UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação - ICMC-USP

Fabio Alves Martins Pereira (NUSP 7987435)

Naylor Garcia Bachiega (NUSP 5567669)

Relatório 2: algoritmo sequencial e paralelo para processamento de imagens com Smooth

SÃO CARLOS

2015

SUMÁRIO

[Lista de Figuras 4](#_Toc433395817)

[Lista de Tabelas 5](#_Toc433395818)

[1 Introdução 6](#_Toc433395819)

[1.2 Objetivos 7](#_Toc433395820)

[2 O Formato PPM 8](#_Toc433395821)

[2.1 O Arquivo PPM 8](#_Toc433395822)

[3 Paralelismo 10](#_Toc433395823)

[*3.1* *OpenMPI* 10](#_Toc433395824)

[3.1.1 Funcionamento 10](#_Toc433395825)

[*3.2* *OpenMP* 12](#_Toc433395826)

[3.2.1 Funcionamento 12](#_Toc433395827)

[3.3 Técnicas de Decomposição 13](#_Toc433395828)

[4 Desenvolvimento e Metodologia 15](#_Toc433395829)

[4.1 Repositório do código 15](#_Toc433395830)

[4.2 Hardware 15](#_Toc433395831)

[4.1 Programa: PPMsequencial 15](#_Toc433395832)

[4.2 Programa: PPMparalelo 16](#_Toc433395833)

[4.3 README.md 17](#_Toc433395834)

[4.4 Makefile 19](#_Toc433395835)

[4.5 Decomposição do Problema 20](#_Toc433395836)

[5 Resultados e Discussões 23](#_Toc433395837)

[5.1 Algoritmo Sequencial 23](#_Toc433395838)

[5.1 Algoritmo Paralelo: Pthreads 24](#_Toc433395839)

[5.1.1 Processamento das Matrizes 25](#_Toc433395840)

[5.2 Algoritmo Paralelo: *OpenMP* 28](#_Toc433395841)

[5.2.1 Processamento das Matrizes 30](#_Toc433395842)

[5.3 SpeedUp 33](#_Toc433395843)

[5.4 Considerações Finais 34](#_Toc433395844)

[Referências 35](#_Toc433395845)

[Anexo A – Execuções sequenciais 38](#_Toc433395846)

[Anexo B – Execuções paralelas: *Pthreads* 40](#_Toc433395847)

[Anexo C – Execuções paralelas: *OpenMP* 43](#_Toc433395848)

# Lista de Figuras

Figura 1 – Benchmark em um único computador (LAMMPS, 2015). 7

Figura 2 – Exemplo de imagem PPM (NETBPM, 2015). 9

Figura 1 – Comunicação ponto-a-ponto MPI (COULOURIS et al, 2005). 10

Figura 1 – Exemplo código MPI (HPC, 2015). 11

Figura 3 – Exemplo de algoritmo utilizando *OpenMP* (LAMMPS, 2015). 13

Figura 4 – Tela inicial do programa PPMsequencial. 16

Figura 4 – Tela de ajuda do programa PPMparalelo. 16

Figura 8 – Execução das matrizes pelo método sequencial. 24

Figura 9 – Execução das matrizes por *Pthreads*. 25

Figura 10 – Execução da matriz de ordem n=250 por *Pthreads*. 25

Figura 11 – Execução da matriz de ordem n=500 por *Pthreads*. 26

Figura 12 – Execução da matriz de ordem n=1000 por *Pthreads*. 26

Figura 13 – Execução da matriz de ordem n=1500 por *Pthreads*. 27

Figura 14 – Execução da matriz de ordem n=2000 por *Pthreads*. 27

Figura 15 – Execução da matriz de ordem n=3000 por *Pthreads*. 28

Figura 16 – Execução da matriz de ordem n=4000 por *Pthreads*. 28

Figura 17 – Execução das matrizes pelo *OpenMP*. 29

Figura 18 – Execução da matriz de ordem n=250 pelo *OpenMP*. 30

Figura 19 – Execução da matriz de ordem n=500 pelo *OpenMP*. 30

Figura 20 – Execução da matriz de ordem n=1000 pelo *OpenMP*. 31

Figura 21 – Execução da matriz de ordem n=1500 pelo *OpenMP*. 31

Figura 22 – Execução da matriz de ordem n=2000 pelo *OpenMP*. 32

Figura 23 – Execução da matriz de ordem n=3000 pelo *OpenMP*. 32

Figura 24 – Execução da matriz de ordem n=4000 pelo *OpenMP*. 33

Figura 25 – SpeedUp entre os algoritmos paralelos e sequencial. 34

# Lista de Tabelas

Tabela 1 – Especificação do hardware utilizado. 15

Tabela 2 – Execução das matrizes pelo método sequencial. 23

Tabela 3 – Execução das matrizes por *Pthreads*. 24

Tabela 4 – Execução das matrizes pelo *OpenMP*. 29

Tabela 5 – Execução da matriz de odem 250 pelo método sequencial. 38

Tabela 6 – Execução da matriz de odem 500 pelo método sequencial. 38

Tabela 7 – Execução da matriz de odem 1000 pelo método sequencial. 38

Tabela 8 – Execução da matriz de odem 1500 pelo método sequencial. 38

Tabela 9 – Execução da matriz de odem 2000 pelo método sequencial. 39

Tabela 10 – Execução da matriz de odem 3000 pelo método sequencial. 39

Tabela 11 – Execução da matriz de odem 250 por *Pthreads*. 40

Tabela 12 – Execução da matriz de odem 500 por *Pthreads*. 40

Tabela 13 – Execução da matriz de odem 1000 por *Pthreads*. 40

Tabela 14 – Execução da matriz de odem 1500 por *Pthreads*. 41

Tabela 15 – Execução da matriz de odem 2000 por *Pthreads*. 41

Tabela 16 – Execução da matriz de odem 3000 por *Pthreads*. 41

Tabela 17 – Execução da matriz de odem 4000 por *Pthreads*. 42

Tabela 18 – Execução da matriz de odem 250 pelo *OpenMP*. 43

Tabela 19 – Execução da matriz de odem 500 pelo *OpenMP*. 43

Tabela 20 – Execução da matriz de odem 1000 pelo *OpenMP*. 43

Tabela 21 – Execução da matriz de odem 1500 pelo *OpenMP*. 44

Tabela 22 – Execução da matriz de odem 2000 pelo *OpenMP*. 44

Tabela 23 – Execução da matriz de odem 3000 pelo *OpenMP*. 44

Tabela 24 – Execução da matriz de odem 4000 pelo *OpenMP*. 45

# Introdução

A ciência da computação é uma área abrangente envolvendo vários aspectos nas mais variadas esferas do conhecimento. Ainda segundo Brookshear (2013):

A ciência da computação é uma disciplina que busca construir uma base científica para tópicos como projeto e programação de computadores, processamento de informação, soluções algorítmicas de problemas e o próprio processamento algorítmico.

Dentro dessa área de conhecimento existem os algoritmos, os quais são importantes para resolver problemas ou criar soluções para os mais diversos paradigmas computacionais. Eles são um conjunto de passos que definem como uma ou mais tarefas serão realizadas (BROOKSHEAR, 2013).

Como o surgimento dos computadores e os algoritmos, o tempo de processamento das tarefas foi reduzido substancialmente em processadores de um núcleo. Com o acoplamento de mais núcleos no processador, algoritmos que dividem suas tarefas entre esses núcleos, tendem a ter um melhor desempenho, de acordo com o tipo de dados e sua possibilidade de paralelização (YANO, 2010).

Sendo assim, é importante que o algoritmo desenvolvido avalie todas as possibilidades de paralelização, para extrair um melhor tempo de execução. Atualmente, existem diversas linguagens de programação e bibliotecas que fornecem ferramentas para paralelização, entre elas pode-se citar *Pthreads[[1]](#footnote-1)*, *OpenMP* [[2]](#footnote-2)e *MPI* [[3]](#footnote-3)(SATO, GUARDIA, 2013)

Além da paralelização de processos na *CPU*, é possível enviar trabalho para a *GPU* através de *CUDA*, que é uma plataforma de computação paralela e um modelo de programação desenvolvido pela NVIDIA. Ela permite aumentos significativos de desempenho computacional ao aproveitar a potência da Unidade de Processamento Gráfico (GPU) (NVIDIA, 2015).

Conforme pode ser observado na Figura 1, há um aumento significativo no tempo de execução para algoritmos que utilizam processamento paralelo, tanto para trabalhos enviados para a *CPU* quanto para a *GPU*. Porém na *GPU*, o tempo de resposta foi menor, pelo desempenho da unidade.

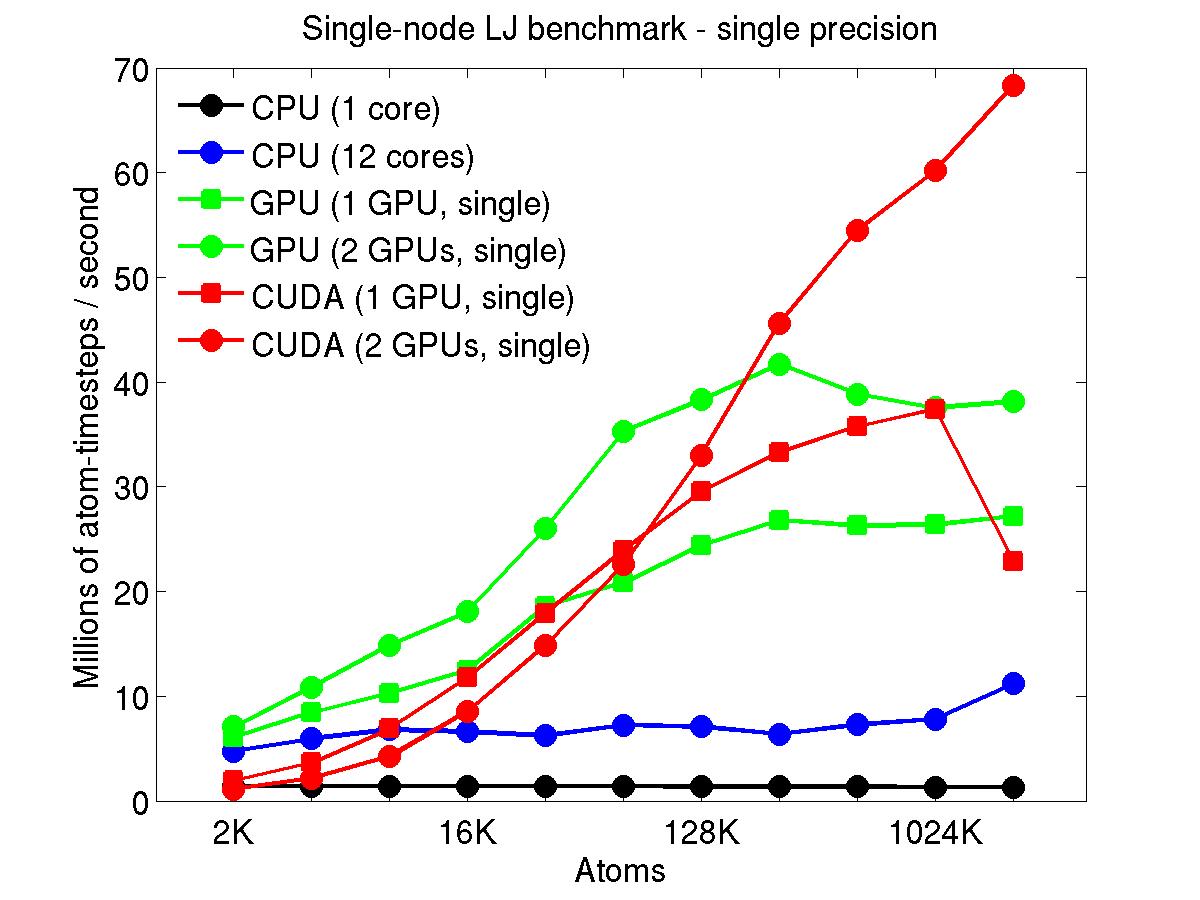


Figura 1 – Benchmark em um único computador (LAMMPS, 2015).

