Empirical Analysis Measuring the Performance of Multi-threading in Parallel Merge Sort

Muhyidean Altarawneh[1], Umur Inan[2]
Department of Computer Science
Maharishi International University, Fairfield, Iowa, USA

Basima Elshqeirat[3]
Department of Computer Science
University of Jordan, Amman, Jordan

***Abstract*—Sorting is one of the most frequent concerns in Computer Science, various sorting algorithms were invented for specific requirements. As these requirements and capabilities grow, sequential processing becomes inefficient. Therefore, algorithms are being enhanced to run in parallel to achieve better performance. Performing algorithms in parallel differ depending on the degree of multi-threading. This study determines the optimal number of threads to use in parallel merge sort. Furthermore, it provides a comparative analysis of various degrees of multithreading. The implementation in this empirical experiment takes a group of devices with various specifications. For each device, it takes fixed-sized data set and executes merge sort for sequential and parallel algorithms. For each device, the lowest average runtime is used to measure the efficiency of the experiment. In all experiments, single-threaded is more efficient when the data size is less than 10^5 since it claimed 53% of the lowest runtime than the multithreaded executions. The overall average of the experiments shows either four or eight threads, with 72% and 28%, respectively, are most efficient when data sizes exceed 10^5.**

Keywords*—Parallel merge sort; sort; multithread; degree of multithreading

I. Introduction

Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945, it is an efficient, general-purpose, comparison-based sorting algorithm [1]. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. A detailed description and analysis of bottom-up merge sort appeared in a report by Goldstine and Neumann as early as 1948 [2]. Such divide and conquer algorithm recursively break down a problem into sub-problems, making it simple to be solved easily, then combine the solutions of the sub-problems until the original problem is solved. In sorting n objects (list of array elements), merge sort is an efficient algorithm that has an average and worst-case performance of O($n$log$n$) [2].

If the running time of merge sort for a list of length n is T(n), then the recurrence T(n) = 2T(n/2) + n follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list and add the n steps taken to merge the resulting two lists). In the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than ($n$log$n$ − 2log n + 1), which is between ($n$log$n$ − n + 1) and ($n$log$n$ + n + O(log$n$)) [3]. In the section below, a pseudo-code of merge sort is illustrated, followed by an example in Fig. 1, using a simple data set of {38,27,43,3,9,82,10} [4].

Fig. 1 illustrates how the algorithm divides all items one by one then combines them recursively. This approach indicates the possibility of applying the algorithm in parallel. Hence, parallel merge sort reduces the complexity to O($n$log$n$/t), where t is the number of threads, by using multi-threaded operations where the data is divided into equal portions and each portion is assigned to a specific thread. The complexity is reduced to O(n) but could vary according to the number of threads used [5].

Merge sort is suitable when the data structure is a linked list because it is a sequential access structure. Implementing a linked list hinders the performance of other algorithms such as quicksort and heapsort [6,7]. Moreover, parallel merge sort is frequently used in various domains, including; sorting NoSql databases [8], high-performance computing environments [9], and massively parallel architectures [10,11].

Algorithm 1 Merge Sort

```
1:  procedure Mergesort
2:  var list left ,right , result
3:  if length(m) ≤ 1 then return m
4:  else
5:      var middle = length(m) / 2
6:      for each x in m up to middle do
7:          add x to left
8:      end for
9:      for each x in m after middle do
10:         add x to right
11:     end for
12:     left ← mergesort(left)
13:     right ← mergesort(right)
14:     result ← merge(left, right) return result
15:
```

When it comes to executing algorithms in parallel, most studies show results of the performance on several processors [12-15]. These results will mainly rely on the specifications of the device and the behavior of the execution in terms of multithreading. The question that led to this research is, what is the suitable degree of multi-threading required for parallel merge sort? This study conducts an empirical experiment and highlights several factors that influence multithreading performance. First, the number of cores that affect multithreading performance and second, the given data size that demands multithreading when a single-threaded performance degrades.

