Day3

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"     # 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. MPI Array Sum Calculation Example

2.1. mpi_arraysum.c

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

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

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

    int n = 1000000; // Size of the array
    long *array = NULL;
    int chunk_size = n / size;
    long *sub_array = (long*)malloc(chunk_size * sizeof(long));

    if (rank == 0) {
```

```c
        array = (long*)malloc(n * sizeof(long));
        for (int i = 0; i < n; i++) {
            array[i] = i + 1; // Initialize the array with values 1 to n
        }

        // Distribute chunks of the array to other processes
        for (int i = 1; i < size; i++) {
            MPI_Send(array + i * chunk_size, chunk_size, MPI_LONG, i, 0, MPI_COMM_WORLD);
        }

        // Copy the first chunk to sub_array
        for (int i = 0; i < chunk_size; i++) {
            sub_array[i] = array[i];
        }
    } else {
        // Receive chunk of the array
        MPI_Recv(sub_array, chunk_size, MPI_LONG, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    // Compute the local sum
    long local_sum = 0;
    for (int i = 0; i < chunk_size; i++) {
        local_sum += sub_array[i];
    }

    if (rank != 0) {
        // Send local sum to process 0
        MPI_Send(&local_sum, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD);
    } else {
        // Process 0 receives the local sums and computes the final sum
        long final_sum = local_sum;
        long temp_sum;
        for (int i = 1; i < size; i++) {
            MPI_Recv(&temp_sum, 1, MPI_LONG, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            final_sum += temp_sum;
        }
        printf("The total sum of array elements is %ld\n", final_sum);
    }

    free(sub_array);
    if (rank == 0) {
        free(array);
    }

    MPI_Finalize();
    return 0;
}
```

2.2. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_array_sum.c
```

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

- Run the program:

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

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

The total sum of array elements is 500000500000

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

3. MPI Array Sum Calculation with Timing

3.1. Introduction to MPI_{Wtime}

`$MPI_{Wtime}$` is a function in MPI that returns the elapsed wall-clock time in seconds since an arbitrary point in the past. It is used to measure the performance and execution time of parallel programs.

3.2. Syntax

```
double MPI_Wtime(void);
```

3.3. mpi_arraysumtimed.c

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

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

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

    int n = 1000000; // Size of the array
    long *array = NULL;
    int chunk_size = n / size;
    long *sub_array = (long*)malloc(chunk_size * sizeof(long));

    double start_time, end_time;

    if (rank == 0) {
        array = (long*)malloc(n * sizeof(long));
        for (int i = 0; i < n; i++) {
            array[i] = i + 1; // Initialize the array with values 1 to n
        }

        // Start timing the computation
        start_time = MPI_Wtime();

        // Distribute chunks of the array to other processes
        for (int i = 1; i < size; i++) {
            MPI_Send(array + i * chunk_size, chunk_size, MPI_LONG, i, 0, MPI_COMM_WORLD);
        }
```

```
        // Copy the first chunk to sub_array
        for (int i = 0; i < chunk_size; i++) {
            sub_array[i] = array[i];
        }
    } else {
        // Receive chunk of the array
        MPI_Recv(sub_array, chunk_size, MPI_LONG, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    // Compute the local sum
    long local_sum = 0;
    for (int i = 0; i < chunk_size; i++) {
        local_sum += sub_array[i];
    }

    if (rank != 0) {
        // Send local sum to process 0
        MPI_Send(&local_sum, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD);
    } else {
        // Process 0 receives the local sums and computes the final sum
        long final_sum = local_sum;
        long temp_sum;
        for (int i = 1; i < size; i++) {
            MPI_Recv(&temp_sum, 1, MPI_LONG, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            final_sum += temp_sum;
        }

        // Stop timing the computation
        end_time = MPI_Wtime();
        printf("The total sum of array elements is %ld\n", final_sum);
        printf("Time taken: %f seconds\n", end_time - start_time);

        free(array);
    }

    free(sub_array);

    MPI_Finalize();
    return 0;
}
```

3.4. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_array_sum_timed.c
```

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

- Run the program:

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

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

The total sum of array elements is 500000500000
Time taken: 0.010608 seconds

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

3.5. Explanation of Timing

- \`$MPI_{Wtime}$()\`: Returns the current time in seconds. It is called before and after the computation to measure the elapsed time.
- \`$start_{time}$ = $MPI_{Wtime}$();\`: Captures the start time before distributing the array.
- \`$end_{time}$ = $MPI_{Wtime}$();\`: Captures the end time after collecting the local sums and computing the final sum.
- \`printf("Time taken: %f seconds\n", $end_{time}$ - $start_{time}$);\`: Prints the total time taken for the computation.