## Objetivos

Tendo em vista os benefícios da paralelização de algoritmos, esse trabalho mostra o desenvolvimento de um algoritmo sequencial e paralelo para o processamento de imagens PPM (*Portable Pixmap Format*), utilizando um filtro para suavização de imagens (denominado *smooth*).

# O Formato PPM

O formato PPM (*Portable Pixmap Format*) é um de arquivo de imagem colorido relativamente simples. Deve notar-se que este formato é notoriamente ineficiente e altamente redundante, também não utiliza algoritmos de compressão de dados. Além disso, esse formato permite muito pouca informação sobre a imagem além da cor base. No entanto, é muito fácil escrever programas para analisar e processar esse arquivo, o que o torna interessante (NETBPM, 2015).

## O Arquivo PPM

Um arquivo PPM consiste em uma sequência de uma ou mais imagens PPM. Não existem dados delimitadores antes, depois ou entre imagens. Segundo Netbpm (2015), cada imagem PPM consiste em:

* Um número para identificar o tipo de arquivo. Exemplo, "P6".
* Espaço em branco (espaço em branco, TABs, CRs, LFs).
* A largura, formatado como caracteres ASCII em decimal.
* Espaço em branco.
* A altura, novamente em decimal ASCII.
* Espaço em branco.
* O valor máximo de cor (MAXVAL), novamente em decimal ASCII. Entre 0 e 65536.
* Um espaço em branco único (geralmente uma nova linha).
* As linhas contendo os dados da imagem. Cada linha é composta por pixels, da esquerda para a direita. Cada pixel é um composto de três valores de cores, vermelho, verde e azul, nesta ordem.

Na Figura 2 é possível visualizar um exemplo de imagem no formato PPM P6.



Figura 2 – Exemplo de imagem PPM (NETBPM, 2015).

A Figura 3 – Exemplo de um programa em linguagem C para manipular imagem PPM (ROSETTA CODE, 2015).Figura 3 contém um código escrito na linguagem C para manipulação de imagem PPM, que escreve uma cor em cada execução.

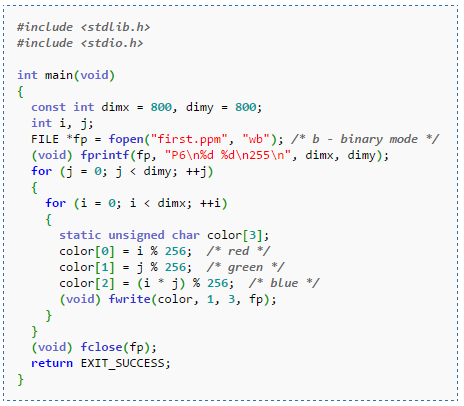


Figura 3 – Exemplo de um programa em linguagem C para manipular imagem PPM (ROSETTA CODE, 2015).

# Paralelismo

Nesse capítulo são expostas as ferramentas para paralelização utilizadas nesse trabalho.

## *OpenMPI*

A interface de passagem de mensagens (MPI) é um padrão desenvolvido para fornecer uma API para um conjunto de operações de transmissão de mensagens com variantes síncronas e assíncronas. O modelo de arquitetura subjacente para MPI é relativamente simples, conforme pode ser visto na Figura 4:

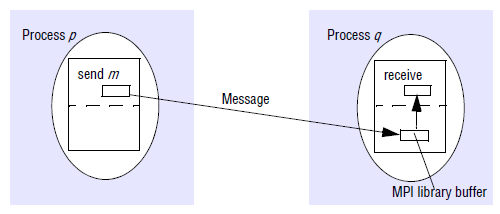


Figura 4 – Comunicação ponto-a-ponto MPI (COULOURIS et al, 2005).

Um processo p envia uma mensagem para um processo q através de uma biblioteca MPI, inicialmente essa mensagem é enviada ao buffer de q, e após, repassada ao processo q (COULOURIS et al, 2005).

### Funcionamento

A interface MPI possui alguns tipos de operação, como bloqueante, não-bloqueante, síncrona e assíncrona:

* MPI\_Send: operação bloqueante, o remetente fica bloqueado até as mensagens serem entregues no *buffer* do destinatário.
* MPI\_Ssend: operação síncrona e bloqueante, remetente e destinatário retornam apenas quando a mensagem final é entregue.
* MPI\_Bsend: bloqueante, o remetente aloca um *buffer* e a chamada retorna quando os dados forem copiados com sucesso para esta memória intermédia.
* MPI\_Rsend: bloqueante, a chamada retorna quando o *buffer* de aplicação do remetente puder ser reutilizado.
* MPI\_Isend: não-bloqueante, a chamada retorna imediatamente após o envio, podendo ser verificado seu progresso com chamadas MPI\_Wait ou MPI\_Test.
* MPI\_Issend: não-bloqueante, porém informado se a mensagem foi entregue ao destinatário.
* MPI\_Ibsend: não-bloqueante, a mensagem é copiada para o *buffer* logo após seu envio.
* MPI\_Irsend: não-bloqueante, em que o programador indica que o receptor que está pronto para receber.

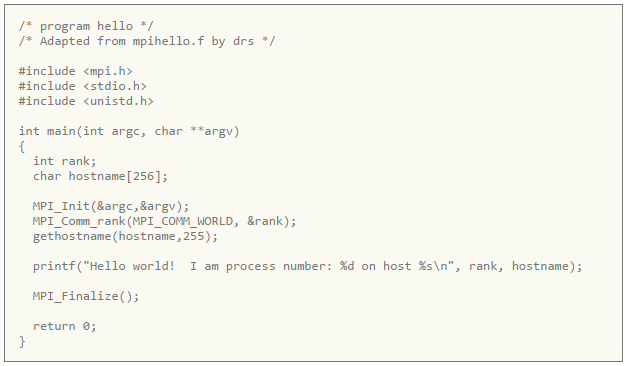


Figura 5 – Exemplo código MPI (HPC, 2015).

Conforme demonstrado na Figura 5, o primeiro passo para a construção de um programa MPI é incluir o arquivo de cabeçalho MPI com #include <mpi.h>. Depois disto, a área de atuação do MPI é definida com MPI\_Init. A próxima linha define MPI\_Comm\_rank, que devolve a classificação de um processo em um comunicador. Cada processo dentro de um comunicador é atribuído um número incremental que começa a partir de zero. As fileiras dos processos são utilizadas principalmente para fins de identificação ao enviar e receber mensagens.

Após é realizado um *print* mostrando o número do processo e o nome do servidor que está localizado. O MPI\_Finalize é usado para limpar o ambiente MPI. Nenhuma chamada MPI pode ser feita após isso.

## *OpenMP*

*OpenMP* é uma API (*Application Program Interface*), definida em conjunto por um grupo de grandes fornecedores de *hardware* e *software*. Fornece um modelo portátil, escalável para desenvolvedores de aplicações paralelas de memória partilhada. A API suporta C/C ++ e Fortran em uma ampla variedade de arquiteturas.

Programas *OpenMP* realizam paralelismo exclusivamente através da utilização de *threads*. Um *thread* é a menor unidade de processamento que pode ser programada por um sistema operacional. Normalmente, o número de *threads* coincide com o número de processadores/núcleos. *OpenMP* é um modelo de programação explícita (não automático), oferecendo ao programador controle total sobre a paralelização. (BARNEY, 2015)

### Funcionamento

Uma das coisas úteis sobre *OpenMP* é que ele permite aos usuários a opção de usar o mesmo código-fonte para compiladores normais quanto para compiladores compatíveis com o *OpenMP*. Isto é possível, utilizando suas diretivas *OpenMP* e comandos ocultos para compiladores normais.

Na Figura 6 é demonstrada uma forma muito simples de programa paralelo *multi-threaded*, escrito em C que irá imprimir "*Hello World*", exibindo o número do *thread* em cada nível de processamento. (KIESSLING, 2009)

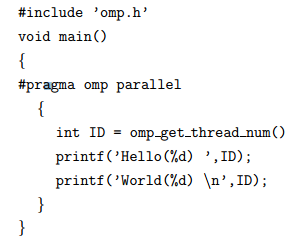


Figura 6 – Exemplo de algoritmo utilizando *OpenMP* (LAMMPS, 2015).

Conforme pode ser visto, a primeira linha é a biblioteca do *OpenMP*. A região paralela é colocada entre a diretiva **#pragma omp parallel** {...}. A instrução omp\_get\_thread\_num() retorna o ID *thread* para o programa.

O *OpenMP* possui uma série de diretivas para tornar o paralelismo customizável pelo programador. Essa e demais opções podem ser encontradas no site e documentação oficial da ferramenta.

## Técnicas de Decomposição

Um dos passos necessários para se resolver um problema em paralelo é dividi-lo em subsistemas para que estes sejam executados em conjunto. Essa divisão é comumente chamada de decomposição. Existem diversas técnicas de decomposição que podem ser utilizadas e às vezes mais de uma técnica pode ser emprega em um problema (GRAMA et al, 2003).

Estas técnicas são amplamente classificadas como:

* Decomposição recursiva: utiliza a abordagem de dividir para conquistar. Sistemas são divididos em sistemas menores e estes são resolvidos recursivamente.
* Decomposição de dados: comumente utilizada para obter a simultaneidade em algoritmos que operam em estruturas de dados grandes. Neste método, a decomposição de cálculos é feito em duas etapas. No primeiro passo, os dados em que os cálculos são efetuados são particionados, e no segundo passo, este particionamento de dados é utilizado para induzir uma compartimentação dos cálculos em tarefas.
* Decomposição exploratória: é usada para decompor problemas cujas computações subjacentes correspondem a uma pesquisa em um espaço de soluções.
* Decomposição especulativa: é usada quando um programa pode assumir um dos muitos possíveis ramos computacionais dependendo da saída de outros cálculos precedentes.

