disto, os fatores de n podem ser encontrados facilmente, sendo um dos fatores o *(x + y)* e outro o *(x – y)*.

Deve-se considerar quando *n* assume um número composto e quando *n* assume um número primo. Quando composto a fatoração de *n* mostra que existe um número inteiro $x > [\sqrt{x}]$ onde $\sqrt{x^2 - n}$ é um número menor que $\frac{(n+1)}{2}$. Quando o *n* em questão se apresenta como número primo, o único valor possível de *x* é $\frac{(n+1)}{2}$ (BARNABÉ, 2009) (ANTUNES, 2002).

4.2. Criptografia RSA

Muitas vezes uma troca de informações entre duas pessoas requer sigilo, e quando esta troca de informações acontece utilizando a internet como meio de comunicação, é inevitável que se utilize algum método para que esta informação não seja interpretada por entidades ou pessoas que não sejam aquelas que realmente devem receber a mensagem.

O método utilizado para codificação da mensagem é a criptografia, que vem sendo utilizado há muito tempo, como o caso da cifra de César, método clássico de criptografia onde as letras do alfabeto eram substituídas por letras de três posições à frente, e no caso das guerras, onde as mensagens eram cifradas para que estratégias de guerras não fossem descobertas pelos inimigos.

Em um período mais recente têm-se dois tipos de criptografia, a criptografia simétrica e a criptografia assimétrica. A criptografia simétrica utiliza uma mesma chave para se criptografar e para descriptografar uma mensagem. Este método é relativamente rápido, porém inseguro pelo fato de ser necessário o compartilhamento da chave, sendo que não há um meio de comunicação seguro para a distribuição da mesma. O algoritmo mais utilizado para este tipo de criptografia hoje em dia é o AES *(Advanced Encryption Standard)* (DETOMINI, 2010).

Na criptografia assimétrica são utilizadas duas chaves, uma para criptografar a mensagem e outra para descriptografar. A chave utilizada para criptografar a mensagem e que pode ser de conhecimento de qualquer pessoa é denominada chave pública, enquanto a chave que é utilizada para descriptografar a mensagem, e que é de conhecimento exclusivo do destinatário é denominada chave privada, cada chave privada deve ser referente a uma única

chave pública, pois apenas ela conseguirá decifrar a mensagem. Um exemplo é quando uma pessoa A deseja enviar uma mensagem para pessoa B, a pessoa A deve utilizar a chave pública de B para criptografar a mensagem, e depois de enviada, a pessoa B usa sua chave privada para decifrar a mensagem. O algoritmo mais utilizado para este tipo de criptografia é o RSA *(Rivest Shamir Adleman)*, que é relativamente mais lento que o AES, porém oferece maior segurança.

O método utilizado pelo RSA para criptografar mensagens é a base para o desenvolvimento deste projeto. Este método utiliza a multiplicação de grandes números primos como parte do seu processo para criptografar uma mensagem, onde esta multiplicação irá gerar um número inteiro extremamente grande, e para que se consiga quebrar a criptografia RSA basta encontrar os dois fatores primos a partir do número inteiro gerado na multiplicação dos mesmos. A maior segurança que oferece este método de criptografia em relação a outros, esta relacionada a dificuldade, em termos de tempo e tecnologia, para se fatorar um número extremamente grande (DETOMINI, 2010).

Para a solução deste problema, existem alguns métodos de fatoração de números inteiros, o método de Fermat assume características que de certa forma facilitam sua implementação e a divisão do problema em partes que podem ser paralelizadas, além de sua ideia estar baseada em um poderoso método de fatoração de números inteiros, o método do Crivo quadrático. Por estes motivos o método de Fermat foi selecionado para ser utilizado na implementação deste projeto.

4.3. Desenvolvimento do Algoritmo

O algoritmo foi desenvolvido inicialmente para uma execução de maneira sequencial, e depois foi adaptado para que fosse possível uma execução paralela.

Para a execução de forma paralela o algoritmo utilizado sofreu algumas alterações para que fosse possível paralelizar o processamento, distribuindo as tarefas entre os nós que compunham o cluster, porém as mudanças ficam apenas nessa questão, pois os cálculos efetuados para chegar ao resultado necessário são exatamente iguais para o algoritmo sequencial como para o algoritmo paralelo.

A seguir será melhor explicado como foi desenvolvido, e o funcionamento do algoritmo sequencial e do algoritmo paralelo distribuído.

4.3.1. Algoritmo Sequencial

O algoritmo sequencial foi desenvolvido na linguagem de programação Java, baseado na implementação de todo o método de Fermat, onde todos os cálculos e iterações necessárias para se chegar aos fatores ficou todo sob responsabilidade de apenas uma maquina, sendo desnecessária a implementação de mais componentes. Abaixo, na Figura 9, está sendo mostrado o trecho do código que realiza os cálculos necessários para implementar o método de Fermat.

```
while(yInteiro == 0)
{
      y = Math.sqrt((x*x) - n);

      resto = y % 1;
      if(resto > 0)
      {
          x++;
      }
      else
      {
         System.out.println("n é inteiro");
         System.out.println("Primeiro Fator: "+(x+y));
         System.out.println("Segundo Fator: "+(x-y));
         yInteiro = 1;
      }
}
```

Figura 9 – Trecho do código sequencial onde são realizados os cálculos para se chegar as fatores.

Neste trecho é possível visualizar o laço de repetição onde é realizado o incremento da variável *x* a cada iteração que *y* não assume um valor inteiro, até o momento onde *y* assume um valor inteiro e se torna possível chegar aos dois fatores.

4.3.2. Algoritmo Paralelo Distribuído

O desenvolvimento do algoritmo para execução paralela também foi caracterizado pela utilização da linguagem de programação Java, e um de seus mecanismos, o Java RMI, que permite a utilização da programação distribuída por meio da aplicação.

Baseando-se no paradigma mestre-escravo, foram implementados um cliente, que possui a função do mestre, de dividir os dados e distribuir as partes para os servidores, que possuem o papel dos escravos dentro do paradigma, estes executam o mesmo código fonte (mesmas instruções) sobre os diferentes dados recebidos do mestre, desta forma, a aplicação se encaixa na classe SIMD (*Single Instruction Multiple Data*) das arquiteturas paralelas de acordo com a taxonomia de Tanenbaum.

A Figura 10 representa de maneira bem clara como é organizado o paradigma mestre-escravo de acordo com a utilização do Java RMI, com os Servidores tendo a mesma função de um escravo, e o Cliente, a de um mestre.

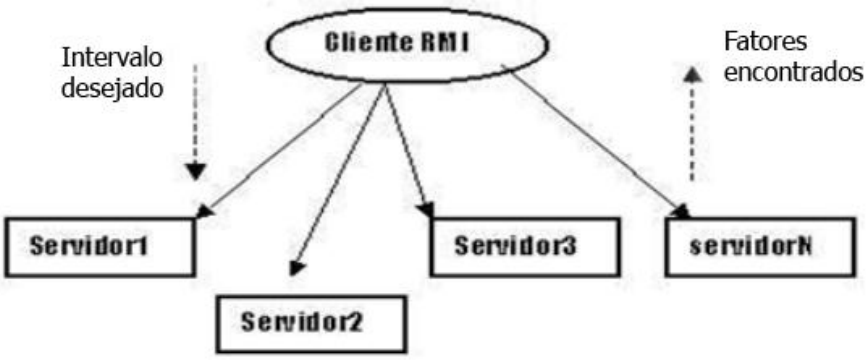

Figura 10 – Organização dos componentes da aplicação e seus respectivos parâmetros.

Para que o cliente RMI disponibilize os intervalos para os servidores, é necessário que os servidores implementem um código que receba estes intervalos, para isso é preciso encontrar no algoritmo sequencial um trecho de código que permita uma divisão do problema, e posteriormente uma paralelização do código.

Esta divisão do problema está representada na Figura 11, sendo que o retângulo azul presente na imagem representa o algoritmo sequencial.

Primeiramente é feita uma análise para identificar um trecho do código sequencial

que apresente características paralelizáveis. Após este trecho ser identificado (representado pelo retângulo vermelho na Figura 11), é feita uma divisão de maneira que as partes geradas possam ser distribuídas pelos servidores, estas partes podem ser caracterizadas por trechos de códigos idênticos ou não. Para este projeto, as partes distribuídas ente os servidores são idênticas, e executam as mesmas instruções, recebendo valores distintos para executarem, onde estes valores são representados por intervalos distintos de números inteiros que cada servidor terá que executar. Na Figura 11, estes intervalos estão sendo representados pelas cores verde, roxo e amarelo.

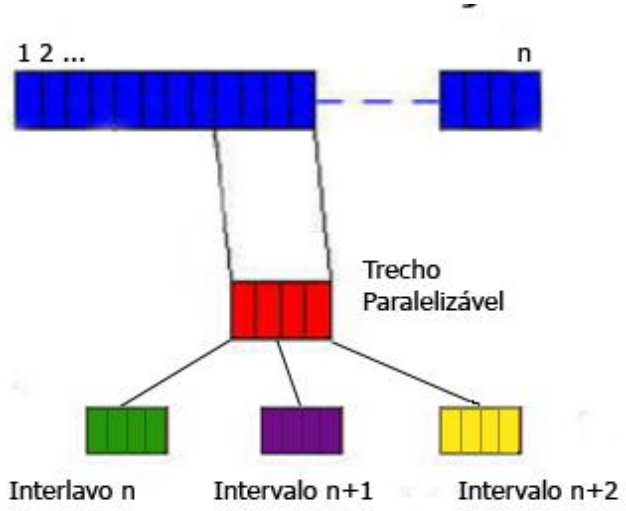

Figura 11 – Método de divisão do algoritmo sequencial.