This updated program measures the time taken to distribute the array, compute local sums, gather the results, and compute the final sum. The timing information helps in evaluating the performance of the parallel program.

4. MPI$_{Scatter}$

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

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

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int n = 10000; // Size of the array
    int *array = NULL;
    int chunk_size = n / world_size;
    int *sub_array = (int*)malloc(chunk_size * sizeof(int));

    if (world_rank == 0) {
        array = (int*)malloc(n * sizeof(int));
        for (int i = 0; i < n; i++) {
            array[i] = i + 1; // Initialize the array with values 1 to n
        }
    }

    // Scatter the chunks of the array to all processes
    MPI_Scatter(array, chunk_size, MPI_INT, sub_array, chunk_size, MPI_INT, 0, MPI_COMM_WORLD);

    // Compute the local sum
    int local_sum = 0;
    for (int i = 0; i < chunk_size; i++) {
        local_sum += sub_array[i];
    }

    // Gather all local sums to the root process
    int final_sum = 0;
    MPI_Reduce(&local_sum, &final_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

```c
    if (world_rank == 0) {
        printf("The total sum of array elements is %d\n", final_sum);
        free(array);
    }

    free(sub_array);

    MPI_Finalize();
    return 0;
}
```

```
bash compile.sh scatter.c
```

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

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

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

The total sum of array elements is 50005000

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

5. MPI Scatter Explanation and Example

5.1. Introduction to $MPI_{Scatter}$

$MPI_{Scatter}$ is a collective communication operation in MPI used to distribute distinct chunks of data from the root process to all processes in a communicator. Each process, including the root, receives a portion of the data.

5.2. Syntax

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
```

- `sendbuf`: Starting address of the send buffer (used only by the root process).
- `sendcount`: Number of elements sent to each process.
- `sendtype`: Data type of send buffer elements.
- `recvbuf`: Starting address of the receive buffer.
- `recvcount`: Number of elements in the receive buffer.
- `recvtype`: Data type of receive buffer elements.
- `root`: Rank of the root process.
- `comm`: Communicator.

5.3. Example Code Explanation

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

int main(int argc, char **argv) {
    int i, myid, size;
    int *sendBuf, recvBuf;
    MPI_Status status;

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

    if (myid == 0) {
        sendBuf = (int*)malloc(size * sizeof(int));
        for (i = 0; i < size; i++) {
            sendBuf[i] = 100 + i * 5 + i; // Initialize the send buffer with some values
```

```
        }
    }
    // Scatter the data from root process to all processes
    MPI_Scatter(sendBuf, 1, MPI_INT, &recvBuf, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (myid == 0) printf("Message broadcasted are: \n");
    printf("Process %d has %d\n", myid, recvBuf);

    if (myid == 0) free(sendBuf);

    MPI_Finalize();
    return 0;
}
```

5.4. Code Explanation

- `MPI$_{Init}$(&argc, &argv);`: Initializes the MPI execution environment.
- `MPI$_{Commsize}$(MPI$_{COMMWORLD}$, &size);`: Gets the number of processes.
- `MPI$_{Commrank}$(MPI$_{COMMWORLD}$, &myid);`: Gets the rank of the current process.

5.5. Root Process (myid == 0)

- `sendBuf = (int*)malloc(size * sizeof(int));`: Allocates memory for the send buffer.
- `for (i = 0; i < size; i++) { sendBuf[i] = 100 + i * 5 + i; }`: Initializes the send buffer with values.

5.6. All Processes

- `MPI$_{Scatter}$(sendBuf, 1, MPI$_{INT}$, &recvBuf, 1, MPI$_{INT}$, 0, MPI$_{COMMWORLD}$);`: Distributes one element of type `MPI$_{INT}$` from the send buffer of the root process to the receive buffer of each process.

5.7. Printing the Results

- Each process prints its received value.
- `if (myid == 0) free(sendBuf);`: Frees the allocated memory on the root process.

- `MPI_Finalize();`: Terminates the MPI execution environment.

