Assignment #2 - Threading

Introduction

This assignment aims to consolidate your understanding of several fundamental topics taught in the Threading theme: threads, tasks, thread pools, asynchronous tasks, fork-join pools, computational tasks.

Deliverables

Return a folder named after your **upi**, containing two **command-line** applications:

I. A computationally intensive application, **SOD**, given by its Java source, **SOD.java** and its corresponding batch file (to compile and run it), **SOD.bat**

II. An asynchronous Web application, given by its Java source, **WEB.java** and its corresponding batch file (to compile and run it), **WEB.bat**

This folder and all its contents must be packaged in a **upi.zip** archive.

I. The SOD (SumOfDivisors) application

1. First, the SOD application starts by reading an input text file given as a command-line parameter. For example:

```
java  SOD  Matrix.txt
```

The input file contains N lines, where N is in the range 1..3000. Each line i contains Mi integer numbers, in the range 1..3000, separated by spaces, where Mi is in the range 1..10000. All given ranges include the limits. For example (N=3, M1=6, M2=6, M3=3):

```
1 2 4 6 15 20
25 50 100 125 250 500
16 8 3
```

2. Next, the SOD application repeats the following intensive computation in two modes: **sequentially** and **parallel**.

2.1. SOD computes the **sum of divisors** of each input number (excluding the number). For example, number 20 has the sum 1+2+4+5+10=22 and our example numbers have the following sums:

```
1 1 3 6 9 22
6 43 117 31 218 592
15 7 1
```

2.2. These sums are summed up line-by-line. For our example:

```
42
1007
23
```

2.3. For each of the above sums, SOD computes its sum of divisors. For our example:

```
54
73
1
```

2.4. Finally, SOD sums up these numbers. For our example:

```
128
```

As mentioned, your application will repeat these two procedures twice: first in **sequential** mode and secondly in **parallel** mode. In the **parallel** mode, each input line is processed by its own parallel *task*.

In both cases, it must use the same method for computing the **sum of divisors** (excluding the number itself). You can use a *naive* computation, because here we are only interested in the relative performance, not the absolute: the parallel evaluation should scale according to the number of processing cores.

The runtime of each computation mode must be computed up to milliseconds and displayed according to the following sample format (for our example):

```
[ 1] SeqSum lines=3
[ 1] SeqSum res=128 secs=0.000


[ 1] ParSum lines =3 procs=2
[ 1] ParSum threads=[8, 9, ]
[ 1] ParSum res=128 secs=0.005


... press <enter> to exit
```

SeqSum indicates the **sequential** mode and **ParSum** the **parallel** mode. The first number between brackets, **[1]**, indicates the **thread id** which prints that line. Then, **lines=** indicates the number of lines in the given input file, **res=** indicates the final result and **secs=** the number of seconds (with 3 decimals).

Only in the **parallel** mode, **procs=** is the number of cores on the machine used and **threads=** lists, *without repetitions*, the **thread ids** of the threads allocated to solve the problem.

In our example, thread **[8]** executed the first task (processing the first line), then thread **[8]** continued with the second task (processing the second line), and thread **[9]** executed the last task (processing the last line). By discarding repetitions, the thread ids list, **[8, 8, 9]**, is mapped to a set, **[8, 9]**, which appears in the sample output.

Our example file contains a few numbers only. In this case, the runtime of the **parallel** mode can be larger than the sequential mode, because of the parallel *overhead*.

However, if the application is properly implemented, large sets of numbers show a more characteristic performance **scaling**, e.g.:

```
[ 1] SeqSum lines=3000
[ 1] SeqSum res=28089648000 secs=21.725

[ 1] ParSum lines=3000 procs=2
[ 1] ParSum threads=[8, 9, 10, 11, ]
[ 1] ParSum res=28089648000 secs=15.707

... press <enter> to exit
```

Please take care to time only the actual computation, *excluding* the time required to print the output.

Note that thread ids and runtimes are execution specific, but results (**res=**) are deterministic.

II. The WEB application

1. First, the WEB application starts by reading an input text file given as a command-line parameter. For example:

```
java  WEB  Urls.txt
```

The input file contains N lines, where N is in the range 1..10 (including the limits). Each line contains a Web URL, accessible over HTTP, and a response line number, starting with 0, separated by spaces. For example (N=3):

```
http://www.cs.auckland.ac.nz/uoa        6
http://www.science.auckland.ac.nz/uoa 6
http://www.auckland.ac.nz/uoa     6
```

2. Next, the WEB application repeats the following operations in two modes: **sequentially** and **parallel**.

2.1. WEB makes an HTTP GET request for each the given URLs. After receiving a response, WEB the indicated response line (line number 6, in our example) and the last response line.

