CPSC 474 - Project 2

GitHub url: https://github.com/CSUF-CPSC-Bein-FA21/project-2-cniles

Group Members: Dalton Caron, dpcaron@csu.fullerton.edu

Jeevika Yarlagadda, jeevikayarlagadda@csu.fullerton.edu

Submission for Project 2 - Distributed Mandelbrot Set

README.md

474-Project-2

Project 2: Implement a distributed algorithm using MPI

Distributed Mandelbrot Set

This project calculates calculates the mandelbrot set with arbitrary percision floating point numbers and distributes the work using MPI.

Group members

Dalton Caron dpcaron@csu.fullerton.edu Jeevika Yarlagadda jeevikayarlagadda@csu.fullerton.edu

Summary:

This project calculates mandelbrot set zooms with arbitrary precision floating point numbers and distributes the work using MPI. Zooming is performed automatically and by a factor of two each iteration. Images are distributed into partitions. A partition with the most entropy, relative to other partitions, is selected to be the image to partition further into deeper partitions of the mandelbrot set. The program is capable of supporting arbitrarily large zoom factors at the cost of performance. Each f(z) iteration may be executed in parallel, so it is easy to distribute the work among different processes. This computation is normally performed on a GPU to perform large batch computations of f(z). The GPU does not support large floating numbers, restricted to the 64 bit double. Using distributed systems, we can recapture lost performance when executing on the CPU and have arbitrary precision floating point numbers to calculate deeper zooms.

Pseudocode:

The below are the two main algorithms used: calculatePartition and performIterations. calculatePartiton is performed on all follower nodes and is the bottleneck of the algorithm. What is not shown in the pseudo code is that all multiplications are made using variable precision floating point numbers. Thus, the multiplication to perform f(z) is not a constant time operation. performIterations distributes the partitions to the follower processes and is looped depending on the amount of zooming desired. performItertions executes on the leader process and decides which image partition to elect for future zooming.

Algorithm 1: calculatePartition(c, offset, size, maxIterations)

Input : a coordinate c, offset = $\frac{boundary}{size-1}$, $size$ of image partition
Output: an image I of dimensions $size \times size$ and entropy e

```
1  cx ← c.re
2  for x ← 0 to size − 1 do
3      for y ← 0 to size − 1 do
4          z ← 0
5          for iteration ← 1 to maxIterations do
6              z ← z^2 + c
7              if |z| > 4 then
8                  break
9          if iteration = maxIterations then
10             I[x][y] = black
11         else
12             I[x][y] = colorFunction(iteration)
13         c.re = c.re + offset.re
14     c.re = cx
15     c.im = c.im − offset.im
16 Compute the entropy of I and store it as e
17 return I, e
```

Algorithm 2: performIteration(S)

Input : a datatructure of settings S
Output: a computed image I and coordinate of partition with most entropy c

```
1  Populate and send the partitions to the follower processes
2  Receive each partition into the set of partitions P
3  I ← (S.size, S.size)
4  maxEntropy ← −∞
5  foreach p ∈ P do
6      Combine partial solution p.I into I
7      if p.entropy > maxEntropy then
8          maxEntropy ← p.entropy
9          successorPartition ← p
10 c ← successorPartition.c
11 return I, c
```

Compile the Code and Run:

- Code is written in C.
- Used makefile for easy compilation.

The user must have GCC, libgmp, and make installed to compile the program. The user's compiler must comply with POSIX standards. To compile the program, use the Makefile as shown below.

Command to compile: make build

If the make file is not installed, please use the following command:

```
mpicc main.c libbmp.c leader.c common.c follower.c -lgmp -lm -g
```

This command produces the a.out executable file.

Command to run: make run

If the make file is not installed, please use the following command:

```
mpirun --mca opal_warn_on_missing_libcuda 0 --oversubscribe -n 5 a.out -i 1000 -z 3 -s 256 -e
```

Depending on the parameters, the algorithm may execute in a few seconds or few hours. The default parameters in the Makefile run in a few seconds on modern machines. The Makefile default is executed using the above command.

To change the number of processors or the iterations, please go to the Makefile file in the directory and change the inputs as mentioned in the readme file.

