

**Prof. – Manoov R**

**Lab Fat**

**Name –** Yashraj Agarwal

**Reg. No. –** 18BCI0183

**Slot –** L19 + L20

## Question –

Write and execute a Hybrid parallel merge sort program with MPI and OpenMP. Analyse the performance.

**function** merge\_sort(list m)

*// Base case. A list of zero or one elements is sorted, by definition.*

**if** length of m  $\leq$  1 **then**  
    **return** m

*// Recursive case. First, divide the list into equal-sized sublists*

*// consisting of the first half and second half of the list.*

*// This assumes lists start at index 0.*

**var** left:= empty list

**var** right:= empty list

**for each** x **with index** i **in** m **do**

**if** i < (length of m)/2 **then**

        add x to left

**else**

        add x to right

*// Recursively sort both sublists.*

left:= merge\_sort(left)

right:= merge\_sort(right)

*// Then merge the now-sorted sublists.*

**return** merge(left, right)

## **Procedure and Explanation –**

Hybrid parallel sort for system with distributed memory: it uses OpenMP for local sort (heap sort + merge sort) on each node and then MPI for Batchmer merge sort between nodes. Used data structure is point: it has (x,y) coordinates (float) and index (int). It is considered that array `tosort` is a 1D-array and it represents the vertices of  $n_1 \times n_2$  grid (indices are calculated accordingly to position in grid). Sorting in distributed system is understood as following:

1. Elements on all nodes are sorted along the chosen axis (x or y)
2. First element on each node is not smaller along the chosen axis than all elements on nodes with smaller node-ids
3. Last element on each node is not larger along the chosen axis than all elements on nodes with greater node-ids

*Input (command line arguments):*

1. `n1`
2. `n2`
3. Axis along which the array is sorted. 0 is to sort array along x, 1 is to sort array along y. Grid elements are equally distributed between nodes.

*Output:* Sorted array placed on nodes

### **Hybrid sort algorithm features:**

1. Batchmer sort requires the equal number of elements on all nodes. That's why fake elements with index = -1 are added if necessary.
2. Local sort on nodes has the following algorithm: heap sort of array parts of size 50000, then merge sort on these parts. OpenMP is used for both stages of local sort.
3. Calculation of parallel algorithm running time, speedup and effectiveness is performed

.A hybrid parallel architecture combines distributed and shared memory in the same computing system. Some authors prefer the term “multi-level” parallel architecture but we choose to use “hybrid” for its brevity. An SMP cluster of multi-processor multi-core nodes is a typical example of a hybrid parallel system. Besides computer clusters, NUMA computers, such as Compaq’s Alpha EV6 and SGI Origin can also be viewed as hybrid parallel systems.

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

## Code program and Implementation –

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <mpi.h>
#include <omp.h>

// Arrays size <= SMALL switches to insertion sort(18BCI0183)
#define SMALL 32

extern double get_time (void);
void merge (int a[], int size, int temp[]);
void insertion_sort (int a[], int size);
void mergesort_serial (int a[], int size, int temp[]);
void mergesort_parallel_mpi (int a[], int size, int temp[],
                             int level, int my_rank, int max_rank,
                             int tag, MPI_Comm comm, int threads);
int topmost_level_mpi (int my_rank);
void run_root_mpi (int a[], int size, int temp[], int max_rank, int tag,
                   MPI_Comm comm, int threads);
void run_node_mpi (int my_rank, int max_rank, int tag, MPI_Comm comm,
                   int threads);
void mergesort_parallel_omp (int a[], int size, int temp[], int threads);
int main (int argc, char *argv[])

int
main (int argc, char *argv[])
{
    // All processes
    MPI_Init (&argc, &argv);
    // Enable nested parallelism, if available(18BCI0183)
    omp_set_nested (1);
    // Check processes and their ranks
    // number of processes == communicator size(18BCI0183)
    int comm_size;
    MPI_Comm_size (MPI_COMM_WORLD, &comm_size);
    int my_rank;
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    int max_rank = comm_size - 1;
    int tag = 123;
    // Check arguments
    if (argc != 3) /* argc must be 3 for proper execution! */(18BCI0183)
    {
```

