Modern Applied Science; Vol. 12, No.10; 2018
ISSN 1913-1844 E-ISSN 1913-1852
Published by Canadian Center of Science and Education

Measuring Parallel Performance of Sorting Algorithms

Bubble Sort and Bucket Sort on IMAN 1

Saher Manaseer[1] & Ahmad K. Al Hwaitat[1]

[1] Department of Computer Science, King Abdullah the II IT School, The University of Jordan, Amman, Jordan

Correspondence: Department of Computer Science, King Abdullah the II IT School, The University of Jordan, Amman, Jordan. E-mail: saher@ju.edu.jo

Received: May 2, 2018 Accepted: May 30, 2018 Online Published: September 12, 2018

doi:10.5539/mas.v12n10p23 URL: https://doi.org/10.5539/mas.v12n10p23

Abstract

The performance evaluation of sorting algorithm play a major role in understanding the behavior which has great benefit in most of the field of sciences, knowing the difference between parallel and sequential performance will help the researchers to choose the best algorithm bucket and bubble sort to use and implement. In this research we study the performance of two sorting algorithm and evaluate the difference in performance in aspect of speed up and efficiency, the two algorithms has been tested on IMAN1 super computer with different evaluate input size and different number of processors. The results showed that he performance of runtime for the bubble and bucket sorting algorithms has been effectively reduced by the parallel computing over the large data size and the number of processor of 64 get the lowest running time, and the parallel performance was better than other methods.

Keywords: parallel bubble sort, parallel packet sort, MPI, supercomputer

1. Introduction

Recently computers has become part of our life. But most of the computers that people use such as laptops are still sequential. It is strongly believed that the future of computing is for parallel computing. Karp and Ramachandran in 1990 state that, “Parallel computation is rapidly becoming a dominant theme in all areas of computer science and its applications. Within this decade, virtually all developments in computer architecture, systems programming, computer applications and the design of algorithms will take place within the context of a parallel computational environment. Jordan, H. F (2002).

Computer scientists have built the fundamental theory of parallel computers, e.g., parallel architectures and parallel algorithms. Parallel computers can be categorized in different types by their synchronization, instruction execution and communication design. A parallel computer is called synchronous if, at each time step, all processors execute an instruction (net necessarily identical) simultaneously, and otherwise is synchronous. A parallel computer is Single Instruction Multiple Data (SIMD) when all processors execute the same instruction at the same time, using possibly different inputs for the instruction. A computer is Multiple Instruction Multiple Data (MIMD) when processors may execute different, instructions simultaneously Manwade K. B. (2010).

Parallel processors may communicate either via shared memory orvia fixed connections such as Message Passing Interface (MPI), Parallel programing implementation using MPI is the most common way for parallel programming Among the three main models for parallel programming (shared memory programming, message-passing paradigm, MPI, Partitioned Global Address Space (PGAS) programming), MPI is used for single processingandparallel processing where many nodes are connected together. Where the standard of MPI is designed specifically for parallel applications and processing Chaarawi M. (2008).

Sorting in computer terminology is defined as the process of rearranging a sequence of values in ascending or descending order. Parallel bubble sort it's and Parallel bucket sort algorithms include two phases: the first phase to separate the ordered lists into p disjoint parts, and the second consists of p parallel sequential in different processors.

The aim of this paper is to analyze the performance of two sorting algorithms Parallel bubble sort it's and Parallel bucket sort and evaluate the difference in performance in aspect of speed up and efficiency, the two algorithms has been tested on IMAN1 super computer with different evaluate input size and different number of processors.

1.1 Overview of Message Passing Interface (MPI) Model

The MPI programming model, as well as the design decisions incorporated in most concurrent MPI implementations, are influenced by the services provided by operating systems on target platforms and the traditional run-time environment of the programming languages used. At the time when basic design decisions (such as mapping of MPI tasks to processes and having one thread of execution in an MPI task) were adopted and implemented, they seemed to be natural because there were bno other portable, standard alternatives available Chaarawi M. (2008).

2. Related Work

Manwade K. B., (2010) this study is to evaluated the performance of parallel merge sort algorithm on loosely coupled architecture and compare it with theoretical analysis

Dawra M. and Priti, (2012) This study presented a new algorithm that will split large array in sub parts and then all sub parts processed in parallel using existing sorting algorithms and finally outcome would be merged To process sub parts in parallel multithreading has been introduced.

Faujdar N. and Ghrera S., (2016) this study mentioned the roadmap of research direction of a GPU based sorting algorithms and the various research aspects to work on GPU based sorting algorithms. They tested and compared the parallel and sequential (Merge, Quick, Count and Odd-Even sort) using dataset. The testing of parallel algorithms is done using GPU computing with CUDA.

The merging algorithms of Akl[1987] and that of Valiant[1975] and Kruskal [1983] (Kruskal improves Valiant's algorithm) differ only in their first phase. Since the time complexity of the second phase is 0 ((m + n)/p), and this is optimal as long as the first phase has a smaller complexity, both algorithms are optimal. Consequently it is in their differing first phase that improvements that can be made.

A shared memory MIMD version of a mergesort algorithm has been analyzed by Youssef and Evans [1987]. The algorithm evenly divides the N elements to be sorted into M subsets, and sorts each subset by an efficient sequential algorithm. Then each pair of sorted lists is merged using any free processor and a binary search merge algorithm to form a sorted list of double length. When there are fewer than p (number of processors) tasks left some processors are idle,At the final step all processors are idle but one.

Quinn [1988] designed a p-way quick/merge algorithm. Each of p processor sorts a contiguous set of about size N /p using sequential quicksort. Then p — 1 evenly-spaced keys from the first sorted list are used as divisors to partition each of the remaining sorted sets into p sub lists. Each processor *, 1 <»< p, performs a p way merge of it h sorted sub lists. This version removes the starvation problems of the Youssef and Evans's algorithm. The size of partitions to be assigned to each processor might be very different which can still create substantial starvation at the end.

