# Programação Paralela - Contando Estrelas

Aluno: José Carlos Tobias
Professor: Paulo Bressan

Universidade Federal de Alfenas - UNIFAL

26 de Junho de 2019

## 1 INTRODUÇÃO

A tarefa nos dada se baseia na proposta de utilizar um processamento paralelo para contagem de estrelas em imagens de grande tamanho, neste caso em torno de $20GB$ de dados separados em 20 partes, cada uma com em torno de $1GB$. A execução deste algoritmo requer então, uma interdisciplinaridade com a disciplina de processamento de imagens, para preparar a imagem a ponto de ficar possível realizar o processamento.

Tal tarefa requer um algoritmo com uma complexidade que pode ser e foi dividido em duas partes, onde o primeiro passo do algoritmo transforma a imagem em níveis de cinza, seguida por uma "binarização"dado um valor limítrofe de separação. Esta execução é essencial para que então, um algoritmo de reconhecimento de componentes conexos em imagens binárias seja executado sobre a imagem, dando como saída o número de componentes conexos encontrados na imagem cujo valor dos *pixels* é subentendido para a cor branca, o qual é de fato o mesmo do número de estrelas.

O processamento paralelo é então utilizado para, dado um processo mestre que tem a intenção de manejar a execução, distribuir os blocos de imagens para que as ações de processamento de imagens, i.e transformação para níveis de cinza, binarização e retorno contagem de componentes conexos, sejam executados sobre cada bloco de maneira paralela. Vários aspectos do processamento paralelo podem ser estudados então sobre cada execução, dependendo do tamanho dos blocos a serem divididos e números de processos a serem executados.

O objetivo da tarefa é nos dar uma ampla visão sobre a utilização do processamento paralelo em problemas que exigem grande poder computacional, utilizando troca de mensagens em uma arquitetura de memória distribuída. Assim sendo, na próxima seção apresentarei, em detalhes, o algoritmo utilizado para a execução da tarefa, na próxima uma breve seção de explicação do método simples utilizado para testes, para uma seção de resultados onde a demonstração de execuções com variados parâmetros serão demonstradas, uma seção de métricas vem após onde serão apresentadas em detalhes as métricas requeridas para o trabalho em documento, finalizando as seções com uma conclusão dos meus pensamentos sobre a tarefa e aprendizados da implementação. A implementação completa pode ser encontrada em: `http://bit.ly/parallel_counting_stars`

# 2 ABSTRAÇÃO

Imagine que temos uma imagem qualquer, com $L$ linhas e $C$ colunas, e para fazer a distribuição dessa imagem devemos separá-la em $n$ blocos de $l$ linhas por $c$ colunas. Sabemos que o tamanho dos blocos pode ser de qualquer tamanho tanto para linhas e tanto para colunas, os últimos blocos de cada linha e coluna possuirão um valor de linhas $l - \alpha$ e colunas de $c - \lambda$, onde $0 \leq \alpha \leq L$ e $0 \leq \lambda \leq C$. A separação deve ser feita então de maneira a permitir que alguns blocos que pertençam a borda tenham tamanho variado. A Figura 1 esquematiza esta separação dos blocos.

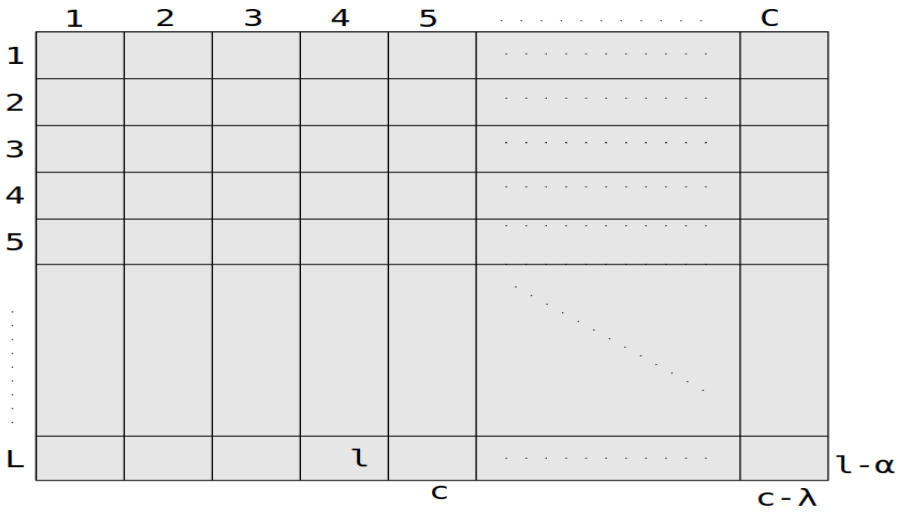


Figura 1: Visão de separação de uma imagem com $L$ linhas e $C$ colunas em $n$ blocos de $l$ linhas por $c$ colunas. Fica evidentes que o tamanho dos blocos sendo arbitrário, poderá haver nas bordas blocos de tamanhos diferentes, indicados por dimensão de $l - \alpha$ e colunas $c - \lambda$.

Os blocos então devem ser distribuídos em um modelo paralelo de processamento, onde cada um dos $k$ processos trabalhadores recebe um bloco de um processo mestre, faz o processamento necessário e envia para um processo mestre os resultados desse processamento; o processo mestre deve então enviar outro bloco para o processo que terminou seu processamento, até que não existam mais blocos. Podemos pensar que o mestre vê os blocos separados como uma lista de $n$ blocos, e simplesmente os envia para os processos pedintes. Este modelo deve evidenciar o fato de computações em blocos de dados não dependentes poderem ser executados de maneira paralela. A Figura 2 demonstra o processo descrito.

Dado este entendimento, estamos prontos para implementar o algoritmo.

