Universidade de Brasília Faculdade UnB Gama

Programação para Sistemas Paralelos e Distribuídos
Prof.: Fernando W. Cruz
Aluno: _____ Matrícula: _____

Laboratório p entrega

Os fractais são baseados no princípio de que um objeto geométrico pode ser dividido em partes menores, cada uma delas semelhante ao objeto original. São, portanto, objetos com muitos detalhes, com similaridade recursiva. Um dos fractais interessantes de se observar é o fractal Julia. Um conjunto Julia (*Julia set*) é uma generalização do famoso conjunto Mandelbrot [https://pt.wikipedia.org/wiki/Conjunto_de_Mandelbrot], que está ilustrado na Figura 1.

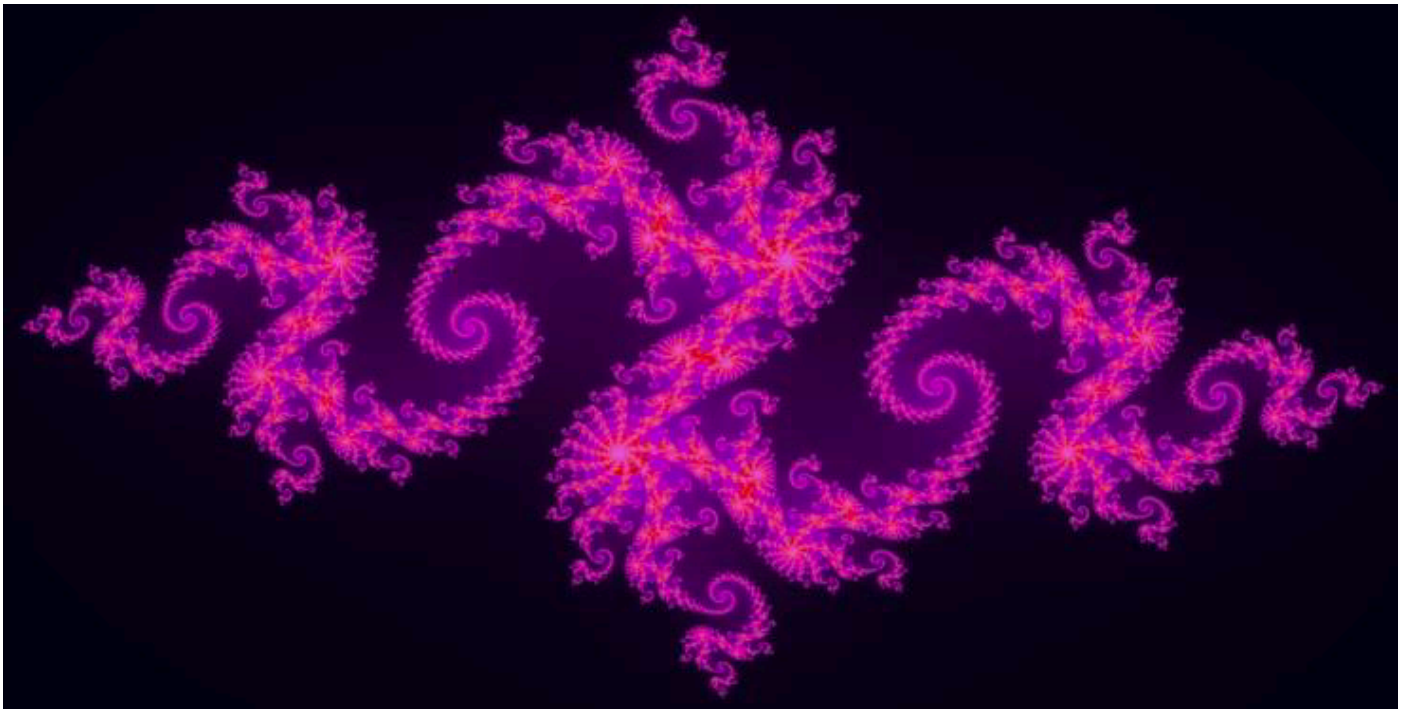

Figura 1 – Fractal Julia

Esse fractal é definido como segue. Dado `z` um ponto no plano complexo 2-D, calculamos a série definida como: `z0 = z` e `zn+1 = zn**2 + c`, onde `c = -0,79 + i * 0,15`, ou seja, um número complexo. Valores diferentes de `c` levam a imagens diferentes, e conhecemos muitos valores que produzem imagens "bonitas". A cor de um pixel correspondente ao ponto z é determinada com base no número de iterações antes que o módulo de `zn` seja superior a 2 (ou até que um número máximo de iterações seja atingido). O programa `fractal.c` é o código que produz a imagem da Figura 1, mas é possível alterá-lo para criar imagens diferentes. Este programa pode ser compilado com o parâmetro a seguir:

```
$ gcc fractal.c -o fractal -lm
```

Para executá-lo, basta digitar o comando

```
$ ./fractal <N>
```

onde N é a altura da figura do fractal (ou número de linhas).

Esse parâmetro é utilizado para o cálculo da largura (2*N) e o cálculo da área do fractal (altura * largura * 3). A saída desse programa é um arquivo em formato bmp (Bitmap) que pode ser aberto com qualquer editor de imagens do seu sistema operacional.

Com base nestas informações iniciais, resolva as questões 1 e 2 a seguir. Os arquivos fonte (.c) dos códigos e READMES com explicações devem ser zipados em um arquivo único, a ser postado no Moodle.

1. Escreva uma versão MPI do código `fractal.c`, com o nome `fractalmpiserial.c`, de modo que N seja dividido pelo número de processos criados e a gravação do arquivo aconteça serialmente e em ordem do menor *rank* para o maior. Por exemplo, se o programa foi chamado da seguinte forma:

```
$ mpirun -np 4 fractalmpiserial 1000
```

Significa que a altura do fractal é 1000 linhas e cada *rank* (calculado em função do parâmetro np) irá gravar 250 linhas no arquivo de saída, de forma não simultânea (paralela), mas serial, na seguinte ordem: *(1º)* o *rank* 0 escreve as linhas de 0 a 249; *(2º)* o *rank* 1 escreve as linhas 250 a 499; *(3º)* o rank 2 escreve de 500 a 749 e, *(4º)* o rank 3 escreve as linhas de 750 a 999.

