High Performance Computing

Practical File

(COCSC18)

Gulshan Kumar

2019UCO1667

Q1. **Write a parallel program to print “Hello World" using MPI.**

Code:

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
   MPI_Init(NULL, NULL);

   int size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   int rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   char processor_name[MPI_MAX_PROCESSOR_NAME];
   int name_len;
   MPI_Get_processor_name(processor_name, &name_len);

   printf("Hello world from processor %s, rank %d out of %d processors\n",
        processor_name, rank, size);

   MPI_Finalize();
}
```

Output:

```
gulshan@DESKTOP-JDS4256:/mnt/c/Users/welcome/Desktop$ mpirun -np 10 ./hello
Hello world from processor DESKTOP-JDS4256, rank 0 out of 10 processors
Hello world from processor DESKTOP-JDS4256, rank 4 out of 10 processors
Hello world from processor DESKTOP-JDS4256, rank 2 out of 10 processors
Hello world from processor DESKTOP-JDS4256, rank 1 out of 10 processors
Hello world from processor DESKTOP-JDS4256, rank 3 out of 10 processors
Hello world from processor DESKTOP-JDS4256, rank 9 out of 10 processors
Hello world from processor DESKTOP-JDS4256, rank 6 out of 10 processors
Hello world from processor DESKTOP-JDS4256, rank 7 out of 10 processors
Hello world from processor DESKTOP-JDS4256, rank 8 out of 10 processors
Hello world from processor DESKTOP-JDS4256, rank 5 out of 10 processors
```

Q2. **Write a program to illustrate basic MPI communication routines Code**

Code:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv)
{

   MPI_Init(NULL, NULL);

   int size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   int rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   char processor_name[MPI_MAX_PROCESSOR_NAME];
   int name_len;

   MPI_Get_processor_name(processor_name, &name_len);
```

```
    printf("Hello world from process %s, rank %d out of %d
processes\n\n",processor_name, rank, size);

    if (rank == 0)
    {
        char *msg = "Hello World!";
        MPI_Send(msg, 12, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
    else
    {
        char msg[12];
        MPI_Recv(msg, 12, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        printf("message received!\n");
        printf("message is : %s\n", msg);
    }

    MPI_Finalize();
    return 0;
}
```

Output:

```
gulshan@DESKTOP-JDS4256:/mnt/c/Users/welcome/Desktop$ mpirun -np 2 ./q2
Hello world from process DESKTOP-JDS4256, rank 0 out of 2 processes

Hello world from process DESKTOP-JDS4256, rank 1 out of 2 processes

message received!
message is : Hello World!DESKTOP-JDS4256
```

Q3. **Write a parallel program to find Sum of an array using MPI.**

Code:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define N 10

int arr[] = {12,24,3,67,45,23,102,34,15,46};
int temp[1000];

int main(int argc, char *argv[])
{

   int pid, num_processor,ele_per_process,num_ele_recieved;

   MPI_Status status;

   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &pid);
   MPI_Comm_size(MPI_COMM_WORLD, &num_processor);

   if (pid == 0)
   {
      int idx, i;
      ele_per_process = N / num_processor;

      if (num_processor > 1)
      {
         for (i = 1; i < num_processor - 1; i++)
         {
            idx = i * ele_per_process;

            MPI_Send(&ele_per_process,1, MPI_INT, i, 0,MPI_COMM_WORLD);
            MPI_Send(&arr[idx],ele_per_process,MPI_INT, i,
0,MPI_COMM_WORLD);
         }
```

```
    }

            idx = i * ele_per_process;
            int elements_left = N - idx;

            MPI_Send(&elements_left,1, MPI_INT,i, 0,MPI_COMM_WORLD);
            MPI_Send(&arr[idx],elements_left,MPI_INT, i, 0,MPI_COMM_WORLD);
        }
        int sum = 0;
        for (i = 0; i < ele_per_process; i++)
            sum += arr[i];
        int tmp;
        for (i = 1; i < num_processor; i++)
        {
            MPI_Recv(&tmp, 1, MPI_INT,MPI_ANY_SOURCE,
0,MPI_COMM_WORLD,&status);
            int sender = status.MPI_SOURCE;

            sum += tmp;
        }

        printf("Sum of array is : %d\n", sum);
    }
    else
    {
        MPI_Recv(&num_ele_recieved,1, MPI_INT, 0,
0,MPI_COMM_WORLD,&status);

