Anting Around
UCT CSC2002S Java Multithreading Module Assignment
Second Semester 2012
Due Date: Friday 21st September 2012 by 9am

Introduction

This project will give you practice using some of the various methods for multithreading in Java.
In this project, you will:

- Write software that processes data in several stages.
- Implement divide-and-conquer parallel algorithms using fork-join parallelism.
- Experience the difficulty of evaluating parallel programs experimentally.
- Write a clear, detailed report to present the interesting parameters of your implementation and how you evaluated them.

Problem description: *Where are the ants?*

For this project, we assume that researchers at UCT have developed a tiny wireless GPS– so small that it can be attached to an insect and will not interfere with its daily movements. The Zoology Department has great interest in this device and has attached a number of them to ants in a local colony of harvester ants on Table Mountain. You are now asked to write a program to process the resulting large files of ant location data and answer queries such as:

- How many times did ants visit a specific area inside the colony's territory?
- What percentage of the total monitoring time did ants spend in this area?

Such questions can reveal where attractants (food sources) and repellants are located and how actively ants of different types (workers, soldiers) explore the surrounding territory. By supporting only rectangles as queries, we can answer queries more quickly. A different shape can be approximated using multiple rectangles, but this will give you "Additional Credit."

As a preprocess, your program will first read in the files of the ants' movements and count the number of times ants visited each particular section of the gridded area to find the range of territory explored by the ants (some versions of the program will then further preprocess the data to build a data structure that can more efficiently answer the queries described above).

The program will then prompt the user for such queries and answer them, until the user chooses to quit. It is this query processing that you will focus your parallelization effort on. For testing and timing purposes, you may also wish to provide an alternative version where queries are read from a second file.

You will implement the program in several ways that vary in their simplicity and efficiency and compare your approaches. Some of the ways will require fork-join parallelism and, in particular, Java's ForkJoin Framework. Others are entirely sequential.

A final portion of this project involves comparing execution times for different approaches and parameter settings. You will want to write scripts to collect timing data for you, and you will want to use a machine that has at least 4 processors.

This project is an experiment where much of the coding details and experimentation are left to you, though we will describe the algorithms you must use. Will parallelism help or hurt? Does it matter given that most of your code runs only in a pre-processing step? The answers may or may not surprise you, but you should learn about parallelism along the way.

Input

Your application should take a parameter file as input, with data lines comprising:

- The number, *n*, of data files to be summarized.
- A list of *n* files containing the data on the travels of individual ants.
- Two integer values, *k* and m, indicating the bin dimensions.

```
<n >
<ant data file name 1 >
<ant  data file name 2 >
```

```
…
<ant data fine name n >
<k> <m>
```

Each ant data file will contain a (potentially very long) record (or time series) of the travels of a particular ant. This is a list of numbers, where each line has the format:

`<counter> <x> <y> <optional other data – ignore>`

Here, the `<counter>` keeps track of the number of readings recorded – it will be sequential, but may count in increments greater than 1 and can become very large. The coordinate values `<x>` and `<y>` represent the location of a particular ant at a particular time. Note that these can take on any value, including negative ones, as the grid 0,0, position is centered on an arbitrary location.

For rapid query processing, it will help to divide the 2D space into blocks and calculate a 2D histogram of the x,y of the location data. The *l* and *m* bin dimensions represent the size of the *divisions* in the x and y dimensions for the histogram grid (the division size will affect the accuracy of your program's output). For example, if k= 2 units and m = 5 units, the grid will consist of rectangular blocks 2x5 square units in size.
Note that you are not given the x- and y-ranges for the data files: you will have to determine these from scans of the file(s), and then divide the ranges into the correct number of bins.

Examples of this input file and location data files are provided for testing.

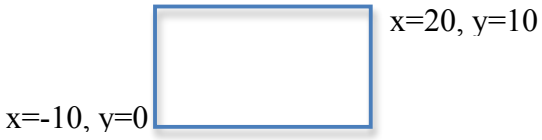
Queries
A query describes a rectangle within the area travelled by an insect. It is simply four numbers, representing the "bottom-left" and "top-right" corners of the rectangle to be queried:

`X_min Y_min X_max Y_max`

For example, the user could type:

`-10 0 20 10`

This would represent the following rectangle:



Your program should print a single one-line prompt asking for these four numbers. Any illegal input (i.e., only 2 integers on one line) indicates the user is done and the program should end. Invalid queries (i.e. X_max < X_min) should result in an error message and reprompt. Otherwise, you should output two numbers:

- The total number of data points in the queried rectangle
- The percentage of time the ant (or ants) spent in the queried rectangle, rounded to two decimal digits, e.g. 10.22%

To implement your program, you will need to determine how many data points lie within the specified query rectangle.
You should then repeat the prompt for another query.
Note that it is possible for the user to specify a query rectangle that is out of the range of the data files – in this case, both results will be zero.

