Journal of Theoretical and Applied Information Technology
31st October 2021. Vol.99. No 20

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

# DESKTOP APPLICATION DEVELOPMENT FOR IMPLEMENTING MERGE SORT ALGORITHM ON DISTRIBUTED SYSTEM TO SORTING NUMBERS

**[1]HANDRIZAL, [2]SRI MELVANI HARDI, [3]WINTO JUNIOR KHOSASIH**

[1,2,3]Department of Computer Science, Faculty of Computer Science and Information Technology,

Universitas Sumatra Utara, Jl. University No. 9-A, Medan 20155, Indonesia

Email: handrizal@usu.ac.id

**ABSTRACT**

A long way back to 2008, the google infrastructure team reported that the google data center handles around 20 Petabytes per day with an average of 100.000 MapReduce jobs spread across its massive computing cluster with standard machine cluster node setup costs approximately $2400 each. According to how google handles things we can conclude even high-performance computers could have difficulties in processing large data, and even not all agencies could afford this kind of setup. Sorting is any process of arranging an item according to a rule with two common, yet ascending (ordering from smallest to largest item) or descending (ordering from largest to smallest item). Sorting is very important for indexing an item to ease searching the item by the system, human, or algorithm. There are many sorting algorithms with time complexity *O(N log N)* for sorting n items one of them is merge sort. Based on google's case, computing performance could be increased by dividing tasks to another node in the cluster computing environment, with this similar approach we could increase the performance of the merge sort algorithm. Merge sort algorithm based on divide and conquer technique that eases converting sequential process to parallel process with dividing it to another computer processor in divide phase. Using this similar dividing approach the system will divide the process into several hosts separated physically in cluster computing by utilizing java RMI as middleware. A research test was conducted using 4 computers hosted for distributed merge sort computing as well as a comparative computer for parallel merge sort computing. The system was tested on two and four-processor usage of parallel merge sort and four hosts to three hosts configuration with two processors and four processors for distributed merge sort, respectively. Performance is measured by comparing computation time of distributed merge sort and parallel merge sort.

**Keywords:** *Distributed MergeSort, High-Performance Computing MergeSort, Distributed System, MergeSort Algorithm, Parallel MergeSort, Java RMI, number sorting, Distributed MergeSort VS Parallel MergeSort, Cluster computing, Computer Network*

## 1. INTRODUCTION

Sorting is one of the main core computational algorithms used in many scientific and engineering applications [8]. There are many algorithms with time complexity O (N log N) for sort N numbers, one of them is merge sort. Merge sort is a sorting algorithm based on the divide and conquer paradigm to solve the problem by dividing tasks into sub-problems recursively as every subproblem is small enough to be solved immediately, every solution from each subproblem will be merged again to obtain a solution from the main problem.

The simplest form of merge sort is sequential merge sort that dividing input array elements into half recursively until one element is left in the array, then conquers the divided elements of the array based on order rule and merging the conquered array. All of this operation executes step by step from half part to other remaining half parts in a single execution thread process line at one time.

As for big data, computation takes place at the application back-end commonly server computers used RISC architecture, such as a mainframe, PC (Personal Computer), supercomputer, etc. Then

high-performance computing must take place in the non - mobile device because the amount of energy used for the computing is high, in the case to provide faster data services in the mobile device without using the limited amount of energy for computing large data set. Desktop PC as a type of the Personal Computer variant with general-purpose computing capability and wide high-performance setup solution to choose for the cost to performance widely used as a server rather than a mainframe.

Now a day, computer technology is constantly developed and more architecture is being introduced to the market. The design of machines with multiple core capabilities involves programming for many logical processors working independently [13]. Several parallel sorting algorithms such as bitonic sort, sample sort, column sort, and partitioned radix sort have been devised to shorten execution time. Parallel sorts usually need a fixed number of data exchange and merging operations. The computation time decreases as the number of processors grows. Since the time is dependent on the number of data each processor has [8].

To take advantage and adapting to parallel computing, a parallel sorting merge sort algorithm was introduced for utilizing all processors to participate in the merging phase that takes place during iteration. Parallel merge sort concept based on sequential merge sort form with the only difference is dividing phase not only divide problem but divide task to be processed by several other execution threads so that multiple execution paths can be executed simultaneously at one time and synchronized the thread during merging phase to prevent thread interference error caused by array access error that does not correspond to the iteration.

Parallel merge sort has limited hardware resources on a device while processing a large data set. To overcome this hardware limitation in computing merge sort algorithm operation, the merge sort algorithm operation was applied on a distributed system. Where in the system the distributed array object was distributed to all existing hosts to be executed in parallel among each hosted resource then synchronization was done when merging back all data from the existing host, this form so-called distributed merge sort.

One of the widely used distributed computing software concepts is MOS (Middleware Operating System) which provides several services for local applications and several independent services for remote applications. The available services in MOS (Middleware Operating System) are finding objects and interface location names, controlling information protocol, synchronization, concurrency, the security of objects, etc. Using the MOS paradigm parallel merge sort algorithm can be implemented into distributed merge sort.

