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

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.

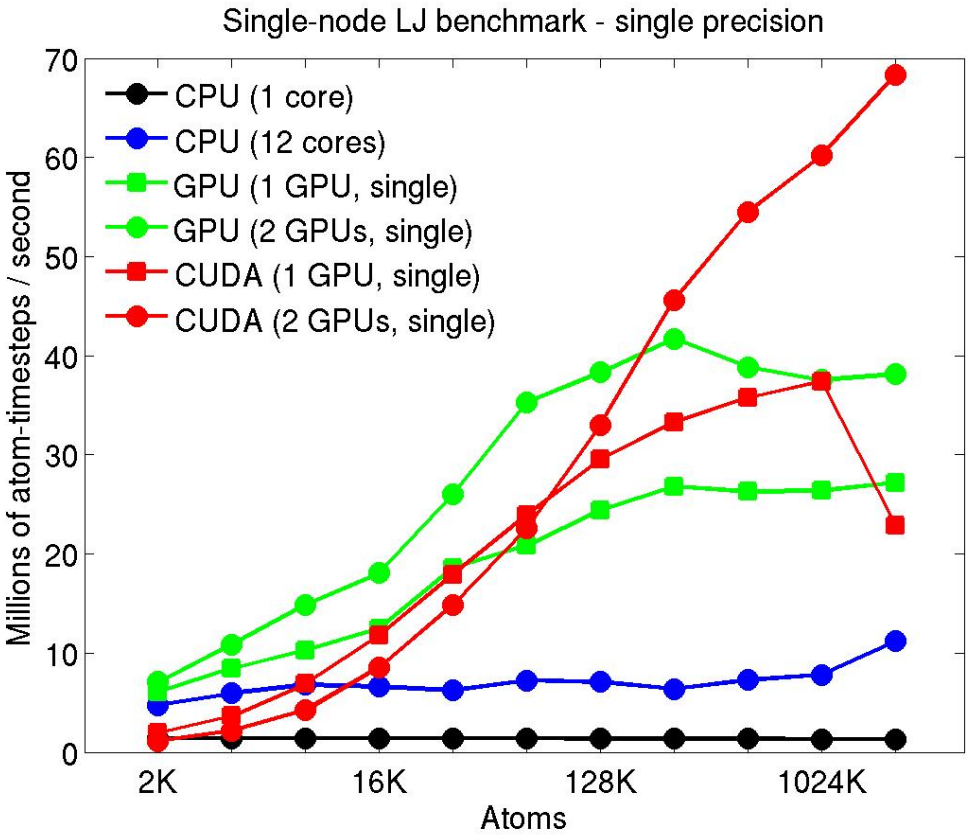

Figura 1 – Benchmark em um único computador (LAMMPS, 2015).

1.2 OBJETIVOS

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

2 O FORMATO PPM E PGM

Os formatos PPM (*Portable Pixmap Format*) e PGM (*Portable Graymap Format*) são arquivos de imagem relativamente simples. Deve notar-se que estes formatos são notoriamente ineficientes e altamente redundantes, também não utilizam algoritmos de compressão de dados. Além disso, esses formatos permitem muito pouca informação sobre a imagem além da cor base. No entanto, é relativamente fácil escrever programas para analisar e processar esses arquivos, o que os tornam interessantes (NETBPM, 2015).

2.1 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.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
  const int dimx = 800, dimy = 800;
  int i, j;
  FILE *fp = fopen("first.ppm", "wb"); /* b - binary mode */
  (void) fprintf(fp, "P6\n%d %d\n255\n", dimx, dimy);
  for (j = 0; j < dimy; ++j)
  {
    for (i = 0; i < dimx; ++i)
    {
      static unsigned char color[3];
      color[0] = i % 256;  /* red */
      color[1] = j % 256;  /* green */
      color[2] = (i * j) % 256;  /* blue */
      (void) fwrite(color, 1, 3, fp);
    }
  }
  (void) fclose(fp);
  return EXIT_SUCCESS;
}
```

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

2.2 O ARQUIVO PGM

Este formato é idêntico ao PPM, a diferença é que o PGM armazena informação em escalas de cinza, ou seja, um valor por pixel em vez de 3 (R, G, B). Há outra diferença na secção de cabeçalho, os identificadores que são "P2" e "P5", estes

correspondem a ASCII e forma binária dos dados, respectivamente. Segundo Netbpm (2015), cada imagem PGM consiste em:

```
P2
# feep.pgm
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Figura 4 – Exemplo de imagem PGM (NETBPM, 2015).

3 PARALELISMO

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

3.1 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 5:

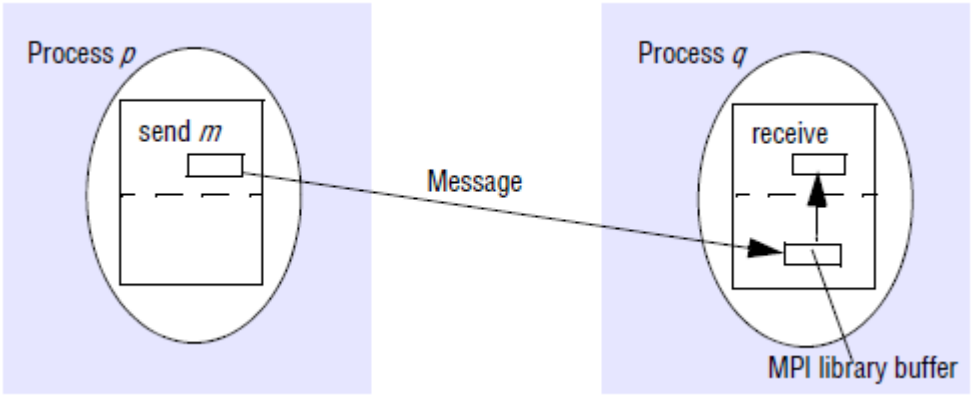

Figura 5 – Comunicação ponto-a-ponto MPI (COULOURIS et al, 2005).

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).

3.1.1 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.

```
/* program hello */
/* Adapted from mpihello.f by drs */

#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
  int rank;
  char hostname[256];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  gethostname(hostname,255);

  printf("Hello world!  I am process number: %d on host %s\n", rank, hostname);

  MPI_Finalize();

  return 0;
}
```

Figura 6 – Exemplo código MPI (HPC, 2015).

Conforme demonstrado na Figura 6, 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.

3.2 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)

3.2.1 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 7 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)

```
#include 'omp.h'
void main()
{
#pragma omp parallel
  {
    int ID = omp_get_thread_num()
    printf('Hello(%d) ',ID);
    printf('World(%d) \n',ID);
  }
}
```

Figura 7 – 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.

3.3 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 8 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.

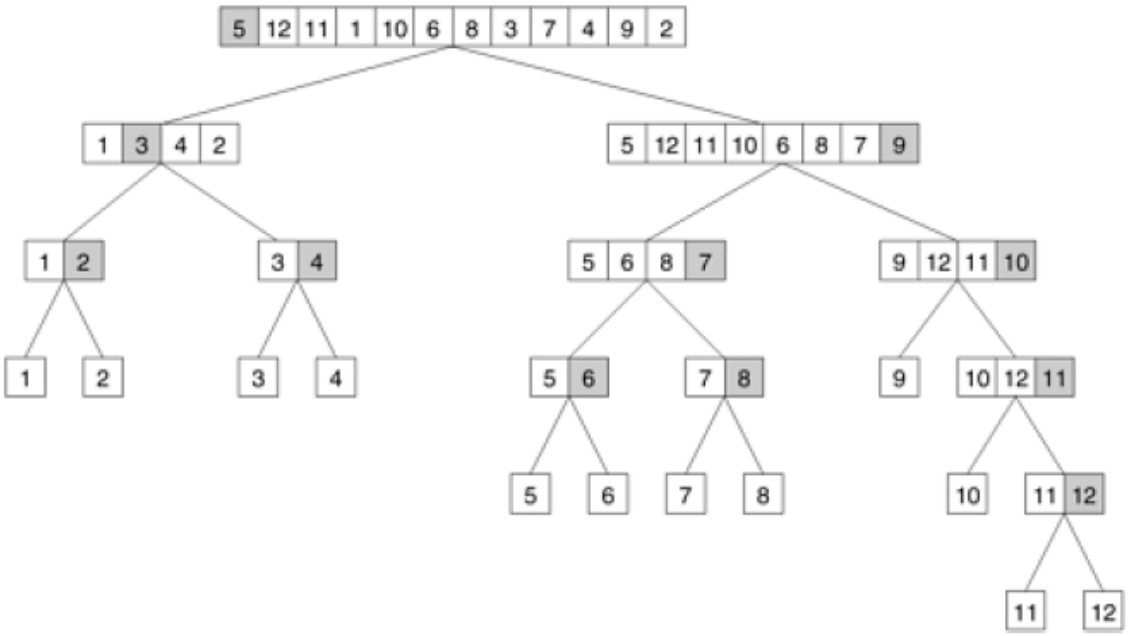

Figura 8 – Exemplo de decomposição: *Quicksort* (GRAMA et al, 2003).

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.

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/Trab02-Grupo11-AePos**

4.2 HARDWARE

Para esse trabalho foi utilizado do *Cluster* Cosmos do LaSDPC (*Laboratory of Distributed Systems and Concurrent Programming*) e o seguintes computadores:

TABELA 1 – COMPUTADORES UTILIZADOS NO *CLUSTER*.

