# COMP3230A Principles of Operating Systems

# Programming Project – Multiprocessing and multithreaded programs

## Total 17 points

(version: 1.0)

## Objectives

1. An assessment related to ILO4 [Practicability] – "demonstrate knowledge in applying system software and tools available in modern operating system for software development".
2. A learning activity related to ILO 2a and ILO 2c.
3. The goals of this programming project are:
   - to have hand-on practice in designing and developing multiprocessing and multithreaded programs;
   - to learn how to use various Unix system functions to create processes as well as to coordinate and interact between processes;
   - to learn how to use POSIX pthreads library to create, manage, and coordinate multiple threads in a shared memory environment;
   - to experience the issues related to synchronization and coordination between processes or threads;
   - to consolidate your understanding on how to structure the multithreaded program in the bounded-buffer problem.

### Compute-Intensive Task

Compute-intensive application is application that demands a lot of computation, which is a good candidate to be executed in parallel, i.e., the computation workload is divided into several smaller computation tasks and executed by multiple processor cores at the same time. In this project, we pick the Mandelbrot set as the compute-intensive task. Your task is to write two programs, a multiprocessing program and a multithreading program, which make use of multiple CPU cores to speed-up the computation in generating the Mandelbrot set image.

Mandelbrot set is a set of points in a complex plane that are quasi-stable (i.e., will increase and decrease, but not exceed some limit) when computed by iterating the function

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

where $z_{k+1}$is the (k+1)th iteration of the complex number $z = a + bi$, $z_k$ 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 of $z$ is zero. The iterations are continued until the magnitude of $z$ is greater than 2 (which indicates that $z$ will eventually become infinite) or the number of iterations reaches some arbitrary limit (e.g., 50000). The magnitude of $z$ is the length of the vector given by

$$|z| = \sqrt{a^2 + b^2}$$

The computation goes through each point ($c$) in the complex plane and determines whether that point is in or not in the Mandelbrot set.

## Specification

The project consists of two parts:

- Part 1 - Convert the sequential program to a multiprocessing program by using fork(), signal()/sigaction(), wait() and pipe() system functions to manage and coordinate all processes to achieve the task. For the multiprocessing program, we are going to use the Boss-Workers model to dynamically distribute the computation tasks to a pool of worker processes.
- Part 2 – Convert the sequential program to a multithreaded program, which uses producer/consumer model for coordination between threads as well as distribution of computation tasks to idle worker threads.

### **Preparation**

Download the sequential program - part0-mandelbrot.c and the two header files: draw.h and Mandel.h from the Course's web site at moodle.hku.hk. The header file "Mandel.h" includes the function *Mandelbrot( c )* for determining whether the point *c* in the complex plane is in the Mandelbrot set; we shall use the returned value for visualization. The header file "draw.h" includes the function *DrawImage( )* for displaying the resulting 2D image. For the project, you do not need to make changes to these two files. The program file "part0-mandelbrot.c" contains the logic in computing the 2D Mandelbrot image. Here is the pseudocode of the program:

```
1.  Measure the startup time of the process
2.  Allocate memory to store the 2D image
3.  Measure the starting time of the Mandelbrot computation
4.  For each pixel in the 2D complex plane
    4.1.  Determine the result value from the Mandelbrot computation
5.  Measure the end time of the Mandelbrot computation and report the total computation time
6.  Measure the end time and report the overall time
7.  Draw the image
```

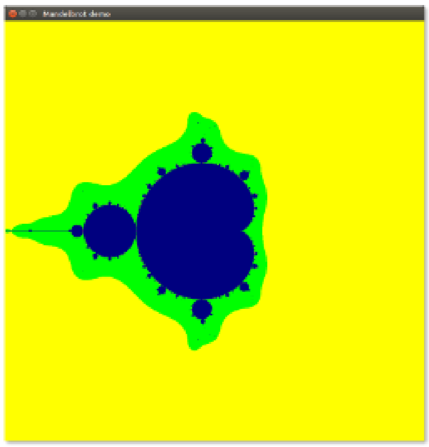
To compile the program, the system must have the SDL2 (Simple DirectMedia Layer version 2) library installed in it as the program uses the SDL2 graphic library to draw the Mandelbrot image. Both the course's VM and the department Ubuntu servers have installed this library already. Use this command to compile the program:

```
gcc  part0-mandelbrot.c  -o 0mandel  -l SDL2  -l m
```

where 0mandel is the output filename for the binary. To run the program, enter this command in a terminal: **./0mandel**