Being able to communicate in distributed computing in addition to requiring a MOS paradigm also required the existence of communication architecture, one of the architectures that are often used to provide APIs in the development of distributed computing applications is CORBA (Common Object Request Broker Architecture). CORBA is a middleware based on an object distributed system that provides interoperability capability for two heterogeneous objects, CORBA usually is used in C, C++, COBOL, and JAVA.

Research by Dr. Mamta C. Padole in 2007 shows that a heterogeneous distributed system is considered as a better alternative to HPC for processing big data without the need for high-performance computing machines integration.

Based on the description above, the problem to be solved is how to develop a desktop application that implements integration in a distributed system to analyze performance gains in sorting large amounts of data.

This research contributes to network-based software developers who need new methods to improve application performance by increasing the computational speed of the algorithms used, especially merge sort through the use of computer networks so that all available computer resources can be used efficiently without having to upgrade hardware specifications. computer to be able to keep up with the services that the application executes against many requests.

The limitations of the problem in this study are as follows:

1. The algorithm used is the Merge Sort Algorithm.
2. The middleware used is JAVA RMI.
3. The system runs closed on a high-speed Local
4. Area Network without any data traffic to the internet.
5. The system is running on a powered host and ready to accept assignments.
6. Messages sent by master and slave have a maximum chunk size of 1 KB.
7. The number of threads of execution per host is a maximum of 4 threads per host. A minimum number of hosts is 3 end systems.
8. The IP address of each host is known before the system starts.

## 2. MERGE SORT ALGORITHM

Merge Sort is one sorting algorithm in computer science designed to meet the sequencing needs of a data set that is not possible to be accommodated in a computer's memory due to the large number size of the data. The algorithm was invented by John Von Neuman in 1945.

Merge sort algorithm data was sorted using divide and conquer that is by breaking then completing each section and merging it again. First data are broken down into 2 parts where the first part is half (if even array size) or half minus one (if odd array size) of all data, then continue resolving each section until it consists of only one data per section.

After breaking down (dividing) the section, the data is re-combined by comparing it on the same section whether the first data is greater than the data to the middle + 1 if true then the data to the middle + 1 is moved as first data, then the data in the first middle is shifted into data to two to the middle + 1, so on until it becomes one whole block as before. Because of this process, the merge sort algorithm requires a recursion function method for its completion. Pseudocode merge sort algorithm is as follows :

**Algorithm 1:** Merge Sort

```
1.  procedure mergesort( var an as array )

2.  if ( n == 1 ) return a
3.  var l1 as array = a[0] ... a[n/2]
4.  var l2 as array = a[n/2+1] ... a[n]
5.  l1 = mergesort( l1 )
6.  l2 = mergesort( l2 )
7.  return merge( l1, l2 )

8.  end procedure

9.  procedure merge( var as an array, var b as an
    array )

10. var c as an array
11. while ( a and b have elements )
12. if ( a[0] > b[0] )
13. add b[0] to the end of c
14. remove b[0] from b
15. else
16. add a[0] to the end of c
17. remove a[0] from a
18. end if
19. end while
20. while ( a has elements )
21. add a[0] to the end of c
22. remove a[0] from a
23. end while
24. while ( b has elements )
25. add b[0] to the end of c
26. remove b[0] from b
27. end while
28. return c

29. end procedure
```

## 3. PARALLEL MERGE SORT

Parallel merge sort consists of two phases: local sort and merges [8]. Once keys in each processor are sorted locally, processors merge them in log $P$ steps as explained below if $P$ processors are used [8]. In the first step, processors are paired as (sender, receiver) [8]. Each sender sends its list of $N/P$ keys to its partner (receiver), then the two lists are merged by each receiver to form a sorted list of $2^1$ $N/P$ keys [8]. Half of the processors work during the merge and the other half sit idling [8]. In the next step, only the senders in the previous step are paired as (sender, receiver), and the same communication and merge operations are

performed by each pair to form a list of $2^2$ $N/P$ keys [8]. The process continues until a complete sort list of $N$ keys is obtained. The algorithm is given as follows :

**Algorithm 2:** Parallel Merge Sort

$P$: the total number of processors
(assume $P = 2^k$ for simplicity)
$P_i$: a processor with index $i$
$h$: the number of active processors
$N$: list of keys

1. Begin
2. $h := P$
3. for all i:= 0 to $P - 1$
4. $P_i$ sorts a list of $N/P$ keys locally.
5. for j:= 0 to (log $P$)-1 do
6. for all i:=0 to $h$-1
7. if(i < $h$/2) then
8. $P_i$ receives $N/h$ keys from $P_{i+h/2}$
9. $P_i$ merge two lists of $N/h$ keys into a sorted list of $2N/h$
10. Else
11. $P_i$ sends its list to $P_{i-}h$ /2
12. $h := h/2$
13. end

## 4. DISTRIBUTED MERGE SORT AND JAVA RMI

A distributed system is a collection of independent computers that appears to its users as a single coherent system [3]. This definition has several important aspects. The first one is that a distributed system consists of components (i.e., computers) that are autonomous [3]. A second aspect is that users (be they people or programs) think they are dealing with a single system [3]. This means that one way or the other autonomous components need to collaborate [3]. How to establish this collaboration lies at the heart of developing distributed systems [3].

