High Performance computing and Sorting Algorithms

Meluleki Dube

Department of Computer science

University of Cape Town

March 26, 2018

1. Introduction

Sorting of data items is a fundamental need in computer science. It is vital process required and useful in multiple use cases. For example for using Uber, when we book a ride the application needs to know which drivers are close to you so it can suggest those for you. This problem of knowing which uber drivers are close to a particular location can be simplified by sorting the locations of these drivers by how much distance they are from a particular user. From there we cab start showing the top n number of drivers to suggest to the user. Another example is when looking for a house to rent on renting sites. You search for a location to look for houses from but now the results are just all over the place in terms of price and you wish you could have a way to show the results in ascending order of price which will be helpful to you. For this to be achieved we need to be able to sort the items that we have.

There are many sorting algorithms that are currently present and used our there and these have their benefits and disadvantages. However, when making product that will be usable by user for example the uber use case or the renting site use case, we would really love to have a sorting algorithm that if very highly efficient. That is we want a sorting algorithm that will be able to sorted millions or billions of data in as little time as possible. One of the sorted algorithm that is largely used out there for giving us this benefit is the quick sort algorithm which has a theoretical running complexity of $O(nlogn)$ where n is the number of items that we are trying to sort.

It is worth noting the fact that computer power has been increasing lately. We now have multiprocessor computers that can now perform sorting 1 million number of items in no time. We also have cluster that can easily distribute work to other nodes I the cluster for faster computations. We therefore in this work are going to look into using these programming constructs in particular we are going to look into implementing two separate sorting algorithms in two computation models which are shared memory and message passing. We are going to use the Open MP library with the C programming language for the shared memory implementation and the MPI standard and the C programming language for message passing. The sorting algorithms that we are going to use are quick sort and parallel soring by regular sampling.

The aim of the work is to discover which of the programming models, shared memory using Open MP or message passing using MPI standard are going to give us better speed ups for the different sorting algorithms. Speed up is by what factor has the run time of the algorithm being reduced from using serial solution to using Open MP or using MPI. This will in translation tell which of the programming models are good to use when trying to reduce the time taken by a particular sorting algorithm.

2. Serial Quick sort

2.1 Description

To calculate speedup, we need to have implementation of the algorithms in different formart and that we will then compare and calculate the speedup if any. Therefore, for this project we first implemented a serial version of the quick sort in the C programming language. This serial version is going to be the base that we will compare all the other implementations with, and the speedup is going to be for the most part be based on it. The quick sort algorithm is one of the divides and conquer algorithms that are available for sorting a list of numbers. It has a best case and Average case performance $O(N \log(N)$ and a worst-case performance of $O(N^2)$ in terms of Big O notation. So, if we plotted a graph of run time vs the size of items sorted for random numbers, we expect a logarithmic type of graph. However, for sorting a list of numbers sorted in reverse order will be highly inefficient of this we would expect a graph of running time and number of items in list we expect a graph that is also most exponential. The expectations do not say the graph is logarithmic or exponential but rather states that the graph expected should be have the same trends with logarithmic and exponential graphs.

To avoid running into the worst-case scenario of quick sort, we randomized the all the sample data that was used to sort. Therefore, we know that our data is not sorted in any form and therefore we know that we are going to expecting exponential growth of run time in relation to increase in data size of the list of items that we are sorting.

Another important thing when implementing a quick sort algorithm is the selection of the pivot as it affects the run time of the algorithm. For this implementation the last item was always selected as the pivot.

2.2 Implementation

Implementation of the quick sort algorithm requires two steps. The first step is the step of partitioning the list at into to two sets. The second step is to recurse into the two partition until no partition is left and at each recursive step, we carry out the same steps where we partition and recurse. The algorithms for both sorting and partitioning the data to complete the quick sort algorithm are presented below as algorithm1 and algorithm2.

ALGORITHM 1: QUICK SORT ALGORITHM

```
input: list of n elements to be sorted, lowest index, highest index
if lowest index less than highest index do
        pivot ← partition(list, 0, n-1)
        call quick sort with list and 0 and pivot-1
        call quick with list and pivot+1 and n-1
output: sorted list
```

The algorithm that was used to partition the data is as below:

ALGORITHM 2: PARTITION A LIST

```
input: list of n elements to be partitioned, lowest index, highest index
pivot ← list [highest index]
i ← lowest index -1
for j= low, low+1, low+2……highest index do
        if list[j] less than pivot then do
                i ← i+1
                temp ←  list [i]
                LIST[I] ← LIST[J]
                LIST [J] ← TEMP
i ← i+1
temp ← list[i]
list[i] ← list[j]
list[j] ← temp
output: i (the midpoint separating the two lists)
```

Both Algorithm 1 and Algorithm 2 were combined to write the final quick sort program which is shown as Figure 1 on the appendix below.

2.3 Validation

There are two aspects to validation of the implementation of the algorithm. The first involves checking if the algorithm is correct in the sense that the indeed follows the quick sort algorithm. The second has to do with validating that the results from the sort algorithm produced the correct output that is produced the sorted data. To validate that the implemented code was an implementation of the quick sort algorithm we first tested the implementation of the partitioning algorithm which is the basis of the quick sort algorithm. After passing the array to the partition implementation we then tested to see if the output of the algorithm was what we expected it to be by printing the resulting array. The expected output was going to be an array containing similar contents as those passed. However, this time the pivot which was the last element in the original array would be correctly sorted that is will be in its correct place with everything before it smaller that it and everything after it greater than it.

To validate the correctness of the algorithm, files with random numbers was generated. The numbers in the file included numbers from -50000 to 1000000. We then passed the array with the contents of the files into the sorting algorithm, and then using the is_sorted function defined in the testing header file. This method would tell if a given array is sorted or not by returning non-zero number if array given to is sorted and a zero if the array given to it is not sorted. Hence all the output from the quick sort implementation were passed into this method and hence validating if the quick sort algorithm.

