RV COLLEGE OF ENGINEERING®,

BENGALURU-560059

(Autonomous Institution Affiliated to VTU, Belagavi)

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

# "Hybrid Programming with MPI + OpenMP"

***ASSIGNMENT REPORT***

***Submitted by***


**Pratheeksha P 1RV17CS112**


***in partial fulfillment for the requirement of 7th Semester***

***Parallel Architecture and Distributed Programming (16CS71)***

**Submitted to**


**Prof. Sandhya S,**
**Assistant Professor,**
**Department of Computer Science and Engineering,**
**R.V. College of Engineering,**
**Bengaluru**


**Academic Year 2020 - 2021**

# Contents:

1. Introduction
2. MPI Support for threads
3. MPI + OpenMP programming styles
4. Code
5. Results
6. References

# 1. Introduction

The basic aims of parallel programming are to decrease the runtime for the solution to a problem and increase the size of the problem that can be solved.The conventional parallel programming practices involve a a pure OpenMP implementation on a shared memory architecture or a pure MPI implementation on distributed memory computer architectures.The largest and fastest computers today employ both shared and distributed memory architecture. This gives a flexibility in tuning the parallelism in the programs to generate maximum efficiency and balance the computational and communication loads in the program. A wise implementation of hybrid parallel programs utilizing the modern hybrid computer hardware can generate massive speedups in the otherwise pure MPI and pure OpenMP implementations.

Hybrid application programs using MPI + OpenMP are now commonplace on large HPC systems. There are essentially two main motivations for this combination of programming models:

1. Reduction in memory footprint, both in the application and in the MPI library (e.g. communication buffers).
2. Improved performance, especially at high core counts where the pure MPI scalability is running out.

# 2. MPI support for threads

In general making libraries thread-safe can be difficult. Internal data structures in the library either have to be replicated (for example, one per thread), or else accesses to these structures must be protected by some form of synchronisation (typically locks). This may add significant overheads which can affect the performance of the library even when only one thread is being used. The MPI standard defines various classes of thread usage. The user can request a certain class for their application and the implementation can return the class that is actually supported.

The MPI specification defines the following four classes of thread safety:

1. **`MPI_THREAD_SINGLE`** – Only one thread will execute.
2. **`MPI_THREAD_FUNNELED`** – The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are funneled to the main thread).
3. **`MPI_THREAD_SERIALIZED`** – The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time.
4. **`MPI_THREAD_MULTIPLE`** – Multiple threads may call MPI, with no restrictions.

# 3. MPI + OpenMP Programming Styles

MPI + OpenMP programs are classified into five distinct styles depending on how multiple OpenMP threads make MPI library calls:

1. **Master-only**
   In Master-Only style all MPI calls are made by the OpenMP master thread, outside of parallel regions.This requires the MPI_THREAD_FUNNELED level of thread support, since other threads will be executing, but not calling MPI. It is implemented by identifying the most time consuming loops and using parallel constructs to multithread them. Disadvantage of this style is that threads other than the master are necessarily idle during MPI calls, and cannot do any useful computation.
   Outline code for this style of MPI + OpenMP program is given by:

```
#pragma omp parallel
{
    work…
}
ierror=MPI_Send(…);
#pragma omp parallel
{
    work…
}
```

2. **Funneled**

   In Funneled style, all MPI calls are made by the OpenMP master thread, but this may include calls from inside OpenMP parallel regions. The advantages of Funneled style are the code is still relatively simple to write and maintain and there are now cheaper ways available to synchronise threads before and after message transfers than closing and opening parallel regions. It is possible for other threads to do useful computation while the master thread is executing MPI calls.

   An outline of code in this style is given by:

```
#pragma omp parallel
{
    … work
    #pragma omp barrier
    #pragma omp master
    {
        ierror=MPI_Send(…);
    }
    #pragma omp barrier
    … work
}
```

3. **Serialized**

   In Serialized style, any thread inside an OpenMP parallel region may make calls to the MPI library, but the threads must be synchronised in such a way that only one thread at a time may be in an MPI call. This style requires MPI_THREAD_SERIALIZED support. Tags or communicators are used to distinguish between messages from (or to) different threads in the same MPI process.

