Day9

Table of Contents

1. Scripts

1.1. compile script

```
#!/bin/sh

#source /opt/ohpc/pub/apps/spack/share/spack/setup-env.sh
#spack load gcc/5i5y5cb
#spack load openmpi/c7kvqyq
source ~/git/spack/share/spack/setup-env.sh
spack load openmpi
```

```
inputFile=$1
outputFile="${1%.*}.out"      # extract the name of the file without extension and adding extension .out
#cmd=`mpicc $inputFile -o $outputFile`
cmd="mpicc $inputFile -o $outputFile -lm"     # running code using MPI
echo "------------------------------------------------------------------"
echo "Command executed: $cmd"
echo "------------------------------------------------------------------"
$cmd

echo "Compilation successful. Check at $outputFile"
echo "------------------------------------------------------------------"
```

1.2. run script

```
#!/bin/sh

#source /opt/ohpc/pub/apps/spack/share/spack/setup-env.sh
#spack load gcc/5i5y5cbc
source ~/git/spack/share/spack/setup-env.sh
spack load openmpi

cmd="mpirun -np $2 $1"
echo "------------------------------------------------------------------"
echo "Command executed: $cmd"
echo "------------------------------------------------------------------"
echo "##################################################################"
echo "##########                    OUTPUT                    ##########"
echo "##################################################################"
echo
mpirun -np $2 $1
echo
echo "##################################################################"
echo "##########                     DONE                     ##########"
echo "##################################################################"
```

2. Serial Matrix Addition

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

```
int main(int argc, char **argv){
    int i, j, myid, size, n = 400;
    int **m1, **m2, **sumMat;
    m1 = (int**)malloc(sizeof(int*) * n);
    m2 = (int**)malloc(sizeof(int*) * n);
    sumMat = (int**)malloc(sizeof(int*) * n);
    for(i = 0; i < n; i++){
        m1[i] = (int*)malloc(sizeof(int) * n);
        m2[i] = (int*)malloc(sizeof(int) * n);
        for(j = 0; j < n; j++){
            m1[i][j] = 1;
            m2[i][j] = 1;
        }
    }
    /*
      //you can also flaten this matrix and stored it in new array for later computation
    int arr[n * n], arr1[n * n];
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
           arr[i * n + j] = m1[i][j];
           arr1[i * n + j] = m2[i][j];
        }
    }*/
    /*
    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            printf("%d ",m1[i][j]);
        }
        printf("\n");
    }
    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            printf("%d ",m2[i][j]);
        }
        printf("\n");
    }*/

    for(i = 0; i < n; i++){
        sumMat[i] = (int*)malloc(sizeof(int) * n);
        for(j = 0; j < n; j++){
            sumMat[i][j] = m1[i][j] + m2[i][j];
        }
    }
    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            printf("%d ",sumMat[i][j]);
```

```
        }
        printf("\n");
    }

    return 0;
}
```

```
bash compile.sh serial_mat_add.c
```

```
------------------------------------------------------------------
Command executed: mpicc serial_mat_add.c -o serial_mat_add.out -lm
------------------------------------------------------------------
Compilation successful. Check at serial_mat_add.out
------------------------------------------------------------------
```

```
bash run.sh ./serial_mat_add.out 10 > output.txt
```

3. Parallel Matrix Addition Using MPI_Scatter and MPI_Gather

This example demonstrates how to parallelize a simple matrix addition code using `MPI_Scatter` and `MPI_Gather`. Each process computes the sum for a portion of the matrix and sends the results back to the root process.