3. Parallel Algorithms Using Open MP

Using the Open MP library, we implemented two parallel sorting algorithms which are, the parallel version of quick sort as well as the regular sampling parallel sorting algorithm. These were implemented and then compared to with the existing serial version of quick sort.

3.1 Quick Sort

3.1.1 Description

The Open MP library was used with the C programming language for the implementation of the parallel version of the quick sort algorithm. As already mentioned, the running time of quick sort as expressed by Big O is $O(nlog(n))$ for random data and $O(n^2)$ for data that is sorted in reverse order. This run time does not change even with the introduction of parallelism. That is, we still expect either a logarithmic type of graph for when plotting the graph of time taken against the size of the data for these algorithms. However, we expect the graphs to be scaled down compared to the serial implementation of the algorithm. That is, we expect for the same data size the time taken by the parallel version of the quick sort algorithm to be lesser than that taken by the serial version of the quick sort algorithm. To be specific we expect that the time taken will be reduced by 2 that is the graph of the parallel quick sort implementation is going to be $\frac{1}{2}$ of the serial one. The reasons for this are because we expect that now at each instance to have 2 threads sorting the array instead of having one thread sorting the array as is the case with serial implementation of the quick sort algorithm.

3.1.2 Implementation

The implementation of the parallel version of the quick sort algorithm is the same as the serial implantation except for the fact that after partitioning the array, two separate threads will then quick sort the sub arrays. That is assuming we had three threads:

STEPS FOR THREAD *N* IN THEORATICAL IMPLEMENTATION OF PARALLEL QUICK SORT
THREAD N: • Gets the input array • Partition the array and gets the pivot • Call 2 more threads to quick sort the sub arrays as well.

The above is the steps that each of the threads are going to be doing. Therefore, at each point we have two threads sorting the array instead of the one that we had with the serial sort. Thus, this is the reason we are expecting a speedup of about 2 for the parallel implementations.

However, this speed is not attainable for the reasons and the above picture of implementing the parallel version of the quick sort algorithm is rather too simplified. The speedup for the parallel implementation is going to be affected by the management of the threads. The way we implemented the quick sort parallel quick sort for this assignment was using the concepts of tasks that is presented by the Open MP library. The steps we used are as follows:

- We create an arbitrary number of threads but preferably 4 threads.
- The first thread will get the initial full array and then partition that and get the pivot.
- The thread then creates two task which is to sort to the left of the pivot and to the right of the pivot. This work is going to be added into a task queue for any of the threads available including this current thread to take.
- The next thread sorting the two sub arrays are also going to follow the steps enlisted above.

The idea of tasks within Open MP is that the task is going to be queued into the task queue and then they are run whenever else possibly. Other threads can continue doing other things unless the taskwait construct is used. If task wait is used, all the threads will first wait for the task queue to be complete before continuing with their execution. However, in this case we don't require such restrictions and the taskwait construct was not used. Since the threads are already created, with the task construct, we don’t create new threads at each place we need to divide the work to different threads. This then ensures that we will have be able to take the advantages brought about by parallelizing the code but also get little over heads for creating new threads at each instance we need to divide work to available threads. The imp implementation of the quick sort algorithm is shown in the appendix of this document.

3.1.3 Validation

The validation of the implementation of the algorithm was like the validation of the serial version of the quick sort algorithm. However, for this instance, we need to ensure that the parallel implementation of the parallel quick sort algorithm was faster than that of the serial version. This was done by sorting arrays of different size and measuring the time between to sort the parallel version and the time to sort the serial version of the algorithm. We then performed a manual check in checking if the time taken by the parallel sort is generally smaller than that taken by the serial implementation of the algorithm.

3.2 Parallel Sorting using Regular Sampling

3.2.1 Description

Parallel Regular Sort by regular sampling is a 6-phase sorting algorithm (Wake Forest University). Theses phases an be summed up as (Wake Forest University):

1. Set up t threads and let thread 0 get the array of size n
2. Let each thread work on some part of the data and quick sort it. All threads then must select samples
3. Thread 0 gets the samples and then merges them after which it selects $t-1$ pivots which we send to other threads
4. We partition the data local to the threads
5. Threads i then receives the i^{th} class of data from the other threads
6. Then thread 0 collects all the data

Since Open MP supports a shared memory model, all communication will be done with threads communicating via the memory. The speedup expected from parallel sort is (Wake Forest University):

$$speed\ up = \frac{plog(n)}{\log(n) + p + 1}$$

For large data sizes we expect a speed up of approximately p according to (Wake Forest University). This speedup is calculated by trying to measure the complexity of parallel sort. The calculation of complexity of parallel sorting by regular sampling takes into consideration the complexities at each step and sums all the complexity of at different phases.

3.2.2 Implementation

The implementation of this algorithm was taken from https://github.com/Fitzpasd/Parallel-sort-by-regular-sampling. Minor aspects where changed to make the program work in for our case.

3.2.3 Validation

The program was validated manually by passing in different kinds of input and using the is_sorted function is the testing header file to find out if the resulting array was sorted002E

4. Parallel Algorithms Using MPI

We also in this practical, using MPI implemented the quick sort and regular sampling parallel sorting. MPI is a standardized and portable message passing interface which is typically used in clusters to facilitate inter process communication. That is, we have two Nodes which want to share data via sharing messages as they can't share memory.