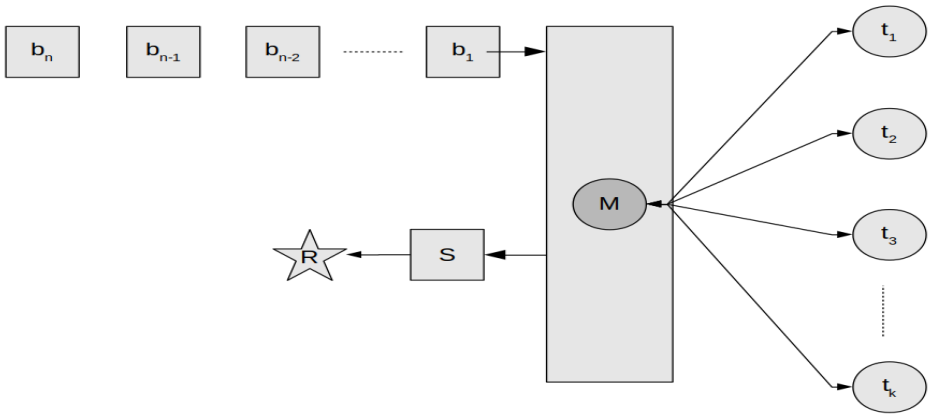


Figura 2: Esquema de envio de $n$ blocos para processamento. Uma lista de n blocos $b$ é criada, cada um dos $k$ processos trabalhadores $t$ recebe, em ciclos, um dos blocos de um processo mestre $M$. Cada processo $t$ executa sobre o seu dado recebido um arcabouço de operações. Os resultados de todos os processos trabalhadores são recebidos pelo mestre ao final, passando por um somador $S$ e guardado em um local de resultados $R$.

# 3 ALGORITMO

A implementação do algoritmo, na linguagem C, foi feita utilizando duas especificações para computação de alta performance na linguagem, sendo elas o OpenMPI (Open Message Passing Interface) e o OpenMP (Open Multi-Processing). O OpenMPI é uma biblioteca que dá suporte para implementação de programas paralelos por meio de passagem de mensagens, nos servindo então para distribuir os processos para os determinados processadores onde os mesmos serão executados - em nossa arquitetura de multicomputadores e memória distribuída. Já a biblioteca OpenMP permite paralelismos em nível de thread local, nos servindo para paralelizar repetições sem dependência de dados dentro de tais processos, uma vez que seus locais de acesso são privativos.

A organização se dá então com um processo mestre, que tem como função fazer a distribuição dos dados das imagens para o processo, enquanto guarda informações de retorno dos mesmos. O processo mestre divide a imagem a ser processada em blocos de tamanho pré-fixado em maior quantidade possível, deixando blocos menores nas extremidades normalmente, e os distribui para processos quando os mesmos ficam ociosos após retornarem os valores achados de estrelas no bloco para o mestre.

As bibliotecas citadas, OpenMPI e OpenMP, nos permitem explorar quase em totalidade o paralelismo em nosso *cluster* de computadores, fazendo com que tiremos um proveito considerável de seu poder computacional. Seguem abaixo as implementações utilizadas para execução do algoritmo:

- Para a biblioteca de imagens *imagelib.h* fica a responsabilidade de leitura e operações sobre os arquivos de imagem, além de operações necessárias para o processamento das mesmas. Sua implementação está disponível no Anexo 8.1.

- Uma estrutura necessária para a implementação do algoritmo de busca de componentes conexos é uma fila para *pixels* pertencentes ao componente, com os métodos relativos a *fifo_pix_queue.h*. A implementação completa está disponível no Anexo 8.2.

- Uma estrutura de fila auxiliar foi criada para ajudar o processo mestre a dividir os blocos das imagens, mantendo o maior fluxo possível para os processos. Os métodos implementados estão em *list_link.h* mostrado abaixo. Implementação disponível no Anexo 8.3.

Estas três partes do algoritmo funcionam de maneira integrada, com intuito de executar todas as operações necessárias para a saída dos resultados. Temos ainda a lógica principal do processo, de onde os processos são iniciados e terminados, como visto em *main.c* no Anexo 8.4.

## 3.1 Utilizando OpenMPI

A utilização do OpenMPI se dá pela necessidade de se trocar mensagens entre os processos e se controlar por exemplo, envios e recebimentos, sendo bloqueantes ou não bloqueantes. No caso da implementação, foi usado um de troca chamado *"persistent connection"*, o qual permite uma troca de mensagens por um canal persistente de comunicação entre dois ou mais processos - no nosso caso, dois processos por canal, sendo o mestre e um outro processo trabalhador.

Este tipo de implementação nos permite utilizar métodos bloqueantes ou não bloquenates de envio e recebimento de mensagens e confirmações. Foi feita a utilização de métodos como *MPI_Startall* que é um método de envio não bloqueante, e *MPI_Waitall* para sincronização dos processos. Uma rotina não bloqueante permite que um processo espere por uma resposta de um determinado processo enquanto executa outras operações com outros processos, recebendo uma resposta de confirmação (*acknowledgement*) em um *buffer* a ser lido posteriormente.

## 3.2 Utilizando OpenMP

As repetições paralelas do OpenMP são utilizadas em três momentos do código: no momento da cópia dos dados da imagem para um bloco a ser enviado, no momento de se aplicar o método de *rec. 601* para passar a imagem de RGB para níveis de cinza, e no momento de ser "binarizar"os níveis de cinza a imagem utilizando um fator limítrofe. Uma vez que estas execuções são feitas sobre todos os dados do bloco, sua importância final para o resultado se torna significante.

Tais operações são utilizadas nestes instantes por não haver dependência na manipulação dos seus dados, sendo todas elas operações pontuais. Operações que ocasionam corrida

de dados não são permitidas em nenhum momento dentro da execução de processos isolados, bem como laços paralelos não podem ser aceitos sob as implementações de filas não bloqueantes já explicitadas.

### 3.3 Binarizando a imagem

Primeiramente, usa-se um algoritmo para transformar a imagem RGB com 8 bits de nível (valores de 0 a 255) para um único valor em níveis de cinza com 8 bits, estilo PGM. Essa mudança é feita utilizando uma padronização chamada "Rec.601", que utiliza uma soma ponderada dos valores RGB para um *pixel* $p$ qualquer, como mostrado na equação abaixo:

$$Rec.601(p) = 0.2989 * p.R + 0.5870 * p.G + 0.1140 * p.B \tag{1}$$

Após então, a binarização dos valores de $p$, $b(p)$, foi feita utilizando-se de uma simples divisão de faixas de valores, dado pela regra abaixo:

$$b(p) = \begin{cases} p > 255 * 0.4 & \implies 1 \\ \text{senão} & \implies 0 \end{cases} \tag{2}$$

# 4 MÉTODO

Os programas paralelos, no modelo a serem testados, estão sujeitos principalmente aos seguintes fatores:

1. número de processos
2. granularidade
3. tempo de comunicação

Uma vez que o tempo de comunicação foge do nosso campo de estudo e controle, o mesmo não será focado. Sendo assim, será feita variação no número de processadores (ou núcleos) utilizados para a execução do algoritmo, e na granularidade escolhida, ou seja o tamanho dos blocos a serem passados a ciclo de execução dos processadores. Desta maneira, o tempo de comunicação poderá ser subentendido pelo *overhead* causado por exemplo por uma granularidade muito fina evidenciada no tempo de execução.

