CSC2002S – Assignment 1

HTGTIM001

Introduction

This assignment required an empirical investigation into the practical performance benefits of parallelizing a so-called "embarrassingly parallel problem". The problem was to classify basins from a grid of height values and because this evaluation was localized, in that it only required a check of the relative heights of one layer of the surrounding basins, it was extremely suited to the *map pattern* of parallel programming. The *map pattern* involves operating on elements independently and does not require the combining of results, this presents an ideal case on which we can test the real-world performance gains of the Fork-Join Framework. The span of the problem should be around O(logn); however, this assumes infinitely many processors, thus the running time with our limited number of processors will likely increase faster than O(logn) as n increases. The work for this problem is O(n). Thus, our expected speedup T_1/T_p, is less than or equal to the theoretical speedup T_1/T_∞, that is to say $T_1/T_p \leq n/\log n$. The Work Law also implies that for a 6-core machine, the speedup (T_1/T_p) is ≤ 6. This is likely to be a closer bound than n/logn.

Methods

The canonical example of an implementation of the Fork-Join Framework involves the splitting of a 1-D array until each sub-array is within the Sequential Cutoff, thus the Boolean basin-status of each *gridpoint* was stored within a 1-D array as this would be simplest to split. The height data of each *gridpoint* was stored as a floating-point number within a 2-D array, otherwise known as an array of arrays, with the first index corresponding to row and the second index corresponding to column. Thus, the solution required a conversion between 1-D and 2-D coordinates. If the objective 2-D position of the element of the 1-D array was tracked this was entirely possible. Keeping track of 2-D position required a column and row count which represented the current coordinates of the 1-D element in the 2-D grid, and the number of columns and rows in the grid. The number of rows and columns in the grid was required for when a *right* parallel object was created as the position in 2-D space would not be the 2-D position in the current thread, as this position was passed to the *left*, but rather a position corresponding to the new *lo* which was a 1-D coordinate. Thus, the 2-D position would have to be calculated from this new *lo*. The row position could be calculated by (new *lo*)/(number of columns) and the column position by (new *lo*)%(number of columns).

Once the issue of 1-D and 2-D coordinates was resolved, the parallelization effort became far easier. A parallel class was created, this class inherited from RecursiveAction as this was the appropriate choice for a *map pattern*. The parallel class overrode the compute method and the method would split the array further, and in doing so create an additional thread, if the current size of the array were not within the sequential cutoff. If the size of the array was within the cutoff, then the sequential portion to the parallel solution was run. Within the sequential portion, *gridpoints* were classified as basins or non-basins by *true* or *false* Boolean values respectively. All edges in a grid can never be basins, as a basin requires a ring of neighbours with relatively greater height values of 0.01 or more. Thus, all edges were excluded from further checks by means of an if statement which relied on the row count, column count and total rows and columns. The relative positions of the neighbours of a potential basin were constant in that they comprised of basins in the row above from the potential basin's column position -1 to the potential basin's column position +1, the *gridpoints* to its immediate left and right, and *gridpoints* in the row below within the same column range as the row above the potential basin. Therefore, knowing the 2-D coordinates of the potential basin it was possible to infer the coordinates of all its neighbours and thus access their height values within the 2-D grid array. If all eight neighbours had greater relative heights of 0.01 or more, the potential basin was classified as a basin by inserting a *true* value into its position in the 1-D array. Otherwise its status as a non-basin was maintained as all positions within the 1-D array were initialized to false by default.

The sequential check of the status of a basin was theoretically sound, but this was further confirmed by testing against the standard datasets and by creating edge cases to test against. The edge cases consisted of: A grid with no basins, a grid with the same values in every position, a rectangular grid, a grid of all zeroes, a grid where a *gridpoint* fails to meet the criteria of a basin by 0.001, a grid where the lowest values are on the edges and a grid where the basin has a relative height deficit of 0.01. The sequential and parallel solutions passed all standard and edge cases except for the last edge case, however after doing some research and testing, this appears to be a floating-point precision issue rather than an algorithmic problem. From this research it was made clear that *BigDecimal* is a more appropriate choice for exact precision than floating-point number; however, seeing as the brief suggested the use of floating-point numbers, it was assumed that this issue would be overlooked.

