Assignment 2 Mandelbrot Set Problem
CSC4005 Distributed and Parallel Computing

Cui Yuncong
School of Data Science
The Chinese University of Hong Kong, Shenzhen
118010045@link.cuhk.edu.cn

1. Introduction

Mandelbrot set is a set of points in a complex number giving the position of the point in the complex plane. All the points are quasistable when computed by iterating the function:

$$z_{k+1} = z_k^2 + c \tag{1}$$

Where z_{k+1} is the $(k+1)$th iteration of the complex number $z = a + b_i$, z_k is the kth iteration of z, and c is a complex number giving the position of the point in the complex plane. The initial value for z is zero and the iteration are continues until the magnitude of z is greater than 2 or the number of iteration reaches some arbitrary limitation. Displaying the Mandelbrot set is an example of processing a bit-mapped image. For message-passing system, the Mandelbrot set is particularly convenient to parallelize since each pixel can be computed without any information about the surrounding pixels. In this assignment, the implemented algorithms are MPI and Pthread respectively.

2. Method

The below figures illustrate the output as well as the execution step of the design of the MPI method and the Pthread method.

2.1. MPI

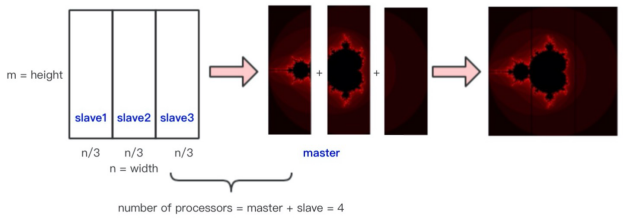

Figure 1: Program Design of Mandelbrot Set Computation using MPI method(eg. Number of Processes = 4)

In the MPI method, each process are assigned with a fixed area of the display of the Mandelbrot set. It it quite feasible to group the whole number set by columns(or rows). As it demonstrated in Figure 1 and Figure 2, if the number of processors is 4, there will be one master process and three slave process. The whole number are orderly and equally distributed to each slave process. the slave processes receive the message from the master process and execute the procedure cal_pixel and save results in an array and then send the whole array to the master in one message. The master will use a wild card to accept messages from slaves in any order. When the master receives messages from slave, it will display the image. Since each process has different computation time, the image of each process will display unsimultaneously. The final image display will be completed until the master receive the last messages from the slave.

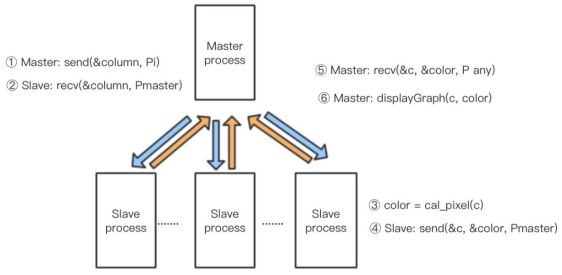

Figure 2: Execution Step of Mandelbrot Set Computation using MPI Method

2.2. Pthread

The Figure 3 and Figure 4 show the implementation of the Pthread method. Pthread method shares some similarity with the MPI method. We also assigned a fixed area of the whole area to each thread. In the Figure 3, we take the number of threads equal to 3. The main thread will fork

the target thread. Each target thread need to execute the procedure, cal_pixel which will compute the value of each given pixel and locaExecution, which will save the results in an array. The main thread will be blocked when not all the threads are joined. After all the threads join together, the image will be displayed in the main thread.

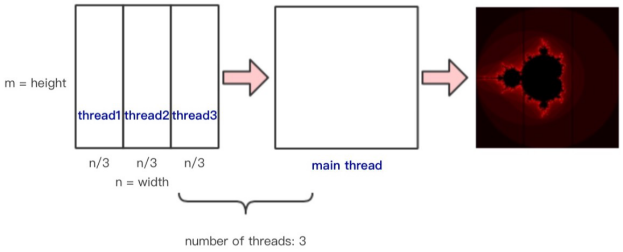

Figure 3: Program Design of Mandelbrot Set Computation using Pthread Method(eg. Number of Threads = 3)

