Elle Mouton (MTNELL004)

Introduction

The aim of this project it to analyse the effectiveness of using a parallel program to accomplish a task and compare it to using a sequential program to accomplish the same task. The reason for using a parallel algorithm over a sequential one is ultimately to produce a speed up in the time that the algorithm takes to complete a task.

Sequential programs only make use of 1 core to complete a task but with a parallel program we can make use of multiple cores and this allows multiple parts of a program to be run simultaneously and thus it is expected that as the number of cores available to a parallel program increases, so should the speed up of that program. In this project, the aim is thus to determine if the a parallel solution to the given problem is more effective than the sequential solution and what factors affect the speed up of the parallel algorithm.

The task that will be used to test the speed up of a parallel program involves calculating the average sunlight per tree canopy as well as the total sunlight per tree when a sun map of a given area is provided along with the size and location of each tree canopy.

Methods

How the algorithm works:

In order to solve this task I created 3 classes namely “SunCalc.java” which contains the main function, “SunThread.java” which extends the RecursiveTask class of the ForkJoin framework as well as a “Tree.java” which is an class used to represent individual trees.

The initial part of the SunCalc program is completely sequential and involves reading in the dataset values from a given text file. From the data a 2D matrix is created and populated and it represents the sun map data. Then an array of type Tree is created and for each tree in the dataset, a Tree object is created that contains the details of the trees position and dimension and each Tree object contains a variable that represents the amount of sunlight that the tree gets and this variable is initially set to zero.

Once all the data has been read from the text file and all the Tree objects in the Tree array have been created, a new SunThread object is created and invoked and the entire Tree array is passed to this thread.

The SunThread thread will handle the data as follows: if the size of the array passed to the thread is larger than the sequential cutoff (a value that is manually set in the SunThread class) then two new SunThread classes will be created by the parent thread, one called “left” and one called “right” and each of which will be passed half of the array that the parent thread received. The parent thread then calls “left.fork()” which allows the thread called “left” to start its own computation and in the mean time the parent thread calls “right.compute()” which runs the compute() method of the right thread. Once the right thread has returned an answer the parent waits for the left thread to complete and then adds the answers from the left and right threads together and returns the sum.

This recursion will continue until a thread is given an array that is smaller than the sequential cutoff. In this case, no new threads will be created and instead the thread in question will do the computations for all the Tree objects in the thread’s Tree array. For each Tree object in the array, the sun map will be consulted and the total amount of sunlight that each tree receives will be added up sequentially and the Tree’s “sunlight” variable will be set to that total amount. The thread that is executing the calculations for the trees sequentially will also add up the total amount of sunlight received by all of its trees and will set its “ans” variable to this total and when it is complete it will return this total to its parent thread. Therefore each thread that runs the program sequentially will return the sum of the total amount of sunlight received by each tree in the array handled by the thread. And each thread that forks off new threads (left and right) will return the sum of the values returned by those threads. Thus the main thread will return the sum of all sunlight values of each tree and as it does this, the sunlight for each tree is calculated and saved.

The last part of the program runs sequentially. It calculates the average sunlight exposure for all trees by dividing the number returned by the main thread by the total number of trees and then loops through the array of trees and prints out, in order, the total sunlight exposure for each tree.

Validating the algorithm:

In order to validate that the program runs correctly I wrote a program that does the entire task sequentially and the ran both of the programs with the same input data and then I wrote a program that compared the outputs of the two programs to ensure that the results were the same.

Analysing the algorithms:

To compare the parallel and sequential programs, timing was used to time the part of the programs where the sunlight calculations were done. This timing was repeated using different sequential cutoffs (ie varying the number of threads used), using different dataset sizes and also running the the programs on different machines architectures.

First I used my computer, a MacBook Pro with 4 cores, to do the testing and timing and I used dataset sizes of 1 million, 2 million and 3 million to test and time the programs. I then tested the programs on a Unix machine with 4 cores and I varied the dataset sizes in the same way and then I tested the programs on a Unix machine with 2 cores in the same way. In all these cases, I would first run the sequential program (1 thread) and record the time taken by the program to run the sunlight calculation part of the program and then I would use the parallel program and would vary the sequential cutoff (and therefore number of threads) to time the same process. Then for each case I could calculate the speedup using the following formula:

$$SpeedUp = \frac{T_1}{T_p}$$

Where T_1 is the time it takes for the sequential program to complete the task and T_p is the time it takes for parallel program to complete the task given a certain sequential cutoff.