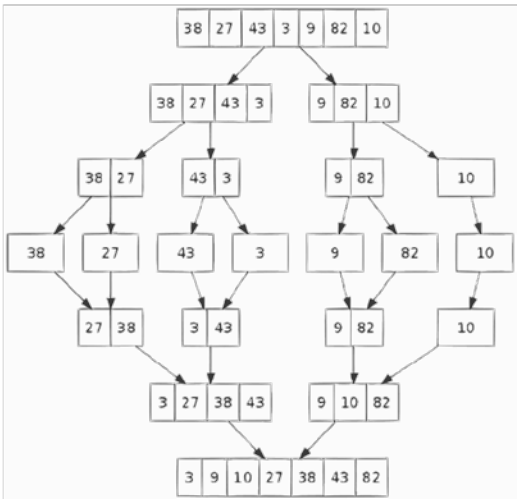

Fig. 1. Merge Sort Algorithm.

The contribution of this paper is to determine the optimal number of threads to use in parallel merge sort. Furthermore, it provides a comparative analysis of various degrees of multithreading. Each data size is examined among a determined number of threads, starting from one thread (sequential), two, four, eight, and sixteen threads (parallel).

In Section 2, related studies were taken to see how parallel merge sort was implemented and what the results were. Section 3 explains and walks through how the experiment was conducted. The results are illustrated in Section 4 and elucidated in the discussion. Finally, Section 5 presents the conclusion of this study.

II. Related Work

There have been several papers that conducted various researches on parallel merge sort, and they have come up with the following.

Jeon [13] improved parallel merge sort by distributing and computing the approximately equal number of keys in all processors throughout the merging phases. Using the histogram information, keys can be divided equally regardless of their distribution, which evaluated the speedup showing a better performance by applying parallel merge sort on two different parallel machines: a Cray T3E and a Pentium III PC cluster on maximum data size of $10^6 \times 4$.

The tested algorithm on loosely coupled parallel machines and the performance of the algorithm has been observed. It has been found that the computational time of the algorithm varies logarithmically for a varying number of processors scenario [14].

Uyar [5] experimented with applying parallel merge sort using multi-threads similar to this experiment. It stated that two threads could perform one merge operation simultaneously. One thread generates the first half of the sorted values that start from the minimums of the two sorted subsets. The other thread generates the second half of the sorted values starting from the maximums of the two sorted subsets. It also compared it with double merging by using four threads implementing it on Java. The comparison focused on array sizes from 10 million up to 50 million. In this study, the array size starts from 5000 up to 50 million to detect when executing in parallel is more efficient than sequential.

A study was conducted on three parallel sorting algorithms (Odd-even transposition sort, Parallel rank sort, and Parallel merge sort) on a number of processors 2, 4, 6, 8, 10, and 12 on 10000 integers [15]. The results proved that parallel merge sort was the fastest, yet the study was comparing only one input size and may differ when the data size increases.

These previous studies show that merge sort could be conducted in parallel in several ways, giving better results than sequential as the array size increases [5,13-15]. Yet, these studies were concerned with enhancing the performance of merge sort without comparing the degree of multi-threading. Only [5] compared different array sizes that were only applied up to four threads on a specific range of sizes, from 10^6 to $10^6 \times 5$. This study experiments parallel merge sort on four different degrees of multi-threading in a broader range of array sizes from 10^5 to 10^7, which is explained in Section 3 maintaining the integrity of the specifications.

III. Experiment

A. Requirements

This experiment was implemented on Java SE8. It was conducted on five devices to ensure diversity in the environment of implementation. Moreover, to verify the results are not dependent on the specifications of a particular device. The specifications of the devices used in this experiment are shown in Table I.

B. Implementation

This experiment takes a specific data set and executes it in two approaches: 1) Sequential (one thread), 2) Parallel (two, four, eight, and sixteen threads). The source code is available on https://github.com/muhyidean/ParallelMergeSort.git.

The implementation in this experiment takes a data set and applies merge sort for sequential and parallel algorithms. For sequential, it executes Algorithm 1. As for parallel, it executes Algorithm 2 based on the following:

1) Data formation: The array sizes for the data sets begin from $10^3 \times 5$, 10^4, $10^4 \times 5$, 10^5, ... up to 10^7. Based on the array size, ten different random data sets are initiated to be implemented in both execution approaches. Each data set will be placed in a separate array and executed in each approach. The average runtime of ten executions for each array size is taken in milliseconds.

