2020.1 Multicore Computing, Project #3

(Due : June 20th)

Although the deadline is June 20th, students are encouraged to finish and submit by May 31th.

Submission Rule

1. Create a directory {studentID#}_proj3 (example: 20123601_proj2). In the directory, create subdirectories 'prob1' and 'prob2'.

2.a For problem 1, write (i)'C with OpenMP' source code prob1.c, (ii)a document that reports the parallel performance of your code, and (iii) readme.txt into the directory "prob1". The document that reports the parallel performance should contain (a) in what environment (e.g. CPU type, memory size, OS type ...) the experimentation was performed, (b) **tables and graphs** that show the execution time (unit:milisecond) for thread number = {1,2,4,8,16}. (c) The document should also contain **explanation on the results and why such results can be obtained**. In **readme.txt** file, you should briefly explain how to compile and execute the source codes you submit. Insert the files (i), (ii), and (iii) into the subdirectory 'prob1'.

2.b For problem 2, write (i) a document, and (ii) ex1.java ~ ex4.java, and insert (i) and (ii) into the subdirectory 'prob2'.

3. zip the directory {studentID#}_proj3 and submit the zip file into eClass homework board.

※ If possible, use quad-core CPU (or CPU with more cores) rather than dual-core CPU for your experimentation, which will better show the performance gains of the parallelism.

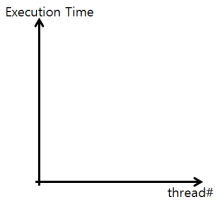
[Problem 1] In our lab. class, we looked at the JAVA program (**ex4.java**) that computes the number of 'prime numbers' between 1 and 200000. The java code (ex4.java) creates threads for parallel computation using static load balancing approach. However, The parallel implementation of **ex4.java** does not give satisfactory performance because of bad load balancing. The problem is that (i) higher ranges have fewer primes and (ii) larger numbers are harder to test. Therefore thread workloads become uneven and hard to predict. For better performance, we implemented dynamic load balancing approach in project 1 where each thread takes a number one by one and test whether the number is a prime number.

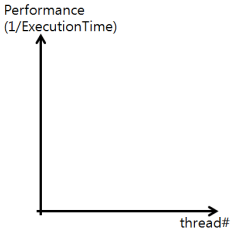
(i) Write 'C with OpenMP' code that computes the number of prime numbers between 1 and 200000. Your program should take two command line arguments: scheduling type number (1=static, 2=dynamic, 3=guided) and number of threads (1, 2, 4, 8, 16) as program input argument. Use **schedule(static) , schedule(dynamic,4) ,** and **schedule(guided,4)**. Your code should print the execution time as well as the number of the prime numbers between 1 and 200000.

command line execution: > **a.out scheduling_type# #_of_thread**

execution example> **a.out 2 8** <---- this means the program use dynamic scheduling using 8 threads.

(ii) Write a document that reports the parallel performance of your code. The graph that shows the execution time when using 1,2,4,8,16 threads. There should be at least three graphs the show the result of static, dynamic, and guided scheduling policy. Your document also should mention which CPU (dualcore? or quadcore?, clock speed) was used for executing your code.





exec time	1	2	4	8	16
static					
dynamic					
guided					

performace (1/exec time)	1	2	4	8	16
static					
dynamic					
guided					

[Problem 2] Visit the website **http://tutorials.jenkov.com/java-util-concurrent/index.html** that introduces and describes JAVA concurrency utilities: **`java.util.concurrent`** package.

1. You are supposed to study and summarize following classes that are important and useful for concurrent programming in JAVA. What you need to do in this problem is to write a document that includes:
(i)-a. Explain the interface/class **`BlockingQueue`** and **`ArrayBlockingQueue`** with your own English sentences. (DO NOT copy&paste)
(i)-b. Create and include (in your document) your example of multithreaded JAVA code (**`ex1.java`**) that is simple and executable. Your example code should use **`put()`** and **`take()`** methods. Also, include example execution results (i.e. output) in your document.
(ii)-a. Do the things similar to (i)-a for the class **`Semaphore`**.
(ii)-b. Do the things similar to (i)-b for **`acquire()`** and **`release()`** methods of Semaphore. (**`ex2.java`**)
(iii)-a. Do the things similar to (i)-a for the class **`ReadWriteLock`**.
(iii)-b. Do the things similar to (i)-b for **`lock()`**, **`unlock()`**, **`readLock()`** and **`writeLock()`** of **`ReadWriteLock`**. (**`ex3.java`**)
(iv)-a. Do the things similar to (i)-a for the class **`AtomicInteger`**.
(iv)-b. Do the things similar to (i)-b for **`get()`**, **`set()`**, **`getAndAdd()`**, and **`addAndGet()`** methods of **`AtomicInteger`**. (**`ex4.java`**)

2. Submit **`ex1.java`**, **`ex2.java`**, **`ex3.java`**, and **`ex4.java`** as well as the document described above.