Para as execuções do algoritmo serão utilizados blocos de diversos tamanhos e diferentes números de processos, mapeados um para cada núcleo disponível e capaz de processador os dados simultaneamente. Para as análises de métricas, não se faz sentido comparar execuções de diferentes granularidades para diferentes números de processos, portanto uma granularidade de blocos de 2000x2000 será fixada para os mesmos afim de se obter algo comparável.

Vale ressaltar, que dentro de um único processador que será mapeado diretamente para um processo, o tempo de comunicação está diretamente ligado com a capacidade dos barramentos de fazer a conexão, e num *cluster* como o que será utilizados, ele está diretamente ligado a capacidade da rede de interconexão dos computadores - no nosso caso com uma taxa de transferência de $1Gbit$.

# 5 RESULTADO

Os testes foram executados em um computador com um processador multinúcleo i7-4790k com 16GB de RAM, e em um *cluster* composto de 4 máquinas iguais, cada uma dispondo da mesma configuração com 8GB de RAM e processador i7-7700. O tempo de execução do algoritmo sequencial no processador multinúcleo em máquina única foi de 597.18 segundos, enquanto o tempo de execução sequencial em um computador do *cluster* utilizado foi de 570.36 segundos.

Para meios de teste e validação do algoritmo, diversas execuções foram feitas em um único processador multinúcleo antes de se executarem testes maiores utilizando-se do *cluster* disponibilizado. Os parâmetros utilizados para a execução dos testes em um único processador podem ser verificados na Tabela 1, enquanto os parâmetros para a utilização do *cluster* na Tabela 2.

| Processos | Dim. Bloco (pixels) | Tempo (segundos) |
|---|---|---|
| 2 | 100x100 | 614.87 |
| 3 | 100x100 | 2811.35 |
| 4 | 100x100 | 2815.49 |
| 2 | 200x200 | 584.64 |
| 3 | 200x200 | 903.32 |
| 4 | 200x200 | 989.93 |
| 2 | 300x300 | 558.29 |
| ... | ... | ... |
| 2 | 2000x2000 | 599.24 |
| 3 | 2000x2000 | 494.94 |
| 4 | 2000x2000 | 406.20 |

Tabela 1: Testes realizados em processador i7-4790k único com 4 núcleos, variando-se a granularidade e número de processos paralelos. Os dados completos podem ser visualizados no link `http://bit.ly/single_time_processor` bem como no Anexo 8.5 e tabela 6

| Processos | Dim. Bloco (pixels) | Tempo (segundos) |
|---|---|---|
| 10 | 500x500 | 944.07 |
| 15 | 500x500 | 829.02 |
| 10 | 1000x1000 | 663.95 |
| 15 | 1000x1000 | 629.60 |
| 10 | 1500x1500 | 614.54 |
| 15 | 1500x1500 | 589.94 |
| 10 | 2000x2000 | 585.98 |
| 15 | 2000x2000 | 596.94 |

Tabela 2: Testes realizados em um *cluster* de 4 máquinas cada uma equipada com um processador i7-7700 de 4 núcleos, variando-se granularidade e número de processos paralelos

A visualização em forma de tabela ajuda na verificação de valores numéricos quando se

procura algo específico, porém a representação pode ser um tanto quanto extensa a ponto ficar confusa em certo ponto. Para isso, várias ferramentas e representações gráficas foram criadas para a melhor análise de programas paralelos. No nosso caso, a representação em forma de gráfico de superfície com eixos X para processos, Y para tamanho de bloco e Z para tempo, deixa de forma clara e concisa a verificação de tendências nos algoritmos. Tais representações podem ser verificadas nas figuras 3 para o processador único e 4 para o *cluster*.

Se percebe pelos números em tabelas, mas também pelos gráficos, que o melhor desempenho para o processador único foi encontrado com 4 processos e um tamanho de bloco de 1000x1000, enquanto para o *cluster* esse ponto foi encontrado para 10 processos em um tamanho de bloco de 2000x2000.

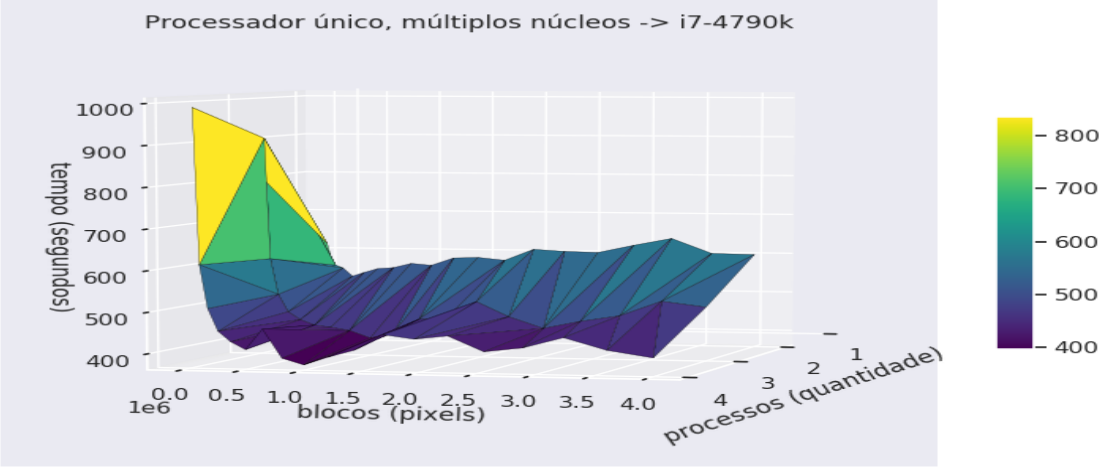


Figura 3: Gráfico da superfície formada pela execução com diversos tamanhos de bloco e número de processos para o processador único. Os valores para o tempo foram truncados em 1000 unidades, de forma a deixar as execuções 2 e 3 da Tabela 1 de fora, pois estes *outliers* causavam uma tremenda perda de detalhe na figura a ser demonstrada.

Na Tabela 3 estão o número de estrelas encontradas pelo algoritmo em cada uma das execuções pelo tamanho do bloco, assim sendo, se percebe que quanto menor o tamanho do bloco, mais há duplicações para o número de estrelas contadas em suas bordas, algo que poderia ser retirado com algumas técnicas e verificações mas não nos foi requisitada a implementação.