TABLE I. Device Specifications

	OS	**Processor**	**# Cores**	**RAM**
Device 1	Windows	Intel i5	4	16
Device 2	Windows	Intel i7	8	16
Device 3	macOS	Intel i5	4	8
Device 4	macOS	Intel i7	8	16
Device 5	Ubuntu	Intel i5	4	4

2) Partition process: The partitioning will be in five categories, one in sequential and four degrees of multi-threading 2, 4, 8, 16. The original data set is considered the first partition, so it will be directly executed (sequentially). Then the same data set is taken and split in half making two data sets, each partition is assigned to a thread to run parallel. The process goes on for the other partitions with respect to the number of threads to be implemented which are two, four, eight, and sixteen.

3) Thread management: The implementation for the parallel merge sort divides the array into sub-arrays to be sorted by the number of threads. The threads sort their assigned sub-arrays independently. Two consecutive sorted sub-arrays are combined by one thread. Each merging thread merges two sorted arrays. The merge operation follows this approach. Whenever the arrays are sorted, the number of arrays is decreased by half. During the last iteration, two sorted arrays are merged to produce a sorted array. This implementation did not use any third-party libraries/frameworks, it was implemented with the java thread package in JDK (Java Development Kit).

Fig. 2 illustrates the partitioning process and the merging mechanism. Each elliptical shape is considered a thread; the shapes labeled with D represent the partition of the original array sorted by merge sort. The shapes labeled with M merge the results from the previous threads until it merges the whole array. To be better illustrated, sixteen threads are not shown Fig. 2 because it follows similar partitioning.

C. Data Analysis

Tables III to VII shows the average runtime for different array sizes on each. Furthermore, they also show how each device performs on different execution approaches (sequential and parallel). For example, the average execution time is calculated by running the algorithm ten times, then the average of times is taken. Table II is one of the execution results for device 4 on array size 10^5. For instance, the result shows that (Th-4) was the most efficient for this case. However, it may differ as the size increases and is subject to the device specifications. For each device, on each data size, it will have a table like Table II.

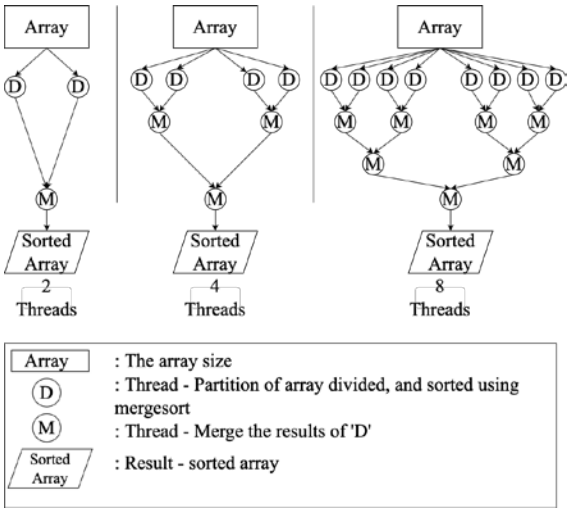

Fig. 2. Parallel Merge Sort using Three Degrees of Multi-threading (2,4,8).

Algorithm 2 Parallel Merge Sort

```
1:  procedure PMergesort
2:  var val ← (v)                                   // v: the number of values here
3:                                                  // x: the number of threads
4:  var list arr test _1[ ], arr test 2[ ], ...arr test x[ ]   // Defining main arrays
5:  var list arr 2[ ] ... arr x[ ]                  // Defining sub arrays
6:                                                  // Defining threads to execute merge sort for each array
7:  threads t1(mergesort(arr 1)), t2(mergesort(arr 2))... t _x(mergesort(arr x))
8:                                                  // Assign random integers to main arrays, to give each same set of random values
9:  for i ← 0 to val do
10: n ← random value in range of (1 − x)
11: arr test1[ ],arr test 2[ ]...arr _test x[ ] ← n
12: end for
13:                                                 // Partition data set and add into x sub arrays for each set of threads
14: var mid ← (length of arr test x/x)              // Get mid points for each partition
15: ∗ repeat code in line 14 for x partitions
16:                                                 // Calculate the time taken for each set of threads
17: var ts ← take current time
18: execute t1 , t2 ... tx                          // Execute threads
19: var te ← take current time
20: ∗ repeat codes in lines (17 − 19) for each set of threads (2,4, 8 ... x )
21: var tr ← ts − te                                // to calculate the time taken in parallel mergesort (x threads)
22: file ← export results(tr1,tr2...trx)            // to take results (time taken in milliseconds)
23: end procedure=0
```