Na Figura 7 é apresentada a decomposição recursiva conhecida como *Quicksort*, que é um algoritmo de dividir e conquistar, que começa selecionando um elemento X, e em seguida particiona em duas seções subsequentes, em que essas seções são menores que X, a inicial. A recursividade termina quando cada seção contiver apenas um único elemento.

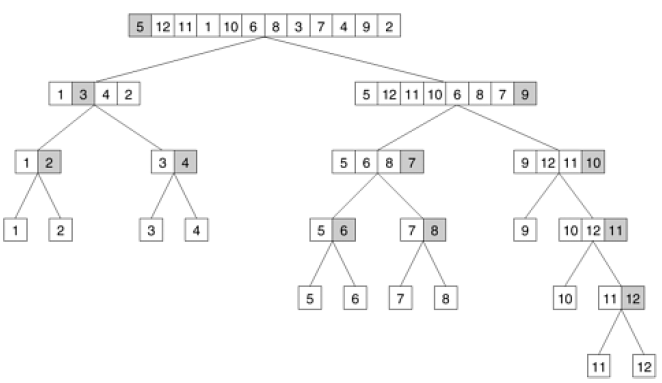


Figura 7 – Exemplo de decomposição: *Quicksort* (GRAMA et al, 2003).

Segundo Grama (2003), inicialmente, existe apenas uma sequência (ou seja, a raiz da árvore), que utiliza apenas um único processo para criar a partição. Da mesma forma, a concorrência continua a aumentar à medida que a árvore se move para baixo. É possível criar outras técnicas para paralelizar a raiz, como por exemplo, definindo mínimos e máximos de elementos.

# Desenvolvimento e Metodologia

Esse capítulo aborda o desenvolvimento dos algoritmos e a metodologia utilizada. Para esse trabalho, foi utilizada linguagem C e Code::Blocks como interface de desenvolvimento.

## Repositório do código

Como o trabalho foi desenvolvido em grupo, o GitHub foi utilizado para compartilhar o código e controlar o versionamento. O repositório pode ser acessado pelo link: **https://github.com/naylor/trabalho2**

## Hardware

A Tabela 1 mostra o hardware utilizado nos experimentos, para os algoritmos sequencial e paralelo.

Tabela 1 – Especificação do hardware utilizado.

|  |  |  |
| --- | --- | --- |
| COMPONENTE | | MODELO |
| Placa­mãe | ASUSTeK COMPUTER INC. S550CA | |
| Sistema Operacional | Linux Ubuntu 14.04.3 LTS | |
| Processador | Intel Core i7 3537U @ 2.00GHz | |
| Memória RAM | 8,00GB Dual-Channel DDR3 798MHz | |
| Hard­Drive | 465GB Western Digital WDC WD5000LPVX-80V0TT0 (SATA) | |

## Programa: PPMsequencial

O trabalho gera dois programas, o sequencial (PPMsequencial) e o paralelo (PPMparalelo). Ambos utilizam as mesmas funções, porém foram divididos pelo fato do MPI ser introduzido no código e também por melhorar a clareza do código. Na versão sequencial, conforme pode ser observado na Figura 8, o usuário pode escolher entre utilizar o menu de navegação ou realizar os lançamentos por linha de comando.

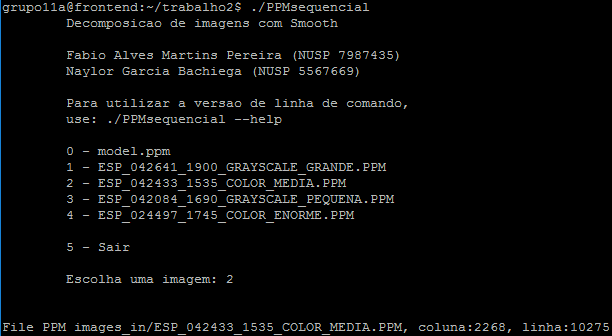


Figura 8 – Tela inicial do programa PPMsequencial.

Ainda é possível utilizar o programa sequencial por linha de comando ou acessar sua tela de ajuda. O próprio sistema carrega automaticamente as imagens contidas no diretório (images\_in) e disponibiliza ao usuário.

## Programa: PPMparalelo

Na versão paralela, o sistema funciona somente por linha de comando. A Figura 9, mostra a tela de ajuda do programa que é informada ao usuário e quais as opções possíveis de lançamento.

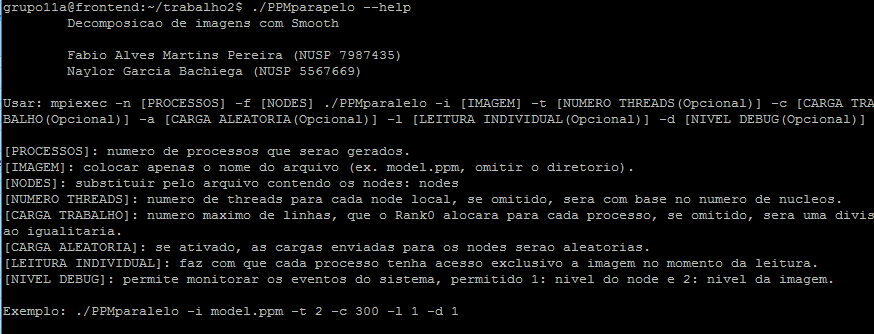


Figura 9 – Tela de ajuda do programa PPMparalelo.

As opções de lançamento são:

* Processos: é o número de processos que o MPI vai lançar do programa.
* Nodes: são os nós que participarão da divisão entre os processos. Nesse caso é um arquivo de texto contendo o *hostname* dos servidores do cluster utilizado no trabalho.
* Número *Threads*: é o número de *threads* que cada processo vai lançar para executar a leitura da imagem e aplicação do filtro.
* Carga Trabalho: permite informar a quantidade de carga de trabalho que cada processo irá receber.
* Carga Aleatória: se ativado, as cargas enviadas para os nodes serão aleatórias.
* Leitura Individual: faz com que os nodes não tenham concorrência ao tentar ler a imagem no disco.
* Nível Debug: permite analisar o comportamento do programa no momento da execução.

O sistema comum foi dividido conforme descrito a seguir:

* **main.c**: faz a inicialização do sistema, carrega o menu e as escolhas do usuário.
* **menu.c**: o menu do usuário e suas escolhas.
* **funcao.c**: funções importantes para o sistema, como enviar o resultados para a tela e imprimir os resultados no arquivo.
* **imagem.c**: contém todas as funções de leitura, manipulação e escrita da imagem.
* **timer.c**: função para imprimir o tempo de execução dos algoritmos.

## README.md

O arquivo README.md contém informação sobre outros arquivos de um projeto ou sistema, como por exemplo autores, procedimentos de instalação, agradecimentos, *bugs*, entre outros.

Abaixo segue o arquivo referente ao sistema desenvolvimento para esse trabalho.

trabalho-prog-conc

Usando MPI e OpenMP para aplicação de Smooth em Imagens PPM

===========================================================

### Dependências

Esse programa divide a imagem em linhas e distribui para os nós pelo MPI.

O rank 0 comanda a comunicação. É responsável por enviar o trabalho para os nós e gerenciar quem pode gravar no disco para não gerar concorrência.

Cada nó divide seu trabalho em Threads que realizam a leitura de sua parte da imagem e aplicam a técnica de Smooth. Quando terminado o trabalho, o nó solicita permissão para gravar o resultado no disco.

### Dependências

1. Pacotes necessários:

\* sudo apt-get install build-essential

\* apt-get install libcr-dev mpich2 mpich2-doc

\* sudo apt-get install gcc-multilib

### Instalação

1. Faça o clone deste projeto:

git clone https://github.com/naylor/trabalho2

2. Entre na pasta do projeto

3. Rode o comando "make"

### Executando a aplicação

1. PPMsequencial (versão sequencial)

\* Utilizando o PPMsequencial por menu

usar: ./PPMsequencial

\* Executando o PPMsequencial pelo terminal

usar: ./PPMsequencial --help

ou

usar: ./PPMsequencial -i [IMAGEM] -d [NÍVEL DEBUG]

2. PPMparalelo (versão paralela)

\* Executando o PPMparalelo pelo terminal

usar: ./PPMparalelo --help

ou

usar: mpiexec -n [PROCESSOS] -f [NODES] ./PPMparalelo -i [IMAGEM] -t [NÚMERO THREADS] -c [CARGA DE TRABALHO] -a [CARGA ALEATORIA] -t [LEITURA INDIVIDUAL] -d [NÍVEL DEBUG]

\* [PROCESSOS]: número de processos que serão gerados.

\* [IMAGEM]: colocar apenas o nome do arquivo (ex. model.ppm, omitir o diretório).

\* [NODES]: substituir pelo arquivo contendo os nodes: nodes.

\* [NUMERO THREADS]: número de threads para cada node local, se omitido, será com base no número de nucleos.

\* [CARGA TRABALHO]: número máximo de linhas, que o Rank0 alocará para cada processo, se omitido, será uma divisão igualitária.

\* [CARGA ALEATORIA]: se ativado, as cargas enviadas para os nodes serao aleatorias.

\* [LEITURA INDIVIDUAL]: faz com que cada processo tenha acesso exclusivo a imagem no momento da leitura.

\* [NIVEL DEBUG]: permite monitorar os eventos do sistema, permitido 1: nível do node e 2: nível da imagem.