# 6 MÉTRICAS

As métricas utilizadas para se medir performance em algoritmos paralelas são usualmente: *Speedup* (S), Eficiência (E), Redundância (R), Utilização (U) e Qualidade (Q). Utilizando-se o tempo sequencial, $t(s)$, para um número de processadores $p$, dado um número $O$ de


Figura 4: Gráfico da superfície formada pela execução com diversos tamanhos de bloco e número de processos para o *cluster*

| Dim. Bloco (pixels) | Número de Estrelas |
|---|---|
| sequencial | 170,398,181 |
| 100x100 | 182,713,152 |
| 200x200 | 178,307,884 |
| 300x300 | 176,854,325 |
| 400x400 | 176,146,381 |
| 500x500 | 175,692,267 |
| 600x600 | 175,401,124 |
| 700x700 | 175,197,662 |
| 800x800 | 175,038,014 |
| 900x900 | 174,922,778 |
| ... | ... |
| 2000x2000 | 174,393,486 |

Tabela 3: Contagem do número de estrelas para diferentes tamanhos de bloco. Os dados completos da tabela podem ser visualizados no link `http://bit.ly/stars_count` bem como no Anexo 8.5 na Tabela 7

operações necessárias para se executar um dado algoritmo, as métricas podem ser definidas como a seguir:

$$S(p) = \frac{t(s)}{t(p)} \tag{3}$$

$$E(p) = \frac{S(p)}{p} = \frac{t(s)}{p * t(p)} \tag{4}$$

$$R(p) = \frac{O(p)}{O(s)} \tag{5}$$

$$U(p) = R(p) * E(p) \tag{6}$$

$$Q(U) = \frac{S(p) * E(p)}{R(p)} \tag{7}$$

Como pode-se ver nas fórmulas acima, duas métricas necessárias sequenciais necessárias para se calcular as métricas de performance em algoritmos paralelos são o tempo sequencial $t(s)$, e o número de operações sequenciais $O(s)$. Para o $t(s)$, temos para o computador e processador multinúcleo um valor de 557.57, e para um computador do *cluster* um valor de 570.36. Enquanto que para o número de operações, temos que $O(s)$ para ambos os casos se dá pelo número de operações sobre cada imagem, assim sendo: transformação em níveis de cinza, binarização e contagem de componentes conexos. Nos dando assim, para as 20 imagens do exemplo de resolução 29000x17380, um valor de

$$\begin{aligned} O(s) &= 29000 * 17380 * 20 * 3 \\ &= 302412 * 10^5 \end{aligned} \tag{8}$$

Para o algoritmo paralelo, temos que as operações de: divisão de blocos, transformação em níveis de cinza, binarização e contagem de componentes conexos, nos dando uma operação a mais sobre todos os dados da imagem. As mensagens trocadas pelos processos trabalhadores e mestre são: tamanho do bloco, o bloco e recebimento de valores, cada mensagem com recebimento e resposta de transferência correta. Assim sendo, temos 3 mensagens por processo trabalhador e 4 atividades em cada mensagem, nos dando um total de 12 mensagens por processo para processamento de cada bloco. Assim sendo e sabendo que utilizamos sempre blocos de 2000x2000 para todas as métricas, temos que a quantidade de operações no algoritmo paralelo é dada por:

$$\begin{aligned} O(p) &= 403216 * 10^5 + 12 * (29000 * 17380/2000 * 2000) * p \\ &\approx 403216 * 10^5 + 1512 * p \end{aligned} \tag{9}$$

sendo o termo da soma um número irrisório perto da magnitude dos dados, incluído apenas para completude.

Dada essa discussão, as métricas para a configurações de processadores e utilizando sempre a configuração de bloco 2000x2000 são mostradas para a execução em processador multinúcleo na Tabela 4 e para o *cluster* na Tabela 5 abaixo:

| Métrica/Processos | 2 | 3 | 4 |
|---|---|---|---|
| S(p) | 0.9304 | 1.1265 | 1.3726 |
| E(p) | 0.4652 | 0.3755 | 0.3431 |
| R(p) | 1.3333 | 1.3333 | 1.3333 |
| U(p) | 0.6202 | 0.5006 | 0.4574 |
| Q(p) | 0.3246 | 0.3172 | 0.3532 |

Tabela 4: Métricas para processador multinúcleo i7-4790k e blocos de tamanho 2000x2000

| Métrica/Processos | 10 | 15 |
|---|---|---|
| S(p) | 0.9733 | 0.9554 |
| E(p) | 0.0973 | 0.0636 |
| R(p) | 1.3333 | 1.3333 |
| U(p) | 0.1297 | 0.0847 |
| Q(p) | 0.0710 | 0.0455 |

Tabela 5: Métricas para *cluster* de 4 máquinas com processadores i7-7700 e blocos de tamanho 2000x2000

# 7 CONCLUSÃO

Se observa pelos resultados que a expertise de se utilizar sistemas paralelas é algo importante para se ter uma noção de resultados futuros. Ao longo do desenvolvimento, para mim e meu nível de conhecimento, achei que as implementações paralelas sempre levariam a resultados melhores do que os resultados sequenciais, o que de fato não ocorreu em nenhum dos cenários de testes - em computador único multinúcleo e *cluster* de máquinas.

É importante refletir sobre o que se leva aos resultados e aprender a identificar pontos críticos, para que no futuro quando tal conhecimento for necessário, saber aplicar técnicas com certeza e demonstrações concretas. Nos experimentos então, desenvolvi um senso crítico, além das técnicas, para saber como melhor tomar decisões em casos complexos.

# 8 ANEXOS

## 8.1 IMAGELIB

```
//
// Created by josetobias on 5/11/19.
//

#ifndef PARALLEL_COUNTING_STARS_IMAGELIB_H
#define PARALLEL_COUNTING_STARS_IMAGELIB_H

#include "fifo_pix_queue.h"
#include "list_link.h"
#include <png.h>
#include <stdlib.h>
#include <string.h>
#include <zlib.h>

typedef struct pix_ {
    int y;
    int x;
    int value;
    int label;
} pixel;

typedef struct figure_ {
    int dim_y;
    int dim_x;
    pixel **pixels;
} figure;

void read_png_file(char *file_path, int *width, int *height, png_byte *
    color_type, png_byte *bit_depth, png_byte ***row_pointers);

void binarizer(png_byte *row_pointers, figure **binary_comps);

void rec_601_gray(png_byte *row_pointers, figure binary_comps);

void divide_range(figure *binary_figure, double factor);

void divide_block(png_byte **src, png_byte *block, int y_init, int y_end, int
    x_init, int x_end);

int scc_count(figure **src);

void malloc_components(figure **src);

figure *initiate_figure(int dim_y, int dim_x);

png_byte *create_block(int stream_max_size);

png_byte **initiate_blocks(int size, int block_y, int block_x);
```

```
void block_list_linking(block_list **l, int dim_y, int dim_x, int block_y, int
    block_x);

void destroy_blocks(void **blocks, int size);

void destroy_figure(figure *src);

void destroy_components(pixel **src, int dim_y);

#endif //PARALLEL_COUNTING_STARS_IMAGELIB_H
```