In principle, a distributed system should also be relatively easy to expand or scale [3]. This characteristic is a direct consequence of having independent computers, but at the same time, hiding how these computers take part in the system as a whole [3]. A distributed system will normally be continuously available, although perhaps some parts may be temporarily out of order [3]. Users and applications should not notice that parts are being replaced or fixed, or that new parts are added to serve more users or applications [3].

To support heterogeneous computers and networks while offering a single system view, distributed systems are often organized using a layer of software-that is, logically between a higher-level layer consisting of users and applications, and a layer underneath consisting of operating systems and basic communication facilities. Such a distributed system is sometimes called **middleware** [3].

An important class of distributed systems is one for high-performance computing tasks [3]. Roughly speaking, one can make a distinction between the two subgroups. In cluster computing, the underlying hardware consists of a collection of similar workstations or PCs, closely connected using a high-speed local-area network and each node runs the same operating system [3]. The situation has become quite different in the case of grid computing [3]. This subgroup consists of distributed systems that are often constructed as a federation of computer systems, where each system may fall under a different administrative domain and may be very different when it comes to hardware, software, and deployed network technology [3].

As communication must support heterogeneous computers and networks in offering a single system view, then interprocess communication is the heart of all distributed systems [3]. Communication in distributed systems is always based on low-level message passing as offered by the underlying network [3]. Expressing communication through message passing is harder than using primitives based on shared memory, available for non-distributed platforms. The modern distributed system often consists of thousands or even millions of processes scattered across a network with unreliable communication such as the internet [3]. Unless the primitive communication facilities of computer networks are replaced by something else, the development of large-scale distributed applications is extremely difficult. There are three widely used models for communication: Remote Procedure Call (RPC), Message Oriented Middleware (MOM), and data streaming [3].

An RPC allowing programs to call procedures located on other machines. When a process on machine *A* calls a procedure on machine *B*, the calling process on *A* is suspended and execution of the called procedure takes place on *B*. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message-passing at all is visible to the programmer [3].

RPC (Remote Procedure Class) and ROI (Remote Object Invocation) contribute to hiding communication in distributed systems, that is, they enhance access transparency. Unfortunately, neither mechanism is always appropriate. In particular, when it cannot be assumed that the receiving side is executing at the time a request is issued, alternative communication services are needed. Likewise, the inherent synchronous nature of RPCs, by which a client is blocked until its request has been processed, sometimes needs to be replaced by Message-Oriented Communication [3]. A class of Message Oriented Communication services generally known as message queueing system or just MOM (Message Oriented Middleware. MOM provides extensive support for persistent asynchronous communication. The essence of these systems is that they offer intermediate-term storage capacity for messages, without requiring either the sender or receiver to be active during message transmission [3].

Communication as discussed so far has concentrated on exchanging more or less independent and complete units of information [3]. Examples include a request for invoking a procedure, the reply to such a request, and messages exchanged between applications as in message queuing systems [3]. The characteristic feature of communication sometimes does not matter at what particular point in time communication takes place [3]. Although a system may perform too slow or too fast, timing does not affect correctness. The question that addresses is which facilities a distributed system should offer to exchange time-dependent information such as audio and video streams [3]. Various network protocols that deal with this are called stream-oriented communication.

### 4.1. Java RMI

Java RMI is a Java API that performs the object-oriented equivalent of Remote Procedure Calls (RPC), with support for the direct transfer of serialized Java classes and distributed garbage collections. The RMI provides a mechanism to create distributed applications in java as explained in the three communication models above. The RMI allows an object residing in one system to invoke methods on an object running in another JVM [1]. RMI is used to build distributed applications such an application is sometimes referred to as a distributed object application.

RMI provides remote communications between java programs with a typical server program that creates some remote objects, makes references to these objects accessible, and waits for clients to invoke a method on these objects [1]. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them [1].

The distributed object applications in java RMI need to do 3 things: locate remote objects, communicate with the remote object and load class definitions for objects that are passed around. The following illustration in figure 1 depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. [1] The server calls the registry to associate (or bind) a name with a remote object [1]. The client looks up the remote object by its name in the server's registry and then invokes a method on it [1]. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed [1].

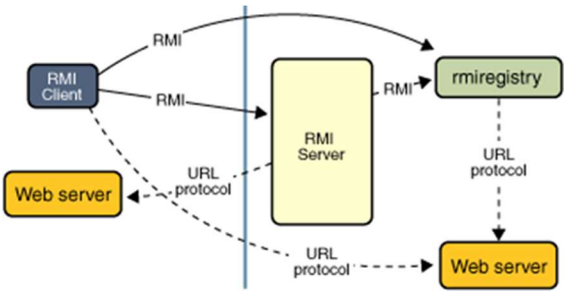


*Figure 1: RMI distributed application*

Java RMI has been designed following a layered architecture approach. Figure 2 presents, from bottom to top, the transport layer, responsible

for managing all communications, the remote reference layer, responsible for handling all references to objects, the stub/skeleton layer, in charge of the invocation and execution, respectively, of the methods exported by the objects; and the client and server layer, also known as service layer [12]. The activation, registry, and distributed collection (DGC) services are also part of this service layer [12].

