UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação - ICMC-USP

Fabio Alves Martins Pereira (NUSP 7987435)

Naylor Garcia Bachiega (NUSP 5567669)

Trabalho 1: algoritmo sequencial e paralelo utilizando o método de Jacobi-Richardson

SÃO CARLOS

2015

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

1 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], *OpenMP* [2]e *MPI* [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

---

[1] *Pthreads* são definidos como um conjunto de tipos de linguagem de programação C e chamadas de procedimento (LLNL, 2015).

[2] *OpenMP* é um conjunto de diretivas do compilador e bibliotecas chamadas através de rotinas para expressar o paralelismo de memória compartilhada (OPENMP, 2015)

[3] *MPI* é uma *API* padronizada normalmente utilizada para computação paralela e/ou distribuída.

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.

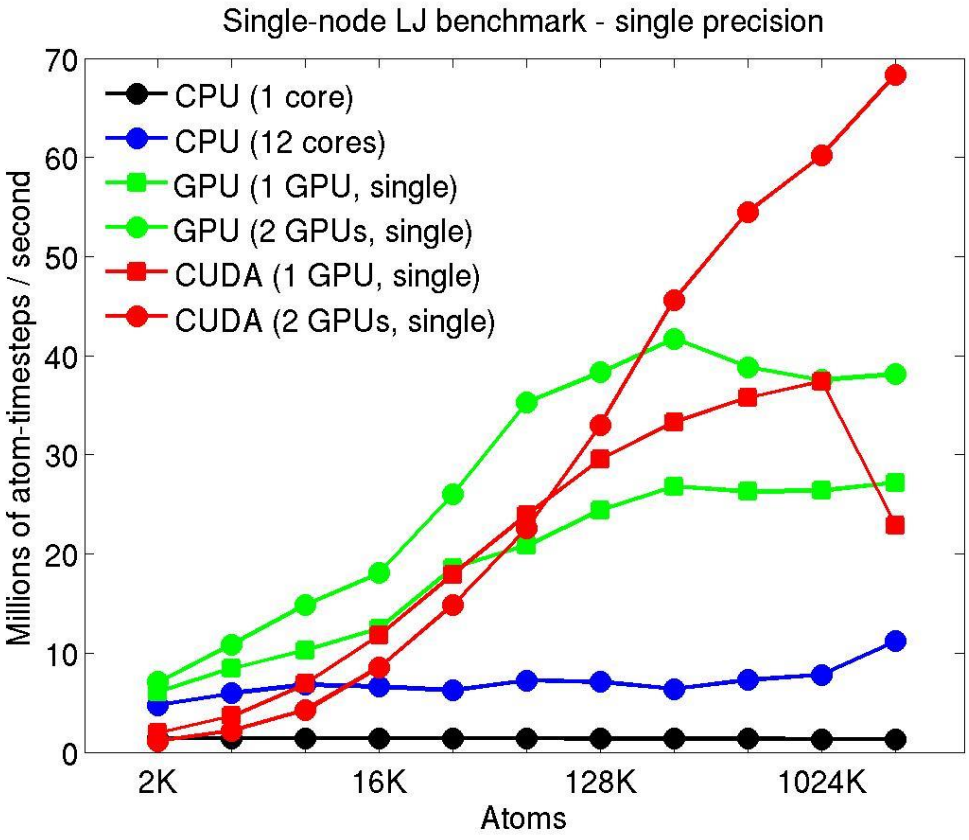

Figura 1 – Benchmark em um único computador (LAMMPS, 2015).

1.2 OBJETIVOS

Tendo em vista os benefícios da paralelização de algoritmos, esse trabalho mostra o desenvolvimento de dois algoritmos para o processamento de matrizes lineares, utilizando o método de Jacobi-Richarson. Após o desenvolvimento, os resultados serão demonstrados através de gráficos e tabelas.

2 MÉTODO DE JACOBI-RICHARSON

Métodos numéricos para solução de sistemas de equações lineares são divididos principalmente em dois grupos (YONA, 2010):

• Métodos Exatos: são aqueles que forneceriam a solução exata, se não fossem os erros de arredondamento, com um número finito de operações.

• Métodos Iterativos: são aqueles que permitem obter a solução de um sistema com uma dada precisão através de um processo infinito convergente e que possui um erro de truncamento.

O método iterativo de Jacobi-Richardson se baseia inicialmente numa verificação de convergência, ou seja, verifica-se se a matriz é estritamente diagonalmente dominante. Se ela atender o critério de convergência, define-se a margem de erro máximo (truncamento) que o resultado terá. Assim, o método inicia-se através da solução inicial (no caso de um sistema linear, zero é uma das soluções possíveis), em seguida, calcula-se o resultado e ocorre a substituição no vetor resultado na próxima iteração, e assim em diante até que se tenha a solução que atenda a margem de erro (YONA, 2010).

