High Performance and Parallel Computing 1TD064 - Parallel Quicksort

Tomas Grahn

July 2024

1 Introduction

One of the foundational sets of algorithms in Computer Science is sorting algorithms, and for good reason. Sorting underlies many higher level applications and functions. From ordering a spreadsheet to enormous database queries, sorting shows up time and time again. Given recent limitations in single core performance the community of Computer Scientists must look to parallel algorithms in order to efficiently use our resources. Quick sort is a notoriously efficient algorithm with a methodology that can be adapted for parallel processing. This project will explore it's implementation and optimisation.

2 Problem

The problem of sorting is too wide to explore in it's entirety so we will place some limitations to enable exploration and optimisation. These limitations are important to keep in mind as they often dictate which optimisations can and should be applied and so these results should be applied with thought. For this project only positive integers will be sorted. Furthermore the experiments will use varying problem sizes and ranges to explore behaviour. The goal is to implement and optimise a sorting algorithm that can efficiently sort a large list of integers while using multiple cores.

3 Solution method

3.1 Algorithm - Parallel Quicksort

In the following explanation a sub-list will refer to what is passed into global sort and a bin will refer to locally sorted section of the sub-list.

1. Split & Sort
   To begin the list is split into equal sized bins. The number of bins is equal to the number of processors available. Each processor then sorts their

respective bin in parallel. Furthermore a list of bin_infos containing the indexing information of each sorted bin is created. The list and bin_infos are then passed into global sort.

2. Global Sort
   This step aims to merge the sorted bins into a single sorted list. It accomplishes this by continually splitting the bins around a pivot and then merging the new bins either side of the pivot.

   2.1 Merge odd bin
   In order to ensure the sort bins step can merge together two bins each without edge cases, a check for an odd number of bins is performed. If there is an odd number of bins then the final bin is merged with second to last bin and sorted. This ensures an even number of bins meaning we can always group 2 bins.

   2.2 Select pivot
   A pivot is selected for the current sub-list. This pivot aims to split the sub-list such that the upper and lower lists contain equal elements.

   2.3 Create lower and upper bins.
   Each bins set of values less than the pivot are moved to a new list. The same is done for the upper values. During this process information for 2 bins are combined into a data structure called bin_info. This data structure maintains information regarding the indexes of the bin with respect to the new list. This data is stored for the following merge and sort bins step.

   2.4 Merge and sort bins
   Using the bin_infos information each 2 bins of lower and upper values are sorted such that now both their values less than the pivot and greater and in new sub-lists and sorted.

3. Recursively perform 2 until fully sorted
   After the sort bins step there is a lower and upper sub-list with bins of partially sorted values. The lower and upper sub-lists are then passed into global sort again along with the bin_infos for the newly sorted bins. This continues until only two bins are merged and hence the sub-list is fully sorted.

3.2 Optimisation

3.2.1 Where to optimise

In order to begin optimising the algorithm it is important to initially analyse how long the program spends doing each operation. This initial timing was done on 100,000 positive integers uniformly distributed between 0 and 100,000. Additionally two threads were used with the median of first bin for the pivot and bubble sort for the initial sort and merging.

Total (s)	Initial Sort (s)	Left Half Merge (s)	Right Half Merge (s)	Other (s)
5.72	3.98	1.73	1.62	0.01

Table 1: Timing in seconds sorting 100,000 integers

Table 1 shows the total time is currently dominated by the initial sorting time and merging the subsequent halves. Together these make up over 99% of the time in the program. Hence, they will be targeted for optimisation.

3.2.2 Sorting

The following algorithms will be explored to improve the initial sorting and merging.

1. Bubble sort
   While bubble sort is simple and quick to implement it does not have very good sorting properties with an average case of $O(n^2)$. It will be included in the experiments to provide a baseline for improvement.

2. Quick sort
   Quick sort while being more complex to implement has far a far better average case of $O(n * log(n))$. It's recursive nature can however lead it to be slower than other algorithms for shorter lists due to excessive function calls. This was demonstrated in Lab 6 - Task 1 with merge sort which has a similar problem.

3. Quick sort + Insertion sort
   To rectify quick sort's slowness for small array's it can be supplemented with another sorting algorithm for small lists. In this case Insertion sort will be used.

4. qsort from stdlib.c
   Finally qsort from the c standard library will also be compared. This contains additional optimisations such as; being non recursive, choosing a better pivot and some memory optimisations.

3.2.3 Splitting the merge

Initially both the initial sort and merging of bins operation were done using a sorting algorithm. However there are some properties of the bins that can be used for a much more efficient merging operation. It is know that when two bins are being merged there are two sorted halves. This is exactly that case of the 'merge' section of merge sort. Avoiding the entire division and many of the merge steps should allow for a very efficient merge operation. However this does still require an additional memory allocation for the a copy of the original list.

