*Journal of Computer Science & Computational Mathematics, Volume 5, Issue 4, December 2015*
DOI: 10.20967/jcscm.2015.04.002

# Hybrid Bubble and Merge Sort Algorithms Using Message Passing Interface

Zaid Abdi Alkareem Alyasseri[1*], Kadhim Al-Attar[2], Mazin Naser Yousif [3]

[1]Faculty of Engineering / ECE Dept., University of Kufa, Iraq
[2]Computer Center /Al-Mustansiriya University, Baghdad, Iraq
[3]Vizocom Company, Basra, Iraq
[*]Corresponding author email: Zaid.alyasseri@uokufa.edu.iq

***Abstract*:** Sorting or ordering is an arrangement of a set of items of a list in a certain order. Several sorting algorithms and methods are available. In this paper, we focus on the simplest but not highly efficient bubble sort algorithm. Rearranging parallel computing, i.e., message passing interface (MPI), it is explained why parallel architectures are now mainstream. Recently, parallel computing has become one of the most important aspects of computer science and it has showed dominance especially in the field of analyzing high-performance problems. One of the major results is the type of data structure obtained by using 3D array of char, which shows faster result, instead of using a vector of string with extra overhead. The threads (slaves) sort part of the list and return to the master thread, which merges all small lists in one final sorted list. This study shows an increase in speed and efficiency with the increase in number of threads, indicating an increase in the efficiency of bubble sort algorithm.

***Keywords*:** bubble sort, MPI, sorting algorithms, parallel computing, parallelize bubble algorithm.

## 1. Introduction

Sorting or ordering a list of objects is a primary research filed in computer science as well as a major problem. This arrangement (sorting) displays a set of items either in ascending order e.g. (1, 3, and 4) or descending one e.g. (3, 2, and 1). The input is a sequence of N items, which are in some random order; at the end of output are sorting or arranging elements of list in specific way either a positive or negative order [1]. Sorting is considered as a fundamental problem-solving method for computer scientists and as an intermediate step to solve other problems. Many sorting algorithms are used in the field of computer science; they differ in their complexity (execution time and space), utilization with different data type, and resource usage [2]. In this paper we focus on the bubble sort algorithm.

### 1.1 Bubble Sort Algorithm

Bubble sort is self-explanatory and uncomplicated but highly inefficient sort and rarely used in practice. The algorithm name, bubble sort, comes from a logical fact (the light weight), small values “bubble” up to the top of data list, whereas (heavy weight) large values fall down to end of data list[4]. The algorithm for bubble sort is as follows: the procedures begin with first element in data set. By comparing the first with the second element; if the first is larger than the second, it changes them. It goes on for every couple of elements until it finishes all the data set. Then start again from the beginning with the first two items, continues swapping elements until no longer swaps occurrence on the final pass [3]. The worst-case and average performance of this algorithm is **O($n^2$).** The location of element has a substantial aspect in defining the performance of the algorithm. For example, the element in the beginning of the list quickly swaps, but those at the end cause an issue considering that bringing the element to the start of the list requires several steps.

### 1.2 Algorithm : Bubble Sort

```
void BubSort(int arr1[], int n1) {
    bool swapps = true;
    int tmp1;
    while (swapps) {
        swapps = false;
        for (int k = 0; k<n1 - j; k++) {
            if (arr1[k] >arr1[k + 1]) {
                tmp1 = arr1[k];
                arr1[k] = arr1[k + 1];
                arr1[k + 1] = tmp1;
                swapps = true;
            } } } }
```

The above algorithm uses a procedure or a function in C++ for bubble sort. The **arr1[]** is the unsorted input list and N is the total size of the data list or number of items in the list. After finalizing the procedure, the array will be sorted. The variable **swapps** is a flag that keeps the iteration on indicates list still unsorted and the value of swapped flag equals to true otherwise iteration stop and list sorted. The variable **tmp1** is temporarily placed to help in the swapping of the two adjacent elements in the list.

As an example, using an array 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 to emulate each step of Bubble Sort algorithm, each pass would be like as shown in Table 1.

By the first step of Bubble algorithm, the large element value sinks down and positions to the end of the array. As the second step, the second large value positions in the second-largest array. The steps continue until each element is ordered in the correct way as shown in Table 1.

