**Name: Abu Butt**

**Professor: Izidor Gertner**

**Class: Parallel Processing**

**Date: July 19, 2017**

**Topic: Mandelbrot Set**

**Introduction:**

The Mandelbrot set is used to refer both to a general class of fractal sets and to a particular instance of such a set. In general, a Mandelbrot set marks the set of points in the complex plane such that the corresponding Julia set is connected and not computable. The Mandelbrot set is obtained from the quadratic recurrence equation zn+1 = zn2 + C. with z0 = C, where points C in the complex plane for which the orbit of zn does not tend to infinity are in the set. Mandelbrot set images may be created by sampling the complex numbers and determining for each sample point c, where the result of iterating the above function goes to infinity.

**Mandelbrot Psuedocode:**

For each pixel (Px, Py) on the screen, do:

{

x0 = scaled x coordinate of pixel (scaled to lie in the Mandelbrot X scale (-2.5, 1))

y0 = scaled y coordinate of pixel (scaled to lie in the Mandelbrot Y scale (-1, 1))

x = 0.0

y = 0.0

iteration = 0

max\_iteration = 1000

// Here N=2^8 is chosen as a reasonable bailout radius.

while ( x\*x + y\*y < (1 << 16) AND iteration < max\_iteration ) {

xtemp = x\*x - y\*y + x0

y = 2\*x\*y + y0

x = xtemp

iteration = iteration + 1

}

// Used to avoid floating point issues with points inside the set.

if ( iteration < max\_iteration ) {

// sqrt of inner term removed using log simplification rules.

log\_zn = log( x\*x + y\*y ) / 2

nu = log( log\_zn / log(2) ) / log(2)

// Rearranging the potential function.

// Dividing log\_zn by log(2) instead of log(N = 1<<8)

// because we want the entire palette to range from the

// center to radius 2, NOT our bailout radius.

iteration = iteration + 1 - nu

}

color1 = palette[floor(iteration)]

color2 = palette[floor(iteration) + 1]

// iteration % 1 = fractional part of iteration.

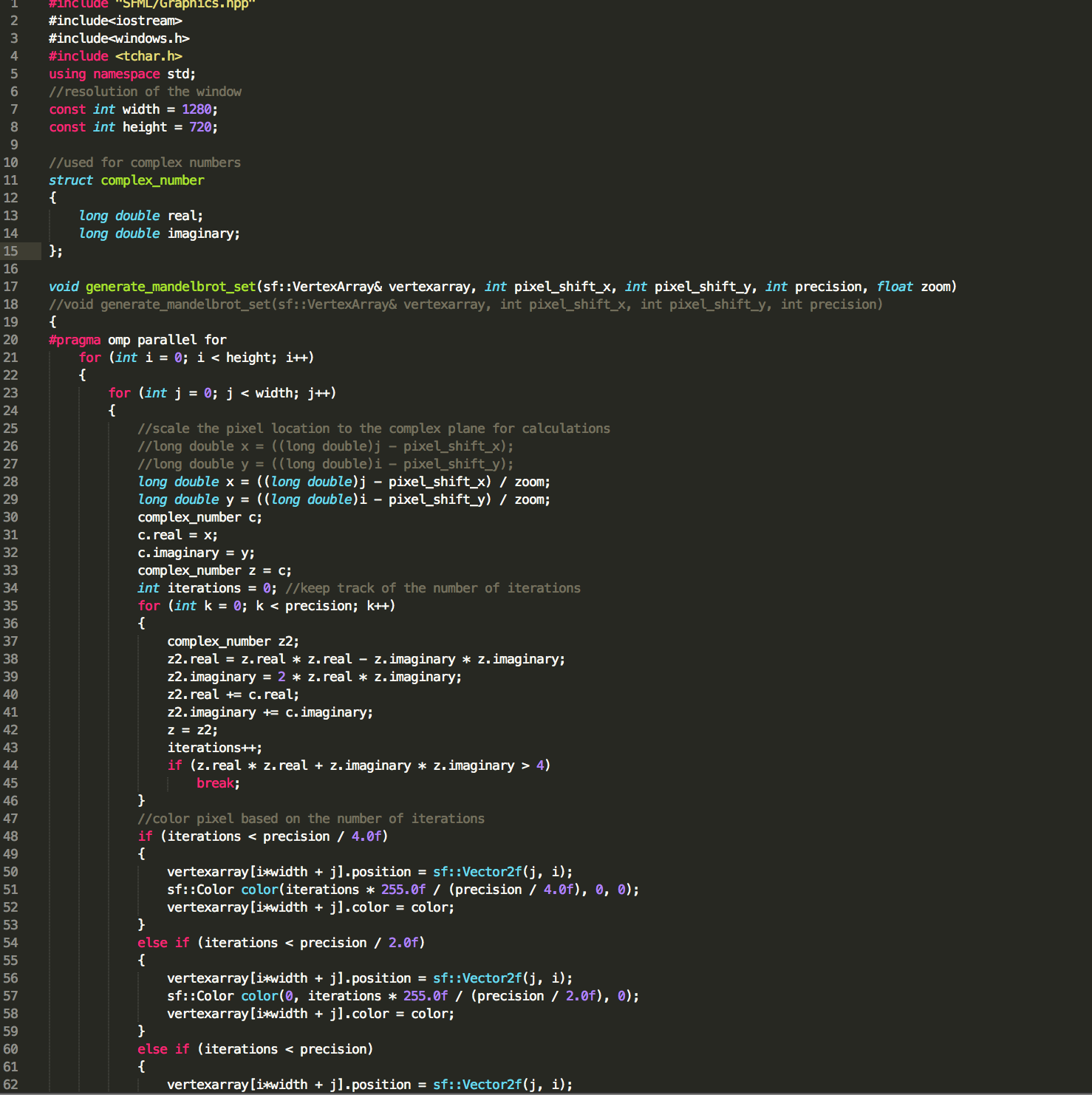
color = linear\_interpolate(color1, color2, iteration % 1)