Parallel Quicksort (PQ) The Quicksort algorithm is considered to be the most efficient general-purpose sorting algorithm, as both empirical and analytic studies have shown. It is a comparative, interchange algorithm basal on recursively partitioning the original data set into smaller subsets. The nature of this algorithm appears suitable to be implemented on a parallel multiprocessor. PQ is a parallel version of Quicksort. It has been implemented by several authors Men, EvYa5, MoSt871. However, their basic results are similar.

3. Research Design

3.1 Overview of Sequential Bucket Sort

It's a linear time sorting algorithm on the case of average over the interval [0, 1). It suggest that the input size is generated by a random process is uniformly distributed.

```
SORT_BUCKET (A)

1. n ← arraysize [A]
2. For i = 1 to n do
3.     Insert A[i] into list B[n A[i]]
4. For i = 0 to n-1 do
5.     Sort list B with Insertion sort
6. Concatenate the lists B[0], B[1], . . B[n-1] together in
   order.
```

Figure 1. Pseudo code Sequential Bucket sort

3.2 Sequential Bucket Sort Description

The main concept behind Bucket sort is to divide the [0, 1) interval into an equal-sized buckets which are subintervals, and then the n input numbers are distributed into the subintervals buckets. Whereas the array of inputs which is uniformly distributed over the interval (0, 1), many numbers are not expected to go down into every bucket. Toget the output from bucket sort the algorithm sort the numbers in every bucket and after that it go through the bucket which have order numbers, then after that it show the list of numbers in each of them Robert S. (1998).

For each element in n-element array A that an s between $0 \leq A[i] \leq 1$. And an auxiliary array B [0... n -1] for the buckets.

3.3 Sequential Bucket sort Analysis

Lines from 1 to 6 except line 5 take O (n) time in the worst case. The running time for bucket algorithm in line 5 is O (n)

The bucket sort algorithm time is also determined by the insertion sort of line 5 that has time of O (n2), when sorting the elements then the total complexity time will be

$$n-1\sum i=0\ O(E[2ni]) = O\ n-1\sum i=0\ (E[2ni])$$

And to know the value of the summation, we will determine every random number n distribution

There is n buckets and n elements. The probability that a given element falls in a bucket B[i] is 1/n. That's why, the probability will have a distribution that is binomial.

The mean of this distribution is: $E[ni] = np = 1$
The variance of this distribution is: $Var[ni] = np(1- p) = 1- 1/n$

For a set of random input matrix:

$$\begin{aligned} E\ [2ni] &= Var[ni] + E2[ni] \\ &= 1 - 1/n + 12 \\ &= 2 - 1/n \\ &= \Theta(1) \end{aligned}$$

So the insertion sort line will have O (n) time complexity and bucket sort will have order of O(n+m).

3.4 Measures and Covariates

Include in the Method section information that provides definitions of all primary and secondary outcome measures and covariates, including measures collected but not included in this report. Describe the methods used to collect data (e.g., written questionnaires, interviews, observations) as well as methods used to enhance the quality of the measurements (e.g., the training and reliability of assessors or the use of multiple observations). Provide information on instruments used, including their psychometric and biometric properties and evidence of cultural validity.

3.5 Parallel Bucket Sort

Given the number of components to be sorted and p be the number of procedures. At first, each procedure is given a block of n/p components, and the number of buckets is chosen to be m = p. The parallel detailing of buckets sort comprises of three stages. In the initial step, each process divides its block of n/p blocks into p sub-blocks, one for each of the p buckets. This is conceivable in light of the fact that each procedure knows the interval (a, b) and in this way the interval for each bucket. In the second phase, each procedure sends sub-blocks to the suitable procedures. After this step, each procedure has just the components having a place with the elements allocated to it. In the third phase, each procedure sorts its bucket inside by utilizing an optimal sequential sorting algorithm Kapur E., et al. (2012).

Utilizing bucket sort A specimen of size s is chosen from the n-component grouping, and the scope of the buckets is specifying by sorting the example and picking m - 1 components from the outcome.

These components called splitters which partition the sample into m same-sized buckets. After that, the calculation continues in the same path as the bucket sort. The execution of test sort relies on the sample size estimate s and the way it is chosen from the n-element sequence.

Consider a splitter determination scheme that ensures that the same quantity of components in each bucket. n is the number of components to be sorted and m be the number of buckets. Then sort each block by using quicksort. Each block selects spaced elements of m-1. For m(m - 1) elements are selected from the blocks represent the sample to specify the buckets. This assurances that the element numbers in each bucket is under 2n/m Manwade K. B. (2010).

in bucket sort, m = p; therefore, toward the end of the calculation, each procedure contains just the components having a single bucket. Each procedure is allotted a piece of n/p components, which it sorts consecutively. It then picks p - 1 equally dispersed components from the sorted block. Each procedure sends its p - 1 test components to one process – P0. Prepare P0 then successively sorts the p(p - 1) sample components and chooses the p - 1 splitters. Lastly, prepare P0 broadcasts the p - 1 splitters to the other procedures. Presently the calculation continues in a way indistinguishable to that of bucket sort.

