PERFORMANCE EVALUATION OF PARALLEL BUBBLE SORT ALGORITHM ON SUPERCOMPUTER IMAN1

Reem Saadeh and Mohammad Qatawneh

Department of Computer Science, King Abdullah II School for Information Technology, The University of Jordan, Amman, Jordan.

ABSTRACT

Parallel sorting algorithms order a set of elements USING MULTIPLE processors in order to enhance the performance of sequential sorting algorithms. In general, the performance of sorting algorithms are EVALUATED IN term of algorithm growth rate according to the input size. In this paper, the running time, parallel speedup and parallel efficiency OF PARALLEL bubble sort is evaluated and measured. Message Passing Interface (MPI) IS USED for implementing the parallel version of bubble sort and IMAN1 supercomputer is used to conduct the results. The evaluation results show that parallel bubble sort has better running time as the number of processors increases. On other hand, regarding parallel efficiency, parallel bubble sort algorithm is more efficient to be applied OVER SMALL number of processors.

KEYWORDS

MPI, Parallel Bubble Sort, Parallel Efficiency, Speed Up.

1. INTRODUCTION

Element sorting is one of the most fundamental process for many algorithms such as searching and load balancing. Different sequential sorting algorithms have been implemented such as selection sort, insertion sort, bubble sort, quicksort, merge sort, and heap sort. Sequential sorting algorithms vary in their time complexity. Table 1 lists the time complexity of different sequential sorting algorithms [1]. In order to enhance the performance of sequential sorting algorithms, parallel versions have been implemented. These parallel algorithms aim to reduce the overall execution time and increase the fault-tolerance of sorting based applications [2n]. The performance of parallel sorting algorithms depends on its implementation and the underlying architecture of the parallel machine [3][4][5][6][7][8][9][10].

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 $(\text{n} \log n)$	O $(\text{n} \log n)$	O $(\text{n} \log n)$
Merge	O $(\text{n} \log n)$	O $(\text{n} \log n)$	O $(\text{n} \log n)$
Quick	O $(\text{n} \log n)$	O $(\text{n} \log n)$	O(n^2)

The Objective of this paper is to evaluate the performance of parallel bubble sort. The parallel algorithm is implemented using Message Passing Interface (MPI) which is used to develop parallel algorithms using many programming languages like C, C++, and FORTAN. Many performance metrics are used for evaluating parallel bubble sort including the running time, speedup, parallel efficiency, communication cost, and parallel granularity according to a different number of processors. Results were conducted using IMAN1 supercomputer which is Jordan's first and fastest supercomputer [2].

The rest of this paper is organized as follows; Section 2 summarizes some related works. Section 3 introduces sequential and parallel Bubble sort. The evaluation results of parallel bubble sort algorithm are discussed in Section 4. Finally, Section 5 concludes the paper.

2. RELATED WORKS

Parallel sorting algorithms is considered to be a rich area of research due to the increasing need for efficient, fast, and reliable sorting algorithms which can be implemented over parallel platforms. The authors in [1] use a Message Passing Interface Chameleon (MPICH2) for Windows on a dual-core processor machine in order to study how number of cores and number of processes affect the performance of parallel bubble sort algorithm. In [11] a new study of parallel bubble sort algorithm for K-Nearest Neighbour (KNN) search in introduces. The study is conducted on a parallel processor or accelerator that favours synchronous operations and has high synchronization cost. The authors in [12] proposed a high-sample rate 3-Dimention (3D) median filtering processor architecture that can reduce the computation complexity in comparison with the traditional bubble sorting algorithm.