Here is the example output display of the image and the printout of the program.

```
Start the computation ...
 ... completed. Elapse time = 24013.668 ms
Total elapse time measured by the process = 24013.723 ms
Draw the image
```



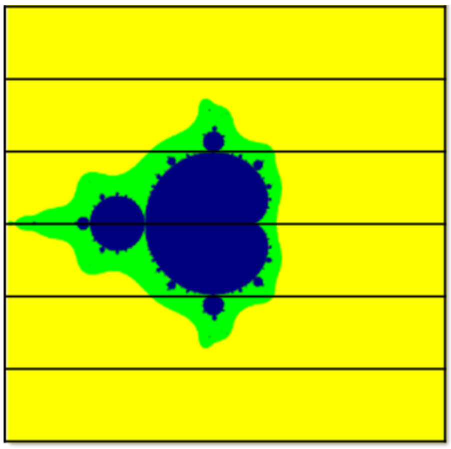
Examine the sequential program and understand its execution logic before moving to Part 1.

### **Part One (9 points)**

In this part, we are going to write a multi-processes program to speed-up the computation. We divide the development process into two stages. In the first stage, we statically divide the workload to all child processes and return all computation results through a pipe to the parent process for display. In the second stage, we adopt a dynamic scheme to distribute the workload to any idle child processes and return the computation results to the parent process for display.

**Stage A**

We can view the computation effort in finding the results of a row of pixels as an independent task. To improve the performance, we evenly divide the workload into blocks of rows and assign each block to a child process to work on. For example, if there are M rows of pixels in the 2D image and N child processes to work on the computation, each child process would get $M/N$ rows of pixels.



Once we know how many rows of pixels each child process will work on, the parent process creates all child processes. Each child process has to determine which block of rows of pixels it is going to work on and then it starts the computation. When a child process finished the computation, it passes the computation results row-by-row back to the parent process (through the Data pipe) and then terminates.

Since all child processes place their results to the pipe, we need to have a standard message format to store each result. For example, we can define a structure to store the results of one row of pixels:

```
typedef struct message {
   int row_index;
   float rowdata[IMAGE_WIDTH];
} MSG;
```

After creating all child processes, the parent process waits and collects all the results from the Data pipe. Then, it waits for all child processes to terminate (using *wait()* or *waitpid()*), collects the execution statistics, and finally display the resulting image.

Like the sequential program, we would like to measure how much time each child process spent in working on the computation and how much time the whole application spent in generating the 2D image (except the drawing part). Each child process should measure the time it spent in computing the results of the block of rows and print out the timing. The parent process should collect the CPU resource usage statistics of its child processes and itself (using *getrusage()*) and print out the timings. Here is the example printout of the stage A program with 5 child processes:

```
Child(13993) : Start the computation ...
Start collecting the image lines
Child(13997) : Start the computation ...
Child(13995) : Start the computation ...
Child(13994) : Start the computation ...
Child(13996) : Start the computation ...
Child(13993) :        ... completed. Elapse time = 7.028 ms
Child(13997) :        ... completed. Elapse time = 7.432 ms
Child(13994) :        ... completed. Elapse time = 3507.499 ms
Child(13996) :        ... completed. Elapse time = 3629.017 ms
Child(13995) :        ... completed. Elapse time = 17327.031 ms
All Child processes have completed
Total time spent by all child processes in user mode = 24460.000 ms
Total time spent by all child processes in system mode = 4.000 ms
Total time spent by parent process in user mode = 4.000 ms
Total time spent by parent process in system mode = 8.000 ms
Total elapse time measured by parent process = 17330.760 ms
Draw the image
```

You can name the stage A program to part1a-mandelbrot.c. To allow us to play around with a different number of child processes, stage A program should accept one input argument, which is the number of child processes to be created.

