CS3014 Assignment Report

Andrew Donegan, 15315962

Introduction

All my files use a text file of 10 million integers as a source to search through and sort. I have an extra file called “createItems.c” ,which needs to be run once before using my search/sort files, that creates the text file of integers.

linear.c is my serial implementation of linear search.
linearPara.c is my parallel implementation of linear search.
mergesort.c is my linear implementation of merge sort.
mergeParallel.c is my parallel implementation of merge sort.
constants.h is my global constants file
makefile includes all the commands from your assignment brief.

System Environment: (LG12 Ubuntu machines)

OS: Ubuntu
CPU: Intel i7 4770
Cores: 4
Threads: 8
Cache: 8MB per processor
RAM: 8GB
Hyperthreading: Yes
Compiler: gcc version 5.4.0

Timing testing:

All algorithms tested on 10,000,000 Items in a text file. Each algorithm ran 3 times. Time values acquired using sys/time.h library. All time values in seconds. Serial values do not change for different number of threads.

Searching

Algorithm:

I did Linear Search. I chose this algorithm as it is the one I am most familiar with and have used the most in the programming career. It is a simple algorithm which uses a for loop to iterate through elements given to search for a value.

Time Complexity:

θ (n) for worst case with serial implementation. N is the number of items to search through. The worst case is when the value that you are searching for is right at the end of the set of values.

Parallelising:

Using openmp to parallelise this search algorithm was quite simple. I added “#pragma omp parallel for” above the for loop that searched for the randomly generated number.
Doing this will spread the for loop iterations over the number of threads specified. So if 2 threads are used, one half of the for loop will be executed on the first thread while the other half will be executed on the other thread at the same time.

Parallelising this made it noticeably quicker, which can be seen in the timing table. However, as can be seen in the table, once you go over 4 threads (from what I tested), it actually gets slower. This is because each thread will only be searching through a small number of values. The time needed to create the thread and execute its part of the code takes longer than using less threads.

Testing:

Table for Program Time: Using sys/time.h library

Number of Threads	1	2	4	8	100
Serial (Seconds)	0.0262, 0..0285, 0.0278	-	-	-	-
Parallel (Seconds)	0.0265, 0.0258, 0.0275	0.0152, 0.0114, 0.0138	0.0094, 0.0091, 0.0102	0.0118, 0.0148, 0.0135	0.0128, 0.0103, 0.0128

Sorting

Algorithm:

I chose to do merge sort for my sorting algorithm. I chose this because it is the sorting algorithm I have used before in C. I also chose it because I was not sure how the algorithm would react to being parallelised, as I wasn't sure how recursion interacts with multiple threads.

Time Complexity:

θ(n * log n) for best and worst case. N items are iterated through log(n) times. This cannot be made quicker as the whole list needs to be sorted.

Parallelising:

To parallelise this algorithm I needed to split up the sorting parts between threads. I used openmp sections to parallelise each sorting section. However since this is a recursive call an excess of threads gets created each time the function gets called again. This causes the program to actually be slower than the serial implementation.

I believe the only way to make merge sort parallelizable is by using an iterative method rather than the recursive method I used. The only part of the code that can be parallelized is sorting the array sections so there are no blocks that can be split up to parallelize.

The merging part of the algorithm has to wait for the sorting of the array sections to be completed before it can complete. It is dependent on the sorting parts to complete before it can begin merging the 2 sorted arrays together.

The array sorting is split into 2 and then having a section for each sorting segment means each thread gets an equal share of the array to sort.

From the timings below it shows that using the recursive implementation of merge sort is not parallelizable. However, when using 1 thread the performance decreases drastically, using a thread count higher than 1 provides a similar output that is much quicker than just 1 thread. When only 1 thread is used, it means nothing runs in parallel so each section has to wait for the previous before it can run.

Table for Program Time: Using sys/time.h library

Number of Threads	1	2	4	8	100
Serial (Seconds)	2.7794, 2.6994, 2.7083	-	-	-	-

Andrew Donegan, 15315962

Parallel (Seconds)	9.4658, 9.6309, 9.6141	5.3916, 5.3533, 5.3069	5.2366, 5.4229, 5.3667	5.5284, 5.3787, 5.2783	5.3332, 5.3440, 5.3854

Conclusion

Parallelising linear search greatly increased performance, however, parallelising merge sort reduced performance before of recursion.

This assignment gave me a better understanding of parallel programming and how threads and running certain parts of code interacts with each other. I learned how to use openmp and how it affects a program when used. I learned that parallelising recursive functions make it more inefficient. I found it useful in implementing what we learned in the lectures.

Andrew Donegan, 15315962