2.2 DESCRIÇÃO DO MÉTODO

Dado um sistema quadrado de equações lineares n: $Ax = b$, onde:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Então, A pode ser decomposto numa diagonal D, e o restante de R:

$$A = D + R \qquad \text{where} \qquad D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \text{ and } R = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}.$$

A solução é, então, obtida iterativamente:

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - R\mathbf{x}^{(k)}),$$

Onde $\mathbf{x}^{(k)}$ é a aproximação de ordem *k* ou iteração de $\mathbf{x}$ e $\mathbf{x}^{(k+1)}$ é o próximo ou a iteração *k* + 1 de $\mathbf{x}$. A fórmula base é:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

2.3 ALGORITMO

Para o trabalho, foi utilizado o algoritmo básico, conforme pode ser visualizado:

Inicialmente, é estimado um valor inicial para $x^{(0)}$ para a solução $k = 0$

enquanto erro não é alcançado

para i := 1 até n faça

$\sigma = 0$

para j := 1 até n faça

se j ≠ i então

$\sigma = \sigma + a_{ij} x_j^{(k)}$

fim se

fim (j-loop)

$x_i^{(k+1)} = \frac{(b_i - \sigma)}{a_{ii}}$

fim (i-loop)

$k = k + 1$

repete (enquanto erro não é alcançado)

3 PTHREADS

Pthread é uma maneira simples e eficaz de criar uma aplicação paralelizada. Quando uma *thread* é criada usando **pthread_create**, a *thread* original e a nova compartilham da mesma base de código e a mesma memória - é como fazer duas chamadas de função ao mesmo tempo (TIM, 2010).

Todos os programas em C usando *pthreads* precisam incluir o arquivo de *header* pthread.h (#include <pthread.h>). No sistema operacional Ubuntu Desktop 15.04 é necessário instalar um pacote através do *apt-get* (ferramenta de instalação e atualização de pacotes):

- sudo apt-get install build-essential
- sudo apt-get install libpthread-stubs0-dev

3.1 FUNCIONAMENTO

Há quatro etapas para a criação de um programa básico utilizando Pthreads (TIM, 2010):

- Definir a variável thread de referência: a variável do tipo pthread_t é uma forma de referenciar *threads*. É preciso haver uma variável pthread_t na existência de cada segmento que está sendo criado. Algo como pthread_t thread0.

- Criar um ponto de entrada para a *thread*: ao criar a *thread* usando *pthreads*, é necessário apontá-la para uma função para ela iniciar a execução. A função deve retornar void * e tomar um único argumento void *. Por exemplo, para que a função pegue um argumento inteiro, é necessário passar o endereço do inteiro. Um exemplo de função seria: void * my_entry_function (void * param);

- Criar a thread: uma vez que a variável pthread_t foi definida e a função de ponto de entrada criada, deve criar o segmento usando pthread_create. Este método tem quatro argumentos: um ponteiro para a variável pthread_t, os atributos extras, um ponteiro para a função a ser chamada e o ponteiro que está sendo passado como argumento para a função. Esta chamada será algo parecido com pthread_create (&thread0, NULL, my_entry_function, e &parameter);

4 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.

4.1 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/trabalho-prog-conc**

4.2 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	ASUS P6T
Sistema Operacional	Linux Mint 17.2 Cinnamon 64bit
Kernel do Linux	3.16.038generic
Processador	Intel(TM) Core(TM) i7 930 – 2.80 GHz (4cores físicos e 8 lógicos)
Memória RAM	6 GB Markvision DDR3 1333Mhz
HardDrive	Western Digital 1 TB – 7200 rpm

4.3 PROGRAMA: TRABALHO-PROG-CONC

O programa principal foi criado para listar as matrizes disponíveis no diretório “matrizes/” e disponibilizar a opção de escolha para o usuário. O usuário, após escolher a matriz, pode especificar se a execução será sequencial ou paralela. No caso da execução paralela, ainda é possível digitar a quantidade desejada de *thread* (Figura 2).

O sistema 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.
- **funcao.c**: funções importantes para o sistema, como enviar o resultados para a tela, imprimir os resultados no arquivo, checar as matrizes disponíveis, carregar a matriz, limpar a memória após execução e checar os critérios de parada.
- **paralelo.c**: faz as chamadas das threads para execução do algoritmo paralelo.
- **sequencial.c**: instruções do algoritmo sequencial.
- **timer.c**: função para imprimir o tempo de execução dos algoritmos.

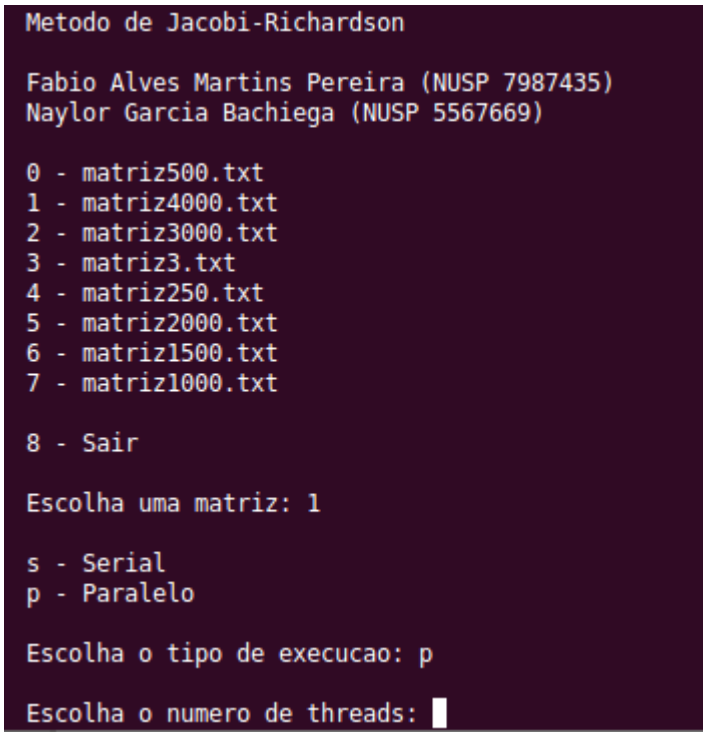

Figura 2 – Tela inicial do programa.

