GHRAHM004 – Ahmed Ghoor – Assignment 1 - CSC2002S

**Introduction**

**-Aim of project**

To compare the Java Fork/Join parallelization framework to a standard sequential Algorithm.

This will be done by creating and analyzing 2 Algorithms (one parallel and one sequential) to find basins in a terrain, so that we may answer the following questions:

* + Is it worth using parallelization (multithreading) to tackle this problem in Java?
  + For what range of data set sizes does your parallel program perform well?
  + What is the maximum speedup obtainable with your parallel approach? How close is this speedup to the ideal expected?
  + What is an optimal sequential cutoff for this problem? (Note that the optimal sequential cutoff can vary based on dataset size.)
  + What is the optimal number of threads on your architecture?

**-Parallel Algorithm**

Have a look at this example of a really small terrain below:



After storing these blocks in an array, a sequential solution would go to each index one by one in order from start to end and carry out a check to see if the respective block is a basin or not. A block is classified as a basin if the 9 blocks around it are each more than or equal to 0.01 units larger. The block at index [2,1] with a depth of 0.6 is an example

The parallel divide-and-conquer solution is similar, except that it would attempt to check multiple blocks at the same time.

It would do this by splitting the array multiple times into smaller sub-grids until a given cutoff, before computing them. Each piece of the grid will be in its own thread, and so it is able to be processed whenever a processor becomes available, without having to wait for the blocks before it to be processed. Since a computer often has more than one processor, multiple threads can be processed at the same time.

Note that the array isn’t broken up into smaller arrays. Rather, there is one array that is split by assigning different start and end points for each thread to classify. Therefore, every block still has access to all the blocks around it.

So, in this example, the grid could be split into two threads. One thread could process the first 2 rows (1.0 to 0.8) and the other thread could process the last two rows (0.85 to 1.0).

**-Expected Speedup/Performance:**

According to Amdahl’s Law: Speed up = T\_1/T\_P = 1/(S+(1-S)/P)

On current Machine Architecture: P=4

S = portion of exe. time that cannot be parallelized = 1/number of splits(or threads created)

**Methods (Approach to solutions)**

**-Sequential:**

For the Sequential solution, I created seqAlgo.java and runSeqAlgo.java. runSeqAlgo.java takes in a given file with the terrain information and uses it to populate a grid. This grid is then passed into a seqAlgo object whose methods are used to classify whether or not each block is a basin, before returning a list of the basins. The classification process is done one-by-one, row-by-row, from the 2nd row and 2nd column till the 2nd last row and 2nd last columns (The edges cannot be basins).

**-Parallel:**

For the Parallel solution, I created forkAlgo.java and runForkAlgo.java. The Fork/Join framework was used to implement a divide-and-conquer solution. The runForkAlgo populated the grid from a file which it then passed into a forkAlgo object along with a sequential cutoff which I will explain. The forkAlgo class contains a recursive method which splits the array into multiple sub-grids/threads until a given cutoff. This is called the sequential cutoff since, at that point, the sub-grids are classified using the standard sequential algorithm and not split any further. The results areconcatenated to a String that is returned at the end.

A close up of text on a white background

Description automatically generated

Note that runForkJoin.class takes in the number of threads and not the exact sequential cutoff as I assume most students would have done. The number of threads wanted is then used to calculate the sequential cutoff for a given dataset size. This was done because:

1. The experiment instructions asked us to vary the amount of threads, not the Sequential Cutoff. The latter was put in brackets
2. The graphs become more readable and understandable since the sequential cutoff varies greatly with different data sizes. A sequential cutoff of 30 for a dataset of 50 results in only 2 threads but for a dataset of a 1000, it will result in 33 threads. This can result in misleading statistics.
3. Threads are more efficient measurements since they are essentially the critical points of sequential cutoffs. For example, for a 50x50 dataset, it doesn’t matter whether the cutoff is at 26 or 49. It will still only split once. Measuring by threads deals with the critical points of 25 vs 26 and 49 vs 50.

**-Validation that the algorithms are correct:**

The output for the provided for the small, medium and large data sets matched the solutions outputted by both the sequential and fork/join parallel programs.

**-Timing:**

9 more evenly spaced datasets between 50x50 and 1000x1000 were generated. A combination of python scripts and excel was then used to run the programs multiple times for each dataset and number of threads, get average time results (which excluded the first few runs for the parallel program), and then the stats stored in a csv file.