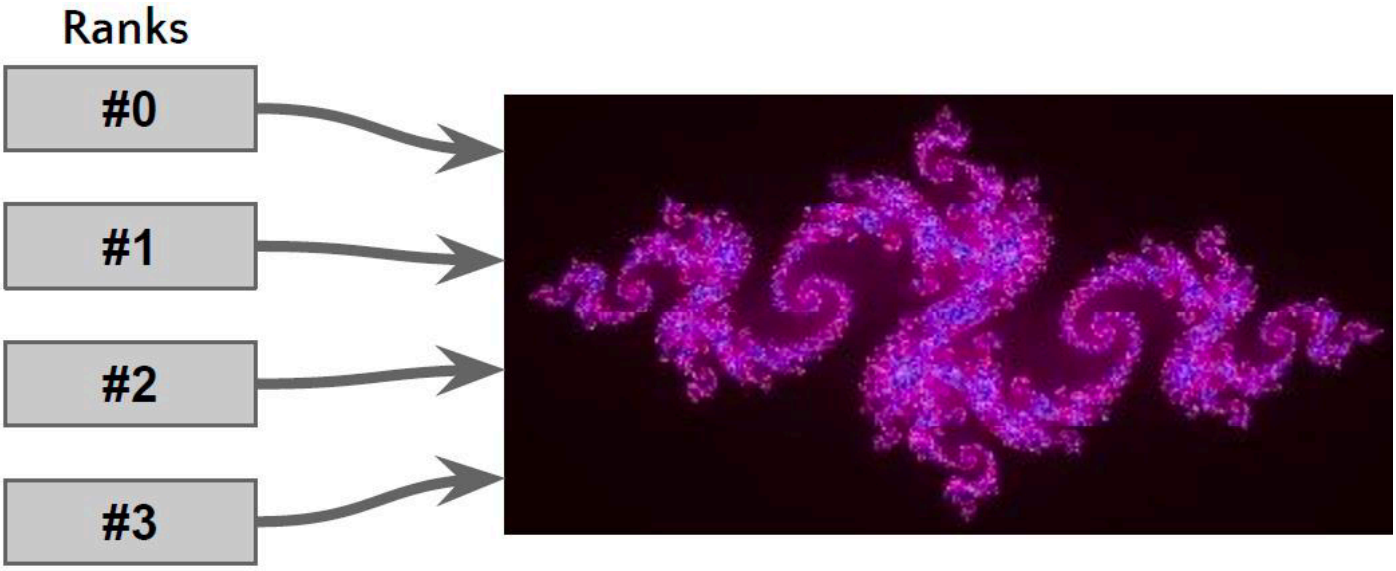

Figura 2 – Exemplo de execução MPI do Fractal Julia

Neste caso, a execução do código deve ser feita no cluster chococino, de modo que cada processo fique ativo em um *host* específico e cada um deles fique responsável por uma porção do arquivo de saída. Use um editor de imagens para ver a figura resultante produzida.

2. Gerar uma segunda versão MPI do código fractal.c, com o nome `fractalmpi_io.c`, de modo que N seja dividido pelo número de processos, mas a gravação aconteça em paralelo, obedecendo o *offset* calculado para cada *rank*, e fazendo uso de alguma função de escrita de arquivo da biblioteca MPI. Conforme apresentado na Tabela 1, são muitas as funções de I/O MPI, mas nem todas elas servem para o problema apresentado neste experimento. Portanto, a sugestão é fazer uma pesquisa, antes de adotar uma solução factível. Para esta questão, os alunos devem tentar resolver o problema com pelo menos duas funções de gravação. Além disto, cada função escolhida deve ser documentada, com explicação associada sobre parâmetros e forma de uso.

Tabela 1 – Funções MPI de I/O (Input/Output) em arquivos

positioning	synchronism	coordination	
		noncollective	collective
explicit offsets	blocking	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	nonblocking	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_iread_at_all MPI_File_iwrite_at_all
	split collective	N/A	MPI_File_read_at_all_begin/end MPI_File_write_at_all_begin/end
individual file pointers	blocking	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	nonblocking	MPI_File_iread MPI_File_iwrite	MPI_File_iread_all MPI_File_iwrite_all
	split collective	N/A	MPI_File_read_all_begin/end MPI_File_write_all_begin/end
shared file pointer	blocking	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	nonblocking	MPI_File_iread_shared MPI_File_iwrite_shared	N/A
	split collective	N/A	MPI_File_read_ordered_begin/end MPI_File_write_ordered_begin/end

Anexo - fractal.c - versão não-MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define OUTFILE "out_julia_normal.bmp"

int compute_julia_pixel(int x, int y, int largura, int altura, float tint_bias,
unsigned char *rgb) {
  // Check coordinates
  if ((x < 0) || (x >= largura) || (y < 0) || (y >= altura)) {
    fprintf(stderr,"Invalid (%d,%d) pixel coordinates in a %d x %d image\n", x, y,
largura, altura);
    return -1;
  }
  // "Zoom in" to a pleasing view of the Julia set
  float X_MIN = -1.6, X_MAX = 1.6, Y_MIN = -0.9, Y_MAX = +0.9;
  float float_y = (Y_MAX - Y_MIN) * (float)y / altura + Y_MIN ;
  float float_x = (X_MAX - X_MIN) * (float)x / largura  + X_MIN ;
  // Point that defines the Julia set
  float julia_real = -.79;
  float julia_img = .15;
  // Maximum number of iteration
  int max_iter = 300;
  // Compute the complex series convergence
  float real=float_y, img=float_x;
  int num_iter = max_iter;
  while (( img * img + real * real < 2 * 2 ) && ( num_iter > 0 )) {
    float xtemp = img * img - real * real + julia_real;
    real = 2 * img * real + julia_img;
    img = xtemp;
    num_iter--;
  }