   Outline code in serialized style is given by:

```
#pragma omp parallel
{
    … work
    #pragma omp critical
    {
        ierror=MPI_Send(…);
    }
    … work
}
```

4. **Multiple**

In Multiple style, any thread inside (or outside) a parallel region may call MPI, and there are no restrictions on how many threads may be executing MPI calls at the same time. This requires MPI_THREAD_MULTIPLE support.Synchronization is taken care of by the MPI library. Compared to Master-Only or Funneled style, the internal synchronisation overheads are high which degrades the communication performance.

Outline code in multiple style is given by:

```
#pragma omp parallel
{
     … work
     ierror=MPI_Send(…);
     … work
}
     ierror=MPI_Send(…);
```

5. **Asynchronous Tasks**

In Asynchronous Tasks style, MPI calls occur inside OpenMP dependent tasks. Since multiple threads may make concurrent MPI calls, this is a special case of Multiple style, and thus requires MPI_THREAD_MULTIPLE support.The idea of using OpenMP dependent tasks is to minimise the synchronisation between threads and allow the OpenMP runtime as much flexibility as possible in scheduling computation and communication.

Outline code for this style is given by:

```
#pragma omp parallel
{
#pragma omp single
{
#pragma omp task depend (out:sbuf)
    { … write sbuf }
… more tasks
#pragma omp task depend (in:sbuf) depend
(out:rbuf)
    { MPI_SendRecv(sbuf,…,rbuf,…); }
… more tasks
#pragma omp task depend (in:rbuf)
    { … use rbuf }
}// end single
}// end parallel
```

# 4. Code

This project demonstrates the implementation of merge sort algorithm using several approaches - serial , parallelised using OpenMP, MPI, hybrid models and to observe the performance in each case.

## Serial MergeSort

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#if _POSIX_TIMERS
#include <time.h>
#ifdef CLOCK_MONOTONIC_RAW
/* System clock id passed to clock_gettime. CLOCK_MONOTONIC_RAW
   is preferred.  It has been available in the Linux kernel
   since version 2.6.28 */
#define SYS_RT_CLOCK_ID CLOCK_MONOTONIC_RAW
#else
#define SYS_RT_CLOCK_ID CLOCK_MONOTONIC
#endif
double get_time(void)
{
    struct timespec ts;
    double t;
    if (clock_gettime(SYS_RT_CLOCK_ID, &ts) != 0)
    {
        perror("clock_gettime");
        abort();
    }
    t = (double)ts.tv_sec + (double)ts.tv_nsec * 1.0e-9;
    return t;
}

#else /* !_POSIX_TIMERS */
#include <sys/time.h>

double get_time(void)
{
    struct timeval tv;
    double t;
    if (gettimeofday(&tv, NULL) != 0)
    {
        perror("gettimeofday");
        abort();
    }
    t = (double)tv.tv_sec + (double)tv.tv_usec * 1.0e-6;
    return t;
}
```

```
#endif


// Arrays size <= SMALL switches to insertion sort
#define SMALL    32

void merge (int a[], int size, int temp[]);
void insertion_sort (int a[], int size);
void mergesort_serial (int a[], int size, int temp[]);
extern double get_time (void);
int main (int argc, char *argv[]);

int main (int argc, char *argv[])
{
  puts ("-Serial Recursive Mergesort-\t");
  // Check arguments
  if (argc != 2)         /* argc must be 2 for proper execution!
*/
    {
      printf ("Usage: %s array-size\n", argv[0]);
      return 1;
    }
  // Get arguments
  int size = atoi (argv[1]);  // Array size
  printf ("Array size = %d\n", size);
  // Array allocation
  int *a = (int*)malloc (sizeof (int) * size);
  int *temp = (int*)malloc (sizeof (int) * size);
  if (a == NULL || temp == NULL)
    {
      printf ("Error: Could not allocate array of size %d\n",
size);
      return 1;
    }
  // Random array initialization
  int i;
  srand (314159);
  for (i = 0; i < size; i++)
    {
      a[i] = rand () % size;
    }
  // Sort
  double start = get_time ();
  mergesort_serial (a, size, temp);
  double end = get_time ();
  printf ("Start = %.2f\nEnd = %.2f\nElapsed = %.2f\n",
       start, end, end - start);
  // Result check
  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]);
        return 1;
      }
    }
  puts ("-Success-");
  return 0;
}