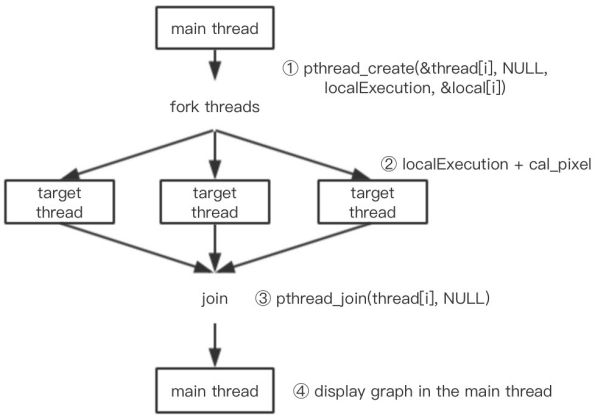

Figure 4: Execution Step of Mandelbrot Set Computation using Pthread method

3. Experiment

3.1. Execution Steps

Type "mpic++ hw2_mpi.cpp -lX11 -o ./hw2_mpi.out" to compile the MPI method.

Type "mpirun -n 3 ./hw2_mpi.out" to execute the MPI method.

Type "g++ hw2_pthread.cpp -lX11 -lpthread -o ./hw2_pthread.out" to compile the pthread method.

Type "mpirun -n 3 ./hw2_mpi.out" to execute the pthread method.

The default setting is not to show the result image. If the image needs to be printed, please change the bool variable PIC from false to true in each source code.

```
root@CSC4005:/home/cuhksz_csc/Desktop/2# g++ hw2_pthread.cpp -lX11 -lpthread -o ./hw2_pthread.out
root@CSC4005:/home/cuhksz_csc/Desktop/2# ./hw2_pthread.out
Name: Cui Yuncong
Student ID: 118010045
Assignment 2, Mandelbrot Set, Pthread implementation
runingTime: 0.116233
root@CSC4005:/home/cuhksz_csc/Desktop/2# mpic++ hw2_mpi.cpp -lX11 -o ./hw2_mpi.out
root@CSC4005:/home/cuhksz_csc/Desktop/2# mpirun -n 3 ./hw2_mpi.out
Name: Cui Yuncong
Student ID: 118010045
Assignment 2, Mandelbrot Set, MPI Implementation
runTime is 10.153004
```

Figure 5: The Command Line of Execution

3.2. Result Image

The Figure 6 and Figure 7 show the result images of the MPI and Pthread method. The size of the image in both case is 1000 * 1000.For mpi method, we set the test program with 3 processors(2 slave and 1 master). For pthread method, we set the test program with 2 threads.

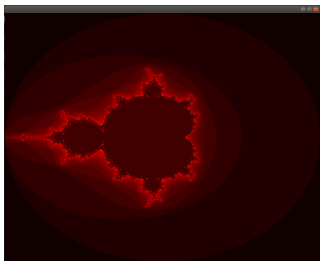

Figure 6: The Image of Mandelbrot Set Computation Generated by MPI Method

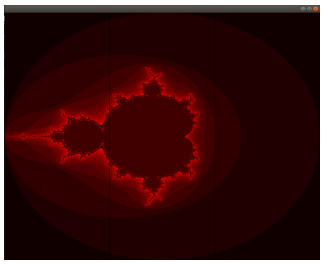

Figure 7: The Image of Mandelbrot Set ComputationGenerated by Pthread Method

3.3. Data Analysis

Noticed that the display of the image is excluded from the final evaluation since the drawing process is extremely timeconsuming on the cluster server and time spent on the computation (or communication) is quite trivial compared with the running time of the drawing. In this case, we will not take the drawing time into consideration. The performance analysis is conducted in two dimensions, varying in image size and the number of processors / threads.

Figure 8 and 9 shows the running time of the MPI, Pthread and sequential method as the image size grows. In

this figure, the running time of MPI, Pthread and sequential all grows when the image size grows. Noticed the value of x-coordinate is not proportionally increase. The running time of sequential method show like a polynomial curve. The running time of sequential method is in proportion to the image size.