4.4 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
==================

Exemplo de como utilizar o programa.

### Dependências
1. Necessária instalação da libpthread:

sudo apt-get install build-essential
sudo apt-get install libpthread-stubs0-dev

### Instalação

1. Faça o clone deste projeto:
      git clone https://github.com/naylor/trabalho-prog-conc.git`

2. Entre na pasta do projeto

3. Rode o comando "make"


### Executando a aplicação

1. ./trabalho-prog-conc

2. Os resultados são gravados na pasta: resultados

3. Matrizes disponíveis na pasta: matrizes
   Obs.: Arquivos de matrizes são carregados automaticamente,
      assim, mais arquivos podem ser adicionados.
```

4.5 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++
SRCDIR=src/
SRCEXT=c
OBJEXT=o
PROG=trabalho-prog-conc

# FLAGS NECESSARIAS
```

```
# PARA COMPILACAO
CFLAGS=-Wall -Wextra
LIB=-lpthread

#------------------------------
# CARREGA AUTOMATICAMENTE OS
# ARQUIVOS .C E .H
#------------------------------
SOURCES=$(wildcard $(SRCDIR)*.c)
HEADERS=$(wildcard $(SRCDIR)*.h)


all: $(PROG)

$(PROG): $(SOURCES:.c=.o)
	$(G) -o $@ $^ $(LIB)

%.o: %.c $(HEADERS)
	$(CC) -g -c $< -o $@

clean:
	rm -f $(SRCDIR)*.o
	rm -f $(PROG)
```

5 RESULTADOS E DISCUSSÕES

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

5.1 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	0,96	0,0018
500	7,49	0,0026
1000	58,3	0,9739
1500	174,08	3,4188
2000	479,97	0,1538
3000	1615,24	0,1657
4000	➢ 32400	NAN

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.

A Figura 3 mostra como o tempo aumenta, consideravelmente, quando a ordem da matriz é acrescida.

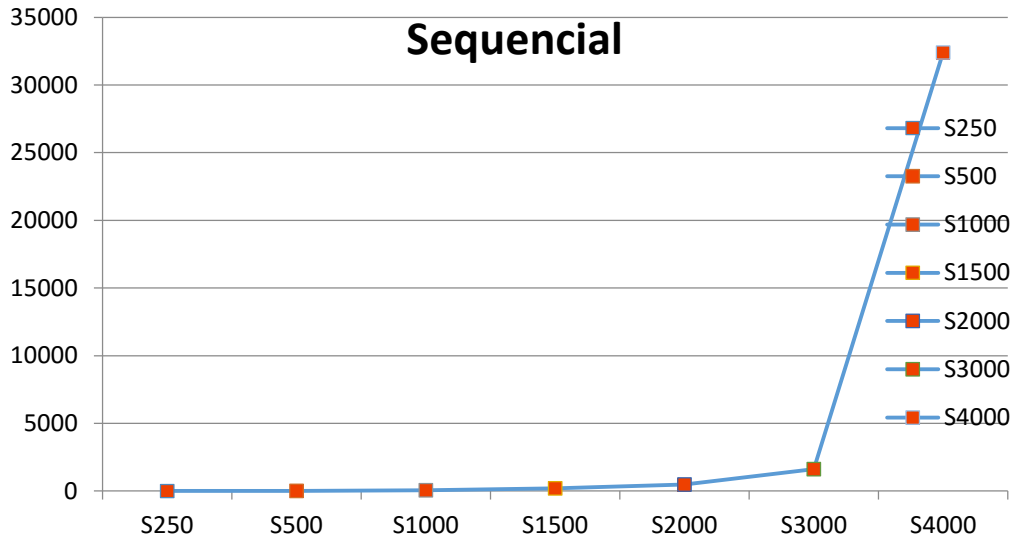

Figura 3 – Execução das matrizes pelo método sequencial.

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

5.2 ALGORITMO PARALELO

Os mesmos testes foram realizados para o algoritmo paralelo, porém utilizando *threads*. Foi observado que, em um determinado momento, aumentar a quantidade de *threads* também aumentava o tempo de execução. Sendo assim, na Tabela 3são 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 PELO MÉTODO PARALELO.

ORDEM ***n x n***	**THREADS**	**MÉDIA (segundos)**	**DESVIO (segundos)**
250	4	0,02931	0,00069
500	4	0,06118	0,00047
1000	5	0,19206	0,00275
1500	5	0,4354	0,00531
2000	5	0,69776	0,01427
3000	6	1,56333	0,02603
4000	8	2,80552	0,02188

Na Figura 4 são apresentados os resultados com os tempos de execução de cada ordem de matriz e a quantidade de *threads* que proporcionou o melhor resultado.