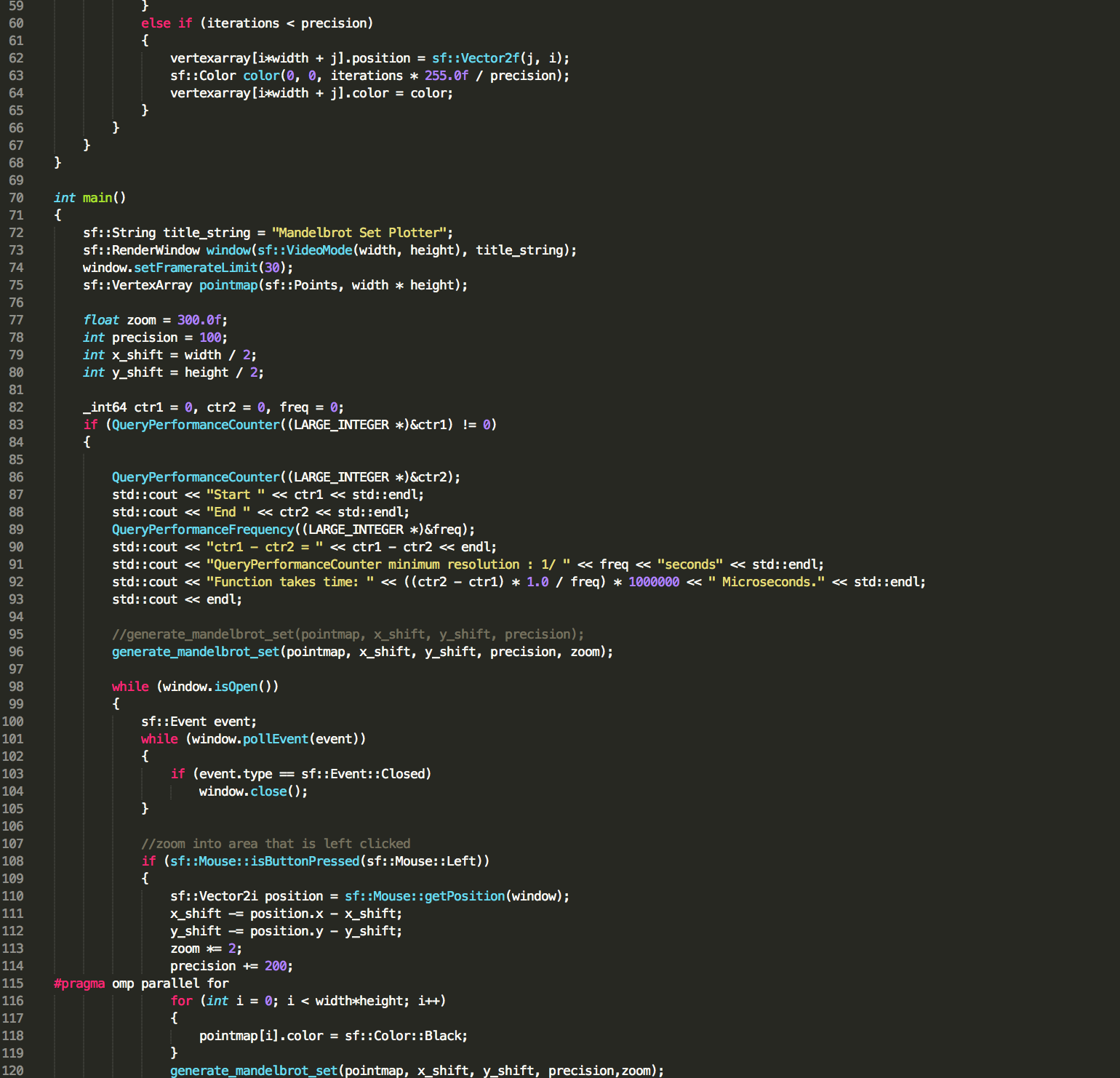
plot(Px, Py, color)

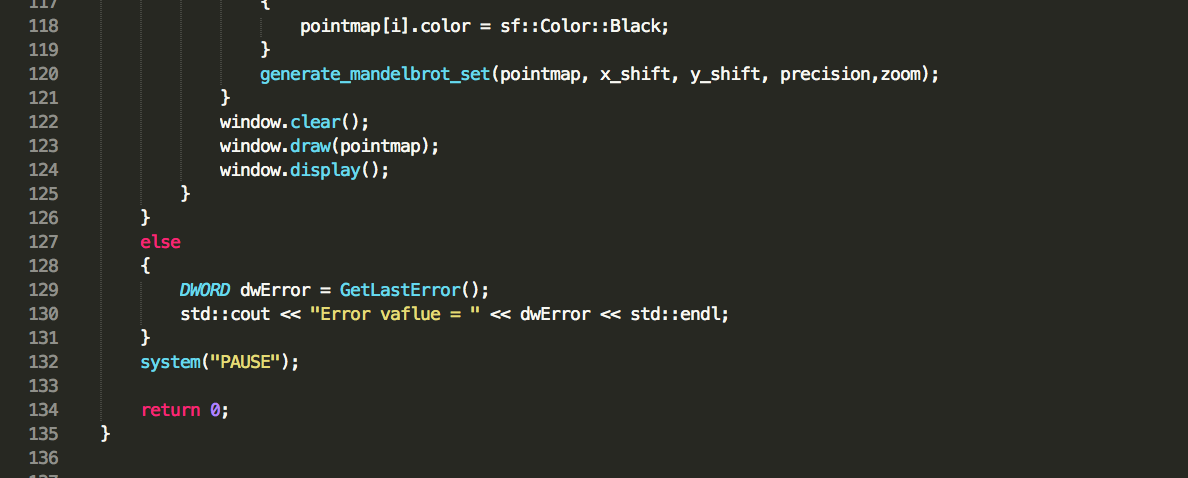
}

Above image is the pseudocode for the Mandelbrot set. Following the Mandelbrot pseudocode, we were to provide either C++ or C code and produce an image and count the timing using query performance using visual studio.

