International Journal of Advanced Science and Technology
Vol.95 (2016), pp.57-72
http://dx.doi.org/10.14257/ijast.2016.95.06

Performance Evaluation of Parallel Sorting Algorithms on IMAN1 Supercomputer

Maha Saadeh[1]*, Huda Saadeh[2] and Mohammad Qatawneh[3]

Department of Computer Science, King Abdullah II School for Information Technology, University of Jordan, Amman, Jordan
[1]maha.k.saadeh@gmail.com, [2]huda.saadeh@gmail.com, [3]mohd.qat@ju.edu.jo

Abstract

Many sorting algorithms have been proposed and implemented in previous years. These algorithms are usually judged by their performance in term of algorithm growth rate according to the input size. Efficient sorting algorithm implementation is important for optimizing the use of other algorithms such as searching algorithms, load balancing algorithms, etc. In this paper, parallel Quicksort, parallel Merge sort, and parallel Merge-Quicksort algorithms are evaluated and compared in terms of the running time, speedup, and parallel efficiency. These sorting algorithms are implemented using Message Passing Interface (MPI) library, and results have been conducted using IMAN1 supercomputer. Results show that the run time of parallel Quicksort algorithm outperforms both Merge sort and Merge-Quicksort algorithms. Moreover, on large number of processors, parallel Quicksort achieves the best parallel efficiency of up to 88%, while Merge sort and Merge-Quicksort algorithms achieve up to 49% and 52% parallel efficiency, respectively.

Keywords: *Parallel Merge Sort, Parallel Quicksort, MPI, Supercomputer*

1. Introduction

The aim of sorting algorithm is to order ***N*** items in an efficient way considering run time, memory space, and data structure complexity. Different types of sorting algorithms have been implemented such as selection sort, insertion sort, bubble sort, Quicksort, merge sort, and heap sort (see Table 1) [1]. In 1960s computer manufacturers estimate that 25% of their computers running time was spent on sorting. As a results, it is important to enhance sorting algorithms and implement them efficiently to enhance the performance of applications with huge amount of data such as search engines [2].

In order to enhance the performance of sorting algorithms, parallel versions have been implemented which reduce the overall execution time and increase the fault-tolerance of sorting based applications [3 - 5]. The performance of these parallel algorithms depends on its implementation and the underlying architecture of the parallel machine [16-21].

The goal of this paper is to evaluate the performance of three different parallel sorting implementations, which are parallel Quicksort, parallel Merge sort, and hybrid Merge-Quicksort algorithms. The three algorithms are implemented using MPI which is a standard library for message passing that can be used to develop portable parallel programs using C, C++ or FORTRAN [6].

The evaluation is done in terms of the required running time, speedup, and parallel efficiency according to different data size and number of processors. The results were

* Corresponding author,
E-mail: maha.k.saadeh@gmail.com
Postal Code: Amman 11942 Jordan

ISSN: 2005-4238 IJAST

conducted using IMAN1 supercomputer which is Jordan's first and fastest supercomputer. It is available for use by academia and industry in Jordan and the region and provides multiple resources and clusters to run and test High Performance Computing (HPC) codes [7].

The rest of the paper is organized as follows; Section 2 summarizes some related works. Section 3 introduces sequential and parallel Quicksort. In Section 4, sequential and parallel Merge sort algorithms are discussed. Next, parallel Merge-Quicksort is discussed in Section 5. The evaluation results of the three algorithms are discussed in Section 6. Finally, Section 7 concludes the paper.

Table 1. The Time Complexity of Different Sorting Algorithms [1]

Sorting Algorithm	Best Case	Average Case	Worst Case
Insertion	O(n)	O(n^2)	O(n^2)
Selection	O(n^2)	O(n^2)	O(n^2)
Bubble	O(n^2)	O(n^2)	O(n^2)
Heap	O(n*log(n))	O(n*log(n))	O(n*log(n))
Merge	O(n*log(n))	O(n*log(n))	O(n*log(n))
Quick	O(n*log(n))	O(n*log(n))	O(n^2)

2. Related Works

Parallel sorting algorithms have been a rich area of research due to the increasing need to efficient, fast, and reliable sorting algorithms which can be implemented in parallel platforms. In [8] and [9] the authors propose a load-balanced merge sort algorithm that utilizes all processors throughout the computation. It evenly distributes data to all processors in each stage. Thus, every processor is forced to work in all phases. In [10-12] different parallel Quicksort algorithms are proposed. In [10] a comparison between sequential and parallel versions of Quicksort algorithms is discussed. The authors in [11] propose a fast parallel Quicksort algorithm that can sort N data items in $\log n$ time using N processors. In [12] another fast parallel Quicksort algorithm is implemented and evaluated using the SUN enterprise 1000.

A study of common patterns in parallel sorting algorithms in terms of load balancing, keys distribution, communication, and computations is conducted in [13]. Another study based on qualitative and quantitative analysis of the performance of parallel sorting algorithms on modern multi-core hardware can be found in [14].

3. An Overview of Sequential and Parallel Quicksort

In 1960 Quicksort was developed by Tony Hoare [1]. It is a sorting in place algorithm that recursively calls itself to sort the items in an array. The algorithm calls itself tow times with input length depends on the partition procedure. Obviously, the main operations are done in the partition procedure; firstly, it chooses a pivot value to sort the items into two sub lists; upper and lower. All items in the former list are greater than or equal to the pivot value. The later list contains the items that are less than the pivot. Then, the Quicksort algorithm is called recursively on each sub list. The same process is repeated to each sub list until partitioning cannot be performed. Finally, each sub list will contain a sorted part of the original list.

The chosen of the pivot element can be done in various ways; 1- It could be the leftmost or rightmost item in the list. 2- It can be chosen randomly. 3- The median value between the leftmost, the rightmost, and the middle items.

In this section, an overview of Quicksort algorithms is discussed. Section 3.1 discusses the sequential algorithm and parallel Quicksort is discussed in 3.2.

