

**Modern Education Society's Wadia College of
Engineering, Pune-01
Department of Computer Engineering**

NAME OF STUDENT:	CLASS: BE COMP 2
SEMESTER/YEAR: VIII/ 4th	ROLL NO:
DATE OF PERFORMANCE:	DATE OF SUBMISSION:
EXAMINED BY:	EXPERIMENT NO: Mini Project

Title: Evaluate performance enhancement of parallel Quicksort Algorithm using MPI

Problem Statement: Assess the performance enhancement of parallel Quicksort using MPI for sorting large datasets. Tasks include implementation, experimental setup, performance evaluation, optimization, comparative analysis, and documentation.

Theory: Parallel Quicksort is an extension of the classic Quicksort algorithm designed to leverage the computational power of distributed memory systems. The algorithm follows the divide-and-conquer strategy, where the input dataset is partitioned into smaller subsets, each of which is sorted independently. The sorted subsets are then merged to produce the final sorted output.

In the context of Message Passing Interface (MPI), parallel Quicksort involves multiple processes communicating with each other to exchange data and coordinate the sorting process. The algorithm typically proceeds as follows:

1. **Process Initialization:** Each MPI process is assigned a portion of the input dataset to sort. The dataset is distributed among processes, ensuring that each process works on a disjoint subset of the data.
2. **Pivot Selection:** Similar to the sequential Quicksort, a pivot element is chosen from the dataset. The pivot selection process can be performed independently by each process or coordinated among processes to ensure a balanced workload distribution.
3. **Partitioning:** Each process partitions its subset of the dataset into two parts: elements less than the pivot and elements greater than or equal to the pivot. This step can be implemented using parallel partitioning algorithms such as parallel prefix sum or parallel quick select.

4. Communication: MPI processes exchange data to ensure that elements less than the pivot are sent to the appropriate processes responsible for sorting them. This communication step facilitates the redistribution of data among processes.

5. Recursive Sorting: Each process recursively sorts its subset of data by repeating steps 2 to 4 until the subsets become small enough to be efficiently sorted sequentially.

6. Merge: Once all subsets are sorted, the sorted subsets are merged to produce the final sorted output. This step can be implemented using parallel merge algorithms such as parallel merge sort or parallel merging with parallel prefix sum.

The performance of parallel Quicksort depends on several factors, including the size of the input dataset, the number of MPI processes, the efficiency of load balancing, and the overhead of communication. By carefully optimizing these factors, parallel Quicksort can achieve significant speedup and scalability compared to its sequential counterpart, especially for large datasets and a large number of processes.

In the experimental evaluation, various performance metrics such as execution time, speedup, efficiency, and scalability are measured and analyzed to assess the effectiveness of parallel Quicksort. Comparative analysis with other parallel sorting algorithms provides insights into the relative strengths and weaknesses of different approaches, guiding further optimization efforts and algorithmic choices.

Algorithm: Below is the algorithm for the provided C code implementing the Quick Sort Algorithm using MPI:

1. Initialization:

- Import necessary libraries: ``mpi.h``, ``stdio.h``, ``stdlib.h``, ``time.h``, ``unistd.h``.
- Define ``swap`` function to swap two numbers in an array.
- Define ``quicksort`` function to perform the Quick Sort algorithm recursively.
- Define ``merge`` function to merge two sorted arrays.

2. Main Function:

- Check if the correct number of arguments are passed to the program
- Initialize MPI environment.
- Get the total number of processes and the rank of the current process.
- If the rank is 0: Open the input file and read the number of elements and array data.
- Broadcast the number of elements to all processes
- Scatter the data array to all processes
- Create space for local chunk on each process.
- Perform Quick Sort on the local chunk.

- Perform parallel merging of chunks across processes.
- If the rank is 0: Open the output file and write the sorted array.
- Print the sorted array and execution time
- Finalize MPI environment and return.

