Assignment 3: Study on Parallel Sorting by Regular Sampling
- Parallel Algorithms -

author: Alexandre Silva and Daniel Malhadas

***Abstract*—This paper reports the effort and work expended by the authors in order to reach conclusions regarding the performance of an implementation of a parallel sorting by regular sampling based on [1]. This work presents a deep study of the obtained work-load balance, time and space complexity and various possible optimisations. The algorithm is then heavily tested on two different machines where the obtained time speed-up for the increasing number of used threads is observed.**

***Index Terms*—Sorting Algorithms, Parallel Sorting by Regular Sampling, Performance Analysis, Parallel and Distributed Computing.**

I. Introduction

A. Contextualisation and Motivation

The problem of sorting has had increasing importance over the years and it's large area of application contributes to increase it's growing relevance. There is no limit to the situations where sorting may be needed, therefore it's study also has an increasing practical relevance.

In the context of parallel algorithms, the algorithm **Parallel Sorting by Regular Sampling (PSRS)** is a relevant sorting algorithm to study as it achieves good results as can be better understood on a more profound study like [1] which was the basis for the implementation this document relates to.

B. Document Structure

This document is structured as follows:

Initially, on **section 2** we mention the environment used to test our implementation and the data-sets used, justifying both the choice of machine and of data-sets. On **section 3** The algorithm is explained in detail after which there is a brief study of the time and size complexity as well as the work-load balance. **Section 4** presents the obtained results and interprets them providing proof to our conclusions. **Section 5** then sums the conclusions reached on section 4 and from them relates relevant possibilities for possible future work.

II. Initial Considerations

A. Characterisation of the used Machines

During this assignment two main machines were used and the results presented from now on will be related to these specific machines. The use of two different machines allows for a much better understanding and certainty from the results of our studies since we get results from two fundamentally different kinds of machine. Our machines are: Cluster SeARCH's node 652 and node 431. Both these machines are fully characterised on the next table:

Machine 1	
Designation	Nodo 431
CPU Microarquitecture	Intel
CPU Model	Dual CPU X5650
CPU Microarquitecture	Nehalem
Clock Frequency	2.67 GHz
#Cores	12, with 24 threads (2 per core)
Cache	L1d: 32kB, L1i: 32kB, L2: 256kB, L3: 12288kB
Memory Access Bandwidth	32GB/s
RAM Memory	48GB
Communication	Gigabit Ethernet e Myrinet 10Gbps

Machine 2	
Designation	Node 652
Manufacturer	Intel
CPU Model	Dual CPU E5-2670v2
CPU Microarquitecture	Ivy Bridge
Clock Frequency	2.50 GHz
#Cores	20, with 40 threads (2 per core)
Cache	L1d: 32kB, L1i: 32kB, L2: 256kB, L3: 25600kB
Memory Access Bandwidth	59.7GB/s
RAM Memory	64GB
Communication	Gigabit Ethernet e Myrinet 10Gbps

This data was obtained from various places. By using the Unix command **lscpu**, it allowed us to know the manufacturer, number of cores and threads per core and size of cache levels. For the remaining items [4], [5] and [6] where accessed.

It can be seen that the choice of these two specific machines was not random. their very different microarquitectures, L3 cache specifications and number of cores/threads allows us to study the implementation on very distinct environments, resulting on a better quality and certainty of the obtained conclusions.

B. Characterisation of the used Data-Sets

It's important to also gather relevant information in order to understand appropriate number of elements for the data-sets we will sort during our tests so that our conclusions are indeed relevant and not just random cases chosen arbitrarily. four relevant data-sets were chosen:

Data-Set 1 - A small data-set that fits well inside both machine caches

Because the 431 Cluster's node has the smallest L3 cache (12288kB as opposed to the 25600kB on the 652 node) we shall base this particular data-set on the 431 node. Because we will test the implementation with the sorting of double precision values (C double type) and each of these values needs 8 bytes, therefore on this cache level we can store 12288000/8B = 1536000 doubles. In order to fit the whole cache level we would then sort 1536000 values. However we can also take into account the size of a cache line. In this case it is 64B, therefore we can fit 8 doubles on a cache line, so our total number of doubles to sort should be a multiple of 16 in order to improve spacial locality on the cache. Two conditions are then established, this data-set should be smaller then 1536000 values to fit well inside both machine's cache and should be multiple of the number 8 to make proper use of spatial locality. A possible and interesting value for this is 62208 elements (62208/8 = 7776).