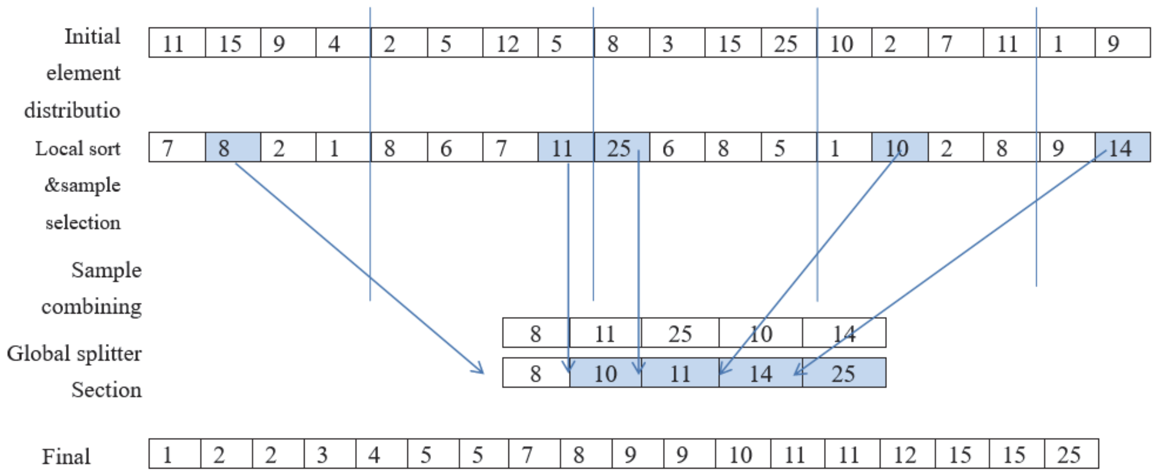

Figure 2. Swapping of elements in Parallel Bucket Sort

Analysis now the intricacy of sample sort on a message-passing PC with p procedures and O (p) separation data transfer capacity is examined.

The inner sort of n/p components requires time ((n/p) log(n/p)), and the selection of p - 1 sample components requires time (p). Sending p - 1 components to prepare P0 is like an accumulate operation, the time required is (p2). The opportunity to inertly sort the p (p - 1) test components at P0 is (p2 log p), and an opportunity to choose p - 1 splitters is (p). The p - 1 splitters are sent to the other procedures by utilizing one-to-all broadcast (Section 4.1), which requires time (p log p). Each procedure can embed these p - 1 splitters in its local sorted block of size n/p by performing p - 1 binary searches. Each procedure in this manner segments its block into p sub-blocks, the time required for this dividing is (p log (n/p)). Each procedure then sends sub-blocks to the proper procedures that is, buckets. The correspondence time for this progression is hard to compute correctly, as it relies on upon the size of the sub-blocks to be communicated. These sub-blocks can differ arbitrarily between 0 and n/p. In this way, the upper bound on the correspondence time is O (n) + O (p log p) Valiant, L. G (1975).

3.6 Sequential Bubble Sort

Bubble sort, is a straightforward sorting calculation that repeats steps through the list to be sorted, looks at each combine of contiguous items and swaps them if they were in the wrong order. The pass through the list is repeated until no swaps are required, which demonstrates that the list is sorted. The calculation, which is an examination sort, is named for the way littler or bigger components "bubble" to the highest priority on the rundown. In spite of the fact that the calculation is basic, it is too moderate and unreasonable for most issues notwithstanding when contrasted with insertion sort. It can be viable if the input is generally in order request yet may have some out-of-order elements almost in position Alnihoud J. and Mansi R. (2010).

Bubble sort has the worst-case and average complexity with O(n2), where n is the quantity of things being sorted. There exist many sorting calculations with considerably better of worst-case or average complexity of O(n log n). Indeed, even other O(n2) sorting calculations, for example, insertion sort, has better performance than bubble sort. Along these lines, bubble sort is not a practical sorting algorithm when n is huge MansotraV.andSourabh K, (2011).,

The only huge advantage that bubble sort has over most other algorithms, even quicksort, however not insertion sort, is that the capability to distinguish that the list is sorted effectively is incorporated with the calculation. At the point when the list is sorted (best-case), the complexity of bubble sort is just O(n). On the other hand, most other algorithms, even those with better average-case complexity, perform their sorting process on the set and in this way are more complex. In any case, not exclusively does insertion sort have this mechanism as well, however it likewise performs better on a list that is substantially sorted (having few reversals).

Bubble sort should not be used in large collections. It won't be efficient in reverse-order lists. Kapur E. et al. (2012).

```
BubbleSort_Sequential(Int B :array M [int M]);
Int X;
{
 While x< n
{
for (X=1 ; M-1 M++)
{
if B[Xi]>B[X+1] then
SWAP(B[X], B[X+1]);
}
LOOP
}
```

Figure 3. The sequential version of bubble sort algorithm

3.7 Parallel Bubble Sort: Odd-Even Sort

The idea behind this calculation is to modify the correlations in however many independent comparisons as circumstances. Comparisons can be done in parallel if each processor is included in at most one compare-and-exchange. This can be accomplished if the processors are assembled into even/odd sets or odd/even combines. odd-even stage the odd procedures p contrast and trade their components and the even processors p + 1 even-odd stage The even procedures contrast and trade their components and the odd processors p + 1 Protopopov et al 2008).

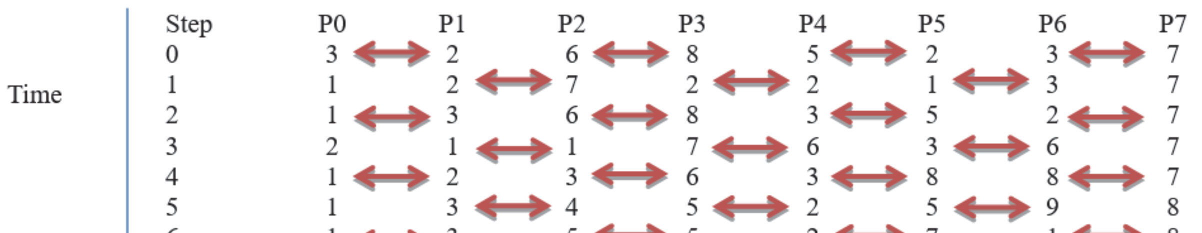

Figure 4. Swapping of elements in Parallel Bubble Sort

```
X = 1, 3, . . . , H – 1
% even phase send(A, X-1);
receive(Y, X-1);
if A < Y A = Y;
end if X <= H-3
% odd phase send(A, p+1);
receive(Y, X+1);
if A > Y A =Y;
end
end
```

```
X = 0, 2, . . . , H – 2
% even phase receive(A, X+1);
send(B, X+1);
if A < B B = A;
end
if X >= 2
% odd phase receive(A, X-1); send(B,
X-1);
if A > B
A = B;
end
end
```

Figure 5. Odd-Even Sort: Implementation

Internal sorting can be utilized when the number of elements is sufficiently little to fit into the main memory. If n expansive and it doesn't fit the main memory then supplementary storage must be utilized as a part of request to finish the sorting operation. The sorting strategies can be isolated into two classes by the complexity of the used algorithms. Blaise B, (2010).