Timing was done via a different set of arguments than simply outputting the results of the basin search to a file. The algorithm to be timed was given by either "pt" for Parallel Timing or "st" for Sequential Timing, the input file was then specified and in the case of Parallel Timing, a sequential cutoff was given as the final argument. Within the main method of FindBasin which handles simply outputting the results of a basin search and timing a particular algorithm, appropriate variables are assigned, the 2-D grid and 1-D array are created as usual and an additional array to store times. System.gc() is called before timing to minimize the likelihood that the garbage collector runs during execution, after this, the appropriate basin finding method is called 50 times and for each run, start time is recorded in nano seconds and after

each run completes the difference between current time and start time is assigned to the array of times. After all iterations are complete, the times are printed out in milliseconds and finally min, mean and max are printed out.

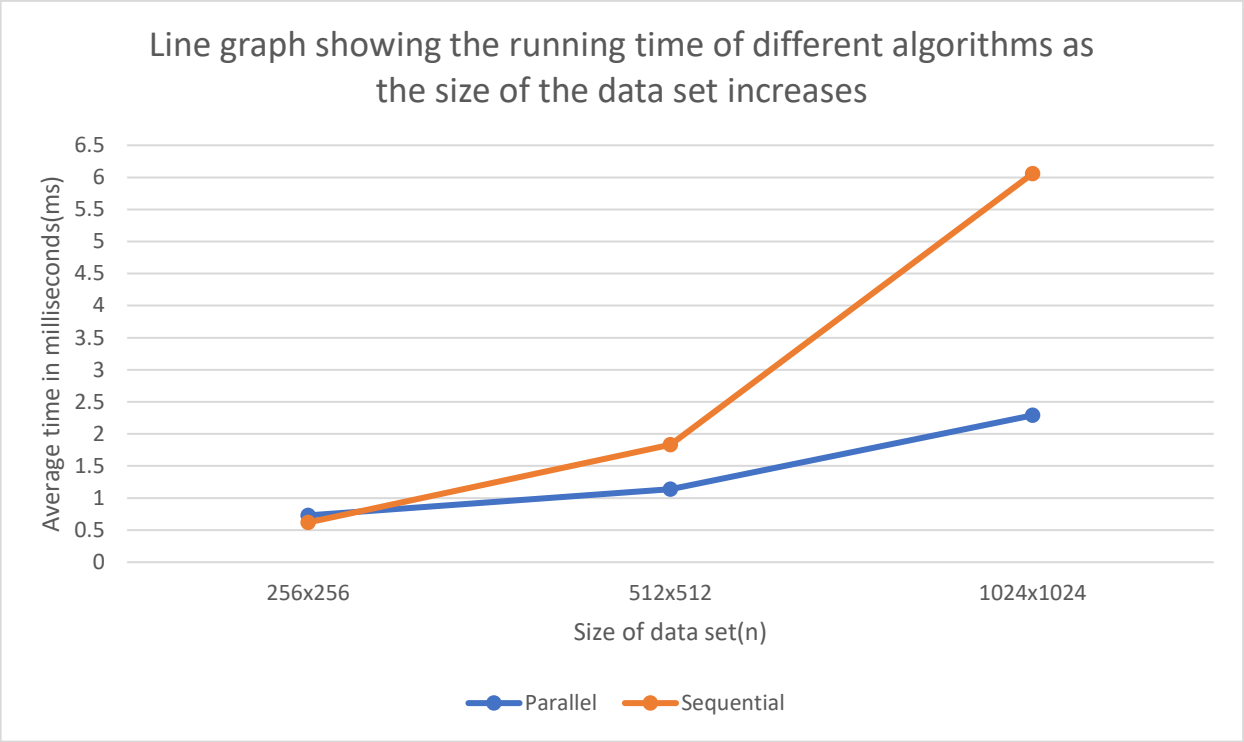
The approach to finding a sequential cutoff for a particular data set was to choose some appropriate starting point and then rapidly increase in sequential cutoff value and when an interval in which a significant speedup occurs was found, subsequent timings with smaller jumps were done within that interval. The emergence of “plateaus” was also taken into account.