3.2.4 Pivots

In Table 1 there is some discrepancy between the time it takes to complete the left and right half. This discrepancy can be minimised by ensuring a more even split between the left and right sub-lists. In order to accomplish this a better pivot must be used. Three options are outlined below:

1. Median of the first bin
   In this method the median of the first bin in the sub-list is used. This method is quick to sample however can lead to unbalanced pivots if the first bin has a median that is not representative of the other bins.

2. Mean of all medians
   In this method all medians from each bin are averaged to construct a value that should balance across the bins. Using the mean can however over-expose the pivot to extreme medians within a bin leading to a inefficient pivot.

3. Mean of two middle most medians
   This method takes the median of each bin, sorts them and then takes the mean of the central 2. This method should provide a balanced pivot but takes longer to sample compares to the other two methods. Whilst longer than the other two compared to the larger sorting and merging operations it is not significant and the gains in a more efficient pivot are expected to be worthwhile.

3.2.5 Stack versus the heap

Due to the structure of memory, stack allocations are generally faster than heap allocations. For this reason it is preferable to use stack allocations where possible. Hence, for additional memory allocations, such as in the merge operation or creating the temporary upper and lower sub-lists, stack allocations will be used. However for sorting larger sets of integers a separate implementation using allocated memory will be used to avoid over-allocating the stack.

3.3 Implementing Parallelisation using OpenMP

To parallelise the program OpenMP was used. As previously explored, the two key operations in the program are the initial sorting and the merging and sorting of the bins. Enabling these operations to run in parallel is what allows this method to outperform serial sorting methods. Parallelising the initial sort was accomplished by adding a single *omp parralel for* pragma to the corresponding loop. The merging operation was parrallelised using two tasks one for the lower list and one for the upper list. Within each of these tasks a parallel for loop was used for the merging operation. The independent nature of the sorting of the sub-lists means that it is also possible to continue recursively performing global sort as soon as the merge operation of the given sub-list is completed.

For this reason it is necessary to enable OpenMP's nested parallelism to ensure the recursive tasks are parallelised.

4 Experiments

4.1 Methodology

The following results were generated on arrhenius.it.uu.se which has Intel Xeon E5520 with 16 available cores. After each experiment the resulting list is compared to the same list sorted by qsort from stdlib.c to ensure correctness. For all experiments a uniform distribution sampling from 0 to number of integers being sorted is used. This is done to ensure a similar probability density function between varying sample sizes. In the tables results below, total refers to the total time for the program to complete, initial sort is the time to complete the initial sorting of the n bins, the left and right merge is the time for the respective half of the list to finish merging and thus be sorted, other is the remaining time not taken by the initial sort and longer of the left and right merge. The program was compiled with the O3, ffast-math and -march=native flags to ensure the compiler performs as many optimisations as possible.

4.2 Results

4.2.1 Pivots

To explore the pivots list sizes of 1000, 1000000 will be compared. 8 threads will be used for all tests as well as bubble sort for the initial sort and merging.

List Size	Total (s)	Initial Sort (s)	Left Merge (s)	Right Merge (s)	Other (s)
1000	0.0081	0.0005	0.0039	0.0035	0.0036
1000000	63.8722	31.1449	32.5655	32.72	0.0006

Table 2: Results from using median of the first bin

List Size	Total (s)	Initial Sort (s)	Left Merge (s)	Right Merge (s)	Other (s)
1000	0.0060	0.0004	0.0029	0.0038	0.0018
1000000	64.0600	31.2381	32.7451	32.308	0.0076

Table 3: Results from using mean of median of all bins

List Size	Total (s)	Initial Sort (s)	Left Merge (s)	Right Merge (s)	Other (s)
1000	0.0070	0.004	0.0043	0.0038	0.00226
1000000	63.7861	31.15	32.5269	32.5911	0.0405

Table 4: Results from using mean of middlemost bin medians

The pivot results reveal that there is not much deviation in the total runtime for the either lists sizes given different pivots. This is likely due to the uniform distribution being used, there is not much likelihood of a very unbalanced split that would effect performance. For example in the smaller 1000 case, split across 8 threads each bin has 125 integers. This bin of 125 is unlikely, with the given distribution, to lead to an unbalanced median. Another observation is that for the smaller lists the other cost is significant making up almost half of the total time. This is to be expected as there are some constant costs such as establishing threads, memory allocations and establishing the bin_info's. Moving forward the third strategy will be used as it did not add significant overhead and albeit small did show improvement in the difference between left and right merge times indicating a more even split.

4.2.2 Sorting

To explore sorting methods list sizes of 50000, 1000000 will be compared. 8 threads will be used for all tests. The third pivot strategy will be used.

