Comparative Analysis of OpenMP and MPI Parallel Computing Implementations in Team Sort Algorithm

Eko Dwi Nugroho 1*, Ilham Firman Ashari 2*, Muhammad Nashrullah 3*, Muhammad Habib Algifari 4*, Miranti Verdiana 5*
* Teknik Informatika, Institut Teknologi Sumatera
eko.nugroho@if.itera.ac.id [1], firman.ashari@if.itera.ac.id [2], muhammad.118140136@student.itera.ac.id [3], muhammad.algifari@if.itera.ac.id [4], miranti.verdiana@if.itera.ac.id [5]

Article Info

Article history:

Received …
Revised …
Accepted …

Keyword:

MPI,
OpenMP,
Parallel Computation,
Parallelization,
Team Sort Algorithm

ABSTRACT

Tim Sort is a sorting algorithm that combines Merge Sort and Binary Insertion Sort sorting algorithms. Parallel computing is a computational processing technique in parallel or is divided into several parts and carried out simultaneously. The application of parallel computing to algorithms is called parallelization. The purpose of parallelization is to reduce computational processing time, but not all parallelization can reduce computational processing time. Our research aims to analyse the effect of implementing parallel computing on the processing time of the Tim Sort algorithm. The Team Sort algorithm will be parallelized by dividing the flow or data into several parts, then each sorting and recombining them. The libraries we use are OpenMP and MPI, and tests are carried out using up to 16 core processors and data up to 4194304 numbers. The goal to be achieved by comparing the application of OpenMP and MPI to the Team Sort algorithm is to find out and choose which library is better for the case study, so that when there is a similar case, it can be used as a reference for using the library in solving the problem. The results of research for testing using 16 processor cores and the data used prove that the parallelization of the Sort Team algorithm using OpenMP is better with a speed increase of up to 8.48 times, compared to using MPI with a speed increase of 8.4 times. In addition, the increase in speed and efficiency increases as the amount of data increases. However, the increase in efficiency that is obtained by increasing the processor cores decreases.

I. INTRODUCTION

A set of procedures or steps used to solve a problem can be called an algorithm [1]. An algorithm is considered efficient when it can produce the desired solution for each input that matches the problem it needs to solve [2]. The duration of data processing by each algorithm varies, also determined by the amount of data input [3]. Efficiency in solving problems can be disrupted due to protracted computation time, even though the results are still as desired.

Parallel computing methods involve computer processing that is carried out simultaneously or divided into several parts [4]. In an effort to increase computational processing efficiency, parallel computing was developed with the main goal being to increase speed or reduce the time required. Research by Aditya and Abba shows that the Quick Double Merge Sort algorithm can increase its performance up to 3.6 times using eight processors [5]. However, not all algorithms are suitable for implementing parallel computing. Findings from research conducted by Favorisen, Aristotle, and Nadila show that the radix sort algorithm can only be accelerated up to 1.2 times even though it already uses four processors [6]. The information used by them is obtained through a random number generation process.

In computer science and programming, there are two types of parallel computing architectures which are based on computer memory management. The first is shared memory and the second is distributed memory [7]. To implement these two types of architecture, we can use the help of libraries. Shared memory has advantages over distributed memory if the algorithm is not too complicated or the data is not too large. However, if the algorithm is too complicated or the data is too large, distributed memory is a better choice than shared memory. The reason is that in distributed memory, there can be more than one computer connected to each other via a network [8], while in shared memory, usually only one computer is used [9].

There are two measures that can be used to evaluate the performance of a parallel algorithm. One way to measure speed is to use speed up, which describes the comparison of the time required by parallel algorithms and sequential algorithms to solve similar problems. The speed up formula is the result of dividing computational time sequentially with parallel computing time [10]. Ideally, the more processors used, the acceleration value will increase linearly. For example, if n processors are used to run a parallel algorithm, then the ideal value for speed increase is n. Parallelization has the ability to increase computational speed in proportion to the number of processors used [11]. In practice, it rarely happens that the speed up value reaches its ideal value. The reason is because not all parts of the algorithm can be executed in parallel and the distribution of tasks between processors is uneven.

Another measurement method is efficiency, which indicates the degree to which the processor takes advantage of the increase in speed on average. According to one source, efficiency can be calculated by dividing the speed up by the number of processors used [12]. In general, the desired degree of parallel algorithm acceleration is linearly increased as the number of processors used increases. In some situations, the processing speed may reach optimal performance when using multiple processing units. If the processor used in the algorithm continues to be added, the speed increase will not continue to increase linearly, but will reach a certain limit.

This occurs as a result of Amdahl's law. According to Amdahl's law, a program's speed up value is restricted by the portion of the program that cannot be parallelized [13]. The maximum speed up value is one divided by one minus the part of the program that can be parallelized, regardless of how much speed up is done to any component of the code.

Sorting algorithms are the most often employed algorithms in a variety of issues, including biological data processing, solving systems of many-dimensional linear equations, and grid function compression [14]. Tim Sort is a combination of the Merge Sort and Binary Insertion Sort sorting algorithms [15]. Team Sort is the default algorithm in several programming languages, including java, swift, rust, and others. The method has an asymptotic time complexity of O(n*log(n)) [16], an average asymptotic time complexity of O(n), and an asymptotic time complexity of O(n). While O(n) is the worst space complexity.