A maneira como funciona a aplicação implementada para este projeto ocorre com o cliente efetuando invocações de maneira paralela aos métodos remotos disponibilizados pelos servidores, que executam as partes que possuem do algoritmo. Após as execuções terem ocorrido, o servidor que obteve o resultado transmite a resposta ao cliente para que este possa mostrar ao usuário o resultado obtido.

Com a utilização do RMI faz-se necessário que sejam implementadas interfaces e outros componentes para que seja possível sua utilização, portanto, o lado do cliente, que será denominado mestre, será composto pela classe cliente e por outros componentes, e o lado dos servidores, que serão denominados escravos, será composto para cada servidor, pela classe servidor e por outros componentes necessários.

A partir destas características o desenvolvimento do algoritmo paralelo foi dividido em duas etapas principais. A primeira etapa visa à implementação dos componentes

que fazem parte do mestre (cliente). Em sequencia, a segunda etapa visa à implementação dos componentes que pertencem ao lado do escravo (servidor).

Inicialmente, será descrito todos os componentes que pertencem ao lado do mestre, e depois, todos os componentes que pertencem ao lado dos escravos.

4.3.2.1. Implementação do Mestre

A responsabilidade pela divisão e distribuição do problema entre os nós escravos, e pela invocação dos métodos remotos é toda do nó mestre. Ele quem vai gerenciar toda a aplicação, distribuindo os intervalos de valores a serem executados pelos escravos e recebendo como retorno os fatores encontrados por algum dos nós escravos.

Para que o problema da comunicação síncrona imposto pela utilização do Java RMI fosse resolvido e a invocação dos métodos remotos ocorresse de maneira paralela, foi necessária a utilização de threads, onde cada thread criada se torna responsável pela invocação a um método remoto relacionado a um único escravo.

Na implementação do código executado pelo nó mestre foram necessárias as seguintes classes: Cliente.java, ThreadEscravo.java, InterfaceEscravo.java.

4.3.2.2. Classe Cliente.java

No inicio do desenvolvimento desta classe, são declaradas e inicializadas no método principal, algumas das variáveis necessárias. Dentre as responsabilidades das mesmas estão: receber tempo inicial do inicio da execução, controle de laço de repetição, quantidade de escravos utilizados, o número que será fatorado, e um vetor para armazenar valores inteiros, que agrupados em pares vão representar os limites iniciais e finais dos intervalos a serem executados pelos nós escravos. O agrupamento dos pares que formam o intervalo é feito de maneira lógica, sendo que ele só pode ser observado no momento em que o par de posições do vetor é passado por referência ao servidor que irá executá-lo. O modo como os pares de posições do vetor são agrupados para formar um intervalo está representado pela Figura 12.

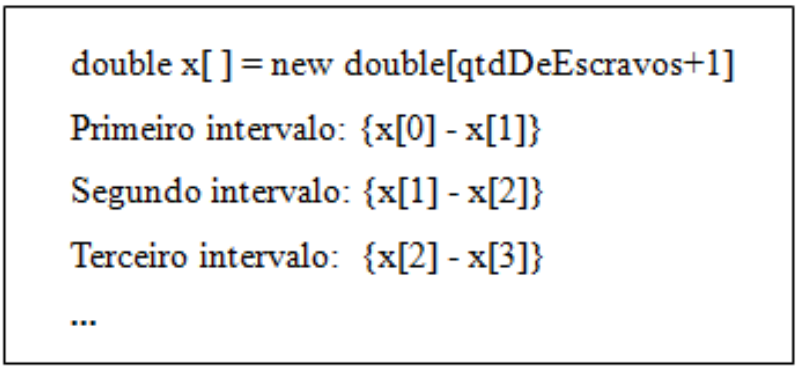

Figura 12 – Divisão dos intervalos para os escravos.

Após as declarações das variáveis utilizadas e a explicação de como o vetor é utilizado para armazenar os intervalos, o método de Fermat começa a ser implementado no lado do cliente. O primeiro passo é calcular a raiz quadrada do valor a ser fatorado. Este valor a ser fatorado será identificado como *n* para facilitar a explicação.

O valor encontrado ao se calcular a raiz quadrada de *n* é incrementado em uma unidade e corresponde ao valor inicial a ser incrementado nas iterações seguintes, este valor é guardado na primeira posição do vetor responsável por formar os intervalos de valores, e será denominado *xInicial*.

A princípio será criado um intervalo principal, que depois será dividido em intervalos menores de acordo com o número de escravos utilizados. Para criar este intervalo principal falta gerar um valor final que represente o limite máximo do intervalo, já que o limite mínimo do intervalo é o *xInicial* descrito acima.

Através de testes preliminares com a fatoração de diversos números inteiros, não foi possível encontrar nenhum padrão ou lógica no número de iterações que acontece até que os fatores sejam encontrados, para que fosse possível definir o final do intervalo de maneira que os intervalos gerados após a divisão para os escravos não ficassem com um tamanho muito extenso e prejudicassem o desempenho.

Com isso, o valor de *xFinal* será ajustado de acordo com as necessidades, para que se consiga mostrar cenários de execuções particulares, portanto o ambiente de testes se torna totalmente controlado. O ajuste será feito controlando o tamanho do intervalo principal que irá afetar também, o intervalo que cada escravo irá receber para executar, este procedimento vai representar o mesmo que se fossem aumentados o número de escravos.

O modo encontrado para se gerar um *xFinal*, foi através da implementação de um método que identifica a quantidade de dígitos que compõe o valor de *n*, e assim criar o maior valor que aquela quantidade de dígitos pode formar, sendo este o valor de *xFinal.*

Esta forma de gerar o *xFinal* também se baseou nos testes preliminares, onde foi possível identificar que o valor de *xInicial*, ao ser incrementado, não irá ultrapassar de maneira alguma o maior valor que se pode formar com a quantidade de dígitos do número *n*. Caso o número *n* tenha 7 dígitos por exemplo, o *xFinal* será correspondido pelo número 9999999, que estará armazenado na última posição do vetor.

Com o valor do *xFinal* definido, já é possível dividir o intervalo entre o *xInicial* e o *xFinal* em quantos outros intervalos for necessário, de acordo com o número de escravos utilizados na execução. Os valores que correspondem aos limites dos intervalos são armazenados dentro do vetor, e após todos os valores serem definidos e armazenados, a responsabilidade da classe Cliente passa a ser a de criar a quantidade de threads necessárias.

A definição da quantidade máxima de threads que podem ser criadas pela aplicação é feita de maneira estática, pois como os testes serão realizados em um número fixo de cenários, e a quantidade de threads será igual à quantidade de escravos utilizados na execução, não há necessidade de uma criação dinâmica de threads.

Antes de uma execução, define-se de maneira manual apenas a quantidade de escravos utilizados, deixando que a aplicação inicie o número correto de threads para o respectivo cenário. As seguintes situações podem ocorrer: a criação de apenas uma thread quando apenas um escravo for utilizado, duas threads para quando dois escravos forem utilizados, três threads para quando três escravos forem utilizados, quatro threads para quando quatro escravos forem utilizados, e cinco threads para quando cinco escravos forem utilizados.

Após estarem criadas, as threads recebem como parâmetro na sua chamada, o valor de *n*, duas posições do vetor que correspondem a um intervalo de valores, e o tempo em que se iniciou a execução da aplicação, como demonstrado na Figura 13.

```
switch(qtdEscravos)
{
case 1:
      new Thread(new ThreadServer01(n,x[i],x[i+1],tempoInicial)).start();
      break;
case 2:
      new Thread(new ThreadServer01(n,x[i],x[i+1],tempoInicial)).start();
      new Thread(new ThreadServer02(n,x[i+1],x[i+2],tempoInicial)).start();
      break;
case 3:
      new Thread(new ThreadServer01(n,x[i],x[i+1],tempoInicial)).start();
      new Thread(new ThreadServer02(n,x[i+1],x[i+2],tempoInicial)).start();
      new Thread(new ThreadServer03(n,x[i+2],x[i+3],tempoInicial)).start();
      break;
case 4:
      new Thread(new ThreadServer01(n,x[i],x[i+1],tempoInicial)).start();
      new Thread(new ThreadServer02(n,x[i+1],x[i+2],tempoInicial)).start();
      new Thread(new ThreadServer03(n,x[i+2],x[i+3],tempoInicial)).start();
      new Thread(new ThreadServer04(n,x[i+3],x[i+4],tempoInicial)).start();
      break;
case 5:
      new Thread(new ThreadServer01(n,x[i],x[i+1],tempoInicial)).start();
      new Thread(new ThreadServer02(n,x[i+1],x[i+2],tempoInicial)).start();
      new Thread(new ThreadServer03(n,x[i+2],x[i+3],tempoInicial)).start();
      new Thread(new ThreadServer04(n,x[i+3],x[i+4],tempoInicial)).start();
      new Thread(new ThreadServer05(n,x[i+4],x[i+5],tempoInicial)).start();
      break;
}
```

Figura 13 – Trecho do código que faz a chamada das threads.