Difficulties that I encountered:

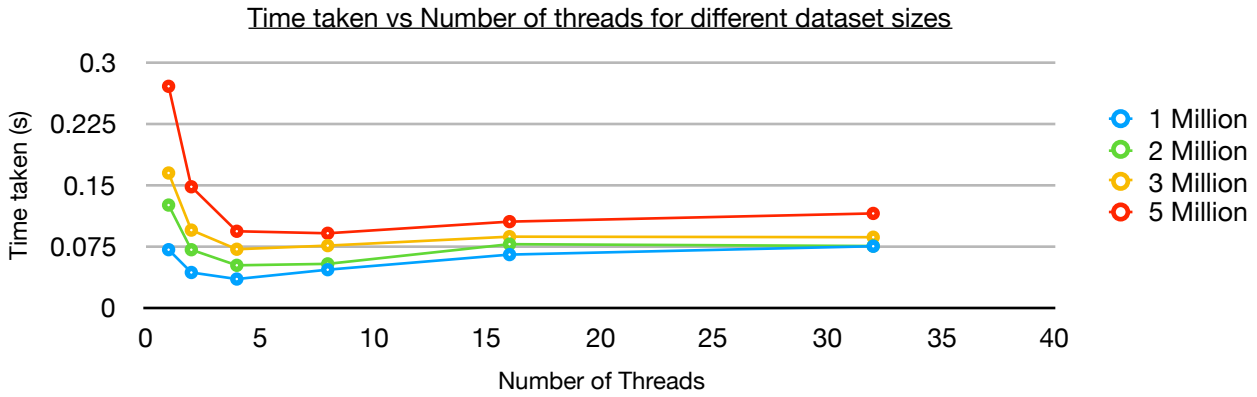
The main problem that I encountered was that the average sunlight calculated by my parallel program would change as the number of threads it used changed and that it would only match up with the answer from my sequential program when 1 thread was used. And another problem was that even my sequential program was not calculating the correct average even though all the calculations for the individual trees where correct. Both these issues were due to the fact that I was using the *float* type for all the sunlight values and calculations and after doing research and doing various test cases, I discovered that the *float* type can be inaccurate for numbers with many decimal points. After changing my program to only the *double* type for all the sunlight values and calculations, all the programs worked and produced the correct answers.

Results and Discussion

4 Core Mac

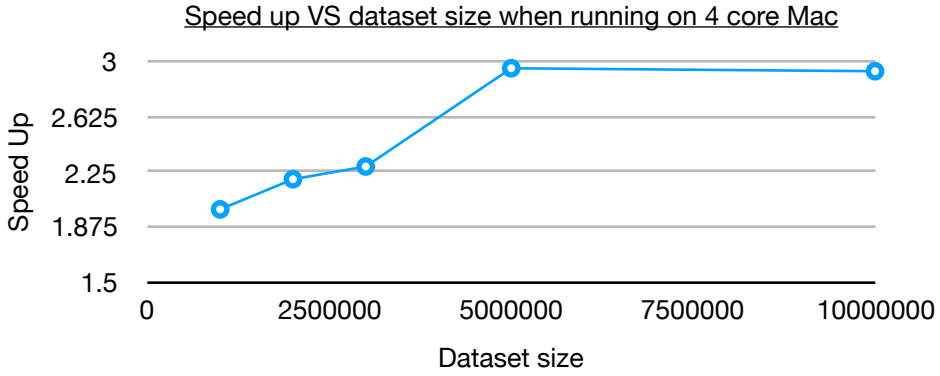
I first did tests on my 4 core Mac by timing the program for different dataset sizes and for each dataset size I would vary the sequential cutoff so that I could determine how the number of threads used by the program would affect the speed of the program. My estimation for the ideal maximum speed up that my 4 core Mac can provide is a speed up of 4: $T_4 >= \frac{T_1}{4}$

The graph below shows how the time taken for the program to run varied as the number of threads used was altered as well as the dataset size.