**Stage B**

You should find that although we do have certain performance improvement, the measured speed-up is not that impressive even with 5 child processes. This is due to the imbalance in the computation workload. Some child processes got more work to do than the others.

In stage B, you are going to extend stage A program to use the Boss-Workers model for dynamically distributing the computation workload to idle child processes. In a Boss-Workers model of program design, the parent process works as the boss and assigns tasks to child processes (workers). Instead of creating a lot of child processes, the parent process creates a fixed number of workers regardless of the number of tasks. In the project, we can have any number of workers between 1 and 10. Each worker waits for the signal (SIGUSR1) from the boss, then picks up a task (i.e., a block of rows of pixels) from the Task pipe, performs the computation, returns computation results row-by-row through the Data pipe (another pipe) to the parent process, and the cycle repeats until the child process receives a termination signal (SIGINT) sent by the parent process.

Same as in stage A, parent and child processes are communicating via pipes. We use two pipes – Task pipe and Data pipe to reduce the complexity in synchronization between the boss and workers. For the Task pipe, only the boss is the writer and all workers are readers. For the Data pipe, the boss is the only reader and all workers are writers (same as stage A). You have to design two data structures for the communications between the boss and workers through the two pipes. One is for specifying the task - the block of rows; when a worker picks up a task, it needs to know which row to start at and how many rows it has to take care of in a task. This is an example data structure to serve for this purpose:

```
typedef struct task {
      int start_row;
      int num_of_rows;
} TASK;
```

The other data structure is for passing the computation results to the parent process. As we would like to distribute the tasks dynamically, the boss needs to find out which worker(s) is/are idle, then it assigns a new task to the worker. The trick is that when a worker returns the computation results to the boss, it includes its identity in the message that carries the computation results of the last row of the just completed task. Once the boss notices that, it can place a new task in the Task pipe and uses the SIGUSR1 signal to inform the worker to work on the new task. Here is an example data structure:

```
typedef struct message {
      int row_index;
      pid_t child_pid;
      float rowdata[IMAGE_WIDTH];
} MSG;
```

A special requirement of this part is to use signals to synchronize between the boss and workers. We use the signal – SIGUSR1 to inform a worker that a new task has been placed in the Task pipe. Thus, a worker only gets a task to work on when it receives the SIGUSER1 signal. To terminate child processes, the boss sends SIGINT signals to its child processes. When a child process receives SIGINT signal, it prints out a message to the display and returns the total number of tasks it has performed as its termination status before termination. On the other hand, the boss collects the termination statuses of all child processes and outputs them to the display.

