

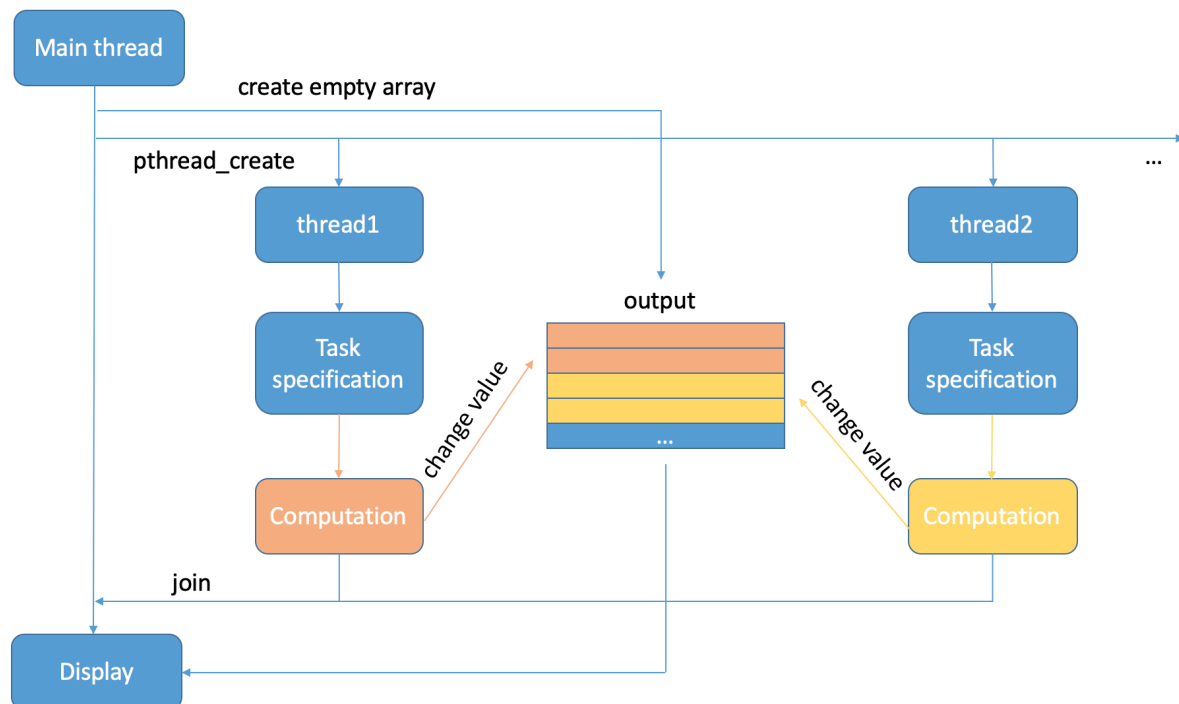
CSC4005 Assignment2 Report

Introduction

The objective of this assignment is to visualize the Mandelbrot Set on an image by doing computations on each pixel and drawing each pixel on the display with Xlib. Since the computation work of each pixel is independent, it could be better if parallel computation is adopted. In this assignment, two versions of Mandelbrot Set are implemented: Pthread version and MPI version. This report is going to elaborate the detailed implementations and analyse the performances of different versions.

Program design

Pthread version



The above figure shows the basic logic of the pthread-version program. In this implementation, the computation tasks is divided in terms of rows. However, the main thread does not do the work of distributing the tasks. Each thread will do the task specification by itself according to its id, so that it could determine the target rows. The following is the detailed implementations:

1. Global variables preparation:

```
int NUM_THREADS = 1; // number of threads, default: 1
int SIZE = 200;      // the size of the image, default: 200*200
```

2. In the main thread, check whether there is a input parameter. Change the global variable if there is a input:

```
int NUM_THREADS = 1;
int SIZE = 200;
```

3. Do some preparation for Xlib use, and then start timing.
4. Prepare an empty array, which is going to be used to plot the image later:

```
int *output=(int *)malloc(sizeof(int) * (SIZE * SIZE));
```

(NOTE: For the programs in this assignment, it is assumed that all the images are square)

5. Start to create pthread, each thread execute the function *cal_func()*:

```
int i, rc;
pthread_t thread[NUM_THREADS];
thread_data input_data[NUM_THREADS];

for (i = 0; i < NUM_THREADS; ++ i) {
    input_data[i].thread_id = i; // deliver the id info
    input_data[i].data = output; // deliver the output array
    rc = pthread_create(&thread[i], NULL, cal_func, &input_data[i]);
    if (rc) {
        fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
        return EXIT_FAILURE;
    }
}
```

the main thread deliver the id info and the output array (pointer) to each thread, so each thread could operate on the initial array according to its own id.