From the graph above, it can be seen that the time taken for the program to run decreases significantly between the change from 1 thread (sequential) to 2 or 4 threads. It can also be seen that this change is larger for larger datasets because the graph clearly shows that the speed up for the dataset size of 1 million is smaller than that for the dataset size of 5 million. Another thing to note from the graph above is that on my 4 core machine, the number of threads that result in the largest speed up seems to consistently be 4 threads since using 4 threads results in the least amount of time taken for the computations to complete for a given dataset size.

From the data collected, the maximum speedup for a given dataset size can be calculated and the figure below shows how the speedup varies with dataset size.

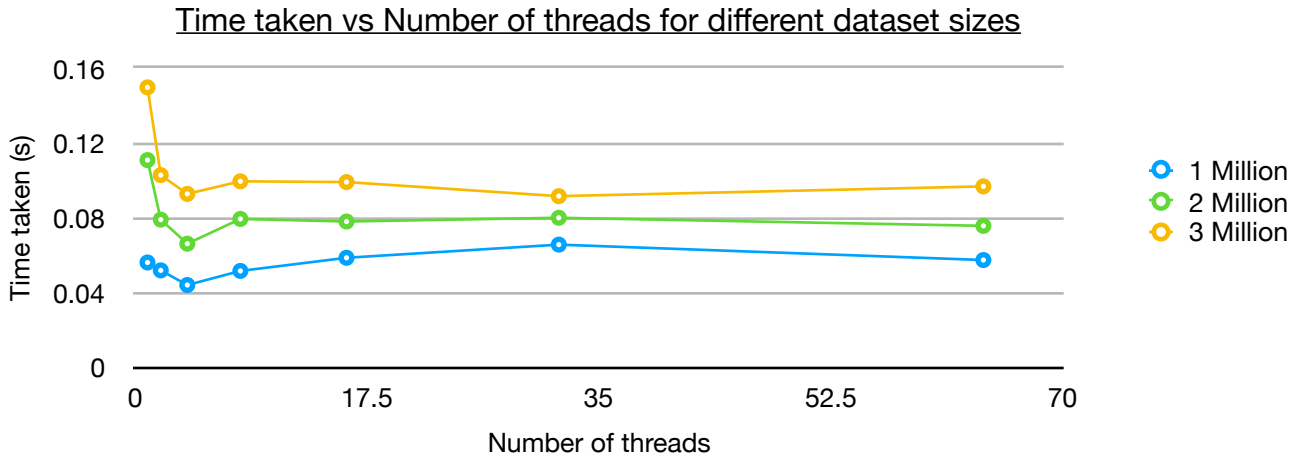


The graph above shows that as the dataset size increases, the speedup that the parallel program offers becomes significantly larger and closer to the ideal speedup that a 4 core machine can provide which is 4. With a dataset size of 5 million, the speed up is 2.9. However the graph also shows that as the dataset size increases between 5 million and 10 million, the speed up that the parallel program offered did not vary much.