TABLE II. DEVICE 1 – RUNTIME ON SIZE 10^5 (MS)

Execution #	Th-16	Th-8	Th-4	Th-2	Th-1
Execution 1	159	14	18	17	46
Execution 2	33	21	37	50	27
Execution 3	37	22	31	36	36
Execution 4	23	26	30	24	44
Execution 5	32	32	16	32	49
Execution 6	38	48	38	32	32
Execution 7	17	16	32	81	44
Execution 8	31	48	32	33	32
Execution 9	35	33	16	49	34
Execution 10	14	16	25	97	16
Average	**41.9**	**27.6**	**27.5**	**45.1**	**36.0**

IV. RESULTS AND DISCUSSION

This section highlights and points out the main findings of the empirical experiment. To measure the efficiency of the experiment, the lowest average execution time (ms) is taken for each data size on each device.

A. Results

In Tables III to VII, it shows the average of 10 executions for each degree of multi-threading. Each column is a different size starting from $10^3 \times 5$ up to 10^7. The rows show the performance of each thread for a specific data size. For example, (Th-1) is one thread, (Th-2) is two threads and goes on. As shown in Tables III to VII, for data size $10^3 \times 5$, all devices perform efficiently in terms of runtime in a single-threaded execution. As for the sizes 10^4 and $10^4 \times 5$, it varies from one to eight threads depending on the number of cores in the device. With data sizes of 10^5 and larger, each device performs better with a certain number of threads, depending on the number of cores. All results are illustrated in Fig. 3 to 7.

Fig. 3 to 7 illustrates the performance graphs according to different data sizes and the number of threads used. Multithreading is clearly more efficient when the data size increases. The appropriate number of threads will generally be visible when the data size exceeds 10^5.

B. Findings

There were two main findings from these results. First, multithreading does not always have the most efficient runtime as it depends on the data size. Second, even when the data size increases, a specific number of threads will determine the optimized performance based on the device specifications. In other words, implementing as many threads as possible will not lead to higher runtime performance.

Tables VIII and IX were presented to highlight the findings of the results, one below 10^5 and the other greater 10^5. Table VIII shows the overall average for each device with data sizes below 10^5. For example, in Device 1, the sequential runtime performance was most efficient. By taking the overall average, single-threaded was more efficient since it claimed 53% of the lowest runtime than the multithreaded executions. Table IX shows the overall average for each device with data sizes above 10^5. As shown in Table IX, multi-threaded implementation with either four or eight threads provided better performance with 72% and 28%. Fig. 8 and 9 visualize which threads performed better in the overall average for different data sizes. A higher percentage indicates that using a specific number of threads is more efficient on a particular data size.

Based on the experiment results, all devices that have four cores achieved efficient runtime performance with four threads. Moreover, all devices with eight cores achieved efficient runtime performance with eight threads. Evidently, the selection of the number of threads is mainly determined by the number of the cores.

C. Discussion

The main question of this study is, what is the optimal number of threads for parallel merge sort considering two main factors: data size and number of cores?

TABLE III. DEVICE 1 - RESULTS - AVERAGE RUNTIME (MS)

Th(x) = number of threads	Array Size							
	10^3 x 5	10^4	10^4 x 5	10^5	10^5 x 5	10^6	10^6 x 5	10^7
Th-16	18	17	25	42	94	165	678	1303
Th-8	10	18	34	28	90	131	588	1173
Th-4	8	13	26	28	83	130	585	1142
Th-2	8	11	38	45	104	179	818	1724
Th-1	2	4	23	36	145	261	1342	2751

TABLE IV. DEVICE 2 - RESULTS - AVERAGE RUNTIME (MS)