3. Os resultados são gravados na pasta: resultados

4. Imagens PPM disponíveis na pasta: images\_in

5. Imagens processados com Smooth na pasta: images\_out

## Makefile

O *make* é utilitário *Unix* que é projetado para iniciar a execução de um *makefile*. Um *makefile* é um arquivo especial, contendo comandos *shell*, geralmente instruções necessárias para a compilação do programação. Para o trabalho, o *Makefile* foi configurado conforme a seguir:

# MAKEFILE #

#INFORMANDO O COMPILADOR,

#DIRETÓRIOS E O

#NOME DO PROGRAMA

CC=gcc

G=g++

MPICC=mpicc

MPIG=mpic++

SRCCOMMON=common/

SRCSEQ=sequencial/

SRCPAR=paralelo/

SEQ=PPMsequencial

PAR=PPMparapelo

# FLAGS NECESSARIAS

# PARA COMPILACAO

CFLAGS=-Wall -Wextra -fopenmp

LIB=-fopenmp

#-------------------------------

# CARREGA AUTOMATICAMENTE OS

# ARQUIVOS .C E .H

#-------------------------------

SOURCESEQ=$(wildcard $(SRCSEQ)\*.c $(SRCCOMMON)\*.c)

HEADERSEQ=$(wildcard $(SRCSEQ)\*.h $(SRCCOMMON)\*.h)

SOURCEPAR=$(wildcard $(SRCPAR)\*.c $(SRCCOMMON)\*.c)

HEADERPAR=$(wildcard $(SRCPAR)\*.h $(SRCCOMMON)\*.h)

all: $(SEQ) $(PAR)

$(PAR): $(SOURCEPAR:.c=.o)

$(MPIG) -o $@ $^ $(LIB)

%.o: %.c $(HEADERPAR)

$(MPICC) -g -c $< -o $@ $(CFLAGS)

$(SEQ): $(SOURCESEQ:.c=.o)

$(G) -o $@ $^ $(LIB)

%.o: %.c $(HEADERSEQ)

$(CC) -g -c $< -o $@ $(CFLAGS)

clean:

rm -f $(SRCSEQ)\*.o $(SRCPAR)\*.o $(SRCCOMMON)\*.o

rm -f $(SEQ) $(PAR)

## Decomposição do Problema

Um algoritmo paralelo é derivado a partir de uma escolha de distribuição de dados. A distribuição de dados deve ser equilibrada, alocar (aproximadamente) o mesmo número de entradas para cada processador; e deverá minimizar a comunicação.

Baseado nos dados mostrados na Seção 3.3, a decomposição deve ser construída de forma que o trabalho seja divido logo no início e que os processos recebam os elementos para processamento. A Figura 10 demonstra o funcionamento do algoritmo paralelo.



Figura 10 – Decomposição do problema.

O algoritmo segue os passos descritos a seguir e as explicações de cada ação:

1. Rank 0 faz a leitura somente do cabeçalho, pois o este contém o tamanho da imagem. Não é necessário ler a imagem inteira.
2. Rank 0 grava o cabeçalho da nova imagem e envia algumas configurações para os Ranks: tamanho da imagem, número de threads, caminho da imagem de leitura e caminho da imagem de gravação, onde a leitura deve iniciar na imagem e onde eles devem escrever na nova imagem.
3. Rank 0 divide o trabalho para os demais Ranks de três formas possíveis:

3.1 Divide a imagem igualitariamente.

3.2 O usuário define uma quantidade de linhas pelo terminal.

3.3 É escolhida uma carga aleatória entre a carga máxima e 50% dela. Nos testes realizados, quanto mais grão fino, pior o tempo de resposta, por isso a limitação a 50% da carga para o valor de mínimo.

1. Rank 0 cria 1 thread para cada processo, cada thread envia o trabalho para seu processo responsável e aguarda pelo processo.
2. Rank 1..N recebe a parte da imagem que deve processar. Cada processo gera 3..N threads (caso o usuário não defina o número de threads, o padrão é 3) para fazer a leitura somente da parte que recebeu, não é lida a imagem inteira. Porém há três tipos de leitura:

5.1 Se o Rank recebeu o início da imagem, ele faz a leitura da sua parte, mais as duas últimas linhas, isso é necessário para poder calcular o Smooth.

5.2 Se o Rank recebeu um bloco do meio da imagem, ele faz a leitura de duas linhas acima e duas abaixo.

5.3 Se o Rank recebeu o fim da imagem, ele faz a leitura das duas linhas acima.

1. Rank 1..N aplica o filtro de Smooth e envia mensagem para sua thread correspondente no Rank 0 solicitando permissão para gravar.
2. A thread responsável no Rank 0 que recebeu solicitação de gravação, tenta acessar a região crítica de memória para tentar ganhar a vez de gravar. Tentará até conseguir. Quando conseguir acessar a região crítica, informará ao Rank, sob sua guarda, que pode gravar e aguardo ele responder.
3. Rank 1..N que recebeu permissão de gravação de sua thread, faz a gravação da sua parte da imagem no disco e informa que terminou a gravação.
4. A thread no rank 0 recebe a mensagem informando do término da gravação e deixa a região crítica de memória, abrindo disputa para outras threads. A thread verifica se ainda há linhas da imagem para processar. Se houver, enviará para o Rank que voltará ao item 4, caso não, informa para ele que não há mais serviços e que este pode finalizar.
5. Rank 1..N finaliza.
6. Rank 0 finaliza.

# Resultados e Discussões

Aqui são apresentados os resultados e discussões com base nos algoritmos sequencial e paralelo.

## Algoritmo Sequencial

Inicialmente foram executadas as matrizes de ordem *n x n* utilizando o algoritmo sequencial, obtendo os resultados conforme a Tabela 2:

Tabela 2 – Execução das matrizes pelo método sequencial.

|  |  |  |
| --- | --- | --- |
| ORDEM  *n x n* | MÉDIA  (segundos) | DESVIO  (segundos) |
| 250 | 0,9556787 | 0,001737338 |
| 500 | 5,2710011 | 0,002590176 |
| 1000 | 58,2954194 | 0,973853303 |
| 1500 | 194,5482731 | 3,243080053 |
| 2000 | 479,9629639 | 0,15378519 |
| 3000 | 1615,233688 | 0,165632233 |
| 4000 | * 999999 | 0 |

Foram realizadas as médias das dez execuções para cada ordem da matriz (ANEXO A). Não foi possível determinar o tempo de execução da matriz de ordem 4000. Após nove horas de execução, o algoritmo foi interrompido e um valor aproximado foi inserido apenas para efeitos de comparação. Nesse caso, essa matriz foi desconsiderada em alguns gráficos comparativos.

A Figura 11 mostra como o tempo aumenta, consideravelmente, quando a ordem da matriz é acrescida.

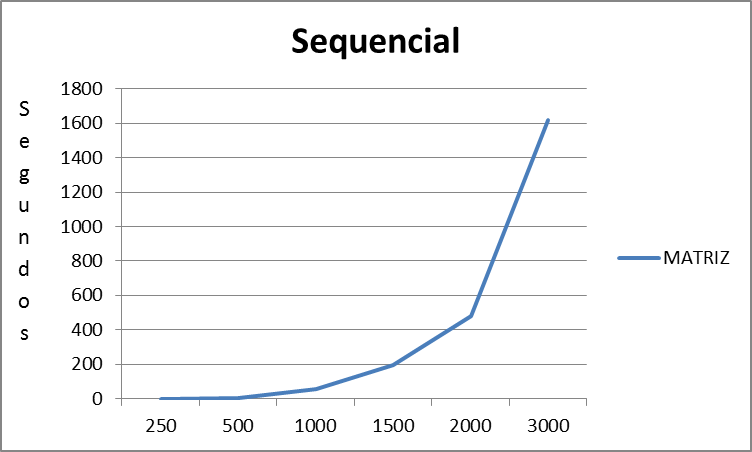


Figura 11 – Execução das matrizes pelo método sequencial.

Apesar de adicionado o desvio padrão no gráfico, não é possível visualizar pela grandeza das diferenças.

## Algoritmo Paralelo: Pthreads

Os mesmos testes foram realizados para o algoritmo paralelo com *Pthreads*. Foi observado que, em um determinado momento, aumentar a quantidade de *threads* também aumentava o tempo de execução. Sendo assim, na Tabela 3 são apresentados os melhores resultados para cada ordem de matriz e suas respectivas quantidades de *threads* para o melhor tempo de execução.

Tabela 3 – Execução das matrizes por *Pthreads*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM  *n x n* | THREADS | MÉDIA  (segundos) | DESVIO  (segundos) |
| 250 | 5 | 0,0193803 | 0,000175189 |
| 500 | 5 | 0,0569816 | 0,000645895 |
| 1000 | 5 | 0,1887689 | 0,002332534 |
| 1500 | 5 | 0,3959849 | 0,004570521 |
| 2000 | 5 | 0,7129205 | 0,010868422 |
| 3000 | 7 | 1,7038281 | 0,018810684 |
| 4000 | 10 | 2,9953613 | 0,021264158 |

Na Figura 12 são apresentados os resultados com os tempos de execução de cada ordem de matriz e a quantidade de *threads* que proporcionou o melhor resultado.

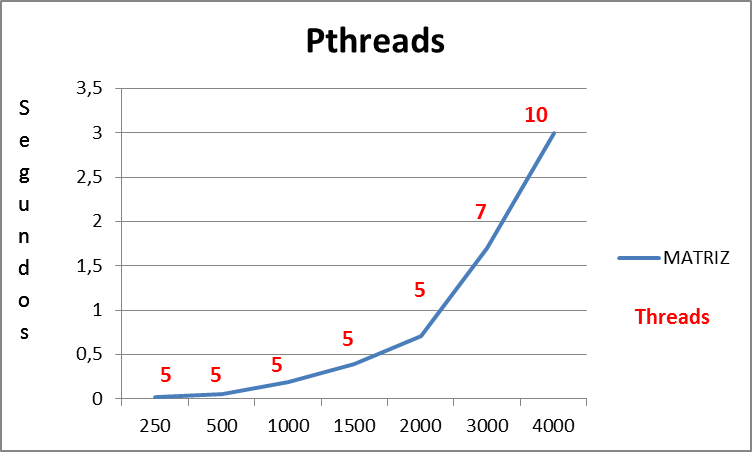


Figura 12 – Execução das matrizes por *Pthreads*.

### Processamento das Matrizes

Na Figura 13 é possível observar o comportamento da execução de acordo com a quantidade de *threads* informada ao programa.

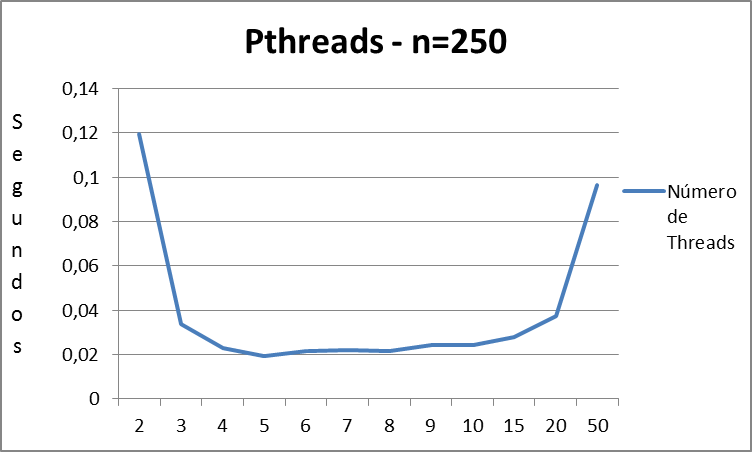


Figura 13 – Execução da matriz de ordem n=250 por *Pthreads*.

Como mencionado, com quatro *threads* o sistema obteve um melhor tempo de execução. O mesmo pode ser observado nas figuras a seguir.

Execução da matriz de ordem n=500.

