Hybrid Sorting Techniques using OpenMP and OpenMPI

Ashwini Kumar
Indian Institute of Technology, Delhi

Abstract

This paper contains various sorting algorithms using parallel programming. Specifically, OpenMP and OpenMPI are used here for writing the programs. We will find out more about Merge Sort, Quick Sort, Radix Sort using hybrid sorting i.e. using both OpenMP and OpenMPI in the same program. At last we will have conclusions that using OpenMP and OpenMPI separately gave better results instead of using them in a hybrid way.

Keywords: OpenMP, OpenMPI, Parallel Sorting Algorithms, Hybrid Sorting, Parallel Merge Sort, Parallel Quick Sort, Parallel Radix Sort

1 Introduction

1.1 OpenMPI

MPI is a specification for the developers and users of message passing libraries. Precisely, it is not a library instead a specification of what such a library should be. MPI primarily addresses the message-passing parallel programming model. OpenMPI [Open MPI: Open Source High Performance Computing] is one such implementation. The Open MPI Project is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners. In MPI, we use a set of library routine calls to communicate and synchronize between processes. It is useful in many supercomputers. It also gets importance in GPU based algorithms for Deep learning techniques being applied to Speech Recognition, Computer Vision, etc. . Horovod [uber/horovod] is a distributed deep learning framework; supports TensorFlow, PyTorch along with the use of OpenMPI. With this one can make use of multiple GPUs across multiple machines for faster training of a model than training it using a single GPU.

1.2 OpenMP

OpenMP is a compiler-directive based shared-memory programming model, which allows sequential programmers to quickly graduate to parallel programming. It is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. The API supports C/C++ and Fortran on a wide variety of architectures.The application programming interface OpenMP supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows. The OpenMP APIiitd supports, on a variety of platforms, programming of shared memory multiprocessing. With OpenMP, C/C++ and Fortran programmers use a set of compiler directives (pragmas), library routines, and environment variables to specify multi-threaded execution that is implicitly managed by the OpenMP implementation.

1.3 Sorting Algorithms

There are multiple sorting techniques implemented till date (for e.g. Merge, Quick, Radix, Bubble, etc.) but in this paper we would be looking at the shared memory, message passing and combination of message passing and shared memory implementation styles for Merge Sort, Quick Sort, and Radix Sort. For a brief introduction, we can say that Merge sort is the best representative for divide and conquer sorting algorithms. It divides an array of data into two halves and then have a recursive call on each half. Quick Sort works in almost the same way but it takes an element as pivot in the array of data and then divides the array into two arrays. A recursive call is made upon each of the arrays created. Radix Sort is an algorithm in which we use bucket sorting technique to put each data element into some numbered bucket. After allotting each element in the bucket, we collect the elements from each bucket from the least numbered bucket to highest numbered bucket. Let us discuss the parallel sorting versions of each of the three sorting algorithms.

2 MERGE SORT

As described in introduction, for serial merge sort, we can show its implementation as described in the figure 1:

```
void mergeSortSerial(int a[], int low, int high){
if(low<high){
mergeSortSerial(a, low, (low+high)/2);
mergeSortSerial(a, ((low+high)/2)+1, high);
merge(a,low, (low+high)/2, high);
}
```

Figure 1: Serial Merge Sort Implementation

2.1 Shared Memory Parallel Programming Model:

The shared memory programming model for Merge sort is implemented using OpenMP. In this(refer to Fig. 2), we use task pragma directive calls in OpenMP to assign work on each of the automatically created threads. Each task calls the mergeSortMP() function recursively on the one half of the original array. The two tasks return the sorted halves and then these are merged in the same way as did in the sequential algorithm for merge sort. We can also use parallel sections in place of tasks but in practical implementation, tasks gave better results.

```
void mergeSortMP(int a[], int size, int threads) {
if ( threads == 1)
{ mergeSortSerial(a, low, high); }
else if (threads > 1) {
#pragma omp parallel sections {
#pragma omp section
mergeSortMP(a, size/2, temp, threads/2);
#pragma omp section
mergeSortMP(a + size/2, size - size/2, temp + size/2, threads - threads/2);
}
merge(a, size, temp);
} // threads > 1
}
```