Th(x) = number of threads	Array Size							
	10^3 x 5	10^4	10^4 x 5	10^5	10^5 x 5	10^6	10^6 x 5	10^7
Th-16	3	3	5	9	30	51	228	523
Th-8	3	4	8	20	36	44	201	415
Th-4	4	3	7	19	34	47	251	523
Th-2	7	1	9	20	43	69	371	778
Th-1	1	2	9	34	69	134	693	1442

TABLE V. DEVICE 3 - RESULTS - AVERAGE RUNTIME (MS)

Th(x) = number of threads	Array Size							
	10^3 x 5	10^4	10^4 x 5	10^5	10^5 x 5	10^6	10^6 x 5	10^7
Th-16	4	5	10	23	57	97	501	1011
Th-8	11	5	16	13	45	86	431	943
Th-4	4	3	9	10	43	85	420	859
Th-2	2	3	7	13	62	130	665	1356
Th-1	1	3	13	27	114	206	1100	2258

TABLE VI. DEVICE 4 - RESULTS - AVERAGE RUNTIME (MS)

Th(x) = number of threads	Array Size							
	10^3 x 5	10^4	10^4 x 5	10^5	10^5 x 5	10^6	10^6 x 5	10^7
Th-16	3	3	7	13	27	47	371	671
Th-8	2	3	4	10	23	37	184	474
Th-4	2	2	5	8	32	45	259	604
Th-2	1	2	7	13	44	75	387	801
Th-1	2	2	11	23	93	160	832	1660

TABLE VII. DEVICE 5 - RESULTS - AVERAGE RUNTIME (MS)

Th(x) = number of threads	Array Size							
	10^3 x 5	10^4	10^4 x 5	10^5	10^5 x 5	10^6	10^6 x 5	10^7
Th-16	6	15	24	29	63	108	446	1198
Th-8	7	15	22	27	88	120	440	1064
Th-4	9	11	21	25	63	104	421	931
Th-2	8	12	20	32	87	120	654	1492
Th-1	2	4	23	36	149	261	1336	2200

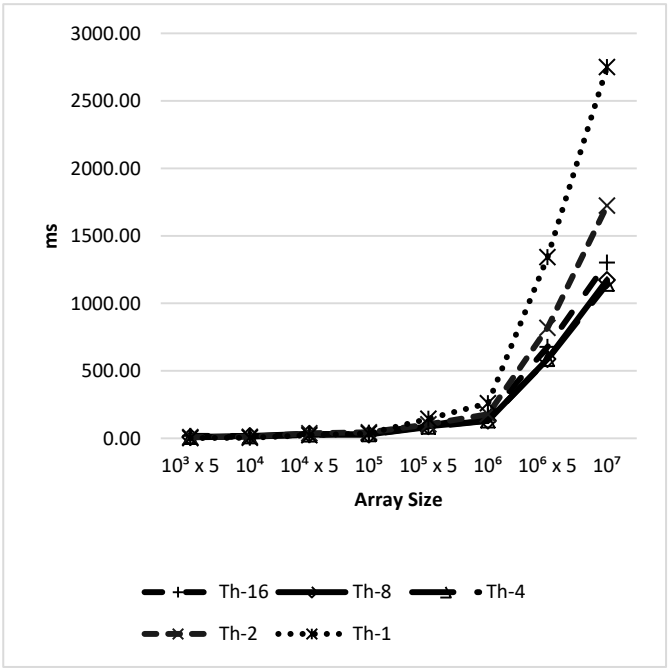

Fig. 3. Device 1 - Results - Average Runtime (MS).

The results of this study had shown that having as many threads as possible will not lead to the best runtime performance. To achieve the best runtime performance, the number of cores present is crucial in determining the optimal number of threads. The cruciallity is due to how multiple threads are executed by the operating system. Correspondingly, the data size determines whether multiple threads are required. In small data sets, the use of multiple threads is unnecessary since one thread can perform more efficiently.

The conclusion is that if the data size is under 10^5, single-threaded will be more efficient. In contrast, having multiple threads will perform better for data sizes that exceed 10^5. In addition, it should not spawn threads more than the number of cores (excluding merging threads).