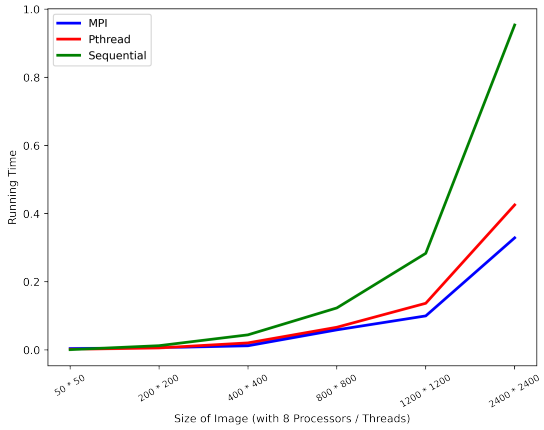

Figure 8: Performance Analysis of the MPI, Pthread and Sequential Method when Image Size Grows (with 8 Processors / Threads)

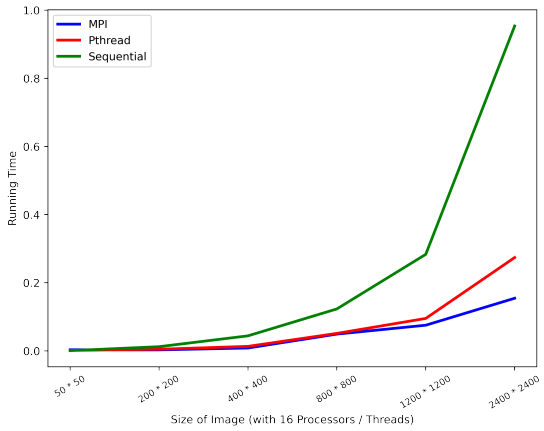

Figure 9: Performance Analysis of the MPI, Pthread and Sequential Method when Image Size Grows (with 16 Processors / Threads)

Figure 10 - 15 demonstrate the performance (running time) of the MPI, Pthread and sequential method when the image size is 50 * 50, 200 * 200, 400 * 400, 800 * 800, 1200 * 1200 and 2400 * 2400.

It should be highlighted that in all the figures below and below, the number of processors in MPI method means the number of processors used in computation since there is a master processor which is responsible for sending and receiving the the messages to or from all the slave processors. Just like Figure 1 shows, when the processors used for computations is 3(slave processor), the actual number of processors is 4(slave + master = 3 + 1 = 4). Since the drawing step is not taken into account, the time spent on the master processor is quite trivial. Thus, we only use the number of slave processor to conduct the performance analysis.

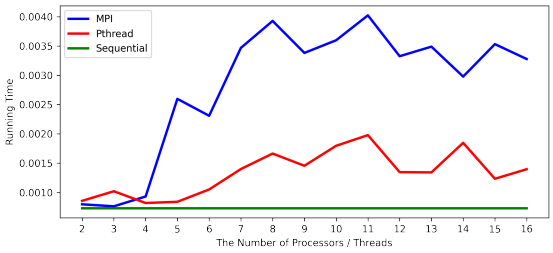

Figure 10: Performance Analysis of the MPI, Pthread and Sequential Method when Image Size is 50 * 50

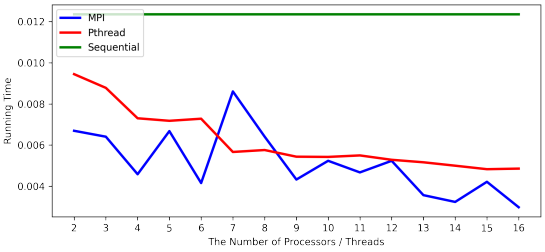

Figure 11: Performance Analysis of the MPI, Pthread and Sequential Method when Image Size 200 * 200

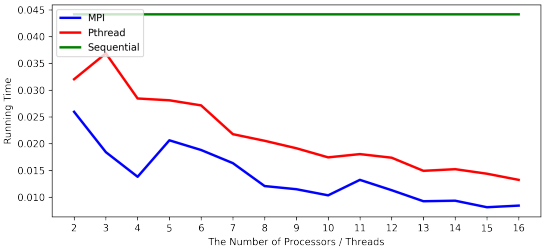

Figure 12: Performance Analysis of the MPI, Pthread and Sequential Method when Image Size is 400 * 400