void
mergesort_serial (int a[], int size, int temp[])
{
  // Switch to insertion sort for small arrays
  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
  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
  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;
    }
}
```

# MergeSort parallelized using OpenMP

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <omp.h>
#if _POSIX_TIMERS
#include <time.h>
#ifdef CLOCK_MONOTONIC_RAW
/* System clock id passed to clock_gettime. CLOCK_MONOTONIC_RAW
   is preferred.  It has been available in the Linux kernel
   since version 2.6.28 */
#define SYS_RT_CLOCK_ID CLOCK_MONOTONIC_RAW
#else
#define SYS_RT_CLOCK_ID CLOCK_MONOTONIC
#endif

double
get_time(void)
{
    struct timespec ts;
    double t;
    if (clock_gettime(SYS_RT_CLOCK_ID, &ts) != 0)
    {
        perror("clock_gettime");
        abort();
    }
    t = (double)ts.tv_sec + (double)ts.tv_nsec * 1.0e-9;
    return t;
}

#else /* !_POSIX_TIMERS */
#include <sys/time.h>

double
get_time(void)
{
    struct timeval tv;
    double t;
    if (gettimeofday(&tv, NULL) != 0)
    {
        perror("gettimeofday");
        abort();
    }
    t = (double)tv.tv_sec + (double)tv.tv_usec * 1.0e-6;
    return t;
}

#endif

// Arrays size <= SMALL switches to insertion sort
#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_omp (int a[], int size, int temp[], int
threads);
void run_omp (int a[], int size, int temp[], int threads);
int main (int argc, char *argv[]);

int
main (int argc, char *argv[])
{
  puts ("-OpenMP Recursive Mergesort-\t");
  // Check arguments
  if (argc != 3)         /* argc must be 3 for proper execution!
*/
    {
      printf ("Usage: %s array-size number-of-threads\n",
argv[0]);
      return 1;
    }
  // Get arguments
  int size = atoi (argv[1]);  // Array size
  int threads = atoi (argv[2]);    // Requested number of
threads
  // Check nested parallelism availability
  omp_set_nested (1);
  if (omp_get_nested () != 1)
    {
      puts ("Warning: Nested parallelism desired but
unavailable");
    }
  // Check processors and threads
  int processors = omp_get_num_procs ();     // Available
processors
  printf ("Array size = %d\nProcesses = %d\nProcessors = %d\n",
       size, threads, processors);
  if (threads > processors)
    {
      printf
     ("Warning: %d threads requested, will run_omp on %d
processors available\n",
      threads, processors);
      omp_set_num_threads (threads);
    }
  int max_threads = omp_get_max_threads ();  // Max available
threads
  if (threads > max_threads)  // Requested threads are more than
max available
    {
```

```
        printf ("Error: Cannot use %d threads, only %d threads
available\n",
             threads, max_threads);
        return 1;
      }
    // Array allocation
    int *a = (int*)malloc (sizeof (int) * size);
    int *temp =(int *) malloc (sizeof (int) * size);
    if (a == NULL || temp == NULL)
      {
        printf ("Error: Could not allocate array of size %d\n",
size);
        return 1;
      }
    // Random array initialization
    int i;
    srand (314159);
    for (i = 0; i < size; i++)
      {
        a[i] = rand () % size;
      }
    // Sort
    double start = get_time ();
    run_omp (a, size, temp, threads);
    double end = get_time ();
    printf ("Start = %.2f\nEnd = %.2f\nElapsed = %.2f\n",
         start, end, end - start);
    // Result check
    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]);
         return 1;
       }
      }
    puts ("-Success-");
    return 0;
  }

  // Driver
  void
  run_omp (int a[], int size, int temp[], int threads)
  {
    // Enable nested parallelism, if available
    omp_set_nested (1);
    // Parallel mergesort
    mergesort_parallel_omp (a, size, temp, threads);
  }
```

```
// OpenMP merge sort with given number of threads
void
mergesort_parallel_omp (int a[], int size, int temp[], int
threads)
{
  if (threads == 1)
    {
//        printf("Thread %d begins serial merge sort\n",
omp_get_thread_num());
      mergesort_serial (a, size, temp);
    }
  else if (threads > 1)
    {
#pragma omp parallel sections
      {
//                      printf("Thread %d begins recursive
section\n", omp_get_thread_num());
#pragma omp section
       {              //printf("Thread %d begins recursive
call\n", omp_get_thread_num());
         mergesort_parallel_omp (a, size / 2, temp, threads / 2);
       }
#pragma omp section
       {              //printf("Thread %d begins recursive
call\n", omp_get_thread_num());
         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
      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
  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
  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
  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;
    }
}
```

