Introduction

The aim of this assignment was to test parallel computing against sequential computing by means of a median filter. A median filter is a method that combs through the data generating a new data set where each item is replaced by the median of its neighbours.

A sequential algorithm was made to run through the input data set to get the achieve this filtering.

The parallel computing application made use of a divide and conquer method, specifilcally the fork-join method, to use multiple threads to try and achieve speedup by these threads working concurrently on different cores.

The theoretical/ideal speed up is the inverse of the time it takes the sequential process divided by the number of cores of the computer.

Methods

The first goal was to take in a file name and filter size and read the input file to create an array with which filtering would occur. This was done using a scanner and a while loop.

A parallel filter class was made which extends recursive action with the intention of using multiple threads.

The first step I took in solving the parallel algorithm is getting the sequential algorithm for when the size of the data set is less than the sequential limit.

The first action in the sequential section was to make sure that the borders were not changed from the input to the output data set. The next step was to incrementally make sub arrays of the input data set of the size of the filter window and take the median of the subarray and add it to the output array.

A parallel component was set up when the size of the data set exceeded the sequential limit. This was achieved by splitting up the array and creating two new parallel filter classes. The fork join method was used to compute these two at the same time as they are different threads.

Correctness of the parallel program was tested by comparing its output to a known filter output given a known input, and it was also compared to the sequential output.

Time taken to compute was done by using System.nanoTime() before and after the filtering process.

An automation program was attempted in python tochange the arguments with which the program was run, however I failed to produce a suitable application to do so by the night before submission so, each argument combination was run 20 times and the average calculated manually.

The impact of input file size was done by keeping the filter size and sequential cutoff the same while changing the number of lines the input file had.

The impact of filter size was done by keeping input file size and sequential cutoff the same, while changing the filter size.

The impact of sequential cutoff was done by keeping the input file size and filter size the same while changing the sequential cutoff.

Results and discussion

Impact of input file size:

Chart, scatter chart

Description automatically generated

From the above graph, increasing the filter has an associated linear slowdown with regards to time taken.

Comparing the parallel program to the sequential program, the sequential program performs slightly better than the parallel program at small filter sizes, however becomes slower than the parallel program as the filter size increases. This could be since for each process integers need to be copied from the array which means that there is an overhead to sending data to the processes and the smaller the window size the more windows that needs to be sent. This favours the sequential application.

Impact of input file size:

Chart, scatter chart

Description automatically generated

The graph shows that increasing the input file size exponentially increases the time taken by both filters but has a far larger effect on the sequential filter. This means that for smaller input files, sequential programming is faster, but as the size of the input file increases, the parallel program becomes faster than the sequential program. This is expected because when the file size becomes large enough the speedup obtained by multithreading outweighs the slowdown of the overhead of creating the threads.

Impact of sequential cutoff on parallel filter:

Chart, scatter chart

Description automatically generated

The points on the graph not having a linear or exponential pattern, but rather going up and down suggests that there is a “goldilocks” zone for when it will be the fastest. This is expected as a sequential cutoff that is too small will wait too long before using the sequential part of the program and will split it up into too many threads and having a sequential cutoff that is too large will cause the program to run the sequential progress too soon, thereby not giving as good of a speedup.

Using the graphs above, the following questions can be answered.

* Is parallel programming worth it for this problem:

Parallel programming will provide a tangible speedup at larger input file sizes and larger filter sizes. And therefore is worth it provided the inputs meet these criteria.

* At what data set sizes and filter sizes does the parallel program perfrm well?

The parallel program has a better performance than sequential when the file size is larger than 100000 numbers and when the filter size is larger than 9.

* What is the maximum speedup and how close is this to ideal?

The maximum speedup observed was 3.14. the ideal for four cores would be 4. This is quite close to ideal speedup. The difference from ideal is caused by overhead due to creating the threads, communicating answers among threads, memory hierachy issues and inherent computayional dependencies related to span.

* Optimal number of threads?

Conclusions

From the data and analysis above, the following conclusions can be made:

By increasing the input file size, the process time increases. The process time increases more rapidly for

The sequential program compred to the parallel program due to the use of multithreading.

Incerasing the filter size linearly increases process time. The gradient of increase is greater for sequential

program than the parallel program.

By changing the sequential cutoff the process time of the parallel program should have a curve with a

dip where the dip is the “goldilocks” zone where the program runs the sequential process on a data set

that is too small thereby creating too many threads and causing “thrashing” or too large where it runs

the sequential process too soon thereby wasting potential speedup.

The sequential program outperformed the parallel program for small input file sizes and small filter

sizes. Therefore the parallel program should be selected over the sequential program in this case where the input file size is larger than 100000

Improvements to the experiment can be made by:

* Achieve a working automater to reliably test in order to get:

1. More data points
2. Test a larger variety of variables
3. Automatically graph results

* Use a more modular approach – using separate classes for every process. This would also make testing easier
* Test on different machines with a different number of cores and a different processor.