Universidade Federal Fluminense

LEONARDO DINIZ

Estudo da Utilização de LAM/MPI com Suporte a Checkpoint/Restart

Volta Redonda

2015

LEONARDO DINIZ

Estudo da Utilização de LAM/MPI com Suporte a Checkpoint/Restart

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:

Diomar Cesar Lobão

Coorientador:

Gustavo Benitez
Tiago Neves

Universidade Federal Fluminense

Volta Redonda

2015

Estudo da Utilização de LAM/MPI com Suporte a Checkpoint/Restart

Leonardo Diniz

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:

Tiago Araújo Neves, D.Sc / MCCT-UFF

Jorge Alberto Rodriguez Durán, D.Sc / UFF/EEIMVR

Alexandre da Silva Galvão, D.Sc / Universidade Federal de São João del Rei, Campos Alto Paraopeba

Volta Redonda, 29 de maio de 2015.

Dedicatoria

Este trabalho é dedicado a meus pais, Irineu Diniz e Regina Helena Diniz.

Meu orientador da graduação e amigo de mestrado , Hélio Soares, descanse em paz e saudades.

Aos meus orientadores.

Aos grandes amigos que fiz nesses 5 anos de UFF.

Ao grande amigo Raphael Landim, que me trouxe o conhecimento do mestrado.

Agradecimentos

Não poderia deixar de agradecer primeiro a Deus, a seu filho unigênito nosso senhor Jesus cristo e a Maria santíssima pela fé e força para continuar quando a cruz foi muito pesada, por ter me julgado merecedor de estar e compartilhar de um ambiente com tantas pessoas que marcaram minha vida, pela fibra nas caminhadas da uff a minha casa em Barra Mansa que tantas vezes foram cansativas, e acima de tudo, por me permitirem chegar até aqui e escrever este agradecimento;

Aos meus pais, pois na pior fase da minha vida e mesmo sem condições na época, me acolheram novamente, me incentivaram a continuar nessa caminhada, que em nenhum momento deixaram abater-me com as dificuldades e sempre estiveram comigo;

Ao mestre com carinho, professor Lobão, que foi um marco e exemplo de vitória nessa minha passagem pela vida. Que Deus o abençoe sempre pois sei que foi um verdadeiro anjo da guarda pra mim. Confiou em mim quando muitos me pisaram e me acharam inepto, me incentivou quando muitos acharam que eu era um fracassado, e acima de tudo, foi e será um exemplo acadêmico a ser seguido por mim. Essas palavras são poucas para agradecer e nem chegam perto de tudo o que me proporcionou, mas saiba que se sou um ser humano melhor, agradeço em muito à seus conselhos.

Ao professor Benitez, pelos momentos de descontração e os momentos de empolgação com ideias para pesquisas como o out-of-core, o quadro iterativo e pelo convite a fazer parte do time de baseball.

Ao professor Tiago que foi um irmão mais velho aqui pra mim. Por toda atenção e paciência que teve quando comecei a trilhar a área de redes e software, que até então não faziam parte do meu cotidiano. Pela descontração na hora de montar os códigos e entender as meta-heurísticas que tanto me fizeram perder o sono.

Ao professor Duncan e meu amigo Cleidinei, pelo convite a ajudar a escrever o artigo do SEGET, proporcionando a mim mais um leque de conhecimento .

A todos os professores do programa de mestrado do MCCT, especialmente aos pro-

fessores Cleyton, Thadeu e Huguenin.

A todos os alunos do curso, especialmente a meu amigo Edgar pelas nossas eternas discussões sobre física e cálculo, ao Roberto e as varias vezes que rimos das coisas mais inusitadas e birutas que poderiam acontecer conosco, Júnior, Juliana, Marcão e Marco Antônio pela companhia e Hélio Soares, orientador da graduação, amigo do mestrado e que deixou muitas saudades.

Adriana e Naldinho da cantina da UFF, pelos momentos de atenção e carinho que proporcionaram a mim enquanto estive aqui.

A todas as pessoas que passaram pela secretaria, Larissa, Rose, Amanda, Lidiane e Karine que se esforçam em fazer um ótimo trabalho sempre, e que tiveram a maior paciência em me explicar os procedimentos burocráticos quando eu precisei de algo e que me aguentaram pedindo café toda hora.

A todos os funcionários da UFF , especialmente ao Pedrão e as meninas da limpeza, que sempre estavam a postos e que fazem seu trabalho com excelência.

E por fim, agradeço à CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pela concessão da bolsa durante todo o período de realização deste mestrado.

Resumo

O presente trabalho tem como objetivo implementar e disponibilizar o conhecimento necessário dos conceitos básicos, aplicações e vantagens do estudo da utilização do processamento paralelo/distribuído, utilizando o LAM/MPI com suporte a *checkpoint/restart* de dados e técnicas de out-of-core. Esta técnica é a execução direta de leitura e escrita em arquivos de dados. Estes estudos são executados em um cluster em nuvem com técnicas de NFS, ou seja, os computadores compartilham de uma mesma pasta visível a todos e por fim, serão utilizados softwares livres como o Fedora 14, que possui um conjunto de ferramentas e aplicativos muito úteis ao estudo, como compiladores Fortran e C. Serão mostradas as comparações entre códigos computacionais paralelos e seriais, com leitura e escrita de arquivos, em aplicações envolvendo distribuições de calor em domínios 1D e 2D e em produtos de matrizes quadradas. Finalmente, serão mostrados os resultados e revelando o ganho de tempo obtido envolvendo paralelismo, assim como os códigos utilizados no ambiente LAM/MPI.

Abstract

This paper aims to show how to implement and provide the necessary knowledge of the basic concepts, applications and advantages of studying the use of parallel processing/ distributed using the LAM/MPI with support for checkpoint/restart data and out-of-core techniques. This technique is the direct implementation of reading and writing files. This study are performed on a cluster cloud with techniques NFS, which is the computers share the same folder visible to all and ultimately free software will be used such as Fedora 14, which has a set of very useful tools and applications to the present study, such as Fortran and C compilers. Comparisons are shown between parallel and serial computer codes with reading and writing files in applications involving heat distributions in a 1D and 2D domains and products of square matrices. Finally, the results are displayed and revealing the time gain obtained involving parallelism, as well as the codes used in the environment LAM / MPI.

Palavras-chave

1. LAM/MPI
2. Cluster
3. Processamento Paralelo
4. Processamento Distribuído
5. Sistema Operacional Linux
6. Fedora
7. Sistema NFS de arquivos
8. Linguagem Fortran/C
9. GNU
10. Bibliotecas MPICH2

Glossário

LAM	:	Local Area Multicomputer;
MPI	:	Message Passing Interface;
COLL	:	Collective;
RPI	:	Request Progression Interface;
CR	:	Checkpoint/Restart
NFS	:	Net File System
FEDORA	:	Operational System
BLCR	:	Berkeley Lab Checkpoint/Restart
DHCP	:	Dynamic Host Configuration Protocol
IEEE	:	Institute of Electrical and Electronic Engineers
GNU	:	GNU's Not Unix
IPS	:	Internet Protocol
WEP	:	Wired Equivalent Privacy
DAEMON	:	Disk and execution monitor
LAMBOOT	:	Inicia o software LAM em cada um DOS NÓS especificados
SSH	:	Secure Shell

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este capitulo retrata as motivações e ideias básicas do LAM/MPI.

1.1 Motivação

Nos últimos anos, o mundo tem acompanhado um aumento exponencial na velocidade com que a informação se propaga pelo globo, gerando dúvidas e pesquisas de como processar tais informações o mais rápido possível. Para isso, o uso de supercomputadores tem sido requeridos para essa árdua tarefa.

Possuindo muitos processadores, grande capacidade de memória e um conjunto de *hardwares* de elevada tecnologia, supercomputadores são computadores com alto desempenho de processamento de informação. Porém, o custo de tais computadores é elevado, tornando sua aquisição muitas vezes inviável.

Uma prática que tem se mostrado muito útil é a técnica de processamento paralelo e distribuído. Basicamente, um ambiente específico de processamento paralelo consiste na utilização de computadores domésticos que através de *softwares*, responsáveis por monitorar, distribuir, executar e agrupar as informações, divide a mesma tarefa entre os processadores do computador, diminuindo o tempo de execução. O processamento distribuído divide a mesma tarefa em vários computadores, a fim de obter um resultado de tempo de execução próximo ou melhor ao da computação de alto desempenho em supercomputadores [**?**].

Um *cluster* é um conjunto de computadores que trocam informações entre si através do uso de uma rede de transmissão de dados de alta velocidade, dividindo entre si partes da execução de uma mesma tarefa, de forma a atuar como um único computador, ou um

grande supercomputador [?].

Atualmente, existem inúmeras ferramentas ou *softwares* que ajudam e facilitam a instalação de pacotes e gerenciamento de serviços nos clusters, possibilitando sua rápida montagem e com um baixo custo em relação aos supercomputadores [?].

Com base nas informações anteriores, o presente trabalho tem como objetivo implementar e disponibilizar o conhecimento necessário dos conceitos básicos, aplicações e vantagens do estudo da utilização do processamento paralelo/distribuído, utilizando o LAM/MPI com suporte a *checkpoint/restart* de dados e técnicas de out-of-core. Para esse fim, pretende-se estudar e utilizar um *software* de controle de processos paralelos e distribuídos conhecido como LAM/MPI, auxiliado por um módulo de suporte conhecido como checkpoint/restart.

1.1.1 Uso de uma estrutura LAM/MPI

A ideia principal é poder criar *softwares*, passando mensagens de forma útil, flexível, portátil, prática e eficiente em ambientes de memória distribuída, rede de workstations, etc. Existem vários modelos de paralelismo, um bem conhecido é o *SPMD-ME (Single Program, Multiple Data - Master-Workers)* [?].

Master-Workers: Um computador é nomeado como *Master* ou nó 0, a fim de coordenar os processos, enquanto que os outros são *workers* ou nós N(nó 1, nó 2 , ...nó n).

Multiple Data: Os dados (matrizes e vetores) são divididos e cada parte é enviada aos nós, ou seja, cada computador recebe informações diferentes.

Single Program: Cada nó executa o mesmo programa (nesse caso, o nó 0 também é responsável por executar parte do programa)

A ideia básica pode ser explicada de acordo com os seguintes passos:
i. Um computador, *Master/Receptor*, recebe a informação a ser processada;
ii. Transforma a informação em blocos de informações (chamados processos) e distribui para os outros computadores, chamados nós;
iii. Os nós realizam os cálculos intensos e logo em seguida retornam o resultado novamente para o *Master/Receptor*;
iv. O *Master/Receptor* reagrupa a informação e fornece o resultado esperado.

Resumindo, *Master/Receptor* coordena a execução do programa, o envio e o recebi-

mento das mensagens do *Master/Receptor* para os *Workers* e vice-versa.

1.2 Estrutura do presente trabalho

O Capítulo 2 contem os tópicos sobre a arquitetura e componentes de comunicação do LAM/MPI. No Capítulo 3 e 4 serão abordados respectivamente a arquitetura do cluster, como *hardware* e *softwares* utilizados e a instalação passo a passo do cluster. No Capítulo 5 serão apresentados estudos de caso utilizando programação paralela/ distribuída e finalizando no capítulo 6, as discussões e resultados.

Capítulo 2

LAM/MPI com Suporte a Checkpoint/-Restart

Este capitulo retrata um pouco da infra-estrutura do LAM/MPI.

2.1 Visão geral

LAM / MPI é uma implementação de código aberto do padrão MPI *(Message Passing Interface)* desenvolvido e mantido na Universidade de Indiana [?]. Ele utiliza o padrão MPI para compilação, controle e execução de códigos [?]. MPI é fruto de décadas de pesquisa e desenvolvimento de um conjunto de funções que permitem aos programadores criarem códigos computacionais paralelos de alto desempenho, ou seja, os códigos são previamente preparados para distribuição dos cálculos entre os processos, utilizando-se das bibliotecas do MPI, fazendo com que os processos comuniquem-se entre si [?]. O LAM/MPI é composto de 2 camadas, sendo a primeira a camada LAM*(Local Area Multicomputer)*, responsável pelo ambiente de execução dos códigos computacionais(RTE *run-time enviroment*), e a segunda composta pela camada MPI responsável pela comunicação dos processos ativos naquele dado instante. Ambas as camadas interagem com o sistema operacional [?].

2.1.1 Camada LAM

Sendo essa a camada, responsável pela compilação e execução dos códigos computacionais paralelos multicomputadores, ela detêm uma biblioteca auxiliar que permite que compiladores como C ou FORTRAN possam interagir com ela, gerando e executando códigos [?]. Essa camada funciona basicamente sobre a execução de *daemons*. *Daemons(*

disk and execution monitor) ou simplesmente gerenciador de disco e execução, são programas independentes que funcionam em segundo plano, sem interferência direta do usuário. Geralmente são responsáveis por controlar, iniciar, terminar um serviço provido pelo seu sistema, como requisições de rede, controle e execução de periféricos (DVDs, placas de rede, ...), etc [**?**].

O processo de utilização é bem simples. O usuário, necessitando executar códigos computacionais, via terminal ou através de *batch files*, deve utilizar o comando *LAMBOOT*, seguido do nome do arquivo de texto onde se encontram os nomes ou os IPs dos nós que farão parte da execução. Este por sua vez, inicia em cada nó, um *daemon* chamado *LAMD*, responsável por dar início em segundo plano a vários outros processos, como acesso remoto via *rsh* ou *ssh*, o *globus* para controle de aplicações remotas, etc [**?**]. Caso não ocorram erros, o MPI está pronto para ser utilizado. Caso necessite-se terminar o processo *LAMD*, utilizá-se o comando *LAMHALT*, finalizando o processo em todos os nós.

2.2 Arquitetura do LAM/MPI

O LAM/MPI em sua essência possui 4 grandes componentes, conforme mostrado na Tabela 2.1.

São eles: O *RPI*, responsável por coletar e processar as informações de comunicação de ponto a ponto ou nó a nó do MPI, como o *send* e *receive* [**?**][**?**]; O *COLL* responsável por coletar e processar as informações de comunicação coletivas do mpi [**?**][**?**]; O *CR* responsável por fazer o *checkpoint* e o *restart* dos processos[**?**][**?**]; Inicialização é responsável por executar o *LAMBOOT* na camada LAM [**?**][**?**].

Tipo de Componente	Módulos disponíveis
RPI	Gm, LAMD, tcp, sysv usysv
COLL	LAM, basic, smp
CR	BLCR
Inicialização	Bgroc, globus, ssh, tm

Tabela 2.1: Módulos disponíveis para cada tipo de componente [**?**]

2.2.1 Quadro de apoio(*Framework*)

Antes de executar o código computacional, geralmente em C ou FORTRAN, é necessário que o *LAM/MPI* possa lidar com os vários componentes, entre eles, de configuração, compilação, instalação e parâmetros arbitrários, tanto no tempo de compilação quanto no tempo de execução [**?**].

Os componentes precisam estar conectados estaticamente e dinâmicamente, além de estarem implementados como *plugins*. *Plugins* são pequenos programas que inserem funções em programas de maior porte, para um resultado mais específico.

2.2.2 Componentes de comunicação ponto a ponto

Entende-se por comunicação ponto a ponto quando um processo envia uma mensagem e um segundo processo recebe, ou seja, envolve troca de mensagem entre apenas dois processos distintos. Segundo o Centro Nacional de Processamento de Alto Desempenho existem 4 formas de comunicação ponto a ponto, sendo elas [**?**]:

• Síncrono: Operação sincronizada entre o processo que envia uma mensagem e o processo que recebe. O processo que envia, bloqueia a execução do programa até que ocorra uma operação de *receive* no processo destino.

• Disposto: Tipo de *send* que pode ser usado se o programador tiver certeza de que exista um *receive* correspondente, já ativo. Possui o menor índice de *overhead* total, no entanto, a execução de um *receive* no processo destino, deve preceder a execução de um *send* no processo de origem.

• Envio Buferizado: Operação de envio na qual se utiliza um novo *buffer* criado e adaptado ao tamanho dos dados que serão enviados. Isso pode ser necessário para evitar o processo de *buffering*, devido ao espaço padrão do *System Buffer*. Permite o controle no tamanho do *buffer*.

• Padrão: Operação básica de recebimento de mensagens. É usado para aceitar os dados enviados por qualquer outro processo. Possui excelente desempenho.

Para cada método mencionado acima, existem duas formas de processamento, sendo elas:

• Bloqueante: O dado deve estar salvo no *buffer* do sistema indicando que o pode ser reutilizado ou ter sido enviado com sucesso;

• Não Bloqueante: Diferente do bloqueante, o controle é da responsabilidade do programador, ou seja, a certeza de que o *buffer* do sistema esteja disponível para ser reutilizado.

2.2.3 Componentes de comunicação coletiva

Chama-se de comunicação coletiva quando os processos são alocados em grupos, onde uma parte comum de informação de um código é manipulada por todos os processos deste grupo.

Elas se utilizam das comunicações ponto a ponto, porém com um bloqueio onde cada processo no grupo será sincronizado, ou seja, cada processo será coordenado e sincronizado pelo seu parâmetro *communicator*.

As rotinas de comunicação coletivas estão divididas em três categorias

• sincronização;

• movimento de dados;

• computação global;

2.2.3.1 Rotina de Sincronização

MPI Barrier

A função da rotina *MPI Barrier* é sincronizar todos os processos de um grupo através do parâmetro *communicator*, ou seja, ao se utilizar o *MPI Barrier*, um processo de um grupo para sua execução, até que todos os processos do mesmo grupo também executem um *MPI Barrier*, sincronizando assim os processos.

2.2.3.2 Movimento de Dados

MPI Broadcast

Rotina que permite a um processo enviar dados, de imediato, para todos os processos de um grupo. Todos os processos do grupo, deverão executar um MPI *Brodcast*, com o mesmo *comm* e *root*. O processo identificado como *root*, enviará os dados, enquanto que o processo que não possui a identificação *root*, receberá os dados.

2.2.3.3 Computação Global

MPI Reduce

Uma das ações mais úteis em operações coletivas são as operações globais de redução ou combinação de operações. O resultado parcial de um processo, em um grupo, é combinado e retornado para um específico processo utilizando-se algum tipo de função de operação.

2.2.4 Componentes do checkpoint/restart

A técnica de *checkpointing* consiste na criação de um arquivo de descrição de um processo em execução, o qual pode ser utilizado para reconstruir o processo. Este arquivo contém uma imagem do estado de execução do processo em um dado instante de tempo. Isto possibilita que o processo possa continuar sua execução a partir do ponto onde o *checkpoint* foi realizado. Devido a esse comportamento, essa técnica é comumente chamada de *checkpoint/restart (C/R)*. Projeto *Checkpoint / Restart* do The Berkeley Lab (BLCR) é uma implementação de verificação de processos, que pode ser usado tanto para aplicações de checkpoint em um único nó, ou por múltiplos nós, em códigos paralelos em execução, criando uma imagem do estado de execução do processo em um dado instante de tempo, possibilitando que o processo possa continuar sua execução a partir do ponto onde o *checkpoint* foi realizado, esta imagem pode ser acessada e utilizada para recomeçar o processo *(restart)* caso necessite-se.

Berkeley *Checkpoint/Restart (BLCR)* é uma ferramenta de *checkpointing* [**?**].

2.2.5 Componentes de inicialização

Compilar aplicações que utilizam rotinas MPI depende inteiramente do tipo de implementação MPI no ambiente. Neste trabalho utiliza-se o Ambiente LINUX com compiladores GNU utilizando-se bibliotecas MPICH2. Tanto os comandos de compilação a seguir quanto, posteriormente, os comandos de execução devem ser executados via terminal.

Fortran77: mpif77 <fonte.f> -o <executável>

Fortran90: mpif90 <fonte.f90> -o <executável>

C Standard: mpicc <fonte.c> -o <executável>

C++: mpicxx <fonte.cxx> -o <executável>

Ao fim da compilação, inicia-se o executável via terminal.

mpirun -np x <executável>, onde x é o número de processos que serão utilizados para a execução do código.

Exemplo da utilização dos comandos acima citados:

Ao se compilar um arquivo nomeado como *teste.c*, contendo um codigo computacional paralelo/distribuido feito em C, deve-se usar o comando *mpicc teste.c -o teste*, onde será gerado um executável nomeado como *teste*. Com o executável *teste* gerado, devemos executá-lo com *mpirun -np 4 teste* onde 4 será o número de processos e o executável gerado se chamará *teste*.

Capítulo 3

Arquitetura do Cluster

Este capitulo aborda a infra-estrutura do cluster, como *hardwares* e *softwares* utilizados.

3.1 *Hardwares* Utilizados

Foram utilizados 4 computadores, sendo um deles o servidor NFS ou nó 0, denominado pc1 e os demais nós denominados como pc2, pc3 e pc4. O servidor NFS, assim como os outros nós, são exatamente homogêneos(idênticos), tanto em capacidade de processamento quanto estrutura de hardware para que não existam diferenças significativas de desempenho e resultados finais. Porém, isto não significa que todos os testes aqui apresentados não possam ser utilizados em configurações heterogêneas(diferentes) de nós.

