Paralell Implementation of Mergesort

High Performance Programming

Author:

Fredrik Gustafsson

Uppsala

March 21, 2019

1 Introduction

Being able to sort data is useful to, as an example, reduce search complexity. A naive way of sorting a list would be to compare every element to every other element in the list. A problem with such an algorithm is scaling since it would have $O(n^2)$ time complexity. A better solution could be a divide and conquer algorithm such as merge sort. However even then we will be limited by the hardware which the code will run on. One way to improve the performance is to parallelise the code.

2 Theory and Implementation

2.1 Theory

Merge sort works by recursively split the array into two halves until each array has a size of one and then merge the arrays back in a sorted order.

2.2 Implementation

Following the reasoning above one can write the following pseudocode.

Data: Array of unsorted numbers
Result: Array of sorted numbers
if *left less than right* **then**
    calculate mid point;
    call self for left half;
    cal self for right half;
    merge both halves;
end

Algorithm 1: Merge sort Pseudocode

Following the pseudocode at least two functions were needed, one which recursively splits the list and one to merge them. In addition to the merge sort program one also needed a way to generate an input array, either reading or creating. Since we were more interested in the complexity of the algorithm and how parallelisation will affect the performance we generated our own array in the code. In the end the program consisted of a main function, the splitting function (parallel and serial), the merging function, a function to generate random double numbers, and a function to print the array.

2.3 Parallelisation

Since we were interested in exploring how parallelisation of the code would affect the performance we needed to find parts of the code to parallelise. The recursive calls for the left and right halves of the array were independent from each other. They did share a variable, the midpoint but it was only read and not changed which made the chances of race conditions virtually nonexistent. Hence it was a part of the code which could be effectively parallelised. Since we always split the array at a midpoint the load balance for each thread should be similar. However the function call is recursive and we wanted to limit how many threads are used hence we needed a way to switch back to the serial implementation which was done with a `if` condition.

In this case OpenMP, which uses the shared address space model, was used to parallelise the code. OpenMP uses threads which have access to global data while also having their own private data on stack. In addition to this OpenMP uses compiler directives instead of functions to do the parallelisation since it's a more high level approach compared to e.g. Pthreads. This can lead to a decrease in performance but make the implementation easier.

3 Optimisation

When doing the optimisation of the code it was executed five time and the timing being done by the time command. For measuring individual parts of the code a start and end time was used. No matter which compiler flags or other techniques tested the program was compiled with the `-fopenmp` flag to ensure that it wouldn't affect the end result. For each execution the length of the array was set to 2000000 with a constant seed. The code was executed on Unbuntu 18.04.1 LTS on Windows 10 using a Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz CPU, compiled with GCC 7.3.0.

3.1 Serial Optimisation

Before trying to do the parallel optimisation the code was optimised for serial use, one thread. The first thing tested was different optimisation flags with the result presented in table 2 below.

Table 1: Real, User and Sys time as reported by the time command for different optimisation flags n = 2000000

	-O0	-O1	-O2	-O3	-Ofast
Real	0.530	0.316	0.322	0.317	0.314
User	0.500	0.281	0.281	0.281	0.281
Sys	0.016	0.031	0.047	0.047	0.031

	-Ofast -funroll-loops	-Ofast, -march=native	-Ofast, -march=native, -funroll-loops	-Ofast, -mtune=native	-Ofast, -mtune=native, -funroll-loops
Real	0.319	0.314	0.324	0.322	0.316
User	0.297	0.281	0.297	0.297	0.281
Sys	0.016	0.031	0.031	0.031	0.031

After choosing the best combination of compiler flags from table 2 some attempts at micro-optimisation of the code was done. Since a test array had to be filled with random numbers it might benefit from loop unrolling the result of different amount is shown in table **??** below.