  // Paint pixel based on how many iterations were used, using some funky colors
  float color_bias = (float) num_iter / max_iter;
  rgb[0] = (num_iter == 0 ? 200 : - 500.0 * pow(tint_bias, 1.2) *  pow(color_bias,
1.6));
  rgb[1] = (num_iter == 0 ? 100 : -255.0 *  pow(color_bias, 0.3));
  rgb[2] = (num_iter == 0 ? 100 : 255 - 255.0 * pow(tint_bias, 1.2) * pow(color_bias,
3.0));

  return 0;
} /*fim compute julia pixel */

int write_bmp_header(FILE *f, int largura, int altura) {

  unsigned int row_size_in_bytes = largura * 3 +
    ((largura * 3) % 4 == 0 ? 0 : (4 - (largura * 3) % 4));

  // Define all fields in the bmp header
  char id[2] = "BM";
  unsigned int filesize = 54 + (int)(row_size_in_bytes * altura * sizeof(char));
  short reserved[2] = {0,0};
  unsigned int offset = 54;

  unsigned int size = 40;
  unsigned short planes = 1;
  unsigned short bits = 24;
  unsigned int compression = 0;
  unsigned int image_size = largura * altura * 3 * sizeof(char);
  int x_res = 0;
```

```
  int y_res = 0;
  unsigned int ncolors = 0;
  unsigned int importantcolors = 0;

  // Write the bytes to the file, keeping track of the
  // number of written "objects"
  size_t ret = 0;
  ret += fwrite(id, sizeof(char), 2, f);
  ret += fwrite(&filesize, sizeof(int), 1, f);
  ret += fwrite(reserved, sizeof(short), 2, f);
  ret += fwrite(&offset, sizeof(int), 1, f);
  ret += fwrite(&size, sizeof(int), 1, f);
  ret += fwrite(&largura, sizeof(int), 1, f);
  ret += fwrite(&altura, sizeof(int), 1, f);
  ret += fwrite(&planes, sizeof(short), 1, f);
  ret += fwrite(&bits, sizeof(short), 1, f);
  ret += fwrite(&compression, sizeof(int), 1, f);
  ret += fwrite(&image_size, sizeof(int), 1, f);
  ret += fwrite(&x_res, sizeof(int), 1, f);
  ret += fwrite(&y_res, sizeof(int), 1, f);
  ret += fwrite(&ncolors, sizeof(int), 1, f);
  ret += fwrite(&importantcolors, sizeof(int), 1, f);

  // Success means that we wrote 17 "objects" successfully
  return (ret != 17);
} /* fim write bmp-header */

int main(int argc, char *argv[]) {
    int n;
    int area=0, largura = 0, altura = 0, local_i= 0;
    FILE *output_file;
    unsigned char *pixel_array, *rgb;

    if ((argc <= 1)||(atoi(argv[1])<1)){
        fprintf(stderr,"Entre 'N' como um inteiro positivo! \n");
        return -1; }
    n = atoi(argv[1]);
    altura = n; largura = 2*n; area=altura*largura*3;
    //Allocate mem for the pixels array
    pixel_array= calloc(area, sizeof(unsigned char));
    rgb = calloc(3, sizeof(unsigned char));
    printf("Computando linhas de pixel %d até %d, para uma área total de %d\n", 0,
n-1, area);

    for (int i = 0; i < n; i++)
        for (int j = 0; j < largura*3; j+=3){
            compute_julia_pixel(j/3, i, largura, altura, 1.0, rgb);
            pixel_array[local_i]= rgb[0]; local_i++;
            pixel_array[local_i]= rgb[1]; local_i++;
            pixel_array[local_i]= rgb[2]; local_i++;
        }
    //Release mem for the pixels array
    free(rgb);
    //escreve o cabeçalho do arquivo
    output_file= fopen(OUTFILE, "w");
    write_bmp_header(output_file, largura, altura);
    //escreve o array no arquivo
    fwrite(pixel_array, sizeof(unsigned char), area, output_file);
    fclose(output_file);
    free(pixel_array);
    return 0;
} /* fim-programa */
```