The complexity of the sorting algorithms is written in the big-O notation. And they are expressed in view of the span of sets the calculation is keep running against. The big O notation represents to theoretical framework whereupon we can compare two or more algorithms. The two classes of the sorting calculations are O(n2), which incorporates the bubble sort are O(n2) which includes the insertion, bubble, selection, shell sorts and O(n log n), which includes the heap, quick sorts and merge. Mundra M. J and Pal M. B. (2015).

The sequential form of the bubble sort is the most established, the easiest and the slowest sorting calculation being used having a complexity level of O(n2). This is the reason that the technique is thought to be the most inefficient sorting calculation in like manner use. The bubble sort works by comparing every item in the list and the item beside it and swapping them if needed. The calculation repeats this procedure until it makes a pass completely through the list without swapping any item. This situation means, to the point that all things are in the correct order. This makes the bigger items to be moved (risen) to the end of the list while smaller items stay towards the start of the list. The sorting calculation closes its execution after a number of maximum n iterations of the main loop. The comparison numbers will be n-1 and the quantity of swaps can be 2 approximated by (n-1)/2. The minimum number of iterations is 1, for the situation when the components of the list are sorted. For this situation the calculation will perform n-1 comparisons and 0 swaps. In light of these outcomes we can infer that the many-sided quality level of the calculation for a typical array is O(n2) Mishra A. D and Garg D. (2008).

The parallel can be acquired in the event that we utilize the odd-even transposition technique that has n stages, each requiring n/2 compares and swaps. In the first stage, called odd stage, the components having odd indexes are compared with the neighbors from the right and the qualities are swapped when needed. In the even stage, the components having even indexes are compared with the elements from the right and the swaps are performed just when needed M. J and Pal M. B. (2015).

```
procedureOddEvenTranspositionSortAlg_Parallel (var O:vector;M:integer);
varX,T:integer;
begin for X:=1 to M do
if odd(X) then begin
for all T:=1 to M div 2 do
if O[2*T-1]>O[2*T] then
Exchange(O[2*T-1],O[2*T]);
end
else begin
 for all T:=1 to (M div 2)+(M mod 2)-1 do if O[2*T]>O[2*T+1] then Exchange(O[2*T],O[2*T+1]);
end;
end;
```

Figure 6. procedure Odd Even Transposition Sort Parallel

4. Results and Evaluation

In this part the performance evaluation of the results as long as the results are shown and discussed. The running time of the algorithms has been recorded on IMAN1 Zaina cluster supercomputer then the speedup and parallel efficiency performance has been evaluated for both of the algorithms the bubble sort and bucket sort algorithms. The hardware and software specifications which are used to implementation the algorithms are shown in table 1.

Table 1. The Hardware and Software Specifications

Hardware	Dual Quad Core Intel Xeon, CPU with SMP, 16 GB RAM
Software	Scientific Linux 6.4 with open MPI 1.5.4, C and C++ compiler.
Data Size	10000, 20000, 40000, 80000, 100000, 160000, 200000 Byte
Number of processors	2, 4, 8,16, 32, 64

4.1 Run Time Evaluation

The run time Evaluation for each of the implemented algorithms according to different data sizes are shown in Figure 7. The results are performed on 32 processor. It can be seen from the figure that the data size and time has an increase relation and as the data size increase the run time increases and this is because that the number of comparisons are increased and the needed for splitting and gathering increased, as the figure shows that the. Parallel bucket sort outperformed the bubble sort run time for both small and large input data size.

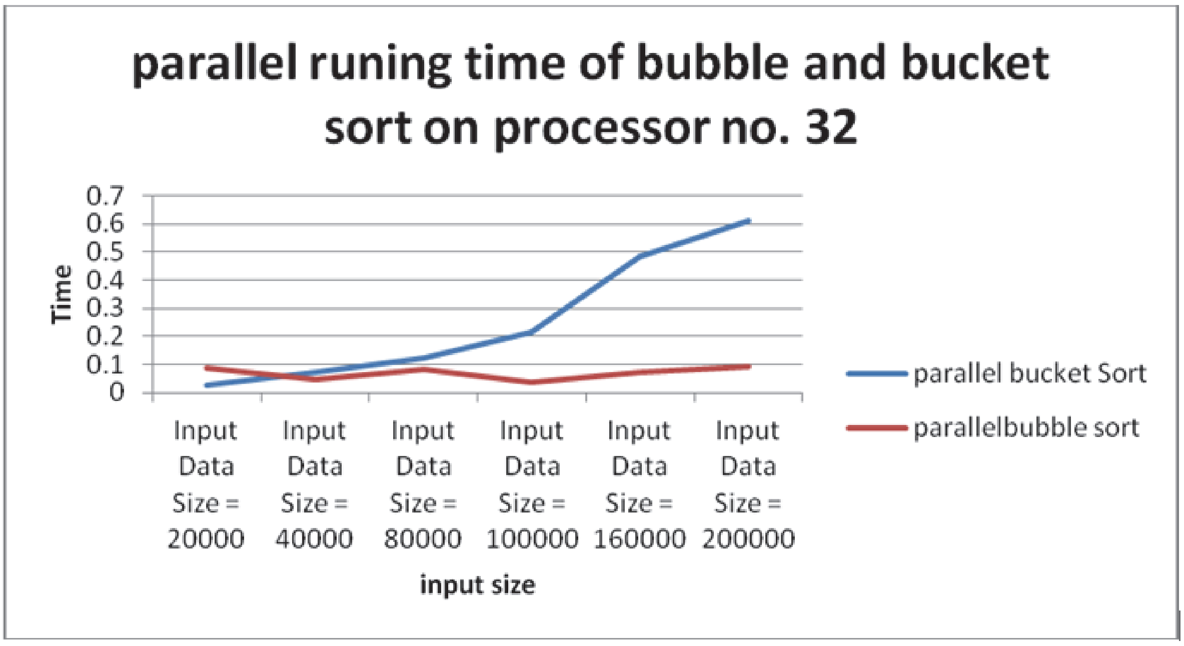

Figure 7. Parallel running time of bubble and bucket sort on processor no. 32

And the overall running time for both bubble and bucket sort on different data size (20,000, 200,000) correspond to small and large data size, respectively and different number of processors of (2, 4, 8, 16, 32) are shown in figure 8 and figure 9 respectively

