Universidade Federal Fluminense

EDGAR ELLER JUNIOR

Estudo de Tecnologias para Computação Paralela e Distribuída: Implementação de um Cluster Beowulf.

VOLTA REDONDA

2013

EDGAR ELLER JUNIOR

Estudo de Tecnologias para Computação Paralela e Distribuída: Implementação de um Cluster Beowulf.

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional em Ciência e Tecnologia da Universidade Federal Fluminense, como requisito parcial para obtenção do título de Mestre em Modelagem Computacional em Ciência e Tecnologia. Área de Concentração: Modelagem Computacional.

Orientador:

Prof. Diomar Cesar Lobão, Ph.D.

Coorientador:

Prof. Gustavo Benitez Alvarez, D.Sc.
Prof. Tiago Araújo Neves, D.Sc.

Universidade Federal Fluminense

VOLTA REDONDA

2013

E45 Eller Junior, Edgar.
Estudo de tecnologias para computação paralela e distribuída: implementação de um Cluster Beowulf. / Edgar Eller Junior. – Volta Redonda, 2013.

163 f.

Dissertação (Mestrado em Modelagem Computacional em Ciência e Tecnologia) – Universidade Federal Fluminense.
Orientador: Diomar Cesar Lobão.

1. Cluster. 2. Software. 3. Alto desempenho. 4. Computação paralela. 5. Comparação distribuída. 6. Código computacional. 7. Processos I. Lobão, Diomar Cesar. II. Título.

CDD 004.36

Estudo de Tecnologias para Computação Paralela e Distribuída:
Implementação de um Cluster Beowulf.

Edgar Eller Junior

Dissertação apresentada ao Programa de Pós-graduação em Modelagem Computacional em Ciência e Tecnologia da Universidade Federal Fluminense. como requisito parcial para obtenção do título de Mestre em Modelagem Computacional em Ciência e Tecnologia. Área de Concentração: Modelagem Computacional.

Aprovada por:

Prof. Diomar Cesar Lobão. Ph.D / MCCT-UFF

Prof. Tiago Araújo Neves. D.Sc. / MCCT-UFF

Prof. Orion de Oliveira Silva. D.Sc. / ITA-UNITAU

Prof. Dilson Pereira Caetano. D.Sc. /VCE-UFF

Prof. Jorge Alberto Rodríguez Durán. D.Sc./ UFF

Prof. Luís Alberto Duncan Rangel. D.Sc. / MCCT-UFF

Volta Redonda. 05 de Setembro de 2013.

"... Seja bendito o nome de Deus de eternidade a eternidade,
porque dele são a sabedoria e a força;
E ele muda os tempos e as estações;
ele remove os reis e estabelece os reis;
ele dá sabedoria aos sábios e conhecimento aos entendidos.
Ele revela o profundo e o escondido;
conhece o que está em trevas, e com ele mora a luz".
Daniel 2:20-22

Dedicatória.

Ao meu pai Edgar Eller.

A minha mãe Maria Eugenia, saudades...

A minha amada esposa Tatiane.

A minha filha Estéfani.

Aos amigos e irmãos em Cristo Jesus

Agradecimentos

A Deus, pela saúde e sabedoria que mais ninguém pode dar.

Ao meu pai, Edgar Eller que sempre me apoiou em meus estudos e crescimento.

A minha querida mãe Maria Eugenia que se dedicou ao meu crescimento e educação, até o fim de sua vida aqui na terra.

A minha amada esposa Tatiane por ter me apoiado nos momentos difíceis e ser uma companheira dedicada e atenciosa.

A minha preciosa filha Estéfani por muitas vezes ser privada de minha presença e companhia.

Ao meu grande mestre Prof. Lobão pela orientação deste trabalho, incentivo e paciência, principalmente por não ser somente professor que transmite conhecimento, mas um grande amigo, que proporciona ensinamentos para toda a vida.

Aos meus coorientadores, Prof. Gustavo Benetiz pelo apoio e conselhos, e ao Prof. Tiago Neves por ter paciência em meio a tantas atividades, na orientação dos códigos e nas diversas correções.

A todos os professores deste programa de mestrado, especialmente aos professores Luis Duncan e Thadeu Penna.

A todos os meus amigos e companheiros de "batalha", especialmente ao Mestre Helio Soares, e aos futuros Mestres Leonardo Diniz e Roberto Carlos.

Ao diretor do Instituto de Ciências Humanas e Sociais Prof. Marcelo Amaral e ao diretor do Instituto de Ciências Exatas Prof. José Augusto Huguenin por todo apoio e incentivo.

Aos demais amigos que me apoiaram e contribuíram para a realização deste trabalho, especialmente ao Fábio Martins, Rômulo e Leandro.

Resumo

O presente trabalho é desenvolvido com o objetivo de mostrar os conceitos básicos, aplicações, vantagens e desvantagens de um *Cluster* Beowulf *High Performance Computer* com software livre para códigos computacionais seriais e paralelos. Motivado por esta demanda, demonstrar as facilidades de instalação e recursos de gerenciamento existentes na utilização do *Cluster* Beowulf HPC utilizando o *Rocks* e seu conjunto de aplicativos, principalmente em atividades acadêmicas e cientificas, tendo como principal vantagem ser um software livre. O *Rocks* possui um conjunto de ferramentas e aplicativos, incluindo compiladores Fortran e C, aplicativos de *benchmark* e o OPENMPI utilizado em códigos computacionais paralelos, que juntamente com o sistema operacional CentOS formam uma completa e prática ferramenta para utilização em *Clusters*. Apresenta também comparações de execução de códigos computacionais seriais e paralelos.

Abstract

The present work is developed with the aim of showing the basic concepts, applications, advantages and disadvantages of a Beowulf Cluster High Performance Computer with free software for serial and parallel computer codes. Driven by this demand, demonstrating the ease of installation and management features existing in the use of HPC Beowulf Cluster using the Rocks and its suite of applications, especially in scientific and academic activities, with the main advantage being a free software. The Rocks has a set of tools and applications, including C and Fortran compilers, applications and OPENMPI benchmark computer codes used in parallel, which along with the CentOS operating system form a complete and practical tool for use in clusters. It also presents comparisons between the implementation of serial and parallel computational codes.

Palavras-chave

1. Cluster
2. Software
3. Alto Desempenho
4. Computação Paralela
5. Computação Distribuída
6. Código Computacional
7. Processos

Glossário

SISD	:	*Single Instruction, Single Data*
MISD	:	*Multiple Instruction, Single Data*
SIMD	:	*Single Instruction, Multiple Data*
MIMD	:	*Multiple Instruction, Multiple Data*
SMP	:	*Symmetric Multi Processing*
UMA	:	*Uniform Memory Access*
NUMA	:	*Nom Uniform Memory Access*
COMA	:	*Cache-Only Memory Architecture*
CC-NUMA	:	*Cache-Coherent*
NORMA	:	*No Remote Memory Acess*
MPP	:	*Massively Parallel Processors*
NOW	:	*Network of Workstations*
COW	:	*Cluster Of Workstations*
PC	:	*Personal Computing*
LAN	:	*Local Área Network*
WAN	:	*Wide Area Network*
MP	:	*Message Passing*
PVM	:	*Parallel Virtual Machine*
MPI	:	*Message Passing Interface*
NIC	:	*Network Interface Card*
MAC	:	*Media Access Control*

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos Computacionais

Capítulo 1

Introdução

A cada dia a Ciência evoluí e com esta evolução os problemas científicos tendem a ser maiores e mais complexos. Na área da pesquisa e tecnologia a computação tem um papel importante em praticamente todos o ramos do conhecimento na resolução destes problemas. A computação paralela tem destaque na busca de soluções rápidas para problemas computacionais de grande complexidade e grande volume de informações. Nas últimas décadas temos presenciado um grande avanço na tecnologia de microprocessadores, que hoje são fundamentais para a resolução de problemas científicos e de modelagem. Outro fator fundamental é a diminuição do preço de recursos computacionais e equipamentos possibilitando uma comunidade cada vez maior de centros de pequisas, acadêmicos, governo e empresas possuírem recursos de resolução de problemas com paralelização.

Pode-se citar dentre os vários serviços que utilizam a computação paralela e distribuída as aplicações intensivas de manipulação de dados, processamento de transações e informações, pesquisa e análise de dados, serviços multimídia e aplicações distribuídas em rede [1].

1.1 Objetivo Geral

Este trabalho tem por objetivo geral a montagem de um Cluster Beowulf de alto desempenho, utilizando o software Rocks e OPENMPI, implementando e analisando o desempenho de códigos computacionais seriais e paralelos.

1.2 Objetivos Específicos

Motivado por todos os fatos expostos acima, dentre os objetivos específicos deste trabalho estão:

1. Levantamento das tecnologias atuais existentes na área de computação paralela e distribuída, apresentando uma visão dos componentes de um *Cluster* Beowulf de alto desempenho.
2. Implantação do *Cluster* Beowulf de alto desempenho com *Rocks* e OPENMPI.
3. Analisar as formas e rotinas de paralelização utilizando o OPENMPI
4. Implementar e analisar o desempenho de diferentes códigos computacionais seriais e paralelos.

1.3 Estrutura

Este trabalho está organizado da seguinte maneira:

No Capítulo 2 uma breve exposição sobre os conceitos de programação paralela e distribuída, apresentando as diversas classificações e evoluções das arquiteturas de computadores paralelos e os conceitos básicos de programação paralela. Também aborda os conceitos de um *Cluster* Beowulf, suas características e requisitos, expondo as características e vantagens do *Rocks*, um *Toolkit* de instalação e gerenciamento escolhido para este trabalho.

O Capítulo 3 aborda os componentes e aspectos da implantação do *Cluster* Beowulf com o *Rocks* do Programa de Pós Graduação em Modelagem Computacional em Ciência e Tecnologia, bem como compreende as especificações técnicas de *hardware*, *software* e rede de conexão. Descreve também os problemas e soluções encontradas durante a instalação, que não foram contempladas nas literaturas de instalação.

O Capítulo 4 apresenta uma ampla e exemplificada visão do MPI, mais especificamente o OPENMPI, com seus comandos de passagem de mensagens, que proporciona a comunicação entre processos paralelizados.

O Capítulo 5 expõe três problemas que foram propostos para resolução e implementação com códigos computacionais seriais e paralelos, demonstrando o funcionamento do *Cluster* VORTEX.

No Capítulo 6 são apresentados e discutidos os resultados, metas e objetivos alcançados neste trabalho, gerando também propostas para futuros trabalhos.

Nos Apêndices A e B estão descritos, respectivamente, a instalação e teste de funcionamento do *Cluster* VORTEX e os códigos computacionais desenvolvidos e utilizados neste trabalho.

Capítulo 2

Computação Paralela e Distribuída

A computação Paralela e Distribuída tem despontado como uma solução de custo-benefício atrativa para a rápida resolução de problemas computacionais, principalmente na área Científica, gerando assim um interesse crescente pela implantação de *Clusters* computacionais, processamento em nuvem e consequentemente o desenvolvimento de aplicações e *softwares* paralelos e aplicações distribuídas.

Neste capítulo são abordados os princípios de computação paralela e distribuída, a classificação e evolução das arquiteturas de computadores paralelos, os conceitos de *Cluster*, com enfoque para o *Cluster* Bewoulf utilizando *Rocks* e sua arquitetura de funcionamento.

2.1 Computação Paralela

A computação paralela pode ser entendida como a aplicação de dois ou mais unidades de processamento para resolução de um problema, sendo estas unidades processadores físicos ou lógicos. Para entendimento da computação paralela é necessário a compreensão de três áreas: Arquiteturas, Linguagens de Programação e Algorítimos [2].

1. Arquiteturas: conhecer as arquiteturas computacionais se faz necessário pois influem diretamente no desempenho e comportamento de um código computacional paralelo. A arquitetura computacional difere na forma como são programados resultando no diferencial de desempenho, sendo algumas arquiteturas menos adequadas para certas aplicações. Os principais componentes de um sistema de computação utilizados em computação paralela são:

- Processador - utiliza os dados para realizar operações matemáticas
- Memória - dispositivo que armazena dados e disponibiliza para outros componentes
- Switch - dispositivo comumente utilizado, pertencente a infra estrutura de rede, que permite que os dados sejam transferidos entre dispositivos.

2. Linguagens de Programação: várias linguagens de programação estão disponíveis para utilização em computação paralela, algumas utilizadas em diferentes arquiteturas computacionais com apenas alterações em sua compilação e outras mais especificas para cada arquitetura. Estas linguagens que proporcionam execução em paralelo possuem recursos para coordenação e controle de tarefas. Um exemplo são as linguagens Fortran e C que veremos mais adiante.

3. Algorítimos: sendo parte fundamental no processo de paralelização, requer a intervenção no código computacional, porém existem métodos que não necessitam desta intervenção, sendo a paralelização menos eficiente. Um dos fatores primordiais na paralelização de um código é a remoção de dependências de dados que existem na maioria dos códigos. Entende-se como dependência de dados em paralelismo uma instrução que depende de outra para execução sendo crítico para o processo de escalonamento, sendo assim necessário determinar quais instruções podem executar em paralelo. Dentro deste conceito dois tipos de dependência são limitadoras para a paralelização de códigos computacionais: dependência de controle que resulta do fluxo de instruções do algoritmo e dependência de dados que resulta do fluxo de dados durante a execução de um código computacional ou seja de como os dados no decorrer do programa são modificados e utilizados ao longo do fluxo. As dependências forçam o sequenciamento de comandos e instruções, prejudicando assim a paralelização do algoritmo.

   Dependendo da estrutura do algoritmo a paralelização se torna difícil e até mesmo inviável. Neste caso os algoritmos devem ser modificados ou desenvolvido novos para a execução paralelizada. Em grande parte das aplicações cientificas encontra-se múltiplos algoritmos que atendem as necessidades, devendo sempre buscar o aumento da eficiência do código computacional paralelo.

Em Computação Paralela tem-se a divisão entre vários processadores das tarefas a serem executadas, ou seja, a divisão de grandes execuções computacionais de partes divisíveis em tarefas menores para vários processadores executando simultaneamente. Estes

processos comunicam-se entre si, para que sejam sincronizados e monitorados em sua execução. Ao contrário da computação paralela, a computação serial executa suas tarefas em série, conforme Figura 2.1.

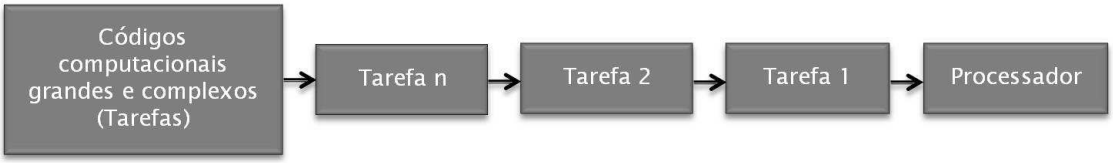

Figura 2.1: Processamento serial.

Observa-se que a simples divisão das tarefas em processadores não diminui o tempo de execução na mesma proporção, o código a ser paralelizado e o tempo de comunicação e sincronismos entre processos devem ser levados em consideração para se obter um menor tempo de execução. A Figura 2.2 indica que o paralelismo deve ser utilizado em complexos códigos computacionais, efetuando a divisão deste código ou tarefas em tarefas menores, distribuídas e executadas simultaneamente por vários processadores [3].

As plataformas de computação paralelas atendem a uma grande variedade de aplicações nas áreas de dinâmica dos fluidos, mecânica estrutural, processamento de dados em transações e recuperação de informação, análise e manipulação de dados, serviços multimídia, meteorologia, nanotecnologia e quântica [1].

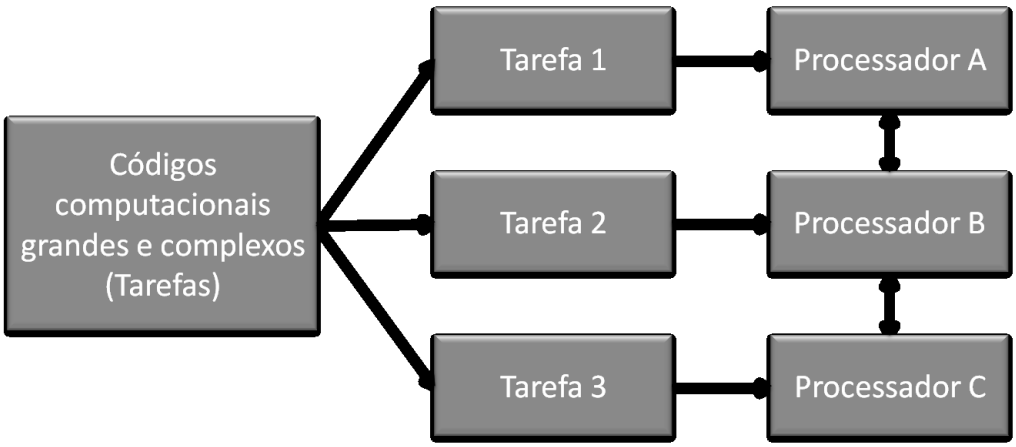

Figura 2.2: Paralelizando tarefas. [3]

2.1.1 Taxonomia de Flynn

Michel Flynn, em 1972, classificou vários modelos de arquitetura de computadores, como mostrado na Figura 2.3, segundo a quantidade de fluxos de dados e de instruções em cada instante [3]:

- SISD (*Single Instruction, Single Data*): é caracterizado por executar somente uma instrução por vez, também chamada de execução serial ou sequencial. São máquinas mais simples com um processador. Exemplos: Computadores pessoais, estações de trabalho com uma única CPU, minicomputadores e mainframes

- MISD (*Multiple Instruction, Single Data*): arquitetura que executa sobre um único fluxo de dados várias instruções simultaneamente. São máquinas de propósito especiais e consideradas úteis quando e mesma entrada *input* é sujeito a várias operações diferentes. Podemos ter como exemplo múltiplos algoritmos de criptografia para decodificar uma mensagem ou múltiplos filtros operando sobre um único fluxo de sinal

- SIMD (*Single Instruction, Multiple Data*): caracterizada por executar os dados em paralelo, ou seja, uma instrução é executada paralelamente utilizando vários fluxos de dados sincronizados, executando um único programa ao mesmo tempo. Este modelo é adequado para o tratamento de conjuntos regulares de dados, como as matrizes e vetores.

- MIMD (*Multiple Instruction, Multiple Data*): apresenta como um conjunto de máquinas SISD onde vários processadores executam programas diferentes. Nesta arquitetura temos cada um dos processadores agindo independentemente, efetivamente executando paralelamente, gerando múltiplos fluxos de instruções e múltiplos dados.

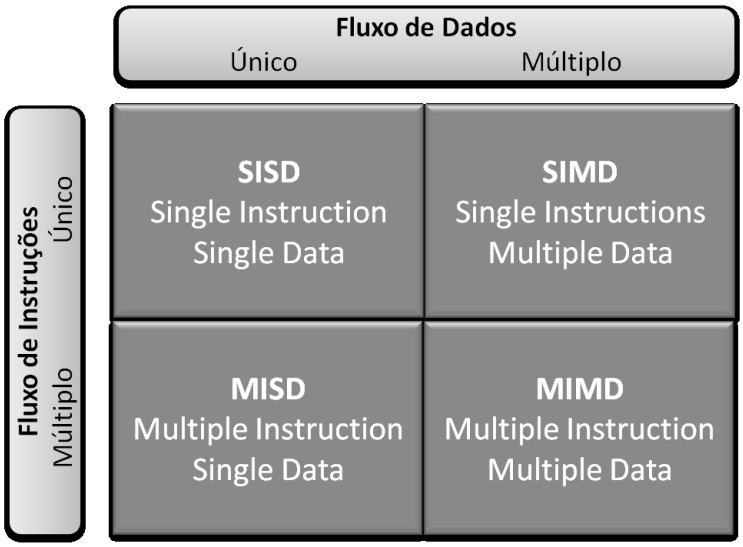

Figura 2.3: Modelo da taxotomia de Flynn. [3]

A taxonomia de Flynn pode ser estendida considerando o grau de acoplamento da memória principal, como mostrado na Figura 2.4. Os Multiprocessadores de memória compartilhada possuem um sistema fortemente acoplado, enquanto os Multicomputadores de troca de mensagens um sistema fracamente acoplado [4].

Como exemplificado anteriormente podemos ainda observar que dois elementos são críticos para o desempenho da computação paralela. A organização das plataformas paralelas podem ser entendida como organização lógica, que expressa as tarefas paralelas e a interação entre estas tarefas; e física, que se refere à organização do *hardware* da plataforma. A organização lógica pode ser referenciada como estrutura de controle e modelo de comunicação [1].

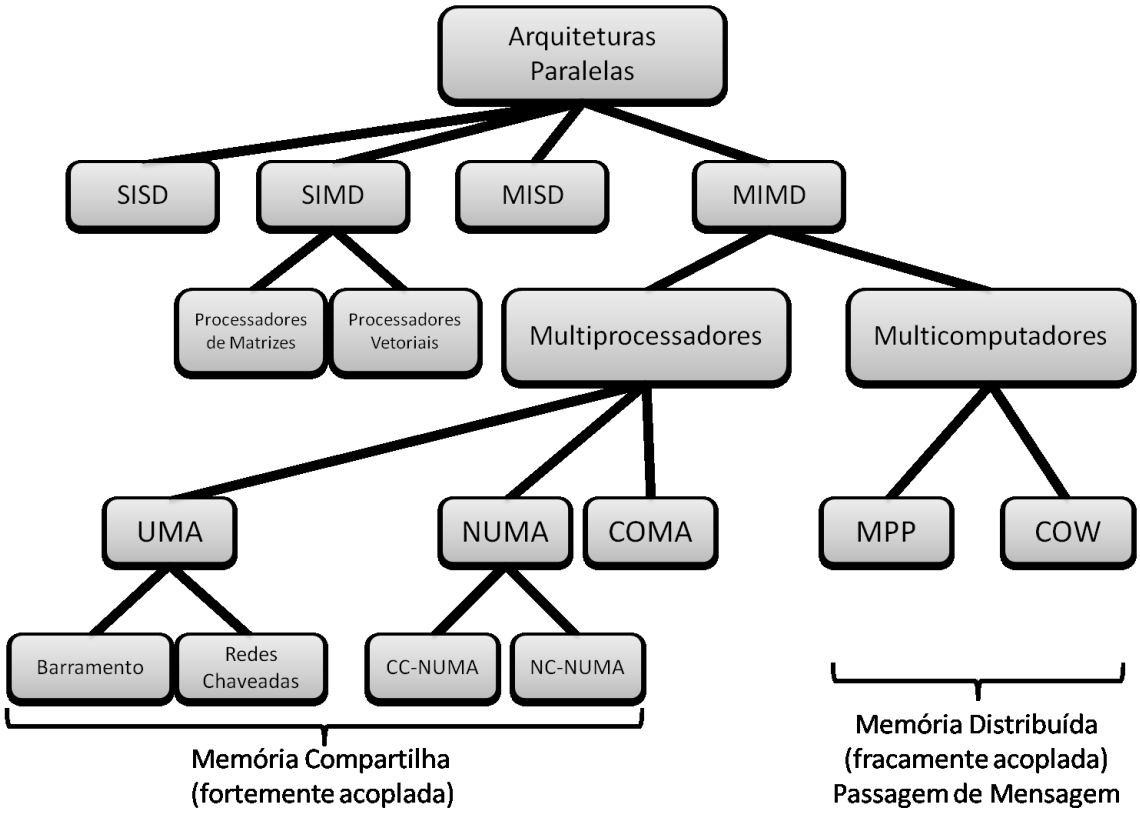

Figura 2.4: Modelo da taxonomia de computadores paralelos.[4]

Os itens deste modelo serão descritos ao longo deste capítulo.

2.1.2 Estrutura de Controle

Considera-se que um código computacional de um conjunto de códigos pode ser entendido como uma tarefa paralela assim como as instruções individuais que compõem este código. Nas arquiteturas SIMD as instruções para cada unidade de processamento são enviadas de uma única unidade de controle sendo a mesma instrução executada por todas as unidades de processamento. Este modelo apresenta resultados satisfatórios para cálculos sobre estruturas paralelas de dados, como matrizes. Em contraste com esta arquitetura, o MIMD onde cada processador é capaz de executar programas diferentes com múltiplos fluxos de dados. Este tipo de sistema requer mais *hardware* comparado com o modelo SIMD pois este possui somente uma unidade de controle global, além de utilizar uma quantidade menor de memória devido ao armazenando de apenas uma cópia do código.

2.1.3 Modelo de Comunicação de Plataformas Paralelas

A troca de dados entre tarefas paralelas pode ser feita através de compartilhamento de memória sendo acessível a todos os processadores ou a troca de mensagens entre eles. Para maior entendimento observa-se dois modos de classificar quanto a utilização da memória [3]:

1. Memória Compartilhada: Caracterizado por vários processadores que compartilham a mesma memória, onde o ambiente é fortemente acoplado a nível de processador e memória em um mesmo sistema local, como mostrado na Figura 2.5.

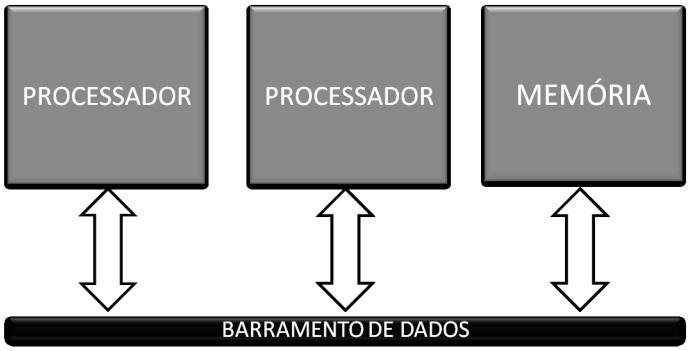

Figura 2.5: Modelo de memória compartilhada. [3]

Vantagens:

- Maior rapidez no compartilhamento de dados entre os processos.

Desvantagens:

- Os processadores são limitados fisicamente devido a largura de banda do barramento.
- Necessidade de sincronização de operações de leitura e escrita dos dados.
- Custo elevado do *hardware* especifico.

2. Memória Distribuída: Várias unidades de processamento cada uma com sua própria memória, ou seja, Multicomputadores com suas memórias locais sendo a comunicação executada por troca de mensagens entres os processos, como mostrado na Figura 2.6.

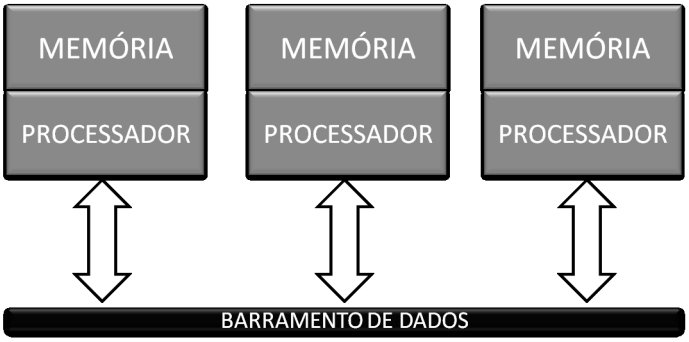

Figura 2.6: Modelo de memoria distribuída. [3]

Vantagens:

- Projeto simples e relativamente barato
- Alta escalabilidade

Desvantagens:

- Paralelismo não é intuitivo e a necessidade dos processos enviarem mensagens explicitas a outros processos.
- O desempenho esta diretamente ligado a comunicação entre processos que por consequência necessita de uma rede de alto desempenho e confiabilidade.

Considerando estes conceitos pode-se descrever quanto ao Modelo de Comunicação de Plataformas Paralelas [3]:

SMP – Multiprocessadores Simétricos

Diferentemente do sistema anterior o SMP é um sistema de processamento simétrico e considerado de arquitetura de memória compartilhada (Multiprocessador), sendo que um barramento ou uma infra estrutura de rede de comunicação dedicada é utilizado para disponibilizar toda a memória do sistema. Observa-se que a comunicação entre tarefas é rápida porém a escalabilidade é limitada pela saturação do barramento de comunicação, interferindo diretamente no tempo de acesso a memória.

A forma de interligação entre processador e memória, em sistemas Multiprocessadores, podem ser categorizados como:

- UMA (*Uniform Memory Access*): memória centralizada e fortemente acoplada. Sua largura de banda é limitada devido ao congestionamento de conexões, sendo minimizado através da utilização de *caches* nos processadores.

- NUMA (*Nom Uniform Memory Access*): consiste em vários módulos de memória distribuída e associada aos processadores, porém com endereçamento único podendo acessar toda a área de memória, propiciando um maior número de processadores interligados.

- COMA (*Cache-Only Memory Architecture*): utiliza-se memória *cache* como uma memória de endereçamento global, não existindo um hierarquia de memória por processador.

- CC-NUMA (*Cache-Coherent*): consiste em um sistema escalável de multiprocessadores, arquitetura de coerência de *cache* e acesso não uniforme. Cada processador possui um espaço de memória reservado, como memória local, porém tendo visão da memoria global. Utilizam-se de memoria *cache* local que reduz o tráfego de rede.

MPP - Processadores Massivamente Paralelos

Caracteriza-se por ser um grande sistema de processamento paralelo com arquitetura de múltiplas memórias locais, composta por vários processadores interconectados por uma rede de alta velocidade. Entende-se como nó um computador ou máquina computacional interconectado a outros nós. Cada nó tem seu próprio sistema operacional, executando aplicações localmente e comunicando através de pacotes de troca de mensagens. Cada nó possui sua própria memória local, não existindo um espaço de endereçamento global com acesso remoto as memórias, e podendo ser formado por um ou mais processadores. O modelo de memória utilizado em sistemas MPP, também denominado Multicomputadores, possui arquitetura fracamente acoplada, também conhecida como NORMA (*No Remote Memory Acess*).

NOW - Redes de Estações de Trabalho

Formada de múltiplas estações de trabalho interligadas com redes de comunicação de baixo custo, como *Ethernet*, utilizadas em aplicações paralelas. Não são máquinas dedicadas a este trabalho, por este motivo não são otimizadas para estas tarefas, porém possui um baixo custo. Podem ser vistas também como máquinas NORMA em relação a utilização de suas memória.

COW - *Cluster* de Estações de Trabalho

Também formada por estações de trabalho, similar ao NOW, porém dedicadas ao processamento paralelo. Seu Sistema Operacional é otimizado para as aplicações paralelas, garantindo uma performance maior na execução de códigos computacionais paralelos. Utiliza-se de redes de alta velocidade, padrão Gigabit Ethernet, e possui aplicações direcionadas ao controle e gerenciamento utilizando um único nó central.

2.2 Computação Distribuída

Um sistema distribuído compreende a agregação de vários computadores ou sistemas computacionais, que interligados através de redes de computadores privadas ou em nuvem, permite o compartilhamento de seus recursos através de *softwares* de sistema distribuído, independente de configurações homogêneas de *hardware* e *software*.

Entende-se como computação em nuvem um conjunto de máquinas ou servidores, físicos ou virtuais, utilizados em um ambiente computacional, provendo recursos de capacidade de processamento, armazenamento, conectividade, plataformas, aplicações e serviços de *Internet* [5]. Uma das grandes vantagens da computação em nuvem e não ter a necessidade de instalação e armazenamento em espaço pré-determinado de equipamentos físicos, visto que estes equipamentos podem estar instalados em locais remotos e serem agregados ao seu ambiente computacional a qualquer momento dependendo da demanda e fatores financeiros.

Computação distribuída tem por objetivo compartilhar recursos de computadores de maneira a utilizá-los otimizadamente, permitindo a formação de SMPs, MPPs, *Clusters* e *Grids* computacionais.

Um sistema distribuído deve possuir algumas características [6]:

- Transparência: Qualquer processo pode ser executado em qualquer máquina da rede de maneira transparente ao usuário;
- Confiabilidade: Alta disponibilidade dos serviços. Atinge-se este patamar utilizando vários equipamentos em conjunto;
- Concorrência: Caracterizado pelo compartilhamento de recursos com uma melhor utilização da carga de processamento entre todas as máquinas;

- Escalabilidade: Capacidade de agregar novos recursos ao sistema, tanto *hardware* quanto *software*;
- Flexibilidade: Flexibiliza as decisões de projeto, permitindo mudanças no sistema, caso necessite;
- Heterogeneidade: Diversidade de elementos computacionais, como plataformas de *hardware* ou *software*.

2.3 *Cluster*

Considera-se como definição de *Cluster* um conjunto de computadores pessoais interligados através de uma rede local ou em nuvem executando um sistema distribuído para a emulação de supercomputadores, porém baseando-se em *Work Stations* e PCs (*Personal Computing*). Define-se como um sistema distribuído uma coleção de computadores independentes que se apresentam ao usuário como um sistema único e consistente [7].

A computação em *Cluster* tem se desenvolvido como uma solução de baixo custo e de fácil acesso face ao alto custo de supercomputadores, para atender as demandas principalmente da área acadêmica [8]. Outro fator relevante para a intensificação da utilização dos *Clusters* é a melhoria das redes locais (LANs – *Local Área Network*) que proporcionam equipamentos trabalhando em altas velocidades (de 1 Gbps a 10 Gbps), juntamente com a evolução dos sistemas em nuvem.

Classifica-se os *Clusters* como homogêneos ou heterogêneos, de acordo com o seu *hardware* e equipamentos de rede. Entende-se por *Cluster* heterogêneos aqueles que possuem diferentes configurações de seus nós ou da rede de comunicação que estão interligados, ou seja, os nós e a rede diferenciam-se por *hardware* e conexões. Contrariamente, o *Cluster* homogênio possui *hardware* e rede de comunicação idêntico em seus nós, sendo muito mais raro de ser encontrado devido a troca ou inclusão de novos equipamentos e o avanço da tecnologia. Uma preocupação quanto a *Clusters* heterogêneos é o balanceamento da carga que cada nó deverá receber. A tendencia é a distribuição igualitária de todas as tarefas pelas quantidade de nós, porém deve-se fazer uma análise da capacidade de processamento de cada nó e utilizar um algoritmo de balanceamento para distribuir a carga compatível com a capacidade de cada nó [3].

Classifica-se os *Clusters* de acordo com suas características de trabalho, como observado a seguir.

1. *Cluster* de Alta Disponibilidade (*High Availability*).
2. *Cluster* para Balanceamento de Carga (*Load Balancing*).
3. *Cluster* de Alto Desempenho (*High Performance Computer*).

2.3.1 *Cluster* de Alta Disponibilidade (*High Availability*)

São *Clusters* baseados em redundância, onde existem nós que podem realizar as tarefas de outros nós no caso de falhas, permitindo que seus sistemas permaneçam ativos por um longo período de tempo e em plena condição de uso, independente de falhas de *hardware* [9]. Neste tipo de *Cluster* seu projeto deve levar em conta as possíveis vulnerabilidades, que possam ocasionar falhas e a partir desse levantamento são consideradas medidas de redundância como a utilização de servidores interligados [10]. São exemplos de *Clusters* de Alta Disponibilidade: servidores de banco de dados, serviços para WEB (páginas, FTP, e-mail) e *backup*.

2.3.2 *Cluster* para Balanceamento de Carga (*Load Balancing*)

Esse tipo de *Cluster* possui a característica de reduzir a latência e aumentar a capacidade de execução de tarefas [9]. Tem como objetivo a distribuição adequada de processos ao longo dos nós do agrupamento de computadores, com a ajuda de algoritmos de escalonamento [3].

2.3.3 *Cluster* de Alto Desempenho (*High Performance Computer*)

Também conhecido como *Cluster* de Alta Performance ou *Cluster* HPC, ele funciona permitindo que ocorra uma grande carga de processamento com um alto volume de operações de ponto flutuante em computadores comuns ou estações de trabalho, utilizando sistema operacional gratuito, o que diminui seu custo [3].

O principal objetivo do *Cluster* HPC é otimizar a resolução de problemas de alta complexibilidade, proporcionando grande poder computacional com um custo pequeno se comparado a supercomputadores.

São características da utilização da computação paralela e distribuída na otimização dos recursos computacionais e uma melhor performance dos sistemas. Podemos enumerar

diversas vantagens da utilização do *Cluster* HPC [3].

- Alto Desempenho: possibilidade da utilização da paralelização de programas complexos que diminui o tempo de processamento e resolução do problema.
- Escalabilidade: novas unidades de processamento, memória e *hardware* em geral podem ser acrescentadas para o aumento do poder de processamento, sem alteração das características primárias do *Cluster*.
- Tolerância a falhas: caso algum nó falhe, desconsiderando o nó principal, existe a possibilidade de utilizarmos os outros nós do sistema, aumentando assim a confiabilidade como um todo.
- Baixo custo: redução do custo utilizando computadores pessoais ou estações de trabalho se comparado a supercomputadores.
- Independência de fornecedores: utilização de *software* livre e *hardware* independente de fabricantes ou tecnologia proprietária.

2.4 *Cluster* Beowulf

Os *Clusters* Beowulf de Alto Desempenho - HPC foram desenvolvidos para atender a crescente necessidade de capacidade de processamento utilizado nas diversas áreas do conhecimento, principalmente na área científica e de pesquisa, com o objetivo de implementar sistemas computacionais cada vez mais poderosos e com um baixo custo. Esta diminuição do custo deve-se também às novas tecnologias de rede e processadores, com equipamentos mais velozes e de baixo custo e a utilização de *software* livre, tanto no Sistema Operacional com as distribuições de Linux, quanto nos demais *softwares* de controle e paralelização que possuem licenças *GNU Public License - GPL e Open Source* [10].

O nome Beowulf foi originado de um poema épico da língua inglesa, que conta a história de um herói dos Gautas que livra os dinamarqueses de dois monstros pois possui grande força e coragem [11].

Pode-se caracterizar este tipo de *Cluster* pela utilização de:

- Computadores Pessoais utilizados em grupo para o aproveitamento de seu potencial computacional.

- Interconexão através de rede local de alta velocidade, padrão *Ethernet*, utilizando para comunicação entre os nós, o *Message Passing*, que veremos nos próximos capítulos.
- Sistema operacional *Open-Source*.

Com estas características os *Clusters* Beowulf tiveram um grande avanço na disseminação deste tipo de *Cluster*, com um custo-benefício muito bom, atendendo as demandas por computação paralela.

2.5 *Cluster* HPC Beowulf com *Rocks*

Em maio de 2000 um grupo de pesquisadores iniciou um trabalho para que a implantação de *Cluster* com *software* livre fosse uma tarefa mais fácil e intuitiva. O *Rocks* (*Open-Source Toolkit for Real and Virtual Clusters*) é um sistema para implantar e gerenciar o *Cluster* visando atender uma grande demanda de usuários científicos, melhorando e facilitando a utilização de ferramentas de execução de códigos computacionais paralelos [12].