Table 2: Time required to fill n = 2000000 array with numbers for different amount of loop unrolling

	i+=1	i+=2	i+=3	i+=4	i+=5	i+=6	i+=7	i+=8	i+=9
Time	0.031250	0.031250	0.031250	0.031250	0.31250	0.031250	0.031250	0.031250	0.031250

Since loop unrolling did not change the time used to create the array it was removed. Finally the parallel optimisation was done. The first thing investigated was the speedup from different amount of threads used when parallelising the sorting part which is shown in figure 1 below.

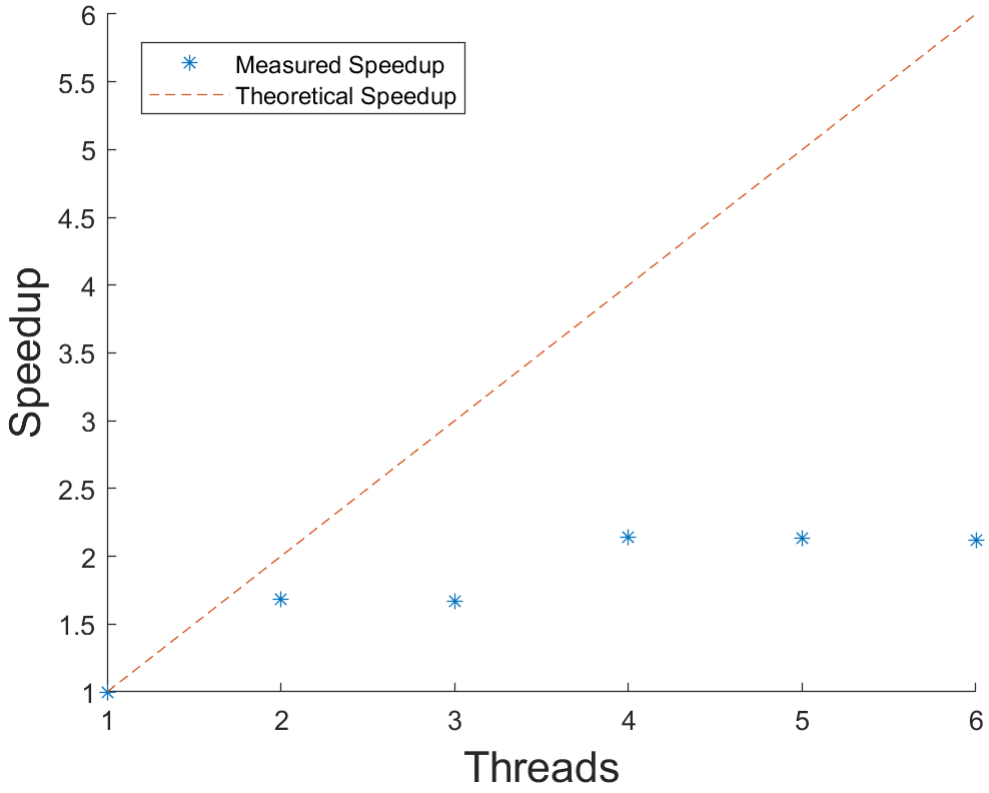

Figure 1: Meassured and theoretical speedup for different amount of threads

After which the affect of adding parallelisation to the creation of the array was investigated. However such a change turned out to decrease the performance which might be due to the cost of synchronisation of the threads outweighing the benefit. Finally the time complexity was investigated when compiling with `-Ofast` and `-march=native` and using OpenMP to parallelise the sorting. This was done for the serial case, two threads and finally four threads, the result together with a n^2 and $nlog(n)$ curve is shown in figure 2 below.