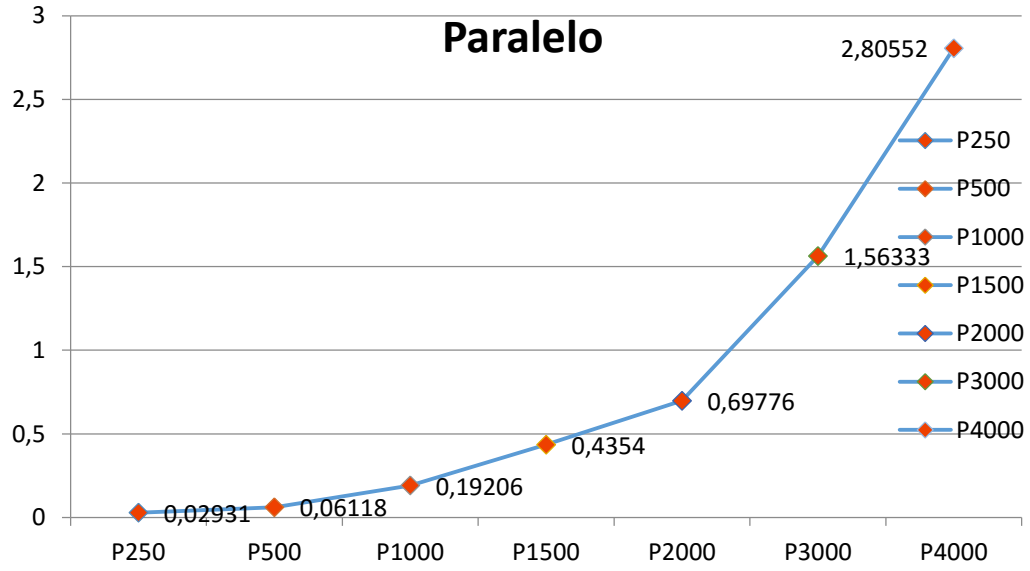

Figura 4 – Execução das matrizes pelo método paralelo.

5.2.1 PROCESSAMENTO DAS MATRIZES

Na Figura 5 é possível observar o comportamento da execução de acordo com a quantidade de *threads* informada ao programa.

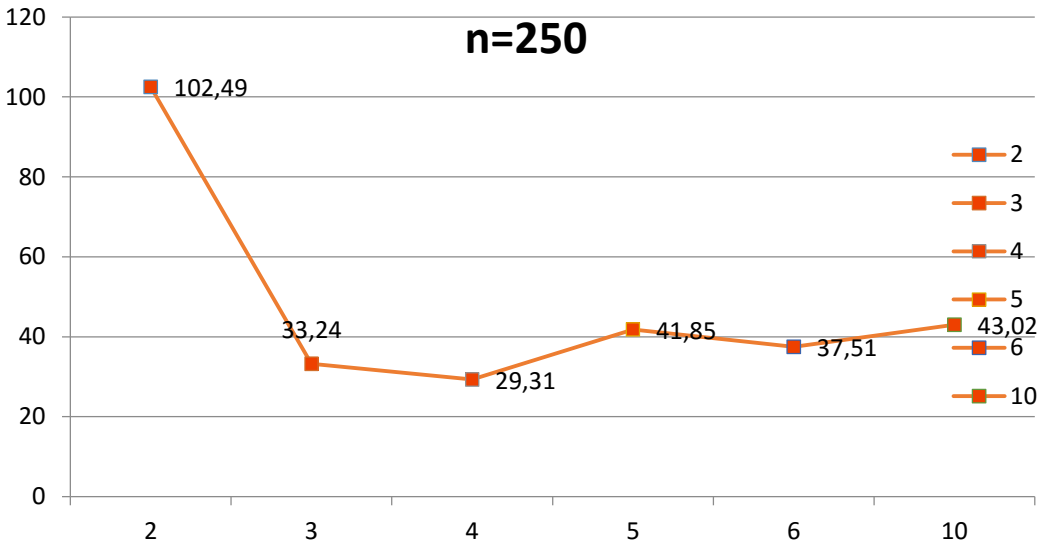

Figura 5 – Execução da matriz de ordem n=250 pelo método paralelo.

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.

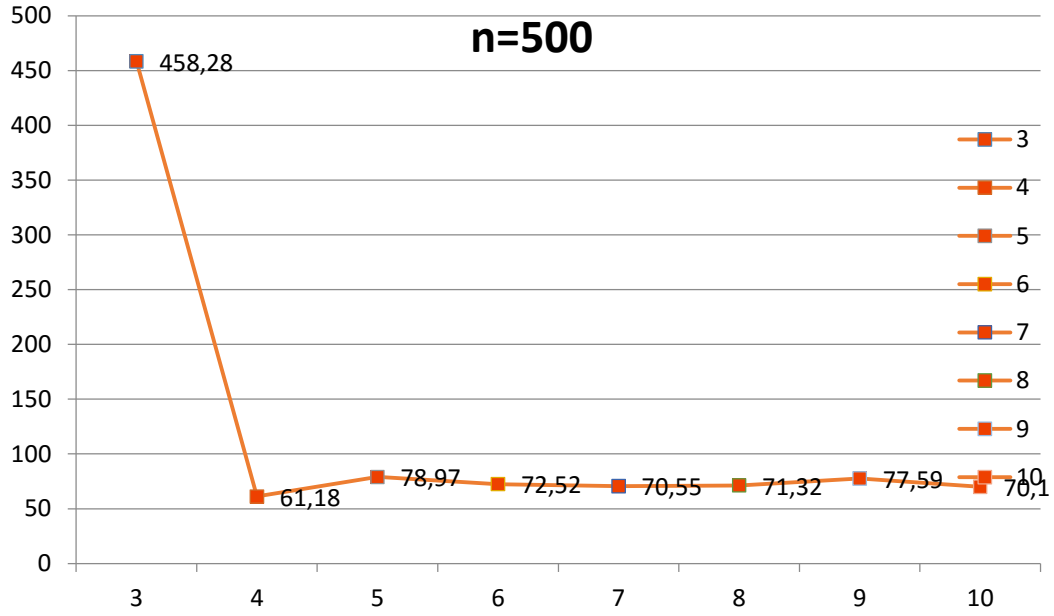

Figura 6 – Execução da matriz de ordem n=500 pelo método paralelo.