Para o gerenciamento de um *Cluster* verifica-se alguns aspectos do funcionamento do *software* [9], conforme Figura 2.7:

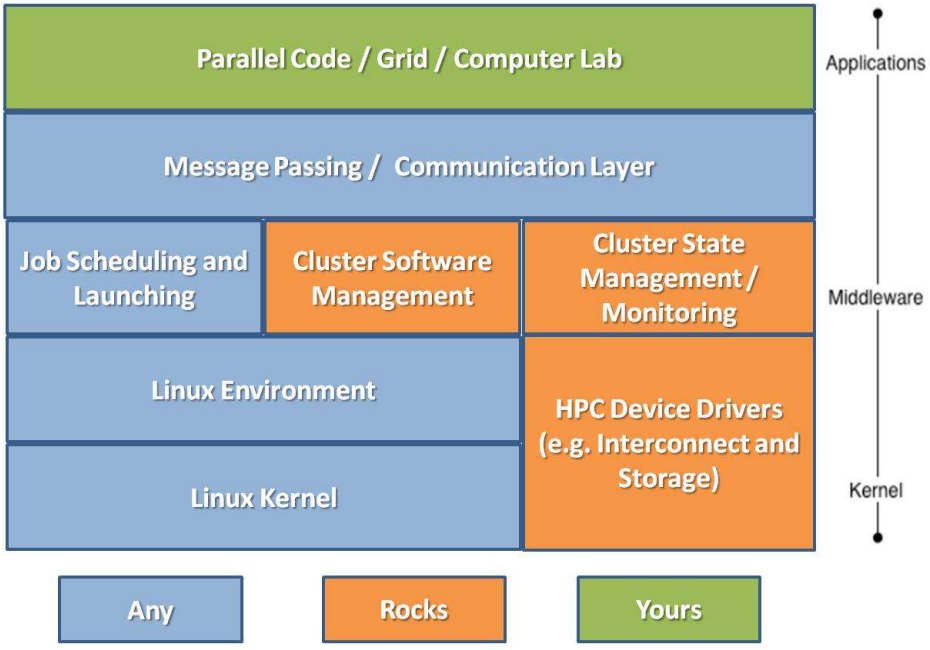

Figura 2.7: Modelo de *softwares* de um *Cluster*.[13]

- Administração: para administração necessita-se conhecer a situação de funcionamento do *frontend* e dos nós interligados ao *Cluster*. Entende-se por *frontend* o nó principal ou computador mestre que gerencia o *Cluster*. Garante-se assim a estabilidade e disponibilidade do sistema. Esta tarefa é realizada pelo *Cluster State Menagement/Monitoring*.

- Aplicações: domínio e coerência das configurações e bibliotecas para um correto funcionamento das aplicações em paralelo. Neste aspecto um fator de grande importância e a homogeneidade das arquiteturas utilizadas. Sabe-se que devido a expansão e manutenção do *Cluster* esta tarefa fica muito difícil, mas deve-se esforçar para conservar a mesma arquitetura. O *Cluster Software Management* e seus módulos são responsáveis por estas aplicações e suas bibliotecas.

- Controle das Tarefas Paralelas: para que o controle de execução, monitoramento e eliminação de tarefas seja efetivo e assim utilizar da melhor maneira possível os recursos, as camadas de *Massagem Passing*, *Comunication Layer e Job Scheluding and Launching*, executam e controlam múltiplas tarefas sequenciais, efetuando a comunicação e monitoramento dos processos.

A camada *Middleware* em um sistema de computação distribuída funciona como uma camada de abstração ou mediação entre sistemas de aplicações diferentes, independente do número de nós, arquiteturas ou programas de diferentes protocolos de comunicação [9]. É responsável por disponibilizar os nós conectados aos sistemas, que são independentes, como um sistema único (SSI - *Single System Image*) [3].

Em razão destas características de gerenciamento de um *Cluster* pode-se listar as principais características de utilização de um *Cluster Rocks* [9]:

Instalação

De fácil instalação em relação a outros *Toolkits*, sendo sua característica mais forte. Em cerca de uma hora podemos ter a instalação básica de um *Cluster Rocks*. Ele permite que a instalação inicie no computador mestre denominado de *frontend*, permitindo ser instalado via Web ou através de mídia. Através desta primeira instalação ele gerencia a instalação dos demais nós, oferecendo instalação via rede ou mídia. Desta forma a instalação dos nós torna-se mais prática e rápida. Possui grande vantagem quando trabalhamos com *Clusters* heterogêneos, pois neste caso o método de geração de imagens de nós não pode ser utilizada, sendo que a instalação do *Rocks* suporta diferentes *hardwares*.

Manutenção dos nós

Todo o controle de manutenção e realizado no *frontend* que atualiza os nós do *Cluster*. Possui uma base de dados contendo as informações de configuração dos nós e seu estado atual. Devido a fácil instalação dos nós, em caso de falha, indica-se a re-instalação do nó com problema, o que em outros sistemas seria uma tarefa muito difícil e demorada.

Variedade de *softwares*

Dispõem de grande variedade de *softwares* de comunicação, segurança, controle e administração de *Cluster* integrados a sua instalação, podendo ser instalados na primeira instalação do *frontend*. Os módulos de *software* são chamados de *Rolls*, contendo todas as instruções de instalação e configuração. Por estar em pacotes pode-se adicionar outros aplicativos a qualquer momento, sendo sua instalação muito intuitiva.

Minimiza o tráfego de rede

Através da instalação do Sistema Operacional no próprio nó do *Cluster* e das aplicações básicas de funcionamento diminuísse o tráfego na rede, pois o tráfego é constituído somente de informações necessárias e imprescindíveis a execução dos códigos computacionais.

Para melhor entendimento dos próximos capítulos alguns componentes de um *Cluster* precisam ser conhecidos, levando em conta a utilização do *Rocks*.

- *frontend* e nó - considera-se o *frontend* como o computador mestre onde serão instalados os *softwares* de gerenciamento e manutenção do *Cluster*, tendo uma porta de comunicação com a Internet pública (WAN - *Wide Area Network*) e outra interface de comunicação com a rede Local (LAN - *Local Area Network*) que ficará totalmente restrita ao *Cluster*. Pode-se interligar a esta rede local uma grande quantidade de nós, dependendo da infra-estrutura de sua rede de comunicação. Nestes nós são instalados o sistema operacional e aplicações básicas, sendo seu gerenciamento efetuado pelo *frontend* [14].

- Sistema Operacional - baseado em *Open-Source*, o Linux se tornou um sistema ideal para este tipo de aplicação, sendo um sistema robusto e permitindo alterações e correções em seu código-fonte aberto [3], que é a tendencia atual. O *Rocks* 5.4.3 (Viper), versão utilizada neste trabalho, traz nativamente o CentOS 5.6 como sistema operacional baseado no Red Hat Linux [13].

- Rede Local - responsável pela interligação do *frontend* e dos demais nós influenciando diretamente no desempenho do *Cluster*. A utilização de equipamentos padronizados em sua tecnologia e velocidade são de fundamental importância para uma rede sem pontos de "gargalo"ou estrangulamento de transmissão de pacotes. Nesta aplicação foi utilizada uma rede Gigabit *Ethernet* [9].

- Protocolos - possibilita a transmissão de dados entre os nós da rede através de um conjunto de regras e padrões de transmissão. Utiliza-se na transmissão de pacotes o protocolo TCP/IP, comumente utilizado nas redes locais e Internet [3].

- Ferramentas de Comunicação - proporciona as ferramentas para a utilização da computação paralela. Pode-se ressaltar o PVM (Máquina Virtual Paralela) para a execução de programas paralelos interligando vários nós e formando um ambiente transparente, como uma única máquina virtual. Possui também versões do MPI (*Message Passing Interface*) que permite a troca de mensagens na programação paralela . As versões nativas do MPI no *Rocks* é o MPICH e OPENMPI, porém apresenta-se neste trabalho o OPENMPI [14].

  Alguns destes itens relacionados acima estão demonstrados na Figura 2.8, que apresenta a visão lógica do *Cluster Rocks*, sendo detalhados ao longo deste trabalho.

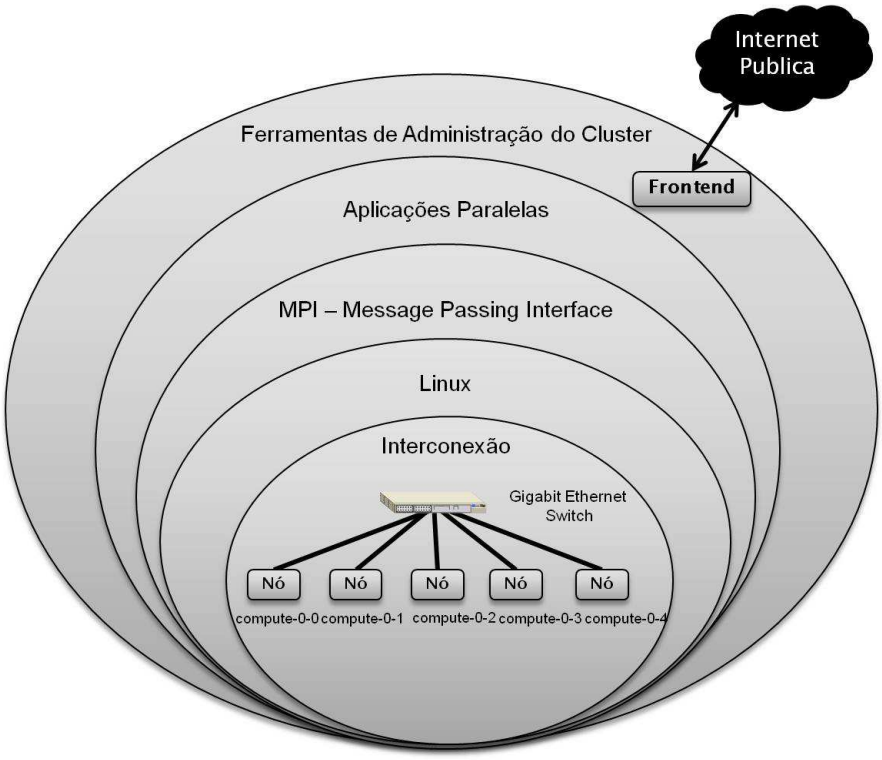

Figura 2.8: Visão lógica do *Cluster Rocks*.

O próximo capítulo apresenta com mais detalhes as características e o descritivo de implantação de um *Cluster Rocks*.

Capítulo 3

Implantação do Cluster Beowulf

É abordado neste capítulo a instalação do *Rocks*, especificamente, para a implantação do *Cluster* denominado VORTEX, que se encontra no Laboratório de D3 do PPG-MCCT - Programa de Pós Graduação de Modelagem Computacional em Ciência e Tecnologia da EEIMVR - Escola de Engenharia, Industrial e Metalúrgica de Volta Redonda da Universidade Federal Fluminense, observando alguns aspectos de *hardware*, *software* e rede de comunicação que serão instalados e utilizados.

3.1 *Hardware*

O *hardware* utilizado para a implementação é composto de seis computadores, sendo um *frontend*, denominado *engenhariavr* e os demais nós denominados *compute-0-0*, em sequencia até o *compute-0-4*. O *frontend* possui uma configuração um pouco melhor do que os nós, em velocidade de processamento, porém todas os outros nós possuem a mesma configuração. Teoricamente este *Cluster* não é heterogêneo mas na prática esta diferença de configuração não é significativa para diferenças de desempenho, visto que o *frontend* recebe maior carga de trabalho.

Hardware do Frontend:

- Processador: Intel Core I7-2600 (8M Cache L3, 3.4 Ghz)
- Placa Mãe: Asus P8h61-M LE
- Memória RAM: 8 GB DDR3 1333MHz,

- Disco Rígido: HD Sata 1 TB - Samsung HD103SJ
- Gravadora de DVD: DVD RW Sata AD-7260S
- Placa de Vídeo: GeForce GT 440 - ZOTAC NVIDIA
- Monitor: 20 polegadas, led, modelo LS20A300 Sansung

Hardware do demais Nós:

- Processador: Intel Core I7-860 (8M Cache L3, 2.8 Ghz)
- Placa Mãe: Intel
- Memória RAM: 8 GB DDR3 1333MHz,
- Disco Rígido: HD Sata 1 TB - Samsung HD103SJ
- Gravadora de DVD: DVD RW Sata TssTCorp
- Placa de Vídeo: GeForce GT 440 - ZOTAC NVIDIA

Nesta instalação em particular utiliza-se um *switch gigabit* de 16 portas, com taxas de transmissão suportados de 10/100/1000Mbps, modelo *Baseline 2816 - 3CBLUG16A* da fabricante 3COM. Entende-se como taxa de transmissão a transmissão de dados entre enlaces ou sistemas finais de comunicação, medida em bps - bits por segundo [15]. Segue as especificações do *switch*:

- Padrões suportados: *Ethernet* 10Base-T, *Ethernet* 100Base-TX, *Ethernet* 1000BaseTX
- Modos de comunicação: *Half-Duplex*, *Full-Duplex*
- Taxa de transferência de dados: 1 Gbps
- Capacidade de comutação: 32 Gbps

A ligação elétrica efetuada com cinco *nobreaks* do modelo Back-UPS RS 1500 do fabricante APC, com potência de 1500 Watts e tensão de 110 Volts.

O *Cluster* foi montado em sala climatizada com ar condicionado, boa iluminação e baixa circulação de pessoas. Colocamos as máquinas em uma estante de aço com os nós

enfileirados e identificados. Para uma melhor administração do *frontend*, o mesmo foi instalado em uma mesa de escritório próximo aos nós, conforme observa-se nas Figuras 3.1 e 3.2. Recomenda-se a instalação de *Clusters* em *racks* devido a sua segurança, confiabilidade e facilidade de inclusão de novos nós.

Figura 3.1: Foto 1 do *Cluster* VORTEX.

Figura 3.2: Foto 2 do *Cluster* VORTEX.

3.2 Infra Estrutura de Rede

Para entendimento da Infra Estrutura de Rede alguns conceitos necessitam ser abordados:

Uma interface de rede, também chamada de adaptador ou *NIC - Network Interface Card*, é uma placa ou cartão constituído de chips, memória RAM e ROM, interface de barramento eletrônico e interface de enlace, utilizada para efetuar o enlace físico ou conexão com a rede. Por padrão os sistemas operacionais Linux identificam estas interfaces como eth0, eth1, eth2 e assim por diante [15].

Um adaptador ou interface de rede possui um endereço de camada de enlace denominado endereço de LAN, endereço físico ou mais comumente chamado endereço *MAC - Media Access Control*. O endereço MAC possui 6 bytes de comprimento expressos como um par de números hexadecimais, separados por um traço a cada byte (exemplo: 0F-CF-6A-87-D1-42). Este endereço é permanente e univoco sendo gravado na memória ROM do adaptador no processo de fabricação [15].

A infra estrutura de comunicação em rede foi montada conforme diagrama, representado na Figura 3.3. Este diagrama representa a ligação do *frontend* em relação aos nós. O *frontend* possui duas interfaces de rede eth0 e eth1. A eth1 é utilizada para a comunicação com a Internet ou rede pública externa. Possui grande importância na utilização e execução de códigos computacionais remotamente. A outra interface eth0 é interligada com o *switch* da rede privada do *Cluster*, sendo sua comunicação direta com os nós. Os nós se conectam ao *switch* através da eth0, onde formam com o *frontend* uma rede privada, somente acessível ao *Cluster*. Os nós possuem acesso aos serviços da Internet, como atualizações e upgrades através do *frontend* que gerencia a conexão.

Os cabos para a conexão dos computadores são do tipo par de fios trançado sem blindagem (*UTP - Unshielded Twisted Pair*), Cat5e, utilizado em redes *LAN* na faixa de transmissão de dados de 10 Mbps a 1 Gbps, sendo seu tamanho padronizado em 1,5 metros e conectores padrão RJ45 no padrão EIA/TIA 568A [15].

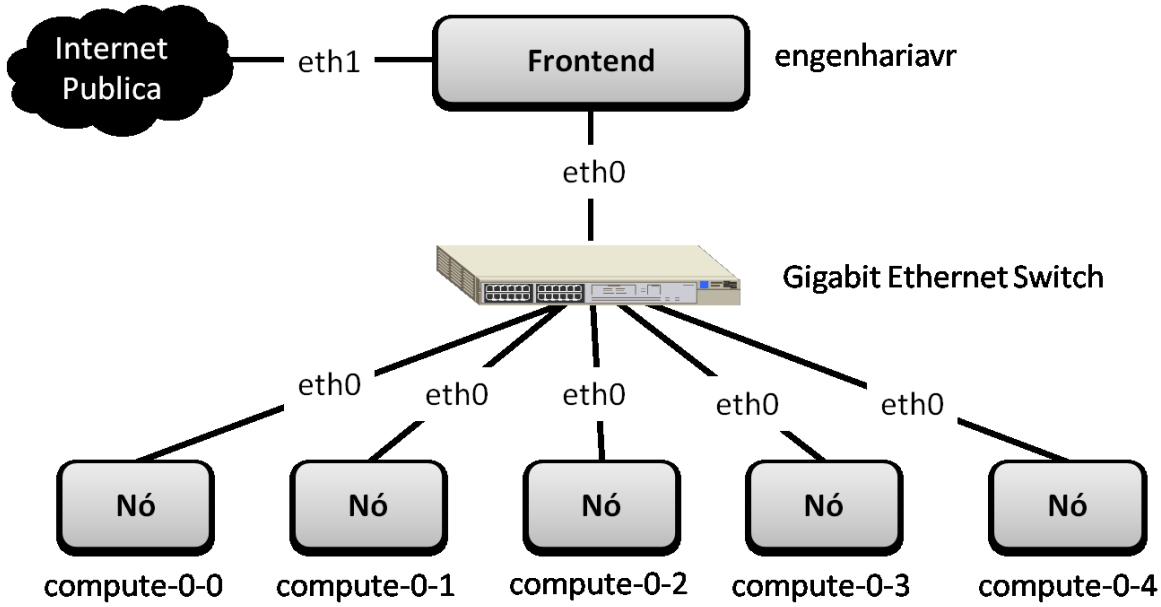

Figura 3.3: Modelo de conectividade do *Cluster* VORTEX [13].

3.3 *Software*

Nesta seção discrimina-se o sistema operacional e os softwares que compreendem a versão do *Rocks Toolkit 5.4.3* (Viper) e suas funcionalidades e especificidades. Os aplicativos e sistema operacional estão agrupados em *Rolls*, relacionados abaixo.

Roll Base de Dados e _Kernel_ de Inicialização

Sistema Gerencial do *Cluster*: O *Rocks* 5.4.3 compreende uma série de comandos básicos de gerenciamento e instalação do *Cluster* [12].

***Roll* OS**

Sistema Operacional: O sistema operacional Linux nativo do *Toolkit* e o CentOS 5.6 baseado em Red Hat Enterprise Linux, porém o *Rocks* permite a escolha de outra distribuição baseada em RHEL 5 *update* 4, devendo-se fornecer todos os pacotes de instalação [12].

HPC

HPC é um conjunto de aplicativos instalados em um *Roll* para fornecer ferramentas para execução de aplicações paralelas. Neste *Roll* inclui os aplicativos *MPI - Message Passing Interface* (OPENMPI, MPICH, MPICH2), *PVM - Parallel Virtual Machine* e *benchmarks (Stream, iperf, IOZone)* [16].

Área 51

Este *roll* é responsável pela segurança do *Cluster*, instalando dois aplicativos: o

Tripware que consiste em um *software* de segurança e integridade de dados que verifica a alteração de arquivos específicos do sistema, sendo seu código fonte aberto. Também disponibiliza o *Chkrootkit* que identifica infecções por um *rootkit* [17]

Bio

Contém uma coleção de ferramentas de bio-informática que são geralmente utilizadas por várias comunidades de pequisa no campo da biologia. Utiliza-se em técnicas matemáticas aplicadas, informática, estatística e ciência da computação [18].

Condor

O Condor, que apartir de 2012 passou a se chamar HTCondor, oferece ao *Cluster Rocks* um ambiente computacional com altas taxas de transferência, efetuando o gerenciamento de carga, mecanismos de gerenciamento de filas, agendamento, regime de prioridade, monitoramento e gerenciamento de recursos. Seu funcionamento se assemelha a um sistema de filas de lotes [19].

Ganglia

É um sistema de monitoramento de *Cluster* que gera gráficos e monitora as atividades dos nós e núcleos dos processadores e sua utilização em processos paralelos. Utiliza a tecnologia XML, XDR e RRDtoll para a representação, compactação, armazenamento e visualização dos dados e gráficos [20]. A Figura 3.4 mostra um exemplo de relatório de carga dos nós do *Cluster* VORTEX, gerado com o Ganglia. Cada gráfico demonstra a carga de processamento de cada nó do *Cluster* em relação ao tempo decorrido.

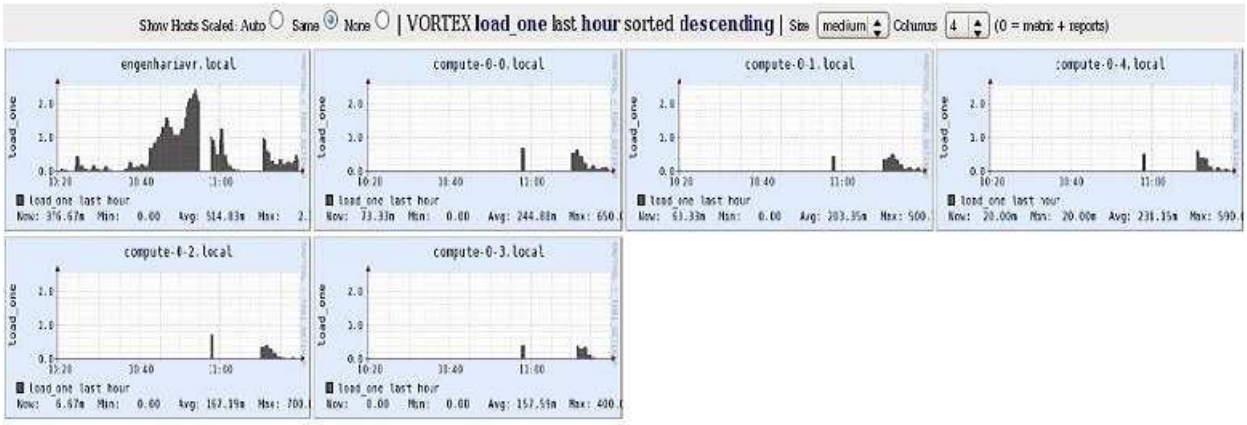

Figura 3.4: Exemplo de relatório do ganglia.

Perl

O *Roll* Perl instala o Perl RPM - *Red Hat Package Manager*, sendo o Perl 5.14.1 uma linguagem de programação amplamente utilizada, caracterizada por trabalhar com projetos de missão crítica, código de alta qualidade e sendo facilmente extensível através de módulos de códigos aberto *CPAN - Comprehensive Perl Archive Network*. Pode-se entender o RPM como um sistema de gerenciamento de pacotes de softwares, utilizado para instalar, desinstalar, atualizar, procurar e verificar pacotes de softwares. Portanto o Perl RPM da suporte ao CPAN e seus módulos. Ele é responsável por construir e criar os RPMs do *Cluster Rocks* [21].

Python

Este Roll instala a linguagem de programação Python 2.7.2, que compila e instala também o Numpy, e o Python 3.2.1. Python é uma linguagem de programação potente e dinâmica utilizada em uma grande variedade de aplicações. Ela é comparada ao Perl, Ruby, Java entre outras [22].

SGE

O *Roll* SGE instala e configura o *software Sun Grid Engine*, que agora chama-se *Oracle Grid Engine* devido a incorporação da *Sun* pela empresa *Oracle*, é uma solução que gerencia a carga de tarefas dentro de um *Cluster*, oferecendo uma grande escalabilidade e adaptação a vários ambientes de trabalho, incluindo computação em nuvem [23].

Torque

Assim como o SGE ele também gerencia os recursos e a carga de tarefas no *Cluster*, oferecendo uma alternativa ao SGE, sendo que o *frontend* somente poderá trabalhar com um dos dois sistemas instalados [24].

Xen

O projeto Xen faz parte da *Linux Fundation*, é desenvolve tecnologias de virtualização em código aberto [25]. O *Roll* Xen instala e configura as máquinas virtuais do *Cluster*, suportando também a construção de *Cluster*s virtuais. Pode-se criar vários grupos de nós que são alocados em máquinas físicas e através de uma interface no *frontend* ou servidor elas são administradas [26]. A comunicação entre nós é realizada de forma independente através do encapsulamento em *VLANs - Virtual Local Area Network*, entendida como uma rede virtual local logicamente independente, separando a topologia lógica das LANs de sua topologia física [27].

3.4 Implantação do *Cluster* VORTEX

Para a instalação mínima do *Cluster* pode-se utilizar vários métodos, como a instalação com mídias ou via rede. Neste exemplo a instalação será feita através de Cds com os *Rolls*. Pode-se obter os *Rolls* de instalação do *Cluster Rocks* 5.4.3 através do site *www.rocksclusters.org*[13], podendo ser baixado em um DVD chamado jumbo ou em Cds contendo:

- Cd *Roll Boot / Kernel*
- Cd *Roll Base*
- Cd *Roll* Sistema Operacional - *Disk 1*
- Cd *Roll* Sistema Operacional - *Disk 2*
- Cd *Web Server*

Antes da instalação devemos ter algumas informações do novo *Cluster*, no caso deste trabalho os dados são do *Cluster* VORTEX:

- Nome de Computador: engenharia.uff.br
- Nome de *Cluster*: VORTEX
- Organização: UFF - Universidade Federal Fluminense
- Localização: EEIMVR / Volta Redonda
- Estado: Rio de Janeiro
- País: BR
- Contato:
- URL: www.uff.br
- Latitude/Longitude: S22.51 W44.10

Abaixo as informações sobre as interfaces de rede do *frontend*. A partir destas informações será configurada o acesso a rede pública na interface eth1 e a rede privada na interface eth0.
Configuração Interface eth0:

- Endereço IP: XX.X.X.X
- Mascara: 255.0.0.0

Configuração Interface eth1:

- Endereço IP: XX.XX.XX.XXX
- Mascara: 255.255.255.0
- *Gateway*: XX.XX.XX.X
- DNS: XXX.XXX.XXX.XX

Configuração de senha do super usuário *root* e de fuso horário:

- Fuso Horário: América/São Paulo
- Servidor NTP: a.ntp.br

O Apêndice A descreve passo a passo e com mais detalhes esta instalação.

3.4.1 Problemas, Soluções e Observações da Implantação do *Cluster* VORTEX

Reconhecimento e Re-Instalação de Nós

Para a instalação dos nós do *Cluster* VORTEX utiliza-se o comando para reconhecimento automático e consequentemente instalação dos nós. Uma observação sobre este aplicativo *"Insert Ethernet Addresses"*, no momento da instalação ele reconhece os novos nós, através do nome e endereço MAC da interface de rede de cada nó, fornecendo endereço IP, porém ele aguarda a instalação do *Rocks*, ficando sua marcação desta forma -( ). Como a opção foi instalar através de Cds, efetuou-se o *boot* em cada nó através do leitor de cd e assim efetuar a instalação do *Rocks*. Somente após este processo o *frontend* reconhece a instalação e confirma em seu banco de dados, alterando sua identificação para - (*).

Uma observação importante: Quando interrompe-se o processo de *"Insert Ethernet Addresses"* ou reinstala-se e reconhece novamente um nó que já foi cadastrado na listagem de nós do *frontend*, deve-se proceder desta maneira:

1. Verificar primeiramente na listagem de *hosts* se o nó em questão consta desta listagem. Pode-se observar através do MAC da interface de rede.
   # Rocks list host

2. Caso esteja preenchida na listagem, retira-se o nó desejado.
   #insert-ethers –remove="nome do nó"

3. Iniciar o reconhecimento novamente do nó, que deve estar neste momento ligado, conectado a rede e com o Cd do *Roll Boot* no leitor.
   # insert-ethers (opção Computer)

4. Após a instalação do *Rocks* aparecerá na tela do *"Insert Appliances"* o MAC do nó, nome e a confirmação da instalação representada pelo - (*).

Instalação da Placa de Vídeo *Off Board*

Para instalação do controlador do dispositivo de vídeo (*driver* de vídeo) no sistema operacional pode-se efetuar o *download* diretamente da página web do fabricante o *software* de instalação que vai automaticamente efetuar a instalação. Se não houver sucesso neste processo ou o fabricante não disponibilizar a atualização automática, segue-se alguns passos para a instalação manualmente:

1. Efetua-se primeiramente a atualização dos pacotes do Sistema Operacional. Para isso utilizamos o comando *yum*.
   # yum update

2. Caso esteja na interface gráfica é necessário sair deste modo, com o comando:
   # init3

3. Logo após executa-se o comando de instalação e configuração do dispositivo de vídeo.
   # system-config-display

4. Para visualizar a instalação e consequentemente as alterações reinicializamos a interface gráfica.
   # init5

Substituição de Placa de Interface de Rede

Para substituição de uma placa de interface de rede do *frontend*, neste caso a interface eth0 que efetua a conexão do *frontend* com a rede privada, ou seja, os nós do *Cluster*, é necessário que as configurações da interface e o MAC sejam refeitas, tanto no SO quanto nas configurações do *Rocks*. Após a troca física da placa de rede, deve-se seguir os seguintes passos:

1. Efetua-se a atualização dos pacotes do Sistema Operacional.
   # yum update

2. Após a configuração execute no terminal do *frontend* a comando de configuração das interfaces de rede.
   # system-config-network

3. Uma caixa de texto será aberta. Marque a interface eth0 e click no botão *edit eth0*. Nesta janela configura-se o endereço IP, máscara de rede e *gateway* conforme informado no item 3.4. Pressione o botão OK para que as configurações possam ser atualizadas.

4. Efetuar a reinicialização da interface.
   # service network restart

5. O próximo passo é reconfigurar o arquivo de interfaces de rede no sistema *Rocks*. Necessita-se do MAC da nova interface de rede.
   # Rocks set host interface iface engenhariavr iface=eth0 mac="mac da interface de rede"

6. Deve-se verificar se a interface de rede foi reconfigurada com o novo MAC.
   # Rocks list host interface engenhariavr
   O retorno do comando deve ser de acordo com a Figura 3.5:

```
[root@engenhariavr ~]# rocks list host interface engenhariavr
SUBNET  IFACE MAC               IP            NETMASK
MODULE NAME         VLAN OPTIONS CHANNEL
private eth0  00:E0:4C:39:0F:2A 10.1.1.1      255.255.0.0
------ engenhariavr ---- ------- -------
public  eth1  F4:6D:04:EF:39:69 10.11.11.136 255.255.255.0
------ engenhariavr ---- ------- -------
```

Figura 3.5: Configurações das interfaces de rede no *Rocks*.

Removendo a Virtualização das Placas de Rede

O *Rocks* possui um pacote de virtualização das interfaces de rede, utilizado pelo Roll

Xen para a virtualização do *Cluster*. Esta virtualização pode ser útil na criação de várias interfaces virtuais, visando a criação de redes virtuais. Porém, neste caso, quando efetuado a atualização dos pacotes do *Cluster*, foi instalado o grupo *Virtualization Support*, nas interfaces dos nós e do *frontend*. Este grupo instala as interfaces virtuais, gerando alguns conflitos ao executar códigos computacionais paralelos.

Para verificar se esta funcionalidade esta ativada utiliza-se o comando de verificação das configurações de rede do Sistema Operacional Linux:

 # ifconfig

Como resultado, as configurações, conforme Figura 3.6:

```
eth0      Link encap:Ethernet  HWaddr 00:27:0E:0A:3A:29
          inet addr:10.1.255.254  Bcast:10.1.255.255
Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:204 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1534 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:24867 (24.2 KiB)  TX bytes:205892 (201.0 KiB)
          Interrupt:233 Memory:fb100000-fb120000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:53 errors:0 dropped:0 overruns:0 frame:0
          TX packets:53 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:5274 (5.1 KiB)  TX bytes:5274 (5.1 KiB)

virbr0    Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          inet addr:192.168.122.1  Bcast:192.168.122.255
Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

Figura 3.6: Configurações das interfaces de rede virtualizadas.

Verifica-se a presença da interface *virbr*, intui-se que a virtualização esta ativada. Descreve-se a seguir as etapas de desinstalação da virtualização de interfaces de redes:

1. Primeiramente deve-se acessar cada nó da rede remotamente através do comando *ssh*.

   # ssh "nome do nó"

   Exemplo: *# ssh compute-0-0*

2. Após o acesso como root ao nó do *Cluster*, efetua-se a desinstalação do grupo *virtualization* com o comando *yum*:
   # yum groupremove "virtualization"

3. Após a remoção de todo o pacote e de suas dependências confirma-se a remoção da interface virtual *virbr*:
   # ifconfig

No próximo capítulo é apresentado as rotinas de troca de mensagens utilizando a biblioteca MPI, juntamente com a implementação nos códigos computacionais.

Capítulo 4

MPI - Message Passing Interface

Uma das características de um *Cluster* Beowulf é o método de troca de mensagens, como observado na secção 2.4, principalmente por se tratar de computadores paralelos escalonáveis com memória distribuída. Pode-se inferir que o conceito básico de troca de mensagens refere-se a finalidade de comunicação entre processos por meio de mensagens, sendo um conceito bem compreendido apesar de existirem muitas variações, porém na prática este sistema pode ser implementado de forma eficiente e portátil. Entende-se como portabilidade a capacidade de execução dos códigos de transmissão de mensagens em uma grande variedade de máquinas e arquiteturas heterogenias.

Defini-se o modelo de passagem de mensagens (MP -*Message Passing*) por um conjunto de processos que possuem acesso a memória local e se comunicam com outros processos, efetuando operações em ambos os processos para envio e recebimento de dados da memória local de um processo para outro. A grande vantagem, dentre outras, da utilização deste modelo de passagem de mensagens em ambientes de computação paralela é o desempenho. Com o avanço e modernização das CPUs se tornando mais rápidas, gerenciamento da memória cache e da hierarquia de memória em geral, o MP proporciona ao programador manipular dados específicos dos processos permitindo um otimização do *hardware* [28].

4.1 Conceitos do MPI

O MPI - *Message Passing Interface* pode ser entendido como uma biblioteca, especificando nomes, sequencias de chamadas, e os resultados das sub-rotinas a serem chamadas de códigos computacionais em Fortran, C e C++. Os códigos computacionais paralelos implementados nestas linguagens são compilados com compiladores comuns porém ligados

com a biblioteca MPI, sendo detalhado melhor posteriormente neste mesmo Capítulo. As bibliotecas MPI devem conter [29]:

- Rotinas de Gerência de Processos, que permite estabelecer processos, gerenciar, finalizar, possibilitar a identificação do processo através de seu ID e a terminação do número de processos estabelecidos na execução do código computacional
- Rotinas de Comunicação, que estabelece a comunicação entre processos ou seja o envio e recebimento de mensagens, com seus parâmetros previamente discriminados.
- Rotinas de Comunicação em Grupo, sendo determinante para sincronização de processos e a coordenação de um ou mais grupos de processos.

4.1.1 Evolução do MPI

A evolução do MPI se fez necessário para tornar-se um padrão de portabilidade, pois existiam dificuldades na transferência de mensagens de um sistema para outro com o mínimo de falhas. Neste tópico enumera-se alguns pontos evolutivos do padrão MPI [1].

- No final da década de 80 existia uma grande dificuldade com as ferramentas de desenvolvimento de programas em ambientes paralelos, principalmente com a portabilidade de bibliotecas de comunicação. Também haviam problemas de funcionalidades e preço, necessitando o desenvolvimento de um padrão.
- Em 1992 foi instituído o Fórum MPI para a definição de um padrão portátil para a troca de mensagens, uma forma totalmente aberta com a participação de empresas fabricantes de computadores e *softwares*, universidades e cientistas, com prazo de conclusão de 1 ano.
- Em Maio de 1994 o MPI*Standard* foi concluído e publicado como domínio público.
- Entre 1995 e 1997 o Fórum MPI reuniu-se para incluir operações adicionais de memória remota, gerenciamento de projeto dinâmico, entre outras funcionalidades.

4.2 O OPENMPI

O OPENMPI (*Open Source High Performance Computing*) possibilita ao *frontend* e aos nós do *Cluster* a comunicação entre todos os seus processos e assim a troca de

mensagens. Este padrão foi desenvolvido pela união das principais funcionalidades de quatro implementações de código aberto MPI: MPI LAM / MPI, LA / MPI (*Los Alamos* MPI) e FT-MPI (*Fault-Tolerant* MPI), gerando assim o OPENMPI [30]. Para que se possa utilizar o OPENMPI, são necessárias algumas linguagens, como os compiladores Fortran (Formula Translation) e C, assim como um sistema operacional, sendo aqui utilizada uma distribuição Linux, conhecida como CentOS.

4.2.1 Linguagem Fortran

Fortran é considerada a primeira linguagem científica de alto nível para computadores. Foi idealizada e desenvolvida pelo engenheiro John Warner Backus, que escreveu uma carta a seu chefe na IBM, propondo pesquisar uma linguagem de alto nível, com o intuito traduzir equações científicas para códigos de computadores. John Backus formou uma equipe composta por Irving Ziller, Harlan Herrick e Robert A Nelson para começar o desenvolvimento. No inicio, os algoritmos eram escritos como equações algébricas para logo depois o Fortran transcrever o código em linguagem de máquina. No inicio o Fortran não possuía muitos recursos, e conforme novos compiladores foram aparecendo, notou-se a necessidade de introduzir atualizações.

Conforme o tempo foi passando, muitas versões foram sendo geradas, entre elas e de grande melhoramento foi os chamados Gfortran (compilador livre Fortran 95/2003/2008) e o HPF (*High Performance Fortran*), uma extensão do Fortran 90, que permitia ter a vantagem de trabalhar com arquiteturas de processamento paralelo. Este último, deu novas características para a linguagem, revelando superioridade em relação aos seus competidores, nas áreas científicas e de aplicações de engenharia, garantindo sua continuidade com sucesso por muito tempo.

Versões do Fortan (*Release date*)

Fortran (1957) => Fortran II (1958) => Fortran IV (1961) => Fortran 66 (1966) => Fortran 77 (1977) => Fortran 90 (1992) => Fortran 95 (1996) => Fortran 2003 (2004) => Fortran 2008 (2010).

4.2.2 Linguagem C

Foi desenvolvida a partir das linguagens B e BCPL, por volta de 1972 por Dennis Ritchie e se tornou difundida como a linguagem de desenvolvimento do sistema operacional UNIX. A linguagem C evoluiu muito no final da década de 70 chegando ao C que

hoje conhecemos como tradicional. Por independer do *hardware* utilizado e por sua portabilidade para a maioria dos computadores o C teve uma expansão muito grande levou a muitas variantes da linguagem, similares, porém com algumas incompatibilidades. Em 1989 um único padrão foi aprovado através de um comitê técnico – X3J11 sob o *American National Standards Committee on Computers em Information Processing.* Hoje a linguagem C é amplamente difundida e disponível, independente de *hardware*, sendo suas aplicações executadas em uma grande variedade de sistemas computacionais com poucas ou nenhuma modificação [31].

Compilador GCC

Utilizamos em nosso trabalho o GCC – *GNU Compiler Collection* que inclui em seu pacote copiladores C e Fortran. Foi escrito como compilador para o sistema operacional GNU, sendo software livre em sua totalidade, regido sob os termos da *GNU Free Documentation License* [32].

4.2.3 Identificadores e Rotinas do OPENMPI

Para um melhor entendimento do funcionamento do OPENMPI é necessário conhecer alguns conceitos básicos e elementos de funcionamento.

- Processo: cada cópia do código computacional executado é chamado de processo. Os processos podem ser executados em uma única máquina ou em várias.
- *Rank*: todo o processo tem uma identificação única, chamada de *rank*, atribuída pelo sistema quando o processo é inicializado. Essa identificação é contínua representada por um número inteiro, começando de zero até *N-1*, onde *N* é o número de processos.
- Grupos: é um conjunto ordenado de *N* processos.
  A Figura 4.1 representa os grupos de processos e seus identificadores.

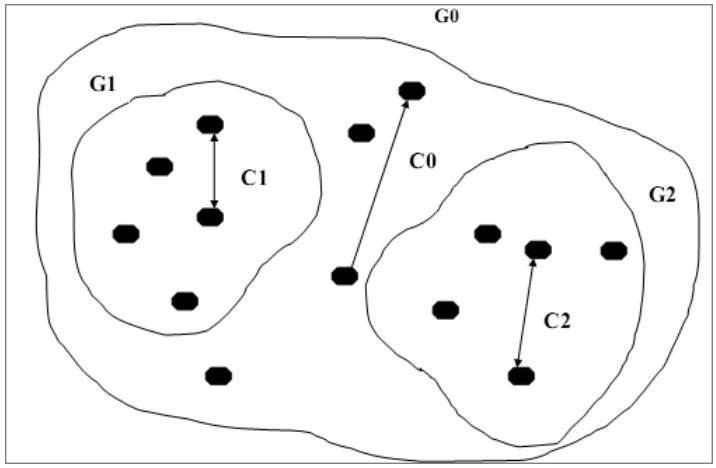

Figura 4.1: Grupos de processos e seus identificadores[29].

- Comunicador: O comunicador é um objeto local que representa o domínio (contexto) de uma comunicação (conjunto de processos que possuem permissão de comunicação). O *MPI_ COMM_ WORLD* é o comunicador predefinido que permite todos os processos se comunicarem dentro deste grupo

4.2.3.1 Rotinas de Gerência de Processos

As rotinas de gerenciamento de processo incluem os processos de iniciar, finalizar, determinar o número de processos, identificar processos e o nome do nó que esta realizando o processo.

Para iniciar o código computacional utilizando o MPI necessita-se declarar o arquivo de cabeçalho *mpi.h* onde se encontra constantes utilizadas pelo MPI e demais estruturas de dados [1]. Trata-se neste trabalho de comandos no formato do compilador C.
Abaixo verifica-se algumas destas rotinas de gerenciamento de processo [1]:

- *MPI_ Init(&argc, &argv);* -> Inicializa um processo MPI. Ele sincroniza todos os processos na inicialização de uma aplicação MPI. Os argumentos *argc* e *argv* são argumentos de linha de comando do programa C.

- *MPI_ Comm_ size(MPI_ COMM_ WORLD, &noprocessos);* -> Retorna o número de processos dentro de um grupo em execução.

- *MPI_ Comm_ rank(MPI_ COMM_ WORLD, &IDprocesso);* -> Identifica o número do processo MPI dentro de um determinado grupo, também conhecido como *rank*, iniciando pelo *rank 0*.

- *MPI_ Get_ processor_ name(computerName, &nome);* -> Retorna o nome do computador cujo processo está sendo executado.
- *MPI_ Finalize();* -> Finaliza um processo MPI, devendo ser colocado no final do código computacional.

4.2.3.2 Rotinas de Comunicação Ponto a Ponto

A passagem de mensagens entre processos ponto a ponto é o método de comunicação básica num sistema de memória distribuída. Uma mensagem contém duas partes: dado a ser transmitido e o envelope com informações da rota dos dados [1], representado pela Figura 4.2.

Dados: dado (variável, vetor ou matriz), número de elementos e tipo do dado.
Envelope: processo origem, processo destino, rótulo da mensagem e comunicador.

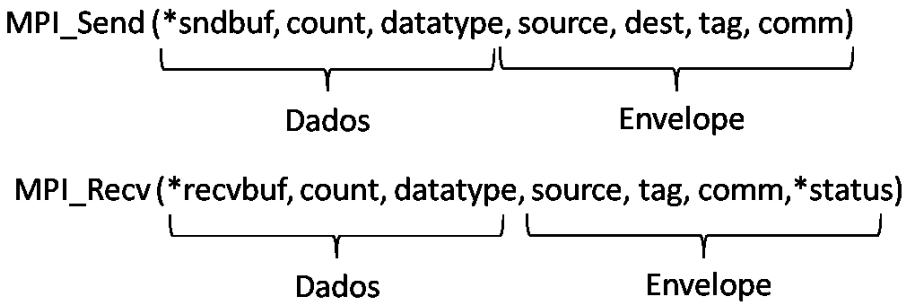

Figura 4.2: Rotina de comunicação ponto a ponto *MPI_ Send* e *MPI_ Recv*.

- *MPI_ Send (*sndbuf, count, datatype, dest, tag, comm)* -> Envia o dado e finaliza quando a memória de envio estiver pronto para ser reutilizado. A rotina só retorna após o dado ter sido enviado.

  onde:
  sndbuf -> endereço inicial da variável a ser enviada.
  count -> número de elementos ou posições a partir do endereço inicial.
  datatype -> tipo de dado manipulado.
  dest -> identificação do destino da mensagem.
  tag -> identificação da mensagem (numérico) ou *MPI_ ANY_ TAG*.
  comm -> identificação do comunicador.

- *MPI_ Recv (*recvbuf, count, datatype, source, tag, comm,*status)* -> Recebe o dado e finaliza quando a memória de recebimento estiver pronto para ser reutilizado. A rotina retorna após o dado ter sido recebido e armazenado.

  onde:
  recvbuf -> endereço inicial da variável a ser recebida.
  count -> número de elementos ou posições a partir do endereço inicial.
  datatype -> tipo de dado manipulado.
  source -> identificação da origem da mensagem ou *MPI_ ANY_ SOURCE* para a identificação de qualquer processo.
  tag -> identificação da mensagem (numérico) ou *MPI_ ANY_ TAG.*
  comm -> identificação do comunicador.
  status ->retorna a situação da execução.

  A tabela 4.1 lista os principais tipos de dados utilizados em rotinas MPI no compilador C.

Tipos de dados no C	
MPI	**C**
MPI_CHAR	*char*
MPI_SHORT	*short int*
MPI_INT	*int*
MPI_FLOAT	*float*
MPI_DOUBLE	*double*
MPI_LONG_DOUBLE	*long double*

Tabela 4.1: Tipos de dados no compilador C

4.2.3.3 Rotinas de Comunicação de Grupos

A utilização destas rotinas tem como objetivo o envio de mensagens entre grupos de processos, envolvendo todos os processos e compartilhando uma parte comum de informação. As rotinas efetuam o bloqueio do processo até receberem a mensagem de resposta ou serem finalizadas. A coordenação das mensagens em grupo são identificadas através do parâmetro comunicador [1].

- *MPI_ Bcast (*buffer, count, datatype, root, comm)* -> envia os dados para todos os

processos do grupo.

onde:
buffer -> endereço inicial da variável a ser enviada.
count -> número de elementos ou posições a partir do endereço inicial.
datatype -> tipo de dado manipulado.
root -> identificação do processo que efetuará o envio da mensagem para todos os processos.
comm -> identificação do comunicador

A Figura 4.3 mostra um exemplo em que todos os 4 processos possuem vetores de 4 posições na qual executam o *MPI_ Bcast*. Esse comando é executado no processo 2 que enviará os dados, no caso a letra A, para a primeira posição do vetor de todos os outros processos, fazendo assim o *broadcast* da informação para todos os processos.

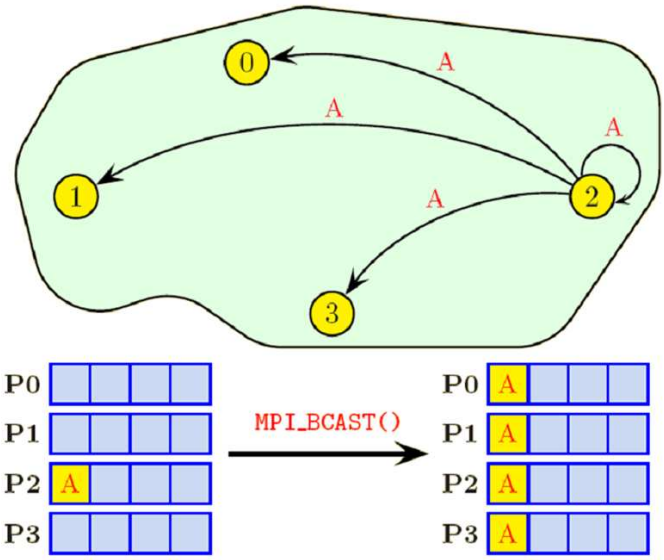

Figura 4.3: Rotina de comunicação em grupo *MPI_ Bcast*[29].

- *MPI_ Barrier (comm)* -> sincroniza os processos de um grupo. .

onde:
comm -> identificação do comunicador

A Figura 4.4 mostra um exemplo em que todos os 4 processos estão em execução porém cada um com um tempo diferente do outro. O *MPI_Barrier* pausa os processos que estejam sendo executados paralelamente com os demais, até que o último processo possa estar sincronizado com os demais e assim continuar a execução.

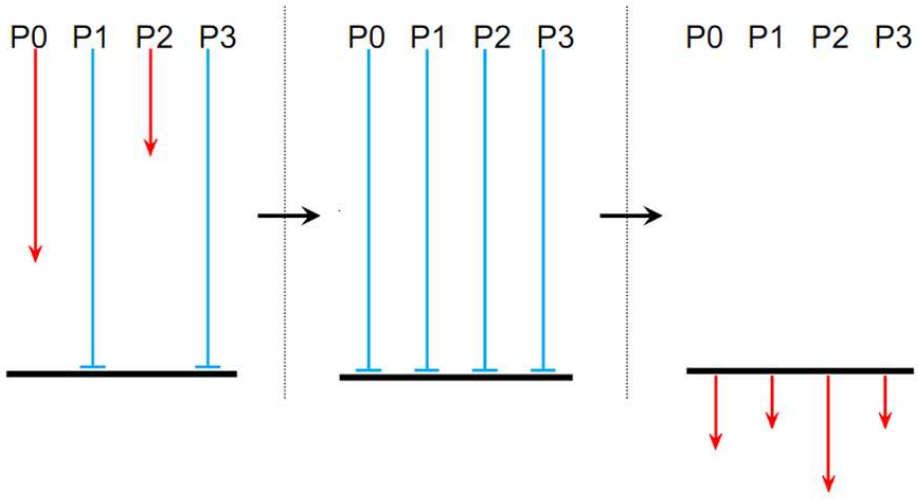

Figura 4.4: Rotina de comunicação em grupo *MPI_Barrier*[29].

- *MPI_Reduce(*sbuf, *rbuf, count, datatype, op, root, comm)* -> recebe dados de todos os processos e efetua a operação descrita na opção *"op"*.

onde:
sbuf -> endereço da variável que participará da operação da redução.
scount -> endereço da variável que receberá o resultado da operação.
count -> número de elementos ou posições a partir do endereço inicial.
datatype -> tipo de dado manipulado.
op -> tipo de operação, conforme tabela abaixo.
root -> identificação do processo que receberá o resultado.
comm -> identificação do comunicador

A Figura 4.5 mostra um exemplo em que 6 processos executam o *MPI_Reduce* com a opção *MPI_PROD*. Todos os processos enviam suas variáveis ao processo 0 que faz o produto de todos os valores, enviando o resultado do produto a todos os outros processos.

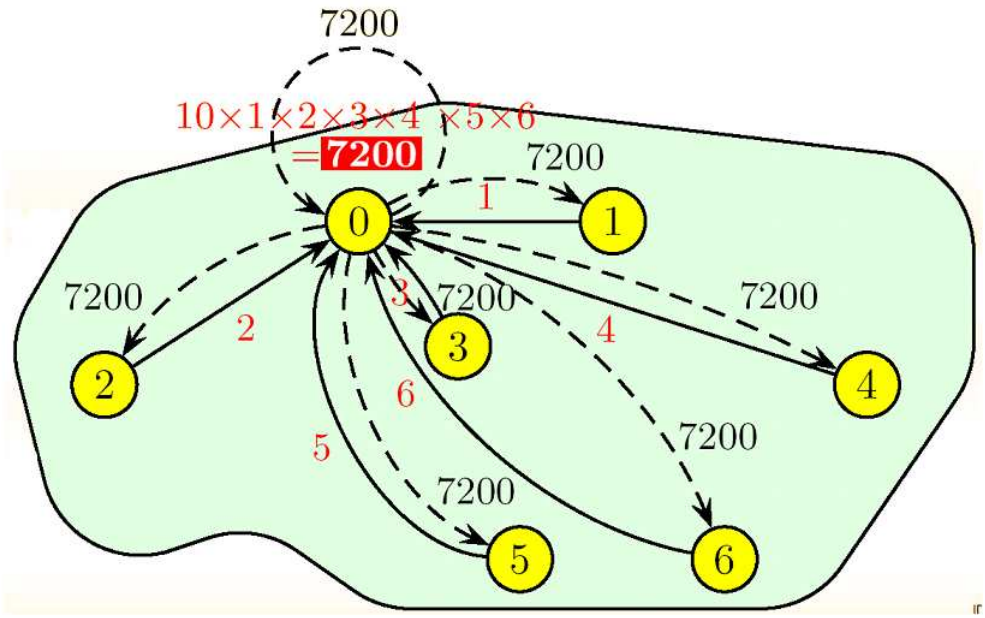

Figura 4.5: Rotina de comunicação em grupo *MPI_Reduce*[29].

Os principais tipos de operação do MPI_Reduce e suas utilizações estão relacionadas na tabela 4.2.

Função	Resultado	Compilador C
MPI_MAX	valor máximo	integer,float
MPI_MIN	valor mínimo	integer,float
MPI_SUM	somatório	integer,float
MPI_PROD	produto	integer,float

Tabela 4.2: Tipos de operações MPI_Reduce

- *MPI_Scatter (*sbuf, scount, datatype, *rbuf, rcount, rtype, root, comm)* -> distribui dados em n seguimentos iguais, tal que n é o número de processos.

onde:

sbuf -> endereço da variável a ser distribuída.

scount -> número de elementos ou posições, em cada processo, que serão distribuí-

dos.

datatype -> tipo de dado manipulado.

rbuf -> endereço onde os dados, em cada processo, serão coletados.

rcount -> número de dados, por processo, que serão coletados.

rtype -> tipo de dado que será coletado.

root -> identificação do processo que receberá o resultado.

comm -> identificação do comunicador.

A Figura 4.6 mostra um exemplo em que o vetor do processo 2 executa o *MPI_ Scatter*, enviando as variáveis contidas em um único vetor, para todos os outros vetores, sempre na primeira posição de cada um deles.

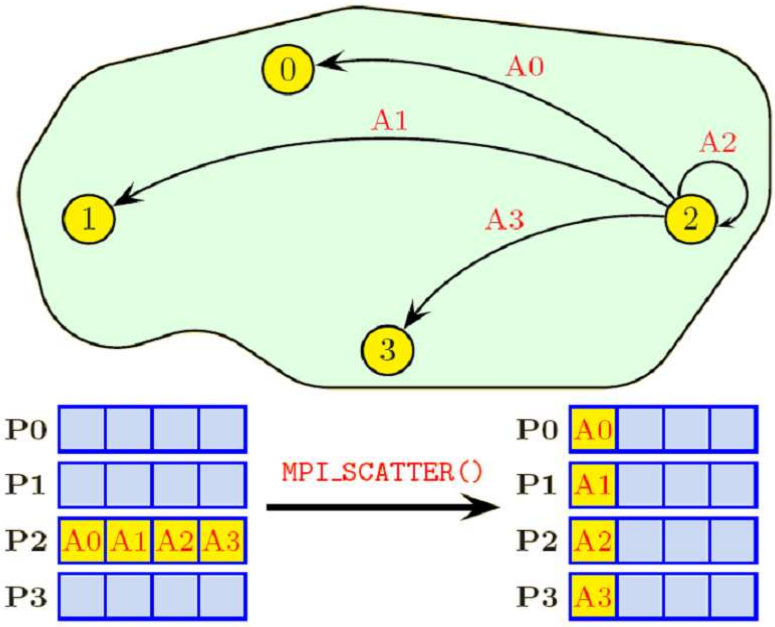

Figura 4.6: Rotina de comunicação em grupo *MPI_ Scatter*[29].

- *MPI_ Gather(*sbuf, scount, datatype, *rbuf, rcount, rtype, root, comm)* -> coleta os dados em n processos do grupo.

onde:

sbuf -> endereço da variável a ser distribuída.

scount -> número de elementos ou posições, em cada processo, que serão distribuí-

dos.

datatype -> tipo de dado manipulado.

rbuf -> endereço onde os dados serão coletados.

rcount -> número de dados que serão coletados.

rtype -> tipo de dado que será coletado.

root -> identificação do processo que coletará os dados.

comm -> identificação do comunicador.

A Figura 4.7 mostra um exemplo em que todos os 4 processos possuem vetores de 4 posições na qual executam o *MPI_ Gather*. Esse comando é executado no processo 2 que recebe os dados, no caso as letra A0, A1, A2 e A3 contidas na primeira posição de cada vetor dos processos P0, P1, P2 e P3, para o vetor do processo 2, recebendo as variáveis A0, A1, A2 e A3 respectivamente nesta ordem.

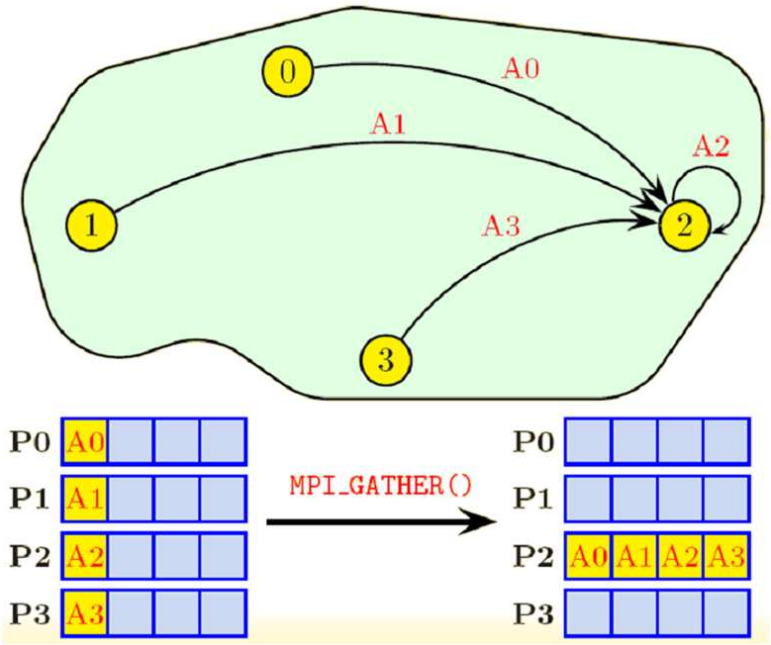

Figura 4.7: Rotina de comunicação em grupo *MPI_ Gather*[29].

- *MPI_ Allgather(*sbuf, scount, datatype, *rbuf, rcount, rtype, comm)* -> os processos coletam os dados de cada processo da aplicação, como se cada processo efetuasse um *brodcast*.

  onde:

sbuf -> endereço da variável a ser enviado.
scount -> número de elementos ou posições que serão enviados.
datatype -> tipo de dado manipulado.
rbuf -> endereço onde os dados serão coletados.
rcount -> número de dados que serão coletados.
rtype -> tipo de dado que será coletado.
comm -> identificação do comunicador.

A Figura 4.8 mostra os 4 processos e cada um deles possui um vetor de 4 posições, com os valores A0, A1, A2 e A3, na primeira posição de cada um deles. Quando da execução desta rotina, os valores são enviados para todos os processos, e os resultados devem ser armazenados nos demais vetores de cada processo. Nota-se que o vetor de cada processo precisa ter o número de posições igual ao número de processos.

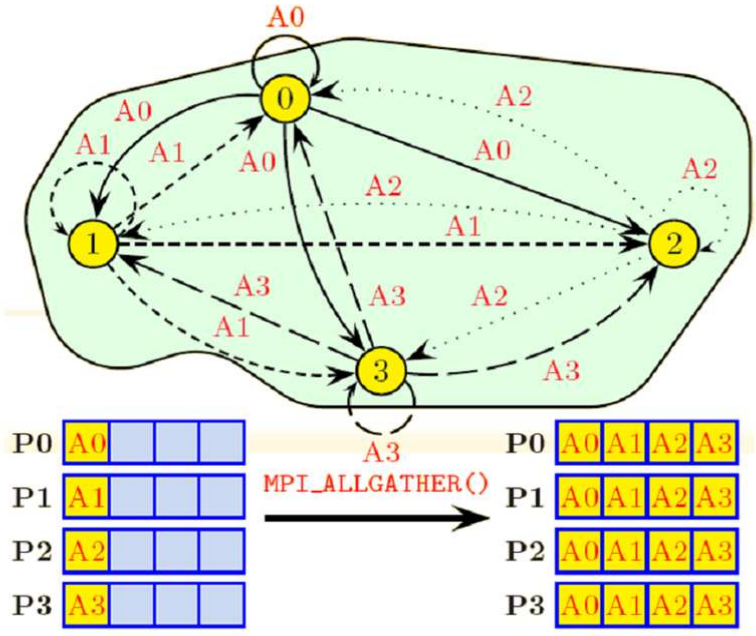

Figura 4.8: Rotina de comunicação em grupo *MPI_ Allgather*[29].

- *MPI_ Alltoall(*sbuf, scount, datatype, *rbuf, rcount, rtype, comm)* -> cada processo envie seus dados para todos os outros processos da aplicação. Seria similar a cada processo efetuar um *scatter*.

onde:

sbuf -> endereço da variável a ser distribuída.

scount -> número de elementos ou posições, que serão distribuídos.

datatype -> tipo de dado manipulado.

rbuf -> endereço onde os dados serão coletados.

rcount -> número de dados que serão coletados

rtype -> tipo de dado que será coletado.

comm -> identificação do comunicador.

A Figura 4.9 mostra os 4 processos com vetores de 4 posições onde cada um deles possui os valores de A0, A1, A2, A3 para o processo 0, B0, B1, B2, B3 para o processo 1, C0, C1, C2 e C3 para o processo 3 e D0, D1, D2 e D3. Ao executar a rotina MPI_Alltoall a primeira posição do vetor de todos os processos foi enviado ao vetor de processo O e assim preencheu todas as posições do seu vetor. Nota-se que o vetor de cada processo precisa ter o número de posições igual ao número de processos.

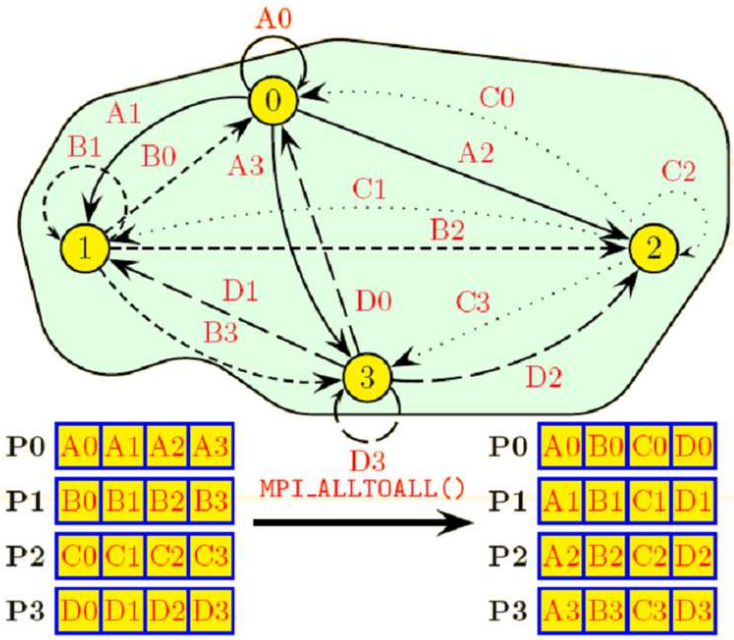

Figura 4.9: Rotina de comunicação em grupo *MPI_ Alltoall*[29].

O próximo capítulo apresenta exemplos de utilização das rotinas de comunicação em códigos computacionais em paralelo e suas comparações com códigos seriais para verificação da correta implementação das rotinas MPI.

Capítulo 5

Implementação de Códigos Computacionais Paralelizados

Neste capítulo é apresentado alguns exemplos de códigos computacionais seriais e paralelos executados no *Cluster* Beowulf VORTEX que foi objeto de implementação neste trabalho. Todos os códigos computacionais foram implementados utilizando a linguagem de programação C, com o compilador GCC 4.1.2, e as bibliotecas do OPENMPI 1.6.4 para paralelização, conforme discutido no Capítulo 4. Os gráficos comparativos foram gerados através do Matlab R2012a. Pode-se definir o *software* Matlab R2012a como um ambiente interativo de linguagem de alto nível para computação numérica, visualização e programação. É utilizado na análise de dados, desenvolvimento de algoritmos, criação e aplicação de modelos [33]. Todos os códigos computacionais utilizados nestes exemplos podem ser observados em sua íntegra no Apêndice B.

Inicialmente os códigos computacionais foram gerados serialmente, de acordo com cada método. Com a execução foi obtido o resultado numérico, tempos de utilização do processador e tempo de execução. Posteriormente foram gerados os códigos em paralelos, aplicando as rotinas do OPENMPI aos códigos seriais e efetuando assim a paralelização do código. Com os resultados obtidos e armazenados em arquivos tipo texto, os gráficos comparativos foram gerados.

Quando executa-se um código serial, com um dado de entrada, um processo é gerado executando cada rotina do código serialmente, gerando uma saída, conhecida como o resultado da execução das rotinas serialmente. Como exemplificado na Figura 5.1, temos uma operação de multiplicação de um *array* B por uma variável K gerando o resultado em um *array* A. Deste modo cada operação de multiplicação deve ser executada serialmente, uma a uma, de acordo com a variável y, que é o índice da instrução *for*.

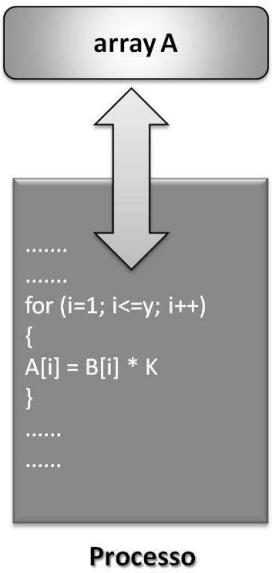

Figura 5.1: Execução de código computacional serial.

Ao paralelizar este código vários processos são gerados, identificados de *0* até *n-1*, onde n é o número de processos de acordo com o proposto pelo programador. Cada processo executa a operação de multiplicação de parte do *array B*, concomitantemente aos outros processos, e retornam, concatenando os resultados no *array A*, explicitado na Figura 5.2

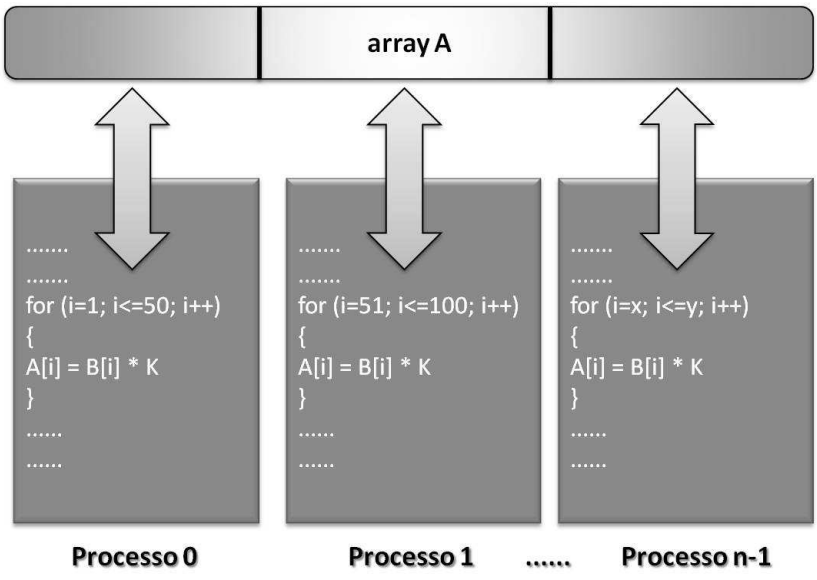

Figura 5.2: Execução de código computacional paralelo.

Com a paralelização objetivamos a diminuição significativa de tempo de execução e processamento em relação ao código serial.

Nos exemplos deste capítulo os códigos computacionais são paralelizados no máximo com 48 processos, pois como citado no Capítulo 3, o *Cluster* VORTEX possui 48 processadores. No caso dos processos excederem o número de processadores verifica-se a concorrência entre processos para o controle do processador e consequentemente gerando um aumento do tempo Real de execução do processo.

Os tempos de execução foram apurados por funções no código computacional chamando funções do *Kernel* do Sistema Operacional, igualmente utilizado nos códigos seriais e paralelos:

- Tempo de CPU -> retorna o tempo de utilização dos recursos dos processadores pelo processo específico. A função *getrusage(RUSAGE_SELF, &usage)*,utilizada no Código Computacional 5.1, retorna as estatísticas de uso dos recursos de CPU demandados para a execução do processo. O campo *ru_utime* retorna o tempo gasto na execução, em microsegundos, sendo convertido em segundos, conforme observado na linha 4 do código computacional [34].

```
1 double cpuTime() {
2   static struct rusage usage;
3   getrusage(RUSAGE_SELF, &usage);
4   return ((double)usage.ru_utime.tv_sec)+(((double)usage.ru_utime.
       tv_usec)/((double)1000000));
5 }
```

Código Computacional 5.1: Tempo de utilização dos recursos dos processadores - cpuTime().

- Tempo Real -> retorna o tempo decorrido da execução código, também chamado de tempo corrido de execução. A função *ftime(&tp)*utilizada no Código Computacional 5.2 retorna a hora atual UTC (Tempo Universal Coordenado) da máquina. Os campos *tp.time* e *tp.millitm* retornam respectivamente o tempo em segundos e milisegundos, sendo convertidos em segundos, conforme observado na linha 5 do código computacional [34].

```
1 double wallClockSeconds() {
2   struct timeb tp;
```

```
3     double sec;
4     ftime(&tp);
5       sec =  ((double)tp.time)+(((double)tp.millitm)/1000.0);
6     return sec;ftime
7   }
```

Código Computacional 5.2: Retorna o tempo total de execução do código - wallClockSeconds().

Desvio Padrão

O desvio padrão é utilizado para a comparação entre conjunto de dados diferentes, determinando a dispersão dos valores em relação a média. O desvio padrão é calculado pela Equação 5.1, definida por [35]:

$$S = \sqrt{\frac{\sum x_i^2 - \frac{1}{n}(\sum x_i)^2}{n-1}} \tag{5.1}$$

onde:

- n -> número de amostras.
- x -> tempo observado na execução do código.

Em cada exemplo os tempos de execução foram calculados por média de todos os tempos dos processos executados. Para cada execução do código observado, calcula-se o desvio padrão de tempo de CPU e de tempo Real. Foram amostrados os tempos de 5 execuções diferentes, excluindo sempre a primeira execução. Em alguns pontos dos gráficos não se visualiza o desvio padrão devido a escala do gráfico e o valor do desvio muito pequeno, porém todos os valores estão listados nas tabelas deste capítulo.

5.1 Regra do Trapézio Repetida

A regra do trapézio e uma aproximação para calcular a área sob uma curva que pode ser obtida pela substituição da função original por uma função discreta mais simples, no caso uma reta, dentro de um intervalo limitado. O Objetivo de aplicarmos a regra do Trapézio é calcular a integral definida pela Equação 5.2 [36]:

$$I = \int_a^b f(x)dx \tag{5.2}$$

O valor de I será associado a uma área conforme Figura 5.3. A área sob a curva *f(x)*, onde x varia entre a e b equivale aproximadamente a área abaixo da linha reta entre os pontos *(a; f(a))* e *(b; f(b))*, demonstrada na Equação 5.3. A área sombreada da figura corresponde ao calculo da integral seguindo esta aproximação [36].

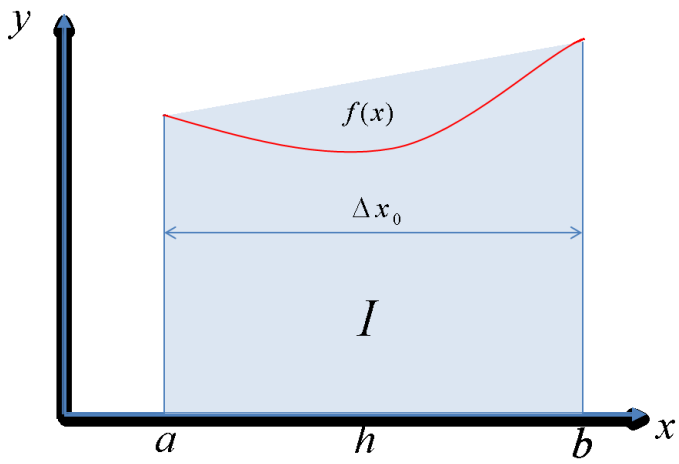

Figura 5.3: Área do Trapézio [36].

$$I \approx T_0 = \frac{1}{2}[f(a) + f(b)]\Delta x_0 \tag{5.3}$$

onde:

$$\Delta x_0 = (b - a) \tag{5.4}$$

Conforme se verifica na Figura 5.4 ao se aproximar uma função por uma reta obtém-se um erro de aproximação. Ao dividir o intervalo *(a-b)* em duas ou mais partes iguais levando a construção de n subintervalos, busca-se um resultado mais próximo da solução analítica, considerando que com estes subintervalos resulta na diminuição de erro do resultado. A área total abaixo da função foi dividida em n subintervalos de comprimento h [36].

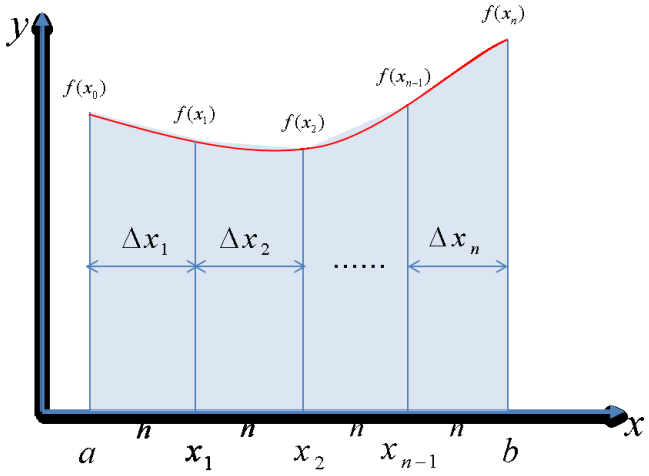

Figura 5.4: Área do Trapézio com subintervalos [36]

Aplicando a regra do trapézio repetido em cada um desses subintervalos obtermos a Equação 5.5, que será o resultado aproximado da integral.

$$I \approx T_n = h\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(x_i)\right) \tag{5.5}$$

e

$$h = \left(\frac{b-a}{n}\right) \tag{5.6}$$

onde:

- n -> número de subintervalos.
- h -> tamanho dos subintervalos

Observa-se na Figura 5.5 que para a paralelização do código, a área é dividida abaixo da função em partes iguais onde as funções de cada parte serão executadas por processos diferentes, trabalhando concomitantemente.

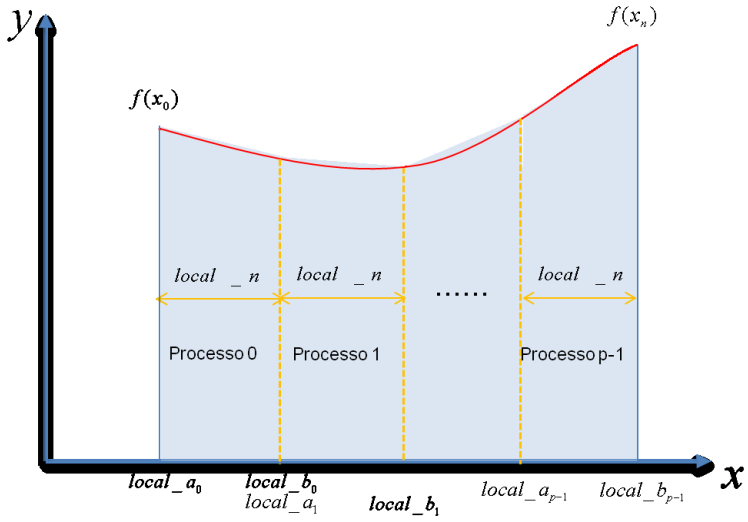

Figura 5.5: Área do Trapézio divida por processos.

Para se determinar as fronteiras das áreas a serem divididas utiliza-se uma variável do tipo inteiro para determinar o início e o final de cada subdivisão da área, afim de que cada processo utilize para cálculo somente sua área.