3.1. Sequential Quicksort

As discussed previously, the main operation of Quicksort takes place in partition procedure. The Pseudo code for sequential Quicksort is shown in Figure 1.

```
Partition(List, left, right)
{
   p=left;
   pivot=List[left];
   for(i = left + 1;i <= right; i++)
   {
      if(pivot > List[i])
      {
         List[p] = List [i];
         List [i] = List [p+1];
         List [p+1] = pivot;
         p = p + 1;
      }
   }
   return p;
}
```

```
Quicksort (List, left, right)
{
  if(left < right)
  {
    Q = Partition (List, left, right)
    Quicksort (List, left, q-1)
    Quicksort (List, q+1, right)
  }
}
```

Figure 1. Sequential Quicksort Pseudo Code

3.1.1. Sequential Quicksort Description

Sequential Quicksort algorithm is a recursive procedure works as follows:

1. Select one of the items as a pivot.
2. Divide the list into two sub lists: a lower list containing numbers smaller than the pivot, and an upper list containing numbers larger than or equal to the pivot.
3. The lower list and the upper list recursively repeat the procedure to sort themselves.
4. The final sorted result is the concatenation of the sorted lower list, the pivot, and the sorted upper list.

To understand the sequential Quicksort we will illustrate the example shown in Figure 2. The list consists of five numbers and the goal is to sort them in ascending order. The inputs are {3, 2, 1, 5, 4}, and we choose the pivot to be the leftmost item {3}. Firstly, the partition procedure will arrange the list into two sub lists at the third index (since the pivot is the middle value); lower sub list {2, 1} and upper sub list {5, 4}. Then the Quicksort algorithm will be performed to each sub list. In the lower sub list the pivot is {2} and the partition index is 2. For the upper sub list the pivot is {5} and the partition index is 5. No more partitioning can be performed for both lower and upper sub lists and thus the resulted list is already sorted. Note that due to the sequential nature the lower sub list will be sorted before the upper one.

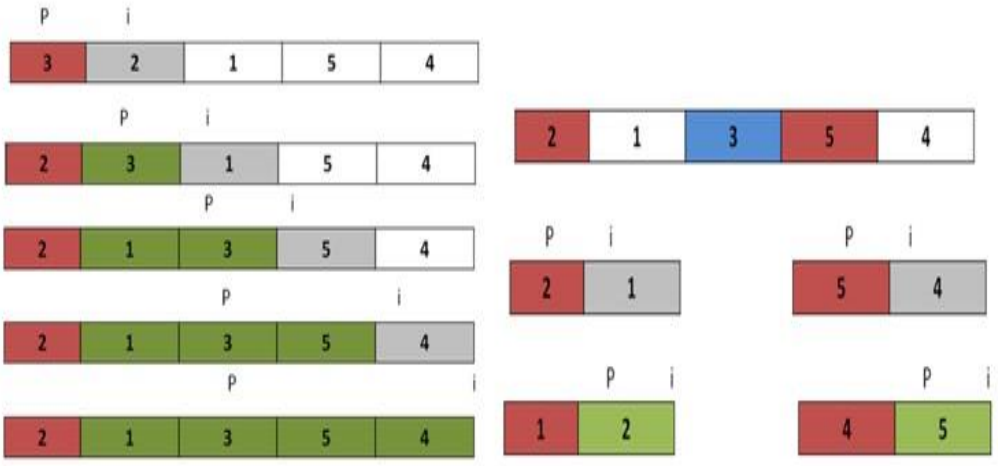

Figure 2. Sequential Quicksort Example

3.1.2. Sequential Quicksort Analysis

Sequential Quicksort has better run time complexity compared with other sorting algorithms such as selection, insertion, and bubble sorting algorithms. Merge and heap sorting algorithms have the same growth rate as in Quicksort except for the worst case run time. However worst case is rarely to occur [15]. In this section we will analyze the time complexity of the sequential Quicksort according to the best, average and worst cases.

The worst case of the Quicksort algorithm is presented in Figure 3. This case occurs when the list is already sorted and the pivot is the left or right most item. As shown in the figure, in each call for partitioning procedure nothing will be moved. The time complexity is $O(n^2)$.

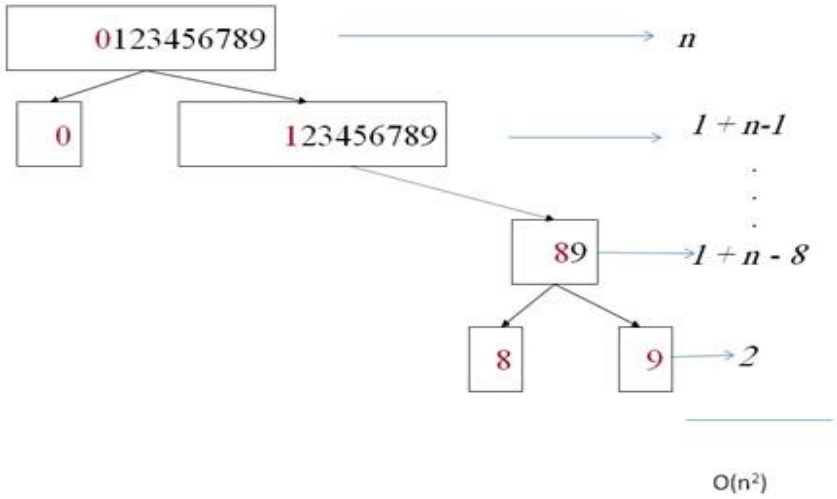

Figure 3. Quicksort Worst Case

Figure 4 and Figure 5 illustrate the complexity of Quicksort in the best case and average case, respectively. In the best case the partitioning is always balanced (the partition procedure returns the middle index). Thus, in each recursive call for Quicksort algorithm the lower and the upper sub lists will contain the same number of items which is the half of the items in the previous call. This means that the

recursive calls will go for log n times each take *O(n)* for partitioning the sub list. As a result the time complexity in the best case is *O(n log n)*. In the average case an alternate between good and bad partitioning occurs results in the complexity of O(n log n).