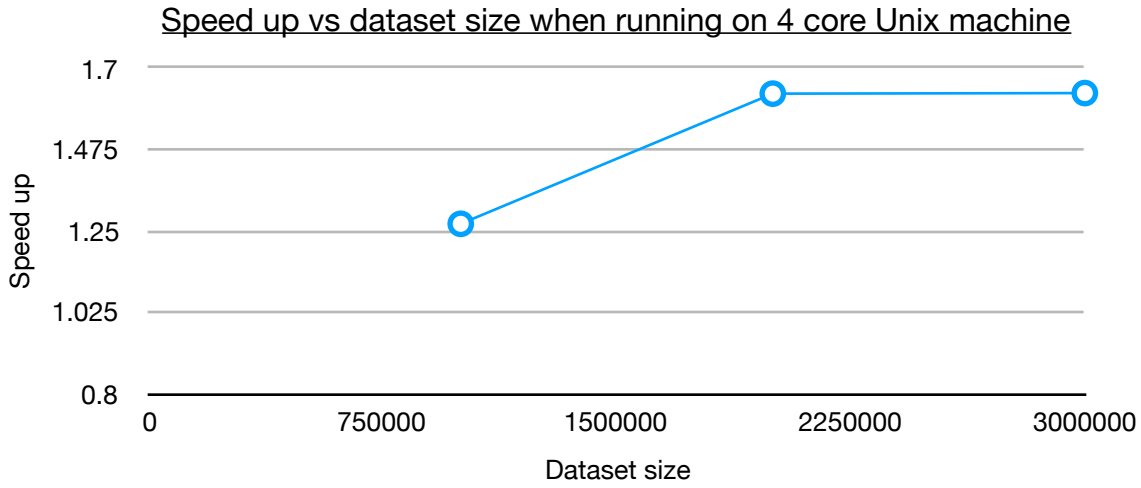
For large dataset sizes of more than 5 million, this parallel solution offers a significant speed up of 2.9 which is 72.3% of the ideal speed up that the machine can offer (4) and I think this makes it worth it to parallelize the problem. For this machine architecture, the ideal number of threads is 4 and so for a dataset size of 5 million the sequential cutoff should be 1250001 and for a dataset size of 10 million it should be 2500001 in order to ensure that 4 threads are being created.

4 Core Unix machine

Similar tests were then done on a 4 core Unix machine. The graph below shows the time taken for the parallel program to run for different ranges of input datasets as the number of threads is varied. Similarly to the data found with the Mac machine, the change in time taken as the number of thread increases from 1 to 4 is larger when the input dataset is larger. It can also be seen that the largest speedup occurs when the number of threads used is 4. Again, my estimation for the maximum speed up that this machine can offer is 4 since it is a 4 core machine.



From the data collected, the maximum speedup for a given dataset size can be calculated and the figure below shows how the speedup varies with dataset size.



The graph above shows that as the dataset size increases, the speedup that the parallel program offers becomes larger but reaches a peak of approximately 1.63 and does not change as the dataset size increases from 2 million to 3 million.

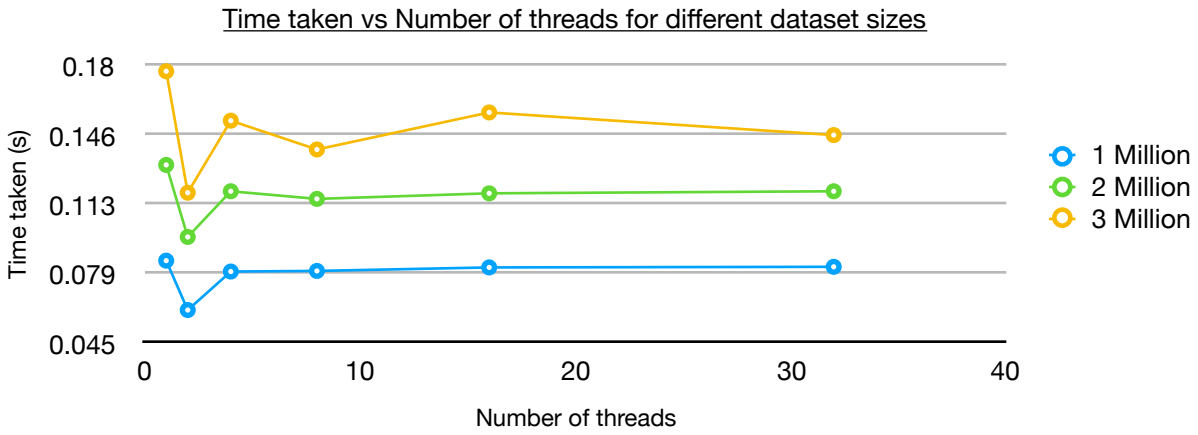
For large dataset sizes of more than 2 million, this parallel solution offers a speed up of 1.63 which is 40.8% of the ideal speed up that the machine can offer (4) and I think this still makes it worth it to parallelize the problem since it means that an algorithm that with a sequential program would take 1 hour to run would with the parallel program take 36 minutes to run which I think is a significant improvement. For this machine architecture, the ideal number of threads is 4 and so for a dataset size of 2 million the sequential cutoff should be 500001 and for a dataset size of 3 million it should be 750001 in order to ensure that 4 threads are being created.