Execução da matriz de ordem n=1000.

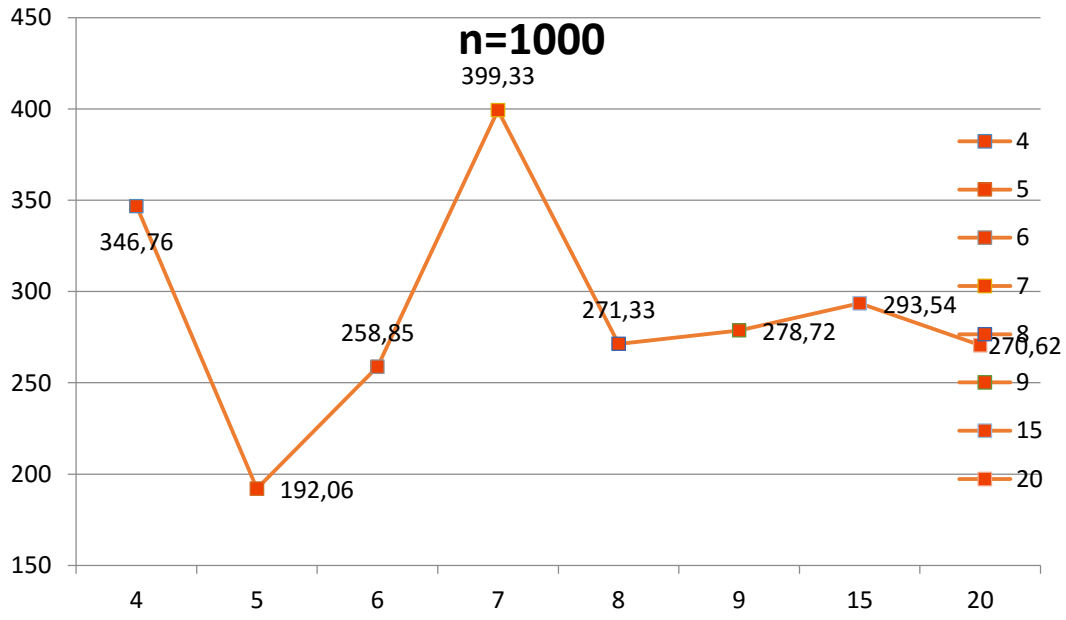

Figura 7 – Execução da matriz de ordem n=1000 pelo método paralelo.

Execução da matriz de ordem n=1500.

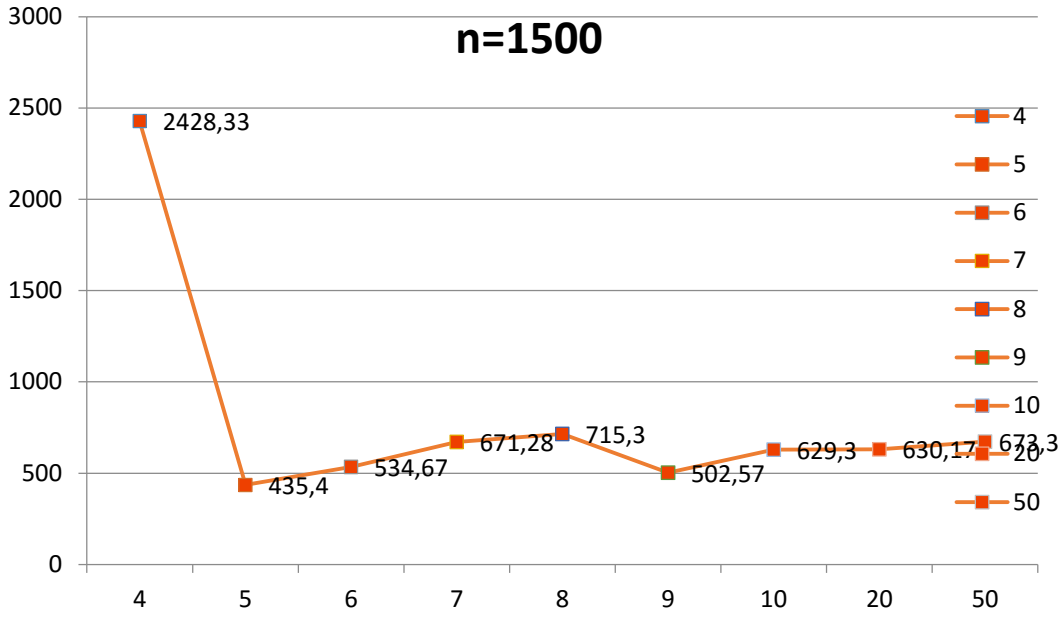

Figura 8 – Execução da matriz de ordem n=1500 pelo método paralelo.

Execução da matriz de ordem n=2000.