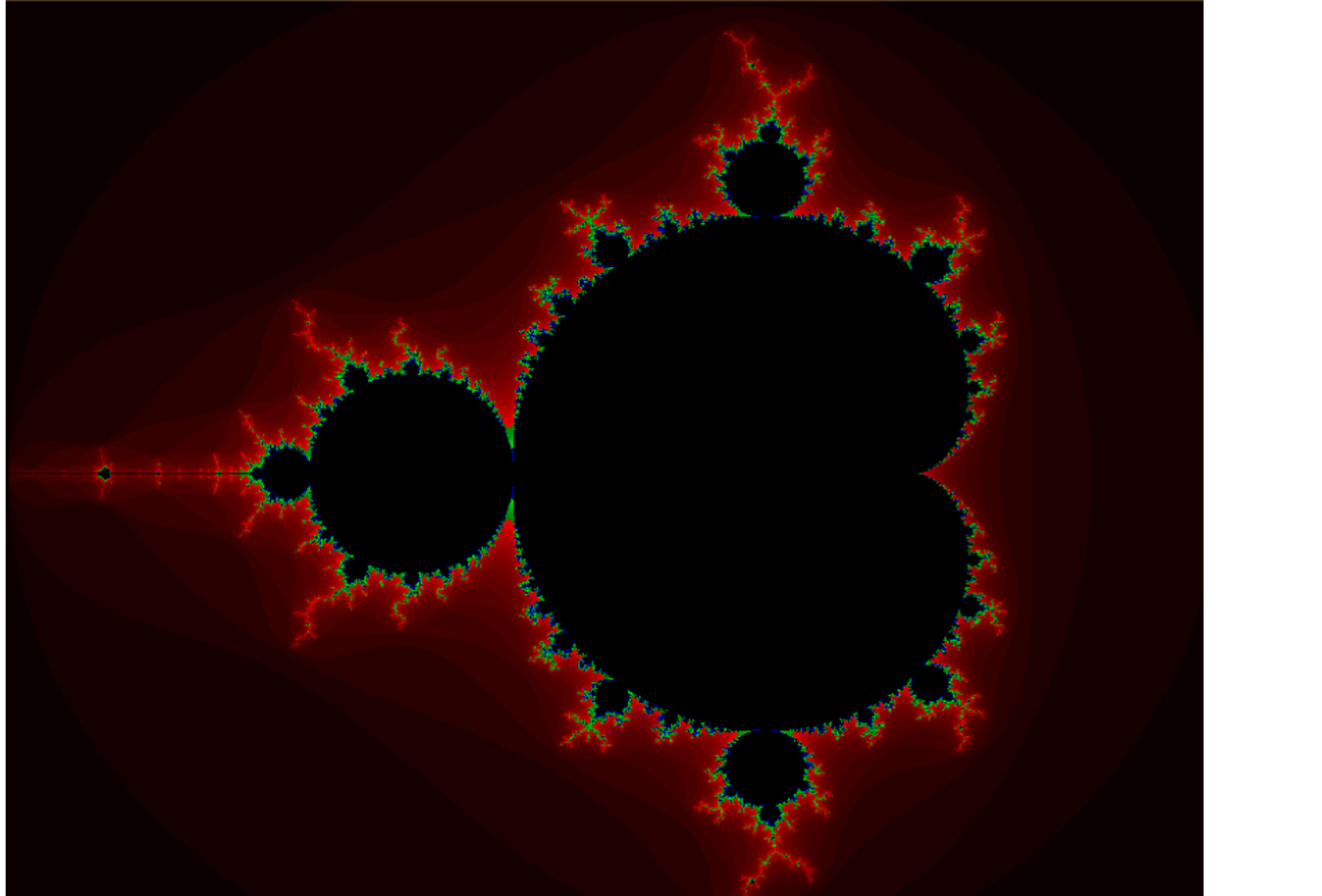
**Mandelbrot Code in C++:**



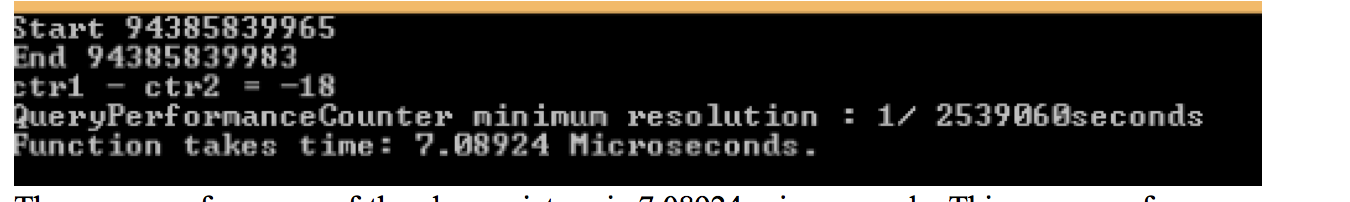




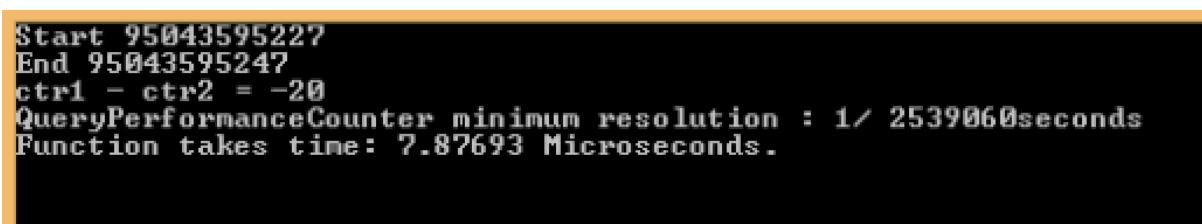
Above 3 pictures are C++ code for Mandelbrot. I have used visual studio to create an image and write the code for Mandelbrot set. To produce a Mandelbrot image I used SFML library in the visual studio. Then we were to measure the Query performance of the Mandelbrot set. Below are pictures of Mandelbrot set image and its query performances;