Our research uses OpenMP sources that support shared memory systems and MPI supports distributed memory systems, and we use the Tim Sort algorithm as our case study. We conducted this research with the aim of increasing computational speed, so that data processing can be performed more efficiently. We also want to compare and test the effect of using OpenMP and MPI parallel computing on Tim Sort's algorithm processing time. The goal to be achieved by comparing the application of OpenMP and MPI to the Team Sort algorithm is to find out and choose which library is better for the case study, so that when there is a similar case, it can be used as a reference for using the library in solving the problem. In addition, we want to investigate how adding processor cores to MPI and increasing the amount of data affect the performance of Tim Sort's algorithm. The benefit of our research is gaining a better understanding of which libraries are most suitable for use in computational processing of various timing problems, as well as the ability to involve parallel computing in solving these problems.

II. METHODOLOGY

A. Research Flow

Our research consists of five main stages, namely identifying problems and conducting related literature studies, designing research models, implementing and testing these models, as well as analysing data and writing research reports. The relationship between these steps is explained through a flow chart which can be seen in Figure 1. The study was conducted for approximately 6 months.

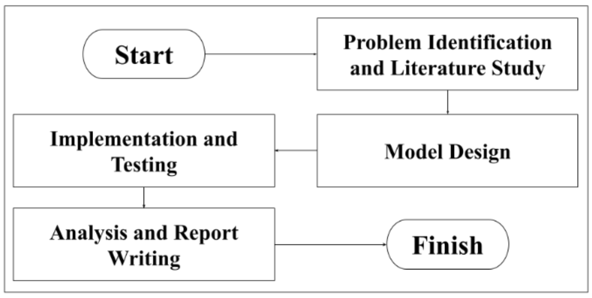

Start
Problem Identification and Literature Study
Model Design
Implementation and Testing
Analysis and Report Writing
Finish

Figure 1. Research Flowchart

1) *Defining the problem and reviewing literature sources*: Researchers examine and evaluate the results of the case studies that have been conducted, after which they create questions, aims, and benefits of the research. Research experts find challenges that can be overcome by several methods obtained from reading sources such as journals and books on parallel computing and algorithms, so as to ensure that the problem has a clearly defined goal. The researchers collected a variety of information about parallel programs, Team Sort algorithms, and evaluation of sorting algorithm performance. In this study, the researcher was able to obtain information regarding guidelines for making parallel programs, the implementation of the Team Sort algorithm, the techniques used in parallel programs, the efficiency of each technique used in parallel programs, and methods for testing model performance. A small experiment was carried out by the researcher to verify whether the method used was appropriate and whether the results met the researcher's expectations.

2) *Model design*: The researcher starts the process of designing a model for Tim Sort's parallel algorithm after obtaining the required information. In designing the model, the researcher separated it into three main stages, namely task mapping, task completion, and combining the results of each task. The design will be described in the form of a flow chart showing the results. In addition to making a model design, the researchers also made a test design for the model. The objective of this trial is to compare the performance of the model and the original algorithm, and to see the impact of adding processor cores on model performance.

3) *Implementation and testing*: In this stage, researchers use and test the pre-designed model. This method is implemented using the C++ programming language and the OpenMP and MPI libraries. The program to be created receives a file containing information that needs to be arranged sequentially. After the program is executed, the result file will contain the sorted data. For this study, we used a dataset that included 4194304 random numbers. After that, the researcher tested the results of the model implementation using the test steps that had been prepared previously. The test results will be used as a record for analysis in the next stage.

4) *Analysis and report writing*: The researcher analyses the data that has been obtained in the previous stage and then writes it in a report. Performance comparison is done by comparing the model and the original algorithm and comparing the performance of the model when using various number of processor cores. This research involves problem analysis, model building, use of equipment, implementation, discussion of results, evaluation, and drawing conclusions which are all explained in the research report.

B. Tools and Materials

1) *Tools*: The equipment we used in this study included: Computer with 16 Core processor specifications, 16 GB RAM, and Ubuntu Linux operating system; GCC version 11.3 is used as the compiler; as well as the C++ programming language version 11.

2) *Materials*: The data used in our study came from a study entitled "Parallel Divide-and-Conquer Algorithm for Bubble Sort, Selection Sort, and Insertion Sort" by Pramod and Rezaul [17]. The data in the study were obtained by generating random numbers. There are 13 choices of numbers in the data, in the order from 2^{10} to 2^{22}.

C. Model Design

The illustration in Figure 2 explains how parallelization is carried out in the Tim Sort algorithm. The data in the image is divided into eight parts and processed in parallel using four threads or processor cores. This process includes the stages of allocating tasks, completing tasks, and combining the results of each task.

