Parallel Merge Sort Algorithm

1. Merge sort Algorithm:

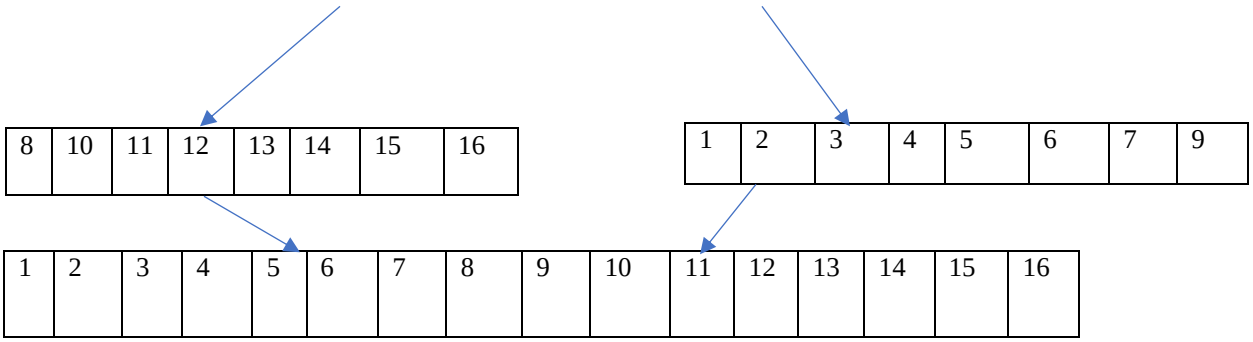
Description:

Merge sort algorithm uses divide and conquer algorithm for sorting. An unsorted list of elements is given as input for the merge sort algorithm. The unsorted list is divided into two equal sub lists and each sub list is again divided into two halves and so on till a sub list with one element is reached.

Now merge two sub lists into a sorted sub list at each step. For this the first element of both the sub lists are compared and the smallest number (for ascending order) is added to the sorted/merged list. If the minimum element is taken from the first list, increment the index of the first list and compare the element at the incremented index of the first list with the first element of the second list.

Whenever the minimum element is selected from a list, copy that minimum element to the sorted/merged list and increment the index of that list and compare with the element at the current index of the other list. Repeat this process and compare all the other elements from each of the sub lists and add the smallest element to the merged list. This process is repeated till the sub lists to be merged are empty and finally we get a single sorted list in ascending order.

Example:



This is the sequential merge sort algorithm and all the steps are sequential and carried out in one process.

2. Parallel Merge Sort Algorithm

Description:

In parallel merge sort implementation, the input unsorted list is divided into 'p' equal parts where 'p' is the number of processes or servers. In this report, 4 processes(servers) are considered for the parallel implementation of the merge sort. So, p=4.

Now the input unsorted list is divided into 4 equal unsorted sub lists and each sub list is given to a server.
Each sub list contains n/4 elements where n is the number of elements in the input list. Each of the 4 servers will perform merge sort in parallel and sort the 4 sub lists. This process is called parallel sort.

Then the sorted sub lists are exchanged in parallel between the servers and merge the sorted sub lists into final sorted list as follows. This step is called parallel merge.

The servers are numbered as 1,2,3,4. For example, when two servers 1 and 2 perform merge of the sorted sub lists, the server 1 sends its n/4 sorted elements to server 2 and then server 2 sends its n/4 sorted elements to server 1.

In the server 1, the first element of the sub list of server 1 is compared with the first element of the sub list received from server 2 and the minimum of the both elements is taken into the new sorted sub list. The index of the sub list from which minimum element is taken is incremented and the element at this incremented index is compared with the element at the current index of the other sub list and minimum element of these both is taken into the new sorted sub list. This process is repeated till the new sorted list contains first minimum n/4 elements from the both sub lists of server 1 and server 2. The remaining n/4 elements are discarded.

In server 2, same process is performed between the sub list of server 2 and the sub list received from server 1. The new sorted sub list in server 2 contains maximum n/4 elements from the both sub lists of server 1 and server 2. The remaining n/4 elements are discarded. This merge process is performed in server 1 and 2 in parallel. Similar merge process is performed by exchanging the sub lists by server 3 and server 4.