Hostname	**IP**	**Hostname**	**IP**
frontend	10.1.1.200	iris	10.1.1.10
flora	10.1.1.20	tetis	10.1.1.30
adria	10.1.1.40	ceres	10.1.1.50
doris	10.1.1.60	hebe	10.1.1.70
irene	10.1.1.80	palas	10.1.1.90
maia	10.1.1.100	isis	10.1.1.110
lidia	10.1.1.120	nisa	10.1.1.130
rosa	10.1.1.140	alice	10.1.1.150

4.3 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 para facilitar a leitura. Na versão sequencial, conforme pode ser observado na Figura 9, o usuário pode selecionar qual imagem deseja aplicar.

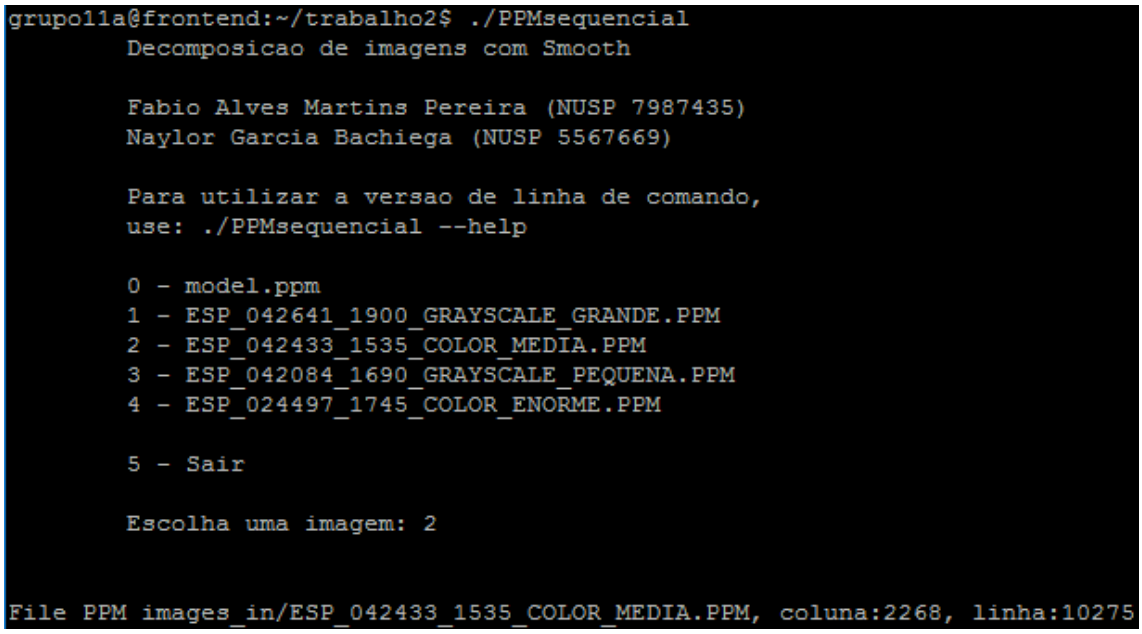

Figura 9 – Tela inicial do programa PPMsequencial.

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.

4.4 PROGRAMA: PPMPARALELO

Na versão paralela, o sistema funciona somente por linha de comando. A Figura 10, mostra a tela de ajuda do programa que é informada ao usuário e quais as opções possíveis de lançamento.

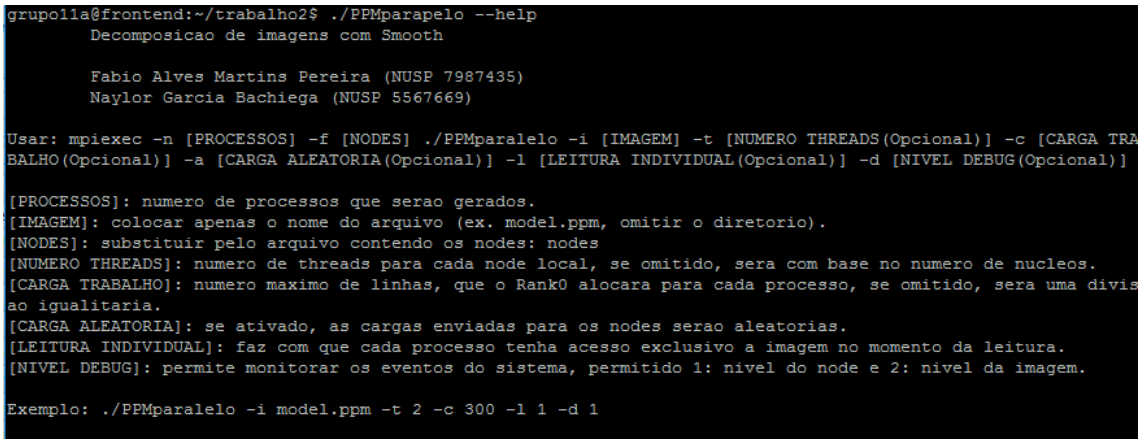

Figura 10 – Tela de ajuda do programa PPMparalelo.

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.

4.5 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.

```
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/Trab02-Grupo11-AePos

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
ALEATÓRIA] -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 núcleos.
  * [CARGA TRABALHO]: número máximo de linhas, que o Rank0
alocará para cada processo, se omitido, será uma divisão
igualitária.
  * [CARGA ALEATÓRIA]: se ativado, as cargas enviadas para os
nodes serão aleatórias.
  * [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 processadas com Smooth na pasta: images_out
```