**Table 1:** Bubble Sort for following input 9, 8, 7, 6, 5, 4, 3, 2, 1, 0.

| | |
|---|---|
| Pass 1 | 8, 7, 6, 5, 4, 3, 2, 1, 0, 9 |
| Pass 2 | 7, 6, 5, 4, 3, 2, 1, 0, 8, 9 |
| Pass 3 | 6, 5, 4, 3, 2, 1, 0, 7, 8, 9 |
| Pass 4 | 5, 4, 3, 2, 1, 0, 6, 7, 8, 9 |
| Pass 5 | 4, 3, 2, 1, 0, 5, 6, 7, 8, 9 |
| Pass 6 | 3, 2, 1, 0, 4, 5, 6, 7, 8, 9 |
| Pass 7 | 2, 1, 0, 3, 4, 5, 6, 7, 8, 9 |
| Pass 8 | 1, 0, 2, 3, 4, 5, 6, 7, 8, 9 |
| Pass 9 | 1, 0, 2, 3, 4, 5, 6, 7, 8, 9 |
| Pass 10 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 |

## 1.3 Parallel Computing

Parallel computing is a type of processing in which many arithmetic operations are accomplish concurrently, which lead to accelerated computations [5]. The main idea is that large problems can be generally split into smaller sub-problems, which are then solved simultaneously. The increased availability of parallel hardware represents a tremendous opportunity to implement a parallel solution [6]. In parallel, the data or functions or sometimes both could usually be decomposed. The computing resource is typically a one-box machine with a number of processors/cores and arbitrary number of such machine (computer) is connected by a network [7].

## 1.4 Message Passing Interface (MPI)

Message passing interface (MPI) is a specification for a standard library for message passing defined by the MPI Forum, a broadly based group of parallel computer vendors, library writers, and application specialists [8]. MPI is a specification, not an implementation. MPI mainly aims to develop a widely used standard for writing message passing programs and achieving practical, portable, efficient, and flexible standard for message passing. MPI can be supported and implanted by C, C++, Fortran-77, and Fortran-95. MPI is a mixture of functions, subroutines, and methods and has a different syntax from other languages as well. MPI is a de-factor standard for message-passing software used for the development of high-performance portable parallel applications. The MPI specification has been implemented for a wide range of computer systems from clusters of networked workstations, running the general purpose operating systems UNIX, Windows [9].

## 1.5 Literature Review

In one study [10], a table summary shows that bubble sort is an exchange method with straightforward, simple, and stable advantage. Disadvantages include low inefficiency on large data.

In another study [11], a new algorithm called Freezing Sort is presented. The results indicate better performance algorithm in a stage of worst case scenario, the results also compared with Bubble sort and other used algorithms. Freezing Sort added strength to the work of Bubble sort by compressing and minimizing the number of comparisons and swaps between the elements of data list.

Another investigation [12], presents a new approach of Bubble Sort technique, the main steps of procedure is rather than swap two variables with a third temporarily variable, using a shift-and-replace procedure. The results indicate that this method takes less time and in the final analyses result gives very good indication on average and worst case.

Based on a study [13], the execution time of the Bubble Sort algorithm is improved by implementing it using a new algorithm, which reduces the execution time by approximately 40%. A stable sorting algorithm has been designed, which can sort the Maximum Number of elements in every pass. The sorting is done using the following two steps: Step 1: the leftmost and rightmost elements are compared if leftmost is larger than the rightmost, then, swapped. Then, left-most bound increases and the right-most bound decreases. The main loop is repeated “n/2” times, provided that “n” is the Length of array. Step 2: In this phase, bubble sort algorithm is applied from left to right, up to the middle element, and from right to left, up to the middle element. Then, array is sorted.

The next study [14] presents a new sorting algorithm based on counting the smallest elements in the array of every element, and keeps their position. The result of implementation is compared with other sorting algorithms. For large data input string, other sorting algorithms like Selection Sort is faster than Bubble Sort and new Counting Position method

Another study [15] presents a new enhanced version of the bubble sort algorithm. The optimized bubble sort has reduced the number of iteration by half compared with the traditional bubble sort, which has a significant improvement in reducing the number of pass iteration.

