**Concurrent Systems**

Assignment

Student Number: 15321767

**Introduction**

This assignment entails implementing one searching algorithm and one sorting algorithm both serially and in parallel. I will attempt to parallelise the linear search algorithm as well as the quicksort algorithm and discover whether it is worthwhile to parallelise these algorithms. The library that I am using is OpenMP which provides functionality for creating and using threads in C programs. I shall be using this library to alter the implementations of the serial versions of these algorithms to create an improved performing parallel version of them.

**Algorithms**

**Linear Search**  
I chose the linear search algorithm as it is an algorithm that is fairly simple to implement as well as having easy access to expose the algorithm to multi-threading capabilities to increase run-time. The algorithm works by simply iterating through an array until the end, comparing at each iteration whether we have found the piece of data we are looking for. This gives this algorithm these performance times:

Worst-case: O(n) (element is the very last element in the list, meaning we have to iterate  
 n times to reach it)

Best-case: O(1) (element is the very first element in the list, meaning on the first comparison   
 we find the element)

Average-case: O(n/2) (element is positioned somewhere in the list between the start and   
 the end, with equal probability of being in any position,  
 this gives us a mean of the element being in the middle  
 of the array giving us n/2)

**QuickSort**I chose the quicksort algorithm for attempting to parallelise it as the quicksort algorithm is an example of an algorithm that is a divide-and-conquer algorithm, that is, the algorithm functions by positioning a pivot in the array such that the elements to its left are less than the pivot and the elements to the right are greater than the pivot, these two groups of elements are then subsequently both implementing this for the left array and the right array. This process repeats until the array is decomposed into single elements all in their respective places.  
**Example:**

[3,1,5,2,7,9,8,4]

[3,1,2]4[5,7,9,8]

[1]2[3]4[5,7]8[9]

[1,2,3,4,5,7,8,9]

Worst-case: O(n^2) (the pivot chosen is the largest or smallest in the sub-array causing each   
 sub-array to be of size n-1 creating a chain of n-1 calls)

Best-case: O(nlogn) (each partition sub divides the array into two equal parts creating a call   
 tree of log2N with each call having a constant overhead)

Average-case: O(nlogn) (in each recursive call the pivot has a random rank that will  
 50% of the time be between the 25% and 75% mark of all values

which will split its sub arrays into ¼ and ¾ meaning it will at most go to log4/3N times)

**Production of Sample Data**

The sample data used for my program is generated by the program generateArraySort.c   
This program generates 100,000,000 integers and places them into a file. All versions both parallel and serial of my linear search and quicksort algorithm use this as the data source. These programs take one parameter which is the sample size of data to be used for that instance.  
For example serialsort 100 // runs serial sort with n = 100

**Linear Search**

**Serial**

My serial implementation of linear search is quite trivial. The program takes an integer argument after the program name as its sample size n. The program processes the integers from the file and places them in an array. A random number is generated in the range of the input and that number within the array is the number to be searched for. The main program then simply calls the linear search function which iterates through a for loop comparing each value with the value we are looking for. This implementation will averagely take (n/2) iterations before finding the number and terminating. The time taken for the serial approach to work grows in accordance with the time complexity.

**Parallel**

The obvious place for us to expose this algorithm and parallelise it is within the for loop that iterates over the entire array until it finds the number we are looking for. We can parallelise this for loop over m threads so that each thread will iterate over n/m integers, each checking if the requested integer is within its subsection of the array.

For example:  
t1: for(i=0;i<(size/NUM\_THREADS\*thread\_number); i++) //iterates  
t2:for(i=(size/NUM\_THREADS\*thread\_number-1);i<(size/NUM\_THREADS\*thread\_number); i++) //iterates  
etc..

Although when executing a serial implementation we can simply return from the function when we found our sought number, through a parallel implementation we cannot immediately return as this will cause an invalid branch for Open MP as we are trying to terminate a slave branch that was created from the master branch that simultaneously created other slave branches. We must wait for all slave branches to iterate over their section of the data first. Instead if we match with the data provided we simply store the index that it was found at in a variable and return that after all threads terminate. This piece of data is shared by all threads.