As shown in figure 8 for both of the algorithms over small data size the run time decrease to a certain pint then after that it start to increase and this is due to the communication overhead increases on small data size the parallelism benefits is also decreased when the number of communication increase.

in figure 9 with large data size of 200000 As the number of processors increases the run time is decreased for the bucket sort and this is because of better load distribution between the processors, better parallelism and this as the no of processors increase from 2 to 32. But for bubble sort it almost zero the required parallel time for all the large data size and this is due to the benefit of parallelism

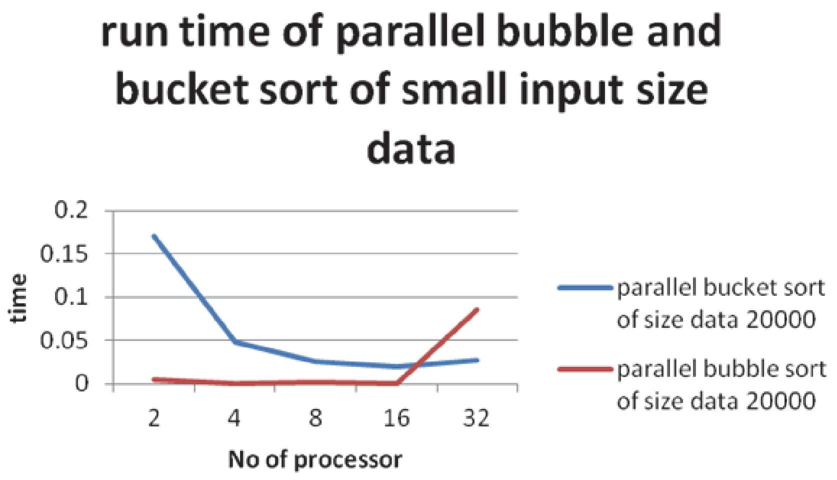

Figure 8. Run time of parallel bubble and bucket sort of small input size data

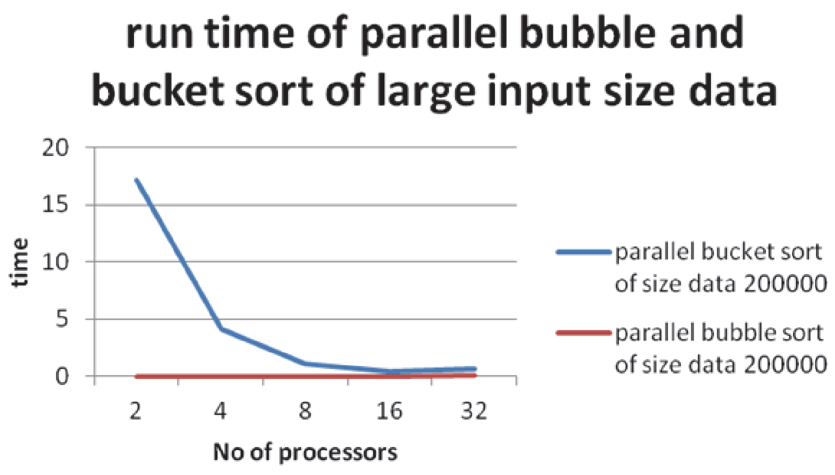

Figure 9. Run time of parallel bubble and bucket sort of large input size data

4.2 Sequential Bubble Running Time

The sequential running time of the bubble sort and bucket sort are shown in figure 10, as it seen from the results that the running time for the bubble sort increase very rapidly and it is slower than the bucket sort, these results will be used to calculate the speedup for both the bubble sort and bucket sort.

