Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 6.017

IJCSMC, Vol. 7, Issue. 10, October 2018, pg.37 – 43

Merge Sort Using Sequential and Parallel Programming

Agnes S. Suguitan[1], Lucille N. Dacaymat[2]

[1] College of Computer Science, Don Mariano Marcos Memorial State University - South La Union Campus, Philippines
[2] College of Computer Science, Don Mariano Marcos Memorial State University - South La Union Campus, Philippines
[1] agnessuguitan28@gmail.com; [2] cil.dacaymat@gmail.com

Abstract– Merge sort is considered to be one of the fastest sorting algorithms. It recursively divides the data set into subarrays until each subarray consists of a single element. The goal of this paper is to analyze the performance of merge sort using sequential and parallel programming. Both algorithms were implemented in Python and were tested for various array sizes. Performance of the sequential and parallel merge sort were evaluated with respect to execution time. The results show that the smaller the number of elements gives good performance when executed by a sequential programming but as the number of elements increases, it is best performed by using parallel programming than sequential.

Keywords: merge sort, sequential, parallel, performance, algorithm, sorting

---

I. INTRODUCTION

Sorting is one of the fundamental problems in computer science. Over the years, researchers have developed many algorithms to solve this problem. Many of these algorithms have been developed to work on single CPU machines. Some of these single CPU sorting algorithms are Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort and Radix Sort.

However, in recent years' computer systems have been using more and more cores in processors. Nowadays, even many types of smartphones have quad core processors. Since the year 2004, the trend in processor technology has been to put more cores instead of increasing the clock speed. This trend requires us to develop parallel algorithms for the important problems in computer science. Therefore, we need to develop more parallel algorithms for the sorting problem.

One of the sorting algorithms is the merge sort. It is an efficient divide-and-conquer sorting algorithm. Because merge sort is easier to understand than other useful divide-and-conquer methods. One common example of parallel processing is the implementation of the merge sort within a parallel processing environment. With computers being networked today, it has become possible to share resources like files, printers, scanners, fax machines, email servers, etc. One such resource that can be shared but is generally not, is the CPU. Today's processors are highly advanced and very fast, capable of thousands of

operations per second. If this computing power is used collaboratively to solve bigger problems, the time taken to solve the problem can reduce drastically. However, the whole operation of parallel processing also depends on the RAM available to the processors for their computation.

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows: divide the n-element sequence to be sorted into two subsequences of n=2 elements each, sort the two subsequences recursively using merge sort, and merge the two sorted subsequences to produce the sorted answer [1]. Algorithm 1 shows the sequential merge sort algorithm while Algorithm 2 shows the definition for the merge function.

Algorithm 1. The algorithm for merge sort

```
MERGE-SORT(A, p, r)
1.    if p < r
2.    q= [(p+r)/2]
3.    MERGE-SORT(A, p, q)
4.    MERGE-SORT(A, q+1, r)
5.    MERGE(A, p, q, r)
```

Algorithm 2. The algorithm for merge

```
MERGE (A, p, q, r)
1.    n1 = q – p + 1
2.    n2 = r – q
3.    let L[1…n1 +1] and R[1…n2 +1] be new arrays
4.    for i = 1 to n1
5.    L[i] = A[p + i – 1]
6.    for j = 1 to n2
7.    R[j] = A[q + j]
8.    L[n1 + 1] = ∞
9.    R[n2 + 1] = ∞
10.   i = 1
11.   j = 1
12.   for k = p to r
13.   if L[i] ⩽ R[j]
14.   A[k] = L[i]
15.   i = i + 1
16.   else A[k] = R[j]
17.   j = j + 1
```

Merge sort recursively divides the data set into subarrays until each subarray consists of a single element. It then merges each subarray back, sorting each new subarray as it builds its way back to a single sorted array. Regardless of the shape of the data set to be sorted, merge sort performs the same number of steps, and will therefore have the same time complexity for all cases, O(n log n), Even though it is an efficient algorithm in terms of sorting, it has a drawback in that it uses O(n) extra memory when sorting. This makes it inefficient if memory usage is a key aspect when choosing a sorting algorithm to use. Due to it being a divide and conquer algorithm, it is possible to implement it in parallel [2].

A sequential sorting algorithm [2] executes itself in sequential, meaning it executes all instructions from start to finish without any extra assistance. In contrast, a parallel algorithm executes multiple instructions at the same time using different processing devices, such as multi-cores or threads, before combining it all back together in the end. What often distinguishes sequential from parallel algorithms are that parallel algorithms can break down a problem into smaller subproblems. Divide and conquer is an algorithm design paradigm that work this way, and is a natural fit for parallel implementation. Quicksort and merge sort are examples of divide and conquer algorithms, that both can be implemented in parallel.