        MPI_Recv(&temp, num_ele_recieved,MPI_INT, 0,
0,MPI_COMM_WORLD,&status);

        int partial_sum = 0;
        for (int i = 0; i < num_ele_recieved; i++)
            partial_sum += temp[i];

        MPI_Send(&partial_sum, 1, MPI_INT,
                0, 0, MPI_COMM_WORLD);
    }
```

```
    MPI_Finalize();

    return 0;
}
```

Output:

```
gulshan@DESKTOP-JDS4256:/mnt/c/Users/welcome/Desktop$ mpirun -np 10 ./q3
Sum of array is : 371
```

Q4 Write a C program for parallel implementation of Matrix Multiplication using MPI.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <mpi.h>
#define MAT_SIZE 5
int main(int argc, char *argv[])
{
    int np, pid;
    MPI_Status status;

    int A[MAT_SIZE][MAT_SIZE] = {{2, 4, 1,7,2}, {6, 0, 0,9,2}, {3, -12, 6,-1,5}, {6,
0,0,12,6}, {6, 0, 0,7,-23}};

    int B[MAT_SIZE][MAT_SIZE] = {{5, 5, 8,8,10}, {6, 2, 4, 12, 5}, {3, 5, 7, 0, 1},
{6,0,0,8,3}, {6, 0, 0,0,0}};
    int C[MAT_SIZE][MAT_SIZE];

    MPI_Init(&argc, &argv);
```

```
   MPI_Comm_rank(MPI_COMM_WORLD, &pid);
   MPI_Comm_size(MPI_COMM_WORLD, &np);
   if (pid == 0)
   {
      int num_rows_per_processor = MAT_SIZE / np;
      for (int i = 1; i < np - 1; i++)
      {
         int index = i * num_rows_per_processor;
         printf("Processor 0: Sending rows %d to %d to processor %d\n",index,
index + num_rows_per_processor - 1, i);

         MPI_Send(&index, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
         MPI_Send(&num_rows_per_processor, 1, MPI_INT, i,
0,MPI_COMM_WORLD);
         MPI_Send(&A[index][0], num_rows_per_processor * MAT_SIZE,MPI_INT, i,
0, MPI_COMM_WORLD);
         MPI_Send(&B[0][0], MAT_SIZE * MAT_SIZE, MPI_INT, i,
0,MPI_COMM_WORLD);
      }
      int index = (np - 1) * num_rows_per_processor;
      int num_rows_sent = MAT_SIZE - index;
      printf("Processor 0: Sending rows %d to %d to processor %d\n",index,index +
num_rows_sent - 1, np - 1);

      MPI_Send(&index, 1, MPI_INT, np - 1, 0, MPI_COMM_WORLD);
      MPI_Send(&num_rows_sent, 1, MPI_INT, np - 1, 0, MPI_COMM_WORLD);
      MPI_Send(&A[index][0], num_rows_sent * MAT_SIZE, MPI_INT, np - 1,
0,MPI_COMM_WORLD);
      MPI_Send(&B[0][0], MAT_SIZE * MAT_SIZE, MPI_INT, np - 1,
0,MPI_COMM_WORLD);

      for (int r = 0; r < num_rows_per_processor; r++)
      {
         for (int c = 0; c < MAT_SIZE; c++)
         {
            C[r][c] = 0;
            for (int k = 0; k < MAT_SIZE; k++)
```

```
          {
             C[r][c] += A[r][k] * B[k][c];
          }
       }
    }
    for (int i = 1; i < np; i++)
    {
       int index, num_rows;
       MPI_Recv(&index, 1, MPI_INT, i, 2, MPI_COMM_WORLD, &status);
       MPI_Recv(&num_rows, 1, MPI_INT, i, 2, MPI_COMM_WORLD, &status);
       MPI_Recv(&C[index][0], num_rows * MAT_SIZE, MPI_INT, i,
2,MPI_COMM_WORLD, &status);
       printf("Processor 0: Received answer from processor
%d\n",status.MPI_SOURCE);
    }

    for (int i = 0; i < MAT_SIZE; i++)
    {
       for (int j = 0; j < MAT_SIZE; j++)
       {
          printf("%d ", C[i][j]);
       }
       printf("\n");
    }
 }
 else
 {
    int num_rows, index;
    MPI_Recv(&index, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(&num_rows, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(&A, num_rows * MAT_SIZE, MPI_INT, 0, 0,
MPI_COMM_WORLD,&status);
    MPI_Recv(&B, MAT_SIZE * MAT_SIZE, MPI_INT, 0, 0,
MPI_COMM_WORLD,&status);
    printf("Processor %d: Received rows %d to %d from processor 0\n",pid,
index, index + num_rows - 1);
    for (int r = 0; r < num_rows; r++)
```

```
      {
        for (int c = 0; c < MAT_SIZE; c++)
        {
          C[r][c] = 0;
          for (int k = 0; k < MAT_SIZE; k++)
          {
            C[r][c] += A[r][k] * B[k][c];
          }
        }
      }
      printf("Processor %d: sending answer to processor 0\n", pid);
      MPI_Send(&index, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
      MPI_Send(&num_rows, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
      MPI_Send(&C, num_rows * MAT_SIZE, MPI_INT, 0, 2, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

Output:

```
gulshan@DESKTOP-JDS4256:/mnt/c/Users/welcome/Desktop$ mpirun -np 5 ./q4
Processor 0: Sending rows 1 to 1 to processor 1
Processor 0: Sending rows 2 to 2 to processor 2
Processor 0: Sending rows 3 to 3 to processor 3
Processor 0: Sending rows 4 to 4 to processor 4
Processor 1: Received rows 1 to 1 from processor 0
Processor 1: sending answer to processor 0
Processor 2: Received rows 2 to 2 from processor 0
Processor 2: sending answer to processor 0
Processor 3: Received rows 3 to 3 from processor 0
Processor 3: sending answer to processor 0
Processor 4: Received rows 4 to 4 from processor 0
Processor 4: sending answer to processor 0
Processor 0: Received answer from processor 1
Processor 0: Received answer from processor 2
Processor 0: Received answer from processor 3
Processor 0: Received answer from processor 4
91 23 39 120 62
96 30 48 120 87
-15 21 18 -128 -27
138 30 48 144 96
-66 30 48 104 81
```

Q5 Write a multithreaded program to generate Fibonacci series using pThreads.

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int n;
int *fibseq;
int i;

void *runn(void *arg);

int main(int argc, char *argv[])
```

```
{
   if (argc != 2)
   {
      printf("format is:./a.out <intgervalue>\n");
      return -1;
   }

   if (atoi(argv[1]) < 0)
   {
      printf("%d must be>=0\n", atoi(argv[1]));
      return -1;
   }

   n = atoi(argv[1]);
   fibseq = (int *)malloc(n * sizeof(int));
   pthread_t *threads = (pthread_t *)malloc(n * sizeof(pthread_t));
   pthread_attr_t attr;

   pthread_attr_init(&attr);

   for (i = 0; i < n; i++)
   {
      pthread_create(&threads[i], &attr, runn, NULL);
      pthread_join(threads[i], NULL);
   }


   printf("The Fibonacci sequence.:");
   int k;

   for (k = 0; k < n; k++)
   {
      printf("%d,", fibseq[k]);
   }
   return 0;
}
void *runn(void *arg)
```

```
{
   if (i == 0)
   {
      fibseq[i] = 0;
      pthread_exit(0);
   }

   if (i == 1)
   {
      fibseq[i] = 1;
      pthread_exit(0);
   }
   else
   {
      fibseq[i] = fibseq[i - 1] + fibseq[i - 2];

      pthread_exit(0);
   }
}
```

Output:

```
The Fibonacci sequence.:0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,
```

Q6 Write a program to implement Process Synchronization by mutex locks using pThreads.