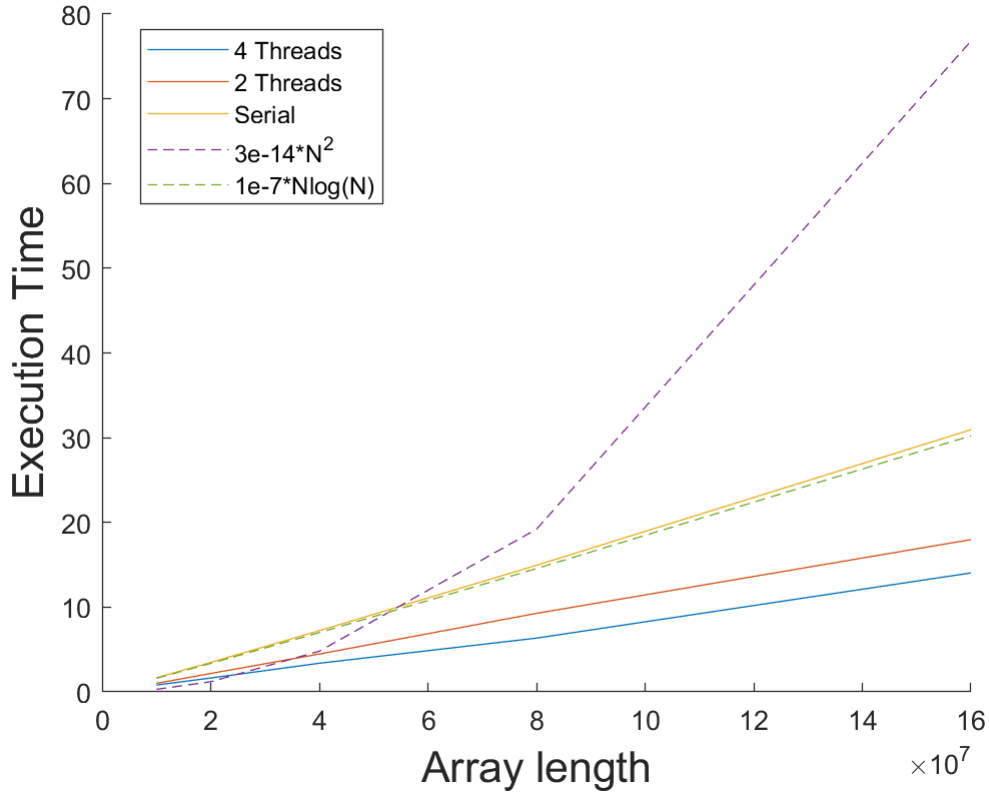

Figure 2: Meassured time complexity for different amount of threads together with a nlog(n) and n^2 curve

4 Discussion

As we can see the best combination of optimisation flags was `-Ofast` together with `-march=native`. Using `-march=native` will make the code less portable since it optimises the code for being run on the machine on which it is compiled on. However at the same time a program which sorts a random array would not be of much use on its own. Instead it might be part of a larger program or being used internally and hence using the `-march=native` flag can be justified. Since `-Ofast` would increase both compiling time and the size of the executable the result of this was measured. The compiling time was under one second and the file size is 19kb hence it was seen as not being a problem. Since `-Ofast` does not introduce errors in the sorting while not affecting compile time or executable size the usage of `-Ofast` to improve the performance can be seen as justified.

The best speedup was achieved with four threads as figure 1 shows. Since the CPU used has four logical cores it makes sense that more threads than four would not improve the performance. Using four threads should then hopefully minimise synchronisation for swithching between threads while still utilise all cores. A reason why the achieved speedup is lower than the theoretical one might be that not all of the code is parallelised. Populating the array is done in serial as is the merging. Another explanation for the achieved speedup might be the need to synchronise the threads before merging.

The reason why merge sort is a better algorithm than a naive sorting algorithm is the time complexity being $O(nlog(n))$. As figure 2 shows the implementation had a $O(nlogn)$ complexity compared to the $O(n^2)$. This shows that although we might not be able to parallelise all code, synchronise the threads and having to create the array we still achieved the desired complexity.

5 Conclusion

The best performance was achieved when compiling the merge sort based program with `-Ofast` and `-march=native` and running on 4 threads.

Appendix

Running the program

After compiling the program using the makefile with the command `make` the program is executed with the command `mergesort N T P` where `N` is the size of the array `T` the number of threads and `P` is if one wants to have the array printed in the terminal or not, 1 to print.