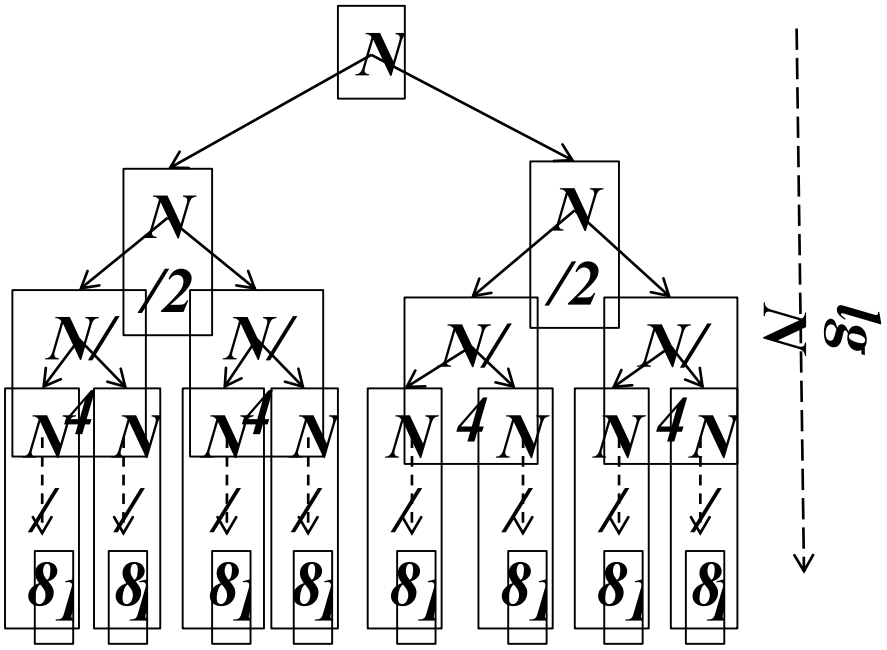

Figure 4. Sequential Quicksort Best Case.

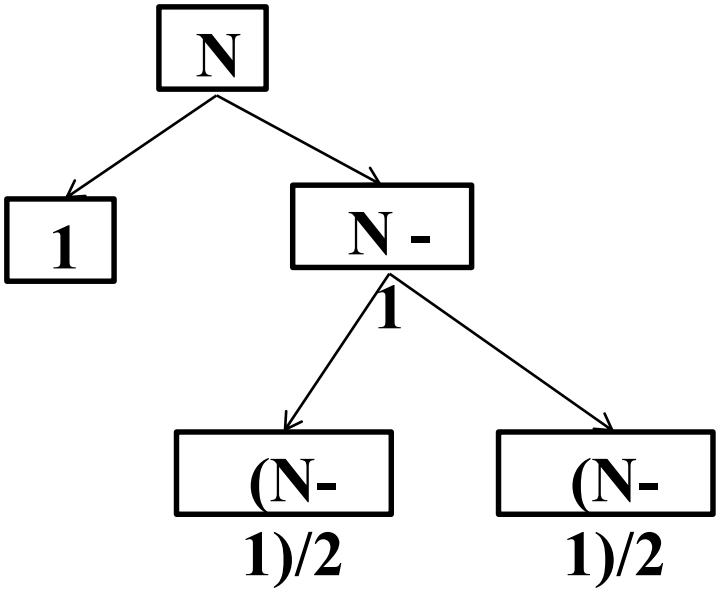

Figure 5. Sequential Quicksort Average Case

3.2. Parallel Quicksort

Parallel Quicksort algorithm can be implemented in different ways. In this section we will summarize the one we used for our implementation. The main idea is to do the partitioning on the original list using a single processor, and then assign the upper sub list to another processor for further partitioning. The steps of this method are listed in Table 2. In the next subsections, the analysis of parallel Quicksort is discussed. For simplicity,

assume that the input size $n = 2^k - 1$, the number of processors $P = 2^m$ where $m < k$, and electrical links are used to connect the processors (Figure 6).

Table 2. Steps of Parallel Quicksort

Step	Action taken by each processor
1	Do partition on the assigned list, go to step 2.
2	Check if any $processor_i$ is available. If yes go to step 3, otherwise go to step 4.
3	Assign the upper sub list to $processor_i$ if its size is greater than 1, and mark $processor_i$ as a child. Then, go back to step 1.
4	Perform sequential Quicksort on the assigned list, and go to step 5.
5	Send results to parent processor. Then, go to step 6.
6	Exit

3.2.1. Partitioning Procedure Analysis

In the first partitioning of the original list the partition procedure will take $n - 1$ time to perform $n - 1$ comparisons (n – the pivot item). Assuming that partitioning is always balanced, the lower and upper sub lists have the same number of items = $(n - 1) / 2$ and this will be the input size for the next partitioning. Taking $(n - 1) / 2$ items, the time for partition procedure is $((n - 1) / 2) - 1 = (n - 3)/2$ on each process with total $n-3$ comparisons (((n-3) / 2) × number of processors that do the partitioning). In the next round 4 sub lists will be running on 4 processors each with input size = $(n - 3) / 4$, partitioning time = $(n - 7) / 4$, and $n - 7$ comparisons. Generally round $_i$ has input size = $(round_{i-1}\ size - 1) / 2$, partition time = $(n – 2^i - 1) / 2^{i-1}$, and number of comparisons = $n – 2^i - 1$ where $i > 1$. This can be performed for log p. Then, the all processors will perform the sequential Quicksort since new processors are not available. According to this analysis the Partitioning time in parallel Quicksort for total log p rounds = $\left[\sum_{i=1}^{(\log p)+1} \frac{(n - 2^i - 1)}{2^{i-1}}\right]$, and the number of comparisons = $\left[\sum_{i=1}^{(\log p)+1} (n - 2^i - 1)\right]$.