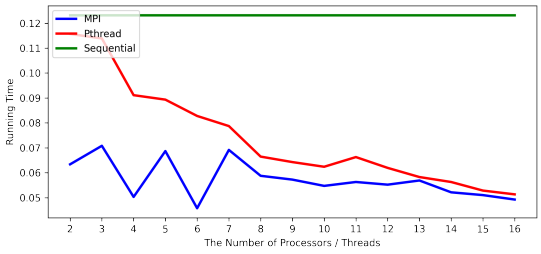

Figure 13: Performance Analysis of the MPI, Pthread and Sequential Method when Image Size is 800 * 800

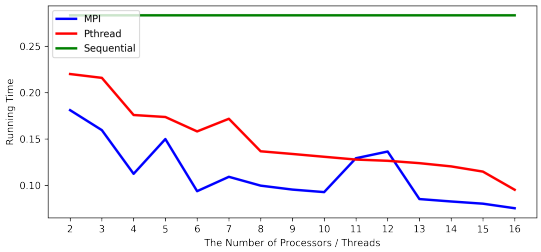

Figure 14: Performance Analysis of the MPI, Pthread and Sequential Method when Image Size is 1200 * 1200

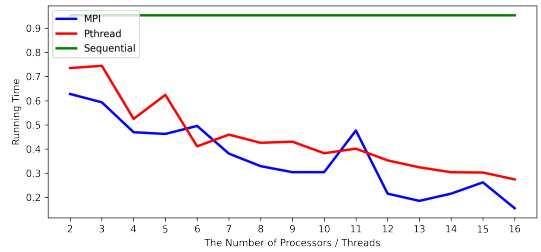

Figure 15: Performance Analysis of the MPI, Pthread and Sequential Method when Image Size is 2400 * 2400

In general these results exhibit reasonable speedup as the number of participating processes increases. The estimated execution-time is $O(nm)$ for serial program. A parallel implementation can be estimated by $O(nm/p)$ where n and m are size and p is number of processes.

As demonstrated in Figure 10 - 15, when the image size is small, the performance of pthread method is better than the MPI method. When the number of processors grows, the contrast is even distinct. And sequential method holds the best performance under this circumstance. Both of the sequential the Pthread method have a generally stable trend. The MPI method, in this case, have a worse performance when the number of processors grows. It is because when the image size is small, the time spent on computation is trivial compares to the time spent on communication. As the number of processors grows, the overhead of the communication is even heavier.

When the image size is medium and large, the ratio of the computation time is larger, the computation time's increment surpasses the communication time's increment. The running time of both MPI and Pthread show a general downward trend, and MPI method conducts a better performance than the Pthread method. The sequential method has the worst performance no matter under medium or large size of image.

It should be noticed that when image size is small, Pthread has a better performance than MPI but when the image size becomes medium and large, MPI has a performance gain. Its is because directly reading from the memory is faster than MPI_Send() and MPI_Recv(). When the image size grows, the computation time surpasses the communication time, that is why the MPI have a better performance.

4. Code Implementation

This section demonstrates the core code of MPI and Pthread method.

4.1. MPI

Figure 16 demonstrates the implementation of the master process. The master process send messages to the slave process and receive messages from it.

```
94  void masterExecution(double real_min, double real_max, double imag_min, double imag_max,
95                       int width, int height)
96  {
97      int receiveSize, index, receiveRank, realDataSize;
98      Local *localX = (Local *)malloc((comm_sz - 1) * sizeof(Local));
99      int *localInfo = (int *)malloc(sizeof(int) * 2);
100     int *colors = (int *)malloc(width * height * sizeof(int) );
101
102     for(int i = 0; i < (comm_sz - 1); i++ )
103     {
104         localX[i].start= i * local_width;
105         if (i * local_width + local_width > width)
106             localX[i].end= width - 1;
107         else
108             localX[i].end= (i * local_width) + local_width - 1;
109     }
110
111     for(int i = 0; i < (comm_sz - 1); i++)
112     {
113         localInfo[0] = localX[i].start;
114         localInfo[1] = localX[i].end;
115         MPI_Isend(&localInfo[0], 2, MPI_INT, (i+1), tag, MPI_COMM_WORLD, &request);
116     }
117
118     for(int i = 0; i < (comm_sz - 1); i++)
119     {
120         receiveSize = local_width * height;
121         int *results = (int *)malloc(receiveSize * sizeof(int));
122
123         MPI_Recv(results, receiveSize, MPI_INT,MPI_ANY_SOURCE, tag, MPI_COMM_WORLD,&status);
124         receiveRank = status.MPI_SOURCE;
125         index =  localX[receiveRank -1].start *height;
126         realDataSize =  localX[receiveRank -1].end - localX[receiveRank-1].start + 1;
127         memcpy(colors + index , results, realDataSize*height*sizeof(int)) ;
128
129         free(results);
130         results=NULL;
131     }
132 }
```