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)
```

4.6 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. O algoritmo foi construído de forma a enviar cargas de trabalhos para os processos, estimulando a concorrência de recursos, conforme demonstrado na Figura 11, assim, enquanto um processo aplica o filtro, o outro pode estar gravando ou lendo a imagem.

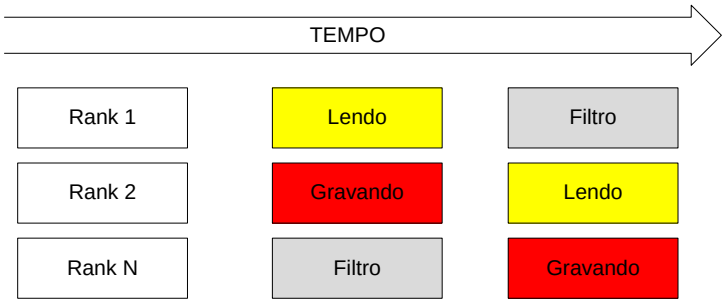

Figura 11 – Concorrência por recursos.

Dessa forma, o algoritmo desenvolvido utiliza da decomposição de dados, 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. A Figura 12 demonstra o funcionamento do algoritmo paralelo.

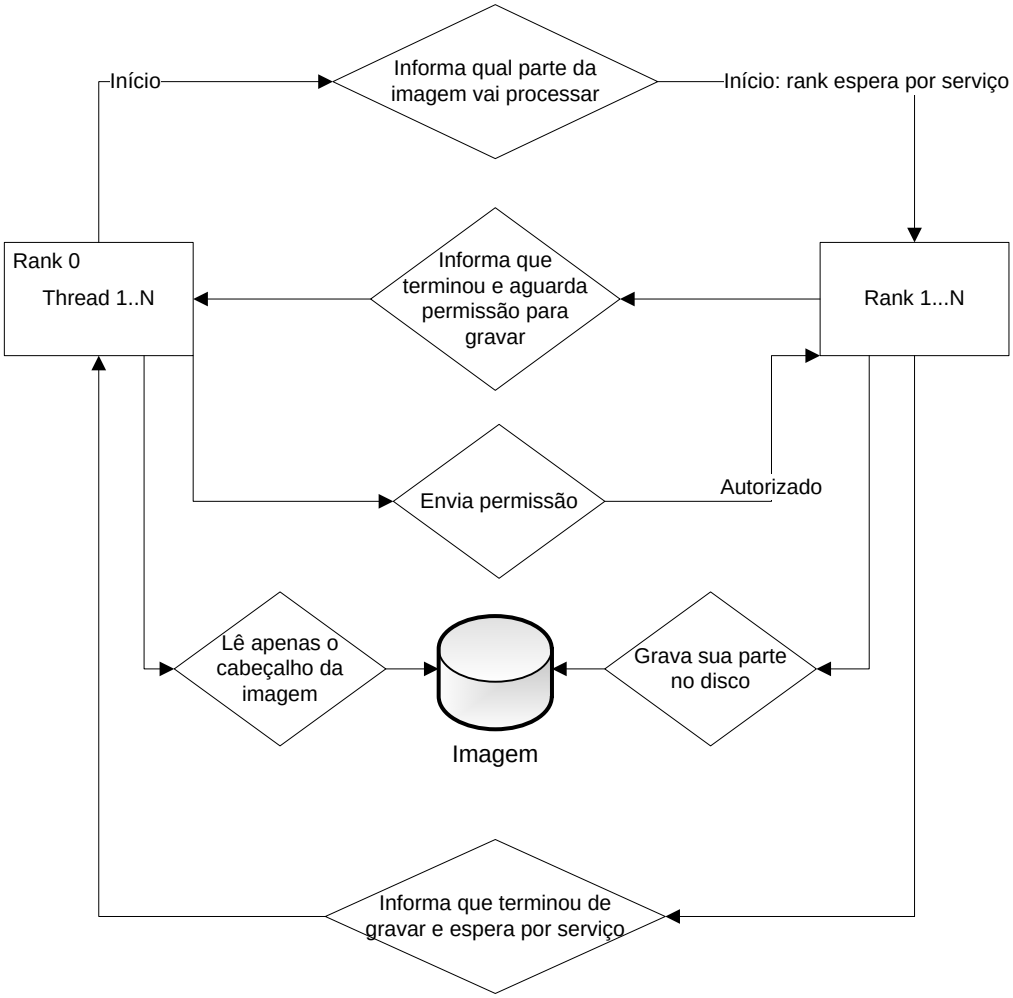

Figura 12 – Decomposição do problema.

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. O Rank 0 inicia os Threads de 1..N de acordo com o número de processos.

4. O Thread N chama uma função que retorna a quantidade de trabalho ainda disponível que pode ser escalonada para o seu Rank N de três formas possíveis:
    a. A função divide a imagem igualitariamente e retorna a parte a ser executada.
    b. A função retorna a quantidade de linhas baseada na carga máxima, informada pelo usuário.
    c. 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.

5. 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:
    a. 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.
    b. Se o Rank recebeu um bloco do meio da imagem, ele faz a leitura de duas linhas acima e duas abaixo.
    c. Se o Rank recebeu o fim da imagem, ele faz a leitura das duas linhas acima.

6. Rank 1..N aplica o filtro de Smooth e envia mensagem para seu thread correspondente no Rank 0 solicitando permissão para gravar.

7. O 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.

8. Rank 1..N que recebeu permissão de gravação de seu thread, faz a gravação da sua parte da imagem no disco e informa que terminou a gravação.

9. O thread no rank 0 recebe a mensagem informando do término da gravação e deixa a região crítica de memória, abrindo disputa para outros threads. O 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.

10. Rank 1..N finaliza.

11. Thread 1..N finaliza

12. Rank 0 finaliza.

4.7 IMAGENS PPM E PGM

Para os testes foram utilizadas imagens do site HiRISE (*High Resolution Imaging Science Experiment*) disponíveis em: http://www.uahirise.org/katalogos.php, tanto imagens coloridas como imagens em escada de cinza, conforme demostrado na Tabela 2:

TABELA 2 – IMAGENS UTILIZADAS NOS TESTES.

Nome: ESP_042084_1690 **Local:** http://www.uahirise.org **Dimensão:** 1678x7105 **Tamanho:** 34,1MB **Tipo:** colorida **Classificação:** pequena	**Nome:** ESP_042084_1690 **Local:** http://www.uahirise.org **Dimensão:** 4096x8624 **Tamanho:** 33,6MB **Tipo:** tons de cinza **Classificação:** pequena
Nome: ESP_042433_1535 **Local:** http://www.uahirise.org **Dimensão:** 2268x10275 **Tamanho:** 66,6MB **Tipo:** colorida	**Nome:** ESP_042433_1535 **Local:** http://www.uahirise.org **Dimensão:** 5626x10272 **Tamanho:** 55,1MB **Tipo:** tons de cinza

Classificação: média	**Classificação:** média
Nome: ESP_042641_1900 **Local:** http://www.uahirise.org **Dimensão:** 4832x16184 **Tamanho:** 223MB **Tipo:** colorida **Classificação:** grande	**Nome:** ESP_042641_1900 **Local:** http://www.uahirise.org **Dimensão:** 18373x21421 **Tamanho:** 375MB **Tipo:** tons de cinza **Classificação:** grande
Nome: ESP_024497_1745 **Local:** http://www.uahirise.org **Dimensão:** 7115x36873 **Tamanho:** 750MB **Tipo:** colorida **Classificação:** grande	**Nome:** ESP_024497_1745 **Local:** http://www.uahirise.org **Dimensão:** 17834x39941 **Tamanho:** 679MB **Tipo:** tons de cinza **Classificação:** grande