6. In each thread (or in function *cal_func()*):

1. Task specification:

according to the id of the thread, each thread could figure out the target rows, by computing the start position and the end position.

```
int start_row, end_row, num_rows, remain;
thread_data *input_data = (thread_data *) arg; // get the input data

num_rows = SIZE / NUM_THREADS; // the number of target rows
remain = SIZE % NUM_THREADS;
// if the total number of rows is not the multiple of the number of
// threads,
// then some threads may compute one more row.
if (input_data->thread_id < remain) {
    num_rows ++;
```

```

        start_row = num_rows * input_data->thread_id;
    }else {
        start_row = num_rows * input_data->thread_id + remain;
    }

    end_row = start_row + num_rows;

```

2. Computation on each pixel in the target rows:

```

Compl    z, c;
int i, j, k;
double  lengthsq, temp;

for (i = start_row; i < end_row; ++i){
    for (j = 0; j < SIZE; ++j){
        z.real = z.imag = 0.0;
        c.real = ((float) j - SIZE/2)/(SIZE/4);
        c.imag = ((float) i - SIZE/2)/(SIZE/4);
        k = 0;
        do {
            temp = z.real*z.real - z.imag*z.imag + c.real;
            z.imag = 2.0*z.real*z.imag + c.imag;
            z.real = temp;
            lengthsq = z.real*z.real+z.imag*z.imag;
            k++;
        } while (lengthsq < 12 && k < 100); //lengthsq and k are the
threshold
        if (k >= 100) {
            input_data->data[i*SIZE+j]=1; // mutate the output array
        }
    }
}

```

7. Join the threads:

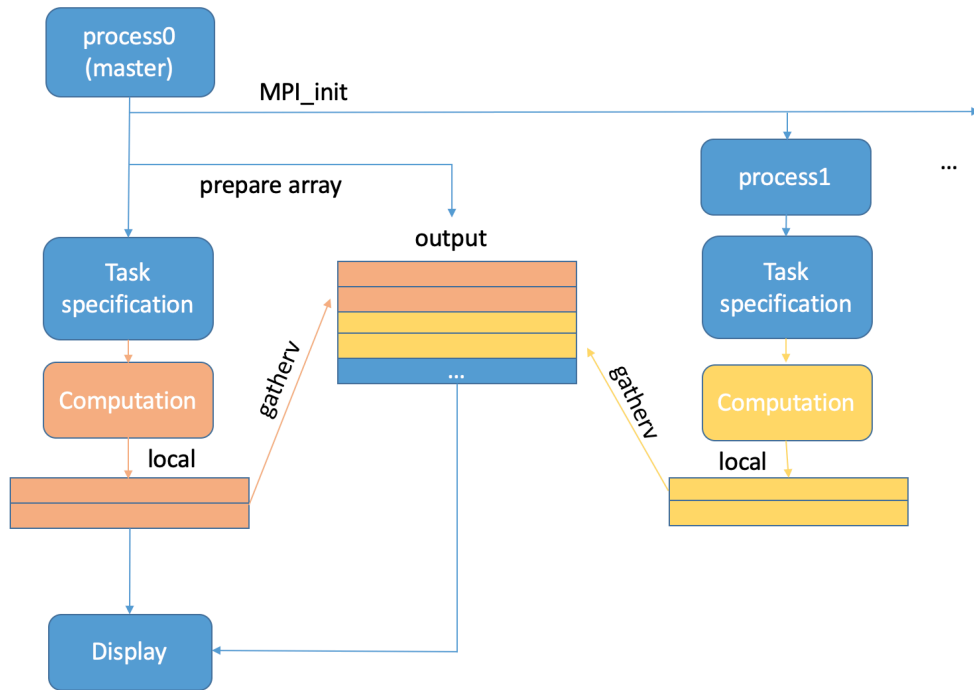
```

for (i = 0; i < NUM_THREADS; ++i) {
    void *return_data;
    pthread_join(thread[i], &return_data);
}

```

8. Stop timing, print out the result and draw the image.

MPI version



The above figure shows the basic logic of the MPI-version program. Similar to pthread-version, in this implementation, the computation tasks is divided in terms of rows and each process works on the task specification by itself according to its own rank. However, here local array was used to store the result of computation and it would be gathered at last. The following is the detailed implementations:

1. The master process do some preparation for Xlib use and then start timing.
2. Initialize the MPI, and get the number of tasks and the process rank (taskid):

```

MPI_Init(&argc, &argv);
int taskid,
numtasks;
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);

```

3. Check whether there is a input parameter to determine the size of image:

(The checking is put over here since only after MPI_Init() the normal parameters could be read)

```

int SIZE = 200; // default size is 200*200
if (argc > 1) SIZE = atoi(argv[1]);

```

4. Task specification, similar to pthread version, except that here a local array is prepared to store the computation result:

```

Compl    z, c;
int i, j, k, quotient, remainder;

```

```

int count[numtasks]; // store the number of target rows for each process
int disp[numtasks]; // store the start position of each process
int * local_data;    // store the computation result locally

disp[0] = 0;
quotient = SIZE / numtasks;
remainder = SIZE % numtasks;
// fill the chart 'count' and 'disp', which will be used in 'gatherv()'
later
for (i = 0; i < numtasks; ++ i) {
    if (i < remainder) {
        count[i] = (quotient + 1) * SIZE; // one more row
    }else{
        count[i] = quotient * SIZE;
    }
    if (i > 0) disp[i] = disp[i-1] + count[i-1];
    if (taskid == i) local_data = (int *)malloc(sizeof(int) * count[i]);
}
int start_row = disp[taskid] / SIZE;
int end_row = start_row + count[taskid] / SIZE;

```

5. Computation on the target rows, same as pthread version, except that it store the result in the local array *local_data*:

```

... // for details, please refer to the computation part of pthread
version
if (k >= 100)
    local_data[(i-start_row)*SIZE+j]=1;
...

```

6. Gather all the local data to the output array in the master process:

```

MPI_Gatherv(local_data, count[taskid], MPI_INT, output, count, disp,
MPI_INT, 0,
            MPI_COMM_WORLD);

```

7. In the master process, stop timing, print out the results and draw the image.

How to run

Pthread version

1. In the directory of *pthread_version*, type the following command and enter to compile the program:

```

g++ mandelbrot_pthread.cpp -lX11 -lpthread -o ./a.out

```

then an executable file *a.out* will be generated in the directory.

2. In the same directory, type the following command and enter to run the program:

```
./a.out (SIZE) (NUM_THREADS)
```

where SIZE and NUM_THREADS are optional:

- SIZE could specify the size of image.
- NUM_THREADS could specify the number of threads used to compute.
(NOTE: to specify the NUM_THREADS, the SIZE should be specify first)
- The default setting is: SIZE = 200, NUM_THREADS = 1.
- For example,

```
./a.out 1000 4
```

this command would display an image with size 1000×1000 by using 4 threads to run.

MPI version

1. In the directory of *mpi_version*, type the following command and enter to compile the program:

```
mpic++ mandelbrot_mpi.cpp -lX11 -o ./a.out
```

then an executable file *a.out* will be generated in the directory.

2. In the same directory, type the following command and enter to run the program:

```
mpirun -np N ./a.out (SIZE)
```

where N specifies the number of processes and SIZE is optional to specify the size of image.

Sample output

Running environment

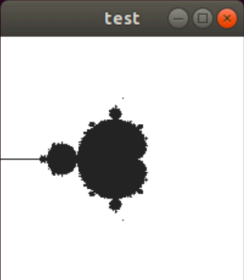
The sample is running on the virtual machine (with *Xlib* and *mpich-3.2.1* installed):



Pthread version

```
File Edit View Search Terminal Help
cuhksz_csc@CSC4005: ~/Documents
cuhksz_csc@CSC4005:~/Documents$ g++ mandelbrot_pthread.cpp -lX11 -lpthread -o ./a.out
mandelbrot_pthread.cpp: In function 'int main(int, char**)':
mandelbrot_pthread.cpp:72:36: warning: ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
    char                *window_name = "test", *display_name = NULL;
    ~~~~~^
/* initialization for a window */

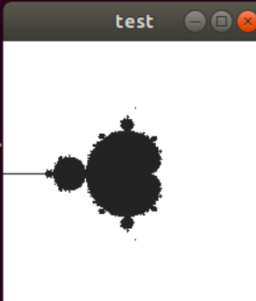
cuhksz_csc@CSC4005:~/Documents$ ./a.out
Name: Hongpeng Yi
Student ID: 117020343
Assignment 2, Mandelbrot Set, Pthread implementation.
RunTime is 0.006626s, with 1 threads and size of 200*200.
█
```



