**CSE-381: Operating Systems**

**Exercise #14**

Max Points: 20

**Note: If you are using your personal machine then prior to commencing work on this exercise, you may need to install XMing, Putty, and WinScp as illustrated in LinuxEnvironment.pdf (and shown in the videos in the Handouts folder).**

|  |
| --- |
| **Objective**: The objective of this exercise is to:   * Develop a data parallel application in Java * Explore the effectiveness of using multiple threads for data parallel applications.   **Submission**: Save this MS-Word document using the naming convention *MUid*\_Exercise14.docx prior to proceeding with this exercise. Upload the following at the end of the lab exercise:   1. This MS-Word document saved with the convention *MUid*\_Exercise14.docx. 2. Program developed in his exercise named with the convention MUid\_Exercise14.java.   You may discuss the questions with your instructor. |

# Preliminaries

1. Log onto the Linux server for this course via the following steps (that were covered in the previous exercises and as illustrated in the [LinuxEnvironment.pdf](https://niihka.muohio.edu/access/content/group/b05b7d30-d8c6-4312-9559-4d980565cbaf/Handouts%20_%20Video%20Tutorials/LinuxEvironment.pdf)):
   1. Run the X-Server Xming.
   2. Use PuTTY to log into the Linux server cse381-f12.csi.muohio.edu.
   3. When you log onto the server, you will be presented with a shell (**$**) prompt. You need to perform various tasks by typing commands at the shell prompt and pressing the enter (↵) key.
   4. Start emacs and ensure you see the graphical screen for emacs.

|  |  |
| --- | --- |
|  | You may develop your Java program using Eclipse on the local Lab PC. However, the runtimes must be measured on cse381-f12.csi.muohio.edu and not on your local box as your local box does not have 4 cores. |

# Part #1: Convert given data parallel C++ program to a data parallel Java program [10 points]

*Estimated time to complete: 45 minutes*

**Background**: Data parallel applications are essentially synchronization free (aka embarrassingly parallel) applications that are easy to parallelize using multiple threads. In a data parallel application, each thread operates on an independent sub-set of data. However, some level of synchronization is needed in data parallel applications to generate final outputs.

**Exercise**: The objective of this exercise is to convert a given data parallel C++ program (developed in an earlier lab exercise) into a corresponding data parallel Java program:

1. Download the supplied raodm\_Ex11.cpp (which is the solution for lab Exercise #11) to the Linux server and briefly review the operation of the program.
2. Using Eclipse (or any other development environment of your choice) develop a Java program that mirrors the operations of the given multi-threaded C++ program, while adhering to the following requirements:
   1. Your Java program should create the same number of threads as the C++ version.
   2. Your Java program should perform the same sequence and set of sequential versus multi-threaded processing.
   3. Note that your Java program should not use the synchronized keyword as it is not required.
   4. Generate random numbers by creating a Random object and using consecutive numbers from it as suggested below (the code fragment below is just to illustrate the use of consecutive numbers as pairs for which GCD is to be computed using multiple threads):

|  |
| --- |
| Random rnd = **new** Random(0);  **for**(**int** i = 0; (i < numPairs); i++) {  **int** num1 = rnd.nextInt(Integer.*MAX\_VALUE*);  **int** num2 = rnd.nextInt(Integer.*MAX\_VALUE*);  *…*  } |

* 1. In order to ease sharing of list of numbers and the range of values that each thread must operate on, it may be easier to suitable define your class as shown below:

|  |
| --- |
| **public** **class** Exercise14 **extends** Thread {  /\*\* A simple class to encapsulate two numbers and their  \* Greatest Common Divisor (GCD)  \*/  **public** **static** **class** ThreeInt {  **…**  }  /\*\*  \* The list of numbers to be processed by multiple threads. The list  \* is static so that it can be accessed by multiple thread classes.  \*/  **private** **static** ArrayList<ThreeInt> *numList* = **new** ArrayList<ThreeInt>();  /\*\*  \* The starting index in the {@link #numList} array from where this thread  \* must process pairs of values.  \*/  **final** **private** **int** startIndex;    /\*\*  \* The number of values in the {@link #numList} array that this thread  \* is supposed to process.  \*/  **final** **private** **int** count;    **public** Exercise14(**int** startIndex, **int** count) {  **this**.startIndex = startIndex;  **this**.count = count;  }  …  } |