Figure 2: Shared Memory Merge Sort Implementation using OpenMP

2.2 Message Passing Parallel Programming Model:

The message passing programming model for merge sort can be interpreted as the root process(rank 0) having all the data elements and then calling rank 1 processor as a helper process. 0th processor splits the data into half and then keep the other half with itself. Then 0th process again calls 2nd processor as a helper process and sends half of the present data to it. Each of the helper processes also call helper processes in the same way.This procedure is recursively called until each process has

received some data elements to sort in a serial manner. Then going from down to up 0th process merges the data obtained from each of the helper process to obtain the finally sorted data.

```
void mergeSortMPI(int a[], int size, int temp[], int level, ) {
// my_rank is used to calculate helper rank:
int helper_rank = my_rank + pow(2, level);
if (helper_rank > max_rank) {
mergeSortSerial(a, size, temp);
} else {
// send second half of array, asynchronous:
MPI_Isend(a+size/2, size-size/2, ..., helper_rank, ...);
// sort first half:
mergeSortMPI(a, size/2, temp, level+1, ...);
// receive second half sorted:
MPI_Recv(a+size/2, size-size/2, ..., helper_rank, ...)
// merge the two sorted sub-arrays:
merge(a, size, temp);
}
}
```

Figure 3: Distributed Memory Merge Sort Implementation using OpenMPI

2.3 Combined(Hybrid) Parallel Programming Model for Shared Memory and Message Passing Programming Models:

For combined implementation using both OpenMPI and OpenMP, we simply use the openmp implementation in place of serial sort used in MPI model. In this architecture, shared and distributed memory are combined. It is also referred to as 'Hybrid'/'Multi Level' architecture. An SMP cluster of multi-processor multi-core nodes is a typical example of a hybrid parallel system.The advantage of using both- OpenMP and OpenMPI is that OpenMPI provides coarse-grain parallelism with recursive calls on each process(as described above in Message Passing model) while OpenMP provides a finer grain parallelism by working over multiple threads in a single MPI process. Although theoretically it improves performance but in practical implementation, it results in slight decrease of performance as compared to individual programming models of shared memory and message passing.

3 QUICK SORT

Quick sort is considered to be one of the reliable sorting algorithm which adapts to parallelization easily. As described in introduction, for serial quick sort, we can show its implementation as described in figure 5:

```
void mergeSortHybrid(int a[], int size, int temp[], int level, int threads, ...  )
{
int helper_rank = my_rank + pow(2, level);
if (helper_rank > max_rank) {
mergeSortMP(a, size, temp, threads);
} else {
MPI_Isend(a+size/2, size-size/2, ..., helper_rank, ...);
mergeSortHybrid(a, size/2, temp, level+1, threads, ...);
MPI_Recv(a+size/2, size-size/2, ..., helper_rank, ...);
merge(a, size, temp);
}
}
```

Figure 4: Fig. 4: Hybrid Merge Sort Implementation using OpenMPI and OpenMP

```
int partition (arr[], low, high){
pivot=arr[low]; i=low-1;
for (j = low; j <= high- 1; j++){
if (arr[j] < pivot){ i++; swap arr[i] and arr[j]; }
}  void quickSortSerial(int a[], int low, int high){
if(low<high){
pi = partition(arr, low, high); pi is partitioning index
quickSortSerial(a, low, pi-1);
quickSortSerial(a, pi+1, high);}
}
```

Figure 5: Serial Quick Sort Implementation

3.1 Shared Memory Parallel Programming Model:

In this programming model we use OpenMP to implement a parallel Quick sort algorithm. First, we start by deciding the pivot element among the given data elements. Then two new arrays are created -one having the elements less than or equal to pivot and the other having elements greater than the pivot. Further, we use parallel sections directive calls in OpenMP which assigns work on each of the automatically created threads. Each section make a recursive call quickSortMP() for each of the two new arrays created. This is continued until we have a reasonable data size from where on we use serial quick sort. Here is the implementation for the shared memory programming model:

```
void quickSortMP(int a[], int low, int high, int temp[], int threads) {
if ( threads == 1) { quickSortSerial(a, size, temp); }
else if (threads > 1) {
pi = partition(arr, low, high);
#pragma omp parallel sections
{
#pragma omp section
quickSortMP(a, size/2, temp, threads/2);
#pragma omp section
quickSortMP(a + size/2, size - size/2, temp + size/2, threads - threads/2);
}
} // threads > 1
}
```

