Day8

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. MPI Groups and Communicators

In MPI, communicators and groups are essential for defining communication contexts and organizing processes. A communicator encapsulates a group of processes that can communicate with each other. Each process within a communicator has a unique rank, starting from 0.

2.1. Groups

A group is an ordered set of processes. Groups are used to define the members of a communicator. Groups are created from existing communicators and can be manipulated using various MPI functions.

2.2. Communicators

A communicator is a communication domain, and it is the primary context for MPI communication operations. The default communicator, `MPI_COMM_WORLD`, includes all the processes in an MPI program. You can create new communicators with different groups of processes.

2.3. Creating and Managing Groups and Communicators

- **Creating Groups**: You can create a new group from an existing communicator using `MPI_Comm_group`, which extracts the group from a communicator.
- **Creating Communicators**: You can create new communicators from existing groups using `MPI_Comm_create` or `MPI_Comm_split`.

2.4. Difference between Groups and Communicators

- **Groups**:
  - A group is a collection of processes identified by their ranks.
  - Groups do not have communication contexts.
  - Groups are used to create new communicators.
- **Communicators**:
  - A communicator includes a group and a communication context.
  - Communicators are used for performing communication operations.
  - The default communicator, `MPI_COMM_WORLD`, includes all processes.

2.5. MPI Syntax and Functions

2.5.1. MPI_Comm_split

This function splits an existing communicator into multiple, non-overlapping communicators based on the color and key values provided.

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm);
```

- **comm**: The original communicator.
- **color**: Determines the group to which a process belongs.
- **key**: Determines the rank within the new communicator.
- **newcomm**: The new communicator.

2.5.2. MPI_Comm_group

This function retrieves the group associated with a communicator.

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group);
```

- **comm**: The communicator.
- **group**: The group associated with the communicator.

2.5.3. MPI_Group_incl

This function creates a new group from a subset of processes in an existing group.

```
int MPI_Group_incl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup);
```

- **group**: The original group.
- **n**: Number of ranks in the new group.
- **ranks**: Array of ranks in the original group to include in the new group.
- **newgroup**: The new group.

2.5.4. MPI_Comm_create_group

This function creates a new communicator from a group.

```
int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm *newcomm);
```

- **comm**: The original communicator.
- **group**: The group defining the new communicator.
- **tag**: Tag for the new communicator.
- **newcomm**: The new communicator.

2.5.5. MPI_Group_free

This function deallocates a group.

```
int MPI_Group_free(MPI_Group *group);
```

- **group**: The group to be deallocated.

2.5.6. MPI_Comm_free

This function deallocates a communicator.

```
int MPI_Comm_free(MPI_Comm *comm);
```

- **comm**: The communicator to be deallocated.

2.6. Example: Creating and Using Groups and Communicators

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

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

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

    // Split the world group into two groups
    int color = world_rank % 2;  // Determine color based on rank
    MPI_Comm new_comm;
    MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &new_comm);

    // Get the new rank and size in the new communicator
    int new_rank, new_size;
    MPI_Comm_rank(new_comm, &new_rank);
    MPI_Comm_size(new_comm, &new_size);

    printf("World Rank: %d, New Rank: %d, New Size: %d\n", world_rank, new_rank, new_size);

    // Perform some communication within the new communicator
    int send_data = new_rank;
    int recv_data;
```

```
    MPI_Allreduce(&send_data, &recv_data, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    printf("World Rank: %d, New Comm Sum: %d\n", world_rank, recv_data);

    // Free the new communicator and group
    MPI_Comm_free(&new_comm);

    MPI_Finalize();
    return 0;
}
```

2.7. Explanation

1. **Extract World Group**:
   - `MPI_Comm_group` is used to get the group of `MPI_COMM_WORLD`.
2. **Split the Communicator**:
   - `MPI_Comm_split` is used to split `MPI_COMM_WORLD` into two new communicators based on the color value (rank modulo 2).
   - This creates two new communicators: one for even ranks and one for odd ranks.
3. **New Rank and Size**:
   - The new rank and size within the new communicator are obtained using `MPI_Comm_rank` and `MPI_Comm_size`.
4. **Communication**:
   - `MPI_Allreduce` is performed within the new communicator to compute the sum of ranks in the new communicator.
5. **Cleanup**:
   - The new communicator and group are freed using `MPI_Comm_free` and `MPI_Group_free`.

2.8. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_groups_communicators.c
```

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

```
------------------------------------------------------------------
```

- Run the program:

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

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

World Rank: 1, New Rank: 0, New Size: 5
World Rank: 1, New Comm Sum: 20
World Rank: 0, New Rank: 0, New Size: 5
World Rank: 0, New Comm Sum: 20
World Rank: 2, New Rank: 1, New Size: 5
World Rank: 2, New Comm Sum: 20
World Rank: 5, New Rank: 2, New Size: 5
World Rank: 5, New Comm Sum: 20
World Rank: 4, New Rank: 2, New Size: 5
World Rank: 4, New Comm Sum: 20
World Rank: 3, New Rank: 1, New Size: 5
World Rank: 3, New Comm Sum: 20
World Rank: 7, New Rank: 3, New Size: 5
World Rank: 7, New Comm Sum: 20
World Rank: 8, New Rank: 4, New Size: 5
World Rank: 8, New Comm Sum: 20
World Rank: 6, New Rank: 3, New Size: 5
World Rank: 6, New Comm Sum: 20
World Rank: 9, New Rank: 4, New Size: 5
World Rank: 9, New Comm Sum: 20

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

This example demonstrates how to create and use groups and communicators in MPI to organize and manage process communication in parallel applications.

3. Task Parallelism

When working with MPI, it's often necessary to divide tasks among different groups of processes. MPI provides various functions to create and manage groups and communicators.

3.1. Example: Task Parallelism with Groups and Communicators

This example demonstrates the use of the above functions to create two groups, assign communicators, and perform different tasks.

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

void perform_computation(int rank) {
    printf("Process %d performing computation\n", rank);
    // Simulate computation by sleeping for a while
    sleep(2);
}

void perform_io_operations(int rank) {
    printf("Process %d performing I/O operations\n", rank);
    // Simulate I/O by sleeping for a while
    sleep(3);
}

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

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

    // Define two groups: one for even ranks and one for odd ranks
    int half_size = world_size / 2;
    int *even_ranks = malloc(half_size * sizeof(int));
    int *odd_ranks = malloc((world_size - half_size) * sizeof(int));

    int even_count = 0, odd_count = 0;
    for (int i = 0; i < world_size; i++) {
        if (i % 2 == 0) {
```

```
            even_ranks[even_count++] = i;
        } else {
            odd_ranks[odd_count++] = i;
        }
    }

    // Create groups
    MPI_Group world_group, even_group, odd_group;
    MPI_Comm_group(MPI_COMM_WORLD, &world_group);
    MPI_Group_incl(world_group, even_count, even_ranks, &even_group);
    MPI_Group_incl(world_group, odd_count, odd_ranks, &odd_group);

    // Create new communicators
    MPI_Comm even_comm, odd_comm;
    MPI_Comm_create_group(MPI_COMM_WORLD, even_group, 0, &even_comm);
    MPI_Comm_create_group(MPI_COMM_WORLD, odd_group, 1, &odd_comm);

    // Perform tasks based on the group
    if (world_rank % 2 == 0 && even_comm != MPI_COMM_NULL) {
        perform_computation(world_rank);
    } else if (world_rank % 2 != 0 && odd_comm != MPI_COMM_NULL) {
        perform_io_operations(world_rank);
    }

    // Free the groups and communicators
    MPI_Group_free(&even_group);
    MPI_Group_free(&odd_group);
    if (even_comm != MPI_COMM_NULL) MPI_Comm_free(&even_comm);
    if (odd_comm != MPI_COMM_NULL) MPI_Comm_free(&odd_comm);
    MPI_Group_free(&world_group);

    free(even_ranks);
    free(odd_ranks);

    MPI_Finalize();
    return 0;
}
```

3.2. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_task_parallelism_manual_groups.c
```

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

- Run the program:

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

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

Process 6 performing computation
Process 8 performing computation
Process 0 performing computation
Process 2 performing computation
Process 4 performing computation
Process 1 performing I/O operations
Process 5 performing I/O operations
Process 3 performing I/O operations
Process 7 performing I/O operations
Process 9 performing I/O operations

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

3.3. Questions and Answers

3.3.1. Is there any way for two different groups to communicate with each other?

Yes, two different groups can communicate using an inter-communicator. `MPI_Intercomm_create` can be used to establish communication between two groups, allowing them to exchange messages.

3.3.2. What are the communication mechanisms in different groups and the same group?

- **Different Groups**:
  - **Inter-communicator**: Allows communication between different groups.
  - **Point-to-Point Communication**: Direct communication between processes in different groups using inter-communicators.
- **Same Group**:
  - **Intra-communicator**: Default communication within the same group using collective operations like `MPI_Bcast`, `MPI_Reduce`, etc.