MPI version

```
File Edit View Search Terminal Help
cuhksz_csc@CSC4005: ~/Documents
cuhksz_csc@CSC4005:~/Documents$ mpic++ mandelbrot_mpi.cpp -lX11 -o ./a.out
mandelbrot_mpi.cpp: In function 'int main(int, char**)':
mandelbrot_mpi.cpp:18:36: warning: ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
    char                *window_name = "test", *display_name = NULL;
    ~~~~~^
/* initialization for a window */

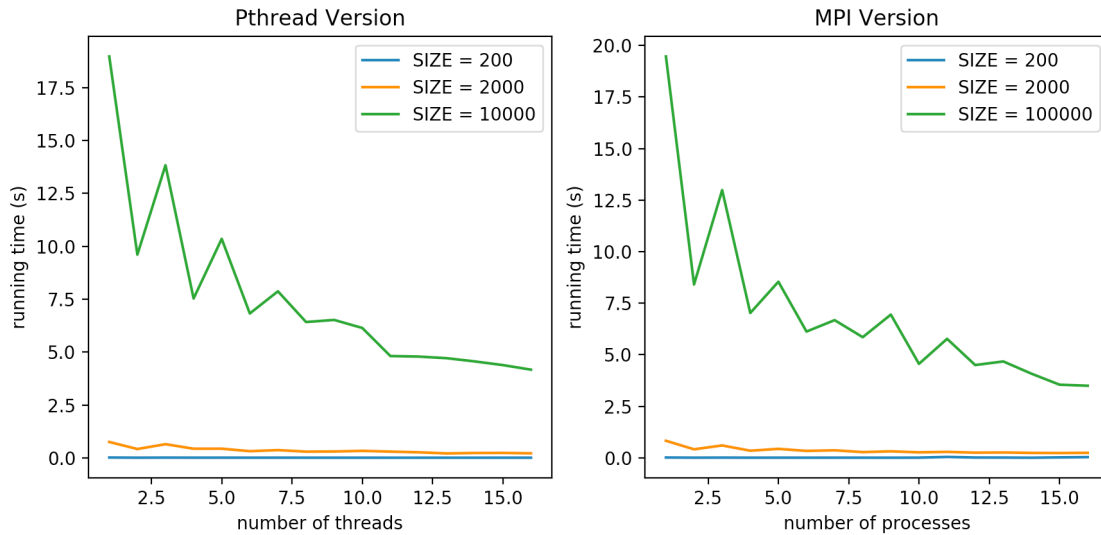
cuhksz_csc@CSC4005:~/Documents$ mpirun -np 2 ./a.out
Name: Hongpeng Yi
Student ID: 117020343
Assignment 2, Mandelbrot Set, MPI implementation.
RunTime is 0.003595s, with 2 processes and size of 200*200.
█
```



Performance analysis

Performance with different number of threads/processes and data size:

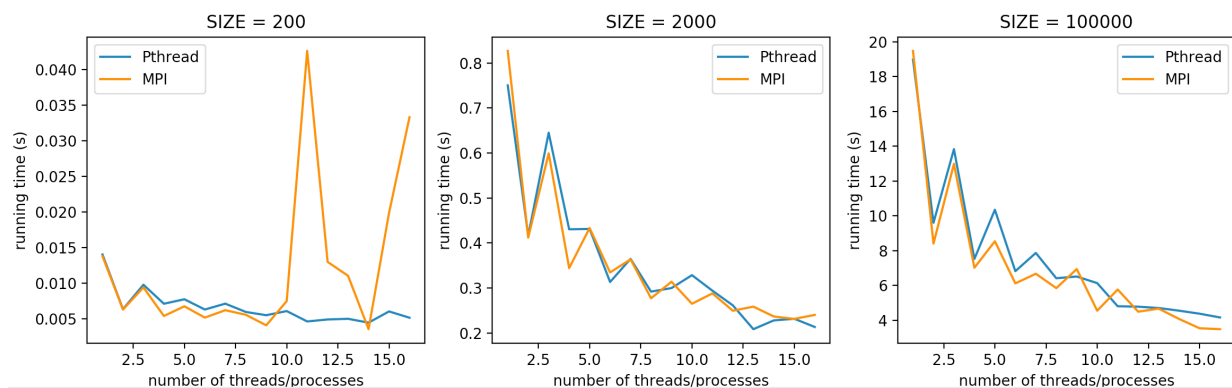
To analyze the performance of these two versions of programs, the programs are run with three different sizes 200, 2000, 10000 and with 1 - 16 processes/threads respectively, and finally get 96 data points plotted as graph:



According to the graph above, which shows the each change of running time with different data sizes, as the number of processes/threads increase , we could get some information:

- Observing the trend of the curve, it is roughly decreasing as the number of threads/processes increases. This does make sense that multithreads or multiprocesses divided the task so that increase the running speed.
- Observing the shape of the curve, it fluctuates a lot especially when the number of threads/processes change from even number to odd number. This may be caused by the weakness of static schedule, it could be regarded as a point for futher improvements.
- Comparing the performance of different data size, the larger the data size, the longer the running time. And the running time increases faster than data size. This shows that the time complexity of this algorithm is larger than $O(n)$. In fact, according to the program design, the complexity is probably $O(n^2)$ since the image is a square.

MPI vs Pthread



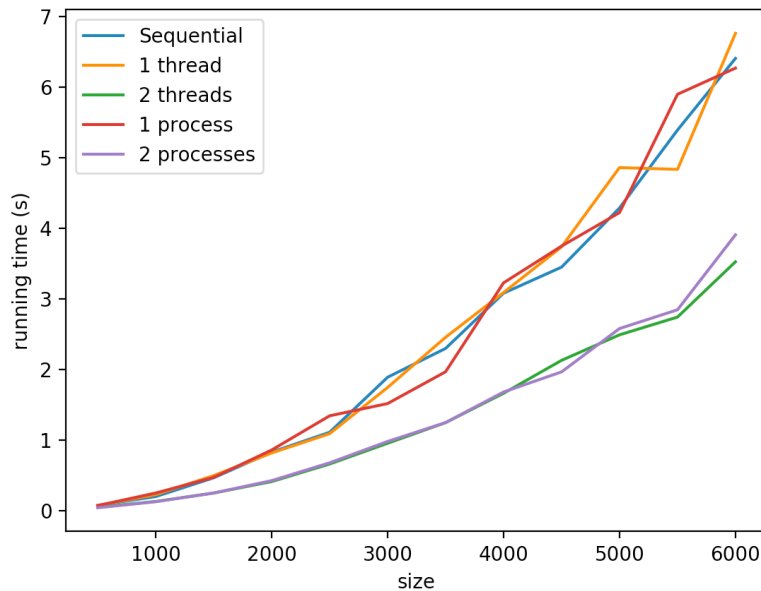
With the same data points, comparing the performance between Pthread version and MPI version, we could get some more information:

- According to the graph of 'SIZE = 200', When the data size is small, the running time of MPI fluctate a lot, and it becomes very high at some points with large number of processes. This could be explained as the communication cost of MPI, which was magnified when the data size is too small since in this case the running time is dominated by communication time. The communication time could be flattened with large running time, as the other two graphs show.

- According to the graph of medium(2000) and large(10000) data size, we could find that they performed very similarly when data size become large.

Sequential vs Parallel

To evaluate the performance of parallelism, the three versions of program were run on data size from 500 to 6000 with interval 500.



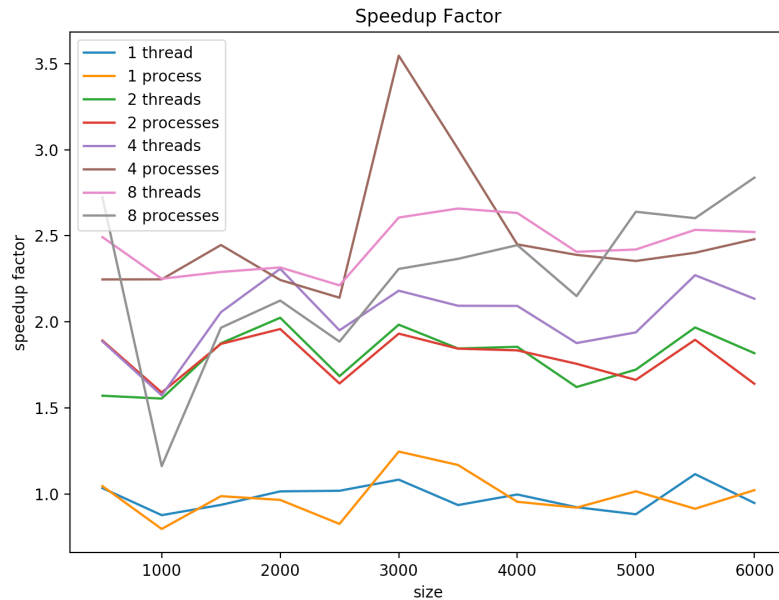
Here 5 curves have been plotted, the curves of 1 thread and 1 process could be regarded as sequential version of pthread and MPI, and the curves of 2 threads and 2 processes could represent the parallel version.