A classe Cliente.java finaliza sua execução neste momento transferindo para as threads a responsabilidade de invocar os métodos remotos.

4.3.2.3. Implementação das Threads

Ao serem criadas e chamadas pela classe Cliente.java as threads têm o objetivo de invocar o método remoto e mostrar o resultado recebido como retorno, as threads implementadas utilizam-se do método *lookup* disponibilizado pela classe *java.rmi.Naming* para procurar pelo método que foi registrado anteriormente por um objeto no servidor de

nomes.

No construtor da thread, são recebidos da classe Cliente.java, o número a ser fatorado, um intervalo de números inteiros, e o tempo inicial da execução. O código do construtor da thread é representado pela Figura 14.

```
public ThreadServer(double n, double x1, double x2, Long tempoInicial)
{
	this.n = n;
	this.x1 = x1;
	this.x2 = x2;
	this.tempoInicial = tempoInicial;
}
```

Figura 14 – Construtor da thread.

O trecho do código onde é feita a invocação do método remoto e exibição do retorno recebido estão contidos no método *run* que pertence a interface Runnable implementada pelas threads. Este código é mostrado na Figura 15.

```
InterfaceServidor server =
(InterfaceServidor) Naming.lookup("rmi://"+ip+"/Servidor");

double fatoresServidor[] = server.parte(n, x1, x2);
if((fatoresServidor[0] > 0.0) && (fatoresServidor[1] > 0.0))
{
     System.out.println("Fator 1: "+fatoresServidor[0]);
     System.out.println("Fator 2: "+fatoresServidor[1]);
}
else
     System.out.println("Fatores não encontrados!");
```

Figura 15 – Trecho do código onde é realizada a invocação ao método remoto.

4.3.2.4. Implementação do Escravo

Dependendo do cenário, uma variação no número de escravos vai ocorrer, porém,

independentemente da quantidade de escravos, a instrução que cada um deles executa é a mesma. O que modifica de um escravo para o outro são os dados que vão atuar sobre as instruções. Esses dados na verdade são intervalos de números inteiros que são recebidos por cada escravo na chamada de seu método remoto. A partir daí, os escravos executam seu código, que é caracterizado pela implementação do método de Fermat, incrementando o primeiro número do intervalo a cada iteração, até que se encontre o valor esperado, ou até que todos os números inteiros do intervalo tenham sido executados.

Ao final, apenas um dos escravos irá encontrar os dois fatores que serão retornados para thread cliente. E quando um dos escravos encontrar os dois fatores, a execução dos outros escravos é ignorada, pois o que interessa é efetuar uma verificação no período em que se inicia a execução até o momento em que são encontrados os dois fatores.

O lado do escravo é composto por uma interface, InterfaceServidor.java, e duas classes: Servidor.java, e a InterfaceServidorImplementada.java.

4.3.2.5. Classe Servidor.java

A tarefa de registrar o objeto remoto no servidor de nomes e manter o escravo escutando invocações fica por conta da classe Servidor.java. O registro acontece através da chamada ao método *rebind*, que tem a função de fazer a ligação entre o nome especificado e o objeto remoto, substituindo qualquer ligação feita anteriormente. O trecho do código que implementa esta funcionalidade segue abaixo representado pela Figura 16.

```
InterfaceServidorImplementada m = new InterfaceServidorImplementada();

Naming.rebind("rmi://"+ip+":1099/Servidor", m);

System.out.println("Servidor executando ...");
```

Figura 16 – Trecho do código onde o objeto remoto é registrado no RMIRegistry.

4.3.2.6. Classe InterfaceServidor.java

As interfaces são implementadas contendo apenas as assinaturas dos métodos, deixando para que uma classe implemente todos métodos da interface. Neste caso, a interface assina apenas um método, denominado parte, referente à parte que um específico servidor vai

executar. Abaixo segue a Figura 17 que representa o código da interface.

```
public interface InterfaceServidor extends Remote {
          public double[] parte(double n, double x1, double x2)
throws RemoteException;
}
```

Figura 17 – Interface com a assinatura de seu método.

Caso haja uma mudança no cenário de execução, e sejam adicionados mais escravos, outras interfaces serão criadas para cada escravo adicionado.

4.3.2.7. Classe InterfaceServidorImplementada.java

A classe InterfaceServidorImplementada.java implementa a interface InterfaceServidor.java, com isto a classe consiste da implementação dos métodos contidos na interface, no caso desta aplicação, apenas um método será implementado.

A implementação do método é definida pelo método de Fermat, onde o escravo tenta resolver uma fatoração de um número inteiro com base no intervalo de valores recebido por uma invocação do cliente. O trecho do método onde é feito o cálculo para encontrar os fatores é mostrado na Figura 18.

```
while((yInteiro == 0) && (x1 <= x) && (x <= x2))
{
      y = Math.sqrt((x*x) - n);

      resto = y % 1;
      if(resto > 0)
      {
          x++;
      }
      else
      {
          System.out.println("n é inteiro");
          System.out.println("Primeiro Fator: "+(x+y));
          System.out.println("Segundo Fator:  "+(x-y));
          yInteiro = 1;
      }
```

Figura 18– Trecho do código onde são realizados os cálculos para se chegar aos fatores.

O código apresentado é idêntico ao código do algoritmo sequencial, a diferença é que na execução paralela, este código estará contido nas implementações das diversas interfaces distribuídas entre os escravos utilizados na execução. O comportamento do código apresentado na Figura 18 acontece de maneira que a execução só será interrompida até que uma destas condições seja satisfeita: o valor de *y* assuma um valor inteiro, ou o valor de *x*, ao ser incrementado, chegue ao final do intervalo passado pelo cliente.

O primeiro passo é multiplicar o primeiro valor do intervalo recebido por ele mesmo, subtrair pelo número a ser fatorado, representado pelo *n*, e encontrar a raiz quadrada deste número que é então é armazenada em *y*.

O segundo passo é verificar se *y* é um número inteiro, e para se verificar esta condição, divide-se *y* por 1, e com o resto da divisão é possível determinar se *y* é um número inteiro, pois caso o resultado seja 0, *y* é um valor inteiro, caso o resultado seja um número diferente de zero, o valor de *x* não é inteiro e então é incrementado e o primeiro passo é novamente realizado, até que o valor de *y* seja um número inteiro.

Após a variável *y* assumir um valor inteiro, os dois fatores já podem ser

encontrados. O primeiro fator é encontrado através da soma do valor de *x* com o valor de *y*, e o outro é encontrado através da subtração do valor de *x* pelo valor de *y*. Estes então são transmitidos como retorno para thread que efetuou a invocação a este escravo e então exibidos pela aplicação.

4.4. Execução da aplicação

A aplicação foi executada de duas maneiras distintas, execução sequencial e execução paralela, de acordo com os códigos implementados anteriormente. A seguir será mostrado como foi executado o algoritmo para dois casos, e em quais cenários as execuções ocorreram.

4.4.1. Execução Sequencial

A execução sequencial foi realizada em uma mesma máquina, com seis cenários que se distinguiam apenas pelo número inteiro utilizado para fatoração. Os números utilizados foram os mesmos dos testes realizados de maneira paralela distribuída, e serão apresentados no item 5.2.

Em cada um dos cenários foram realizadas 10 execuções de onde se calculou a média, que foi o resultado levado em conta como medida de desempenho, de onde foi possível calcular outras medidas para análise. Os resultados da execução sequencial serão apresentados no capítulo 5, em uma tabela respectiva ao cenário de execução, juntamente com os resultados da execução paralela, para facilitar as análises e uma comparação entre ambas.

4.4.2. Execução Paralela Distribuída

Após a execução e coleta dos resultados provenientes da execução sequencial, chega à vez de se testar a aplicação em um ambiente paralelo distribuído, e para isso é necessário que sejam feitas alterações no código e implementação de novos componentes que vão executar em outros nós do sistema, como exemplificado no item 4.3.2.

Após toda implementação dos componentes necessários para compor a aplicação, é hora de executar o código em diferentes tipos de cenários.

Ao todo foram organizados seis cenários, diferenciados apenas pelo número

inteiro a ser fatorado. A execução em cada um dos cenários leva em consideração outros seis modos de execução, são eles: execução do algoritmo sequencial; execução do algoritmo utilizando uma máquina mestre e uma máquina escravo; execução utilizando uma máquina mestre e duas máquinas escravos; execução utilizando uma máquina mestre e três máquinas escravos, execução utilizando uma máquina mestre e quatro máquinas escravos, e uma execução utilizando uma máquina mestre e cinco escravos. Cada cenário realizou os seis modos de execução, permitindo que fossem realizadas comparações entre eles.

Todos os cenários foram executados em um ambiente distribuído paralelo controlado, onde o número inteiro a ser fatorado será escolhido e inserido manualmente no algoritmo antes da execução de cada cenário. A escolha do número inteiro será feita utilizando critérios para que os testes realizados possam mostrar casos onde ocorreu perda de desempenho e onde ocorreu ganho de desempenho, dando a possibilidade de se explicar cada caso, e também para que o tempo de uma única execução seja viável para realização de todos os testes, pois serão realizadas 10 execuções para o processamento sequencial e 10 execuções para cada quantidade de escravos utilizados na execução paralela. Alguns dos critérios utilizados que interferem diretamente no desempenho são: o tamanho e a distância dos fatores, que interferem diretamente no número de iterações que irá ocorrer na execução.

