Day19

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 -fopenmp"     # 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 Multiplication

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

int main() {
    int n = 1000;
    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;
        }
    }

    double starttime = omp_get_wtime();
    // 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];
```

```
            }
        }
    }
    double endtime = omp_get_wtime();

    // Print result
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }
    printf("execution time: %lf\n", endtime - starttime);

    // 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 -fopenmp
---------------------------------------------------------------------
Compilation successful. Check at serial_matrix_multiplication.out
---------------------------------------------------------------------
```

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

3. 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 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 -fopenmp
------------------------------------------------------------------
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.

4. OpenMP Tasking

4.1. Introduction to OpenMP Tasking

- OpenMP tasking is a powerful feature introduced to handle irregular and dynamic workloads.

- It allows the creation of tasks, which are units of work that can be executed independently.
- Tasks are distributed among threads for execution, enabling efficient parallelization of applications with unpredictable workloads.

4.2. Key Concepts

- **Task:**
  - A unit of work created using the `#pragma omp task` directive.
  - Contains code that can be executed independently.
- **Tasking Constructs:**
  - `#pragma omp task`
  - `#pragma omp taskwait`
  - `#pragma omp taskgroup`

4.3. When to Use Tasking

- Divide-and-conquer algorithms (e.g., quicksort, mergesort).
- Recursive computations.
- Workloads with dynamically varying tasks.
- Problems where work cannot be evenly divided in advance.

4.4. Task Directive: Syntax

```
#pragma omp task [clauses]
   structured-block
```

- **Clauses:**
  - `if(expression)`: Specifies whether the task should be created based on the condition.
  - `default(shared | none)`: Specifies variable sharing.
  - `private(list)`, `firstprivate(list)`, `shared(list)`: Data-sharing clauses.

4.5. Example 1: Simple Task Creation

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

void work(int id) {
    printf("Task %d is being executed by thread %d\n", id, omp_get_thread_num());
}


int main() {
    #pragma omp parallel num_threads(5)
    {
        #pragma omp single
        {
            printf("%d is creating the task\n", omp_get_thread_num());
            for (int i = 0; i < 5; i++) {
                #pragma omp task
                work(i);
            }
        }
    }
    return 0;
}
```

```
gcc task1.c -fopenmp
```

```
./a.out
```

```
1 is creating the task
Task 1 is being executed by thread 1
Task 0 is being executed by thread 0
Task 3 is being executed by thread 3
Task 2 is being executed by thread 1
Task 4 is being executed by thread 3
```

4.6. Explanation:

- The `single` construct ensures that only one thread creates tasks.
- Tasks are executed by any available thread in the team.

4.7. Taskwait Directive

- Ensures that all tasks created in the current context are completed before proceeding.
- Syntax:

```
#pragma omp taskwait
```

4.8. Example 2: Task Synchronization

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

void work(int id) {
    printf("Task %d is being executed by thread %d\n", id, omp_get_thread_num());
}

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < 5; i++) {
                #pragma omp task
                work(i);
            }
            #pragma omp taskwait
            printf("All tasks are completed.\n");
        }
    }
    return 0;
}
```

```
gcc task2.c -fopenmp
```

```
./a.out
```

```
Task 0 is being executed by thread 1
```

```
Task 2 is being executed by thread 9
Task 1 is being executed by thread 3
Task 3 is being executed by thread 4
Task 4 is being executed by thread 6
All tasks are completed.
```

4.9. Taskgroup Directive

- Groups tasks together for synchronization.
- Ensures that all tasks in the group are completed before proceeding.
- Syntax:

```
#pragma omp taskgroup
   structured-block
```

4.10. Example 3: Using Taskgroup

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

void work(int id) {
    printf("Task %d is being executed by thread %d\n", id, omp_get_thread_num());
}

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp taskgroup
            {
                for (int i = 0; i < 5; i++) {
                    #pragma omp task
                    work(i);
                }
            }
            printf("All tasks in the group are completed.\n");
        }
    }
    return 0;
```

```
}
```

```
gcc task3.c -fopenmp
```

```
./a.out
```

```
Task 4 is being executed by thread 11
Task 2 is being executed by thread 4
Task 3 is being executed by thread 3
Task 0 is being executed by thread 5
Task 1 is being executed by thread 6
All tasks in the group are completed.
```

4.11. Advanced Features

- **Task Dependencies:**
  - Allows you to specify dependencies between tasks using the `depend` clause.
  - Syntax:

```
#pragma omp task depend(dependency-type : list)
   structured-block
```

- **Dependency Types:**
  - `in`: Task depends on the data being available.
  - `out`: Task produces data required by another task.
  - `inout`: Task both consumes and produces data.