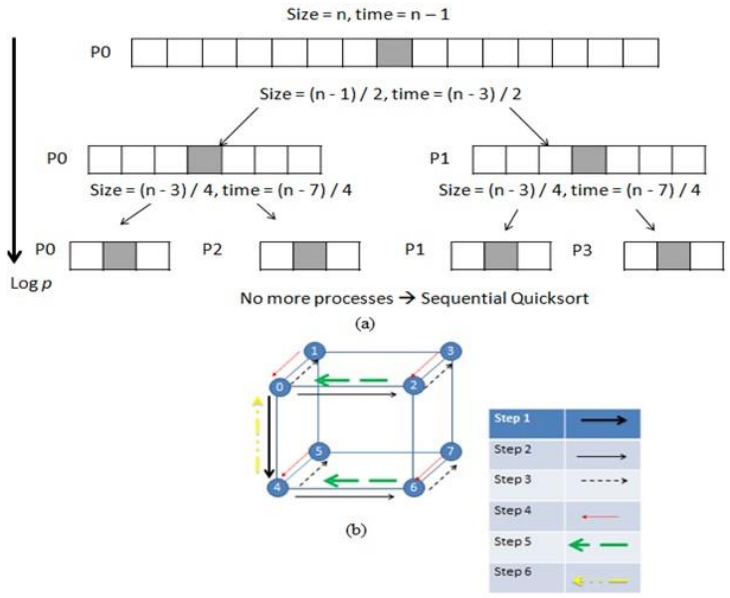

Figure 6. Parallel Quicksort Analysis. (a) Tree View and (b) Hypercupe View

3.2.2. Parallel Analysis

Parallel Quicksort is analyzed according to the number of communication steps, complexity, speed, and execution time.

***Communication steps**:* this includes the number of steps required for data splitting and results gathering. As we can see in Figure 6, we need 2 steps to scatter the data among 4 processors and 3 steps to scatter it among 8 processors. As a result, the number of communication steps that are required to scatter the data depends on the number of processors *P*, which is $\log P$. on the other hand, we need same number of communication steps to gather the results from all processors. So the total number of communication steps is $2 \times \log P$.

***Complexity**:* this is the time required to perform Quicksort locally on each processor and the time of partition procedure. In the best case, each processor will sort *n/p* of elements using sequential Quicksort at the same time. So the time required for all processors to sort the data is $= O(\frac{n}{P} \times \log \frac{n}{P})$. In addition, the partition procedure take $\left[\sum_{i=1}^{(\log p)+1} \frac{(n - 2^i - 1)}{2^{i-1}}\right]$ time, so the total complexity is $O(\frac{n}{P} \times \log \frac{n}{P}) + \left[\sum_{i=1}^{(\log p)+1} \frac{(n - 2^i - 1)}{2^{i-1}}\right]$.

***Speed**:* this is the communication steps times the speed of the electrical links. Assuming that the speed of electrical links = 250Mb/s, the speed is $2 \times \log P \times 250$ Mb/s.

***Execution time**:* this is the complexity of sorting + communication time. As discussed previously, the complexity of parallel sorting is $O(\frac{n}{P} \times \log \frac{n}{P}) + \left[\sum_{i=1}^{(\log p)+1} \frac{(n - 2^i - 1)}{2^{i-1}}\right]$.The communication time depends on the data that is transmitted over the link in each step. We mentioned previously that we need $\log P$ communication steps to scatter the data. In the best case the data is divided into equal parts. So, the data size in the first step = $\frac{n}{2}$, in the second step = $\frac{n}{4}$, in the ith step = $\frac{n}{2^i}$. As a result, the total is $\sum_{i=1}^{\log P} \frac{n}{2^i}$. In addition, we need to add the data size in the communication steps for data gathering. Thus, the total communication time is $2 \times \sum_{i=1}^{\log P} \frac{n}{2^i}$ and the total execution time is $(2 \times \sum_{i=1}^{\log P} \frac{n}{2^i}) + O(\frac{n}{P} \times \log \frac{n}{P}) + \left[\sum_{i=1}^{(\log p)+1} \frac{(n - 2^i - 1)}{2^{i-1}}\right]$.

4. An Overview of Sequential and Parallel Merge Sort

Merge sort is another divide and conquer based sorting algorithm. In this section both sequential and parallel Merge sort algorithms are discussed.

4.1. Sequential Merge Sort

Merge sort depends on dividing a list of data into smaller sub lists then merging those sorted lists into one sorted list. This process is implemented recursively by dividing the list each time into two smaller sub lists each with *n/2* data size. Then, re-dividing each of those sub lists again until each sub list has one element. After that, the algorithm will merge all sub lists into one sorted list (see Figure 7). The dividing process forms a binary tree in which the merge process is done in a bottom-up manner. An example of Merge sort is illustrated in Figure 8.

4.2. Parallel Merge Sort

In parallel Merge sort, input elements are divided into two equal parts and each part is processed by a separate processor. Like parallel Quicksort, each division is mapped to a new processor until every processor has its own data to be sorted. This

division forms a tree which can be implemented as a hypercube. After division, all processors will perform sequential Merge sort to sort their own *n/p* data locally, where *p* is the number of processors and n is the number of items. To gather the results, each processor will merge its own results with its parent results in a bottom up manner. Table 3 lists the main steps of parallel Merge sort algorithm.

In the next sub sections, the analysis of parallel Merge sort is discussed in details. For simplicity, we will assume that the input size $n = 2^k$, the number of processors $P = 2^m$ where $m < k$, and electrical links are used to connect the processors.