Sabendo disso, foi utilizado um mesmo modelo para a construção de todos os cenários, onde este modelo está sendo representado pela Tabela 1. Nesta figura, a letra *X* que vem após a palavra "cenário", representa o número do cenário, que pode ser 01, 02, 03, 04, 05, 06, e a letra *Y* representa o número inteiro que será fatorado no cenário em questão.

Tabela 1 – Modelo de construção dos cenários

Cenário *X*: Número *Y*	**Total de execuções**
Execução Sequencial	10
Execução com 1 escravo	10
Execução com 2 escravos	10
Execução com 3 escravos	10
Execução com 4 escravos	10
Execução com 5 escravos	10

A escolha dos dois números primos foi feita de maneira aleatória, observando e garantindo que a multiplicação dos dois não gere um número que tenha acima de treze dígitos, pois um número maior que esta quantidade de dígitos tomaria um tempo ainda maior para que

todas as execuções dentro de todos os cenários fossem concluídas, sabendo que o tempo e a infra-estrutura necessária muitas vezes é escasso. Porém, com um número de treze dígitos para ser fatorado, já é possível efetuar as análises e conclusões necessárias.

5. TESTES E RESULTADOS

Os testes foram todos realizados no laboratório 01 do Centro Universitário Eurípides Soares da Rocha de Marília – UNIVEM. As máquinas utilizadas possuíam configuração homogênea, onde executavam o sistema operacional Windows 7 Professional 32 bits, com processador Intel Core 2 Duo - E7500 de 2.94GHz e memória de 4Gbytes. As mesmas estavam conectadas em uma rede padrão ethernet 100Mbits, interligadas através da utilização de um switch.

A presença da plataforma Java nas máquinas possibilitou a execução dos testes que faz uso do mecanismo de comunicação Java RMI para a comunicação distribuída.

O desempenho foi analisado em cima de testes em diferentes tipos de cenários em um ambiente paralelo distribuído controlado. Foram necessários seis cenários para compor todos os testes, sendo que a composição de cada um deles será melhor detalhada nos capitulo 5.3.

Ao término da execução dos testes, foi calculada a média dos tempos de execução de cada um dos cenários, o que possibilitou efetuar uma análise estatística dos resultados coletados. Algumas das medidas calculadas foram: desvio padrão, *speedup* e eficiência.

5.1. Speedup e Eficiência

Speedup é uma métrica utilizada para verificar a razão entre o tempo da execução sequencial (em um único processador) pelo tempo da execução em paralelo (em *p* procesadores), determinando o desempenho da aplicação. A equação 1 representa o cálculo para obtenção do *speedup*.

$$S = \frac{Ts}{Tp} \qquad (1)$$

Onde:

S = *speedup* obtido através da razão de *Ts* e *Tp*.

Ts = tempo de execução da aplicação sequencial.

Tp = tempo de execução da aplicação em paralelo, com p processadores.

Eficiência também é uma métrica utilizada para verificar o tempo em que os processadores estão ativos, e o quanto da potência computacional disponível foi utilizado na execução. É calculada através da razão entre o s*peedup* e o número de processadores utilizados (SABATINE, 2007). A equação 2 representa este cálculo.

$$\mathrm{E} = \frac{Sp}{p} \tag{2}$$

Onde:

E = eficiência obtida através da razão entre *Sp* e *p*.

Sp = *speedup* obtido através da razão de *Ts* e *Tp*.

p = número de processadores utilizados.

Estas duas métricas serão aplicadas utilizando os tempos médios obtidos através dos testes realizados nos diferentes cenários de execução e estarão disponíveis nas tabelas com outras métricas do respectivo cenário.

5.2. Cenários de Teste

Os resultados obtidos através dos testes foram coletados conforme o tempo de execução dos mesmos, e convertidos para uma única medida de tempo, os segundos (s). Todos os testes realizados serão descritos por meio de uma breve explicação de como foi realizado e quais dados foram utilizados em cada um dos cenários, além da construção de uma tabela, onde foi possível apresentar algumas medidas para análise, são elas: média, desvio padrão, *speedup*, e eficiência.

As tabelas proporcionaram a construção de gráficos que facilitam representar uma comparação entre o tempo médio da execução sequencial com os tempos de cada uma das execuções paralelas.

Ao serem mostrados cenários com características semelhantes em relação ao desempenho obtido, uma conclusão parcial foi realizada buscando verificar quais fatores influenciaram no desempenho similar entre eles.

5.2.1. Cenário 01

Neste primeiro cenário, o número a ser fatorado é o 2.100.044.300.161, onde seus fatores são os números primos, 1.500.007 e 1.400.023. A utilização deste número se deu por causa do tempo de execução viável que este apresenta, pelos fatores estarem a uma distância relativamente pequena um do outro, e por apresentar resultados interessantes de serem analisados. Ao ser testado de maneira sequencial e paralela, em um ambiente distribuído, com os testes da execução paralela sendo realizados com um, dois, três, quatro e cinco escravos, foram obtidos os resultados que estão sendo apresentados pela Tabela 2.

Tabela 2 – Resultados obtidos com testes realizados no cenário 01

CENÁRIO 01						
	Sequencial	1 escravo	2 escravos	3 escravos	4 escravos	5 escravos
Média	0,414	0,507	0,509	0,51	0,513	0,515
Desvio Padrão	0,0105	0,0052	0,0149	0,0119	0,0113	0,0086
Speedup		0,8165	0,8133	0,8117	0,807	0,8038
Eficiência		0,8165	0,2711	0,2705	0,2017	0,1608

Após observação dos resultados obtidos, foi possível observar que houve perda de desempenho em relação às execução em paralelo com a execução sequencial. E conforme o número de escravos utilizados nos testes foi aumentando, o tempo de execução também aumentou. Este fato fica bem caracterizado ao ser analisada a eficiência, que diminui à medida que novos escravos vão sendo adicionados. No Gráfico 1, mostrado abaixo, fica bem visível a crescente no tempo médio de execução conforme o número de escravos aumenta.

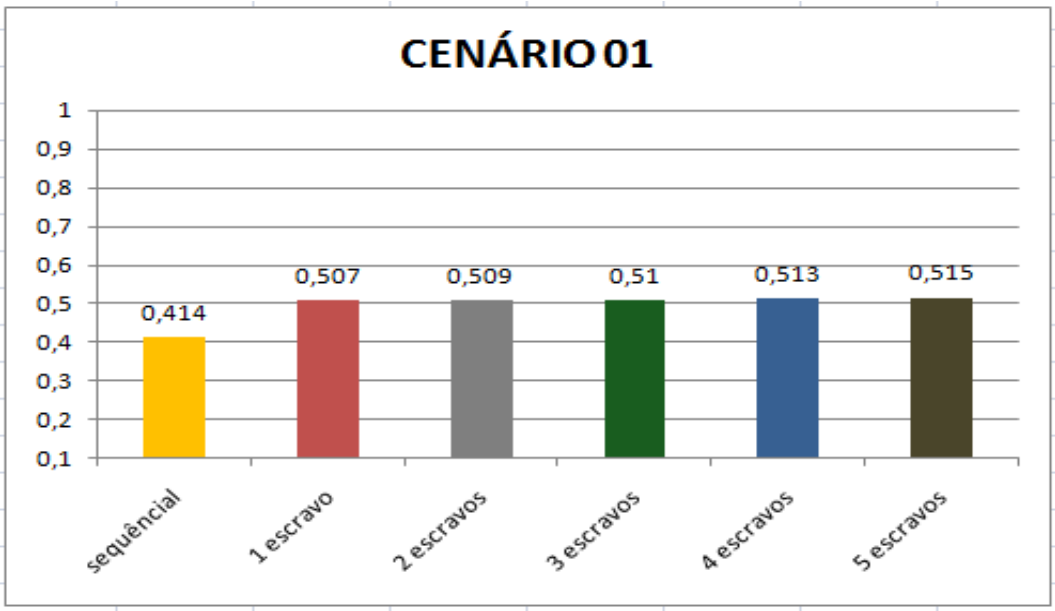

Gráfico 1 – Distribuição das médias dos tempos de execução no cenário 01.

5.2.2. Cenário 02

Para este segundo cenário, o número escolhido para ser fatorado foi o 1.125.033.750.133, onde seus fatores são os números primos, 1.500.007 e 750.019. Este número foi escolhido, pois possui um tempo de execução viável para ser testado nos diversos cenários, pelos fatores estarem em uma distancia média um do outro, diferentemente da distância pequena entre os fatores do primeiro cenário, e por apresentar resultados interessantes para serem analisados. Ao ser testado de maneira sequencial e paralela, em um ambiente distribuído, com os testes da execução paralela sendo realizados com um, dois, três, quatro e cinco escravos, foram obtidos os resultados que estão sendo apresentados pela Tabela 3.