Parallel computing is now considered as standard way for computational scientists and engineers to solve problems in areas as diverse as galactic evolution, climate modeling, aircraft design, and molecular dynamics. Parallel computer has roughly classified as multicomputer and multiprocessor. Multi-core technology means having more than one core inside a single chip. This opens a way to the parallel computation, where multiple parts of a program are executed in parallel at same time. Thread-level parallelism could be a well-known strategy to improve processor performance. So, this results in multithreaded processors.

For this paper, the researchers choose merge sort algorithm to compare its implementation in a sequential and parallel program. The following section presents our work into context with existing publications. Section III presents the results of testing cases, comparing the performance of sequential and parallel implementations with respect to the number of elements (n) sorted. In Section IV, we draw some conclusions from the findings.

II. REVIEW OF LITERATURE

One of the fundamental issues in computer science is ordering a list of items. Although there is a huge number of sorting algorithms, sorting problem has attracted a great deal of research; because efficient sorting is important to optimize the use of other algorithms. Sorting algorithms have been studied extensively since past three decades. Their uses are found in many applications including real-time systems, operating systems, and discrete event simulations. In most cases, the efficiency of an application itself depends on usage of a sorting algorithm [3].

Divide and conquer [4] is a design perspective that works with multi branch recursion. Since recursion is utilized as a part of divide and conquer paradigm for solving sub problems, so it's a need that each sub problem should be smaller enough in compared to the original problem and there should be a base case for sub problems. A problem is broken up into small sub problems of the same kind using the recursion; this practice is repeated until problem gets to be adequately basic to be explained easily as shown in figure 1. The solution of these small arrangements is done and it is joined to give arrangement of the first problem.

Merge sort is recursive algorithm that works on divide-and-conquer approach [5], it always partitions the input array into two equal parts, this process partitioning continues until each sub array contains one element. Recursion is used for splitting the array into two equal arrays [6].

The complexity of merge sort [7] is supposing T (n) is the number of comparisons needed to sort an array of n elements by the Merge Sort algorithm. By splitting an array in two parts we reduced a problem to sorting two parts but smaller sizes, namely n/2. Each part can be sort in T(n/2). Finally, on the last step we perform n-1 comparisons to merge these two parts in one. All together, we have the following equation $T(n) = 2*T(n/2) + n - 1$.

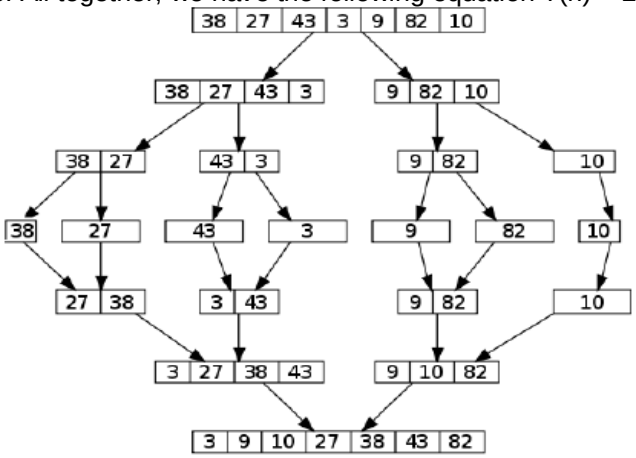

Figure 1. Example of Merge Sort

In the study of Manwade, [8] evaluated the performance of parallel merge sort algorithm on loosely coupled architecture and compare it with theoretical analysis. It has been found that the computational time of the algorithm varies logarithmically for varying number of processors scenario. Also it is found that for varying number of elements the computational time varies linearly. It is also found that the practical analysis closely matches with theoretical analysis.

A similar study was made by Mahafzah (2013). [9] Instead of using multithreading in Java, POSIX Threads (or Pthreads) was utilized and implemented in C++. It was found out that a parallel implementation of quicksort is more time-efficient than a traditional sequential one. In this experiment, the implementation that utilized four threads proved to be most efficient. This is due to the fact that the test environment being used, a dual-core processor, offered two threads to be run for each core, thus enables four threads to run simultaneously, increasing the resources available.

Conventional programs [10] can change the values of variables during the course of execution and so to ensure uniquely defined results, sequential execution is required. The advantage of sequential approach is that it is easy to implement

and easy to follow the execution steps. The analysis and debugging of these programs is hard in view of the fact that program statements are not that independent of each other and so many interactions need to be considered; and a particular value depend on a change made many operations from the past. Parallel programming is even harder, particularly if the programmer explicitly handles the coordination.