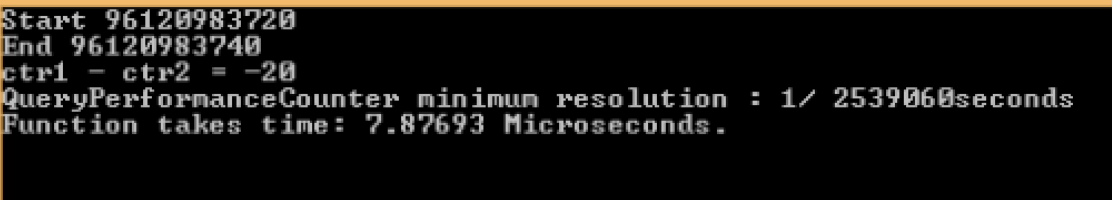
**Query Performances:**



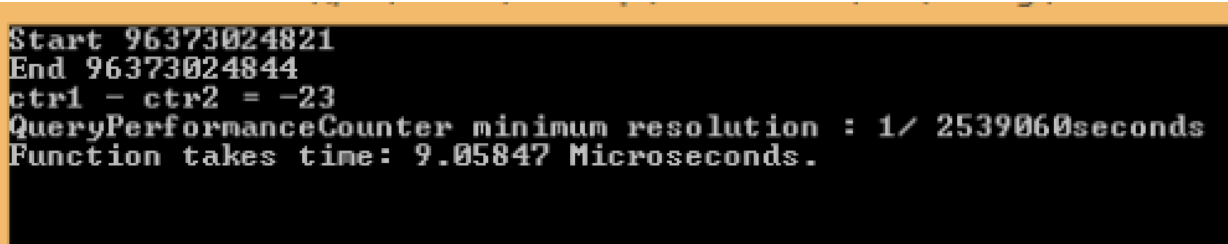
The query performance of the above picture is 7.08924 microseconds. This query performance was done when Qpar and vectorization were disabled.



The query performance of the above picture is 7.87693 microseconds. This query performance was done when Qpar was No and for vectorization I used SSE2. This performance took a little more time since the Mandelbrot set needed to be vectorized.



The query performance of the above picture is 7.87693 microseconds. This query performance was done when Qpar was enable and for vectorization I used AVX2. It has same query performance as the above picture.



The query performance of the above picture is 9.05847 microseconds. This query performance was done when Qpar was enable and for vectorization I used AVX. This performance took the most time for the Mandelbrot set to be vectorized.

**Mandelbrot AVX Intrinsic:**

To make the speed faster of the Mandelbrot set, we can use AVX intrinsic instructions. Below is the function for the AVX intrinsic instruction for Mandelbrot set.

void MandelbrotAVX(float x1, float y1, float x2, float y2, int width, int height, int maxIters, unsigned short \* image)

{

float dx = (x2-x1)/width;

float dy = (y2-y1)/height;

// round up width to next multiple of 8

int roundedWidth = (width+7) & ~7UL;

float constants[] = {dx, dy, x1, y1, 1.0f, 4.0f};

\_\_m256 ymm0 = \_mm256\_broadcast\_ss(constants); // all dx

\_\_m256 ymm1 = \_mm256\_broadcast\_ss(constants+1); // all dy

\_\_m256 ymm2 = \_mm256\_broadcast\_ss(constants+2); // all x1

\_\_m256 ymm3 = \_mm256\_broadcast\_ss(constants+3); // all y1

\_\_m256 ymm4 = \_mm256\_broadcast\_ss(constants+4); // all 1's (iter increments)

\_\_m256 ymm5 = \_mm256\_broadcast\_ss(constants+5); // all 4's (comparisons)

float incr[8]={0.0f,1.0f,2.0f,3.0f,4.0f,5.0f,6.0f,7.0f}; // used to reset the i position when j increases

\_\_m256 ymm6 = \_mm256\_xor\_ps(ymm0,ymm0); // zero out j counter (ymm0 is just a dummy)

for (int j = 0; j < height; j+=1)

{

\_\_m256 ymm7 = \_mm256\_load\_ps(incr); // i counter set to 0,1,2,..,7

for (int i = 0; i < roundedWidth; i+=8)

{

\_\_m256 ymm8 = \_mm256\_mul\_ps(ymm7, ymm0); // x0 = (i+k)\*dx

ymm8 = \_mm256\_add\_ps(ymm8, ymm2); // x0 = x1+(i+k)\*dx

\_\_m256 ymm9 = \_mm256\_mul\_ps(ymm6, ymm1); // y0 = j\*dy

ymm9 = \_mm256\_add\_ps(ymm9, ymm3); // y0 = y1+j\*dy

\_\_m256 ymm10 = \_mm256\_xor\_ps(ymm0,ymm0); // zero out iteration counter (ymm0 is just a dummy)

\_\_m256 ymm11 = ymm10, ymm12 = ymm10; // set initial xi=0, yi=0

unsigned int test = 0;

int iter = 0;

do

{

\_\_m256 ymm13 = \_mm256\_mul\_ps(ymm11,ymm11); // xi\*xi

\_\_m256 ymm14 = \_mm256\_mul\_ps(ymm12,ymm12); // yi\*yi

\_\_m256 ymm15 = \_mm256\_add\_ps(ymm13,ymm14); // xi\*xi+yi\*yi

ymm15 = \_mm256\_cmp\_ps(ymm15,ymm5, \_CMP\_LT\_OQ); // xi\*xi+yi\*yi < 4 in each slot