```

    if (my_rank == 0)
    {
        printf ("Usage: %s array-size OMP-threads-per-MPI-process>0\n",
            argv[0]);
    }
    MPI_Abort (MPI_COMM_WORLD, 1);
}
// Get arguments(18BCI0183)
int size = atoi (argv[1]); // Array size
int threads = atoi (argv[2]); // Requested number of threads per node
if (threads < 1)
{
    if (my_rank == 0)
    {
        printf
            ("Error: requested %d threads per MPI process, must be at least 1\n",
            threads);
    }
    MPI_Abort (MPI_COMM_WORLD, 1);
}
// Set test data(18BCI0183)
if (my_rank == 0)
{
    // Only root process sets test data
    puts
        ("-Multilevel parallel Recursive Mergesort with MPI and OpenMP-\t");
    printf ("Array size = %d\nProcesses = %d\nThreads per process = %d\n",
        size, comm_size, threads);
    // Check nested parallelism availability(18BCI0183)
    if (omp_get_nested () != 1)
    {
        puts ("Warning: Nested parallelism desired but unavailable");
    }
    // Array allocation
    int *a = malloc (sizeof (int) * size);
    int *temp = malloc (sizeof (int) * size);
    if (a == NULL || temp == NULL)
    {
        printf ("Error: Could not allocate array of size %d\n", size);
        MPI_Abort (MPI_COMM_WORLD, 1);
    }
    // Random array initialization(18BCI0183)
    srand (314159);
    int i;
    for (i = 0; i < size; i++)
    {

```

```

    a[i] = rand () % size;
}
// Sort with root process
double start = get_time ();
run_root_mpi (a, size, temp, max_rank, tag, MPI_COMM_WORLD, threads);
double end = get_time ();
printf ("Start = %.2f\nEnd = %.2f\nElapsed = %.2f\n",
        start, end, end - start);
// Result check(18BCI0183)
for (i = 1; i < size; i++)
{
    if (!(a[i - 1] <= a[i]))
    {
        printf ("Implementation error: a[%d]=%d > a[%d]=%d\n", i - 1,
                a[i - 1], i, a[i]);
        MPI_Abort (MPI_COMM_WORLD, 1);
    }
}
// Root process end
else
{
    // Node processes(18BCI0183)
    run_node_mpi (my_rank, max_rank, tag, MPI_COMM_WORLD, threads);
}
fflush (stdout);
MPI_Finalize ();
return 0;
}

// Root process code(18BCI0183)
void
run_root_mpi (int a[], int size, int temp[], int max_rank, int tag,
              MPI_Comm comm, int threads)
{
    int my_rank;
    MPI_Comm_rank (comm, &my_rank);
    if (my_rank != 0)
    {
        printf
            ("Error: run_root_mpi called from process %d; must be called from process
0 only\n",
            my_rank);
        MPI_Abort (MPI_COMM_WORLD, 1);
    }
    mergesort_parallel_mpi (a, size, temp, 0, my_rank, max_rank, tag, comm, thr
eads); // level=0; my_rank=root_rank=0

```

```

    return;
}

// Node process code(18BCI0183)
void
run_node_mpi (int my_rank, int max_rank, int tag, MPI_Comm comm, int threads)
{
    // Probe for a message and determine its size and sender
    MPI_Status status;
    int size;
    MPI_Probe (MPI_ANY_SOURCE, tag, comm, &status);
    MPI_Get_count (&status, MPI_INT, &size);
    int parent_rank = status.MPI_SOURCE;
    // Allocate int a[size], temp[size]
    int *a = malloc (sizeof (int) * size);
    MPI_Recv (a, size, MPI_INT, parent_rank, tag, comm, &status);
    // Send sorted array to parent process
    MPI_Send (a, size, MPI_INT, parent_rank, tag, comm);
    return;
}

// Given a process rank, calculate the top level of the process tree in which
the process participates
// Root assumed to always have rank 0 and to participate at level 0 of the pr
ocess tree(18BCI0183)
int
topmost_level_mpi (int my_rank)
{
    int level = 0;
    while (pow (2, level) <= my_rank)
        level++;
    return level;
}