Figure 6: Shared Memory Quick Sort Implementation using OpenMP

3.2 Message Passing Parallel Programming Model:

In this programming model, we use OpenMPI to sort the data elements. We get the data at the root process. We now separate the communicator into two halves- upper and lower half of processes. Each process in the communicator has a partner process(which can be taken as a process symmetric w.r.t the current process). As a first step of the algorithm, we decide a pivot to be some element as the mid element on each process. After this, we identify elements less than or equal to the pivot and elements greater than pivot which results in creation of two new arrays. Now, if the current process belongs to the lower half of the list of processes, then we keep the array of elements less than or equal to pivot with itself and send the array of elements greater than pivot to the partner process(which is in the upper half of list of processes) and vice versa. We use MPI_Send() and MPI_Recv() routine calls for communication between the processes. We then make a recursive call of quickSortMPI() on the modified array on each process. Here is the implementation for the above:

3.3 Combined(Hybrid) Parallel Programming Model for Shared Memory and Message Passing Programming Models:

For combined implementation using both OpenMPI and OpenMP, we simply use the openmp implementation in place of serial sort used in MPI model. In this architecture, shared and distributed memory are combined. It is also referred to as 'Hybrid'/'Multi Level' architecture. An SMP cluster of multi-processor multi-core nodes is a typical example of a hybrid parallel system.The advantage of using both- OpenMP and OpenMPI is that OpenMPI provides coarse-grain parallelism with recursive

```
void quickSortMPI(int* arr, int size, int process_rank, int max_rank, int
rank_index)
{ int partner_process = process_rank + pow(2, rank_index); rank_index ++;
if(partner_process > max_rank)
{ quickSortSerial(arr, 0,size-1); return;}
int pivot = arr[size/2];
int partition_index = partition(arr, pivot, size,(size/2) -1);
int offset = partition_index + 1;
if(rank==0)
{ if (offset > size - offset)
{ MPI_Send((arr + offset), size - offset,..., share_pr, offset,...);
quickSortMPI(arr, offset, pr_rank, max_rank, rank_index);
MPI_Recv((arr + offset), size - offset,..., share_pr,...); }
else
{ MPI_Send(arr, offset,..., ch_pr,...);
quickSortMPI((arr + offset), size - offset, process_rank, max_rank, rank_index);
MPI_Recv(arr, offset,..., ch_pr,...); } }
else
{ int* subarray = NULL;
int sub_arr_size = 0;
int index_count = 0;
int source_process = 0;
while(pow(2, index_count) <= rank)
index_count ++;
MPI_Probe(MPI_ANY_SOURCE,...);
MPI_Get_count(msgSt, MPI_INT, sub_arr_size);
source_process = msgSt.MPI_SOURCE;
subarray = (int*)malloc(sub_arr_size * sizeof(int));
MPI_Recv(subarray, sub_arr_size, ..., MPI_ANY_SOURCE,...);
int pivot = subarray[(sub_arr_size / 2)];
quickSortMPI(subarray, sub_arr_size, rank, max_rank -1, rec_count);
MPI_Send(subarray, sub_arr_size,..., source_process,...); }
}
```