* 1. To compile and run your Java program on the Linux server (cse381-f12.csi.muohio.edu), compile and run your program using the following commands:

|  |
| --- |
| $ javac *MUid*\_Exercise14.java  $ java *MUid*\_Exercise14 |

**Sample output**:

Run your program with the same value for numPairs (first command-line argument) but different number of threads (second command-line argument). The output from your program should be consistent immaterial of the number of threads used:

|  |  |  |
| --- | --- | --- |
| **One Thread** |  | **Two Threads** |
| $ ./raodm\_Ex11 12 1  gcd(1569741360, 1785505948) = 4  gcd(516548029, 1302116447) = 1  gcd(1368843515, 663681053) = 1  gcd(1182054491, 251269761) = 1  gcd(1283218719, 1678332854) = 1  gcd(715581077, 542832677) = 1  gcd(827187473, 1316484262) = 1  gcd(2114931095, 2110588844) = 1  gcd(1888030084, 49567875) = 1  gcd(2021317241, 377907320) = 1  gcd(590459143, 672024888) = 1  gcd(276804524, 790221947) = 1 |  | $ ./raodm\_Ex11 12 2  gcd(1569741360, 1785505948) = 4  gcd(516548029, 1302116447) = 1  gcd(1368843515, 663681053) = 1  gcd(1182054491, 251269761) = 1  gcd(1283218719, 1678332854) = 1  gcd(715581077, 542832677) = 1  gcd(827187473, 1316484262) = 1  gcd(2114931095, 2110588844) = 1  gcd(1888030084, 49567875) = 1  gcd(2021317241, 377907320) = 1  gcd(590459143, 672024888) = 1  gcd(276804524, 790221947) = 1 |
|  |  |  |
| **Three Threads** |  | **Four Threads** |
| $ ./raodm\_Ex11 12 3  gcd(1569741360, 1785505948) = 4  gcd(516548029, 1302116447) = 1  gcd(1368843515, 663681053) = 1  gcd(1182054491, 251269761) = 1  gcd(1283218719, 1678332854) = 1  gcd(715581077, 542832677) = 1  gcd(827187473, 1316484262) = 1  gcd(2114931095, 2110588844) = 1  gcd(1888030084, 49567875) = 1  gcd(2021317241, 377907320) = 1  gcd(590459143, 672024888) = 1  gcd(276804524, 790221947) = 1 |  | $ ./raodm\_Ex11 12 4  gcd(1569741360, 1785505948) = 4  gcd(516548029, 1302116447) = 1  gcd(1368843515, 663681053) = 1  gcd(1182054491, 251269761) = 1  gcd(1283218719, 1678332854) = 1  gcd(715581077, 542832677) = 1  gcd(827187473, 1316484262) = 1  gcd(2114931095, 2110588844) = 1  gcd(1888030084, 49567875) = 1  gcd(2021317241, 377907320) = 1  gcd(590459143, 672024888) = 1  gcd(276804524, 790221947) = 1 |

# Part #2: Threading Analysis [10 points]

*Estimated time to complete: 45 minutes*

**Background**: The application being converted to a data parallel implementation is a CPU-bound task – *i.e.*, all of the runtime is spent in performing computations on the CPU. CPU-bound tasks can gain considerable performance improvements when run on a multi-core or multi-CPU machine such as cse381-f12.csi.muohio.edu that is backed by 4 real cores. However, to realize performance benefits a certain amount of computation needs to be performed in order to offset the overheads of multi-threading. The overheads include generation of random numbers, creation of threads, any synchronization overheads, and waiting for threads to finish.

In data parallel applications the synchronization overheads are very minimal, if not totally absent. Consequently, at least in theory, linear performance improvements are to be expected – *i.e.*, if the program takes *t* seconds to run with 1 thread, then ideally it would take only *t*/*n* seconds to run when run using *n* threads. This part of the exercise explores the effectiveness of data parallel implementation by comparing the time taken to run the program with varying number of threads.