2.2. In the **sequential** mode, all operations are executed by the same thread, requests are started in the given URL order, a new request is started after the completion of the previous.

2.3. In the **parallel** mode, each request/response operation is independently executed by its own thread. Note that, because of thread interleaving, requests and responses do not appear in any predictable order (except that, of course, a request arrives after the corresponding request). This ensures parallel processing, but could waste thread resources in real applications. However, here we accept this simple solution.

WEB's output format is somehow similar to SOD's output format. For example:

```
[ 1]  SeqFetch urls=3

[ 1]  0       : http://www.cs.auckland.ac.nz/uoa
[ 1]  0.6     :    Department of Computer Science - ...</title>
[ 1]  0.673   : </html>
[ 1]  1       : http://www.science.auckland.ac.nz/uoa
[ 1]  1.6     :    Faculty of Science - The University of Auckland</title>
[ 1]  1.643   : </html>
[ 1]  2       : http://www.auckland.ac.nz/uoa
[ 1]  2.6     :    Welcome to The University of Auckland - ...</title>
[ 1]  2.1007  : </html>

[ 1]  SeqFetch secs=1.224

[ 1]  ParFetch urls=3 procs=2

[ 9]  0       : http://www.cs.auckland.ac.nz/uoa
[10]  1       : http://www.science.auckland.ac.nz/uoa
[11]  2       : http://www.auckland.ac.nz/uoa
[10]  1.6     :    Faculty of Science - The University of Auckland</title>
[11]  2.6     :    Welcome to The University of Auckland - ...</title>
[11]  2.1007  : </html>
[10]  1.643   : </html>
[ 9]  0.6     :    Department of Computer Science - ...</title>
[ 9]  0.673   : </html>

[ 1]  ParFetch secs=0.513

[ 1]  ... press <enter> to exit
```

SeqFetch indicates the **sequential** mode and **ParFetch** the **parallel** mode. Then, **urls=** indicates the number of input lines (URLs), **secs=** the number of seconds (with 3 decimals) and, in **parallel** mode, **procs=** is the number of processing cores.

All **sequential** processing is done by the same thread, the main thread, **[1]**. Next:

- Index 0 indicates the first URL, "http://www.cs.auckland.ac.nz/uoa".
- Indices 0.6 indicate response line 6 for first URL, ": Department of Computer Science - ...</title>".
- Indices 0.673 indicate last response line for first URL, "</html>".
- ...
- Finally, the **sequential** mode took a total of 1.224 seconds.

In **parallel** mode, each request/response is processed by its own thread. For example:

- Thread **[9]** takes care of the first URL, http://www.cs.auckland.ac.nz/uoa.
- Thread **[10]** takes care of the second URL, http://www.science.auckland.ac.nz/uoa.
- Thread **[11]** takes care of the third URL, http://www.auckland.ac.nz/uoa.
- ...
- Note that results arrived in an unpredictable order.
- Finally, the **parallel** mode took a total of 0.513 seconds.

Submission

Submit electronically, to the COMPSCI web dropbox, a **upi.zip** archive containing a **upi** folder with all your source files, **WEB.java**, **WEB.bat**, **SOD.java**, **SOD.bat**. Please keep your receipt and follow the instructions given in our Assignments web page (please read these carefully, including our policy on plagiarism):

http://www.cs.auckland.ac.nz/courses/compsci230s1c/assignments/

Deadline

Saturday 19 May, 2012, 17:00. Please do not leave it for the last minute. Remember that you can resubmit and, by default, we only consider your last submission.

Late submission will incur penalties, 10% off for each day late, for up to three days.

Change summary for v2

Added appendices: A, B, C, D, E

Change summary for v3

Updated the limits for SOD (important!) and corrected Appendix C, Example 2. All changes are marked in red fonts and highlighted.