4.12. Example 4: Task Dependencies

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

int main() {
    int data = 0;
```

```
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task depend(out: data)
            {
                data = 42;
                printf("Task 1: Produced data = %d\n", data);
            }

            #pragma omp task depend(in: data)
            {
                printf("Task 2: Consumed data = %d\n", data);
            }
        }
    }
    return 0;
}
```

4.13. Best Practices

- Use `if` clauses to limit task creation overhead for small tasks.
- Combine tasks with `taskgroup` for efficient synchronization.
- Use `depend` clauses for precise dependency management.
- Avoid excessive task creation to reduce runtime overhead.

5. test

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

#define N 10000

void sum(int* arr, int start, int end, int* result) {
    int sum = 0;
    for (int i = start; i < end; i++) {
        sum += arr[i];
    }
    *result = sum;
}
```

```c
void totalSum(int* result1, int* result2, int* total) {
    *total = *result1 + *result2;
}

int main() {
    int* arr = (int*) malloc(N * sizeof(int));
    int result1 = 0, result2 = 0, total = 0;

    // Initialize the array
    for (int i = 0; i < N; i++) {
        arr[i] = i + 1;
    }

    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            sum(arr, 0, N/2, &result1);

            #pragma omp task
            sum(arr, N/2, N, &result2);

            #pragma omp taskwait

            #pragma omp task
            totalSum(&result1, &result2, &total);
        }

    }

    printf("Total sum: %d\n", total);

    free(arr);
    return 0;
}
```

```
gcc test.c -fopenmp
```

```
./a.out
```

```
Total sum: 50005000
```

6. test2

```c
/* Try to read and analyze the code and also change some of the parameters
 * according to your needs. I have also added comments to make you aware of my
 * thought process while doing the code.*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define N 10000

//function to calculate sum
void sum(int* arr, int start, int end, int* result) {
    int sum = 0;
    for (int i = start; i < end; i++) {
        sum += arr[i];
    }
    *result = sum;
}

//function to calculate totalSum
void totalSum(int* result, int size, int* total) {
    int sum = 0;
    for(int i = 0; i < size; i++){
        sum+= result[i];
    }
    *total = sum;
}

int main() {
    omp_set_num_threads(5); //setting total number of threads
    int* arr = (int*) malloc(N * sizeof(int));  //creating and allocating array
    int *result, total = 0;
    int start = 0, end = 0;

    //allocating spaces for resultant sum array
    //I want to store the sum by each task at a specific thread index
    //Here size of resultant array will be equal to total_no_of_threads
    //because each thread will do task of calculating there some and store
    //it in there location which will be result[threadId]
```

```
result = (int*) malloc(omp_get_num_threads() * sizeof(int));

// Initialize the array
for (int i = 0; i < N; i++) {
    arr[i] = i + 1;
}

int chunksize = 0;
#pragma omp parallel
{
    //here chunksize will be equal to N / total number of threads
    chunksize = N / omp_get_num_threads();
    #pragma omp single
    {
        for(int i = 0; i < omp_get_num_threads(); i++){
            //first task will start from 0 to chunksize
            //second task will start from 1 * chunsize to its (start + chunksize)
            start = i * chunksize;
            if(i == omp_get_num_threads() - 1){
                //if your thread is last thread then we want to give all the remaining
                //iterations to last threads if there's any reminder threads
                end = N;
            }
            else{
                end = start + chunksize;
            }
            //creating tasks here and storing the result in result[i]
            #pragma omp task
            sum(arr, start, end, &result[i]);
        }
        //taskwait for synchronization
        //try to remove taskwait and analyze the result
        //your code more likely to be involved in race condition
        #pragma omp taskwait
        //task for final sum calculation
        //below I used omp_get_num_threads to give the total size of result array
        //which in my case will be equal to total number of threads
        //bcz I created tasks equal to total number of threads
        #pragma omp task
        totalSum(result, omp_get_num_threads(), &total);
    }

}

//printing total sum by tasking and by natural number sum formula
printf("Total sum by tasking: %d\n", total);
printf("Total sum by formula: %ld\n", ((N * 1L) * (N + 1)) / 2);
```

```
    //resources deallocation
    free(arr);
    free(result);
    return 0;
}
```

```
gcc test2.c -fopenmp
```

```
./a.out
```

```
Total sum by tasking: 50005000
Total sum by formula: 50005000
```