4.1 Quick Sort

4.1.1 Description

We implemented the MPI version of quick sort also using the c programming language. The big O notation run time of the algorithm is still not affected by the fact that now we have different computer running the quick sort algorithm which is the case when running on the cluster. So, we still expect a graph that has the structure of $nlogn$ which is the same as that of serial and as that of Open MP implementation of the quick sort. However, we expect that the graph will be scaled down from the serial version. That is the graph is below that of the serial implementation but then above that of the Open MP implementation for the following reasons:

4.1.2 Implementation

The implementation for the MPI version of quick sort was inspired by (Perera, 2009). MPI style of programming involves having different processors sharing data via message passing. That is if a process A wants to talk to another process B then process A needs to send the message to process B. Below are the steps that show the process of sorting an array of number using quick sort under MPI.

STEPS FOR PERFORMING QUICK SORT WITH MPI FOR PROCESS *N*

PROCESSOR N:

- Gets the input array
- Partition the array and gets the pivot
- If there are more processors left:
    - Check which of the partitions s less
    - Send the less partition to one of the remaining processes
    - Sort larger partition using the same steps
- Else:
    - Perform a parallel quick sort using OMP/ or perform a simple serial quick sort.

The first thing to note from above is that at some point in time one process will need to share the array with another process. For this there are two things to consider:

- Which of the processes does this process share with?
- How are we sharing with the other processes? That is which of the partition do we send to the other process for sorting?

The answer to the second question is quite an easy one compared to the answer to the first question. For the second question *which of the partitions does the current process send to the next processor?* We decided it will be more efficient to send the partition with less items and remain with the partition with the most items. This is was done to reduce the overheads of sending large data across to the other processor which was going to take more time. Therefore, we reduced that overhead by sending the partition with the small number of items to the other process.

Choosing which process to share the data with.

There are two possible techniques. The first technique will be to always have a process broadcast to the other processes once it has been given work. For instance, processor 0 will first broadcast that it has work and thus the other processor will now know that they can share with any other process except processor 0. When processor 0 shares with process 1, processor 1 will also broadcast to other processors to alert them that it is working on something. The processors can also broadcast when they are done. However as can be seen this approach is quite inefficient as we will have the overhead of receiving messages every time one process shares work with the other processor. Also, this approach will have required a lot of synchronizations that were going to cost even more. Therefore, there was need to look for a more efficient solution.

This solution was taken from (Perera, 2009). To answer it we will begin with an illustration of how we want the sharing to be done. The following illustration was taken from (Perera, 2009) and illustrated which process will share with which process and how we can compute that information.

Process	Sharing Set	Sharing rank calculation
0	1	$0 + 2^0$
0 1	1, 2 3	$0 + 2^0$,0+2^1 $1 + 2^1$
0 1 2 3	1,2,4 3, 5 6 7	$0 + 2^0, 0 + 2^1, 0 + 2^2$ $1 + 2^1, 1 + 2^2$ $2 + 2^2$ $3 + 2^2$

Table 1 Illustrating how process sharing will occur

The above assumes we have 8 processes are going to be used. From the table above, (Perera, 2009) the managed to deduce the following equation for deducing which other processes will one process be able to share data with:

$$Process = r + 2^n$$

Where Process is the process number we are trying to share with, r is the process number for the current process, and n must satisfy the following inequality:

$$2^{n-1} \leq r < 2^n$$

Therefore using this, each process will then in each iteration try to calculate the new process that it can share with, if the Process value calculated is greater than the number of processes we have, we then default to sorting using either OMP or the serial version of quick sort.

4.1.3 Validation

There where three validations done for the MPI version of quick sort. The first validation was to validate if we indeed where sending the lowest partition and for this we validated by having the process we are sharing with print out the number of items it received, and then the current process also printed how many items it how many items it was going to sort. Therefore, in this aspect the validation was manually done.

We also had to validate the sorting algorithm in the same manner that the serial version was validated.

4.2 Parallel Sorting using Regular Sampling

4.2.1 Description

Most of the description for regular sampling are like the ones made for Open MP. We still expect the same speed up although this time the speed up is going to have little overheads that are caused by communication issues. Therefore, even though the maximum speed up expected for regular sampling is p we expect that the speed up experience by Open MP is going to be a bit larger than that of MPI due to the communication overheads. What changes from the phases is that we are now using processors instead of threads? That is the steps are now:

1. Set up p processes and let process 0 get the array of size n
2. Let each process work on some part of the data and quick sort it. All processes then must select samples
3. Process 0 gets the samples and then merges them after which it selects $p - 1$ pivots which broadcast to other processes
4. We partition the data local to processes
5. Process i then receives the i^{th} class of data from the other threads
6. Then process 0 collects all the data

Communication will now be done via sending of messages as MPI does not support shared memory model of programming.

4.2.2 Implementation

Implementation for this was taken from http://www.cse.iitd.ernet.in/~dheerajb/MPI/codes/day-3/c/samplesort.c. We then did a few tweaks to ensure that the code was working.

4.2.3 Validation

Validation was also done manually using the is_sorted function provided in the testing header file.

5. Benchmarking

5.1 Methods

To conduct experiments comparing the serial version of quick sort, OMP version of quick sort and MPI version of quick sort as well as regular sampling parallel sort in OMP and MPI we had to undergo some steps. One step that was common to all these is file generation. We also needed some mechanism to measure time taken by the different sorting algorithms. We discuss how these were accomplished in the following discussions.

5.1.1 File generation

Files were generated manually by using an online random number generation site. (NumberGenerator.org) was used to generate different sizes of number separated by white spaces. We generated 10 different files. The file with the smallest number of items contains only 40 items and the file with the greatest number of items contains 1 million items. We tried to generate numbers randomly by generating numbers from -500000 to 10000000. This was done to ensure that we reduce the possibility of having a sequence of numbers that are sorted in reverse order to avoid the worst-case scenario of quick sort at any scenario.