Linux provides a time utility to measuring the time taken to run a program that can be used as shown in the sample output below (the actual timings you observe will be different and that is to be expected):

|  |
| --- |
| $ /usr/bin/time java MUid\_Exercise14 1200000 1 > /dev/null  3.80user 0.02system 0:03.83elapsed 99%CPU (0avgtext+0avgdata 51984maxresident)k  0inputs+0outputs (0major+3305minor)pagefaults 0swaps |

Each time a program is run, the actual time taken to run the program will vary depending on the load and other activities occurring on the system. Consequently, on multi-user, multi-tasking systems timing measurements have to be repeated in order to ensure that consistent timings (no more than ±5% different from each other) are obtained and the consistent timings are averaged to obtain a suitable runtime value.

**Exercise**: Using the time command (shown above) measure and record runtime (to determine GCD of 1.2 million randomly generated pairs of numbers) of the data-parallel version of the program in the table below. Ensure you run the program several times in each configuration and record only more-or-less consistent timings in the table. **The %CPU used for multiple threads should be above 100%.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Timing Observations to compute GCD for 1,200,000 randomly generated pairs of numbers** | | | | | | |
| #Threads | Observation #1 | | Observation #2 | | Observation #3 | |
| Elapsed Time (sec) | %CPU | Elapsed Time (sec) | %CPU | Elapsed Time (sec) | %CPU |
| 1 |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |

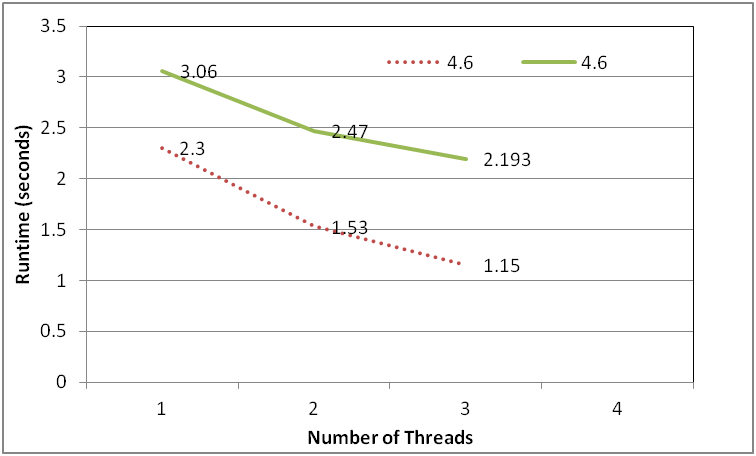
The average (of three consistent runs) runtime using one thread is assumed to the reference value. Compute the reference value by averaging the runtime for one thread recorded in the previous table.

|  |  |
| --- | --- |
| **The reference runtime value is:** |  |
| *(The reference runtime is the average runtime measured when using a single thread)* |  |

Using the timing data in the above table, compute and enter the average elapsed time values in the following table:

|  |  |  |
| --- | --- | --- |
| **Average (of 3 runs) Runtime (seconds)** | | |
| # Threads | Expected Theoretical Runtime  (seconds) | Observed Average Runtime  (seconds) |
| 1 | <*reference runtime*> | <*reference runtime*> |
| 2 | <*reference runtime*> / 2 | <average computed from previous table> |
| 3 | <*reference runtime*> / 3 | <average computed from previous table> |
| 4 | <*reference runtime*> / 4 | <average computed from previous table> |

Using the above data plot a chart (using Excel) with number of threads on X-axis and runtime on Y-axis similar to the example chart below. Copy-paste the chart you have generated replacing the chart shown below (the chart shown below is fictitious and your chart look different. **However, ensure you include axis titles, legend, and data-point labels in your chart!**).



Using the above chart answer the following question:

1. Are the observed average runtime and expected theoretical runtime the same? Do they follow a similar trend? What could plausibly explain any difference between the theoretical and observed behaviors of your program?

|  |
| --- |
|  |

# Part 3: Submit files to Niihka

Upload just the following files to Nihhka:

1. This MS-Word document saved with the convention *MUid*\_Exercise14.docx.
2. Program developed in Part #1 named with the convention MUid\_Exercise14.java.