Table 3. Steps of Parallel Merge Sort

Step	Action taken by each processor
1	Divide the data into two equals parts; left and right. Then, go to step 2.
2	Check if any $processor_i$ is available. If yes go to step 3, otherwise go to step 4.
3	Assign the right part to $processor_i$, and mark $processor_i$ as a child. Then, go back to step 1.
4	Perform sequential Merge sort on the assigned data part, and go to step 5.
5	Send results to the parent processor to be merged with its results. Then, go to step 6.
6	Exit

```
Mergesort(data, first, last)
  if (Length(data) > 1)
{
    Mid = Length(data) / 2
    Mergesort(data, first, Mid)
    Mergesort(data, Mid+1, last)
    Merge(data, first, Mid, last - Mid)
   }
End of Mergesort
```

```
Merge(data,first,n1,n2)
  counter1=0
  counter2=0
  counter=0
  while ((counter1 < n1) and (counter2< n2))
  {
     if (data[first + counter1) < data(first + n1 + counter2))
     {
      temp(counter) = data(first + (counter1))
      increase both counter and counter1
     }
     else
     {
      temp(counter) = data(first + n1 + (counter2))
      increase both counter and counter2
     }
  }
  // Copy any remaining entries in the left and right sub-arrays.
  while (counter1 < n1)
     temp(counter) = data(first + (counter1))
  while (counter2< n2)
     temp(counter) = data(first + n1 + (counter2))
  // Copy from temp back to the data array.
  for i = 0 to n1+n2
     data(first + i) = temp(i)
End of Merge
```

Figure 7. The Pseudo Code of Sequential Merge Sort

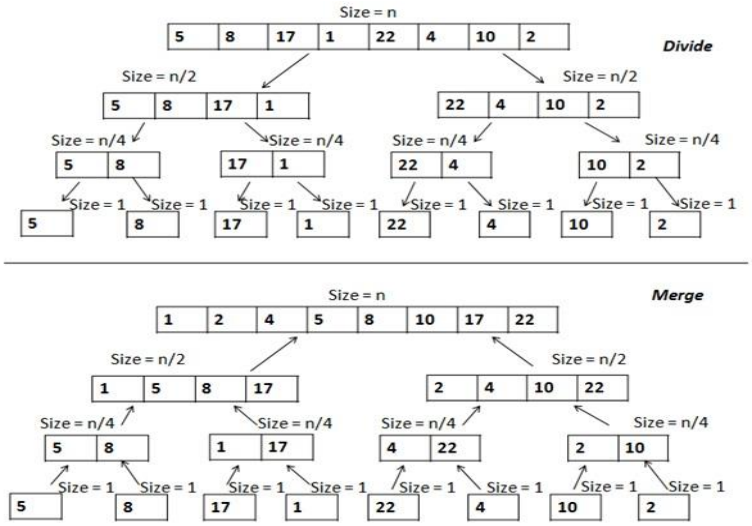

Figure 8. An Example of Sequential Merge Sort

4.2.1. Merge Procedure Analysis

Since data is divided among processors to form a binary tree, merging operation will take log n steps to merge the whole results in bottom up manner. If data size $n = 2^k$, then log $n = k$. In each step, the data size is doubled since it is been merged with the parent results which can be expressed as $\sum_{i=1}^{k} 2^i = \sum_{i=0}^{k} 2^i - 1$. Solving this equation the time complexity for merge procedure is O(n).

4.2.2. Parallel Analysis

As for parallel Quicksort, parallel Merge sort is analyzed according to the number of communication steps, complexity, speed, and execution time.

Communication steps: this includes the number of steps required for data division and results merging. The number of communication steps that are required to divide the data among processors depends on the number of processors P, which is$\log P$. on the other hand, we need same number of communication steps to merge the results from all processors. So the total number of communication steps is $2 \times \log P$.

Complexity: this is the time required to perform Merge sort locally on each processor and time for merge procedure. In the best case, each processor will sort n/p of elements using sequential Merge sort at the same time. So the time required for all processors to sort the data is $=\frac{\mathrm{n}}{\mathrm{P}} \times \log\frac{n}{P}$. As discussed previously, the merge procedure take O(n) time. As a result, the total is $\mathrm{O}(\frac{\mathrm{n}}{\mathrm{P}} \times \log\frac{n}{P})$ + O(n).

Speed*:* this is the communication steps times the speed of the electrical links. Assuming that the speed of electrical links = 250Mb/s, the speed is $2 \times \log P \times 250$ Mb/s.

Execution time: this is the complexity of sorting + communication time. As discussed previously, the complexity of parallel sorting is $\mathrm{O}(\frac{\mathrm{n}}{\mathrm{P}} \times \log\frac{n}{P})$ + O(n). The communication time depends on the data that is transmitted over the link in each step. We mentioned previously that we need $\log P$ communication steps to divide the data. The data is divided into two equal parts. So, the data size in the first step = $\frac{n}{2}$, in the second step = $\frac{n}{4}$, in the ith step = $\frac{n}{2^i}$. As a result, the total is $\sum_{i=1}^{\log P} \frac{n}{2^i}$. In addition, we need to add the data size in the

communication steps for data merging. Thus, the total communication time is $2 \times \sum_{i=1}^{\log P} \frac{n}{2^i}$ and the total execution time is $(2 \times \sum_{i=1}^{\log P} \frac{n}{2^i}) + (\frac{\mathrm{n}}{\mathrm{P}} \times \log \frac{n}{P}) + O(n)$.

5. An Overview of Parallel Merge-Quicksort

This algorithm is a combination of both Merge sort and Quicksort. It uses parallel Merge sort to divide the data among processors and sequential Quicksort to sort the data locally at each processor. The parallel analysis for this algorithm is the same as parallel Merge sort, since in the best case sequential Quicksort has the same complexity as the one for sequential Merge sort which is $O(\frac{\mathrm{n}}{\mathrm{P}} \times \log \frac{n}{P})$.

6. Performance Evaluation and Results

In this section, the results are discussed and evaluated in terms of running time, speedup and parallel efficiency performance metrics. IMAN1 Zaina cluster is used to conduct our experiments and open MPI library is used in our implementation of the following parallel sorting algorithms; Merge sort, Quicksort, and Merge-Quicksort. The algorithms are evaluated according to different input sizes and different number of processors. An average of multiple runs is considered to record the results. The hardware and software specifications along with the used implementation parameters are listed in Table 4.