Tabela 3 –Resultados obtidos com testes realizados no cenário 02

CENÁRIO 02						
	Sequencial	1 escravo	2 escravos	3 escravos	4 escravos	5 escravos
Média	19,6	19,82	19,85	19,87	19,95	19,96
Desvio Padrão	0,0232	0,0748	0,0279	0,032	0,0254	0,0336
Speedup		0,9889	0,9874	0,9864	0,9824	0,9819
Eficiência		0,9889	0,4937	0,3288	0,2456	0,1964

Ao analisar os resultados obtidos neste cenário 02 foi possível observar que

aconteceu uma situação semelhante a do cenário 01, onde houve perda de desempenho em relação às execuções em paralelo com a execução sequencial, e a medida que o número de escravos aumentou, o tempo médio de execução aumentou, e a eficiência diminuiu. Neste cenário 02, o tempo médio de todas as execuções foi superior ao tempo de execução obtido em todas as execuções do cenário 01.

No Gráfico 2 mostrado abaixo também é possível verificar a piora no desempenho através da crescente que ocorre nas colunas do gráfico ao se aumentar o número de escravos. O tempo médio das execuções em comparação com o tempo médio do cenário 1 também aumentou.

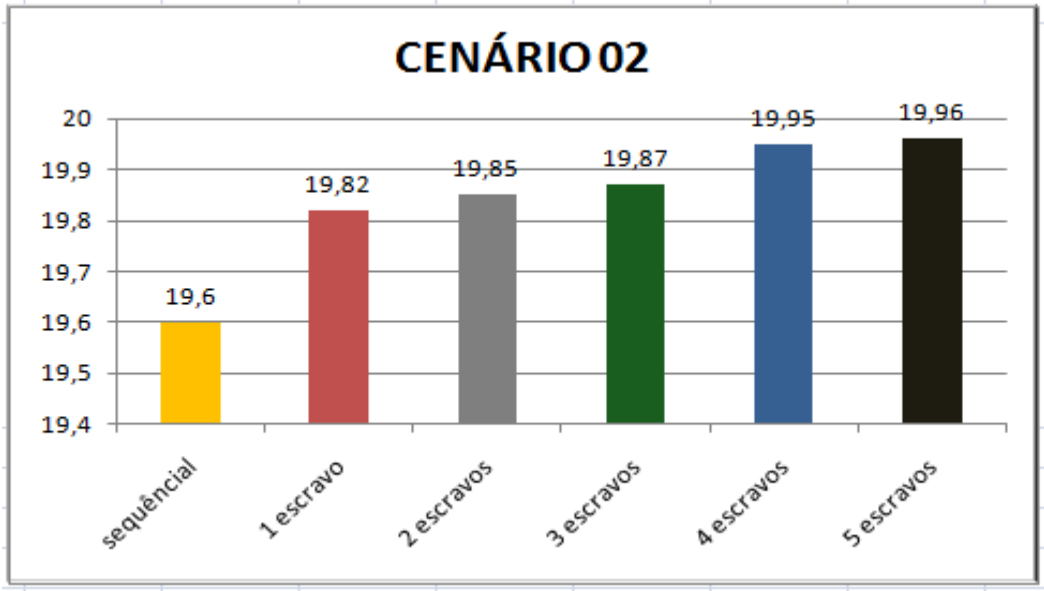

Gráfico 2 – Distribuição das médias dos tempos de execução no cenário 02.

5.2.3. Cenário 03

Neste terceiro cenário, o número escolhido para ser fatorado foi o 150.005.200.021, pois possui fatores mais distantes um do outro em relação aos cenários anteriores, onde seus fatores são os números primos, 1.500.007 e 100.003, além de apresentar um tempo viável de execução e resultados interessantes de serem analisados. Ao ser testado de maneira sequencial e paralela, em um ambiente distribuído, com os testes da execução paralela sendo realizados com um, dois, três, quatro e cinco escravos, foram obtidos os resultados que estão sendo apresentados pela Tabela 4.

Tabela 4 – Resultados obtidos com testes realizados no cenário 03

CENÁRIO 03						
	Sequencial	1 escravo	2 escravos	3 escravos	4 escravos	5 escravos
Média	124,39	125,4	125,42	125,45	125,99	126,3
Desvio Padrão	0,2413	0,1313	0,1206	0,1012	0,1963	0,1585
Speedup		0,9919	0,9917	0,9915	0,9873	0,9848
Eficiência		0,9919	0,4958	0,3305	0,2468	0,197

Observando e analisando os resultados obtidos neste cenário 03, foi possível identificar que aconteceu uma situação semelhante a do cenário 01 e cenário 02, onde houve perda de desempenho em relação às execuções em paralelo com a execução sequencial, onde o aumento do número de escravos utilizados nos testes fez com que o tempo de execução da aplicação paralela também aumentasse. Outro fator a se considerar é que todas as médias dos tempos de execução neste cenário foram superiores a todas as médias dos tempos de execução obtidos nos testes realizados nos cenários 01 e 02. No cenário 01, por exemplo, a média de todos os tempos de execução ficou em torno de 0,5 segundos, já no cenário 02, essa média ficou entre 19,5 e 20,0 segundos, e no cenário 03, em torno dos 125,0 segundos.

Esse aumento no tempo médio das execuções facilitou a observação da diferença entre os tempos de execução de testes com diferentes quantidades de escravos no mesmo cenário. Abaixo no Gráfico 3 é possível verificar a crescente no tempo de execução a medida que o número de escravos aumenta, e o tempo médio das execuções em comparação com os tempos médios dos cenários anteriores.

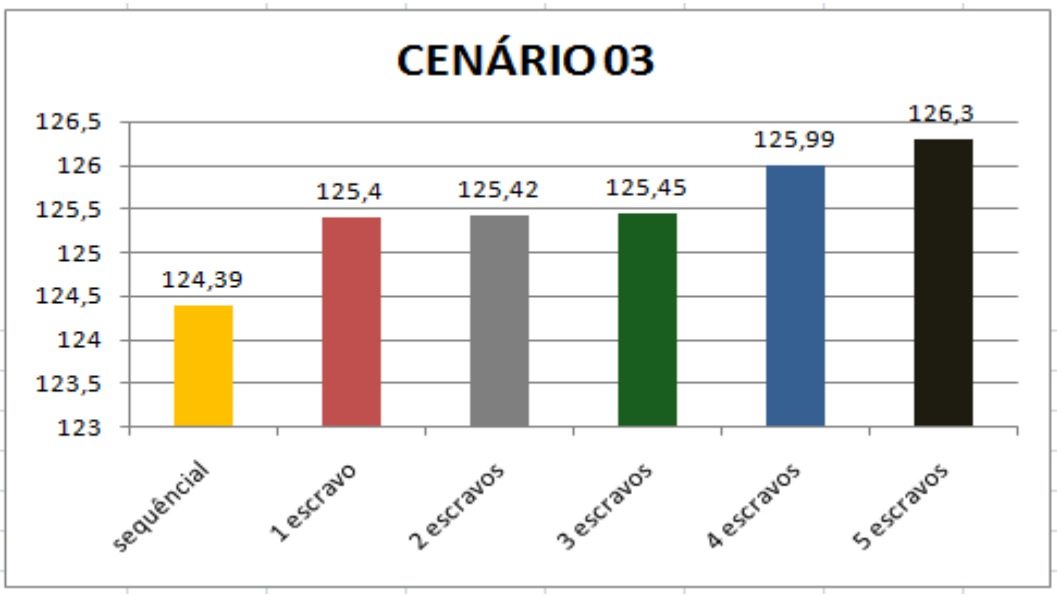

Gráfico 3 – Distribuição das médias dos tempos de execução no cenário 03.

Conclusão Parcial dos Três Primeiros Cenários

Nestes três cenários de testes até aqui relatados, foi possível observar que a execução em sequencial obteve um melhor desempenho do que todas as execuções em paralelo. Tanto nas execuções com apenas um, ou dois escravos, como na execução com cinco escravos, o desempenho foi inferior ao desempenho da execução sequencial. Isso porque, ao serem analisadas as execuções e o comportamento do algoritmo, foi possível observar que ao se dividir e distribuir os intervalos aos escravos, o tamanho de todos os intervalos gerados ainda continuaram muito grandes. Esta situação ocorre por causa da quantidade de escravos utilizados, que neste caso, os cinco escravos utilizados nos testes podem ser considerados um número pequeno de escravos, pois com essa quantidade ainda não há uma divisão do processamento ou quantidade de intervalos necessários para que se consiga algum ganho de desempenho.

Esta divisão em cinco intervalos faz com que o número que precise ser encontrado pelo método de Fermat para se chegar aos fatores, permaneça ainda sempre no primeiro intervalo, como mostrado na Figura 19, ou seja, a aplicação vai sempre executar um mesmo número de vezes em todas as execuções efetuadas, diferenciando apenas no número de escravos utilizados em cada uma delas. O aumento do número de escravos neste caso é o único fator que vai impactar no desempenho, interferindo para uma piora no tempo de processamento, pois como o número a ser encontrado sempre vai estar no primeiro intervalo, e o primeiro intervalo estará contido no primeiro escravo, de nada adiantará aumentar o

número de escravos, pois o mestre terá que efetuar uma invocação a mais a cada escravo adicionado, aumentando a utilização da rede para troca de mensagens.

A Figura 19 mostra um exemplo de quando a divisão dos intervalos propicia um melhor desempenho para execução sequencial, o exemplo mostrado não faz referência a nenhum dos testes realizados anteriormente, mas representa o mesmo que aconteceu em cada um dos três primeiros cenários.