Based on Elnashar [16], the paradigm has three main stages, namely, scatter, sort, and merge. The first stage is responsible for scattering the data in some way and then distributing the unsorted data list between the MPI processes in such a way that each of the processes accept part of unordered list. In the second phase after each process received its part of data, the process of sort phase starts, each process sorts its local unsorted data list using any of sorting algorithms. After each process “slave” finished sorting its part of unordered list, the third phase starts, by sending the sorted list of each process to the main process “master” and merging all lists in one big list in order to generate the final sorted data list. In each process different sort algorithms are applied and one of them is Bubble sort algorithm. The results show that this algorithm is the slowest compared with other algorithms, but the execution time decreased rapidly when the number of processes increased.

## 2. Methodology

Dataset: Two types of text file dataset have been provided in this paper (HAMLET, PRINCE OF DENMARK by William Shakespeare), which are different in size and length. The first dataset is equal to 190 KB, and the second one is equal to 1.38 MB taken from (http://www.booksshouldbefree.com) .

Pre-processing: The first phase of removing/ ignoring the special characters from the text file keeps only words with different lengths .

The sequential code (all jobs done by one CPU) implements two approaches based on the Data structure, which has a huge effect on the functions of algorithm and on the required amount of time to complete the job. The first was implemented based on vectors of string. The second was implemented based on 3D array of char. The result indicates that the second type of data structure is faster and better .

Two different ways to sort the dataset using bubble sort algorithm are as follows: The first sort is based on the length of characters; each group has a different number of words based on the same number of characters. The second type of sort is based on the arrangement of all the words in alphabetical order using bubble sort algorithm .

The last part is implementing MPI code using Master/Slave communications , where the first step is to calculate the size of chunks equal the length of the array, divided by the number of processors**, "Whenever the chunk size small the parallel time is better"**[17]. Then, the master will send the data to the slaves, and each slave will perform a bubble sort on its data. At the end of this step, the slaves send the result to the master. In the final step all the chunks are merged and the results are printed. Figure 1 describes the proposed method for sorting Datasets 1 and 2.