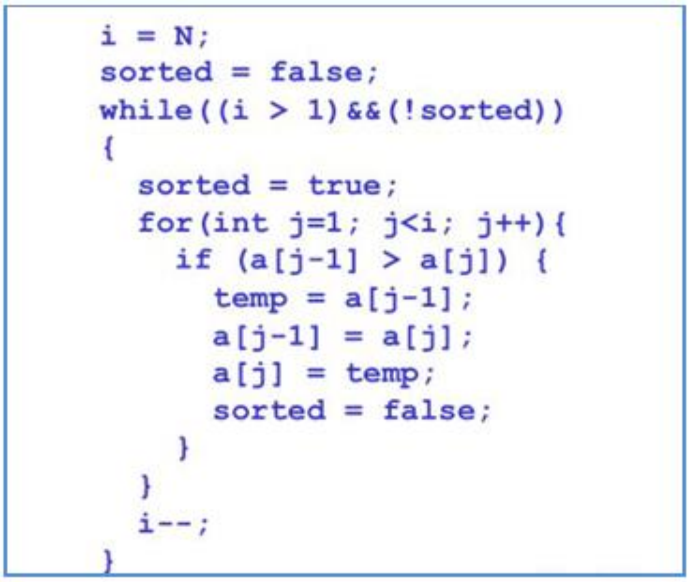
A study of common patterns of parallel sorting algorithms in terms of load balancing, keys distribution, communication cost, and computations cost is conducted in [13]. Another study based on qualitative and quantitative analysis of the performance of parallel sorting algorithms on modern multi-core hardware can be found in [14] [25] [26]. Moreover, several mechanisms for secured message passing use bubble sort algorithm to tackle security problems of information by enhancing the process of encryption and decryption the text message [15] [16] [17] [18] [19]. Sorting algorithms can be used to improve the performance of parallel implementation of different security algorithms such blowfish, RSA, etc. [20] [21] [22] [23].

3. AN OVERVIEW OF SEQUENTIAL AND PARALLEL BUBBLE SORT.

Bubble sort is a simple sorting algorithm for a given dataset, it compares the first two elements and if the first one is greater than the second element, it swaps them. This process is performed for each pair of adjacent elements until it reaches the end of the given dataset. The whole process is repeated again until no swaps have occurred on the last pass. The average run time and worst-case run time for sequential bubble sort are O (n^2). Consequently, sequential bubble sort is rarely used to sort large datasets [1]. In this section, an overview of bubble sort algorithm is discussed. Section 3.1 discusses the sequential algorithm while section 3.2 discusses the parallel version of bubble sort. As discussed previously, the main two operations of Bubble sort are the comparison between two adjacent elements and the swap. In this subsection, sequential bubble sort is discussed in details as follows: For a given array “a” of unsorted elements, sequential Bubble sort algorithm is a recursive procedure works as follows:

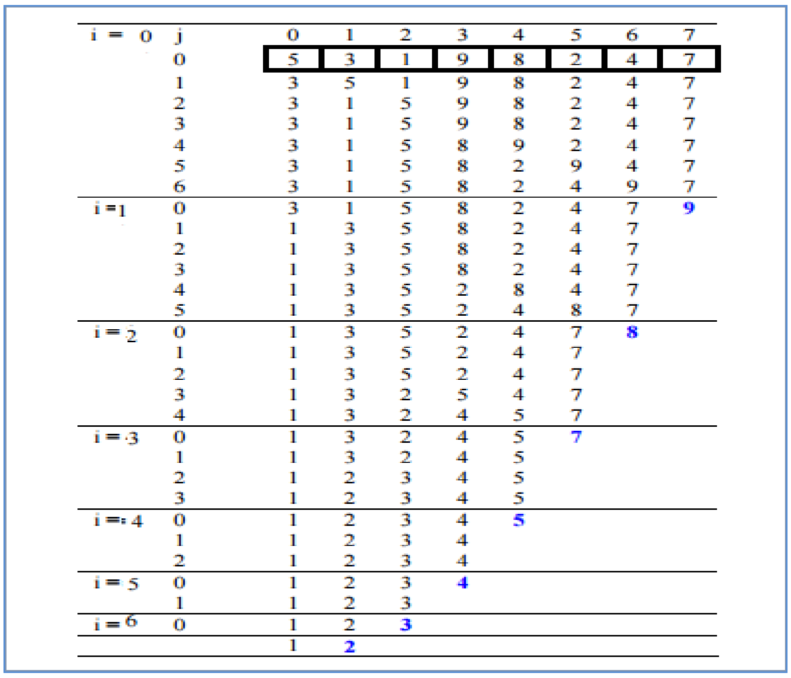
- ***Step 1:*** For the first round, compare a[1] and a[2], swap if needed, than compare a[2] and a[3] until the greatest value moves to the end of the array.
- ***Step 2:*** In the second round, repeat the process in step 1 until the second greatest value next to the greatest value.
- ***Step 3:*** Repeat until no swaps operations are performed. After moving all the comparatively greater values towards the end of the array, this array is sorted.