Listing 1: imagelib.h

```
//
// Created by josetobias on 5/11/19.
//

#include "imagelib.h"

png_byte ***row_pointers) {
    FILE *fp = fopen(file_path, "rb");

    png_structp png = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL
    , NULL);
    if (!png)
        abort();

    png_infop info = png_create_info_struct(png);
    if (!info)
        abort();

    if (setjmp(png_jmpbuf(png)))
        abort();

    png_init_io(png, fp);

    png_read_info(png, info);

    *width = png_get_image_width(png, info);
    *height = png_get_image_height(png, info);
    *color_type = png_get_color_type(png, info);
    *bit_depth = png_get_bit_depth(png, info);

    // Read any color_type into 8bit depth, RGBA format.
    // See http://www.libpng.org/pub/png/libpng-manual.txt

    if (*bit_depth == 16)
        png_set_strip_16(png);

    if (*color_type == PNG_COLOR_TYPE_PALETTE)
        png_set_palette_to_rgb(png);

    if (*color_type == PNG_COLOR_TYPE_GRAY && *bit_depth < 8)
        png_set_expand_gray_1_2_4_to_8(png);
```

```
    if (png_get_valid(png, info, PNG_INFO_tRNS))
        png_set_tRNS_to_alpha(png);

    if (*color_type == PNG_COLOR_TYPE_RGB || *color_type ==
   PNG_COLOR_TYPE_GRAY ||
        *color_type == PNG_COLOR_TYPE_PALETTE)
        png_set_filler(png, 0xFF, PNG_FILLER_AFTER);

    if (*color_type == PNG_COLOR_TYPE_GRAY || *color_type ==
   PNG_COLOR_TYPE_GRAY_ALPHA)
        png_set_gray_to_rgb(png);

    png_read_update_info(png, info);

    *row_pointers = (png_byte **) malloc(sizeof(png_byte *) * *width);
    for (int y = 0; y < *height; y++)
        (*row_pointers)[y] = (png_byte *) malloc(png_get_rowbytes(png, info));

    png_read_image(png, *row_pointers);

    png_destroy_read_struct(&png, &info, NULL);

    fclose(fp);
}

void malloc_components(figure **src) {
    (*src)->pixels = (pixel **) malloc(sizeof(pixel *) * (*src)->dim_y);
    for (int y = 0; y < (*src)->dim_y; y++)
        (*src)->pixels[y] = (pixel *) malloc(sizeof(pixel) * (*src)->dim_x);
}

void binarizer(png_byte *row_pointers, figure **binary_comps) {
    // malloc components, transform the image to gray levels, and divide the
   range into binary to a factor
    malloc_components(&(*binary_comps));
    rec_601_gray(row_pointers, *(*binary_comps));
    divide_range(*binary_comps, 0.4);
}

void rec_601_gray(png_byte *row_pointers, figure binary_comps) {
#pragma omp parallel for
    for (int y = 0; y < binary_comps.dim_y; y++) {
        for (int x = 0; x < binary_comps.dim_x; x++) {
            png_byte *px = &(row_pointers[(y * binary_comps.dim_x * 3 + x * 3)
   ]);
            binary_comps.pixels[y][x].value = (0.2989 * px[0]) + (0.5870 * px
   [1]) + (0.1140 * px[2]);
            binary_comps.pixels[y][x].label = -1;
            binary_comps.pixels[y][x].y = y;
            binary_comps.pixels[y][x].x = x;
        }
    }
}
```

```
fifo_queue *create_queue() {
    fifo_queue *q = (fifo_queue *) malloc(sizeof(fifo_queue));
    q->size = 0;
    return q;
}

void divide_range(figure *binary_figure, double factor) {
#pragma omg parallel for
    for (int y = 0; y < (*binary_figure).dim_y; y++) {
        for (int x = 0; x < (*binary_figure).dim_x; x++) {
            // binarize value to a factor chosen
            if ((*binary_figure).pixels[y][x].value > 255 * factor)
                (*binary_figure).pixels[y][x].value = 1;
            else
                (*binary_figure).pixels[y][x].value = 0;
        }
    }
}

int scc_count(figure **src) {
    fifo_queue *q = create_queue();
    int label = 0;

    for (int y = 0; y < (*src)->dim_y; y++) {
        for (int x = 0; x < (*src)->dim_x; x++) {
            if (((*src)->pixels[y][x].value == 1) && ((*src)->pixels[y][x].
   label == -1)) {
                pixel *p = &(*src)->pixels[y][x];
                enqueue(&q, p);

                // keep expanding the label for all pixels in the 8-neighbor
   component
                while (q->size != 0) {
                    pixel *c = dequeue(&q);
                    c->label = label + 1;

                    pixel *l, *r, *u, *d, *ru, *rd, *lu, *ld;
                    if ((c->y - 1) >= 0) {
                        u = &(*src)->pixels[c->y - 1][c->x];
                        if (u->value == 1 && u->label == -1) {
                            u->label = label;
                            enqueue(&q, u);
                        }

                        if ((c->x + 1) < (*src)->dim_x) {
                            ru = &(*src)->pixels[c->y - 1][c->x + 1];
                            if (ru->value == 1 && ru->label == -1) {
                                ru->label = label;
                                enqueue(&q, ru);
                            }
                        }

                        if ((c->x - 1) < (*src)->dim_x) {
```

```
                            lu = &(*src)->pixels[c->y - 1][c->x - 1];
                            if (lu->value == 1 && lu->label == -1) {
                                lu->label = label;
                                enqueue(&q, lu);
                            }
                        }
                    }

                    if ((c->x - 1) >= 0) {
                        l = &(*src)->pixels[c->y][c->x - 1];
                        if (l->value == 1 && l->label == -1) {
                            l->label = label;
                            enqueue(&q, l);
                        }
                    }

                    if ((c->x + 1) < (*src)->dim_x) {
                        r = &(*src)->pixels[c->y][c->x + 1];
                        if (r->value == 1 && r->label == -1) {
                            r->label = label;
                            enqueue(&q, r);
                        }
                    }

                    if ((c->y + 1) < (*src)->dim_y) {
                        d = &(*src)->pixels[c->y + 1][c->x];
                        if (d->value == 1 && d->label == -1) {
                            d->label = label;
                            enqueue(&q, d);
                        }

                        if ((c->x - 1) < (*src)->dim_x) {
                            ld = &(*src)->pixels[c->y + 1][c->x - 1];
                            if (ld->value == 1 && ld->label == -1) {
                                ld->label = label;
                                enqueue(&q, ld);
                            }
                        }

                        if ((c->x + 1) < (*src)->dim_x) {
                            rd = &(*src)->pixels[c->y + 1][c->x + 1];
                            if (rd->value == 1 && rd->label == -1) {
                                rd->label = label;
                                enqueue(&q, rd);
                            }
                        }
                    }
                }
                label++;
            }
        }
    }
    free(q);
    return label;
```

```
}

void destroy_figure(figure *src) {
    destroy_components(src->pixels, src->dim_y);
    free(src);
}

void destroy_components(pixel **src, int dim_y) {
    for (int y = 0; y < dim_y; y++)
        free(src[y]);

    free(src);
}

png_byte **initiate_blocks(int size, int block_y, int block_x) {
    png_byte **blocks = (png_byte **) malloc(sizeof(png_byte *) * size);
    for (int y = 0; y < size; y++)
        blocks[y] = (png_byte *) malloc(sizeof(png_byte) * (block_y * block_x
   * 3));

    return blocks;
}

png_byte *create_block(int stream_max_size) {
    // allocate memory only for values RGB of pixels, ignoring A values
    png_byte *block = (png_byte *) malloc(sizeof(png_byte) * (stream_max_size
   * 3));

    return block;
}

void divide_block(png_byte **src, png_byte *block, int y_init, int y_end, int
   x_init, int x_end) {
    int dim_y = (y_end - y_init);
    int dim_x = (x_end - x_init);

#pragma omg parallel for
    for (int y = 0; y < dim_y; y++) {
        for (int x = 0; x < dim_x; x++) {
            // create block divisions ignoring A values, using only RGB then
            png_byte *ptr = &(src[y_init + y][(x_init + x * 4)]);
            block[(y * dim_x * 3 + x * 3)] = ptr[0];
            block[(y * dim_x * 3 + x * 3 + 1)] = ptr[1];
            block[(y * dim_x * 3 + x * 3 + 2)] = ptr[2];
        }
    }
}

figure *initiate_figure(int dim_y, int dim_x) {
    figure *fig = (figure *) malloc(sizeof(figure));

    fig->dim_y = dim_y;
    fig->dim_x = dim_x;
```

```
    return fig;
}

void destroy_blocks(void **blocks, int size) {
    for (int y = 0; y < size; y++)
        free(blocks[y]);

    free(blocks);
}

void block_list_linking(block_list **l, int dim_y, int dim_x, int block_y, int
    block_x) {
    for (int y = 0; y < dim_y; y += block_y) {
        for (int x = 0; x < dim_x; x += block_x) {
            // generate y boundaries
            int y_init = y;
            int y_end = y + block_y;
            if (y_end > dim_y)
                y_end = dim_y;

            // generate x boundaries
            int x_init = x;
            int x_end = x + block_x;
            if (x_end > dim_x)
                x_end = dim_x;

            // insert block into queue
            inlink((*l), y_init, y_end, x_init, x_end);
        }
    }
}
```