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

int main() {
    int i, j, rank, size, n = 10000;
    int *m1, *m2, *sumMat, *sub_m1, *sub_m2, *sub_sumMat;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int chunksize = (n * n) / size;

    // Allocate memory for the full matrices on the root process
```

```
if (rank == 0) {
    m1 = (int*)malloc(n * n * sizeof(int));
    m2 = (int*)malloc(n * n * sizeof(int));
    sumMat = (int*)malloc(n * n * sizeof(int));
    for (i = 0; i < n * n; i++) {
        m1[i] = 1;
        m2[i] = 1;
    }
}

// Allocate memory for the submatrices on each process
sub_m1 = (int*)malloc(chunksize * sizeof(int));
sub_m2 = (int*)malloc(chunksize * sizeof(int));
sub_sumMat = (int*)malloc(chunksize * sizeof(int));

double startTime = MPI_Wtime();
// Scatter the elements of the matrices to all processes
MPI_Scatter(m1, chunksize, MPI_INT, sub_m1, chunksize, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatter(m2, chunksize, MPI_INT, sub_m2, chunksize, MPI_INT, 0, MPI_COMM_WORLD);

// Perform the addition on the submatrices
for (i = 0; i < chunksize; i++) {
    sub_sumMat[i] = sub_m1[i] + sub_m2[i];
}

// Gather the results from all processes
MPI_Gather(sub_sumMat, chunksize, MPI_INT, sumMat, chunksize, MPI_INT, 0, MPI_COMM_WORLD);

double endTime = MPI_Wtime();
// Print the result on the root process
if (rank == 0) {
    int flag = 1;
    for (i = 0; i < n * n; i++) {
        if (sumMat[i] != 2) {
            flag = 0;
            break;
        }
    }
    if (flag){
      printf("_____PASS_____\n");
      printf("Execution time: %lf\n", endTime - startTime);
    }
    else printf("_____FAIL_____\n");
    // Free the allocated memory
    free(m1);
    free(m2);
    free(sumMat);
```

```
    }
    free(sub_m1);
    free(sub_m2);
    free(sub_sumMat);
    MPI_Finalize();
    return 0;
}
```

- Compile

```
bash compile.sh mpi_matrix_addition1.c
```

```
---------------------------------------------------------------------
Command executed: mpicc mpi_matrix_addition1.c -o mpi_matrix_addition1.out -lm
---------------------------------------------------------------------
Compilation successful. Check at mpi_matrix_addition1.out
---------------------------------------------------------------------
```

- Run

```
bash run.sh ./mpi_matrix_addition1.out 10
```

```
---------------------------------------------------------------------
Command executed: mpirun -np 10 ./mpi_matrix_addition1.out
---------------------------------------------------------------------
######################################################################
##########                    OUTPUT                    ##########
######################################################################

_____PASS_____
Execution time: 1.178934

######################################################################
##########                    DONE                      ##########
######################################################################
```

4. Serial Matrix Multiplication

This example demonstrates a simple serial matrix multiplication code.

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

int main() {
    int n = 400;
    int i, j, k;

    // Allocate memory for matrices
    int **A = (int **)malloc(n * sizeof(int *));
    int **B = (int **)malloc(n * sizeof(int *));
    int **C = (int **)malloc(n * sizeof(int *));
    for (i = 0; i < n; i++) {
        A[i] = (int *)malloc(n * sizeof(int));
        B[i] = (int *)malloc(n * sizeof(int));
        C[i] = (int *)malloc(n * sizeof(int));
    }

    // Initialize matrices
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            A[i][j] = 1;
            B[i][j] = 1;
            C[i][j] = 0;
        }
    }

    // Matrix multiplication
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            for (k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    // Print result
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    // Free allocated memory
```

```
    for (i = 0; i < n; i++) {
        free(A[i]);
        free(B[i]);
        free(C[i]);
    }
    free(A);
    free(B);
    free(C);

    return 0;
}
```

```
bash compile.sh serial_matrix_multiplication.c
```

```
-----------------------------------------------------------------
Command executed: mpicc serial_matrix_multiplication.c -o serial_matrix_multiplication.out -lm
-----------------------------------------------------------------
Compilation successful. Check at serial_matrix_multiplication.out
-----------------------------------------------------------------
```

```
bash run.sh ./serial_matrix_multiplication.out 10 > output.txt
```

5. Parallel Matrix Multiplication Using MPI

This example demonstrates parallel matrix multiplication using `MPI_Scatter` and `MPI_Gather`.