```
1    h = (b-a)/n;
2    local_n =n/p;
3    local_a = a + my_rank * local_n * h;
4    local_b = local_a + local_n * h;
```

Código Computacional 5.3: Divisão de submatrizes - trapmpisr.c e trapmpir.c

onde:

- a -> ponto inicial da função
- b -> ponto final da função
- n -> número de subintervalos
- h -> cumprimento total da função
- p -> quantidade de processos na execução em paralelo
- local_n -> quantidade de subdivisões para cada processo
- local_a -> inicio da subdivisão
- local_b -> final da subdivisão
- my_rank -> identificação do processo que esta executando o código

Observar-se no Código Computacional 5.3, na linha 2, a variável *local_ n* representa o números de subdivisões da área da função. Nas linhas 3 e 4, a variável *local_ a* foi utilizada para determinar o início da subdivisão e a variável *local_ b* para indicar o final da subdivisão. Desta maneira pode-se dividir igualitariamente a área da função em subdivisões, identificando o inicio e fim de cada uma, atribuindo um processo a cada subdivisão.

Os resultados de cada parte da função são transmitidos ao processo principal ou *rank 0*para gerar um único resultado. Conforme abordado no Capítulo 4 o processo identificado como *rank 0* é o processo que concatena todos os dados gerados por outros processos, apresentando o resultado final da operação. A comunicação ou envio e recebimento de mensagens pode ser implementada através de duas rotinas do OPENMPI: *MPI_ Send*/*MPI_ Recv* ou *MPI_ Reduce*.

Os processos que executam partes da função enviam através da rotina *MPI_ Send* o resultado de seus cálculos e o processo *rank 0* recebe através da rotina *MPI_ Recv*, efetuando o somatório e gerando assim o resultado final, conforme demonstrado no Código Computacional 5.4

```
1  if(my_rank == 0) {
2     total = integral;
3     for(source = 1; source < p; source++) {
4        MPI_Recv(&integral, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD,
             &status);
5        total = total + integral;
6     } }
7  else {
8     MPI_Send(&integral, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
9  }
```

Código Computacional 5.4: Utilizando *MPI_ Send* e *MPI_ Recv* - trapmpisr.c

Outra rotina abordada neste exemplo e a comunicação entre processos através do *MPI_ Reduce*. Pode-se observa que utiliza-se a opção *MPI_ SUM* que executa o somatório dos resultados de todos os processos, na operação de redução. Esta função esta implementada no Código Computacional 5.5

```
1 MPI_Reduce(&integral, &total, 1, MPI_DOUBLE,MPI_SUM, 0, MPI_COMM_WORLD)
    ;
```

Código Computacional 5.5: Utilizando *MPI_ Reduce* com a opção *MPI_ SUM* - trapmpir.c

5.1.1 Dados e Parâmetros para Execução do Código Computacional Serial e Paralelo

A função de x foi definida conforme a equação abaixo e será calculada neste exemplo.

$$I = \int_0^1 e^x dx \therefore f(x) = e^x \tag{5.7}$$

Para execução do código computacional serial e paralelo utiliza-se as variáveis abaixo:

- a = 0
- b = 1
- n = 2.400.000 subintervalos.
- p = 2, 4, 8, 16, 32 e 48 processos.

O número de subintervalos foi definido em 2.400.000 levando em consideração a precisão de até 15 dígitos significativos ou seja a limitação computacional do sistema, verificado pelo Código Computacional *teste.c* listado no Apêndice B.

Neste exemplo executa-se 3 códigos computacionais, sendo 1 serial e 2 paralelos utilizando rotinas de comunicação ponto a ponto, *MPI_Send* e *MPI_Recv*, e rotinas de paralelização em grupo, *MPI_Reduce*.

5.1.2 Resultados

Os resultados foram obtidos com até 15 dígitos significativos, devido à limitação computacional mencionado no subitem anterior. O resultado obtido foi de 1,718281828459050, sendo que o código serial obtido foi de 1,718281828426647 e os resultados paralelos, em relação a quantidade de processos, estão representados na Tabela 5.1.

Resultados Obtidos - Regra do Trapézio		
Processos	**Paralelo MPI_Recduce**	**Paralelo *MPI_Send* e *MPI_Recv***
2	1,718281828423420	1,718281828423420
4	1,718281828441920	1,718281828441920
8	1,718281828450660	1,718281828450670
16	1,718281828454910	1,718281828454910
32	1,718281828457000	1,718281828457000
48	1,718281828457690	1,718281828457690

Tabela 5.1: Resultados da regra do Trapézio

A Tabela 5.2 mostra os resultados de tempo CPU e o desvio padrão do código serial e dos códigos paralelos.

Trapézio - tempo CPU						
Processos	**Tempo 1**	**Tempo 2**	**Tempo 3**	**Tempo 4**	**Tempo 5**	**Desvio Padrão**
Serial	0,25596100	0,25196100	0,26096100	0,25096100	0,25996000	0,00452747
Paralelo - Método *MPI_ Send* e *MPI_ Recv*						
2	0,09098600	0,08948650	0,09098600	0,10698350	0,08998600	0,00746206
4	0,04074375	0,04499275	0,04249350	0,04324350	0,04624325	0,00214593
8	0,02849563	0,02499613	0,02724588	0,02762075	0,02787062	0,00133736
16	0,01493500	0,01562244	0,01543525	0,01574737	0,01568512	0,00032892
32	0,00874875	0,00859250	0,00859256	0,00871756	0,00856116	0,00008446
48	0,00649904	0,00651983	0,00568665	0,00618667	0,00637400	0,00034334
Paralelo - Método MPI_Reduce						
2	0,09298600	0,09048650	0,08748650	0,09198600	0,09398550	0,00253439
4	0,04199350	0,04549325	0,03799425	0,03849400	0,04374325	0,00326108
8	0,02974513	0,02662063	0,02524600	0,02699600	0,02137188	0,00305741
16	0,01874712	0,01537256	0,01512231	0,01780981	0,01537250	0,00167365
32	0,00856131	0,00877991	0,00881134	0,00890503	0,00884247	0,00013071
48	0,00674898	0,00656148	0,00664481	0,00670737	0,00676973	0,00008461

Tabela 5.2: Trapézio - tempo CPU e desvio padrão

A Tabela 5.3 mostra os resultados de tempo Real e o desvio padrão do código serial e dos códigos paralelos.

Trapézio - Tempo REAL						
Processos	**Tempo 1**	**Tempo 2**	**Tempo 3**	**Tempo 4**	**Tempo 5**	**Desvio P.**
Serial	0,25699997	0,25099993	0,26099992	0,25099993	0,25999999	0,00479584
Paralelo - Método *MPI_ Send* e *MPI_ Recv*						
2	0,12549996	0,12700009	0,12600017	0,14999986	0,12700009	0,01058527
4	0,06400013	0,06349993	0,06274998	0,06374991	0,08200002	0,00828668
8	0,03125012	0,03150007	0,03162506	0,04325017	0,03162509	0,00525703
16	0,01593754	0,01581250	0,01837493	0,01581246	0,01599999	0,00111398
32	0,00956254	0,00934369	0,00962495	0,00868759	0,00965628	0,00040320
48	0,00722909	0,00693753	0,00662500	0,00683343	0,00693748	0,00021817
Paralelo - Método *MPI_ Reduce*						
2	0,12649989	0,12599993	0,12599993	0,12649989	0,12649989	0,00027384
4	0,06299996	0,06299996	0,06299996	0,06299996	0,06324995	0,00011180
8	0,03112498	0,03124991	0,02987504	0,04337502	0,03025007	0,00573136
16	0,02843748	0,01725003	0,01643749	0,02400003	0,01600003	0,00553493
32	0,00987499	0,00981245	0,00996890	0,01059377	0,01009382	0,00031204
48	0,00768756	0,00775001	0,00850002	0,00791677	0,00781249	0,00032788

Tabela 5.3: Trapézio - tempo real e desvio padrão

No gráfico representado pela Figura 5.6 observa-se que independente do número de processos o tempo serial fica acima dos tempos apurados nos processos paralelos. Com o aumento do número de processos também observa-se uma diminuição significativa dos tempos de CPU. Entre os métodos de paralelização *MPI_ Send* e *MPI_ Recv* e o *MPI_ Reduce* os tempos são praticamente os mesmos sem variações significativas.

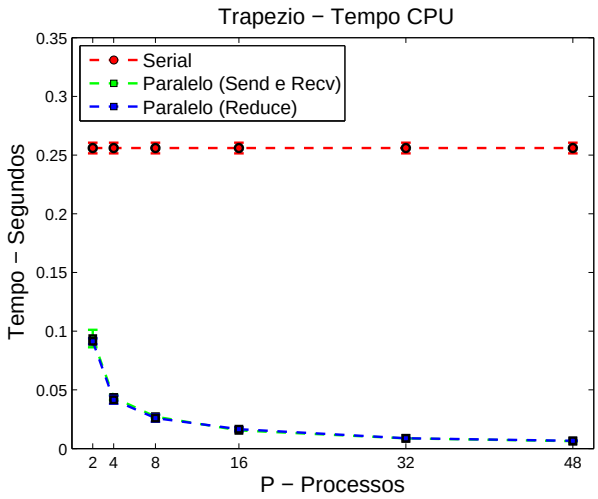

Figura 5.6: Gráfico de comparação entre o código serial e paralelo - tempo de CPU.

O gráfico representado pela Figura 5.7 apresenta os tempos reais de execução do código computacional, se comportando da mesma maneira que o gráfico representado pela Figura 5.6, porém com resultados de tempo distintos.

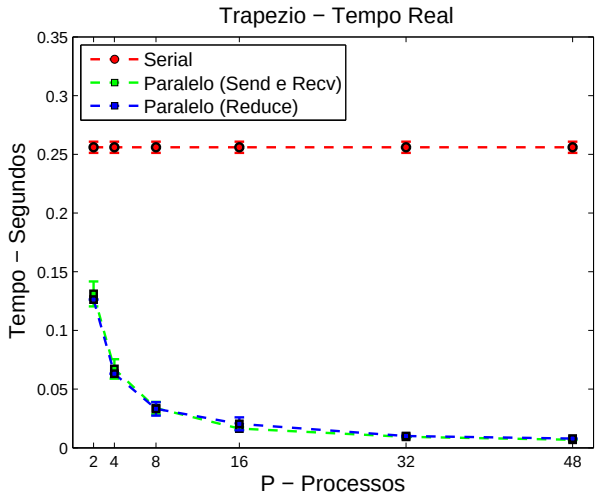

Figura 5.7: Gráfico de comparação entre o código serial e paralelo - tempo Real.

5.1.3 Verificação do Erro

Calcula-se o erro com os resultados obtidos, em relação ao resultado analítico e os resultados seriais e paralelos, com uma precisão de aproximadamente 15 dígitos significativos. O erro do código serial é de aproximadamente 0,000000000032405 e os códigos paralelos estão demonstrados na Tabela 5.4.

Verificação do Erro - Regra do Trapézio		
Processos	**Paralelo com *MPI_ Reduce***	**Paralelo com *MPI_ Send* e *MPI_ Recv***
2	0,000000000035625	0,000000000035625
4	0,000000000017125	0,000000000017125
8	0,000000000008385	0,000000000008375
16	0,000000000004135	0,000000000004135
32	0,000000000002045	0,000000000002045
48	0,000000000001355	0,000000000001355

Tabela 5.4: Erro da regra do Trapézio

Pode-se observar que o aumento de processos resulta na diminuição do erro, pois o erro não se propaga de uma subdivisão para outra, evitando assim o aumento acumulativo do erro a cada cálculo da integral.

O gráfico representado pela Figura 5.8 compara o erro relativo entre o código computacional serial e paralelo em relação aos processos paralelos executados.

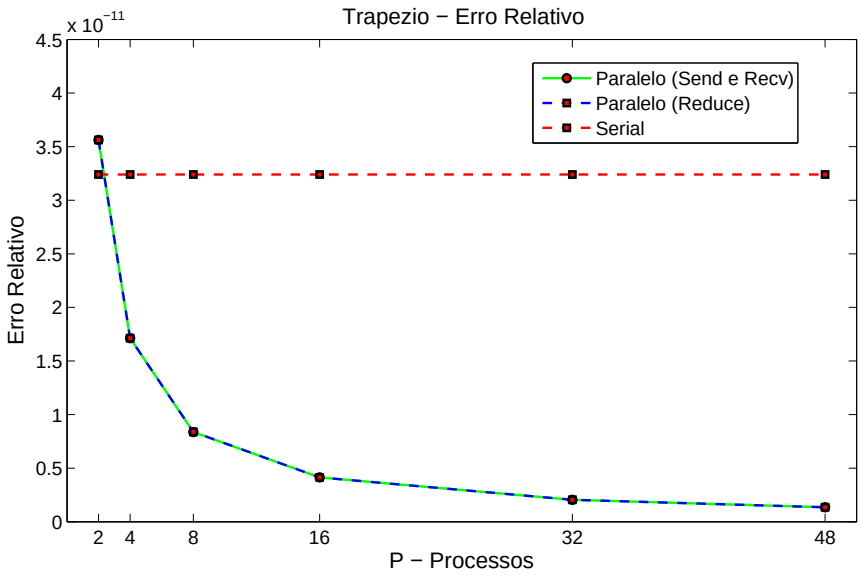

Figura 5.8: Gráfico comparativo entre o código serial e paralelo - erro relativo.

5.2 Equação de Laplace Estacionária em 2D

Demonstra-se neste exemplo a equação da propagação do calor ou equação de Laplace em duas dimensões, representada pela equação 5.8. O exemplo aplica a equação no estado estacionário. A equação de calor pode ser representada por[37]:

$$\frac{\partial u}{\partial t} = \alpha \bigtriangledown^2 u \tag{5.8}$$

onde

$$\alpha = K/c_p\rho \tag{5.9}$$

onde α é o termo difusivo e os componentes deste termo estão demonstrados abaixo:

- α -> difusão térmica
- K -> condutividade térmica
- ρ -> densidade da massa
- c_p -> capacidade térmica específica
- h -> distância entre os pontos da malha discretizada, incremento no espaço
- Δt -> incremento no tempo

O operador de segunda ordem, descrevendo a condução do calor em estado estacionário, em duas dimensões, é dado pela equação 5.10 [37]:

$$\bigtriangledown^2 u = 0 \tag{5.10}$$

onde

$$\bigtriangledown^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{5.11}$$

No método das diferenças finitas, cada derivada é aproximada de forma discreta em relação às duas dimensões do espaço. Utiliza-se o método numérico explícito de Jacobi [38].

5.2.1 Condição de Contorno e Condição Inicial

A condição de contorno neste exemplo é dado por uma fonte de calor de 100 graus nas fronteiras da esquerda e direita do domínio e nas demais fronteiras 0 grau [37]. A condição inicial é igual a 0 em toda a malha. Podemos representar através da Figura 5.9 as condições de contorno.

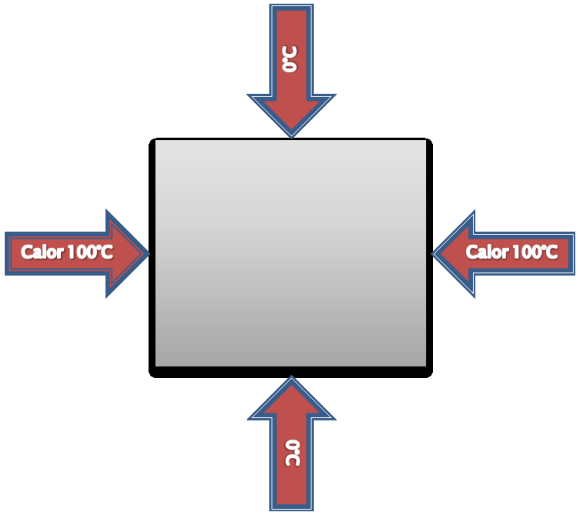

Figura 5.9: Condição inicial e de contorno.

5.2.2 Discretização do Domínio da Integração

Para discretização do domínio da integração confecciona-se a malha das diferenças finitas que será calculada e alocada em uma matriz u(i,j), conforme Figura 5.10

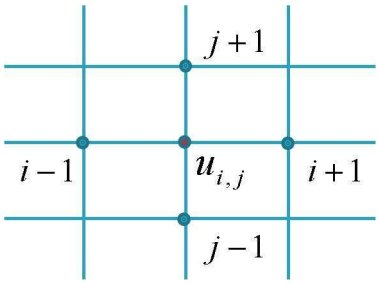

Figura 5.10: Malha discretizada equação de Laplace estacionária.

Cada ponto da malha discretizada e calculado com base nos pontos a frente *(i+1)* e a ré *(i-1)*, acima *(j+1)* e abaixo *(j-1)*.

5.2.3 Paralelização do Código Computacional

Para paralelizar um código computacional é necessário verificar qual a tarefa pode ser processada sem dependências de outros processos. No caso deste exemplo pode-se dividir a matriz a ser calculada em várias partes e distribuí-la entre vários processos onde cada submatriz e submetida as iterações do método de resolução afim de gerar os

resultados de cada submatriz. Como observado na Figura 5.11 cada matriz foi divida em submatrizes que por sua vez gerou vários processos. Esta divisão pode ser feita por linhas, colunas ou quadrantes, a critério do programador. Como já referenciado neste trabalho o método de paralelização via MPI e explicito, ou seja, a divisão destes processos, e consequentemente da matriz, é realizada pelo programador no momento de implementação do código computacional.

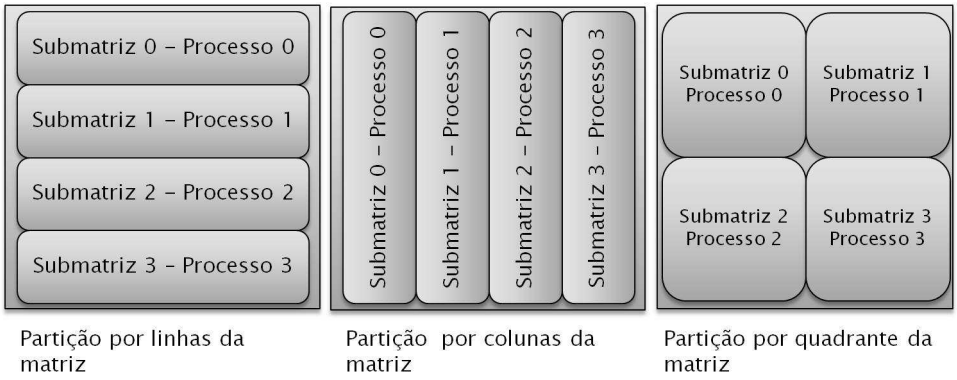

Figura 5.11: Divisão de uma matriz em submatrizes e processos.

5.2.4 Parâmetros para Execução do Código Computacional Serial e Paralelo

Neste exemplo foi utilizado a divisão da matriz em linhas, conforme representado na Figura 5.12. Um fator importante é a escolha da quantidade de processos em relação ao tamanho da matriz. O resultado da divisão das dimensões da matriz pelo número de processos utilizados na paralelização necessita ser um número inteiro, sendo diferente, tem-se um erro na divisão das submatrizes e consequentemente na resolução do problema.

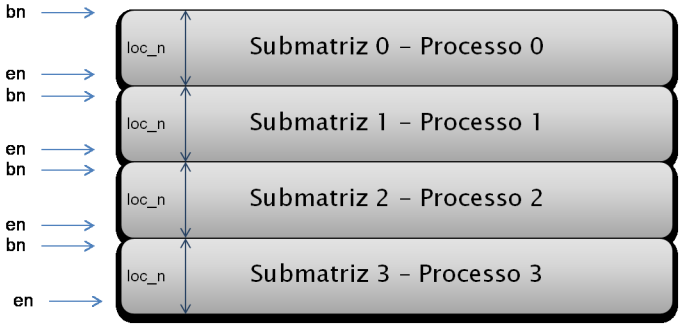

Figura 5.12: Divisão de submatrizes em linhas.

Um dos métodos para se determinar as fronteiras da submatriz e a utilização de uma variável inteira para determinar o início e o final de cada submatriz, afim de que cada processo utilize para calculo somente sua submatriz [37].

Observa-se no Código Computacional 5.6, na linha 1, a variável *loc_n* representa o números de linhas de cada submatriz. Nas linhas 2 e 3, a variável *bn* foi utilizada para determinar a linha início da submatriz e a variável *en* indica a última linha da submatriz. Desta maneira pode-se dividir igualitariamente a matriz global em submatrizes, identificando o inicio e fim de cada uma.

```
1  loc_n = (n-1)/p;
2  bn = 2 + (my_rank * loc_n);
3  en = bn + loc_n -1;
```

Código Computacional 5.6: Divisão em submatrizes - heat2dmpi.c e laplacempi.c

onde:

- loc_n -> quantidade de linhas para cada processo
- n -> dimensão da matriz
- p -> quantidade de processos
- bn -> número da linha inicial da submatriz de cada processo
- en -> número da linha final da submatriz de cada processo
- my_rank -> identificação do processo que esta executando o código

5.2.4.1 Comunicação entre Processos

Conforme explanado no início deste exemplo trata-se de um processo iterativo, sendo que cada ponto da malha discretizada e calculado com base nos pontos a frente e a ré, acima e abaixo. Desta forma quando paraleliza-se o código computacional, e consequentemente dividi-se a matriz global em submatrizes, é necessário a comunicação entre as fronteiras das submatrizes, a cada iteração, enviando ao processo anterior e ao posterior a sua fronteira e recebendo também a fronteira calculada por estes processos, para que o processo iterativo possa efetuar os cálculos dos pontos conforme exemplificado na Figura 5.13.

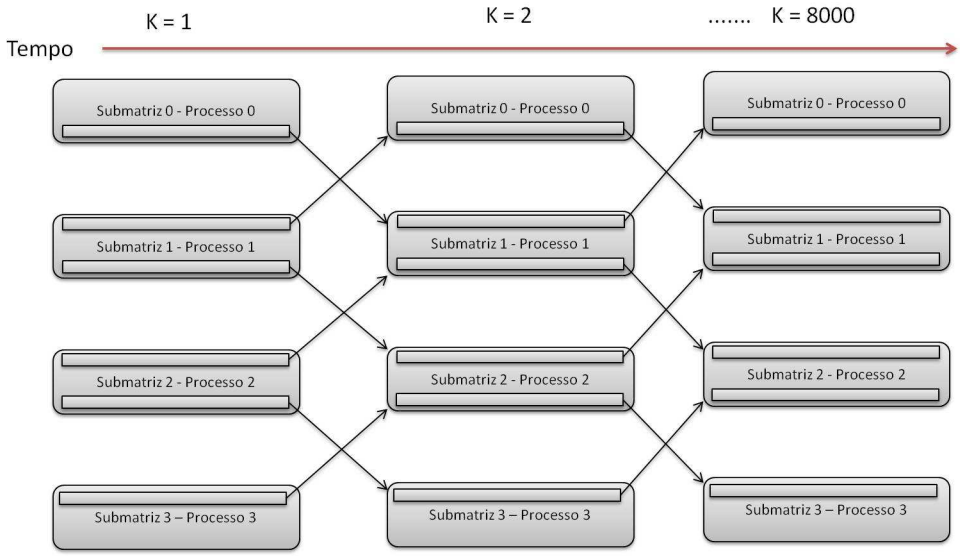

Figura 5.13: Envio e recebimento de fronteiras entre submatrizes e processos.

Nota-se através do exemplo que a cada iteração K o processo 1 envia a fronteira superior da submatriz 1 (*u(bn,1)*) ao processo 0, ou seja, o processo anterior. A fronteira inferior do processo 1 (*u(en,1)*) também é enviada ao processo 2, ou seja, processo posterior. Reciprocamente o processo 0 envia sua fronteira inferior (*u(en,1)*) ao processo 1, ou seja, ao processo posterior e o processo 2 envia sua fronteira superior (*u(bn,1)*) também ao processo 1, ou seja, ao processo anterior. Intui-se que como o processo 0 não possui processos anteriores, sua fronteira não é alterada, assim como o processo 3 não possui processos posteriores. Ressalta-se que este envio e recebimento de fronteiras entre os processos é efetuado a cada iteração[37].

O envio de mensagens, neste caso das fronteiras das submatrizes, é implementado através das rotinas de comunicação MPI. Neste exemplo utilizamos os *MPI_Send* e *MPI_Recv*, explicitado no Código Computacional 5.7.

```
1 if (my_rank == 0){
2     MPI_Recv(&uold[en+1][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
          MPI_COMM_WORLD, &status);
3   MPI_Send(&uold[en][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
       MPI_COMM_WORLD);}
4 if ((my_rank > 0) && (my_rank < p-1) && (my_rank%2==1)) {
5   MPI_Send(&uold[en][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
       MPI_COMM_WORLD);
6   MPI_Recv(&uold[en+1][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
       MPI_COMM_WORLD, &status);
```