Listing 2: imagelib.c

## 8.2 FIFO PIX QUEUE

```
//
// Created by josetobias on 5/12/19.
//

#ifndef PARALLEL_COUNTING_STARS_FIFO_PIX_QUEUE_H
#define PARALLEL_COUNTING_STARS_FIFO_PIX_QUEUE_H

#include "imagelib.h"

typedef struct q_comp_ {
    struct pix_ *c;
    struct q_comp_ *last;
} q_comp;

typedef struct fifo_queue_ {
    int size;
```

```
    q_comp *head;
} fifo_queue;

void enqueue(fifo_queue **q, struct pix_ *c);

struct pix_ *
dequeue(fifo_queue **q);

#endif //PARALLEL_COUNTING_STARS_FIFO_PIX_QUEUE_H
```

Listing 3: fifo_pix_queue.h

```
//
// Created by josetobias on 5/12/19.
//

#include "fifo_pix_queue.h"

void enqueue(fifo_queue **q, pixel *c) {
    q_comp *new_comp = (q_comp *) malloc(sizeof(q_comp));
    new_comp->c = c;
    if ((*q)->size == 0) {
        new_comp->last = NULL;
        (*q)->head = new_comp;
    } else {
        new_comp->last = (*q)->head;
        (*q)->head = new_comp;
    }
    (*q)->size++;
}

pixel *dequeue(fifo_queue **q) {
    if ((*q)->size == 0) {
        return NULL;
    } else {
        q_comp *tmp = (*q)->head;
        (*q)->head = tmp->last;
        struct pix_ *comp = tmp->c;
        free(tmp);
        (*q)->size--;
        return comp;
    }
}
```

Listing 4: fifo_pix_queue.h

```
//
// Created by josetobias on 5/12/19.
//

#include "fifo_pix_queue.h"

void enqueue(fifo_queue **q, pixel *c) {
    q_comp *new_comp = (q_comp *) malloc(sizeof(q_comp));
```

```
    new_comp->c = c;
    if ((*q)->size == 0) {
        new_comp->last = NULL;
        (*q)->head = new_comp;
    } else {
        new_comp->last = (*q)->head;
        (*q)->head = new_comp;
    }
    (*q)->size++;
}

pixel *dequeue(fifo_queue **q) {
    if ((*q)->size == 0) {
        return NULL;
    } else {
        q_comp *tmp = (*q)->head;
        (*q)->head = tmp->last;
        struct pix_ *comp = tmp->c;
        free(tmp);
        (*q)->size--;
        return comp;
    }
}
```

Listing 5: fifo_pix_queue.c

## 8.3 LIST LINK

```
//
// Created by jose on 5/16/19.
//

#ifndef PARALLEL_COUNTING_STARS_LIST_LINK_H
#define PARALLEL_COUNTING_STARS_LIST_LINK_H

#include <stdlib.h>

typedef struct block_link_ {
    int y_init, y_end;
    int x_init, x_end;
    struct block_link_ *next;
} block_link;

typedef struct block_list_ {
    int size;
    struct block_link_ *first;
} block_list;

void
inlink(struct block_list_ *l, int y_init, int y_end, int x_init, int x_end);

struct block_link_ *
```

```
outlink(struct block_list_ *l);

#endif //PARALLEL_COUNTING_STARS_LIST_LINK_H
```