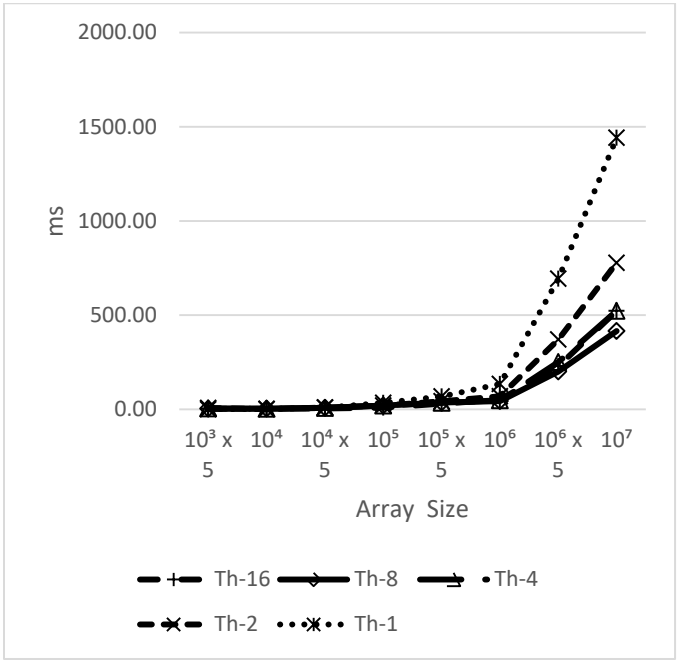

Fig. 4. Device 2 - Results - Average Runtime (MS).

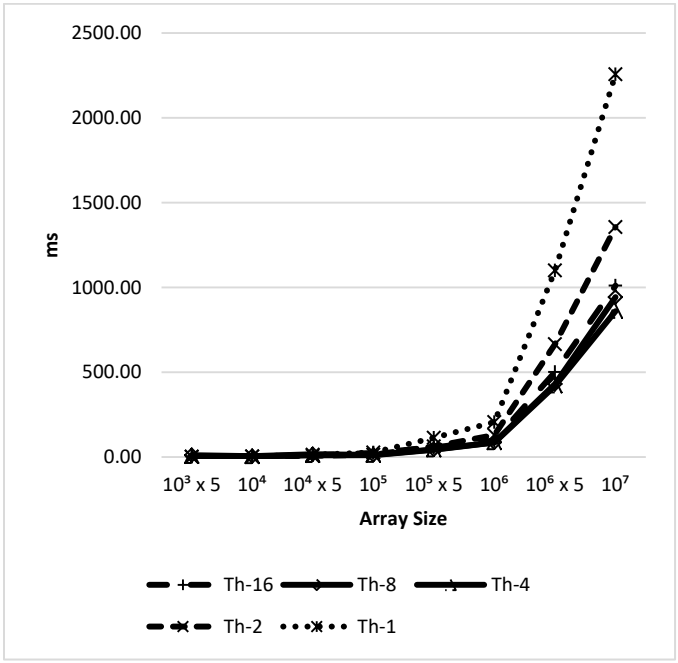

Fig. 5. Device 3 - Results - Average Runtime (MS).

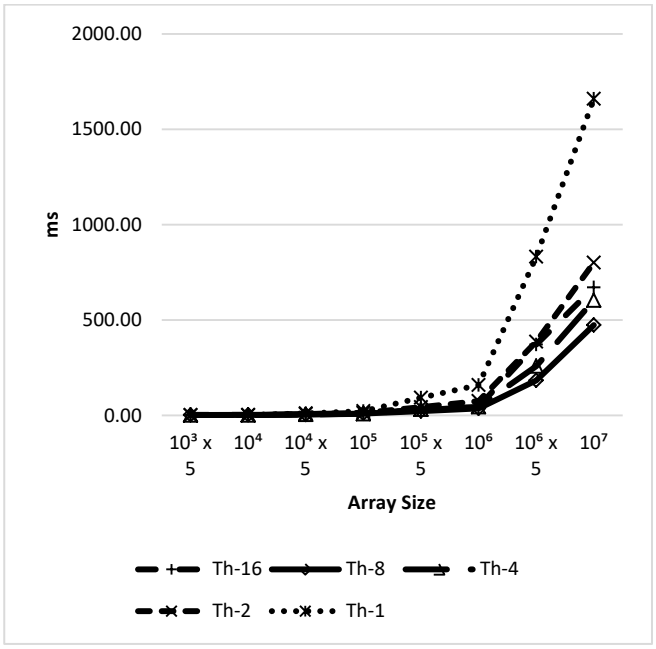

Fig. 6. Device 4 - Results - Average Runtime (MS).