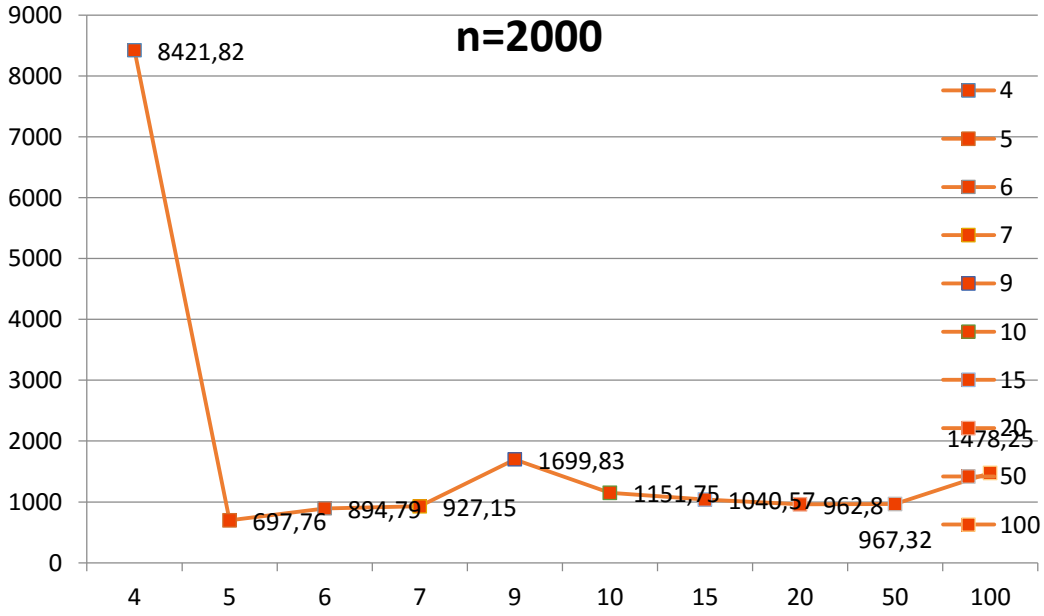

Figura 9 – Execução da matriz de ordem n=2000 pelo método paralelo.

Execução da matriz de ordem n=3000.

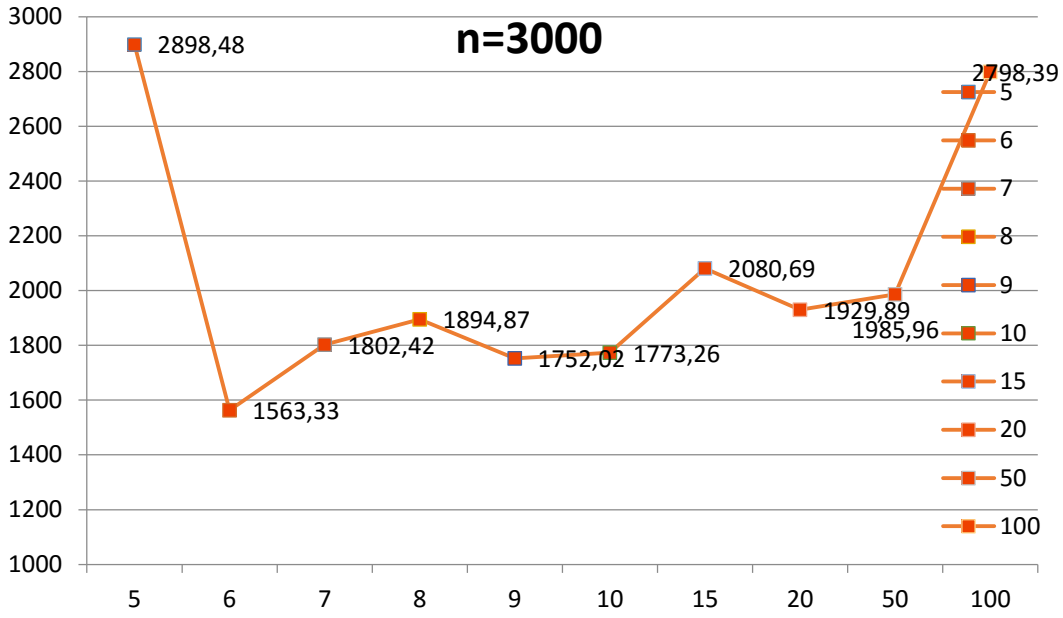

Figura 10 – Execução da matriz de ordem n=3000 pelo método paralelo.

Execução da matriz de ordem n=4000.

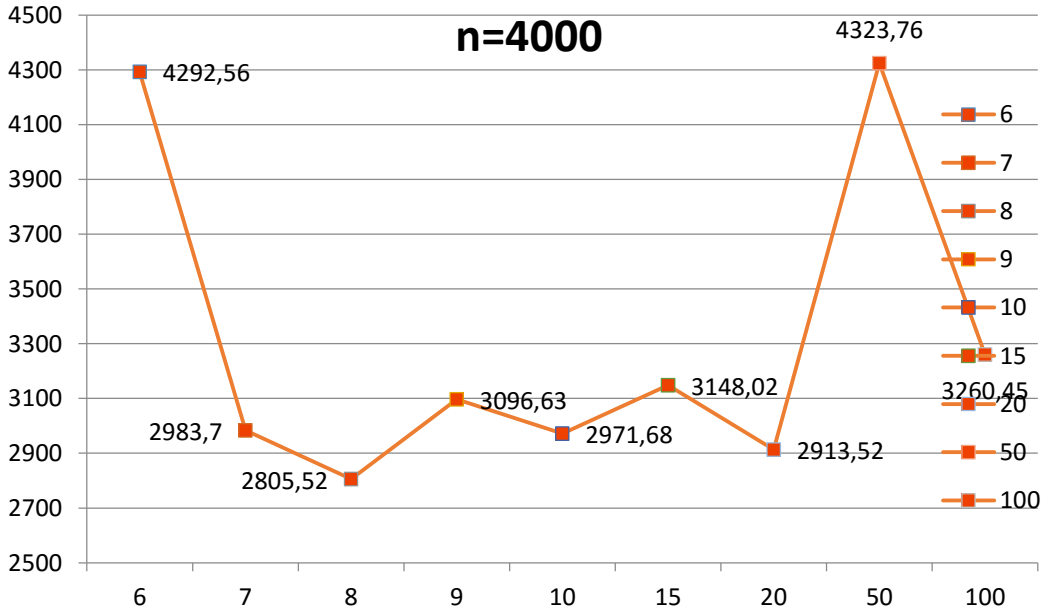

Figura 11 – Execução da matriz de ordem n=4000 pelo método paralelo.