3.3.3. Do the two groups have the same communicator?

No, each group will have its unique communicator. When groups are created from a world communicator, they each get a new communicator that allows them to operate independently. An inter-communicator is required for communication between these separate groups. ng Groups and Communicators Task parallelism focuses on distributing tasks (rather than data) across different processes. Each task can perform different operations, allowing for concurrent execution of multiple tasks. By using groups and communicators, you can organize processes to perform specific tasks independently.

3.4. Example: Task Parallelism

In this example, we divide processes into two groups: one for computing and one for I/O operations. Each group performs its respective task concurrently.

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

void perform_computation(int rank) {
    printf("Process %d performing computation\n", rank);
    // Simulate computation by sleeping for a while
    sleep(2);
}

void perform_io_operations(int rank) {
    printf("Process %d performing I/O operations\n", rank);
    // Simulate I/O by sleeping for a while
    sleep(3);
```

```
}

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

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


    // Split the world group into two groups based on rank
    int color = world_rank % 2;  // Determine color based on rank
    MPI_Comm new_comm;
    MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &new_comm);

    // Get the new rank and size in the new communicator
    int new_rank, new_size;
    MPI_Comm_rank(new_comm, &new_rank);
    MPI_Comm_size(new_comm, &new_size);

    if (color == 0) {
        perform_computation(world_rank);
    } else {
        perform_io_operations(world_rank);
    }

    // Free the new communicator and group
    MPI_Comm_free(&new_comm);
    MPI_Group_free(&world_group);

    MPI_Finalize();
    return 0;
}
```

3.5. Explanation

1. **Task Functions**:
    - `perform_computation` simulates a computation task.
    - `perform_io_operations` simulates an I/O task.
2. **Extract World Group**:
    - `MPI_Comm_group` is used to get the group of `MPI_COMM_WORLD`.
3. **Split the Communicator**:
    - `MPI_Comm_split` is used to split `MPI_COMM_WORLD` into two new communicators based on the

color value (rank modulo 2).
    - This creates two new communicators: one for even ranks (compute group) and one for odd ranks (I/O group).
4. **Perform Task**:
    - Each group performs its respective task concurrently based on the rank's color.
5. **Cleanup**:
    - The new communicator and group are freed using `MPI_Comm_free` and `MPI_Group_free`.

3.6. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_task_parallelism.c
```

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

- Run the program:

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

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

Process 1 performing I/O operations
Process 0 performing computation
Process 2 performing computation
Process 6 performing computation
Process 4 performing computation
Process 9 performing I/O operations
```

```
Process 7 performing I/O operations
Process 3 performing I/O operations
Process 8 performing computation
Process 5 performing I/O operations

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

3.7. Communication between Groups

1. **Different Groups**:
    - **Inter-communicator**: MPI provides `MPI_Intercomm_create` to create an inter-communicator that allows communication between two different groups.
    - **Point-to-Point Communication**: `MPI_Send` and `MPI_Recv` can be used for direct communication between processes in different groups using the inter-communicator.
2. **Same Group**:
    - **Intra-communicator**: The default communicator within the same group is an intra-communicator (like `MPI_COMM_WORLD`). All standard communication operations (e.g., `MPI_Bcast`, `MPI_Reduce`) work within the same group.

3.8. Summary

Dividing processes into groups and communicators helps in organizing and managing tasks effectively in MPI. Different groups can communicate using inter-communicators, while communication within the same group uses intra-communicators.

- **Different Groups Communication**:
    - Use inter-communicators (`MPI_Intercomm_create`) and point-to-point communication (`MPI_Send`, `MPI_Recv`).
- **Same Group Communication**:
    - Use intra-communicators like `MPI_COMM_WORLD` and collective operations.

Groups and communicators do not share the same communicator. Each group can have its unique communicator, allowing for flexible and organized communication in parallel applications.

4. Task1

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

long long sumOfSquares(long long *arr, int size){
    long long sum = 0;
    for(int i = 0; i < size; i++){
        sum+= arr[i] * arr[i];
    }
    return sum;
}

long long sum(long long *arr, int size){
    long long sum = 0;
    for(int i = 0; i < size; i++){
        sum+= arr[i];
    }
    return sum;
}


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

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

    // Split the world group into two groups
    int color = world_rank % 2;  // Determine color based on rank
    MPI_Comm new_comm;
    MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &new_comm);

    // Get the new rank and size in the new communicator
    int new_rank, new_size;
    MPI_Comm_rank(new_comm, &new_rank);
    MPI_Comm_size(new_comm, &new_size);

    //printf("World Rank: %d, New Rank: %d, New Size: %d\n", world_rank, new_rank, new_size);
    const int data_size = 10000;
    long long data[data_size];
    if(new_rank == 0){
        for(int i = 0; i < data_size; i++){
            data[i] = i + 1;
```