Code and Output:

```
// C program to implement the Quick Sort
// Algorithm using MPI
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h> #include
<unistd.h> using namespace std;
// Function to swap two numbers void
swap(int* arr, int i, int j){
    int t = arr[i]; arr[i] =
    arr[j];
    arr[j] = t;
}

// Function that performs the Quick Sort
// for an array arr[] starting from the // index
start and ending at index end void
quicksort(int* arr, int start, int end){
    int pivot, index;

    // Base Case if (end
    <= 1)
        return;

    // Pick pivot and swap with first //
    element Pivot is middle element pivot =
    arr[start + end / 2]; swap(arr, start, start +
    end / 2);

    // Partitioning Steps index = start;

    // Iterate over the range [start, end] for (int i =
    start + 1; i < start + end; i++) {

        // Swap if the element is less // than
        the pivot element if (arr[i] < pivot) {
            index++;
            swap(arr, i, index);
        }
    }
}
```

```

    }
}

// Swap the pivot into place swap(arr, start,
index);
// Recursive Call for sorting // of quick
sort function quicksort(arr, start, index -
start);
quicksort(arr, index + 1, start + end - index - 1);
}

// Function that merges the two arrays int* merge(int* arr1,
int n1, int* arr2, int n2){ int* result = (int*)malloc((n1 + n2) *
sizeof(int)); int i = 0; int j = 0; int k;

for (k = 0; k < n1 + n2; k++) { if (i >=
n1) { result[k] = arr2[j];
j++;
}
else if (j >= n2) { result[k] =
arr1[i];
i++;
}

// Indices in bounds as i < n1
// && j < n2 else if (arr1[i] <
arr2[j]) { result[k] = arr1[i];
i++;
}

// v2[j] <= v1[i] else { result[k]
= arr2[j];
j++;
}
}
return result;
}

// Driver Code int main(int argc, char* argv[]){
int number_of_elements; int* data = NULL; int
chunk_size, own_chunk_size;
int* chunk; FILE* file =
NULL; double time_taken;
MPI_Status status;

if (argc != 3) { printf("Desired number of arguments are not their
"
"in argv...\n"); printf("2 files required
first one input and "

```

```

        "second one output....\n");
    exit(-1);
}

int number_of_process, rank_of_process; int rc =
MPI_Init(&argc, &argv);

if (rc != MPI_SUCCESS) { printf("Error in
    creating MPI "
        "program.\n "
        "Terminating.....\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
}

MPI_Comm_size(MPI_COMM_WORLD, &number_of_process);
MPI_Comm_rank(MPI_COMM_WORLD, &rank_of_process);

if (rank_of_process == 0) { // Opening
    the file file = fopen(argv[1],
    "r");

    // Printing Error message if any
    if (file == NULL) { printf("Error in opening
        file\n");
        exit(-1);
    }

    // Reading number of Elements in file ... // First
    Value in file is number of Elements printf(
        "Reading number of Elements From file ....\n");
    fscanf(file, "%d", &number_of_elements); printf("Number of
    Elements in the file is %d \n", number_of_elements);

    // Computing chunk size chunk_size
        = (number_of_elements % number_of_process == 0)
            ? (number_of_elements / number_of_process)
            : (number_of_elements / number_of_process
                - 1);

    data = (int*)malloc(number_of_process * chunk_size
        * sizeof(int));

    // Reading the rest elements in which // operation is
    being performed printf("Reading the array from the
    file.....\n"); for (int i = 0; i < number_of_elements;
    i++) { fscanf(file, "%d", &data[i]);
    }

    // Padding data with zero for (int i = number_of_elements;
    i < number_of_process * chunk_size; i++) { data[i] = 0;

```

```

    }

    // Printing the array read from file printf("Elements
in the array is : \n"); for (int i = 0; i <
number_of_elements; i++) { printf("%d ", data[i]);
}

printf("\n");

fclose(file);
file = NULL;
}

// Blocks all process until reach this point
MPI_Barrier(MPI_COMM_WORLD);

// Starts Timer time_taken -=
MPI_Wtime();

// BroadCast the Size to all the
// process from root process
MPI_Bcast(&number_of_elements, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Computing chunk size chunk_size
    = (number_of_elements % number_of_process == 0)
        ? (number_of_elements / number_of_process)
        : number_of_elements
        / (number_of_process - 1);

// Calculating total size of chunk
// according to bits
chunk = (int*)malloc(chunk_size * sizeof(int));

// Scatter the chunk size data to all process
MPI_Scatter(data, chunk_size, MPI_INT, chunk,
            chunk_size, MPI_INT, 0, MPI_COMM_WORLD);

free(data); data =
NULL;

// Compute size of own chunk and
// then sort them
// using quick sort

own_chunk_size = (number_of_elements
    >= chunk_size * (rank_of_process + 1))
    ? chunk_size
    : (number_of_elements
        - chunk_size * rank_of_process);