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

int main(int argc, char **argv) {
    int i, j, k, rank, size, n = 400;
    int *A, *B, *C, *sub_A, *sub_C;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int chunksize = n * n / size;
```

```
// Allocate memory for matrices on the root process
if (rank == 0) {
    A = (int*)malloc(n * n * sizeof(int));
    B = (int*)malloc(n * n * sizeof(int));
    C = (int*)malloc(n * n * sizeof(int));
    for (i = 0; i < n * n; i++) {
        A[i] = 1;
        B[i] = 1;
        C[i] = 0;
    }
} else {
    B = (int*)malloc(n * n * sizeof(int));
}

// Allocate memory for submatrices
sub_A = (int*)malloc(chunksize * sizeof(int));
sub_C = (int*)malloc(chunksize * sizeof(int));
for (i = 0; i < chunksize; i++) {
    sub_C[i] = 0;
}

// Broadcast matrix B to all processes
MPI_Bcast(B, n * n, MPI_INT, 0, MPI_COMM_WORLD);

// Scatter the rows of matrix A to all processes
MPI_Scatter(A, chunksize, MPI_INT, sub_A, chunksize, MPI_INT, 0, MPI_COMM_WORLD);

// Perform the multiplication on the submatrices
for (i = 0; i < chunksize / n; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            sub_C[i * n + j] += sub_A[i * n + k] * B[k * n + j];
        }
    }
}

// Gather the results from all processes
MPI_Gather(sub_C, chunksize, MPI_INT, C, chunksize, MPI_INT, 0, MPI_COMM_WORLD);

// Print the result on the root process
if (rank == 0) {
    int flag = 1;
    for (i = 0; i < n * n; i++) {
        if (C[i] != n) {
            flag = 0;
            break;
```

```
            }
        }
        if (flag) printf("_____PASS_____\n");
        else printf("_____FAIL_____\n");

        // Free allocated memory
        free(A);
        free(B);
        free(C);
    } else {
        free(B);
    }

    free(sub_A);
    free(sub_C);

    MPI_Finalize();
    return 0;
}
```

```
bash compile.sh parallel_matrix_multiplication.c
```

```
---------------------------------------------------------------------
Command executed: mpicc parallel_matrix_multiplication.c -o parallel_matrix_multiplication.out -lm
---------------------------------------------------------------------
Compilation successful. Check at parallel_matrix_multiplication.out
---------------------------------------------------------------------
```

```
bash run.sh ./parallel_matrix_multiplication.out 10
```

```
---------------------------------------------------------------------
Command executed: mpirun -np 10 ./parallel_matrix_multiplication.out
---------------------------------------------------------------------
#####################################################################
##########                    OUTPUT                     ##########
#####################################################################

_____PASS_____

#####################################################################
##########                    DONE                       ##########
```