5 RESULTADOS E DISCUSSÕES

Aqui são apresentados os resultados e discussões com base nos algoritmos sequencial e paralelo. Para cada diretiva testada (número de processos, *threads* e nós) foram geradas dez execuções com as mesmas condições de ambiente.

5.1 ALGORITMO SEQUENCIAL

Os testes foram realizados com imagens coloridas e em tons de cinza utilizando o algoritmo sequencial, obtendo os resultados conforme a Tabela 3 e Tabela 4:

TABELA 3 – EXECUÇÃO DAS IMAGENS COLORIDAS PELO ALGORITMO SEQUENCIAL.

IMAGEM ***n x n***	**MÉDIA (milissegundos)**	**DESVIO (milissegundos)**
7105x1678	12058,46	11,96
10275x2268	23571,64	30,97
16184x4832	81594,26	8246,31
36873x7115	274080,74	27743,12

TABELA 4 – EXECUÇÃO DAS IMAGENS EM TONS DE CINZA PELO ALGORITMO SEQUENCIAL.

IMAGEM ***n x n***	**MÉDIA (milissegundos)**	**DESVIO (milissegundos)**
8624x4096	32355,48	41,25
10272x5626	52799,06	12,96
21421x18373	361657,63	6923,92
39941x17834	705895,10	79936,82

Foram realizadas as médias das dez execuções para cada imagem e o seu desvio padrão. A Figura 13 mostra os tempos de processamento das imagens coloridas e com tons de cinza. Apesar das duas categorias apresentarem tamanhos semelhantes, as imagens com tons de cinza (PGM) apresentam um processamento mais lento, pois a leitura, gravação e aplicação do *Smooth* são realizadas por pixel.

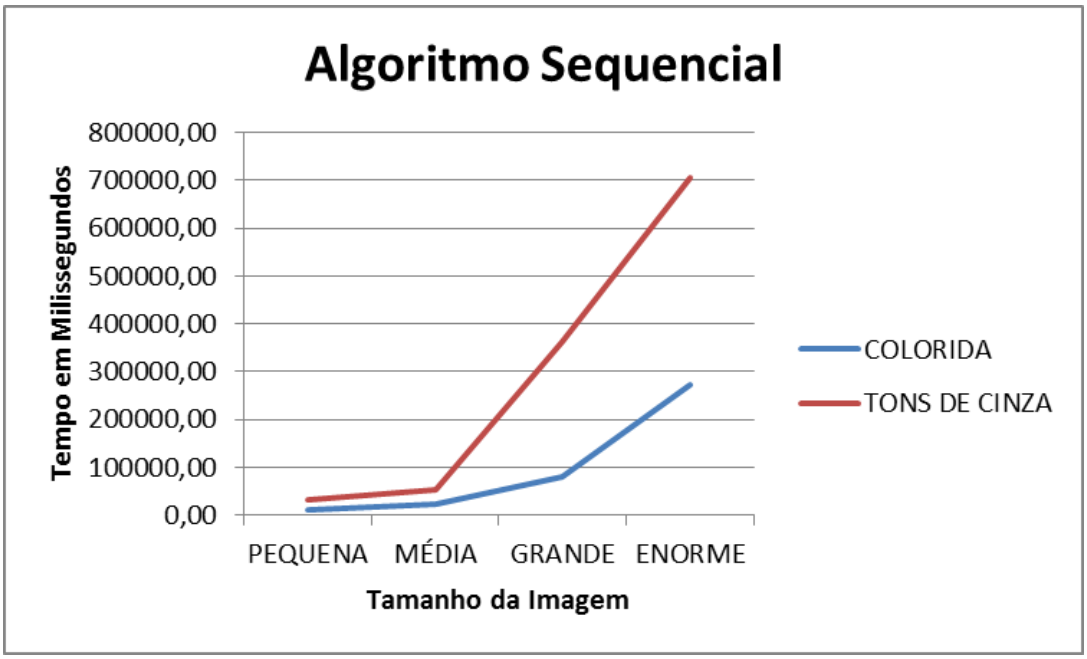

Figura 13 – Execução das imagens pelo método sequencial.

5.2 ALGORITMO PARALELO: *OPENMP E OPENMPI*

Aqui são demonstrados os testes com o *OpenMP* e *OpenMPI*. Após analisar a Seção 3.3, alguns testes foram realizados para determinar o melhor tempo resposta nas execuções.

5.2.1 CARGAS ALEATÓRIAS X CARGAS DETERMINADAS

Uma das formas de equilibrar a disputa quando se tem grandes quantidades de informações, é enviar cargas aleatórias, baseado no algoritmo *QuickSort*. Desse modo, uma das opções de execução do sistema desenvolvido é enviar a *Flag* **a = 1**. Isso faz que o sistema comece a enviar para os nós cargas aleatórias baseadas na seguinte fórmula.

Carga = Randômico (Mínimo, Máximo);

Onde,

Mínimo = Número Máximo de Linhas/2 e

Máximo = Número Máximo de Linhas*2

O número máximo de linhas é a razão entre a quantidade de linhas da imagem pelo número de processos mais o seu número de threads.

Na Tabela 5 é possível verificar a média e o desvio padrão da execução pelo método de cargas aleatórias e comparativamente com cargas determinadas da imagem de ordem 10275x2268 (Tipo colorida, média).

TABELA 5 – EXECUÇÃO COM CARGAS ALEATÓRIAS X DETERMINADAS EM IMAGEM COLORIDA.

NODES	PROCESSOS	DETERMINADA	DESVIO	ALEATÓRIA	DESVIO
5	5	4061,67	878,50	4726,28	550,32
5	10	2907,93	578,71	3018,70	553,59
5	15	1749,69	215,19	2050,54	221,27
5	20	2395,76	502,08	2689,53	407,58
10	5	6486,63	2212,86	6068,47	2070,41
10	10	3010,57	575,75	3396,11	956,62
10	15	2485,65	626,98	2784,24	483,38
10	20	1751,08	213,49	1843,97	303,62
10	30	1899,06	229,18	2153,64	321,33
10	40	1554,67	172,84	1814,85	248,39
15	5	3732,11	765,46	5395,15	978,33
15	10	3521,02	1004,83	3759,05	745,01
15	15	1796,87	169,74	2112,51	358,48
15	20	2377,19	421,27	3179,62	628,16
15	30	2142,57	341,76	2233,37	482,62
15	40	1780,17	217,20	2049,37	332,83
15	50	1907,97	264,16	1841,38	156,04
MÉDIA		**2680,03**		**3006,86**	