For Example

Input file data:

```
1
antfile.txt
1 1
```

antfile.txt contains:

```
1 2 3
2 4 1
3 2 2
4 1 1
5 2 1
6 1 1
7 2 1
8 2 2
9 2 3
10 3 3
```

This could be processed to form the following histogram, with bin size of 1x1 (the bottom and left row and column are the x and y dimension labels respectively):

3	0	2	1	0
2	0	2	0	0
1	2	2	0	1
	1	***2***	***3***	***4***

For the query

```
2 2 4 3
```

the result would be

```
5 50%
```

Parallel Implementations

You will implement four versions of your program.

Version 1: Simple and Sequential

Before processing any queries, process the data to find the four corners of the data range (we will call this corner-finding) and then bin the data into histograms. Then, for each query, do a sequential traversal of the histogram to answer the query (determining for each bin whether it is in the query rectangle).

Version 2: Simple and Parallel

This version is the same as version 1, *except* that the traversal for each query should use the ForkJoin Framework effectively. The span for the query should lower to O(log n).

Version 3: Smarter and Sequential

This version will, like version 1, not use any parallelism, but it will perform additional preprocessing so that each query can be answered in O(1) time. This involves two additional steps:

Modify the histogram so that, instead of each bin holding the total for that position, it holds the total for all positions that are neither farther "right" nor farther "down". In other words, bin *g* stores the total population in the rectangle whose upper-left is the top left corner of the grid and the lower-right corner is *g*. This can be done in time O(x*y), but you need to be careful about the order you process the elements. Keep reading….

.

For example, suppose after step 1 we have this grid:

```
0   11   1   9
1   7    4   3
2   2    0   0
9   1    1   1
```

Then step 2 would update the grid to be:

```
0   11   12   21
```

```
1   19  24  36
3   23  28  40
12  33  39  52
```

There is an arithmetic trick to completing the second step in a single pass over the grid. Suppose our grid positions are labeled starting from (1,1) in the bottom-left corner. (You can implement it differently, but this is how queries are given.) So our grid is:

```
(1,4)  (2,4)  (3,4)  (4,4)
(1,3)  (2,3)  (3,3)  (4,3)
(1,2)  (2,2)  (3,2)  (4,2)
(1,1)  (2,1)  (3,1)  (4,1)
```

Now, using standard Java array notation, notice that after step 2, for any element not on the left or top edge: `grid[i][j]=orig+grid[i-1][j]+grid[i][j+1]-grid[i-1][j+1]` where `orig` is `grid[i][j]` after step 1. So you can do all of step 2 in O(x*y) by simply proceeding one row at a time top to bottom -- or one column at a time from left to right, or any number of other ways. The key is that you update (i-1 , j), (i , j+1) and (i-1 , j+1) before (i , j).

Given this unusual grid, we can use a similar trick to answer queries in O(1) time. Remember a query gives us the corners of the query rectangle. In our example above, suppose the query rectangle has corners (3,3), (4,3), (3,2), and (4,2). The initial grid would give us the answer 7, but we would have to do work proportional to the size of the query rectangle (small in this case, potentially large in general). After the second step, we can instead get 7 as 40 - 21 - 23 + 11. In general, the trick is to:

- Take the value in the bottom-right corner of the query rectangle.
- Subtract the value just above the top-right corner of the query rectangle (or 0 if that is outside the grid).
- Subtract the value just left of the bottom-left corner of the query rectangle (or 0 if that is outside the grid).
- Add the value just above *and* to the left of the upper-left corner of the query rectangle (or 0 if that is outside the grid).

Notice this is O(1) work. Draw a picture or two to convince yourself this works.

Note: A simpler approach to answering queries in O(1) time would be to pre-compute the answer to every possible query. But that

would take $O(x^2y^2)$ space and pre-processing time.

Version 4: A) ForkJoin for faster Parallel Initial Binning OR B) Parallel Prefix for faster queries

There are two options for this version: you can either try to a) parallelize the initial binning procedure or else b) try to speedup queries with parallel prefix.