Added appendices: F (simplifying restrictions), G (rough marking schedule), H (FAQ)

Appendix A

Reading a text file into an ArrayList of Strings

```
List<String> Lines = new ArrayList<String>();

BufferedReader in =
        new BufferedReader(
                new FileReader(filename));

for (;;) {
        String inLine = in.readLine();
        if (inLine == null) break;
        //System.out.format("%s %n", inLine);
        Lines.add(inLine);
}

in.close();
```

Appendix B

Reading a web page into an ArrayList of Strings

```
List<String> Lines = new ArrayList<String>();

URLConnection http = new URL(url).openConnection();

BufferedReader in =
        new BufferedReader(
                new InputStreamReader(
                        http.getInputStream()));

for (;;) {
        String inLine = in.readLine();
        if (inLine == null) break;
        //System.out.format("%s %n", inLine);
        Lines.add(inLine);
}

in.close();
```

Appendix C

Computing the sum of divisors (suggestion)

Sketch of a simple, but less naive, algorithm

Start with Sum=1

1. Example for n=20

Test	20%2==0?	20%3==0?	20%4==0?
Sum +=	2, 10		4, 5

Result: sum=22

2. Example for n=36

Test	36%2==0?	36%3==0?	36%4==0?	36%5==0?	36%6==0?
Sum +=	2, 18	3, 12	4, 9		6

Result: sum=55

Appendix D

Sketch of SOD solution (exception processing is required but omitted here)

1. Assert args.length==1
2. Filename = args[0]
3. Read input file (filename) into an array of strings
4. **Sequential solution**
   - 4.1. Start stopwatch
   - 4.2. For each input line
     - 4.2.1. Parse the current input line into integers
     - 4.2.2. For each integer in the current input line
       - 4.2.2.1. Compute the sum-of-divisors of this integer
       - 4.2.2.2. Add this to the cumulated sum-of-divisors
     - 4.2.3. Compute the sum-of-divisors of this cumulated sum
     - 4.2.4. Add this to the grand total
   - 4.3. Stop stopwatch
   - 4.4. Print result summary
5. **Parallel solution**
   - 5.1. Start stopwatch
   - 5.2. For each input line
     - 5.2.1. Parse the current input line into integers
     - 5.2.2. For each integer in the current input line
       - 5.2.2.1. Compute the sum-of-divisors of this integer
       - 5.2.2.2. Add this to the cumulated sum-of-divisors
     - 5.2.3. Compute the sum-of-divisors of this cumulated sum
     - 5.2.4. Add this to the grand total
   - 5.3. Stop stopwatch
   - 5.4. Print result summary

Conceptually, the **parallel solution** differs from the sequential solution by running the main part of loop 5.2, i.e. lines 5.2.1, 5.2.2, 5.2.2.1, 5.2.2.2 and 5.2.3, as a parallel task.

Appendix E

Sketch of WEB solution (exception processing is required but omitted here)

1. Assert args.length==1
2. Filename = args[0]
3. Read input file (filename) into an array of strings
4. **Sequential solution**
   - 4.1. Start stopwatch
   - 4.2. For each input line
     - 4.2.1. Parse the current input line into an url and an integer
     - 4.2.2. Start an HTTP web request to the given url
     - 4.2.3. Print the required lines of the web response
   - 4.3. Stop stopwatch
   - 4.4. Print result summary
5. **Parallel solution**
   - 5.1. Start stopwatch
   - 5.2. For each input line
     - 5.2.1. Parse the current input line into an url and an integer
     - 5.2.2. Start an HTTP web request to the given url
     - 5.2.3. Print the required lines of the web response
   - 5.3. Stop stopwatch
   - 5.4. Print result summary

Conceptually, the **parallel solution** differs from the sequential solution by running the loop 5.2, i.e. lines 5.2.1, 5.2.2 and 5.2.3, as a parallel task.

Appendix F

New limits for SOD, N and Mi

The input file contains N lines, where N is in the range 1..3000. Each line i contains Mi integer numbers, in the range 1..3000, separated by spaces, where Mi is in the range 1..10000. All given ranges include the limits.

These restrictions bring two benefits:

1. The largest test files are smaller and more manageable.
2. All sum-of-divisors and *intermediate* sums can be calculated using **int** types

There is only one computation for which the **int** type is not enough: the *grand total* res sum, which needs **long** types (to avoid arithmetic wraparound).

For example, one of our large tests, SOD_3000_3000_1000.txt (the actual test used to create the output mentioned on page 4), has the final result 28089648000, which is greater than the maximum **int** number, Int32.MaxValue=2147483647.