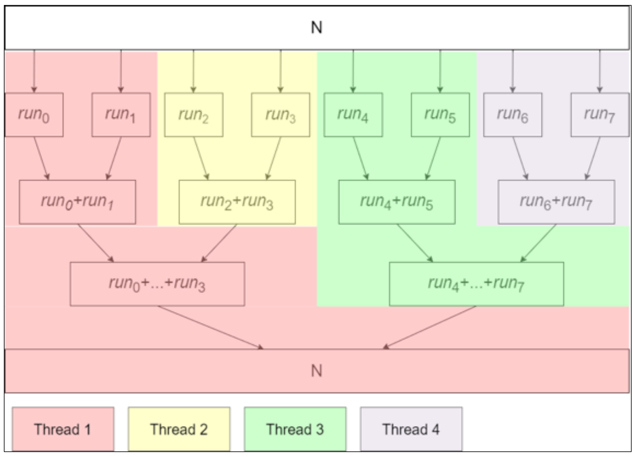
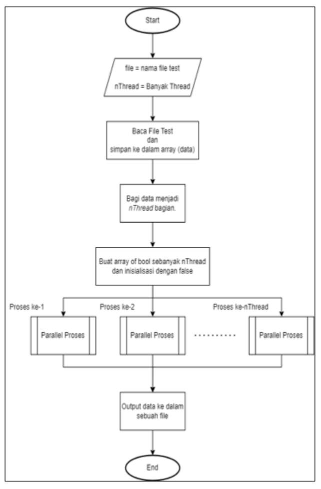


Figure 2. Illustration of the Parallelization of the Tim Sort Algorithm using 4 Processor Threads/Cores

The following is an explanation for each step contained in Figure 2. The first process that must be carried out is the division of tasks, where the data will be divided evenly for each process. The task is completed by sorting the data obtained, using the Team Sort algorithm sequentially in each process. The process of combining and unifying the results of each task on the data parts that have been sorted into one unit. This fusion is implemented using the merging technique in the Team Sort algorithm. In each stage of aggregation, two data sets will be combined into one at the same time, so that the number of data sets in the next stage will be reduced by half from the previous one in parallel.

In the illustration in Figure 3 on the left side, the software receives input in the form of a file name which contains a list of numbers that need to be sorted along with a number "n threads/processor cores" which indicates the number of threads/processor cores to be created. Then, the application will take all the numbers contained in the file and divide it into "n threads/processor cores" parts of almost the same size. The program will also create an array of bools with the same size as the number of "n threads/processor cores". The bool array will be used as an indicator that a thread or processor core has successfully completed its task. After that, the program generates a number of "n threads/processor cores" and assigns one piece of data to each thread. After all the processor core threads have finished their work, the program will return the results that have been arranged in the order.



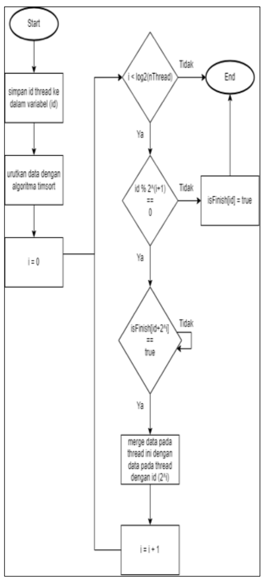

Start
simpan id thread ke dalam variabel (id)
urutkan data dengan algoritma timsort
i = 0
i < log2(nThread)
Tidak
End
Ya
id % 2^(i+1) == 0
Tidak
isFinish[id] = true
Ya
isFinish[id+2^i] == true
Tidak
Ya
merge data pada thread ini dengan data pada thread dengan id (2^i)
i = i + 1

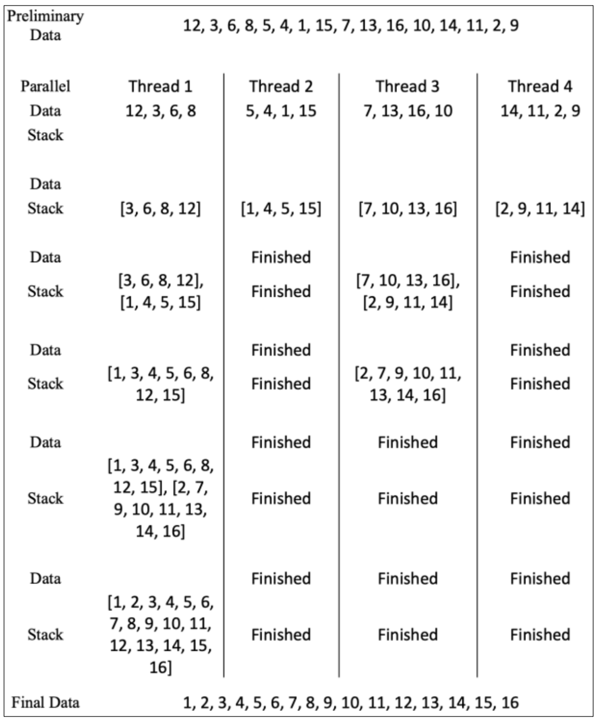
Figure 3. Main Program Flowchart (Left) and Parallel Process Program Flowchart (Right)

The diagram on the right in Figure 3 illustrates the workflow that occurs on each thread or processor core. First, the processor thread/core will request an identification. The data that has been obtained will be arranged sequentially using the Team Sort algorithm technique successively. Then, the first half thread or processor core will fetch information from the second half thread or processor core. Part of the second processor thread or core is considered complete. The process will continue until there is only one processor core/brain left. Processor threads/cores have the ability to find out the extent to which part of the many processor threads/cores can be identified based on their ID. There is an illustrative in Figure 4 which showing data of 16 numbers and 4 threads or processor cores.