A) The initial binning procedure may not benefit for parallelism because the files can be too big to fit into memory. However, you may experiment with using independent threads to process different files, or the same file.
To parallelize the first binning step, you will need each parallel subproblem to return a grid. To combine the results from two subproblems, you will need to add the contents of one grid to the other. The grids may be small enough that doing this sequentially is okay, but for larger grids you will want to parallelize this as well using another ForkJoin computation. (To test that this works correctly, you may need to set a sequential-cutoff lower than your final setting.)

Note that your ForkJoin tasks will need several values that are the same for all tasks: the input array, the grid size, and the overall corners. Rather than passing many unchanging arguments in every constructor call, it is cleaner and probably faster to pass an object that has fields for all these unchanging values. You will have to experiment with and carefully document how your approach affects runtime (it may be worse!).

B) For parallel prefix, as in version 3, you should create the grid that allows O(1) queries. In version 3, the grid-building is still entirely sequential, running in time O(x*y). We can use parallel prefix computations to improve this -- the most straightforward approach involves two different parallel-prefix computations where the second uses the result of the first. Implement this so that the span of for this grid-building step is O(log x + log y). Run experiments to determine the effect of this change on running time.

Additional Work

Extra credit will be given for additional work (or doing both parts of version 4). If you have time, you may implement:.

Polygonal queries: You can choose to support queries that are arbitrary polygons instead of rectangles. For a polygon, the user can enter any set of grid positions and the polygon should be the shape that connects these points in order (connecting the last back to the first). Reject a query in which any lines cross each other. Then answer the query by transforming it into as few rectangular queries as possible.

Explicit threads: You can choose to investigate whether explicit threads and locks will give a performance benefit over the Java fork-join framework, particularly for Part 4A.

Report

Your project will be marked primarily on the basis of your report, which should be a concise, clear account of the work you have done.

Your report must document the performance (running time) of the various implementations with different parameter settings. You should compare the running time on different numbers and sizes of files, different bin sizes and with different numbers of threads and report your results as graphs. Note that, as you may not have the time or resources to experiment with every combination of every parameter, you will need to choose wisely to reach appropriate conclusions in an effective way.

You are required to run the code on a machine with at least four processors – the machine's characteristics should be documented in your report. (You will have a blade server with 4 cores made available for this practical, but are welcome to test on other machines that you have access to.)

You should write extra code to perform the experiments, but this code should not interfere with your working program. You do *not* want to enter queries manually in order to collect timing data.
To ensure that your timing values make sense for programs run many times or on big data files, your timing experiments should:

- Separate out the time require to read the initial files

- Time the actual code answering a query, not including the time for a (very slow) human to enter the query.
- Allow the Java Virtual Machine and the ForkJoin Framework to "warm up" before taking measurements. The easiest way to do this is to put the initial processing in a loop and not count the first few loop iterations, which are likely to be slower.

You must graph the results, discuss them in the text and reach appropriate conclusions.

A good report is essential for a good mark. An example of a model report will be provided. Aside from the requirements listed above, your report should answer the following questions:

What assistance did you receive on this project? Include anyone or anything *except* the course staff and the course materials / textbook.

How long did the project take? Which parts were most difficult? How could the project be better?

Were there any safety and liveness issues to be dealt with in your program? Which synchronization constructs – e.g. join, barriers, latches, atomic variables – did you use and why were they necessary?

How did you dealt with handling multiple files?

How did you validate that your program works correctly?

Which "Additional Work" aspects did you implement? What was interesting or difficult about them? Describe how you implemented them.

Did you try any alternative parallelization approaches? If so, what did you do and why?

How did you test your program? Which parts did you test in isolation and how? Which inputs did you create so that you could check your answers? What boundary cases did you consider? What were the results?

How does varying the sequential cut-off of the ForkJoin Framework affect the performance of code using this toolkit? What is the optimal value for this cutoff? Note that if the sequential cut-off is high enough to eliminate all parallelism, then you should see performance close to the sequential algorithms, but evaluate this claim empirically.

How many queries are necessary before the pre-processing is worth it?

How does the performance of version 4 vary as the resolution of the grid changes?

Handin procedure and late penalties

Hand in an archive of your report (pdf) format and Java source code on the Vula course site.

Handin is due on the 21st September 2012, at 9am.

There will be a 5% penalty for submission **after** 9am on 21st September, but **before** 9am on Tuesday 25th September. Thereafter, 10% will be subtracted per day, or part thereof.

Note that plagiarism will receive a mark of zero and all offending documents will be submitted to the university court for trial.

Acknowledgments

This project was adapted from Dan Grossman's Java Parallel-Programming Project – "Where are the People?" (http://www.cs.washington.edu/homes/djg/teachingMaterials/spac/grossmanSPAC_project.html - last accessed 24 August 2012)