Data-Set 2 - One data-set that completely fits all cache levels and forces the CPU to load it from the main memory

Because the 652 Cluster's node has the biggest L3 cache we shall base this particular data-set on this node. On this cache level we can store 25600000/8B = 3200000 doubles. In order to fit the whole cache level we would then sort 3200000 values. With this out of the way, the two conditions we need to meet for this data-set are that this data-set should be significantly bigger then 3200000 values to completely fill both machine's cache and force the CPUs to load it from the DRAM, also it should be multiple of the number 8 to make proper use of spatial locality. A possible and interesting value for this is 4000000 elements (4000000/8 = 500000).

Data-Set 3 - An even bigger data-set to test the implementation further

The next two data-sets are calculated the same way with the only condition being to be bigger then the last data-set with the intention of testing the implementation even further allowing for a better understanding of how scalable this implementation is in terms of data size. A possible and interesting value for this data-set is 6000000 elements (6000000/8 = 750000).

Data-Set 4 - An even bigger data-set to test the implementation even further

A possible and interesting value for this data-set is 8000000 elements (8000000/8 = 1000000).

III. Implementation

A. What is PSRS?

PSRS is a MIMD sorting algorithm [3] [1] that distinguishes itself from other MIMD algorithms by it's good work-load balancing, modest communication needs and good locality of memory reference [2]. While other MIMD algorithms are optimised for particular architectures and get worse when used with others, the PSRS algorithm is generally good for a vast range of MIMD architectures. This is the main vantage that makes this algorithm such a relevant one to study. If we want to deploy a certain big project on many different MIMD architectures or don't have time to explore the particular strengths or weaknesses of a specific architecture then PSRS is a solid choice that will, most likely, perform well. Of course another algorithm particularly optimised for a certain architecture is expected to perform even better.

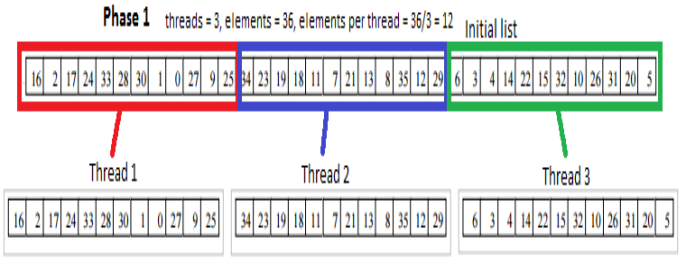
B. Algorithm explanation

The present implementation consists of four different phases.

Phase 1:

Initially, in phase one, the values to be sorted are divided into different contiguous lists, one for each thread. In order to divide our elements to sort in different chunks we actively try to divide as equally as possible so that a work-load balance is maintained between all threads because, as can be expected, a better performance is expected when all threads execute similar work, as opposed to having one do the vast majority while others only do simple computations. However, when utilising this implementation the available number of threads may not divide perfectly the number of elements to sort. An easy but crude solution would be to divide as perfectly as possible having only one final thread with a slightly bigger or smaller set to sort. However this decision does not bring the best results as can be intuitively understood because the work-load balance is not as good as it could be. On these cases, instead of manipulating the elements to sort this way we rather manipulate the number of threads used. This decision comes from the assertion we make and believe to be very valid for these types of algorithms: **We should only use more threads when more threads are relevant**. Perhaps if the number of threads do not divide the input perfectly then, in order to maintain a perfect work-load balance, less threads can be used allowing for each thread to truly do relevant computations, reducing the overhead of the Fork&Join operations that would be even more noticeable because more threads equals less work done by each which means the Fork&Join overhead is more relevant. Experience has proved that a good work-load balance is reached when the **number of threads to the power of three is less then the number of elements to sort** [2], [3] and [1]. In order to decide the perfect number of threads we can round it to be the biggest even number that fulfils that condition (therefore also dividing perfectly all our data-sets).

As an example we see the following image depicting this phase with an example small list to sort:



Phase 2:

The second phase relies on an independent sorting, by each thread, of the chunk it was assigned to in phase one. This independent sorting could be done with any sequential sorting algorithm so quicksort is used because of it's average case complexity of **O(n log n)**. It's relevant to note that quicksort has a worst case complexity of **O(n*n)**, this situation is not common and therefore not very relevant but if it eventually ends up being a problem in the future for some reason then a merge sort could be used instead with it's worst case complexity of **O(n log n)**.