```
7    MPI_Send(&uold[bn][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
         MPI_COMM_WORLD);
8    MPI_Recv(&uold[bn-1][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
         MPI_COMM_WORLD, &status);  }
9  if ((my_rank > 0) && (my_rank < p-1) && (my_rank%2==0)) {
10   MPI_Recv(&uold[bn-1][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
         MPI_COMM_WORLD, &status);
11   MPI_Send(&uold[bn][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
         MPI_COMM_WORLD);
12   MPI_Recv(&uold[en+1][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
         MPI_COMM_WORLD, &status);
13   MPI_Send(&uold[en][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
         MPI_COMM_WORLD); }
14 if (my_rank == p-1) {
15   MPI_Send(&uold[bn][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
         MPI_COMM_WORLD);
16   MPI_Recv(&uold[bn-1][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
         MPI_COMM_WORLD, &status);   }
```

Código Computacional 5.7: Utilização de *MPI_Send* e *MPI_Recv* para comunicação entre as fronteiras das submatrizes

Observa-se que a estrutura *if* verifica primeiramente qual o *my_rank*, ou seja, qual a identificação do processo e consequentemente da submatriz que vai enviar e receber a fronteira. Após esta verificação o comando *MPI_Send* envia um ponteiro da matriz *&uold[en][1]* para o processo posterior ou *&uold[bn][1]* para o processo anterior, juntamente com o número de elementos *n+1* correspondente a fronteira, para o processo destino. Consequentemente recebe através da rotina *MPI_Recv* os elementos correspondentes a fronteira, ou seja um ponteiro da matriz *&uold[en+1][1]* ou *&uold[bn-1][1]* de outro processo posterior ou anterior. Esta rotina deve ser feita igualmente para as submatrizes anterior e posterior. Nota-se que o processo 0 envia e recebe a fronteira inferior de seu processo posterior e o processo *n-1* envia e recebe a fronteira superior de seu processo anterior [37]. A Figura 5.14 demonstra a troca de fronteiras entre os processos.

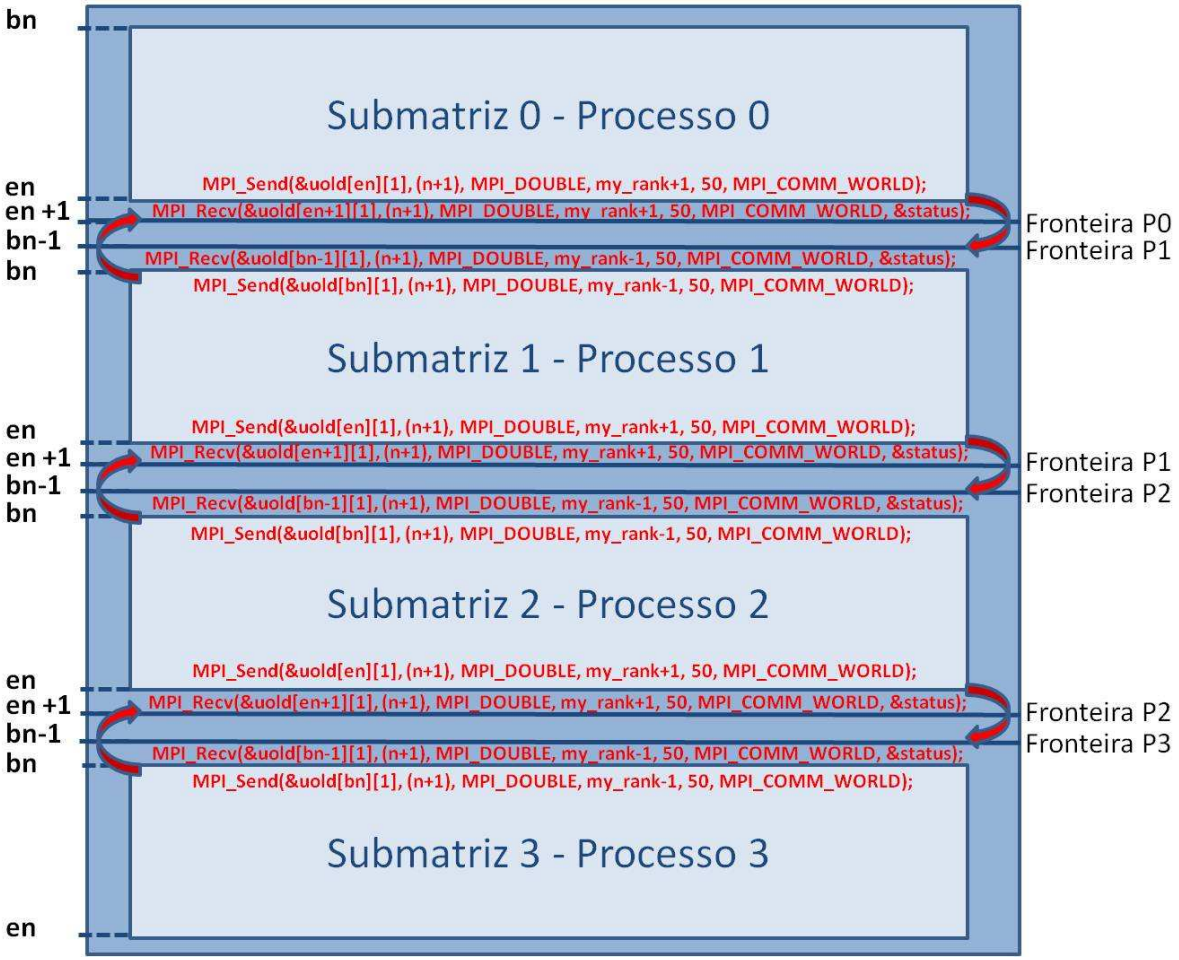

Figura 5.14: Exemplo de transmissão de fronteiras entre os processos utilizando *MPI_Send* e *MPI_Recv*.

5.2.5 Dados para Execução do Código Computacional Serial e Paralelo

As demais variáveis para execução do código computacional nesta secção estão relacionados aos parâmetros e variáveis deste exemplo.

- n = 290 -> tamanho da matriz
- P = 4, 8, 16, 32, 48 -> quantidade de processos utilizados para a execução em paralelo
- k = 8000 -> quantidade de iterações para o método

5.2.6 Resultados

Após a execução dos códigos no Cluster VORTEX os resultados foram obtidos e representados em forma de gráficos. Foram obtidos por códigos computacionais serial e paralelo, obtendo assim a confirmação da correta implementação da paralelização. Demonstra-se na Figura 5.15 o resultado da equação de Laplace no modo estacionário com o código computacional serial.

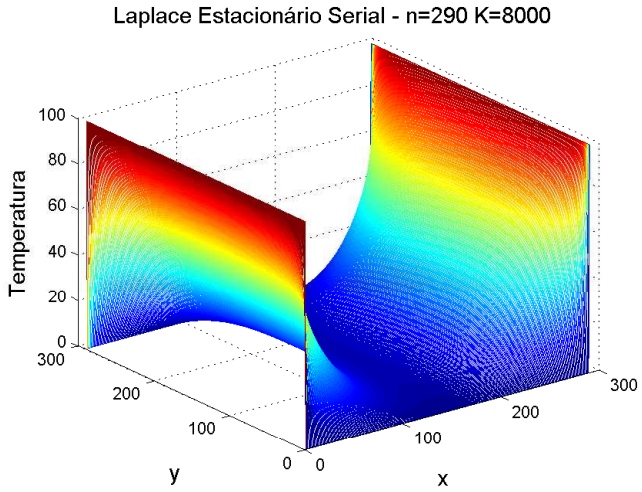

Figura 5.15: Gráfico da equação de Laplace estacionária serial.

Observa-se na Figura 5.16 que o resultado do código computacional paralelizado com 8 processos foi similar ao resultado do código computacional serial, não interferindo assim o método de paralelização no resultado da equação.

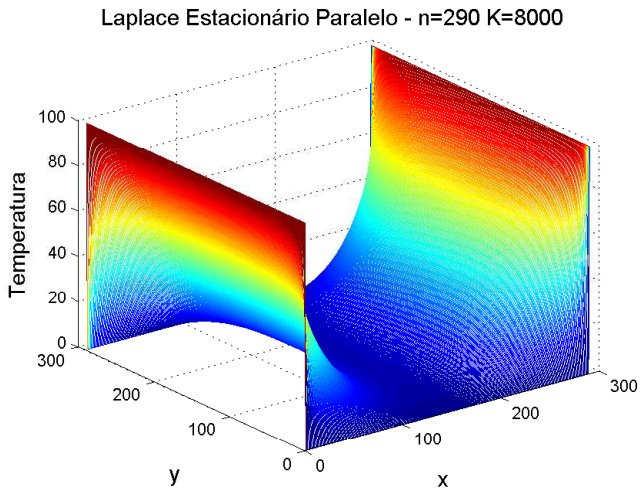

Figura 5.16: Gráfico da equação de Laplace estacionária paralelo.

A Tabela 5.5 mostra os resultados de tempo de CPU e o desvio padrão do código serial e paralelo.

Laplace Estacionário - Tempo CPU						
Serial/Paralelo	**Tempo 1**	**Tempo 2**	**Tempo 3**	**Tempo 4**	**Tempo 5**	**Desvio Padrão**
Serial	5,36218400	5,30019400	5,30619300	5,26919800	5,30219300	0,03368761
4 processos	1,90671025	1,91370875	1,85571775	1,92445725	1,88371350	0,02742276
8 processos	1,66774650	1,42715787	1,01247100	1,15382450	1,07133700	0,27482940
16 processos	0,79719125	0,76407137	0,94923062	1,00034794	0,84112219	0,10077602
32 processos	1,02178209	0,93941962	0,90267528	0,97500816	0,98572531	0,04553187
48 processos	1,23479144	0,99116185	1,03059329	0,96089560	1,13926425	0,11359673

Tabela 5.5: Laplace estacionário - tempo CPU e desvio padrão

A Tabela 5.6 mostra os resultados de tempo de Real e o desvio padrão do código serial e paralelo.

Laplace Estacionário - Real						
Serial/Paralelo	**Tempo 1**	**Tempo 2**	**Tempo 3**	**Tempo 4**	**Tempo 5**	**Desvio Padrão**
Serial	5,36599994	5,30200005	5,30900002	5,26999998	5,30499983	0,03473181
4 processos	4,75224996	4,71025002	4,75649989	4,78825009	4,74275005	0,02803125
8 processos	4,00200009	3,98700005	3,87425005	4,02250001	4,01362503	0,06052452
16 processos	3,93549997	3,83906248	4,11468755	3,81168756	3,94687504	0,11904938
32 processos	6,48150007	6,47837508	6,59078126	6,52465628	6,48465623	0,04789371
48 processos	9,63754892	9,51708330	10,32060423	9,49908334	10,11332453	0,37571524

Tabela 5.6: Laplace estacionário - tempo Real e desvio padrão.

O gráfico representado pela Figura 5.17 demonstra os resultados comparativos de tempo de CPU entre os códigos computacionais em serial e paralelo. Pode-se notar que o tempo de CPU nas execuções paralelas foram menor que o tempo na execução serial, sendo o melhor valor com 16 processos.

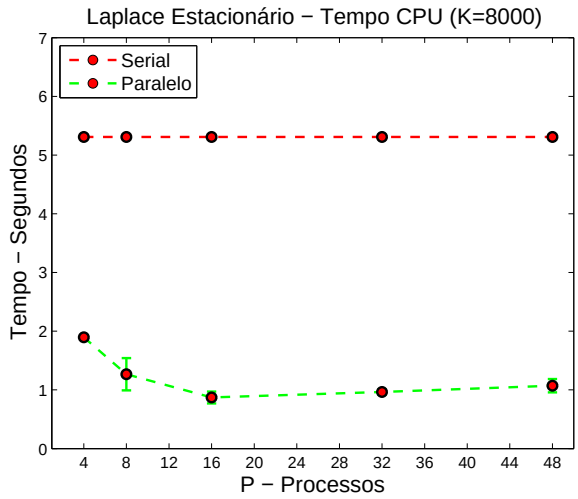

Figura 5.17: Comparação de tempo de CPU em relação a execução serial e paralelo da equação de Laplace estacionária.

Nota-se no gráfico representado pela Figura 5.18 que o tempo Real das as execuções paralelas até 16 processos estão abaixo da execução do código serial, porém os tempos apurados com 32 e 48 processos estão acima da linha que indica o tempo serial. Observa-se que, ao contrario do tempo de CPU, o tempo Real do código paralelo considera o tempo de troca de mensagens entre os processos, que ocorre a cada iteração, resultando no aumento de tempo em relação ao processo serial.

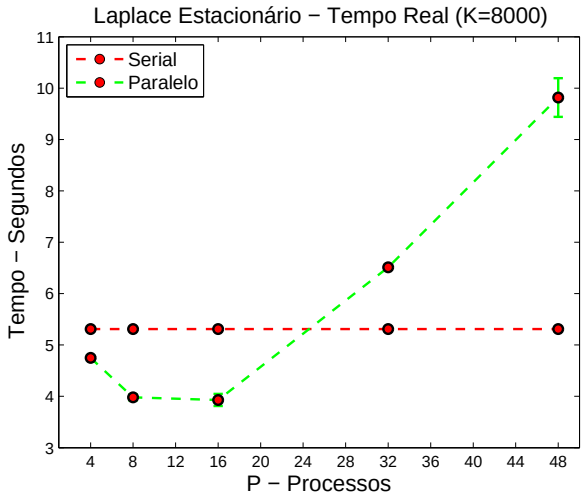

Figura 5.18: Comparação de tempo real em relação a execução serial e paralelo da equação de Laplace estacionária.

5.3 Equação de Laplace Transiente em 2D

Neste último exemplo demonstra-se a equação da propagação do calor ou equação de Laplace em duas dimensões, representada pela Equação 5.12. Neste caso a equação está no estado transiente, ou seja, dependente do tempo. A equação de calor pode ser representada por [37]:

$$\frac{\partial u}{\partial t} = \alpha \bigtriangledown^2 u \tag{5.12}$$

onde

$$\alpha = \left(\frac{K}{c_p}\right)\left(\frac{\Delta t}{h^2}\right) \tag{5.13}$$

onde α é o termo difusivo e os componentes deste termo estão aoscrito no subitem 5.2:

Utiliza-se neste exemplo o Método de Euler explicito. Discretização avante no tempo. O Método de Diferenças Finitas: avante para o tempo e centrada para o espaço

5.3.1 Condição de Contorno e Condição Inicial

A condição de contorno neste exemplo é alterada com o decorrer do tempo. A fronteira inicial e dado por uma fonte de calor de 380 graus na fronteira da esquerda e nas demais fronteiras 70 graus, conforme demonstrado na Figura 5.19. A condição inicial é igual a 70 graus, onde *u(x,y,0)=70* [37].

Quando o tempo>= 1000 as condições de fronteira são igualadas em 70 graus, conforme demonstrado na Figura 5.20.

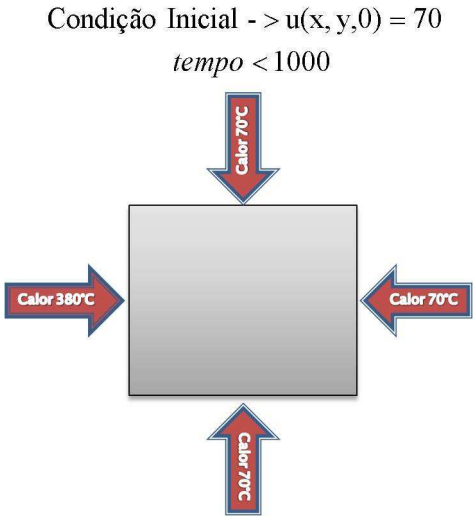

Figura 5.19: Condição inicial e de contorno para tempo $<$ 1000.

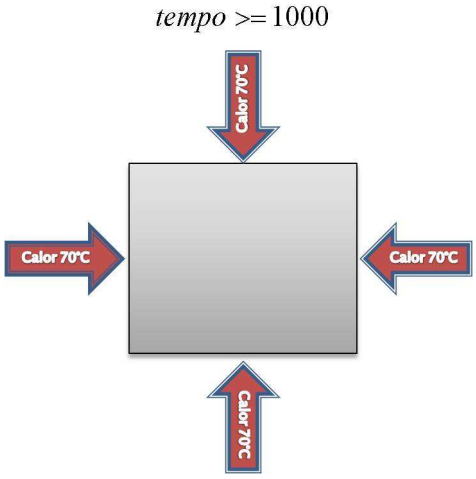

Figura 5.20: Condição de contorno para tempo >=1000.

5.3.2 Discretização do Domínio da Integração

Como no exemplo anterior para discretização do domínio da integração utiliza-se a malha das diferenças finitas que será calculada e alocada em uma matriz u(i,j), conforme Figura 5.21

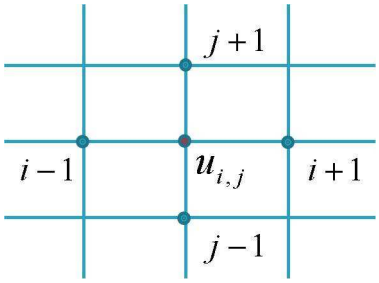

Figura 5.21: Malha discretizada equação de Laplace transiente.

Cada ponto da malha discretizada e calculado com base nos pontos a frente *(i+1)* e a ré *(i-1)*, acima *(j+1)* e abaixo *(j-1)*.

5.3.3 Parâmetros para Execução do Código Computacional Serial e Paralelo

Neste exemplo foi utilizado a divisão da matriz em linha, conforme descrito no subitem 5.2.4 .

5.3.3.1 Comunicação entre Processos

A comunicação entre processos foi descrita no subitem 5.2.4.1.

5.3.4 Dados para Execução do Código Computacional

Neste subitem estão relacionados os parâmetros e variáveis para a execução do código computacional deste exemplo.

- L = 1.0 -> comprimento na direção x
- W = 1.0 -> comprimento na direção y
- Tend = 14,31 / 6,38 / 3,59 (Δt) - incremento no tempo
- Tmaxk = 5000 -> total de iterações
- maxk = 500 / 1000 / 2000 / 3500 / 5000 -> iterações
- n = 194 / 290 / 386 -> matriz
- h = dx = dy -> distância entre os pontos da malha discretizada, incremento no espaço.

- cond = 0.002 -> condutividade térmica (K)
- rho = 1.0 -> densidade da massa (ρ)
- spheat = 1.0 -> capacidade térmica especifica (c_p)

5.3.5 Resultados

Após processamento no *Cluster* VORTEX os resultados obtidos foram representados em forma de gráficos. Os resultados foram obtidos por códigos computacionais serial e paralelo, obtendo assim a confirmação da correta implementação da paralelização.

As Figuras 5.22, 5.23 e 5.24 expõem o resultado da equação de Laplace no modo transiente com o código computacional serial, nas dimensões de matrizes quadradas de 194, 290 e 386 respectivamente e nos tempos de 500, 1000, 2000 e 5000.

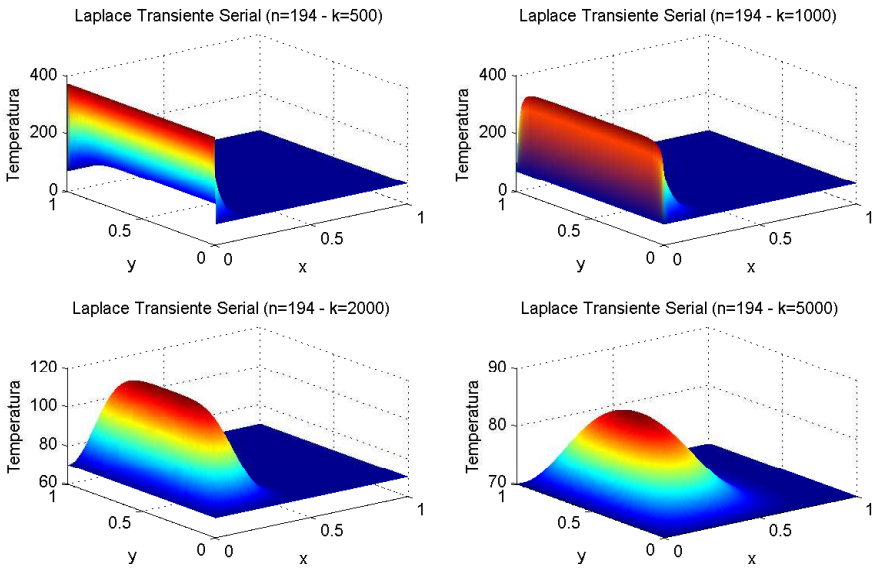

Figura 5.22: Gráficos do resultado da equação de Laplace transiente - serial n=194.

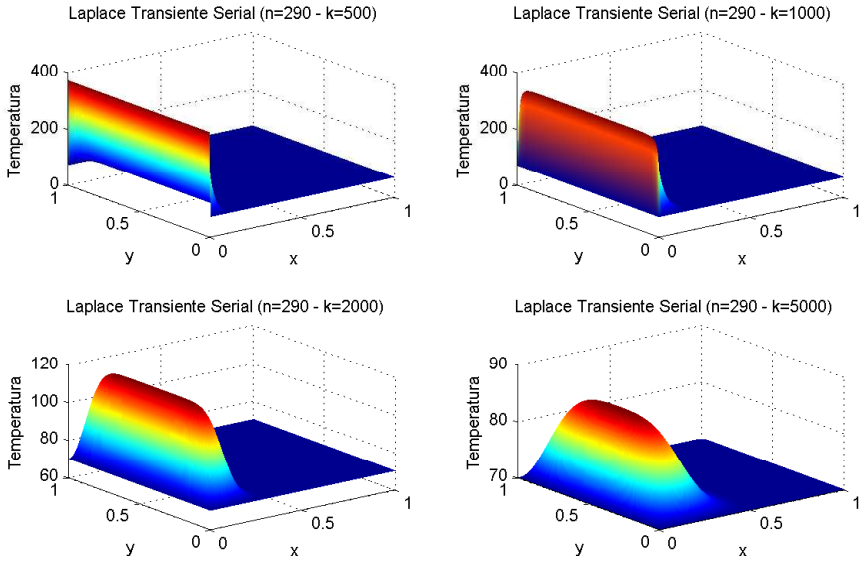

Figura 5.23: Gráficos do resultado da equação de Laplace transiente - serial n=290.

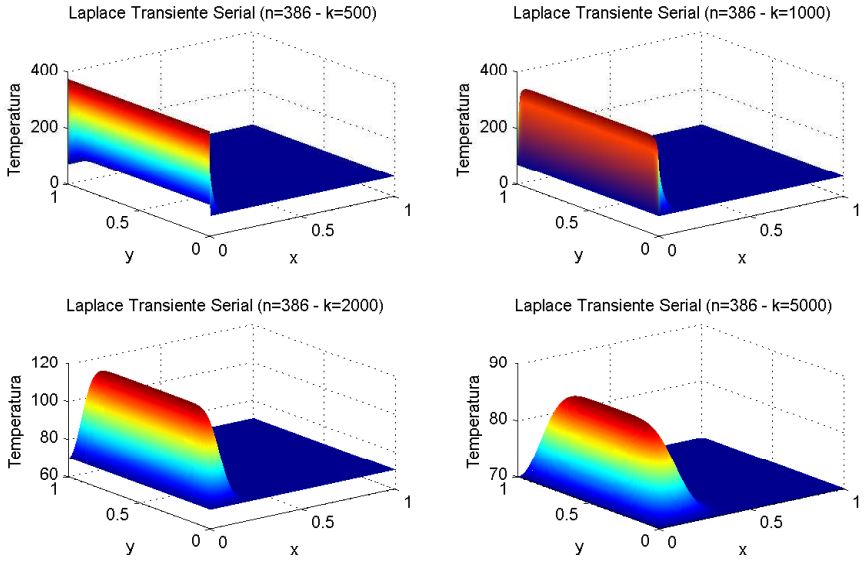

Figura 5.24: Gráficos do resultado da equação de Laplace transiente - serial n=386.

As Figuras 5.25, 5.26 e 5.27 apresentam os resultados da equação de Laplace no modo transiente com o código computacional paralelo, nas dimensões de matrizes quadradas de 194, 290 e 386 respectivamente e nos tempos de 500, 1000, 2000 e 5000.

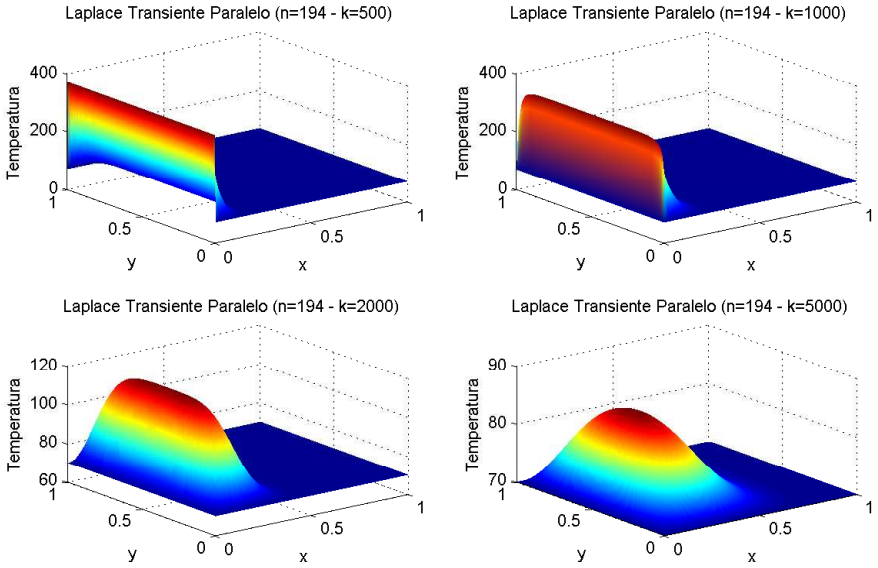

Figura 5.25: Gráficos do resultado da equação de Laplace transiente - paralelo n=194.

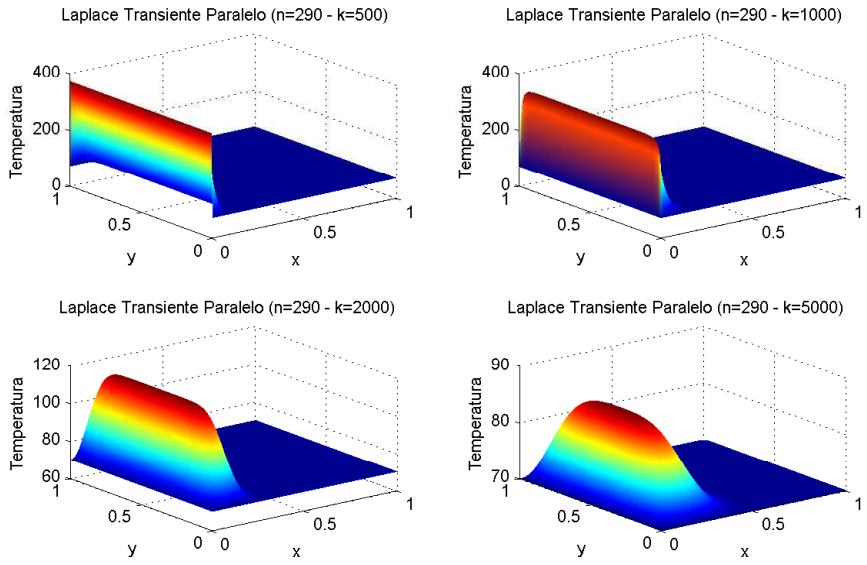

Figura 5.26: Gráficos do resultado da equação de Laplace transiente - paralelo n=290.

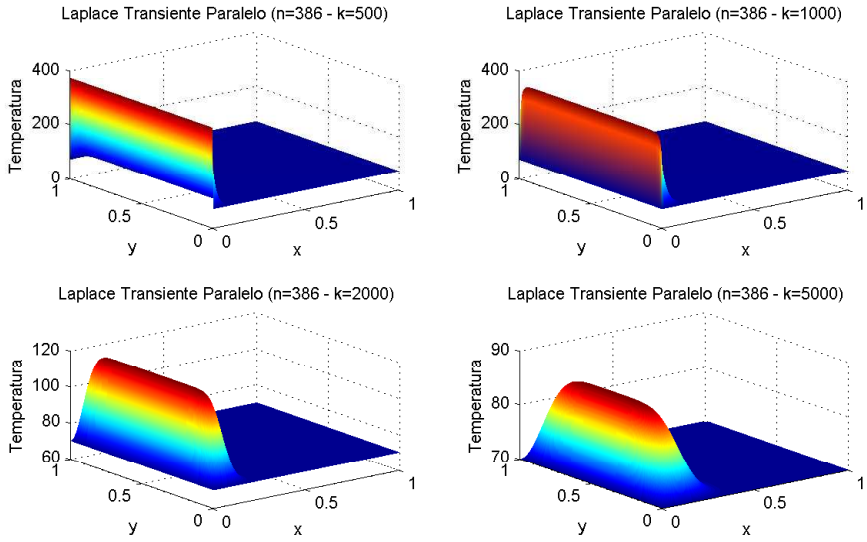

Figura 5.27: Gráficos do resultado da equação de Laplace transiente - paralelo n=386.

As Tabelas 5.7, 5.8 e 5.9 mostram os resultados de tempo de CPU e o desvio padrão do código serial e paralelo para matrizes quadradas de 194, 290 e 386 respectivamente.

Laplace Transiente - Tempo CPU n=194						
Serial/Paralelo	**Tempo 1**	**Tempo 2**	**Tempo 3**	**Tempo 4**	**Tempo 5**	**Desvio Padrão**
Serial k=500	0,24596300	0,24996200	0,25596100	0,24996200	0,25196200	0,00363249
Serial k=1000	0,48392700	0,47792700	0,47992700	0,49192600	0,48292600	0,00536618
Serial k=2000	0,95485500	0,96185400	0,95785400	0,97985100	0,96185400	0,00973503
Serial k=3500	1,65974800	1,69574200	1,66174800	1,65974800	1,65474900	0,01663448
Serial k=5000	2,45562700	2,50162000	2,49862100	2,46362600	2,50561900	0,02348679
4 processos k=500	0,13699450	0,07073925	0,09173625	0,15797600	0,13347950	0,03578295
4 processos k=1000	0,16923925	0,17573875	0,16797450	0,16947425	0,15772625	0,00650276
4 processos k=2000	0,25147725	0,28245700	0,34994675	0,36369425	0,38794125	0,05760766
4 processos k=3500	0,60493875	0,62588050	0,62640475	0,64168275	0,62615500	0,01308482
4 processos k=5000	0,80066925	0,81687625	0,82140575	0,94352750	0,93778625	0,07038239
16 processos k=500	0,03993144	0,04311838	0,04605550	0,04661794	0,04155631	0,00286795
16 processos k=1000	0,08167519	0,08554956	0,08961131	0,07661325	0,09836025	0,00824895
16 processos k=2000	0,19034619	0,15347662	0,15447656	0,17641087	0,17372362	0,01567196
16 processos k=3500	0,23390194	0,31920156	0,34844719	0,34019831	0,29045587	0,04632588
16 processos k=5000	0,40737556	0,38794094	0,50117381	0,40712569	0,44162031	0,04472211
48 processos k=500	0,03918154	0,04220187	0,04101454	0,03943150	0,04195187	0,00139828
48 processos k=1000	0,07942544	0,08188333	0,07711327	0,08234165	0,07969615	0,00210616
48 processos k=2000	0,15660117	0,15762183	0,15564296	0,15414323	0,15753856	0,00145386
48 processos k=3500	0,27895750	0,28980998	0,27506238	0,31413977	0,27233360	0,01704658
48 processos k=5000	0,46878273	0,39431502	0,44911919	0,39250279	0,40156388	0,03526858

Tabela 5.7: Laplace transiente - tempo CPU e desvio padrão para n=194.

Laplace Transiente - Tempo CPU n=290						
Serial/Paralelo	**Tempo 1**	**Tempo 2**	**Tempo 3**	**Tempo 4**	**Tempo 5**	**Desvio Padrão**
Serial k=500	0,39394000	0,40293900	0,39993900	0,40093900	0,40393800	0,00391088
Serial k=1000	0,80487800	0,80887700	0,81387700	0,81987600	0,80787800	0,00588990
Serial k=2000	1,60875500	1,60475600	1,60375700	1,62375400	1,64875000	0,01898945
Serial k=3500	2,79357600	2,83157000	2,83656900	2,80057500	2,82157100	0,01893627
Serial k=5000	3,99039400	4,00339100	4,00739100	3,98939400	3,99639300	0,00790421
4 processos k=500	0,14997700	0,16272525	0,15622600	0,15497600	0,15872575	0,00470775
4 processos k=1000	0,28270700	0,30695325	0,29870450	0,31445250	0,30970325	0,01245742
4 processos k=2000	0,60690800	0,62215525	0,64690175	0,69864400	0,65690050	0,03531050
4 processos k=3500	1,03109300	1,06183850	1,03159325	0,99834850	1,08333550	0,03252703
4 processos k=5000	1,46327750	1,63650150	1,56701175	1,57201125	1,33854675	0,11675614
16 processos k=500	0,06092812	0,06474006	0,05542906	0,08954875	0,06355281	0,01319150
16 processos k=1000	0,11491988	0,13141769	0,11348262	0,12585575	0,13229256	0,00893688
16 processos k=2000	0,23327706	0,24483769	0,26533469	0,22177869	0,26520969	0,01931774
16 processos k=3500	0,46380431	0,38025475	0,42143594	0,42287319	0,36169500	0,04002985
16 processos k=5000	0,49230006	0,51410938	0,57828700	0,66452406	0,57116312	0,06706423
48 processos k=500	0,05597058	0,05642898	0,05642898	0,05642898	0,05642898	0,00020500
48 processos k=1000	0,10269265	0,10654625	0,10831681	0,10450487	0,10783775	0,00235665
48 processos k=2000	0,21813350	0,21448810	0,22936092	0,20761429	0,21652962	0,00788108
48 processos k=3500	0,37565125	0,37987977	0,37515133	0,38117119	0,38429579	0,00385018
48 processos k=5000	0,54531296	0,55639450	0,55022877	0,54193842	0,53333562	0,00868514

Tabela 5.8: Laplace transiente - tempo CPU e desvio padrão para n=290.

Laplace Transiente - Tempo CPU n=386						
Serial/Paralelo	**Tempo 1**	**Tempo 2**	**Tempo 3**	**Tempo 4**	**Tempo 5**	**Desvio Padrão**
Serial k=500	0,95085600	0,94585600	0,93985700	0,94285700	0,94985600	0,00463632
Serial k=1000	1,96770100	1,99769700	1,98069900	1,97370000	1,97770000	0,01127680
Serial k=2000	3,98739400	3,94540000	3,97639600	3,93340200	3,99339300	0,02643852
Serial k=3500	6,96994000	6,95594300	6,94194500	6,96994000	6,94794400	0,01269416
Serial k=5000	9,91449300	9,85350200	9,80750900	9,82150700	9,95248700	0,06187197
4 processos k=500	0,35919550	0,35269600	0,35369600	0,35394625	0,36294500	0,00440619
4 processos k=1000	0,69339450	0,69814350	0,70764275	0,70114325	0,70164350	0,00521356
4 processos k=2000	1,41753450	1,38728925	1,36054325	1,33529700	1,39928725	0,03244570
4 processos k=3500	2,48462200	2,43662950	2,48637175	2,47862350	2,42063175	0,03055783
4 processos k=5000	3,35948950	3,42022975	3,44672575	3,50721675	3,53471250	0,06977030
16 processos k=500	0,12010694	0,12048194	0,12554338	0,11873194	0,11798212	0,00295940
16 processos k=1000	0,25102444	0,25489875	0,23246462	0,25477387	0,25202412	0,00941791
16 processos k=2000	0,47567750	0,43074713	0,50392344	0,52848206	0,50598550	0,03755306
16 processos k=3500	0,94885575	0,84849606	0,78588044	0,89330162	0,80056594	0,06718987
16 processos k=5000	1,16494794	1,16582287	1,20019256	1,11226844	1,18794431	0,03368466
48 processos k=500	0,09496485	0,10396325	0,09188188	0,09267348	0,10629633	0,00669692
48 processos k=1000	0,18286804	0,19634517	0,18143081	0,18124335	0,17949350	0,00685222
48 processos k=2000	0,52937790	0,37887987	0,35990365	0,37115194	0,48517629	0,07710398
48 processos k=3500	0,64213156	0,65208831	0,63379946	0,68593733	0,65631694	0,01986154
48 processos k=5000	1,09460446	0,92294300	1,11374733	0,98682915	0,94656440	0,08662875

Tabela 5.9: Laplace transiente - tempo CPU e desvio padrão para n=386.

As Tabelas 5.10, 5.11 e 5.12 mostram os resultados de tempo de CPU e o desvio padrão do código serial e paralelo para matrizes quadradas de 194, 290 e 386 respectivamente.

Laplace Transiente - Tempo Real n=194						
Serial/Paralelo	**Tempo 1**	**Tempo 2**	**Tempo 3**	**Tempo 4**	**Tempo 5**	**Desvio Padrão**
Serial k=500	0,24600005	0,25000000	0,25600004	0,25000000	0,25200009	0,00363319
Serial k=1000	0,47799993	0,48000002	0,49099994	0,48300004	0,48400021	0,00496991
Serial k=2000	0,95600009	0,96199989	0,95899987	0,98000002	0,96500015	0,00934346
Serial k=3500	1,65899992	1,69500017	1,66199994	1,66000009	1,65599990	0,01613391
Serial k=5000	2,45700002	2,50500011	2,49900007	2,46399999	2,50599980	0,02374235
4 processos k=500	0,26350003	0,26875001	0,26800007	0,26425004	0,26400000	0,00247109
4 processos k=1000	0,51500005	0,51325005	0,52100003	0,52399993	0,52724993	0,00591761
4 processos k=2000	1,01349998	1,01400012	1,02099997	1,01999998	1,03300005	0,00787083
4 processos k=3500	1,76550001	1,76574993	1,76799995	1,76999998	1,80299997	0,01606471
4 processos k=5000	2,51975000	2,55775005	2,54724997	2,52999997	2,52050000	0,01684805
16 processos k=500	0,22025001	0,21862513	0,21343745	0,21149999	0,21743752	0,00365802
16 processos k=1000	0,42287506	0,43818749	0,42581254	0,42881250	0,43468748	0,00629859
16 processos k=2000	0,85068755	0,83943745	0,83068737	0,84231248	0,84699997	0,00766665
16 processos k=3500	1,51643752	1,46187499	1,52199998	1,50993751	1,46356252	0,02956770
16 processos k=5000	2,06156255	2,14368749	2,05512506	2,09293753	2,09618749	0,03519290
48 processos k=500	0,46964584	0,47062498	0,46922915	0,46947915	0,46927082	0,00057024
48 processos k=1000	0,93602076	0,95045832	0,93489587	0,93483329	0,94945837	0,00807769
48 processos k=2000	1,86785417	1,86966663	1,86222917	1,90254169	1,88629169	0,01652080
48 processos k=3500	3,29702079	3,26327083	3,29427080	3,90779170	3,30733338	0,27656215
48 processos k=5000	5,70804168	4,66389587	5,55327081	4,70689582	4,68410415	0,52108068

Tabela 5.10: Laplace transiente - tempo Real e desvio padrão para n=194.

Laplace Transiente - Tempo Real n=290						
Serial/Paralelo	**Tempo 1**	**Tempo 2**	**Tempo 3**	**Tempo 4**	**Tempo 5**	**Desvio Padrão**
Serial k=500	0,39400005	0,40300012	0,39899993	0,40099978	0,40300012	0,00374168
Serial k=1000	0,80499983	0,80900002	0,81300020	0,81999993	0,80700016	0,00593296
Serial k=2000	1,61000013	1,60399985	1,60399985	1,62799978	1,64800000	0,01905782
Serial k=3500	2,79500008	2,83300018	2,83699989	2,80099988	2,82200003	0,01883615
Serial k=5000	3,99300003	4,00400019	4,00800014	3,99099994	3,99900007	0,00717644
4 processos k=500	0,33824992	0,33700001	0,33675009	0,33550000	0,33300006	0,00198900
4 processos k=1000	0,67474997	0,65675008	0,66400003	0,68899989	0,66674989	0,01230212
4 processos k=2000	1,29550004	1,28175002	1,30474997	1,32900000	1,32100004	0,01905961
4 processos k=3500	2,25600004	2,25849992	2,29525006	2,24500006	2,23725003	0,02230794
4 processos k=5000	3,21375000	3,28924996	3,21700001	3,24724996	3,29124999	0,03754429
16 processos k=500	0,25099993	0,25387508	0,25043747	0,46299988	0,24799991	0,09490911
16 processos k=1000	0,48799992	0,52199998	0,51681243	0,50662500	0,49018750	0,01532325
16 processos k=2000	0,96756250	0,99637492	1,17100000	0,96912491	1,04712500	0,08501301
16 processos k=3500	1,74093747	1,74243748	1,94268756	1,95431249	1,68962507	0,12467611
16 processos k=5000	2,43368752	2,56406254	2,59100002	2,49337500	2,41237496	0,07825974
48 processos k=500	0,59760418	0,63787502	0,59760418	0,63787502	0,59760418	0,02205725
48 processos k=1000	1,19447916	1,18645839	1,18683332	1,18764581	1,18647914	0,00344372
48 processos k=2000	2,36389584	2,36570831	2,64189585	2,37125000	2,37122923	0,12252459
48 processos k=3500	4,13739588	4,14877086	4,14424996	4,17275001	4,13545832	0,01497150
48 processos k=5000	5,92962504	6,12216667	5,91441671	5,90202082	5,95045838	0,09038983

Tabela 5.11: Laplace transiente - tempo Real e desvio padrão para n=290.

Laplace Transiente - Tempo Real n=386						
Serial/Paralelo	**Tempo 1**	**Tempo 2**	**Tempo 3**	**Tempo 4**	**Tempo 5**	**Desvio Padrão**
Serial k=500	0,95099998	0,94600010	0,94000006	0,94299984	0,95000005	0,00463682
Serial k=1000	1,96700001	1,99799991	1,98000002	1,97399998	1,97799993	0,01152385
Serial k=2000	3,98699999	3,94600010	3,98099995	3,93400002	3,99699998	0,02741346
Serial k=3500	6,97900009	6,95700002	6,95099998	6,97099996	6,94999981	0,01283751
Serial k=5000	9,91900015	9,85699987	9,81400013	9,82699990	9,95499992	0,06058712
4 processos k=500	0,58349997	0,58674997	0,58125001	0,58699995	0,60574996	0,00974454
4 processos k=1000	1,15225011	1,15050006	1,15400004	1,15599996	1,14899993	0,00277038
4 processos k=2000	2,27800006	2,31925005	2,27750009	2,27050000	2,29399997	0,01955089
4 processos k=3500	3,99500006	3,97124994	4,00574988	3,98575008	3,98600000	0,01275611
4 processos k=5000	5,66800004	5,66999996	5,66525000	5,65024990	5,65599990	0,00843473
16 processos k=500	0,36999993	0,35399999	0,34999999	0,35999998	0,35499997	0,00769413
16 processos k=1000	0,69887505	0,69999994	0,70356253	0,70500016	0,69700003	0,00331737
16 processos k=2000	1,40625007	1,43362486	1,39493759	1,38974988	1,36893746	0,02375299
16 processos k=3500	2,40337504	2,38468751	2,31900001	2,36674999	2,35831252	0,03167756
16 processos k=5000	3,34681255	3,33312494	3,38537501	3,42299990	3,34000005	0,03791021
48 processos k=500	0,80666668	1,00160420	0,81672910	0,81266671	1,04281250	0,11609787
48 processos k=1000	1,58518752	1,82695831	1,62677089	1,58758340	1,58533335	0,10469093
48 processos k=2000	3,17385418	3,40297917	3,18637502	3,17385418	4,99022916	0,79132534
48 processos k=3500	5,55102090	5,55750003	5,72712502	6,52922918	5,59181251	0,41858381
48 processos k=5000	10,87010409	8,11402079	11,17735408	9,12116671	8,87816669	1,32801516

Tabela 5.12: Laplace transiente - tempo Real e desvio padrão para n=386.

O gráfico representado pelas Figura 5.28, mostra os resultados comparativos da execução serial e paralela com 4, 16 e 48 processos.

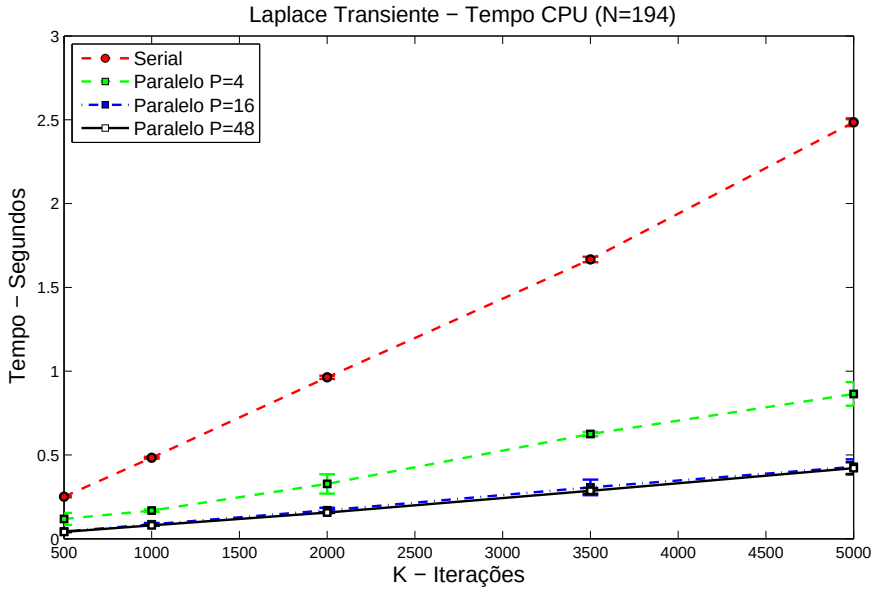

Figura 5.28: Comparação de tempo de CPU serial e paralelo para a equação de Laplace transiente com n=194.

Nota-se que em todo o intervalo de iterações as execuções em paralelo tiveram um tempo menor de CPU. Este mesmo padrão é verificado nos gráficos representados pelas

Figuras 5.29 e 5.30, sendo que o tamanho da matriz foi alterado para 290 e 386 respectivamente. Pode-se observar que nos três gráficos obteve-se o menor tempo de CPU com 48 processos.

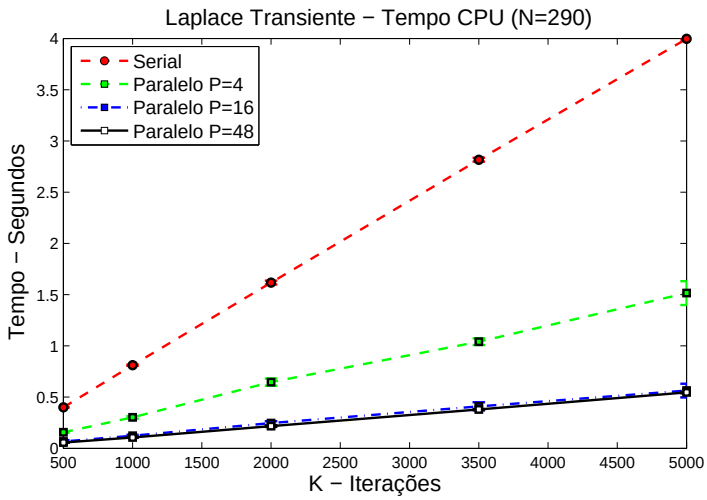

Figura 5.29: Comparação de tempo de CPU serial e paralelo para a equação de Laplace transiente com n=290.

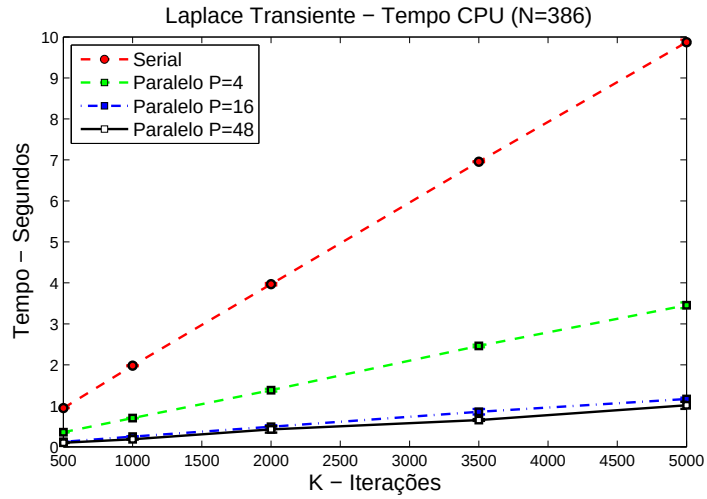

Figura 5.30: Comparação de tempo de CPU serial e paralelo para a equação de Laplace transiente com n=386.

O gráfico representado pelas Figura 5.31, mostra os resultados comparativos da execução serial e paralela com 4, 16 e 48 processos e com a matriz de tamanho 194, porém

com o tempo Real. Neste caso observa-se que o tempo com 4 e 48 processos se manteve maior que o processo serial, porém o gráfico com 16 processos se manteve com o tempo abaixo do serial.

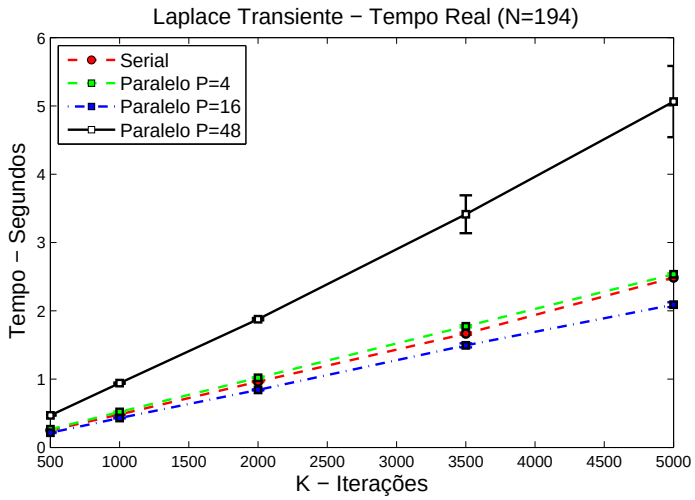

Figura 5.31: Comparação de tempo Real em relação a execução serial e paralelo da equação de Laplace transiente com n=194.

O gráfico representado pela Figura 5.32, mostra uma alteração na tendencia do tempo Real. Constata-se que com o aumento da matriz para 290 as execuções paralelas com 4 e 16 processos mantiveram abaixo do tempo Real serial.

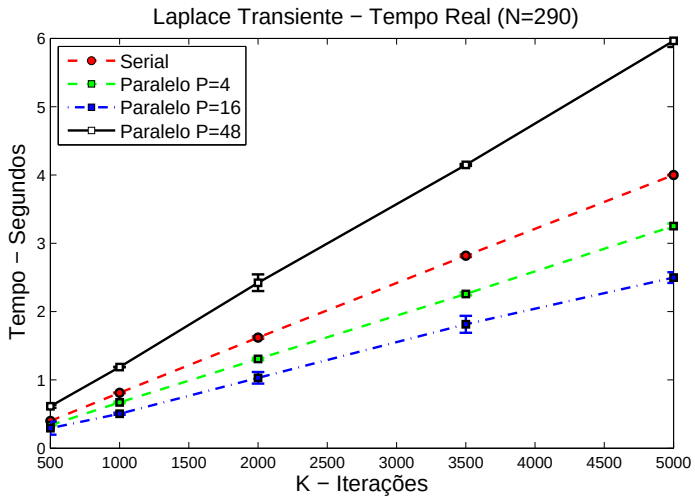

Figura 5.32: Comparação de tempo Real em relação a execução serial e paralelo da equação de Laplace transiente com n=290.

Os resultados mostrados pelo gráfico representado pela Figura 5.33, com uma matriz de 386, mostra que os tempos de execução em paralelo ficaram abaixo do tempo serial. Nota-se um desvio padrão maior em alguns pontos da execução paralela com 48 processos, atribuindo a esse fato o tempo Real que pode ser afetado por rotinas do Sistema Operacional ou processos de outros aplicativos em execução.

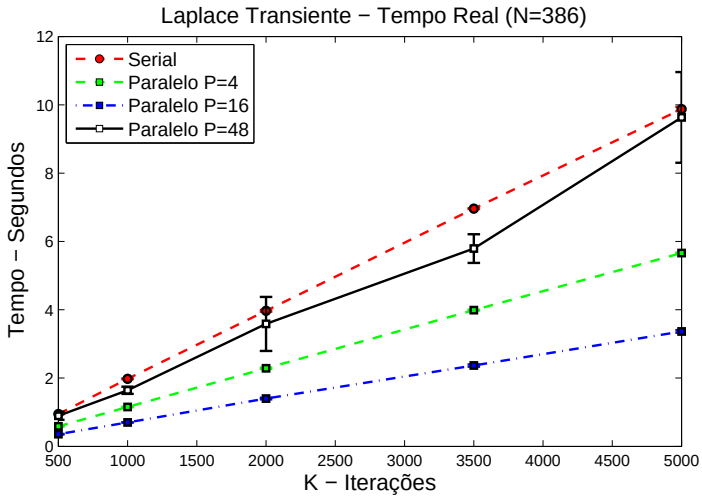

Figura 5.33: Comparação de tempo Real em relação a execução serial e paralelo da equação de Laplace transiente com n=386.

5.3.6 Verificação do Resíduo dos Métodos Iterativos na Normal L2

Os residuais dos métodos apresentados neste trabalho foram calculados de acordo com a equação 5.14 até atingir o número de dígitos significativos da precisão do computador, que é de aproximadamente 15 dígitos significativos. Utiliza-se neste cálculo a norma L2, também chamada de norma Euclidiana, representada pela equação 5.15 [39].

$$Residual = resid[K] = log10\left(\frac{l_2}{(n-1)(n-1)}\right) \tag{5.14}$$

$$l_2 = \left(\sum_{i,j=1}^{n} \left|a_{ij}^{k} - a_{ij}^{k-1}\right|^2\right)^{1/2} \tag{5.15}$$

O gráfico representado pela Figura 5.34 mostra o residual da equação de Laplace 2D estacionária, executado serialmente.

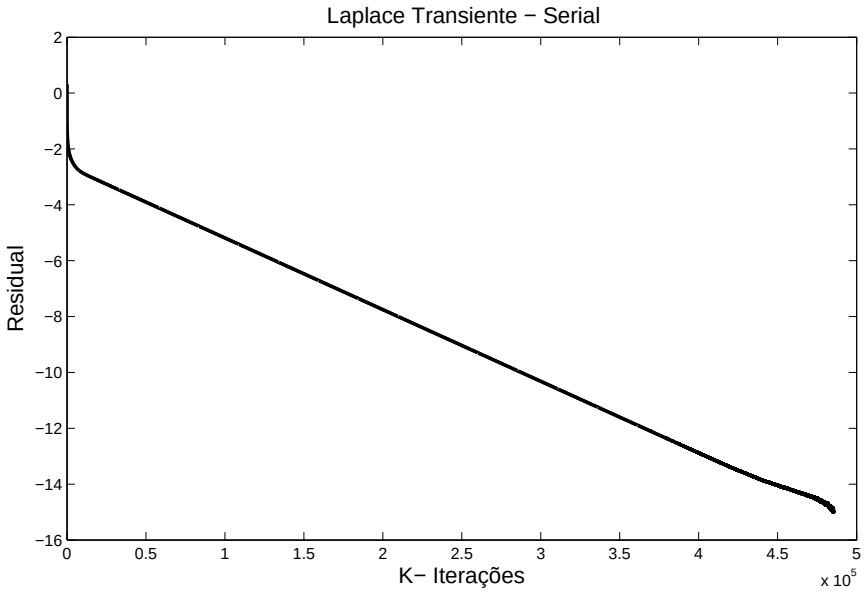

Figura 5.34: Residual Laplace estacionário - serial.

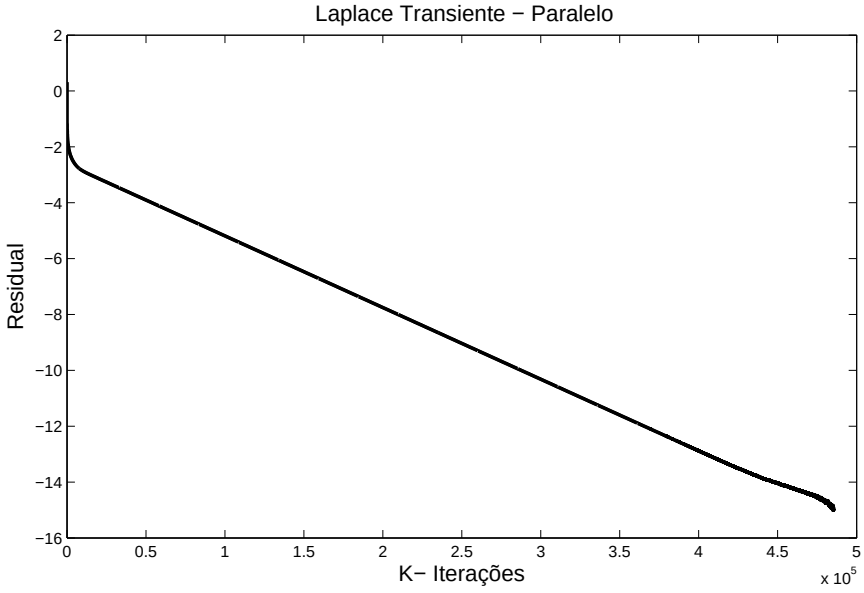

Figura 5.35: Residual Laplace estacionário - paralelo.

O gráfico representado pela Figura 5.35 mostra o residual da equação de Laplace 2D estacionária, executado paralelamente.

Nota-se que em ambos os casos os gráficos tendem para a convergência atingindo porém a capacidade computacional do sistema, referenciado no inicio desta secção.

Os gráficos 5.36 e 5.37 demonstram o residual do exemplo da Equação de Laplace transiente, serial e paralelo respectivamente.

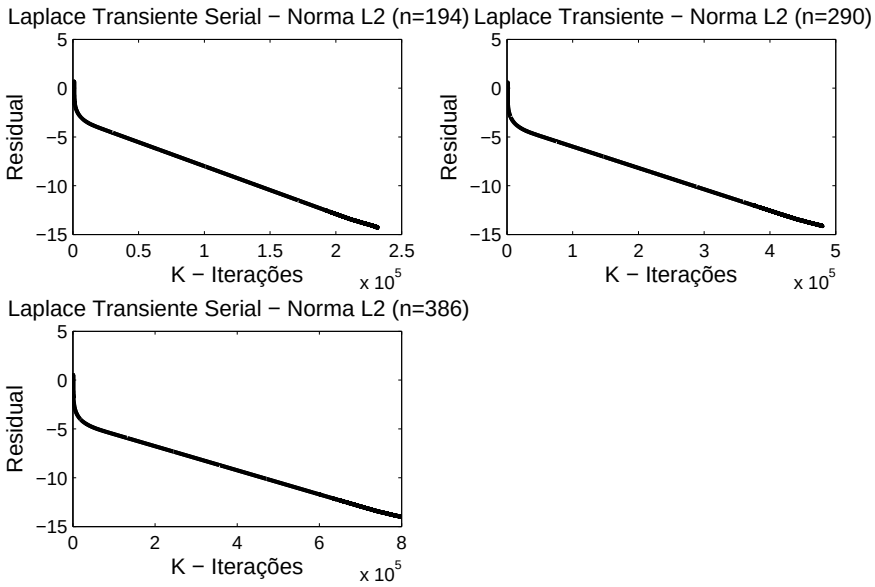

Figura 5.36: Residual Laplace transiente - serial.

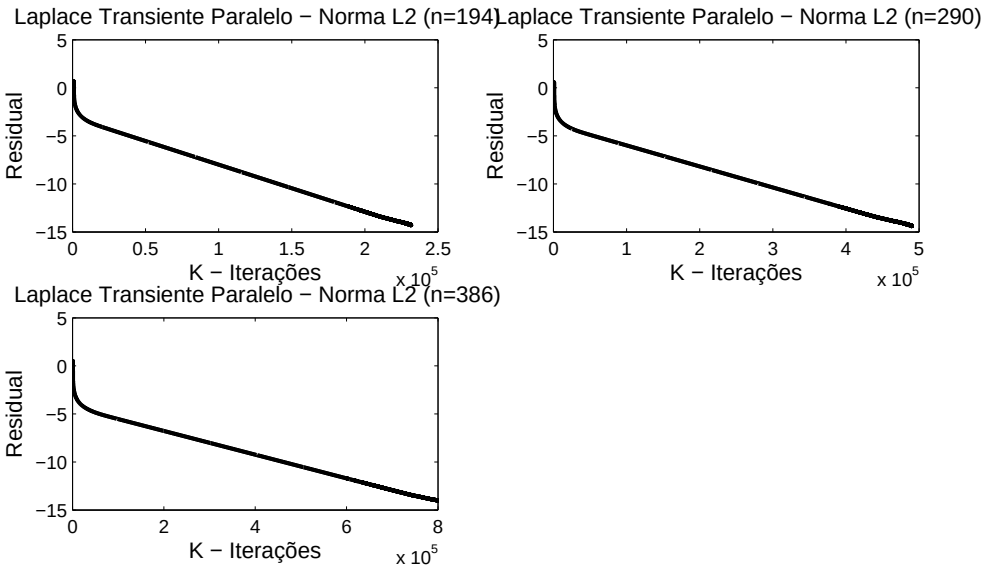

Figura 5.37: Residual Laplace transiente - paralelo.

Os resultados serão analisados no próximo capítulo, assim como avaliado o alcance dos objetivos que motivaram este trabalho.

Capítulo 6

Análise dos Resultados, Conclusões e Trabalhos Futuros

Ao longo deste trabalho foram apresentadas as principais características do processamento paralelo e distribuído, como enfatizado no Capítulo 2, como também do *Cluster* Beowulf. A implantação do *Cluster* VORTEX foi descrita no Capítulo 3, como também os desafios e problemas de *hardware* e *software* solucionados em sua instalação, gerando assim um manual técnico de instalação, anexado a este trabalho como Apêndice A.

No Capítulo 4 foi detalho os softwares de paralelização e execução dos códigos computacionais, com enfase ao OPENMPI que possibilitou a paralelização dos códigos. Foram descritas ainda várias rotinas de execução da biblioteca do OPENMPI afim de gerar exemplos práticos de sua utilização em códigos computacionais demonstrados e as inúmeras possibilidades de utilização destes recursos.

Após a implantação do *Cluster* VORTEX e detalhado todos os softwares para a paralelização dos códigos computacionais, o Capítulo 5 descreve a implementação dos códigos computacionais serias e paralelos. Os parâmetros de execução dos códigos foram definidos assim como a metodologia para verificação do tempo de execução. Para cada código e cada parâmetro foram executados e consequentemente obtidos 5 resultados, gerando o desvio padrão dos tempos de CPU e Real. Após a execução dos exemplos e a plotagem dos resultados através do software MATLAB, considerando os códigos computacionais seriais e paralelos e seus tempos de execução, pode-se analisar e comparar alguns aspectos, relacionando os resultados e conclusões.

6.1 Análise dos Resultados

Regra do Trapézio

Através da paralelização da regra do Trapézio obteve-se uma diminuição significativa de tempo Real e do tempo CPU em relação ao código serial. Calcula-se a variação percentual (VP), através da Equação 6.1, dos tempos paralelos em relação aos tempos seriais [35]:

$$VP = \frac{T_p - T_s}{T_s} \tag{6.1}$$

onde:

Tp -> tempo paralelo.

Ts -> tempo serial.

Observa-se na Tabela 6.1 que o melhor tempo paralelo de CPU e Real foram obtidos com 48 processos, com uma melhora de -97% em relação ao tempo serial. Nota-se uma tendência da diminuição do tempo de execução do código paralelo em relação ao código serial a medida que aumentamos o número de processos.

Trapézio - Tempo CPU		
Processos	**Média Tempos**	**Variação**
Serial	0,25596080	-
Paralelo - Método MPI_Send e Recv	**Média Tempos**	**Variação**
2	0,09368560	-63%
4	0,04354335	-83%
8	0,02724580	-89%
16	0,01548504	-94%
32	0,00864251	-97%
48	0,00625324	-98%
Paralelo - Método MPI_Reduce	**Média Tempos**	**Variação**
2	0,09138610	-64%
4	0,04154365	-84%
8	0,02599593	-90%
16	0,01648486	-94%
32	0,00878001	-97%
48	0,00668647	-97%
Trapézio - Tempo REAL		
Processos	**Média Tempos**	**Variação**
Serial	0,25599995	-
Paralelo - Método MPI_Send e Recv	**Média Tempos**	**Variação**
2	0,13110003	-49%
4	0,06719999	-74%
8	0,03385010	-87%
16	0,01638748	-94%
32	0,00937501	-96%
48	0,00691251	-97%
Paralelo - Método MPI_Reduce	**Média Tempos**	**Variação**
2	0,12629991	-51%
4	0,06304996	-75%
8	0,03317500	-87%
16	0,02042501	-92%
32	0,01006879	-96%
48	0,00793337	-97%

Tabela 6.1: Trapézio - tempo CPU e Real - variação percentual

Com o aumento do número de processos na execução do código paralelo observa-se uma diminuição também no erro do resultado, em relação ao resultado serial.

Não houve diferença significativa do resultado numérico em relação aos códigos serial e paralelo. Em relação aos tempos de CPU e tempo Real de execução, quando se utiliza diferentes métodos de paralelização com OPENMPI (*MPI_Reduce ou MPI_Send/MPI_Recv*), não foi observado diferença significativa do resultado.

Equação de Laplace Estacionário

Na execução dos códigos computacionais paralelos obteve-se uma diminuição significativa do tempo de CPU em relação ao código serial.

Observa-se na Tabela 6.2 que o melhor tempo paralelo de CPU e Real foi obtido com 16 processos, com uma melhora de -84% e -26% respectivamente em relação ao tempo serial.

Laplace Estacionário - Tempo CPU - variação percentual		
Serial/Paralelo Laplace CPU	**Média Tempos**	**Variação**
Serial	5,30799240	-
4 processos	1,89686150	-64%
8 processos	1,26650737	-76%
16 processos	0,87039267	-84%
32 processos	0,96492209	-82%
48 processos	1,07134129	-80%
Laplace Estacionário - Real		
Serial/Paralelo Laplace Real	**Média Tempos**	**Variação**
Serial	5,31039996	-
4 processos	4,75000000	-11%
8 processos	3,97987505	-25%
16 processos	3,92956252	-26%
32 processos	6,51199378	23%
48 processos	9,81752886	85%

Tabela 6.2: Laplace estacionário - tempo CPU e Real - variação percentual

Em relação aos resultados de tempo Real de execução nota-se que o tempo do código paralelo com 32 e 48 processos se encontra acima do tempo serial. Percebe-se este aumento do tempo Real em decorrência do tempo utilizado para envio e recebimento de mensagens entre os processos. Para os demais tempos os valores da execução em paralelo ficaram abaixo da execução serial.

Equação de Laplace Transiente

Como no exemplo anterior obteve-se significativa e desejável diminuição do tempo Real de execução e de CPU do código paralelo em relação ao código serial.

Em relação ao tempo de CPU nota-se que conforme a Tabela 6.3 o melhor tempo foi observado com uma matriz quadrada de 386 (n) e 48 processos em todas as iterações (k) do experimento. Esse mesmo comportamento é observado com n = 290 e 194, conforme as Tabelas 6.4 e 6.5. Observa-se também que todos os tempos de CPU em paralelo ficaram significativamente abaixo dos tempos de execução serial.

Laplace Transiente - Tempo CPU - N=386		
Serial/Paralelo	**Média Tempos**	**Variação**
Serial k=500	0,94585640	-
4 processos k=500	0,35649575	-62%
16 processos k=500	0,12056926	-87%
48 processos k=500	0,09795596	-90%
Serial k=1000	1,97949940	-
4 processos k=1000	0,70039350	-65%
16 processos k=1000	0,24903716	-87%
48 processos k=1000	0,18427617	-91%
Serial k=2000	3,96719700	-
4 processos k=2000	1,37999025	-65%
16 processos k=2000	0,48896313	-88%
48 processos k=2000	0,42489793	-89%
Serial k=3500	6,95714240	-
4 processos k=3500	2,46137570	-65%
16 processos k=3500	0,85541996	-88%
48 processos k=3500	0,65405472	-91%
Serial k=5000	9,86989960	-
4 processos k=5000	3,45367485	-65%
16 processos k=5000	1,16623522	-88%
48 processos k=5000	1,01293767	-90%

Tabela 6.3: Laplace transiente - tempo CPU n=386

Laplace Transiente - Tempo CPU - N=290		
Serial/Paralelo	**Média Tempos**	**Variação**
Serial k=500	0,40033900	-
4 processos k=500	0,15652600	-61%
16 processos k=500	0,06683976	-83%
48 processos k=500	0,05633730	-86%
Serial k=1000	0,81107720	-
4 processos k=1000	0,30250410	-63%
16 processos k=1000	0,12359370	-85%
48 processos k=1000	0,10597967	-87%
Serial k=2000	1,61795440	-
4 processos k=2000	0,64630190	-60%
16 processos k=2000	0,24608756	-85%
48 processos k=2000	0,21722529	-87%
Serial k=3500	2,81677220	-
4 processos k=3500	1,04124175	-63%
16 processos k=3500	0,41001264	-85%
48 processos k=3500	0,37922987	-87%
Serial k=5000	3,99739260	-
4 processos k=5000	1,51546975	-62%
16 processos k=5000	0,56407672	-86%
48 processos k=5000	0,54544205	-86%

Tabela 6.4: Laplace transiente - tempo CPU n=290

Laplace Transiente - Tempo CPU - N=194		
Serial/Paralelo	**Média Tempos**	**Variação**
Serial k=500	0,25076200	-
4 processos k=500	0,11818510	-53%
16 processos k=500	0,04345591	-83%
48 processos k=500	0,04075626	-84%
Serial k=1000	0,48332660	-
4 processos k=1000	0,16803060	-65%
16 processos k=1000	0,08636191	-82%
48 processos k=1000	0,08009197	-83%
Serial k=2000	0,96325360	-
4 processos k=2000	0,32710330	-66%
16 processos k=2000	0,16968677	-82%
48 processos k=2000	0,15630955	-84%
Serial k=3500	1,66634700	-
4 processos k=3500	0,62501235	-62%
16 processos k=3500	0,30644097	-82%
48 processos k=3500	0,28606065	-83%
Serial k=5000	2,48502260	-
4 processos k=5000	0,86405300	-65%
16 processos k=5000	0,42904726	-83%
48 processos k=5000	0,42125672	-83%

Tabela 6.5: Laplace transiente - tempo CPU n=194

Percebe-se um outro comportamento em relação ao tempo de Real. Nota-se que, conforme a Tabela 6.6, o melhor tempo foi observado com uma matriz quadrada de 386 (n) e 16 processos em todas as iterações (k) do experimento. Os tempos oscilaram com melhora entre -62% e -66% em relação ao tempo serial. Esse mesmo comportamento é observado com n = 290, conforme a Tabela 6.7, porém no caso da

execução com 48 processos o tempo paralelo tende a ser maior que o tempo serial, oscilando entre 47% e 53% a mais.

Laplace Transiente - Tempo Real - N=386		
Serial/Paralelo	**Média Tempos**	**Variação**
Serial k=500	0,94600001	-
4 processos k=500	0,58884997	-38%
16 processos k=500	0,35779997	-62%
48 processos k=500	0,89609584	-5%
Serial k=1000	1,97939997	-
4 processos k=1000	1,15235002	-42%
16 processos k=1000	0,70088754	-65%
48 processos k=1000	1,64236669	-17%
Serial k=2000	3,96900001	-
4 processos k=2000	2,28785003	-42%
16 processos k=2000	1,39869997	-65%
48 processos k=2000	3,58545834	-10%
Serial k=3500	6,96159997	-
4 processos k=3500	3,98874999	-43%
16 processos k=3500	2,36642501	-66%
48 processos k=3500	5,79133753	-17%
Serial k=5000	9,87439999	-
4 processos k=5000	5,66189996	-43%
16 processos k=5000	3,36566249	-66%
48 processos k=5000	9,63216247	-2%

Tabela 6.6: Laplace transiente - tempo Real n=386

Laplace Transiente - Tempo Real - N=290		
Serial/Paralelo	**Média Tempos**	**Variação**
Serial k=500	0,40000000	-
4 processos k=500	0,33610002	-16%
16 processos k=500	0,29326245	-27%
48 processos k=500	0,61371252	53%
Serial k=1000	0,81080003	-
4 processos k=1000	0,67024997	-17%
16 processos k=1000	0,50472497	-38%
48 processos k=1000	1,18837916	47%
Serial k=2000	1,61879992	-
4 processos k=2000	1,30640001	-19%
16 processos k=2000	1,03023747	-36%
48 processos k=2000	2,42279585	50%
Serial k=3500	2,81760001	-
4 processos k=3500	2,25840002	-20%
16 processos k=3500	1,81400001	-36%
48 processos k=3500	4,14772501	47%
Serial k=5000	3,99900007	-
4 processos k=5000	3,25169998	-19%
16 processos k=5000	2,49890001	-38%
48 processos k=5000	5,96373752	49%

Tabela 6.7: Laplace transiente - tempo Real n=290

Pode-se observar que esta tendência permanece na Tabela 6.5, onde o melhor tempo foi constatado com 16 processos, porém ouve uma diminuição do ganho de tempo do código paralelo em relação ao serial. Constata-se também que com 4 e 48 processos os resultados foram acima do tempo serial.

Laplace Transiente - Tempo Real - N=194		
Serial/Paralelo	**Média Tempos**	**Variação**
Serial k=500	0,25080004	-
4 processos k=500	0,26570003	6%
16 processos k=500	0,21625002	-14%
48 processos k=500	0,46964999	87%
Serial k=1000	0,48320003	-
4 processos k=1000	0,52010000	8%
16 processos k=1000	0,43007501	-11%
48 processos k=1000	0,94113332	95%
Serial k=2000	0,96440000	-
4 processos k=2000	1,02030002	6%
16 processos k=2000	0,84202496	-13%
48 processos k=2000	1,87771667	95%
Serial k=3500	1,66640000	-
4 processos k=3500	1,77444997	6%
16 processos k=3500	1,49476250	-10%
48 processos k=3500	3,41393750	105%
Serial k=5000	2,48620000	-
4 processos k=5000	2,53505000	2%
16 processos k=5000	2,08990002	-16%
48 processos k=5000	5,06324167	104%

Tabela 6.8: Laplace transiente - tempo Real n=194

Intui-se neste exemplo que existe uma relação direta entre o tamanho da matriz e o ganho de tempo com a paralelização do código serial. Justifica-se esse fato devido ao tempo de comunicação entre os processos, registrado no tempo Real, tendo em vista o tamanho da matriz.

6.2 Conclusões

A construção e implementação do *Cluster* VORTEX baseado no sistema Beowulf foi concluída com sucesso. O sistema é estável e confiável atendendo a demanda de paralelização dos usuários. A instalação dos sistemas auxiliares seguiram o estabelecido nos seus devidos manuais, com intervenções que estão relatadas neste trabalho.

Foi atingido o objetivo de diminuição de tempo Real de execução e de tempo de CPU em relação aos códigos seriais. Em relação aos resultados de melhoria de tempo do código paralelo em relação ao código serial, nota-se que a diminuição de tempo possui direta relação com:

1. o tamanho do vetor ou matriz paralelizada no código computacional, sendo que tende a ser mais eficiente com o aumento das dimensões.
2. o número de processos estabelecidos na execução, que dependendo de cada problema existe uma quantidade que resulta em um melhor resultado de tempo
3. a quantidade de mensagens utilizadas pelos processos, gerando assim tráfego de rede e *overhead*, afetando diretamente o desempenho da paralelização.

6.3 Trabalhos Futuros

Para trabalhos futuros sugere-se alguns itens que podem ser estudados e ampliam a pesquisa em computação paralela e distribuída com foco em Clusters utilizando Rocks.

1. Paralelização de códigos computacionais afim de solucionar problemas com grandes matrizes, NxN » 386.
2. Virtualização do *Cluster Rocks* para utilização com execução de vários códigos computacionais paralelas simultaneamente.
3. Instalação de um *Cluster Rocks* em nuvem e o comparativo de desempenho entre *Clusters* em rede local privada e em nuvem.

Referências

[1] Anshul Gupta Vipin Kumar Ananth Grama, George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison-Wesley, 2003. ISBN 0201648652.

[2] L. Ridgway Scott, Terry Clark, and Babak Bagheri. Scientific parallel computing. Princeton University Press, Princeton, NJ, 2005.

[3] M. Pitanga. *Construindo Supercomputadores com Linux*. BRASPORT. ISBN 9788574523729.

[4] A.S. Tanenbaum. *Organização estruturada de computadores*. Pearson Prentice Hall, 2007. ISBN 9788576050674.

[5] C. Taurion. *Cloud Computing - Computação em Nuvem*. BRASPORT, 2009. ISBN 9788574524238.

[6] Marcel Fagundes Souza. Estudo de um modelo de cluster baseado em um ambiente wmpi. *UNIMINAS*, 2004.

[7] A.S. Tanenbaum. *Distributed Systems: Principles and Paradigms*, volume 2. Globo, 2006. 2 pp.

[8] Daniela de Campos Quitete Vianna. Um sistema de gerenciamento de aplicações mpi para ambientes grid. *UFF - Niteroi*, 1:166, 2005.

[9] Ignacio Fernandez Martinez. Creacion y validacion de um cluster de computacion cientifica baseado em rocks. *Escuela Politecnica Superior, Leganes, Universidad Carlos III de Madrid*, 2009.

[10] Johnny Marcus Gomes Rocha. Cluster beowulf: Aspectos de projeto e implementação. *UFPA – CT - PPGEE*, 2003.

[11] Marco Haurelio. *A SAGA DE BEOWULF*. Aquariana, 2013.

[12] The Rocks Cluster Group at the San Diego Supercomputer Center. *Rocks Base Users Guide 5.4.3*. University of California, Aug 11 2011. Disponível em:http://www.rocksclusters.org/roll-documentation/base/5.4.3/. Acesso em: 02/06/2013.

[13] University of California. http://www.rocksclusters.org/, 2013. Disponível em:http://www.rocksclusters.org/. Acesso em: 05/06/2013.

[14] University of California Regents. http://www.rocksclusters.org/presentations/tutorial/tutorial-1.pdf, 2008.

[15] J.F. Kurose and K.W. Ross. *Redes de computadores e a Internet: uma abordagem top-down*. Pearson Addison Wesley, 2006. ISBN 9788588639188.

[16] University of California. Hpc users guide, 2011. Disponível em:http://www.rocksclusters.org/roll-documentation/hpc/5.4.3/. Acesso em: 03/06/2013.

[17] University of California. Area51 users guide, 2011. Disponível em:http://www.rocksclusters.org/roll-documentation/area51/5.4.3/. Acesso em: 03/06/2013.

[18] University of California. Biousersguide, 2013. Disponível em:http://www.rocksclusters.org/roll-documentation/bio/5.4.3/. Acesso em: 05/06/2013.

[19] Htcondor high throughput computing, 2013. Disponível em: http://research.cs.wisc.edu/htcondor/index.html. Acesso em: 03/06/2013.

[20] University of California. Ganglia monitoring system, 2013. Disponível em: http://ganglia.sourceforge.net/ . Acesso em: 05/06/2013.

[21] University of California. Perl roll: Users guide, 2011. Disponível em:http://www.rocksclusters.org/roll-documentation/perl/5.4.3/index.html. Acesso em: 05/06/2013.

[22] Python Software Foundation. Python programming language - site oficial, 2013. Disponível em: http://www.python.org/. Acesso em: 05/06/2013.

[23] Oracle Grid Engine. Sun grid engine, 2013. Disponível em: http://www.oracle.com/technetwork/oem/grid-engine-support-215299.html. Acesso em: 05/06/2013.

[24] Roy Dragseth. Torque roll documentation, 2010. Disponível em: http://neams.rpi.edu/roll-documentation/torque/5.4/torque-roll-userguide.html. Acesso em: 03/06/2013.

[25] Linux Fundation Collaborative Projects. Xen project, 2013. Disponivel em: http://xen.org/. Acesso em: 03/06/2013.

[26] University of California. Xen users guide, 2011. Disponível em: http://www.rocksclusters.org/roll-documentation/xen/5.4.3/index.html. Acesso em: 05/06/2013.

[27] A.S. Tanenbaum. *Redes de computadores*. Elsevier, 2003. ISBN 9788535211856.

[28] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, second edition, 1999. ISBN ISBN: 0-262-57134-3.

[29] Paul Burton. *An Introduction to MPI Programming*. ECMWF, 2012. Disponível em: http://www.ecmwf.int/services/computing/training/material/hpcf/. Acesso em: 21/07/2013.

[30] Jeffrey M. Squyres. *The Archicture of Open Source Applications*, volume ii, chapter 15. Self published, April 2012.

[31] H.M. Deitel. *Como Programar em C*. LTC, 1999. ISBN 9788521611912.

[32] GNU Compiler Collection GCC. The gnu compiler collection, 2012. Disponível em <http://gcc.gnu.org/>. Acesso em 10/10/2012.

[33] MathWorks. Visão global matlab, 2013. Disponível em:http://http://www.mathworks.com/products/matlab/. Acesso em: 24/07/2013.

[34] The Open Group Base Specifications Issue 6. http://pubs.opengroup.org/onlinepubs/009695399/mindex.html, 2004. Disponível em: http://pubs.opengroup.org/onlinepubs/009695399/mindex.html. Acesso em: 29/07/2013.

[35] Sonia Maria Barros Barbosa Correa. *Probabilidade e Estatística*. 2 edition, 2003. 116p pp.

[36] Foulis David J. Nunem, Mustafa A. *CALCULO 1*. Number v. 1. LTC, 1 edition, 1982. ISBN 9788521610540. 605 pp.

[37] Robert E. White. *Computational mathematics - models, methods, and analysis with MATLAB and MPI.* Chapman&Hall/CRC Press, 2004. ISBN 978-1-58488-364-7. I-XVI, 1-385 pp.

[38] N.B. Franco. *Cálculo Numérico*. Pearson, 2006. ISBN 9788576050872.

[39] João Manuel Patrício Joaquim João Júdice. Sistemas de equações lineares. *Departamento de Matemática da Universidade de Coimbra*, 1996.

[40] University of California. Base users guide, 2011. Disponível em:http://www.rocksclusters.org/roll-documentation/base/5.4.3/. Acesso em: 29/07/2013.

APÊNDICE A – Procedimentos de Instalação do Rocks no Cluster VORTEX

Neste apêndice estão aoscritos os procedimentos de instalação do Rocks e seus componentes nos computadores do *Cluster* VORTEX:

A.1 Instalando o *frontend*

1. Insira o CD denominado *Kernel/Boot* configure o *setup* para na opção de *Boot* pelo CD/DVD.

2. A Figura A.1 apresenta a tela para escolher trabalhar com IP dinâmico (DHCP) ou IP Manual. A opção DHCP é sempre mais interessante quando em sua rede existe um servidor DHCP, pois o servidor reconhece os computadores na rede e configura seu endereço IP automaticamente. Devemos lembrar que esta configuração é para a interface de rede ligada a internet. Desabilitar a opção *Enable* IPv6.

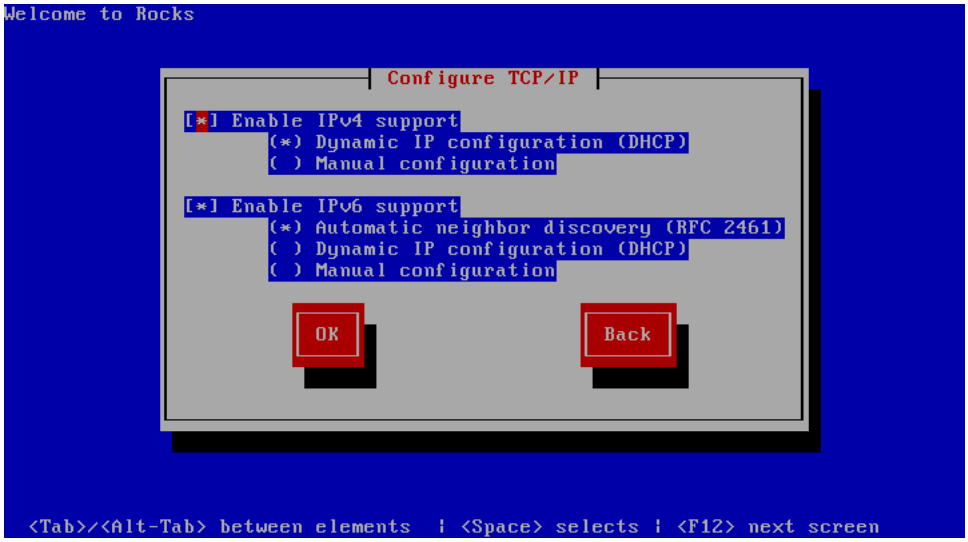

Figura A.1: Tela de boas vindas do *Rocks* [40].

3. Para instalação com o CD escolha a opção CD/DVD *Base Roll*, como mostrado na Figura A.2. Caso necessite de instalar via internet escolha a opção *Download*. Inserir o CD/DVD Base *Roll* e clicar em Continue, conforme indicado na Figura A.3.

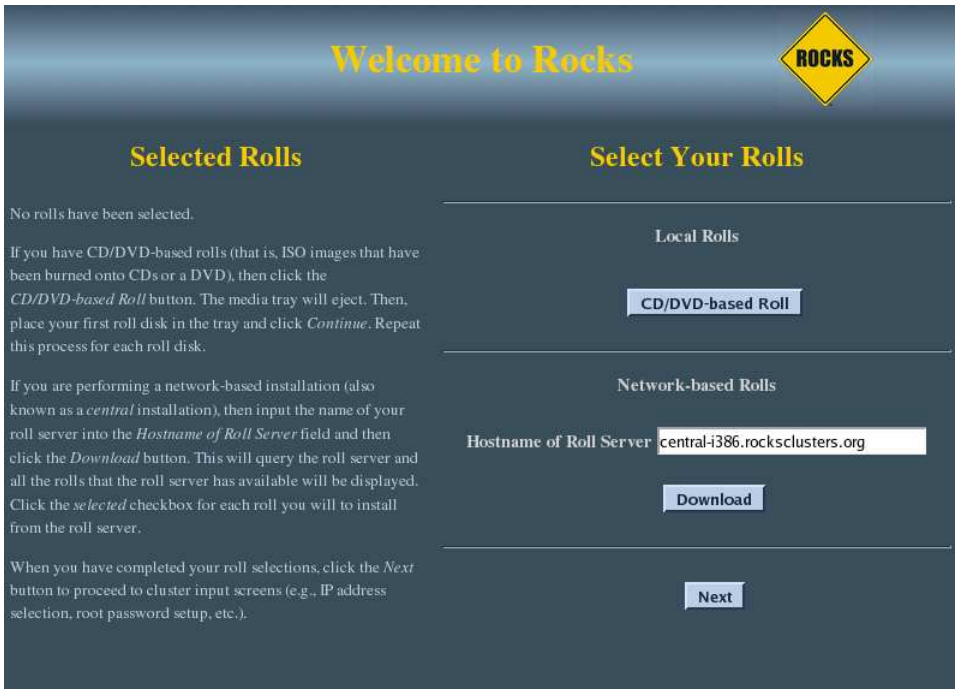

Figura A.2: Seleção de *Rolls* [40].

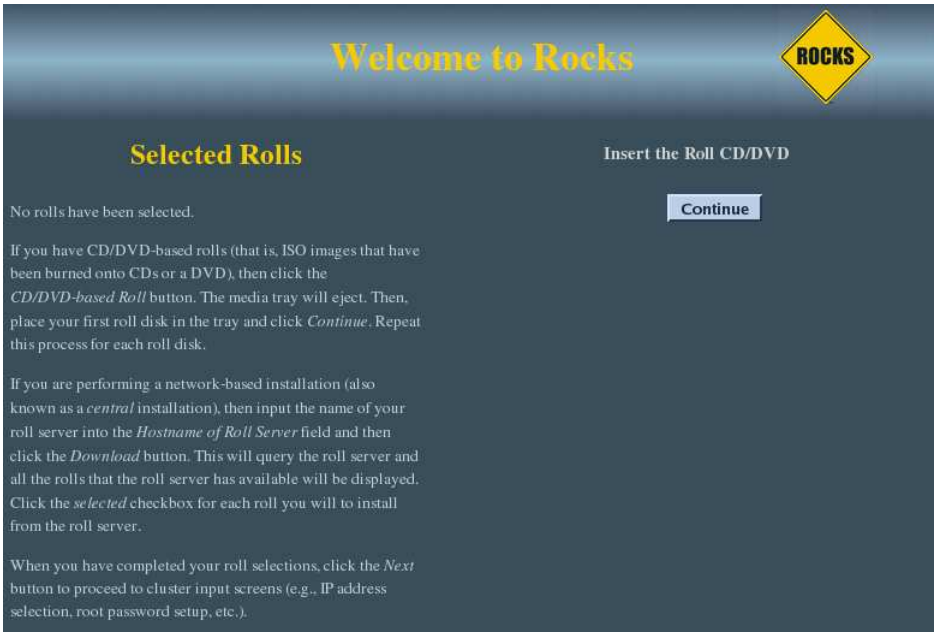

Figura A.3: Inserção do disco *Rolls* [40].

4. Para selecionar os pacotes a serem instalados, mostrados nas Figuras A.4 e A.5, coloca-se o disco com os devidos complementos (pode-se utilizar mídias separadas ou a mídia com todas as instalações – DVD jumbo). Outra opção e selecionar os arquivos a serem instalados através da internet, utilizando o botão Download e marcando as opções desejadas.

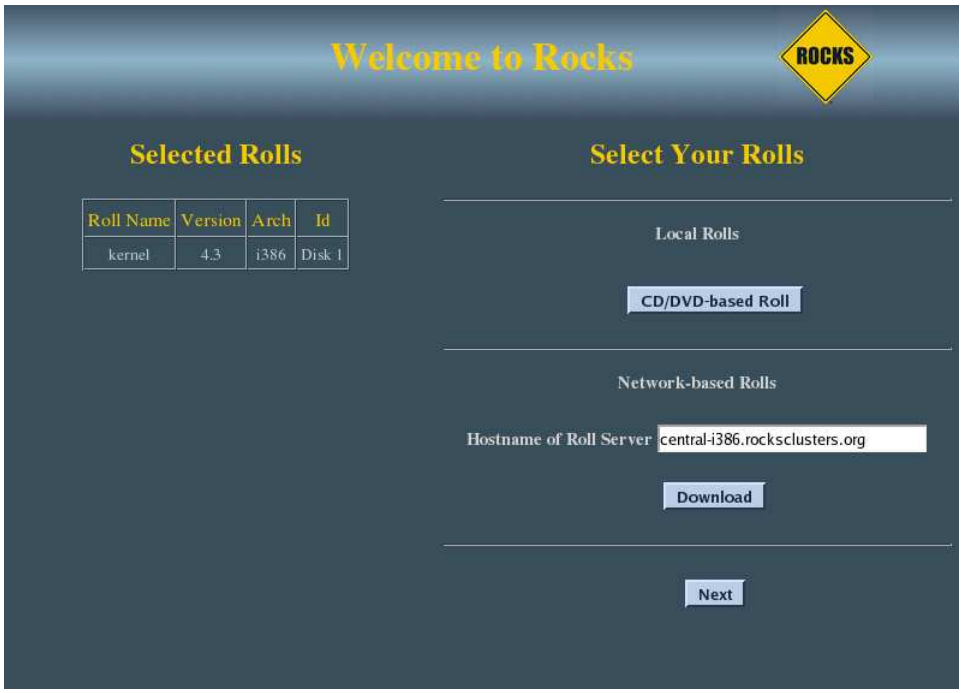

Figura A.4: *Rolls* carregados para instalação [40].

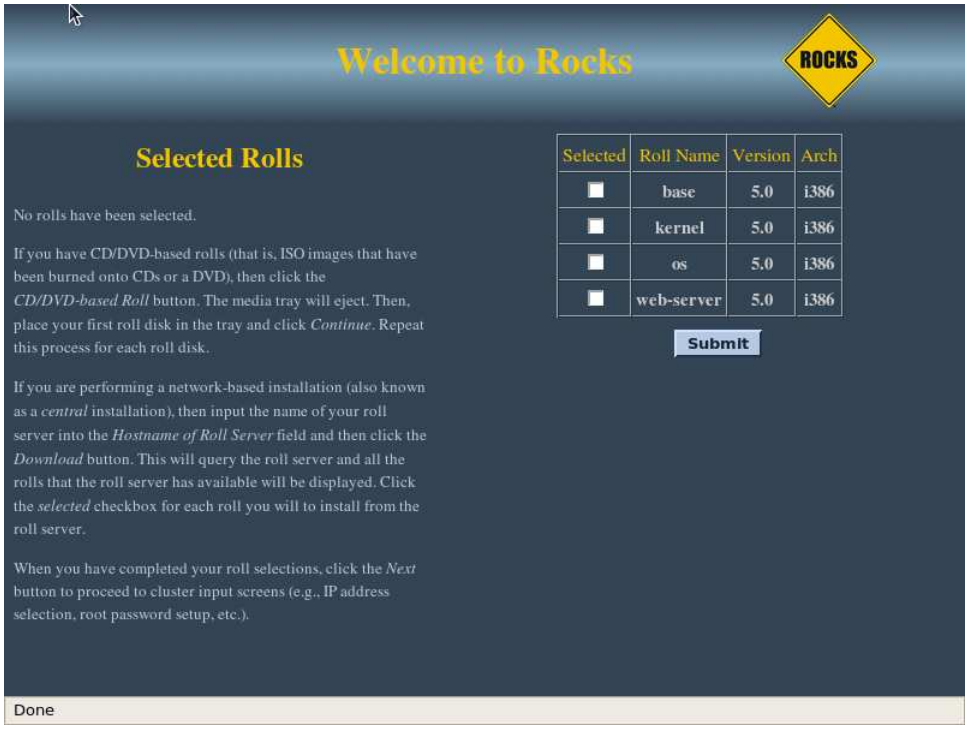

Figura A.5: Lista de *Rolls* disponíveis [40].

5. Na tela mostrada na Figura A.6 inserimos as informações sobre o *Cluster*, incluindo seu nome, localização, e-mail, e demais dados citados no Capítulo 3, deverão ser introduzidos neste passo.

Figura A.6: Informações do *Cluster* [40].

6. Configura-se o endereço IP e a *Netmask* da interface de rede com a Internet, conforme a Figura A.7, geralmente a eth0. Caso esteja em uma rede com servidor DHCP este endereço será configurado automaticamente. Configurar também o endereço IP da interface com a rede LAN (*Local Area Network*) dos nós (se não for atribuído nenhum endereço o *frontend* definirá automaticamente para toda a rede – recomendado), indicado na Figura A.8. Na próxima tela, Figura A.9, configurar o *Gateway* e o DNS da rede local interna (LAN).

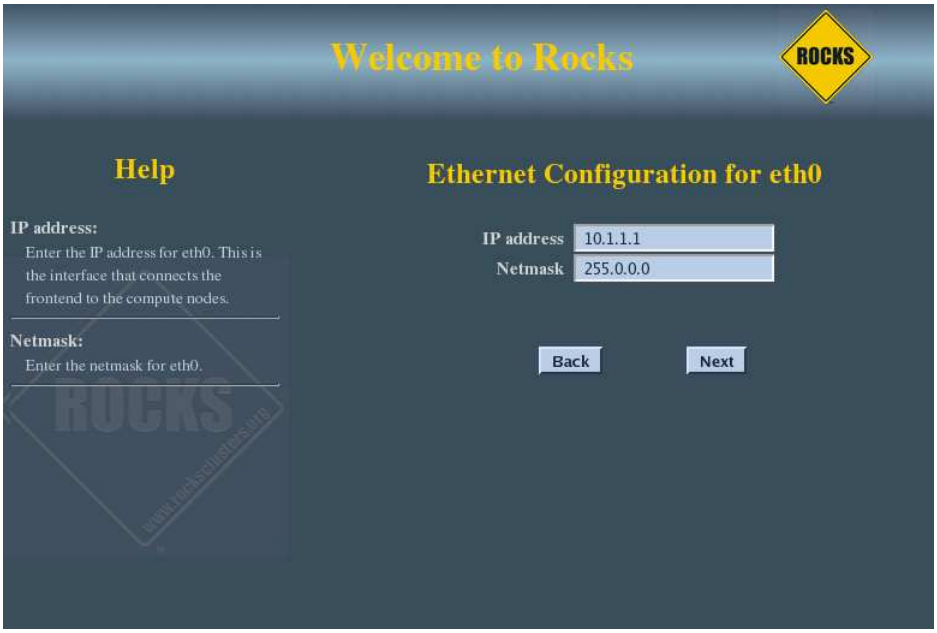

Figura A.7: Configurações da interface de rede eth0 [40].

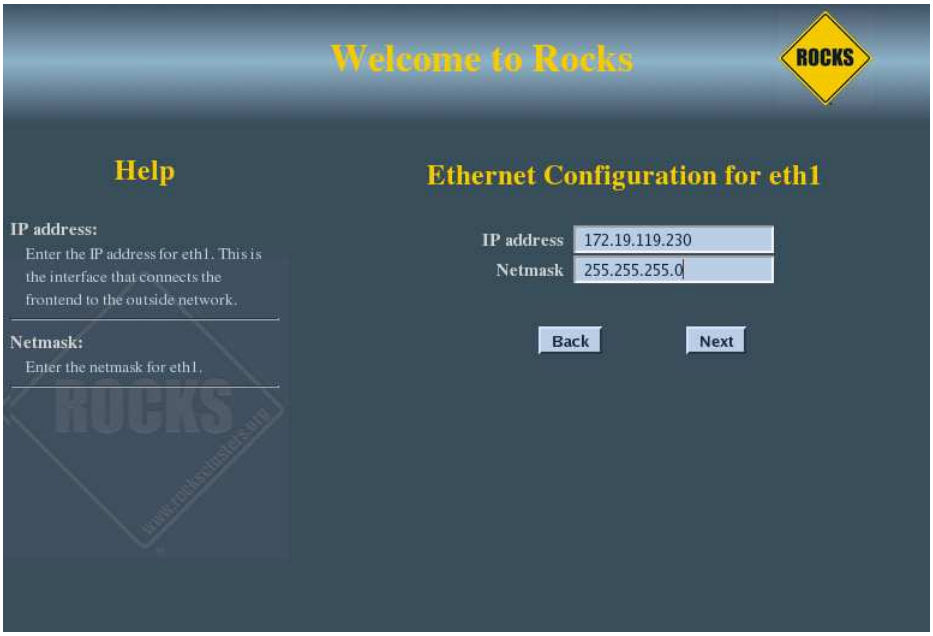

Figura A.8: Configurações da interface de rede eth1 [40].

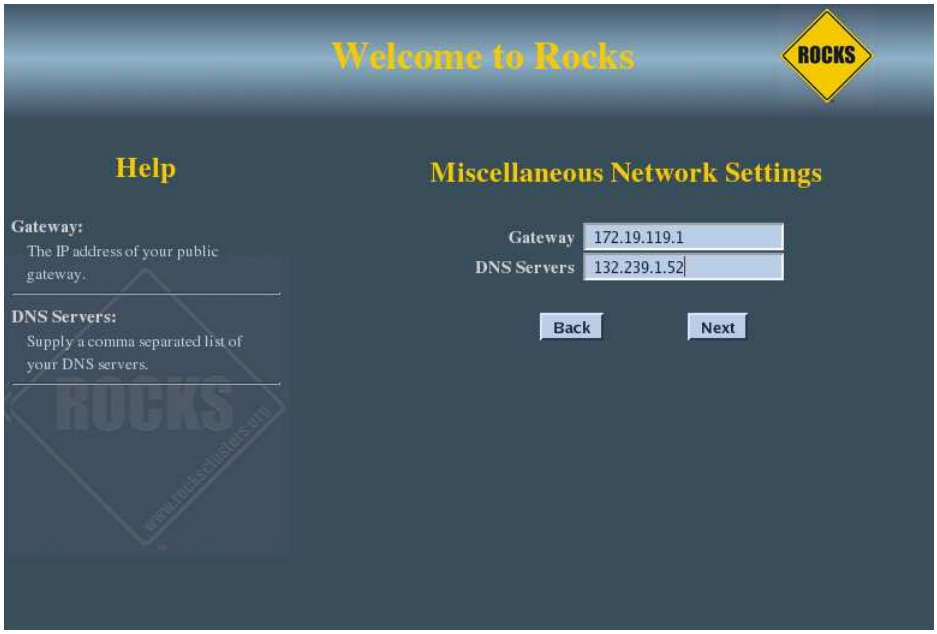

Figura A.9: Configurações de Gateway e DNS [40].

7. Na Figura A.10 definir a senha do *root* (super usuário) e as configurações de data e hora na tela mostrada pela Figura A.11. Utilizamos no Brasil o NTP *Server*: a.ntp.br.

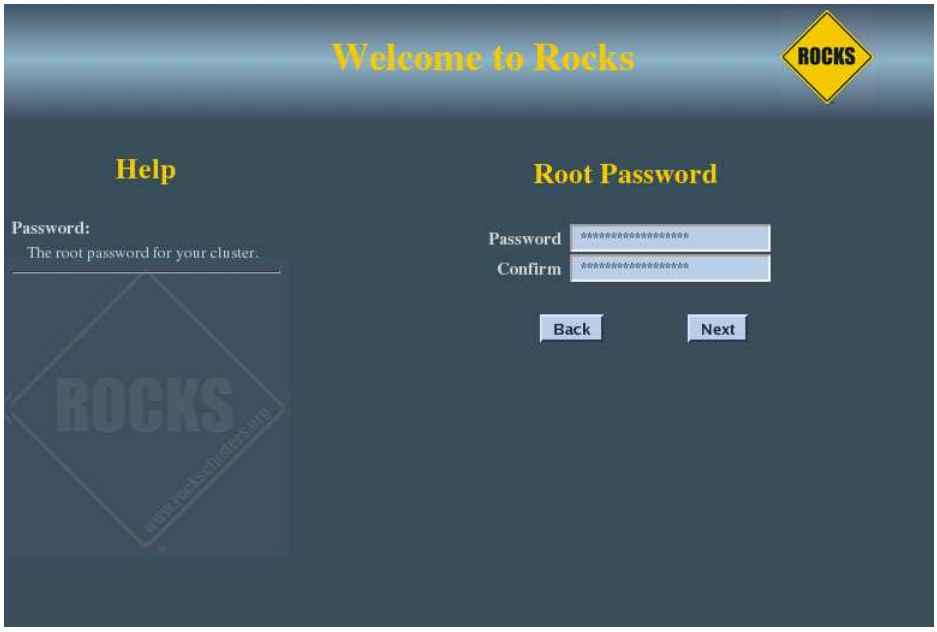

Figura A.10: Configurações da senha do *root* [40].

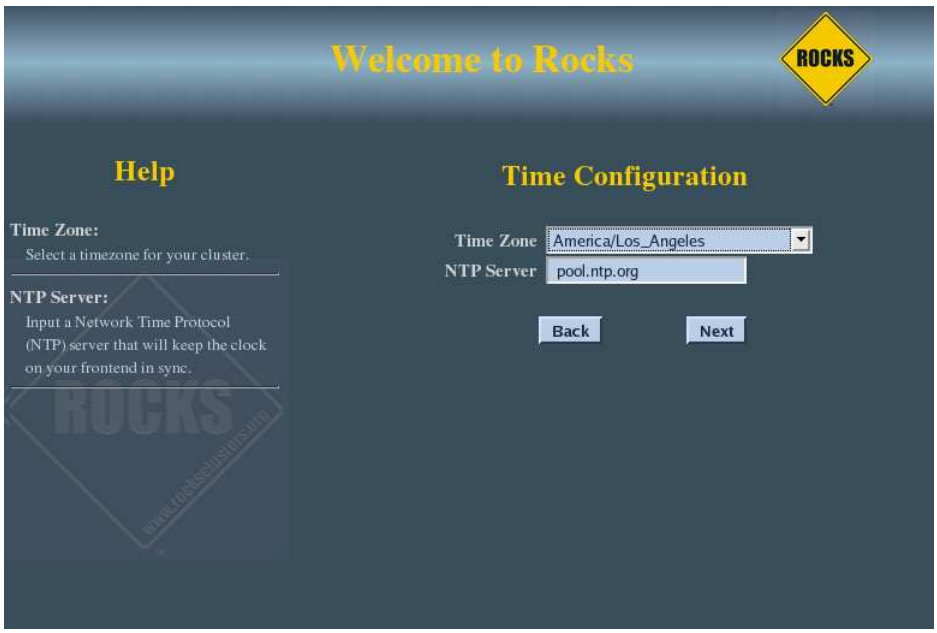

Figura A.11: Configurações do servidor de data e hora [40].

8. Nesta etapa será efetuado o particionamento do disco rígido, conforme mostrada na Figura A.12. Recomenda-se efetuar o particionamento automático, porém tem-se a opção de particionamento manual. Nesta opção manual deve-se levar em consideração o tamanho para as partições, conforme recomendações, definido por seus desenvolvedores [40]: Partição / - 16GB, partição /var - 4GB, Swap - 1GB e partição /export (todo o restante do disco).

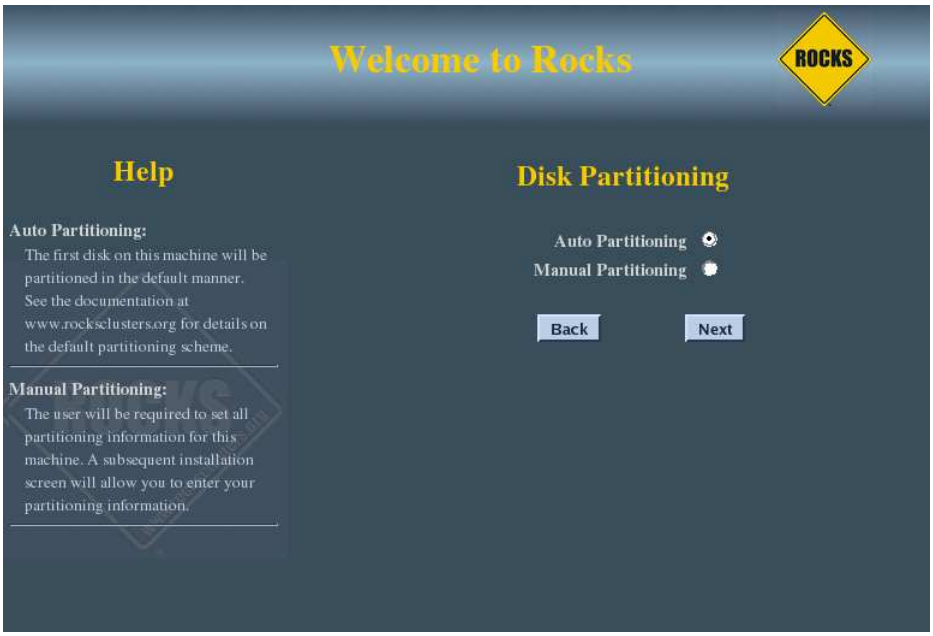

Figura A.12: Configurações do particionamento do disco rígido [40].

9. Após a instalação, o *fronted* irá reiniciar.

A.2 Instalando os Nós do *Cluster*

1. No terminal *shell*, verificar a listagem de computadores já instalados, como na listagem de *host*, mostrado na Figura A.13.
   #rocks list host