D. Test Design

1) Optimal minimal run testing: tests conducted to find a method of determining the smallest number of runs that can produce optimal computation time. There are differences in the data used in the minimal optimal run test scheme compared to other test schemes. The steps carried out in the testing process are as follows*:*

- Determines how the minimum run value is determined. There are two types of methods used, namely static methods and dynamic methods. Regardless of the amount of data, static methods are used to determine the minimum run value.
- Implement the Team Sort method in an action algorithm
- Observing the length of time it takes to execute the algorithm
- Making comparisons regarding the length of execution time of the algorithms and looking for the most optimal one.



Preliminary Data	12, 3, 6, 8, 5, 4, 1, 15, 7, 13, 16, 10, 14, 11, 2, 9			
Parallel	Thread 1	Thread 2	Thread 3	Thread 4
Data Stack	12, 3, 6, 8	5, 4, 1, 15	7, 13, 16, 10	14, 11, 2, 9
Data Stack	[3, 6, 8, 12]	[1, 4, 5, 15]	[7, 10, 13, 16]	[2, 9, 11, 14]
Data		Finished		Finished
Stack	[3, 6, 8, 12], [1, 4, 5, 15]	Finished	[7, 10, 13, 16], [2, 9, 11, 14]	Finished
Data		Finished		Finished
Stack	[1, 3, 4, 5, 6, 8, 12, 15]	Finished	[2, 7, 9, 10, 11, 13, 14, 16]	Finished
Data		Finished	Finished	Finished
Stack	[1, 3, 4, 5, 6, 8, 12, 15], [2, 7, 9, 10, 11, 13, 14, 16]	Finished	Finished	Finished
Data		Finished	Finished	Finished
Stack	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]	Finished	Finished	Finished
Final Data	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16			

Figure 4. Data Illustration of 16 Numbers and 4 Processor Threads/Cores

2) Testing the accuracy of the algorithm: Testing is carried out to evaluate the success of the parallel algorithm that has been developed in sorting data accurately, without any errors or discrepancies in the amount of data. Following are the actions taken in the testing process:

- Generates a random sequence of data with values between 1 and n. Variable n is the amount of data generated.
- Executing the Team Sort parallel algorithm using the data that has been created
- Verify the result of algorithm n

3) Comparative testing between parallel algorithms and Sequential Algorithms: In this test, the main goal is to compare the performance of parallel algorithms and sequential algorithms. We will test how fast the parallel algorithm can increase the speed compared to the existing sequential algorithm. Following are the procedures carried out in the testing phase:

- Generates data in a random way.
- Execute every available algorithm using the generated data
- Observing the duration of the algorithm execution
- Comparing the execution time of each algorithm

4) Testing the effect of increasing core usage on speed up: This test was conducted to determine the impact of increasing processor core usage on algorithm acceleration. The testing process involves a series of steps as follo*ws:*

- Generate random information
- Execute algorithms that run simultaneously using the data that has been generated
- Measuring the time taken by the algorithm to execute
- Perform calculations to determine performance improvements and efficiency of an algorithm. The length of time for executing the algorithm sequentially was found through previous experiments
- Measuring the speed increase achieved by a more efficient parallel algorithm

III. Results And Discussion

A. Research Results

1) Implementation of Tim Sort's algorithm using OpenMP: Successfully coded to implement Team Sort's algorithm in parallel using OpenMP. Input checking involves verifying the input data (using multiple threads and file directories) and the process of reading the file. The user enters all input via arguments on the command line. The preparation process includes sharing data for each thread and initial setting of supporting variables. One of the supporting factors is the variable "isFinish" which is an array that functions as an indicator of the status of a thread. The paraphrase of the text is "In a parallel process, previously prepared data begins to be arranged in sequence." This process includes organizing and combining information.

2) Implementation of Tim Sort algorithm using MPI: Implementation of parallel code of Tim Sort algorithm with MPI has been successfully completed. The process of capturing input data involves verifying the input data (including the number of processes and file addresses) and entering files. The process is only executed by process number 0. The number of processes can be obtained by using the MPI_Comm_Size function. However, the file location is obtained via command line arguments. Process preparation involves dividing the data for each process and initializing the supporting variables. In order to share data, the process is carried out using two functions, namely MPI_Bcast and MPI_Scatterv.

MPI_Bcast serves to spread a number of data to be sorted to all processes in the MPI communicator. Later, this information will be used to start the initial value of the supporting variables. Meanwhile, the MPI_Scatterv function will be used to spread the data to be sorted. The layout process involves organizing and aggregating data. The merging process involves two steps, but only one of those steps will merge the process. The process involved will send information about the amount of data and the amount of data it has to the process that is doing the merging. This can be done through the use of the MPI_Send function. To combine data, the information will be received by the process through the MPI_Recv function.

B. Analysis and Discussion

1) Analysis of the results of the optimum minimum run test data: In Table 1 there is an average testing computation time on the results of the optimum minimum run test data. In Table 1 it can be seen that the computation time will decrease when the minimum run value decreases from 2 to 16. By using a small minimum run value, the process of forming one run can be done more quickly. However, this phenomenon also causes the number of runs that are formed to increase when compared to the minimum value which is larger and the time required to combine them becomes longer.

TABLE I
Average Computation Time Of Minimum Optimum Test Run

No.	Minimum Run Value (Data)	Average Compute Time (Milliseconds)
1	2	81,086
2	4	76,647
3	8	67,875
4	16	64,469
5	32	67,675
6	64	80,286
7	128	111,468
8	Dynamic	75,899

The solution to this problem is to increase the minimum number of runs. By increasing the minimum run value limit, the time needed to make one run will be longer, but the number of runs formed will be reduced. Thus, the time needed to combine the runs will be shorter. When compared, the increase in computation time in forming the entire run is less than the reduction in computation time in the concatenation, so that the total computation time becomes shorter.

Continuously increasing the minimum run value limit does not necessarily result in a reduction in the total computation time. From a minimum run value of 16 to a minimum run value of 128, it can be observed in Table 1. During this interval, increasing the minimum run value will cause the overall computation time to become longer. This is due to the fact that the time taken to calculate all the individual runs is longer than the time taken to combine them. This is what causes dynamic run to have a longer average computation time. At a minimum run of 16, the difference between the reduction of the combined computation time and the computation time of the formation of all runs is the largest compared to the other minimum run values. Therefore, the best computation time performance can be achieved with a minimum run value of 16.

2) Analysis of the results of testing the accuracy of the algorithm: Testing the accuracy of the algorithm is carried out to evaluate the extent to which modifications that have been made to the algorithm can produce correct results. After testing, the results obtained reached an accuracy value of 100%. The results of these tests conclude that the algorithm that has been made is successful in sorting the data correctly and no data is lost during the sorting process.

3) Analysis of the results of comparative testing between parallel algorithms and sequential algorithms: The test is carried out using 16 processor cores and a minimum number of operations of 16 (results of the most efficient test with a minimum number of operations). Table 2 shows the test results. In Table 2, it can be seen that the performance of the parallel Sort Team has increased significantly. This performance increase can be observed from the processing time which can be up to 8.4 times faster on the MPI implementation and 8.48 times faster on the MPI implementation when compared to the processing time of the Tim Sort algorithm sequentially on large amounts of data 4194304.

TABLE II
DATA RESULTS OF COMPARISON TESTING BETWEEN PARALLEL AND SEQUENTIAL ALGORITHMS

No.	Lots of Data	Compute Time (Milliseconds)		
		Sequential Algorithm	Parallel Algorithm	
			OpenMP	MPI
1	1024	0,273	0,737	0,157
2	2048	0,556	0,782	0,210
3	4096	1,226	1,037	0,322
4	8192	2,362	1,071	0,542
5	16384	4,864	1,418	0,969
6	32768	10,126	2,012	1,786
7	65536	20,988	3,648	3,652
8	131072	43,464	6,950	7,024
9	262144	89,743	12,224	14,110
10	524288	186,050	24,652	27,925
11	1048576	383,766	47,882	51,949
12	2097152	790,304	95,737	95,839
13	4194304	1642,169	193,330	195,899

From this information, it can be seen that the increase in the amount of data is in line with the increase in speed. When the amount of data to be sorted is small (less than or equal to 16384), the MPI implementation experiences a higher speed increase than the MPI implementation. This is because OpenMP involves creating threads.

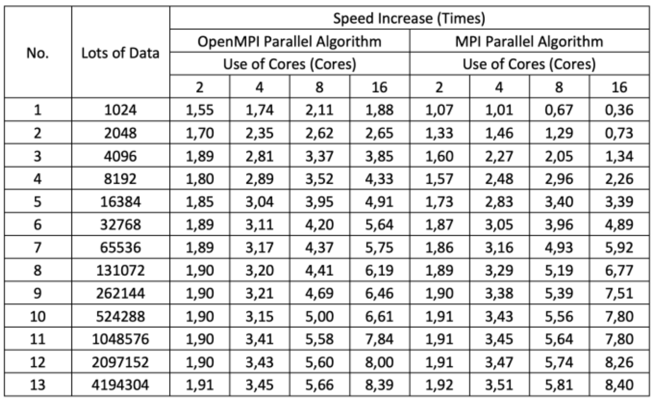
The creation of the thread does not take a long time, less than 1 millisecond, but even so, the short time has a significant impact when compared to the time needed for sequential computation. Similarly, a similar situation occurs for MPI when faced with large data sets of 1024 and 2048. The sequential processing time for that large amount of data is less than 1 millisecond, so if the processing times are increased from 0 to 1 millisecond, the processing time using MPI can exceed the sequential processing time.

Success parameters include average execution time, fastest time, and longest time for each implementation, as follows. With a lot of data ≥ 65536, the OpenMP computing time is smaller than the MPI computing time and MPI has a smaller computing time than OpenMP with a lot of data < 65536. The average execution time is 30.114 milliseconds for OpenMP and 30.8 for MPI, so that in total on average OpenMP is faster than MPI for all amounts of data and processor cores. The fastest time for 1024 data and 16 processor cores from OpenMP is 0.737 milliseconds and from MPI is 0.157 milliseconds. The longest time for the amount of data 4194304 and 16 processor cores from OpenMP is 193,330 milliseconds and from MPI is 195,899 milliseconds. This happens because information is passed between threads in OpenMP at a higher speed than information is passed between processes in MPI. Although the process of creating threads in OpenMP requires computation time, the total computation time for thread creation plus OpenMP communication time is smaller than MPI communication time.

Communication in OpenMP can run more efficiently because each thread in OpenMP uses the same memory to communicate. As a result, threads can share information by writing and reading messages directly from memory, without the need to go through a complicated communication process. Communication between processes in MPI is done by sending messages to each other. The larger the message sent, the longer it will take to send it.

4) Analysis of the results of testing the effect of increasing the use of processor cores on increasing speed (speed up): The process of analysing test results is carried out with the aim of understanding how the use of more processor cores can affect the increase in speed (speed up) resulting from the algorithm that has been developed. The trial was carried out using a minimum run of 16 times, which is the result of the optimal minimum run test. Table 3 shows the test results. The increase in speed in Table 3 is estimated based on the difference in execution time of the sequential algorithm in Table 2.

TABLE III
TESTING RESULTS DATA EFFECT OF USING MANY PROCESSORS CORE ON SPEED UP



No.	Lots of Data	Speed Increase (Times)							
		OpenMPI Parallel Algorithm				MPI Parallel Algorithm			
		Use of Cores (Cores)				Use of Cores (Cores)			
		2	4	8	16	2	4	8	16
1	1024	1,55	1,74	2,11	1,88	1,07	1,01	0,67	0,36
2	2048	1,70	2,35	2,62	2,65	1,33	1,46	1,29	0,73
3	4096	1,89	2,81	3,37	3,85	1,60	2,27	2,05	1,34
4	8192	1,80	2,89	3,52	4,33	1,57	2,48	2,96	2,26
5	16384	1,85	3,04	3,95	4,91	1,73	2,83	3,40	3,39
6	32768	1,89	3,11	4,20	5,64	1,87	3,05	3,96	4,89
7	65536	1,89	3,17	4,37	5,75	1,86	3,16	4,93	5,92
8	131072	1,90	3,20	4,41	6,19	1,89	3,29	5,19	6,77
9	262144	1,90	3,21	4,69	6,46	1,90	3,38	5,39	7,51
10	524288	1,90	3,15	5,00	6,61	1,91	3,43	5,56	7,80
11	1048576	1,90	3,41	5,58	7,84	1,91	3,45	5,64	7,80
12	2097152	1,90	3,43	5,60	8,00	1,91	3,47	5,74	8,26
13	4194304	1,91	3,45	5,66	8,39	1,92	3,51	5,81	8,40

Table 3 shows the scalability of OpenMP and MPI implementations with increasing number of processor cores and data used. The performance of both implementations improves as resources increase, but as data increases this will affect the scalability of the OpenMP and MPI implementations. In Table 3, it can be seen that the order of increasing the speed from highest to lowest when using MPI is when using 16 cores, 8 cores, 4 cores, and 2 cores processor. The speed ups were 8.4 times, 5.69 times, 3.46 times and 1.91 times, respectively. When using OpenMP, the speed increases from the highest to the lowest, including when using 16 cores, 8 cores, 4 cores and 2 cores. The addition of speed (speed up) each of 8.48 times, 5.74 times, 3.51 times, and 1.91 times. From this information, it can be concluded that the increase in data usage will be in line with the increase in speed obtained by using the same core processor.

In OpenMP, the increase in processor core usage is proportional to the speed up (speed up) of the same large amount of data. However, this effect is only seen in the amount of data that is greater than or equal to 32768. In many cases, the speed up on data smaller than 32768 will experience an increase which will then decrease or even continue to decrease. The decrease in computation time resulting from the increase in the use of processor cores, is offset by the increase in the time required for communication. As a result, the total time required for computation and increased speed is lower.

Some data shows that OpenMP has a speed increase of less than 1, which means that the sequential computation time is faster than OpenMP. This situation occurs when using an 8-core processor with a data set of 1024, and when using a 16-core processor with a data base of 1024 and 2048. This occurs because the time required to calculate the thread creation and communication processes is longer than the time saved by reducing computing.

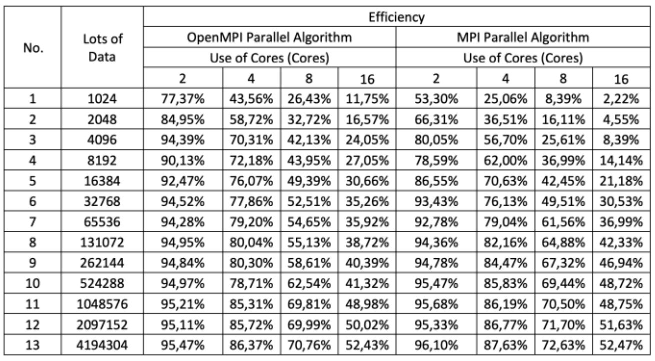
The more threads to be created, the longer the computation time required for the thread creation process. This phenomenon is clearly seen in Table 3, where the use of a processor with 8 cores results in a shorter computation time for the thread creation process compared to the reduction in computation time resulting when the data is multiplied up to 2048. When using a processor with 16 cores, the new situation is similar occurs when the amount of data is increased to 4096, or a 4-fold increase.

In Table 3, the more data ≥ 262144, the more similar the speed up of OpenMP and MPI will be. This phenomenon occurs because in that interval, the ratio between the computation time when creating threads plus the computation time when communicating and subtracting the computation time in OpenMP will be increasingly similar to the comparison between the computing time communicating and reducing the computation time in MPI. MPI speed is not always faster than OpenMP.

The efficiency of computing resource use is measured for each implementation. Success parameters include the extent to which the implementation utilizes the potential of available computing resources compared to optimal resource use. The efficiency figures in Table 4 are calculated based on the information contained in Table 3. In Table 4, it can be seen that the MPI implementation has varying levels of efficiency depending on the number of cores used. The order of MPI implementation efficiency from highest to lowest is using 2 cores, 4 cores, 8 cores, and 16 cores of processors. The efficiency percentages are 95.53%, 86.38%, 71.08% and 52.47% respectively. In the OpenMP implementation, processor efficiency levels based on the number of cores from highest to lowest are 2 cores, 4 cores, 8 cores, and 16 cores. The efficiency of each processor is 95.61%, 87.68%, 72.75% and 52.99%.

Based on Table 4, it can be seen that using 8 processor cores with 1024 data and using 16 processor cores with 1024 and 2048 data produces an efficiency below 10%. This happens because when the speed of using the processor core increases with the same amount of data, the speed (speed up) does not exceed 1 or the computation time becomes longer compared to the sequential computing time. The causes of this phenomenon can be found in the explanations given in Table 3. Apart from these three data, there is one additional data which shows an efficiency below 10%. This occurs when using as many as 16 processor cores with the amount of data reaching 4096.

TABLE IV
TESTING RESULTS DATA EFFECT OF USING MANY PROCESSORS CORE ON EFFICIENCY



No.	Lots of Data	Efficiency							
		OpenMPI Parallel Algorithm				MPI Parallel Algorithm			
		Use of Cores (Cores)				Use of Cores (Cores)			
		2	4	8	16	2	4	8	16
1	1024	77,37%	43,56%	26,43%	11,75%	53,30%	25,06%	8,39%	2,22%
2	2048	84,95%	58,72%	32,72%	16,57%	66,31%	36,51%	16,11%	4,55%
3	4096	94,39%	70,31%	42,13%	24,05%	80,05%	56,70%	25,61%	8,39%
4	8192	90,13%	72,18%	43,95%	27,05%	78,59%	62,00%	36,99%	14,14%
5	16384	92,47%	76,07%	49,39%	30,66%	86,55%	70,63%	42,45%	21,18%
6	32768	94,52%	77,86%	52,51%	35,26%	93,43%	76,13%	49,51%	30,53%
7	65536	94,28%	79,20%	54,65%	35,92%	92,78%	79,04%	61,56%	36,99%
8	131072	94,95%	80,04%	55,13%	38,72%	94,36%	82,16%	64,88%	42,33%
9	262144	94,84%	80,30%	58,61%	40,39%	94,78%	84,47%	67,32%	46,94%
10	524288	94,97%	78,71%	62,54%	41,32%	95,47%	85,83%	69,44%	48,72%
11	1048576	95,21%	85,31%	69,81%	48,98%	95,68%	86,19%	70,50%	48,75%
12	2097152	95,11%	85,72%	69,99%	50,02%	95,33%	86,77%	71,70%	51,63%
13	4194304	95,47%	86,37%	70,76%	52,43%	96,10%	87,63%	72,63%	52,47%

The increase in the speed of the data has a ratio of more than 1 or higher than the sequential computing time, but the increase in time obtained is less than 1.8. This phenomenon occurs because the difference in the time required to enumerate threads, the time to send and receive communications, and the reduction in computation time is still insufficient to achieve a speed increase of 1.8 or more. From the available information, it appears that the increase in the use of processor cores is inversely related to the level of efficiency achieved.

The more processor cores are used, the less efficiency that can be achieved. The reason is because there is inequality in the division of tasks in the parallelization carried out in our research. An imbalance in the division of tasks can be found in the merging process. In the time of merging into one, only half the number of processor cores are used. When combining the two, a fraction of the total processor cores is currently running. The activity will continue to run, using half the processor core capacity of the previous activity, until it reaches the log2(n) process.

Apart from that, the communication process between processor cores and data transfer also causes a lack of efficiency. This is in line with the concept of division, namely when divided by two, the reduction is halved, but when divided again by two, the reduction is not half, but smaller, and so on. Therefore, the reduction in time can still be smaller, but the amount of time reduced will decrease.

If the number of processor cores used is still small (≤ 4), the impact of the unequal distribution of tasks is not too visible. This can be seen from the test results in Table 3, where the highest efficiency reached 95.53% and reached 86.38%. As the number of processor cores used increases, an imbalance in the distribution of tasks becomes visible. Evidence of this can be seen from the test results listed in Table 4.5, where the highest efficiency reached 71.08 percent and the lowest efficiency reached 52.47 percent.

The OpenMP implementation and the MPI implementation are similar in terms of performance. The difference in the highest level of efficiency obtained for each processor core is only around 0% to 2% with the application of OpenMP technology which shows better results compared to MPI technology. Differences appear when the amount of data to be sorted is small (less than 16384), in this case the MPI implementation shows more optimal performance (with an increase in efficiency of 5% to 25%) compared to the OpenMP implementation. While there is a significant difference in efficiency scores, the difference in execution time between the two implementations is only slight (between 50 microseconds to 500 microseconds), with the MPI implementation outperforming the OpenMP implementation in terms of speed.

The working concept of OpenMP is like people having a discussion at a table, when one of them asks a question, the others can immediately hear and answer. MPI's working concept is like people having a discussion but not in the same room, when one of them wants to ask a question, it is done via sending messages, and the others cannot immediately hear and answer. The ability of MPI implementation to overcome disruptions or failures in communication is by freeing up all kinds of data transferred on the same communication line, thereby reducing tolerance for time delays. However, the advantage of MPI over OpenMP is that it is easier to implement for an increasing number of processor cores and resources.

IV. CONCLUSION

The increase in speed using OpenMP in parallelizing the Team Sort algorithm is better, reaching up to 8.48 times, when compared to using MPI which only achieved a speed increase of 8.4 times for testing using 16 processor cores and data is also used. The speed increase is proportionally related to the use of processor cores on the same large amount of data, whether implementing OpenMP or MPI. OpenMP has a faster computation than MPI when the data being processed is large enough, which is more than or equal to 65536. However, MPI has a smaller computation time than OpenMP when the data is less than 65536. The more data that is used on the use of processor cores the more the same, then the speed and efficiency will increase proportionally. The use of processor cores has a conflicting relationship with efficiency when using a lot of similar data.

REFERENCES

[1] D. Maitra, Beginner's Guide to Code Algorithms, Boca Raton: CRC Press, 2022.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Fourth Edition, London: MIT Press, 2022.

[3] S. S. Skiena, The Algorithm Design Manual, New York: Springer, 2020.

[4] D. Eddelbuettel, "Parallel Computing With R: A Brief Review," *WIREs Comput Stat,* vol. 13, no. 2, p. e1515, 2021.

[5] I. N. A. Yudiswara and A. Suganda.

[6] F. R. Lumbanraja, A. and N. R. Muttaqina, "Analisa Komputasi Paralel Mengurutan Data Dengan Metode Radix Dan Selection," *Jurnal Komputasi,* vol. 8, no. 2, pp. 77-93, 2020.

[7] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, Cambridge: Morgan Kaufmann, 2019.

[8] V. D. Thoke, "THEORY OF DISTRIBUTED COMPUTING AND PARALLEL PROCESSING WITH ITS APPLICATIONS, ADVANTAGES AND DISADVANTAGES," in *nternational Conference on Innovative Minds*, Vita, 2015.

[9] W. Tang, W. Feng, J. Deng, M. Jia and H. Zui, "Parallel Computing for Geocomputational Modeling," in *GeoComputational Analysis and Modeling of Regional Systems*, New York, Springer, Cham, 2018, pp. 37-54.

[10] P. Czarnul, Parallel Programming for Modern High Performance Computing Systems, Boca Raton: CRC Press, 2018.

[11] E. D. Nugroho, M. E. Wibowo and R. Pulungan, "Parallel Implementation of Genetic Algorithm for Searching Optimal Parameters of Artificial Neural Networks," in *International Conference on Science and Technology-Computer*, Yogyakarta, 2017.

[12] P. S. Pacheco and M. Malensek, An Introduction to Parallel Programming, Massachusetts: Morgan Kaufmann, 2022.

[13] R. Trobec, B. Slivnik, P. Bulić and B. Robič, Introduction to Parallel Computing From Algorithms to Programming on State-of-the-Art Platforms, New York: Springer, Champ, 2018.

[14] V. Kasilov, P. Drobintsev and N. Voinov, "High-performance genome sorting program," in *International Young Scientists Conference on Computational Science*, St. Petersburg, 2021.

[15] M. R. Hanafi, M. A. Faadhilah, D. Pradeka and M. T. Dwi Putra, "Comparison Analysis of Bubble Sort Algorithm with Tim Sort Algorithm Sorting Against the Amount of Data," *Journal of Computer Engineering, Electronics and Information Technology,* vol. 1, no. 1, pp. 9-13, 2022.

[16] N. Auger, V. Juge, C. Nicaud and C. Pivoteau, “On the Worst-Case Complexity of TimSort,” in *Leibniz International Proceedings in Informatics,* Wadern, 2018.

[17] P. Ganapathi and R. Chowdhury, “Parallel Divide-and-Conquer Algorithms for Bubble Sort, Selection Sort and Insertion Sort,” *The British Computer Society,* vol. 00, no. 0, pp. 1-11, 2021.