```
        }
    }
    //data broadcasted to each process in new_comm
    int chunk_size = data_size / new_size;
    long long local_array[chunk_size];
    long long local_sum = 0;
    long long local_square_sum = 0;
    MPI_Scatter(data, chunk_size, MPI_LONG_LONG, local_array, chunk_size, MPI_LONG_LONG, 0, new_comm);
    // Perform some communication within the new communicator
    if(color == 0){
        local_sum = sum(local_array, chunk_size);
    }
    if(color == 1){
        local_square_sum = sumOfSquares(local_array, chunk_size);
    }
    long long final_sum = 0;
    long long final_square_sum = 0;
    MPI_Allreduce(&local_sum, &final_sum, 1, MPI_LONG_LONG, MPI_SUM, new_comm);
    MPI_Allreduce(&local_square_sum, &final_square_sum, 1, MPI_LONG_LONG, MPI_SUM, new_comm);
    if(new_rank == 0){
        if(color == 0)
            printf("World Rank: %d, Sum of arrays: %lld\n", world_rank, final_sum);
        if(color == 1)
            printf("World Rank: %d, Sum of squares of arrays: %lld\n", world_rank, final_square_sum);
    }

    // Free the new communicator and group
    MPI_Comm_free(&new_comm);

    MPI_Finalize();
    return 0;
}
```

```
bash compile.sh task1.c
```

```
---------------------------------------------------------------------
Command executed: mpicc task1.c -o task1.out -lm
---------------------------------------------------------------------
Compilation successful. Check at task1.out
---------------------------------------------------------------------
```

```
bash run.sh ./task1.out 10
```

```
--------------------------------------------------------------------
Command executed: mpirun -np 10 ./task1.out
--------------------------------------------------------------------
######################################################################
##########                    OUTPUT                    ##########
######################################################################

World Rank: 0, Sum of arrays: 50005000
World Rank: 1, Sum of squares of arrays: 333383335000

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

5. PI calculator serial

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<sys/time.h>
#define N 999999999
int main()
{
    int i, j;
    double area, pi;
    double dx, y, x;
    double exe_time;
    struct timeval stop_time, start_time;
    dx = 1.0/N;
    x = 0.0;
    area = 0.0;
    gettimeofday(&start_time, NULL);
    for(i=0;i<N;i++){
        x = i*dx;
        y = sqrt(1-x*x);
        area += y*dx;
    }
    gettimeofday(&stop_time, NULL);
    exe_time = (stop_time.tv_sec+(stop_time.tv_usec/1000000.0)) - (start_time.tv_sec+(start_time.tv_usec/1000000.0));
    pi = 4.0*area;
    printf("\n Value of pi is = %.16lf\n Execution time is = %lf seconds\n", pi, exe_time);
    return 0;
}
```

- Compile the program:

```
gcc pi_serial.c -o pi_serial.out -lm
```

- Run the program:

```
./pi_serial.out
```

```
Value of pi is = 3.1415926555902138
Execution time is = 3.660779 seconds
```

6. Parallel Pi Computation Using MPI

This example demonstrates how to parallelize the computation of π using MPI. The interval `[0, 1]` is divided among multiple processes, and each process computes its partial sum. The partial sums are then reduced to compute the final value of π.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <sys/time.h>

#define N 999999999

int main(int argc, char** argv) {
    int rank, size, i;
    double dx, x, y, local_area, total_area;
    double start_time, end_time, execution_time;

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

    dx = 1.0 / N;
    local_area = 0.0;

    start_time = MPI_Wtime();

    for (i = rank; i < N; i += size) {
        x = i * dx;
        y = sqrt(1 - x * x);
        local_area += y * dx;
    }

    MPI_Reduce(&local_area, &total_area, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    end_time = MPI_Wtime();
    execution_time = end_time - start_time;

    if (rank == 0) {
        double pi = 4.0 * total_area;
        printf("\nValue of pi is = %.16lf\nExecution time is = %lf seconds\n", pi, execution_time);
    }
```