```
[root@engenhariavr ~]# rocks list host
HOST           MEMBERSHIP CPUS RACK RANK RUNACTION INSTALLACTION
engenhariavr: Frontend   8    0    0    os         install
compute-0-0:  Compute    8    0    0    os         install
compute-0-1:  Compute    8    0    1    os         install
compute-0-2:  Compute    8    0    2    os         install
compute-0-3:  Compute    8    0    3    os         install
compute-0-4:  Compute    8    0    4    os         install
```

Figura A.13: Resultado do comando *rocks list host* [40].

2. Em caso de reinstalação de nós e esta listagem já esteja preenchida com o computador que você deseja efetuar a instalação novamente, é necessário remove-lo da listagem, com o comando:
   #insert-ethers –remove="nome do computador"

3. Iniciar o reconhecimento dos nós, mostrado na Figura A.14. Os nós devem estar desligados.
   # insert-ethers

   Opção: Compute

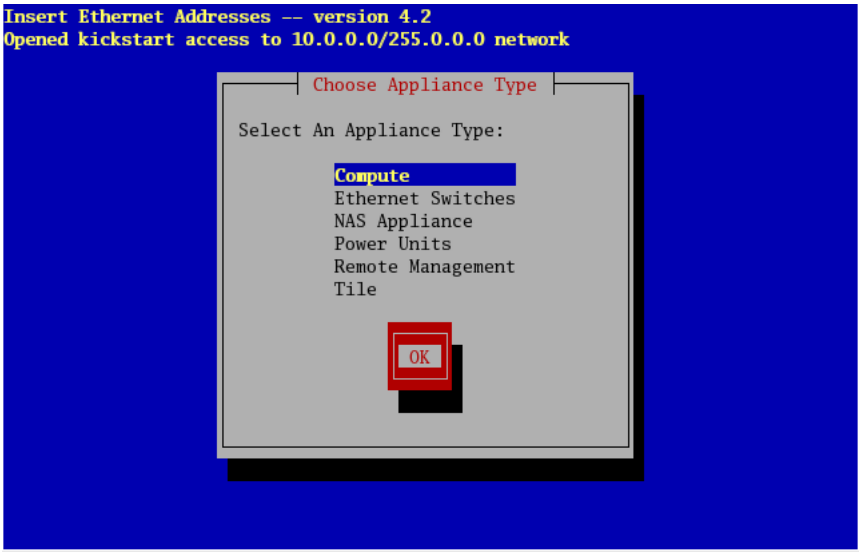

Figura A.14: Inserindo os nós no *Cluster* [40].

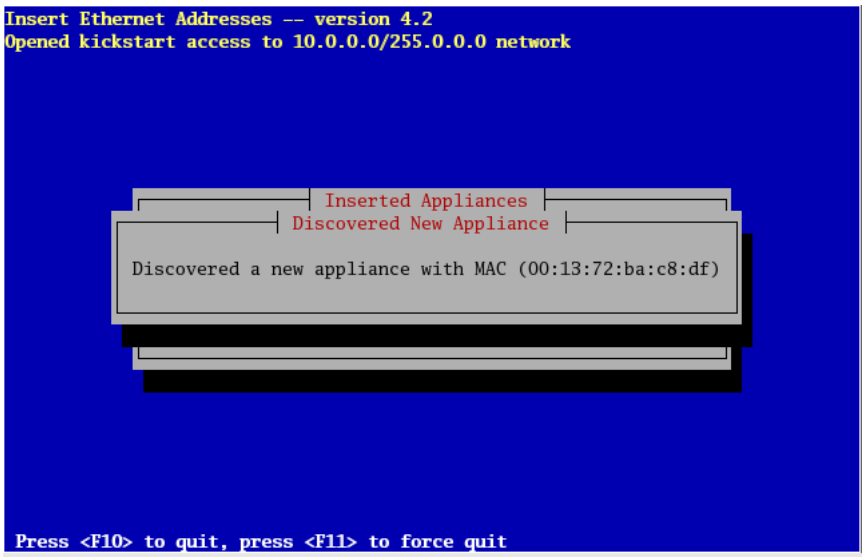

Figura A.15: Reconhecendo os nós do *Cluster* [40].

4. Ligar os nós, um de cada vez, efetuando o *Boot* pelo DVD contendo *Boot Roll* e instalar um nó de cada vez, conforme Figura A.16.

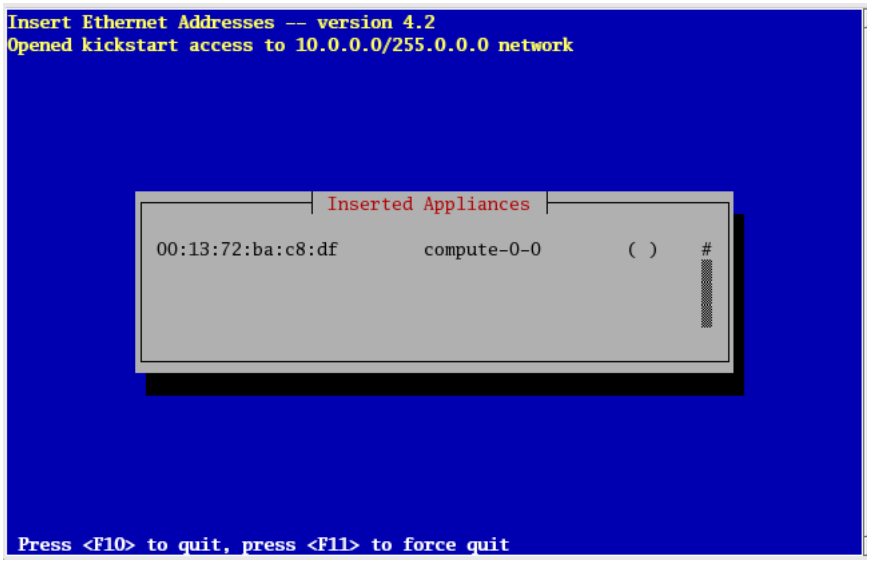

Figura A.16: Identificando os nós do *Cluster* [40].

5. Na tela *Insert appliances*, mostrada na Figura A.17 deve constar a tabela indicando o número MAC, nome e instalação SO (*) para cada nó.

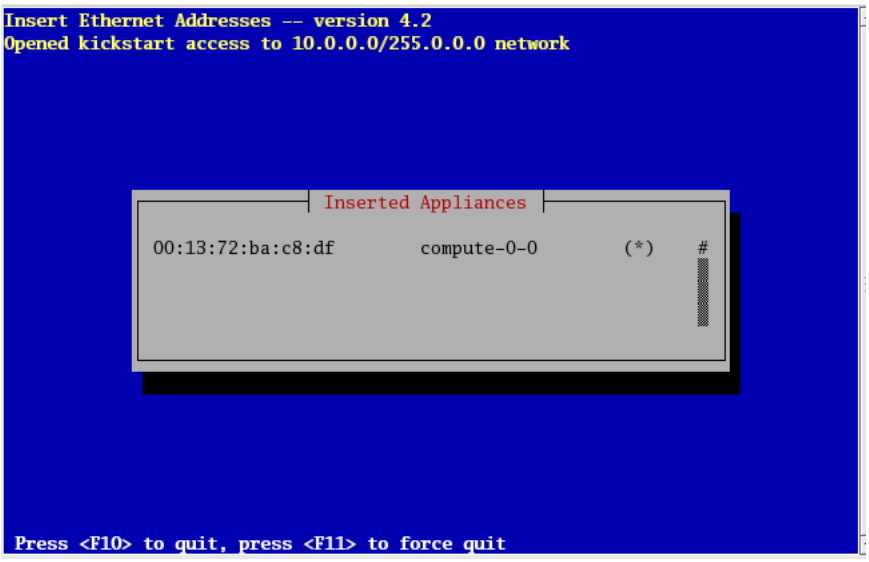

Figura A.17: Instalação do sistema operacional nos nós do *Cluster* [40].

Existem duas etapas:
( ) O *frontend* reconheceu o nó porém ainda não instalou o Sistema Operacional.
(*) O *frontend* reconheceu o nó e instalou o sistema operacional.

Este procedimento pode durar vários minutos. Caso necessite de executar novamente repetir os passos 2, 3, 4, 5.

A.3 Executando Código Computacional no *Cluster*

Após a instalação deve-se efetuar um teste para verificar o funcionamento do OPENMPI e dos nós do *Cluster*. Acessando o *frontend* deve-se iniciar o terminal, compilar e executar o código computacional. Para a compilação utilizamos o comando *mpicc* para códigos em C e *mpif90* para códigos em Fortran 90.

 # mpicc -o arquivo arquivo.c

 # mpif90 -o arquivo arquivo.f90

Após a compilação é gerado um arquivo binário que pode ser executado no *frontend* com vários processos, que são declarados com *-np* "numero de processos", como a seguir.

 # mpirun -np 3 arquivo

Para execução em vários nós do *Cluster* indica-se no comando *–hostfile* o arquivo texto onde estão listados o nome dos nós registrados na instalação do *Cluster*, no caso deste exemplo o nome do arquivo é *my_ target.*

 # mpirun –hostfile my_ target -np 6 arquivo

Para o *Cluster* VORTEX foi executado o código computacional de teste *teste_ cluster.c*, sendo compilado e executado com 6 processos, pelos comandos abaixo:

 # mpicc -o teste_ cluster teste_ cluster.c

 # mpirun –hostfile my_ target -np 6 teste_ cluster

O resultado obtido mostra cada processo sendo executado na respectiva máquina, utilizando 6 nós do *Cluster* VORTEX e cada um rodando um processo simultaneamente.

Cluster ->VORTEX<-: Processo 0 na Maquina compute-0-0.local

Cluster ->VORTEX<-: Processo 1 na Maquina compute-0-1.local

Cluster ->VORTEX<-: Processo 2 na Maquina compute-0-2.local

Cluster ->VORTEX<-: Processo 3 na Maquina compute-0-3.local

Cluster ->VORTEX<-: Processo 4 na Maquina compute-0-4.local

Cluster ->VORTEX<-: Processo 5 na Maquina engenhariavr.uff.br

APÊNDICE B – Códigos Computacionais

Neste apêndice estão listados os códigos computacionais implementados neste trabalho.

B.1 Código Computacional para Teste dos Nós do *Cluster* VORTEX