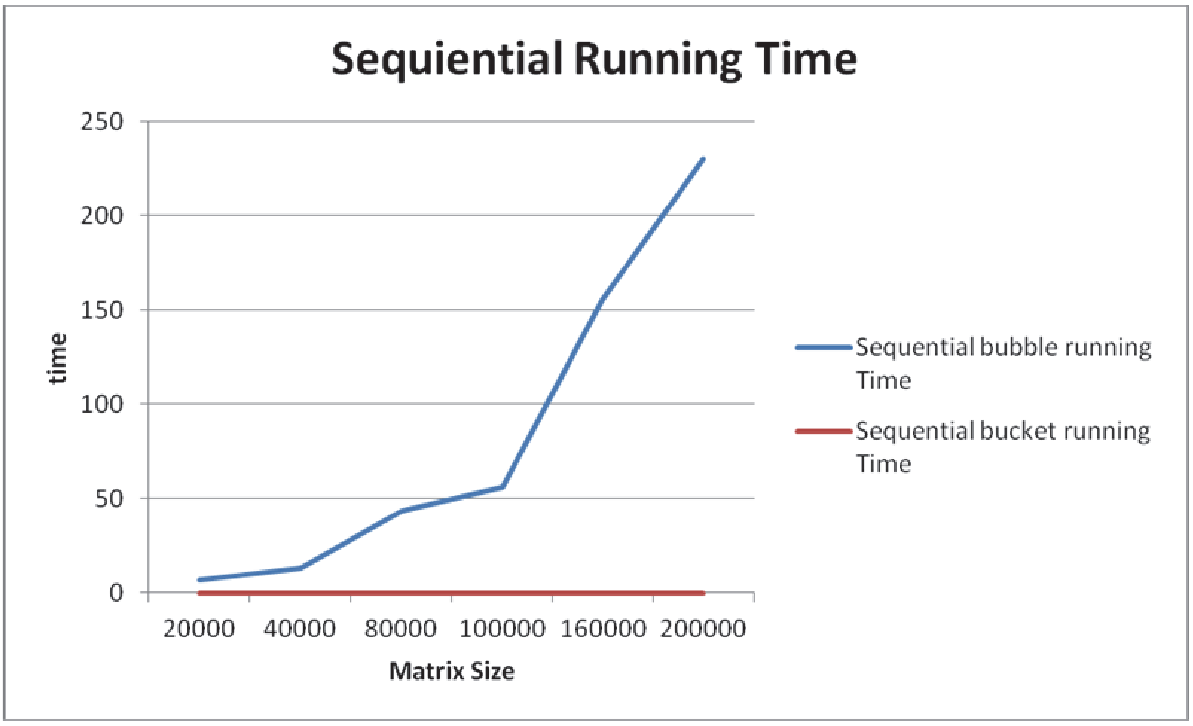

Figure 10. Sequential Running Time bubble and bucket sort

4.3 Multithreaded Bucket Sort Runtime

The run time of the multithreaded bucket sort are shown in figure 11. Over 10 threads and different data input size.

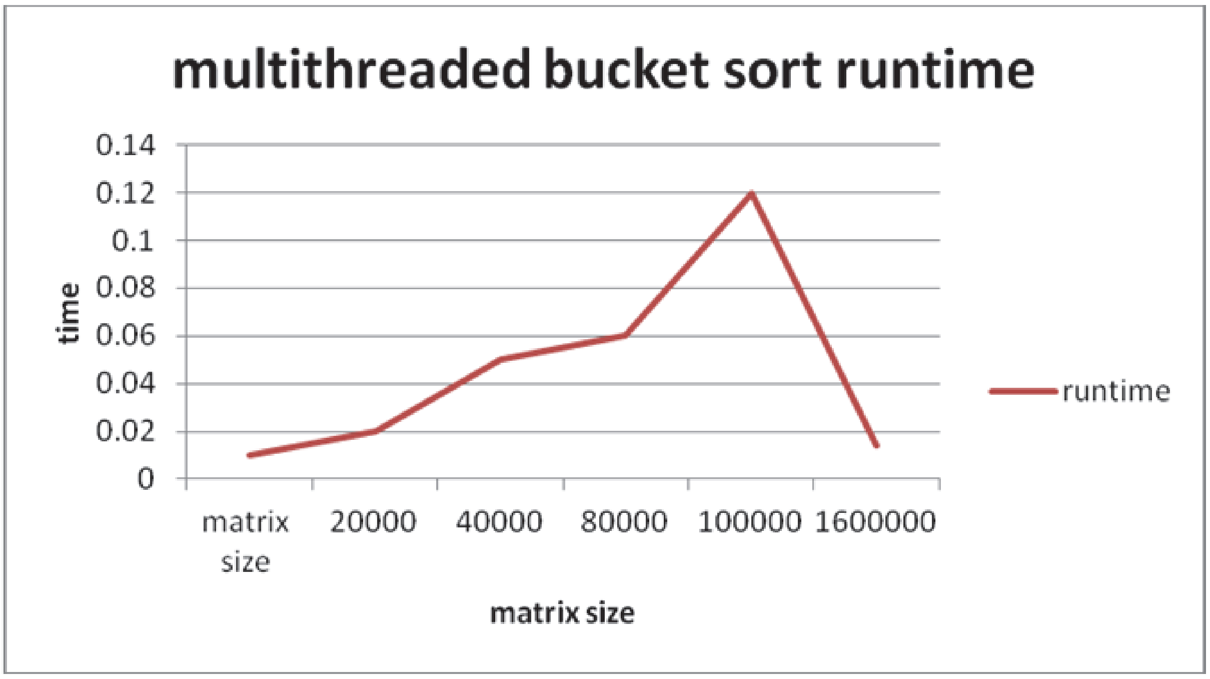

Figure 11. Multithreaded bucket sort runtime

4.4 Speedup Evaluation

The speedup for both of the algorithms has been calculated according to that the the speedup defined by the the ratio between the sequential time and the parallel time. The results for the speed up has been shown in Figure 12 and Figure13 which show the speedup of the two algorithms on the different number of processors (2, 4, 8, 16, 32) over the small data size and large data size (20000, 40000, 80000, 100,000, 160,000, 200,000)