Hardware dos computadores:

- Processador: Intel Core I7-2600 (8M Cache L3, 3.4 Ghz)
- Memória RAM: 8 GB DDR3 1333MHz, 3.1 Hardware 43
- Disco Rígido: HD Sata 1 TB - Samsung HD103SJ
- Gravadora de DVD: DVD RW Sata AD-7260S

Neste trabalho em particular, ocorreram duas fases em relação a comunicação entre os nós. Na primeira fase foi utilizado um Roteador, com taxas de transmissão de até 150MPBS sem fio, DIR-610 N150 de 4 portas, que foi configurado para que cada nó tivesse um endereço IP fixo para melhor administração dos mesmos. A seguir seguem as especificações do roteador:

- Padrões suportados: IEEE 802.11b/g/n, IEEE 802.3 ou IEEE 802.3u
- Modos de comunicação: Full-Duplex
- Segurança: Criptografia WEP 64/128-bit ou Criptografia Wi-Fi Protected Access (WPA/WPA2)

A segunda fase foi utilizada uma rede cabeada de comunicação de alto desempenho de 1 Gigabyte, onde não se tinha qualquer acesso por parte dos usuários ao servidor DHCP, que em primeira instância, distribuía a cada nó um IP dinâmico, o que levou a se fazer posteriormente uma configuração em cada nó para que se fixassem definitivamente os IPS recebidos sem alterações posteriores.

O *Cluster* foi montado utilizando-se os computadores localizados no laboratório D3 EEIMVR/VR UFF para testes e desenvolvimento de processamentos de alto desempenho conforme Figura 3.1, possuindo ar condicionado, boa iluminação e acesso restrito apenas a alunos e professores envolvidos em testes e/ou dissertações sobre processamentos paralelos e distribuídos. Os equipamentos (nós) foram colocados em fila, pois não é- necessário destacar quem será o *frontend*, além de ser uma estrutura clássica para um ambiente de ensino, uma vez que os nós também são independestes e podem ser utilizados nas aulas.

Figura 3.1: Laboratório de alto desempenho - D3

3.2 Softwares Utilizados

- Fedora versão 14

O Sistema operacional utilizado em todos os nós, nesse trabalho, foi uma distribuição de software livre, conhecida como *Fedora versão 14*, devido sua boa compatibilidade com o LAM/MPI e o BLCR. Segundo o Fedora project [**?**],"O Fedora é um sistema operacional baseado em Linux, uma coleção de softwares que torna seu computador operacional. Pode-se usar o Fedora em adição, ou no lugar de outros sistemas operacionais como o Microsoft Windows ou o Mac OS X, sendo completamente livre de custos para você utilizar e compartilhar". Resumindo, o Fedora é um sistema operacional feito por uma comunidade mundial de desenvolvedores conhecidos como Fedora project e é completamente livre para uso, estudo e compartilhamento. Fedora project é patrocinada pela Red Hat [**?**].

- Configurador gráfico de NFS

O *Software* utilizado para este trabalho como configurador gráfico de NFS.

Ele atua diretamente no arquivo /etc/exports, sem a necessidade de uma configuração escrita e manual mais delicada. Sua instalação é simples conforme demonstra Figura 3.2.

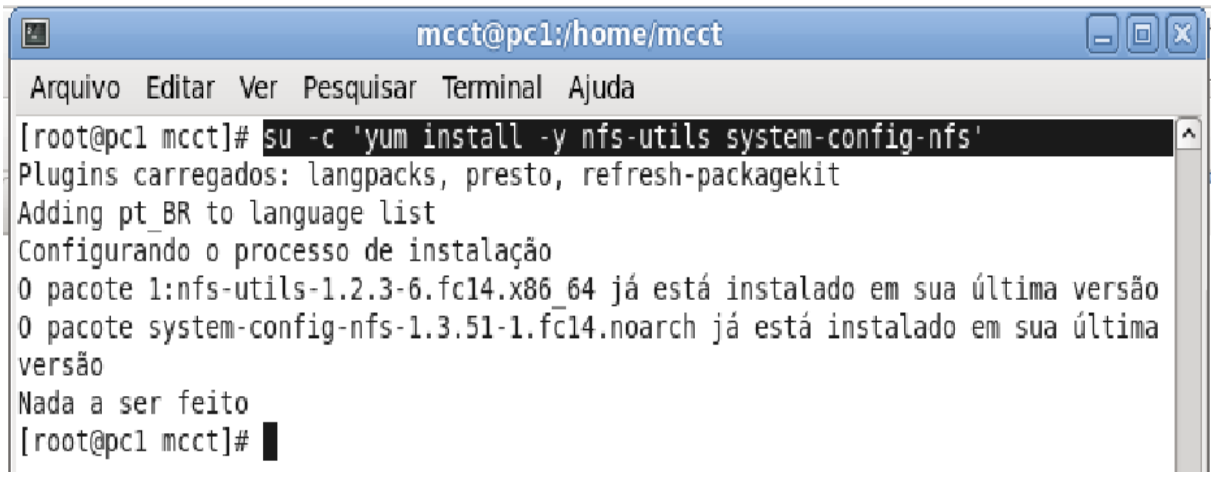

Figura 3.2: Tela capturada do terminal do fedora durante o processo de instalação do configurador do NFS

Conforme Figura 3.2, o software de configuração está instalado no sistema operacional, onde sua utilização e praticidade serão descritas no próximo capítulo, conforme figuras 4.8 e 4.10

- Linguagem C

É uma linguagem de programação estruturada muito utilizada. Por sua independência do *hardware* utilizado, suas bibliotecas de comunicação com *hardware* e por sua portabilidade para a maioria dos computadores, cresceu muito e teve muitas variações similares, porém com algumas incompatibilidades. 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. Para este trabalho em específico, foi utilizado o *software* livre GCC – GNU Compiler Collection que possui em seu pacote compiladores C e Fortran [**?**].

No próximo capítulo serão descritos os procedimentos necessários a instalação e configuração do ambiente LAM/MPI, para o *cluster* que será utilizado no presente trabalho.

Capítulo 4

Instalação

Este capitulo aborda a configuração necessária para a construção do cluster aberto.

4.1 Configuração do Conjunto de Programas e Bibliotecas do LAM/MPI em Cada Nó

Nesta seção, a instalação do LAM/MPI é abordada passo a passo, assim como o módulo CHECKPOINT/RESTART e demais programas auxiliares no sistema operacional. Todo o procedimento será executado via terminal, em todos os nós, que farão parte do cluster. Terminal é uma interface de comando onde executá-se outros programas ou serviços via comandos escritos.

4.1.1 Atualização

O primeiro passo a ser executado deve ser uma atualização dos programas que fazem parte do sistema operacional, pois, um programa desatualizado pode não ser executado como esperado, pode gerar erros na execução dos *daemons* ou dos códigos computacionais. Para isso, usa-se o comando *yum update* conforme demonstra a Figura 4.1. Toda vez que o comando é solicitado, é feita uma checagem dos repositórios, baixando os *headers* do pacotes e calculando as dependências ou pré-requisitos dos softwares antes de confirmar a instalação [?].

Figura 4.1: Tela capturada do terminal durante o processo de atualização do sistema operacional

Ao final da execução, todas as dependências do sistema estão atualizados e prontas para receberem instalações de outros aplicativos.

4.1.2 Compiladores C e Fortran

A função do compilador é traduzir um código computacional para uma linguagem de máquina, ou seja, traduzem códigos escritos em forma de texto, em uma linguagem específica como C ou FORTRAN, em um programa que podem ser executados e gerar resultados específicos.

Neste trabalho serão utilizados os compiladores C e Fortran, através do GCC, conforme Figura 4.2, onde é feita a instalação usando o seguinte comando destacado:

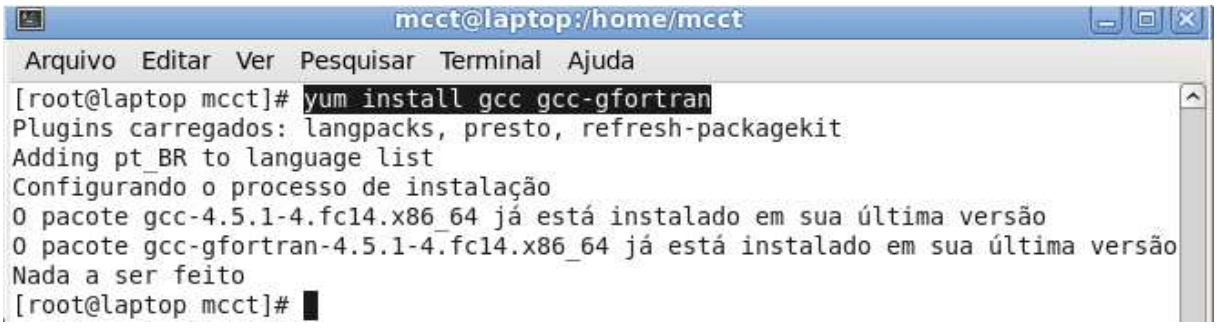

Figura 4.2: Tela capturada do terminal durante o processo de instalação dos compiladores C e FORTRAN

Ao final do instalação, o sistema uma informaçãono terminal que os compiladores C e FORTRAN estão instalados.

4.1.3 Cabeçalhos

Os *Kernel headers* são partes do código fonte, e são chamados quando o compilador precisa compilar um determinado programa. Em sua estrutura eles possuem funções, classes, variáveis, e etc. Os *headers* específicos de cada *kernel* contém o código para as funções de baixo nível(hardware), pois módulos se acoplam ao *kernel* , e como estes mesmos módulos não poderiam se comunicar com o *kernel* através de bibliotecas, eles têm que de fato compartilhar o mesmo código para determinadas funções. Para este trabalho usa-se o kernel 2.6.35.14-106.fc14.x86 conforme Figura 4.3. Deve-se reiniciar o sistema operacional neste momento.

Figura 4.3: Tela capturada do terminal durante o processo de instalação dos cabeçalhos

A instalação/atualização correta dos cabeçalhos são de extrema importância, conforme Figura 4.3.

4.1.4 LAM/MPI

Após feita a atualização dos *Kernel headers*, deve-se fazer download do pacote de instalação do LAM/MPI [**?**] .

A instalação do LAM/MPI exige a execução dos 3 comandos respectivos:

- *sudo ./configure –without-fc –with-rsh="ssh -x"*; onde *–without-fc* significa que não se levará em conta a versão do fortran que está instalado e *–with-rsh="ssh -x* para definir que será utilizado o *ssh* ao invés do rsh, que é nativo do fedora 14.

- *sudo make all*;

- *sudo make install*;

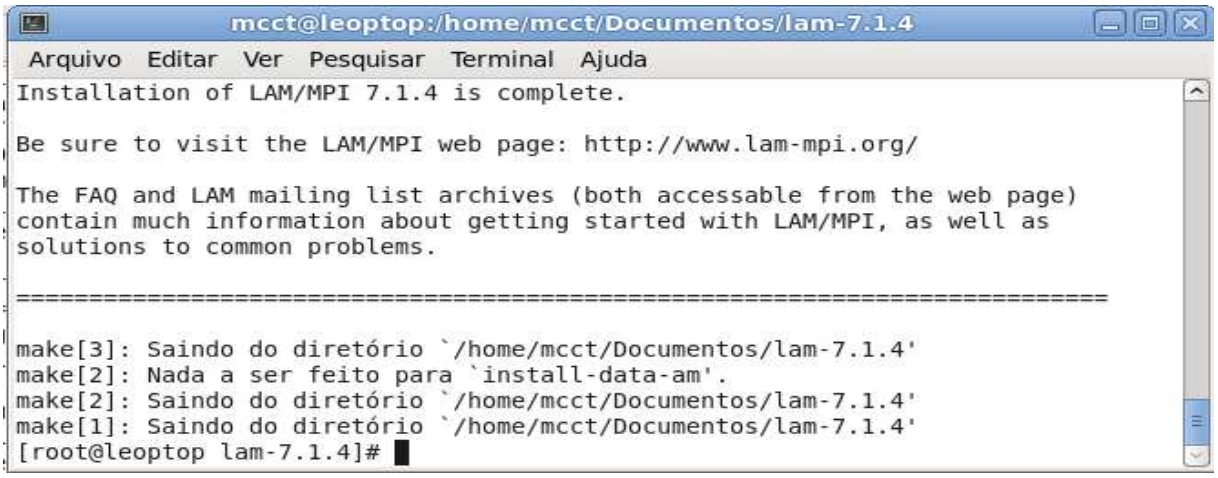

Figura 4.4: Tela capturada do terminal durante o processo de instalação do LAM/MPI

Conforme visto Figura 4.4, o LAM/MPI foi instalado e deve-se adicionar no arquivo de configurações do sistema operacional o local de execução do LAMMPI, para que as bibliotecas se tornem globais.

- Digitar no terminal *gedit /etc/ld.so.conf*
- Adicionar */usr/local/lib* conforme Figura 4.5

Figura 4.5: Instalação do LAM/MPI

- Ao final desse procedimento, a execução do LAM/MPI pode ser chamada globalmente, ou seja, de qualquer pasta via terminal. Para finalizar, ainda no terminal, digitar *ldconfig* para carregar imediatamente as bibliotecas.

4.1.5 BLCR(Berkeley Lab Checkpoint/Restart)

Como descrito antes, o BLCR é um *checkpoint/restart*. Neste trabalho, ele foi utilizado para fazer *checkpoints* dos processos que estavam sendo executados, para que no caso de falhas nos processamentos, eles seriam reiniciados para dar continuidade ao trabalho. Para todos os testes realizados no Capitulo 5, foi simulado um *restart* no último teste. Sem o

BLCR, qualquer processo que incorra em erro, seria perdido e todo o procedimento deveria ser reiniciado do inicio. O pacote contendo a instalação do BLCR pode ser acessado conforme referência [?].

Neste ponto começa a instalação do BLCR(*Berkeley Lab Checkpoint/Restart*) conforme Figura 4.6 e deve ser instalado sempre depois da instalação do LAM/MPI, pois caso seja instalado antes, podem ocorrer discrepâncias em relação as suas bibliotecas [?].

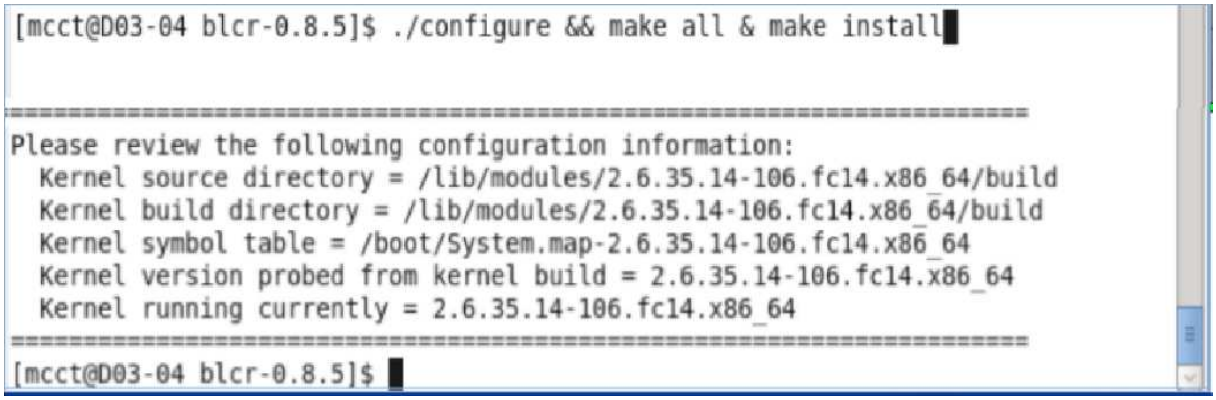

Figura 4.6: Instalação do BLCR

Neste ponto, o BLCR encontra-se instalado e pronto para uso. Porém, para que o LAM/MPI execute a chamada das funções do BLCR, é necessário que o sistema operacional carregue seus módulos. Para que não se precise ficar carregando via terminal as bibliotecas, deve-se inserir esses comandos, conforme Figura 4.7, ao fim do arquivo /etc/rc.local, que é executado na inicialização do sistema operacional.

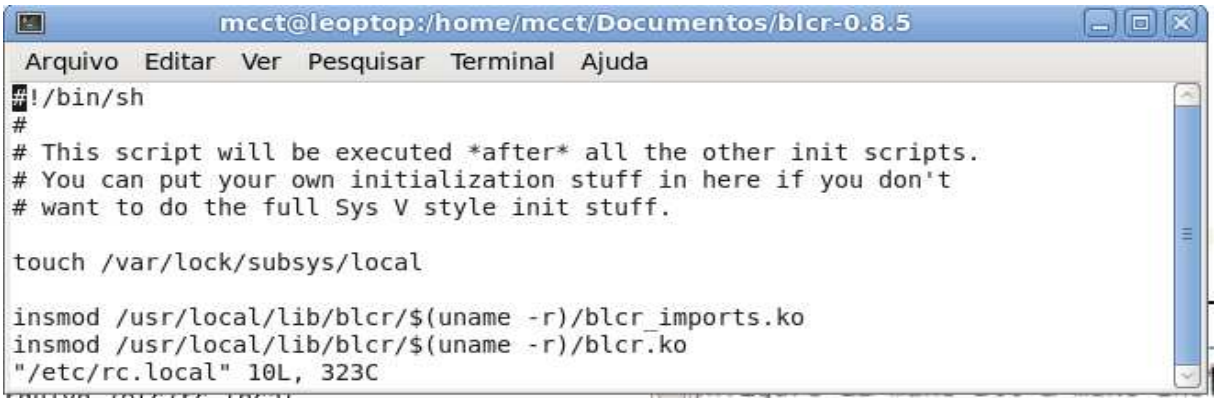

Figura 4.7: Conteúdo do arquivo rc.local

Feito isso, sempre que o sistema operacional iniciar e depois que todos os outros processos principais de *boot* forem executados, ele carregará as bibliotecas do BLCR.

4.2 Servidor e cliente NFS

O NFS ou Network File System(NFS) permite que um servidor possa compartilhar diretórios e arquivos com os outros nós do cluster, a fim de que os usuários e programas possam acessar arquivos como se eles estivessem armazenados localmente, tornando esse artifício extremamente importante para o LAM/MPI, pois cada nó deve possuir o código computacional.

Para isso, deve-se parametrizar o servidor para apontar qual pasta será compartilhada, que nós serão possíveis candidatos a ter acesso e se existirá a possibilidade de somente ler os arquivos ou *ler/escrever*, conforme Figura 4.8.

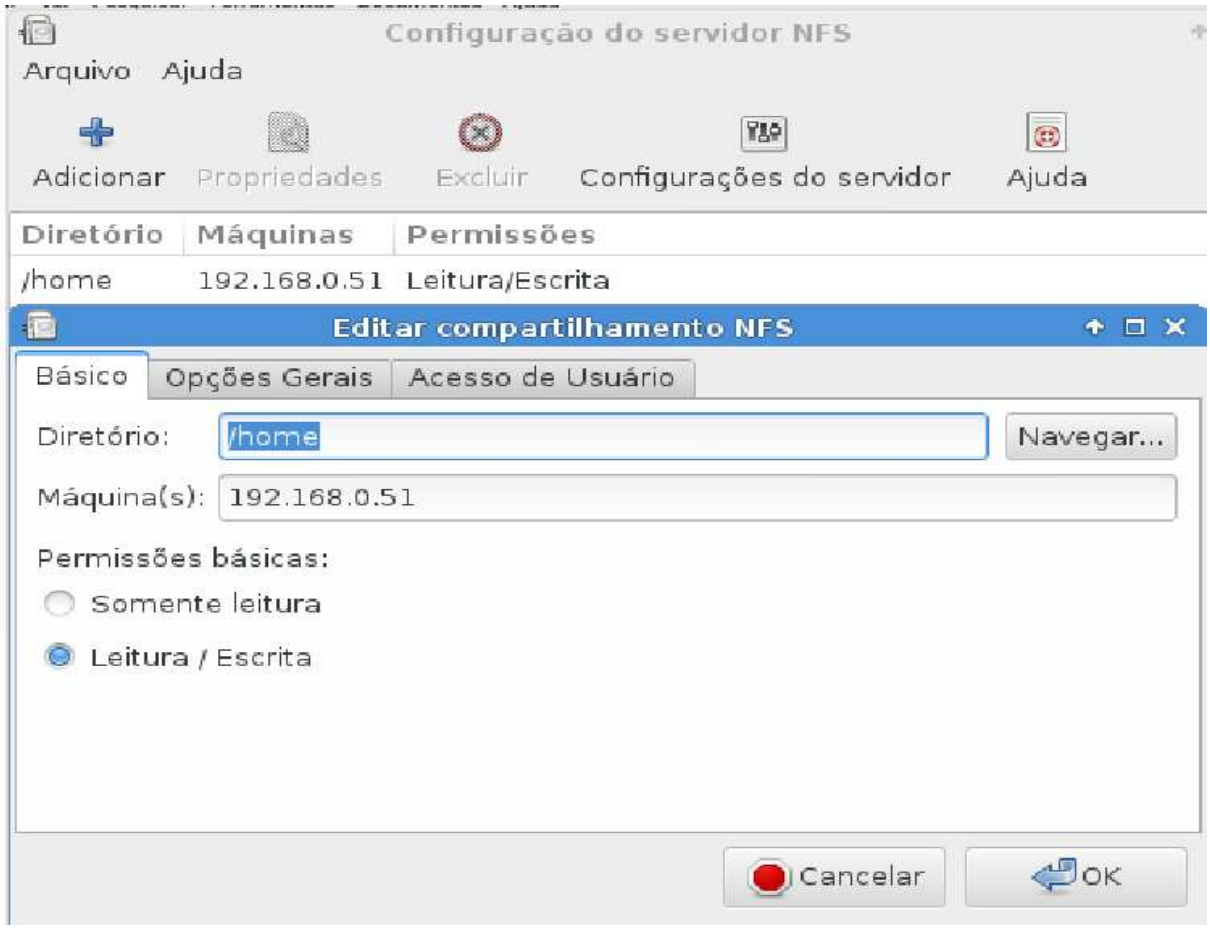

Figura 4.8: Configuração de nós do NFS

Agora, deve-se acessar o arquivo */etc/hosts.allow* e digitar nos parametros *lockd: mountd: rpcbind: rquotad: statd:* quais nós terão permissão de acessar a pasta compartilhada, conforme Figura 4.9, a seguir.