List Size	Total (s)	Initial Sort (s)	Left Merge (s)	Right Merge (s)	Other (s)
50000	0.2335	0.0911	0.1300	0.12499	0.0123
1000000	63.7861	31.15	32.5269	32.5911	0.0405

Table 5: Results from bubble sort

List Size	Total (s)	Initial Sort (s)	Left Merge (s)	Right Merge (s)	Other (s)
50000	0.0104	0.0012	0.5954	0.04896	0.0032
1000000	0.117	0.0197	0.0849	0.0849	0.013

Table 6: Results from using quick sort

List Size	Total (s)	Initial Sort (s)	Left Merge (s)	Right Merge (s)	Other (s)
50000	0.0082	0.0012	0.0528	0.0507	0.0018
1000000	0.1055	0.0176	0.0737	0.0733	0.0141

Table 7: Results from using quick sort with insertion sort

List Size	Total (s)	Initial Sort (s)	Left Merge (s)	Right Merge (s)	Other (s)
50000	0.0074	0.0017	0.03659	0.0397	0.0018
1000000	0.0856	0.0298	0.0431	0.0447	0.0110

Table 8: Results from using qsort

Comparing Table 5 with Table 6 the benefit of quick sort is very evident with a reduction in total run time in the 1000000 list case from 63.7861 seconds to 0.117 seconds. This difference is coming for significant reductions in both the initial sort and merge. This is to be expected as the sorting is impacting both these operations. The difference between the different version is quick sort is only slight with the insertion sort version and qsort performing slightly better. Comparing Table 7 with Table 8 shows that the insertion sort version performs better for initial sort but not for the merge. This indicates that the qsort may have some additional optimisations for smaller lists as this is primary difference between the initial sort and merge operations. Moving forward the quick sort with insertion sort will be used as the merge method explored in the next set of results nullifies the qsort's better merge timings. It was interesting to see that the stdlib qsort was outperformed by a standard implementation of quick sort with insertion sort. This may be due to the algorithm being built to be more general or a better break point having been selected for when to switch to insertion sort.

4.2.3 Merging

To explore the new merging method list sizes of 50000, 1000000 will be compared. 8 threads will be used for all tests. Quick sort with insertion sort will be used as well as the third pivot strategy.

List Size	Total (s)	Initial Sort (s)	Left Merge (s)	Right Merge (s)	Other (s)
50000	0.0102	0.0012	0.0054	0.0037	0.0035
1000000	0.399	0.188	0.0115	0.0118	0.0094

Table 9: Results from using merge

Comparing 9 with 7 we see that there is a significant reduction in the merge timing. The new method is about 7x faster for the same list. This degree of improvement was surprising but shows how many operations can be saved by using the existing properties of the bins. This method will be used moving forward despite having an increased memory cost.

4.2.4 Scaling up

Having performed some significant optimisations larger lists will be explored. In order to explore larger lists the implementation had to change stack allocations involving the sub-lists to heap allocations. 8 threads will be used for all tests. Quick sort with insertion sort, with the separate merge method as well as the third pivot strategy.

List Size	Total (s)	Initial Sort (s)	Left Merge (s)	Right Merge (s)	Other (s)
50000	0.0076	0.0012	0.0033	0.0044	0.0019
1000000	0.0358	0.0191	0.0092	0.0094	0.0066
2000000	0.0695	0.4166	0.1567	0.1598	0.0012
5000000	0.1582	0.0921	0.368	0.0355	0.0292
100000000	3.392	1.935	0.7916	0.5744	0.6650
400000000	13.5856	8.5000	2.8700	3.7600	3.4400

Table 10: Results from larger lists

Comparing the first two entries of Table 10 with 9 it is surprising to see that the heap allocated method performs faster. Furthermore the performance of the algorithm has clear improvements over the single threaded case. Running stdlib.c qsort on the 100,000,000 case takes 25.3011 seconds while the current implementation performs the same sort in 3.392 seconds. Table 10 demonstrates the algorithms ability to perform large sorts in reasonable time frames. Table 10 also demonstrates that there is a growing element in the other time, with it consistently taking around the same time as the merge. This is expected to primarily be the cost involved with locating the pivoting index in the bin before the merge step.

4.2.5 Multi-threading

To explore sorting with various threads methods a list of 100,000,000 was sorted using quick sort with insertion sort, the separate merge method and the third pivot strategy.