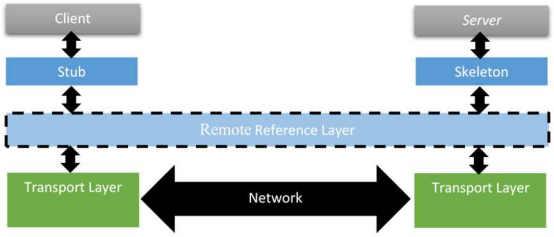


*Figure 2: Java RMI layered architecture*

RMIregistry is a namespace on which all server objects are placed [7]. Each time the server creates an object, it registers this object with the RMIregistry (using **bind()** or **reBind()** methods) [7]. These are registered using a unique name known as **bind name** [7].

To invoke a remote object, the client needs a reference of that object [7]. At that time, the client fetches the object from the registry using its bind name (using the **lookup()** method) [7]. The following illustration explains the entire process :

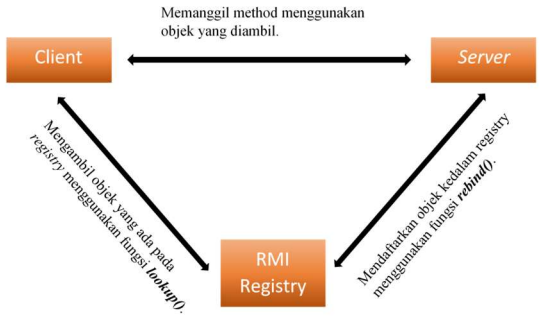


*Figure 3: Java RMI Process Summary*

Java RMI as discussed so far has offered high-speed network support, a high-performance library for cluster computing, remote communication abstraction to the programmer, security of transmitted objects. The question addressed is why use java RMI in the development of this desktop application for implementing merge sort on a distributed system. Java technology runs consistently on any operating system supported by the java platform. The main problem faced in deployments of distributed computing is that each node must have installed JRE (Java Runtime Environment) in their system. This makes scalability of distributed computing easier, different variety systems can participate dynamically in the cluster without every node has to be configured individually.

### 4.2. Distributed Merge Sort

Distributed merge sort has a similar approach with parallel merge sort counterpart, the main and only differences are local sort and merge process in distributed merge sort run in different host except for the final sort and merge process is done by the local host (client). Once all host $H$ is located, the $N$ list will be divided by $H$ host. In the first step, the hosts are paired to their list to sort tasks according to division results. Then each task to send the sorting task for $H$ host is assigned in the queue waiting to be processed by $cP$ processors on sender host (client), then each sender sends its list of $N/H$ keys to the listed receiver host (server) simultaneously, after that, the receiver host (server) executes parallel merge sort locally in their machines then send back their result of the sorted list to sender host (client).

The sender host (client) sits idling, waiting for the completing task returned from the listed receiver host (server). While completing the sorted list task returned from a receiver host (server), the sender host (client) will begin to execute the local sort and merge it with the other completed list from another receiver host (server), this process continues until a complete sort list of $N$ keys are obtained. The algorithm of distributed merge sort is summarized as follows :

**Algorithm 3:** Distributed Merge Sort

$P$: the total number of receiver host (server) processors (assume $P = 2^k$ for simplicity)
$P_i$: server processor with index $i$
$cP$: the total number of sender host (client) processors (assume $cP = 2^k$ for simplicity)
$cP_i$: client host processor with index $i$

*h*: the number of active processors
*H*: the number of available receiver hosts (server)
$H_i$: receiver host (server) with index *i*
*L*: list of receiver returned sorted keys

1. procedure parallel mergesort(var *N* as an array): *N* as an array
2. *h* := *P*
3. for all i:= 0 to *P* – 1
4. $P_i$ sorts a list of *N/P* keys locally.
5. for j:= 0 to (log *P*)-1 do
6. for all i:=0 to *h*-1
7. if(i < *h*/2) then
8. $P_i$ receives *N/h* keys from $P_{i+h/2}$
9. $P_i$ merge two lists of *N/h* keys into a sorted list of 2*N/h*
10. Else
11. $P_i$ sends its list to $P_{i-}$ *h* /2
12. *h* := *h*/2
13. end procedure

14. procedure distributedMergeSort(var *N* as array)
15. *cP* := *H*
16. for i:= 0 to *H-1*
17. $cP_i$ send a list of *N/H* keys to $H_i$ => list add ($H_i$ invoke parallelMergeSort(*N/H*) local in $H_i$'s machine)
18. end for
19. i :=0
20. while( i < *H-1)*
21. if(receives *N/H* keys from $H_i$)
22. if( sizeOf(*L*) > 2)
23. merge two list of *L* into single sorted list of *L*
24. end if
25. i++
26. end if
27. end while
28. end for
29. end procedure

## 5. UNDERSTANDING THE WORKING ENVIRONMENT

To perform the experimentation, different hardware and software configurations are chosen. Here, we will understand the need for different configurations with their specification. Each node in the system is preloaded by some computation and the processing job with its details is discussed in the following subsection.