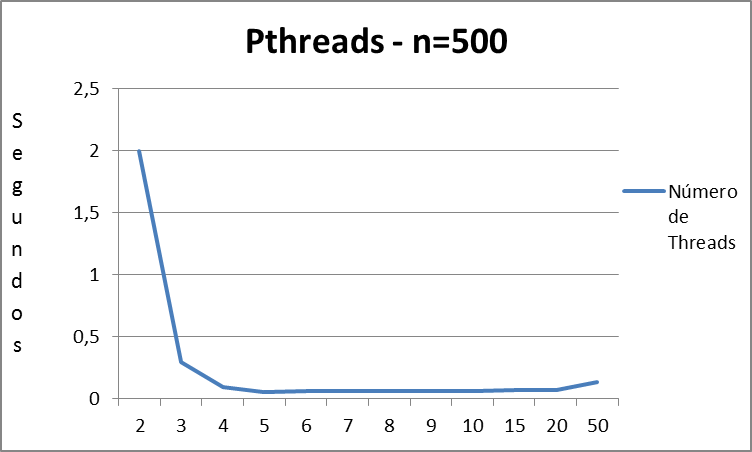


Figura 14 – Execução da matriz de ordem n=500 por *Pthreads*.

Execução da matriz de ordem n=1000.

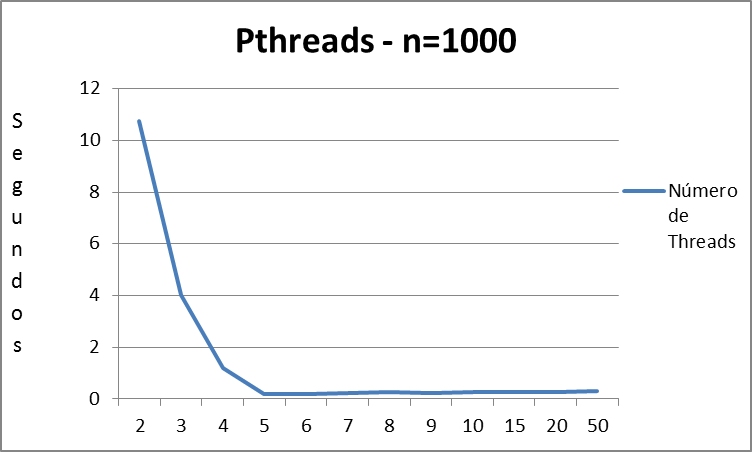


Figura 15 – Execução da matriz de ordem n=1000 por *Pthreads*.

Execução da matriz de ordem n=1500.

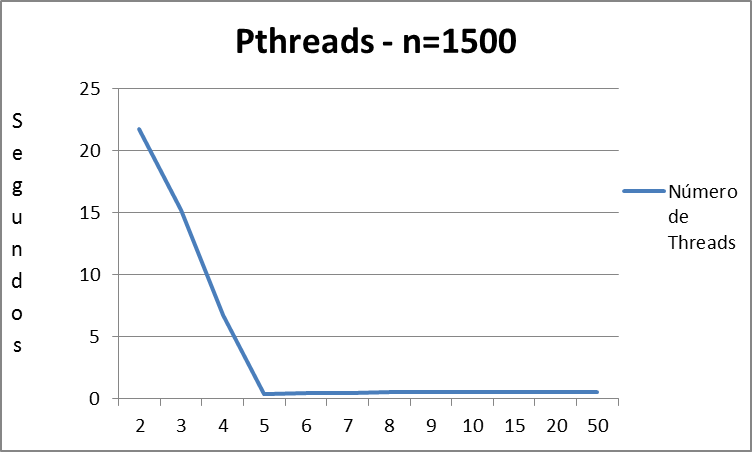


Figura 16 – Execução da matriz de ordem n=1500 por *Pthreads*.

Execução da matriz de ordem n=2000.

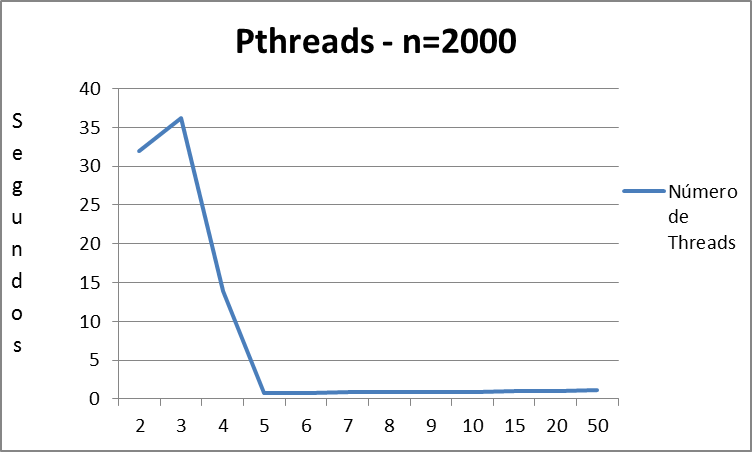


Figura 17 – Execução da matriz de ordem n=2000 por *Pthreads*.

Execução da matriz de ordem n=3000.

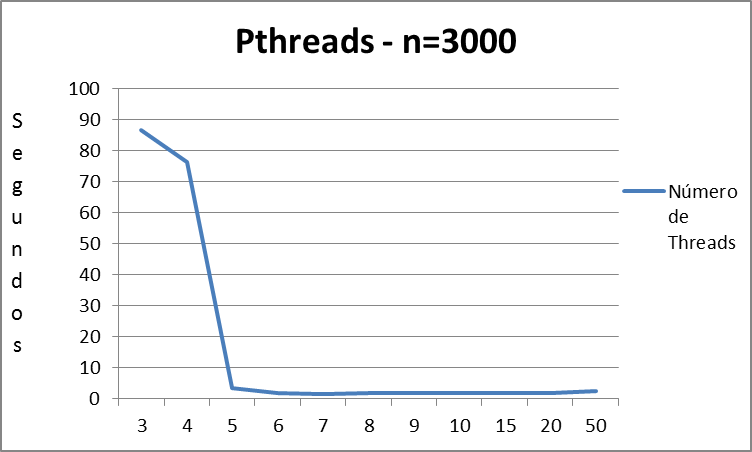


Figura 18 – Execução da matriz de ordem n=3000 por *Pthreads*.

Execução da matriz de ordem n=4000.

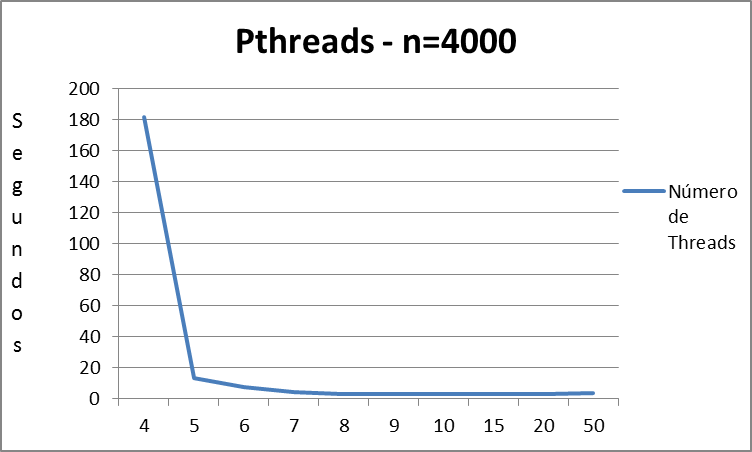


Figura 19 – Execução da matriz de ordem n=4000 por *Pthreads*.

## Algoritmo Paralelo: *OpenMP*

Aqui são demonstrados os testes com o *OpenMP*. Também foi observado que, em um determinado momento, aumentar a quantidade de *threads* matinha o resultado estável em relação ao tempo de execução. Sendo assim, na Tabela 3 são apresentados os melhores resultados para cada ordem de matriz e suas respectivas quantidades de *threads* para o melhor tempo de execução.

Tabela 4 – Execução das matrizes pelo *OpenMP*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM  *n x n* | THREADS | MÉDIA  (segundos) | DESVIO  (segundos) |
| 250 | 5 | 0,0243607 | 3,29274E-05 |
| 500 | 5 | 0,0826408 | 7,89174E-05 |
| 1000 | 5 | 0,3148744 | 0,000220465 |
| 1500 | 6 | 0,6800495 | 0,000296371 |
| 2000 | 2 | 1,1732628 | 0,003280427 |
| 3000 | 3 | 2,6221947 | 0,001255182 |
| 4000 | 2 | 4,4473706 | 0,004068199 |

Na Figura 12 são apresentados os resultados com os tempos de execução de cada ordem de matriz e a quantidade de *threads* que proporcionou o melhor resultado.

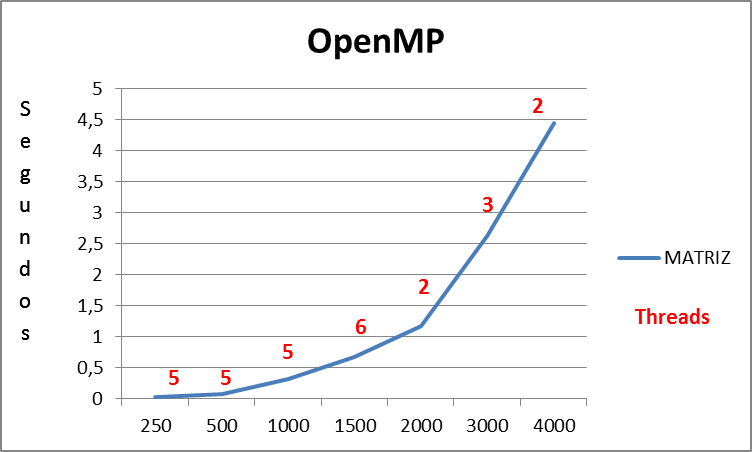


Figura 20 – Execução das matrizes pelo *OpenMP*.

### Processamento das Matrizes

Na Figura 13 é possível observar o comportamento da execução de acordo com a quantidade de *threads* informada ao programa.

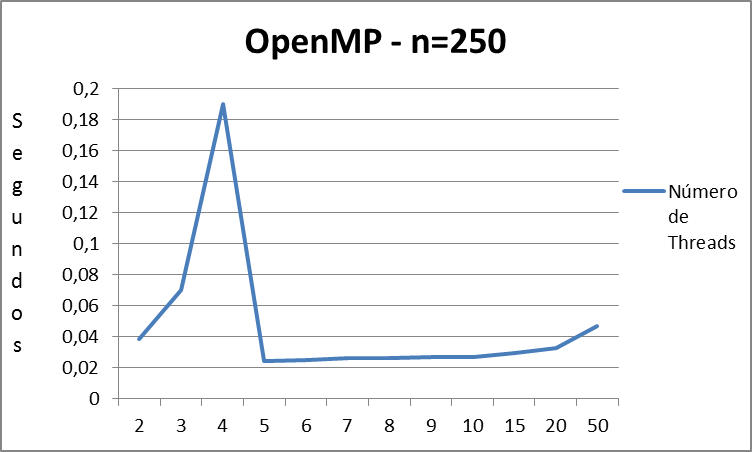


Figura 21 – Execução da matriz de ordem n=250 pelo *OpenMP*.