The Pseudo code for sequential Bubble sort is shown in Figure 1. To understand the sequential Bubble sort, 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 {5, 1, 12, -5, 16}. Firstly, the algorithm will compare the first two elements {5,1} if 5 is greater than 1 then bubble sort algorithm will swap them, secondly the second and third elements will be compared, 5 is less than 12 then nothing will happen then repeat these steps until all the list get sorted.



```
i = N;
sorted = false;
while((i > 1)&&(!sorted))
{
  sorted = true;
  for(int j=1; j<i; j++){
    if (a[j-1] > a[j]) {
      temp = a[j-1];
      a[j-1] = a[j];
      a[j] = temp;
      sorted = false;
    }
  }
  i--;
}
```

Figure 1. Sequential Bubble sort Pseudocode.



i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i =1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						

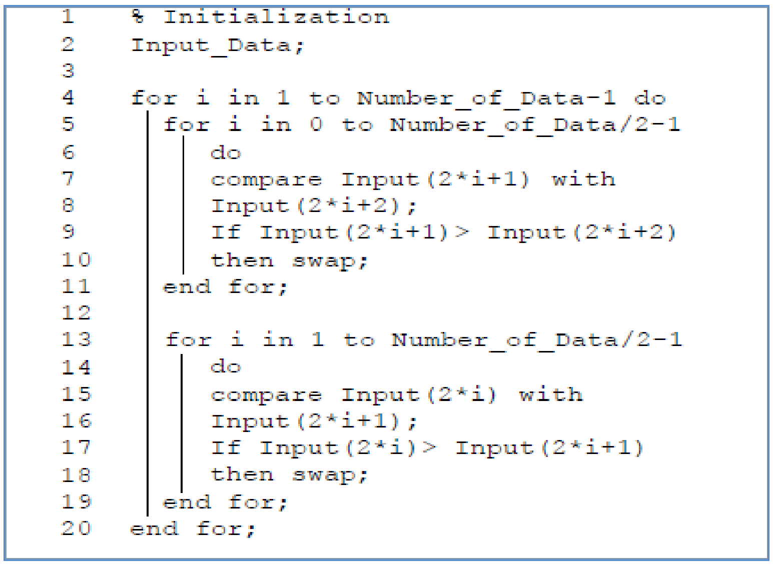
Figure 2. Sequential Bubble Sort Example.

3.1 Sequential Bubble Sort Analysis

Bubble sort has worst-case and average complexity both $O(n^2)$, where n is the number of items being sorted. There exist many sorting algorithms, such as merge sort with substantially better worst-case or average complexity of $O(n \log n)$. Even other $O(n^2)$ sorting algorithms, such as insertion sort, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large. The only significant advantage that bubble sort has over most of the other implementations is that the ability to detect that the list is sorted efficiently is built into the algorithm. When the list is already sorted (best-case), the complexity of bubble sort is only $O(n)$ [1].

3.2 Parallel Bubble Sort

Parallel Bubble sort algorithm, also known as odd-even bubble sort, and can be implemented in different ways. The main idea in parallel bubble sort is to compare all pairs in the input list in parallel, then, alternate between odd and even phases. The parallel bubble sort pseudo code is shown in Figure 3.



```
% Initialization
Input_Data;

for i in 1 to Number_of_Data-1 do
    for i in 0 to Number_of_Data/2-1
        do
        compare Input(2*i+1) with
        Input(2*i+2);
        If Input(2*i+1)> Input(2*i+2)
        then swap;
    end for;

    for i in 1 to Number_of_Data/2-1
        do
        compare Input(2*i) with
        Input(2*i+1);
        If Input(2*i)> Input(2*i+1)
        then swap;
    end for;
end for;
```

Figure 3. The parallel bubble sort algorithm [24]

Assuming that the input list size is n, the parallel version of bubble sort will perform n iterations of the main loop and for each iteration, a number of n-1 comparisons and (n-1)/2 exchanges will be performed in parallel. If the number of processors is lower than n/2, every processor will execute (n/2)/p comparisons for each inner loop, where p is the number of processors. In this case, the complexity level of the algorithm will be $O(n^2 /2p)$. If the parallel system has a number of processors p higher than n/2, the complexity level of the inner loops is O(1) because all the iterations are performed in parallel. The outer loop will be executed for maximum n times. Consequently, the complexity of parallel bubble sort equals to O(n) which is much better than the complexity of the fastest known sequential sorting algorithm with O(n log n) complexity [1]. An example of parallel bubble sort tis illustrated in Figure 4.