```
/*==================================================================
// Testar comunicao x processos dos nos do cluster
// Implementado por: Edgar Eller Junior
// E-mail: edgar.jrr@gmail.com
// Criado em: Junho, 2012          Versao: 001
==================================================================*/
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
int main (int argc, char * argv[])
{
int processId;
int noProcesses;
int nameSize;
char computerName[MPI_MAX_PROCESSOR_NAME];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &noProcesses);
MPI_Comm_rank(MPI_COMM_WORLD, &processId);
MPI_Get_processor_name(computerName, &nameSize);
printf("Cluster ->VORTEX<-: Processo %d na Maquina %s\n",
processId, computerName);
MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
return 0;
}
```

Código Computacional B.1: Teste dos nós do *Cluster* VORTEX - teste_cluster.c

B.2 Código Computacional para Verificação da Precisão da Máquina

```
/*====================================================================
// Verificacao da Precisao da Maquina
// Implementado por: Edgar Eller Junior
// E-mail: edgar.jrr@gmail.com
// Criado em: Julho, 2013          Versao: 001
====================================================================*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <unistd.h>

int main(int argc, char* argv[])
 {
 float ex, g=1 ,eps;
 do{
 g=g/2;
 ex=g*0.98+1;
 ex=ex-1;
 printf("g =%15.40e ex=%15.40e\n",g,ex);
   if (ex>0)
      eps=ex;
  }while (ex>0);

        printf("\nMAchine Epsilon= %32.40e\n \n", eps);

}
```

Código Computacional B.2: Verificação da precisão da máquina - epsilon

B.3 Código Computacional Regra do Trapézio Repetido

B.3.1 Código da Regra do Trapézio Repetido Serial

```
/*====================================================================
// Regra do Trapezio Repetido - Serial
// Implementado por: Edgar Eller Junior
// E-mail: edgar.jrr@gmail.com
// Criado em: Julho, 2013          Versao: 001
====================================================================*/
#include <stdio.h>
#include <math.h>
#include <sys/timeb.h>
#include <sys/resource.h>
#include <sys/times.h>
#include <time.h>

double cpuTime();
double wallClockSeconds();

int main(int argc, char** argv) {
   int          my_rank,i;
   int          p=1;
   double       a = 0.0;
   double       b = 1.0;
   int          n = 2400000;
   double       h;
   double       local_a;
   double       local_b;
   int          local_n;
   double       integral;
   double       total;
   int          source;
   int          dest = 0;
   int          tag = 0;

   double Trap(double local_a, double local_b, int local_n, double h);
   /* Inicio Tempo */
   double t0=cpuTime();
   double t2=wallClockSeconds();
   /****** Divisao dos subintervalos ******/
```

```
38     h = (b - a) / n;
39     local_n = n ;
40     local_a = a * local_n*h;
41     local_b = local_a + local_n*h;
42     integral = Trap(local_a, local_b, local_n, h);
43     total = total + integral;
44     /* Final Tempo */
45     double t1=cpuTime();
46     double t3=wallClockSeconds();
47 
48     printf("n = %d trapezios\n", n);
49     printf("Valor de integral de  %g a %g = %20.16g\n", a, b, total);
50     printf("Tempo CPU = %20.8f   Rank = %d\n", t1-t0, my_rank);
51     printf("Tempo Real = %20.8f  Rank = %d\n", t3-t2, my_rank);
52 
53 }
54 /****** Funcoes para calculo da integral  ******/
55 double Trap(double local_a, double local_b, int local_n, double h) {
56     double integral;
57     double x;
58     int i;
59 double f(double x);
60     integral = (f(local_a) + f(local_b))/2.0;
61     x = local_a;
62     for(i = 1; i <= local_n-1; i++) {
63        x = x + h;
64        integral = integral + f(x);
65     }
66     integral = integral * h;
67     return integral;
68 }
69 double f(double x) {
70     double return_val;
71     return_val=exp(x);
72     return return_val;
73 
74 /****** Funcoes para calculo do tempo de CPU e Real  ******/
75 }
76 double cpuTime() {
77   static struct rusage usage;
78   getrusage(RUSAGE_SELF, &usage);
79   return ((double)usage.ru_utime.tv_sec)+(((double)usage.ru_utime.
       tv_usec)/((double)1000000));
```

```
80
81 }
82 double wallClockSeconds() {
83   struct timeb tp;
84   double sec;
85
86   ftime(&tp);
87    sec =  ((double)tp.time)+(((double)tp.millitm)/1000.0);
88
89   return sec;
90 }
```

Código Computacional B.3: Regra do trapézio repetido serial - trap.c

B.3.2 Código da Regra do Trapézio Repetido Paralelo

```
1  /*====================================================================
2  // Regra do Trapezio Repetido - Paralelizado com metodo MPI_Send e Recv
3  // Implementado por: Edgar Eller Junior
4  // E-mail: edgar.jrr@gmail.com
5  // Criado em: Julho, 2013          Versao: 001
6  ====================================================================*/
7  #include <stdio.h>
8  #include <math.h>
9  #include "mpi.h"
10 #include <sys/timeb.h>
11 #include <sys/resource.h>
12 #include <sys/times.h>
13 #include <time.h>
14 double cpuTime();
15 double wallClockSeconds();
16 int main(int argc, char** argv) {
17    int         my_rank;
18    int         p;
19    double      a = 0.0;
20    double      b = 100.0;
21    int         n = 2400000;
22    double      h;
23    double      local_a;
24    double      local_b;
25    int         local_n;
26    double      integral;
```

```
27    double      total;
28    int         source;
29    int         dest = 0;
30    int         tag = 0;
31    MPI_Status   status;
32
33    double Trap(double local_a, double local_b, int local_n, double h);
34    /* Declaracao das rotinas de gerenciamento de processos MPI */
35    MPI_Init(&argc, &argv);
36    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
37    MPI_Comm_size(MPI_COMM_WORLD, &p);
38    /* Inicio Tempo */
39    double t0=cpuTime();
40    double t2=wallClockSeconds();
41    /****** Divisao dos processos ******/
42    h = (b - a) / n;
43    local_n = n / p;
44    local_a = a + my_rank  * local_n*h;
45    local_b = local_a + local_n*h;
46    integral = Trap(local_a, local_b, local_n, h);
47
48    /* Rotina de comunicacao entre processos MPI_Send e MPI_Recv */
49    if(my_rank == 0) {
50       total = integral;
51       for(source = 1; source < p; source++) {
52          MPI_Recv(&integral, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD,
              &status);
53          total = total + integral;
54       }
55    } else {
56       MPI_Send(&integral, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
57    }
58    /****** Calculo do tempo de CPU e Real******/
59    /* Final Tempo */
60    double t1=cpuTime();
61    double t3=wallClockSeconds();
62    double cpu = t1-t0;
63    double real = t3-t2;
64    double tcpu, treal;
65    /* Media dos tempos dos processos */
66    MPI_Reduce(&cpu, &tcpu, 1, MPI_DOUBLE,MPI_SUM, 0, MPI_COMM_WORLD);
67    MPI_Reduce(&real, &treal, 1, MPI_DOUBLE,MPI_SUM, 0, MPI_COMM_WORLD);
68
```

```
69     if(my_rank == 0) {
70        printf("n = %d trapezios\n", n);
71        printf("Valor de integral de  %g a %g = %20.16g\n", a, b, total);
72        printf("CPU = %20.8f    Rank = %d\n", tcpu/p, my_rank);
73        printf("Real = %20.8f    Rank = %d\n", treal/p, my_rank);
74            }
75     MPI_Finalize();
76 }
77 
78 /****** Funcoes para calculo da integral  ******/
79 double Trap(double local_a, double local_b, int local_n, double h) {
80     double integral;
81     double x;
82     int i;
83  double f(double x);
84     integral = (f(local_a) + f(local_b))/2.0;
85     x = local_a;
86     for(i = 1; i <= local_n-1; i++) {
87        x = x + h;
88        integral = integral + f(x);
89     }
90     integral = integral * h;
91     return integral;
92 }
93 double f(double x) {
94     double return_val;
95     return_val = exp(x);
96     return return_val;
97 }
98 /****** Funcoes para calculo do tempo de CPU e Real  ******/
99 double cpuTime() {
100   static struct rusage usage;
101   getrusage(RUSAGE_SELF, &usage);
102   return ((double)usage.ru_utime.tv_sec)+(((double)usage.ru_utime.
        tv_usec)/((double)1000000));
103 }
104 double wallClockSeconds() {
105   struct timeb tp;
106   double sec;
107   ftime(&tp);
108     sec =  ((double)tp.time)+(((double)tp.millitm)/1000.0);
109   return sec;
110 }
```

Código Computacional B.4: Regra do trapézio repetido paralelo com MPI_Send e Recv - trapmpisr.c

```
/*====================================================================
// Regra do Trapezio Repetido - Paralelizado com metodo MPI_Reduce
// Implementado por: Edgar Eller Junior
// E-mail: edgar.jrr@gmail.com
// Criado em: Julho, 2013          Versao: 001
====================================================================*/
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#include <sys/timeb.h>
#include <sys/resource.h>
#include <sys/times.h>
#include <time.h>
double cpuTime();
double wallClockSeconds();
main(int argc, char* argv[]) {
    int         my_rank;
    int         p;
    double      a = 0.0;
    double      b = 100.0;
    int         n = 2400000;
    double      h;
    double      local_a;
    double      local_b;
    int         local_n;
    double      integral;
    double      total = 0.0;
    int         dest = 0;
    int         tag = 50;
    int         i,j,source;
    double      x;
    double      overhead;
    double      start, finish;
    double      f(double x);
    double tcpu,treal,cpu,real;
    /* Declaracao das rotinas de gerenciamento de processos MPI */
    MPI_Status   status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
40      MPI_Comm_size(MPI_COMM_WORLD, &p);
41      /* Inicio Tempo */
42      double t0=cpuTime();
43      double t2=wallClockSeconds();
44      /****** Divisao dos processos ******/
45      h = (b-a)/n;
46      local_n =n/p;
47      local_a = a + my_rank * local_n * h;
48      local_b = local_a + local_n * h;
49      integral = (f(local_a) + f(local_b))*.5;
50      x = local_a;
51      for (i = 1; i <= local_n-1; i++) {
52          x = x + h;
53          integral = integral +  f(x);
54      }
55      integral = integral * h;
56       /****** Rotina de comunicacao entre processos MPI_Reduce ******/
57      MPI_Reduce(&integral, &total, 1, MPI_DOUBLE,MPI_SUM, 0,
           MPI_COMM_WORLD);
58
59      /****** Calculo do tempo de CPU e Real******/
60      /* Final Tempo */
61      double t1=cpuTime();
62      double t3=wallClockSeconds();
63      cpu = t1-t0;
64      real = t3-t2;
65      /* Media dos tempos dos processos */
66      MPI_Reduce(&cpu, &tcpu, 1, MPI_DOUBLE,MPI_SUM, 0, MPI_COMM_WORLD);
67      MPI_Reduce(&real, &treal, 1, MPI_DOUBLE,MPI_SUM, 0, MPI_COMM_WORLD);
68
69   if (my_rank == 0) {
70     printf("n = %d trapezios\n", n);
71     printf("Valor de integral de  %g a %g = %20.16g\n", a, b, total);
72     printf("CPU = %20.8f    Rank = %d\n", tcpu/p, my_rank);
73     printf("Real = %20.8f    Rank = %d\n", treal/p, my_rank);
74      }
75      MPI_Finalize();
76  }
77
78  /****** Funcoes para calculo da integral  ******/
79  double Trap(double local_a, double local_b, int local_n, double h) {
80     double integral;
81     double x;
```

```
82     int i;
83 double f(double x);
84     integral = (f(local_a) + f(local_b))/2.0;
85     x = local_a;
86     for(i = 1; i <= local_n-1; i++) {
87        x = x + h;
88        integral = integral + f(x);
89     }
90     integral = integral * h;
91     return integral;
92 }
93 double f(double x) {
94     double return_val;
95     return_val = exp(x);
96     return return_val;
97
98 /****** Funcoes para calculo do tempo de CPU e Real  ******/
99 }
100 double cpuTime() {
101   static struct rusage usage;
102   getrusage(RUSAGE_SELF, &usage);
103   return ((double)usage.ru_utime.tv_sec)+(((double)usage.ru_utime.
        tv_usec)/((double)1000000));
104 }
105 double wallClockSeconds() {
106   struct timeb tp;
107   double sec;
108   ftime(&tp);
109     sec =  ((double)tp.time)+(((double)tp.millitm)/1000.0);
110   return sec;
111 }
```

Código Computacional B.5: Regra do trapézio repetido paralelo com MPI_Reduce - trapmpir.c

B.4 Código Computacional Equação de Laplace Estacionária

B.4.1 Código Equação de Laplace Estacionária Serial

```
1 /*==========================================================================
```