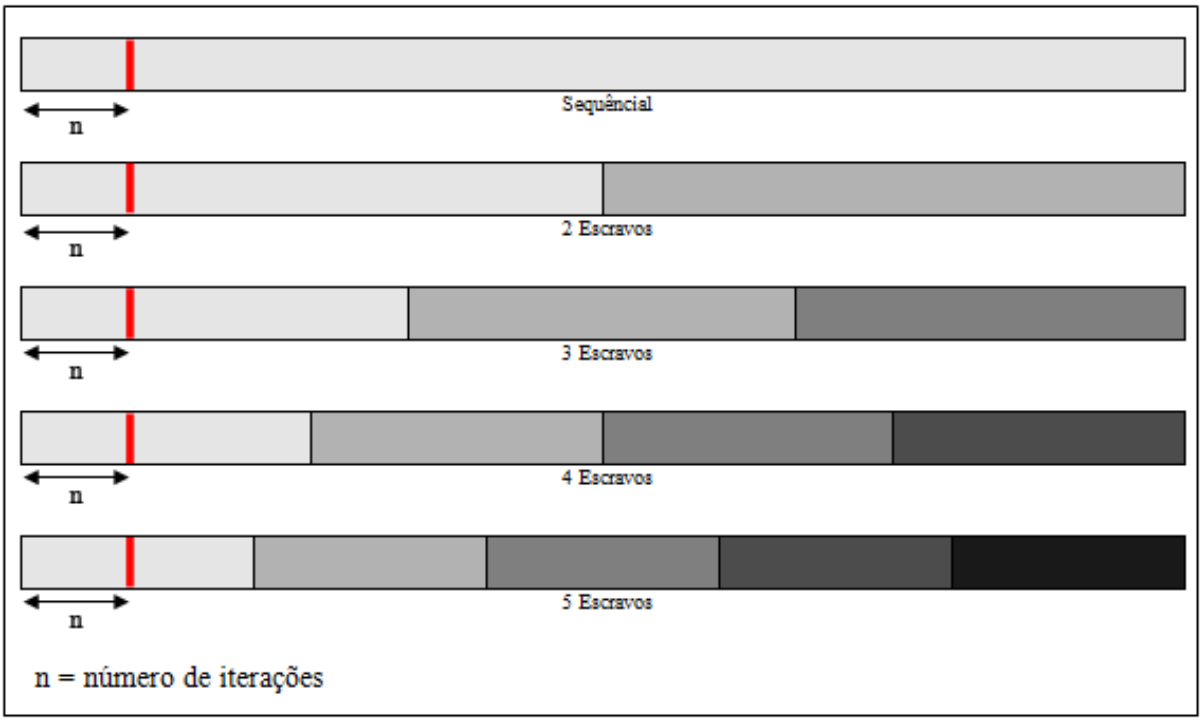

Figura 19– Exemplo de divisão dos intervalos onde o número de iterações sempre é o mesmo.

Na Figura 19 o traço em vermelho representa o número que o método de Fermat precisa para conseguir gerar os fatores, e observando, consegue-se verificar que cinco escravos não é o suficiente para fazer com que este número modifique de intervalo e proporcione um número menor de iterações.

Foi possível observar também o aumento do tempo médio de todas as execuções do cenário 01 para todas as execuções do cenário 02, e do cenário 02 para o cenário 03, este fato acontece devido à distância dos fatores primos. Conforme a distância entre os fatores aumenta, o tempo de execução também aumenta, então, quanto mais próximo estiverem os fatores, mais rápida será a execução.

A seguir foram efetuados mais três testes em outros três cenários, onde o mesmo ambiente de execução dos primeiros três cenários foi mantido, modificando apenas o número

a ser fatorado.

5.2.4. Cenário 04

Após se verificar e explicar a perda de desempenho obtida nos cenários 01, 02, e 03, outros testes foram realizados onde foi possível observar características diferentes em relação a estes três primeiros cenários.

No quarto cenário, o número escolhido para ser fatorado foi o 2.832.547, onde seus fatores são os números primos, 10.009 e 283. Este número foi escolhido, pois possui um tempo de execução viável para ser testado, e por apresentar resultados interessantes e diferentes dos resultados dos cenários anteriores. Ao ser testado de maneira sequencial e paralela, em um ambiente distribuído, com os testes da execução paralela sendo realizados com um, dois, três, quatro e cinco escravos, foram obtidos os resultados que estão sendo apresentados pela Tabela 5.

Tabela 5 – Resultados obtidos com testes realizados no cenário 04

CENÁRIO 04						
	Sequencial	1 escravo	2 escravos	3 escravos	4 escravos	5 escravos
Média	1,21	1.31	1,32	0,45	0,67	0,28
Desvio Padrão	0,0084	0,0063	0,0097	0,0108	0,0067	0,0063
Speedup		0,9236	0,9167	2,69	1,806	4,3214
Eficiência		0,9236	0,4583	0,967	0,4515	0,8643

As análises efetuadas sobre os resultados obtidos neste cenário 04, possibilitou identificar situações onde houve ganho de desempenho, neste caso, a execução com três, quatro, e cinco escravos apresentou um menor tempo de processamento que a execução sequencial, e também com as execuções com um, e dois escravos.

Após se verificar a melhora de desempenho na execução com três escravos, foi possível verificar que nas execuções seguintes, com quatro e cinco escravos, o desempenho sempre continuou melhor que o processamento sequencial. Porém, ao analisar o tempo médio das execuções que tiveram desempenho melhor que a execução sequencial, é possível observar, que na execução com quatro escravos o tempo médio de processamento foi maior do que na execução com três escravos, e na execução com cinco escravos o tempo médio voltou a ser menor que na execução com quatro escravos e menor que em todas as execuções

anteriores. O motivo de isto acontecer será explicado ao término da descrição dos próximos cenários.

Abaixo os dados referentes ao tempo médio das diferentes execuções foram distribuídos pelo Gráfico 4, facilitando uma melhor interpretação dos resultados deste cenário 04.

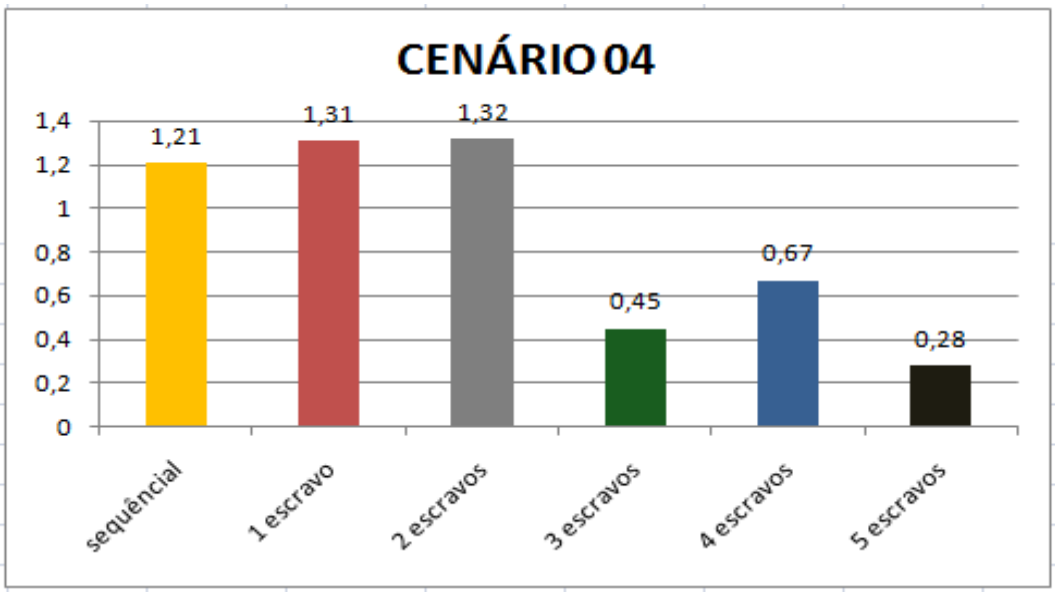

Gráfico 4 – Distribuição das médias dos tempos de execução no cenário 04.

5.2.5. Cenário 05

No cenário 05, o número escolhido para ser fatorado foi o 905.131.559, onde seus fatores são os números primos, 100.559 e 9.001. A escolha deste número se deu pelo fato dele possuir um tempo de execução viável para ser testado, e por apresentar resultados interessantes para serem analisados e explicados.

Após ser realizada a execução de maneira sequencial e paralela, em um ambiente distribuído com os testes da execução paralela sendo realizados com um, dois, três, quatro e cinco escravos, foram obtidos os resultados que estão sendo apresentados pela Tabela 6.

Tabela 6 – Resultados obtidos com testes realizados no cenário 05

CENÁRIO 05						
	Sequencial	1 escravo	2 escravos	3 escravos	4 escravos	5 escravos
Média	7,72	7,85	7,86	0,67	2,45	3,53
Desvio Padrão	0,1218	0,0141	0,0206	0,0067	0,0052	0,0116
Speedup		0,9834	0,9822	11,5224	3,151	2,1869
Eficiência		0,9834	0,4911	3,8408	0,7877	0,4374

Após observação e análise dos resultados obtidos neste cenário 05 foi possível identificar que na execução com três escravos houve o melhor desempenho registrado, com o *speedup* e a eficiência assumindo valores extremamente altos, e que após esta melhora, o tempo médio foi aumentando novamente com o aumento dos escravos, porém ainda sim se manteve inferior ao tempo médio da execução sequencial. A semelhança com o cenário 04 acontece apenas no momento em que há uma melhora no desempenho, que é verificada na utilização de três escravos. Neste mesmo momento também pode ser verificada uma melhora considerável na eficiência.