```
################################################################
```

Explanation:

1. The program initializes the MPI environment and retrieves the rank and size of the processes.
2. Memory for the matrices is allocated, and matrices are initialized with 1's.
3. The matrix B is broadcasted to all processes to ensure each process has the full matrix B.
4. Matrix A is scattered among all processes so that each process receives a portion (submatrix).
5. Each process performs the multiplication on its portion of the matrix.
6. The resulting submatrices are gathered back into the full matrix C on the root process.
7. The root process verifies and prints the result, and all allocated memory is freed.

6. Alter Parallel Matrix Multiplication Using MPI

This example demonstrates parallel matrix multiplication using `MPI_Scatter` and `MPI_Gather`.

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

int main(int argc, char **argv) {
    int i, j, k, rank, size, n = 400;
    int *A, *B, *C, *sub_A, *sub_C;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int rows_per_process = n / size;
    int remainder = n % size;

    // Allocate memory for matrices on the root process
    if (rank == 0) {
        A = (int*)malloc(n * n * sizeof(int));
        B = (int*)malloc(n * n * sizeof(int));
        C = (int*)malloc(n * n * sizeof(int));
        for (i = 0; i < n * n; i++) {
            A[i] = 1;
            B[i] = 1;
            C[i] = 0;
```

```
    }
} else {
    B = (int*)malloc(n * n * sizeof(int));
}

// Allocate memory for submatrices
int sub_n = rows_per_process + (rank < remainder ? 1 : 0);
sub_A = (int*)malloc(sub_n * n * sizeof(int));
sub_C = (int*)malloc(sub_n * n * sizeof(int));
for (i = 0; i < sub_n * n; i++) {
    sub_C[i] = 0;
}

// Broadcast matrix B to all processes
MPI_Bcast(B, n * n, MPI_INT, 0, MPI_COMM_WORLD);

// Scatter the rows of matrix A to all processes
int *sendcounts = (int*)malloc(size * sizeof(int));
int *displs = (int*)malloc(size * sizeof(int));
int offset = 0;
for (i = 0; i < size; i++) {
    sendcounts[i] = (rows_per_process + (i < remainder ? 1 : 0)) * n;
    displs[i] = offset;
    offset += sendcounts[i];
}
MPI_Scatterv(A, sendcounts, displs, MPI_INT, sub_A, sendcounts[rank], MPI_INT, 0, MPI_COMM_WORLD);

// Perform the multiplication on the submatrices
for (i = 0; i < sub_n; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            sub_C[i * n + j] += sub_A[i * n + k] * B[k * n + j];
        }
    }
}

// Gather the results from all processes
MPI_Gatherv(sub_C, sendcounts[rank], MPI_INT, C, sendcounts, displs, MPI_INT, 0, MPI_COMM_WORLD);

// Print the result on the root process
if (rank == 0) {
    int flag = 1;
    for (i = 0; i < n * n; i++) {
        if (C[i] != n) {
            flag = 0;
            break;
        }
```

```
        }
        if (flag) printf("_____PASS_____\n");
        else printf("_____FAIL_____\n");

        // Free allocated memory
        free(A);
        free(B);
        free(C);
    } else {
        free(B);
    }

    free(sub_A);
    free(sub_C);
    free(sendcounts);
    free(displs);

    MPI_Finalize();
    return 0;
}
```

```
bash compile.sh parallel_matrix_multiplication1.c
```

```
----------------------------------------------------------------
Command executed: mpicc parallel_matrix_multiplication1.c -o parallel_matrix_multiplication1.out -lm
----------------------------------------------------------------
Compilation successful. Check at parallel_matrix_multiplication1.out
----------------------------------------------------------------
```

```
bash run.sh ./parallel_matrix_multiplication1.out 10
```

```
----------------------------------------------------------------
Command executed: mpirun -np 10 ./parallel_matrix_multiplication1.out
----------------------------------------------------------------
################################################################
##########                    OUTPUT                  ##########
################################################################

_____PASS_____

################################################################
```

```
#########                    DONE                    #########
##################################################################
```

Explanation:

1. The program initializes the MPI environment and retrieves the rank and size of the processes.
2. Memory for the matrices is allocated, and matrices are initialized with 1's.
3. The matrix B is broadcasted to all processes to ensure each process has the full matrix B.
4. Matrix A is scattered among all processes so that each process receives a portion (submatrix).
5. Each process performs the multiplication on its portion of the matrix.
6. The resulting submatrices are gathered back into the full matrix C on the root process.
7. The root process verifies and prints the result, and all allocated memory is freed.

7. MPI Initialization: MPI_Init vs. MPI_Init_thread

MPI provides two main functions to initialize the MPI environment: `MPI_Init` and `MPI_Init_thread`. The primary difference is that `MPI_Init_thread` allows you to specify the desired level of thread support.