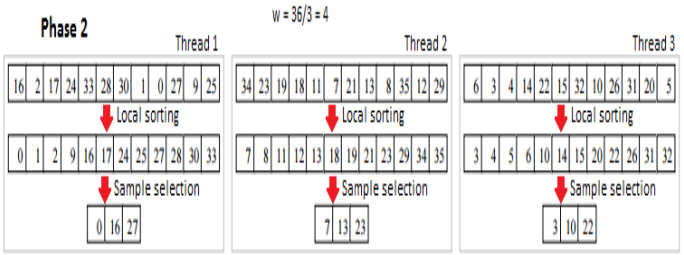
After sorting, each thread will take a certain number of samples from it's sorted list. These samples will allow the rest of the system to understand the balance of the values of the elements on each thread's list. To know which indexes of the contiguous list each thread should choose for it's samples the following formula is used:

$$1, w+1, 2w+1, ..., (t-1)w+1 \tag{1}$$

where t is the number of total threads used and w is a factor calculated with:

$$w = \frac{n}{t^2} \tag{2}$$

as an example we see the following image depicting this phase, continuing on from the example image presented on phase 1:



Phase 3:

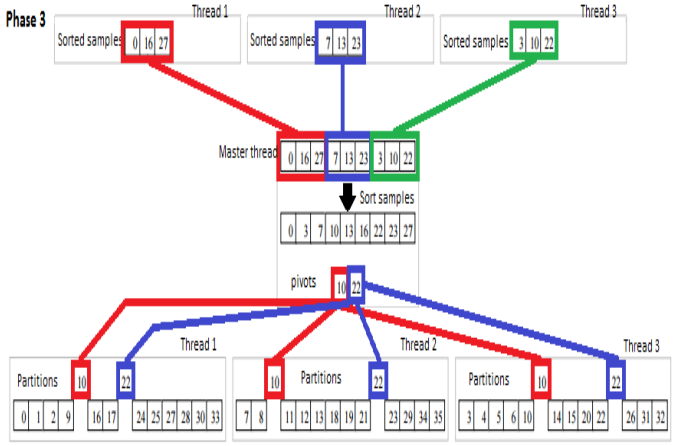
At this moment all threads join into a single master thread and the samples gathered on the previous phase are sorted. Once again any sorting algorithm can be used, however this time there is a real advantage to the use of merge sort because the samples from each thread are already ordered, for that reason merge sort was used here.

After sorting the samples a certain amount of them are chosen as pivots. The indexes chosen for these pivots are defined by the following equation:

$$t+t, 2t+t, 3t+t, ..., (t-1)+t \tag{3}$$

This new list of ordered indexes is then proliferated to each thread. Each thread will then divide it's independent list sorted previously on phase two into t sets based on the pivots it received.

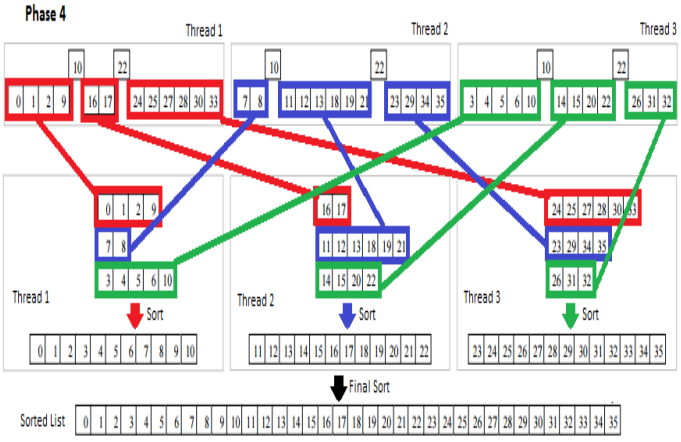
As an example we present the next image that continues on from the latest images:



Phase 4:

This time we fork the threads again and each thread takes it's partitions, keeps the one that corresponds to it's own thread number (as an example: thread 1 keeps partition 1, thread 2 keeps partition 2, ...) and passes on to the other threads their corresponding partition (as an example: thread 1 sends thread 2 the second partition, thread 3 the third partition, ...). In other words, each thread thread keeps one partition and reassigns t-1 partitions.

Lastly each thread sorts the partitions it received with the one it kept using quicksort. After this last sorting we get the final sorted list. Of course here we could also use merge sort or other algorithms however, according to [1], through extensive tests, it was concluded quicksort provides the best results here.