After exchanging data, merge process is done in parallel and first n/4 minimum elements are taken in server 3 and maximum n/4 elements in server 4. When two servers exchange data always the new sorted sub list of a lower numbered server stores first n/4 minimum elements and the high numbered server stores the n/4 maximum elements, discarding the remaining elements.

In this implementation, note that if the number of elements in the input unsorted list (n) is not a multiple of 4, then the list is appended with zeroes to make n as the multiple of 4. The exchange of data is performed between all the servers till the first quarter of the final sorted list is at server 1, second quarter at server 2, third quarter at server 3 and fourth quarter at server 4. All the sorted sub lists from these servers are concatenated to form a final sorted list. Extra zeroes that are appended to the input list to make n as multiple of 4 are removed from the final sorted list.

Below is the example for the design if parallel merge sort algorithm.

Example:

Step 1:

P0

16	14	13	15	11	10	12	8	7	9	5	4	3	6	2	1

Step 2:

P1

16	14	13	15

P2

11	10	12	8

P3

7	9	5	4

P4

3	6	2	1

Step 3:

P1

13	14	15	16

P2

8	10	11	12

P3

4	5	7	9

P4

1	2	3	6

Step 4:

P1

13	14	15	16

8	10	11	12

P2

8	10	11	12

13	14	15	16

P3

4	5	7	9

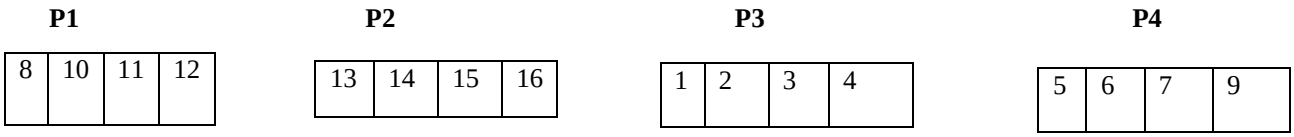
1	2	3	6

P4

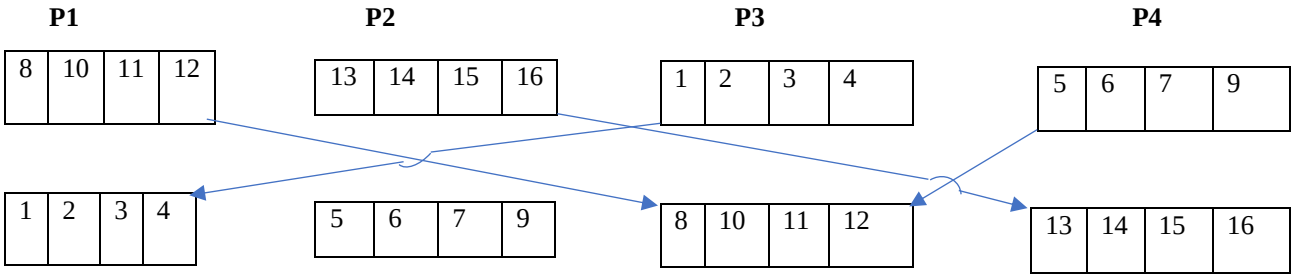
1	2	3	6

4	5	7	9

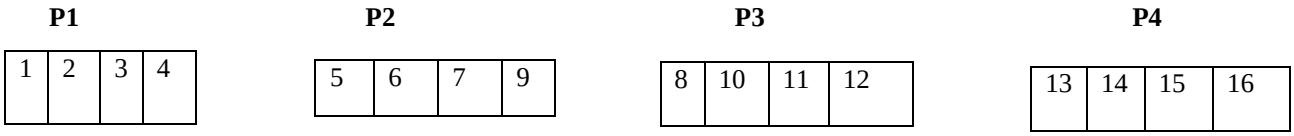
Step 5:



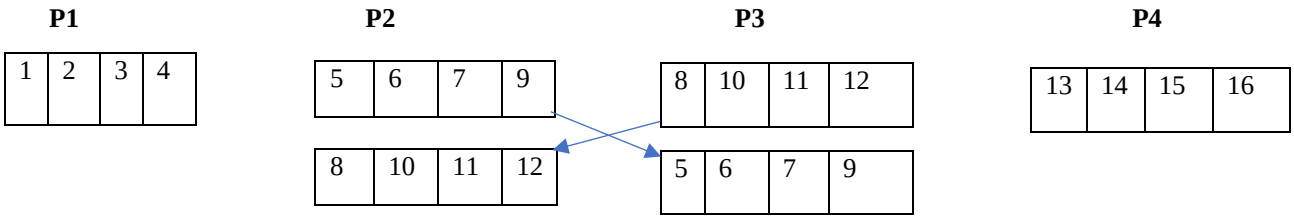
Step 6:



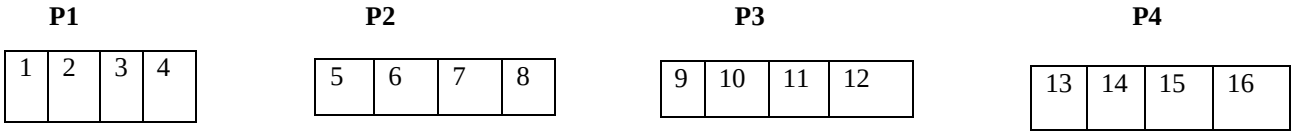
Step 7:



Step 8:



Step 9:



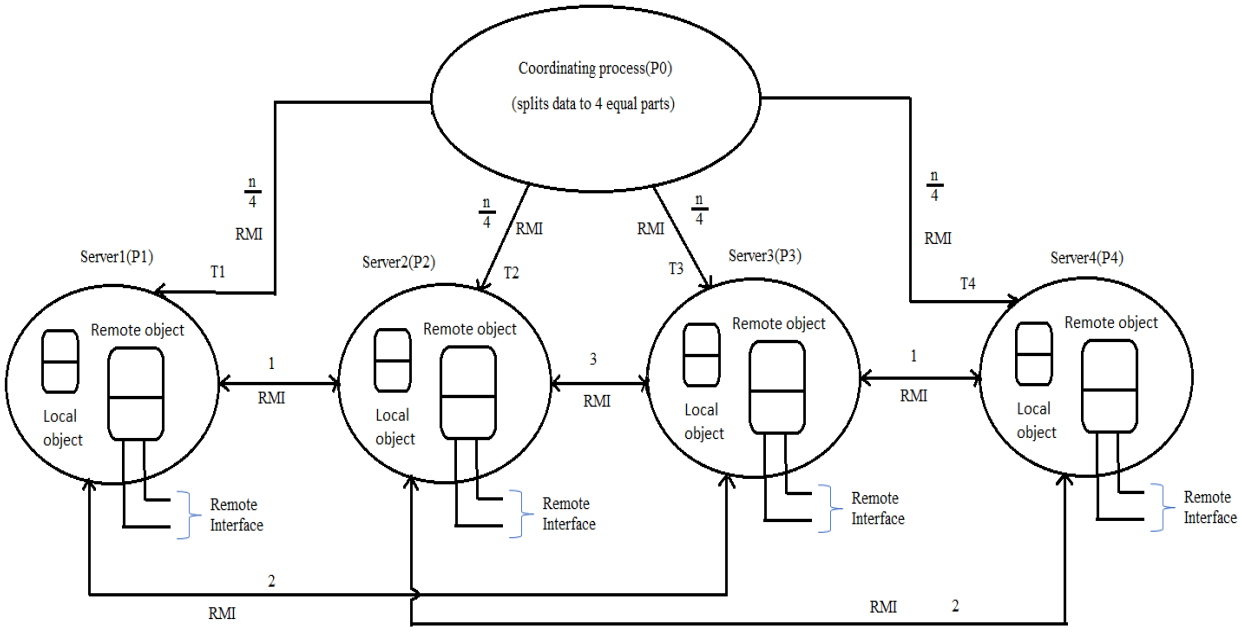
Step 10:

P0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

3. Design of the Parallel Merge sort Algorithm

In this design, we have 5 processes as shown in the above example. One Process (P0) acts as the coordinating process and the remaining 4 processes (P1, P2, P3, P4) acts as 4 servers. Note that the processes are created in the same machine and used to serve the purpose of 4 different machines in a distributed environment. The implementation of parallel merge sort uses Java Remote Method Invocation as Inter Process Communication.