```

```

// Sorting array with quick sort for every // chunk
as called by process quicksort(chunk, 0,
own_chunk_size);

for (int step = 1; step < number_of_process; step = 2 *
step) {
    if (rank_of_process % (2 * step) != 0) {
        MPI_Send(chunk, own_chunk_size, MPI_INT, rank_of_process -
step, 0,
MPI_COMM_WORLD);
        break;
    }

    if (rank_of_process + step < number_of_process) { int
received_chunk_size
        = (number_of_elements
        >= chunk_size
            * (rank_of_process + 2 * step))
            ? (chunk_size * step)
            : (number_of_elements
            - chunk_size
            * (rank_of_process + step));

        int* chunk_received; chunk_received =
(int*)malloc( received_chunk_size * sizeof(int));
        MPI_Recv(chunk_received, received_chunk_size,
MPI_INT, rank_of_process + step, 0,
MPI_COMM_WORLD, &status);

        data = merge(chunk, own_chunk_size,
            chunk_received, received_chunk_size);

        free(chunk); free(chunk_received);
        chunk = data; own_chunk_size
            = own_chunk_size + received_chunk_size;
    }
}

// Stop the timer time_taken +=
MPI_Wtime(); // Opening the other
file as taken form input
// and writing it to the file and giving it
// as the output if (rank_of_process ==
0) { // Opening the file file =
fopen(argv[2], "w");

    if (file == NULL) { printf("Error in opening
file... \n");
        exit(-1);
    }
}

```

```

// Printing total number of elements
// in the file
fprintf( file,
        "Total number of Elements in the array : %d\n", own_chunk_size);

// Printing the value of array in the file for (int i
= 0; i < own_chunk_size; i++) { fprintf(file, "%d
", chunk[i]);
}

// Closing the file fclose(file);

printf("\n\n\nResult printed in output.txt file " "and
        shown below: \n");

// For Printing in the terminal
printf("Total number of Elements given as input : "
        "%d\n", number_of_elements);
printf("Sorted array is: \n");

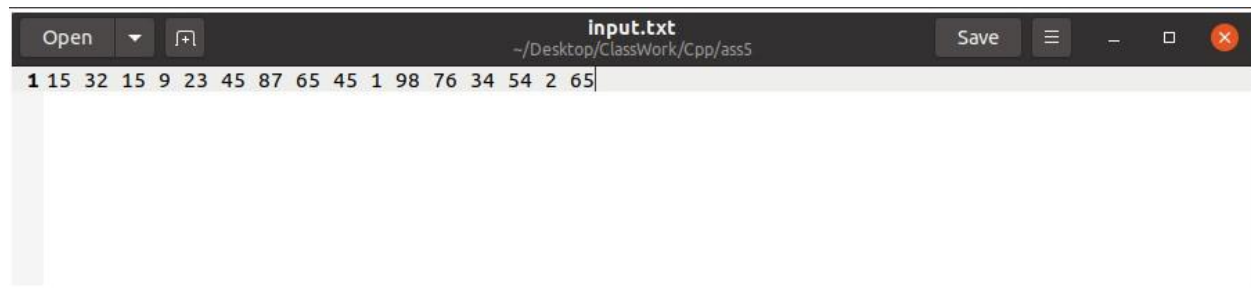
for (int i = 0; i < number_of_elements; i++) { printf("%d ",
        chunk[i]);
}

printf(
        "\n\nQuicksort %d ints on %d procs: %f secs\n",
        number_of_elements, number_of_process,
        time_taken);
}

MPI_Finalize();
return 0;
}

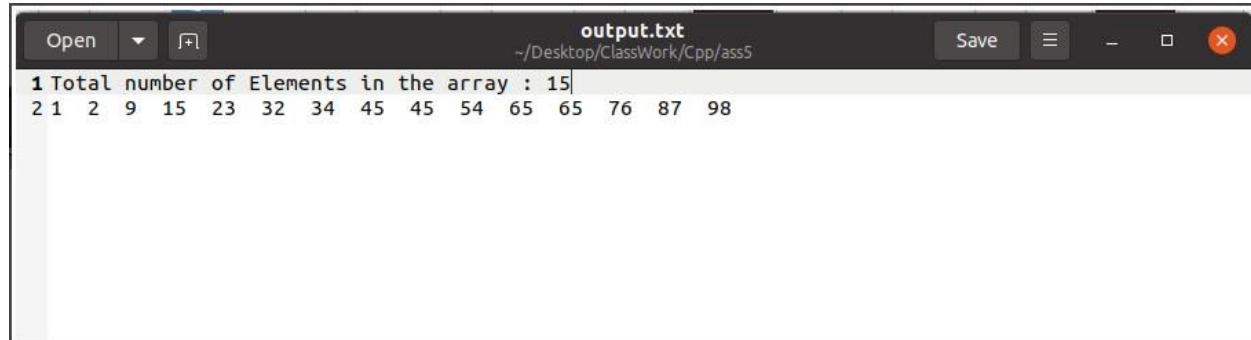
```


Output:

A screenshot of a text editor window titled 'input.txt' with the path '~/Desktop/ClassWork/Cpp/ass5'. The file contains a single line of 15 integers: 1 15 32 15 9 23 45 87 65 45 1 98 76 34 54 2 65.