C. Complexity and Load Balance

Our implementation was made on a shared memory environment, therefore all thread communication is done that way, however the study of the size complexity of the communications is still relevant in order to understand how it could impact performance. Initially, on Phase 1, the list to sort is divided in t parts, here the proliferation of each part to each thread results on t communications of size $O(t)$. Then, when each thread has to send $t-1$ partitions to the remaining threads, each partition sent will result on a size of $O(n/t)$ with n as the number of elements to sort.

Now we look into the time complexity. From the three sorts done throughout the algorithm (one merge sort and two

quick sorts) we conclude a best case of $O(n/tlogn)$, also corroborated by [3] and [1]. Of course this is the best case scenario witch can only be guaranteed when $n >= t * t * t$ as it provides the optimal work-load balance, as referenced before and corroborated by the same sources.

Regarding load balance, once again we assert it is optimal on phase one, where the list to sort is evenly divided between all threads. The remaining of the algorithm is either sequential or depends on the machine's architecture, however, on phase four is where the load balance is more relevant. On [2] we can observe a deep study on the work load balance for this phase where very promising results are obtained, both in theory and in practice, even when there are multiple duplicates on the list to sort.

IV. Tests and Measurements

A. Characterisation of the Measuring Process

It's relevant to mention that all results presented over the course of this section resulted on the code being compiled using **g++** distributed with **gnu 4.9.0** with the added flags**-O3** for added optimisations and also **-fopenmp** for the use of **OpenMP** directives.

The library **sys/time.h** was also used which allowed for the use of thread safe time measurements with the **gettimeofday()** function and the expression **(end_time.tv_sec - start_time.tv_sec) * 1000000L + (end_time.tv_usec - start_time.tv_usec))**, where start_time represents the starting time and end_time the stopping time. This time measuring method allowed us to visualise the time in milliseconds with a the precision of a **C long long** variable.

The next subsection presents speed-up graphs that characterise the behaviour of our implementation on the different machines. These speed-up graphs were measured the following way: First the sequential time was measured, we'll call it **Tseq**, next for each number of threads intended (we'll call it **NumT**), we measured the time for the OpenMP version a total of ten times. Of those ten times the arithmetic average was calculated between all values that did not deviate more then 10% of the best time measured, let's call this average **Tomp**. We then have the speed-up calculated the following way:

$$speedup = \frac{Tseq}{Tomp} \tag{4}$$

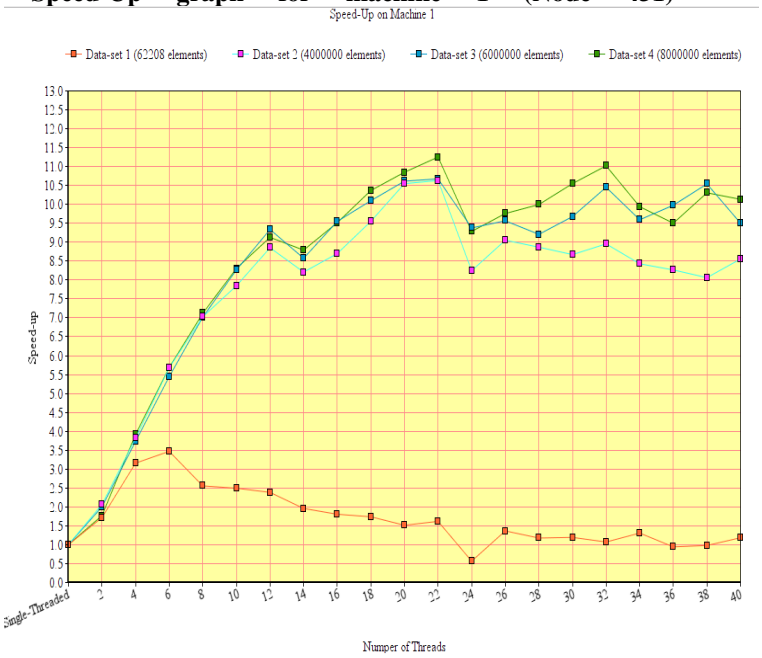
Also, in order to guarantee the correction of the implemented sort, every time it is run, the final sorted list is compared against the result of a sequential quick sort provided by the library **math.h**, which is expected to be correct itself. During all tests ran, no situation resulted on different sorted lists from these algorithms, therefore we can assume the correction of our implementation.

In order to generate the intended data-sets the **rand()** function was used on the following expression: **(double)rand() / (double)RAND_MAX** therefore providing different pseudo-random sets each time.

B. Interpretation of the results

Because no machine has more then 20 physical cores (40 threads with HyperThreading) then it's only relevant to study the speed-up for a maximum of 40 threads.