7.1. Levels of Thread Support

- `MPI_THREAD_SINGLE`: Only one thread will execute.
- `MPI_THREAD_FUNNELED`: The process may be multi-threaded, but only the main thread will make MPI calls.
- `MPI_THREAD_SERIALIZED`: Multiple threads may make MPI calls, but only one at a time.
- `MPI_THREAD_MULTIPLE`: Multiple threads may make MPI calls with no restrictions.

7.2. MPI_Init Example

This example uses `MPI_Init` to initialize the MPI environment.

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

int main(int argc, char** argv) {
```

```
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Print off a hello world message
    printf("Hello world from processor %d out of %d processors\n", world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
    return 0;
}
```

7.3. Compilation and Execution (MPI_Init)

- Compile the program:

```
bash compile.sh mpi_init.c
```

```
----------------------------------------------------------------
Command executed: mpicc mpi_init.c -o mpi_init.out -lm
----------------------------------------------------------------
Compilation successful. Check at mpi_init.out
----------------------------------------------------------------
```

- Run the program:

```
  bash run.sh ./mpi_init.out 6
```

```
  ----------------------------------------------------------------
  Command executed: mpirun -np 6 ./mpi_init.out
  ----------------------------------------------------------------
```

```
##################################################################
##########                    OUTPUT                    ##########
##################################################################

Hello world from processor 1 out of 6 processors
Hello world from processor 3 out of 6 processors
Hello world from processor 4 out of 6 processors
Hello world from processor 2 out of 6 processors
Hello world from processor 5 out of 6 processors
Hello world from processor 0 out of 6 processors

##################################################################
##########                    DONE                      ##########
##################################################################
```

7.4. MPI_Init_thread Example

This example uses `MPI_Init_thread` to initialize the MPI environment with thread support.

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

int main(int argc, char** argv) {
    int provided;

    // Initialize the MPI environment with thread support
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);

    // Check the level of thread support provided
    if (provided !=  MPI_THREAD_MULTIPLE) {
        printf("MPI does not provide required thread support\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Print off a hello world message
```

```
    printf("Hello world from processor %d out of %d processors with thread support level %d\n", world_rank, world_size, provided);

    // Finalize the MPI environment.
    MPI_Finalize();
    return 0;
}
```

7.5. Compilation and Execution (MPI_Init_thread)

- Compile the program:

```
bash compile.sh mpi_init_thread.c
```

```
---------------------------------------------------------------
Command executed: mpicc mpi_init_thread.c -o mpi_init_thread.out -lm
---------------------------------------------------------------
Compilation successful. Check at mpi_init_thread.out
---------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./mpi_init_thread.out 5
```

```
  ---------------------------------------------------------------
  Command executed: mpirun -np 5 ./mpi_init_thread.out
  ---------------------------------------------------------------
  ###############################################################
  ##########                    OUTPUT                   ##########
  ###############################################################

  Hello world from processor 0 out of 5 processors with thread support level 3
  Hello world from processor 1 out of 5 processors with thread support level 3
  Hello world from processor 4 out of 5 processors with thread support level 3
  Hello world from processor 2 out of 5 processors with thread support level 3
  Hello world from processor 3 out of 5 processors with thread support level 3

  ###############################################################
  ##########                     DONE                    ##########
```

```
##################################################################
```

7.6. Summary

- `MPI_Init` is used for standard MPI initialization without considering threading.
- `MPI_Init_thread` allows the program to specify and check the level of thread support.
    - Important for applications that require multi-threading in conjunction with MPI.
    - Ensures that the required thread support is available.