Listing 6: list_link.h

```
//
// Created by jose on 16/05/19.
//

#include "list_link.h"

void inlink(struct block_list_ *l, int y_init, int y_end, int x_init, int
    x_end) {
    block_link *new = (block_link *) malloc(sizeof(block_link));

    new->y_init = y_init;
    new->y_end = y_end;
    new->x_init = x_init;
    new->x_end = x_end;
    if (l->size == 0) {
        new->next = NULL;
        l->first = new;
    } else {
        new->next = l->first;
        l->first = new;
    }
    l->size++;
}

struct block_link_ * outlink(struct block_list_ *l) {
    if (l->size == 0) {
        abort();
    } else {
        struct block_link_ *ret = l->first;
        l->first = ret->next;
        l->size--;
        return ret;
    }
}
```

Listing 7: list_link.c

## 8.4 MAIN

```
#include "imagelib.h"
#include "list_link.h"
#include "mpi.h"
#include <dirent.h>
#include <png.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

#define MASTER 0
#define DEBUG 0
#define VERBOSE 0

int main(int argc, char **argv) {
    // MPI initialization
    MPI_Init(&argc, &argv);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    /* Setup */
    if (world_rank == MASTER) {
        char *dir_path = argv[1];
        DIR *dir;
        struct dirent *sd;

        if ((dir = opendir(dir_path)) == NULL) {
            printf("Problem opening directory <%s> - program will abort.\n",
   dir_path);
            abort();
        }

        printf("Establishing connection with processes...\n");

        int stars_total = 0;

        // needed block parameters passing
        int stars_image = 0;
        int block_y, block_x;
        block_y = atoi(argv[2]);
        block_x = atoi(argv[3]);
        int stream_max_size = block_y * block_x;
        MPI_Bcast(&stream_max_size, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

        // MPI Persistent mode configuration
        MPI_Request req_send_stars_block[(world_size - 1)];
        MPI_Status stts_send_stars_block[(world_size - 1)];

        MPI_Request req_recv_count[(world_size - 1)];
        MPI_Status stts_recv_count[(world_size - 1)];

        MPI_Request req_send_block_size[(world_size - 1)];
        MPI_Status stts_send_block_size[(world_size - 1)];

        png_byte **block = initiate_blocks((world_size - 1), block_y, block_x)
   ;
        int **block_size = (int **) malloc(sizeof(int *) * (world_size - 1));
        for (int p = 0; p < world_size - 1; p++)
            block_size[p] = (int *) malloc(sizeof(int) * 2);
```

```
    int *stars_counted = (int *) calloc((world_size - 1), sizeof(int));

    for (int p = 1; p < world_size; p++) {
        MPI_Send_init(block_size[(p - 1)], 2, MPI_INT, p, 0,
MPI_COMM_WORLD, &req_send_block_size[(p - 1)]);
        MPI_Send_init(block[(p - 1)], (block_x * block_y * 3),
MPI_UNSIGNED_CHAR, p, 0, MPI_COMM_WORLD,
                      &req_send_stars_block[(p - 1)]);
        MPI_Recv_init(&stars_counted[(p - 1)], 1, MPI_INT, p, 0,
MPI_COMM_WORLD, &req_recv_count[(p - 1)]);
        stts_send_stars_block[(p - 1)]._cancelled = 0;
    }

    printf("Connections successfully established.\n");
    printf("Processing files in folder <%s> with blocksize <%d, %d>...\n",
 dir_path, block_y, block_x);
    while ((sd = readdir(dir)) != NULL) {
        // skip directory self pointer and exit to parent
        if (!(strcmp(sd->d_name, ".") && strcmp(sd->d_name, "..")))
            continue;

        // build file path
        char *file_path = (char *) malloc(sizeof(char) * strlen(dir_path)
+ strlen(sd->d_name));
        strcpy(file_path, dir_path);
        strcat(file_path, sd->d_name);
        if (VERBOSE >= 1)
            printf("Executing algorithm for file %s...\n", sd->d_name);

        // setup variables
        int width, height;
        png_byte color_type, bit_depth;
        png_byte **row_pointers;
        read_png_file(file_path, &width, &height, &color_type, &bit_depth,
 &row_pointers);

        block_list *l = (block_list *) malloc(sizeof(block_list));
        l->size = 0;
        block_list_linking(&l, height, width, block_y, block_x);

        // log
        if (VERBOSE >= 2) {
            printf("width = %d\n", width);
            printf("height = %d\n", height);
            printf("color_type = %d\n", color_type);
            printf("bit_depth = %d\n", bit_depth);
        }
        if (DEBUG)
            getchar();

        /* Execution */
        do {
            int p;
            for (p = 0; (p < (world_size - 1)) && (l->size > 0); p++) {
```

```
                    block_link *bl = outlink(l);
                    block_size[p][0] = (bl->y_end - bl->y_init);
                    block_size[p][1] = (bl->x_end - bl->x_init);
                    divide_block(row_pointers, block[p], bl->y_init, bl->y_end
, bl->x_init, bl->x_end);
                    if (VERBOSE >= 3) {
                        printf("send y_init = %d\n", bl->y_init);
                        printf("send y_end = %d\n", bl->y_end);
                        printf("send x_init = %d\n", bl->x_init);
                        printf("send x_end = %d\n", bl->x_end);
                    }
                    if (DEBUG)
                        getchar();
                    free(bl);
                }
                if (p == (world_size - 1)) {
                    MPI_Startall((world_size - 1), req_send_block_size);
                    MPI_Startall((world_size - 1), req_send_stars_block);
                    MPI_Startall((world_size - 1), req_recv_count);

                    MPI_Waitall((world_size - 1), req_send_block_size,
stts_send_block_size);
                    MPI_Waitall((world_size - 1), req_send_stars_block,
stts_send_stars_block);
                    MPI_Waitall((world_size - 1), req_recv_count,
stts_recv_count);
                } else {
                    for (int pc = 0; pc < p; pc++) {
                        MPI_Start(&req_send_block_size[pc]);
                        MPI_Start(&req_send_stars_block[pc]);
                        MPI_Start(&req_recv_count[pc]);

                        MPI_Wait(&req_send_block_size[pc], &
stts_send_block_size[pc]);
                        MPI_Wait(&req_send_stars_block[pc], &
stts_send_stars_block[pc]);
                        MPI_Wait(&req_recv_count[pc], &stts_recv_count[pc]);
                    }
                }
                for (int pc = 0; pc < p; pc++)
                    stars_image += stars_counted[pc];
            } while (l->size > 0);

            if (VERBOSE >= 1)
                printf("Image stars count = %d\n", stars_image);
            stars_total += stars_image;

            // program reset of needed variables
            stars_image = 0;
            destroy_blocks((void **) row_pointers, height);
            free(file_path);
            free(l);
        }
```

```
        for (int p = 0; p < (world_size - 1); p++) {
            MPI_Request_free(&req_send_block_size[p]);
            MPI_Wait(&req_send_block_size[p], &stts_send_block_size[p]);

            MPI_Request_free(&req_send_stars_block[p]);
            MPI_Wait(&req_send_stars_block[p], &stts_send_stars_block[p]);

            MPI_Request_free(&req_recv_count[p]);
            MPI_Wait(&req_recv_count[p], &stts_recv_count[p]);
        }

        printf("Result of stars found = %d\n", stars_total);
        closedir(dir);

        // abort execution
        MPI_Abort(MPI_COMM_WORLD, 0);
    } else {
        // needed block parameters receiving
        int stream_max_size;
        MPI_Bcast(&stream_max_size, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

        // MPI Persistent mode configuration
        MPI_Request req_recv_block_size;
        MPI_Status stts_recv_block_size;

        MPI_Request req_recv_block;
        MPI_Status stts_recv_block;

        MPI_Request req_send_count;
        MPI_Status stts_send_count;

        png_byte *stars_received = create_block(stream_max_size);
        int *block_size = (int *) malloc(sizeof(int) * 2);
        int stars_send = 0;

        MPI_Recv_init(block_size, 2, MPI_INT, MASTER, 0, MPI_COMM_WORLD, &
    req_recv_block_size);
        MPI_Recv_init(stars_received, (stream_max_size * 4), MPI_UNSIGNED_CHAR
    , MASTER, 0, MPI_COMM_WORLD,
                      &req_recv_block);
        MPI_Send_init(&stars_send, 1, MPI_INT, MASTER, 0, MPI_COMM_WORLD, &
    req_send_count);

        /* Execution */
        while (1) {
            MPI_Start(&req_recv_block_size);
            MPI_Wait(&req_recv_block_size, &stts_recv_block_size);

            figure *binary_fig = initiate_figure(block_size[0], block_size[1])
    ;

            MPI_Start(&req_recv_block);
            MPI_Wait(&req_recv_block, &stts_recv_block);
```

```
            binarizer(stars_received, &binary_fig);
            stars_send = scc_count(&binary_fig);

            MPI_Start(&req_send_count);
            MPI_Wait(&req_send_count, &stts_send_count);

            destroy_figure(binary_fig);
        }
    }

    // MPI finalizer
    // a formality to have it here, but processes will most likely never come
    here
    MPI_Finalize();
    return 0;
}
```