# MergeSort parallelized using MPI

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <mpi.h>
#if _POSIX_TIMERS
#include <time.h>
#ifdef CLOCK_MONOTONIC_RAW
#define SYS_RT_CLOCK_ID CLOCK_MONOTONIC_RAW
#else
#define SYS_RT_CLOCK_ID CLOCK_MONOTONIC
#endif

double get_time(void)
{
    struct timespec ts;
    double t;
    if (clock_gettime(SYS_RT_CLOCK_ID, &ts) != 0)
    {
        perror("clock_gettime");
        abort();
    }
    t = (double)ts.tv_sec + (double)ts.tv_nsec * 1.0e-9;
    return t;
}

#else /* !_POSIX_TIMERS */
#include <sys/time.h>
double get_time(void)
{
    struct timeval tv;
    double t;
    if (gettimeofday(&tv, NULL) != 0)
    {
        perror("gettimeofday");
        abort();
    }
    t = (double)tv.tv_sec + (double)tv.tv_usec * 1.0e-6;
    return t;
}
#endif
```

```
#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 my_topmost_level_mpi (int my_rank);
void run_root_mpi (int a[], int size, int temp[], int max_rank,
int tag,MPI_Comm comm);
void run_helper_mpi (int my_rank, int max_rank,int tag,
MPI_Commcomm);
int main (int argc, char *argv[]);

int main (int argc, char *argv[])
{
  // All processes
  MPI_Init (&argc, &argv);
  // Check processes and their ranks
  // number of processes == communicator size
  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;
  // Set test data
  if (my_rank == 0)
    {                    // Only root process sets test data
      puts ("-MPI Recursive Mergesort-\t");
      // Check arguments
      if (argc != 2){
       printf ("Usage: %s array-size\n", argv[0]);
       MPI_Abort (MPI_COMM_WORLD, 1);
     }
      // Get argument
      int size = atoi (argv[1]);   // Array size
      printf ("Array size = %d\nProcesses = %d\n", size,
     comm_size);
      // 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
      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);
      double end = get_time ();
      printf ("Start = %.2f\nEnd = %.2f\nElapsed = %.2f\n",
           start, end, end - start);
      // Result check
      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
    {                    // Helper processes
      run_helper_mpi (my_rank, max_rank, tag, MPI_COMM_WORLD);
    }
  fflush (stdout);
  MPI_Finalize ();
  return 0;
}
```

```
// Root process code
void run_root_mpi (int a[], int size, int temp[], int max_rank,
int tag,MPI_Comm comm)
{
  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);
  /* level=0; my_rank=root_rank=0; */
  return;
}

// Helper process code
void run_helper_mpi (int my_rank, int max_rank, int tag,
MPI_Comm comm)
{
  int level = my_topmost_level_mpi (my_rank);
  // 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);
  int *temp = malloc (sizeof (int) * size);
  MPI_Recv (a, size, MPI_INT, parent_rank, tag, comm, &status);
  mergesort_parallel_mpi (a, size, temp, level, my_rank,
max_rank, tag, comm);
  // Send sorted array to parent process
  MPI_Send (a, size, MPI_INT, parent_rank, tag, comm);
  return;
}
```

```
int my_topmost_level_mpi (int my_rank)
{
  int level = 0;
  while (pow (2, level) <= my_rank)
    level++;
  return level;
}

// MPI merge sort
void mergesort_parallel_mpi (int a[], int size, int temp[],
               int level, int my_rank, int max_rank,
               int tag, MPI_Comm comm)
{
  int helper_rank = my_rank + pow (2, level);
  if (helper_rank > max_rank)
    {                    // no more processes available
      mergesort_serial (a, size, temp);
    }
  else
    {
//printf("Process %d has helper %d\n", my_rank, helper_rank);
      MPI_Request request;
      MPI_Status status;
      // Send second half, asynchronous
      MPI_Isend (a + size / 2, size - size / 2, MPI_INT,
 helper_rank, tag,
           comm, &request);
      // Sort first half
      mergesort_parallel_mpi (a, size / 2, temp, level + 1,
my_rank, max_rank,tag, comm);
      MPI_Request_free (&request);
      MPI_Recv (a + size / 2, size - size / 2, MPI_INT,
 helper_rank, tag,comm, &status);
      // Merge the two sorted sub-arrays through temp
      merge (a, size, temp);
    }
  return;
}