The machine on which the timing was done is x64-based laptop with an 8th generation Intel CPU with 6 cores.

Initially when testing, timings were assigned to floats, however this led to some timings being 0, once the type was changed to long, this issue no longer persisted. Presumably this was a precision issue, given that nanoTime() was used for timing.

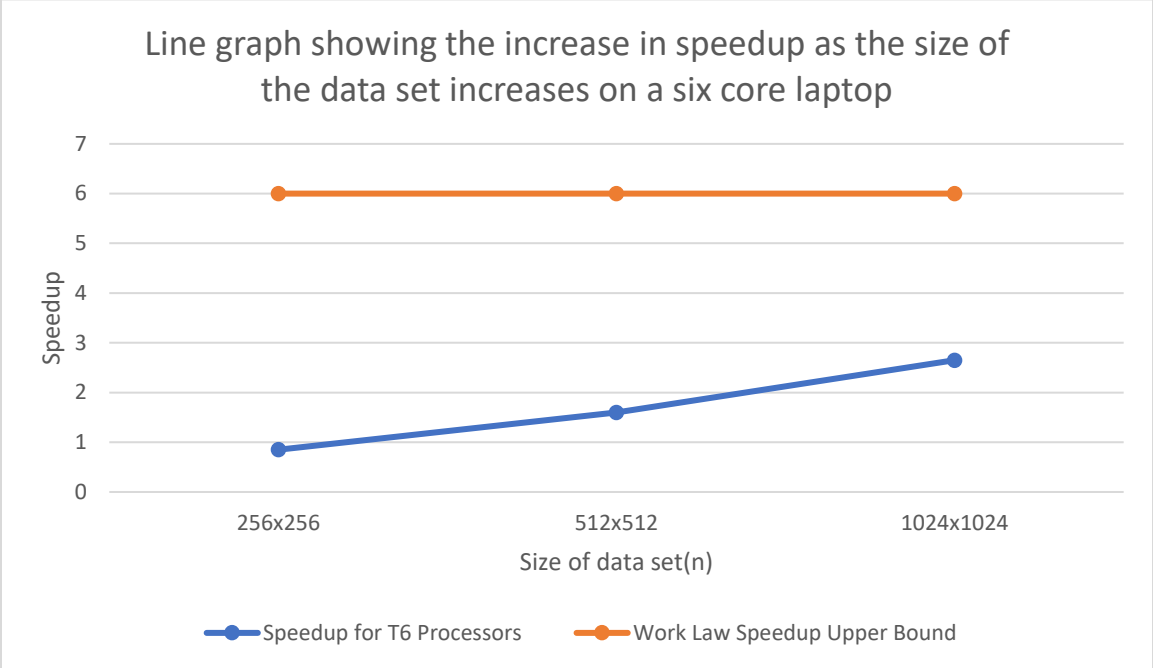
Results and Discussion

All timings were done for 50 runs, the garbage collector was called before the timing blocks and timing was isolated to identification of basins.