5.1.2 Testing Methods

To conducting an experiment, we needed to get the numbers that were generated above from the files they are stored in and then pass them to the sorting functions as arrays. While doing this, we were measuring the time that was taken by each sorting function to sort the number of arrays and this time was recorded. The actual steps that were taken to perform the experiments are as follows:

1. Read file with the numbers
2. Save the numbers into an array
3. Pass the array into a sorting algorithm and immediately record the starting time
4. Perform sorting in the sorting function (ensure that there are no other operations done during this e.g. printing)
5. Collect the time immediately the sorting function returns and calculate difference between the time the sorting function was called and the current time.
6. Record this time as the duration.
7. Repeat this process with the same number of file and find average until you are satisfied with outcome.
8. Change the file you are reading from and repeat steps 1 to 8 (changing the file will change the number of items that need to be sorted.)

For measuring time, the wall clock was used. For MPI and OMP the MPI_Wtime and omp_getwtime methods were used to measure the time. While for the serial version quick sort we used get_time method provided by the Linux system. The time was recorded in seconds.

5.1.3 Measuring Speedup

Speedup for the parallel versions was measured by comparing the time taken by the parallel algorithm to sort and that taken by the serial version of quick sort. The following equation is used to calculate the speedup:

$$Speed\ up = \frac{Time\ taken\ to\ sort\ using\ serial\ sort}{Time\ taken\ to\ sort\ using\ parallel\ sort}$$

5.2 System Architecture

The following systems where used to conduct the experiments:

- HPC Cluster (Curie Partition)
    - **Memory** 128 GB
    - **Process** Opteron 6376
        - 64 cores

5.3 Results

5.3.1 Comparing speedup graphs between two 4 threads of OMP and 8 threads of OMP implementation of Quick Sort

To start the analysis, we will consider the performance of parallel Open MP quick sort implementation. The results for drawing the graph where collected from the UCT HPC cluster. To get the speedup, we ran the serial version of the quick sort program 100 times and got the average running time for sorting an array of a size. This same process was done for Open MP parallel implementation. We then took the time that it took to the serial version to sort the items and divided that by the time it took the parallel quick sort to sort the items for the same number of items as suggested by the equation above. This then gave us the speedup which we have plotted against the number of elements that where sorted.

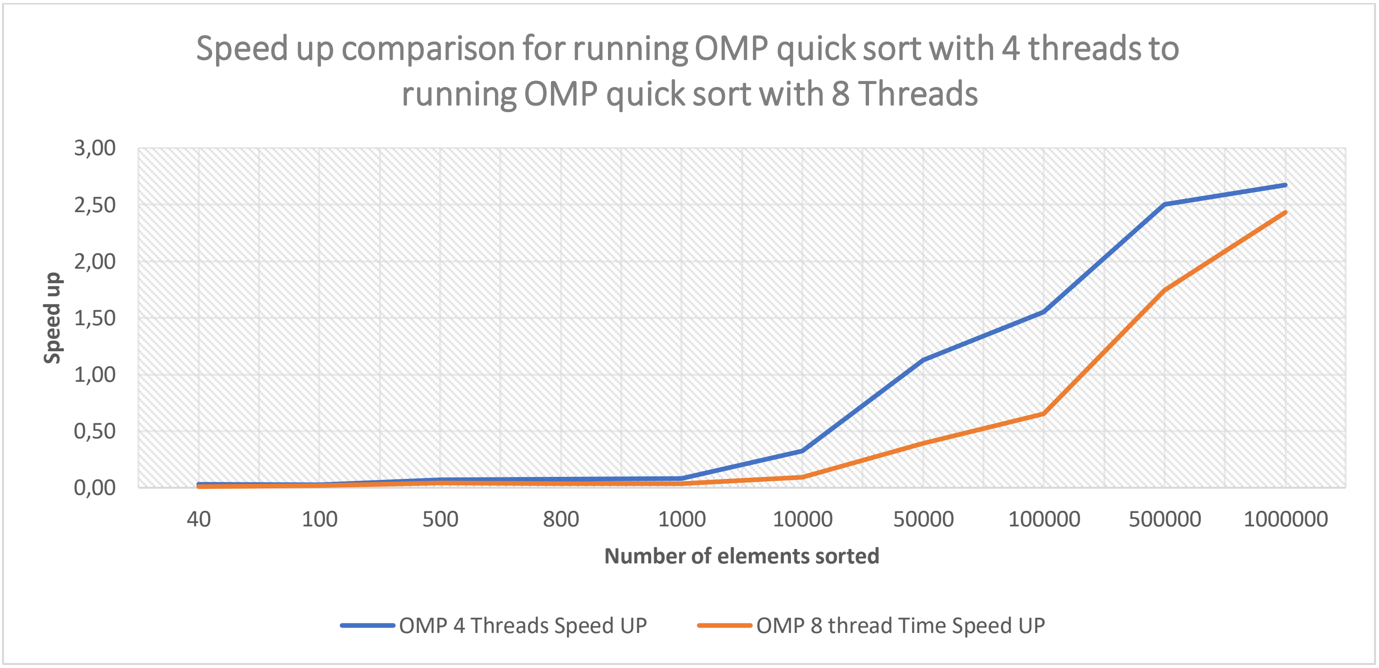


Figure 1 Showing the speedup comparisons for running Open MP quick sort with 4 threads vs with 8 threads.