void mergesort_serial (int a[], int size, int temp[])
{
  // Switch to insertion sort for small arrays
  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
  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
  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;
    }
}
```

## MergeSort Using Hybrid (MPI + OpenMP)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <mpi.h>
#include <omp.h>
#if _POSIX_TIMERS
#include <time.h>
#ifdef CLOCK_MONOTONIC_RAW
/* System clock id passed to clock_gettime. CLOCK_MONOTONIC_RAW
   is preferred.  It has been available in the Linux kernel
   since version 2.6.28 */
#define SYS_RT_CLOCK_ID CLOCK_MONOTONIC_RAW
#else
#define SYS_RT_CLOCK_ID CLOCK_MONOTONIC
#endif

double
get_time(void)
{
    struct timespec ts;
    double t;
    if (clock_gettime(SYS_RT_CLOCK_ID, &ts) != 0)
    {
        perror("clock_gettime");
        abort();
    }
    t = (double)ts.tv_sec + (double)ts.tv_nsec * 1.0e-9;
    return t;
}

#else /* !_POSIX_TIMERS */
#include <sys/time.h>

double
get_time(void)
{
    struct timeval tv;
    double t;
    if (gettimeofday(&tv, NULL) != 0)
    {
        perror("gettimeofday");
        abort();
    }
    t = (double)tv.tv_sec + (double)tv.tv_usec * 1.0e-6;
    return t;
}

#endif
```

```
// Arrays size <= SMALL switches to insertion sort
#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
  omp_set_nested (1);
  // Check processes and their ranks
  // number of processes == communicator size
  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! */
    {
      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
  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
  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
      if (omp_get_nested () != 1)
         {
             puts ("Warning: Nested parallelism desired but
unavailable");
         }
      // Array allocation
      int *a = (int *)malloc (sizeof (int) * size);
      int *temp = (int *)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
      srand (314158);
      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);

   }                // Root process end
  else
    {                     // Node processes
      run_node_mpi (my_rank, max_rank, tag, MPI_COMM_WORLD, threads);
    }
  //fflush (stdout);
  MPI_Finalize ();
  return 0;
```

```
}

// Root process code
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, threads);     // level=0; my_rank=root_rank=0
  return;
}

// Node process code
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 = (int *) 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 process tree
int topmost_level_mpi (int my_rank)
{
  int level = 0;
  while (pow (2, level) <= my_rank)
    level++;
  return level;
}

// MPI merge sort
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
      MPI_Isend (a + size / 2, size - size / 2, MPI_INT, helper_rank,
tag,comm, &request);
      // Sort first half with OpenMP
      // 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
      merge (a, size, temp);
    }
  return;
}

// OpenMP merge sort with given number of threads
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
      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
  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
  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
  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;
    }
}
```

# 5. Results

The execution of the above program gives the following output:

Mergesort - Serial execution:

```
pratz@LAPTOP-R3NCRLFU: /mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ g++ -lrt serial_mergesort.c
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 1000000
-Serial Recursive Mergesort-
Array size = 1000000
Start = 2633.10
End = 2633.24
Elapsed = 0.15
-Success-
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 10000000
-Serial Recursive Mergesort-
Array size = 10000000
Start = 2638.01
End = 2639.65
Elapsed = 1.64
-Success-
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 100000000
-Serial Recursive Mergesort-
Array size = 100000000
Start = 2653.31
End = 2702.49
Elapsed = 49.17
-Success-
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$
```

MergeSort - Openmp execution:

```
pratz@LAPTOP-R3NCRLFU: /mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ g++ -lrt omp_mergesort.c -fopenmp
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 100000 2
-OpenMP Recursive Mergesort-
Array size = 100000
Processes = 2
Processors = 8
Start = 1609340349.23
End = 1609340349.24
Elapsed = 0.01
-Success-
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 1000000 2
-OpenMP Recursive Mergesort-
Array size = 1000000
Processes = 2
Processors = 8
Start = 1609340353.59
End = 1609340353.67
Elapsed = 0.08
-Success-
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 10000000 2
-OpenMP Recursive Mergesort-
Array size = 10000000
Processes = 2
Processors = 8
Start = 1609340357.12
End = 1609340358.02
Elapsed = 0.90
-Success-
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 100000000 2
-OpenMP Recursive Mergesort-
Array size = 100000000
Processes = 2
Processors = 8
Start = 1609340373.65
End = 1609340402.53
Elapsed = 28.89
-Success-
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$
```