Figure 7: Distributed Memory Quick Sort Implementation using OpenMPI

calls on each process(as described above in Message Passing model) while OpenMP provides a finer grain parallelism by working over multiple threads in a single MPI process. Although theoretically it improves performance but in practical implementation, it results in slight decrease of performance as compared to individual programming models of shared memory and message passing.

```
void quickSortHybrid(int* arr, int size, int process_rank, int max_rank, int
rank_index)
{ int partner_process = process_rank + pow(2, rank_index); rank_index ++;
if(partner_process > max_rank)
{ quickSortMP(arr, 0,size-1); return;}
int pivot = arr[size/2];
int partition_index = partition(arr, pivot, size,(size/2) -1);
int offset = partition_index + 1;
if(rank==0)
{ if (offset > size - offset)
{ MPI_Send((arr + offset), size - offset,..., share_pr, offset,...);
quickSortHybrid(arr, offset, pr_rank, max_rank, rank_index);
MPI_Recv((arr + offset), size - offset,..., share_pr,...); }
else
{ MPI_Send(arr, offset,..., ch_pr,...);
quickSortHybrid((arr + offset), size - offset, process_rank, max_rank, rank_index);
MPI_Recv(arr, offset,..., ch_pr,...); } }
else
{ int* subarray = NULL;
int sub_arr_size = 0;
int index_count = 0;
int source_process = 0;
while(pow(2, index_count) <= rank)
index_count ++;
MPI_Probe(MPI_ANY_SOURCE,...);
MPI_Get_count(msgSt, MPI_INT, sub_arr_size);
source_process = msgSt.MPI_SOURCE;
subarray = (int*)malloc(sub_arr_size * sizeof(int));
MPI_Recv(subarray, sub_arr_size, ..., MPI_ANY_SOURCE,...);
int pivot = subarray[(sub_arr_size / 2)];
quickSortHybrid(subarray, sub_arr_size, rank, max_rank -1, rec_count);
MPI_Send(subarray, sub_arr_size,..., source_process,...); }
}
```

Figure 8: Hybrid Quick Sort Implementation using OpenMPI and OpenMP

4 RADIX SORT

As described in introduction, for serial radix sort, we can show its implementation as described in the figure 9:

```
void radixSortSerial(int a[], int bit_number_from_left){
a=bucketSortSerial(a, 10, bit_number_from_left);// returns array after bucket sort
radixSortSerial(a, bit_number_from_left+1);//recursive call for next bit
}
```

Figure 9: Serial Radix Sort Implementation

4.1 Shared Memory Parallel Programming Model:

In this programming model we use OpenMP to implement a parallel Radix sort algorithm. We start to use bucket sort from Least Significant Bit(LSB). We use base 10 bucket sort but the same algorithm can be implemented for base 2 bucket sort. We first put each element into some bucket numbered from 0-9. For doing this, we assign some portion of data array to each available thread and then each thread puts in the respective buckets. So, a parallelism is created while we put elements into the bucket. This is repeated till the maximum number of digits any data element(or the key of data element) can have. Finally, we start to collect the elements from least numbered bucket to highest numbered bucket. This works well for positive numbered data elements but for implementing the same for negative numbers, we can store the negative numbers in reverse order in the final step and keep all the positive numbers ahead of all the negative numbers.

4.2 Message Passing Parallel Programming Model:

In this programming model, we use OpenMPI to sort the data elements. Similar to shared memory programming, we start to use bucket sort from Least Significant Bit(LSB). We use base 10 bucket sort but the same algorithm can be implemented for base 2 bucket sort. For this, we first locally use serial bucket sort on each process to keep the elements into the buckets. After doing these on all processes, we collectively empty the buckets starting from process with rank 0. The process with rank 0 keeps a certain number of elements by itself and sends the rest to the next processor. This procedure is followed till all the buckets get emptied. We do this for each digit of the data elements till we reach the last digit of the number having maximum number of digits.

```
void radixSortMP(int a[],int size) {
#pragma omp parallel shared (S, D, Gl) private (LCacc)
for (int dig = least_significant_digit; dig< most_significant_digit;dig++){
for(int i = 0:i< 10;i++)
Gl[i][pid]= 0;
}
#pragma omp for private (i, value) schedule (static)
for (int i = 0;i< N;i++){
value = get digit value(S[i], dig)
Gl[value][pid] = Gl[value][pid]+1
}
partial sum (Gl, LCacc)
#pragma omp for private (i, value) schedule (static)
for (int i = 0;i< N;i++){
value = get digit value(S[i], dig)
D[LCacc[value]] = S[i]
LCacc[value] = LCacc[value]+1}
#pragma omp single
swap addresses ( S, D )
}
```

Figure 10: Shared Memory Radix Sort Implementation using OpenMP

```
void radixSortMPI(int a[], int size, int temp[], int level, ) {
for (int dig = least_significant_digit; dig< most_significant_digit;dig++){
initialize (LC, nbuckets);
for (int i = 0;i< N;i++){
value= get digit value(S[i], dig);
LC[value]= LC[value]+1;}
tmp =LC[0]; LC[0]= 0;
for(int i = 0:i< 10;i++){
accum = tmp + LC[i -1];
tmp = LC[i];
LC[i] = accum;}
for (int i = 0;i< N;i++){
value = get digit value(S[i], dig);
D[LC[value]] = S[i];
LC[value] =LC[value]+1;}
Gl[pid][0] = LC[0];
for(int i = 0:i< 10;i++) {
Gl[pid][i] ← LC[i] -LC[i  1]; }
allgather (Gl[pid][0], nbuckets, Gl);
bucket distribution(Gl);
data communication(S, D, Gl);
swap addresses ( S, D );
}
```

Figure 11: Distributed Memory Radix Sort Implementation using OpenMPI

```
void radixSortHybrid(int a[], int size, int temp[], int level, ) {
for (int dig = least_significant_digit; dig< most_significant_digit;dig++){
for(int i = 0:i< 10;i++)
Gl[i][pid]= 0;
}
#pragma omp for private (i, value) schedule (static)
for (int i = 0;i< N;i++){
value = get digit value(S[i], dig)
Gl[value][pid] = Gl[value][pid]+1
}
partial sum (Gl, LCacc)
#pragma omp for private (i, value) schedule (static)
for (int i = 0;i< N;i++){
value = get digit value(S[i], dig)
D[LCacc[value]] = S[i]
LCacc[value] = LCacc[value]+1}
allgather (Gl[pid][0], nbuckets, Gl);
bucket distribution(Gl);
data communication(S, D, Gl);
swap addresses ( S, D );
}
```

Figure 12: Hybrid Radix Sort Implementation using OpenMPI and OpenMP

4.3 Combined(Hybrid) Parallel Programming Model for Shared Memory and Message Passing Programming Models:

As we do in Hybrid Quick and Merge Sort, here also we replace the serial computation by using multi threading i.e. OpenMP. The serial computation in which we place data elements into buckets in the distributed memory model, is being replaced by bucketSortMP(). The remaining computation is then kept the same as that in the distributed memory model.