The first thing to notice is that the speedup graph for running with 8 threads and for running with 4 threads has the same structure. That is starts low below 1 and gradually increases with increase in number of elements. The initial assumption made, was that with 2 more threads, maximum speed up for sorting items would have been 2. However, we see that that is not really the case as can be seen for working with 4 threads we don't really achieve a 4-fold increase. The maximum speedup achieved was 2.67. This is because of the discussed implication of thread handling that are added when working with threads. The creation of threads is an expensive process. For this same reason we realize that for input less than 10000 items, for sorting with 4 threads, sorting the numbers using serial version of quick sort was much better. This is shown by the speedup for these sizes being less than one. A speedup of less than 1 suggests that the serial version was faster than the parallel version as speedup is calculated as below:

$$speed\ up = \frac{time\ to\ run\ serial\ version}{time\ to\ run\ the\ parall\ version}$$

Therefore, the only way we ca get speedup less than 1 is if the serial time is less than the parallel time meaning that the serial code was faster than the parallel code.

Another thing to note, for the 8 threads, it takes a much longer time to attain the same speedup as the 4 thread did. For instance, the implementation using 4 threads of the parallel quick sort reaches a speedup of 2.5 on an input size of 500000. On the other hand, with 8 threads we reach that speedup on 1million items. This also suggests that adding more threads to a problem does not mean we will have more speedup. In fact, for small problems, we will have less speedup as the price we pay to handle threads is much more than the benefits we get from using more threads. Therefore, when the input size is of less than 1million we can see it is better to use 4 threads with OMP however, when the input is more then we can use 8 threads.

One last thing with the is that as we can see, the speedup graph for OMP with 4 threads is reaching a constant speedup after 500000 items. This is shown by the decrease in the increase of speedup with increase in data size. We can then predict that the same behaviour will have been witnessed if we had continued increasing the size of the arrays, we were sorting for the 8 threads OMP implementation. That is after a certain input, we don't experience any increase in speedup as we have reached the maximum speed up. We also predict that this speedup is going to be less than 8 as well for the 8 threads implementation.

5.3.2 Comparing OMP quick sort and OMP PSRS

We also considered the comparison of the different sorting algorithms using OMP approach and this is shown by then graph below. The graph below shows that the speed up graphs for a quick sort implementation using OMP with 4 threads and for a parallel sort by regular sampling using OMP with 4 threads as well.

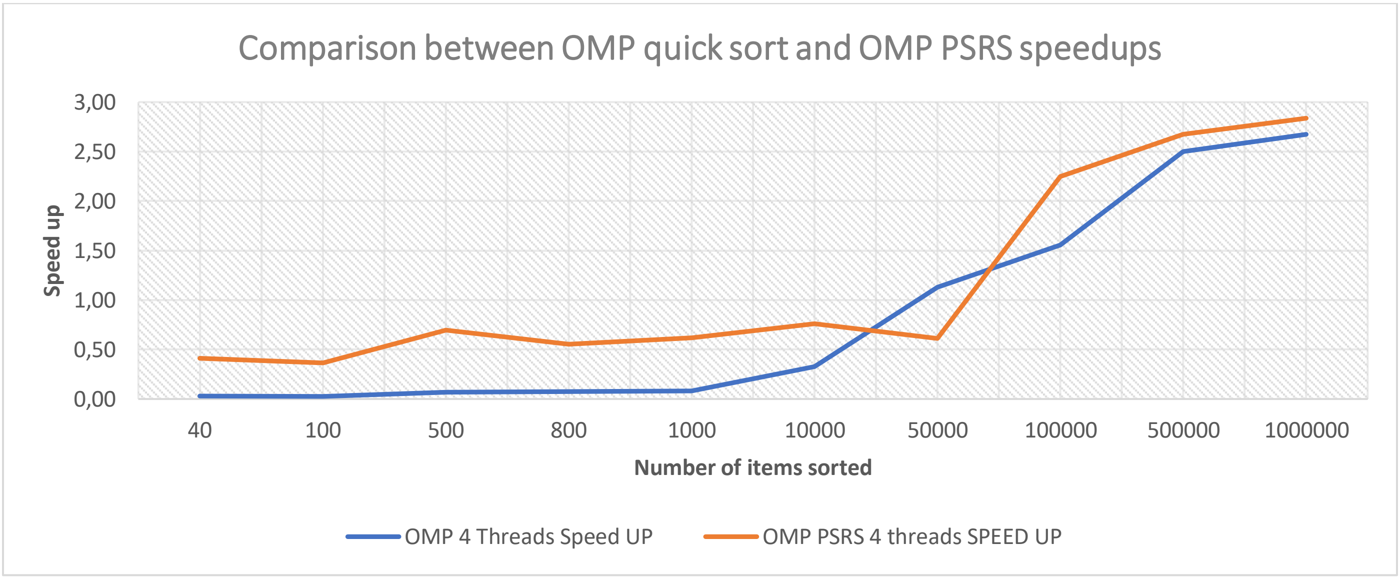


Figure 2 showing Comparison between OMP quick sort and OMP PSRS speedups

The graph above indicates to us that the parallel sort implementation has a higher speed up compared to the quick sort implementation except when they are both reaching the constant speed up value where they are bot almost equal. However, we still get the same structure within the graph where by we have speed of less than one below sorting a certain number of items. We also realize that the benefits of using 4 threads with parallel sorting by regular sampling are only experience when we have way over 50thousands items to sort while those of using 4 threads for quick sort are noticed at about 50thousand items. In overall, parallel quick sort gives more speed up, however the speed up experience is still not far from those experienced when using quick sort.

5.3.3 Comparison between MPI quick sort with different configurations

The different configurations that were used for experimenting with the MPI quick sort implementation include, running with 4 nodes 4 processors using serial quick sort when out of processes, running with 4 nodes using 4 threads for OMP sorting and lastly 8 processors using serial sorting when out of processors. The hypothesis before running the experiment was that increasing the number of processors should yield better results and also using serial sorting was going to also yield better results as we remove the thread handling overhead that will be on top of inter-process communication handling. The graph below shows the results that we had.