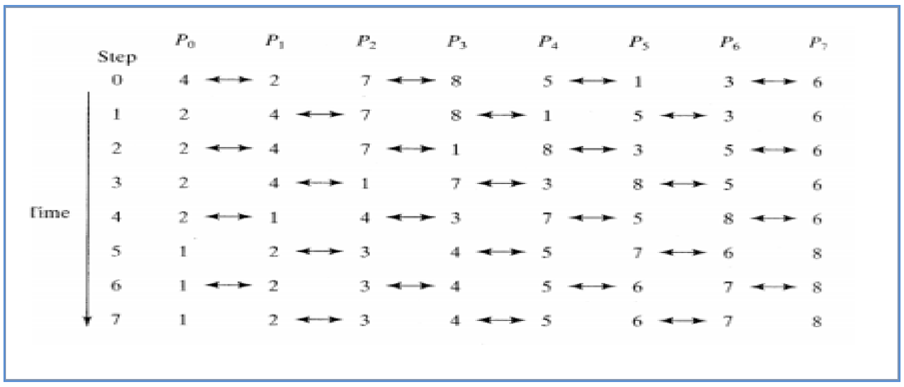


Figure 4. Example of Parallel Bubble Sort

3.2.1 Parallel Bubble Sort Analysis.

Parallel Bubble sort is analyzed according to the communication cost, complexity and execution time.

- **Communication cost**: the communication cost depends on the pre byte transfer time which is based on the channel bandwidth (t_b), the diameter of the used topology (L), and the message size (m). The communication cost also depend on the time that a node requires to prepare the message to be transmitted, it consists of adding a header, trailer etc. This time is called startup time (t_s). Based on this, The total communication cost = ($t_s + m*t_b$)*L.
- **Complexity**: this is the time required to perform bubble sort locally on each processor and the time of partition procedure. In the best case, each processor will sort n/p of elements using sequential bubble sort at the same time. As we can see in Figure 3, there are two loops, the outer loop will take n steps and the inner loop depends on the odd or even phase which takes n/2 steps and that time will be divided by the number of processors. So the time complexity that is required for all processors to sort the dataset is O (n^2 /2p).
- **Execution time**: this time includes both the time complexity of sorting and the communication cost. As discussed previously, the complexity of parallel sorting is O (n^2 /2p). The communication cost depends on the message that is transmitted over the link in each step, so the execution time = O (n^2 /2p) + ($t_s + m*t_b$)*L.

4. PERFORMANCE EVALUATION AND RESULT

In this section, the results are discussed and evaluated in terms of running time, speedup and parallel efficiency. IMAN1 Zaina cluster is used to conduct the experiments and open MPI library is used in the implementation of the parallel bubble sort algorithm. Parallel bubble sort algorithm is evaluated according to different number of processors.

4.1 Run Time Evaluation

Figure 5 shows the run time for the algorithm according to a different number of processors including the sequential time (p = 1). a fixed data size is chosen to conduct the experiments. The general behavior can be summarized as follows: As the number of processors increases the run time is reduced due to better parallelism and better load distribution among more processors. This is the case when moving from 2 to 8 or to 16 or to 32 processors. but, as shown in the figure, when using 64 or 128 processors the run time increases which indicates that the communication overhead increases too, as a result, the benefit of parallelism decreases.

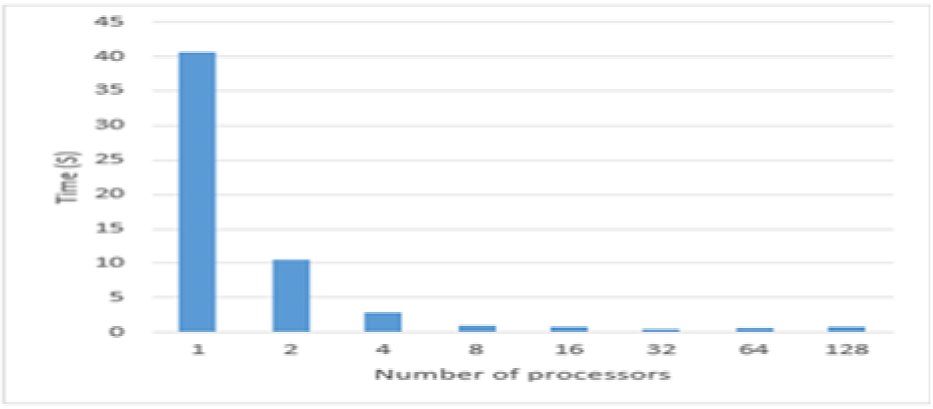


Figure 5. Run time According to Different Number of Processors

4.2 Speedup Evaluation

The speedup is the ration of the serial execution time of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on P processors. The Speedup = sequential processing time/parallel processing time. Figure 6 illustrates the super linear speedup of the parallel bubble sort algorithm on 2, 4, 8, 16, 32, 64 and 128 processors with fixed data size. As can be noticed, parallel speedup increases when the number of processors increases. The parallel algorithm achieves the best speedup on a large number of processors. When using 16 and 32 processors the parallel bubble sort achieves up to 54 and 74 speedup, respectively. On the other hand, for 64 and 128 processors the speedup decreases due to the communication overhead which increases when using 64 and 128 processors.