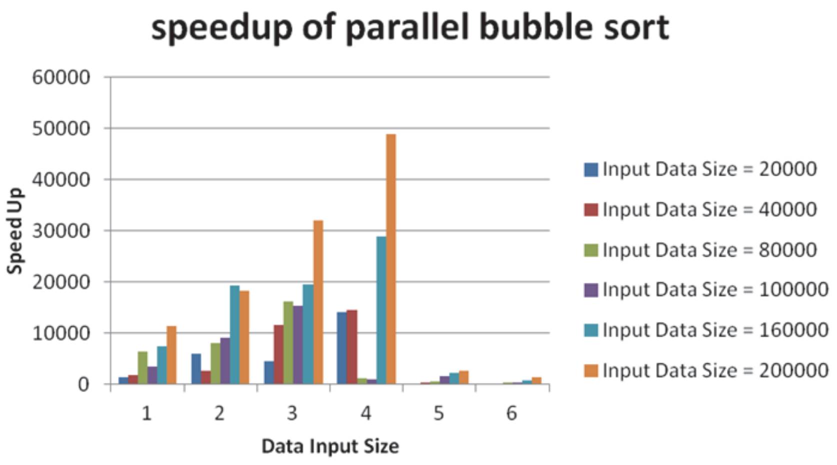

Figure 12. Speedup of parallel bubble sort

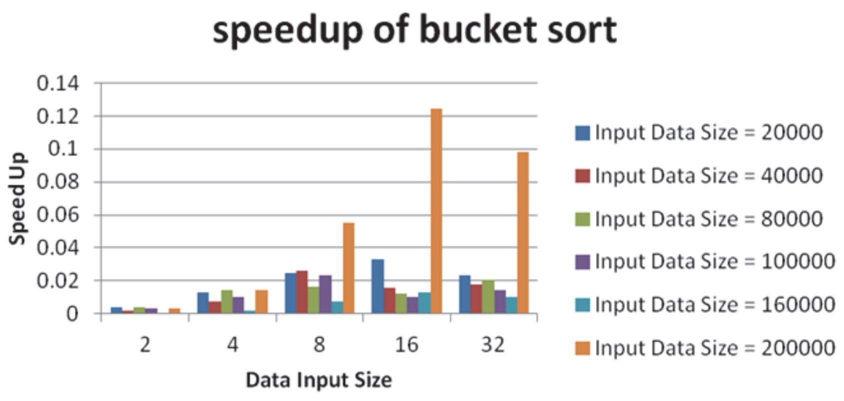

Figure 13. Speedup of bucket sort

Figure 14 and Figure15 which shows a comparison between the speedup of bubble and bucket sort at different number of processors (2, 4, 8, 16, 32) over the small data size and large data size respectively.

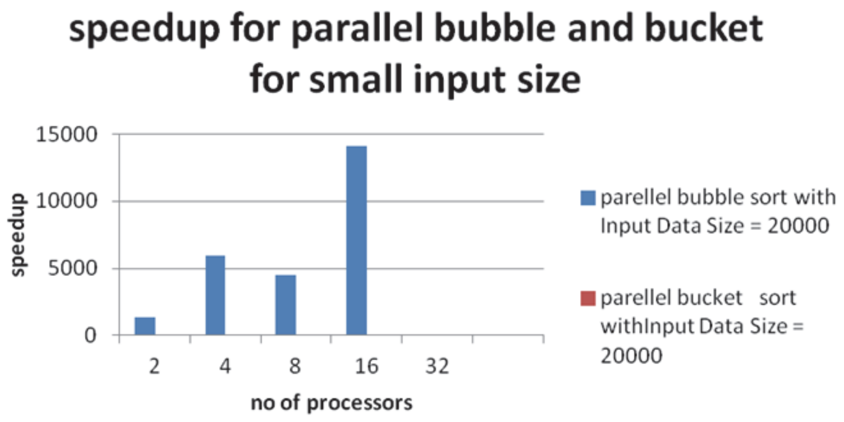

Figure 14. Speedup for parallel bubble and bucket for small input size

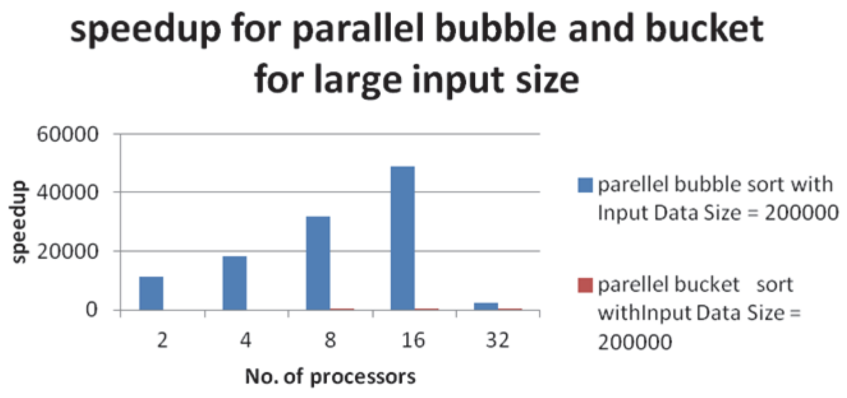

Figure 15. Speedup for parallel bubble and bucket for large input size

4.5 Parallel Efficiency Evaluation

The efficiency of both of the algorithms has been calculated, the efficiency is the ratio between speedup, the results of the efficiency is shown in Figure 16 and Figure 17 which show the parallel efficiency for both algorithms according to different input size 200000, the and different number of processors