### 5.1. Heterogeneous Cluster Node Configuration

In this system, two setups are used for both the setup we are varying the number of processors used in the node with 2 and 4 CPU usage. To understand the effect of computation-distribution we are varying the number of nodes in the cluster from 3 to 4 in a distributed merge sort setup, the purpose is to understand the effect of computation-distribution techniques like memory, the number of the CPU core usage, and the number of cluster nodes for comparison with parallel merge sort computation in similar CPU core usage configuration. This is to see the effect of processing, overall performance, and adaptiveness of the algorithm in a cluster, in different configuration parameters.

For the projected system, 4 different Personal Computers models are used. The hardware configurations are as follows.

*Table 1: Hardware configurations of cluster nodes*

| PC Index | Processor | Memory | Operating System |
|---|---|---|---|
| PC_1 | AMD E-350 @ 1,6GHz (2 CPUs) | 8GB DDR3L | Windows 7 ultimate 32-bit (build 6.1.7601) |
| PC_2 | Intel(R) Core(TM) i7-8550U CPU @ 1,8GHz (4 CPUs) | 8GB DDR4 | Windows 10 Home Single Language 64-bit (build 10.0.1863.1198) |
| PC_3 | Intel(R) Core(TM) i5-8400 CPU @ 2,8GHz (6 CPUs) | 16GB DDR4 | Windows 10 Pro 64-bit (build 10.0.19042.572) |

| | | | |
|---|---|---|---|
| PC_4 | Intel(R) Core(TM) i5-2450M CPU @ 2,5Ghz (2 CPU) | 4GB DDR3 | Windows 7 Professional 32-bit (build 6.1.7600) |

The PC_3 node is the most higher hardware performance, following by PC_2 as the second-highest and PC_4 as the third-highest. Meanwhile, PC_1 is the lowest hardware performance in terms of processing power. The PC_3 node is higher in terms of processing, communication, and memory. Purposely the higher node is introduced in the system so that the system behavior can be studied over the other parameters and a good algorithm can be designed accordingly. In all varying nodes, the java RMI client node is always PC_3. Other computing parameters on which the system performance relies are bus speed, internal clock rate, instruction sets architecture, cache memory, and processor. The details of the distributed merge sort configuration are given below in table 2.

*Table 2: Distributed merge sort configuration*

| Experimental Configuration | Participating Nodes |
|---|---|
| 3 Nodes | PC_2, PC_3, PC_4 |
| 4 Nodes | PC_1, PC_2, PC_3, PC_4 |

Meanwhile, for a parallel merge sort setup, each node will execute the process locally using a different number of processor core usage. Thus, our system comprises three setups that generating elapse time of execution data for each chosen setup.

### 5.2. System Design and Processing Job Description

Merge sort as sorting algorithm need access array and comparing array, this process has interval time from starting point to the finishing point. This time interval is in milliseconds and by comparing it with two different computation setups (i.e. parallel and distributed) we can find whether high-performance computing between two methods has advantages on the heterogeneous cluster.

The main problem we faced when calculating elapsed execution time is a consistent result of elapsed time value, then developed application must have the process's looping feature for average measurement of execution time. The result of calculating average time will be used as an observation variable in comparing the performance of both computation types.

Java RMI has tight–coupling middleware, it needs client and server applications to know each other. So a node must determine to act as either a server or a client. The server node is a node that performs given tasks and a client node is a node in charge of dividing tasks to the server nodes, client node is the actual main application while the server node is only modules/parts of the main application.

The client application contains five phases: choosing algorithms, initializing the algorithm configuration, choosing logging type, starting the process, process result. The initializing algorithm phase depends on the type of algorithm selected at the beginning of the choosing algorithm phase. The basic initialization parameter supported by all chosen algorithms is the number of process loop/stress tests and a random number input file to sort. The initialization parameter number of processor cores used is a special configuration for parallel merge sort. Initialization parameter for distributed merge sort consists of the number of available server hosts, the number of processor cores used in the client, the number of processor cores used for each server host, and the amount of data sent to the host server.

As for choosing the logging type phase in the client application is configured to set the message format displayed in real-time windows at the starting process phase. Lastly, the process result phase is a process that tasks to create and display the result of sorted numbers and sorting process results in execution time data.

The data used in the experiment are a random number with values ranging between 0 and 10.000. The amount of random data used in the experiment process ranges from ten million to one hundred million. Similar random data is used as input data

for all existing node sorting processes to preserve consistent results. These data will be sorted repeatedly according to the number of process loop/stress test parameters during the initialization phase so that the average value of execution time data can be calculated.

## 6. SORTING USING PARALLEL MERGE SORT

Parallel merge sort experimentation was performed on four nodes with different hardware specifications. Each experiment is carried out with two and four-processor core configurations. The test is conducted on ten files measuring from ten million to one hundred million random numbers. The execution time result of four nodes with each number of processor's core usage configuration is dual-core (2 processor cores) and quad-core (4 processor cores) are presents using graphs will explain in the following.

### 6.1. Parallel Merge Sort Execution Time Result on PC_1

As discussed in previous sections, parallel merge sort on PC_1 will be performed on two numbers of processor's core usage configuration that is 2 cores and 4 cores, each configuration will be looped ten times for each file contains ten million to eighty million random numbers. The average execution time results are shown in figure 4.

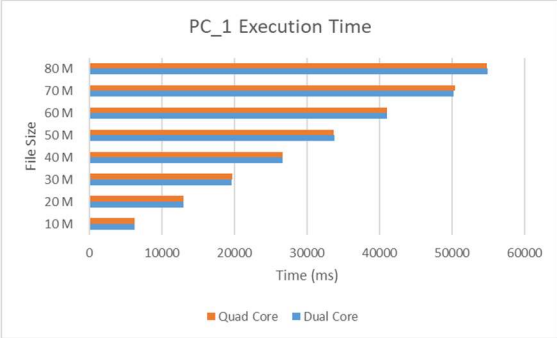


*Figure 4: PC_1 Execution Time*

For the system setup mentioned and is used to get figure 4 results, the performance observed in eight file sizes is not showing improvement. This is because the number of node's physical processor cores is only two and processors not supporting hyperthreading feature, increasing the number of processor core usage in system configuration will be useless. After all, the configuration dropped to the maximum physical cores (that is two) automatically by PC_1's operating system. As noticed in figure 4, the file size only counts to eight of ten, this is because the PC_1 system does not have enough memory to store all the contents of the file from the hard disk drive.

### 6.2. Parallel Merge Sort Execution Time Result on PC_2

Similar to the previous section, parallel merge sort on PC_2 also performed on two numbers of processor's core usage configuration that is 2 cores and 4 cores, each configuration will be looped ten times for each file contains ten million to one hundred million random numbers. The average execution time results are shown in figure 5.

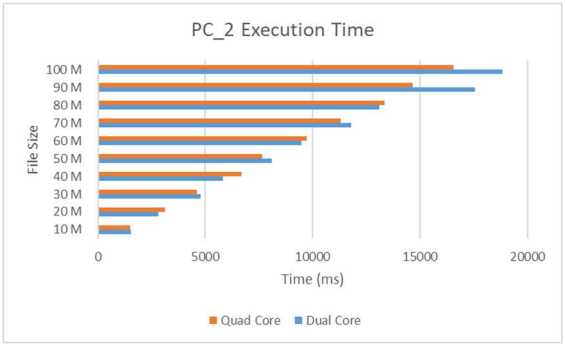


*Figure 5: PC_2 Execution Time*

Based on figure 5 results, the performance observed in ten file sizes showing some improvement in the quad-core configuration for PC_2 nodes. This is because the number of node's physical processor's core reaching four, here significant improvement is shown while processing ninety million and one hundred million file size, but having some performance degradation between ten million to forty million file sizes. This result shows that PC_2 having CPU throttling issues in the experiment. Hence, further in this paper, the result would be taken as a reference for comparison to other nodes and computation setup.

### 6.3. Parallel Merge Sort Execution Time Result on PC_3

Parallel merge sort on PC_3 is similarly performed on two numbers of processor's core usage configuration that is 2 cores and 4 cores, each configuration will be looped ten times for each file contains ten million to one hundred million random numbers. The average execution time results are shown in figure 6.

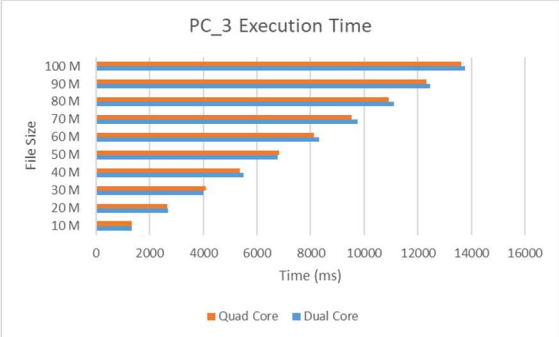


*Figure 6: PC_3 Execution Time*

Figure 6 shows that PC_3 has the best execution time compared to other nodes, the dual-core and quad-core performance of PC_3 shows a relatively consistent improvement compared to PC_2 performance for the same configuration even though it has more physical processor cores compared to other nodes.

### 6.4. Parallel Merge Sort Execution Time Result on PC_4

Similar to PC_1 discussed in the previous section, parallel merge sort on PC_4 also performed on two numbers of processor's core usage configuration that is 2 cores and 4 cores, each configuration will be looped ten times for each file contains ten million to eighty million random numbers. The average execution time results are shown in figure 7.

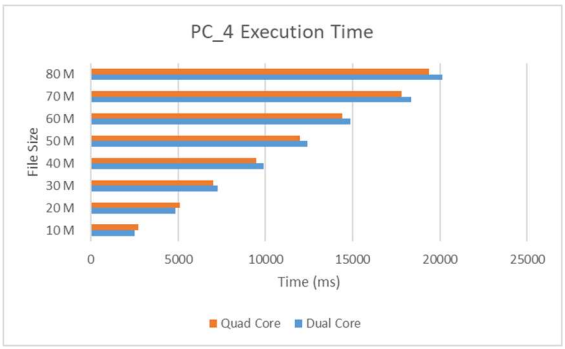


*Figure 7: PC_4 Execution Time*

For the system set up in PC_4 having similar hardware specification as PC_1, but figure 7 showing a slight improvement in PC_4 execution time while increasing numbers of the processor's core used configuration, this performance improvement remains consistent as the file size increase. This is because of the hyperthreading feature of the processor, even the number of the physical processor is only two, hyperthreading enables the processor to act as two logical processors in a physical processor. This hyperthreading feature is hidden from the application and shown as four physical processor cores on the application side.

## 7. SORTING USING DISTRIBUTED MERGE SORT

Distributed merge sort was performed on four setups: distributed merge sort 3 nodes dual-core configuration, distributed merge sort 3 nodes quad-core configuration, distributed merge sort 4 nodes dual-core configuration, and distributed merge sort quad-core configuration. Each experiment is carried out on ten files measuring from ten million to one hundred million random numbers. The following section will present the execution time result in graphs for each setup.

### 7.1. Distributed Merge Sort Execution Time Result on 3 nodes

The following graph showing the average execution time result for each file contains ten million to one hundred million random numbers in 3 nodes configuration with two processor's cores and four processor's cores configuration.

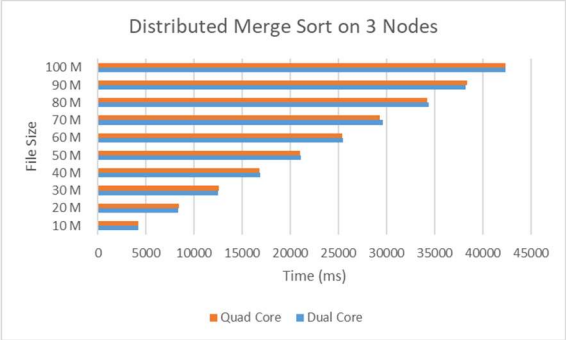


*Figure 8: Distributed Merge Sort 3 Nodes Execution Time*

Execution time results in figure 8 graphs showing a huge performance hit, there was unbalance node bottlenecking in a cluster environment that slowing performance further after increasing the number of processor's core usage.

### 7.2. Distributed Merge Sort Execution Time Result on 4 nodes

The following graph showing the average execution time result of each file contains ten million to one hundred million random numbers in 4 nodes configuration with two processor's cores and four processor's cores configuration.

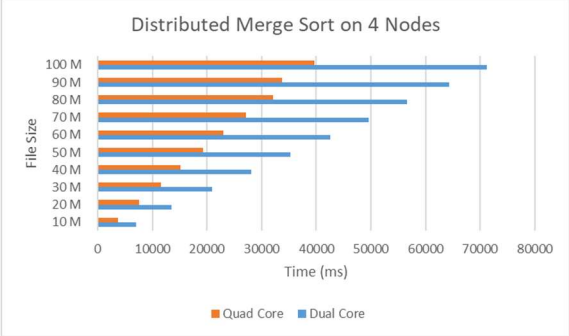


*Figure 9: Distributed Merge Sort 4 Node Execution Time*

Figure 9 graphs showing dramatic performance enhancement for the dual-core to the quad-core configuration in a four-node cluster environment. The result of this experiment based on figure 9 graph shows an increasing number of participant nodes and the number of processor's cores configuration is very influential at the level of performance in merge sort algorithm execution implementation of the distributed system cluster environment.

## 8. DISTRIBUTED MERGE AND PARALLEL MERGE SORT COMPARISON BASED ON CPU USAGE

This section presents graphs for comparing parallel merge sort performance with distributed merge sort performance. The previous discussion, we find there was four parallel merge sort setup with dual cores (two core configuration) and quad cores (four-core configuration) configuration and we find there was two distributed merge sort setup also with dual cores and quad cores configuration, this two setup have a similar characteristic in several processor's core usage, then the following subsection explaining distributed merge sort and parallel merge sort comparison based on two groups.

### 8.1. Dual Cores Comparison

The following graph showing a line diagram to visualize the difference performance between four parallel merge sort setups and two distributed merge sort setups in two processor's core used configuration, higher lines indicate worsening performance because the execution time becomes longer and vice versa lower lines indicate better performance because of the execution time is getting faster.

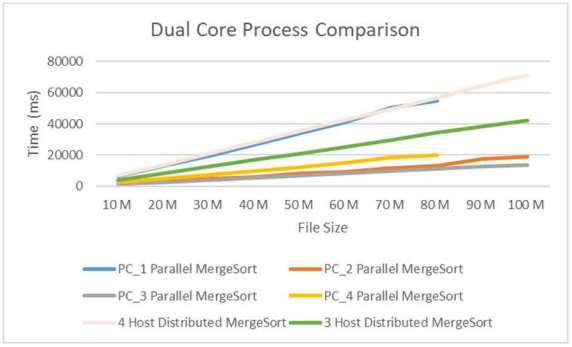


*Figure 10: Performance Comparison in Dual Core Configuration*

The performance observes base on the Figure 10 graph showing that distributed merge sort overall performance is worse than parallel merge sort overall performance. its performance is even much worse than nodes with the lowest specification compared with distributed merge sort with the most nodes setup, oddly distributed merge sort with the least node setup has better

performance than distributed merge sort with the most nodes setup.