Appendix G

Rough marking schedule

1. [50] SOD
   1.1. [10] …
   1.2. [20] Sequential
      1.2.1. [10] Correct result (with required details)
      1.2.2. [5] Acceptable performance (at most 1.5 the demo runtime)
      1.2.3. [5] Code inspection (clarity, brevity, efficiency)
   1.3. [20] Parallel
      1.3.1. [10] Correct result (with required details)
      1.3.2. [5] Acceptable performance (at most 1.5 the demo runtime)
      1.3.3. [5] Code inspection (clarity, brevity, efficiency)

   SOD testing will use files similar to the ones accompanying the demo.

2. [50] WEB
   2.1. [10] …
   2.2. [20] Sequential
      2.2.1. [10] Correct result (with required details)
      2.2.2. [5] Acceptable performance (at most 1.5 the demo runtime)
      2.2.3. [5] Code inspection (clarity, brevity)
   2.3. [20] Parallel
      2.3.1. [10] Correct result (with required details)
      2.3.2. [5] Acceptable performance (at most 1.5 the demo runtime)
      2.3.3. [5] Code inspection (clarity, brevity)

   WEB testing will use files similar to the ones accompanying the demo, but with standardized web services (where we can fully control the page contents and serving delays).

Appendix H

FAQ

- Q: In part 1 of the assignment, it states that the thread IDs used must be printed without repetition; so if thread 8 executed the first 2 tasks, then 9 completed the third, the set would be [8,9]. What if thread 8 completed the first process, then 9 the second, 8 the third and 9 the fourth? Would this set be [8,9] or would it be [8,9,8,9]?
  - A: The set must be listed as a set, i.e., either [8,9] or [9,8]. The order is not relevant and actual thread id numbers may differ from run to run.

- Q: Does the reading of textfile have to be done sequentially and parallelly? For my assignment I basically read the input textfile and converted it into a 2D array and then start the sequential and parallel mode.
  - A: This is correct, but not optimal. Reading lines must be sequential, coz there is no other way. However, anything else, starting by converting already read lines into number sequences, can be done in the selected mode, sequential or parallel. BTW, you do not really need any 2D array of numbers…

- Q: How many threads and tasks should I use in the parallel versions:
  - A:
    - Suggestions for SOD:
      - Threads = number of available processors (or up to double this number, not more)
      - Tasks = number of lines (each line is a task)
      - Avoid explicit threads and submit all your tasks to a ForkJoinPool
    - Suggestions for WEB:
      - Threads = tasks = number of lines (each line is a task)
      - Avoid explicit threads and submit all your tasks to a thread pool (e.g., newFixedThreadPool). This is “good enough” in our case (we only have up to 10 urls), but, please be warned, won’t be optimal for concurrent downloading from a substantially larger number of urls.

- Q: How to determine which thread read which lines …?
  - A: Everything runs in a thread, even if this is not explicit. You can always locate the Thread instance reifying your current thread, by calling the Thread API used in our very first Runnable example, please see slide 21 of 09 - Threading Intro.

- Q: How to collect these thread numbers for SOD, in the parallel case?
  - A: There are several options
    - You can add them to a collection, while the threads are running. However, in this case, please take care to use a thread-safe collection. The classical collections are NOT thread safe or are inefficient when used concurrently. Please recall our counter and bank account examples -- you risk similar issues if you do not use a thread-safe collection. Collections in java.util.concurrent are thread-safe.
    - Alternatively, return the thread ids from your Callable tasks, together with their line results. Instead of returning just one integer, each Callable can return a complex object, containing both the integer result and the thread id of thread that was automatically allocated. After all Callable tasks end, you can separate the integer numbers that must be further summed and the thread ids.
    - In both cases, the collected thread ids must be filtered by removing duplicates. There are simple API's for doing this (even in one line).

- Q. My SOD program returns the correct results, but takes longer (or much longer) than the demo. Will I be penalised?
  - A: Possibly, please check the draft marking schedule.

- Q: My WEB program seems to work well, but, compared to the demo and specs, the line numbers are off by one…
  - A: Please start numbering lines from 0, as we use array indices.

- Q: My WEB program seems to work well, but, compared to the specs, the last line numbers are (very) different
  - A: This is possible, as the sample test pages change frequently. For marking, we'll set up a test web service with controllable content and delays…