// now ymm15 has all 1s in the non overflowed locations

test = \_mm256\_movemask\_ps(ymm15)&255; // lower 8 bits are comparisons

ymm15 = \_mm256\_and\_ps(ymm15,ymm4); // get 1.0f or 0.0f in each field as counters

ymm10 = \_mm256\_add\_ps(ymm10,ymm15); // counters for each pixel iteration

ymm15 = \_mm256\_mul\_ps(ymm11,ymm12); // xi\*yi

ymm11 = \_mm256\_sub\_ps(ymm13,ymm14); // xi\*xi-yi\*yi

ymm11 = \_mm256\_add\_ps(ymm11,ymm8); // xi <- xi\*xi-yi\*yi+x0 done!

ymm12 = \_mm256\_add\_ps(ymm15,ymm15); // 2\*xi\*yi

ymm12 = \_mm256\_add\_ps(ymm12,ymm9); // yi <- 2\*xi\*yi+y0

++iter;

} while ((test != 0) && (iter < maxIters));

// convert iterations to output values

\_\_m256i ymm10i = \_mm256\_cvtps\_epi32(ymm10);

// write only where needed

int top = (i+7) < width? 8: width&7;

for (int k = 0; k < top; ++k)

image[i+k+j\*width] = ymm10i.m256i\_i16[2\*k];

// next i position - increment each slot by 8

ymm7 = \_mm256\_add\_ps(ymm7, ymm5);

ymm7 = \_mm256\_add\_ps(ymm7, ymm5);

}

ymm6 = \_mm256\_add\_ps(ymm6,ymm4); // increment j counter

}

}

After running the AVX intrinsic code, I could observe the better running performance for Mandelbrot set. It was faster than the regular C or C++ code.

**Mandelbrot Set with openMP and MPI:**

The purpose of this code was to make the performance more efficient and reduce the number of cycles of Mandelbrot set. Since we build a raspberry pi cluster, which has 8 nodes and each node has 4 processes. Which means our code should be 32 times faster using mpi and openMP. Using MPI we divide the code into 8 pieces. Where each node will be assigned a piece of code to be executed. Since each node has 4 cores. We can write openMP code to divide each node’s code into 4 cores. Which means the code that each node has will be divided into 4 parts and assign each part to each core using openMP.

**How to Run the Code on the Raspberry Pi Cluster:**

1. First We make a directory of our name in the master node.

mkdir ~/mpich/{YOUR\_NAME}

1. Then we send the command to the rest of the nodes

parallel-ssh -i -h ~/pssh\_hosts mkdir ~/mpich/{YOUR\_NAME}

1. Then we copy the code from the flash drive into the master code

cp /media/pi/{YOUR\_FLASH\_DRIVE}/{YOUR\_CODE.CPP} ~/mpich/{YOUR\_NAME}/

1. Then compile the code with mpicc

mpicc -o {YOUR\_EXEC} {SOURCE.CPP}

1. Then copy the executable file to all other nodes

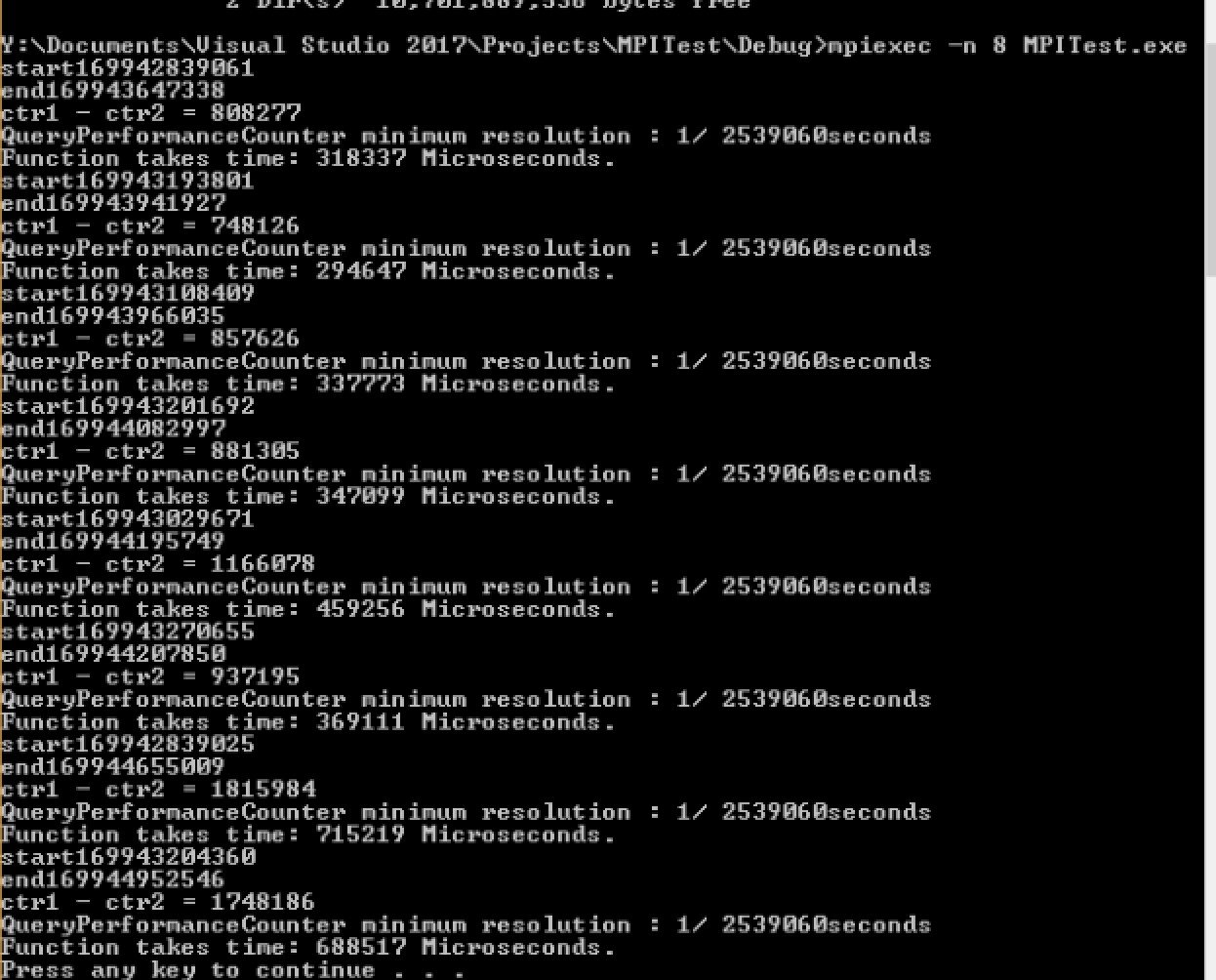
parallel-scp -v -h ~/pssh\_hosts ~/mpich/{YOUR\_NAME}/{YOUR\_EXEC} ~/mpich/{YOUR\_NAME}/

1. Run the executable file on the master node using mpirun or mpiexec

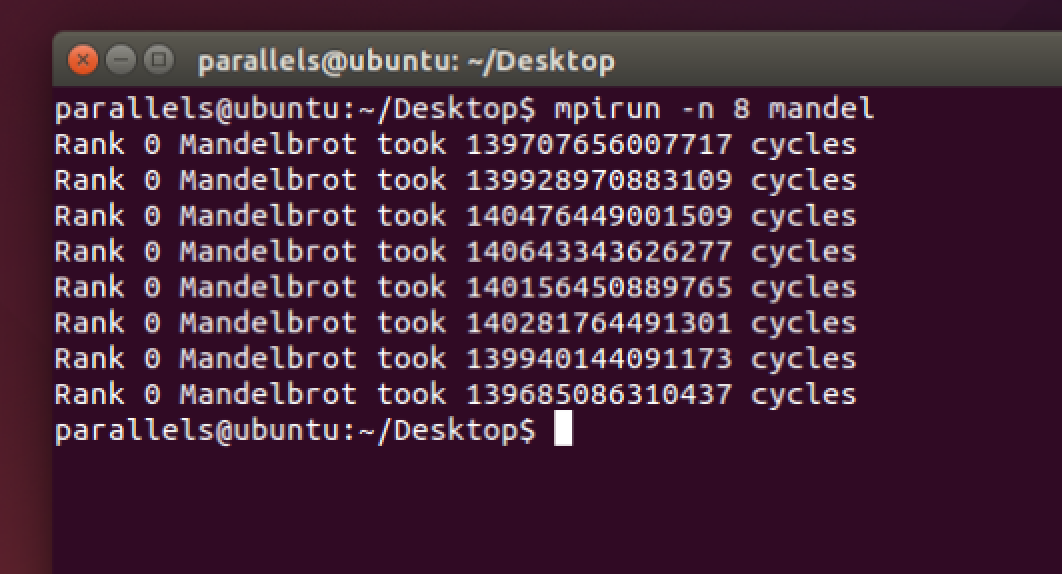
mpirun -n 8 -hostfile ~/host\_file ~/mpich/{YOUR\_NAME}/{YOUR\_EXEC}

**Mandelbrot in MPI:**

I have 2 ways to run the Mandelbrot code with MPI. First I ran the code using Visual studio compiler and the other one is perf event open in Linux.



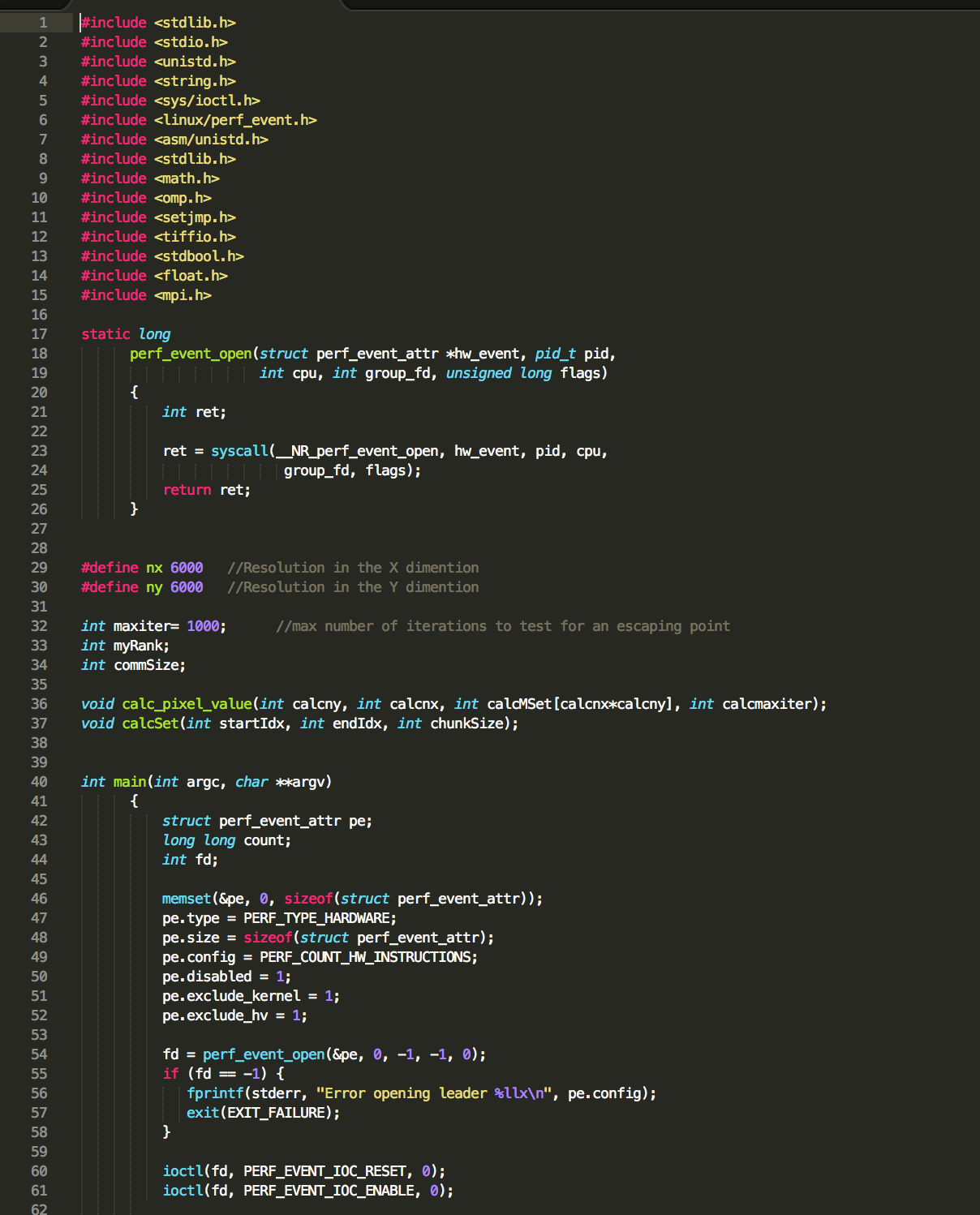
As you can see in the picture that every node has different timing. Because it does not have load balance. And I did not assign the specific code to any of the specified node. Since the Mandelbrot is a symmetric image so it takes more time to compute the code from the center of the image. The execution of the top of the image takes less timing than the middle part of the image.