### 8.2. Quad Cores Comparison

The following graph also showing a line diagram to visualize the difference performance between four parallel merge sort setups and two distributed merge sort setups in four processor's core used configuration, using similar measurement approaches like the dual cores line graph counterpart that is a higher line indicates worsening performance because the execution time becomes longer and vice versa lower lines indicate better performance because of the execution time is getting faster.

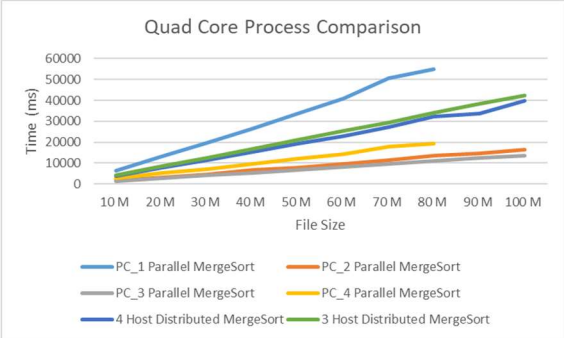


*Figure 11: Performance Comparison in Quad-Core Configuration*

Performance comparison observed based on figure 11 showing dramatic improvement of distributed merge sort with most nodes performance. Although the distributed merge sort with most nodes setup is better than the distributed merge sort with the least nodes and parallel merge sort with the lowest hardware specification, the overall performance of distributed merge sort still worse than the overall performance of parallel merge sort in quad cores configuration.

## 9. CONCLUSION AND FUTURE DIRECTION

Based on the designing, analysis, and test results and discussion, from the implementation of merge sort algorithms on a distributed system, then the conclusions obtained are as follows :

- Distributed merge sort algorithm can improve the performance of parallel computing merge sort on the nodes with low hardware specifications, however, raises the bottleneck effect on nodes with higher hardware specification when combined with a node with the lowest hardware specification in a distributed system environment.
- Distributed merge sort performance on dual-core configuration is 167.119% worse than the parallel merge sort performance on dual-core configuration and distributed merge sort performance on quad-core configuration is 93.5708% worse than the performance of parallel merge sort on the quad-core configuration. So it can be summed up a different performance of parallel merge sort with distributed merge sort is -130.345% or 2.3x slower.
- Parallel merge sort algorithm and distributed merge sort algorithm speed heavily influenced the number of processors utilized based on the system configuration.

Based on the result of conducting research, high-performance computing with implementing merge sort algorithm on the distributed system has no performance improvement. In the development and testing of the system, the author finds some weaknesses and errors in this study. Here are some suggestions and factors to be aware of as consideration for research, repair, and research development in the future, that is :

- To achieve a better and more accurate result in the next research, the entire used nodes in the system's cluster environment needs to be homogenous.
- Try using another divide and conquer algorithm such as quicksort, bitonic sort, column sort, sample sort, partitioned radix sort, etc.
- Try using another middleware such as manta RMI, MPI, restful API, SOAP, etc.
- Try using more participant nodes in the cluster environment.
- For the next research better use another programming approach in division task held in the client node rather than just dividing it based on the number of available nodes.

## REFERENCES:

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[1] An Overview of RMI Applications, https://docs.oracle.com/javase/tutorial/rmi/overview.html
[2] Alyasseri, Zaid & Al-Attar, Kadhim & Nasser, Mazin, Parallelize Bubble and Merge Sort Algorithms Using Message Passing Interface (MPI), 2014.
[3] Andrew, Maarten, *Distributed Systems Principles and Paradigms*, Pearson Education Inc, 2016.
[4] George, Jean et. all, *DISTRIBUTED SYSTEMS Concept and Design*, Addison-Wesley, 2012.
[5] Grosso, W., *Java RMI* (1st ed.), O'Reilly, 2001.
[6] Janeš, I., An Overview of Distributed Programming Techniques, University of Zagreb, 2019.
[7] Java RMI – Introduction, https://www.tutorialspoint.com/java_rmi/java_rmi_introduction.htm.
[8] Minsoo Jeon, Dongseung Kim, Load-Balanced Parallel Merge Sort on Distributed Memory Parallel computers, Korea University, 2002.
[9] Nugroho, A. a. , Dcom, Corba, Java Rmi: Konsep Dan Teknik Dasar Pemrograman. *Jurnal Sistem Informasi, 7*, 2011, pp. 132-142.
[10] Radenski, A., Shared Memory, Message Passing, and Hybrid Merge Sort for Standalone and Clustered SMPs. *CSREA Press*, 2011.
[11] Shi, Feng & Yan, Zhiyuan & Wagh, Meghanad., An Enhanced Multiway Sorting Network Based on n-Sorters. *2014 IEEE Global Conference on Signal and Information Processing*, GlobalISIP 2014, 2014.
[12] Taboada, G. &. T. C. &. T. J., High-Performance Java Remote Method Invocation for Parallel Computing on Clusters, 2007, pp. 233 – 239.
[13] Zbigniew Marzalek, Parallelization of Modified Merge Sort Algorithm, *Symmetry 176*, 2017.