Listing 8: main.c

## 8.5 DADOS DAS EXECUÇÕES

| Processos | Bloco | Tempo |
|---|---|---|
| 2 | 100x100 | 614,87 |
| 3 | 100x100 | 2811,35 |
| 4 | 100x100 | 2815,49 |
| 2 | 200x200 | 584,64 |
| 3 | 200x200 | 903,32 |
| 4 | 200x200 | 989,93 |
| 2 | 300x300 | 558,29 |
| 3 | 300x300 | 601,14 |
| 4 | 300x300 | 612,37 |
| 2 | 400x400 | 551 |
| 3 | 400x400 | 511,61 |
| 4 | 400x400 | 506,92 |
| 2 | 500x500 | 529,71 |
| 3 | 500x500 | 467,5 |
| 4 | 500x500 | 452,31 |
| 2 | 600x600 | 539,67 |
| 3 | 600x600 | 448,17 |
| 4 | 600x600 | 427,4 |
| 2 | 700x700 | 550,32 |
| 3 | 700x700 | 434,96 |
| 4 | 700x700 | 409,04 |
| 2 | 800x800 | 553,52 |
| 3 | 800x800 | 430,03 |
| 4 | 800x800 | 459,37 |
| 2 | 900x900 | 565,12 |
| 3 | 900x900 | 422,75 |
| 4 | 900x900 | 389,63 |
| 2 | 1000x1000 | 560,93 |
| 3 | 1000x1000 | 404,49 |
| 4 | 1000x1000 | 374,58 |
| 2 | 1100x1100 | 580,85 |
| 3 | 1100x1100 | 418,77 |
| 4 | 1100x1100 | 394,1 |
| 2 | 1200x1200 | 582,98 |
| 3 | 1200x1200 | 453,46 |
| 4 | 1200x1200 | 411,91 |

| Processos | Bloco | Tempo |
|---|---|---|
| 2 | 1300x1300 | 576,51 |
| 3 | 1300x1300 | 457,41 |
| 4 | 1300x1300 | 450,53 |
| 2 | 1400x1400 | 606 |
| 3 | 1400x1400 | 500,16 |
| 4 | 1400x1400 | 441,07 |
| 2 | 1500x1500 | 601,59 |
| 3 | 1500x1500 | 463,72 |
| 4 | 1500x1500 | 450,01 |
| 2 | 1600x1600 | 599,79 |
| 3 | 1600x1600 | 435,9 |
| 4 | 1600x1600 | 413,78 |
| 2 | 1700x1700 | 619,49 |
| 3 | 1700x1700 | 454,88 |
| 4 | 1700x1700 | 424,42 |
| 2 | 1800x1800 | 638,31 |
| 3 | 1800x1800 | 470,28 |
| 4 | 1800x1800 | 450,62 |
| 2 | 1900x1900 | 600,91 |
| 3 | 1900x1900 | 507,91 |
| 4 | 1900x1900 | 422,19 |
| 2 | 2000x2000 | 599,24 |
| 3 | 2000x2000 | 494,94 |
| 4 | 2000x2000 | 406,2 |

Tabela 6: Tabela com tempos de processador único i7-4790k

| block size | count |
|---|---|
| 100x100 | 182,713,152 |
| 200x200 | 178,307,884 |
| 300x300 | 176,854,325 |
| 400x400 | 176,146,381 |
| 500x500 | 175,692,267 |
| 600x600 | 175,401,124 |
| 700x700 | 175,197,662 |
| 800x800 | 175,038,014 |
| 900x900 | 174,922,778 |
| 1000x1000 | 174,809,230 |
| 1100x1100 | 174,735,086 |
| 1200x1200 | 174,677,759 |
| 1300x1300 | 174,629,219 |
| 1400x1400 | 174,571,432 |
| 1500x1500 | 173,547,773 |
| 1600x1600 | 174,511,414 |
| 1700x1700 | 174,487,360 |
| 1800x1800 | 174,444,393 |
| 1900x1900 | 174,435,228 |
| 2000x2000 | 174,393,486 |

Tabela 7: Contagem do número de estrela para execuções com tamanhos de blocos diferentes.