Serial Mandelbrot

Aaron Morgenegg

10/5/18

OpenMPI Serial Mandelbrot

This program generates a mandelbrot image using multiple mpi processes. I ran several tests comparing the runtimes of the program at different resultions, and comparing it to the runtime of the serial version of the program. The results are below.

I ran both versions of the program on my old laptop with 2 cores. Still, the parallel version of the program ran in about 2/3's of the time. While not quite twice as fast as you would expect from 2 cores, there is some overhead to using MPI, as well as the fact that only process 0 is outputting the data to the file.

Resolution	512	1024	2048	4096	8192	16384
serial	.718	2.899	11.346	45.410	3:01.47	12:03.11
parallel	0.615	2.025	7.767	30.791	2:01.65	8:09.90

```
/*
Example of Compiling and running code.

mpic++ main.cpp
mpiexec -np 2 a.out

#/

#include <iostream>
#include <fstream>
#include <time.h>
#include <stdlib.h>
#include <mpi.h>
#include <unistd.h>
#include <algorithm>
#include <math.h>
#include <string>
#include <vector>

const int MAX_ITERATION = 1000;
const int RESOLUTION = 512;
const bool INVERT_COLORS = false;
const std::string OUTPUT_FILE = "parallel_mandelbrot.ppm";

struct Color{
public:
int r;
int g;
int b;

Color(){r,g,b=0;}

Color(int r, int g, int b){
this->r = r;
this->g = g;
this->b = b;
}
};
```

```
Color getColor(int iteration){
int r = iteration%145;
int g = iteration%200;
int b = iteration%255;
if(INVERT_COLORS){
r = 255 - r;
g = 255 - g;
b = 255 - b;
}
return Color(r,g,b);
}

std::ofstream setupFile(){
std::ofstream mandelbrot_file(OUTPUT_FILE);
mandelbrot_file << "P3" << std::endl;
mandelbrot_file << RESOLUTION << " " << RESOLUTION << std::endl;
mandelbrot_file << "255" << std::endl;
return mandelbrot_file;
}

void writeToFile(std::string message, std::ofstream &mandelbrot_file){
mandelbrot_file << message;
}

void plotImage(std::vector<int> plot){
std::ofstream mandelbrot_file = setupFile();
for(int i = 0; i < RESOLUTION; i++){
for(int j = 0; j < RESOLUTION; j++){
Color c = getColor(plot[i*RESOLUTION+j]);
std::string color = std::to_string(c.r) + " " + std::to_string(c.g) + " " + std::t
writeToFile(color, mandelbrot_file);
}
writeToFile("\n", mandelbrot_file);
}
mandelbrot_file.close();
}
```

```
int calculatePixel(int px, int py){
double x0 = -2 + px * (2.5/RESOLUTION);
double y0 = -1.25 + py * (2.5/RESOLUTION);
double x = 0.0;
double y = 0.0;
int iteration = 0;
while(x*x + y*y < 4 && iteration < MAX_ITERATION){
double xtemp = x*x - y*y + x0;
y = 2*x*y + y0;
x = xtemp;
iteration += 1;
}
return iteration;
}

std::vector<int> initPlot(int size){
std::vector<int> plot(size*RESOLUTION,0);
return plot;
}

void mandelbrot(int world_size, int world_rank){
int offset = RESOLUTION/world_size;
std::vector<int> plot = initPlot(RESOLUTION);
int start = world_rank*offset;
for(int i = start; i < start+offset; i++){
for(int j = 0; j < RESOLUTION; j++){
plot[i*RESOLUTION+j] = calculatePixel(j, i);
}
}
if(world_rank==0){
std::vector<int> recv_data = initPlot(RESOLUTION);
MPI_Gather(&plot[start*RESOLUTION],RESOLUTION*offset,MPI_INT,&recv_data.front(),RE
plotImage(recv_data);
} else{
MPI_Gather(&plot[start*RESOLUTION],RESOLUTION*offset,MPI_INT,NULL,RESOLUTION*offse
}
}
```

```
int main(int argc, char** argv) {
MPI_Init(&argc, &argv);
srand(time(NULL));
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

mandelbrot(world_size, world_rank);
MPI_Finalize();
return 0;
}
```