5 Performance and Observations:

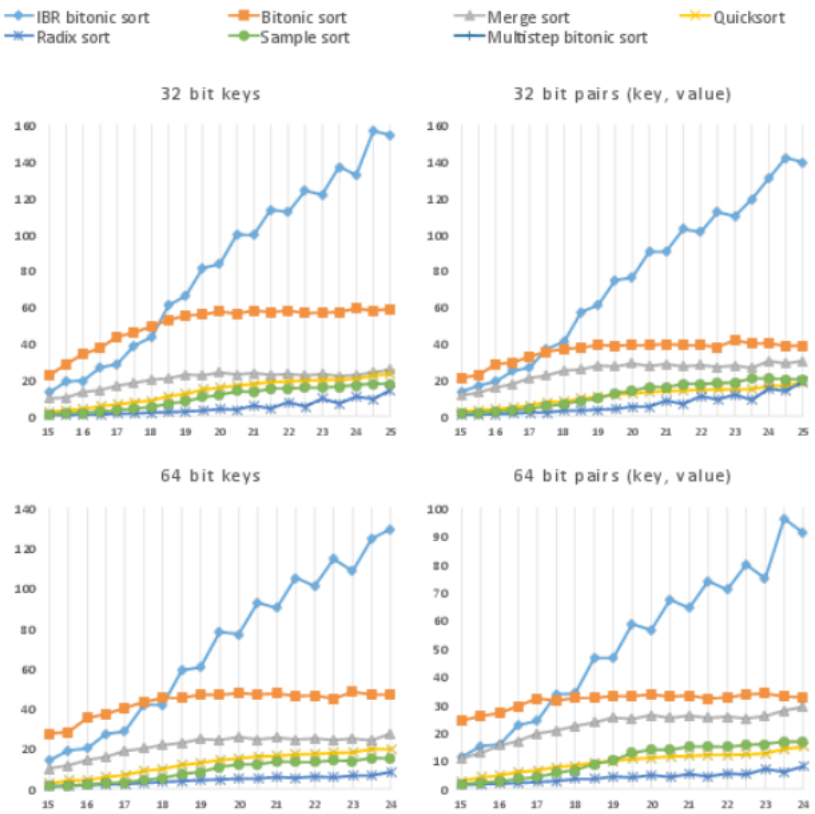

Figure 13: Sorting Comparison: Time(in M/s) vs data size

Data Size	Merge Sort	Quick Sort	Radix Sort
2^{10}	0.00168208	0.000318627	0.00312351
2^{15}	0.0098825	0.003530493	0.0017384
2^{18}	0.0837457	0.0383349	0.0510034
2^{20}	0.343039	0.118931	0.0917004
2^{22}	1.42095	0.39033	0.152871
2^{24}	5.95737	1.92097	1.94514
2^{26}	18.6	9.0159	8.511

Table 1: Shared Memory Performance Analysis

Number of processors	Merge Sort	Quick Sort	Radix Sort
1	1.1	1.004	1.26
4	1.9	2.2	1.54
16	3.78	6	3.57
32	9.3	13	8.7
64	28.9	35.2	21.6

Table 2: Distributed Memory Performance Analysis(on data size = 2^{20})

Data Size	Merge Sort	Quick Sort	Radix Sort
2^{10}	0.00572	0.000809	0.002355
2^{15}	0.01205	0.012691	0.0103814
2^{18}	0.2754	0.317935	0.150034
2^{20}	0.93	0.674325	0.517004
2^{22}	1.50547	1.393	1.2871
2^{24}	4.6574	3.727	3.456
2^{26}	24.983	15.0589	20.511

Table 3: Combined(Hybrid) Shared and Distributed Memory Performance Analysis

6 Conclusion

This paper introduces three sorting algorithms- Merge Sort, Quick Sort, Radix Sort of which each of them has been implemented in Shared Memory Programming Model(using OpenMP), Message Passing Programming Model(using OpenMPI) and Hybrid Programming Model(using OpenMP and OpenMPI). The speedup obtained by multiprocessing and multi threading environments has been theoretically and experimentally studied. This paper puts forward the observation for each of the three sorting algorithms with each implemented in Shared Memory, Message Passing and Hybrid Model. Of all these, the Shared Memory programming model proved to perform best for all the three sorting algorithms. Performance of hybrid model lies between Shared Memory model and Message Passing model. Among Merge sort, Quick Sort, and Radix Sort, Quick Sort proved to be most efficient in practice. The performance of a programming model is solely dependent on the fact

that how many cores each process has and how well do these processes communicate.

7 References:

[1] Bozidar, Darko Dobravec, Tomaž. (2015). Comparison of parallel sorting algorithms.

[2] Jiménez-González, Dani Larriba-Pey, Josep-Lluis Navarro, Juan. (2003). Case Study: Memory Conscious Parallel Sorting. 10.1007/3-540-36574-5_16.

[3] Elnashar, Alaa. (2011). Parallel Performance of MPI Sorting Algorithms on Dual-Core Processor Windows-Based Systems. International Journal of Distributed and Parallel systems. 2. 10.5121/ijdps.2011.2301.

[4] Sharma, Prince. (2016). Analysis of Parallel and Sequential Radix Sort for Graph Exploration using OpenMP and CUDA: A Review.