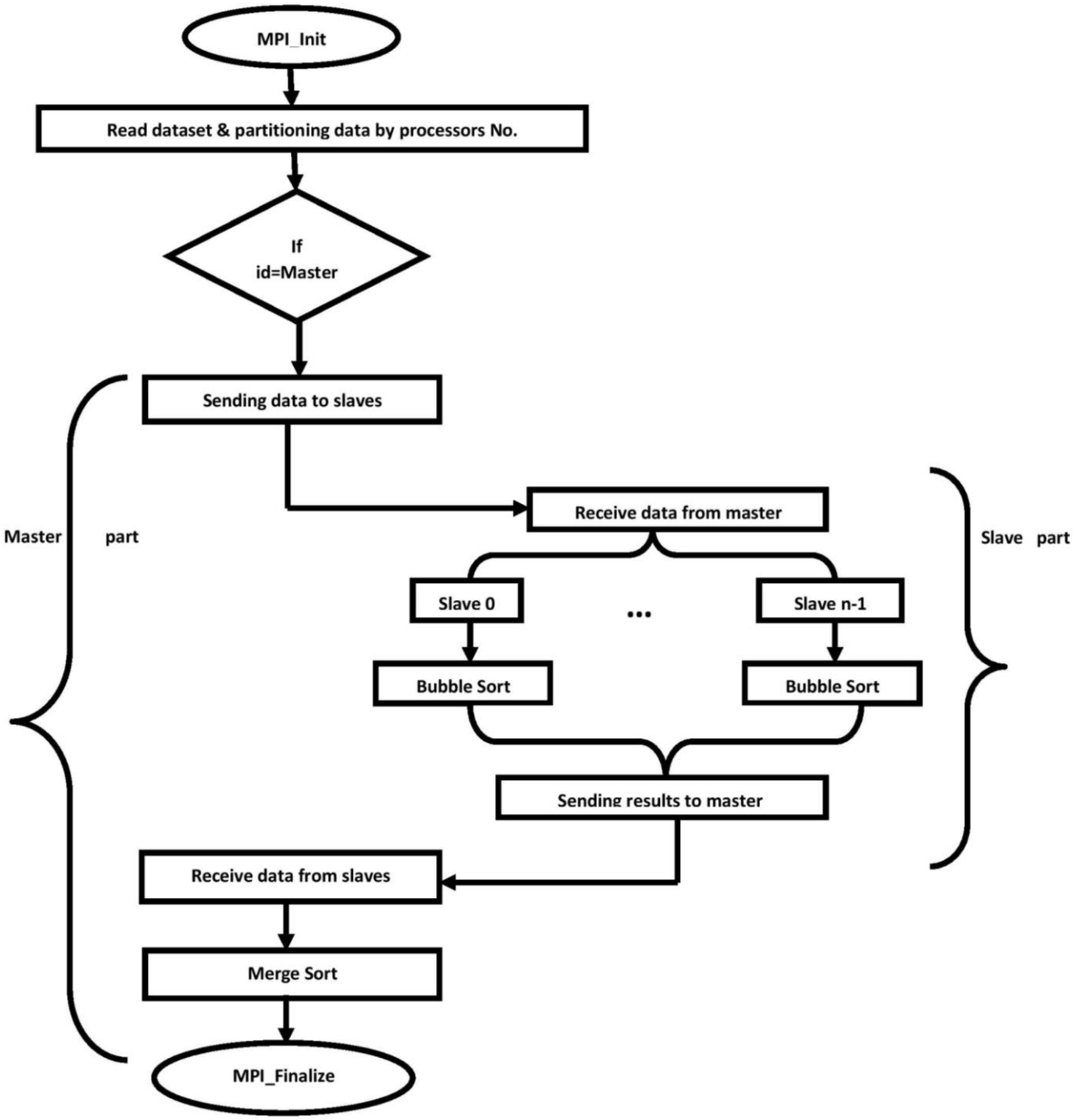


**Figure 1.** The proposed method for sorting Datasets 1 and 2

## 3. Results and Analysis

The dataset has been tested 10 times to obtain the average. The text files used for this experiment are in English, as shown in the previous section. The sequential performance has been implemented on ASUS A55V, Windows 7 Home Premium 64-bit with Intel core i7-3610QM, (8 CPUs), 2.30GH and 8 GB of RAM.

**Table 2** The time required for the dataset1 and dataset2 pre-processing

| Function | Dataset 1 (190KB) | Dataset2 (1.38MB) |
|---|---|---|
| **Preprocessing to remove the special characters from a text file** | 0.265 second | 1.154 second |
| **Bubble sort based on alphabetic order** | 274.528 second | - |
| **Bubble sort based on the length of the word** | 230.215second | - |
| **Bubble sort on array of vector each vector has the same length of the words** | 42.589 second | 1620.44 second |

Table 2 shows a result of pre-processing on both texts for dataset1 and dataset2, which remove the special characters from a text file, keeping only words in one line. In this case, the datasets are ready for sorting. The results indicate that when the data becomes larger, it requires extra time to remove special characters form the text, and only keeps words.

After the pre-processing has entered the data as a vector of string, and then sorting is done using bubble sort on different order. The first sort is in alphabetical order form A–Z. The second sort is based on the length of the word, starting from the shortest word to the longest word. The third sort is based on sorting a vector of the same word length alphabetically. All the time, the recorded data structure is based on vector of string in C++.

In general, 42.589 seconds is needed for third bubble sort, in which an array of vector shows that each vector has the same length of the words. This has taken less time than the other bubble sort based on alphabetical order or word length for dataset1.

For dataset2, the third type of bubble sort has taken around half hour because the other two have not recorded the time. Consequently, the run was aborted, considering its impracticality with over half an hour to carry out the action.

Table 3 shows how using different data structures could affect the run time for the algorithm, even with the same dataset. Table 3 had run two datasets with sizes 190KB and 1.38MB and two different data structures. The first one is a vector of string, and the second is the 3D array of char. The results indicate that the first data structure for both datasets has taken longer execution time compared with 3D array of char. This result is explained by the vector of string being saved during programming. However, it takes extra overhead, which in turn, takes additional time. In other data structure, which displays direct mapping of data to memory through pointers, 3D array of char is faster with direct access to data. However, steps must ensure not to enter an invalid position in memory during programming.