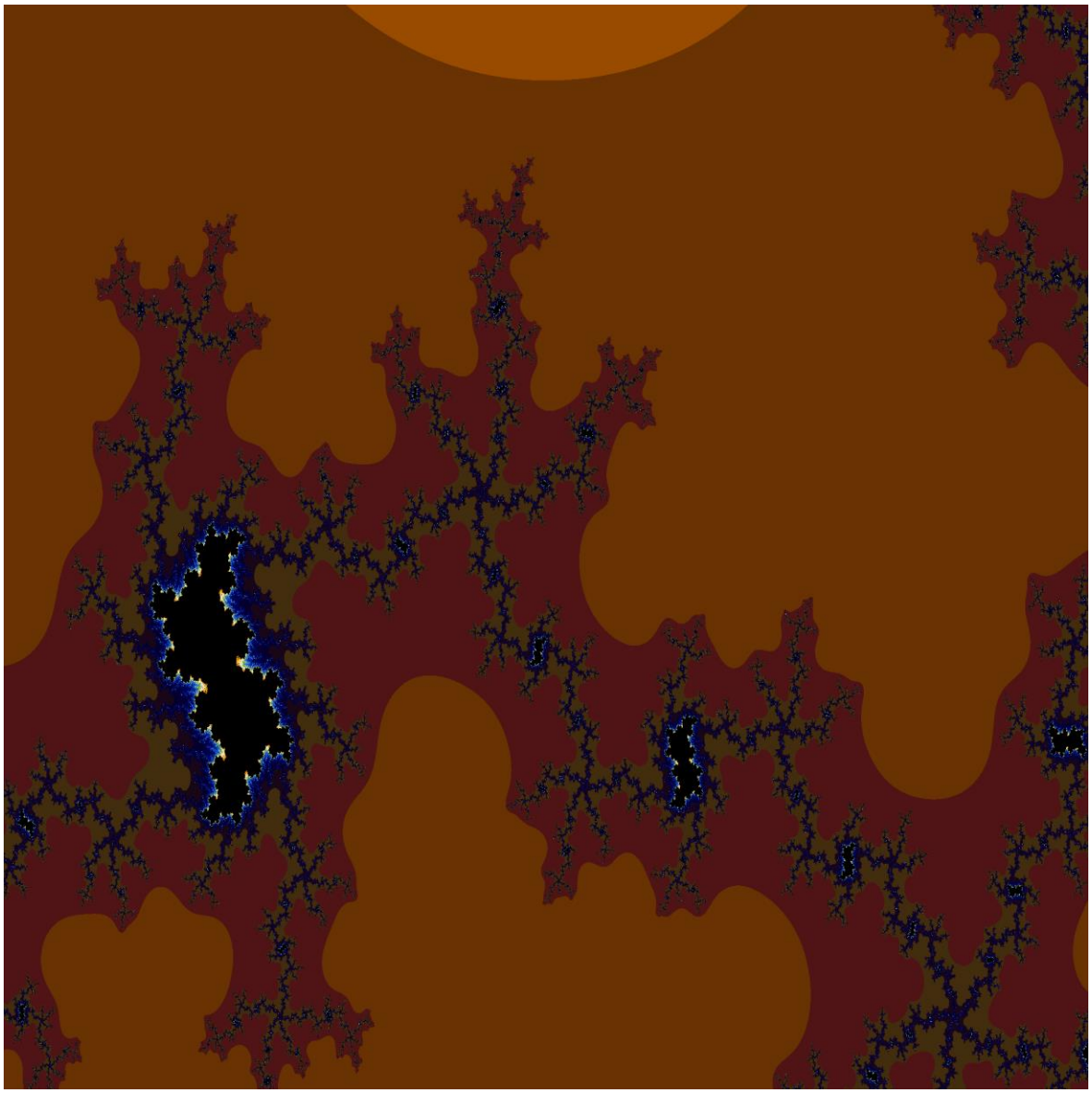
Snapshots of the code with test data:

Test data 1:

Command executed:

```
mpirun --mca opal_warn_on_missing_libcuda 0 --oversubscribe -n 17 a.out -i 2000 -z 17 -s 2048
```

Output image:

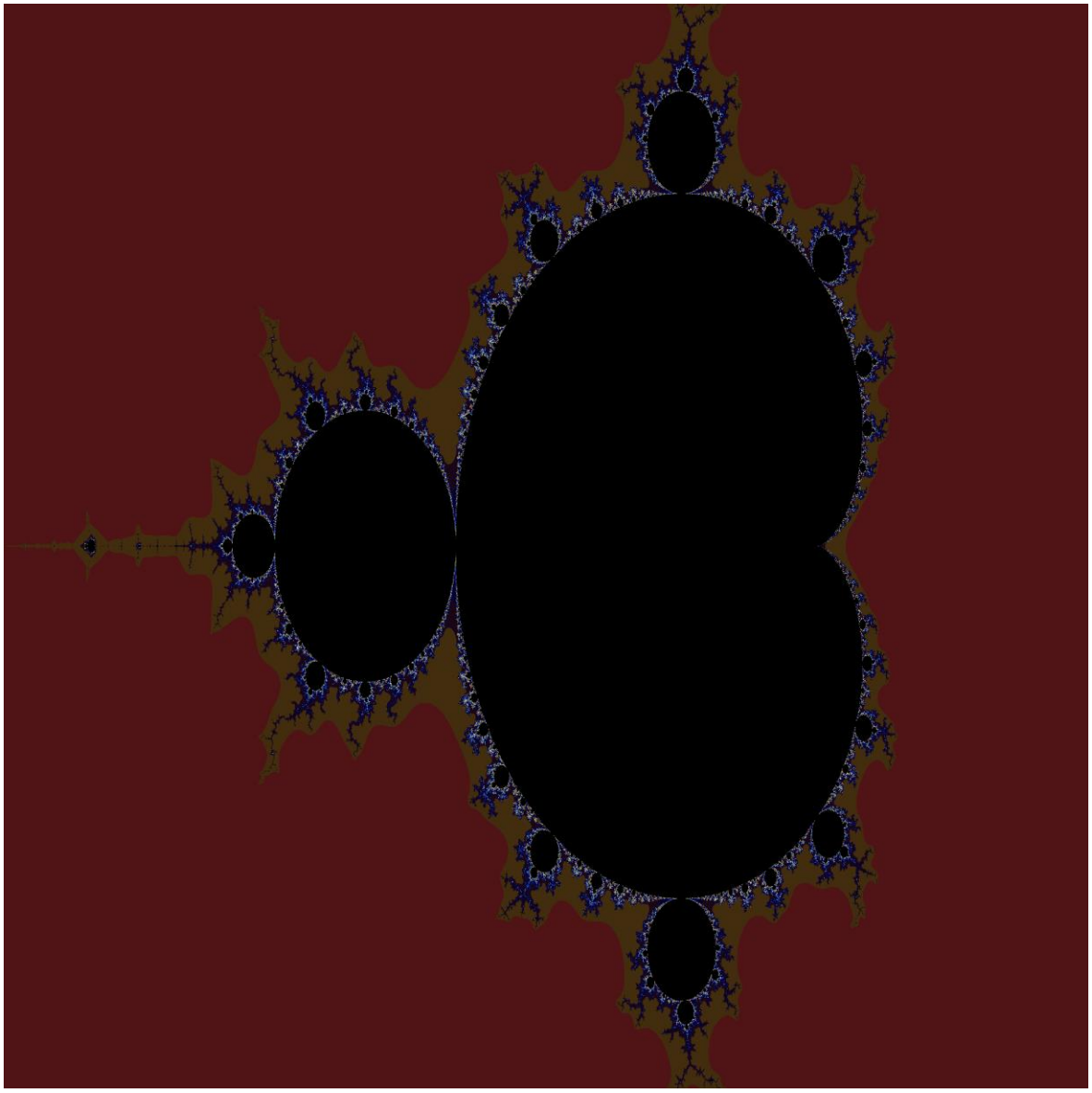


Test data 2:

Command executed:

```
mpirun --mca opal_warn_on_missing_libcuda 0 --oversubscribe -n 5 a.out -i 2000 -z 1 -s 2048
```

Output image:



Test data 3:

Command executed:

```
mpirun --mca opal_warn_on_missing_libcuda 0 --oversubscribe -n 5 a.out -i 20000 -z 40 -s 256
```

Output image: