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 $z_{n+1} = z_n^2 + C$. with $z_0 = C$, where points C in the complex plane for which the orbit of z_n 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.

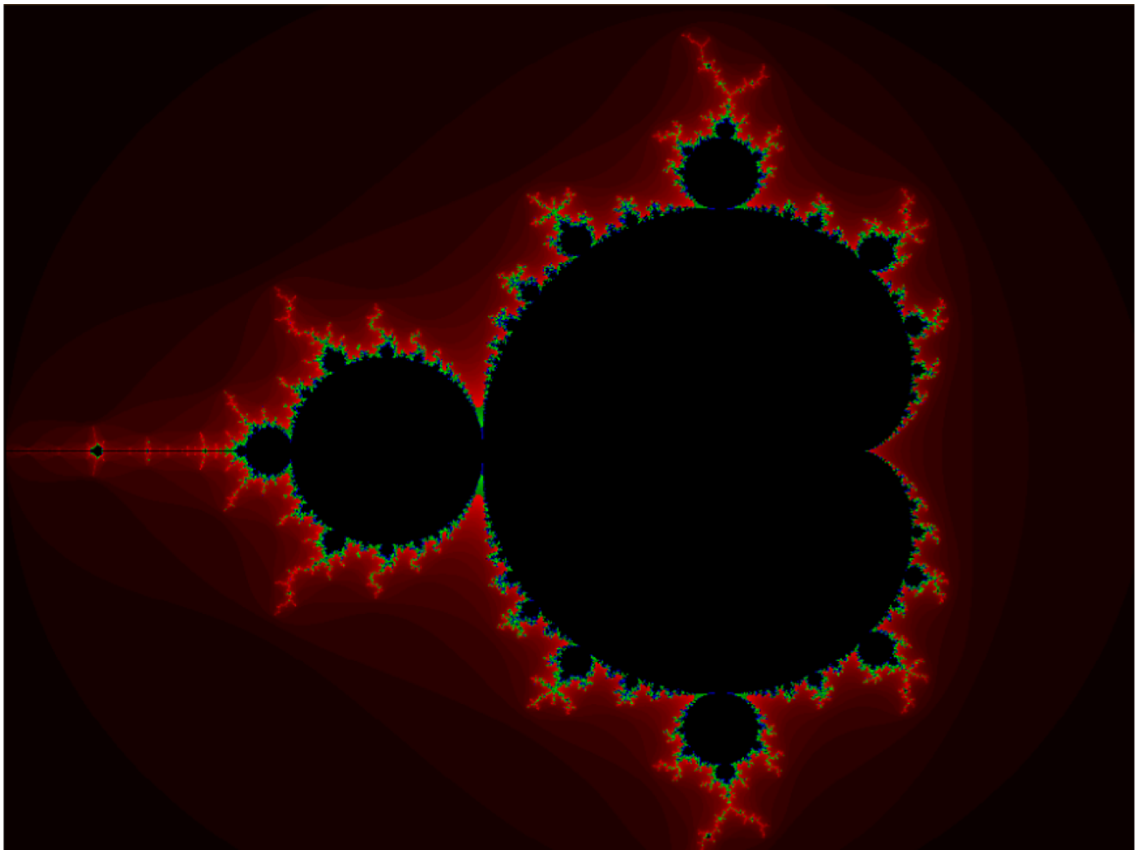
Mandelbrot Code in C++:

```cpp
#include "SFML/Graphics.hpp"
#include<iostream>
#include<windows.h>
#include <tchar.h>
using namespace std;
//resolution of the window
const int width = 1280;
const int height = 720;

//used for complex numbers
struct complex_number
{
    long double real;
    long double imaginary;
};

void generate_mandelbrot_set(sf::VertexArray& vertexarray, int pixel_shift_x, int pixel_shift_y, int precision, float zoom)
//void generate_mandelbrot_set(sf::VertexArray& vertexarray, int pixel_shift_x, int pixel_shift_y, int precision)
{
#pragma omp parallel for
    for (int i = 0; i < height; i++)
    {
        for (int j = 0; j < width; j++)
        {
            //scale the pixel location to the complex plane for calculations
            //long double x = ((long double)j - pixel_shift_x);
            //long double y = ((long double)i - pixel_shift_y);
            long double x = ((long double)j - pixel_shift_x) / zoom;
            long double y = ((long double)i - pixel_shift_y) / zoom;
            complex_number c;
            c.real = x;
            c.imaginary = y;
            complex_number z = c;
            int iterations = 0; //keep track of the number of iterations
            for (int k = 0; k < precision; k++)
            {
                complex_number z2;
                z2.real = z.real * z.real - z.imaginary * z.imaginary;
                z2.imaginary = 2 * z.real * z.imaginary;
                z2.real += c.real;
                z2.imaginary += c.imaginary;
                z = z2;
                iterations++;
                if (z.real * z.real + z.imaginary * z.imaginary > 4)
                    break;
            }
            //color pixel based on the number of iterations
            if (iterations < precision / 4.0f)
            {
                vertexarray[i*width + j].position = sf::Vector2f(j, i);
                sf::Color color(iterations * 255.0f / (precision / 4.0f), 0, 0);
                vertexarray[i*width + j].color = color;
            }
            else if (iterations < precision / 2.0f)
            {
                vertexarray[i*width + j].position = sf::Vector2f(j, i);
                sf::Color color(0, iterations * 255.0f / (precision / 2.0f), 0);
                vertexarray[i*width + j].color = color;
            }
            else if (iterations < precision)
            {
                vertexarray[i*width + j].position = sf::Vector2f(j, i);
```

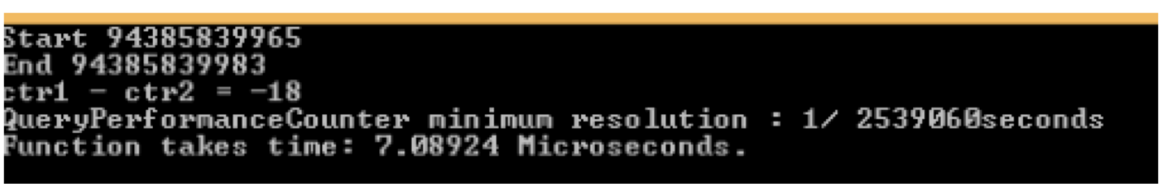
```
            }
            else if (iterations < precision)
            {
                vertexarray[i*width + j].position = sf::Vector2f(j, i);
                sf::Color color(0, 0, iterations * 255.0f / precision);
                vertexarray[i*width + j].color = color;
            }
        }
    }
}

int main()
{
    sf::String title_string = "Mandelbrot Set Plotter";
    sf::RenderWindow window(sf::VideoMode(width, height), title_string);
    window.setFramerateLimit(30);
    sf::VertexArray pointmap(sf::Points, width * height);

    float zoom = 300.0f;
    int precision = 100;
    int x_shift = width / 2;
    int y_shift = height / 2;

    _int64 ctr1 = 0, ctr2 = 0, freq = 0;
    if (QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) != 0)
    {

        QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);
        std::cout << "Start " << ctr1 << std::endl;
        std::cout << "End " << ctr2 << std::endl;
        QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
        std::cout << "ctr1 - ctr2 = " << ctr1 - ctr2 << endl;
        std::cout << "QueryPerformanceCounter minimum resolution : 1/ " << freq << "seconds" << std::endl;
        std::cout << "Function takes time: " << ((ctr2 - ctr1) * 1.0 / freq) * 1000000 << " Microseconds." << std::endl;
        std::cout << endl;

        //generate_mandelbrot_set(pointmap, x_shift, y_shift, precision);
        generate_mandelbrot_set(pointmap, x_shift, y_shift, precision, zoom);

        while (window.isOpen())
        {
            sf::Event event;
            while (window.pollEvent(event))
            {
                if (event.type == sf::Event::Closed)
                    window.close();
            }

            //zoom into area that is left clicked
            if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
            {
                sf::Vector2i position = sf::Mouse::getPosition(window);
                x_shift -= position.x - x_shift;
                y_shift -= position.y - y_shift;
                zoom *= 2;
                precision += 200;
#pragma omp parallel for
                for (int i = 0; i < width*height; i++)
                {
                    pointmap[i].color = sf::Color::Black;
                }
                generate_mandelbrot_set(pointmap, x_shift, y_shift, precision,zoom);
```

```
                {
                    pointmap[i].color = sf::Color::Black;
                }
                generate_mandelbrot_set(pointmap, x_shift, y_shift, precision,zoom);
            }
            window.clear();
            window.draw(pointmap);
            window.display();
        }
    }
    else
    {
        DWORD dwError = GetLastError();
        std::cout << "Error vaflue = " << dwError << std::endl;
    }
    system("PAUSE");

    return 0;
}
```

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;



<u>Query Performances:</u>



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

```
Start 95043595227
End 95043595247
ctr1 - ctr2 = -20
QueryPerformanceCounter minimum resolution : 1/ 2539060seconds
Function takes time: 7.87693 Microseconds.
```

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.

```
Start 96120983720
End 96120983740
ctr1 - ctr2 = -20
QueryPerformanceCounter minimum resolution : 1/ 2539060seconds
Function takes time: 7.87693 Microseconds.
```

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.

```
Start 96373024821
End 96373024844
ctr1 - ctr2 = -23
QueryPerformanceCounter minimum resolution : 1/ 2539060seconds
Function takes time: 9.05847 Microseconds.
```

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}

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

   parallel-ssh -i -h ~/pssh_hosts mkdir ~/mpich/{YOUR_NAME}

3. 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}/

4. Then compile the code with mpicc

   mpicc -o {YOUR_EXEC} {SOURCE.CPP}

5. Then copy the executable file to all other nodes

   parallel-scp -v -h ~/pssh_hosts ~/mpich/{YOUR_NAME}/{YOUR_EXEC}
   ~/mpich/{YOUR_NAME}/

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

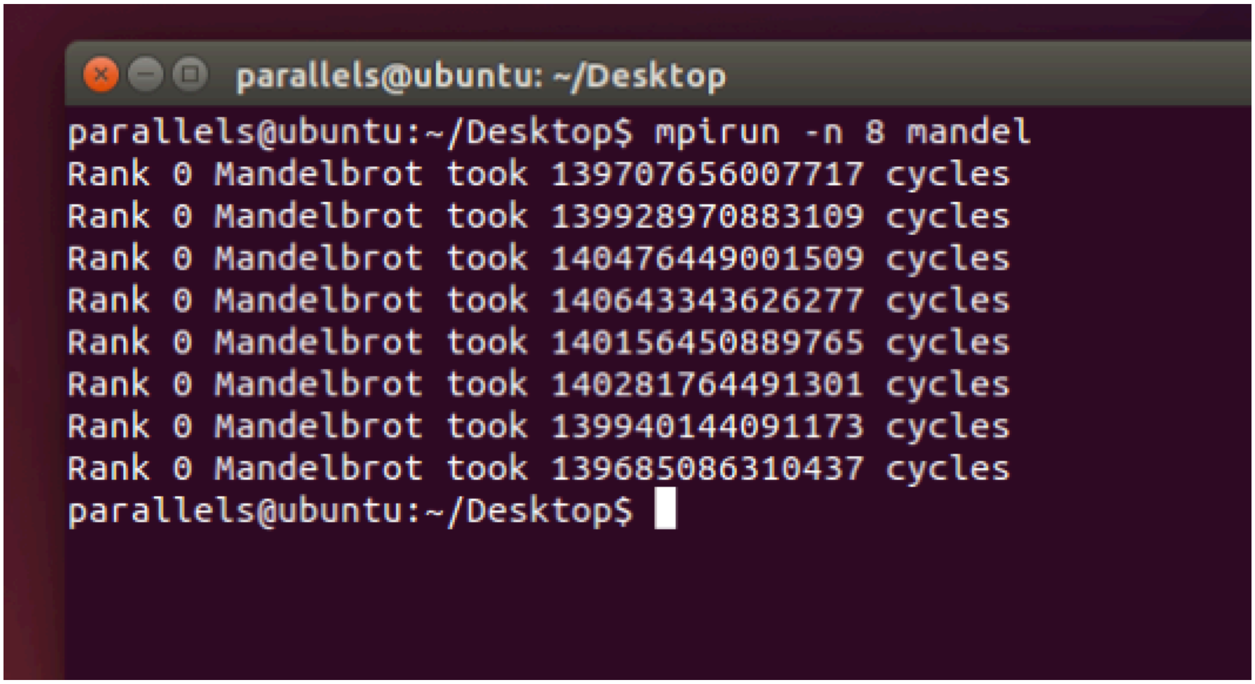
   mpirun -n 8 -hostfile ~/host_file ~/mpich/{YOUR_NAME}/{YOUR_EXEC}

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.

```
Y:\Documents\Visual Studio 2017\Projects\MPITest\Debug>mpiexec -n 8 MPITest.exe
start169942839061
end169943647338
ctr1 - ctr2 = 808277
QueryPerformanceCounter minimum resolution : 1/ 2539060seconds
Function takes time: 318337 Microseconds.
start169943193801
end169943941927
ctr1 - ctr2 = 748126
QueryPerformanceCounter minimum resolution : 1/ 2539060seconds
Function takes time: 294647 Microseconds.
start169943108409
end169943966035
ctr1 - ctr2 = 857626
QueryPerformanceCounter minimum resolution : 1/ 2539060seconds
Function takes time: 337773 Microseconds.
start169943201692
end169944082997
ctr1 - ctr2 = 881305
QueryPerformanceCounter minimum resolution : 1/ 2539060seconds
Function takes time: 347099 Microseconds.
start169943029671
end169944195749
ctr1 - ctr2 = 1166078
QueryPerformanceCounter minimum resolution : 1/ 2539060seconds
Function takes time: 459256 Microseconds.
start169943270655
end169944207850
ctr1 - ctr2 = 937195
QueryPerformanceCounter minimum resolution : 1/ 2539060seconds
Function takes time: 369111 Microseconds.
start169942839025
end169944655009
ctr1 - ctr2 = 1815984
QueryPerformanceCounter minimum resolution : 1/ 2539060seconds
Function takes time: 715219 Microseconds.
start169943204360
end169944952546
ctr1 - ctr2 = 1748186
QueryPerformanceCounter minimum resolution : 1/ 2539060seconds
Function takes time: 688517 Microseconds.
Press any key to continue . . .
```

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.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/ioctl.h>
#include <linux/perf_event.h>
#include <asm/unistd.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include <setjmp.h>
#include <tiffio.h>
#include <stdbool.h>
#include <float.h>
#include <mpi.h>

static long
        perf_event_open(struct perf_event_attr *hw_event, pid_t pid,
                        int cpu, int group_fd, unsigned long flags)
        {
            int ret;

            ret = syscall(__NR_perf_event_open, hw_event, pid, cpu,
                          group_fd, flags);
            return ret;
        }


#define nx 6000   //Resolution in the X dimention
#define ny 6000   //Resolution in the Y dimention

int maxiter= 1000;      //max number of iterations to test for an escaping point
int myRank;
int commSize;

void calc_pixel_value(int calcny, int calcnx, int calcMSet[calcnx*calcny], int calcmaxiter);
void calcSet(int startIdx, int endIdx, int chunkSize);


int main(int argc, char **argv)
        {
            struct perf_event_attr pe;
            long long count;
            int fd;

            memset(&pe, 0, sizeof(struct perf_event_attr));
            pe.type = PERF_TYPE_HARDWARE;
            pe.size = sizeof(struct perf_event_attr);
            pe.config = PERF_COUNT_HW_INSTRUCTIONS;
            pe.disabled = 1;
            pe.exclude_kernel = 1;
            pe.exclude_hv = 1;

            fd = perf_event_open(&pe, 0, -1, -1, 0);
            if (fd == -1) {
                fprintf(stderr, "Error opening leader %llx\n", pe.config);
                exit(EXIT_FAILURE);
            }

            ioctl(fd, PERF_EVENT_IOC_RESET, 0);
            ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);
```

```c
MPI_Init(&argc, &argv);
	//Here we create a 1 dimensional array that will be the total size of the image (in pixels). Because
	//this array can get so big, we need to declare it on the heap and NOT the stack (i.e. how one would normally
	//declare an array. So we use malloc();
	int *MSet = (int*)malloc(nx*ny*sizeof(int));
	memset(MSet, 0, nx*ny*sizeof(int));
	//Get rank number and total number of processes that were launched
	MPI_Comm_size(MPI_COMM_WORLD, &commSize);
	MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
	//Check to see if the resolution and number of processes are compatable.
	if (myRank == 0)
	{
		if ((nx*ny)%(commSize-1) != 0)
		{
			printf("Incompatable number of processes requested\nExiting...\n");
			exit(EXIT_FAILURE);
		}
	}
	int totalRes = (nx*ny);
	int chunkSize = totalRes/(commSize-1);

	if (myRank == 0)
		printf("Starting Calculation of Mandelbrot set...\n");

	int i;
	MPI_Barrier(MPI_COMM_WORLD);
	//This only happens on the manager rank (rank 0). Basically it will compute the boundaries of the section of MS
	//compute nodes will process and the size of the region and send them to the appropriate process. Then it will
	//the completed section of the array it was assigned. When its recieved it will copy it into the right place in
	if (myRank == 0)
	{
		int myStart = 0, myEnd = 0;
		int* tempMSet = (int*)malloc((chunkSize)*sizeof(int));
		for (i = 1; i<commSize; i++)
		{
			myStart = (chunkSize/ny)*(i-1);
			myEnd = (myStart + (chunkSize/ny));

			//send to each node
			//tag 0 = startIdx
			//tag 1 = endIdx
			//tag 2 = chunkSize

			MPI_Send(&myStart, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
			MPI_Send(&myEnd, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
			MPI_Send(&chunkSize, 1, MPI_INT, i, 2, MPI_COMM_WORLD);
			printf("Sent chunk data to node %d\n", i);
		}
		int sender = -1;
		MPI_Status status;
		//Recieve the completed chunks and memcpy it onto MSet
		for (i = 1; i<commSize; i++)
		{
			MPI_Recv(tempMSet, chunkSize, MPI_INT, MPI_ANY_SOURCE, 3, MPI_COMM_WORLD, &status);
			sender = status.MPI_SOURCE;
			printf("Recieved chunk from %d\n",sender);
			memcpy(MSet+(chunkSize*(sender-1)), tempMSet, chunkSize*sizeof(int));
			printf("Memcpy'd data from rank %d into MSet\n", sender);
			sender = -1;
		}
```

```
    //set of points then send the completed region back to the manager rank.
    else if (myRank != 0)
    {
      int myStart = 0;
      int myEnd = 0;
      int chunkSize = 0;
      MPI_Status status;
      MPI_Recv(&myStart, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
      MPI_Recv(&myEnd, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
      MPI_Recv(&chunkSize, 1, MPI_INT, 0, 2, MPI_COMM_WORLD, &status);
      //DO THE MANDEL
      calcSet(myStart, myEnd, chunkSize);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    //Once the final array (MSet) is completed, it will call the same tiff writing funcion from tiff.c
    if (myRank == 0)
    {
      printf("Starting to write image\n");
      calc_pixel_value(nx,ny,MSet,maxiter);
    }
    //End the MPI instance
    MPI_Finalize();
  }

  void calcSet(int startIdx, int endIdx, int chunkSize)
  {
    int position = 0;
    printf("Calculating Mandelbrot chunk on process %d\n", myRank);
    //Here, you can change the minimum and maximum values for the "window" that the image sees. increa
    //and you will zoom out. Decrease them, and you will effectively zoom into the image.
    int xmin=-3, xmax= 1;
    int ymin=-2, ymax= 2;
    double threshold = 1.0;
    double dist = 0.0;
    int ix, iy;
    double cx, cy;
    int iter, i = 0;
    double x,y,x2,y2 = 0.0;
    double temp=0.0;
    double xder=0.0;
    double yder=0.0;
    double huge = 100000;
    bool flag = false;
    const double overflow = DBL_MAX;
    double delta = (threshold*(xmax-xmin))/(double)(nx-1);
    int rowSize = ny/(commSize-1);
    int *localMSet = (int*)malloc((chunkSize)*sizeof(int));
    int count = 0;
    #pragma omp parallel
    {
      double xorbit[maxiter+1];
      xorbit[0] = 0.0;
      double yorbit[maxiter+1];
      yorbit[0] = 0.0;
      //Start OpenMP code
      //We use a nested loop here to effectively traverse over each part of the grid (pixel of the ima
      //determined and then used as the basis of the computaion. Effectively, it will loop over each p
      //the value that the mathematical function returns on each iteration it will determine whether o
      //number.) or not. If it takes few iterations to escape then it will decide that this point is N
      //index in MSet. If it takes nearly all or all of the iterations to escape, then it will decide
      //put a 1 in its place in MSet.
      //The use of the OpenMP pragma here will divide up the iterations between threads and execute th
```

Line 1, Column 1

```
        #pragma omp for
        for (iy = startIdx; iy<endIdx; iy++)
        {
          cy = ymin+iy*(ymax-ymin)/(double)(ny-1);
          for (ix = 0; ix<=(nx-1); ix++)
          {
            iter = 0;
            i = 0;
            x = 0.0;
            y = 0.0;
            x2 = 0.0;
            y2 = 0.0;
            temp = 0.0;
            xder = 0.0;
            yder = 0.0;
            dist = 0.0;
            cx = xmin +ix*(xmax-xmin)/(double)(ny-1);
            //This is the main loop that determins whether or not the point escapes or not. It breaks ou
            for (iter =0; iter<maxiter; iter++)
            {
              temp = x2-y2 +cx;
              y = 2.0*x*y+cy;
              x = temp;
              x2 = x*x;
              y2 = y*y;
              xorbit[iter+1]=x;
              yorbit[iter+1]=y;
              if (x2+y2>huge) break;  //if point escapes then break to next loop
            }
            //if the point escapes, find the distance from the set, just incase its close to the set. if
            if (x2+y2>=huge)
            {
              xder, yder = 0;
              i = 0;
              flag = false;

              for (i=0;i<=iter && flag==false;i++)
              {
                temp = 2.0*(xorbit[i]*xder-yorbit[i]*yder)+1;
                yder = 2.0*(yorbit[i]*xder+xorbit[i]*yder);
                xder = temp;
                flag = fmax(fabs(xder), fabs(yder)) > overflow;
              }
              if (flag == false)
              {
                dist=(log(x2+y2)*sqrt(x2+y2))/sqrt(xder*xder+yder*yder);
              }

            }
            //Assign the appropriate values to MSet in the place relating to the point in question
            if (dist < delta)
              localMSet[count*(nx)+ix] = 1;
            else
              localMSet[count*(nx)+ix] = 0;

          }
          count++;
        }
      }
      printf("Sending calculated set back to master from rank %d\n", myRank);
      //Send the array back to be memcpy'ed into MSet
```

```
                    count++;
                }
            }
            printf("Sending calculated set back to master from rank %d\n", myRank);
            //Send the array back to be memcpy'ed into MSet
            MPI_Send(localMSet, chunkSize, MPI_INT, 0, 3, MPI_COMM_WORLD);


            ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
            read(fd, &count, sizeof(long long));

            printf("Used %lld instructions\n", count);

            close(fd);
        }

```

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.