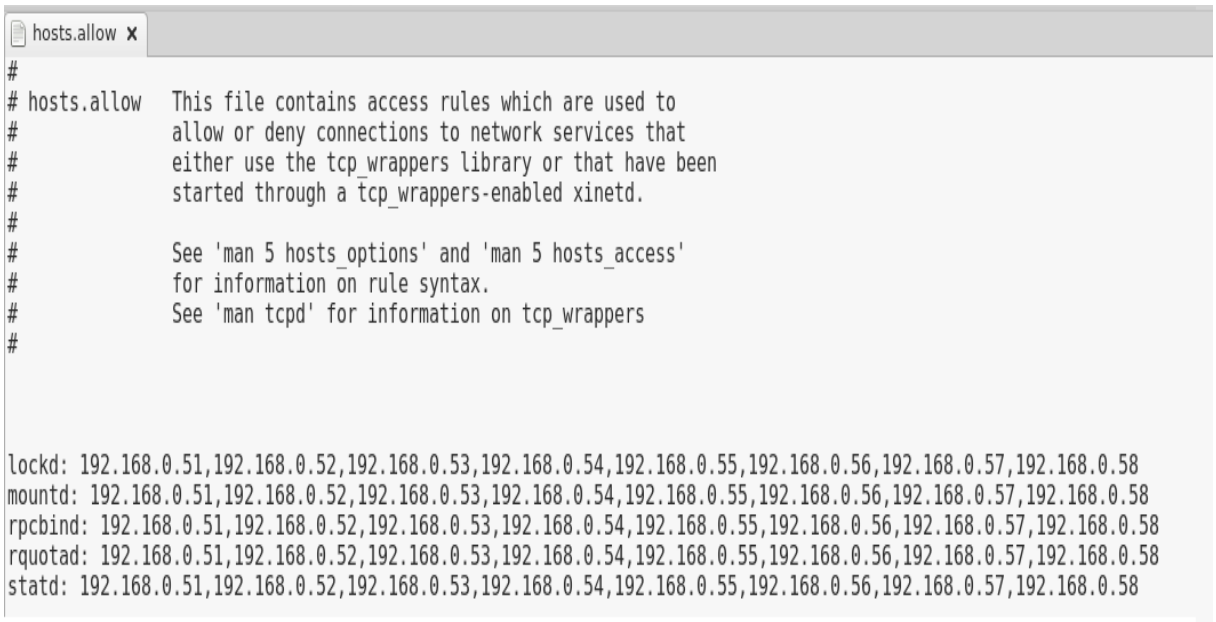

Figura 4.9: Conteúdo do arquivo hosts.allow

Conforme Figura 4.9, cada linha tem sua função e é comentada a seguir:

mountd: Esse processo recebe os pedidos dos nós e verifica se a pasta de arquivos solicitada existe e está compartilhada.

lockd: Este processo permite aos nós bloquearem arquivos no servidor.

statd: Este serviço avisa aos nós quando o serviço é reiniciado bruscamente.

rquotad: Este serviço fornece a cota de informações de usuário para os nós.

rpcbind: Mapeia e redireciona os pedidos dos nós para as portas e serviços requisitados.

Com isso, os nós cadastrados podem acessar a pasta compartilhada. Depois de configurado as pastas a serem compartilhadas e dito quais nós serão candidatos a ter acesso a elas, é necessário definir alguns parâmetros. Um ponto muito importante é a sincronização, ou seja, neste caso, sob demanda de utilização da pasta pelo código. Foram marcadas também as opções de utilização da porta 1024 para comunicação de utilização, conforme Figura 4.10.

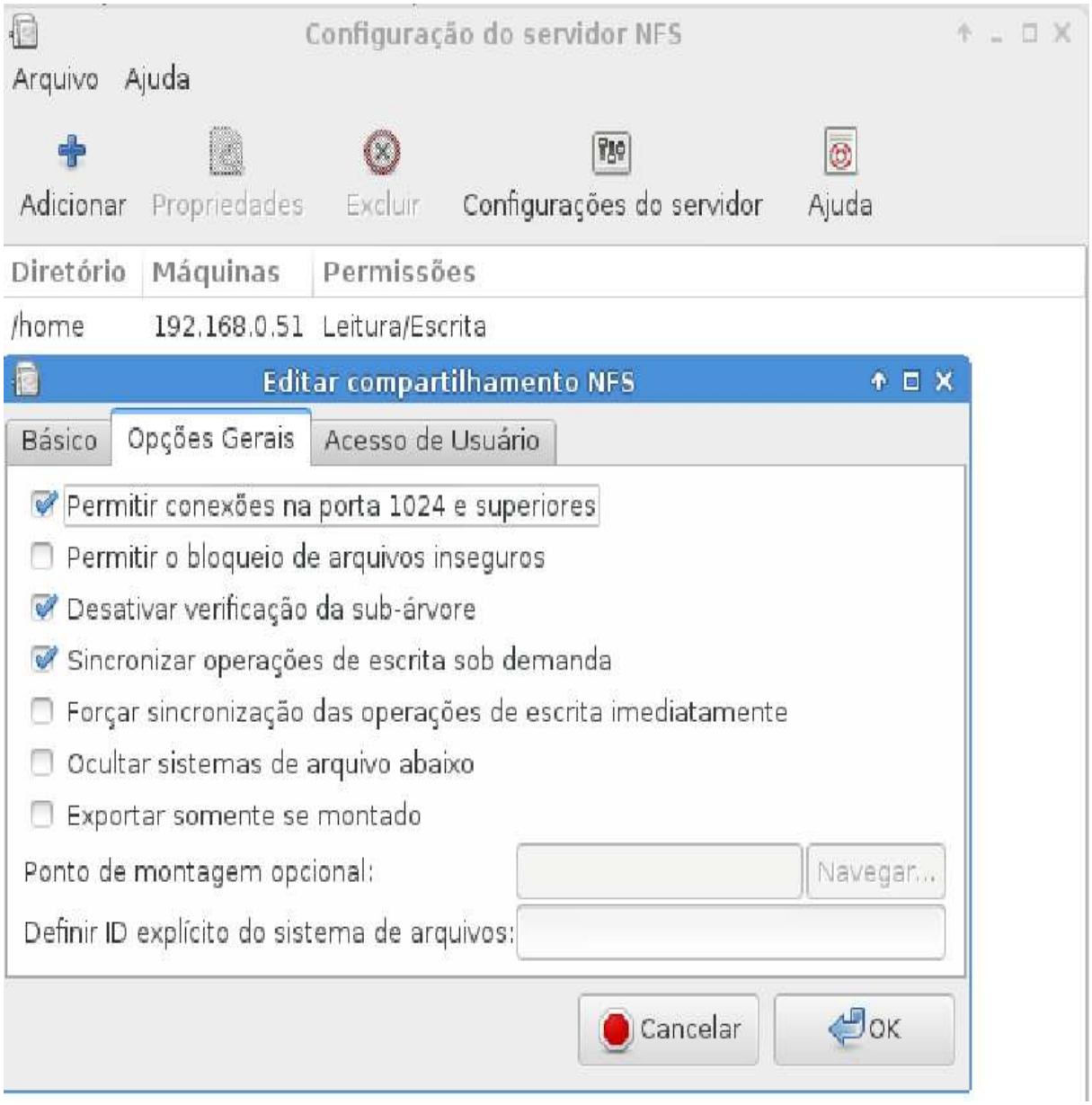

Figura 4.10: Configuração de acesso a pasta

O último ponto é o acesso de usuário *root* dos nós ao servidor. Caso seja necessário, permitir que os usuários *root* dos nós possam acessar o servidor como *root* local, conforme Figura 4.11. Os outros parâmetros foram desconsiderados para este trabalho.

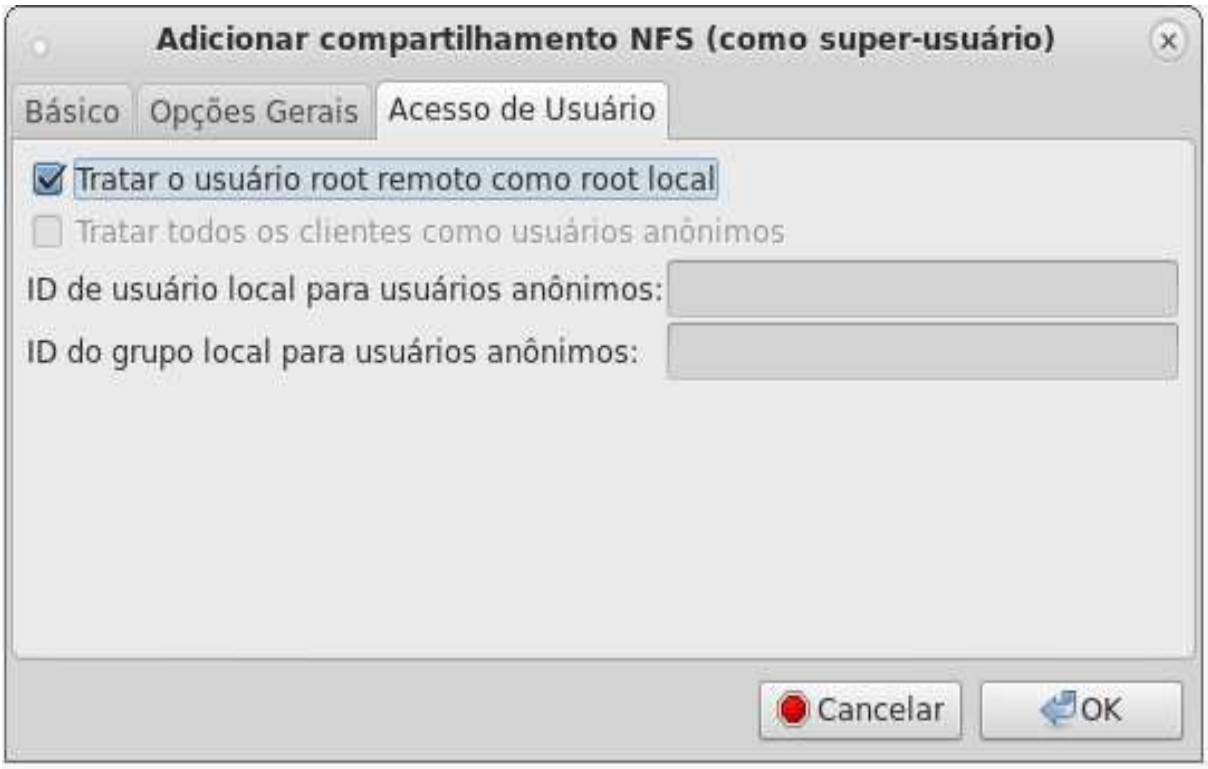

Figura 4.11: Configuração de usuário

Em seguida, os serviços responsáveis que são nfs, nfslock e o rpcbind, são inicializados conforme figuras 4.12, 4.13 e 4.14.

```
[root@server etc]# service nfs start
Iniciando os serviços NFS:                 [  OK  ]
Iniciando as quotas NFS:                   [  OK  ]
Iniciando o servidor NFS:                  [  OK  ]
Iniciando o NFS mountd:                    [  OK  ]
```

Figura 4.12: Instalação do NFS

```
[root@server etc]# service nfslock restart
Parando o NFS locking:                     [  OK  ]
Desligando o NFS statd:                    [  OK  ]
Iniciando o NFS statd:                     [  OK  ]
```

Figura 4.13: Instalação do NFSLOCK

```
[root@server etc]# service rpcbind restart
Parando o rpcbind:                         [  OK  ]
Iniciando o rpcbind:                       [  OK  ]
```

Figura 4.14: Instalação do RPCBIND

Para que os processos de *nfs,nfslock e rpcbind* não precisem ser chamados toda vez que o servidor for reinicializado, basta alocar os comandos na pasta /etc/rc.local no fim do arquivo. Conforme mostra Figura 4.15.

```
mcct@pc1:/home/mcct
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
#!/bin/sh
#
# This script will be executed *after* all the other init scripts.
# You can put your own initialization stuff in here if you don't
# want to do the full Sys V style init stuff.

touch /var/lock/subsys/local


insmod /usr/local/lib/blcr/$(uname -r)/blcr_imports.ko

insmod /usr/local/lib/blcr/$(uname -r)/blcr.ko

service nfs start
service nfslock restart
service rpcbind restart
```

Figura 4.15: Conteúdo do arquivo rc.local

Conforme Figura 4.15, o conteúdo do arquivo *rc.local* deve ficar dessa maneira, finalizado com a inicialização dos serviços *nfs, nfslock e rpcbind.*

4.3 Relação de segurança

4.3.1 Habilitar os serviços de acesso via ssh

A instalação do *ssh* substitui o antigo acesso via *rsh*, a fim de construir uma segurança maior contra espionagem, sequestro de conexão e outros ataques, pois fornece capacidades de tunelamento seguras e vários métodos de autenticação, já que os antigos acessos como *telnet, rlogin* e ftp transmitiam as senhas sem criptografia. O acesso *ssh* possui uma serie de procedimentos de acesso que incluem um protocolo formado de ip, endereço *MAC*, etc. todos criptografados. O comando deve ser executado como *ROOT*, e no caso desse trabalho, também no usuário *DEFAULT*.

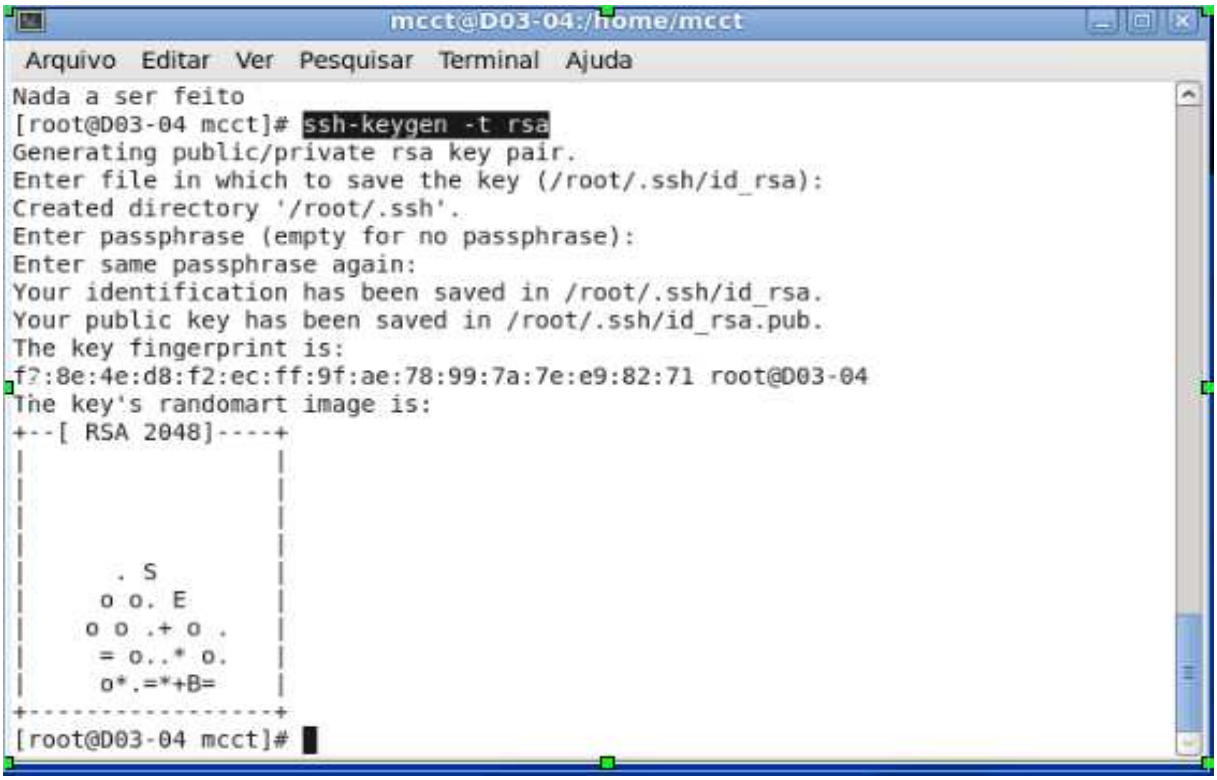

Figura 4.16: Instalação do ssh

Conforme Figura 4.16, em cada nó serão instalados as chaves públicas e privadas, que serão geradas na pasta (/root/.ssh). Neste momento, deve-se copiar e enviar a chave privada de acesso de cada nó, nomeada como *id rsa*, para a localização de chaves ou *pcs* autorizados dos outros nós, conforme mostra Figura 4.17, de forma que a conexão entre eles seja automática e sem senha, ou seja, cada nó deve enviar a chave privada de *ROOT* e do usuario *DEFAULT* para todos os outros *ROOT* e *DEFAULT* dos outros nós. Na primeira vez em que enviar a chave privada, será necessário digitar a senha do usuário do nó que receberá a chave.

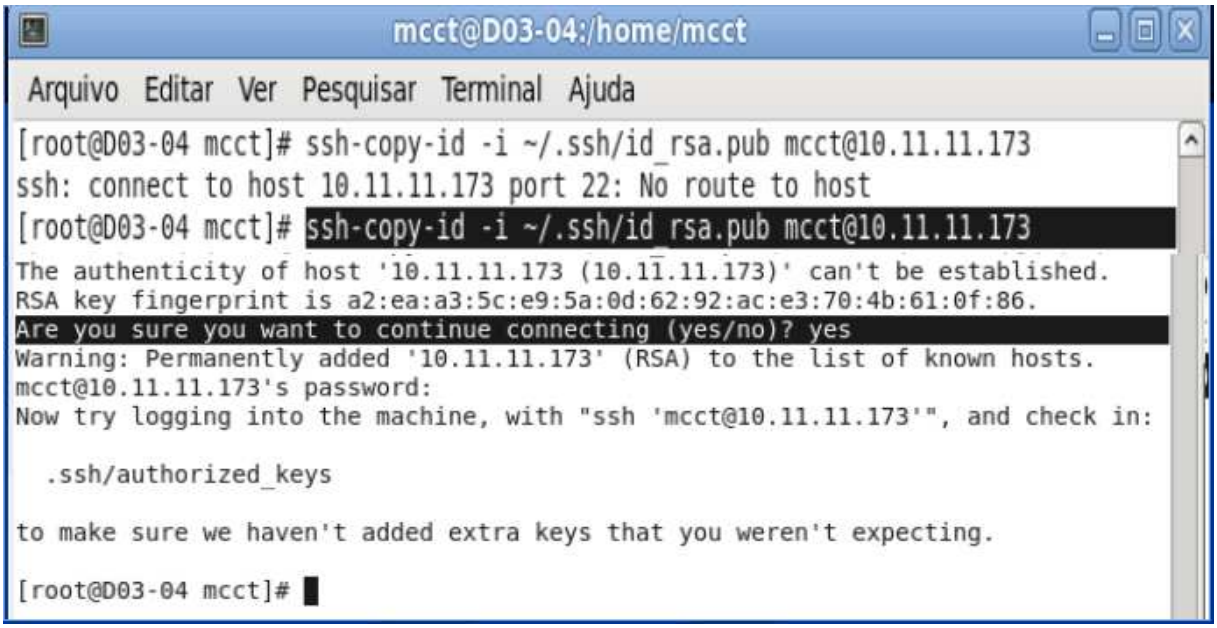

Figura 4.17: Instalação do ssh

O resultado desta última operação é a permissão de acesso sem senha entre nós. Essa segurança se faz necessária, pois caso haja uma invasão da rede e o intruso simule o endereço IP de um nó, existem outros parâmetros (endereço *MAC*, nome do nó, etc) que acusarão uma invasão e o acesso será bloqueado automaticamente. Esse acesso usa a porta 22, que deverá estar liberada no roteador.

Os fluxogramas das figuras 4.18, 4.19, 4.20, 4.21 e 4.22 mostram a construção do cluster.

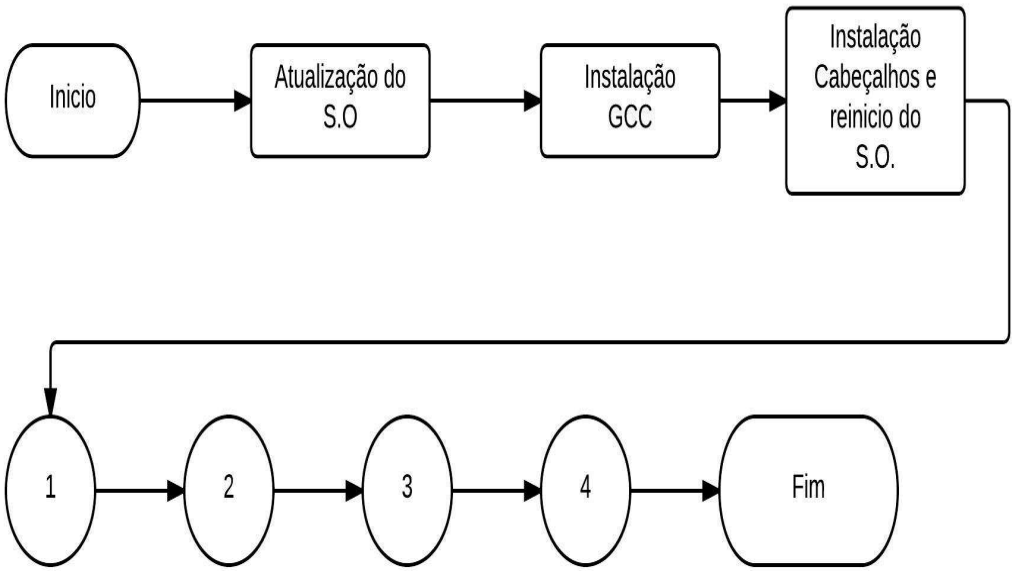

Figura 4.18: Instalação do cluster em nuvem

Veja as definições a seguir.