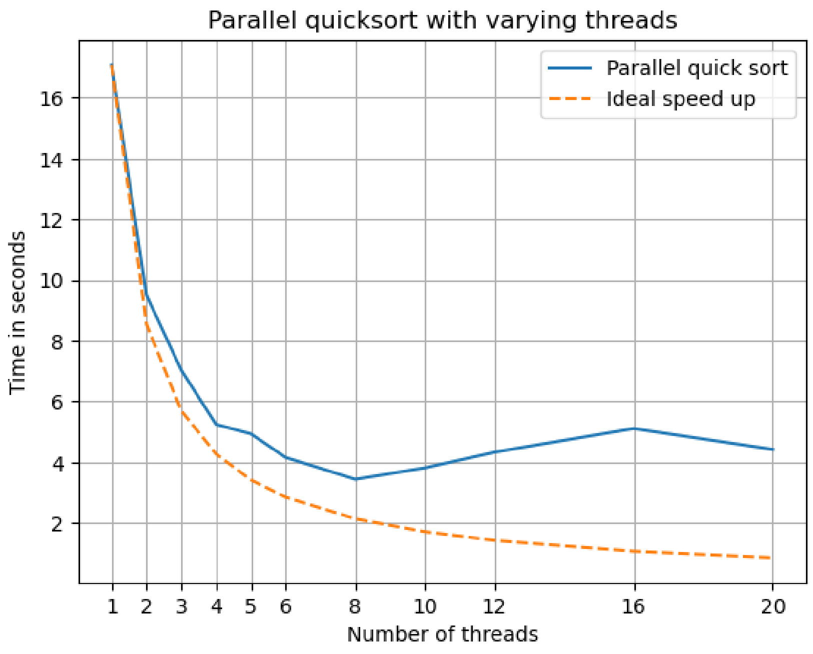

Figure 1: Timings for sorting a 100,000,000 integers with varying threads

Graph 1 shows that on the 16 core processor the algorithm achieves it's maximum speed at 8 threads. After which there is an increase in processing time. Furthermore the gap between the ideal speed up and timed performance increases as the number of threads increases, until 8 threads. This is expected as overheads from using additional threads stack up, memory and resources become in contention.

5 Conclusions

Overall this implementation of parallel quick sort proved successful. Given the 16 core processor the final optimised algorithm outperformed the stdlib's qsort by 746%. In order to accomplish this the use of a better sorting method was essential as it dramatically reduced the algorithm's initial sort time. Furthermore an efficient merging strategy also provided a significant performance improvement. While the pivot optimisations had little impact, this may not be the case given a different distribution of elements to sort.

The current algorithm could be further explored in a number of ways:

1. Investigate performance on differing random distributions
   As the distribution of data being sorted used can make a significant impact on performance exploring additional distributions would give a more holistic view of performance.

2. Additional types
   The type of data being sorted can have an impact on the logical units available to the machine and can lead to differing performance between types. For this reason it could prove interesting to explore additional data types.

3. Reducing Other time
   Table 10 demonstrated a growing other time. As this time has not been investigated and optimised it would make for a good starting point for additional optimisation.

4. Insertion sort break point
   The optimal point at which using insertion sort over quick sort is highly variable and can depend on many factors. Only 20, 40 and 100 were quickly compared in order to find a suitable point. Rigorously expanding this search could provide some small performance gains.

5. Resource contention
   The difference between the optimal case observed in graph 1 indicates there is a fair amount of resource contention that could potentially be investigated and optimised.

6. OpenMP scheduling and chunking
   OpenMP provides additional pragmas flags for adjusting the strategy used for assigning work and chunking loops. These were not explored in this project and could provide some additional time savings due to better workload distribution between cores.

There is also scope for a number of larger changes that could yield additional improvements:

1. Vectorising operations
   The current algorithm does not make use of vectorising lower level operations. Utilising SIMD instructions can provide large efficiency gains if used effectively. One area of the algorithm this could prove useful is in the determination of the pivot index. Currently the pivot index is determined by finding the first element greater than the pivot, Given a perfect pivot this will take $bin\ length/2$ comparisons. Given a 4 lane simd unit thi could effectively be reduced to $bin\ length/4$ comparisons, halving the time.

2. Sorting networks
   For sorting short lists sorting networks are a very efficient approach. As with use insertion sort with quick sort for the shorter lists, a similar approach could be taken with sorting networks for improved performance.

3. Other high level algorithms
   This project only implements a single parallel sorting algorithm and many such algorithms exist. If the goal is to find an optimal solution for sorting

a large list using multiple cores then other parallel algorithms should also be explored, optimised and compared.

6 References

Parallel Quick Sort

1. Based on the provided QuickSort.pdf, The pivot strategies were also taken from the final slide.

Bubble sort implementation

1. Provided bubble sort implementation from Lab 6 - Task 1 was used.

Merge implementation

1. Provided merge sort implementation from Lab 6 - Task 1 was adapted.

Quick sort implementation

1. https://en.wikipedia.org/wiki/Quicksort
2. https://stackoverflow.com/questions/7198121/quicksort-and-hoare-partition

Insertion sort implementation

1. https://www.geeksforgeeks.org/insertion-sort-algorithm/