Pelos testes e os resultados, é possível observar que com um número menor de processos e nós, o algoritmo aleatório obteve melhores resultados. Porém, quando a aplicação se torna mais distribuída e com mais processos envolvidos, distribuir a carga igualitariamente mostrou-se mais eficiente. Isso também acontece pela concorrência por recursos entre processos, no caso aqui, pela leitura de disco e processamento. Esse fato por si, já promove um balanceamento de carga na concorrência por recursos. Na Figura 14 é possível observar a eficiência dos dois modos de execução.

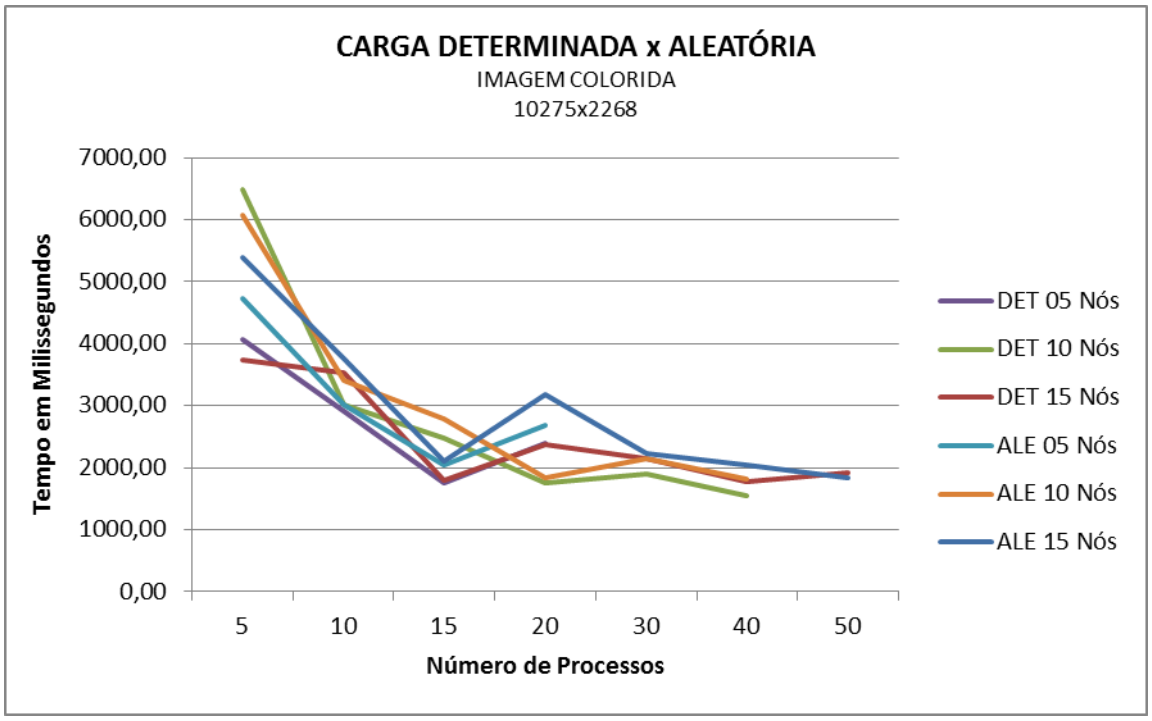

Figura 14 – Teste com cargas aleatórias x determinadas em imagem colorida.

O mesmo teste é realizado para imagens com tons de cinza. Na Tabela 5 é possível verificar a média e o desvio padrão da imagem de ordem 10275x2268 (média).

TABELA 6 – EXECUÇÃO COM CARGAS ALEATÓRIAS X DETERMINADAS EM IMAGEM COM TONS DE CINZA.

NODES	PROCESSOS	DETERMINADA	DESVIO	ALEATÓRIA	DESVIO
5	5	8408,28	1613,80	11384,70	1630,75
5	10	7662,32	1300,66	7635,65	1435,16
5	15	4169,25	493,89	4875,25	1098,42
5	20	6137,64	740,66	5817,62	1187,75
10	5	13131,34	3484,20	17150,67	4520,21
10	10	6781,06	1766,80	7841,64	1665,23
10	15	6383,79	1386,46	7231,65	971,23
10	20	3641,18	549,92	5042,16	1365,70
10	30	4176,77	339,78	4668,57	900,47
10	40	3382,99	469,81	3675,77	421,92
15	5	9022,24	2429,41	10400,63	4323,36
15	10	9346,31	2594,26	7473,08	1423,79
15	15	4393,69	683,73	4338,12	586,99
15	20	5511,20	887,26	5021,62	1337,15
15	30	3699,00	338,40	4661,90	739,62

15	40	3361,28	432,16	3936,50	537,57
15	50	3072,69	259,21	3242,61	454,03
MÉDIA		6016,53		6729,30	

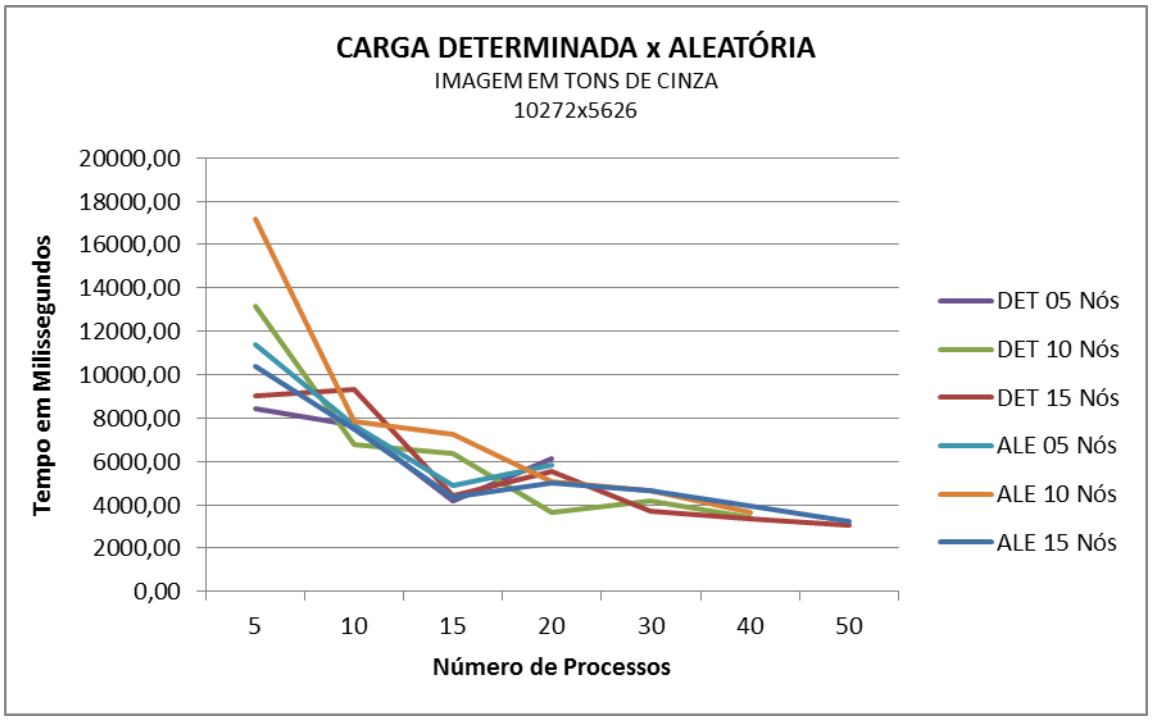

Figura 15 – Teste com cargas aleatórias x determinadas em imagem com tons de cinza.

O mesmo observado na leitura anterior, repetiu nesse experimento. Executar os testes com cargas determinadas mostrou-se mais eficiente para processos com um maior número de nós.

5.2.2 LEITURA SEQUENCIAL X LEITURA CONCORRENTE

Outro experimento interessante para analisar o comportamento do algoritmo e do cluster, é verificar se remover a concorrência de leitura da imagem, tornaria o processo mais eficiente. Para esse teste foram utilizados 15 nós com 40 processos, a Tabela 7 mostra a diferença de resposta nessa modalidade.

TABELA 7 – EXECUÇÃO COM LEITURA SEQUENCIAL X CONCORRENTE EM IMAGEM COLORIDA.

IMAGEM ***n x n***	**LEITURA**	**MÉDIA (milissegundos)**	**DESVIO (milissegundos)**
10275x2268	Sequencial	276485,63	12479,15
10275x2268	Concorrente	1780,17	217,20

Conforme pode ser observado na Figura 16, a leitura sequencial mostrou ter o pior desempenho, aproximadamente 155 vezes mais lenta que a leitura concorrente. Assim, os demais testes foram realizados com base na leitura concorrente pelos processos nos nós.

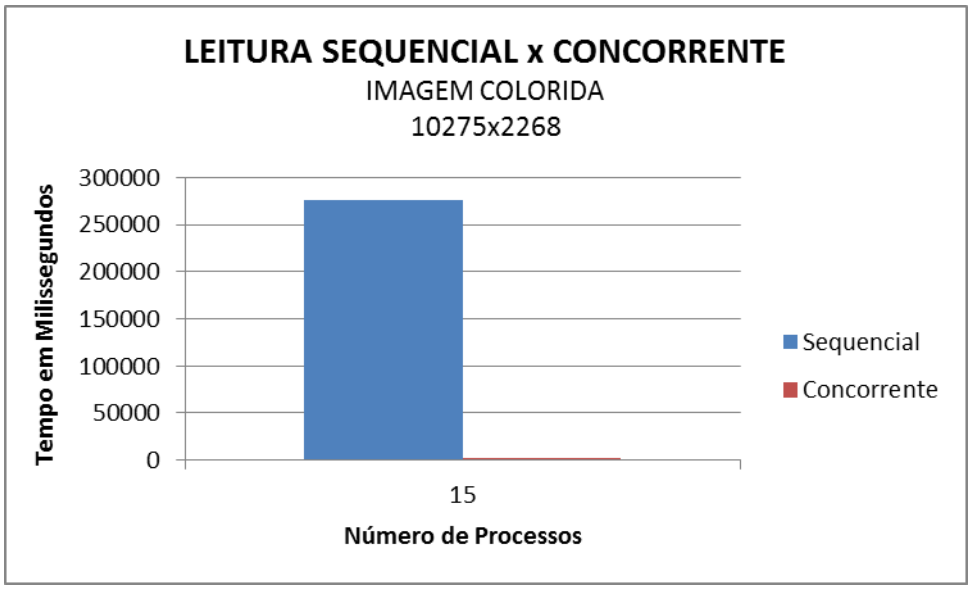

Figura 16 – Teste com leitura sequencial x concorrente em imagem colorida.

5.2.3 DEFINIÇÃO DO NÚMERO DE *THREADS*

Além do número de processos e de nós, outra variável a considerar é a quantidade de *threads* que cada processo utilizará para ler e aplicar o filtro na imagem. Inicialmente os testes foram realizados com um padrão de três *threads* por processo e foi possível observar que a quantidade de threads é influenciada pela quantidade de processos em execução em cada nó.

Levando em consideração uma execução com 40 processos em 10 nós, cada núcleo de cada nó executará um processo, sendo assim, aumentar a quantidade de *threads* também aumentará a concorrência nesse tipo de execução.

Na Figura 17 é possível observar esse comportamento, o teste foi realizado com 10 nós e 40 processos, onde a média dos dez lançamentos teve um melhor desempenho com apenas um *thread*.

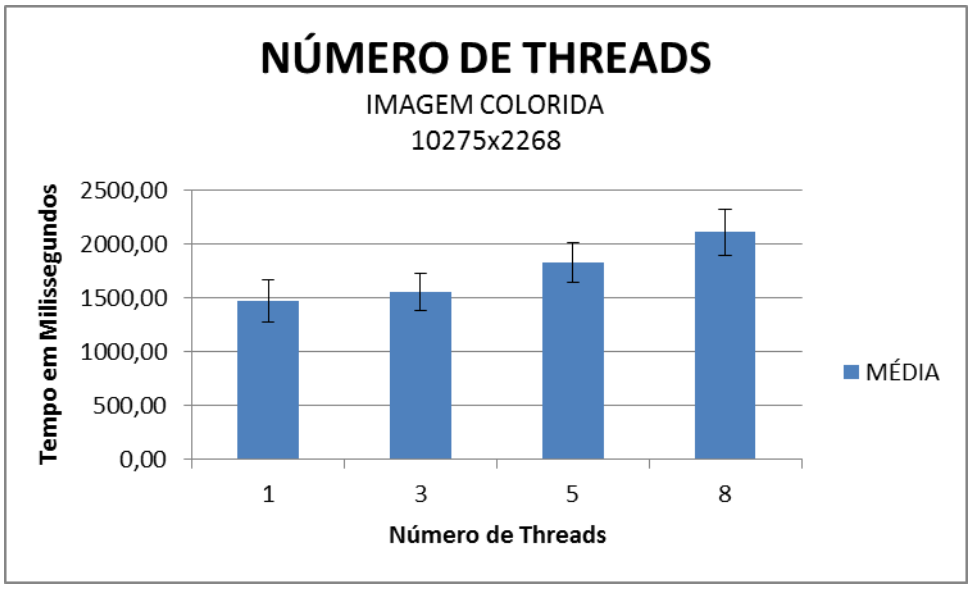

Figura 17 – Definição do número de *threads* em imagem colorida.