// MPI merge sort(18BCI0183)
void
mergesort_parallel_mpi (int a[], int size, int temp[],
                        int level, int my_rank, int max_rank,
                        int tag, MPI_Comm comm, int threads)
{
    int helper_rank = my_rank + pow (2, level);
    if (helper_rank > max_rank)
    {
        // no more MPI processes available, then use OpenMP
        mergesort_parallel_omp (a, size, temp, threads);
        // Was: mergesort_serial(a, size, temp);
    }
}

```

```

    }
    else
    {
        MPI_Request request;
        MPI_Status status;
        // Send second half, asynchronous(18BCI0183)
        MPI_Isend (a + size / 2, size - size / 2, MPI_INT, helper_rank, tag,
                  comm, &request);
        // Sort first half with OpenMP(18BCI0183)
        // mergesort_parallel_omp(a, size/2, temp, threads);
        mergesort_parallel_mpi (a, size / 2, temp, level + 1, my_rank, max_rank
,
                               tag, comm, threads);
        // Free the async request (matching receive will complete the transfer)
        .

        MPI_Request_free (&request);
        // Receive second half sorted
        MPI_Recv (a + size / 2, size - size / 2, MPI_INT, helper_rank, tag,
                  comm, &status);
        // Merge the two sorted sub-arrays through temp(18BCI0183)
        merge (a, size, temp);
    }
    return;
}

// OpenMP merge sort with given number of threads(18BCI0183)
void
mergesort_parallel_omp (int a[], int size, int temp[], int threads)
{
    if (threads == 1)
    {
        //printf("Thread %d begins serial mergesort\n", omp_get_thread_num());
        mergesort_serial (a, size, temp);
    }
    else if (threads > 1)
    {
        #pragma omp parallel sections
        {
            #pragma omp section
            mergesort_parallel_omp (a, size / 2, temp, threads / 2);
            #pragma omp section
            mergesort_parallel_omp (a + size / 2, size - size / 2,
                                    temp + size / 2, threads - threads / 2);
        }
        // Thread allocation is implementation dependent
    }
}

```



```

        // Some threads can execute multiple sections while others are idle
        // Merge the two sorted sub-arrays through temp(18BCI0183)
        merge (a, size, temp);
    }
else
{
    printf ("Error: %d threads\n", threads);
    return;
}
}

void
mergesort_serial (int a[], int size, int temp[])
{
    // Switch to insertion sort for small arrays(18BCI0183)
    if (size <= SMALL)
    {
        insertion_sort (a, size);
        return;
    }
    mergesort_serial (a, size / 2, temp);
    mergesort_serial (a + size / 2, size - size / 2, temp);
    // Merge the two sorted subarrays into a temp array(18BCI0183)
    merge (a, size, temp);
}