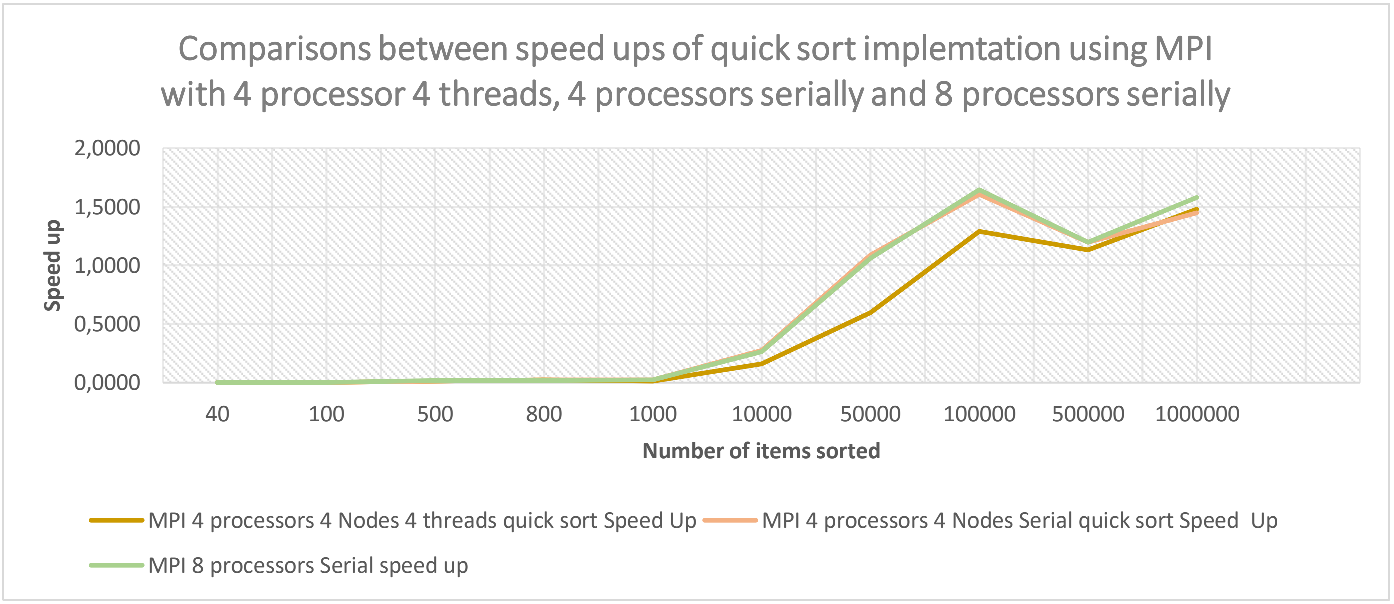
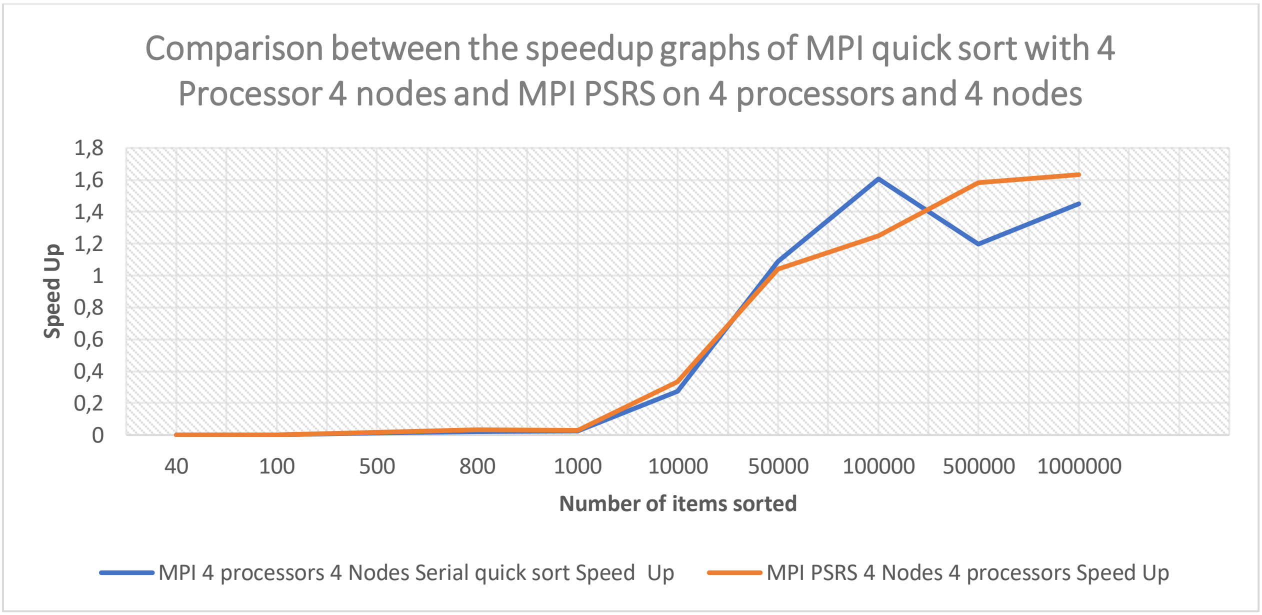


Figure 3 showing a graph of speed up for different configuration of quick sort implemented with MPI

The first thing to note is that the two graphs have the exact same structure and they only differ by the fact that one of the graph, for using MPI with 4 Nodes, 4 processors and 4 threads with OMP for process sorting, is scaled down version of the other two graphs. As can be seen, using serial quick sort when out of processors performed way better than using OMP threads which is seen by the high speed up experienced when using file sizes of grater than 1000. We predict that as we continuously increase the size of items that need sorting, the speed ups at they end will be constant but then the speed of the 2 serial implementations after running out of processes will still be greater than that of the parallel implementation after running out of processes. It is interesting to note that the using 8 processors and using 4 processors did not really make any difference in the end.