Abaixo os dados referentes ao tempo médio de execução foram distribuídos pelo Gráfico 5, facilitando uma melhor interpretação dos resultados deste cenário 05.

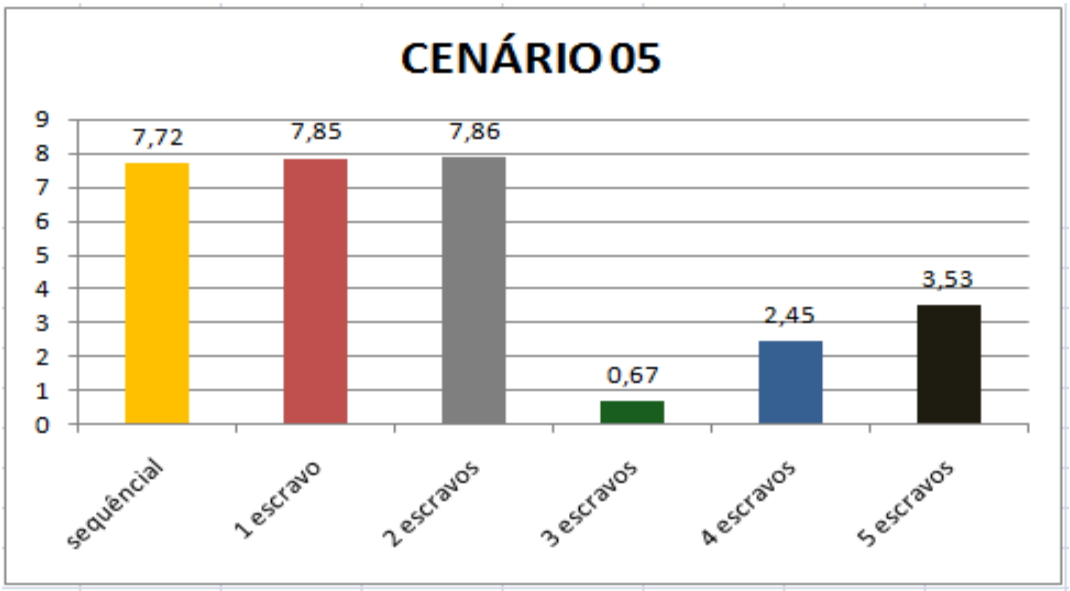

Gráfico 5 – Distribuição das médias dos tempos de execução no cenário 05.

5.2.6. Cenário 06

Neste sexto e último cenário a ser testado, o número escolhido para ser fatorado foi o 16.765.699.193, onde seus fatores são os números primos, 880.043 e 19.051. Este número também foi selecionado pelo fato de possuir um tempo de execução viável para ser testado, e por apresentar resultados interessantes para serem analisados. Ao ser testado de maneira sequencial e paralela, em um ambiente distribuído, com os testes da execução paralela sendo realizados com um, dois, três, quatro e cinco escravos, foram obtidos os resultados que estão sendo apresentados pela Tabela 7.

Tabela 7– Resultados obtidos com testes realizados no cenário 06

CENÁRIO 06						
	Sequencial	1 escravo	2 escravos	3 escravos	4 escravos	5 escravos
Média	97,55	97,68	97,89	9,4	31,4	44,65
Desvio Padrão	0,1898	0,3973	0,1623	0,0169	0,0484	0,0935
Speedup		0,9986	0,9965	10,3776	3,1067	2,1848
Eficiência			0,4982	3,4592	0,7767	0,437

Os resultados observados neste cenário 06 foram semelhantes aos resultados verificados no cenário anterior, que começou a apresentar ganhos de desempenho na execução com três escravos e depois o desempenho foi piorando conforme o número de escravos ia aumentando, porém esta piora não chegou a tornar o desempenho da execução sequencial melhor que o desempenho da execução com quatro e cinco escravos.

Os resultados deste cenário foram distribuídos pelo Gráfico 6, onde ficou mais fácil efetuar uma comparação entre as diferentes execuções.

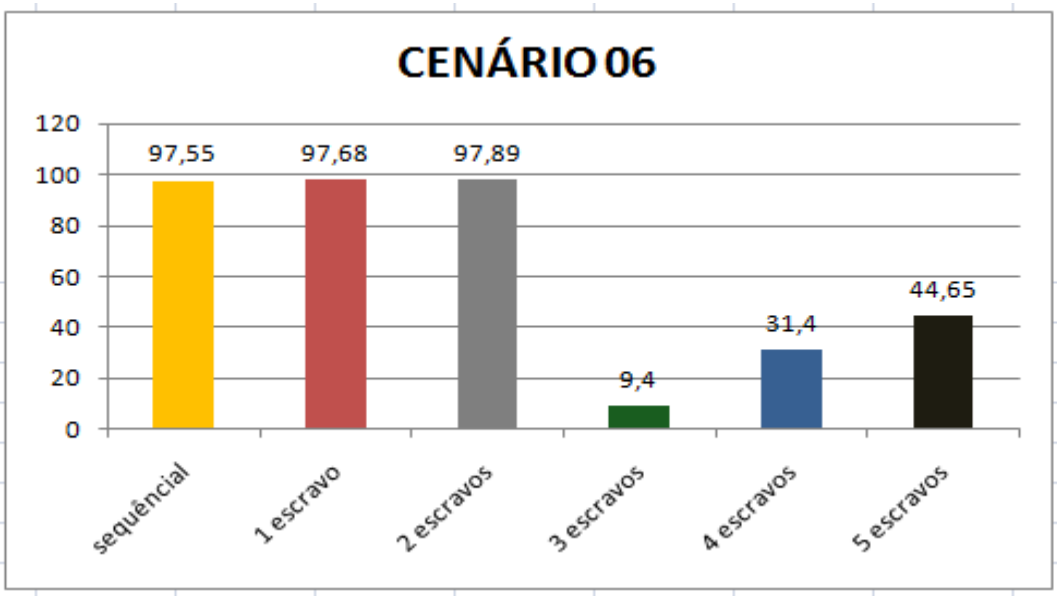

Gráfico 6 – Distribuição das médias dos tempos de execução no cenário 06.

Conclusão Parcial dos Três Últimos Cenários

Nestes três últimos cenários de testes relatados foi possível observar que em algum dos casos a execução em paralelo obteve um melhor desempenho em relação à execução sequencial.

O melhor desempenho obtido em algumas das execuções em paralelo foi efeito da diminuição do intervalo proposto inicialmente, que afetou o tamanho dos intervalos que cada escravo executou.

A diminuição do tamanho do intervalo inicial gerou os mesmos resultados que se fossem aumentados o número de escravos utilizados na execução, pois assim como aumentar o número de escravos, diminuir o tamanho do intervalo inicial fez com que as divisões posteriores para a quantidade de escravos utilizados em uma determinada execução gerassem intervalos de tamanho menor.

Com intervalos de tamanho menor, os escravos executam um menor número de iterações, e consequentemente diminuem seu tempo de processamento. O ajuste no tamanho do intervalo foi a maneira encontrada para que não fosse necessário utilizar um número grande de máquinas, por ser inviável em questão do tempo que seria utilizado nos testes, e pela disponibilidade do hardware necessário.

Ao se dividir os intervalos para os escravos executarem, o número que precisava ser encontrado por algum deles para chegar aos dois fatores primos não ficou localizado sempre no mesmo intervalo, devido à diminuição no tamanho dos intervalos, que fez com que

este número mudasse de intervalo, podendo estar localizado mais próximo do inicio de um dos intervalos, diminuindo o número de iterações necessárias e o tempo de processamento. Diferentemente dos três primeiros cenários analisados, estes três últimos cenários obtiveram um número menor de iterações nas execuções com três ou mais escravos e por este motivo foi possível verificar uma melhora no desempenho em relação à execução sequencial.

A Figura 20 mostra um exemplo de quando a divisão dos intervalos propicia um melhor desempenho para execução paralela, o exemplo mostrado não faz referência a nenhum dos testes realizados, mas representa o mesmo que aconteceu nestes três últimos cenários.