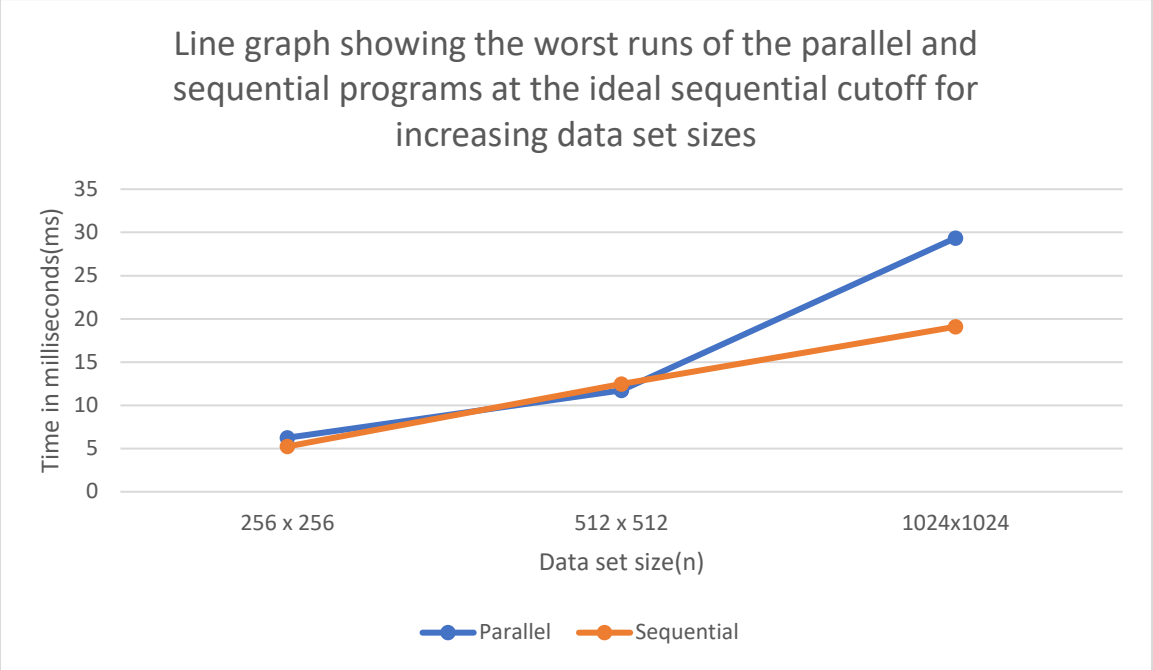


For trivially sized datasets, less than or equal to 256x256 and potentially between 256x256 and 512x512, the parallel algorithm yields worse results than the sequential algorithm. This is likely due to the thread overhead dominating the running time more so than the cost of the computation occurring in each thread. It is also important to note that the above graph represents the runtime of the parallel program when using ideal sequential cutoffs, poorly selected sequential cutoffs would produce even worse results at trivially small data set sizes. Thus, when determining whether parallelization is worth doing, in addition to the parallelizability of the problem, the size of the datasets involved in the problem should be strongly considered