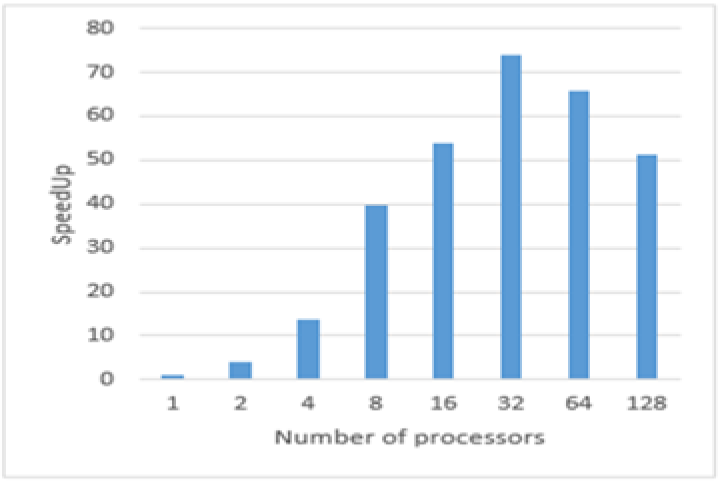


Figure 6. Speedup According to Different Number of Processors

4.3 Parallel Efficiency Evaluation

Parallel efficiency is the ratio between speedup and the number of processors, where the Parallel efficiency = S / P. Figure 7 shows the parallel efficiency for parallel bubble sort algorithm according to different number of processors. The algorithm achieves the best efficiency when using small number of processors, where the number of processors is equal to two, the parallel bubble sort achieves about 26% parallel efficiency. On the other hand, when using medium and large number of processors (4,8,16 and 32), parallel bubble sort achieves 7.3%, 2.5%, 1.9% and 1.4% parallel efficiency, respectively.

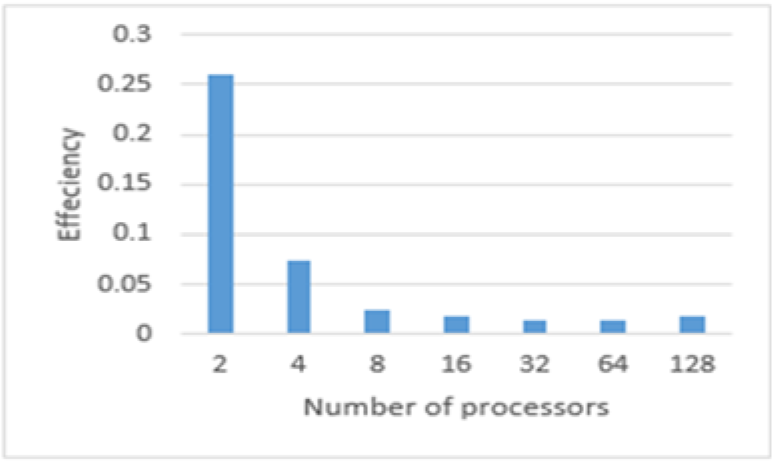


Figure 7. Efficiency According to Different Number of Processors

4.4 Parallel Cost Evaluation

Cost of solving a problem on a parallel system is the product of parallel run time and the number of processors. On the other hand, the cost of solving a problem on a single processor is the execution time of the fastest sequential algorithm. Parallel Cost = p T_p , Sequential Cost = T_s. Figure 8 shows the sequential and parallel cost for parallel bubble sort algorithm according to different number of processors. As the number of processor increases the cost is reduced, this is the case when moving from 2 to 4 or to 8 processors. On the other hand, when moving from 16 to 32 or from 64 to 128 processors the cost 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.