2 Core Unix machine

I then repeated the tests on a 2 core Unix machine to see how the number of cores available would change the effectiveness of the parallel program. My approximation fo the maximum ideal speed up that a 2 core machine can provide is 2 due to the following formula:

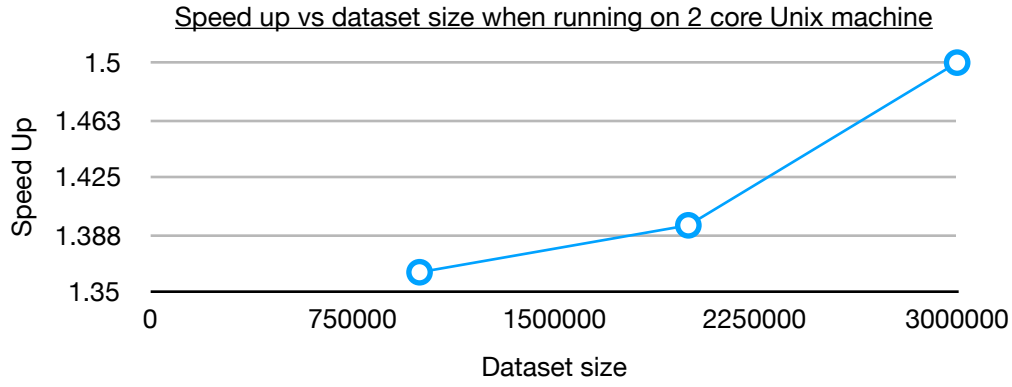
$$T_2 >= \frac{T_1}{2}$$

The graph below shows how the time taken for the program to run varied as the number of threads used was altered as well as the dataset size.



From the graph above, it can be seen that the time taken for the program to run decreases significantly between the change from 1 thread (sequential) to 2 threads. It can also be seen that this change is larger for larger datasets because the graph shows that the speed up for the dataset size of 1 million is smaller than that for the dataset size of 3 million. Another thing to note from the graph above is that on the 2 core machine, the number of threads that result in the largest speed up seems to consistently be 2 threads since using 2 threads results in the least amount of time taken for the computations to complete for a given dataset size.

From the data collected, the maximum speedup for a given dataset size can be calculated and the figure below shows how the speedup varies with dataset size.



The graph above shows that as the dataset size increases, the speedup that the parallel program offers becomes significantly larger and closer to the ideal speedup that a 2 core machine can provide which is 2. With a dataset size of 3 million, the speed up is 1.5 which is very close to the ideal speed up of 2.

For large dataset sizes of more than 3 million, this parallel solution offers a speed up of 1.5 which is 75% of the ideal speed up that the machine can offer (2) and I think this makes it worth it to parallelize the problem since it means that an algorithm that with a sequential program would take 1 hour to run would with the parallel program take 40 minutes to run which I think is a significant improvement. For this machine architecture, the ideal number of threads is 2 and so for a dataset size of 2 million the sequential cutoff should be 1000001 and for a dataset size of 3 million it should be 1500001 in order to ensure that 2 threads are being created.

Conclusions

The first conclusion that can be drawn is that for larger dataset sizes, the parallel algorithm provides the most speedup. This is true regardless of which machine architecture was used. However it was also seen that there is a limit to the speed up of the parallel algorithm and that after a certain dataset size, the speed up no longer continues to increase. Thus it seems that given a certain number of threads, each machine architecture has a limit to how much of a speed up a parallel algorithm will be able to provide.

The second conclusion that can be drawn is that the largest speed up on all the machine architectures always occurs when the number of threads created is equal to the number of cores that the machine has. This makes logical sense because the machines can run threads on the separate cores simultaneously and when each core is given 1 thread, the speed up will be the most. This is particularly true for a problem like this where each thread does roughly the same amount of computation and so there is no load imbalance problem.

A third conclusion that can be noted is that for non of the machine architectures tested did the speed up ever reach the theoretical idea speed up of the machine. And so it is perhaps the case that the operating system does not allow java to use all the cores of the machine.