Same as in stage A, each worker prints out the computation time spent in handling a task and the boss collects the CPU resource usage statistics of its child processes and itself and prints out the timings. Here is the example printout of the stage B program with 5 child processes:

```
Child(21866) : Start up. Wait for task!
Child(21867) : Start up. Wait for task!
Child(21868) : Start up. Wait for task!
Child(21869) : Start up. Wait for task!
Start collecting the image lines
Child(21866) : Start the computation ...
Child(21868) : Start the computation ...
Child(21869) : Start the computation ...
Child(21867) : Start the computation ...
Child(21870) : Start up. Wait for task!
Child(21870) : Start the computation ...
Child(21866) :          ... completed. Elapse time = 0.876 ms
Child(21868) :          ... completed. Elapse time = 0.910 ms
Child(21867) :          ... completed. Elapse time = 0.954 ms
Child(21869) :          ... completed. Elapse time = 0.976 ms
Child(21870) :          ... completed. Elapse time = 1.006 ms
Child(21866) : Start the computation ...
Child(21867) : Start the computation ...
Child(21868) : Start the computation ...
Child(21869) : Start the computation ...
Child(21870) : Start the computation ...
Child(21866) :          ... completed. Elapse time = 1.026 ms
Child(21866) : Start the computation ...
Child(21867) :          ... completed. Elapse time = 1.044 ms
Child(21868) :          ... completed. Elapse time = 1.131 ms
Child(21869) :          ... completed. Elapse time = 1.148 ms
Child(21867) : Start the computation ...
Child(21868) : Start the computation ...
Child(21869) : Start the computation ...
Child(21866) :          ... completed. Elapse time = 5.031 ms
Child(21870) :          ... completed. Elapse time = 5.663 ms
Child(21866) : Start the computation ...
Child(21870) : Start the computation ...
Child(21867) :          ... completed. Elapse time = 141.016 ms
Child(21867) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 389.282 ms
Child(21868) : Start the computation ...
Child(21869) :          ... completed. Elapse time = 499.403 ms
Child(21869) : Start the computation ...
Child(21866) :          ... completed. Elapse time = 1361.569 ms
Child(21866) : Start the computation ...
Child(21870) :          ... completed. Elapse time = 1568.870 ms
Child(21870) : Start the computation ...
Child(21867) :          ... completed. Elapse time = 1782.524 ms
Child(21867) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 2054.256 ms
Child(21868) : Start the computation ...
Child(21869) :          ... completed. Elapse time = 2583.502 ms
Child(21869) : Start the computation ...
Child(21866) :          ... completed. Elapse time = 2798.525 ms
Child(21866) : Start the computation ...
Child(21870) :          ... completed. Elapse time = 2719.551 ms
Child(21870) : Start the computation ...
Child(21867) :          ... completed. Elapse time = 2559.109 ms
Child(21867) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 2131.709 ms
Child(21868) : Start the computation ...
Child(21869) :          ... completed. Elapse time = 1805.886 ms
Child(21869) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 338.953 ms
Child(21868) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 1.636 ms
Child(21868) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 3.121 ms
Child(21868) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 0.636 ms
Child(21868) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 0.628 ms
Child(21868) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 0.553 ms
Child(21868) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 0.540 ms
Child(21868) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 0.536 ms
Child(21868) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 0.535 ms
Child(21868) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 0.534 ms
Child(21868) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 0.495 ms
Child(21868) : Start the computation ...
Child(21868) :          ... completed. Elapse time = 0.479 ms
Child(21867) :          ... completed. Elapse time = 502.480 ms
Child(21869) :          ... completed. Elapse time = 113.751 ms
Child(21870) :          ... completed. Elapse time = 1414.422 ms
Child(21866) :          ... completed. Elapse time = 1602.743 ms
Process 21866 is interrupted by ^C. Bye Bye
Process 21867 is interrupted by ^C. Bye Bye
Process 21868 is interrupted by ^C. Bye Bye
Process 21869 is interrupted by ^C. Bye Bye
Process 21870 is interrupted by ^C. Bye Bye
Child process 21866 terminated and completed 6 tasks
Child process 21868 terminated and completed 17 tasks
Child process 21869 terminated and completed 6 tasks
Child process 21870 terminated and completed 5 tasks
Child process 21867 terminated and completed 6 tasks
All Child processes have completed
Total time spent by all child processes in user mode = 26400.000 ms
Total time spent by all child processes in system mode = 28.000 ms
Total time spent by parent process in user mode = 0.000 ms
Total time spent by parent process in system mode = 12.000 ms
Total elapse time measured by parent process = 5777.566 ms
Draw the image
```

Here is the pseudocode that reflects the logic of the worker:

```
1. print start up message
2. forever
   2.1. wait for SIGUSR1 signal
   2.2. get a task from Task pipe
   2.3. process the task
   2.4. display the computation time
   2.5. for each row of pixels in the task
        2.5.1. write result to Data pipe
```

Here is the pseudocode that reflects the logic of the boss:

```
1. install handlers for SIGUSR1 and SIGINT
2. create two pipes
3. create N workers
4. distribute a task to each worker and send a SIGUSR1 signal to them
5. while not all results have returned
```

```
        5.1.  read in a result message from Data pipe and store the data
        5.2.  check if the result message includes worker id
            5.2.1.  if still have unassigned task
                5.2.1.1.     place a task in Task pipe and send a SIGUSR1 signal to that worker
    6.  inform all workers to terminate by sending SIGINT signals
    7.  collect the termination statuses of all workers and display them
    8.  output the timing statistics and display the 2D image
```