The coordinating process splits the unsorted array of n elements into 4 equal sub arrays, each sub array has n/4 elements and sends each sub array to each server by RMI calls in parallel using 4 threads. Thread T1 for sending data to server 1, T2 to server 2, T3 to server 3 and T4 to server 4.

The servers perform merge sort on the sub arrays received from coordinating process and these sorted sub arrays are exchanged with other servers using RMI calls between the servers to merge the sorted sub arrays.
Below are the exchanges done in order between the servers.

- **Step 1:** The RMI calls numbered '1' in the above figure between server 1 and server 2 and between server 3 and server 4 are done in parallel. Server 1 and server 2 exchange the sub arrays and server 3 and server 4 exchange the sub arrays. After merging, server 1 will store sorted n/4 minimum elements and server 2 will store only sorted n/4 maximum of both sub arrays of server 1 and server 2. server 3 will store sorted n/4 minimum elements and server 4 will store sorted n/4 maximum of both sub arrays of server 3 and server 4. Remaining elements are discarded.
- **Step 2**: The RMI calls numbered '2' in the above figure between server 1 and server 3 and between server 2 and server 4 are done in parallel. Server 1 and server 3 exchange the sub arrays and server 2 and server 4 exchange the sub arrays. After merging, server 1 will store sorted n/4 minimum elements

and server 3 will store sorted n/4 maximum of both sub arrays of server 1 and server 3. server 2 will store sorted n/4 minimum elements and server 4 will store sorted n/4 maximum of both sub arrays of server 2 and server 4. Remaining elements are discarded.

- **Step 3**: The RMI calls numbered '3' in the above figure between server 2 and server 3. Server 1 and server 3 exchange the sub arrays. After merging, server 2 will store sorted n/4 minimum elements and server 3 will store sorted n/4 maximum of both sub arrays of server 2 and server 3.

At the end of all the above-mentioned exchanges and parallel merges between the servers, server 1 will have the first quarter of n/4 elements of the final sorted array. Server 2 will have the second quarter of n/4 elements of the final sorted array. Server 3 will have the third quarter of n/4 elements of the final sorted array. Server 4 will have the fourth quarter of n/4 elements of the final sorted array.

Coordinating process will wait till all the parallel merge sort is done between the servers and collect all the sub arrays and concatenate them to form a single final sorted array.

Remote Method Invocation:

Methods defined in the remote interface of a remote object of a process/server can only be accessed by the objects in other processes/servers. Objects in the same process (local objects) can access the methods in the remote interface as well as the other methods implemented by the remote object.

The implementation in this report has a Remote Interface named 'MergeSort' defined in MergeSort.java file. It is like any other java interface but extends an interface 'Remote' as below. Please refer MergeSort.java in Source Code section for the methods defined in Remote Interface.

public interface **MergeSort** *extends* **Remote**

A remote object is an instance of a class that implements the Remote interface. Methods of a remote interface are implemented in MergeSortServer Class 'MergeSortServer.java' file. All the servers have same methods in the remote interface. Each server creates an instance of the class which implements the methods of the remote interface.

MergeSort *obj1 = new* **MergeSortServer**(*subarr1*)

Here, MergeSortServer is the class that implements the methods of the remote interface MergeSort. 'obj1' is the instance. **MergeSortServer**(*subarr1*) is the constructor of MergeSortServer class which takes the sub array (as an argument) to be sent to that server for which the instance is created.

serverStub1 = (MergeSort) **UnicastRemoteObject.exportObject**(obj1, 0);

UnicastRemoteObject.exportObject is used to make the above created instance(obj1) available to receive RMI invocations by an anonymous TCP port. If '0' is specified in the place of TCP port, then it implies that RMI calls are received by anonymous port.
Ports are specified for severs and binding is done using **createRegistry** and **rebind** methods.