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
void *fun1();
void *fun2();
```

```
int shared = 1;
pthread_mutex_t l;

int main()
{
   pthread_mutex_init(&l, NULL);
   pthread_t thread1, thread2;

   pthread_create(&thread1, NULL, fun1, NULL);
   pthread_create(&thread2, NULL, fun2, NULL);

   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

   printf("Final value of shared variable is %d\n", shared);
}

void *fun1()
{
   int x;
   printf("Thread-1 trying to acquire lock\n");

   pthread_mutex_lock(&l);
   printf("Thread-1 acquired lock\n");

   x = shared;
   printf("Thread-1 reads the value of shared variable as %d\n", x);

   x++;
   printf("Local updation by Thread-1 : %d\n", x);
   sleep(1);

   shared = x;
   printf("Value of shared variable updated by Thread-1 is: %d\n", shared);

   pthread_mutex_unlock(&l);
   printf("Thread-1 released the lock\n");
```

```
}

void *fun2()
{
   int y;
   printf("Thread-2 trying to acquire lock\n");

   pthread_mutex_lock(&l);
   printf("Thread-2 acquired lock\n");

   y = shared;
   printf("Thread-2 reads the value as %d\n", y);

   y--;
   printf("Local updation by Thread-2: %d\n", y);
   sleep(1);

   shared = y;
   printf("Value of shared variable updated by Thread-2 is: %d\n", shared);
   pthread_mutex_unlock(&l);
   printf("Thread-2 released the lock\n");
}
```

Output:

```
Thread-1 trying to acquire lock
Thread-1 acquired lock
Thread-1 reads the value of shared variable as 1
Local updation by Thread-1 : 2
Thread-2 trying to acquire lock
Value of shared variable updated by Thread-1 is: 2
Thread-1 released the lock
Thread-2 acquired lock
Thread-2 reads the value as 2
Local updation by Thread-2: 1
Value of shared variable updated by Thread-2 is: 1
Thread-2 released the lock
Final value of shared variable is 1
```

Q7 Write a C program to demonstrate multitask using OpenMP.

Code:

```
#include <stdlib.h>

#include <stdio.h>

#include <math.h>


#include <time.h>

#include <omp.h>


int main();

int *prime_table(int prime_num);

double *sine_table(int sine_num);
```

```
void timestamp();

int main()
{
    int prime_num;
    int *primes;
    int sine_num;
    double *sines;
    double wtime;
    double wtime1;
    double wtime2;

    timestamp();
    printf("\n");

    prime_num = 20000;
    sine_num = 20000;

    wtime = omp_get_wtime();

    #pragma omp parallel shared(prime_num, primes, sine_num, sines)
    {
        #pragma omp sections
        {
```

```
        #pragma omp section
        {
            wtime1 = omp_get_wtime();
            primes = prime_table(prime_num);
            wtime1 = omp_get_wtime() - wtime1;
        }
        #pragma omp section
        {
            wtime2 = omp_get_wtime();
            sines = sine_table(sine_num);
            wtime2 = omp_get_wtime() - wtime2;
        }
    }
}
wtime = omp_get_wtime() - wtime;

printf("Number of primes computed was %d\n", prime_num);
printf("Last prime was %d\n", primes[prime_num - 1]);
printf("Number of sines computed was %d\n", sine_num);
printf("Last sine computed was %g\n", sines[sine_num - 1]);
printf("\n");
printf("Elapsed time = %g\n", wtime);
printf("Task 1 time = %g\n", wtime1);
printf("Task 2 time = %g\n", wtime2);
```

```
    free(primes);
    free(sines);

    printf("\n");
    printf("Normal end of execution.\n");
    printf("\n");
    timestamp();

    return 0;
}

int *prime_table(int prime_num)