Table 4. The Hardware and Software Specifications

Hardware specification	Dual Quad Core Intel Xeon CPU with SMP, 16 GB RAM
Software Specification	Scientific Linux 6.4 with open MPI 1.5.4, C and C++ compiler.
Data Size (Number of Input Items)	2^{22}, 2^{23}, 2^{24}, 2^{25}, 2^{26} (Element) 16, 32, 64, 128, 256 (MB)
Number of Processors	1, 2, 4, 8, 16, 32

6.1. Run Time Evaluation

Figure 9 show the run time for each algorithm according to different data sizes. All results are performed on 8 processors. As illustrated in the figure, as the data size increases, the run time increases due to the increased number of comparisons and the increased time required for data splitting and gathering. Parallel Quicksort has the best run time for both small and large data size followed by Merge-Quicksort algorithm. Finally, Merge sort algorithms has the worst run time results.

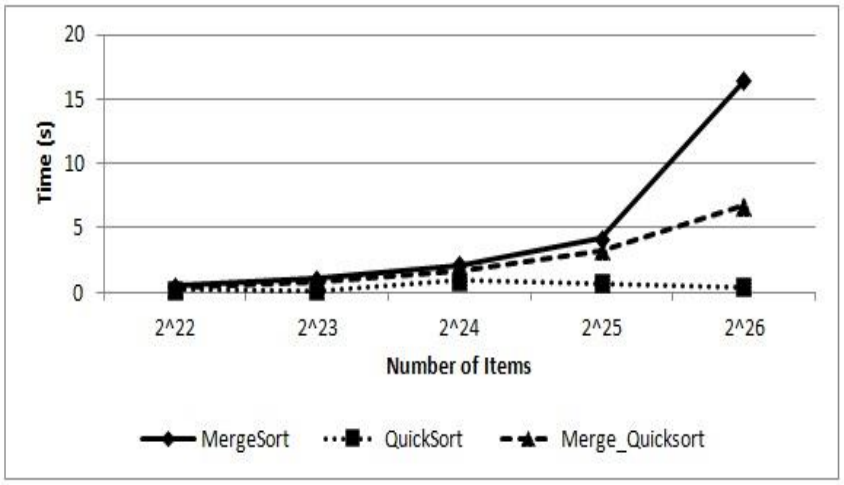

Figure 9. Comparison of the Three Algorithms According to Different Data Size

Figure 10 and Figure 11 illustrate the run time according to different number of processors including the sequential time ($p = 1$). We chose two different data size to conduct our experiments 2^{22} and 2^{26} correspond to small and large data size, respectively. The general behavior is the same for the three algorithms and can be summarized in the following points:

1- As the number of processors increases, the run time is reduced due to better parallelism, better load distribution among more processors. This is the case when moving from 2 to 8 or to 16 processors.

2- As the number of processor increases the run time is increases. This is because as the number of processors increases on specific data size, the communication overhead increases too, consequently, the benefits of parallelism are decreased. This is the case when moving from 16 to 32 processors. Due to this behavior that appears in the three algorithms, we did not use more than 32 processors.

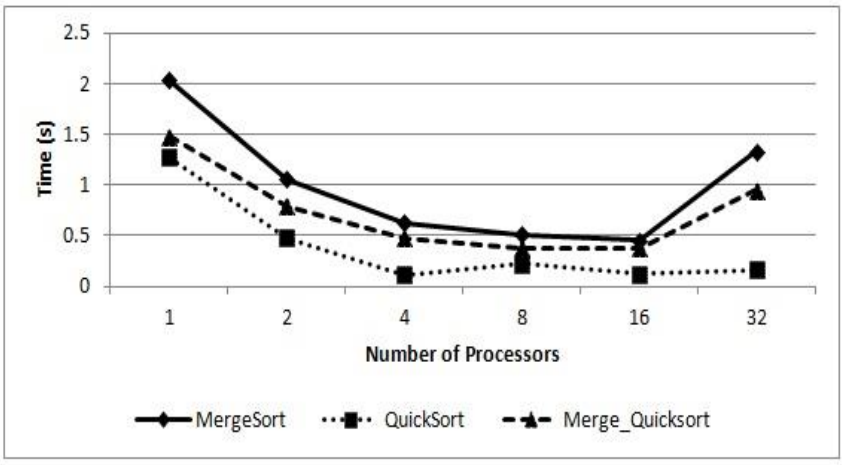

Figure 10. Comparison of the Three Algorithms According to the Number of Processors with 2^{22} Data Size

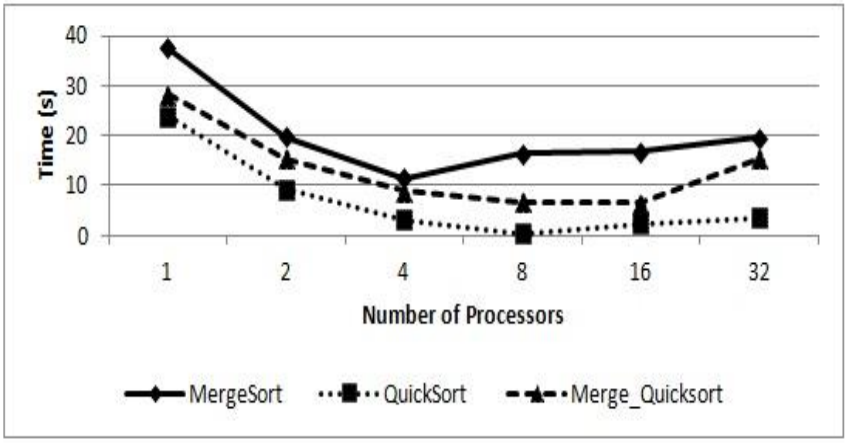

Figure 11. Comparison of the Three Algorithms According to the Number of Processors with 2^{26} Data Size

6.2. Speedup Evaluation

The speedup is the ratio between the sequential time and the parallel time. Figure 12 and Figure 13 illustrate the speedup of the three algorithms on 2, 4, 8, 16, and 32 processors with 2^{22} and 2^{26} data size, respectively. The results show that parallel Quicksort achieves the best speedups values, up to 10, especially on large number of processors.