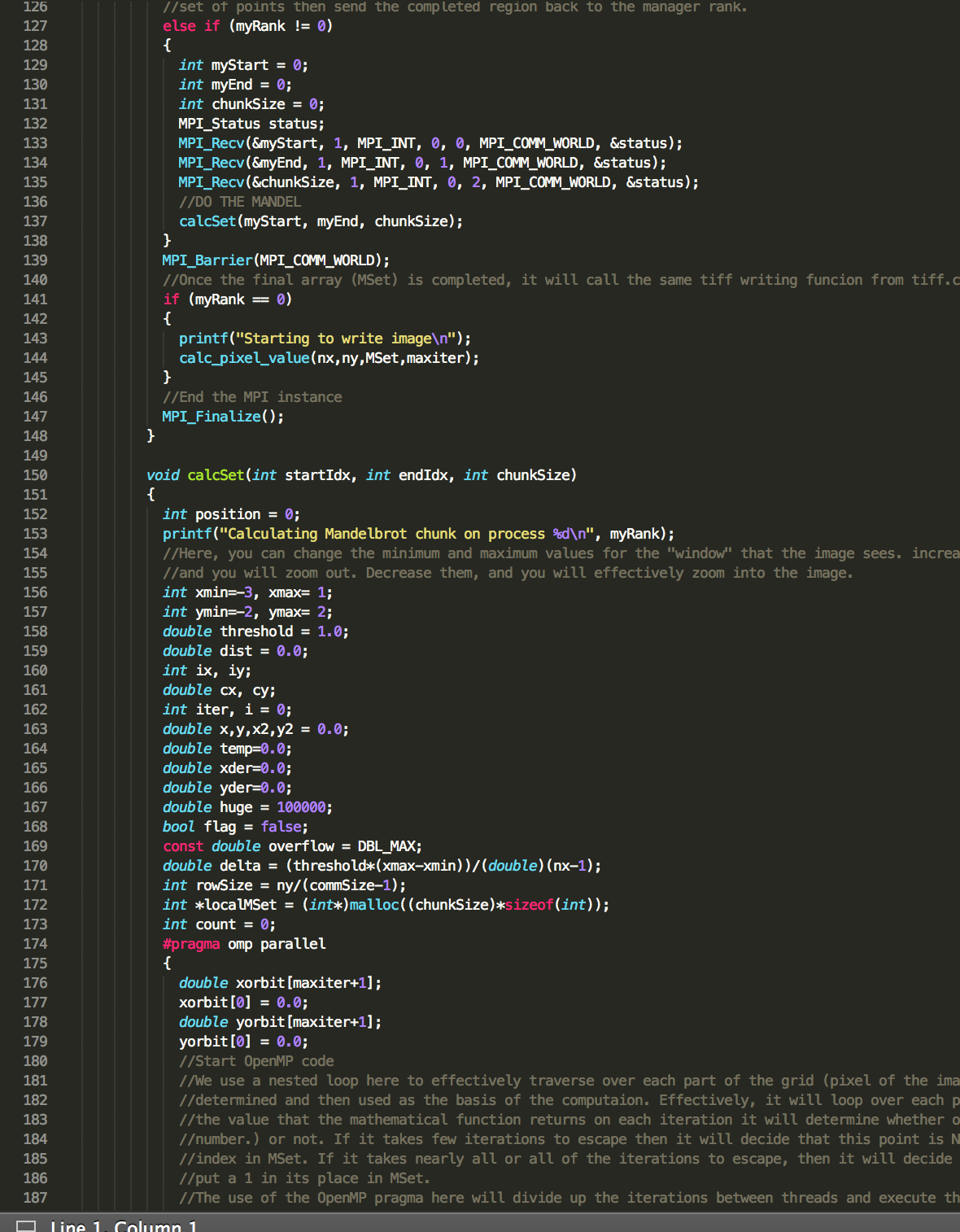
This image is from Linux terminal. Where we used perf event open command to perform the timing. Since this Mandelbrot code has load balancing, where each processes is assigned different parts of the code. What load balance does is it divides the code into 16 pieces, then assign the middle and upper part of the code to the process; Since upper part takes less than middle part of the image in the Mandelbrot set.