From the same table, the results indicate that the second data structure is faster and best for programming, thereby it can be regarded as the basis for other results and tests compared with parallel processing

**Table 3.** The performance for sequential for two different data structure

| Test/ number | Sorting time in second for Vector of string | | Sorting time in second for 3D array of char | |
|---|---|---|---|---|
| | Dataset1 | Dataset2 | Dataset1 | Dataset2 |
| Test1 | 44.239 | 1694.51 | 6.695 | 188.185 |
| Test2 | 44.013 | 1697.98 | 6.624 | 188.194 |
| Test3 | 44.239 | 1702.21 | 6.615 | 188.247 |
| Test4 | 44.503 | 1696.88 | 6.694 | 188.169 |
| Test5 | 44.23 | 1698.3 | 6.531 | 188.348 |
| Test6 | 44.746 | 1692.79 | 6.654 | 188.289 |
| Test7 | 44.433 | 1687 | 6.654 | 188.154 |
| Test8 | 44.36 | 1686.96 | 6.614 | 188.512 |
| Test9 | 44.267 | 1674.82 | 6.646 | 188.343 |
| Test10 | 44.704 | 1630.32 | 6.672 | 188.181 |
| Ave | 44.737 | 1686.177 | 6.639 | 188.262 |

**Table 4**. Sequential & Parallel MPI Bubble Sort

| *Thread No.* | *Data1 Time/ Sec* | *Data2 time/ Sec* | *Speed up Data 1* | *Speed up Data 2* | *Efficiency Data 1* | *Efficiency Data 2* |
|---|---|---|---|---|---|---|
| 1 | 4.703 | 181.908 | 1 | 1 | 100 | 100 |
| 2 | 1.258 | 49.836 | 3.738 | 3.650 | 186.924 | 182.507 |
| 4 | 0.507 | 20.497 | 9.276 | 8.875 | 231.903 | 221.871 |
| 6 | 0.308 | 15.277 | 15.269 | 11.907 | 254.491 | 198.455 |
| 8 | 0.237 | 13.804 | 19.844 | 13.178 | 248.049 | 164.724 |
| 10 | 0.22 | 13.84 | 21.377 | 13.144 | 213.773 | 131.436 |
| 16 | 0.189 | 12.946 | 24.884 | 14.051 | 155.522 | 87.821 |

Table 4 shows that for datasets 1 and 2, a sequential execution with one thread exists, with an increase in the number of threads with MPI. **Speed up = (sequential time with 1 thread / parallel time with n thread), while efficiency = (speed up / number of thread) * 100**. The table starts running with one thread up to sixteen threads, with record execution time for datasets 1 and 2. Then, we calculate the speedup and efficiency for each result.

In speedup result, a constant increasing occurs when the number of threads is increased, considering that the implemented MPI using bubble and merge sort on the same CPU, not through a network or distributed network, provides a huge advantage.