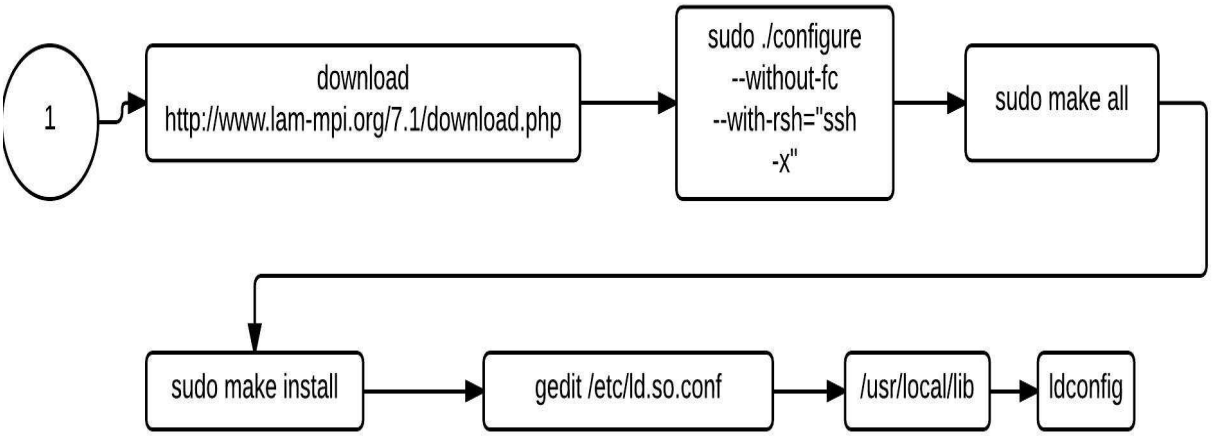

Figura 4.19: Instalação do cluster em LAM/MPI

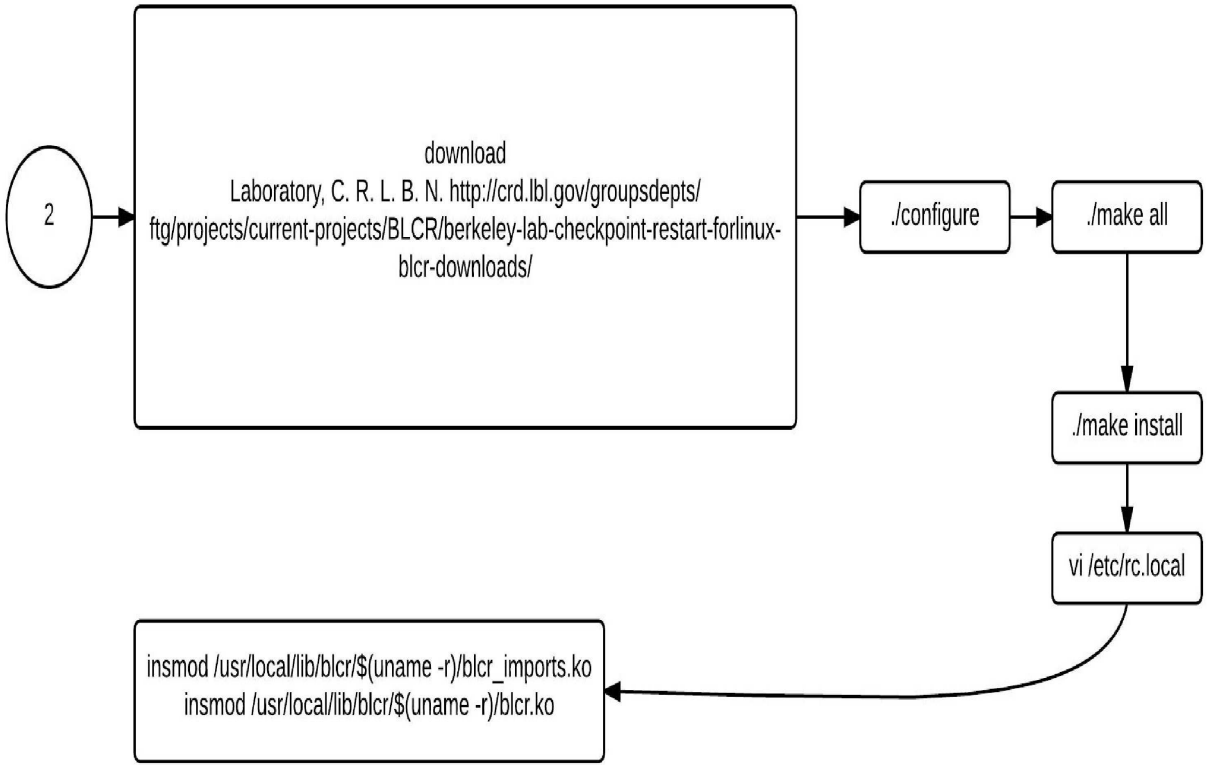

Figura 4.20: Instalação do BLCR

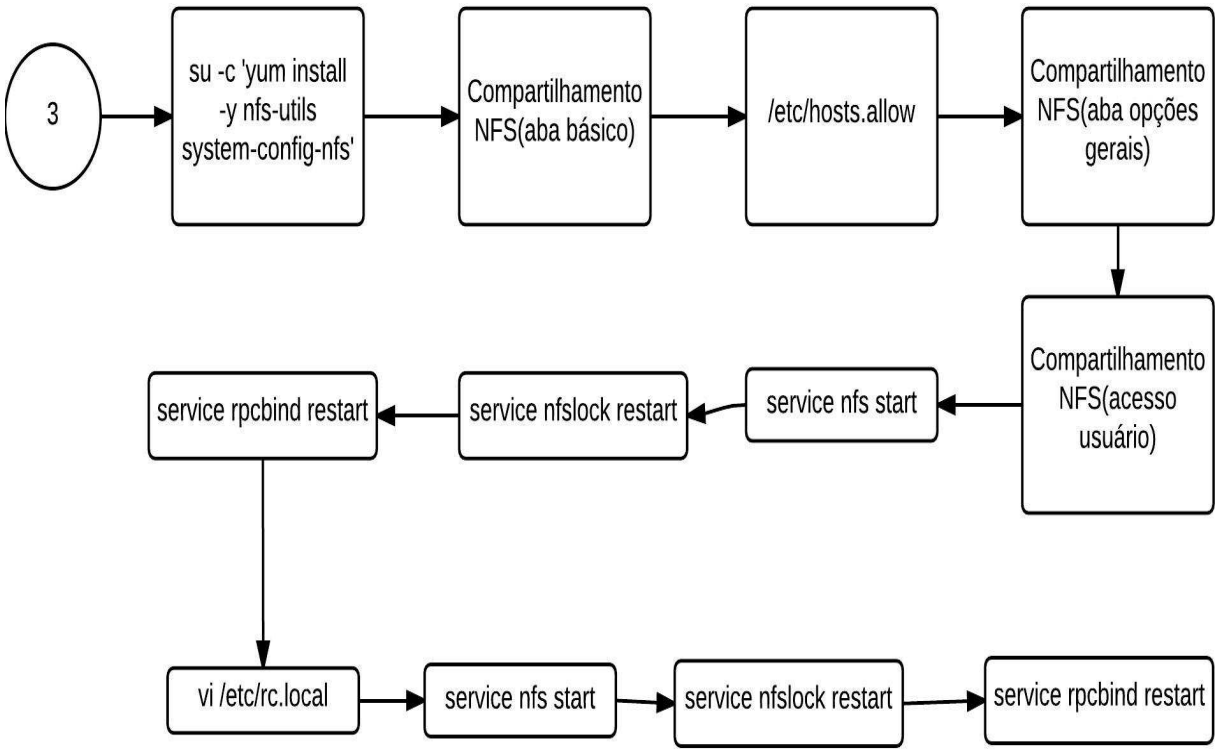

Figura 4.21: Instalação do NFS

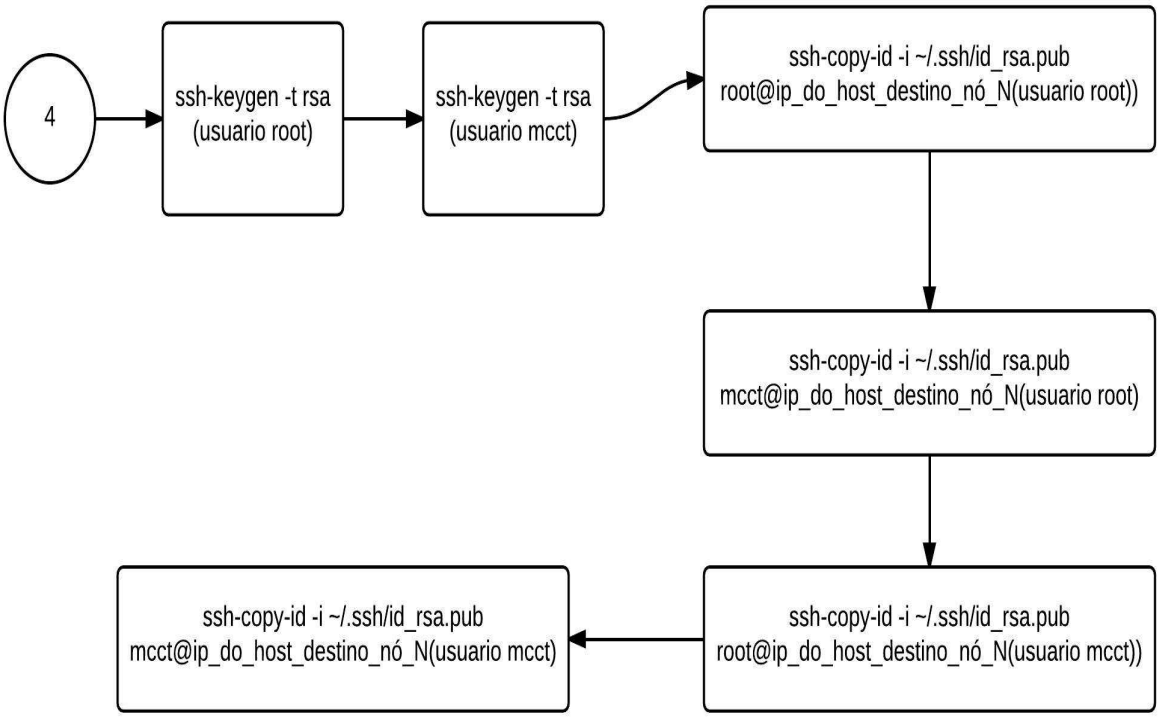

Figura 4.22: Instalação do ssh

Capítulo 5

Teste de validação

Este capitulo retrata dos testes, seus resultados e implementações

5.1 Distribuição de calor 1D

Para o teste abordado, usá-se um método explicito de regime não permanente de diferenças finitas para avaliar a distribuição de calor. Segundo a equação dada por

$$\frac{\partial T}{\partial t} + \frac{\partial^2 T}{\partial x^2} = 0 \tag{5.1}$$

sendo t o tempo, x o espaço e T a temperatura. A solução da equação 5.1 é obtida utilizando diferenças finitas de primeira ordem avante para a derivada temporal e aproximação de segunda ordem centrada no espaço para a derivada de segunda ordem no espaço.Desta maneira se escreve

$$T_i^{n+1} = -\frac{\Delta t}{\Delta x^2}(T_{i-1}^n - 2T_{i-1}^n + T_{i+1}^n) + T_i^n \tag{5.2}$$

Com condições de contorno imutáveis do tipo Dirichlet nas fronteiras de T=100. É utilizado um vetor de 500 posições para discretizar o espaço. De imediato, cria-se para cada processo um vetor de 500 posições, onde cada processo é responsável pelo seu domínio. Entende-se domínio, como ponto de inicio e fim, que é passado via *send/receive* pelos processos vizinhos, ou seja, num vetor com 500 posições, e tendo 2 processos, cada processo criaria seu vetor de 500 posições e atuaria em apenas 250 pontos, tendo um excesso de 250 pontos.

Contudo, se houvesse um vetor de tamanho igual ao número de pontos a serem analisados pelo processo, seria extremamente útil, pois gastar-se-ia menos espaço de memória, o carregamento seria agilizado e a transferência de dados das fronteiras seria mais simples,

tendo apenas que utilizar o *send/receive* para envio do inicio e do fim.

Usando o exemplo, sendo 500 pontos a serem analisados, e houvesse 2 processos, cada processo criaria apenas um vetor de 250 posições e necessitaria apenas das fronteiras (início e fim) para se trabalhar.

Na presente seção, usa-se 2 processos para um vetor de 500 posições, onde usa-se 2 processos. Cada processo cria um vetor de 250 posições mais 2 posições (para recebimento do início e fim do processos vizinhos). No primeiro ponto do primeiro processo, o valor de fronteira é fixado em 100, da mesma forma que o último ponto do último processo receberá o valor de 100.

Inicialmente, determina-se o número de pontos que serão trabalhados no vetor. No presente caso será igual a 500. A seguir, encontra-se a parte de um código em C que executa essa tarefa.

```
int main(int argc, char *argv[])
{
int n=500;
}
```

Em seguida, declara-se a variável *size*, que no momento da execução, receberá o número de processos (*-np x*) que serão executados naquele dado instante através do processo *mpi comm size*.

```
int  size,nameSize;
```

Ao se executar em seguida o comando *mpirun -np 2 teste*, por exemplo, a variável size receberia o valor 2, indicando 2 processos naquele dado instante.

Com isso executado, pode-se agora determinar a variável vetor e o tamanho que ela terá para se trabalhar no código.

Define-se as variáveis *uold* e *unew* para representar nesses vetores. Usa-se uma divisão simples igual a (n / size), ou seja, o número de pontos dividido pelo pelo número de processos. Nesse caso tem-se 500 / 2 = 250. Então cada processo tem que criar um vetor de 250 posições apenas para seu trabalho (posição 1 até 250), e não mais um vetor de 500 posições para cada processo, conforme Figura 5.1. Criar-se-a também mais 2 posições para recebimento do início e fim do processos vizinhos, ficando (500 / 2)+2=252. Veja fragmento de código em C

```
inicio = 1

fim = (n/size);
```

Dessa forma, assegura-se que cada processo criará apenas um vetor do tamanho que necessita e duas posições a mais apenas para recebimento de fronteiras vizinhas. Para o *rank* zero, a contagem deve começar em 2, pois a posição 1 possui a condição de contorno igual a 100 e a posição zero nao esta sendo utilizada, enquanto que no último *rank*, o fim deve ser o tamanho do vetor menos 1, pois a última posição recebe o valor de 100, que também é uma condição de contorno. Deve-se zerar os pontos do vetor, ignorado a posição 0, antes de começar e logo depois alocar as condições de contorno no valor de 100 na primeira posição do vetor no nó 1 e na última posição do vetor no nó N,conforme figura 5.1. Veja fragmento de código em C.

```
for (i=(inicio-1);i<=(fim+1);i++)
{
uold[i] = 0.0;
        unew[i] = 0.0;
}
uold[1]=100.0;
unew[1]=100.0;
uold[(n/size)]=100.0;
unew[(n/size)]=100.0;
```

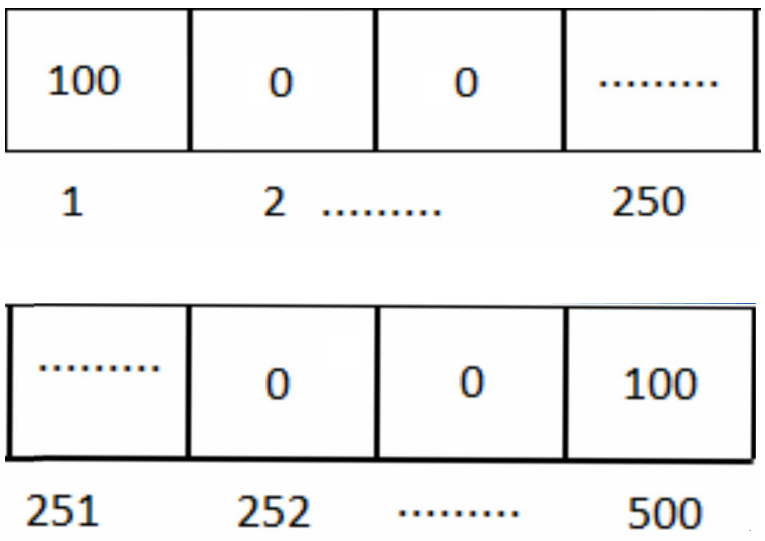

Figura 5.1: Vetor inicial

em seguida, adota-se o número de iterações que serão realizadas no processo. Lembrando que quanto mais iterações houver,mais preciso será o resultado. Adota-se k como contador, e *maxk* como limite de iteração, sendo seu valor igual a 10000. Veja fragmento de código em C.

```
int k,maxk = 10000;
for(k=1;k<=maxk;k++)
   {
   .
   .
   .
   }
```

Agora, cada processo percorrerá cada ponto do vetor e a cada ponto analisado fará uma diferença entre seus pontos vizinhos *(uold[i-1]+ uold[i+1])/2*, conforme Figura 5.2. Um contador i será adotado para esse fim, indo do inicio ao fim do vetor. Veja fragmento de código em C.

```
for(i=inicio;i<=fim;i++)
{
unew[i] = ((uold[i-1]+(uold[i+1]))/2);
}
```

Ao fim desse processo, cada ponto do vetor sofre uma atualização para que se possa recomeçar o processo. Sem esse artificio, depois da primeira iteração, nada se alteraria.

```
for(i=inicio;i<=fim;i++)
{
uold[i]= unew[i];
}
```

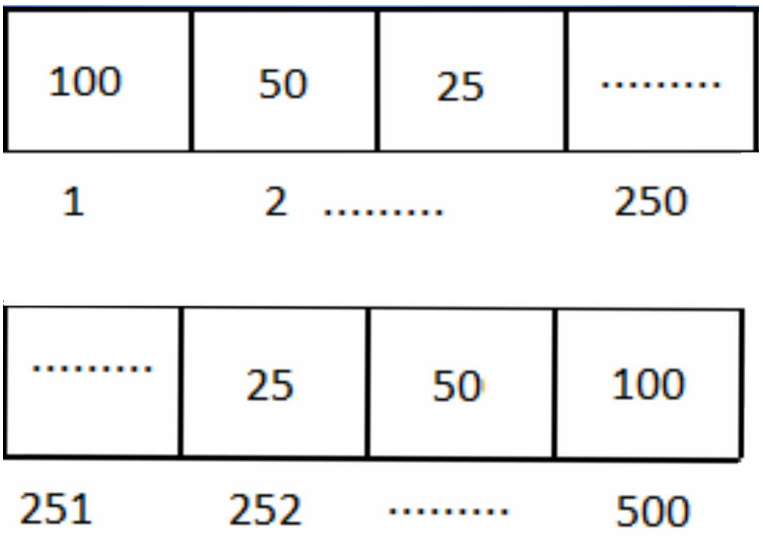

Figura 5.2: Distribuição de calor.

Neste momento, cada processo envia suas condições de contorno para os outros nós,conforme Figura 5.3 para as futuras iterações que serão alocadas nas posições extras que cada nó possui, conforme descrito anteriormente.

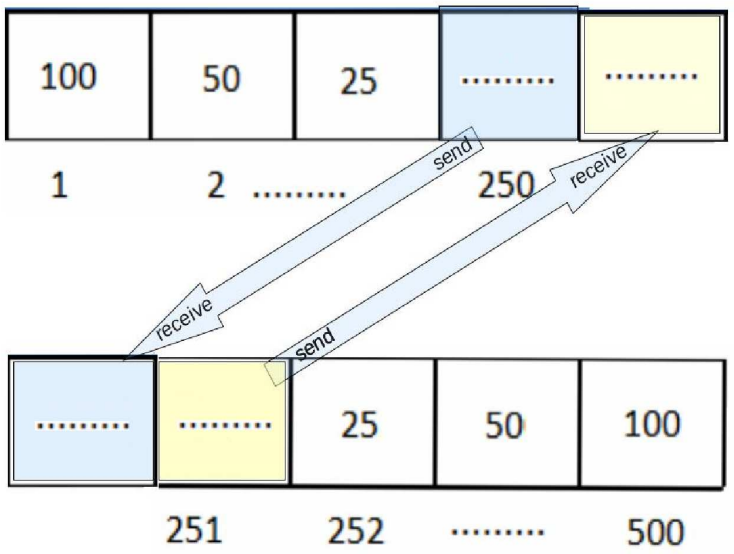

Figura 5.3: Distribuição de calor

A cada 1000 iterações, um *checkpoint* do processo é feito, de maneira que se algum nó sofresse uma parada qualquer que fosse, o processo poderia ser ativado novamente, conforme Figura 5.4.

```
[mcct@pc1 compartilhado]$ mpicc parabola_final.c -o par
[mcct@pc1 compartilhado]$ mpirun -np 2 par
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
checkpoint
[mcct@pc1 compartilhado]$
```

Figura 5.4: Checkpoint sendo feito a cada 1000 iterações.

Ao final desse processo, gera-se um arquivo de texto com os resultados da distribuição de calor em 1 dimensão discretizado de 500 pontos, conforme Figura 5.5. Observa-se que o resultado estabelece a distribuição de calor das fronteiras para o interior do domínio como esperado.