8. Test1

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

#define NUM_THREADS 4

void *thread_function(void* arg) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int thread_id = *(int*)arg;
  printf("Thread %d in process %d: Hello World!\n", thread_id, rank);

  // Simulate some work done by the thread
  for (int i = 0; i < 100000; i++) {
    // Do some calculations or operations here
  }

  return NULL;
}

int main(int argc, char* argv[]) {
  int thread_provided;
  int provided = MPI_THREAD_SINGLE;
  int thread_level = MPI_Init_thread(&argc, &argv, provided, &thread_provided);

  // Check the level of thread support provided by MPI
  if (thread_level != MPI_SUCCESS) {
    printf("Error initializing MPI threads\n");
```

```
    return 1;
  }

  if (thread_provided != MPI_THREAD_MULTIPLE) {
    printf("Warning: MPI_THREAD_MULTIPLE requested but not provided\n");
  }

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (size != 4) {
    printf("This program requires exactly 4 processes\n");
    MPI_Finalize();
    return 1;
  }


  // Create threads within each process
  pthread_t threads[NUM_THREADS];
  int thread_ids[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++) {
    thread_ids[i] = i;
    pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]);
  }

  // Wait for all threads to finish
  for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }

  MPI_Finalize();

  return 0;
}
```

```
bash compile.sh test1.c
```

```
-----------------------------------------------------------------
Command executed: mpicc test1.c -o test1.out -lm
-----------------------------------------------------------------
Compilation successful. Check at test1.out
-----------------------------------------------------------------
```

```
bash run.sh ./test1.out 4
```

```
-----------------------------------------------------------------
Command executed: mpirun -np 4 ./test1.out
-----------------------------------------------------------------
#################################################################
##########                     OUTPUT                    ##########
#################################################################

Warning: MPI_THREAD_MULTIPLE requested but not provided
Thread 0 in process 1: Hello World!
Warning: MPI_THREAD_MULTIPLE requested but not provided
Warning: MPI_THREAD_MULTIPLE requested but not provided
Warning: MPI_THREAD_MULTIPLE requested but not provided
Thread 1 in process 1: Hello World!
Thread 0 in process 3: Hello World!
Thread 0 in process 2: Hello World!
Thread 2 in process 1: Hello World!
Thread 0 in process 0: Hello World!
Thread 3 in process 0: Hello World!
Thread 3 in process 1: Hello World!
Thread 1 in process 2: Hello World!
Thread 1 in process 3: Hello World!
Thread 2 in process 2: Hello World!
Thread 2 in process 3: Hello World!
Thread 1 in process 0: Hello World!
Thread 3 in process 2: Hello World!
Thread 3 in process 3: Hello World!
Thread 2 in process 0: Hello World!

#################################################################
##########                      DONE                     ##########
#################################################################
```

9. Test2

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <mpi.h>
#include <omp.h>

#define NUM_THREADS 6

int main(int argc, char* argv[]) {
  int thread_provided;
  int provided = MPI_THREAD_SINGLE;
  int thread_level = MPI_Init_thread(&argc, &argv, provided, &thread_provided);
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  // Check the level of thread support provided by MPI
  if (thread_level != MPI_SUCCESS) {
    printf("Error initializing MPI threads\n");
    return 1;
  }
  if (thread_provided != MPI_THREAD_MULTIPLE) {
    printf("Warning: MPI_THREAD_MULTIPLE requested but not provided\n");
  }
  #pragma omp parallel num_threads(NUM_THREADS)
  {
        printf("Hello openmp from thread %d inside process %d\n", omp_get_thread_num(), rank);
  }

  MPI_Finalize();
  return 0;
}
```

```
bash compile.sh test2.c
```

```
-----------------------------------------------------------------
Command executed: mpicc test2.c -o test2.out -lm
-----------------------------------------------------------------
Compilation successful. Check at test2.out
-----------------------------------------------------------------
```

```
bash run.sh ./test2.out 4
```

```
--------------------------------------------------------------
Command executed: mpirun -np 4 ./test2.out
--------------------------------------------------------------
##############################################################
##########                   OUTPUT                   ##########
##############################################################


##############################################################
##########                    DONE                    ##########
##############################################################
```

Author: Abhishek Raj
Created: 2024-07-13 Sat 08:04