Figure 16: The Implementation of the Master Process

Figure 17 demonstrates the implementation of the slave process. The slave process is responsible for the result in each part.

```
void slaveExecution(double real_min, double real_max, double imag_min, double imag_max,
                    int width, int height)
{
    int *localInfo = (int *)malloc(sizeof(int) * 2);
    MPI_Recv(&localInfo[0],  2, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    int matrixSize = local_width * height;
    int *colors = (int *)malloc( matrixSize * sizeof(int) );
    calculation(real_min, real_max, imag_min, imag_max, localInfo[0], localInfo[1],
                width, height, colors);
    MPI_Isend(colors, matrixSize , MPI_INT, MASTER, tag, MPI_COMM_WORLD,&request);
}
```

Figure 17: The Implementation of the Slave Process

4.2. Pthread

Figure 18 demonstrates the implementation of the each thread. Figure 19 demonstrates the implementation of the fork and join in the main thread.

```
void *localExecution(void *idp)
{
    Local *local = (Local*)idp;
    Compl z, c;
    int repeats;
    double scale_real = (local->xEnd - local->xStart ) / local->width;
    double scale_imag = (local->yEnd - local->yStart) / local->height;
    double x, y;

    int local_width = local->width / local->numThreads;
    if (local->width % local->numThreads != 0)
        local_width = local_width + 1;

    int start = local_width * local->threadId;
    int end;
    if ((local_width * local->threadId + local_width) > local->width)
        end = local->width-1;
    else
        end = local_width * local->threadId + local_width-1;

    for (int i = start; i < end; i++)
    {
        for (int j = 0; j < local->height; j++)
        {
            x= local->xStart + i * scale_real;
            y= local->yStart + j * scale_imag;
            z.real = 0.0;
            z.imag = 0.0;
            c.real= x;
            c.imag = y;
            repeats = cal_pixel(c);
            int index = (i * local->height + j);
            local->output[index] = repeats;
        }
    }
}
```

Figure 18: The Implementation of Each Thread

```
void MandelbrotPthread(double real_min, double real_max, double imag_min,
                       double imag_max, int width, int height, int *output)
{
    int rc;
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    pthread_t threads[NUM_THREADS];
    struct timeval timeStart, timeEnd, timeSystemStart;
    double runTime=0, systemRunTime;
    gettimeofday(&timeStart, NULL );

    Local *local = (Local *)malloc(sizeof(Local) * NUM_THREADS);

    for (int i = 0; i < NUM_THREADS; i++)
    {
        local[i].xStart = real_min;
        local[i].xEnd = real_max;
        local[i].yStart = imag_min;
        local[i].yEnd = imag_max;
        local[i].width = width;
        local[i].height = height;
        local[i].output = output;
        local[i].threadId = i;
        local[i].numThreads = NUM_THREADS;
    }

    for (int i = 0; i < NUM_THREADS; i++)
        rc = pthread_create(&threads[i], NULL, localExecution, &local[i]);

    for (int i = 0;i < NUM_THREADS; i++)
        pthread_join(threads[i], NULL);

    gettimeofday( &timeEnd, NULL );
    runTime = (timeEnd.tv_sec - timeStart.tv_sec ) + (double)(timeEnd.tv_usec
              - timeStart.tv_usec) / 1000000;
    printf("runingTime: %lf\n", runTime);

    pthread_attr_destroy(&attr);
}
```

Figure 19: The Implementation of the Main Thread

5. Conclusion

The advancement of multicore systems demands applications with more threads. In order to facilitate this demand, parallel programming models are developed.

In this project, the Mandelbrot Set image is displayed by using MPI, Pthread and sequential method. The above performance analysis clarifies why the MPI method will show a downward trend of performance and both of the Pthread and sequential have better performance than MPI when the image size is small. When the image size grows larger, both of the MPI method and Pthread method shows a general downward trend of running time. The MPI method conducts a best performance and sequential performs worst.