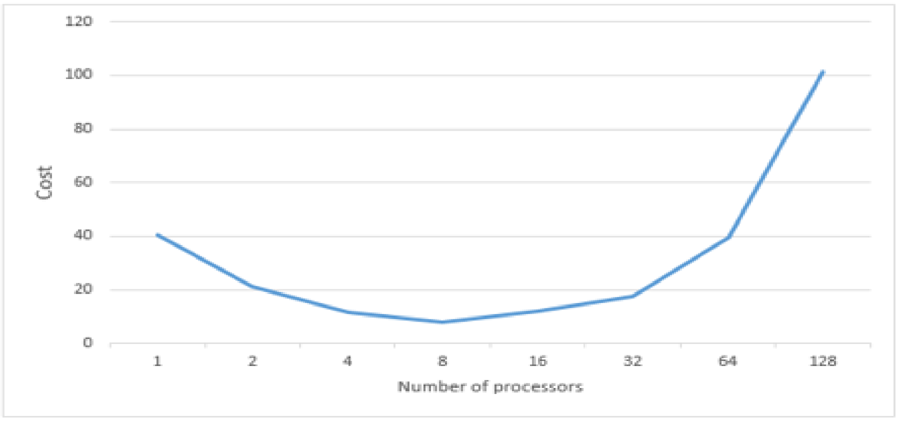


Figure 8. Cost According to Different Number of Processors

4.5 Difference Between Analytical and Implementation Results.

In this subsection, a comparison between the analytical and the implementation time results is discussed. Table 2 lists the differences in time execution results. Figure 9 illustrates the error between the analytic and the implementation results.

Table 2. The Difference between analytical time results and implementation time results

Number of processors	Analytical	Implemented
1-Processor	40	40.622
2-Processors	10.4105	10.518
4-Processors	5.225	2.949
8-Processors	2.605	1.018
16-Processors	1.348	0.754
32-Processors	0.697	0.55
64-Processors	0.379	0.617
128-Processors	0.1913	0.791

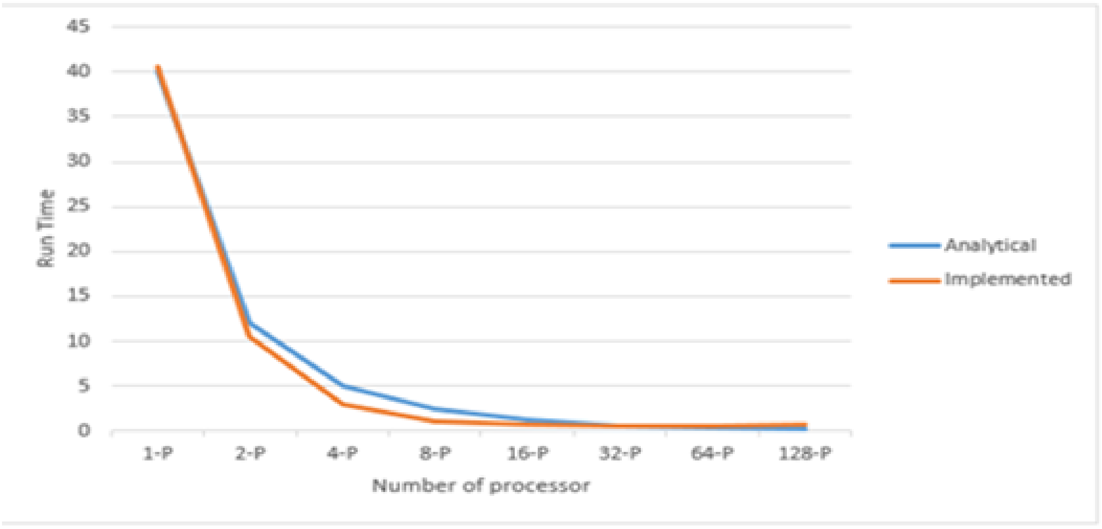


Figure 9: The Difference between Analytical and Implementation Values.

4.6 Granularity

Granularity is the ratio of the time required for a basic computation to the time required for basic communication. Granularity = Computation time/ communication time. There are three types of granularity:

1. **Fine Grained**: In fine grained parallelism, a program is broken down to a large number of small tasks. These tasks are assigned individually for many processors. Consequently, the ration between the computation and the communication cost is small.

2. **Coarse Grained**: In coarse grained parallelism, a program is split into large tasks. Due to this, a large amount of computation takes place in processors. Consequently, the ration between the computation and the communication cost is small.

3. **Medium Grained:** Medium-grained parallelism is a compromise between fine-grained and coarse-grained parallelism in which we have task size and communication time greater than fine-grained parallelism and lower than coarse-grained parallelism.