5.8. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_scatter.c
```

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

- Run the program:

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

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

Message broadcasted are:
Process 0 has 100
Process 1 has 106
Process 8 has 148
Process 2 has 112
Process 3 has 118
Process 9 has 154
Process 4 has 124
Process 5 has 130
Process 6 has 136
Process 7 has 142

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

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

This example demonstrates how to use `MPI$_{Scatter}$` to distribute individual elements from an array on the root process to all processes in the communicator. Each process receives one element and prints it.

6. MPI Scatter and Reduce Example with long datatype

7. mpi$_{\text{scatterreducelong.c}}$

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

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

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    long n = 1000000; // Size of the array
    long *array = NULL;
    long chunk_size = n / world_size;
    long *sub_array = (long*)malloc(chunk_size * sizeof(long));

    if (world_rank == 0) {
        array = (long*)malloc(n * sizeof(long));
        for (long i = 0; i < n; i++) {
            array[i] = i + 1; // Initialize the array with values 1 to n
        }
    }

    // Scatter the chunks of the array to all processes
    MPI_Scatter(array, chunk_size, MPI_LONG, sub_array, chunk_size, MPI_LONG, 0, MPI_COMM_WORLD);

    // Compute the local sum
    long local_sum = 0;
    for (long i = 0; i < chunk_size; i++) {
```

```
        local_sum += sub_array[i];
    }

    // Gather all local sums to the root process
    long final_sum = 0;
    MPI_Reduce(&local_sum, &final_sum, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);

    if (world_rank == 0) {
        printf("The total sum of array elements is %ld\n", final_sum);
        free(array);
    }

    free(sub_array);

    MPI_Finalize();
    return 0;
}
```

7.1. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_scatter_reduce_long.c
```

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

- Run the program:

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

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

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

The total sum of array elements is 500000500000

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

In this example, the array is initialized with long integers and the `MPI$_{Scatter}$` function is used to distribute chunks of the array to all processes. Each process computes the local sum of its chunk and the `MPI$_{Reduce}$` function is used to gather the local sums and compute the final sum in the root process.

8. MPI Broadcast and Gather

8.1. MPI$_{Bcast}$ Example

8.1.1. mpi$_{bcastexample.c}$

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

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

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int data;
    if (rank == 0) {
        data = 100;  // Root process initializes the data
    }

    // Broadcast the data from the root process to all processes
    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Process %d received data %d\n", rank, data);
```

```
    MPI_Finalize();
    return 0;
}
```

8.1.2. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_bcast.c
```

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

- Run the program:

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

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

Process 0 received data 100
Process 2 received data 100
Process 4 received data 100
Process 1 received data 100
Process 3 received data 100

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

In this example, the integer `data` is initialized to 100 in the root process (process 0). The `MPI$_{Bcast}$` function is called to broadcast the value of `data` to all processes in the communicator. After the broadcast, each process prints the received value.

8.2. MPI$_{Gather}$ Example

8.2.1. mpi$_{gatherexample.c}$

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int send_data = rank; // Each process sends its rank
    int *recv_data = NULL;
    if (rank == 0) {
        recv_data = (int*)malloc(size * sizeof(int)); // Allocate memory for receiving data
    }
    // Gather the data from all processes to the root process
    MPI_Gather(&send_data, 1, MPI_INT, recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Gathered data at root process: ");
        for (int i = 0; i < size; i++) {
            printf("%d ", recv_data[i]);
        }
        printf("\n");
        free(recv_data);
    }
    MPI_Finalize();
    return 0;
}
```