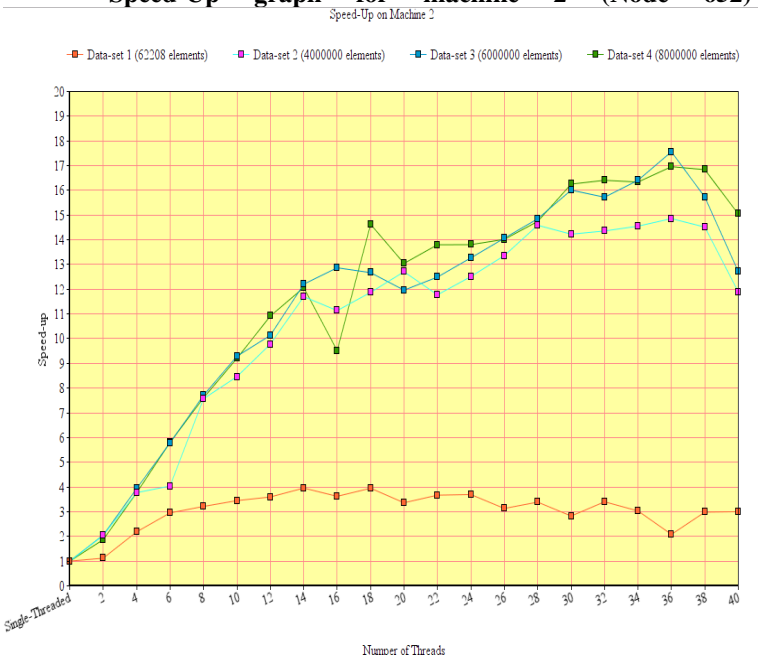
Speed-Up graph for machine 1 (Node 431)



On this graph we see the speed-up curve for the 431 node. For the smallest data-set the scalability regarding the number of threads is very low, it raises to a maximum of around 3.5 with 6 threads and then keeps decreasing all throughout the rest of the curve with a lowest point of 0.5 with 24 threads. It's very noticeable the difference in obtained speed-up between the smallest data-set and the remaining ones. This implies that this algorithm and this implementation are not meant for small data-sets which was to be expected. Previously we mentioned that the load balance is optimal when $t*t*t <= n$, even though this is valid for the smallest data-set for all thread numbers besides 40 we still see a very low speed-up. The problem is also not the communication between threads or the size of said communication because they are done with shared memory resulting on a minimal overhead. If the problem does not lie with the load balance nor with communication then we can only imagine it lies with the sequential sorting algorithms used over the course of the various phases, these may be the best for large data sizes as stated on [1] but as observed here they may be only average when an already small data-set is divided among so many threads. Another possible explanation may be that such a small data-set provides an already very small sequential execution time which will limit how much better the parallel version can ever hope to perform. This may very well be the reason because even though there is a low speed-up for this data-set we still notice that it exists only dropping below the sequential version after crossing the 24 threads threshold which can be explained by noticing that this machine only has 12 physical cores and 24 threads with HyperThreading, therefore when using more then 24 threads the performance is expected to be lower because some threads that are being used are unable to operate in parallel like they should.

For the remaining data-sets we notice that they achieve very good speed-up values easily surpassing the smallest data-set. This leads us to believe this algorithm and implementation are best used when large data sizes are present, on this case, all of them are bigger then the biggest cache level and are still faster then the smaller data-set. This corroborates the idea that perhaps for smaller data-sets alternative algorithms should be used. We notice all these three last data-sets reach a peak speed-up with the use of 22 threads achieving a maximum of around 11.7 with the biggest data-set. Just like the smallest data-set all others have a sudden decrease on speed-up with 24 threads which is expected to be for the same reason that the same was observed for the smallest data-set. The third and Forth data-set then increase a bit after this sudden drop however never achieve the same performance as before and even decrease further at the end proving that the speed-up stagnated here and the peak truly is the 22 threads. For the second data-set we observe a total stagnation after the sudden drop which leads us to conclude the same thing.