Como mencionado, com quatro *threads* o sistema obteve um melhor tempo de execução. O mesmo pode ser observado nas figuras a seguir.

Execução da matriz de ordem n=500.

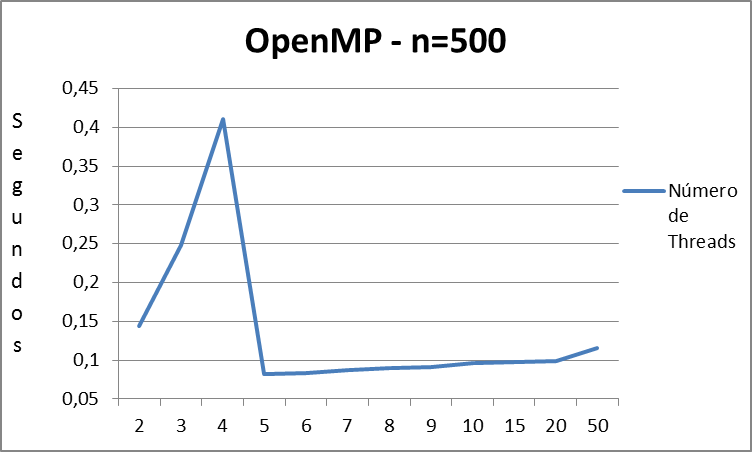


Figura 22 – Execução da matriz de ordem n=500 pelo *OpenMP*.

Execução da matriz de ordem n=1000.

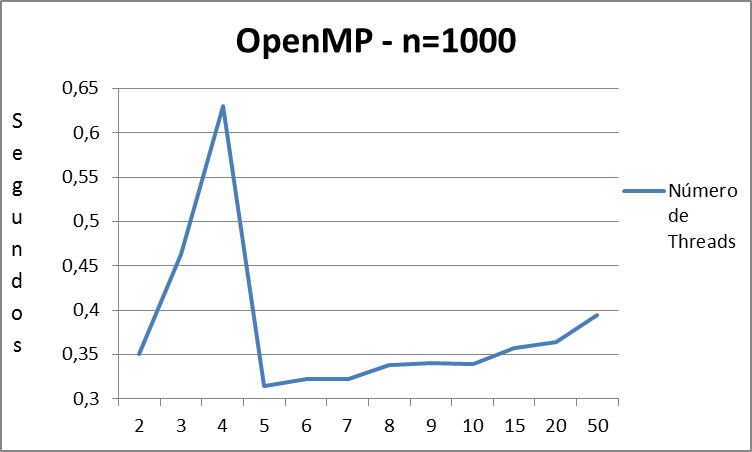


Figura 23 – Execução da matriz de ordem n=1000 pelo *OpenMP*.

Execução da matriz de ordem n=1500.

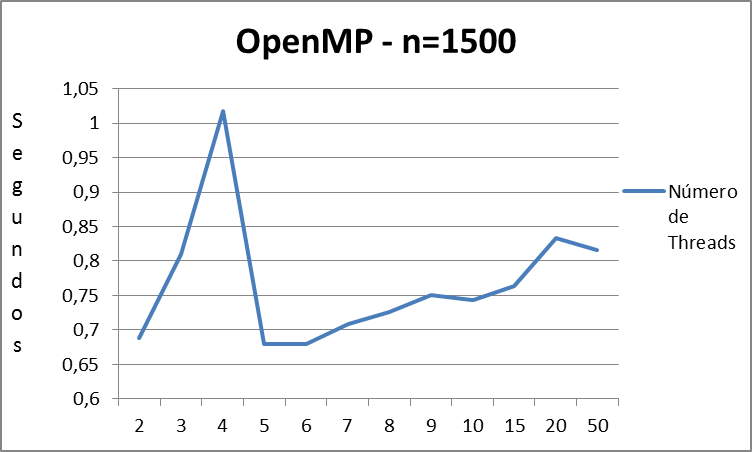
****

Figura 24 – Execução da matriz de ordem n=1500 pelo *OpenMP*.

Execução da matriz de ordem n=2000.

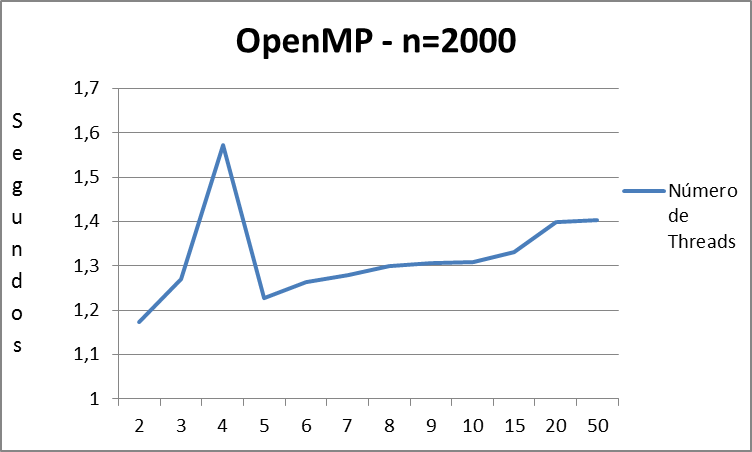
****

Figura 25 – Execução da matriz de ordem n=2000 pelo *OpenMP*.

Execução da matriz de ordem n=3000.

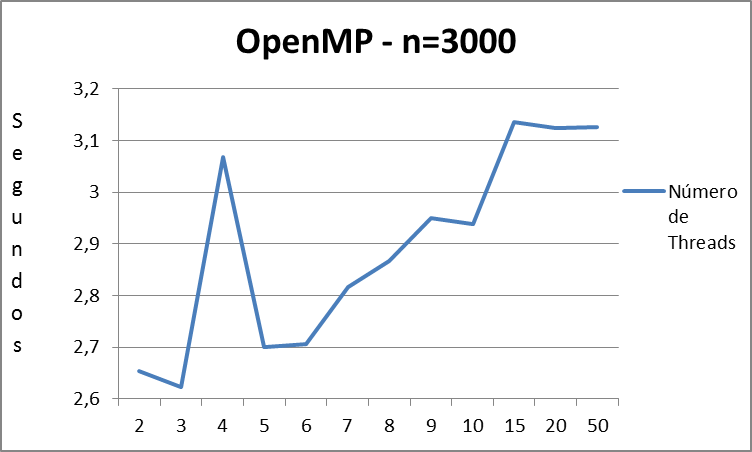
****

Figura 26 – Execução da matriz de ordem n=3000 pelo *OpenMP*.

Execução da matriz de ordem n=4000.

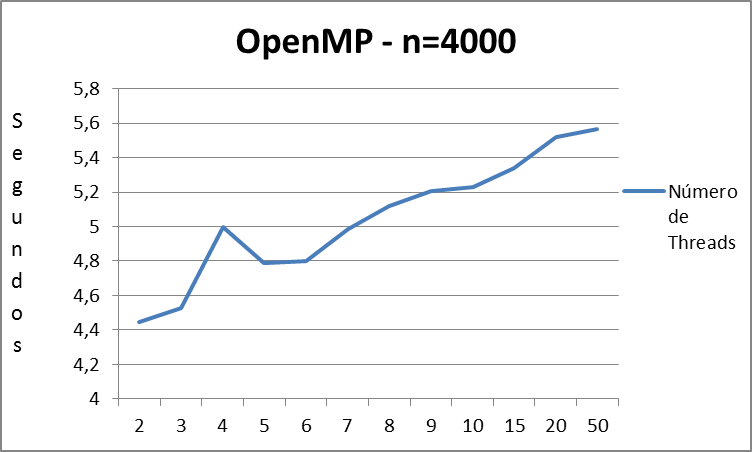
****

Figura 27 – Execução da matriz de ordem n=4000 pelo *OpenMP*.

## SpeedUp

Para medir o aumento de desempenho do processamento utiliza-se o cálculo chamado de *Speedup* que determina a relação existente entre o código executado em *threads* e sequencial. A medida tem por objetivo determinar a relação existente entre o tempo dispensado para executar um algoritmo em um único processador (T1) e o tempo gasto para executá-lo em p processadores (Tp): Speedup = T1/Tp (ROHDE et al, 2015). A Figura 28 mostra essa relação.

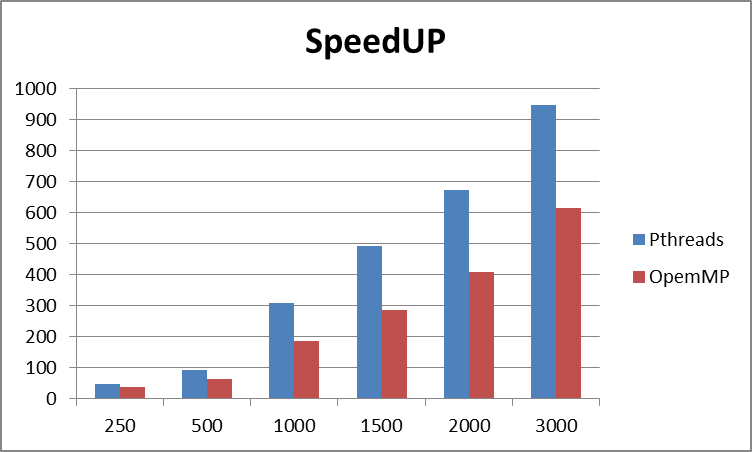


Figura 28 – SpeedUp entre os algoritmos paralelos e sequencial.

## Considerações Finais

Conforme pode ser observado ao longo desse trabalho, paralelizar aplicações permitiu um ganho de tempo substancial em vista de algoritmos sequenciais. Deve-se ressaltar que nem todas as aplicações são paralelizáveis e que o método de abordagem do programa paralelizado também influencia no tempo de execução.

Além disso, foram encontradas algumas dificuldades no desenvolvimento desse trabalho, como entender o funcionamento das *Pthreads*, o modo como *OpenMP* distribui as tarefas, problemas com testes na matriz de ordem n=4000 e construir o algoritmo paralelo pensando na melhor forma de execução.

# Referências

ABREU, Luciano. **Calculo Numérico - Método de Jacobi-Richardson**. Disponível em: <https://www.youtube.com/watch?v=2AORqeCrQEc>. Acesso em: 30 ago. 2015

ALVES, Carlos J. S. **Métodos Iterativos para Sistemas Lineares**. Disponível em: < http://www.math.ist.utl.pt/~calves/cursos/SisLin-Iter.htm>. Acesso em: 30 ago. 2015

ANDRETTA, Marina. **Sistemas lineares - Método Iterativo de Jacobi-Richardson**. São Carlos, 2008.