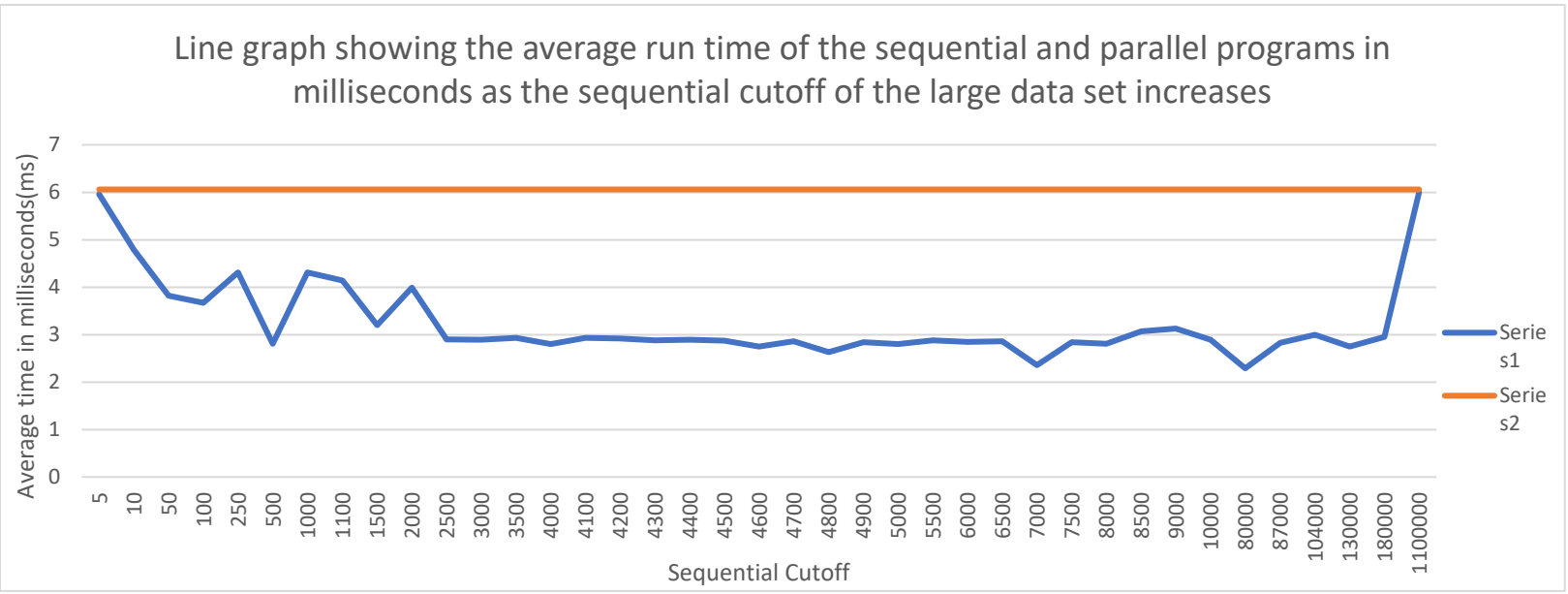
For sufficiently large datasets, the parallelization of this problem yields significantly better results which appear to improve as the size of the dataset increases. A speedup of 1.6x was achieved for a 512x512 dataset with an optimal sequential cutoff, and this speedup increased to 2.65x at a 1024x1024 size, this suggests that with increasingly large data sets, greater performance gains are to be expected. The Work Law states that: $T_p \geq T_1/P$ (ignoring superlinear speedup) where T_1 is the running time with 1 processor and T_p is the running time with P processors. Speedup is calculated by T_1/T_p, therefore the Work Law implies that T_1/T_p(the speedup) ≤ P(the number of cores). Thus, the theoretical upper bound of our speedup is P = 6 