Mergesort - MPI execution:

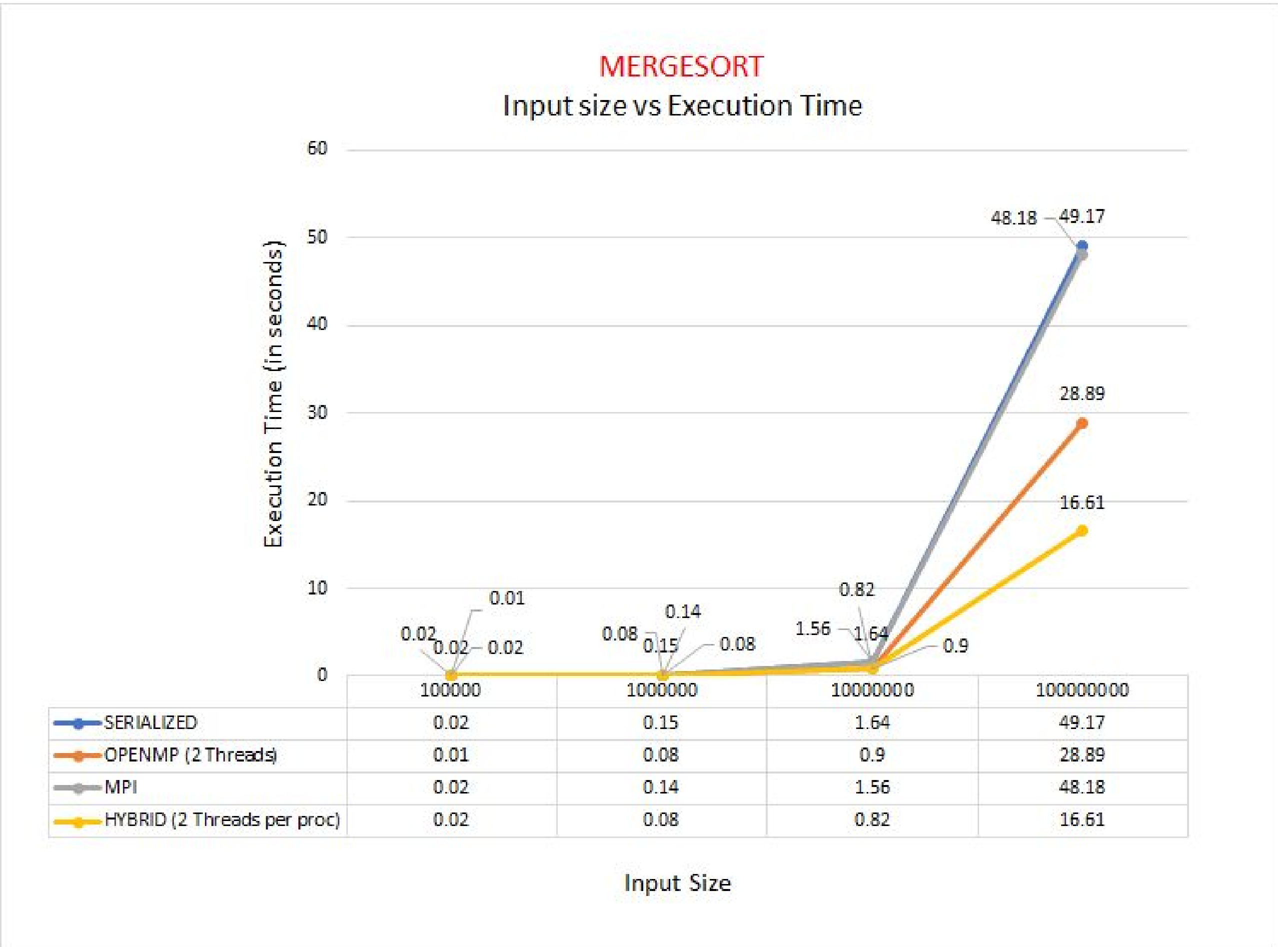
```
pratz@LAPTOP-R3NCRLFU: /mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ mpicc -lrt mpi_mergesort.c -lm
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 100000
-MPI Recursive Mergesort-
Array size = 100000
Processes = 1
Start = 1609340566.56
End = 1609340566.58
Elapsed = 0.02
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 1000000
-MPI Recursive Mergesort-
Array size = 1000000
Processes = 1
Start = 1609340573.86
End = 1609340574.01
Elapsed = 0.14
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 10000000
-MPI Recursive Mergesort-
Array size = 10000000
Processes = 1
Start = 1609340585.16
End = 1609340586.71
Elapsed = 1.56
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 100000000
-MPI Recursive Mergesort-
Array size = 100000000
Processes = 1
Start = 1609340604.21
End = 1609340652.39
Elapsed = 48.18
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$
```

