2025.1 Multicore Computing, Project #1

(Due : April 20, 11:59pm)

Submission Rule

1. Create a directory "proj1". In the directory, create two subdirectories, "problem1" and "problem2".
2. In each of the directory "problem1" and "problem2", Insert (i) JAVA source code files, (ii) a document (.pdf format) that reports the parallel performance of your code, and (iii) video file (**.mp4 format**) that shows compilation and execution of your code. You may use your smartphone camera or screen recorder program to generate this video file. **Make sure to include the valid demo video file in your submission. If not, you may get zero or minimal score.**
The document that reports the parallel performance should contain (a) in what environment (e.g. CPU type, number of cores, memory size, OS type ...) the experimentation was performed, (b) **tables and graphs** that show the execution time (unit:milisecond) for the number of entire threads = {1,2,4,6,8,10,12,14,16,32}. (c) The document should also contain **explanation/analysis on the results and why such results are obtained with sufficient details**. (d) The document should also contain screen capture image of program execution and output. In the document (i.e. report), you should briefly explain how to compile and execute the source code you submit. You should use JAVA language.
3. zip the directory "proj1" into "proj1.zip" and submit the zip file into eClass homework board.
※ If possible, please experiment in a PC equipped with a CPU that has 4 or more cores.

problem 1. Following JAVA program (pc_serial.java) computes the number of 'prime numbers' between 1 and 200000 using a single thread.

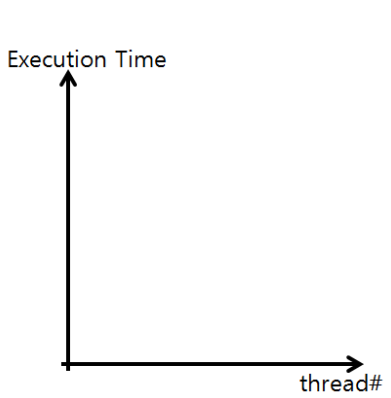
```
public class pc_serial {
  private static int NUM_END = 200000;   // default input
  private static int NUM_THREADS = 1;    // default number of threads
  public static void main (String[] args) {
    if (args.length==2) {
      NUM_THREADS = Integer.parseInt(args[0]);
      NUM_END = Integer.parseInt(args[1]);
    }
    int counter=0;
    int i;
    long startTime = System.currentTimeMillis();
    for (i=0;i<NUM_END;i++) {
      if (isPrime(i)) counter++;
    }
    long endTime = System.currentTimeMillis();
    long timeDiff = endTime - startTime;
    System.out.println("Program Execution Time: " + timeDiff + "ms");
    System.out.println("1..." + (NUM_END-1) + " prime# counter=" + counter);
  }
  private static boolean isPrime(int x) {
    int i;
    if (x<=1) return false;
    for (i=2;i<x;i++) {
      if (x%i == 0)  return false;
    }
    return true;
  }
}
```

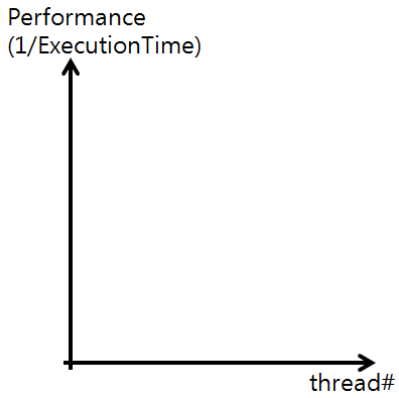
(i) Implement a multithreaded version of **pc_serial.java** using static load balancing based on block decomposition, static load balancing based on cyclic decomposition, and dynamic load balancing. Submit the multithreaded JAVA codes (**"pc_static_block.java"**, **"pc_static_cyclic.java"** and **"pc_dynamic.java"**). Your program should print the (1) execution time of each thread and (2) execution time of the program, and (3) performance (i.e. 1/execution_time) of the program, and (4) the number of 'prime numbers'.

FYI, the static load balancing approach performs work division and task assignment while you do programming, which means your program pre-determines which thread tests which numbers. A static load balancing approach can use a block decomposition method or a cyclic decomposition method.
For example, assuming 4 threads and 200000 numbers, task assignment using
block decomposition method: {0-49999}, {50000-99999}, {100000-149999}, {150000-199999}
cyclic decomposition method (task size: 10 numbers, which means each task-unit has 10 numbers): {1~10, 41~50, 81~90, ...} {11~20, 51~60, 91~100, ...}, {21~30, 61~70, 101~110, ...}, {31~40, 71~80, 111~120, ...}.

The dynamic load balancing approach assigns tasks to threads during execution time. For example, we may let each thread take a number one by one and test whether the number is a prime number or not. (The recommended size for one task is 10, which means each thread processes 10 numbers as one unit of task. However, you may choose another task size if you think it would be better.

(ii) Write a document that reports the parallel performance of your programs. The graphs and tables that show the execution time when using 1, 2, 4, 6, 8, 10, 12, 14, 16, 32 threads. You should include graphs, for static load balancing (block), for static load balancing (cyclic), and for dynamic load balancing. Your document also should mention which CPU type (dualcore? or quadcore?, hyperthreading on or off?, clock speed) was used for executing your code. Your document should also include your interpretation of the parallel results.





exec time	1	2	4	…	32
static (block)					
static (cyclic) [task size : 10 numbers]					
dynamic [task size : 10 numbers]					

performace (1/exec time)	1	2	4	…	32
static (block)					
static (cyclic) [task size : 10 numbers]					
dynamic [task size : 10 numbers]					

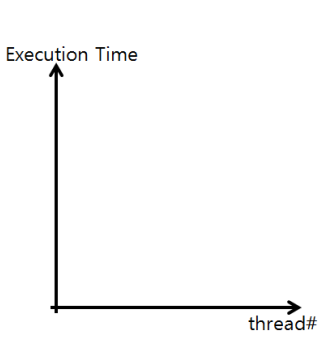
(iii) Create a demo video file (.mp4 format) that shows compilation and execution of your codes. Showing execution using two threads and four threads for each of static(block), static(cyclic), and dynamic cases would be enough for the demo video file. The size of the demo video file should be less than 30MB. (You may use KakaoTalk that automatically reduces the video size when trying to transmit the video.) Make sure to include the valid demo video file in your submission. If not, you may get zero or minimal score.

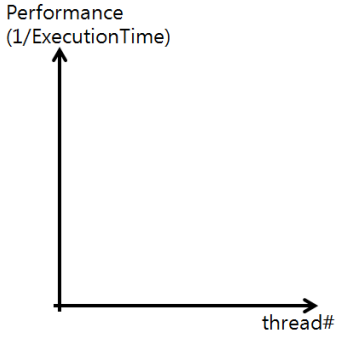
Problem 2. (i) Given a JAVA source code for matrix multiplication (the source code **`MatmultD.java`** is available on our class webpage), modify the JAVA code to implement parallel matrix multiplication that uses multi-threads. You should use a static load balancing approach. You may choose either block decomposition method or cyclic decomposition method. You may also choose the size of each task. Your program should print as output (1) the execution time of each thread, (2) execution time when using all threads, and (3) sum of all elements in the resulting matrix. Use the matrix **`mat1000.txt`** (available on our class webpage) as file input (standard input) for the performance evaluation. **`mat1000.txt`** contains two matrices that will be used for multiplication.

command line execution example in cygwin terminal> j**`ava MatmultD 6 < mat1000.txt`**
In eclipse, set the argument value and file input by using the menu [Run]->[Run Configurations]->{[Arguments], [Common -> Input File].

Here, 6 means the number of threads to use, **`< mat1000.txt`** means the file that contains two matrices is given as standard input.

(ii) Write a document that reports the parallel performance of your code. The graph that shows the execution time when using 1, 2, 4, 6, 8, 10, 12, 14, 16, 32 threads. Your document also should mention decomposition method (block or cyclic?), task size, and CPU type (quadcore?, clock speed) that was used for executing your code.





	1	2	4	…	32
exec time					

	1	2	4	…	32
performace (1/exec time)					

(iii) Create a demo video file (.mp4 format) that shows compilation and execution of your codes (showing execution using two threads and four threads is enough for the demo video file.). The size of the demo video file should be less than 30MB. (You may use KakaoTalk that automatically reduces the video size when trying to transmit the video.) Make sure to include the valid demo video file in your submission. If not, you may get zero or minimal score.