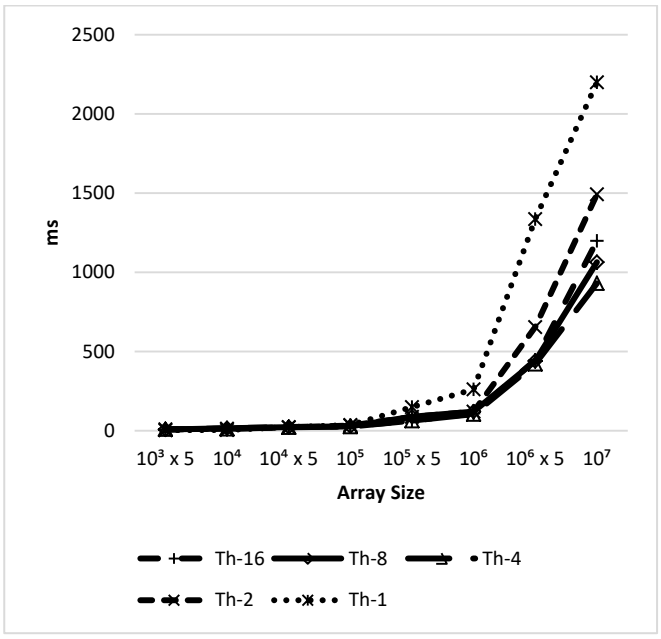

Fig. 7. Device 5 - Results - Average Runtime (MS).

TABLE VIII. MULTITHREADING EFFICIENCY PERCENTAGE (< 50000)

Device	Th(x)= number of threads				
	Th-1	**Th-2**	**Th-4**	**Th-8**	**Th-16**
Device 1	1.00	0	0	0	0
Device 2	0.33	0.33	0.33	0	0
Device 3	0.33	0.66	0	0	0
Device 4	0	0.66	0	0.33	0
Device 5	1.00	0	0	0	0
Average	**0.53**	0.33	0.06	0.06	0

TABLE IX. MULTITHREADING EFFICIENCY PERCENTAGE (> 50000)

Device	Th(x)= number of threads				
	Th-1	**Th-2**	**Th-4**	**Th-8**	**Th-16**
Device 1	0	0	1.00	0	0
Device 2	0	0	0.40	0.60	0
Device 3	0	0	1.00	0	0
Device 4	0	0	0.20	0.80	0
Device 5	0	0	1.00	0	0
Average	0	0	**0.72**	**0.28**	0

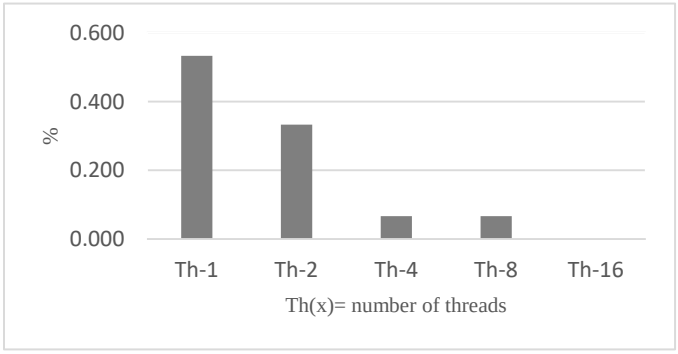

Fig. 8. Multithreading Efficiency Percentage (< 50000).