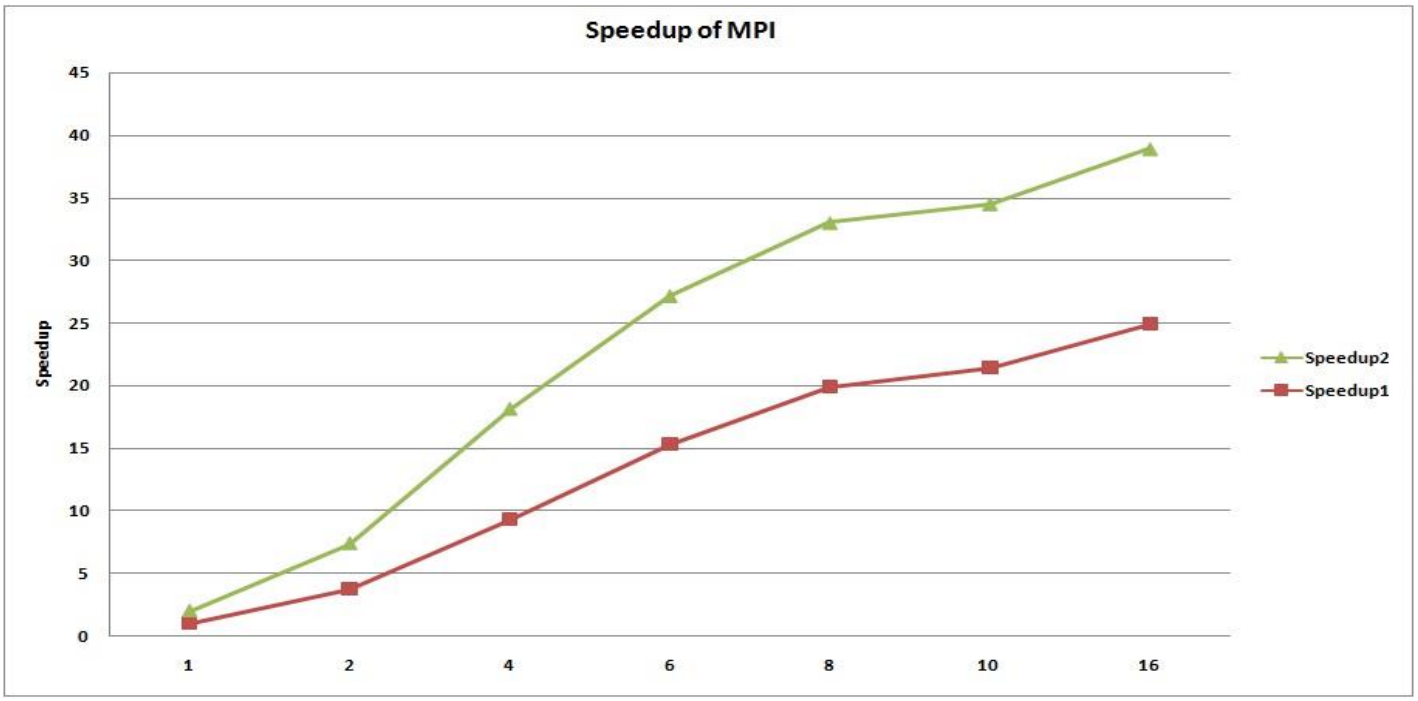


**Figure 2.** Speedup of MPI with Dataset1 and Database2

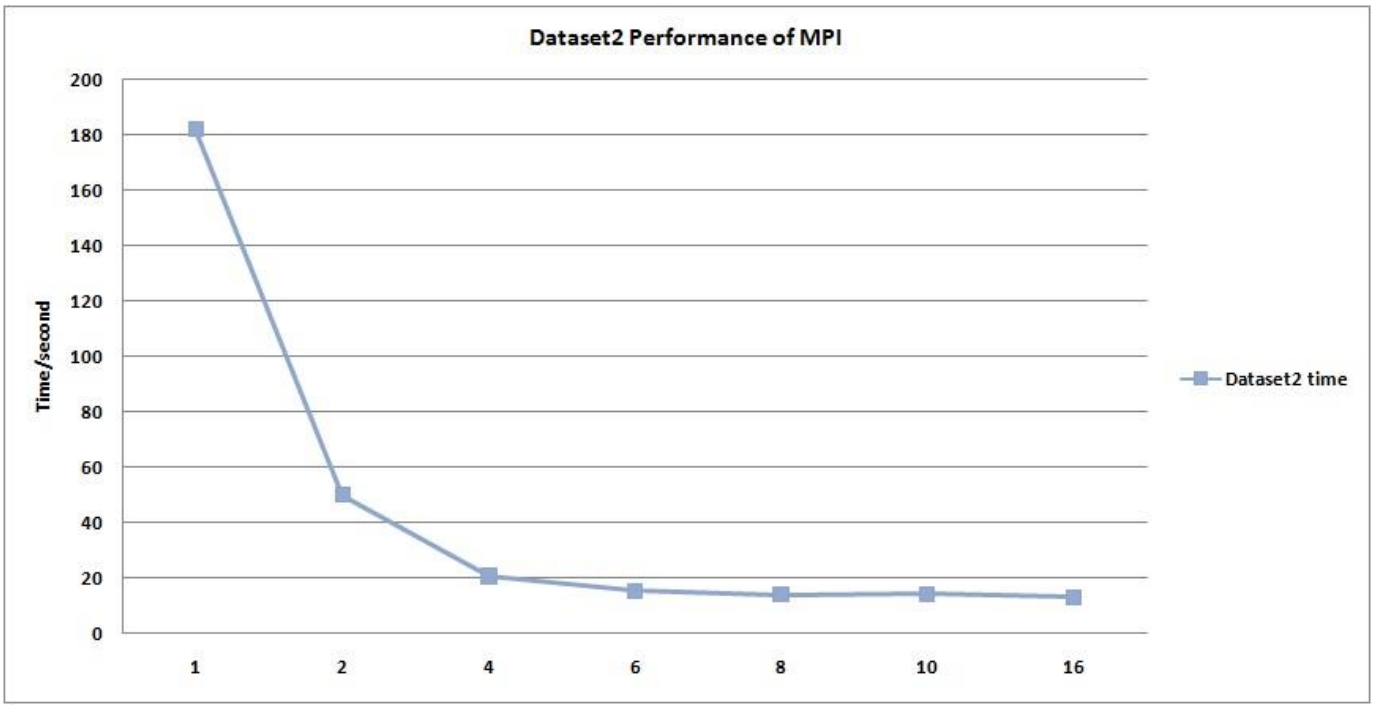


**Figure 3** (a) Efficiency of MPI for Database2

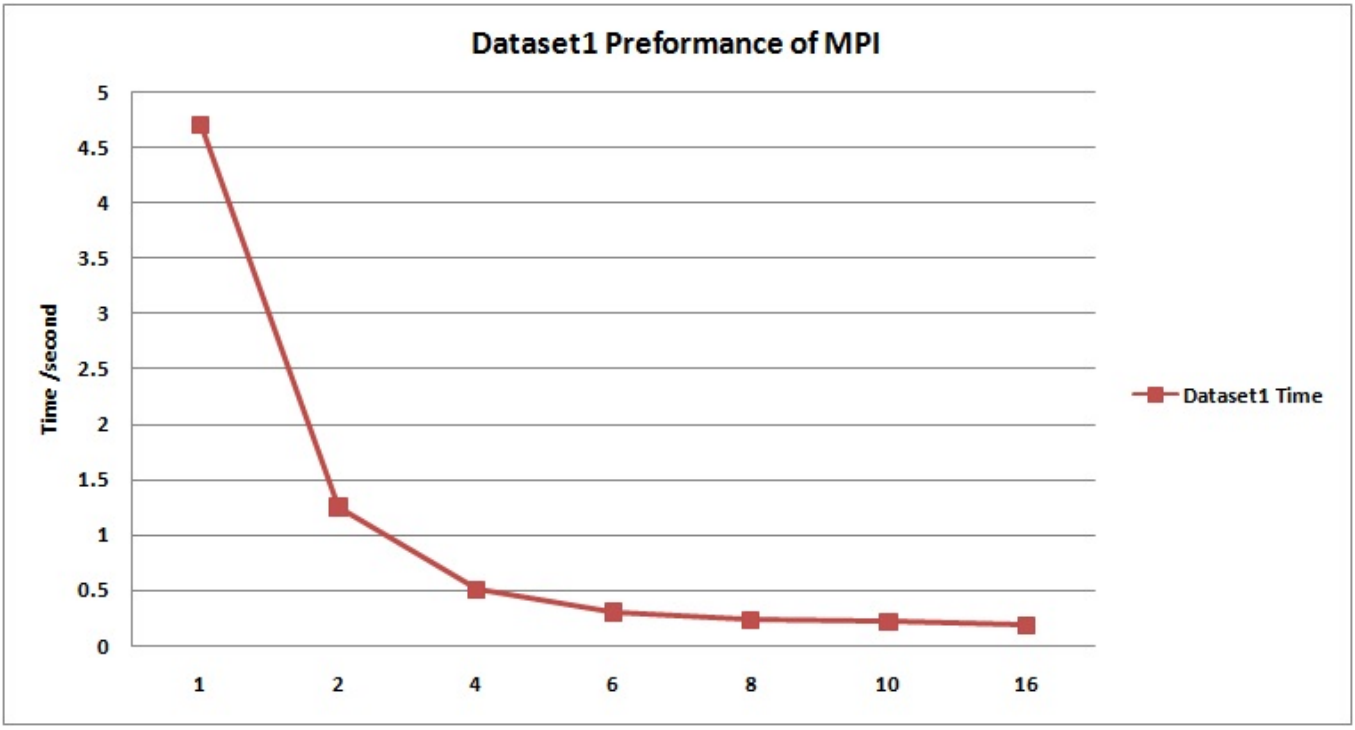


**Figure 3** (b) Efficiency of MPI for Dataset1

Efficiency is on the highest point, when the number of thread equals to one. This phenomenon occurs because this thread does all the work, but the efficiency will drop when the number of thread increases because some of the threads will require some time to let the other thread work, or the required time to manage working thread. Therefore, these issues lead to a reduced efficiency of working threads.

## 4. Conclusions

In this paper, we use parallel computing (i.e., MPI) to enhance the performance and efficiency of bubble sort. Results of this study indicate an increase in speed and efficiency when the number of thread increases, in which part of the data is sorted by one of the slaves, then, they are merged in one list at the end. Furthermore, the early test shows that the data structure has an effect on the performance of the bubble sort, which our 3D array of char performs significantly better from the vector of string in C++, considering the vector-added overhead. Future work is still required to study the use of parallel processing (e.g., GPGPU) to implement bubble sort.