Mergesort - MPI + OpenMP (Hybrid) execution:

```
pratz@LAPTOP-R3NCRLFU: /mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ mpicc -lrt hybrid_mergesort.c -lm -fopenmp
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 100000 2
-Multilevel parallel Recursive Mergesort with MPI and OpenMP-
Array size = 100000
Processes = 1
Threads per process = 2
Start = 1609341049.05
End = 1609341049.07
Elapsed = 0.02
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 1000000 2
-Multilevel parallel Recursive Mergesort with MPI and OpenMP-
Array size = 1000000
Processes = 1
Threads per process = 2
Start = 1609341053.72
End = 1609341053.79
Elapsed = 0.08
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 10000000 2
-Multilevel parallel Recursive Mergesort with MPI and OpenMP-
Array size = 10000000
Processes = 1
Threads per process = 2
Start = 1609341058.96
End = 1609341059.78
Elapsed = 0.82
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 100000000 2
-Multilevel parallel Recursive Mergesort with MPI and OpenMP-
Array size = 100000000
Processes = 1
Threads per process = 2
Start = 1609341065.94
End = 1609341095.12
Elapsed = 29.18
pratz@LAPTOP-R3NCRLFU:/mnt/c/Users/Pratheeksha Prasad/Desktop/Pratz/PADP/mergesort$ ./a.out 100000000 4
-Multilevel parallel Recursive Mergesort with MPI and OpenMP-
Array size = 100000000
Processes = 1
Threads per process = 4
Start = 1609341116.52
End = 1609341133.14
Elapsed = 16.61
```

The tabulation of execution times obtained by varying the input sizes with respect to different serial and parallel programming models is shown below:

| INPUT SIZE | SERIALIZED | OPENMP (2 Threads) | MPI | HYBRID (2 Threads per proc) |
|---|---|---|---|---|
| 100000 | 0.02 | 0.01 | 0.02 | 0.02 |
| 1000000 | 0.15 | 0.08 | 0.14 | 0.08 |
| 10000000 | 1.64 | 0.90 | 1.56 | 0.82 |
| 100000000 | 49.17 | 28.89 | 48.18 | 16.61 |




**Observation** - the hybrid model starts to perform better as the input size and the no of core counts increases.

# 6. Conclusion

Hybrid programming is an attempt to maximize the best from both OpenMP and MPI paradigms. However, it does not always imply better performance.This may be due to a number of reasons. OpenMP has less scalability due to implicit parallelism while MPI allows multi-dimensional blocking. All threads are idle except one while MPI communication. There is a thread creation overhead. The chances of cache miss problems increase due to data placement problems and larger dataset. Pure OpenMP performs slower than pure MPI within a node due to lack of optimized OpenMP libraries. Hybrid programs tend to work better when the communication to computation ratio is high.

Converting pure MPI codes into Hybrid codes is not a particularly difficult task. It is worth a try because if the nature of the problem is such that it can allow for faster hybrid codes, the speedups can be huge. Currently, there are very few benchmarked results for hybrid programs, and all of them are very problem specific.

# 7. References


1. Kushal Kedia,**Hybrid Programming with OpenMP and MPI,** MIT press, 2019
2. Rolf Rabenseifner, Georg Hager,Gabriele Jost, **Hybrid MPI and OpenMP Parallel Programming,** https://www.openmp.org//wp-content/uploads/HybridPP_Slides.pdf
3. https://www.cac.cornell.edu/education/Training/Intro/Hybrid-090529.pdf
4. https://docs.rc.fas.harvard.edu/kb/hybrid-mpiopenmp-codes-on-odyssey/