BALBO, Antonio Roberto. **Métodos iterativos de solução de sistemas lineares**. Disponível em: <http://wwwp.fc.unesp.br/~arbalbo/Iniciacao\_Cientifica/sistemaslineares/teoria/jacobi\_richardson.pdf>. Acesso em: 26 ago. 2015.

BARNEY, Blaise. **OpenMP**. Disponível em: < https://computing.llnl.gov/tutorials/openMP/>. Acesso em: 03 out. 2015.

BROOKSHEAR, J. Glenn. **Ciência da computação: uma visão abrangente**. 11ª ed. Porto Alegre: Bookman, 2013.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Distributed Systems - Concepts and Design**. Addison Wesley Publ. Comp., 4th edition, 2005.

GRAMA, A. Gupta, G. Karypis, and V. Kumar. **Introduction to Parallel Computing**. 2nd edition, Addison Wesley; 2003.

HPC. **Examples of MPI Programs**. Disponível em <https://hpcc.usc.edu/support/documentation/examples-of-mpi-programs/>. Acesso em: 23 out. 2015.

KIESSLING, Alina. **An Introduction to Parallel Programming with OpenMP**. Disponível em: <http://www.roe.ac.uk/ifa/postgrad/pedagogy/2009\_kiessling.pdf>. Acesso em: 04 out. 2015

LAMMPS, Molecular Dynamics Simulator. ***GPU and USER-CUDA package benchmarks on Desktop system with Fermi GPUs***. Disponível em: <https://computing.llnl.gov/tutorials/pthreads/#Pthread>. Acesso em: 05 set. 2015

LLNL. ***POSIX Threads Programming***. Disponível em: <http://lammps.sandia.gov/bench.html>. Acesso em: 02 set. 2015

NETBPM. **PPM**. Disponível em: <http://netpbm.sourceforge.net/doc/ppm.html>. Acesso em: 23 de out. 2015.

NVIDIA. **O que é *CUDA*?**. Disponível em: <http://www.nvidia.com.br/object/cuda\_home\_new\_br.html>. Acesso em: 02 set. 2015

ROHDE, M. Tiago; DESTEFANI, Luciano; FERRARI, Edilaine; MARTINS, Rogério. **As diferentes técnicas de implementação paralela de algoritmos recursivos em C.** Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2012/0034.pdf>. Acesso em: 02 set. 2015.

ROSETTA CODE. **Bitmap/Write a PPM file**. Disponível em: <http://rosettacode.org/wiki/Bitmap/Write\_a\_PPM\_file>. Acesso em 23 de out. 2015.

SATO, Liria Matsumoto; GUARDIA, Hélio Crestana. **Programando para múltiplos processadores: *Pthreads, OpenMP* e *MPI*.** Disponível em: <http://erad.dc.ufscar.br/mc/eradsp2013-multiproc-3.pdf>. Acesso em: 02 set. 2015.

SOUZA, Marcone Jamilson Freitas. **Sistemas Lineares**. Disponível em: <http://www.decom.ufop.br/marcone/Disciplinas/CalculoNumerico/Sistemas.pdf>. Acesso em: 03 set. 2015.

OPENMP. ***What is OpenMP*?**. Disponível em: <http://www.openmp.org/mp-documents/paper/node3.html>. Acesso em: 02 set. 2015.

OPEN-MPI. ***General information about the Open MPI Project***. Disponível em: <https://www.open-mpi.org/faq/?category=general>. Acesso em: 05 set. 2015.

YANO, Luis Gustavo Abe. **Avaliação e comparação de desempenho utilizando tecnologia CUDA**. São José do Rio Preto: 2010.

# Anexo A – Execuções sequenciais

Tabela 5 – Execução da matriz de odem 250 pelo método sequencial.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DATA | TEMPO (MILISSEGUNDOS) |
| 250 | 0 | 05/09/15 14:35 | 937355 |
| 250 | 0 | 05/09/15 14:37 | 944146 |
| 250 | 0 | 05/09/15 14:41 | 969399 |
| 250 | 0 | 05/09/15 14:45 | 941395 |
| 250 | 0 | 05/09/15 14:45 | 944782 |
| 250 | 0 | 05/09/15 14:45 | 966939 |
| 250 | 0 | 05/09/15 14:45 | 973535 |
| 250 | 0 | 05/09/15 14:46 | 983961 |
| 250 | 0 | 05/09/15 14:46 | 934186 |
| 250 | 0 | 05/09/15 14:46 | 961089 |

Tabela 6 – Execução da matriz de odem 500 pelo método sequencial.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DATA | TEMPO (MILISSEGUNDOS) |
| 500 | 0 | 04/10/15 09:45 | 5261713 |
| 500 | 0 | 04/10/15 09:45 | 5312198 |
| 500 | 0 | 04/10/15 09:45 | 5267882 |
| 500 | 0 | 04/10/15 09:45 | 5293331 |
| 500 | 0 | 04/10/15 09:46 | 5298443 |
| 500 | 0 | 04/10/15 09:46 | 5289794 |
| 500 | 0 | 04/10/15 09:46 | 5246556 |
| 500 | 0 | 04/10/15 09:46 | 5240019 |
| 500 | 0 | 04/10/15 09:46 | 5260129 |
| 500 | 0 | 04/10/15 09:46 | 5239946 |

Tabela 7 – Execução da matriz de odem 1000 pelo método sequencial.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DATA | TEMPO (MILISSEGUNDOS) |
| 1000 | 0 | 05/09/15 17:21 | 30592120 |
| 1000 | 0 | 05/09/15 17:23 | 61448066 |
| 1000 | 0 | 05/09/15 17:25 | 60549921 |
| 1000 | 0 | 05/09/15 17:26 | 61599126 |
| 1000 | 0 | 05/09/15 17:32 | 61432035 |
| 1000 | 0 | 05/09/15 17:34 | 61482450 |
| 1000 | 0 | 05/09/15 17:35 | 61447323 |
| 1000 | 0 | 05/09/15 17:37 | 61587433 |
| 1000 | 0 | 05/09/15 17:39 | 61402718 |
| 1000 | 0 | 05/09/15 17:40 | 61413002 |

Tabela 8 – Execução da matriz de odem 1500 pelo método sequencial.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DATA | TEMPO (MILISSEGUNDOS) |
| 1500 | 0 | 05/09/15 17:42 | 102248984 |
| 1500 | 0 | 05/09/15 17:46 | 204817838 |
| 1500 | 0 | 05/09/15 17:49 | 204655042 |
| 1500 | 0 | 05/09/15 17:53 | 204801189 |
| 1500 | 0 | 05/09/15 18:00 | 204747024 |
| 1500 | 0 | 05/09/15 18:11 | 204735267 |
| 1500 | 0 | 05/09/15 18:16 | 204966621 |
| 1500 | 0 | 05/09/15 18:20 | 204843678 |
| 1500 | 0 | 05/09/15 18:25 | 204931821 |
| 1500 | 0 | 05/09/15 18:32 | 204735267 |

Tabela 9 – Execução da matriz de odem 2000 pelo método sequencial.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DATA | TEMPO (MILISSEGUNDOS) |
| 2000 | 0 | 05/09/15 15:44 | 484082565 |
| 2000 | 0 | 05/09/15 15:53 | 478247389 |
| 2000 | 0 | 05/09/15 16:01 | 480237392 |
| 2000 | 0 | 05/09/15 16:10 | 479556307 |
| 2000 | 0 | 05/09/15 16:18 | 479689229 |
| 2000 | 0 | 05/09/15 16:31 | 479342916 |
| 2000 | 0 | 05/09/15 16:55 | 479548405 |
| 2000 | 0 | 05/09/15 17:04 | 479640614 |
| 2000 | 0 | 05/09/15 17:12 | 479334809 |
| 2000 | 0 | 05/09/15 17:20 | 479950013 |

Tabela 10 – Execução da matriz de odem 3000 pelo método sequencial.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DATA | TEMPO (MILISSEGUNDOS) |
| 3000 | 0 | 05/09/15 19:03 | 1613572110 |
| 3000 | 0 | 05/09/15 19:39 | 1617364883 |
| 3000 | 0 | 05/09/15 20:09 | 1615317931 |
| 3000 | 0 | 05/09/15 21:03 | 1613572110 |
| 3000 | 0 | 05/09/15 21:31 | 1617364883 |
| 3000 | 0 | 05/09/15 22:00 | 1615317931 |
| 3000 | 0 | 05/09/15 22:36 | 1613572110 |
| 3000 | 0 | 05/09/15 23:11 | 1617364883 |
| 3000 | 0 | 05/09/15 23:50 | 1615317931 |
| 3000 | 0 | 06/09/15 00:33 | 1613572110 |

# Anexo B – Execuções paralelas: *Pthreads*

Tabela 11 – Execução da matriz de odem 250 por *Pthreads*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 250 | 2 | 0,005844909 | 0,1193635 |
| 250 | 3 | 0,001697169 | 0,0338745 |
| 250 | 4 | 0,000699474 | 0,0229959 |
| 250 | 5 | 0,000175189 | 0,0193803 |
| 250 | 6 | 0,000337322 | 0,0216497 |
| 250 | 7 | 0,000246099 | 0,0218962 |
| 250 | 8 | 0,000244771 | 0,0217701 |
| 250 | 9 | 0,000204559 | 0,0240882 |
| 250 | 10 | 0,00020909 | 0,0243559 |
| 250 | 15 | 0,000197461 | 0,0277417 |
| 250 | 20 | 0,000341694 | 0,0374482 |
| 250 | 50 | 0,001177456 | 0,0962887 |

Tabela 12 – Execução da matriz de odem 500 por *Pthreads*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 500 | 2 | 0,030735347 | 2,001331 |
| 500 | 3 | 0,012561123 | 0,29431 |
| 500 | 4 | 0,001868237 | 0,0932098 |
| 500 | 5 | 0,000645895 | 0,0569816 |
| 500 | 6 | 0,000751651 | 0,0611769 |
| 500 | 7 | 0,000943233 | 0,0656162 |
| 500 | 8 | 0,000988676 | 0,0650518 |
| 500 | 9 | 0,00056376 | 0,0651392 |
| 500 | 10 | 0,000624722 | 0,0658769 |
| 500 | 15 | 0,000498003 | 0,0695445 |
| 500 | 20 | 0,000565597 | 0,0675954 |
| 500 | 50 | 0,001827839 | 0,1362438 |