as the machine on which timing was done has 6 cores. The largest speedup that was achieved was around 44% of the ideal theoretical speedup, this is of course somewhat disappointing, however Gustafson's weak scaling tells us that as problem size increases (ie. Data set size), the ratio of sequential running time (T_s) to parallel running time (T_p) is not fixed and that T_p typically grows faster than T_s, thus for larger dataset sizes we benefit more from parallelism. Therefore, for larger data sets we expect to achieve speedup closer to the Work Law's upper bound.



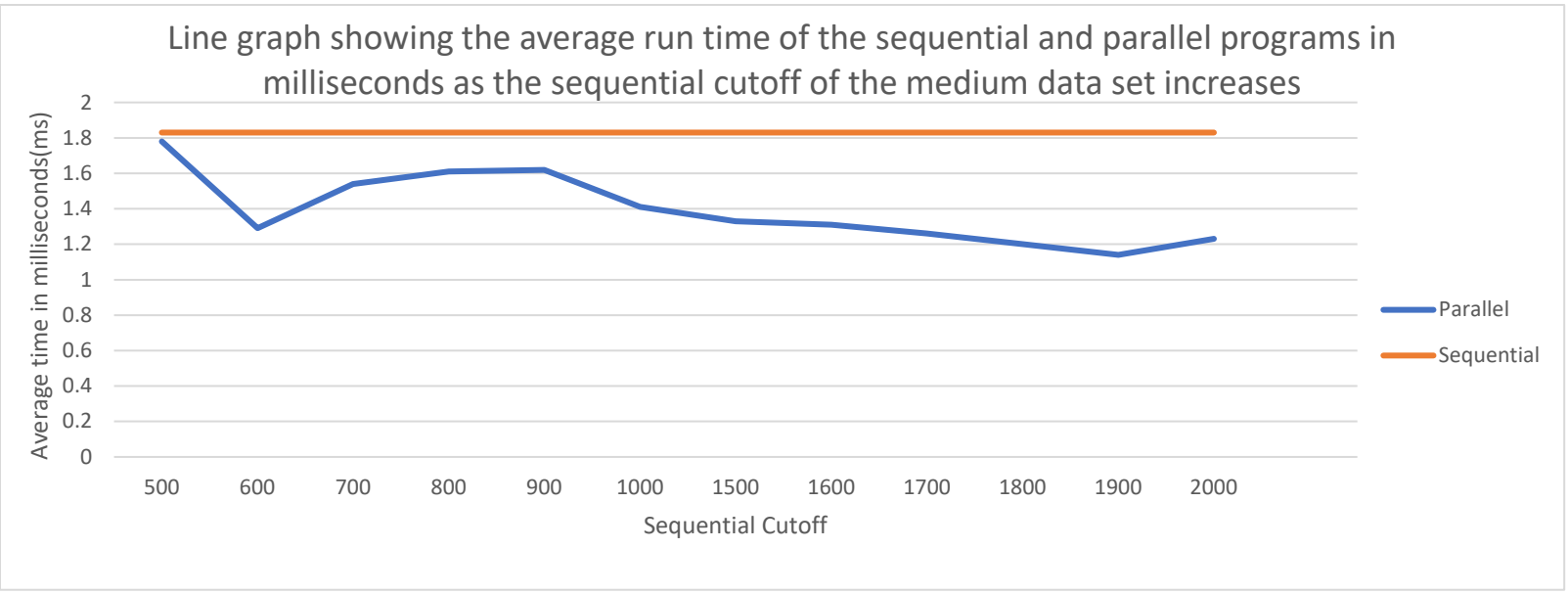
If the number of cores as well as the size of the dataset were to be scaled up, our theoretical maximum speed up (assuming infinitely many cores) is T_1/T_∞, which is O(n/logn).