Em outro teste realizado com 15 nós e 50 processos, execuções com três *threads* tiveram um melhor desempenho. Nesse caso, dividindo 50 processos por 15 nós, tem-se uma média 3,33 processos por nó. Aparentemente as execuções com apenas um *thread* deveriam ter um melhor desempenho, mas isso não acontece pelo fato do OpenMPI utilizar o algoritmo *round-robin* para fazer a distribuição de processos. Nesse caso pode acontecer de alguns nós receberem mais processos que outros, justificando os resultados contidos na Figura 18:

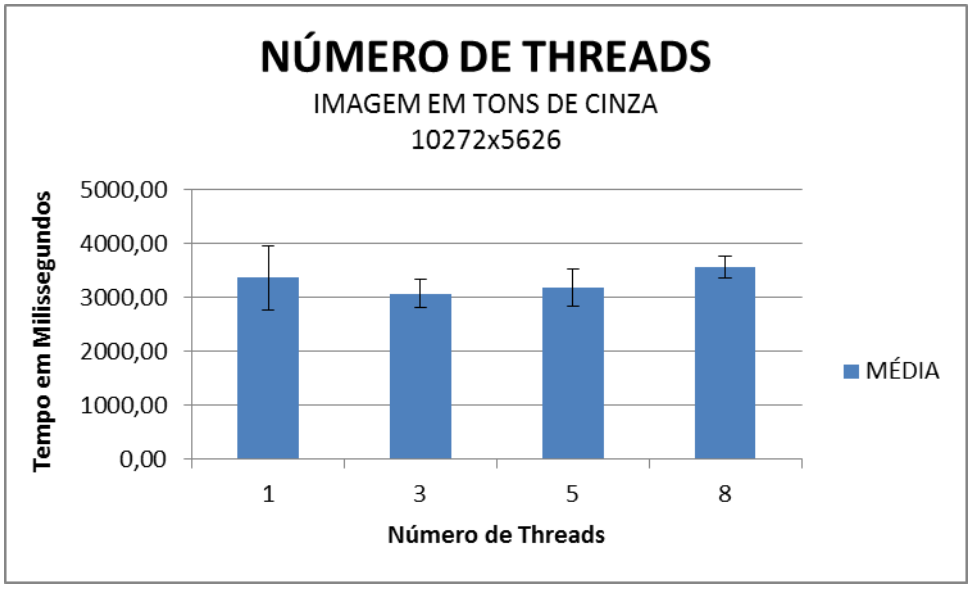

Figura 18 – Definição do número de *threads* em imagem com tons de cinza.

Para contornar esse problema, uma solução interessante é cada processo verificar quantos outros processos existem em seu nó e definir a quantidade de threads baseado nesse resultado, extraindo todo o desempenho do nó, evitando concorrência por recursos.

A Tabela 8 mostra os resultados do experimento:

TABELA 8 – DEFINIÇÃO DO NÚMERO DE *THREADS* EM IMAGEM DO TIPO MÉDIA.

IMAGEM ***n x n***	**THREADS**	**MÉDIA (milissegundos)**	**DESVIO (milissegundos)**
10275x2268 (colorida)	1	1472,27	191,29
10275x2268 (colorida)	3	1554,67	172,84
10275x2268 (colorida)	5	1828,46	181,93
10275x2268 (colorida)	8	2111,10	214,15
10272x5626 (tons de cinza)	1	3363,15	592,65
10272x5626 (tons de cinza)	3	3072,69	259,21
10272x5626 (tons de cinza)	5	3176,77	347,29
10272x5626 (tons de cinza)	8	3568,37	201,78

5.2.4 IMAGENS COLORIDAS

Nessa seção são demonstrados os resultados para imagens coloridas utilizando o algoritmo paralelo. As execuções foram organizadas conforme demonstrado na Tabela 9:

TABELA 9 – DEFINIÇÃO DOS PARÂMETROS DE EXECUÇÃO.

Quantidade de Nós	**Número de Processos**	**Número de Threads**	**Leitura Sequencial**	**Carga**
5	5, 10, 15 e 20	3	Não	Determinada
10	5, 10, 15, 20, 30 e 40	3	Não	Determinada
15	5, 10, 15, 20, 30, 40 e 50	3	Não	Determinada

Nos testes preliminares, a quantidade de *threads* influência nos resultados, porém observou-se que três *threads* correspondiam à média dos melhores desempenhos. Como gerar testes para todas as possibilidades produziria muitos resultados e não seria possível conciliar o cronograma estipulado para o trabalho, logo essa quantidade foi utilizada como padrão para as execuções. Como mencionado anteriormente, uma solução interessante é verificar quantos processos estão em execução em um determinado nó e definir a quantidade de *threads* baseado nesse resultado.

A Figura 19 mostra a execução de imagem colorida, do tipo pequena, pelo método paralelo:

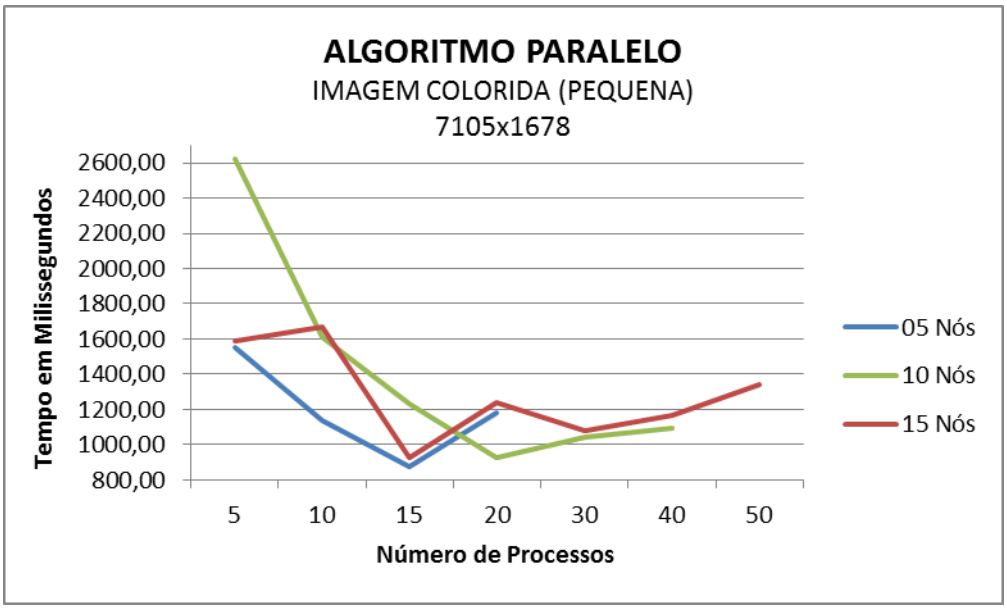

Figura 19 – Execução de imagem colorida pelo método paralelo (pequena).

A Figura 20 mostra a execução de imagem colorida, do tipo média, pelo método paralelo:

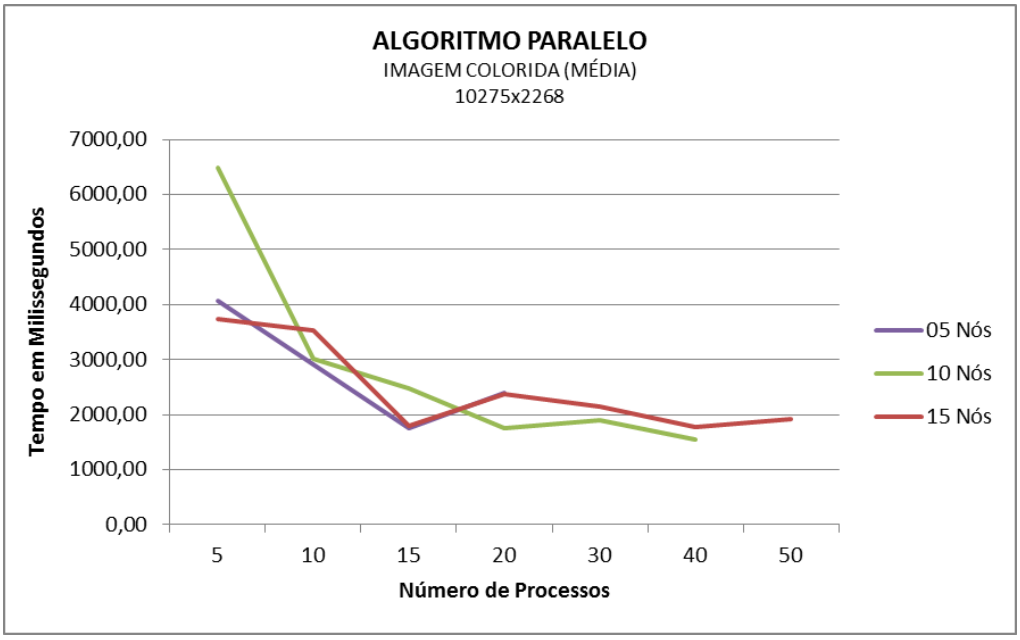

Figura 20 – Execução de imagem colorida pelo método paralelo (média).

A Figura 21 mostra a execução de imagem colorida, do tipo grande, pelo método paralelo:

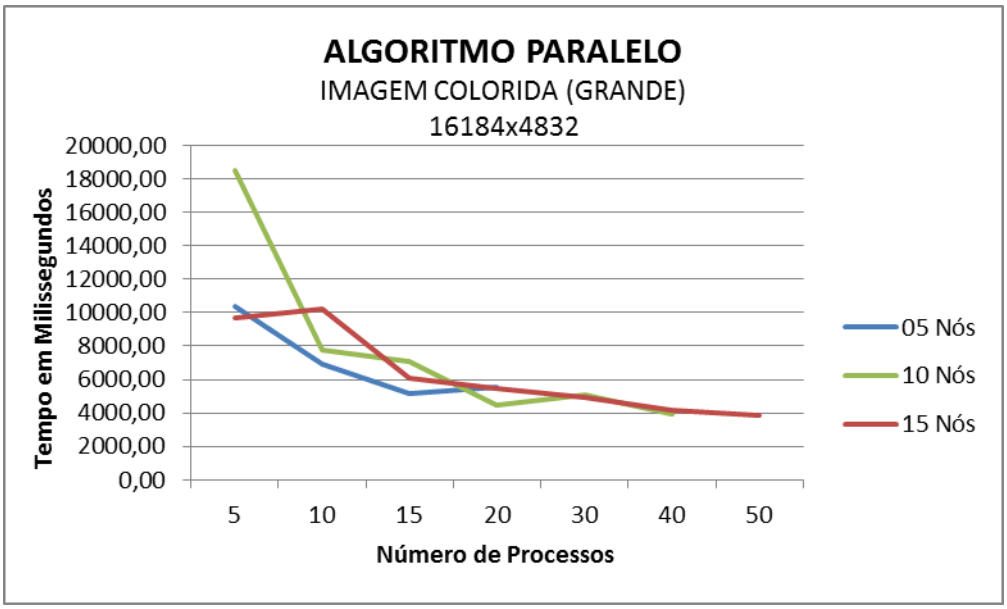

Figura 21 – Execução de imagem colorida pelo método paralelo (grande).

A Figura 21 mostra a execução de imagem colorida, do tipo enorme, pelo método paralelo:

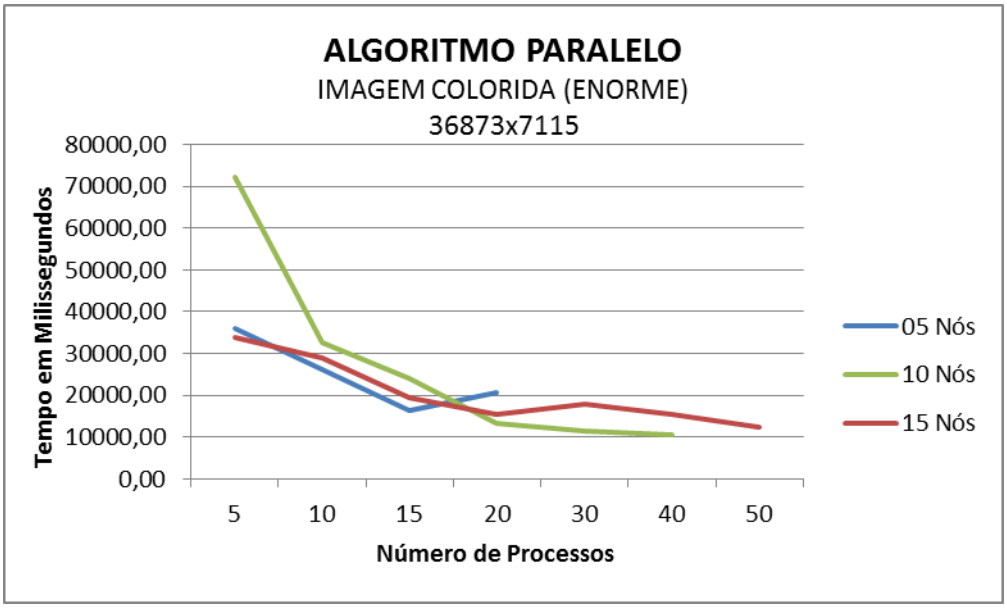

Figura 22 – Execução de imagem colorida pelo método paralelo (enorme).

5.2.5 IMAGENS EM TONS DE CINZA

Os testes com imagens com tons de cinza foram executados nos mesmos padrões estabelecidos na seção 5.2.4.

A Figura 23 Figura 21 mostra a execução de imagem com tons de cinza, do tipo pequena, pelo método paralelo:

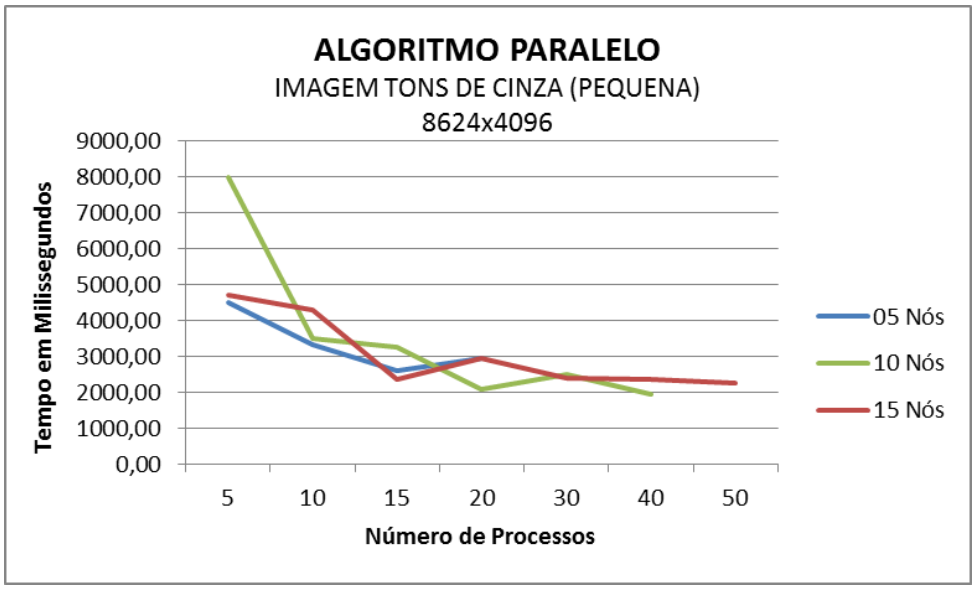

Figura 23 – Execução de imagem em tons de cinza pelo método paralelo (pequena).