Tabela 13 – Execução da matriz de odem 1000 por *Pthreads*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 1000 | 2 | 0,151144293 | 10,7570312 |
| 1000 | 3 | 0,15401344 | 4,0049701 |
| 1000 | 4 | 0,047888413 | 1,1855495 |
| 1000 | 5 | 0,002332534 | 0,1887689 |
| 1000 | 6 | 0,002525459 | 0,2018273 |
| 1000 | 7 | 0,001979472 | 0,2424278 |
| 1000 | 8 | 0,002088994 | 0,263045 |
| 1000 | 9 | 0,002350543 | 0,2341624 |
| 1000 | 10 | 0,001965605 | 0,2629905 |
| 1000 | 15 | 0,001965948 | 0,2543275 |
| 1000 | 20 | 0,001991757 | 0,2500758 |
| 1000 | 50 | 0,002732976 | 0,2992521 |

Tabela 14 – Execução da matriz de odem 1500 por *Pthreads*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 1500 | 2 | 0,356756841 | 21,7196997 |
| 1500 | 3 | 0,235295018 | 15,1989657 |
| 1500 | 4 | 0,385301527 | 6,7724884 |
| 1500 | 5 | 0,004570521 | 0,3959849 |
| 1500 | 6 | 0,005380396 | 0,4523447 |
| 1500 | 7 | 0,003714276 | 0,5043017 |
| 1500 | 8 | 0,006416579 | 0,5604127 |
| 1500 | 9 | 0,004841777 | 0,5634784 |
| 1500 | 10 | 0,004278872 | 0,5597507 |
| 1500 | 15 | 0,005361765 | 0,5881193 |
| 1500 | 20 | 0,004137842 | 0,5821005 |
| 1500 | 50 | 0,003772527 | 0,5719834 |

Tabela 15 – Execução da matriz de odem 2000 por *Pthreads*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 2000 | 2 | 0,276105207 | 31,9962087 |
| 2000 | 3 | 0,813639209 | 36,1758708 |
| 2000 | 4 | 0,676481274 | 13,9111002 |
| 2000 | 5 | 0,010868422 | 0,7129205 |
| 2000 | 6 | 0,006755462 | 0,744742 |
| 2000 | 7 | 0,007042875 | 0,8955244 |
| 2000 | 8 | 0,009752613 | 0,9208012 |
| 2000 | 9 | 0,009307143 | 0,936028 |
| 2000 | 10 | 0,008589566 | 0,9163774 |
| 2000 | 15 | 0,006416224 | 0,9771568 |
| 2000 | 20 | 0,007335715 | 1,0605202 |
| 2000 | 50 | 0,006444757 | 1,1164191 |

Tabela 16 – Execução da matriz de odem 3000 por *Pthreads*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 3000 | 2 | 2,534311793 | 384,0293968 |
| 3000 | 3 | 1,108038448 | 86,5759956 |
| 3000 | 4 | 1,425758076 | 76,3255225 |
| 3000 | 5 | 0,026958126 | 3,5252359 |
| 3000 | 6 | 0,013318767 | 1,7827783 |
| 3000 | 7 | 0,018810684 | 1,7038281 |
| 3000 | 8 | 0,012875233 | 1,7294243 |
| 3000 | 9 | 0,011639586 | 1,8574677 |
| 3000 | 10 | 0,013544456 | 1,8728928 |
| 3000 | 15 | 0,021190148 | 1,8790246 |
| 3000 | 20 | 0,01012595 | 2,0332112 |
| 3000 | 50 | 0,015153726 | 2,3587592 |

Tabela 17 – Execução da matriz de odem 4000 por *Pthreads*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 4000 | 2 | 6,702310421 | 1258,059046 |
| 4000 | 3 | 3,72768057 | 284,267874 |
| 4000 | 4 | 3,633183493 | 181,7880312 |
| 4000 | 5 | 0,112139899 | 13,462303 |
| 4000 | 6 | 0,037009648 | 7,3738155 |
| 4000 | 7 | 0,046051486 | 4,4916316 |
| 4000 | 8 | 0,031419668 | 3,0813353 |
| 4000 | 9 | 0,028077313 | 3,0283891 |
| 4000 | 10 | 0,021264158 | 2,9953613 |
| 4000 | 15 | 0,015352764 | 3,3155479 |
| 4000 | 20 | 0,035937811 | 3,2145234 |
| 4000 | 50 | 0,021273221 | 3,9147345 |

# Anexo C – Execuções paralelas: *OpenMP*

Tabela 18 – Execução da matriz de odem 250 pelo *OpenMP*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 250 | 2 | 0,00021646 | 0,0383757 |
| 250 | 3 | 0,000628303 | 0,0701944 |
| 250 | 4 | 0,008024047 | 0,19019 |
| 250 | 5 | 3,29274E-05 | 0,0243607 |
| 250 | 6 | 0,000131192 | 0,0247212 |
| 250 | 7 | 0,000101744 | 0,0260262 |
| 250 | 8 | 5,88187E-05 | 0,0264609 |
| 250 | 9 | 7,3506E-05 | 0,0268512 |
| 250 | 10 | 5,40789E-05 | 0,0270525 |
| 250 | 15 | 8,20358E-05 | 0,0297134 |
| 250 | 20 | 0,000327338 | 0,0325976 |
| 250 | 50 | 0,000209633 | 0,047059 |

Tabela 19 – Execução da matriz de odem 500 pelo *OpenMP*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 500 | 2 | 0,000141217 | 0,1436495 |
| 500 | 3 | 0,001230141 | 0,2488983 |
| 500 | 4 | 0,000556295 | 0,4114003 |
| 500 | 5 | 7,89174E-05 | 0,0826408 |
| 500 | 6 | 9,47682E-05 | 0,0831002 |
| 500 | 7 | 0,000107461 | 0,0870604 |
| 500 | 8 | 7,13836E-05 | 0,0891213 |
| 500 | 9 | 0,000142679 | 0,0904144 |
| 500 | 10 | 0,001177023 | 0,0960052 |
| 500 | 15 | 0,000152508 | 0,0970359 |
| 500 | 20 | 0,000106334 | 0,0988332 |
| 500 | 50 | 0,000123694 | 0,1160879 |

Tabela 20 – Execução da matriz de odem 1000 pelo *OpenMP*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 1000 | 2 | 0,000193063 | 0,3501861 |
| 1000 | 3 | 0,0003422 | 0,4631207 |
| 1000 | 4 | 0,000686309 | 0,6308179 |
| 1000 | 5 | 0,000220465 | 0,3148744 |
| 1000 | 6 | 0,000200477 | 0,3228454 |
| 1000 | 7 | 0,000220983 | 0,3227508 |
| 1000 | 8 | 0,000782032 | 0,3385973 |
| 1000 | 9 | 0,000709274 | 0,3405521 |
| 1000 | 10 | 0,000282927 | 0,3388063 |
| 1000 | 15 | 0,000266086 | 0,357204 |
| 1000 | 20 | 0,000274323 | 0,3644371 |
| 1000 | 50 | 0,000292011 | 0,3943974 |

Tabela 21 – Execução da matriz de odem 1500 pelo *OpenMP*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 1500 | 2 | 0,000489791 | 0,6887524 |
| 1500 | 3 | 0,002399717 | 0,8101095 |
| 1500 | 4 | 0,002543567 | 1,017067 |
| 1500 | 5 | 0,000262572 | 0,6802059 |
| 1500 | 6 | 0,000296371 | 0,6800495 |
| 1500 | 7 | 0,000456466 | 0,7088291 |
| 1500 | 8 | 0,00054612 | 0,7257934 |
| 1500 | 9 | 0,00194818 | 0,7511781 |
| 1500 | 10 | 0,000551944 | 0,7429973 |
| 1500 | 15 | 0,000746868 | 0,7631534 |
| 1500 | 20 | 0,011868508 | 0,8328276 |
| 1500 | 50 | 0,000560257 | 0,8162348 |

Tabela 22 – Execução da matriz de odem 2000 pelo *OpenMP*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 2000 | 2 | 0,003280427 | 1,1732628 |
| 2000 | 3 | 0,0008538 | 1,2708791 |
| 2000 | 4 | 0,007101083 | 1,5736235 |
| 2000 | 5 | 0,001138713 | 1,226486 |
| 2000 | 6 | 0,018626818 | 1,264386 |
| 2000 | 7 | 0,004655617 | 1,2793232 |
| 2000 | 8 | 0,004412968 | 1,2993697 |
| 2000 | 9 | 0,000829727 | 1,3053968 |
| 2000 | 10 | 0,00080161 | 1,3088045 |
| 2000 | 15 | 0,000775741 | 1,3321501 |
| 2000 | 20 | 0,000934982 | 1,3995464 |
| 2000 | 50 | 0,000960965 | 1,4036731 |

Tabela 23 – Execução da matriz de odem 3000 pelo *OpenMP*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 3000 | 2 | 0,039994628 | 2,6532659 |
| 3000 | 3 | 0,001255182 | 2,6221947 |
| 3000 | 4 | 0,009022777 | 3,0681601 |
| 3000 | 5 | 0,002562502 | 2,6994174 |
| 3000 | 6 | 0,002740349 | 2,7065048 |
| 3000 | 7 | 0,001913408 | 2,8164997 |
| 3000 | 8 | 0,001188279 | 2,8660061 |
| 3000 | 9 | 0,005271193 | 2,949759 |
| 3000 | 10 | 0,002717784 | 2,937724 |
| 3000 | 15 | 0,045620909 | 3,1358197 |
| 3000 | 20 | 0,002815769 | 3,1235518 |
| 3000 | 50 | 0,001771854 | 3,1249703 |

Tabela 24 – Execução da matriz de odem 4000 pelo *OpenMP*.

|  |  |  |  |
| --- | --- | --- | --- |
| ORDEM | THREADS | DESVIO PADRÃO | MÉDIA (MILISSEGUNDOS) |
| 4000 | 2 | 0,004068199 | 4,4473706 |
| 4000 | 3 | 0,002029364 | 4,5262573 |
| 4000 | 4 | 0,005399699 | 4,9987777 |
| 4000 | 5 | 0,00435111 | 4,7895119 |
| 4000 | 6 | 0,003169563 | 4,7986727 |
| 4000 | 7 | 0,001875792 | 4,9871919 |
| 4000 | 8 | 0,002362364 | 5,1206465 |
| 4000 | 9 | 0,004011034 | 5,2086168 |
| 4000 | 10 | 0,00516372 | 5,2305637 |
| 4000 | 15 | 0,004097391 | 5,3386516 |
| 4000 | 20 | 0,003916922 | 5,521651 |
| 4000 | 50 | 0,005531817 | 5,5659338 |

1. *Pthreads* são definidos como um conjunto de tipos de linguagem de programação C e chamadas de procedimento (LLNL, 2015). [↑](#footnote-ref-1)
2. *OpenMP* é um conjunto de diretivas do compilador e bibliotecas chamadas através de rotinas para expressar o paralelismo de memória compartilhada (OPENMP, 2015) [↑](#footnote-ref-2)
3. *MPI* é uma *API* padronizada normalmente utilizada para computação paralela e/ou distribuída. [↑](#footnote-ref-3)