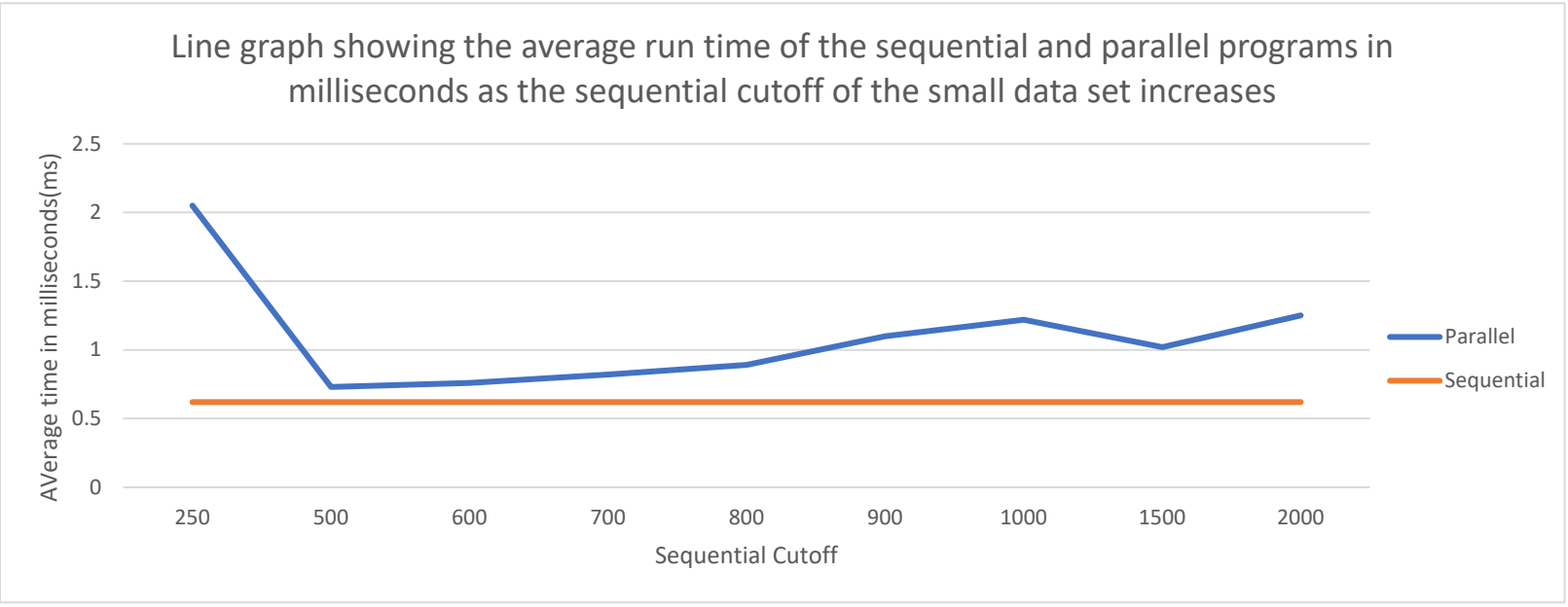


As the assignment is primarily concerned with average run time these results are not especially important as over 50 runs their impact on the average is reduced. The tendency of these worst runs to occur within the first few runs for both the parallel and sequential program suggests that this a behaviour which is Java-specific, rather than parallelism-specific.



At a very low sequential cutoff, the thread overhead significantly degrades performance and thus the parallel performance is similar to the sequential performance. With even a modest increase in the sequential cutoff from 5 to 10, a large performance gain occurs. This is largely due to around half the number of threads being created, and therefore a comparable decrease in thread overhead occurs. For larger sequential cutoffs this dramatic affect is no longer apparent as the average run time appears to plateau at just under 3ms with occasional dips. This suggests that after increasing the sequential cutoff past a certain point, the thread overhead starts to become largely negligible. The largest dip in the plateau is around a cutoff of 80000(around 13 threads) and this was chosen as the ideal sequential cutoff. Once the sequential cutoff gets closer toward the dataset size of 1024^2 the performance of the parallel program degenerates to that of the sequential solution because it eventually performs all of its computation on a single thread, just as the sequential solution does.





The two smaller data sets were timed across a smaller range of sequential cutoffs, in the case of the smallest data set it was because the average run time appeared to trend away from the average sequential time rather rapidly and it was expected that the parallel algorithm would perform worse for this size of data set. The smallest data set was at its fastest with a sequential cutoff of 500(around 130 threads). In the case of the medium data set, it began to plateau at around 1.2-1.3 ms quite quickly and didn't leave this range even for sequential cutoffs as high as 22000 which would produce the same number of threads(around 13) as the optimal sequential cutoff for the large dataset. The medium data set was at its fastest with a sequential cutoff of 1900 (around 130 threads). The optimal number of threads appears to be around 130

threads for smaller datasets and for larger data sets around 13 threads. The CPU on which testing was done has 6 cores so the theoretically optimal amount would be 6 threads, however not all of its cores may be available at a given point and thus the best approach to determining a sequential cutoff is through empiricism.

Conclusions

The parallelization of an "embarrassingly parallelizable" problem via the *map pattern* is worthwhile for datasets of 512x512 and larger, with significant performance gains emerging from around 1024x1024, our results, although well below the Work Law's bound, taken in conjunction Gustafsons Weak Scaling lead us to expect greater gains with increased dataset size. It should however be noted that the cost-benefit of parallelization is also dependent on hardware, thus in the case of machines severely limited by core counts an opposite conclusion should be drawn as no significant performance gains can be made.

When timing, the first 1-4 runs of the parallel program tended to be far slower and this behaviour was also observed to a lesser extent for the sequential program, this suggests that perhaps some "under the hood" JVM optimization needs to occur before a true measure of a program's performance can be ascertained.(Due in part perhaps to Java's extremely general-purpose design: "Write once, run anywhere") As this behaviour was consistent, it leads to the caveat that parallelization in Java is only worth it when a program is going to be run continuously(or many times) or for such large data sets that even the initial parallel runs yield worthwhile performance gains.

Ideally, this assignment should have been carried out across multiple machine architectures to allow for a greater depth of analysis. However, within this limited scope similarly limited conclusions about parallelization in Java can be drawn as discussed above.

Going the extra mile

- Parallelizing the final extraction of the list of basins.
- Testing for edge cases as this was not explicitly required in the brief.
- Using git branches, github and a .gitignore, as this presents a more realistic workflow than committing everything to master locally.