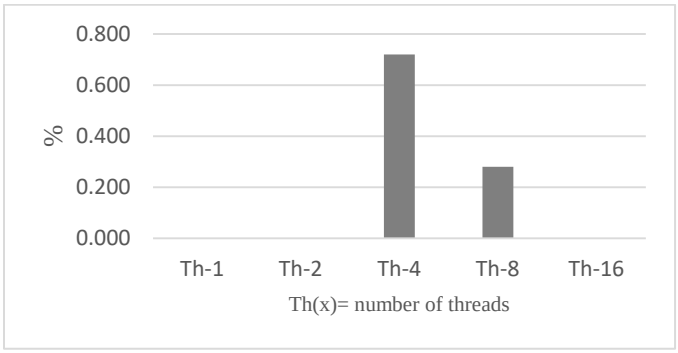

Fig. 9. Multithreading Efficiency Percentage (> 50000).

V. CONCLUSION

This study conducts an empirical experiment to determine the optimal number of threads to use in parallel merge sort. Several factors are discussed in this study to answer this question. First is the number of cores that impact multithreading performance. Second is the given data size that requires the use of multiple cores.

The implementation in this experiment takes a group of devices with various specifications. For each device, it takes fixed-sized data set and applies merge sort for sequential and parallel algorithms. For each device, the lowest average execution time (ms) is used to measure the efficiency of the experiment. Taking the average for all experiments, single-threaded is more efficient when the data size is less than 10^5 since it claimed 53%. Whereas, for data sizes exceeding 10^5, multi-threaded implementation has better performance. The overall average of the experiments shows either four or eight threads are most efficient, with 72% and 28% respectively.

There were two main findings from these results. First, multithreading does not always have the most efficient runtime as it depends on the data size. Second, even when the data size increases, a specific number of threads will determine the optimized performance based on the device specifications. In other words, implementing as many threads as possible will not lead to higher runtime performance.

The conclusion is that if the data size is under 10^5, single-threaded will be more efficient. In contrast, having multiple threads will perform better for data sizes that exceed 10^5. In addition, the number of threads spawned should not exceed the number of cores (excluding merging threads).

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. “Introduction to algorithms,” MIT press, 2009.

[2] J. Katajainen, T. Pasanen, and J. Teuhola. “Practical in-place mergesort,” Nord. J. Comput., 3(1):27–40, 1996.

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

[4] Merge Sort, howpublished = https://www.geeksforgeeks.org/merge-sort/, note = Accessed: 2021-12-01.

[5] A. Uyar. “Parallel merge sort with double merging,” In 2014 IEEE 8th International Conference on Application of Information and Communication Technologies (AICT), pages 1–5. IEEE, 2014.

[6] N. Parlante. “Linked list problems,” Stanford CS Education Library, 1:33, 2002.

[7] A. Abu Dalhoum, T. Kobbay, A. Sleit, M. Alfonseca, and A. Ortega. “Enhancing quicksort algorithm using a dynamic pivot selection technique,” WULFENIA Journal, Austria, 19(10), 2012.

[8] Z. Marszałek. "Parallelization of modified merge sort algorithm." Symmetry 9.9 : 176, 2017.

[9] J. Holke, et al. "Data-adapted Parallel Merge Sort." European Conference on Parallel Processing. Springer, Cham, 2019.

[10] D. P. Singh, Dhirendra Pratap, I. Joshi, and J. Choudhary. "Survey of GPU based sorting algorithms." International Journal of Parallel Programming 46. : 1017-1034, 2018.

[11] K. Raju, N. N. Chiplunkar, and K. Rajanikanth. "A CPU-GPU Cooperative Sorting Approach." 2019 Innovations in Power and Advanced Computing Technologies (i-PACT). Vol. 1. IEEE, 2019.

[12] S. W. Hijazi, and M. Qatawneh. "Study of Performance Evaluation of Binary Search on Merge Sorted Array Using Different Strategies." International Journal of Modern Education and Computer Science 9.12:1, 2017.

[13] M. Jeon and D. Kim. “Parallel merge sort with load balancing,” International Journal of Parallel Programming, 31(1):21–33, 2003.

[14] K. B. Manwade. “Analysis of parallel merge sort algorithm,” International Journal of Computer Applications, 1(19):66–69, 2010.

[15] H. Rashid and K. Qureshi. “A practical performance comparison of parallel sorting algorithms on homogeneous network of workstations,” WSEAS Transactions on Computers, 5(7):1606–1610, 2006.