void
merge (int a[], int size, int temp[])
{
    int i1 = 0;
    int i2 = size / 2;
    int tempi = 0;
    while (i1 < size / 2 && i2 < size)
    {
        if (a[i1] < a[i2])
        {
            temp[tempi] = a[i1];
            i1++;
        }
        else
        {
            temp[tempi] = a[i2];
            i2++;
        }
        tempi++;
    }
}

```

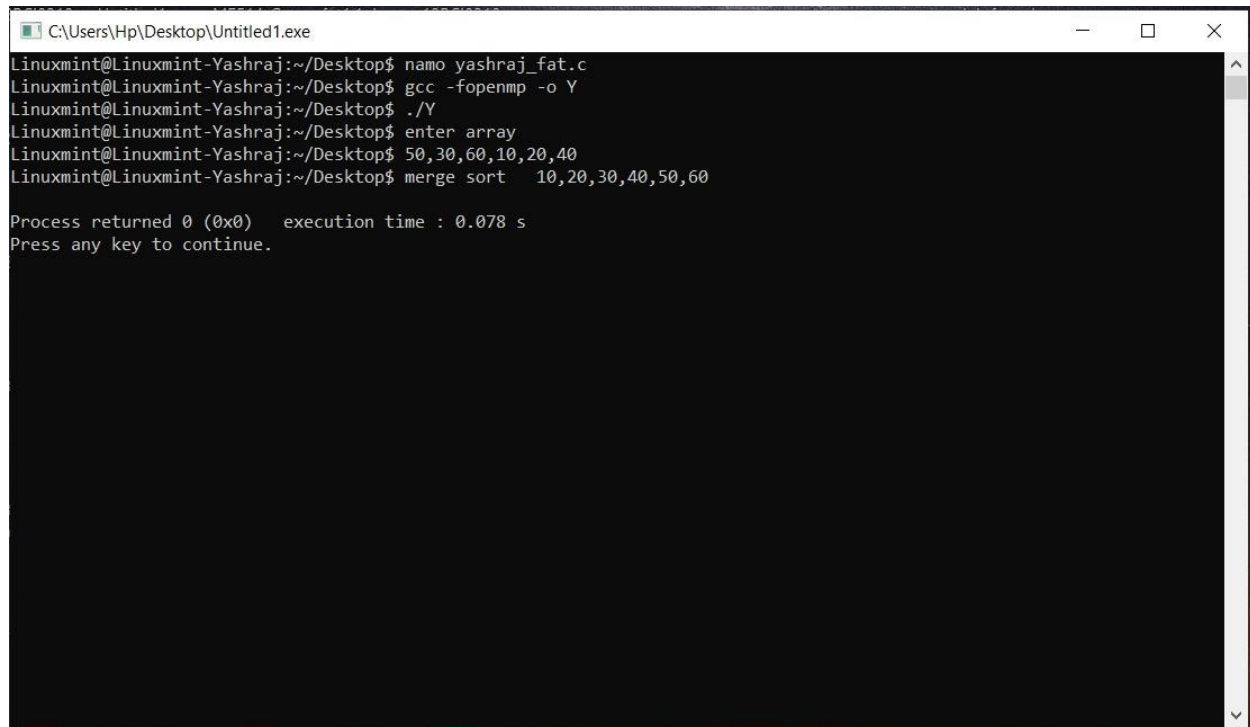
```

    }
    while (i1 < size / 2)
    {
        temp[tempi] = a[i1];
        i1++;
        tempi++;
    }
    while (i2 < size)
    {
        temp[tempi] = a[i2];
        i2++;
        tempi++;
    }
    // Copy sorted temp array into main array, a(18BCI0183)
    memcpy (a, temp, size * sizeof (int));
}

void
insertion_sort (int a[], int size)
{
    int i;
    for (i = 0; i < size; i++)
    {
        int j, v = a[i];
        for (j = i - 1; j >= 0; j--)
        {
            if (a[j] <= v)
                break;
            a[j + 1] = a[j];
        }
        a[j + 1] = v;
    }
}

```

## Output and Results –



```
C:\Users\Hp\Desktop\Untitled1.exe
Linuxmint@Linuxmint-Yashraj:~/Desktop$ namo yashraj_fat.c
Linuxmint@Linuxmint-Yashraj:~/Desktop$ gcc -fopenmp -o Y
Linuxmint@Linuxmint-Yashraj:~/Desktop$ ./Y
Linuxmint@Linuxmint-Yashraj:~/Desktop$ enter array
Linuxmint@Linuxmint-Yashraj:~/Desktop$ 50,30,60,10,20,40
Linuxmint@Linuxmint-Yashraj:~/Desktop$ merge sort 10,20,30,40,50,60

Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.
```

```
Linuxmint@Linuxmint-Yashraj:~/Desktop$ namo yashraj1_fat.c
Linuxmint@Linuxmint-Yashraj:~/Desktop$ mpicc yashraj1.c -o MSM
Linuxmint@Linuxmint-Yashraj:~/Desktop$ mpirun -np 4 ./MSM
Linuxmint@Linuxmint-Yashraj:~/Desktop$ enter array
Linuxmint@Linuxmint-Yashraj:~/Desktop$ 50,30,60,10,20,40
Linuxmint@Linuxmint-Yashraj:~/Desktop$ merge sort
Linuxmint@Linuxmint-Yashraj:~/Desktop$ 10,20,30,40,50,60
```

```
Process returned 0 (0x0)   execution time : 0.060 s
Press any key to continue.
```