TDT4200 --- Fall 2024
PS 1b -- Mandelbrot Fractals & MPI

Author: Anne C. Elster. Due date: Sept 9 by 10pm. Grading: Pass/Fail.

This problem set / exercise give you an opportunity to write an MPI program from scratch based on a cool program that has a nice graphical interface, like you had for PS0.

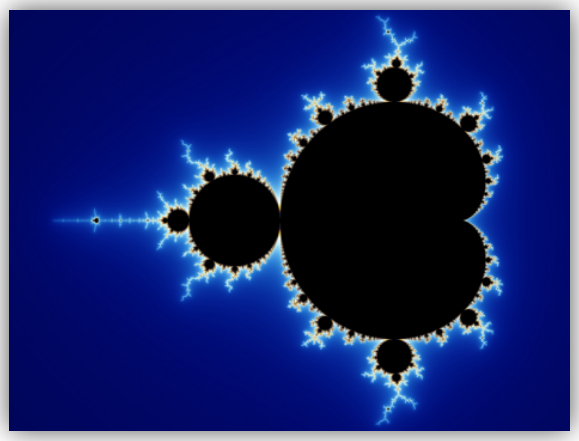
This Problem Set is Optional, and as such will not be graded in detail. However, we strongly recommend you do this assignment to familiarize yourself with MPI and our compute cluster for teaching, Snotra. See also information under Course info on BlackBoard.on how you log into Snotra.

This assignment, like the others, should be done individually and handed in by the posted deadline, preferably by 10pm on Monday Sept. 9.

Introduction

The Mandelbrot set is one of the most well-known mathematical visualizations methods used to generate fractal images – images that have "infinite resolution", i.e. for each step it gets more detailed. It is described by the quadratic polynomial $z_{n+1} = z_n^2 + c$, where c is a part of the Mandelbrot set if the absolute value of z_n remains bounded.

To achieve great level of detail (LOD), a lot of computations are required. However, **by utilizing the power of the GPU, amazingly detailed fractal images can be generated on-the-fly. We will be using the GPU for this in PS5.**



3783OT_01_01.png

Figure 1. Fractal image of a Mandelbrot set from Wiki
http://en.wikipedia.org/wiki/Mandelbrot_set

Another animated fractal generation is available here:

http://en.wikipedia.org/wiki/File:Animation_of_the_growth_of_the_Mandelbrot_set_as_you_iterate _towards_infinity.gif

More info on generalization and implementation:

http://en.wikipedia.org/wiki/Mandelbrot_set#Generalizations

Setting up benchmark basic C program (CPU) – mandel-cpu.c

In this section, we will describe how we developed the Mandelbrot CPU-based C code from scratch. We wrote this code as a benchmark to run against the MPI code you will be developing.

The main difference between the codes provided by the CUDA SDK and the ones we develop here, is that the code we develop here has fewer options and uses a more classical dive-and-conquer parallelization technique. This makes the codes easier to understand when explaining the main differences between CPU C codes and MPI/GPU CUDA codes. It also makes them more efficient for the cases they handle.

The codes we present here were developed by Ruben Spaans as a programming assignment Dr. Elster asked him to developed for her parallel computing course. The codes are based on the Mandelbrot algorithm described on the Wiki http://en.wikipedia.org/wiki/Mandelbrot_set.

C Implementation – step-by-step

To help students get into C, we have here included the core implementation steps. A complete listing of the mandel_cpu.c – a the CPU C source code we include in a that Ruben Spaans wrote as part of our assignment based on the Mandelbrot algorithm for generation Mandelbrot sets provided on the Wiki http://en.wikipedia.org/wiki/Mandelbrot_set, is included as a separate file.

1. After using `#include` to include the standard C library headerfiles "stdio.h", stdlib.h"and "math.h", set the size of the image in x and y as well as MAXITER, the set the maximum number of iterations, as follows:

```
#define XSIZE 2560
#define YSIZE 2048

#define MAXITER 255
```

2. The **c** values are typically scaled so that the real part is contained in [−2, 1] and the imaginary part is contained in [−1,1]. The variables xleft, xright and ycenter can be changed if you want to draw another portion of the Mandelbrot fractal., but we have defined them as follows:

```
double xleft=-2.01;
double xright=1;
double yupper,ylower;
double ycenter=1e-6;
```