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 rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Print off a hello world message
    printf("Hello world from processor %d out of %d processors\n", 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 -fopenmp
----------------------------------------------------------------
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 2 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 5 out of 6 processors
Hello world from processor 0 out of 6 processors
Hello world from processor 1 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_FUNNELED, &provided);

    // Check the level of thread support provided
    if (provided !=  MPI_THREAD_FUNNELED) {
        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 rank;
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Print off a hello world message
    printf("Hello world from processor %d out of %d processors with thread support level %d\n", 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 -fopenmp
----------------------------------------------------------------
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 1 out of 5 processors with thread support level 1
  Hello world from processor 0 out of 5 processors with thread support level 1
  Hello world from processor 3 out of 5 processors with thread support level 1
  Hello world from processor 2 out of 5 processors with thread support level 1
  Hello world from processor 4 out of 5 processors with thread support level 1
```

```
################################################################
##########                    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.

7.7. hybrid

```c
#include <mpi.h>
#include <omp.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 rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);


    #pragma omp parallel num_threads(5)
    {
```

```
        printf("thread %d inside rank %d\n", omp_get_thread_num(), rank);
    }

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

7.8. Compilation and Execution (MPI_Init_thread)

- Compile the program:

```
bash compile.sh h1.c
```

```
---------------------------------------------------------------------
Command executed: mpicc h1.c -o h1.out -lm -fopenmp
---------------------------------------------------------------------
Compilation successful. Check at h1.out
---------------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./h1.out 5
```

```
  ---------------------------------------------------------------------
  Command executed: mpirun -np 5 ./h1.out
  ---------------------------------------------------------------------
  ######################################################################
  ##########                    OUTPUT                    ##########
  ######################################################################

  thread 1 inside rank 0
  thread 4 inside rank 0
  thread 2 inside rank 0
  thread 3 inside rank 0
  thread 0 inside rank 0
  thread 1 inside rank 1
  thread 4 inside rank 1
```

```
thread 2 inside rank 1
thread 3 inside rank 1
thread 0 inside rank 1
thread 4 inside rank 3
thread 1 inside rank 3
thread 2 inside rank 3
thread 3 inside rank 3
thread 0 inside rank 3
thread 4 inside rank 2
thread 1 inside rank 2
thread 2 inside rank 2
thread 3 inside rank 2
thread 0 inside rank 2
thread 4 inside rank 4
thread 1 inside rank 4
thread 2 inside rank 4
thread 3 inside rank 4
thread 0 inside rank 4

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

8. test3

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

int main(int argc, char** argv) {
    int rank, size;
    long long data_size = 1000000000;
    int num_threads = 10;
    long long *data = NULL;
    long long chunksize;
    long long *local_data = NULL;
    long long local_sum = 0;
    long long global_sum = 0;

    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
    if(provided != MPI_THREAD_FUNNELED){
        printf("MPI_THREAD_FUNNELED not supported\n");
        MPI_Finalize();
        return 0;
    }
    chunksize = data_size / size;

    if (rank == 0) {
        data = (long long *)malloc(data_size * sizeof(long long));
        for (long long i = 0; i < data_size; i++) {
            data[i] = i + 1;
        }
    }

    local_data = (long long *)malloc(chunksize * sizeof(long long ));

    MPI_Scatter(data, chunksize, MPI_LONG_LONG, local_data, chunksize, MPI_LONG_LONG, 0, MPI_COMM_WORLD);

    #pragma omp parallel for num_threads(num_threads) reduction(+:local_sum)
    for (long long i = 0; i < chunksize; i++) {
        local_sum += local_data[i];
    }

    MPI_Reduce(&local_sum, &global_sum, 1, MPI_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Final sum: %lld\n", global_sum);
    }

    if (rank == 0) {
        free(data);
    }
    free(local_data);
    MPI_Finalize();

    return 0;
}
```

- Compile the program:

```
bash compile.sh test3.c
```

```
------------------------------------------------------------------
Command executed: mpicc test3.c -o test3.out -lm -fopenmp
------------------------------------------------------------------
Compilation successful. Check at test3.out
------------------------------------------------------------------
```

- Run the program:

```
bash run.sh ./test3.out 10
```

```
  ------------------------------------------------------------------
  Command executed: mpirun -np 10 ./test3.out
  ------------------------------------------------------------------
  ##################################################################
  ##########                    OUTPUT                    ##########
  ##################################################################

  Final sum: 500000000500000000

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

Author: Abhishek Raj
Created: 2025-01-13 Mon 17:42