You can name stage B program to part1b-mandelbrot.c. To allow us to play around with a different number of child processes and different number of rows of pixels in each task, stage B program should accept two input arguments; the first argument gives the number of child processes to be created and the second argument gives the number of rows in each task. You can assume the number of child processes is between 1 to 16 and the number of rows in each task is between 1 to 50. The program interface becomes:

```
./part1b-mandelbrot  [number of child processes]  [number of rows in a task]
```

### *Submission*

For stage A program – part1a-mandelbrot.c, you are not required to submit the program; however, if you are not sure whether stage B program is working correctly, you should submit stage A program to get the partial credits of Part One.

For stage B program – part1b-mandelbrot.c, you are required to submit this program to get the full credits of Part One.

Submit the program(s) to the Part One submission page of the course's web site at moodle.hku.hk.

### *Grading Criteria*

| Part 1 (9 points) | | Stage A | Stage B |
|---|---|---|---|
| | Documentation<br>• Include required student's info at the beginning of the program [0.5]<br>• Include necessary documentation to clearly indicate the logic of the program [0.5] | ☑ | ☑ |
| | • The program should be compiled and executed successfully and the total no. of processes involved in the execution should be equaled to the no. of child processes plus one [0.5] | ☑ | ☑ |
| | • Child processes must be executed in parallel [0.5] | ☑ | ☑ |
| | • Child processes must display the computation time spent in handling a task [0.5] | ☑ | ☑ |
| | • Successfully collect all results via the pipe and display the 2D image correctly [1.5] | ☑ | ☑ |
| | • Parent process should wait for all child processes to terminate before displaying all timing statistics [1.0] | ☑ | ☑ |
| | • All processes should terminate successfully and no processes should be left behind in the system [0.5] | ☑ | ☑ |
| | • Working correctly with different number of child processes and different number of rows in a task [1.0] | | ☑ |
| | • Successfully make use of signals to coordinate child processes [1.5] | | ☑ |
| | • Parent process can successfully collect and display the number of tasks performed by each child process [1.0] | | ☑ |

## **<u>Part Two (8 points)</u>**

In this part, we are going to implement the Boss-Workers model by using a single process with multiple threads; each created thread is working as a worker and the main thread is working as the boss. We are going to structure the solution in the form of producer/consumer interaction, such that the main (boss) thread works as the producer that places tasks (each is a block of rows of pixels) to a task pool, while the worker threads work as the consumers that get the tasks from the task pool and process them. To implement the producer/consumer interaction, you are going to make use of mutex locks and condition variables (or semaphores) for synchronization.

Like part one stage B, the program only creates a fixed number of worker threads regardless of the number of tasks. The boss (producer) communicates with all workers (consumers) via a task pool (the bounded-buffer), which is implemented as an array of TASKS in this application. The array has *X* entries; each is for storing a task. The parameter *X* is a runtime argument obtained from one of the command line arguments. You can assume the number of worker threads is between 1 to 16 and the number of rows in each task is between 1 to 50, and the bounded-buffer size *X* is between 1 to 10.

The boss places the tasks one by one to the task pool. If the task pool becomes full, i.e., no empty entries, the boss should be blocked waiting for empty entries in the task pool. This process would continue until all the tasks have been placed to the task pool.

Workers should continuously check whether tasks are available in the task pool. If task pool is empty, workers should be blocked waiting for new tasks to arrive. If there is a task in the task pool, one of the waiting workers will get the task and remove it from the task pool.

We make use of the logic defined in Figure 30.12 (Section 30.2) of our E-textbook as the base for our implementation of this producer/consumer interaction. Here is the pseudocode that reflects the logic of producer/consumer interaction:

```
Producer
  1.  while still have tasks
      1.1.  mutex_lock(pool_lock)
      1.2.  while task pool is full
            1.2.1.  cond_wait(task pool becomes not full)
      1.3.  put a task in the next unused buffer of the task pool
      1.4.  update the free pointer to point to next unused buffer
      1.5.  cond_signal(any waiting consumers that there is a new task)
      1.6.  mutex_unlock(pool_lock)
  2.  inform all worker threads that no more tasks will be assigned and they should terminate after
      finished all pending tasks
  3.  collect the termination statuses of all workers and display them
  4.  output the timing statistics and display the 2D image

Consumer
  1.  while not yet terminated
      1.1.  mutex_lock(pool_lock)
      1.2.  while task pool is empty
            1.2.1.  cond_wait(task pool becomes not empty)
      1.3.  get and remove a task from the next available task from the task pool
      1.4.  update the next pointer to point to next available task
      1.5.  cond_signal(the producer on this newly available buffer)
      1.6.  mutex_unlock(pool_lock)
      1.7.  process the task
      1.8.  display the computation time
      1.9.  return the computation result
      1.10.   is it time to terminate?
```

With this logic, the boss can pass all tasks via the task pool to worker threads to work on the computation. However, we still have to solve two issues: (i) how can worker threads pass the results to the boss? (ii) how can worker threads know that it is time to terminate, i.e., no more new tasks? The main requirement of this part is to implement a multithreaded program using producer/consumer interaction for coordination between the boss and all workers. Thus, you have to use the above producer/consumer logic in distributing tasks to workers. However, you have the flexibility to select your own mechanisms in handling (i) collection of results and (ii) signaling the termination condition.

Same as in part one stage B, each worker prints out the computation time spent in handling a task. The boss collects and prints out the number of tasks handled by each worker as well as the CPU resource usage statistics of itself (including all threads) and prints out the timings. Here is the example printout of the part two program with 5 child processes:

```
Worker(0) : Start up. Wait for task!
Worker(1) : Start up. Wait for task!
Worker(1) : Start the computation ...
Worker(1) :             ... completed. Elapse time = 0.669 ms
Worker(1) : Start the computation ...
Worker(1) :             ... completed. Elapse time = 0.928 ms
Worker(1) : Start the computation ...
Worker(1) :             ... completed. Elapse time = 0.714 ms
Worker(1) : Start the computation ...
Worker(1) :             ... completed. Elapse time = 0.733 ms
Worker(1) : Start the computation ...
Worker(3) : Start up. Wait for task!
Worker(4) : Start up. Wait for task!
Worker(2) : Start up. Wait for task!
Worker(0) : Start the computation ...
Worker(3) : Start the computation ...
Worker(2) : Start the computation ...
Worker(4) : Start the computation ...
Worker(1) :             ... completed. Elapse time = 0.760 ms
Worker(1) : Start the computation ...
Worker(0) :             ... completed. Elapse time = 0.690 ms
Worker(0) : Start the computation ...
Worker(3) :             ... completed. Elapse time = 0.832 ms
Worker(3) : Start the computation ...
Worker(4) :             ... completed. Elapse time = 0.927 ms
Worker(4) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 1.035 ms
Worker(2) : Start the computation ...
Worker(0) :             ... completed. Elapse time = 5.320 ms
Worker(0) : Start the computation ...
Worker(1) :             ... completed. Elapse time = 5.851 ms
Worker(1) : Start the computation ...
Worker(3) :             ... completed. Elapse time = 139.569 ms
Worker(3) : Start the computation ...
Worker(4) :             ... completed. Elapse time = 379.427 ms
Worker(4) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 527.476 ms
Worker(2) : Start the computation ...
Worker(0) :             ... completed. Elapse time = 1340.056 ms
Worker(0) : Start the computation ...
Worker(1) :             ... completed. Elapse time = 1586.350 ms
Worker(1) : Start the computation ...
Worker(3) :             ... completed. Elapse time = 1712.583 ms
Worker(3) : Start the computation ...
Worker(4) :             ... completed. Elapse time = 1991.579 ms
Worker(4) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 2507.898 ms
Worker(2) : Start the computation ...
Worker(0) :             ... completed. Elapse time = 2742.174 ms
Worker(0) : Start the computation ...
Worker(3) :             ... completed. Elapse time = 2447.473 ms
Worker(3) : Start the computation ...
Worker(4) :             ... completed. Elapse time = 2017.753 ms
Worker(4) : Start the computation ...
Worker(1) :             ... completed. Elapse time = 2802.177 ms
Worker(1) : Start the computation ...
Worker(1) :             ... completed. Elapse time = 292.608 ms
Worker(1) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 1723.355 ms
Worker(2) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 1.341 ms
Worker(2) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 2.464 ms
Worker(2) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 0.454 ms
Worker(2) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 0.411 ms
Worker(2) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 0.373 ms
Worker(2) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 0.364 ms
Worker(2) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 0.357 ms
Worker(2) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 0.339 ms
Worker(2) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 0.289 ms
Worker(2) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 0.277 ms
Worker(2) : Start the computation ...
Worker(2) :             ... completed. Elapse time = 0.257 ms
Worker(1) :             ... completed. Elapse time = 109.225 ms
Worker(4) :             ... completed. Elapse time = 436.735 ms
Worker(3) :             ... completed. Elapse time = 1249.937 ms
Worker(0) :             ... completed. Elapse time = 1542.625 ms
Worker thread 0 has terminated and completed 5 tasks.
Worker thread 1 has terminated and completed 10 tasks.
Worker thread 2 has terminated and completed 15 tasks.
Worker thread 3 has terminated and completed 5 tasks.
Worker thread 4 has terminated and completed 5 tasks.
All worker threads have terminated
Total time spent by process and its threads in user mode = 25560.000 ms
Total time spent by process and its threads in system mode = 4.000 ms
Total elapse time measured by the process = 5636.160 ms
Draw the image
```

Name part two program to part2-mandelbrot.c. To control the number of worker threads, the amount of workload in a task, and the size of the task pool, this program needs to accept three arguments. The program interface becomes:

```
./part2-mandelbrot  [number of workers]  [number of rows in a task]  [number of buffers]
```

### *Submission*

Submit the program part2-mandelbrot.c to the Part Two submission page of the course's web site at moodle.hku.hk.

### *Grading Criteria*

| | |
|---|---|
| Part 2 (8 points) | Documentation<br>• Include required student's info at the beginning of the program [0.5]<br>• Include necessary documentation to clearly indicate the logic of the program [0.5] |
| | • The program should be compiled and executed successfully and the total no. of threads created in this program should be equaled to the no. of workers (input parameter) [0.5] |
| | • Must make use of producer/consumer interaction to coordinate the boss and all workers [2.0] |
| | • Worker threads must display the computation time spent in handling a task [0.5] |
| | • Working correctly with different numbers of worker threads, sizes of workload, and sizes of task pool [2.0] |
| | • Master thread can successfully collect and display the number of tasks performed by all worker threads [1.0] |
| | • Master thread should wait for all worker threads to terminate before displaying all timing statistics [1.0] |

## Documentation

1. At the beginning of all submitted programs, state clearly the
    - File name
    - Student's name
    - Student Number
    - Date and version
    - Development platform
    - Compilation – describe how to compile your program
2. Inline comments (try to be detailed so that your code could be understood by others easily)

## Computer platform to use

For this project, you are expected to develop and test your programs under Ubuntu 16.04. Your programs must be written in C and successfully compiled with gcc. It would be nice if you develop and test your program on your own machine (with native Ubuntu (or Linux installation) or the course's VM). After fully tested, upload the program to the department's X2Go server for the final performance tests. Please note that there is a dedicated Linux (virtual) server been allocated to the course for testing of our programs. Please read the briefing note for the information on how to access and use that dedicated server.

## Submission Deadlines

| | **Deadline** | **Submission** |
|---|---|---|
| **Part One** | Nov 1, 2017 (Wed) 5:00 pm | • part1a-mandelbrot.c (optional but recommended)<br>• part1b-mandelbrot.c |
| **Part Two** | Nov 22, 2017 (Wed) 5:00 pm | • part2-mandelbrot.c |

*Late submission policy:* At most 3 days with 10% penalty for each day of delay.

## Plagiarism

Plagiarism is a very serious offence. Students should understand what constitutes plagiarism, the consequences of committing an offence of plagiarism, and how to avoid it. **Please note that we may**

**request you to explain to us how your program is functioning as well as we may also make use of software tools to detect software plagiarism.**