8.2.2. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_gather.c
```

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

- Run the program:

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

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

Gathered data at root process: 0 1 2 3

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

In this example, each process sends its rank as `send`$_{data}$. The `MPI`$_{Gather}$ function is called to gather the values of `send`$_{data}$ from all processes to the `recv`$_{data}$ array in the root process. After gathering the data, the root process prints the gathered values.

8.3. Summary

- **`MPI`$_{Bcast}$**: Broadcasts data from the root process to all other processes in the communicator.
- **`MPI`$_{Gather}$**: Gathers data from all processes in the communicator and collects it at the root process.

These collective communication functions are essential for distributing data and collecting results in parallel programs using MPI. MPI Array Sum Calculation Example using $MPI_{Scatter}$

8.4. $mpi_{arraysumscatter.c}$

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

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

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

    int n = 100; // Size of the array
    int *array = NULL;
    int chunk_size = n / size;
    int *sub_array = (int*)malloc(chunk_size * sizeof(int));

    if (rank == 0) {
        array = (int*)malloc(n * sizeof(int));
        for (int i = 0; i < n; i++) {
            array[i] = i + 1; // Initialize the array with values 1 to n
        }
    }

    // Scatter the chunks of the array to all processes
    MPI_Scatter(array, chunk_size, MPI_INT, sub_array, chunk_size, MPI_INT, 0, MPI_COMM_WORLD);

    // Compute the local sum
    int local_sum = 0;
    for (int i = 0; i < chunk_size; i++) {
        local_sum += sub_array[i];
    }

    // Gather all local sums to the root process
    int final_sum = 0;
    MPI_Reduce(&local_sum, &final_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("The total sum of array elements is %d\n", final_sum);
```

```
        free(array);
    }

    free(sub_array);

    MPI_Finalize();
    return 0;
}
```

8.5. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_array_sum_scatter.c
```

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

- Run the program:

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

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

The total sum of array elements is 5050

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

9. test

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

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

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

    int n = 100000; // Size of the array
    long *array = NULL;
    int chunk_size = n / size;
    long *sub_array = (long*)malloc(chunk_size * sizeof(long));

    double start_time, end_time, total_time;
    double data_transfer_time = 0.0, computation_time = 0.0;

    if (rank == 0) {
        array = (long*)malloc(n * sizeof(long));
        for (int i = 0; i < n; i++) {
            array[i] = i + 1; // Initialize the array with values 1 to n
        }

        // Start timing the data transfer
        start_time = MPI_Wtime();

        // Distribute chunks of the array to other processes
        for (int i = 1; i < size; i++) {
            MPI_Send(array + i * chunk_size, chunk_size, MPI_LONG, i, 0, MPI_COMM_WORLD);
        }

        // Stop timing the data transfer
        end_time = MPI_Wtime();
        data_transfer_time = end_time - start_time;

        // Copy the first chunk to sub_array
        for (int i = 0; i < chunk_size; i++) {
            sub_array[i] = array[i];
        }
    } else {
```

```c
        // Start timing the data transfer
        start_time = MPI_Wtime();

        // Receive chunk of the array
        MPI_Recv(sub_array, chunk_size, MPI_LONG, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Stop timing the data transfer
        end_time = MPI_Wtime();
        data_transfer_time = end_time - start_time;
    }

    // Start timing the computation
    start_time = MPI_Wtime();

    // Compute the local sum
    long local_sum = 0;
    for (int i = 0; i < chunk_size; i++) {
        local_sum += sub_array[i];
    }

    // Stop timing the computation
    end_time = MPI_Wtime();
    computation_time = end_time - start_time;

    if (rank != 0) {
        // Send local sum to process 0
        MPI_Send(&local_sum, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD);
    } else {
        // Process 0 receives the local sums and computes the final sum
        long final_sum = local_sum;
        long temp_sum;

        // Start timing the data transfer for receiving local sums
        start_time = MPI_Wtime();

        for (int i = 1; i < size; i++) {
            MPI_Recv(&temp_sum, 1, MPI_LONG, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            final_sum += temp_sum;
        }

        // Stop timing the data transfer
        end_time = MPI_Wtime();
        data_transfer_time += end_time - start_time;

        printf("The total sum of array elements is %ld\n", final_sum);
        printf("Time taken for data transfer: %f seconds\n", data_transfer_time);
        printf("Time taken for computation on process 0: %f seconds\n", computation_time);
```

```
    }

    printf("Process %d time taken for local computation: %f seconds\n", rank, computation_time);

    free(sub_array);
    if (rank == 0) {
        free(array);
    }

    MPI_Finalize();
    return 0;
}
```

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

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

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

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

Process 1 time taken for local computation: 0.000044 seconds
Process 2 time taken for local computation: 0.000063 seconds
Process 3 time taken for local computation: 0.000063 seconds
The total sum of array elements is 5000050000
Time taken for data transfer: 0.000721 seconds
Time taken for computation on process 0: 0.000045 seconds
Process 0 time taken for local computation: 0.000045 seconds

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

10. Assignment

- PI calculator using MPI.
- Prime number calculator using MPI. (Calculate number of primes between 0 to N).

Author: Abhishek Raj
Created: 2024-07-04 Thu 14:39