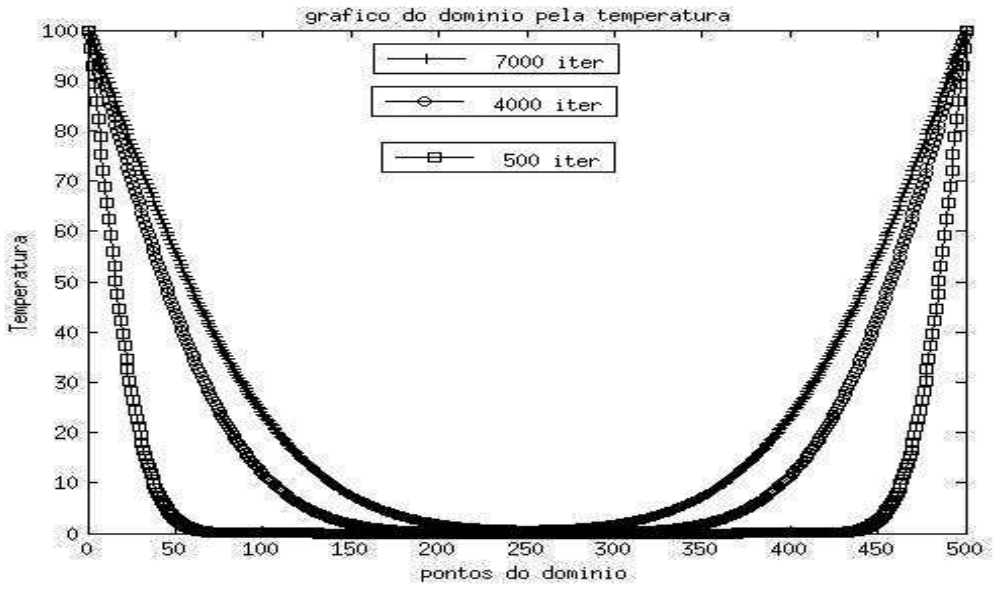

Figura 5.5: Distribuição de calor.

5.2 Produto de Matrizes Densas

Para o caso a ser abordado a seguir, usa-se um método de produto de matrizes, onde trabalha-se com 2 matrizes densas de m dimensões variando entre 600x600 até 2000x2000 posições alocadas em arquivos *.txt* distintos. São matrizes consideráveis e servem para o exemplo, mas neste momento adota-se nas matrizes 1000x1000 através da equação 5.2:

$$A[i,j] * B[i,j] = C[i,j] \tag{5.3}$$

Cria-se um arquivo de texto com a matriz A de 1000x1000 elementos. De imediato, teria-se que percorrer o arquivo onde ela se encontra e criar para cada processo uma matriz de 1000 x 1000 posições, onde cada processo seria responsável pela sua área de atuação. Entende-se área de atuação como ponto de início e fim, que é passado via *send/receive* pelos processos vizinhos, ou seja, uma matriz de 1000x1000 posições, contendo 4 processos, cada processo criaria uma matriz de 1000x1000 posições e atuaria em apenas 1/4 dela, tendo um excesso do restante.

Porém, dividindo-se a matriz em linhas por processo e criar matrizes de tamanho igual ao número de pontos a serem analisados pelo processo, seria extremamente útil, pois se gastaria menos espaço de memória, o carregamento seria agilizado e a transferência de dados seria mais simples.

Usando o exemplo anterior, considerando-se a matriz de 1000x1000 elementos a serem analisados, e contendo 4 processos, cada processo criaria apenas uma matriz de 250x1000 posições e necessitaria de uma menor área de espaço de memória para se trabalhar.

A primeira tarefa a se fazer, é determinar o número de elementos que serão trabalhados nas matrizes. No presente caso será igual a 1000x1000.

O próximo passo importante é declarar a variável *size*. Através dela, é possível receber o número de processos (*-np x*) que serão executados naquele dado instante quando executar o código. Então, ao executar o comando *mpirun -np 5* teste, por exemplo, a variável *size* receberia o valor (5 - 1), indicando 4 processos naquele dado instante, pois ignora-se o nó 0(*master*), que não será atuante, apenas para um melhor controle dos processos.

Feito isso, a variável matriz recebe o valor 4, sendo este o tamanho que ela terá para

se trabalhar o código. Usa-se um divisão simples igual a

$$\frac{n}{size - 1} \tag{5.4}$$

, ou seja, o número de linhas dividido pelo números de processos. Nesse caso tem-se 1000 / (5-1) = 250. Então cada processo teria que criar um matriz de 250 linhas (posição 1 ate 250). Mantendo as colunas pois não será necessário modificá-las, conforme Figura 5.6.

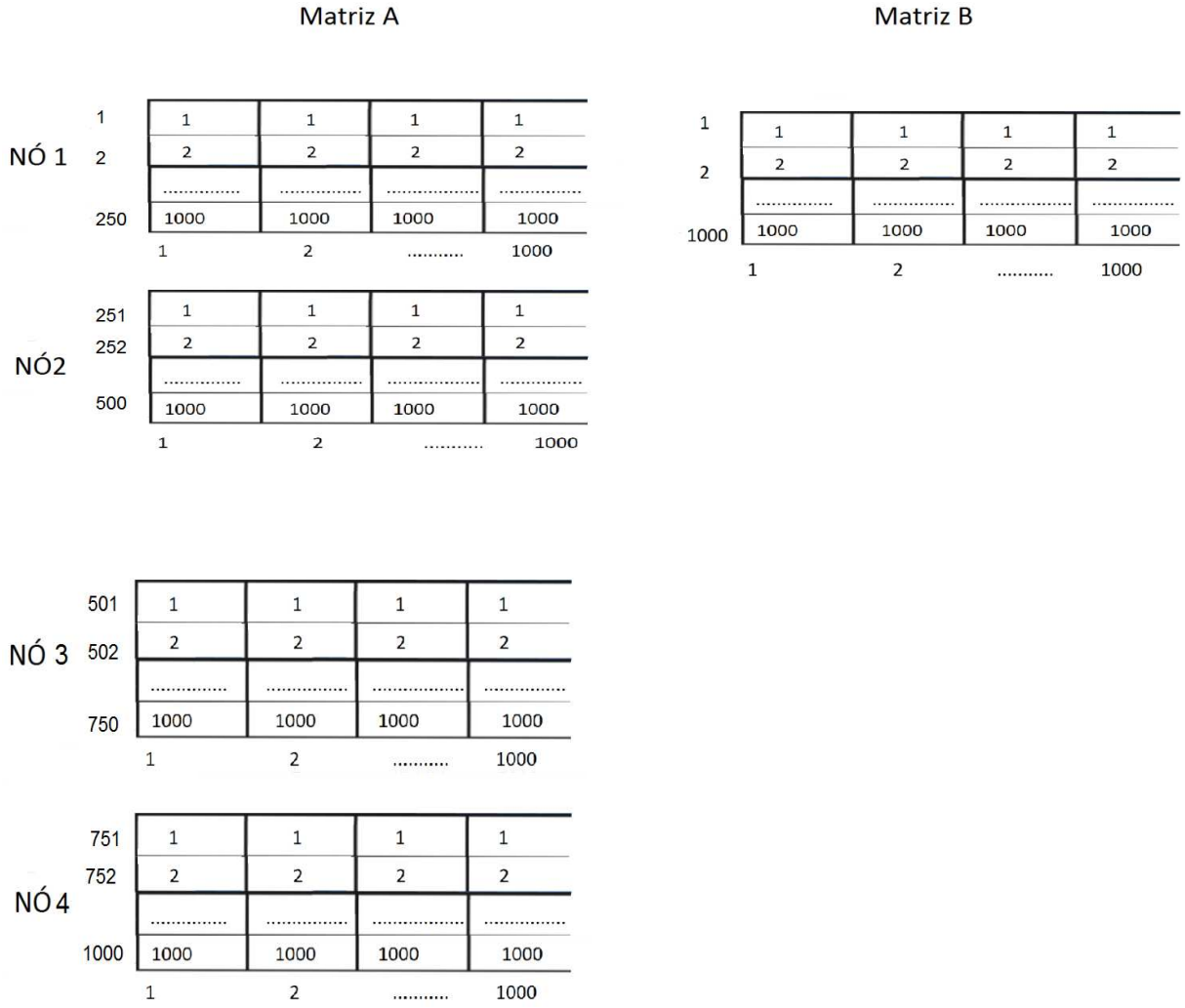

Figura 5.6: Produto de Matrizes

Dessa forma, assegura-se que cada processo criará uma matriz contendo apenas a quantidade de linhas em que atuará e sem mudanças na quantidade de colunas.

Deve-se zerar os pontos da matriz e, só então, cada processo deverá ler a respectiva parte do arquivo da matriz A que lhe corresponde e alocá-las nas matrizes criadas. A partir desse momento, com a matriz A dividida e alocada, deve-se ler o arquivo da matriz B e começar a multiplicação. Lembrando que cada processo lerá a matriz do arquivo

B, ponto a ponto, onde uma variável auxiliar será a responsável por guardá-lo e fazer os cálculos necessários. Sabendo-se que o produto de matrizes, se dá pela equação 5.4:

$$(A * B)[i, j] = A[i, 1]B[1, j] + A[i, 2]B[2, j] + ... + A[i, n]B[n, j] \tag{5.5}$$

Porém, com o intuito de agilizar o processo, utilizá-se um artificio de transposição. Dessa forma, aproveita-se de melhor forma a evolução da leitura do arquivo da matriz B, pois a primeira coluna das matrizes A dos processos deve ser multiplicada por toda primeira linha da matriz B, a segunda coluna das matriz A deve ser multiplicada por toda segunda linha da matriz B e assim por diante, onde ao final da leitura do arquivo B tem-se um arquivo de texto com o produto, alcançando-se um resultado esperado e rápido, conforme Figura 5.7 .

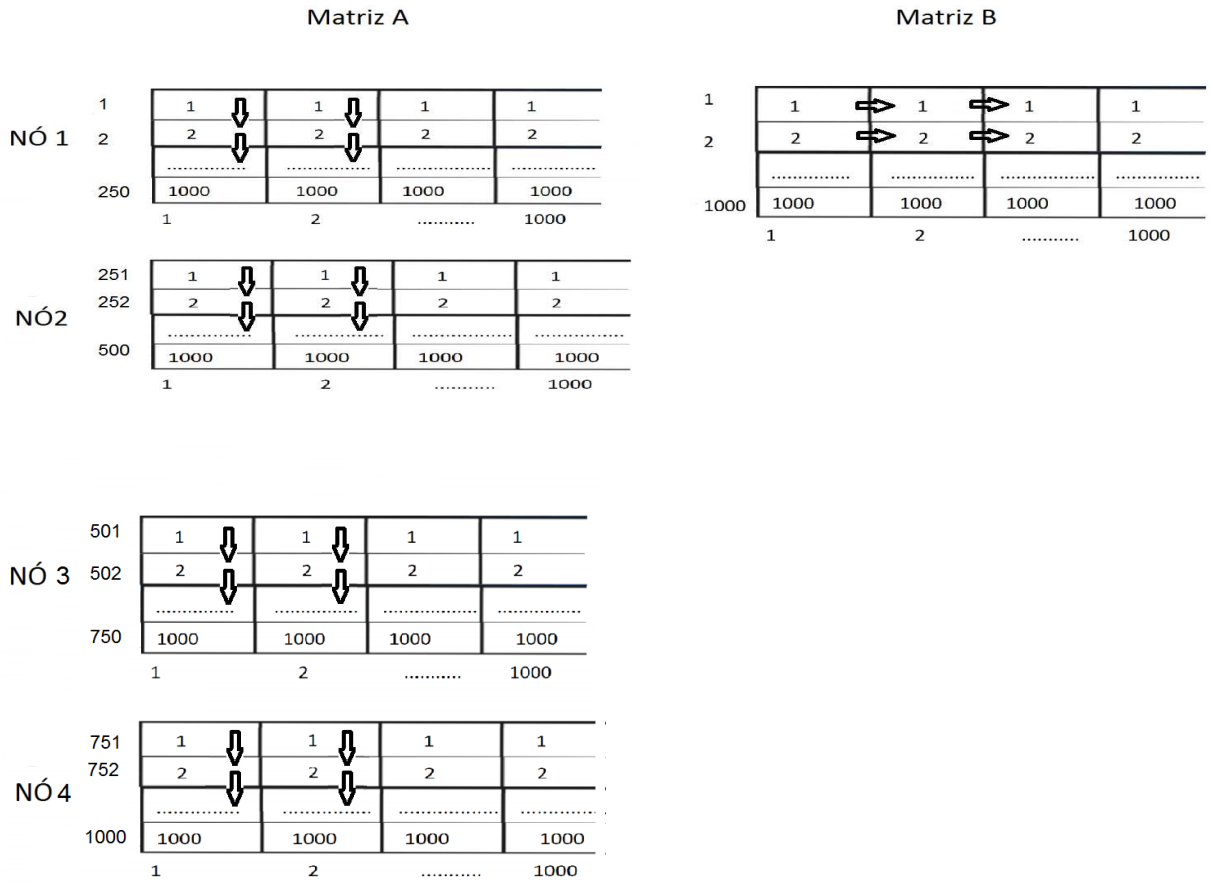

Figura 5.7: Produto de Matrizes

Neste momento, serão simulados os resultados e/ou tempos da aplicação de produto de matrizes, onde obtém-se o resultado de eficiência de tempo do código paralelo em relação ao código serial. Os tempos serão medidos desde o começo da leitura da base de dados das matrizes até seu término com a impressão do resultado obtido, levando-se em conta a execução do BLCR, ou seja, um restart dos processos a fim de se provar o

seu funcionamento e eficácia. O código será executado 6 vezes e será feita uma média aritmética dos tempos, incluindo o cálculo de desvios padrões(quanto de variação ou "dispersão"existe em relação à média).

Conforme gráfico da Figura 5.8 e a tabela 5.1, utilizando 4 processos, obtém-se um ganho de tempo acima de 400% de tempo utilizando-se processamento distribuído em relação ao serial. Pode-se observar também que o processamento serial não teve capacidade de processamento de cálculo ao tentar fazer o produto de matrizes quadradas acima de 1200X1200, enquanto o distribuído obteve o resultado esperado.

Matrizes	Tempo Serial	Tempo Distribuído	Desvio Padrão	Ganho de tempo
600X600	33,00 s	07,667 s	0,5164	471,4285%
800x800	115,00 s	17,833 s	0,4082	644,8598%
1000x1000	151,00 s	37,333 s	0,5164	404,46428%
1200x1200	382,00 s	64,167 s	0,4082	596,875 %
1400x1400	———-	102,167 s	0,4082	———
1600x1600	———-	153,33 s	0,5164	———
1800x1800	———-	218,337 s	0,4082	———
2000x2000	———-	300,50 s	0,5164	———

Tabela 5.1: Ganho de tempo entre o processamento serial e o paralelo

Como pode-se observar, para o processamento serial os produtos de matrizes acima de 1200X1200 a quantidade de cálculos excedeu a capacidade de memória do computador.

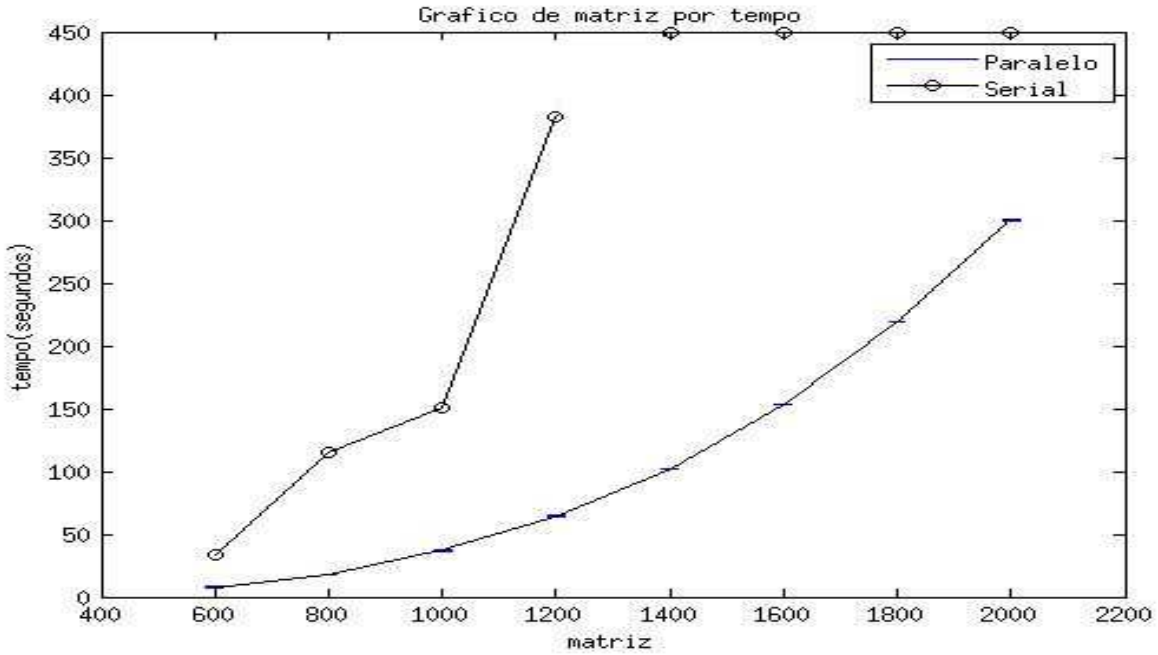

Figura 5.8: Tempo de execução do codigo serial x paralelo.

Para finalizar este primeiro caso, como esperado, observa-se que o gráfico Figura 5.8 comprova que os tempos do cálculos em paralelo/distribuído são menores que os seriais, trazendo assim grandes benefícios.

5.3 Produto de Matrizes Densas *Out-Of-Core*

Segundo Sivan Toledo [**?**], "Quando as estruturas de dados de algoritmos são grandes demais para caber na memória principal de um computador, as estruturas de dados devem ser armazenadas em discos rígidos. O acesso a dados que são armazenados nos discos é um processo mais lento. Para alcançar um desempenho aceitável , um algoritmo deve acessar os dados armazenados em discos de grandes blocos contíguos e reutilizar os dados que são armazenados na memória principal muitas vezes. Os algoritmos que são projetados para alcançar o alto desempenho em suas estruturas de dados que são armazenados em discos são chamados de *Out-Of-Core*"

Neste momento, inicia-se os procedimentos usando-se a técnica de *out-of-core* [**?**]. Será reproduzido o mesmo exemplo de produto de matrizes da seção 5.2 com a diferença que, no exemplo anterior, cada nó carregou sua parte da matriz A em memória, utilizando vetores de matrizes, seguido da leitura de um arquivo contendo a matriz B, enquanto que a partir de agora, existirá apenas a leitura de 2 arquivos nomeados como *matriza* e *matrizb*, sem vetores de matrizes para seu armazenamento.

O primeiro passo é ler o arquivo contendo a matriz A. De imediato, cada processo teria que percorrer o arquivo contendo a matriz A original e cada processo atuante criaria um arquivo, nomeado com o rank atuante mais a extensão *.txt*, escrevendo nesse arquivo as linhas da matriz original em que atuará. Entende-se área de atuação como ponto de inicio e fim, ou seja, uma matriz de 1000x1000 posições, contendo 4 processos, cada processo criaria um arquivo de 250x1000 posições, apagando a matriz original N logo depois, conforme Figura 5.9.

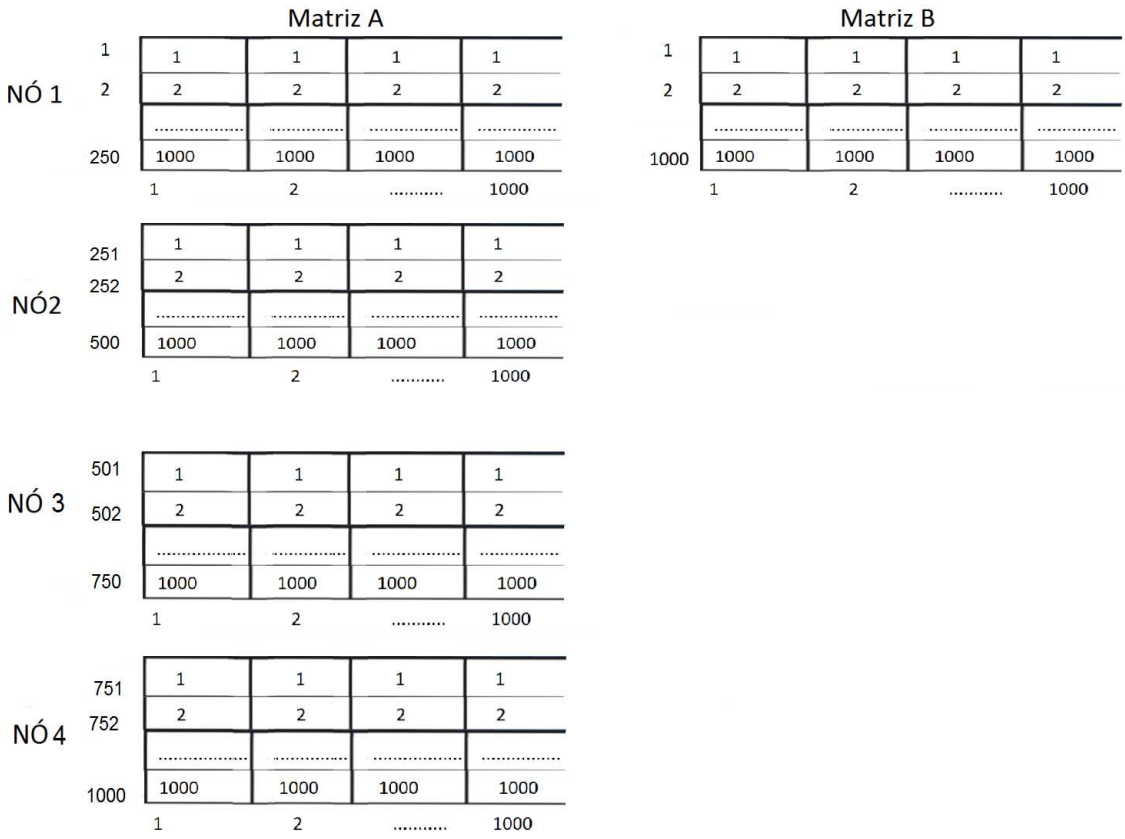

Figura 5.9: Produto de matrizes.