5.3.4 Comparison between MPI quick sort and MPI PSRS

Another discussion worth having is looking into speedups experienced by MPI implementation of the PSRS algorithm and that experiences by MPI implementation quick sort. The configurations used are 4 nodes with 4 processors for both quick sort and parallel sorting by regular sampling. We used serial version of quick sort for sorting in MPI in the instance we ran out of processors to share with. Like other experimentations we ran this with elements up to 1million. The expected maximum speed up was something just below 4 if not 4. The results are shown and discussed below.



The basic structure of the two graphs is the same. And the graphs actually seem to be equal for items below 1000 and there is a slight difference for items at 10000. Otherwise the performance difference is actually seen for items greater than 50000. While for items greater than 50000 speed up increase for the quick sort implementation is greater than that parallel sorting by regular sampling implementation the reverse is true for items just over 100000. It can be assumed that the results speed up experienced by the quicksort implementation is an anomaly an that the correct structure of the graph should be that of parallel sorting by regular sampling. For items greater than 100000 MPI sort has better speed up increase which then decreases after we start sorting with 500000 items. In overall the Parallel regular sampling sorting algorithm is much more efficient for larger sorting larger items when using MPI for implementing a sorting algorithm.

5.3.5 Comparison between OMP Quick Sort, MPI Quick Sort

Another comparison that was made was the speed up of implementing quick sort with OMP and implementing quick sort using MPI. Before the implementation, as described in the Description of these implementation, we expected a faster OMP meaning we expected greater speed up for the OMP solution as the MPI solution had more complex overhead which involved the communication between the nodes. We ran the MPI with 4 different Nodes and 8 Processors using he serial sort quick sort when we ran out of processors. We then ran the OMP version of quick sort with 4 threads. These specific configurations where selected for the respective implementations and they had produced the best results for the used the data sizes. The graph below shows the results obtained

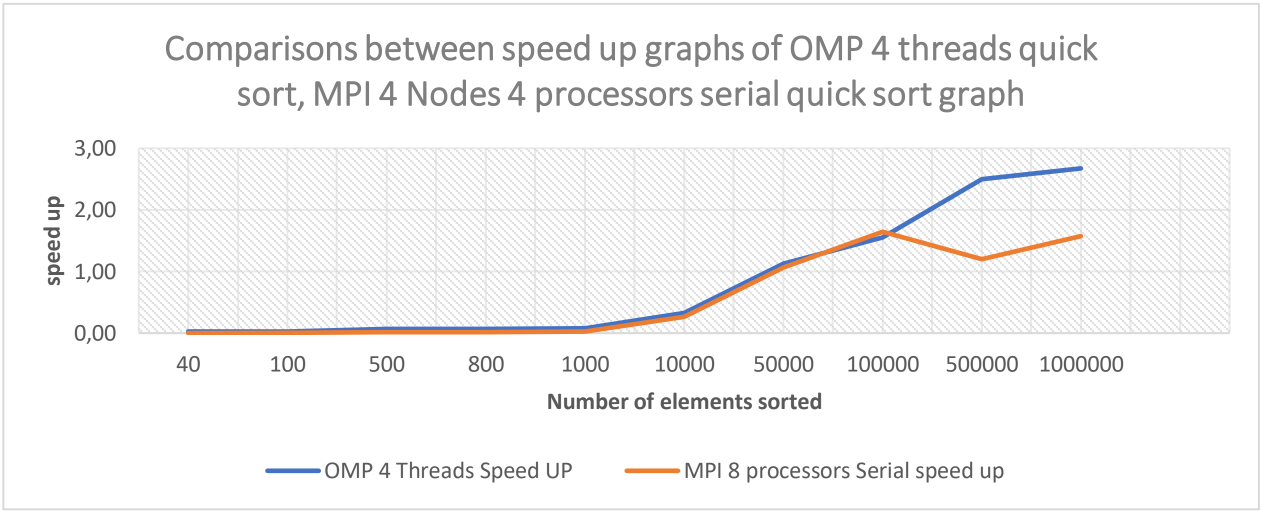


Figure 4 showing a graph comparing the speed ups of using OMP with 4 threads to implement quick sort vs using MPI with 4 nodes and 8 processors to implement quick sort

As can be seen from the fig above, the results obtained are mostly the same with difference being noticed from data size of 100thousand items going up. The results actually confirm the hypothesis of the speedup of OMP being more than the speed up of the MPI implementation. For the 1million items used in max the OMP version obtains a speed up as high as 2.67 while the MPI implementation obtains the speed of 1.57. The graph for the MPI implementation is incomplete as it does not show the true maximum speed up obtained. Our assumption is that the maximum speed up is going to be the same as that of OMP if not less than it due to the overheads of inter-process communication.

5.3.6 Comparison between MPI PSRS and OMP PSRS

Lastly we looked to compare parallel sorting by regular sampling implementation on the two programming models, the Message Passing Interface and Open MP which is a shared memory model. The configuration for used to compare the MPI PSRS is 4 Nodes with 4 processors. For OMP on the other hand we used 4 threads. These configurations where chosen as we expected, the same maximum speed up for the two operation that is a speed up of 4 for both MPI and OMP implementations. This is because as stated above the speed up expected when the size of the data to be sorted is way more than the number of processors and the number of threads, we expect to speed up by the number of the resources added. The results obtained are below in the graph.