LocateRegistry.createRegistry(*1901*)

The above method creates and exports a Registry instance on the local host that accepts requests on the specified port(eg:1901) on which the registry accepts requests instead of anonymous port.
Naming.rebind("*rmi://localhost:1901/mergeSort*", *serverStub1*)

The above method is used by a server to register the identifier of a remote object(eg:*serverStub1*) by name, which is the URL of the server, here. (e.g.*'rmi://localhost:1901/mergeSort'*). Here serverStub1 is the remote object of server 1.

MergeSort *stub1*=(MergeSort) **Naming.lookup**("*rmi://localhost:1901/mergeSort*")

The above method is used by a client or other servers to look up the remote object reference of a remote object of a server mentioned by server URL and these stubs are used to send RMI to that remote object.

Here stub1 can be created at servers 2,3 and 4 and used to access the methods of remote interface of server 1. Stub1 created at server 1 is local object to server 1 and can access all the methods implemented by the remote object of server 1.

In this section, all the above methods are described for server 1. Same methods are implemented for servers 2,3 and 4 with name changes in objects and ports. Please refer the MergeSortServer.java in Source Code (section 5) for the detailed implementation.

Parallel Merge sort using 2 processes:

Parallel Merge sort can be implemented using 2 processes in the same way described in the sections 2 and 3. Here the input array is split into 2 equal parts and send to 2 servers. There will be one exchange of data between servers instead of 5 exchanges in case of 4 servers.

In the source code (section 5), sequential merge sort, parallel merge sort using 4 processes, parallel merge sort using 2 processes is implemented and time taken for sorting in each case is calculated.

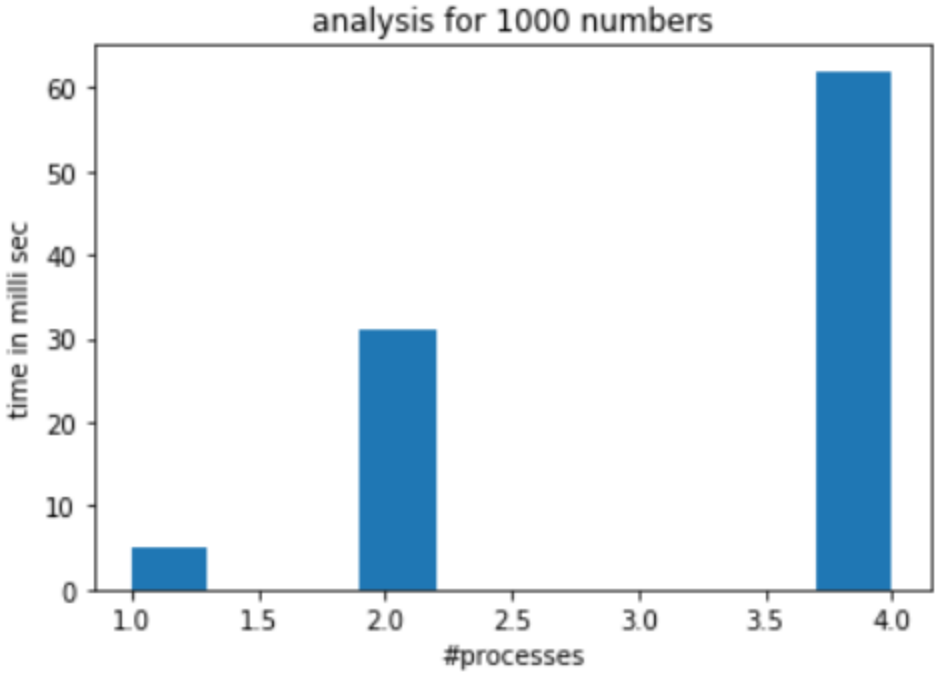
Servers 1,2,3,4 are used for implementing parallel merge sort using 4 processes and Servers 5,6 are used for implementing parallel merge sort using 2 processes.
Ports used for server 1,2,3,4,5,6 are 1901,1902,1903,1904,1905,1905,1906 respectively.