Table 3. Granularity

Number of processors	Computation	Communication	Granularity
2-P	10.155	0.255	39.82
4-P	5.07	0.155	32.7
8-P	2.5	0.105	23.8
16-P	1.26	0.08	15.75
32-P	0.63	0.0675	9.33
64-P	0.317	0.0625	5.072
128-P	0.16	0.0313	5.11

As shown in Figure 10 the ratio between computation and communication is large, when used a small number of processors. For two processors, i.e. coarse grained, the granularity is up to 39.8. This means that the program is split into large tasks. The advantages of this type of parallelism are low communication overhead and the low synchronization overhead. On the other hand, when using a large number of processors, i.e. 32, the program is broken down to a large number of small tasks, this type is called fine grained and the communication and synchronization overhead will be increased in this type. When using a medium number of processors, i.e.8, the task size and communication time will be greater than fine-grained parallelism and lower than coarse-grained parallelism and this type called medium grained.

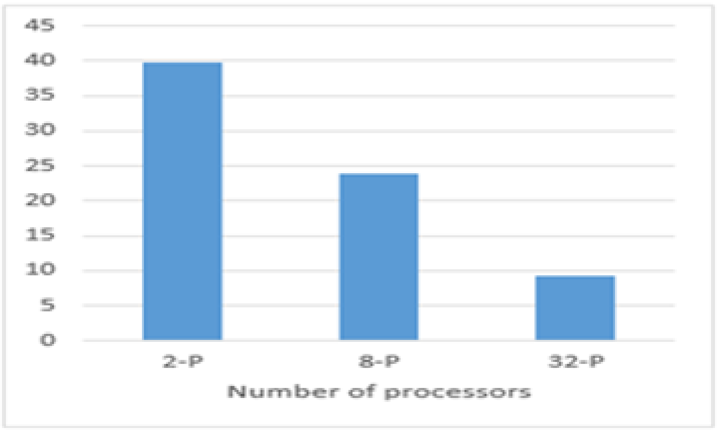


Figure 10: The Ratio between Computation and Communication

5. Conclusion

In this paper, the performance evaluation of parallel bubble sort algorithm in terms of running time, speedup, and efficiency is presented. The algorithm is implemented using open MPI library and the experiments are conducted using IMAN1 supercomputer. The evaluation of the algorithm is based on varying number of processors. Results show that parallel bubble sort has better running time on large number of processors. According to parallel efficiency, parallel bubble sort algorithm is more efficient to be applied on small number of processors i.e.2 processors which achieves up to 26% parallel efficiency.

References

[1] A. I. Elnashar, (2011) "Parallel performance of mpi sorting algorithms on dual-core processor windows-based systems," arXiv preprint arXiv:1105.6040.

[2] M. Saadeh, H. Saadeh, and M. Qatawneh, “Performance evaluation of parallel sorting algorithms on iman1 supercomputer,” International Journal of Advanced Science and Technology, vol. 95, pp. 57–72, 2016.

[3] M. Qatawneh, (2005) “Embedding linear array network into the tree-hypercube network,” European Journal of Scientific Research, vol. 10, no. 2, pp. 72–76.

[4] N. Islam, M. S. Islam, M. Kashem, M. Islam, and M. Islam, (2009), “An empirical distributed matrix multiplication algorithm to reduce time complexity,” in Proceedings of the International Multi Conference of Engineers and Computer Scientists, vol. 2, pp. 18–20.

[5] M. Qatawneh, (2011) “Multilayer hex-cells: a new class of hex-cell interconnection networks for massively parallel systems,” International journal of Communications, Network and System Sciences, vol. 4, no. 11, p.704.

[6] M. Qatawneh, (2011) “Embedding binary tree and bus into hex-cell interconnection network,” Journal of American Sciences, vol. 7, no. 12, p. 0.

[7] M. Qatawneh, (2016) “New efficient algorithm for mapping linear array into hex-cell network,” International Journal of Advanced Science and Technology, vol. 90, pp. 9–14.

[8] M. Qatawneh, “Adaptive fault tolerant routing algorithm for tree hypercube multicomputer,” Journal of computer Science, vol. 2, no. 2, pp. 124–126, 2006.

[9] M. Qatawneh, A. Alamoush ,J. Al Qatawneh, (2015) “Section Based Hex-Cell Routing Algorithm (SBHCR),” International Journal of Computer Networks &Communications, vol. 7, no. 1, p. 167.

