Assignment 5

Stuart Hopkins, A02080107

February 2020

1 Mandelbrot

To create a parallel Mandelbrot program I use a master slave setup. The master distributes rows to the child processes to compute. The rows are gathered back to the master process. After the entire data set is calculated the data is printed to the picture file. The process then terminates. My code is as follows:

```
#include <iostream>
#include <fstream>
#include <time.h>
#include <mpi.h>
#include <unistd.h>
#include <stdlib.h>

#include <chrono>
using namespace std::chrono;

using namespace std;

#define MCW MPI_COMM_WORLD

using namespace std;

struct Complex {
  double r;
  double i;
};

Complex operator + (Complex s, Complex t) {
  Complex v;
  v.r = s.r + t.r;
  v.i = s.i + t.i;
  return v;
}

Complex operator * (Complex s, Complex t) {
  Complex v;
  v.r = s.r * t.r - s.i * t.i;
```

```
  v.i = s.r * t.i + s.i * t.r;
  return v;
}


int rcolor(int iters) {
  if(iters == 255)
    return 266;
  return 266 - 32 * (iters % 8);
}

int gcolor(int iters) {
  if(iters == 255)
    return 0;
  return 32 * (iters % 8);
}

int bcolor(int iters) {
  if(iters == 255)
    return 0;
  return 32 * (iters % 8);
}

int mbrotIters(Complex c, int maxIters) {
  int i = 0;
  Complex z;
  z = c;
  while(i < maxIters && z.r * z.r + z.i * z.i < 4) {
    z = z * z + c;
    i++;
  }
  return i;
}

// Use command line arguments to initialize main.
// (Real0, Imaginary0, Real1)
int main(int argc, char **argv){

  Complex c1, c2, c3;
  Complex c;
  int rank, size;
  const int DIM = 500;
  int data[DIM][DIM];
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MCW,&rank);
  MPI_Comm_size(MCW,&size);
  MPI_Request myRequest1, myRequest2, myRequest3;
  MPI_Status myStatus1, myStatus2, myStatus3;
```

```
int flag1 = 1;
int flag2 = 1;
int flag3 = 1;

if(argc == 4) {
  c1.r = atof(argv[1]);
  c1.i = atof(argv[2]);
  c2.r = atof(argv[3]);
  c2.i = atof(argv[3]) - atof(argv[1]);
}
else {
  c1.r = 2;
  c1.i = 2;
  c2.r = -2;
  c2.i = -2;
}

// Use process 0 as the master process
if(!rank) {
  auto start = high_resolution_clock::now();

  ofstream fout;
  fout.open("image.ppm");

  c3 = c1 + c2;
  cout << c3.r << " + " << c3.i << "i" << endl;

  c3 = c1 * c2;
  cout << c3.r << " * " << c3.i << "i" << endl;

  fout << "P3" << endl;
  fout << DIM << " " << DIM << endl;
  fout << 255 << endl;

  for (int i=0; i < DIM; ++i) {
    while(true) {
      if(flag1) {
        flag1 = 0;
        MPI_Send(&i, 1, MPI_INT, 1, 0, MCW);
        MPI_Irecv(&data[i], DIM, MPI_INT, 1, 1, MCW, &myRequest1);
        break;
      }
      else if(flag2) {
        flag2 = 0;
        MPI_Send(&i, 1, MPI_INT, 2, 0, MCW);
        MPI_Irecv(&data[i], DIM, MPI_INT, 2, 2, MCW, &myRequest2);
        break;
      }
```

```
      else if(flag3) {
        flag3 = 0;
        MPI_Send(&i, 1, MPI_INT, 3, 0, MCW);
        MPI_Irecv(&data[i], DIM, MPI_INT, 3, 3, MCW, &myRequest3);
        break;
      }

      MPI_Test(&myRequest1, &flag1, &myStatus1);
      MPI_Test(&myRequest2, &flag2, &myStatus2);
      MPI_Test(&myRequest3, &flag3, &myStatus3);
    }
  }

  for(int i = 1; i < size; i++) {
    int tmpData = -1;
    MPI_Send(&tmpData, 1, MPI_INT, i, 0, MCW);
  }

  for (int j=0; j < DIM; ++j) {
    for( int i=0; i < DIM; ++i) {
      fout << rcolor(data[j][i]) << " ";
      fout << gcolor(data[j][i]) << " ";
      fout << bcolor(data[j][i]) << " ";
    }
  }

  fout << endl;
  fout.close();

  auto stop = high_resolution_clock::now();
  auto duration = duration_cast<microseconds>(stop - start);
  cout << "TIME: " << duration.count() << " microseconds" << endl;
}


else {
  int recData = -1;
  while(true) {
    MPI_Recv(&recData, 1, MPI_INT, MPI_ANY_SOURCE, 0, MCW, MPI_STATUS_IGNORE);

    if(recData == -1)
      break;

    int iters[DIM];

    for( int i=0; i < DIM; ++i) {
      c.r = (i*(c1.r - c2.r) / DIM) + c2.r;
      c.i = (recData*(c1.i - c2.i) / DIM) + c2.i;
```

```
        iters[i] = mbrotIters(c, 255);
      }

      MPI_Send(&iters, DIM, MPI_INT, 0, rank, MCW);
    }
  }

  MPI_Finalize();
  return 0;
}
```

To verify it works here is the command line arguments and output. It is a requirement if useing custom numbers they are entered in the patter of: (Real0, Imaginary0, Real1). The program will compute the final Imiginary1 using the entered values.

It worked, here is the code:

```
$ mpic++ mbrot_parallel_row.cpp
$ mpirun -np 4 a.out 1 -1 1

2 + -1i
1 * -1i
TIME: 74508 microseconds
```

If we view the image it isn't too interesting but we can try it with the default values of 2 -2 2 by leaving the input blank. This gives the console output of:

```
$ mpic++ mbrot_parallel_row.cpp
$ mpirun -np 4 a.out

0 + 0i
0 * -8i
TIME: 96186 microseconds
```

Now that looks a lot better! To compare the performance we will run the included serial mandelbrot:

```
$ g++ mbrot_serial.cpp
$ /a.out

0 + 0i
0 * -8i
TIME: 218208 microseconds
```

The parallel program is obviously much faster! Lastly, it is a requirement to run this program with 4 processors.