Speed-Up graph for machine 2 (Node 652)

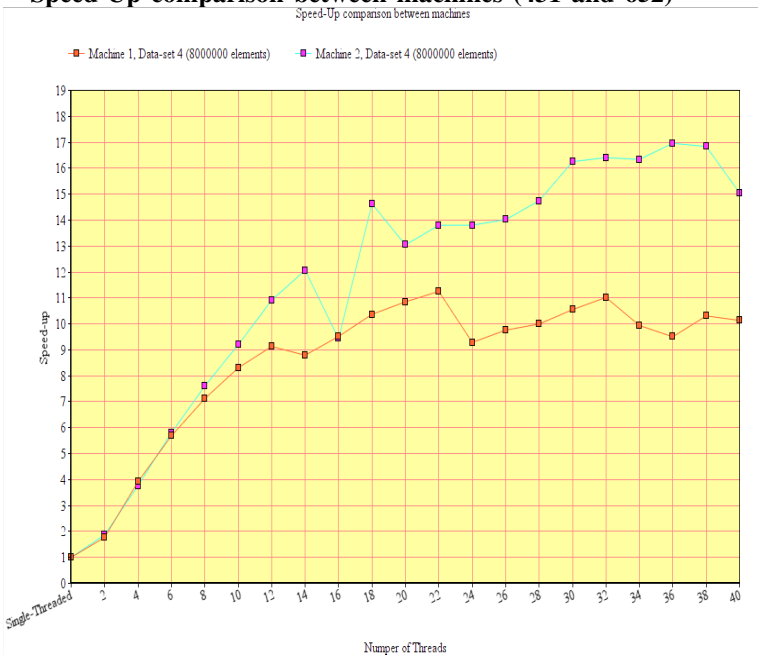


On the second machine (node 652) we observe similar results however some differences should be mentioned. Here the smallest data-set is still the worst by far however achieves a higher speed-up (around 4 with 14 and 18 threads) and never drops below the sequential threshold. It's speed-up seems to be stagnated throughout almost the entirety of the curve therefore the same conclusion can be derived.

The remaining data-sets provide a bigger speed-up with the third data-set achieving the maximum value of around 16.6 with 36 threads. All of these three bigger data-sets suffer a decrease in speed-up after this peak point on 36 threads, this takes us to believe that this 36 threads value is the true scalability peak in terms of threads of the algorithm for big data-sets. While the previous machine achieved a peak at 22 threads that peak was not "fair" to the algorithm because it was simply the hardware limit of the machine (it no longer had threads able to run in parallel), on this second machine we have 40 threads available so if after 36 it starts decreasing it's safe to assume it's the algorithm's upper bound and not a hardware limitation like before.

Next we will compare the speed-up on both machines more directly by observing the obtained behaviour for the bigger data-set on both with the same graph.

Speed-Up comparison between machines (431 and 652)



We can now notice that, even before 22 threads the performance on the second machine (652) far surpasses the one on the first machine (431). This was to be expected because even though the first machine has a higher clock frequency the second has a larger third level cache and also a larger memory access bandwidth. these things make the use of the second machine far superior especially for this algorithm with big data-sets because all of them are big enough to force the CPU to load them from the DRAM on both machines, therefore the larger memory access bandwidth on machine 2 is very relevant to achieve great performances. In other words, if possible, one should choose to use the second machine rather then the first if one intends to use this algorithm. Even though we see advantages in using the second machine we recognise that the speed-up is good on both machines, therefore corroborating the assertion made before and also stated on [2] where was mentioned that this algorithm is good for a large amount of different architectures.

V. Conclusions and Further Work

In the end, looking at all the conclusions from the previous section, we understand this algorithm to be indeed good on different architectures and recognise it's strength on the work-load balance. We also recognise its better performance with larger data-sets and with larger memory access bandwidth.

Regarding future work we believe it would be relevant if the performance of this OpenMP implementation was compared against an MPI implementation. Because this algorithm is intended for MIMD architectures [1] [2] it would be relevant

to implement the algorithm on a distributed memory environment and understand how the communication impacts the performance instead of just studying it in theory (like done on this document).

In conclusion, we believe this project was a success and that we could comprehend and measure the algorithm's pros and cons with enough detail to make a fair final judgement of the best situations to use it.

REFERENCES

[1] Hammao Shi, Jonathan Schaeffer, Parallel Sorting by Regular sampling, Department of Computing Science, University of Alberta, Canada, Journal of Parallel and Distributed Computing, 14(4):361-372, 1992.
[2] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, Hammao Shi, On the Versatility of Parallel Sorting by Regular sampling, Department of Computing Science, Canada
[3] Hammao Shi, Parallel Sorting on Multiprocessor Computers. Master's thesis, University of Alberta, Canada, 1990.
[4] site: cpu-world.com, accessed on June 2017
[5] site: ark.intel.com, accessed on June 2017
[6] site: http://search6.di.uminho.pt/wordpress/?page_id=55, accessed on June 2017