3. The array `pixel` is one-dimensional, and is arranged in a row-major order, which means that coordinate (x, y) is stored in index x + y*XSIZE. The provided macro `INDEX` converts a pair of coordinates (x, y) to a one-dimensional index (see the main loop of the function calculate() for an example usage).

```
int pixel[XSIZE*YSIZE];
#define INDEX(i,j) ((i)+(j)*XSIZE)
```

4. The core of the Mandelbrot algorithm in implemented in the function “calculate” is a classical C code that follows the following Mandelbrot algorithm. Here, the coordinates of a given point in a Mandelbrot set are represented by a complex number **c** where the real and imaginary parts can be interpreted as the x and y-coordinates respectively. Let the input be **c** and the output **z** and for each pixel:

> 1. Let z ← c and iteration← 0. Define max. number of iterations MAX you want to perform.
>
> 2. If |z| ≥ 2 or iteration = MAX then terminate.
>
> 3. Let z ← z2 + c.
>
> 4. Let iteration←iteration+1.
>
> 5. Go to 2

5. If z never diverges (that is, its absolute value never exceeds 2) the point **z** is said to be in the Mandelbrot set. Otherwise, the number of iterations is the escape time for this point, and it will be used to determine the pixel color. The absolute value of a complex number a + bi is defined as $\sqrt{a^2 + b^2}$. In our implementation, we use the equivalent and computationally cheaper condition $a^2 + b^2 \geq 4$ to check for divergence.

6. Be aware that the first parameter is the x coordinate, and the second parameter is the y coordinate:

```
void calculate() {
  for(inti=0;i<XSIZE;i++) {
      for(int j=0;j<YSIZE;j++) {
        complex_tc,z,temp;
        intiter=0;
        c.real = (xleft + step*i);
        c.imag = (ylower + step*j);
        z = c;
/* Calculate the number of iterations until divergence for each
pixel. If divergence never happens, return MAXITER */

        while(z.real*z.real + z.imag*z.imag<4) {
            temp.real = z.real*z.real - z.imag*z.imag + c.real;
            temp.imag = 2*z.real*z.imag + c.imag;
            z = temp;
            if(++iter==MAXITER) break;
          } /* end while */
          pixel[INDEX(i,j)]=iter;
      }/* end for YSIZE */
    } /* end for SIZE */
}
```

7. Function that sets the display color of a point based on the iteration number:

```
/* given iteration number, set a color */
void fancycolour(uchar *p,intiter) {
	if(iter==MAXITER);
	elseif(iter<8) { p[0]=128+iter*16; p[1]=p[2]=0; }
	elseif(iter<24) { p[0]=255; p[1]=p[2]=(iter-8)*16; }
	elseif(iter<160) { p[0]=p[1]=255-(iter-24)*2; p[2]=255; }
	else { p[0]=p[1]=(iter-160)*2; p[2]=255-(iter-160)*2; }
}
```

8. Save the result in to a 24-bit bmp file :

```
/* save 24-bits bmp file, buffer must be bmp format: upside-down
*/
void savebmp(char *name,uchar *buffer,int x,int y) {
	FILE *f=fopen(name,"wb");
	if(!f) {
		printf("Error writing image to disk.\n");
		return;
	}
	unsigned int size=x*y*3+54;
	uchar
header[54]={'B','M',size&255,(size>>8)&255,(size>>16)&255,size>>24
,0,

	0,0,0,54,0,0,0,40,0,0,0,x&255,x>>8,0,0,y&255,y>>8,0,0,1,0,2
4,0,0,0,0,0,0,
		0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	fwrite(header,1,54,f);
	fwrite(buffer,1,XSIZE*YSIZE*3,f);
	fclose(f);
}
```

Notice that the bitmap needs to be written “upside-down” so that it will be rendered correctly on the screen.

9. In the main function we calculate the range of the y-axis (yupper and ylower) in a way that preserves the aspect ratio:

```
/* Calculate range in y-axis to preserve aspect ratio */
	step=(xright-xleft)/XSIZE;
	yupper=ycenter+(step*YSIZE)/2;
	ylower=ycenter-(step*YSIZE)/2;
```

10. Finally, we let the user decide whether the image should be saved to disk, and make a call to “calculate”. If the user run the executable with the “1” option call to “fancycolor”, else save to disk with `savebmp`:

```
        if(strtol(argv[1],NULL,10)!=0) {/* if command_line entered
0
         create nice image from iteration counts.
         take care to create it upside down (bmp format) */
               unsigned char *buffer=calloc(XSIZE*YSIZE*3,1);
               for(int i=0;i<XSIZE;i++) {
                     for(int j=0;j<YSIZE;j++) {
                           int p=((YSIZE-j-1)*XSIZE+i)*3;
                           fancycolor(buffer+p,pixel[PIXEL(i,j)]);
                     }
               }
         /* write image to disk */
          savebmp("mandel2.bmp",buffer,XSIZE,YSIZE);
        }
```

Compiling and running this code with for instance gcc will generate a Mandelbrot image very similar to Figure 2. Unlike the SDK sample, the color settings and window size are here hard coded.