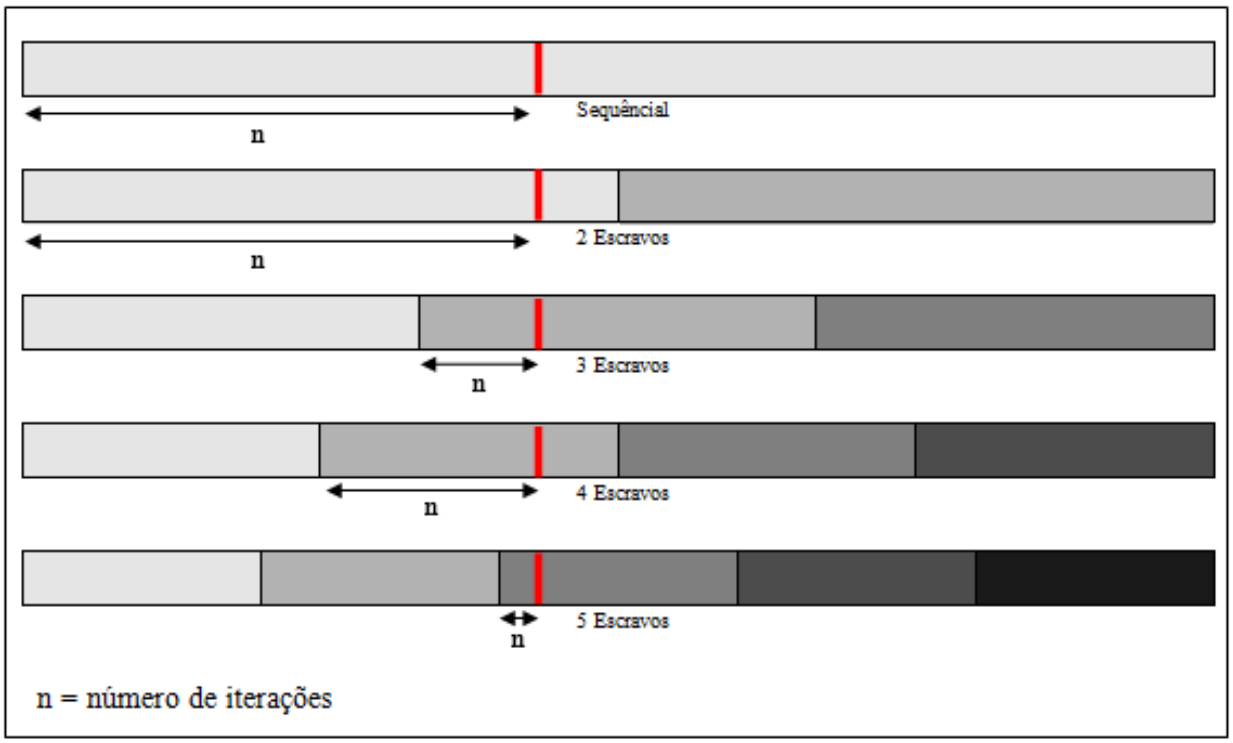

Figura 20– Exemplo de divisão dos intervalos onde o número de iterações modifica com a adição de escravos.

Com a divisão em vários intervalos é possível verificar que o número de escravos não é responsável diretamente pelo melhor desempenho obtido nos testes, no caso do exemplo da Figura 20 sim, pois a execução com cinco escravos obteve o melhor desempenho dentre as outras execuções, porém, ao observar a execução com três e quatro escravos é possível verificar que a execução com três escravos obteve melhor desempenho que a execução com quatro escravos. O fator que influencia diretamente no desempenho é a posição do número que o método de Fermat precisa encontrar dentro do intervalo, visto que quanto mais perto do inicio do intervalo estiver, menor o número de iterações para se chegar até ele, e melhor o desempenho. Apesar da quantidade de escravos não ser um fator que vai impactar diretamente

no desempenho, o número de escravos pode assegurar que quanto maior for sua quantidade, maior será a possibilidade de se obter o melhor desempenho dentre todas as execuções posteriores, pois a cada escravo adicionado, menor serão os intervalos.

5.3. Conclusões Finais

Após todos os cenários serem devidamente testados, é possível chegar a uma conclusão final sobre o custo/beneficio da execução do algoritmo para fatoração de números inteiros.

A execução de modo sequencial apresenta um custo/beneficio melhor do que a execução paralela quando o número que o método de Fermat procura para conseguir gerar os fatores primos do número inteiro fica localizado sempre no primeiro intervalo, mesmo que o número de escravos aumente. Isso acontece, pois a quebra do intervalo proposto inicialmente ainda acaba por gerar intervalos grandes para serem distribuídos, devido ao tamanho do intervalo inicial ser muito grande para ser dividido pela pequena quantidade de escravos utilizados em determinada execução. Ao executar a aplicação, o número que precisa ser encontrado sempre fica localizado no primeiro escravo, modificando apenas o número de invocações efetuadas pelo mestre, e consequentemente o aumento do uso da rede. Portanto a execução sequencial geralmente vai apresentar melhor desempenho quando são utilizados poucos escravos na execução, devido justamente a divisão do intervalo inicial gerar intervalos de grande tamanho.

Para que a execução paralela tenha uma melhora no custo/beneficio em relação à execução sequencial, o ideal seria a utilização de dezenas de máquinas, para que haja uma divisão em mais intervalos e estes fiquem ainda menores, porém com a diminuição manual do tamanho dos intervalos já é possível visualizar ganhos de desempenho. Essa diminuição no tamanho dos intervalos causa os mesmos resultados que um aumento no número de escravos, facilitando a realização dos testes pela não necessidade da utilização de muitas máquinas.

Conforme o número de escravos aumenta e os intervalos vão ficando menores, o número que o método de Fermat procura pode mudar de intervalo, e dependendo da sua localização no novo intervalo, o tempo de processamento será modificado. A melhor das soluções é quando este número se encontra no começo do intervalo, pois um menor número de iterações vai ocorrer, e a pior das soluções é quando este número se encontra ao final do intervalo. No momento em que novos escravos vão sendo adicionados, o tamanho dos

intervalos vai diminuindo proporcionalmente, e com uma grande divisão de intervalos dificilmente haverá um caso onde a execução sequencial terá melhor desempenho que a execução paralela, devido a uma execução com muitos escravos gerar intervalos pequenos para serem distribuídos entre os mesmos, maximizando as chances do número necessário para a resolução da fatoração estar localizado no inicio do intervalo, gerando um número menor de iterações.

5.4. Trabalhos Futuros

Para trabalhos futuros, podem ser feitos testes mais refinados contendo uma imagem igual à Figura 19 e a Figura 20 para cada um dos testes realizados, mostrando onde ficou localizado o número necessário para se chegar aos dois fatores com um exemplo real. Outra possibilidade é modificar o código do algoritmo paralelo distribuído para que este não se limite apenas à execução com cinco escravos, deixando irrestrita a alteração na quantidade de escravos necessários para uma determinada execução, para que após sejam feitos testes mais profundos com a utilização de dezenas de máquinas por exemplo. Além dessas possibilidades, outras diversas mudanças ou melhorias podem ser feitas neste projeto, buscando encontrar novas características e mostrar de maneira cada vez mais clara os fundamentos das conclusões tiradas.

Referências Bibliográficas

ANTUNES, C. M. "**Métodos de Fatoração de Números Inteiros**". Porto Alegre, 2002.

BACELLAR H. V. "**Cluster: Computação de Alto Desempenho**". Instituto de Computação, 2010.

BARNABÉ, V. C. "**Uma introdução aos métodos de fatoração de inteiros de Fermat e Pollard**". Dourados, Universidade Estadual do Mato Grosso do Sul, 2009.

BERNARDI, E. F. F. **"Utilização de programação paralela e distribuída para quebra de chaves de criptografia RSA"**. Porto Alegre: PUCRS, 2006.

COULOURIS G; DOLLIMORE J; KINDBERG T. "**Sistemas Distribuídos".**4 ed., São Paulo: Artmed, 2007.

DANTAS, M. A. R. "**Computação Distribuída de Alto Desempenho: Redes, Clusters e Grids Computacionais**". Editora Axcel, 2005.

DA SILVA, D; FRANCO, C. E. C; AVELINO, F. D,**"Implementação de Sockets e Threads no desenvolvimento de sistemas cliente /Servidor: Um estudo em VB.NET"**. Resende, Associação Educacional Dom Bosco, 2006

DE ROSE, C. A. F. **"Arquiteturas Paralelas"**. Pontifícia Universidade Católica do Rio Grande do Sul, 2001.

DETOMINI, R. C. "**Exploração de Paralelismo em Criptografia utilizando GPUs**". S. J. R. Preto, Universidade Julio de Mesquita Filho, 2010.

JÚNIOR, E.P. F. **"Construindo Supercomputadores com Linux: Cluster Beowulf"**. Centro Federal de Educação Tecnológica de Goiás, Goiânia, 2005.

NODA, A. K. "**Mecanismo de Invocação Remota de Métodos em Máquinas Virtuais**". Pontifícia Universidade Católica do Paraná, 2005.

PEREIRA, F. M. Q. "**Arcademis: Um Arcabouço para Construção de Sistemas de Objetos Distribuídos em Java"**, Belo Horizonte: UFMG Dezembro, 2003.

PITANGA, M. **"Construindo Supercomputadores com Linux"**. 3 ed, Rio de Janeiro, Brasport, 2008.

QUINN, M. J. **"Designing Efficient Algorithms for Parallel Computers"**. McGraw-Hill, Inc., Nova Iorque, NY, EUA, 1987.

SABATINE, R. J. "**Implementação de Algoritmo Paralelo para Apoiar o Processamento de Imagens utilizando JPVM**". Marília: Univem, 2007, pp. 23-43.

SAITO, P. T. M. "**Otimização do Processamento de Imagens Médicas Utilizando a Computação Paralela**". Marília: Univem, 2007, pp. 22-62.

SARAMAGO, J. A. F. G. "**Um Middleware para Computação Paralela em Clusters de Multicores**". Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2012.

TANENBAUM, A. S; STEEN M. V. "**Sistemas Distribuídos**". 2 ed., Rio de Janeiro: Prentice-Hall, 2007.

TOLOUEI, D. V. L. "**Clusters Computacionais de Alto Desempenho**". Vila Velha, 2010.