- Comparing 1-thread curve and 1-process curve with sequential curve, we could find that, these three curves are very similar to each other, which means that when the number of threads/processes is 1, the Pthread and MPI programs do not obviously waste some unnecessary time. We could somehow conclude that the running time of the sequential part of Pthread version and MPI version is similar to that of sequential version.
- Comparing 2-threads curve and 2-processes curve with sequential, it is obvious that the running time reduced a lot when parallel version is adopted. We could say that these two parallel implementations, pthread and MPI, do a good job on speed up the running of the program.

Speedup factor

In the comparison between parallel version and sequential version, *speedup factor* could give a more clear insight about how the parallel computation performs, which measures the increase in speed by using multiprocessors.

The *speedup factor* is given as $S(n) = t_s / t_p$, where t_s is the execution time on a single processor (sequential) and t_p is the execution time on a multiprocessor. Therefore, the speedup factors for different numbers of threads/processes with different data size could be computed, and they were plotted as follows:

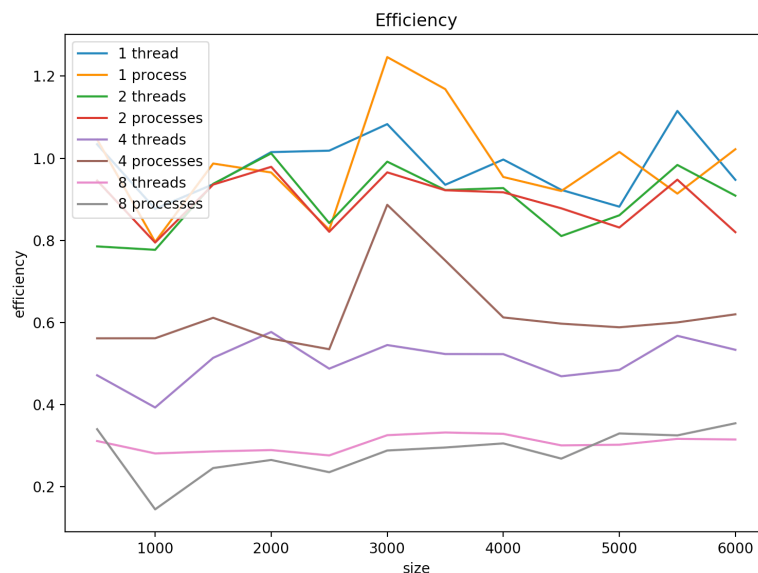


According to the graph, we could find that there is an apparent speedup when using multiprocessors, so the parallel computation could make a considerable increase on running speed, and the speedup remains around the same level with different data size.

Efficiency

The graph of speedup factor indicated that the parallel computation could increase the running speed a lot. However, it seems that more processes added later does not result in an impressive improvement. To specify whether the added processes make reasonable contributions, we use *efficiency* to evaluate, which gives fraction of time that processors are being used on computation.

The *efficiency* is given as $E = t_s / (t_p * n) = S(n) / n$, by calculating the efficiency, the following graph could be generated:



According to the graph of efficiency, we could find that:

- When the number of threads/processes is small, the program has a high efficiency.
- When the number of threads/processes is large, the program has a low efficiency.
- The larger the number of processes is, the lower the efficiency.

The reason for these observations may be that:

- When the number of threads/processes is small, the program spend relatively long time on parallel part of the computation, the sequential time is relatively short so that it could be ignored. Therefore,
the newly added process can contribute a lot to reducing the running time of parallel part as well as the runing time whole program.
- When the number of threads/processes is large, the program has reduce the running time of the parallel part a lot, then the running time of sequential part become significant but could not be improved by adding process. Therefore, the newly added process now only contribute a little to reducing the running time of parallel part, which impact less on the running time of the whole program.

Conclusion

In this assignment, the two implementations of Mandelbrot Set, MPI version and Pthread version, have been successfully designed and test. And the performance analysis on these two programs and sequential version have been presented. The parallel computation does a good job on speedup and it is efficient when small amount of processors are used. There are also some possible improvements could be considered, like dynamic scheduling, which is not covered in this experiment. It could be necessary for further improvement.