A partir desse momento, com os arquivos contendo a matriz A dividida e criados, deve-se ler o arquivo da matriz B e começar a multiplicação. Lembrando que cada processo lerá a matriz do arquivo B, ponto a ponto, onde uma variável auxiliar será a responsável por guardá-lo e fazer os cálculos necessários, conforme equação 5.4 mencionada na seção 5.2.

Dessa forma, a primeira linha das matrizes A dos processos devem ser multiplicadas por todas as colunas da matriz B, a segunda linha das matrizes A devem ser multiplicadas por todas as colunas da matriz B e assim por diante, onde ao final, cada processo criará um arquivo de texto com a sua respectiva parte do produto, alcançando-se o resultado esperado.

Porém, diferente do método anterior, com o intuito de agilizar o processo de leitura da matriz B, utiliza-se um artifício de transposição, pois um arquivo lido linha a linha é bem mais ágil que um arquivo lido coluna por coluna. Ao inicializar o processo de divisão da matriz A, um processo extra se encarregará de ler o arquivo com a matriz B e criar um outro arquivo com a mesma matriz B transposta.

Dessa forma, a leitura dos arquivos será muito mais ágil pois serão lidos por linha e

sequencialmente, ou seja, o primeiro elemento da primeira linha da matriz A será multiplicado pelo primeiro elemento da primeira linha da matriz B, o segundo elemento da primeira linha da matriz A sera multiplicado pelo segundo elemento da primeira linha da matriz B, etc. Quando se finaliza a leitura da primeira linha dos dois arquivos, a matriz A recomeça o primeiro elemento da linha atual novamente enquanto que a matriz B começa a segunda linha, etc. até que se finalize a leitura total da matriz B, para então sim, a matriz A comece a leitura da segunda linha pela matriz B e assim por diante, conforme Figura 5.10

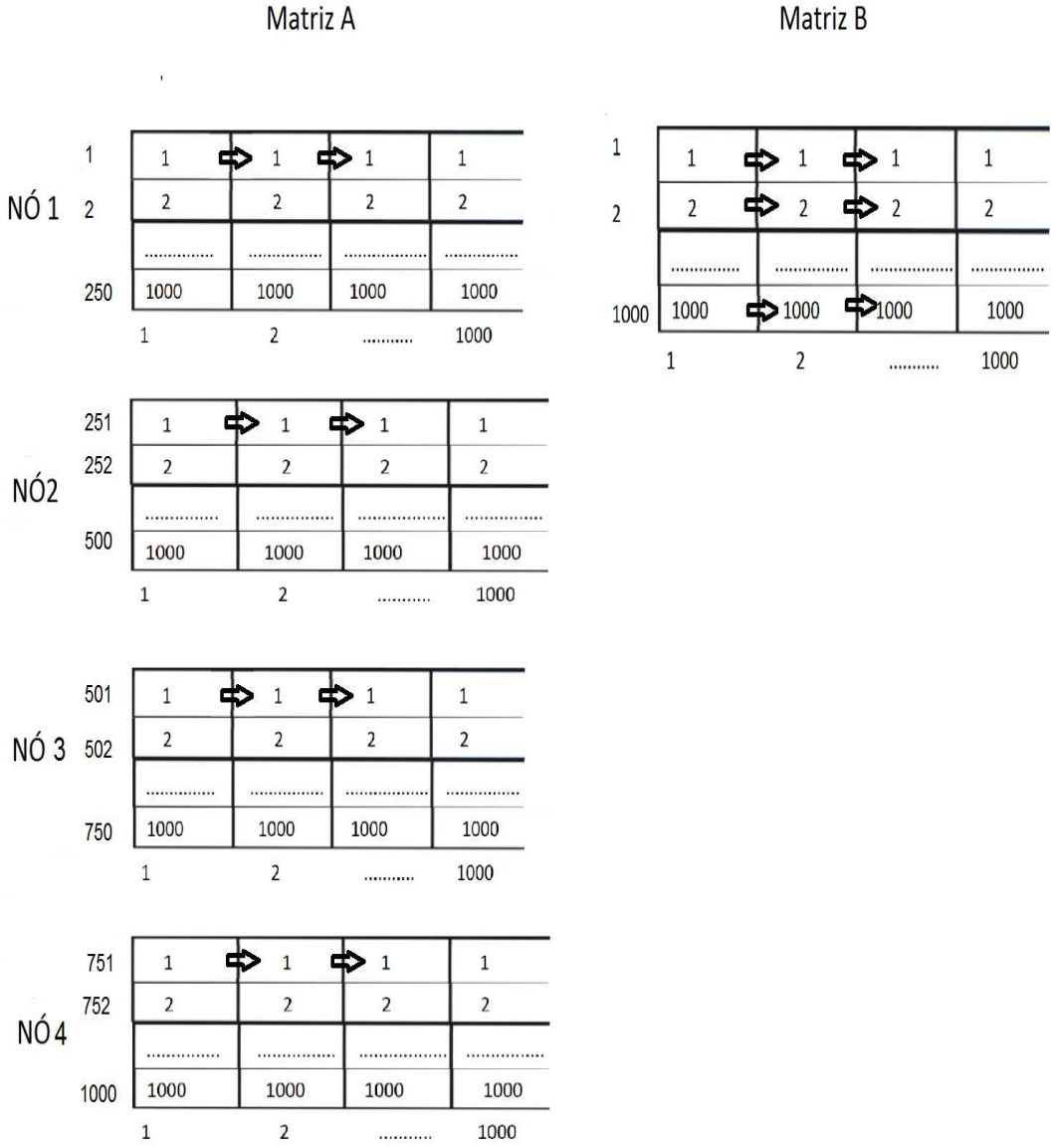

Figura 5.10: Produto de matrizes

Como no exemplo anterior, a partir de agora serão simulados os resultados e/ou

tempos da aplicação comparando os resultados de eficiência de tempo dos códigos paralelo e serial utilizando-se da técnica de *out-of-core*. Novamente os códigos serão executados 6 vezes e só então será feita uma média dos tempos, além de calcular também seus desvios padrão(quanto de variação ou "dispersão"existe em relação à média).

Matrizes	Tempo Serial	Tempo Distribuído	Desvio Padrão	Ganho de tempo
600X600	92,000 s	27,667 s	2,0816	332,530 %
800x800	217,000 s	62,833 s	0,897	345,358 %
1000x1000	425,000 s	137,333 s	1,607	309,466 %
1200x1200	735,000 s	213,167 s	1,154	344,800 %
1400x1400	1247,000 s	353,167 s	2,217	353,091 %
1600x1600	1759,000 s	501,333 s	0,5477	35,.864 %
1800x1800	2597,000 s	740,337 s	3,023	350,787 %
2000x2000	3436,000 s	996,500 s	1,802	344,806 %

Tabela 5.2: Ganho de tempo entre o processamento serial e o paralelo out-of-core

Através da tabela acima, pode-se observar que diferente do caso anterior, o processamento serial não excedeu a capacidade de memória do computador ao fazer o produto de matrizes quadradas acima de 1200X1200.

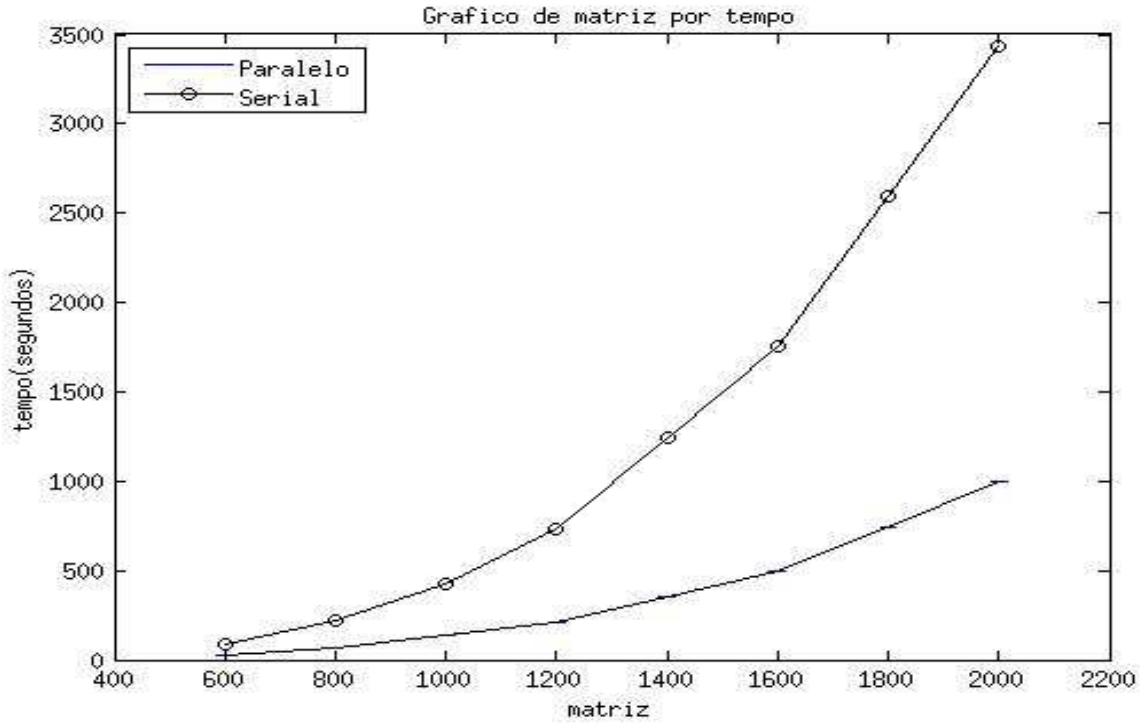

Figura 5.11: tempo de execução do codigo serial x paralelo em cada dimensão de matriz

Conforme gráfico da Figura 5.11, utilizando 4 processos, os tempos paralelos/distribuídos mantiveram-se abaixo do serial e como no exemplo anterior, obtém-se novamente

um ganho de tempo acima de 300% utilizando-se processamento distribuído em relação ao serial.

5.4 Distribuição de calor 2D *Out-Of-Core*

Neste último teste de caso, será aplicado a equação da propagação do calor ou equação de Laplace em duas dimensões usando a técnica de out-of-core segundo a equação 5.6

$$\frac{\partial T}{\partial t} + \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{5.6}$$

sendo x e y o espaço. Com condições de contorno do tipo Dirichlet nas fronteiras de T=1.

Serão utilizadas diferenças finitas para aproximar as derivadas de segunda ordem com diferenças centradas de segunda ordem para x e y.Pode-se escrever a equação discreta iterativa explicita da seguinte forma:

$$T_{ij}^{n+1} = \frac{1}{4}[T_{i-1,j}^{n} + T_{i+1,j}^{n} + T_{i,j-1}^{n} + T_{i,j+1}^{n}] \tag{5.7}$$

O domínio será inicialmente a matriz 200 x 200 contida em um arquivo de texto que será lido e executado através de 4 processos.

De imediato, cria-se para cada processo um arquivo de 200 posições onde cada processo é responsável pelo seu domínio. Entende-se domínio, como ponto de início e fim, ou seja, numa matriz de 200 X 200 , e tendo 4 processos, cada processo criaria seu arquivo de 100 X 100 posições e necessita apenas das fronteiras (início e fim) dos processos vizinhos para se trabalhar.

A primeira tarefa a se fazer, é determinar a matriz que será alocada no arquivo. No presente caso inicialmente será igual a 200 X 200 com condições de contorno globais igual 1 enquanto que o interior será 0. Veja o fragmento do código.

```
int main(int argc, char *argv[])
{
int n=200;
}
```

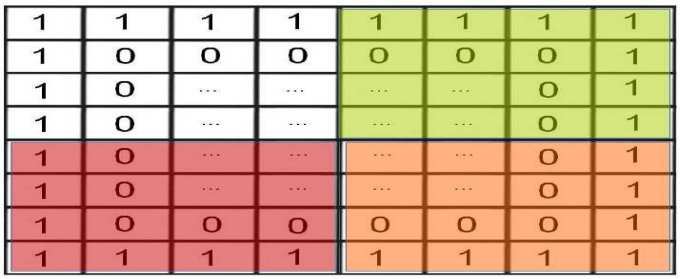

Figura 5.12: Matriz Original

Com a matriz definida, cada processo, utilizando de um laço de repetição de i e j, percorrerá o arquivo com a matriz original, linha a linha, lendo somente o quadrante que lhe corresponde, ou seja, cada processo lerá somente sua parte da matriz, conforme Figura 5.12;

- processo 0 quadrante noroeste; BRANCO
- processo 1 quadrante sudoeste; VERMELHO
- processo 2 quadrante nordeste; VERDE
- processo 3 quadrante sudeste; LARANJA

No próximo passo, cada processo criará apenas 2 arquivos contendo as bordas internas de cada quadrante, que será nomeada neste trabalho como condição de contorno local, conforme Figura 5.13.

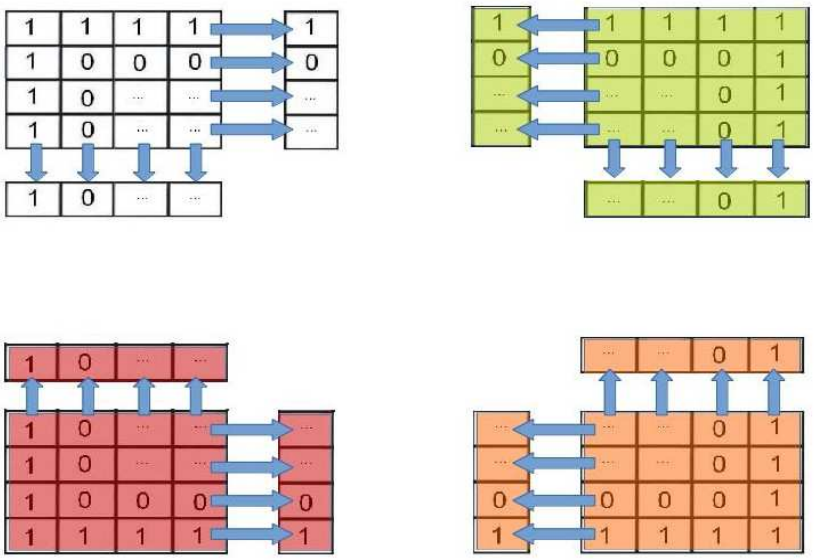

Figura 5.13: Arquivos gerados pelos processos

Conforme Figura 5.14, serão comutados entre os processos vizinhos os arquivos contendo as condições de contorno local, para que posteriormente, os processos vizinhos possam utilizá-las como condição de contorno. Veja fragmento de código a seguir:

```
 if(rank==0 && i<=l && j<=l)
 {
 fprintf(fv,"%lf ",x);
 if(i==(n/2) && j>1){fprintf(ext,"%lf ",x);}
 if(j==(n/2) && i>1){fprintf(ext2,"%lf \n",x);
 }
```

```
if(rank==1 && i>l && j<=l)
 {
 fprintf(fv,"%lf ",x);
 if(i==(n/2)+1 && j>1){fprintf(ext,"%lf ",x);}
 if(j==(n/2) && i<n)  {fprintf(ext2,"%lf \n",x);
 }
```

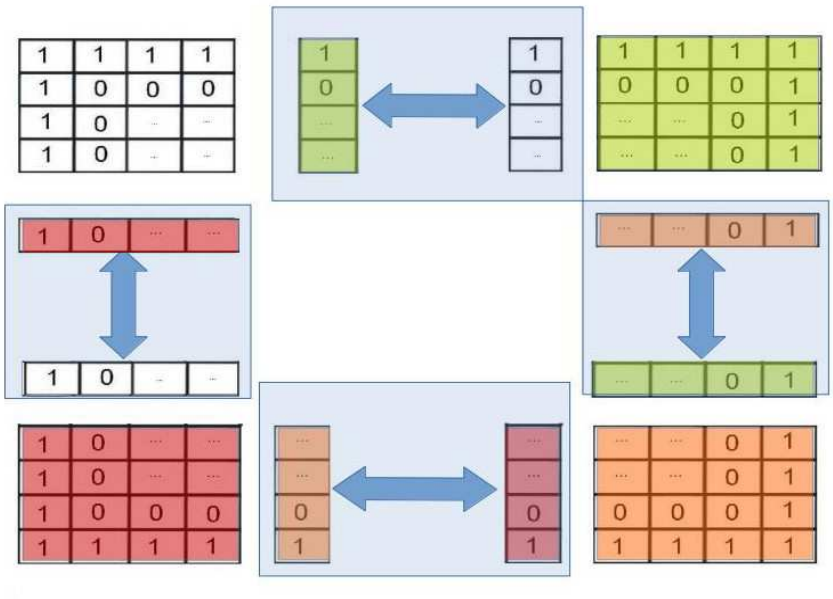

Figura 5.14: troca de arquivos contendo condições de contorno locais

Quando o processo se encerra, cada processo tem sua respectiva matriz A alocada em um arquivo e suas respectivas condições locais dos processos vizinhos alocadas em outros

arquivos, conforme Figura 5.15.

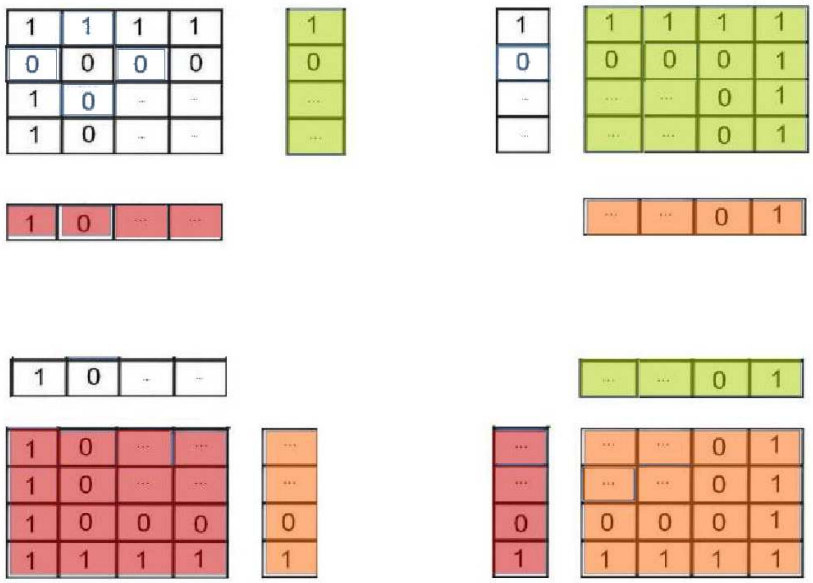

Figura 5.15: média dos pontos vizinhos no ponto analisado em cada processo

Agora deve-se adotar o número de iterações que serão realizadas no processo. Lembrando que quanto mais iterações, maior será a precisão dos resultados, pois trata-se de um processo iterativo onde se procura o resíduo mínimo estabelecido por uma tolerância estabelecida. Adota-se k como contador, e $maxk$ como variável de iteração, sendo seu valor igual a 3000, conforme fragmento do código.

```
int k,maxk = 3000;

for(k=1;k<=maxk;k++)
   {
   .   processo de cálculo do produto de matrizes out-of-core
   .
   }
```

Agora, cada processo percorrerá, linha a linha, cada ponto da matriz, que não seja condição de contorno global, em seus respectivos arquivos. A cada ponto analisado, cada processo fará uma diferença entre seus pontos vizinhos acima, abaixo, a esquerda e a direita, sendo que cada processo terá que recorrer ao arquivo contendo a condição de contorno local dos outros processos quando for necessário , conforme Figura 5.16, como

por exemplo, o processo 0 necessitaria do arquivo contendo a condição de contorno local dos processos 1 e 2. Veja fragmento de código em C exemplificando processo 0.

```
if(rank==0)
{
    x=ftell(fp);
    fseek(fp, -((l+1)*tamanho),SEEK_CUR);fscanf(fp,"%lf",&soma);
    fseek(fp,(l-2)*tamanho,SEEK_CUR);fscanf(fp,"%lf",&y);soma=soma+y;
    if(j==l){fscanf(ext,"%lf",&y);soma=soma+y; }else
        {fseek(fp,x,SEEK_SET);fscanf(fp,"%lf",&y);soma=soma+y; }

    if(i==l){fscanf(ext2,"%lf",&y);soma=((soma+y)/4);}else
        {fseek(fp((l-2)*tamanho),SEEK_CUR);fscanf(fp,"%lf",&y);soma=((soma+y)/4);}
    fseek(fp,x,SEEK_SET);
    .
    .
    }
```

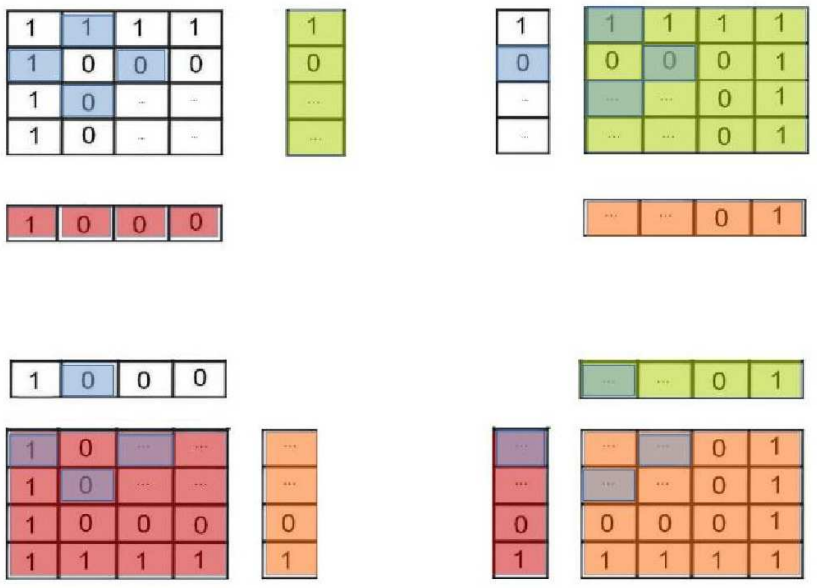

Figura 5.16: Arquivos gerados pelos processos

Ao final, cada processo terá feito as médias dos pontos, e sofre uma atualização, ou seja, aloca esses valores em outro arquivo, conforme Figura 5.17, além de criar os novos arquivos contendo os novos valores de condição de contorno local.