[10] M. Qatawneh and H. Khattab, (2015), “New routing algorithm for hex-cell network,” International Journal of Future Generation Communication and Networking, vol. 8, no. 2, pp. 295–306.

[11] N. Sismanis, N. Pitsianis, and X. Sun, (2012), “Parallel search of k-nearestneighbors with synchronous operations,” in 2012 IEEE Conference onHigh Performance Extreme Computing. IEEE, 2012, pp. 1–6.

[12] Mm. Jiang and D. Crookes. (2006), “High-performance 3D median filter architecture for medical image despeckling”. Electronics Letters. 2006. 42(24): p. 1379-1380.

[13] Kale V, Solomonik E. (2010), “Parallel sorting pattern. In Proceedings of the 2010 Workshop on Parallel Programming Patterns. p. 10. ACM.

[14] Pasetto D, Akhriev A. (2011) “A comparative study of parallel sort algorithms,”. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. p. 203-204. ACM.

[15] M. Qatawneh, A. Sleit, W. Almobaideen. (2009), “Parallel Implementation of Polygon Clipping Using Transputer”. American Journal of Applied Sciences 6 (2): 214-218, 2009.

[16] O. Surakhi, M. Qatawneh, H. Al ofeishat, (2017), “A Parallel Genetic Algorithm for Maximum Flow problem”. International Journal of Advanced Computer Science and Applications, Vol. 8, No. 6, 2017.

[17] S. Hijazi and M. Qatawneh. (2017), “Study of Performance Evaluation of Binary Search on Merge Sorted Array Using Different Strategies”. International Journal of Modern Education and Computer Science, 12, 1-8.

[18] O. AbuAlghanam, M. Qatawneh, H.al Ofeishat, O. adwan, A. Huneiti. (2017), “A New Parallel Matrix Multiplication Algorithm on Tree-Hypercube Network using IMAN1 Supercomputer”. International Journal of Advanced Computer Science and Applications, Vol. 8, No. 12, 2017.

[19] M. Haj Qasem and M. Qatawneh, (2018), “Parallel matrix multiplication for business applications,” vol. 662, 01 pp. 24–36.

[20] A. Bany Doumi and M. Qatawneh. PERFORMANCE EVALUATION OF PARALLEL INTERNATIONAL DATA ENCRYPTION ALGORITHM ON IMAN1 SUPER COMPUTER. International Journal of Network Security & Its Applications (IJNSA) Vol. 11, No.1, January 2019.

[21] H. Harahsheh and M. Qatawneh. (2018), “Performance Evaluation of Twofish Algorithm on IMAN1 Supercomputer”. International Journal of Computer Applications, Vol. 179 (50).

[22] A.Al-Shorman, M. Qatawneh. (2018), “Performance of Parallel RSA on IMAN1 Supercomputer”. International Journal of Computer Applications, Vol. 180 (37)

[23] M. Asassfeh ,M. Qatawneh, F.AL-Azzeh. (2018), “PERFORMANCE EVALUATION OF BLOWFISH ALGORITHM ON SUPERCOMPUTER IMAN1”. International Journal of Computer Networks & Communications (IJCNC), Vol. 10 (2), 2018.

[24] D. Purnomo, J. Marhaendro , A. Arinaldi, D. Priyantini, A. Wibisono, and A. Febrian. (2016), “mplementation of Serial and Parallel Bubble Sort on FPGA.” Jurnal Ilmu Komputer dan Informasi 9, no. 2: 113-120.

[25] Azzam Sleit, Wesam Almobaideen, Mohammad Qatawneh, and Heba Saadeh. “Efficient processing for binary submatrix matching”. American Journal of Applied Science, Vol. 6(1), 2008.

[26] Wesam Almobaideen, Mohammad Qatawneh, Azzam Sleit, Imad Salah and Saleh Al-Sharaeh. “Efficient Mapping Scheme of Ring Topology onto Tree-Hypercubes”. Journal of Applied Sciences 7(18), 2007.

AUTHOR

Professor Mohammad Qatawneh (mohd.qat@ju.edu.jo) is a Professor at computer science department, the University of Jordan. He received his Ph.D. in computer engineering from Kiev University in 1996 and his M.Sc. degree in computer engineering from University of Donetsk – USSR in 1988. His research interests include: Wireless network, parallel computing, embedding system Computer and network security, Routing protocols, Internet of Things, Blockchain and Security.