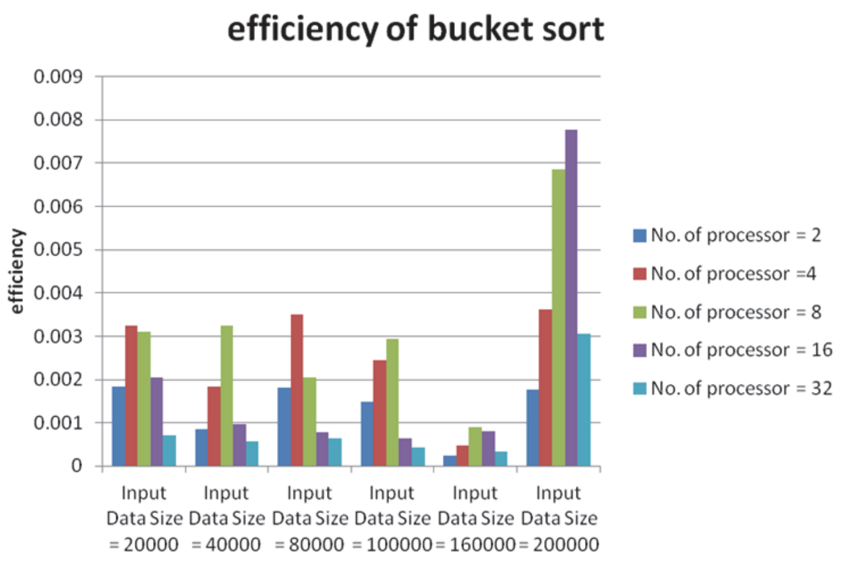

Figure 16. Efficiency of bucket sort

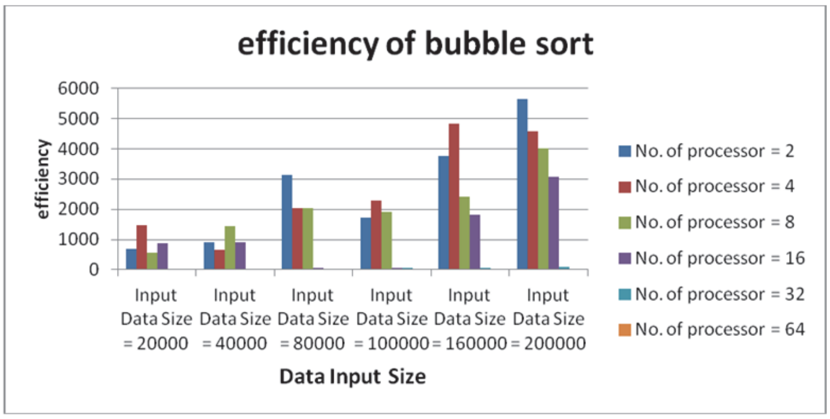

Figure 17. Efficiency of bubble sort Conclusion

5. Conclusion

This paper evaluate the performance two sorting algorithm bucket and bubble sort and evaluate the difference in performance in aspect of speed up and efficiency, the two algorithms has been tested on IMAN1 super computer with different evaluate input size and different number of processors. And the results has been discussed and represented which shows that he performance of runtime for the bubble and bucket sorting algorithms has been effectively reduced by the parallel computing over the large data size of 200000 and the number of processor of 64 get the lowest running time, and the parallel performance was better than the sequential.

References

Akl, S. G. (1985). Parallel Sorting Algorithms, Academic Press, New York. Well written. An early book.

Alnihoud, J., & Mansi, R. (2010). An Enhancement of Major Sorting Algorithms. *International Arab Journal of Information Technology, 7*(1), 55-62.

Blaise, B. (2010). Introduction to Parallel Computing", Lawrence Livermore National Laboratory. https://doi.org/10.1.1.363.5046

Chaarawi, M., Squyres, J. M., Gabriel, E., & Feki, S. (2008). A tool for optimizing runtime parameters of open mpi," in Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. Berlin, Heidelberg: SpringerVerlag, pp. 210-217. https://doi.org/10.1007/978-3-540-87475-1_30

Chand, S., Chaudhary, T., & Parveen, R. (2011). Upgraded Selection Sort. *International Journal on Computer Science and Engineering (IJCSE), ISSN: 0975-3397, 3*(4), 1633-1637.

Dawra, M., & Priti (2012). Parallel Implementation of Sorting Algorithms. International Journal of Computer Science Issues, *9*(4), No 3, 164-169.

Dodescu, G., Oancea, B., & Răceanu, M. (2002). Parallel Processing", Economic Publishing House, Bucharest.

Evans, D. J., & Dunbar, R. C. (1982). The Parallel Quicksort Algorithm Part 2- Simulation. *Inter. J. Computer Math., 12*, 125-133.

Faujdar, N., & Ghrera, S. (2016). Performance Analysis of Parallel Sorting Algorithms.

Inselberg, A. (2004). Parallel Coordinates, Springer. ISBN:0-8186-2083-8.

Jordan, H. F. (2002). Fundamentals of Parallel Computing, Prentice Hall

Joseph, J., & Fellenstein, C. (2003). Grid Computing. *Prentice Hall, 43*(4), 2004. https://doi.org/10.1147/sj.434.0624

Kapur, E., Kumar, P., & Gupta, S. (2012). Proposal of a two way sorting algorithm and performance comparison with existing algorithms. *International Journal of Computer Science, Engineering and Applications (IJCSEA), 2*(3), 61-78.

Kruskal, C. P. (1983). Searching, Merging, and Sorting in Parallel Computation. *IEEE Trans, on Comp., c-32*(10), 942-946, Oct..

Mansotra, V., & Sourabh, K. (2011). Implementing Bubble Sort Using a New Approach", Proceedings of the 5th National Conference; INDIACom, New Delhi.

Manwade, K. B. (2010). Analysis of Parallel Merge Sort Algorithm. *International Journal of Computer Application, 1*(19), 66-68.

Mishra, A. D., & Garg, D. (2008). Selection of Best Sorting Algorithm. *International Journal of Intelligent Information Processing, 2*(2), 363-368.

Moses, O. O. (2009). Improving the performance of bubble sort using a modified diminishing increment sorting. *Scientific Research and Essay, 4*(8), 740-744.

Mundra, M. J., & Pal, M. B. (2015). Minimizing Execution Time of Bubble Sort.

Protopopov, B. V., & Skjellum, A. (2008). A multithreaded message passing interface (MPI) architecture: Performance and program issues. *Journal of Parallel and Distributed Computing, 61*(4), 449.

Quinn, M. J. (1988). Parallel Sorting Algorithms for Tightly Coupled Multiprocessors.

Sabot, G. (1995). High Performance Computing, Addison-Wesley.

Sedgewick, R. (1998). Algorithms in C. Addison Wesley, 273-279.

Sedgewick, R. (1998). Algorithms, Addison-Wesley. *Parallel Compute, 6*, 349-357.

Sen, S., & Chatterjee, S. (2000). Towards a theory of cache-efficient algorithms. 11th ACM Symposium of Discrete Algorithms, 829-838.

Sodhi, T. S., Kaur, S., & Kaur, S. (2013). Enhanced Insertion Sort Algorithm. *International Journal of Computer Applications, 64*(21), 35-39.

Valiant, L. G. (1975). Parallelism in Comparison Problems", SIAM J. Compute., 4(3), 348-355, Sept.

Wyrzykowski, R. (2004). Parallel Processing and Applied Mathematics, Springer.

Yousif, N. Y., & Evans, D. J. (1986). The Parallel Odd- Even Merge Algorithm. *Intern. J. Comp. Math., 18*, 265-273.