Introduction

In order to anticipate the effects of climate change on vegetation, it is required that vegetation patterns be simulated as a forecasting method. These models scale in computational complexity with the size of the input terrain, so in order to speed up the process, parallel programing techniques can be implemented. A basic model will be analyzed, with each data point in the map representing the average sunlight over a 1x1 square, and the tree data which is represented as a square, with the start point in the top left corner of the tree, and a cover of the specified size. This will be used to find the average sunlight per tree, as well as the total average sunlight across the terrain. Trees which have out of bounds components, will only be calculated for the sections which are in bounds, and overlap of trees is ignored. In order to parallelize the simulation, the trees sunlight calculations will be done in parallel, with the expectation that the speedup time will for a P processor system will be $T_p = \frac{T_1}{P}$ assuming the number of trees is larger than the number of processors in a system. This gives a O(log n) speed-up.

Methods

The following solution to this problem was solved, by initially reading in all input values, and assigning them to their respective variables, this is done sequentially by the readInFile( ) method, once completed, the data is iterated over thrice before actions are performed on the data, this is so that the data will be in the cache which should speed up the performance of the parallel algorithm. Once the data is in the cache, the sequential threshold is altered, from the full data set on one thread, to one thread per action, or 99 threads, if data size is larger than 100 items. For each value of the sequential threshold, parallel sorting of the data is done. The parallel algorithm splits the work by number of trees, trees are evaluated on a in parallel to find the average sunlight hitting the tree, which is then stored by the tree object, the average sunlight for each thread consists of the sum of the average sunlight values for all trees evaluated on that thread, once all data has been worked through, the average sunlight per thread is summed together to get a global average value for sunlight hitting the canopies. The time for each iteration is recorded, with the thread number and sequential threshold, these values for all data sizes can be found in the sample_output.txt files. The most efficient sequential cutoff, is also evaluated while the algorithm executes, by comparing each iterations time taken to the previous minimum time taken. Once the testing was completed, the algorithms is called again, with the most efficient parameters found previously, and the required output is generated for the data, into the sample_output_vals.txt files.

Code was tested on a laptop (Intel i7-6700: 4 Core, 8 Threads) and the Raspberry pi (4× ARM Cortex-A53), speed-up times were calculated, by dividing the sequential result by the most efficient result to get maximum speed up for that iteration of the data. Interestingly the most efficient value for the thread count was not consistently the same for successive iterations of the same data, this means that there is no one best solution, since other factors must be taken into consideration, such as processing power readily available at any given point in time. Data for the large input could not be handled on the Raspberry pi so has been excluded. This is likely due to the limited processing power output by the ARM processor

My algorithm was proved correct by comparing results of small data sizes to their expected values, this was only done for smaller values because the task is a recursive one, meaning if it works for the base set, it will work for smaller sets. Testing the boundary conditions was the critical part.

Results and Discussion

Sequential Results:

Threads:1

Sequential Cutoff: 1000000

Elapsed Time: 0.0636

Speed-Up: 4.58 times faster

```
Most Efficient:
Threads: 43.0
Sequential Cutoff: 23255.0
Elapsed Time: 0.013975303000000001
```

Large Sample input Laptop

Sequential Results:

Threads:1

Sequential Cutoff: 2

Elapsed Time: 5.3335E-5

Speed-Up: 1 times faster

Sequential Results:

Threads:1

Sequential Cutoff: 20

Elapsed Time: 8.15E-4

Speed-Up: 9.108 times

Sequential Results:

Threads:1

Sequential Cutoff: 160

Elapsed Time: 1.17396E-4

Speed-Up: 1.155 times faster

```
Most Efficient:
Threads: 1.0
Sequential Cutoff: 2.0
Elapsed Time: 5.3333000000000006E-5
```

Small Sample input Pi

```
Most Efficient:
Threads: 9.0
Sequential Cutoff: 2.0
Elapsed Time: 8.9479E-5
```

Small Sample input 2 Pi

```
Most Efficient:
Threads: 8.0
Sequential Cutoff: 20.0
Elapsed Time: 1.01667E-4
```

Small Sample input 3 Pi

Sequential Results:

Threads:1

Sequential Cutoff: 2

Elapsed Time: 1.3432E-5

Speed-Up: 1 times faster

Sequential Results:

Threads:1

Sequential Cutoff: 20

Elapsed Time: 7.5061E-5

Speed-Up: 1.3 times faster

Sequential Results:

Threads:1

Sequential Cutoff: 160

Elapsed Time: 5.0133E-4

Speed-Up: 4.77 times faster

```
Most Efficient:
Threads: 1.0
Sequential Cutoff: 2.0
Elapsed Time: 1.3432000000000001E-5
```

Small Sample input Laptop

```
Most Efficient:
Threads: 4.0
Sequential Cutoff: 5.0
Elapsed Time: 5.5703000000000005E-5
```

Small Sample input 2 Laptop

```
Most Efficient:
Threads: 49.0
Sequential Cutoff: 3.0
Elapsed Time: 1.05087E-4
```

Small Sample input 3 Laptop

Discussion

As seen from the results above, the parallel performance varied widely based on the architecture. On both machines, parallelization would be worth the performance increase, however it was seen that on the raspberry pi, as the data size increased, the performance decreased, this means that for small data sizes, it is worth parallelizing the task, but as the complexity increases, the performance decays. On the i7 processor, it was seen that for small problem sizes, it is unnecessary to parallelize the task, because the performance increase is negligible and not worth the increased complexity, however as the data size increased, the performance of the i7 improved, and remained consistent for large data types, this of course, will not remain true as the data size is further increased, this is because parallelization has limits. For the pi, the optimal thread count appeared to be around 43-49 threads, and I believe will vary based on data size. The sequential cutoff was also very dependent on the data size. Using the formula,$S = \frac{1}{1-p+\frac{p}{n}}$, using P as 100% due to only timing the parallel component of the code, with 4 processors, the theoretical maximum would be a 4x speed up, however due to the computers not giving optimal performance due to resource sharing, the speedup can be a lot faster if the background applications use a lot of processing power during sequential analysis. The inverse is true, the system can also be slowed down.

Conclusion

It has been seen from this assignment, that different architectures have different levels of parallelization, noting that some work far better than others. However, it must also be said that theoretical performance and actual performance will never be exact, and will likely vary depending on the strain on the processor, although the results show a good correlation of the data, and allow for analysis of the performance on different architectures, I would not say they are reliable when trying to find individual performance of any one device. This is because when using an operating system that many services and background activities, the performance of any singular process will suffer. And thus in turn leads to results which are not consistent with theoretics.