## References

[1] T. H. Cormen, "Introduction to algorithms", *MIT press*, 2009.

[2] J. B. Hayfron-Acquah, Obed Appiah, K. Riverson. "Improved Selection Sort Algorithm", *International Journal of Computer Applications*, vol. 110, no.5, pp. 29-33, *2015*.

[3] S. Altukhaim, "Bubble Sort Algorithm", *Florida Institute of Technology*, 2003.

[4] N. *Wirth*, "Algorithms and Data Structures, Upper Saddle River", *NJ: Prentice-Hall*, 1978.

[5] G. S. Almasi, Allan Gottlieb, "Highly parallel computing" 1988.

[6] L. Snyder C. Lin, "Principles of parallel programming" *Addison-Wesley Publishing Company*, 2011.

[7] B. Blaise, "Introduction to Parallel Computing", *Lawrence Livermore National Laboratory*, 2010.

[8] W. Gropp, E. Lusk, N. Doss, A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard", *Parallel computing*, vol. 22, no. 6, pp. 789-828, 1996.

[9] B. V. Protopopov, A. Skjellum, "A multithreaded message passing interface (MPI) architecture: Performance and program issues", *Journal of Parallel and Distributed Computing*, vol. 61, no. 4, pp. 449-466, 2001.

[10] A. D. Mishra, D. Garg, "Selection of Best Sorting Algorithm", *International Journal of Intelligent Information Processing*, vol. 2, no. 2, pp. 363-368, 2008.

[11] S. Kaur, T. S. Sodhi, P. Kumar. "Freezing Sort", *International Journal of Applied Information Systems* vol. 2, no. 4, pp. 18-21, 2012.

[12] V. Mansotra, K. Sourabh. "Implementing Bubble Sort Using a New Approach", *proceedings of 5th National Conference*, 2011.

[13] M. J. Mundra, M. B. Pal. "Minimizing Execution Time of Bubble Sort Algorithm", (2015).

[14] N. Arora, S. Kumar, V. K. Tamta, "A Novel Sorting Algorithm and Comparison with Bubble Sort and Insertion Sort", *International Journal of Computer Applications*, vol. 45, no. 1, 2012.

[15] R. M. Patelia, S. D. Vyas, P. S. Vyas, "An Analysis and Design of Optimized Bubble Sort Algorithm", *IJRIT*

*International Journal of Research in Information Technology*, vol. 3, no. 1, pp. 65-68, 2015.

[16] A. I. Elnashar, "Parallel Performance of MPI Sorting Algorithms on Dual-Core Processor Windows-Based Systems", *arXiv preprint arXiv:1105.6040* (2011).

[17] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, A. Gatherer, "Implementing OpenMP on a high performance embedded multicore MPSoC", *IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS*, pp. 1-8, Jan. 2009.