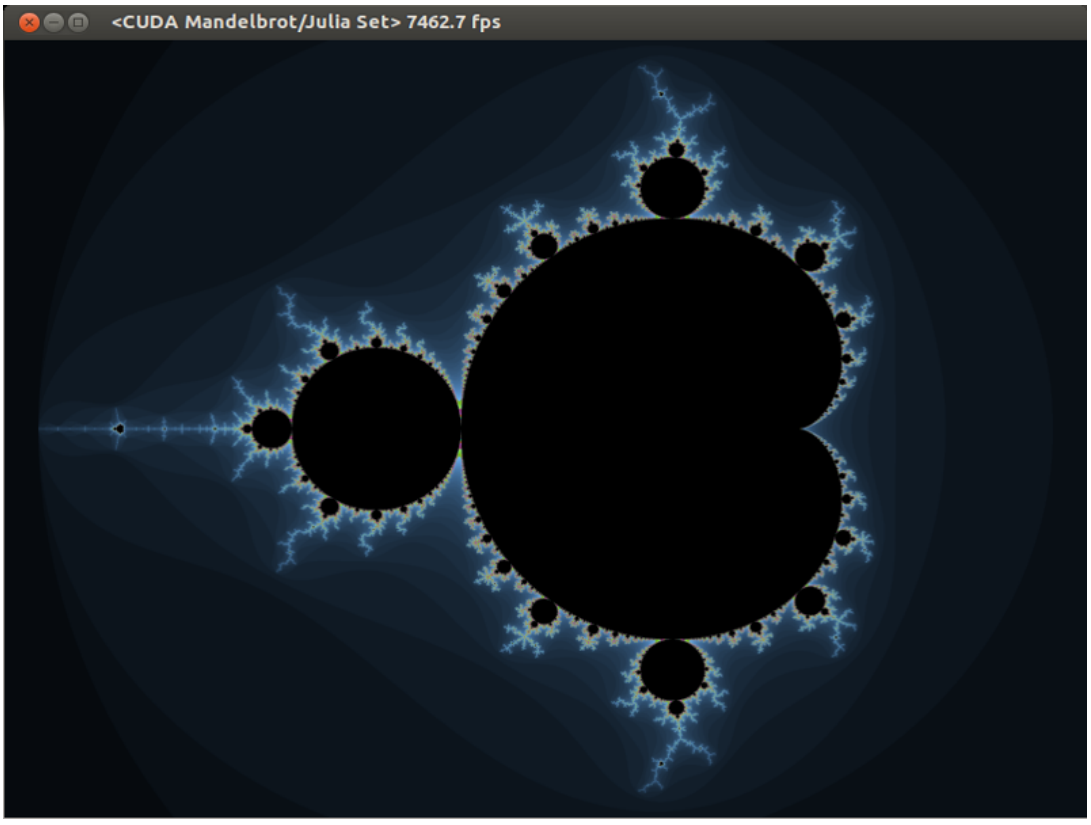


Figure 2 Mandelbrot set [output]

TO DO: Write an MPI version

Your task is to implement an MPI version of the provided C code that utilizes 3 processes.

(You can try more on your own computer).

1) Have each of the three ranks calculate a subset of the image.

   Hint: The easiest partition would be to give a subset of rows to each of the processes.

2) Have each of the processes send the result to Process 0 using MPI_Send and MPI-Recv
3) Have Process 0 print out the resulting image.

Extra: Add some timing functions and check if there is a speed-up.

You can also use MPI_Gather, or MPI_SendRcv, but for this assignment do the above first.

Questions to be answered in submitted PDF file:

1) Show the formula for Speed up. Did you get one and/or think you should have gotten one? Why or why not?

2) Explain the main difference between weak and strong scaling.

3) Explain the differences between SIMD and SPMD. Which one is MPI?