The data that we are using for this parallelisation is shared data, the array will be accessed by all threads which is not a hazard as we are never writing to this array, simply using it for data access for comparison, so we do not explicitly have to specify any private variables or critical sections for this parallel implementation. The variable “found” which is a placeholder for the index as to where the data was found in the array can be shared as we only need 1 index to be returned of the location of the element in the array, that is we don’t necessarily mind if that index is overwritten by another thread that has also found our data in its sub array.

The index for the individual threads that will be used to iterate will be private to those threads.

**Serial Vs Parallel Results  
Serial**

For the serial version of the program, a significant runtime only appears after 100,000 numbers are inputted which gives a runtime of 0.046 seconds.   
For 1,000,000 : 0.46 seconds  
For 10,000,000: 4.95.  
This is all expected as 4.95/10 == 0.46 seconds

**Parallel**

For the parallel version of the program, for smaller values the parallel version was more expensive (timewise) to execute. This is due to the cost of beginning a thread being more expensive than the CPU being able to iterate through values.

For: 100,000:0.024 seconds  
For:1,000,000: 0.249 seconds  
For:10,000,000: 2.53 seconds

The parallelisation achieved a 50 % increase in efficiency

**Quicksort**

**Serial**

The serial implementation for quicksort that I created takes a parameter after the program name similarly to my linear search algorithm, this parameter sets the size of data that we are to be using. We then process this data into an integer array to be sorted. I call the function quicksort from my main which then proceeds to sort the data. Quicksort recursively calls itself twice from within the function for the two sub arrays it has to the left and right of the pivot. It does this until each sub array is of size 1, where it terminates leaving us with an array that is in a sorted order. The serial implementation has a call stack somewhat of a binary tree order, that is the first instance will call 2 more sorts(left, right) which will subsequently call 2 more each (so long as the sub-array>1) (leftleft, leftright, rightleft, rightright)

**Parallel**

The obvious place to parallelise this algorithm is when the algorithm creates 2 sub arrays to be sorted to the left and right of the pivot. We want to thread the sub arrays to a reasonable degree, so much as to not lose efficiency by creating all these threads, a naïve solution would have each call create a thread, but as already mentioned these calls only terminate when the size of the sub array is 1. That means if we were to use a thread per call without restriction we would be creating a thread for each element in the array, that is we would be costing ourselves C \* n seconds where C = cost (in time) to create a thread.

A better solution would be to parallelise our sorting to a degree where we are creating threads up to the number of threads we have available to not lose time on waiting for threads to terminate before we can initialise new ones. That is, we should only use the amount of threads that we can have alive simultaneously for this algorithm. If we have 8 threads able to be alive at once we should parallelise our two left and right calls 3 times to achieve 2^3 = 8 threads of execution for sorting the data. After that we should not attempt to parallelise anymore and allow each thread to serially sort any subsequent sub arrays generated by the algorithm.

The array that is to be sorted is shared across all the threads but the indices that the data will be extracted from and that we will be sorting, will be different. This

**Serial Vs Parallel Results**

For the serial version of the program, a significant runtime only appears after 100,000 numbers are inputted which gives a runtime of 0.016 seconds.  
For 1,000,000: 0.25 seconds  
For 10,000,000: 2.5 seconds  
This is also expected 2.5/10 == 0.25

For the parallel version of this program I struggled to get results. I wanted to attempt to use this:

#pragma omp parallel sections

{

#pragma omp section num\_threads(NUM\_THREADS/2)

{

quicksort(array, low, index-1);

}

#pragma omp section num\_threads(NUM\_THREADS/2)

{

quicksort(array, index+1, high);

}

}

So as to parallelise it as I stated before but I had trouble with num\_threads()

**Execution Environment**

1 process 4 threads.

**Test Machine Physical Environment**

Intel core 17-4510U CPU @ 2GHZ 4CPU’s

8GB RAM

**Compiler**

MinGW32-gcc 5.3.0-2

**To run tests**

serialsearch 100000