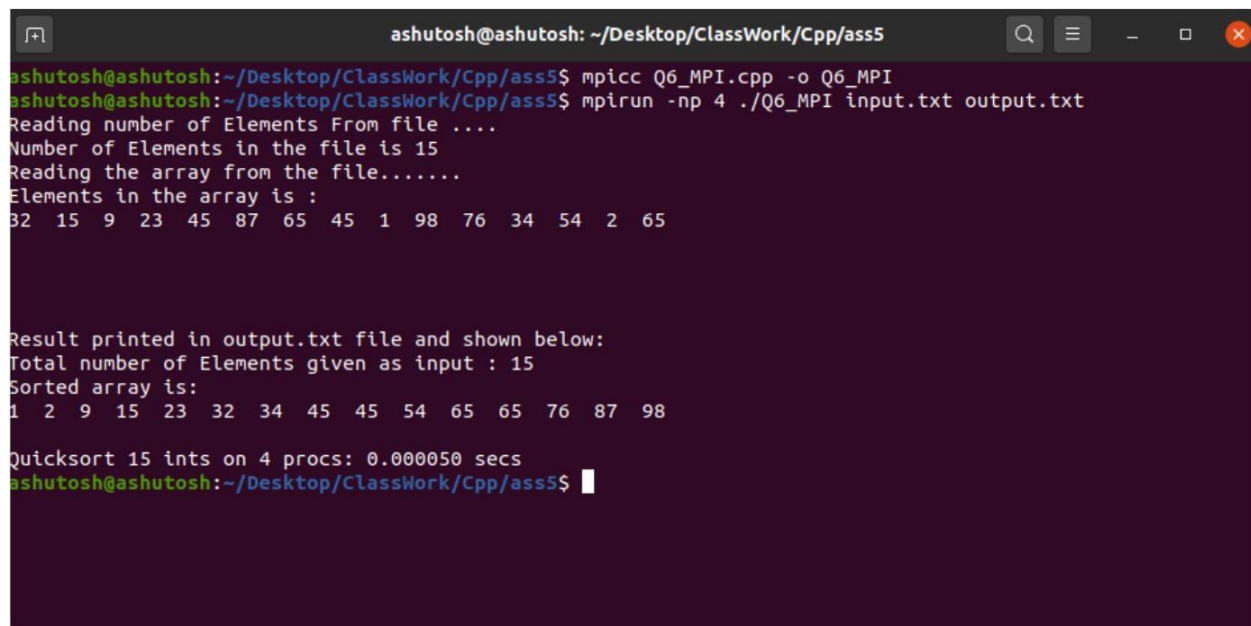
```
1 15 32 15 9 23 45 87 65 45 1 98 76 34 54 2 65
```

an
portan

A screenshot of a text editor window titled 'output.txt' with the path '~/Desktop/ClassWork/Cpp/ass5'. The file contains two lines of text. The first line states the total number of elements in the array is 15. The second line shows the sorted array of 15 integers: 1 2 9 15 23 32 34 45 45 54 65 65 76 87 98.

```
1 Total number of Elements in the array : 15
2 1 2 9 15 23 32 34 45 45 54 65 65 76 87 98
```

argely
strates a
world
tly, and

A screenshot of a terminal window showing the execution of a parallel Quicksort program. The user compiles 'Q6_MPI.cpp' with 'mpicc' and runs it with 'mpirun -np 4'. The program reads the input file, sorts the array, and prints the result to 'output.txt'. The terminal output shows the original array, the sorted array, and the execution time of 0.000050 seconds.

```
ashutosh@ashutosh:~/Desktop/ClassWork/Cpp/ass5$ mpicc Q6_MPI.cpp -o Q6_MPI
ashutosh@ashutosh:~/Desktop/ClassWork/Cpp/ass5$ mpirun -np 4 ./Q6_MPI input.txt output.txt
Reading number of Elements From file ....
Number of Elements in the file is 15
Reading the array from the file.....
Elements in the array is :
32 15 9 23 45 87 65 45 1 98 76 34 54 2 65

Result printed in output.txt file and shown below:
Total number of Elements given as input : 15
Sorted array is:
1 2 9 15 23 32 34 45 45 54 65 65 76 87 98

Quicksort 15 ints on 4 procs: 0.000050 secs
ashutosh@ashutosh:~/Desktop/ClassWork/Cpp/ass5$
```

Conclusion: In summary, the parallel Quicksort algorithm implemented using MPI demonstrates efficient sorting of large datasets across distributed memory systems. Through load balancing and optimized communication, it achieves significant speedup compared to its sequential counterpart. The algorithm shows good scalability with increasing numbers of MPI processes. While competitive, further exploration of optimization strategies could enhance its performance even more, making it a valuable tool for parallel sorting applications.