[5] https://www.codeproject.com/KB/threads/Parallel_Quicksort/Parallel_Quick_sort_without_merge.pdf

[6] https://www.diva-portal.org/smash/get/diva2:839729/FULLTEXT02

[7] https://www3.nd.edu/ zxu2/acms60212-40212-S12/Cray09-hybrid-MPI-OpenMP.pdf

[8] Radenski, Atanas. (2011). Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs. 367-373.

8 Design for Assignment3

COL730 Assignment 3

Ashwini Kumar

2018MT60778

In this assignment, we have implemented merge sort and quick sort using a combined version of OpenMP and OpenMPI. Let us explain them.

9 Quick Sort:

As we did in the OpenMPI version,first we take up the last processor and then choose the pivot value for all processors to be the mid element of the 0th processor. The rank 0 processor broadcasts it to every processor. Then we take a partner process for each process to be the process at rank=(number_of_processes-my_rank). Then keep the data less than or equal to (in both the current process and the partner process) in the current process if it belongs to the lower half and send the data elements greater than pivot to the partner process which belongs to the upper half of the list of processors. Then split the communicator into two halves and then call recursively on each of the communicators.

This communication is continued till the point there is only one processor in the communicator. When there is only one processor, then we use OpenMP implementation or the shared memory implementation to carry out further computation i.e. when we have only one processor ,then use multithreading to sort the data elements.
For OpenMP/Shared memory implementation, first we decide a pivot to be the value as the last element of the array and then calculate the partition index as we do in sequential version of quick sort. The array is then separated into two portions- left and right array. We then use two sections in a recursive call of quick sort and assign left half to one of them and the other half to other section. This is continued recursively till we reach a threshold data size where we switch to sequential quick sort to remove the overhead created by parallelization on a small data set.

This way we sort the given data set using Combined Quick sort(OpenMP +OpenMPI).

9.1 Some important points:

The significant differences from the algorithm mentioned in the term paper and my implementation is that
(1) In OpenMP implementation, I have used sequential quick sort when data size becomes equal to some certain data size whereas the algorithm mentioned in the term paper uses number of threads as the criteria to decide when to switch to sequential version.
(2) I have splitted the communicator into two communicator instead of using groups as we saw in the term paper.

9.2 Observations:

OpenMP performed the best on the same data size as compared to Combined(Hybrid) version and then OpenMPI.
i.e. OpenMP/Shared Memory Model> Combined(Hybrid) Model > OpenMPI/Distributed Memory Model.

10 Merge Sort:

We start by having the data on each processor so then we choose a helper process on which we send one half of the data elements. This processor then sends the half of this data sent, to another helper process, in a recursive call. This is carried out until every process has at least once received the data elements from some parent process. Thus each process now has some data elements for which we use OpenMP/Shared memory implementation of merge sort.

For OpenMP implementation, we split the given array into two halves. We then similarly use sections and assign each half of the array to each one of the sections. These sections call merge sort recursively on each half of the array. This recursive call is made till we reach a certain data size after which we switch to sequential merge sort. After each half is sorted, we merge these two halves as we do in sequential merge sort.

This way data elements are sorted using Combined Merge Sort(OpenMP +OpenMPI).

10.1 Some important points:

The significant differences from the algorithm mentioned in the term paper and my implementation is that

(1) In OpenMP implementation, I have used sequential merge when data size becomes equal to some certain data size whereas the algorithm mentioned in the term paper uses number of threads as the criteria to decide when to switch to sequential version.

(2) In term paper we assume data is given at one process but here we have data distributed on each process and then we sort them such that ith processor contains elements less than elements at the jth processor for i<j.

10.2 Observations:

OpenMP performed the best on the same data size as compared to Combined(Hybrid) version and then OpenMPI.

i.e. OpenMP/Shared Memory Model> Combined(Hybrid) Model > OpenMPI/Distributed Memory Model.

Data Size	Merge Sort	Quick Sort
2^{10}	0.00572	0.000809
2^{15}	0.01205	0.012691
2^{18}	0.2754	0.317935
2^{20}	0.93	0.674325
2^{22}	1.50547	1.393
2^{24}	4.6574	3.727
2^{26}	24.983	15.0589

Table 4: Combined(Hybrid) Shared and Distributed Memory Performance Analysis