6.3. Parallel Efficiency Evaluation

Parallel efficiency is the ratio between speedup and the number of processors. Figure 14 and Figure 15 show the parallel efficiency for each algorithm according to different number of processors. The results are corresponding to 2^{22} and 2^{26} data size, respectively. On small number of processors (2 and 4) Parallel Merge sort achieves up to 96% efficiency followed by Merge-Quick sort with up to 93%, while Quicksort achieves up to 72%. On the other hand, parallel Quicksort achieves the best efficiency on large number of processors (8, 16, and 32), comparing with the other algorithms, with up to 88%, while Merge and Merge-Quicksort achieve up to 49% and 52% efficiency, respectively. This is because Quicksort has better speedup values when applied on 8 or more processors.

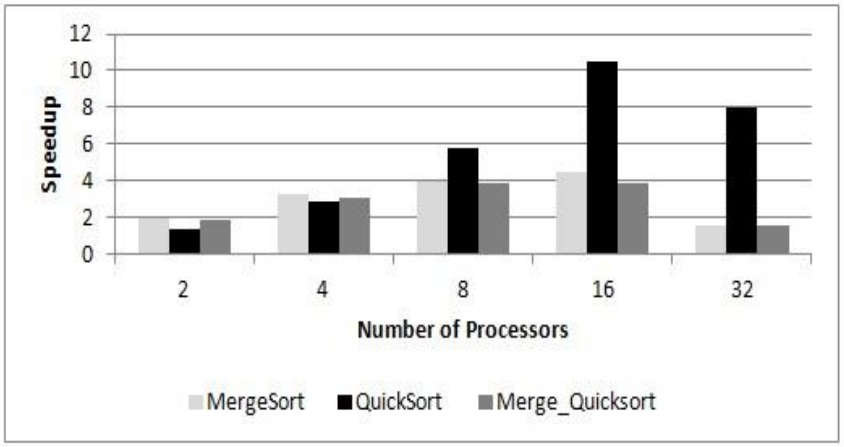

Figure 12. The Speedup of the Three Algorithms on Different Number of Processors with 2^{22} Data Size

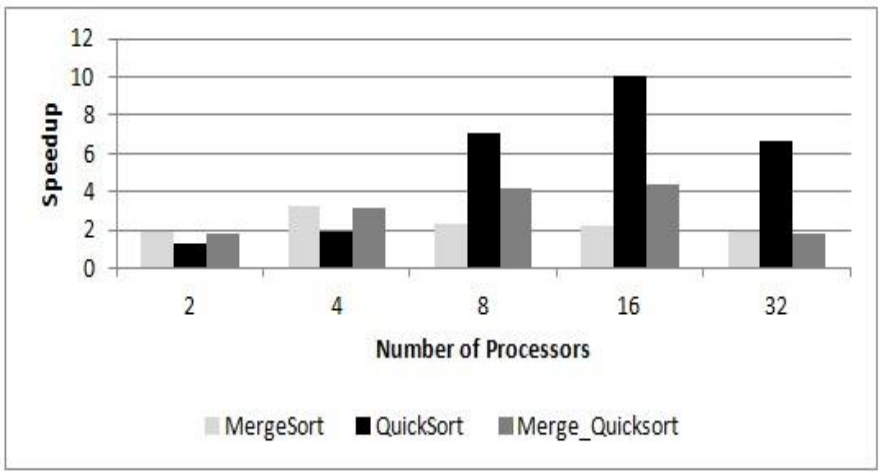

Figure 13. The Speedup of the Three Algorithms on Different Number of Processors with 2^{26} Data Size

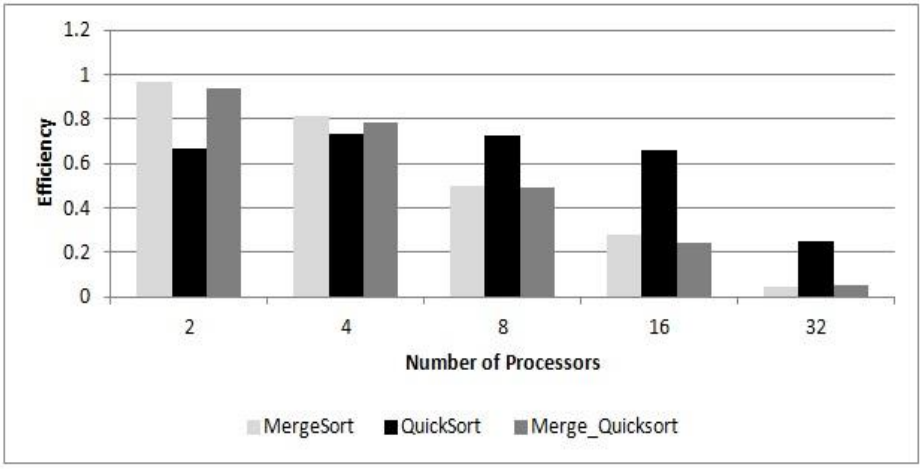

Figure 14. The Efficiency of the Three Algorithms on Different Number of Processors with 2^{22} Data Size

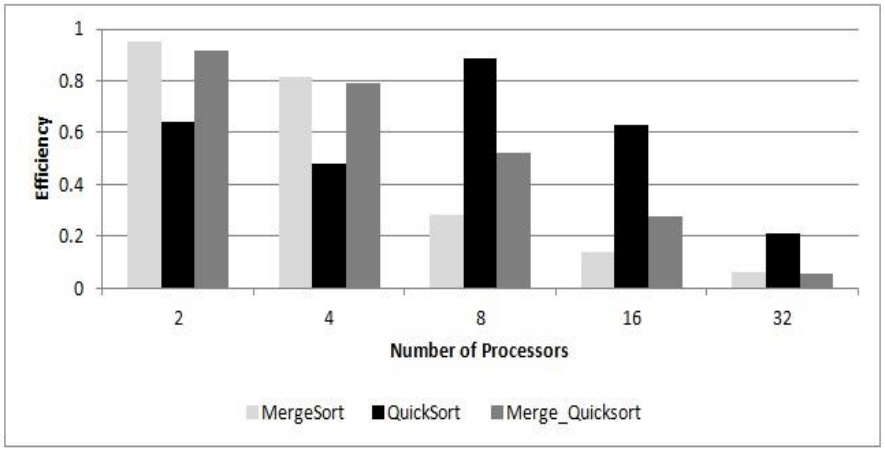

Figure 15. The Efficiency of the Three Algorithms on Different Number of Processors with 2^{26} Data Size

7. Conclusion

In this paper, we present performance evaluation of parallel Quicksort, parallel Merge sort, and parallel Merge-Quicksort algorithms in terms of running time, speedup, and efficiency. The three algorithms are implemented using open MPI library and the experiments are conducted using IMAN1 supercomputer. The evaluation of the three algorithms is based on varying number of processors and input size.