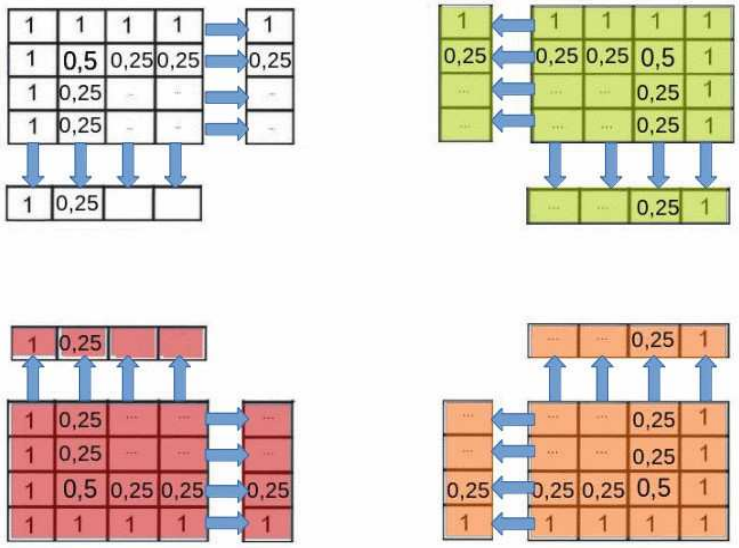

Figura 5.17: Arquivos gerados pelos processos

Após isso, renomeia os novos arquivos com o nomes dos antigos arquivos. Sem esse artificio, depois da primeira iteração, nada se alteraria, pois trata-se de um processo iterativo. Veja fragmento de código em C

```
if(rank==0){fclose(ext);fclose(ext2);fclose(new);fclose(new2);
rename("1.cimanew","1.cima");rename("2.esquerdanew","2.esquerda");}
```

Em cada cem iterações, um *checkpoint* do processo é feito, de maneira que em determinado momento foi simulado um erro, onde cada nó realizou um restart do último *checkpoint*, retornando o processamento, conforme Figura 5.18.

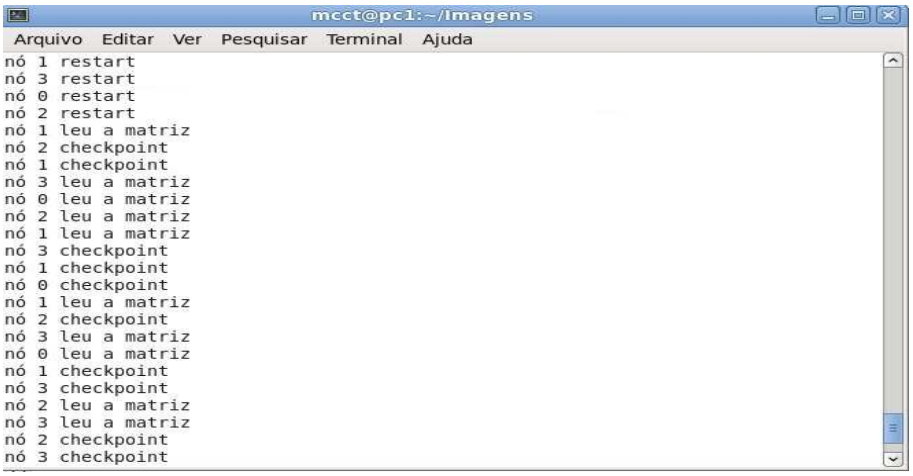

Figura 5.18: checkpoint sendo feito a cada 100 iterações

Ao final, cada processo gera um arquivo de texto com os resultados da parte que lhe corresponde a distribuição de calor em 2 dimensões. Ao plotar o resultado usando-se o software MATLAB, obtêm-se os gráficos da distribuição, conforme Figura 5.19.

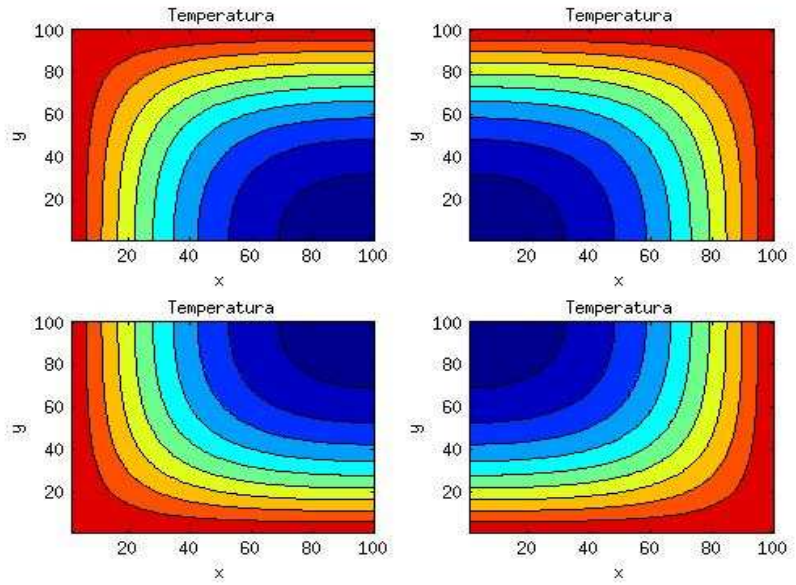

Figura 5.19: Distribuição de calor em cada sub-domínio

Pode-se observar a distribuição de calor em cada subdomínio obtido por cada processo, sendo observado que a linhas nas condições de contorno local dos subdomínios se conectam perfeitamente com as dos subdomínios vizinhos, sendo o resultado esperado, dadas as condições de contorno e o interior do domínio conforme figuras 5.20 e 5.21.

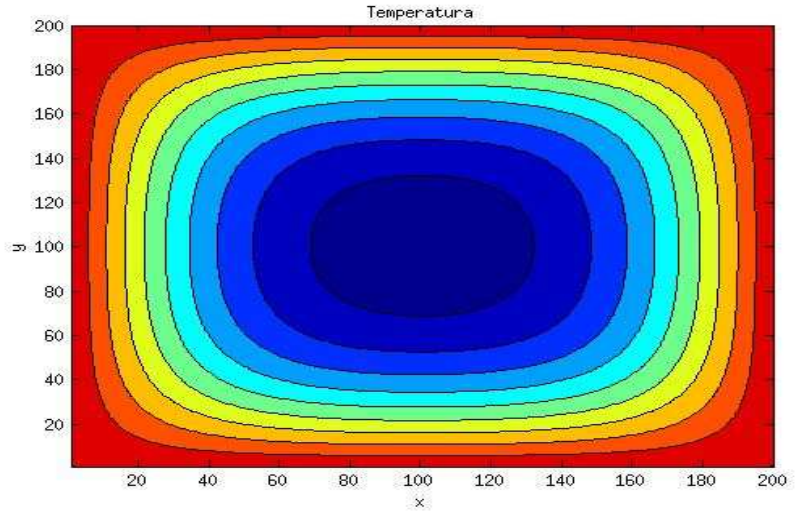

Figura 5.20: união dos 4 sub-domínios

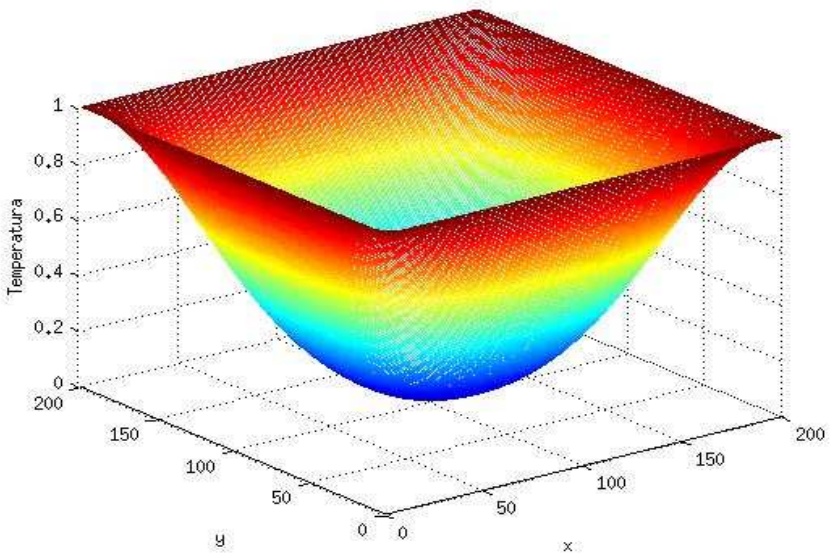

Figura 5.21: Distribuição de calor em visualização via 3D

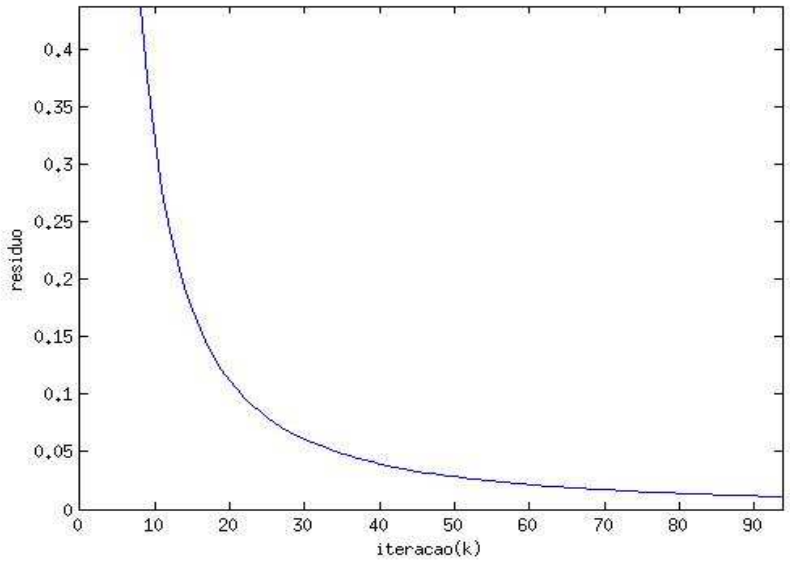

Figura 5.22: Residual da equação de Laplace 2D

Nota-se que o gráfico do resíduo tende para a convergência, com uma tolerância de 0,000001 representado na Figura 5.22. O resíduo foi calculado segundo a norma euclidiana dada por:

$$residuo(r) = \sqrt[2]{\sum_{i,j=1}^{N} (a_{ij}^{k} - aij^{k-1})^2} \tag{5.8}$$

onde k é a iteração atual.

Capítulo 6

Discussão e Conclusão

Este capitulo retrata as conclusões baseados nos testes previamente feitos.

6.1 Resultados

A construção e implementação de um cluster em nuvem usando o LAM/MPI e o BLCR foi concluída, sendo seguro sua utilização, ja que o *Restart* dos processos ocorreu perfeitamente, além de que, os códigos executados corresponderam aos resultados esperados.

1 - Conforme esperado, houve um ganho significativo de tempo , quando se utiliza um processamento paralelo/distribuído em relação ao serial, validando assim, a implementação.

2 - Foi observado que ao utilizar as técnicas de *OUT-OF-CORE*, tanto em processamentos seriais quanto paralelos/distribuídos, evita-se a falta de memória já que seu acesso torna-se recorrente;

3 - O BLCR combinado com as técnicas de *OUT-OF-CORE* e processamento paralelo/distribuído, torna-se uma ferramenta excelente, já que caso ocorram falhas, seu reinicio ocorre do ultimo *checkpoint* feito.

6.2 Trabalhos Futuros

Para trabalhos futuros, alguns pontos podem ser utilizados usando-se os conhecimentos adquiridos aqui, como:

1 - Aumentar os sistemas e ver o seu comportamento, ou seja, alocar arquivos contendo matrizes acima de Gbytes, condições de contorno diferentes das usadas aqui, etc.

2 - Com o recente crescimento de aparelhos moveis no mercado e com pouca memória interna disponível, estudar formas de instalar e utilizar-se da técnica de paralelização e de OUT-OF-CORE nos mesmos, a fim de tornar esses aparelhos utilizáveis como nós extras.

3 - Executar análises de problemas reais que exigem grande custo computacional.

Referências

[1] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(1):479–493, 2005.

[2] Andersown Becher Paes de Barros. Computacao em cluster. *Instituto Cuiabano de Educacao http://www.ice.edu.br/TNX/storage/webdisco/2011/03/11/outros/ffd949f568904419f485df9f969a2c27.pdf*, 09 2011.

[3] Jeffrey M. Squyres, Andrew Lumsdaine, William L. George, John G. Hagedorn, and Judith E. Devaney. The interoperable message passing interface (mpi) extensions to lam/mpi. In *Proceedings, MPIDC'2000*, March 2000.

[4] Amir Ashraf Kamil. Single program, multiple data programming for hierarchical computations. *University of California, Berkeley*, Fall 2012.

[5] LAM team. Lam/mpi parallel computing.

[6] LAM/Team. Lam/mpi parallel computing, 09 2013. www.lam-mpi.org - site acessado em 15/11/2013.

[7] LAM/Team. Lam faq: Lam terms and definitions, 09 2013. http://www.lam-mpi.org/faq/category1.php3 - site acessado 03/6/2013 as 12:40.

[8] Ragen Dazs. Entendendo um pouco sobre os daemons, 04 2006. http://www.vivaolinux.com.br/artigo/Entendendo-um-pouco-sobre-os-daemons - site acessado 25/10/2013 as 12:10.

[9] Centro Nacional de Processamento de Alto Desempenho e Universidade Estadual de Campinas. Introducao ao mpi. http://www.cenapad.unicamp.br/servicos/treinamentos/apostilas/ - site acessado em 30/01/2014.

[10] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003.

[11] Fedoraproject. Sobre o fedora, 09 2013. http://fedoraproject.org/en/about-fedora - site acessado as 01/3/2013 as 14:40 16:25.

[12] Herbert Schildt. *C Completo e Total*. Makron Books (grupo pearson), 1996.

[13] Carlos E. Morimoto. Tutorial do centos, 2008. http://www.hardware.com.br/tutoriais/centos/pagina3.html - site acessado 07/3/2013.

[14] LAM team. http://www.lam-mpi.org/7.1/download.php.

[15] Lawrence Berkeley National Laboratory. http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/berkeley-lab-checkpoint-restart-for-linux-blcr-downloads/. site acessado 06/06/2014 as 14:40em 05,2013.

[16] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms and Visualization*, 50:161–179, 1999.

APÊNDICE A - Anexo

A.0.1 Distribuição de calor 1D

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"
#include <libcr.h>

static int my_callback(void *data_ptr)
{
        int did_restart;

        /* realiza o checkpoint */

        did_restart = cr_checkpoint(0);

        if(did_restart) {
                printf("restart\n");
        } else {
                printf("checkpoint\n");
        }

        return 0;
}

int main(int argc, char *argv[])
{
  int i, j, k ,id,x=0,rank,size,n=500,tag=0,inicio,fim,nameSize;
      char computer_Name[MPI_MAX_PROCESSOR_NAME],string[4];
  int handle = 0,maxk = 500;
  FILE *pFile;
```

```
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Get_processor_name(computer_Name, &nameSize);


float uold[(n/size)+2],unew[(n/size)+2];
char* name;
  /*copia o nome pra dentro da nova variavel*/
name = computer_Name;

  /* armazena dentro do valor 0 do vetor o valor do rank,transformando de
       decimal para char */
sprintf(string,"%d", rank);

  /* concatena no nome do computador ,o valor do rank armazenado em
     STRING */
strcat(name,string);

  /*cria o arquivo com o nome do computador e o rank dele*/
pFile = fopen(name,"wt");



inicializa a libcr
      if (cr_init() < 0)
      {
          printf("cr_init() failed!\n");
          return 0;
      }

       //registra a funcao de callback
       id = cr_register_callback(my_callback, &handle, CR_THREAD_CONTEXT)
           ;
```

```
   inicio = 1; //inicio de cada processo
   fim = ((n/size)); // fim de cada processo

   for (i=(inicio-1);i<=(fim+1);i++) //zerar todos os pontos do vetor
   {
   uold[i] = 0.0;
         unew[i] = 0.0;
   }



   if(rank==0)
   {
   inicio = (inicio+1); //devemos evitar o contorno do primeiro rank
   }
   else
   if(rank==(size-1))
   {
   fim = (fim -1);//devemos evitar o contorno do ultimo rank
   }

for(k=1;k<=maxk;k++)  // numero de iteracoes
 {

      if(k % 1000 ==0)
   {
   cr_request();/* requisita um checkpoint */
               }


   if(rank == 0)
   {
   uold[1]=100.0;  //rank 0 tera o primeiro valor de 100 devido  a ser
       contorno
   unew[1]=100.0;  //rank 0 tera o primeiro valor de 100 devido  a ser
       contorno

      if(size>1){
   MPI_Send(&uold[fim],1, MPI_FLOAT, (rank + 1), tag, MPI_COMM_WORLD);  //
       envia para o proximo rank a ultima posicao do vetor

   MPI_Recv(&uold[fim+1], 1, MPI_FLOAT, (rank + 1), tag, MPI_COMM_WORLD, &
       status); //recebe do proximo rank a primeira posicao do vetor
```

```
    }
        else
      {
            uold[(n)]=100.0;  //rank (size -1) tera o ultimo valor de 100
                devido  a ser  contorno
      unew[(n)]=100.0;  //rank (size -1) tera o ultimo valor de 100 devido
          a ser  contorno
    }

    }
    else
    {
      if(rank==(size-1))
      {


        uold[(n/size)]=100.0;  //rank (size -1) tera o ultimo valor de 100
            devido  a ser status de contorno
        unew[(n/size)]=100.0;  //rank (size -1) tera o ultimo valor de 100
            devido  a ser status de contorno

        MPI_Send(&uold[inicio],1, MPI_FLOAT, (rank - 1), tag,
            MPI_COMM_WORLD);  //envia para o proximo rank a ultima posicao
            do vetor

        MPI_Recv(&uold[inicio -1], 1, MPI_FLOAT, (rank - 1), tag,
            MPI_COMM_WORLD, &status); //recebe do proximo rank a primeira
            posicao do vetor


      }
      else
      {
      MPI_Send(&uold[fim],1, MPI_FLOAT, (rank + 1), tag, MPI_COMM_WORLD);
          //envia para o proximo rank a ultima posicao do vetor

      MPI_Recv(&uold[fim+1], 1, MPI_FLOAT, (rank + 1), tag, MPI_COMM_WORLD,
          &status); //recebe do proximo rank a primeira posicao do vetor

      MPI_Send(&uold[inicio],1, MPI_FLOAT, (rank - 1), tag, MPI_COMM_WORLD)
          ;  //envia para o proximo rank a ultima posicao do vetor
```

```
      MPI_Recv(&uold[inicio -1], 1, MPI_FLOAT, (rank - 1), tag,
          MPI_COMM_WORLD, &status); //recebe do proximo rank a primeira
          posicao do vetor
      }

    }

if(size==1)
{
for(i=inicio;i<=fim-1;i++)
  {
    unew[i] = ((uold[i-1]+(uold[i+1]))/2);
  }


  for(i=inicio;i<=fim-1;i++)
  {
  uold[i]= unew[i];//   ! UPDATE
  }
}
else
{
  for(i=inicio;i<=fim;i++)
  {
    unew[i] = ((uold[i-1]+(uold[i+1]))/2);
  }


  for(i=inicio;i<=fim;i++)
  {
  uold[i]= unew[i];//   ! UPDATE
  }
}


  }
        for(i=1;i<=((n/size));i++)
  {
    fprintf(pFile, "%f \n",(unew[i]));
  }

fclose(pFile);
MPI_Finalize;
```

```
system("cat pc* > dados");
exit(0);
}
```

A.0.2 Produto de Matrizes Densas

```
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1
#include <string.h>
#include <unistd.h>
#include <libcr.h>
#include "mpi.h"

int main(int argc, char *argv[])
{

   int rank,size,cont,ini,final,ierr;
   char *name;
   char string[5];

         MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
       MPI_Comm_size(MPI_COMM_WORLD, &size);
       printf( "rank d inicio\n",rank);fflush(stdout);

        sprintf(string,"%d", rank);
  strcat(string,".txt");
  name = string;
   int n, k,i,j,v,numelem;
   n=1000;
   float vetor[n+1],matriz[n/(size-1)+1][n+1],x;
   FILE *fp,*fv;
   numelem=(n*n);
   i=1;
   j=1;
   ini=((numelem/(size-1))*(rank-1));
   final=((numelem/(size-1))*(rank));

if(rank!=0)
{
```

```
    fp = fopen( "matriza100", "r");
    if (fp == NULL)
    {

        exit( EXIT_FAILURE);
    }
    for(cont=0;cont<numelem;cont++) //responsavel pela contagem de elementos
    {
        k = fscanf( fp, "f ", &matriz[i][j]);

   if(cont>=ini && cont<=final)
      {
         if (j<n)
         {

             j++;   //anda pelas colunas
         }
         else
         {
       i++;  //muda de linha
             j=1;
         }
         if (k != 1) break;
      }
    }

  printf( "leu a matriz\n");fflush(stdout);
for(cont=1;cont<=(n/(size-1));cont++)
{
    if(rank!=0)
    {
    fp = fopen( "matrizb100", "r");
    fv = fopen( name, "a");
    i=1;
    j=1;

    while(TRUE)
    {
      k = fscanf(fp, "%f ", &x);
      //else
      {
      if(i<=1)
      {
```