A Figura 24Figura 21 mostra a execução de imagem com tons de cinza, do tipo média, pelo método paralelo:

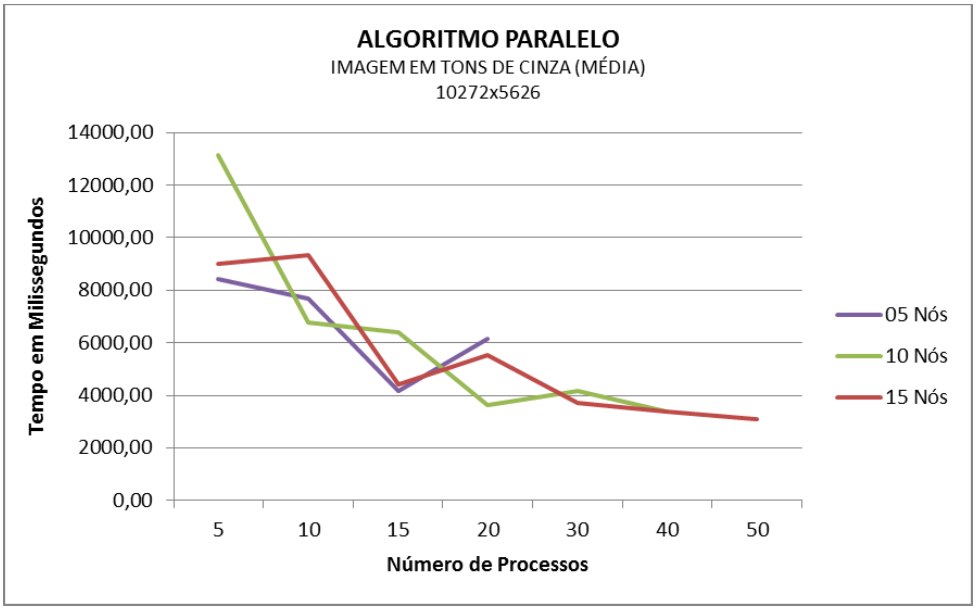

Figura 24 – Execução de imagem em tons de cinza pelo método paralelo (média).

A Figura 25Figura 21 mostra a execução de imagem com tons de cinza, do tipo grande, pelo método paralelo:

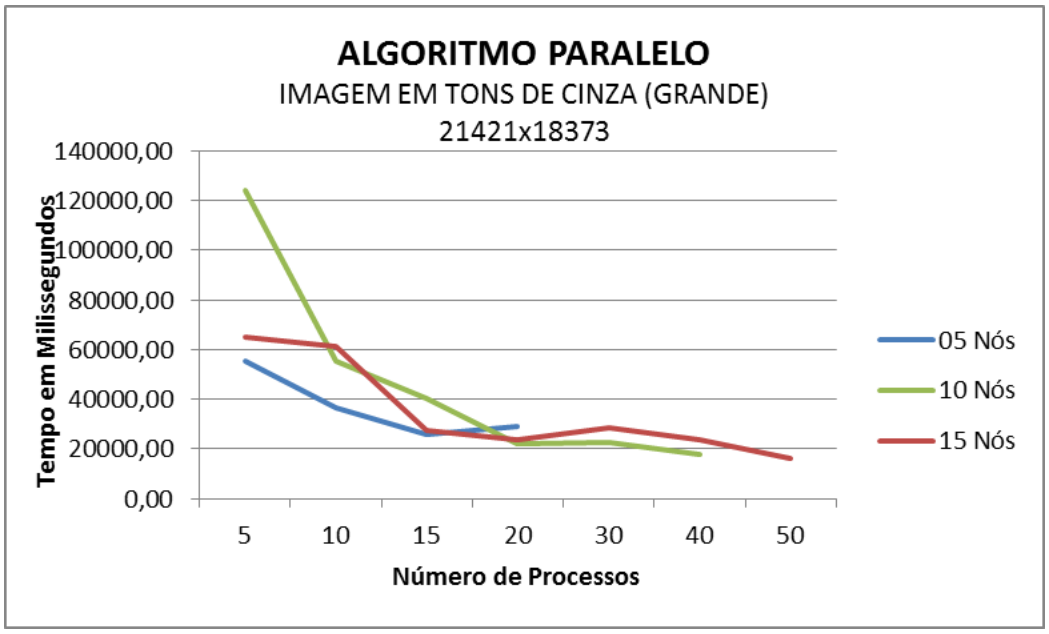

Figura 25 – Execução de imagem em tons de cinza pelo método paralelo (grande).

A Figura 26Figura 21 mostra a execução de imagem com tons de cinza, do tipo enorme, pelo método paralelo:

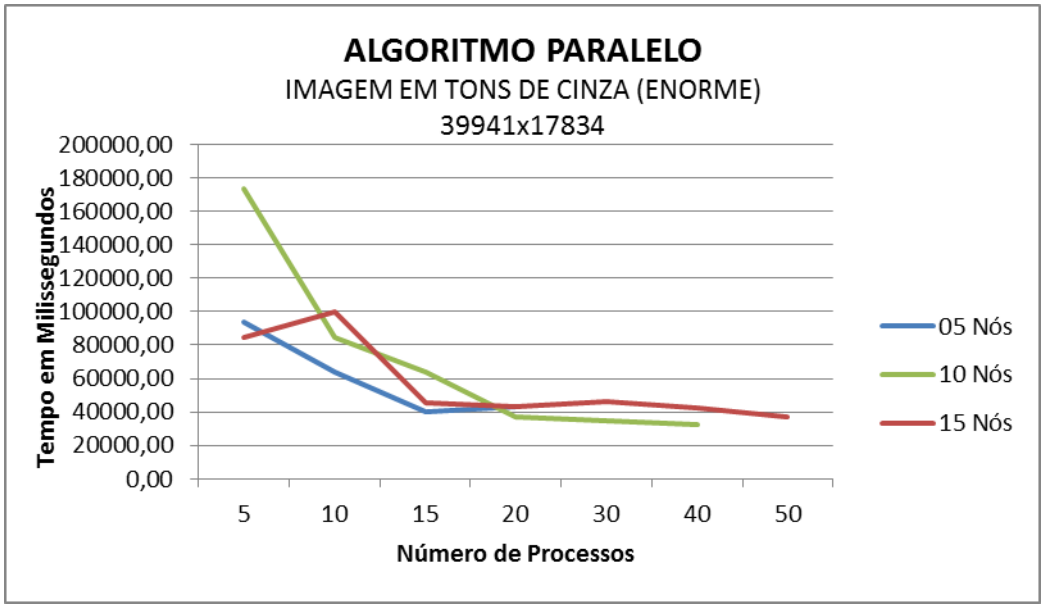

Figura 26 – Execução de imagem em tons de cinza pelo método paralelo (enorme).

5.3 EFICIÊNCIA DO ALGORITMO DESENVOLVIDO

A Tabela 10 mostra a eficiência do algoritmo sequencial comparado com o melhor resultado, após a média de dez iterações, do algoritmo paralelo:

TABELA 10 – EFICIÊNCIA DO ALGORITMO SEQUENCIAL E PARALELO.

IMAGEM	MÉDIA SEQUENCIAL (milissegundos)	MÉDIA PARALELO (milissegundos)
Pequena (colorida)	12058,46	875,02
Média (colorida)	23571,64	1751,08
Grande (colorida)	81594,26	3831,87
Enorme (colorida)	274080,74	10455,87
Pequena (tons de cinza)	32355,48	3072,69
Média (tons de cinza)	52799,06	3072,69
Grande (tons de cinza)	361657,63	16053,68
Enorme (tons de cinza)	705895,10	32780,35

A Figura 27 evidencia a diferença da solução paralelizada comparativamente com a sequencial.

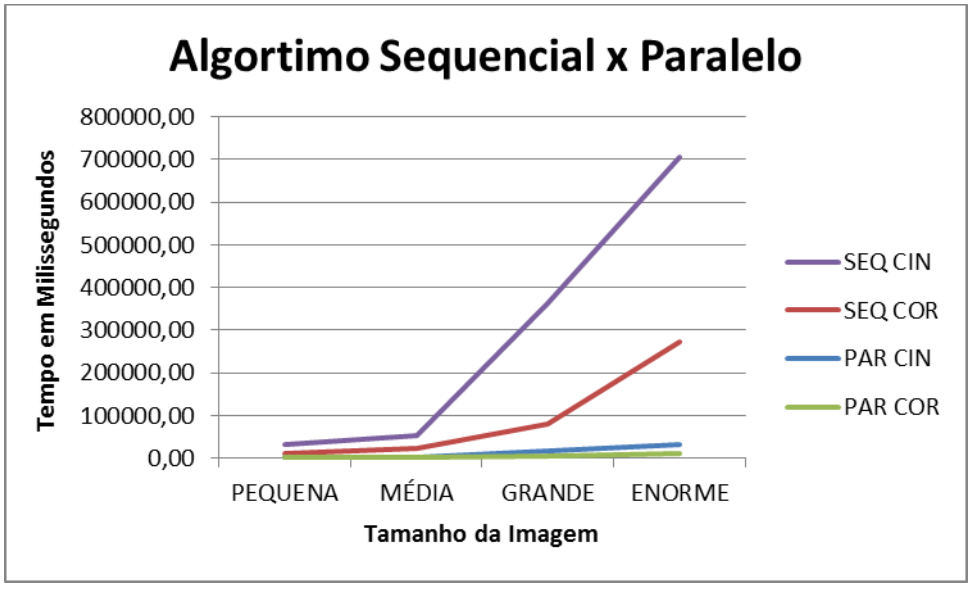

Figura 27 – Eficiência do algoritmo sequencial e paralelo.

5.4 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.

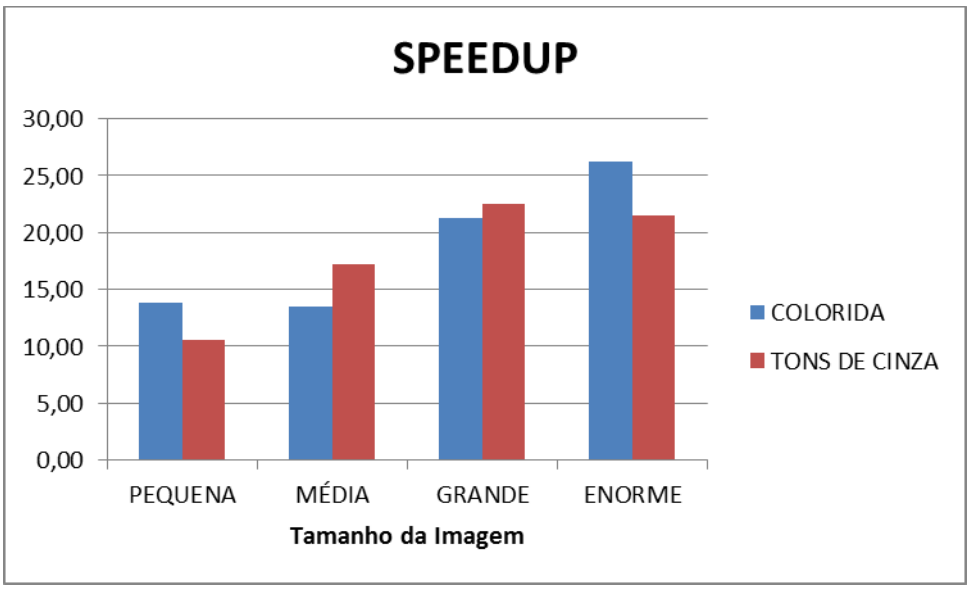

Figura 28 – SpeedUp entre os algoritmos paralelo e sequencial.

5.5 DIFICULDADES ENCONTRADAS

Algumas dificuldades foram encontradas no decorrer do trabalho, essa seção aponta quais foram elas e quais as soluções adotadas.

5.5.1 PROBLEMA DE ESCRITA COM CONCORRÊNCIA EM DISCO

Nesse trabalho, foi estabelecido que os nós gravariam diretamente sua parte da imagem no disco, evitando assim, excesso de comunicação na rede. Inicialmente nenhum tipo de semáforo havia sido definido para controlar qual processo poderia

gravar sua parte da imagem no disco.

- Problema: nos testes com imagens pequenas nenhum problema foi detectado, porém com imagens maiores foi constatado que algumas partes da imagem não eram gravadas. Isso acontece, pois para um processo gravar uma parte de uma imagem, ele precisa saber o que há no arquivo para conservar o conteúdo existente de outro processo que já tenha gravado. Todavia, se dois processos gravarem ao mesmo tempo, um não saberá o conteúdo do outro e o último processo que finalizar a escrita, conservará apenas o conteúdo anterior da imagem e o seu conteúdo.

- Solução: para contornar esse problema, foi colocado um semáforo no Rank 0, que controla qual o nó que poderá gravar no disco. Assim, quando um processo terminar de aplicar o filtro, este solicita para seu *thread* responsável no Rank 0 para gravar em disco.

5.5.2 DISPUTA POR LEITURA EM DISCO POR PROCESSOS

Inicialmente acreditava-se que a disputa em disco por processos, para realizar a leitura da imagem, poderia diminuir o desempenho da aplicação. Nesse caso, também foi criado um semáforo para leitura, porém nos testes ficou claro que a leitura não compromete o desempenho.

Acredita-se que cache de disco e do NFS, a concorrência natural pelos processos e a quantidade pequena de linhas que são lidas, pode justificar esse resultado.

5.5.3 COMPORTAMENTO INESPERADO DO OPENMPI

Em alguns testes com um grande número de processos e consequentemente, um grande número de mensagens na rede, foi observado alguns erros intermitentes:

```
mca_btl_tcp_frag_recv: readv failed: Connection reset by peer
(104)

Your Open MPI job may now fail.

  Local host: node10
  PID:        17144
  Message:    did not receive entire connect ACK from peer
```

Aparentemente, por algum motivo o nó rejeitou a mensagem. Pelas pesquisas

realizadas, na tentativa de descobrir o erro, alguns fóruns e outras aplicações com o mesmo problema, apontam um problema na versão do OpenMPI, porém nenhuma solução foi apresentada. Como o erro é intermitente e acontece poucas vezes, não gerou transtornos para executar os testes.

5.5.4 DEPURAÇÃO DE SOFTWARE COM OPENMPI

Outro problema encontrado é a forma como o OpenMPI apresenta os seus erros e os erros na aplicação. Estes, muitas vezes, são difíceis de interpretar e falta informação, fazendo-se necessário recorrer para aplicações paralelas, como por exemplo, valgrind.

5.6 CONSIDERAÇÕES FINAIS

Conforme pode ser observado ao longo desse trabalho, paralelizar aplicações permitiu um ganho de tempo substancial em vista de algoritmos sequenciais. Algumas dificuldades foram encontradas utilizando OpenMPI, porém o resultado é bastante satisfatório.

Pelos testes realizados, a aplicação paralela desenvolvida é escalável com imagens grandes, porém não é tão eficiente com imagens pequenas. Para imagens pequenas, determinar a quantidade de linhas por nós e processos é fundamental para conseguir os melhores resultados.

Para imagens grandes, a divisão da carga igualitariamente retornou um bom resultado. Alguns testes foram realizados diminuindo a carga e estimulando o uso equilibrado por recursos, porém a quantidade de mensagens geradas na rede tornou a solução não satisfatória.

Uma possibilidade para melhorar o desempenho do algoritmo desenvolvido nesse trabalho é criar um mecanismo para determinar a quantidade de processos existentes nos nós e permitir que cada processo decida quantos *threads* utilizará para ler a imagem e aplicar o filtro. Isso é necessário, pois foi verificado que nós, com quatro núcleos, que já possuem quatro processos iniciados, têm um pior desempenho caso iniciem mais de um *thread* por processo, gerando uma concorrência maior por recursos.

6 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.