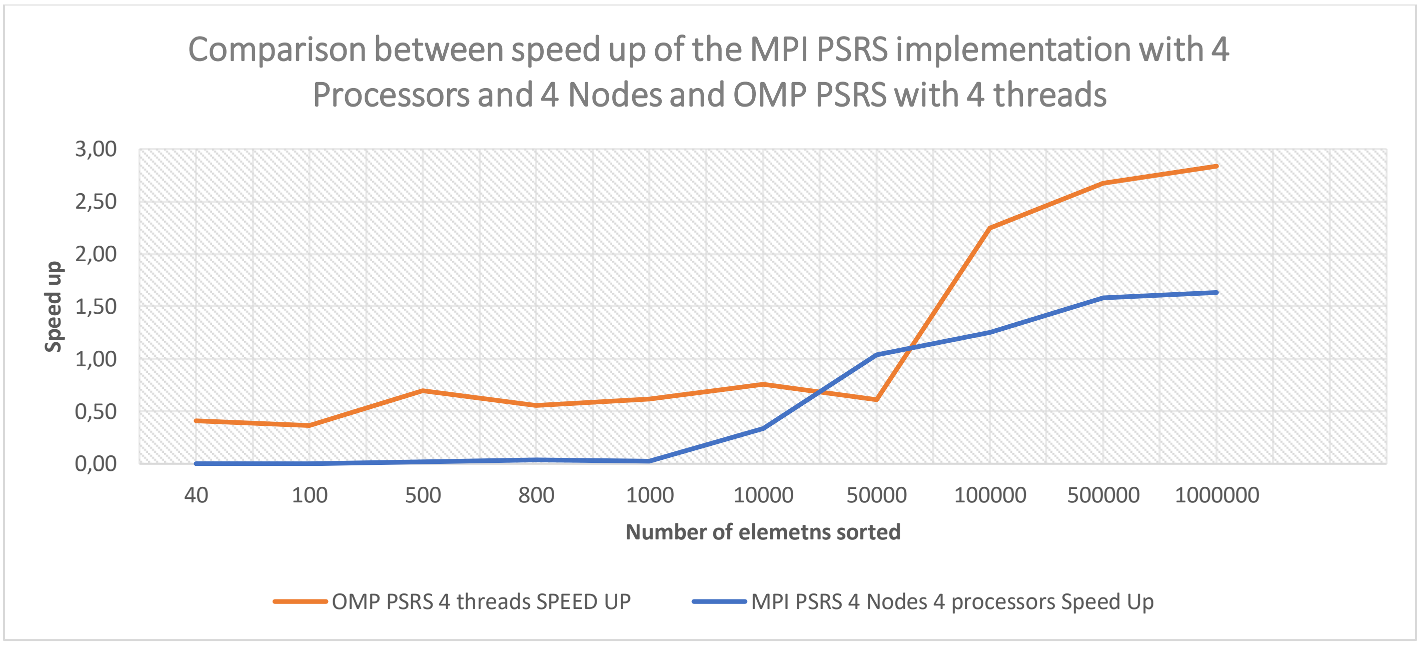


Figure 5 showing a graph comparing the speed ups of using OMP with 4 threads to implement Parallel sorting using regular sampling vs using MPI with 4 nodes and 4 threads to implement the same algorithm.

The graph for using MPI has a simple structure, starts up with speed up being approximately 0 and constant for items of size one thousand, and there after the speed up increases at faster rate to reach 1 for fifty thousand items being sorting. After which it increases with a rather slower rate and this increases keeps decreases with more added items to sort. However We can kind of tell that the speed up is going to reach a peak value such that after it adding items won't speed up the algorithm.

ON the other hand, the OMP speed up graph for PSRS, is rather different. Generally, the structure of the graph is the same of that of MPI implementation in the sense that we start with a speed up less than 1 and gradually increase the speed up as we add more items to sort. Eventually we can see that we are almost reaching the peak speed up for OMP ad around 2.8. The speed up for MPI when soring one million items is 1.63. it is assumed that the reasons for the speedup differences is MPI has added overheads of communicating with other nodes to in attempts to sort the array.

6. Conclusion

We have thus looked into two main sorting algorithms, quick sort and parallel sorting using regular sapling. We implemented the quick sort algorithm serially so we can use as the base for calculating the speed ups for the other implementations. We then implemented quick sort using two different programming models, the first being shared memory via using the Open MP library for c, the second being message passing implemented using MPI and specifically the mpich-3.3 implementation version for the MPI standard. We also implemented parallel sorting by regular sampling using these two programming models as well and tried calculating speed ups for sorting different number of items under different configurations. We found the results that we then presented above.

We have then come to the conclusion that using multiple nodes to implement sorting with MPI can be a very big overhead as we experienced minimal speedups for using 4 Nodes. Increasing the number of threads also for sorting small data sets is just an adding more overheads which makes decreases the speedup expected. Finally we feel that the shared memory model is best for implementing sorting algorithms as we experienced higher speed ups for this implementation.

Computations were performed using facilities provided by the University of Cape Town's ICTS High Performance Computing team: hpc.uct.ac.za

7. References

NumberGenerator.org. (n.d.). Retrieved from https://numbergenerator.org/

Perera, P. (2009). Parallel Quicksort using MPI. Retrieved from https://www.codeproject.com/KB/threads/Parallel_Quicksort/Parallel_Quick_sort_without_merge.pdf

Shi, H., & Schaeffer, J. (1992). Parallel sorting by regular sampling. *J. Parallel Distrib. Comput.*, 361-372.

Wake Forest University. (n.d.). Retrieved from http://csweb.cs.wfu.edu/bigiron/LittleFE-PSRS/build/html/PSRSalgorithm.html