Parallel computing [11] operates on the principle that large problems can often be divided into smaller ones, which are then solved concurrently to save time (wall clock time) by taking advantage of non-local resources and overcoming memory constraints.

The main fact to 'parallelise' [11] the program code, is to reduce the amount of time it takes to run. Consider the time it takes for a program to run (T) to be the number of instructions to be executed (I) multiplied by the average time it takes to complete the computation on each instruction (tav)

$$T = I \times tav.$$

In this case, [12] it will take a serial program approximately time T to run. If you wanted to decrease the run time for this program without changing the code, you would need to increase the speed of the processor doing the calculations. However, it is not viable to continue increasing the processor speed indefinitely because the power required to run the processor is also increasing. With the increase in power used, there is an equivalent increase in the amount of heat generated by the processor which is much harder for the heat sink to remove at a reasonable speed.

Parallel computing has been everywhere for several years but it is only recently that awareness has grown due to the advent of multicore processor at a reasonable value for the people.

III. OBJECTIVE

We have implemented sequential and parallel algorithm using Python to analyze the merge sort execution time. A sequential program, executing on a single processor can only perform one computation at a time, whereas the parallel program executed in parallel and divides up perfectly among the multi-processors. The main objective of this study is to compare and evaluate the performance of merge sort in terms of using sequential and parallel.

IV. OVERVIEW

The researchers used the Python to achieve the performance of merge sort algorithm through sequential and parallel programming.

The merge sort algorithm is implemented in developing a sequential and parallel program. The program accepts number of elements that will be sorted using merge sort and executed through sequential and parallel. When the program has executed the sequential and parallel, it will calculate the execution time according to its best case, average case and worst case. Lastly, the researchers compared and analyzed the result of the program.

V. RESULTS AND DISCUSSIONS

Both sequential merge sort and parallel merge sort are implemented in Python. The performance of the implemented algorithms was evaluated on a data sequence consisting of 100 to 1 million elements, in average case, best case and worst case.

Results show that parallel merge sort is better than sequential for sorting large data sequence. For small size of data sequence up to 100,000 elements, sequential merge sort is better than parallelized merge sort. But for data sequence with more than 100,000 elements, parallel merge sort takes less execution time than sequential merge sort.

Table I shows the execution time of sequential and parallel merge sort in average case. In this case, the data sequence to be sorted consists of randomly generated numbers, where the algorithm is expected to perform in average number of steps. Fig. 2 shows that sequential merge sort offers better performance for small data sequence with up to 100,000 elements while parallel merge sort gives almost 1.2x speed up than sequential merge sort for large data sequence. This is because of the number of threads that work independently and simultaneously to sort the large data sequence.

TABLE I. EXECUTION TIME OF SEQUENTIAL AND PARALLEL MERGE SORT (AVERAGE CASE)

Number of Elements	Execution Time (in seconds)	
	Sequential Merge Sort	***Parallel Merge Sort***
100	0.001000166	0.222999811
1K	0.007999897	0.250000000
10K	0.116000175	2.032999990
100K	2.049999952	2.141000030
1M	131.375000000	108.272000070

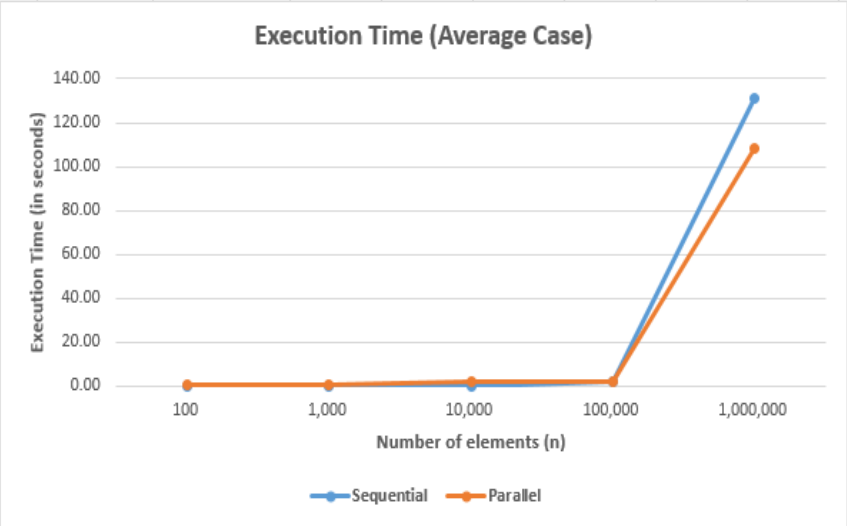

Fig. 2 Performance Comparison of Sequential and Parallel Merge Sort (Average Case)

Table II shows the execution time of both sequential and parallel merge sort in best case. In this case, the merge sort algorithm is implemented on a sorted data sequence. For small data sequence with up to 100,000 elements, sequential merge sort gives better performance while parallel merge sort takes less execution time in sorting for large data sequence. As shown in Fig. 3.