4. Discussion

The parallel merge sort algorithm implemented in this report is efficient in a distributed environment of machines which has less memory and where sorting need to be performed memory efficient. Because the input list is split into sub lists and each server is required to handle the data of atmost double the data of a sub list at any time. And also parallel merge sort is advantageous compared to sequential only for larger data because for smaller amounts of data, time taken for communication overhead (RMI calls) between servers ,exchanges of data between servers and synchronization of parallel tasks is large compared to the time taken for sorting the data.

Performance Analysis:

Time taken for sequential merge sort, parallel merge sort using 2 processes and 4 processes for 1000 numbers are calculated and compared.



From the above graph it is clear that as the number of processes increased ,the time taken for sorting has also increased. This is because all the processes which are in the same machine act as servers.So the time taken for sorting includes RMI calls between servers (increase with processes), exchange of data between servers for merging (number of exchanges increase as the number of processes increase) , synchronization of parallel tasks and context switching of the processes. So the time taken for parallel sorting with 4 processes is large compared to parallel sorting with 2 processes. Parallel sorting with 2 processes is large compared to sequential merge sort.

Future Work:

In this implementation, the exchange of data between the servers and the number of processes are defined manually. The algorithm can be improved by making the servers to exchange the data among themselves dynamically for parallel merge. The algorithm can be implemented to scale dynamically for any number of processes.

Results:

Below are step by step results of sequential merge sort, parallel merge sort using 4 processes and parallel merge sort using 2 processes.

```
Enter the number of elements :
16
Initial unsorted array :[72, 64, 0, 38, 9, 72, 12, 37, 90, 70, 82, 74, 20, 57, 60, 16]

Final sorted array using sequential merge sort :[0, 9, 12, 16, 20, 37, 38, 57, 60, 64, 70, 72, 72, 74, 82, 90]

Parallel merge sort using 4 servers

Initial sorted sub array in server2:[9, 12, 37, 72]

Initial sorted sub array in server1:[0, 38, 64, 72]

sorted sub array in server 2 after exchange with server 1 and merging:[38, 64, 72, 72]

sorted sub array in server 1 after exchange with server 2 and merging:[0, 9, 12, 37]

Initial sorted sub array in server4:[16, 20, 57, 60]

Initial sorted sub array in server3:[70, 74, 82, 90]

sorted sub array in server 3 after exchange with server 4 and merging:[16, 20, 57, 60]

sorted sub array in server 4 after exchange with server 3 and merging:[70, 74, 82, 90]

final sorted sub array in server 1 after exchange with server 3 and merging:[0, 9, 12, 16]

sorted sub array in server 3 after exchange with server 1 and merging:[20, 37, 57, 60]

final sorted sub array in server 4 after exchange with server 2 and merging:[72, 74, 82, 90]

sorted sub array in server 2 after exchange with server 4 and merging:[38, 64, 70, 72]

final sorted sub array in server 3 after exchange with server 2 and merging:[60, 64, 70, 72]

final sorted sub array in server 2 after exchange with server 3 and merging:[20, 37, 38, 57]

Time for merge sort using 4 processes: 57

Final Sorted Array : [0, 9, 12, 16, 20, 37, 38, 57, 60, 64, 70, 72, 72, 74, 82, 90]

Parallel merge sort using 2 servers

Initial sorted sub array in server6:[16, 20, 57, 60, 70, 74, 82, 90]

Initial sorted sub array in server5:[0, 9, 12, 37, 38, 64, 72, 72]

sorted sub array in server 6 after exchange with server 5 and merging:[60, 64, 70, 72, 72, 74, 82, 90]

sorted sub array in server 5 after exchange with server 6 and merging:[0, 9, 12, 16, 20, 37, 38, 57]

Time for sorting using 2 processes: 28

Final Sorted Array using 2 processes: [0, 9, 12, 16, 20, 37, 38, 57, 60, 64, 70, 72, 72, 74, 82, 90]
```