{
    int i;
    int j;
    int p;
    int prime;
    int *primes;
    int sum = 0;

    for(int i=0;i<prime_num;i++){
        sum+=i;
```

```
        printf("Curent sum %d for iteration %d\n ",sum,i);

    }
    return primes;
}

double *sine_table(int sine_num)
{
    double a;
    int i;
    int j;
    double pi = 3.141592653589793;
    double *sines;

    long long int prod = 1;

    for(int i=1;i<=sine_num;i++){
        prod= prod*i;
        printf("Current prod %lld for iteration %d\n",prod,i);
    }
    return sines;
}

void timestamp()
```

```
{
#define TIME_SIZE 40

    static char time_buffer[TIME_SIZE];
    const struct tm *tm;
    time_t now;

    now = time(NULL);
    tm = localtime(&now);

    strftime(time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm);

    printf("%s\n", time_buffer);

    return;
#undef TIME_SIZE
}
```

Output:

```
 Curent sum 198174186 for iteration 19908
 Curent sum 198194095 for iteration 19909
 Curent sum 198214005 for iteration 19910
 Curent sum 198233916 for iteration 19911
 Curent sum 198253828 for iteration 19912
 Curent sum 198273741 for iteration 19913
 Curent sum 198293655 for iteration 19914
 Current prod 0 for iteration 18922
Current prod 0 for iteration 18923
Current prod 0 for iteration 18924
Current prod 0 for iteration 18925
Current prod 0 for iteration 18926
Current prod 0 for iteration 18927
Curent sum 198313570 for iteration 19915
 Curent sum 198333486 for iteration 19916
 Current prod 0 for iteration 18928
Current prod 0 for iteration 18929
Current prod 0 for iteration 18930
Current prod 0 for iteration 18931
Current prod 0 for iteration 18932
```

Q8 Write a C program to demonstrate default, static and dynamic loop scheduling using OpenMP.

Code:

```c
#include <omp.h>
 #include <stdio.h>
 #include <stdlib.h>
 int main()
 {
    int i, N = 10, THREAD_COUNT = 3, CHUNK_SIZE = 3;
    printf("Default Scheduling\n");
 #pragma omp parallel for num_threads(THREAD_COUNT)
```

```
    for (i = 0; i < N; i++)
        printf("ThreadID: %d, iteration: %d\n", omp_get_thread_num(), i);
    printf("\nStatic Scheduling\n");
#pragma
omp parallel for num_threads(THREAD_COUNT) schedule(static, CHUNK_SIZE)
    for (i = 0; i < N; i++)
        printf("ThreadID: %d, iteration: %d\n", omp_get_thread_num(), i);
    printf("\nDynamic Scheduling\n");
#pragma
omp parallel for num_threads(THREAD_COUNT) schedule(dynamic, CHUNK_SIZE)
    for (i = 0; i < N; i++)
        printf("ThreadID: %d, iteration: %d\n", omp_get_thread_num(), i);
    return 0;
}
```

Output:

```
Number of processors available = 4
Number of threads = 4

                          Default        Static       Dynamic
       N     Pi(N)           Time          Time          Time

       1         0       0.000220      0.000002      0.000002
       2         1       0.000013      0.000001      0.000002
       4         2       0.000001      0.000001      0.000002
       8         4       0.000001      0.000001      0.000002
      16         6       0.000001      0.000002      0.000002
      32        11       0.000002      0.000003      0.000003
      64        18       0.000003      0.000006      0.000006
     128        31       0.000006      0.000009      0.000010
     256        54       0.000018      0.000020      0.000051
     512        97       0.000069      0.000053      0.000053
    1024       172       0.000227      0.000158      0.000158
    2048       309       0.000821      0.000500      0.000504
    4096       564       0.003051      0.005700      0.002100
    8192      1028       0.010348      0.005495      0.004402
   16384      1900       0.022729      0.016660      0.014216
   32768      3512       0.065383      0.050545      0.049570
   65536      6542       0.272042      0.188187      0.194251
  131072     12251       1.016691      0.699084      0.720553

Normal end of execution.
```