```
      vetor[j]=x * matriz[cont][i];
      }
      else
      {
      vetor[j]=vetor[j] + (x * matriz[cont][i]);
      }
      if(i==n )
      {
      fprintf(fv, "f ",vetor[j]);
      //fprintf(fv,"d",cont);
      }
      }//if (k != 1) break;
      if(j<n)
         {
             j++;
         }
         else
         {
       i++;
             j=1;
         }
         if (k != 1) break;
   }

   fprintf(fv,"\n");
   fclose(fp);
   fclose( fv);
   }
}
}
       MPI_Finalize();
       return EXIT_SUCCESS;
}
```

A.0.3 Produto de Matrizes Densas OUT-OF-CORE

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1      //definido esse modo para fazer o arquivo ser lido ate o
    final
#include <string.h>
```

```
#include <unistd.h>
#include <libcr.h>
#include "mpi.h"
#include <sys/time.h>




int main(int argc, char *argv[])
{
   int rank,size,cont,ini,final,ierr,linha,coluna,cont2;
   char *name,*name2;
   char string[10],string2[10];
   struct timeval Inicio,Fim;

         MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
       MPI_Comm_size(MPI_COMM_WORLD, &size);

         sprintf(string,"%d",rank);
         sprintf(string2,"%d",rank);
  strcat(string,".txt");
         strcat(string2,".dat");
  name = string;
         name2= string2;


   int k,k2,i,j,l,numelem,n,o,p,q;
   n=1000;
   double x,tamanho,y,soma,z,sec,temp,residuo,teste,res;

   FILE *fp,*fv,*fb,*fx;

   ini=(n/size) * (rank) ;
   final=(n/size) * (rank+1) ;



fp = fopen( "matriza", "r");

fv = fopen(name,"w");
```

```
for(p=1;p<=n;p++)
{
       for(q=1;q<=n;q++)
       {
                fscanf(fp,"%lf ", &y);


      if(p>ini && p<=final)
      {
         fprintf(fv,"%lf ",y);
      }
        }fprintf(fv,"\n");
}
fclose(fp);
fclose(fv);


if(rank ==0)
{ fb = fopen("matrizb","r");fx = fopen("inv","w");
      for(l=1;l<=n;l++)
       {
       for(i=1;i<=n;i++)
       {

           fscanf(fb, "%lf",&x);
           if(i==1){o=ftell(fb);}


           fprintf(fx,"%lf ",x);

           for (j=1;j<n;j++ )
           {
           fscanf(fb, "%lf",&x);
   }

       } fprintf(fx,"\n"); fseek(fb,o,SEEK_SET);
       }
fclose(fx);
}

 MPI_Barrier(MPI_COMM_WORLD);
```

```
gettimeofday(&Inicio, NULL);
fp = fopen( name, "r");
fb = fopen("inv","r");
fv = fopen(name2,"w");
//fx = fopen(name2,"w");

for(i=1;i<=(n/size);i++)
{printf("%d",i);
    for(l=1;l<=n;l++)
    {soma=0;
        if(l==1){k=ftell(fp);} else{fseek(fp,k,SEEK_SET);}

  for(j=1;j<=n;j++)
  {
    fscanf(fp,"%lf ", &y);
                fscanf(fb, "%lf",&x);
                soma = soma + (x*y);

  }fprintf(fv,"%lf ",soma);


    }fprintf(fv,"\n ");
}



 gettimeofday(&Fim, NULL);
sec =Fim.tv_sec - Inicio.tv_sec;
if(rank==0){printf("    %lf      \n ",sec);}
fclose(fp);
fclose(fb);
 MPI_Finalize();
 return EXIT_SUCCESS;
}
```

A.0.4 Distribuição de calor 2D OUT-OF-CORE

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1      //definido esse modo para fazer o arquivo ser lido ate o
    final
```

```
#include <string.h>
#include <unistd.h>
#include <libcr.h>
#include "mpi.h"
#include <sys/time.h>

static int my_callback(void *data_ptr)
{
        int did_restart;

        // realiza o checkpoint
        did_restart = cr_checkpoint(0);

        if(did_restart) {
                printf("restart\n");
        } else {
                printf("checkpoint\n");
        }

        return 0;
}


int main(int argc, char *argv[])
{
   int rank,size,cont,ini,final,ierr,linha,coluna,cont2;
   char *name,*name2;
   char string[10],string2[10];
   struct timeval Inicio,Fim;

         MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
       MPI_Comm_size(MPI_COMM_WORLD, &size);

        sprintf(string,"%d", rank); /*armazena dentro da posicao 0 de
           STRING O valor do rank,transformando de decimal para char */
        sprintf(string2,"%d", rank); /*armazena dentro da posicao 0 de
           STRING O valor do rank,transformando de decimal para char */
  strcat(string,".txt");//concatena no nome do computador ,o valor do rank
     armazenado em STRING
        strcat(string2,".dat");//concatena no nome do computador ,o valor
           do rank armazenado em STRING
```

```
  name = string;
        name2= string2;


   int n,k,i,j,l,iter,kmax=3000,handle=0,id;
   n=200;
   double x,tamanho,y,soma,z,sec,L,W,dt,temp,residuo,teste,res;

   FILE *fp,*fv,*ext,*ext2,*new,*new2,*fresiduo;

  if(size==4){l=(n-(n/2));}else{l=n;} // cada processo tera uma matriz de
     500X500. o calculo e feito para matrizes impares tambem,como matrizes
     1001X1001

   // inicializa a libcr
       if (cr_init() < 0) {
                printf("cr_init() failed!\n");
                return 0;
        }

        // registra a funcao de callback
        id = cr_register_callback(my_callback, &handle, CR_THREAD_CONTEXT);

//for(iter=0;iter<=kmax;iter++)
{



{
  fp = fopen("matrizb", "r");// abre o arquivo contendo a matriz original
  fv = fopen(name,"w"); // cada processo abre seu arquivo respectivo para
     receber sua parte da matriz original; gerando o 0.txt, 1.txt, 2.txt e
     3.txt.
  if(rank==0 && size==4){ext = fopen("1.cima","w");ext2 = fopen("2.esquerda
     ","w");}
  if(rank==1){ext = fopen("0.baixo","w");ext2 = fopen("3.esquerda","w");}
  if(rank==2){ext = fopen("3.cima","w");ext2 = fopen("0.direita","w");}
  if(rank==3){ext = fopen("2.baixo","w");ext2 = fopen("1.direita","w");}


  for (i=1;i<=n;i++ )
  {
      for (j=1;j<=n;j++ )
```

```
    {
      k = fscanf(fp, "%lf",&x);  // fscanf e o responsavel por ler o
          arquivo da matriz original como um arquivo de inteiros
            if(rank==0 && i<=l && j<=l && size==4)
            {
   fprintf(fv,"%lf ",x); //imprime no arquivo 0.txt sua respectiva
       parte da matriz
   if(i==(n/2) && j>1){fprintf(ext,"%lf ",x);} //gera o arquivo com a
       condicao de contorno de baixo
            if(j==(n/2) && i>1){fprintf(ext2,"%lf \n",x);} //gera o
                arquivo com a condicao de contorno da direita
   }
            if(rank==0 && size==1)
        {
   fprintf(fv,"%lf ",x); //imprime no arquivo 0.txt sua respectiva
       parte da matriz
    }


       if(rank==1 && i>l && j<=l && size==4)
       {
          fprintf(fv,"%lf ",x);  //imprime no arquivo 1.txt sua
              respectiva parte da matriz
                          if(i==(n/2)+1 && j>1){fprintf(ext,"%lf ",x)
                              ;} //gera o arquivo com a condicao de
                              contorno de cima
                          if(j==(n/2) && i<n)  {fprintf(ext2,"%lf \n"
                              ,x);} //gera o arquivo com a condicao de
                               contorno da direita


       }
       if(rank==2 && j>l && i<=l && size==4)
       {
           fprintf(fv,"%lf ",x);   //imprime no arquivo 2.txt sua
               respectiva parte da matriz
                             if(i==(n/2) && j<n){fprintf(ext,"%lf ",x)
                                 ;} //gera o arquivo com a condicao de
                                 contorno de baixo
                             if(j==(n/2)+1 && i>1){fprintf(ext2,"%lf \
                                 n",x);} //gera o arquivo com a
                                 condicao de contorno da esquerda


       }
```

```
            if(rank==3 && j>1 && i>1 && size==4)
            {
                  fprintf(fv,"%lf ",x); //imprime no arquivo 3.txt sua
                      respectiva parte da matriz
                  if(i==(n/2)+1 && j<n){fprintf(ext,"%lf ",x);} //gera o
                      arquivo com a condicao de contorno de cima
                                        if(j==(n/2)+1 && i<n){fprintf(ext2,
                                            "%lf \n",x);} //gera o arquivo
                                            com a condicao de contorno da
                                            esquerda

            }
       }fprintf(fv,"\n");
  }
  fclose(fp);  //fecha o arquivo da matriz original
  fclose(fv);  //cada processo fecha seu arquivo respectivo contendo sua
      parte da matriz
         if(rank==0 && size==4){fclose(ext);fclose(ext2);}
         if(rank==1){fclose(ext);fclose(ext2);}
         if(rank==2){fclose(ext);fclose(ext2);}
         if(rank==3){fclose(ext);fclose(ext2);}
         MPI_Barrier(MPI_COMM_WORLD);//Espera todos os processos terem feita
              a leitura das condicoes de contorno e gerados os respectivos
             arquivos
 gettimeofday(&Inicio, NULL);
}


fresiduo = fopen("residuo.txt","w"); // cada processo abre seu arquivo
    respectivo para receber sua parte da matriz original; gerando o 0.txt,
    1.txt, 2.txt e 3.txt.



for(iter=1;iter<=kmax;iter++)//else
{
if(iter % 1000 ==0)
    {
    cr_request();/* requisita um checkpoint */
                }
```

```
        if(rank==0)

        if(iter % 1000 ==0){printf(""); }
        MPI_Barrier(MPI_COMM_WORLD);//Espera todos os processos terem
            feita a leitura das condicoes de contorno e gerados os
            respectivos arquivos

        fp = fopen( name, "r");  // cada processo abre seu arquivo
            respectivo contendo sua parte da matriz original;   0.txt, 1.
            txt, 2.txt e 3.txt.
        fv = fopen(name2, "w");
        if(rank==0 && size==4){ext = fopen( "0.direita", "r");rewind(ext);
            ext2 = fopen( "0.baixo", "r");rewind(ext2);new = fopen("1.
            cimanew","w");new2 = fopen("2.esquerdanew","w");}  //processo 0
             abre os arquivos com as condicoes de contorno de baixo e
            direita que ele necessita
        if(rank==1 && size==4){ext = fopen( "1.direita", "r");ext2 = fopen
            ( "1.cima", "r");new = fopen("0.baixonew","w");new2 = fopen("3.
            esquerdanew","w");}   //processo 1 abre os arquivos com as
            condicoes de contorno da cima e da direita que ele necessita
       if(rank==2 && size==4){ext = fopen( "2.esquerda", "r");ext2 = fopen
           ( "2.baixo", "r");new = fopen("3.cimanew","w");new2 = fopen("0.
           direitanew","w");}  //processo 2 abre os arquivos com as
           condicoes de contorno da baixo e esquerda que ele necessita
       if(rank==3 && size==4){ext = fopen( "3.esquerda", "r");ext2 = fopen
           ( "3.cima", "r");new = fopen("2.baixonew","w");new2 = fopen("1.
           direitanew","w");}   //processo 3 abre os arquivos com as
           condicoes de contorno da cima e da esquerda que ele necessita
tamanho = (sizeof(x)+1);

        res = 0;
 for (i=1;i<=l;i++ )
   {
         for (j=1;j<=l;j++ )
         {
           fscanf(fp,"%lf",&y);
             temp = y;
        if(rank==0 && i<=l && j<=l &&(i==l || j==1 && size==1) ){fprintf
           (fv,"%lf ",y);}else//imprime a condicao de contorno global da
            esquerda e ultima linha
                if(rank==0 && i<=l && j<=l &&(i==1 || j==l && size==1) )
                   {fprintf(fv,"%lf ",y);}else //imprime a condicao de
                   contorno global
```

```
            if(rank==0 && i<=l && j<=l &&(i==l || j==l && size==1) )
                {fprintf(fv,"%lf ",y);}else //imprime a condicao de
                contorno global da direita


        if(rank==0 && i<=l && j<=l &&(i==1 || j==1) ){fprintf(fv,"%lf
            ",y);}  //imprime a condicao de contorno global
                    else{
    if(rank==0)
    {
                      x=ftell(fp);//guarda a posicao em Bytes da
                          posicao do ponteiro
                      fseek(fp, -((l+1)*tamanho),SEEK_CUR);fscanf(fp
                          ,"%lf",&soma);// posiciona o ponteiro para a
                           leitura do vizinho de cima
           fseek(fp,(l-2)*tamanho,SEEK_CUR);fscanf(fp,"%lf",&y);soma=
               soma+y; //posiciona para a leitura do vizinho da
               esquerda
           if(j==l && size==4){fscanf(ext,"%lf",&y);soma=soma+y; }else
                //posiciona para a leitura do arquivo do vizinho da
               direita que e a condicao de contorno
            {fseek(fp,x,SEEK_SET);fscanf(fp,"%lf",&y);soma=soma+y; }
                //posiciona para a leitura do vizinho da direita
                      if(i==l && size==4){fscanf(ext2,"%lf",&y);soma
                          =((soma+y)/4);}else //posiciona para a
                          leitura do arquivo do vizinho de baixo que e
                           a condicao de contorno
     {fseek(fp, ((l-2)*tamanho),SEEK_CUR);fscanf(fp,"%lf",&y);soma=((
         soma+y)/4); } //posiciona para a leitura do vizinho de baixo
     fseek(fp,x,SEEK_SET); //retorna o ponteiro para a posicao
         original
                        fprintf(fv,"%lf ",soma);  // imprime o
                            resultado
    res = res + pow(soma - temp,2);


                      if(i==l && j>1 && size==4){fprintf(new," %lf ",
                          soma);} //gera-se os novos arquivos com a
                          condicao de contorno de baixo
                      if(j==l && i>1 && size==4){fprintf(new2,"%lf \n"
                          ,soma);} //gera-se os novos arquivos com a
                          condicao de contorno da direita


                    res = pow(res,2);
```

```
        }
                    }


if(rank==1 && i<=l && j<=l &&(i==l || j==1 && size==4) ){fprintf(fv
    ,"%lf ",y);}//imprime a condicao de contorno global
else{
  if(rank==1 && size==4)
  {
                    x=ftell(fp); //guarda a posicao em Bytes da
                        posicao do ponteiro
                    if(i==1){fscanf(ext2,"%lf",&soma);}else//
                        posiciona para a leitura do arquivo do
                        vizinho de cima que e a condicao de contorno
                    {fseek(fp,x - ((l+1)*tamanho),SEEK_SET);fscanf(
                        fp,"%lf",&soma);}// posiciona o ponteiro
                        para a leitura do vizinho de cima
  fseek(fp,(l-2)*tamanho,SEEK_CUR);fscanf(fp,"%lf",&y);soma=soma+y;
      //posiciona para a leitura do vizinho da esquerda
  if(j==l){fscanf(ext,"%lf",&y);soma=soma+y;  }else //posiciona
     para a leitura do arquivo do vizinho da direita que e a
     condicao de contorno
    {fseek(fp,1*tamanho,SEEK_CUR);fscanf(fp,"%lf",&y);soma=soma+y;
        } //posiciona para a leitura do vizinho da direita
   {fseek(fp, ((l-2)*tamanho),SEEK_CUR);fscanf(fp,"%lf",&y);soma=(
      soma+y)/4;} //posiciona para a leitura do vizinho de baixo
   fseek(fp,x,SEEK_SET); //retorna o ponteiro para a posicao
      original

                    fprintf(fv,"%lf ",soma);// imprime o resultado
                      if(i==1){fprintf(new," %lf ",soma);}  //gera
                          -se os novos arquivos com a condicao de
                          contorno de cima
                      if(j==l && i<l){fprintf(new2,"%lf \n",soma)
                          ;} //gera-se os novos arquivos com a
                          condicao de contorno da direita

        }}


if(rank==2 && i<=l && j<=l &&(i==1 || j==l && size==4) ){fprintf(fv
    ,"%lf ",y);}else //imprime a condicao de contorno global
        {
```

```
if(rank==2 && size==4)
{
                    x=ftell(fp);rewind(fp);
                    fseek(fp, x- ((l+1)*tamanho),SEEK_CUR);fscanf(
                        fp,"%lf",&soma); // cima

if(j==1){fscanf(ext,"%lf",&y);soma=soma+y;  } //esquerda
else{fseek(fp,(l-2)*tamanho,SEEK_CUR);fscanf(fp,"%lf",&y);soma=
    soma+y;}

      {fseek(fp,x,SEEK_SET);fscanf(fp,"%lf",&y);soma=soma+y;  }
          //direita

                     if(i==l){fscanf(ext2,"%lf",&y);soma=(soma+y)
                         /4;}else
{fseek(fp, ((l-2)*tamanho),SEEK_CUR);fscanf(fp,"%lf",&y);soma=(
    soma+y)/4;} //baixo

 fseek(fp,x,SEEK_SET); //retorna o ponteiro para a posicao
     original
                   fprintf(fv,"%lf ",soma);

if(i==l && j<l){fprintf(new," %lf ",soma);}
                    if(j==1 && i>1){fprintf(new2,"%lf \n",soma);}
    }}



               if(rank==3 && i<=l && j<=l &&(i==l || j==l && size
                   ==4) ){fprintf(fv,"%lf ",y);}else
    {
if(rank==3 && size==4)
{
                    x=ftell(fp);
                    if(i==1){fscanf(ext2,"%lf",&soma);}else
                    {fseek(fp,x - ((l+1)*tamanho),SEEK_SET);fscanf(
                        fp,"%lf",&soma);}//fprintf(fv,"%lf ",x);
                    if(j==1){fscanf(ext,"%lf",&y);soma=soma+y;  }
                        else
{fseek(fp,(l-2)*tamanho,SEEK_CUR);fscanf(fp,"%lf",&y);soma=soma+y
    ;} //-2 porque ele se posiciona 2 atras e quando le,le o
    anterior
```

```
            {fseek(fp,x,SEEK_SET);fscanf(fp,"%lf",&y);soma=soma+y;   }

            {fseek(fp,  ((l-2)*tamanho),SEEK_CUR);fscanf(fp,"%lf",&y);soma=(
                soma+y)/4;}
            fseek(fp,x,SEEK_SET);

                                fprintf(fv,"%lf ",soma);

             if(i==1 && j<l){fprintf(new," %lf ",soma);}
                                if(j==1 && i<l){fprintf(new2,"%lf \n",soma);}

               }}







      }if(iter==kmax){fprintf(fv," \n");}
    }

residuo = (res/pow(l,2)); //raiz quadrada;

if(rank==0){fprintf(fresiduo,"%lf \n",res);}




//Espera todos os processos terem feita a leitura das condicoes de contorno
    e gerados os respectivos arquivos
if(rank ==1 && size==4){MPI_Send(&x,1,  MPI_DOUBLE,0,0,MPI_COMM_WORLD);
   MPI_Recv(&x,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD, &status);}
if(rank ==2 && size==4){MPI_Send(&x,1,  MPI_DOUBLE,0,0,MPI_COMM_WORLD);
   MPI_Recv(&x,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD, &status);}
if(rank ==3 && size==4){MPI_Send(&x,1,  MPI_DOUBLE,0,0,MPI_COMM_WORLD);
   MPI_Recv(&x,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD, &status);}


if(rank==0 && size==4){
MPI_Recv(&x,1,MPI_DOUBLE,1,0,MPI_COMM_WORLD, &status);MPI_Send(&x,1,
   MPI_DOUBLE,1,0,MPI_COMM_WORLD);
MPI_Recv(&x,1,MPI_DOUBLE,2,0,MPI_COMM_WORLD, &status);MPI_Send(&x,1,
   MPI_DOUBLE,2,0,MPI_COMM_WORLD);
```

```
MPI_Recv(&x,1,MPI_DOUBLE,3,0,MPI_COMM_WORLD, &status);MPI_Send(&x,1,
    MPI_DOUBLE,3,0,MPI_COMM_WORLD);
}

fclose(fp);
fclose(fv);
if(rank==0 && size==4){fclose(ext);fclose(ext2);fclose(new);fclose(new2);/*
    remove("1.cima");remove("2.esquerda");*/rename("1.cimanew","1.cima");
    rename("2.esquerdanew","2.esquerda");} // (atualizacao)
if(rank==1 && size==4){fclose(ext);fclose(ext2);fclose(new);fclose(new2);
    remove("0.baixo");remove("3.esquerda");rename("0.baixonew","0.baixo");
    rename("3.esquerdanew","3.esquerda");}// (atualizacao)
if(rank==2 && size==4){fclose(ext);fclose(ext2);fclose(new);fclose(new2);/*
    remove("3.cima");remove("0.direita");*/rename("3.cimanew","3.cima");
    rename("0.direitanew","0.direita");}// (atualizacao)
if(rank==3 && size==4){fclose(ext);fclose(ext2);fclose(new);fclose(new2);/*
    remove("1.direita");remove("2.baixo");*/rename("1.direitanew","1.direita
    ");rename("2.baixonew","2.baixo");}// (atualizacao)
rename(name2,name);// (atualizacao)



gettimeofday(&Fim, NULL);
sec =Fim.tv_sec - Inicio.tv_sec;

}


}
 fclose(fresiduo);
 MPI_Finalize();
 return EXIT_SUCCESS;
}
```