TABLE II. EXECUTION TIME OF SEQUENTIAL AND PARALLEL MERGE SORT (BEST CASE)

Number of Elements	Execution Time (in seconds)	
	Sequential Merge Sort	***Parallel Merge Sort***
100	0.001000166	0.309000020
1K	0.006000040	0.260999920
10K	0.074000120	0.447999950
100K	1.194000006	1.539999960
1M	66.468000170	54.568000080

Fig. 3 Performance Comparison of Sequential and Parallel Merge Sort (Best Case)

Table III shows the execution time of sequential merge sort and parallel merge sort tested in worst case. In this case, the data sequence is in descending order. As shown in Fig. 4, sequential merge sort takes less than a second in sorting a data sequence with up to 10,000 elements and performs better that parallel merge sort when the number of elements increase to 100,000. Parallel merge sort is more efficient when sorting large data sequence consisting of millions of elements.

TABLE III. EXECUTION TIME OF SEQUENTIAL AND PARALLEL MERGE SORT (WORST CASE)

Number of Elements	Execution Time (in seconds)	
	Sequential Merge Sort	***Parallel Merge Sort***
100	0.000999930	0.235000130
1K	0.005999800	0.322000030
10K	0.074001200	0.508000140
100K	1.214999100	1.603000160
1M	66.240999937	50.454999924

Fig. 4 Performance Comparison of Sequential and Parallel Merge Sort (Worst Case)

VI. CONCLUSIONS

Based from the results, the researchers arrived at the following conclusions: (1) the smaller the number of elements or data give good performance when executed by a sequential programming, (2) as elements increases, it is best performed by using parallel programming than sequential.

REFERENCES

[1] T. Cormen, C. Leiserson., R. Rivest, and C. Stein, "*Introduction to Algorithms*", 3rd ed. MIT Press, 2009.

[2] K. Jouper. and H., "*Nordin Performance Analysis of Multithreaded Sorting Algorithms*", Dept. Computer Science & Engineering, Blekinge Institute of Technology SE–371 79 Karlskrona, Sweden, 2015.

[3] G. Kocher and N. Agrawal, "*Analysis and Review of Sorting Algorithms*", International Journal of Scientific Engineering and Research (IJSER), Vol. 2 (3), pp. 81-84, 2014.

[4] T. Singh and D. K. Srivastava, "*Threshold Analysis and Comparison of Sequential and Parallel Divide and Conquer Sorting Algorithms*", International Journal of Computer Applications (0975 – 8887), Volume 145 – No.10, July 2016.

[5] Sabahat Saleem, M. IkramUllah Lali1, M. Saqib Nawaz1 and Abou Bakar Nauman, "*Multi-Core Program Optimization: Parallel Sorting Algorithms in Intel Cilk Plus*", International Journal of Hybrid Information Technology, Vol.7, No.2 pp.151-164, 2014.

[6] Alaa Ismail El-Nashar, "*Parallel Performance Of Mpi Sorting Algorithms On Dual–Core Processor Windows-Based Systems*", International Journal of Distributed and Parallel Systems (IJDPS), Vol.2, No.3, May 2011.

[7] G. Kocher and N. Agrawal, "*Analysis and Review of Sorting Algorithms*", International Journal of Scientific Engineering and Research (IJSER), Volume 2 Issue 3, March 2014.

[8] K. Manwade, "*Analysis of Parallel Merge Sort Algorithm*", International Journal of Computer Applications (0975 - 8887), Vol. 1(19), pp. 70-73, 2010.

[9] B. A. Mahafzah, "*Performance Assessment Of Multithreaded Quicksort Algorithm on Simultaneous Multithreaded Architecture*", The Journal of Supercomputing, 66(1), 339-363, 2010.

[10] R. B. Yehezkel, "*First Steps in Computer Science: Sequential or Parallel*," University of Reims, France,May 2000. *(references)*

[11] B. R. Nanjesh, K.S. Vinay Kumar, C. K. Madhu and K. G. Hareesh, "*MPI Based Cluster Computing for Performance Evaluation of Parallel Applications*", Proceedings of 2013 IEEE Conference on Information and Communication Technologies (ICT 2013).

[12] B. Mustafa and W. Ahmed, "*Parallel Algorithm Performance Analysis Using OpenMP for Multicore Machines*", International Journal of Advanced Computer Technology (IJACT), ISSN:2319-7900, Volume 4, Number 5.