5.2.2 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 12 mostra essa relação.

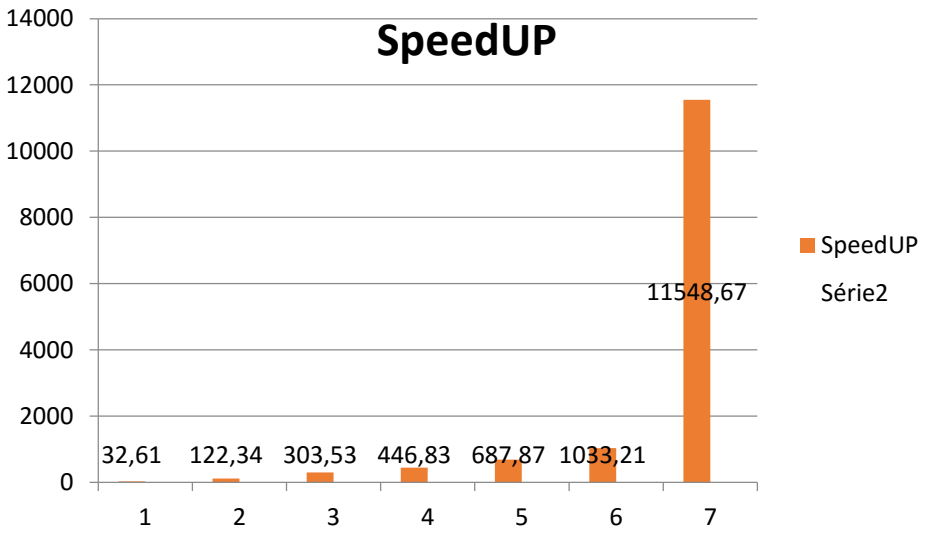

Figura 12 – SpeedUp entre os algoritmos paralelo e sequencial.

5.2.3 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*, 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.

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

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

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.

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.

RICARDO, Luis; PAULINO, Diogo; CARVALHO, Paulo. **Paralelização do algoritmo do Método Jacobi**. Disponível em: <https://drive.google.com/file/d/0B639uUhZ62fgV012UkZvSDdZeWs/view>. Acesso em: 30 ago. 2015.

TIM, C. ***Pthreads in C – a minimal working example***. Disponível em: <http://timmurphy.org/2010/05/04/pthreads-in-c-a-minimal-working-example/>. Acesso em: 30 ago. 2015.

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

ANEXO A – EXECUÇÕES SEQUENCIAIS

TABELA 4 – EXECUÇÃO DA MATRIZ DE ODEM 250 PELO MÉTODO SEQUENCIAL.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
250	0	05/09/15 14:35	937,355
250	0	05/09/15 14:37	944,146
250	0	05/09/15 14:41	969,399
250	0	05/09/15 14:45	941,395
250	0	05/09/15 14:45	944,782
250	0	05/09/15 14:45	966,939
250	0	05/09/15 14:45	973,535
250	0	05/09/15 14:46	983,961
250	0	05/09/15 14:46	934,186
250	0	05/09/15 14:46	961,089

TABELA 5 – EXECUÇÃO DA MATRIZ DE ODEM 500 PELO MÉTODO SEQUENCIAL.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
500	0	05/09/15 14:57	7485,075
500	0	05/09/15 14:57	7476,078
500	0	05/09/15 15:17	7471,182
500	0	05/09/15 15:17	7474,005
500	0	05/09/15 15:17	7462,591
500	0	05/09/15 15:18	7481,204
500	0	05/09/15 15:18	7555,521
500	0	05/09/15 15:18	7480,306
500	0	05/09/15 15:18	7481,225
500	0	05/09/15 15:19	7478,332

TABELA 6 – EXECUÇÃO DA MATRIZ DE ODEM 1000 PELO MÉTODO SEQUENCIAL.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
1000	0	05/09/15 17:21	30592,12
1000	0	05/09/15 17:23	61448,066
1000	0	05/09/15 17:25	60549,921
1000	0	05/09/15 17:26	61599,126
1000	0	05/09/15 17:32	61432,035
1000	0	05/09/15 17:34	61482,45
1000	0	05/09/15 17:35	61447,323
1000	0	05/09/15 17:37	61587,433
1000	0	05/09/15 17:39	61402,718
1000	0	05/09/15 17:40	61413,002

TABELA 7 – EXECUÇÃO DA MATRIZ DE ODEM 1500 PELO MÉTODO SEQUENCIAL.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
1500	0	05/09/15 17:42	102248,984
1500	0	05/09/15 17:46	204817,838
1500	0	05/09/15 17:49	204655,042

1500	0	05/09/15 17:53	204801,189
1500	0	05/09/15 18:00	204747,024
1500	0	05/09/15 18:11	204735,267
1500	0	05/09/15 18:16	204966,621
1500	0	05/09/15 18:20	204843,678
1500	0	05/09/15 18:25	204931,821
1500	0	05/09/15 18:32	204735,267