Results show that parallel Quicksort has better running time for both small and large data size, followed by Merge-Quicksort and then Merge sort algorithms. According to parallel efficiency, Quicksort algorithm is more efficient to be applied on large number of processors (8 and more). It achieves up to 88% parallel efficiency, while Merge and Merge-Quicksort achieve up to 49% and 52% parallel efficiency, respectively.

Acknowledgments

We would like to acknowledge eng. Zaid Abudayyeh for his support to accomplish this work.

References

[1] http://en.wikipedia.org/wiki/Sorting_algorithm, Accessed on **(2013)** September 16.
[2] B. A. Mahafzah, "Performance assessment of multithreaded Quicksort algorithm on simultaneous multithreaded architecture", Journal of Supercomput, vol. 66, **(2013)**, pp. 339-363.
[3] Concurrent Programming, Addison wisely http://www.nondot.org/sabre/Mirrored/ AdvProgLangDesign/-finkel07.pdf.
[4] Q. Mohammad, "Adaptive fault tolerant routing algorithm for Tree-Hypercube multicomputer", Journal of computer Science, vol. 2, no. 2, **(2006)**, pp. 124-126.
[5] M. Qatawneh, A. Sleit and W. Almobaideen, "Parallel implementation of polygon clipping using transputer", American Journal of Applied Sciences, vol. 6, no. 2, **(2009)**, pp. 214-218.
[6] A. Ananthgrama, G. Kerypis and Vipinkumar, "Introduction to Parallel Computing", Second Edition, Addison Wesley, **(2003)**.
[7] http://www.iman1.jo/iman1/, Accessed on **(2014)** July 10.
[8] M. Jeon and D. Kim, "Load-Balanced Parallel Merge Sort on Distributed Memory Parallel Computers", Proceedings of the IEEE International Parallel and Distributed Processing Symposium, IPDPS.02, **(2002)**.
[9] M. Jeon and D. Kim, "Parallel Merge Sort with Load Balancing", International Journal of Parallel Programming, vol. 31, no. 1, **(2003)**, pp. 21-33.
[10] I. Singh Rajput, B. Kumar and T. Singh, "Performance Comparison of Sequential Quick Sort and Parallel Quick Sort Algorithms", International Journal of Computer Applications, vol. 57, no. 9, **(2012)**, pp. 14-22.
[11] C. U. Martel and D. Gusfield, "A fast parallel Quicksort algorithm", Information Processing Letters, vol. 30, no. 2, **(1989)**, pp. 97-102.
[12] P. Tsigas and Y. Zhang, "A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000", 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP, **(2003)**.
[13] V. Kale and E. Solomonik, "Parallel Sorting Pattern", The 2nd Annual Conference on Parallel Programming Patterns (ParaPLoP), ACM 978-1-4503-0127-5, **(2010)** March 30-31st.
[14] D. Pasetto and A. Akhriev, "A Comparative Study of Parallel Sort Algorithm", IBM Dublin Research Lab, Mulhuddart, Dublin 15, Ireland.
[15] Luis Quiles, "Quick Sort", Florida Institute of Technology, http://cs.fit.edu/~pkc/classes/writing/hw15/luis.pdf.
[16] W. Almobaideen, M. Qatawneh, A. Sleit, I. Salah and S. Al-Sharaeh, "Efficient Mapping Scheme of Ring Topology onto Tree-Hypercubes", Journal Applied Sci, vol. 7, no. 18, **(2007)**, pp. 2666-2670.
[17] Q. Mohammed, "Embedding Linear Array Network Into The Tree-Hypercube Network", European Journal of Scientific Research, vol. 10, no. 2, **(2005)**.
[18] A. Sleit, W. AlMobaideen, M. Qatawneh and H. Saadeh, "Efficient Processing for Binary Submatrix Matching", American Journal of Applied Sciences, vol. 6, no. 1, **(2008)**, pp. 78-88.
[19] Q. Mohammad and H. Khattab, "New Routing Algorithm for Hex-Cell Network", International Journal of Future Generation Communication and Networking, vol. 8, no. 2, **(2015)**, pp. 295-306.
[20] Q. Mohammad, "Embedding Binary Tree and Bus into Hex-Cell Interconnection Network", Journal of American Science, vol. 7, no. 12, **(2011)**, pp. 367-370.

Authors

Maha Saadeh, is a Ph.D. student in computer science at the University of Jordan. She worked as research and teaching assistant at the computer science department, The University of Jordan from September 2009 to September 2010. Then she received her M.Sc. degree in computer science from the same university in 2011. She has a number of publications in a number of local and international journals and conferences. Her research interests are: wireless networks, network security, and the Internet of Things (IoT).

Huda Saadeh, is a Ph.D. student in computer science at the University of Jordan. She received her M.Sc. degree in computer science from the University of Jordan in 2006. She is working as lecturer at Petra university, computer information systems department. She has a number of publications in international journals. Her research interests are: network security, the Internet of Things (IoT), and image processing.

Mohammad Qatawneh, is a Professor at computer science department, the University of Jordan. He received his Ph.D. in computer engineering from Kiev University in 1996. Dr. Qatawneh published several papers in the areas of parallel algorithms, networks and embedding systems. His research interests include parallel computing, embedding system, and network security.