```
2  // Equacao de Leplace Estacionaria - Serial
3  // Implementado por: Edgar Eller Junior
4  // E-mail: edgar.jrr@gmail.com
5  // Criado em: Julho, 2013          Versao: 001
6  ========================================================================*/
7  #include <stdio.h>
8  #include <string.h>
9  #include <math.h>
10 #include <sys/timeb.h>
11 #include <sys/resource.h>
12 #include <sys/times.h>
13 #include <time.h>
14 #include <unistd.h>
15 #include <sys/times.h>
16 #define n     289     // matriz
17 #define maxk  8000    // numero de iteracoes
18 char outputFilename[] = "laplaceN289M8000.txt";  //arquivo de saida
19 double cpuTime();
20 double wallClockSeconds();
21 int main(int argc, char* argv[])
22  {
23    int m, dest, tag;
24    int j, i, k;
25    int sbn, r;
26    double unew[n+2][n+2], uold[n+2][n+2];
27    double t0,t1,t2,t3;
28 // Inicio Tempo
29 t0=cpuTime();
30 t2=wallClockSeconds();
31 // Condicao inicial
32   for(j=1; j<=n+1; j++){
33         uold[1][j] = 100.0;
34         unew[1][j] = 100.0;
35       uold[n+1][j] = 100.0;
36         unew[n+1][j] = 100.0;}
37 // Metodo iterativo
38 for(k=1; k<=maxk; k++) {
39     for(i=2; i<=n; i++){
40       for(j=2; j<=n; j++){
41           unew[i][j] = 0.25*(uold[i+1][j] + uold[i-1][j] + uold[i][j+1]
                + uold[i][j-1]); }}
42     for(i=2 ;i<=n; i++){
43         for(j=2; j<=n; j++){
```

```
44            uold[i][j] = unew[i][j]; }}
45  }
46  // Final Tempo
47  t1=cpuTime();
48  t3=wallClockSeconds();
49  // Criando arquivo de saida
50    FILE *ifp, *ofp;
51    int score;
52    ofp = fopen(outputFilename, "w");
53    if (ofp == NULL) {
54      fprintf(stderr, "Can't open output file %s!\n", outputFilename);
55    }
56    else
57    {
58       for(i=1; i<=n+1; i++) {
59         for(j=1; j<=n+1; j++){
60           fprintf(ofp, "%d   %d   %20.8f \n",i,j, uold[i][j]);  }}
61    }
62    fclose(ofp);
63
64       printf("CPU = %20.8f    \n", t1-t0);
65       printf("Real = %20.8f   \n", t3-t2);
66
67  }
68  // Funcao Tempo CPU
69  double cpuTime() {
70    static struct rusage usage;
71    getrusage(RUSAGE_SELF, &usage);
72    return ((double)usage.ru_utime.tv_sec)+(((double)usage.ru_utime.
         tv_usec)/((double)1000000));
73
74  }
75  // Funcao Tempo Real
76  double wallClockSeconds() {
77    struct timeb tp;
78    double sec;
79
80    ftime(&tp);
81      sec =  ((double)tp.time)+(((double)tp.millitm)/1000.0);
82
83    return sec;
84  }
```

Código Computacional B.6: Equação de Laplace estacionária serial - laplace.c

B.4.2 Código Equação de Laplace Estacionária Paralelo

```
/*====================================================================
// Equacao de Leplace Estacionaria - Paralelo
// Implementado por: Edgar Eller Junior
// E-mail: edgar.jrr@gmail.com
// Criado em: Julho, 2013          Versao: 001
====================================================================*/
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"
#include <sys/timeb.h>
#include <sys/resource.h>
#include <sys/times.h>
#include <time.h>
#include <unistd.h>
#include <sys/times.h>
#define n     289     // matriz
#define maxk  8000    // numero de iteracoes
char outputFilename[] = "laplacempiN289M8000.txt";  //arquivo de saida
double cpuTime();
double wallClockSeconds();
int main(int argc, char* argv[])
 {
   int my_rank, p, m,source, dest, tag,loc_n;
   int mpi_status_size, bn, en, j, i, k;
   int sbn, r;
   double unew[n+2][n+2], uold[n+2][n+2];
   double t0,t1,t2,t3,cpu,real,tcpu,treal;
   int number_amount;
// Declaracao das rotinas de gerenciamento de processos MPI
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

// Inicio Tempo
t0=cpuTime();
```

```
38 t2=wallClockSeconds();
39 // Condicao inicial
40   for(j=1; j<=n+1; j++){
41         uold[1][j] = 100.0;
42         unew[1][j] = 100.0;
43         uold[n+1][j] = 100.0;
44         unew[n+1][j] = 100.0;}
45 // Divisao das submatrizes dos processos
46 loc_n = (n-1)/p;
47 bn = 2 + (my_rank * loc_n);
48 en = bn + loc_n -1;
49 source=0;
50 sbn=0;
51 // Metodo iterativo
52 for(k=1; k<=maxk; k++) {
53     for(i=bn; i<=en; i++){
54         for(j=2; j<=n; j++){
55           unew[i][j] = 0.25*(uold[i+1][j] + uold[i-1][j] + uold[i][j+1]
                + uold[i][j-1]); }}
56     for(i=bn ;i<=en; i++){
57       for(j=2; j<=n; j++){
58           uold[i][j] = unew[i][j]; }}
59 // Comunicacao entre os processos
60 if (my_rank == 0){
61     MPI_Recv(&uold[en+1][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
          MPI_COMM_WORLD, &status);
62   MPI_Send(&uold[en][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
       MPI_COMM_WORLD);
63 }
64 if ((my_rank > 0) && (my_rank < p-1) && (my_rank%2==1)) {
65   MPI_Send(&uold[en][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
       MPI_COMM_WORLD);
66   MPI_Recv(&uold[en+1][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
       MPI_COMM_WORLD, &status);
67   MPI_Send(&uold[bn][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
       MPI_COMM_WORLD);
68   MPI_Recv(&uold[bn-1][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
       MPI_COMM_WORLD, &status);
69 }
70 if ((my_rank > 0) && (my_rank < p-1) && (my_rank%2==0)) {
71   MPI_Recv(&uold[bn-1][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
       MPI_COMM_WORLD, &status);
```

```
72    MPI_Send(&uold[bn][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
          MPI_COMM_WORLD);
73    MPI_Recv(&uold[en+1][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
          MPI_COMM_WORLD, &status);
74    MPI_Send(&uold[en][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
          MPI_COMM_WORLD);
75  }
76  if (my_rank == p-1) {
77    MPI_Send(&uold[bn][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
          MPI_COMM_WORLD);
78    MPI_Recv(&uold[bn-1][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
          MPI_COMM_WORLD, &status);
79  }
80  }
81  if (my_rank == 0) {
82    for (source = 1; source<p; source++){
83      sbn = 2 + (source*loc_n);
84      MPI_Recv(&uold[sbn][1], ((n+2)*loc_n), MPI_DOUBLE, source, 50,
            MPI_COMM_WORLD, &status);  }
85  }
86  else
87  {
88    MPI_Send(&uold[bn][1], ((n+2)*loc_n), MPI_DOUBLE, 0, 50,
          MPI_COMM_WORLD);
89  }
90  // Sincronismo dos processos
91  MPI_Barrier(MPI_COMM_WORLD);
92  // Final Tempo
93  t1=cpuTime();
94  t3=wallClockSeconds();
95  cpu=t1-t0;
96  real=t3-t2;
97  // Media dos tempos dos processos
98  MPI_Reduce(&cpu, &tcpu, 1, MPI_DOUBLE,MPI_SUM, 0, MPI_COMM_WORLD);
99  MPI_Reduce(&real, &treal, 1, MPI_DOUBLE,MPI_SUM, 0, MPI_COMM_WORLD);
100 // Criando arquivo de saida
101  if (my_rank == 0){
102   FILE *ifp, *ofp;
103   int score;
104   ofp = fopen(outputFilename, "w");
105
106   if (ofp == NULL) {
107     fprintf(stderr, "Can't open output file %s!\n", outputFilename);
```

```
108   }
109   else
110   {
111      for(i=1; i<=n+1; i++) {
112        for(j=1; j<=n+1; j++){
113            fprintf(ofp, "%d   %d   %20.8f \n",i,j, uold[i][j]);  }}
114   }
115   fclose(ofp);
116
117   printf("CPU = %20.8f     Rank = %d\n", tcpu/p, my_rank);
118   printf("Real = %20.8f     Rank = %d\n", treal/p, my_rank);
119 }
120 MPI_Finalize();
121 }
122 // Funcao Tempo CPU
123 double cpuTime() {
124   static struct rusage usage;
125   getrusage(RUSAGE_SELF, &usage);
126   return ((double)usage.ru_utime.tv_sec)+(((double)usage.ru_utime.
        tv_usec)/((double)1000000));
127
128 }
129 // Funcao Tempo Real
130 double wallClockSeconds() {
131   struct timeb tp;
132   double sec;
133   ftime(&tp);
134     sec =  ((double)tp.time)+(((double)tp.millitm)/1000.0);
135   return sec;
136 }
```

Código Computacional B.7: Equação de Laplace estacionária paralelo - laplacempi.c

B.5 Código Computacional Equação de Laplace Transiente

B.5.1 Código Equação de Laplace Transiente Serial

```
1 /*====================================================================
2 // Equacao de Leplace Transiente - Serial
3 // Autor: Edgar Eller Junior
4 // E-mail: edgar.jrr@gmail.com
```

```
5  // Criado em: Julho, 2013          Versao: 001
6  ========================================================================*/
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <math.h>
11 #include <sys/timeb.h>
12 #include <sys/resource.h>
13 #include <sys/times.h>
14 #include <time.h>
15 #include <unistd.h>
16 #include <sys/times.h>
17 #define n      289      // matriz
18 #define maxk   3500     // iteracoes grafico (tempo)
19 #define maxkT  5000     // iteracoes total
20 #define Tend   6.38
21
22 double cpuTime();
23 double wallClockSeconds();
24 int main(int argc, char* argv[])
25  {
26    char outputFilename[]  = "heat2dN289M3500";  //arquivo de saida
27    int my_rank, p, m,source, dest, tag,loc_n;
28    int mpi_status_size, bn, en, j, i, k;
29    int sbn, r;
30    double unew[n+2][n+2], uold[n+2][n+2], x[n+2][n+2], y[n+2][n+2], time
          [n+2];
31    double t0,t1,t2,t3,f,cond,dt,dx,dy,alpha,cpu,real,tcpu,treal,L,W,h,
          spheat,rho,a,b;
32    int number_amount;
33    double aux,resid[maxk], aux2, aux3, auxn;
34 // Dados iniciais
35 L=1.0;
36 W=L;
37 dt=Tend/maxkT;
38 dx=L/n;
39 dy=W/n;   // onde dx=dy=h
40 h=dx;
41 b=dt/(h*h);
42 cond=.002;    //k
43 spheat=1.0;
44 rho=1.0;
45 a=cond/(spheat*rho);
```

```
alpha = a*b;
loc_n=0;
bn=0;
en=0;
// Condicao inicial
 for(i=1; i<=n+1; i++){
  for(j=1; j<=n+1; j++){
  x[i][j] = 0.0;
  y[i][j] = 0.0;
// Condicao de contorno
    uold[i][j] = 70.0;
    unew[i][j] = 70.0;
    uold[1][j] = 370.0;
    unew[1][j] = 370.0;
    }}
// Inicio Tempo
t0=cpuTime();
t2=wallClockSeconds();
// Metodo iterativo
for(k=1; k<=maxk; k++) {
   aux=0;
// Alteracao condicao de contorno
   if(k==1000){
    for(i=1; i<=n+1; i++){
     for (j=1; j<=n+1; j++) {
     uold[1][j] = 70.0;
        unew[1][j] = 70.0;  }}}
// Metodo iterativo
  for(i=2; i<=n; i++){
  for(j=2; j<=n; j++){
      unew[i][j] = 0.* dt/(spheat*rho)+alpha*(uold[i+1][j] + uold[i-1][j
         ] + uold[i][j+1] + uold[i][j-1])+(1.0-4.0 * alpha)* uold[i][j];
          }}
  for(i=2 ;i<=n; i++){
    for(j=2; j<=n; j++){
       uold[i][j] = unew[i][j]; }}
}
// Final Tempo
t1=cpuTime();
t3=wallClockSeconds();
cpu=t1-t0;
real=t3-t2;
// Criando arquivo de saida
```

```
87  FILE *ifp, *ofp;
88    int score;
89    ofp = fopen(outputFilename, "w");
90
91    if (ofp == NULL) {
92      fprintf(stderr, "Can't open output file %s!\n", outputFilename);
93    }
94    else
95    {
96    fprintf(ofp, "%d %d\n",n+1,n+1);
97
98      for(i=1; i<=n+1; i++) {
99         for(j=1; j<=n+1; j++){
100         x[i][j] = (i-1)*h;
101       y[i][j] = (j-1)*h;
102         fprintf(ofp, "%f %f %20.8f \n" ,x[i][j],y[i][j], uold[i][j]);
              }}}
103
104    printf("maxk= %d, n %d\n",maxk, n);
105    printf("CPU = %20.8f    \n", cpu);
106    printf("Real = %20.8f   \n ", real);
107
108
109 }
110 // Funcao Tempo CPU
111 double cpuTime() {
112   static struct rusage usage;
113   getrusage(RUSAGE_SELF, &usage);
114   return ((double)usage.ru_utime.tv_sec)+(((double)usage.ru_utime.
        tv_usec)/((double)1000000));
115
116 }
117 // Funcao Tempo Real
118 double wallClockSeconds() {
119   struct timeb tp;
120   double sec;
121   ftime(&tp);
122     sec =  ((double)tp.time)+(((double)tp.millitm)/1000.0);
123   return sec;
124 }
```

Código Computacional B.8: Equação de Laplace transiente serial - heat2d.c

B.5.2 Código Equação de Laplace Transiente Paralelo

```
/*====================================================================
// Equacao de Leplace Transiente - Paralelo
// Implementado por: Edgar Eller Junior
// E-mail: edgar.jrr@gmail.com
// Criado em: Julho, 2013          Versao: 001
====================================================================*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <mpi.h>
#include <sys/timeb.h>
#include <sys/resource.h>
#include <sys/times.h>
#include <time.h>
#include <unistd.h>
#include <sys/times.h>
#define n     385     // matriz
#define maxk  5000    // iteracoes grafico (tempo)
#define maxkT 5000    // iteracoes total
#define Tend  3.59
char outputFilename[]  = "heat2dmpiN385M5000.txt";  //arquivo de saida
double cpuTime();
double wallClockSeconds();
int main(int argc, char* argv[])
 {
   int my_rank, p, m,source, dest, tag,loc_n;
   int mpi_status_size, bn, en, j, i, k;
   int sbn, r;
   double unew[n+2][n+2], uold[n+2][n+2], x[n+2][n+2], y[n+2][n+2], time
      [n+2];
   double t0,t1,t2,t3,f,cond,dt,dx,dy,alpha,cpu,real,tcpu,treal,L,W,h,
      spheat,rho,a,b;
   int number_amount;
// Declaracao das rotinas de gerenciamento de processos MPI
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
status.MPI_ERROR=0;
// Dados iniciais
```

```
L=1.0;
W=L;
dt=Tend/maxkT;
dx=L/n;
dy=W/n;   // onde dx=dy=h
h=dx;
b=dt/(h*h);
cond=.002;    //k
spheat=1.0;
rho=1.0;
a=cond/(spheat*rho);
alpha = a*b;
loc_n=0;
bn=0;
en=0;
// Condicao inicial
 for(i=1; i<=n+1; i++){
  for(j=1; j<=n+1; j++){
  x[i][j] = 0.0;
  y[i][j] = 0.0;
// Condicao de contorno
    uold[i][j] = 70.0;
    unew[i][j] = 70.0;
    uold[1][j] = 370.0;
    unew[1][j] = 370.0;
    }}
// Divisao das submatrizes dos processos
loc_n = (n-1)/p;
bn = 2 + (my_rank * loc_n);
en = bn + loc_n -1;
source=0;
sbn=0;
// Sincronismo dos processos
MPI_Barrier(MPI_COMM_WORLD);
// Inicio Tempo
t0=cpuTime();
t2=wallClockSeconds();
// Metodo iterativo
for(k=1; k<=maxk; k++) {
// Alteracao condicao de contorno
   if(k==1000){
    for(i=1; i<=n+1; i++){
     for (j=1; j<=n+1; j++) {
```

```
83        uold[1][j] = 70.0;
84          unew[1][j] = 70.0;  }}}
85
86      for(i=bn; i<=en; i++){
87        for(j=2; j<=n; j++){
88           unew[i][j] = 0.* dt/(spheat*rho)+alpha*(uold[i+1][j] + uold[i
                -1][j] + uold[i][j+1] + uold[i][j-1])+(1.0-4.0 * alpha)*
                uold[i][j];}}
89
90      for(i=bn ;i<=en; i++){
91        for(j=2; j<=n; j++){
92           uold[i][j] = unew[i][j]; }}
93  // Comunicacao entre os processos
94  if (my_rank == 0){
95      MPI_Recv(&uold[en+1][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
           MPI_COMM_WORLD, &status);
96    MPI_Send(&uold[en][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
        MPI_COMM_WORLD);
97  }
98  if ((my_rank > 0) && (my_rank < p-1) && (my_rank%2==1)) {
99    MPI_Send(&uold[en][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
        MPI_COMM_WORLD);
100   MPI_Recv(&uold[en+1][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
        MPI_COMM_WORLD, &status);
101   MPI_Send(&uold[bn][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
        MPI_COMM_WORLD);
102   MPI_Recv(&uold[bn-1][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
        MPI_COMM_WORLD, &status);
103 }
104 if ((my_rank > 0) && (my_rank < p-1) && (my_rank%2==0)) {
105   MPI_Recv(&uold[bn-1][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
        MPI_COMM_WORLD, &status);
106   MPI_Send(&uold[bn][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
        MPI_COMM_WORLD);
107   MPI_Recv(&uold[en+1][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
        MPI_COMM_WORLD, &status);
108   MPI_Send(&uold[en][1], (n+1), MPI_DOUBLE, my_rank+1, 50,
        MPI_COMM_WORLD);
109 }
110 if (my_rank == p-1) {
111   MPI_Send(&uold[bn][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
        MPI_COMM_WORLD);
```

```
112   MPI_Recv(&uold[bn-1][1], (n+1), MPI_DOUBLE, my_rank-1, 50,
          MPI_COMM_WORLD, &status);
113 }
114 }
115 if (my_rank == 0) {
116   for (source = 1; source<p; source++){
117     sbn = 2 + (source*loc_n);
118     MPI_Recv(&uold[sbn][1], ((n+2)*loc_n), MPI_DOUBLE, source, 50,
            MPI_COMM_WORLD, &status);
119 }
120 }
121 else
122 {
123   MPI_Send(&uold[bn][1], ((n+2)*loc_n), MPI_DOUBLE, 0, 50,
          MPI_COMM_WORLD);
124 }
125 // Sincronismo dos processos
126 MPI_Barrier(MPI_COMM_WORLD);
127 // Final Tempo
128 t1=cpuTime();
129 t3=wallClockSeconds();
130 cpu=t1-t0;
131 real=t3-t2;
132 // Media dos tempos dos processos
133 MPI_Reduce(&cpu, &tcpu, 1, MPI_DOUBLE,MPI_SUM, 0, MPI_COMM_WORLD);
134 MPI_Reduce(&real, &treal, 1, MPI_DOUBLE,MPI_SUM, 0, MPI_COMM_WORLD);
135
136 // Criando arquivo de saida
137 if (my_rank == 0){
138
139   FILE *ifp, *ofp;
140   int score;
141   ofp = fopen(outputFilename, "w");
142
143   if (ofp == NULL) {
144     fprintf(stderr, "Can't open output file %s!\n", outputFilename);
145   }
146   else
147   {
148   fprintf(ofp, "%d %d\n",n+1,n+1);
149   for(i=1; i<=n+1; i++) {
150      for(j=1; j<=n+1; j++){
151         x[i][j] = (i-1)*h;
```

```
152       y[i][j] = (j-1)*h;
153         fprintf(ofp, "%f %f %20.8f \n" ,x[i][j],y[i][j], uold[i][j]);
              }}}
154
155    printf("P= %d maxk= %d, n %d\n",p,maxk, n);
156    printf("CPU = %20.8f     Rank = %d\n", tcpu/p, my_rank);
157    printf("Real = %20.8f     Rank = %d\n", treal/p, my_rank);
158
159 }
160 MPI_Finalize();
161 }
162 // Funcao Tempo CPU
163 double cpuTime() {
164   static struct rusage usage;
165   getrusage(RUSAGE_SELF, &usage);
166   return ((double)usage.ru_utime.tv_sec)+(((double)usage.ru_utime.
        tv_usec)/((double)1000000));
167
168 }
169 // Funcao Tempo Real
170 double wallClockSeconds() {
171   struct timeb tp;
172   double sec;
173   ftime(&tp);
174     sec =  ((double)tp.time)+(((double)tp.millitm)/1000.0);
175   return sec;
176 }
```

Código Computacional B.9: Equação de Laplace transiente paralelo - heat2dmpi.c

B.6 Código Computacional para Plotagem dos Resultados

B.6.1 Plotagem dos Resultados do Código Equação de Laplace Estacionário Serial e Paralelo

```
1 %============================================================
2 % Plot2D
3 % Author: Prof. Diomar Cesar Lobao PhD
4 % Created: 2012-08-21
5 %============================================================
```

```
%
clear all;clc;close all;
%
% XX2.txt has the x coordinates
% ZZ2.txt has the y coordinates
% Tnew2.txt has the Tnew solution
fid1=fopen('laplacempiN289M8000.txt','r'); %Open file to read
%
%read the file line by line
disp('Reading...')
%
mnodex = fscanf(fid1,'%d',1)
mnodey = fscanf(fid1,'%d',1)
xx1=zeros(mnodex,mnodey);
yy1=zeros(mnodex,mnodey);
TnewP=zeros(mnodex,mnodey);
%
% As was saved by colunm has to be read by colun
for i=1:mnodex
    for j=1:mnodey
        xx1(i,j) = fscanf(fid1,'%d',1);
        yy1(i,j) = fscanf(fid1,'%d',1);
        TnewP(i,j)= fscanf(fid1,'%f',1);
    end
end
%
%
  figure(1)
  ;

  %pcolor(xx1,yy1,TnewP); shading interp, colorbar
  mesh(xx1,yy1,TnewP);
  %contourf(xx1,yy1,TnewP);
  xlabel('x ','fontsize',14)
  ylabel('y ','fontsize',14)
  zlabel('Temperatura','fontsize',14)
  title('Laplace Estacionario Paralelo - n=290 K=8000','fontsize',14);
 % title('Temperatura')
 %
```

Código Computacional B.10: Plotagem do resultado da equação de Laplace estacionário - plotlaplace_ok.m

B.6.2 Plotagem dos Resultados do Código Equação de Laplace Transiente Serial e Paralelo

```
%=====================================================================
% Plotagem do resultado da equacao de Laplace transiente
% Autor: Prof. Diomar Cesar Lobao PhD
% Criado em: 2012-08-21          Versao: 001
%
% Autor: Edgar Eller Junior
% Alterado em: Julho, 2013       Versao: 002
%=====================================================================*/
%
clear all;clc;close all;
%
% XX2.txt has the x coordinates
% ZZ2.txt has the y coordinates
% Tnew2.txt has the Tnew solution
fid1=fopen('heat2dmpiN193M500.txt','r'); %Open file to read
fid2=fopen('heat2dmpiN193M1000.txt','r'); %Open file to read
fid3=fopen('heat2dmpiN193M2000.txt','r'); %Open file to read
fid4=fopen('heat2dmpiN193M5000.txt','r'); %Open file to read
%
%read the file line by line
disp('Reading...')
%
mnodex1 = fscanf(fid1,'%d',1)
mnodey1 = fscanf(fid1,'%d',1)
xx1=zeros(mnodex1,mnodey1);
yy1=zeros(mnodex1,mnodey1);
TnewP1=zeros(mnodex1,mnodey1);
%
% As was saved by colunm has to be read by colun
for i=1:mnodex1
    for j=1:mnodey1
        xx1(i,j) = fscanf(fid1,'%f',1);
        yy1(i,j) = fscanf(fid1,'%f',1);
        TnewP1(i,j)= fscanf(fid1,'%f',1);
    end
end
%read the file line by line
%
mnodex2 = fscanf(fid2,'%d',1)
mnodey2 = fscanf(fid2,'%d',1)
```

```
xx2=zeros(mnodex2,mnodey2);
yy2=zeros(mnodex2,mnodey2);
TnewP2=zeros(mnodex2,mnodey2);
%
% As was saved by colunm has to be read by colun
for i=1:mnodex2
    for j=1:mnodey2
        xx2(i,j) = fscanf(fid2,'%f',1);
        yy2(i,j) = fscanf(fid2,'%f',1);
        TnewP2(i,j)= fscanf(fid2,'%f',1);
    end
end
%read the file line by line
%
mnodex3 = fscanf(fid3,'%d',1)
mnodey3 = fscanf(fid3,'%d',1)
xx3=zeros(mnodex3,mnodey3);
yy3=zeros(mnodex3,mnodey3);
TnewP3=zeros(mnodex3,mnodey3);
%
% As was saved by colunm has to be read by colun
for i=1:mnodex3
    for j=1:mnodey3
        xx3(i,j) = fscanf(fid3,'%f',1);
        yy3(i,j) = fscanf(fid3,'%f',1);
        TnewP3(i,j)= fscanf(fid3,'%f',1);
    end
end
%
%read the file line by line
%
mnodex4 = fscanf(fid4,'%d',1)
mnodey4 = fscanf(fid4,'%d',1)
xx4=zeros(mnodex4,mnodey4);
yy4=zeros(mnodex4,mnodey4);
TnewP4=zeros(mnodex4,mnodey4);
%
% As was saved by colunm has to be read by colun
for i=1:mnodex4
    for j=1:mnodey4
        xx4(i,j) = fscanf(fid4,'%f',1);
        yy4(i,j) = fscanf(fid4,'%f',1);
        TnewP4(i,j)= fscanf(fid4,'%f',1);
```

```
84      end
85  end
86
87  figure(1)
88    title('Temperature');
89    subplot(2,2,1);
90    surf(xx1,yy1,TnewP1); shading interp,
91    xlabel('x ')
92    ylabel('y ')
93    zlabel('Temperatura')
94    title('Laplace Transiente Paralelo (n=194 - k=500)')
95
96    subplot(2,2,2);
97    surf(xx2,yy2,TnewP2); shading interp,
98    xlabel('x ')
99    ylabel('y ')
100   zlabel('Temperatura')
101   title('Laplace Transiente Paralelo (n=194 - k=1000)')
102
103   subplot(2,2,3);
104   surf(xx3,yy3,TnewP3); shading interp,
105   xlabel('x ')
106   ylabel('y ')
107   zlabel('Temperatura')
108   title('Laplace Transiente Paralelo (n=194 - k=2000)')
109
110   subplot(2,2,4);
111   surf(xx4,yy4,TnewP4); shading interp,
112   xlabel('x ')
113   ylabel('y ')
114   zlabel('Temperatura')
115   title('Laplace Transiente Paralelo (n=194 - k=5000)')
116   %figure(3);
```

Código Computacional B.11: Plotagem do resultado da equação de Laplace transiente - plotheat2d4graf.m

B.6.3 Plotagem dos Gráficos Comparativos dos Resultados da Regra do Trapézio - Serial e Paralelo

```
1  %=====================================================================
```

```
% Plotagem dos Graficos comparativos de tempo x processos da regra do
    trapezio
% Autor: Edgar Eller Junior
% E-mail: edgar.jrr@gmail.com
% Criado em: Julho, 2013          Versao: 001
%=========================================================================*/
clear all;clc;close all;
data2 = load('resulTrapezio_N2400000_REAL.txt');
 x= [2,4,8,16,32,48];
 n=3*8
 for i=1:n
 ee(1,i)=std(data2(i,:));
 yy(1,i)=mean(data2(i,:));
 end
 y1=yy(1,1:6);
 y2=yy(1,9:14);
 y3=yy(1,17:22);
 e1=ee(1,1:6);
 e2=ee(1,9:14);
 e3=ee(1,17:22);
 figure(1)
 hold on
 errorbar (x,y1,e1,'--ro','LineWidth',1.5, 'MarkerEdgeColor','k','
    MarkerFaceColor','r','MarkerSize',5);
 errorbar (x,y2,e2,'--gs','LineWidth',1.5, 'MarkerEdgeColor','k','
    MarkerFaceColor','g','MarkerSize',5);
 errorbar (x,y3,e3,'--bs','LineWidth',1.5, 'MarkerEdgeColor','k','
    MarkerFaceColor','b','MarkerSize',5);
 box on
 set(gca,'XTick',[2,4,8,16,32,48])
 xlabel('P - Processos','fontsize',14)
 ylabel('Tempo - Segundos','fontsize',14)
 title('Trapezio - Tempo CPU','fontsize',14)
 hleg1 =legend ('Serial','Paralelo (Send e Recv)','Paralelo (Reduce)')
 set(hleg1,'Location','NorthWest')
 set(hleg1,'fontsize',12);
```

Código Computacional B.12: Plotagem comparativa de tempo x processos da regra do trapézio - plotresultTrapezio3linhasdesvio.m

B.6.4 Plotagem dos Gráficos Comparativos dos Resultados da Equação de Laplace Estacionária Serial e Paralelo

```
%=====================================================================
% Plotagem dos graficos comparativos de tempo x processos da regra da
    equacao de Laplace estacionaria
% Autor: Edgar Eller Junior
% E-mail: edgar.jrr@gmail.com
% Criado em: Julho, 2013          Versao: 001
=====================================================================*/
clear all;clc;close all;
data2 = load('resultlaplaceREAL_M8000.txt');
x= [4,8,16,32,48];
for i=1:10
ee(1,i)=std(data2(i,:));
yy(1,i)=mean(data2(i,:));
end
y1=yy(1,1:5);
y2=yy(1,6:10);
e1=ee(1,1:5);
e2=ee(1,6:10);

 figure(1)
 hold on
 errorbar (x,y1,e1,'--ro','LineWidth',1.5, 'MarkerEdgeColor','k','
    MarkerFaceColor','r','MarkerSize',6);
 errorbar (x,y2,e2,'--go','LineWidth',1.5, 'MarkerEdgeColor','k','
    MarkerFaceColor','r','MarkerSize',6);
 box on
 axis([1,50,0,7])
 set(gca,'XTick',4:4:48)
 xlabel('P - Processos','fontsize',14)
 ylabel('Tempo - Segundos','fontsize',14)
 title('Laplace Estacionario - Tempo CPU (K=8000)','fontsize',14)
 hleg1 =legend ('Serial','Paralelo')
 set(hleg1,'Location','NorthWest')
 set(hleg1,'fontsize',12);
```

Código Computacional B.13: Plotagem comparativa de tempo x processos da equação de Laplace estacionária - plotresultLaplace2linhasdesvio.m

B.6.5 Plotagem dos Gráficos Comparativos dos Resultados da Equação de Laplace Transiente Serial e Paralelo

```
%=====================================================================
% Plotagem dos Graficos comparativos de tempo x processos da regra da
    equacao de Laplace transiente
% Autor: Edgar Eller Junior
% E-mail: edgar.jrr@gmail.com
% Criado em: Julho, 2013          Versao: 001
%=====================================================================
 clear all;clc;close all;
 data2 = load('Resultheat2dN193CPU_desvio.txt');
 x= [500,1000,2000,3500,5000];
 n=5*4
 for i=1:n
 ee(1,i)=std(data2(i,:));
 yy(1,i)=mean(data2(i,:));
 end
 y1=yy(1,1:5);
 y2=yy(1,6:10);
 y3=yy(1,11:15);
 y4=yy(1,16:20);
 e1=ee(1,1:5);
 e2=ee(1,6:10);
 e3=ee(1,11:15);
 e4=ee(1,16:20);
 figure(1)
 hold on
 errorbar (x,y1,e1,'--ro','LineWidth',1.5, 'MarkerEdgeColor','k','
    MarkerFaceColor','r','MarkerSize',5);
 errorbar (x,y2,e2,'--gs','LineWidth',1.5, 'MarkerEdgeColor','k','
    MarkerFaceColor','g','MarkerSize',5);
 errorbar (x,y3,e3,'-.bs','LineWidth',1.5, 'MarkerEdgeColor','k','
    MarkerFaceColor','b','MarkerSize',5);
 errorbar (x,y4,e4,'-ks','LineWidth',1.5, 'MarkerEdgeColor','k','
    MarkerFaceColor','w','MarkerSize',5);
 box on
 axis([500,5000,0,3])
 set(gca,'XTick',500:500:5500)
 xlabel('K - Iteracoes','fontsize',14)
 ylabel('Tempo - Segundos','fontsize',14)
 title('Laplace Transiente - Tempo CPU (N=194)','fontsize',14)
 hleg1 =legend ('Serial','Paralelo P=4','Paralelo P=16','Paralelo P=48')
```

```
 set(hleg1,'Location','NorthWest')
 set(hleg1,'fontsize',12);
errorbar(X,Y,E)
```

Código Computacional B.14: Plotagem comparativa de tempo x processos da equação de Laplace transiente - plotresultHeat2d5linhasdesvio.m

B.7 Código Computacional para Plotagem do Residual

```
%=====================================================================
% Plotagem do erro da regra do trapezio
% Autor: Prof. Diomar Cesar Lobao PhD
% Criado em: 2012-08-21          Versao: 001
%
% Autor: Edgar Eller Junior
% Alterado em: Julho, 2013       Versao: 002
%=====================================================================*/
%
clear all;clc;close all;
%
% XX2.txt has the x coordinates
% ZZ2.txt has the y coordinates
% Tnew2.txt has the Tnew solution
data1=load('resulttrapezioerro_revista.txt'); %Open file to read
%
%read the file line by line
disp('Reading...')
%
y= data1(:,1);
xx1= data1(:,2);
xx2= data1(:,3);
xx3= data1(:,4);
figure(1)
  subplot(2,2,1);
  plot (y,xx1,'-k','LineWidth',2.2, 'MarkerEdgeColor','k','
     MarkerFaceColor','r','MarkerSize',5)
  xlabel('P - Processos','fontsize',12)
  ylabel('Residual','fontsize',12)
  title('Traps.c Serial - Erro Relativo','fontsize',12)

  subplot(2,2,2);
```

```
  plot (y,xx2,'-k','LineWidth',2.2, 'MarkerEdgeColor','k','
     MarkerFaceColor','r','MarkerSize',5)
  xlabel('P - Processos','fontsize',12)
  ylabel('Residual','fontsize',12)
  title('Trapmpisr.c Paralelo - Erro Relativo','fontsize',12)

  subplot(2,2,3);
  plot (y,xx3,'-k','LineWidth',2.2, 'MarkerEdgeColor','k','
     MarkerFaceColor','r','MarkerSize',5)
  xlabel('P - Processos','fontsize',12)
  ylabel('Residual','fontsize',12)
  title('Trapmpir.c Paralelo - Erro Relativo','fontsize',12)
```

Código Computacional B.15: Plotagem do erro da regra do trapézio - plotresidtrapezio3graf.m

```
%=========================================================================
% Plotagem do residual da equacao de Laplace estacionario
% Autor: Prof. Diomar Cesar Lobao PhD
% Criado em: 2012-08-21          Versao: 001
%
% Autor: Edgar Eller Junior
% Alterado em: Julho, 2013       Versao: 002
%=========================================================================*/
%
clear all;clc;close all;
%
% XX2.txt has the x coordinates
% ZZ2.txt has the y coordinates
% Tnew2.txt has the Tnew solution
data1=load('laplaceR500000_teste.txt'); %Open file to read
data2=load('laplacempiRM500000.txt'); %Open file to read
%
%read the file line by line
disp('Reading...')
%
xx1= data1(:,1);
xx2= data2(:,1);
  subplot(1,2,1);
  plot (xx1,'-k','LineWidth',2.2, 'MarkerEdgeColor','k','MarkerFaceColor
     ','r','MarkerSize',6)
  xlabel('K - Iteracoes','fontsize',12)
  ylabel('Residual','fontsize',12)
```

```
  title('Lapalce.c Serial - Norma L2','fontsize',12)

  subplot(1,2,2);
  plot (xx2,'-k','LineWidth',2.2, 'MarkerEdgeColor','k','MarkerFaceColor
     ','r','MarkerSize',6)
  xlabel('K - Iteracoes','fontsize',12)
  ylabel('Residual','fontsize',12)
  title('Laplacempi.c Paralelo - Norma L2','fontsize',12)
```

Código Computacional B.16: Plotagem do residual da equação de Laplace estacionário - plotresidlaplace2d2graf.m

```
%=====================================================================
% Plotagem do residual da equacao de Laplace transiente
% Autor: Prof. Diomar Cesar Lobao PhD
% Criado em: 2012-08-21          Versao: 001
%
% Autor: Edgar Eller Junior
% Alterado em: Julho, 2013       Versao: 002
%=====================================================================*/
%
clear all;clc;close all;
%
% XX2.txt has the x coordinates
% ZZ2.txt has the y coordinates
% Tnew2.txt has the Tnew solution
data1=load('residheat2dmpi500000N193.txt'); %Open file to read
data2=load('residheat2dmpi500000N289.txt'); %Open file to read
data3=load('residheat2dmpi500000N385.txt'); %Open file to read
%
%read the file line by line
disp('Reading...')
%
xx1= data1(:,1);
xx2= data2(:,1);
xx3= data3(:,1);
figure(1)
 %
  subplot(2,2,1);
  plot (xx1,'-k','LineWidth',2.2, 'MarkerEdgeColor','k','MarkerFaceColor
     ','r','MarkerSize',5)
  xlabel('K - Iteracoes','fontsize',12)
  ylabel('Residual','fontsize',12)
```

```
31  title('Heat2d Paralelo - Norma L2 (n=193)','fontsize',12)
32
33  subplot(2,2,2);
34  plot (xx2,'-k','LineWidth',2.2, 'MarkerEdgeColor','k','MarkerFaceColor
        ','r','MarkerSize',5)
35  xlabel('K - Iteracoes','fontsize',12)
36  ylabel('Residual','fontsize',12)
37  title('Heat2d Paralelo - Norma L2 (n=289)','fontsize',12)
38
39  subplot(2,2,3);
40  plot (xx3,'-k','LineWidth',2.2, 'MarkerEdgeColor','k','MarkerFaceColor
        ','r','MarkerSize',5)
41  xlabel('K - Iteracoes','fontsize',12)
42  ylabel('Residual','fontsize',12)
43  title('Heat2d Paralelo - Norma L2 (n=385)','fontsize',12)
```

Código Computacional B.17: Plotagem do residual da equação de Laplace transiente - plotresidheat2d3graf.m