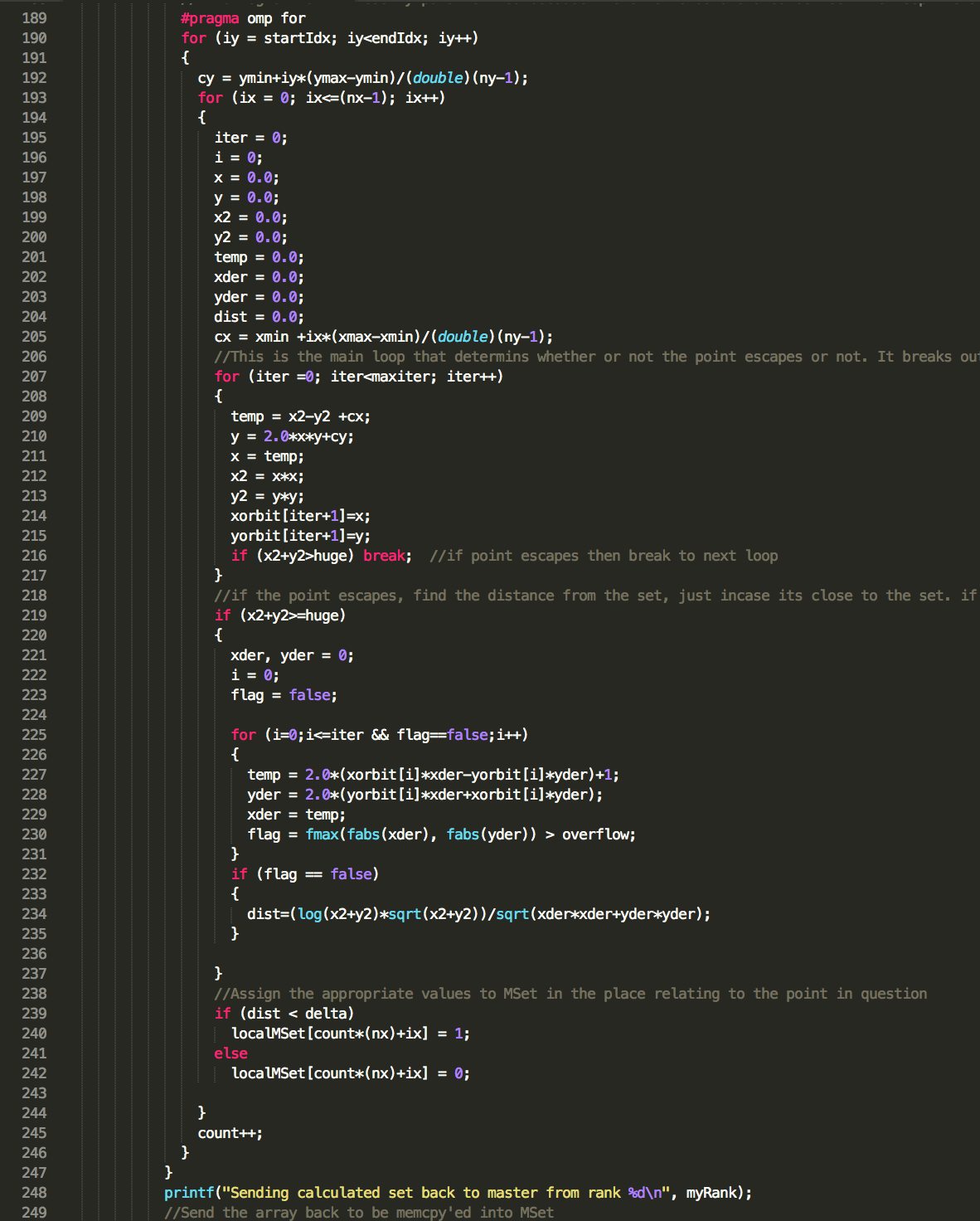
**Mandelbrot in MPI and openMP:**

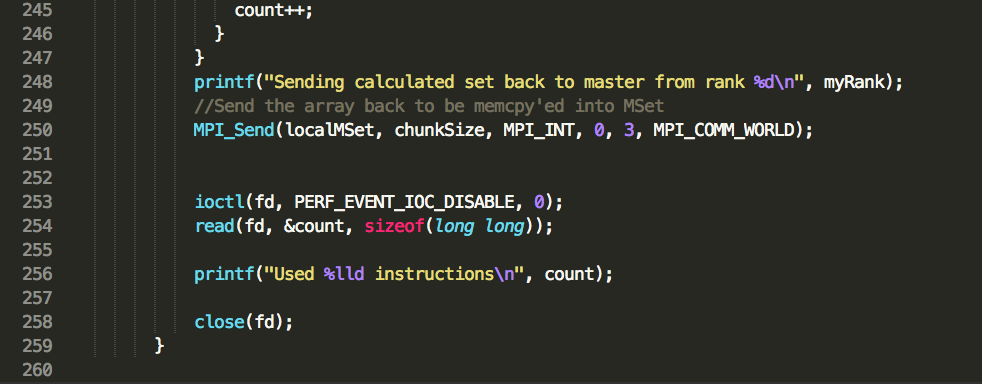
This Mandelbrot set code is written using MPI and openMP. The purpose was to reduce the numbers of cycles for the Mandelbrot set and improve the speed and efficiency. To record the timing and number of cycles, I used perf event open in Linux. Which keeps track of the timing of the execution of the code.











This is the corresponding for the Mandelbrot code using MPI and openMP. As you can see I have divided the code into the specified node and used openMP to assign the task to all cores in the nodes. This code produces much better results in terms of timing and performance.

**Conclusion:**

The purpose of the project was write Mandelbrot set code using C++ or C and check the query performance. Then we had to use MPI and openMP to reduce the number of cycles and improve the timing and efficiency of the code. As a result the Mandelbrot code with MPI and openMPI was faster than the rest of the other code.