Table: Time ~ Data-size for different number of threads

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **50** | **125** | **250** | **375** | **500** | **625** | **750** | **875** | **1000** |
| **Sequential Alg. (1 thread)** | 0.032 | 0.053 | 0.085 | 0.224 | 0.54 | 1.093 | 2.192 | 4.641 | 8.645 |
| **2 threads** | 0.037 | 0.05824 | 0.07659 | 0.13453 | 0.28311 | 0.47871 | 0.82617 | 1.42582 | 2.40264 |
| **4 threads** | 0.04376 | 0.05794 | 0.07971 | 0.10424 | 0.18388 | 0.33476 | 0.49888 | 0.74812 | 1.25682 |
| **8 threads** | 0.04353 | 0.05782 | 0.06888 | 0.08824 | 0.12871 | 0.22423 | 0.297 | 0.48653 | 0.73494 |
| **16 threads** | 0.04147 | 0.06041 | 0.0665 | 0.08318 | 0.10905 | 0.15394 | 0.19576 | 0.28547 | 0.42347 |
| **32 threads** | 0.04406 | 0.05859 | 0.06359 | 0.07676 | 0.10012 | 0.11482 | 0.15012 | 0.20918 | 0.29458 |

**-Speed-Up:**

Using the data collected above, Speed-up was measured for each dataset and number of threads. (Execution time for the Sequential Solution divided by the execution time for Parallel Solution)

Table: Speedup ~ Data-size for different number of threads

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **50** | **125** | **250** | **375** | **500** | **625** | **750** | **875** | **1000** |
|  |  |  |  |  |  |  |  |  |  |
| **2 threads** | 0.864865 | 0.910027 | 1.109805 | 1.665056 | 1.907386 | 2.283219 | 2.653207 | 3.254969 | 3.598125 |
| **4 threads** | 0.731261 | 0.914739 | 1.066366 | 2.148887 | 2.936698 | 3.265026 | 4.393842 | 6.20355 | 6.878471 |
| **8 threads** | 0.735125 | 0.916638 | 1.23403 | 2.538531 | 4.195478 | 4.874459 | 7.380471 | 9.53898 | 11.76286 |
| **16 threads** | 0.771642 | 0.877338 | 1.278195 | 2.692955 | 4.951857 | 7.100169 | 11.19738 | 16.2574 | 20.41467 |
| **32 threads** | 0.726282 | 0.904591 | 1.336688 | 2.918187 | 5.393528 | 9.519248 | 14.60165 | 22.18663 | 29.34687 |

**-Machine Architecture used to test code:**

Intel Core i5-3320M @ 2.60GHz - Cores: 2, Threads: 4

RAM: 8 GB

Operating System: Windows 10 64-bit

**-Interesting problems/difficulties:**

1. The array indexing during the recursive split was a fun problem. One had to make sure they didn’t miss a line during the split. On my first try, I missed a line at the split and the second last lines of the grid. I also got a stack overflow for the first time, so it was interesting to learn about that.
2. Counting the number of basins in the Fork/Join Algorithm. Couldn’t simply increment a and return at the end since it couldn’t be returned with the recursive method. Simply counted the number of lines at the end.

**Results and Discussion**

Tables of graphs drawn below are in the previous section.

**Is it worth using parallelization (multithreading) to tackle this problem in Java?**

From the graphs, we can see that that there it definitely worth using parallelization as the data size grows larger and larger. The speed increases exponentially.

**For what range of data set sizes does your parallel program perform well?**

The larger the dataset, the better it performs. In terms of the datasets tested in this report, any dataset from 500 upwards performs visibly well relative to the Sequential solution.

**What is the maximum speedup obtainable with your parallel approach? How close is this speedup to the ideal expected?**

The maximum speedup obtained was 29.34, by creating 32 threads for the 1000x1000 grid.

The ideal expected, using Amdahl’s Law, would be:

1/(S+(1-S)/P) = 1/(1/32+(31/32)/4) = 3.65

Which means either I misapplied Amdahl’s Law, have a problem in my code, or I just disproved Amdahl’s Law (in which case I would like to submit my work as a PhD proposal).

**What is an optimal sequential cutoff for this problem? (Note that the optimal sequential cutoff can vary based on dataset size.)**

Since, for data-sets 250 upwards, the speedup was still increasing as the sequential cutoff was decreasing, I can only comment on datasets 50 and 125 should not be parallelized (or have a cutoff greater than the number rows i.e. so that it never gets split). For the bigger datasets, smaller cutoff’s are still proving more and more optimal.

**What is the optimal number of threads on your architecture?**

By observation, for most datasets, it seems to be at least 32.

**Conclusions**

* What my results tell me: Parallelization becomes exponentially more efficient as the dataset increases.
* How reliable: They are reliable on a small scale. They are hardly reliable in relation to what is possible given time and equipment. Further experimentation will need to be done with different system architectures, more repetitions, more datasets etc. There’s always a lot of room for improvement.