TABELA 8 – EXECUÇÃO DA MATRIZ DE ODEM 2000 PELO MÉTODO SEQUENCIAL.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
2000	0	05/09/15 15:44	484082,565
2000	0	05/09/15 15:53	478247,389
2000	0	05/09/15 16:01	480237,392
2000	0	05/09/15 16:10	479556,307
2000	0	05/09/15 16:18	479689,229
2000	0	05/09/15 16:31	479342,916
2000	0	05/09/15 16:55	479548,405
2000	0	05/09/15 17:04	479640,614
2000	0	05/09/15 17:12	479334,809
2000	0	05/09/15 17:20	479950,013

TABELA 9 – EXECUÇÃO DA MATRIZ DE ODEM 3000 PELO MÉTODO SEQUENCIAL.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
3000	0	05/09/15 19:03	1613572,11
3000	0	05/09/15 19:39	1617364,88
3000	0	05/09/15 20:09	1615317,93
3000	0	05/09/15 21:03	1613572,11
3000	0	05/09/15 21:31	1617364,88
3000	0	05/09/15 22:00	1615317,93
3000	0	05/09/15 22:36	1613572,11
3000	0	05/09/15 23:11	1617364,88
3000	0	05/09/15 23:50	1615317,93
3000	0	06/09/15 00:33	1613572,11

ANEXO B – EXECUÇÕES PARALELAS

TABELA 10 – EXECUÇÃO DA MATRIZ DE ODEM 250 PELO MÉTODO PARALELO.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
250	4	05/09/15 14:49	29263
250	4	05/09/15 14:49	42333
250	4	05/09/15 14:49	25888
250	4	05/09/15 14:49	23302
250	4	05/09/15 14:49	19636
250	4	05/09/15 14:49	37959
250	4	05/09/15 14:52	31745
250	4	05/09/15 14:52	23770
250	4	05/09/15 14:53	28850
250	4	05/09/15 14:53	30316

TABELA 11 – EXECUÇÃO DA MATRIZ DE ODEM 500 PELO MÉTODO PARALELO.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
500	4	06/09/15 08:51	57358
500	4	06/09/15 08:52	66304
500	4	06/09/15 08:52	58203
500	4	06/09/15 08:54	54954
500	4	06/09/15 08:54	62135
500	4	06/09/15 08:54	58800
500	4	06/09/15 08:54	56323
500	4	06/09/15 08:55	67591
500	4	06/09/15 08:55	63455
500	4	06/09/15 08:55	66640

TABELA 12 – EXECUÇÃO DA MATRIZ DE ODEM 1000 PELO MÉTODO PARALELO.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
1000	5	06/09/15 08:58	170293
1000	5	06/09/15 08:58	188233
1000	5	06/09/15 08:59	181265
1000	5	06/09/15 08:59	215563
1000	5	06/09/15 08:59	208618
1000	5	06/09/15 09:00	216012
1000	5	06/09/15 09:00	218414
1000	5	06/09/15 09:00	202051
1000	5	06/09/15 09:00	191307
1000	5	06/09/15 09:00	128827

TABELA 13 – EXECUÇÃO DA MATRIZ DE ODEM 1500 PELO MÉTODO PARALELO.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
1500	5	06/09/15 09:01	419224
1500	5	06/09/15 09:02	479914
1500	5	06/09/15 09:02	467649

1500	5	06/09/15 09:02	352117
1500	5	06/09/15 09:02	480725
1500	5	06/09/15 09:03	511415
1500	5	06/09/15 09:04	438674
1500	5	06/09/15 09:05	438393
1500	5	06/09/15 09:05	412802
1500	5	06/09/15 09:05	353072

TABELA 14 – EXECUÇÃO DA MATRIZ DE ODEM 2000 PELO MÉTODO PARALELO.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
2000	5	06/09/15 09:06	652362
2000	5	06/09/15 09:07	388140
2000	5	06/09/15 09:07	689230
2000	5	06/09/15 09:07	689324
2000	5	06/09/15 09:07	687726
2000	5	06/09/15 09:08	887246
2000	5	06/09/15 09:09	900173
2000	5	06/09/15 09:09	652822
2000	5	06/09/15 09:09	766986
2000	5	06/09/15 09:10	663559

TABELA 15 – EXECUÇÃO DA MATRIZ DE ODEM 3000 PELO MÉTODO PARALELO.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
3000	6	06/09/15 09:15	1478927
3000	6	06/09/15 09:15	2003301
3000	6	06/09/15 09:18	1735997
3000	6	06/09/15 09:18	1564277
3000	6	06/09/15 09:19	1511238
3000	6	06/09/15 09:22	1687891
3000	6	06/09/15 09:22	1772268
3000	6	06/09/15 09:22	1233317
3000	6	06/09/15 09:22	1107470
3000	6	06/09/15 09:23	1538589

TABELA 16 – EXECUÇÃO DA MATRIZ DE ODEM 4000 PELO MÉTODO PARALELO.

ORDEM	THREADS	DATA	TEMPO (MILISSEGUNDOS)
4000	8	06/09/15 09:31	2862776
4000	8	06/09/15 09:31	2696047
4000	8	06/09/15 09:31	3040590
4000	8	06/09/15 09:31	2807275
4000	8	06/09/15 09:31	2683767
4000	8	06/09/15 09:34	3259406
4000	8	06/09/15 09:34	2849113
4000	8	06/09/15 09:34	2470979
4000	8	06/09/15 09:35	2688591
4000	8	06/09/15 09:35	2696606