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

6.1. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_parallel_pi.c
```

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

- Run the program:

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

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


Value of pi is = 3.1415926555895561
Execution time is = 1.623518 seconds

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

6.2. Explanation

- **MPI Initialization**:
  - `MPI_Init`: Initializes the MPI execution environment.
  - `MPI_Comm_rank`: Gets the rank of the process.
  - `MPI_Comm_size`: Gets the number of processes.
- **Interval Division**:
  - The interval `[0, 1]` is divided among the processes.
  - Each process computes its partial sum of areas.
- **Partial Sum Computation**:
  - Each process computes the area for its assigned part of the interval.
- **Reduction**:
  - `MPI_Reduce`: Reduces all partial sums to compute the total area.
- **Timing**:
  - `MPI_Wtime`: Measures the execution time.

**Benefits

- **Parallelism**: The workload is distributed among multiple processes.
- **Efficiency**: The parallel version is faster for large values of `N` due to concurrent execution.
- **Scalability**: The program can scale with the number of processes.

By using MPI to parallelize the computation of π, you can significantly reduce the execution time and handle larger computations more efficiently.

7. Parallel Pi Computation Using MPI

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<sys/time.h>
#include "mpi.h"
#define N 999999999

int main(int argc, char **argv)
{
    int i, j, myid, size,start,end;
    double area, pi, recv_area;
    double dx, y, x;
    double exe_time;
```

```
struct timeval stop_time, start_time;

dx = 1.0/N;
x = 0.0;
area = 0.0;

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

start = myid * (N/size);
end = start + (N/size);

if(myid == (size - 1))
{
    end = N;
}

if(myid == 0)
{
    gettimeofday(&start_time, NULL);
}

for(i=start; i<end; i++)
{
    x = i*dx;
    y = sqrt(1-x*x);
    area += y*dx;
}

if(myid != 0)
{
    MPI_Send(&area, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
else
{
    for(i=1; i<size; i++)
    {
        MPI_Recv(&recv_area, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        area = area + recv_area;
    }

    gettimeofday(&stop_time, NULL);
    exe_time = (stop_time.tv_sec+(stop_time.tv_usec/1000000.0)) - (start_time.tv_sec+(start_time.tv_usec/1000000.0));

    pi = 4.0*area;
    printf("\n Value of pi is = %.16lf\n Execution time is = %lf seconds\n", pi, exe_time);
```

```
    }

    //End MPI Environment
    MPI_Finalize();
}
```

7.1. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_parallel_pi1.c
```

```
------------------------------------------------------------------
Command executed: mpicc mpi_parallel_pi1.c -o mpi_parallel_pi1.out -lm
------------------------------------------------------------------
Compilation successful. Check at mpi_parallel_pi1.out
------------------------------------------------------------------
```

- Run the program:

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

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


Value of pi is = 3.1415926555895561
Execution time is = 1.529297 seconds

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

8. Prime number count

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>
#include<sys/time.h>

#define N 1000000
/*
                N  PRIME_NUMBER

                1           0
               10           4
              100          25
            1,000         168
           10,000       1,229
          100,000       9,592
        1,000,000      78,498
       10,000,000     664,579
      100,000,000   5,761,455
    1,000,000,000  50,847,534

*/

int main()
{
    int i, j;
    int count, flag;
    double exe_time;
    struct timeval stop_time, start_time;

    count = 1; // 2 is prime. Our loop starts from 3

    gettimeofday(&start_time, NULL);


    for(i=3;i<N;i++)
    {
        flag = 0;
        for(j=2;j<i;j++)
        {
            if((i%j) == 0)
            {
                flag = 1;
```

```
                break;
            }
        }
        if(flag == 0)
        {
            count++;
        }
    }

    gettimeofday(&stop_time, NULL);
    exe_time = (stop_time.tv_sec+(stop_time.tv_usec/1000000.0)) - (start_time.tv_sec+(start_time.tv_usec/1000000.0));

    printf("\n Number of prime numbers = %d \n Execution time is = %lf seconds\n", count, exe_time);

}
```

- Compile the program:

```
gcc prime_count_serial.c -o prime_count_serial.out -lm
```

- Run the program:

```
./prime_count_serial.out
```

```
Number of prime numbers = 78498
Execution time is = 119.418229 seconds
```

9. Parallel Prime Number Counting Using MPI

This example demonstrates how to parallelize the prime number counting code using MPI. The range of numbers `[2, N]` is divided among multiple processes, each of which counts the primes in its assigned range. The results are then gathered to compute the total number of primes.

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

```c
#include <mpi.h>
#include <sys/time.h>

#define N 10000000

int main(int argc, char** argv) {
    int rank, size, i, j, count, flag, local_count;
    double start_time, end_time, execution_time;

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

    int range = N / size;
    int start = rank * range + 2;
    int end = (rank + 1) * range + 1;
    if (rank == size - 1) {
        end = N;
    }

    local_count = 0;
    if (rank == 0) {
        local_count = 1; // 2 is prime. Our loop starts from 3
    }

    start_time = MPI_Wtime();

    for (i = start; i <= end; i++) {
        flag = 0;
        for (j = 2; j <= sqrt(i); j++) {
            if ((i % j) == 0) {
                flag = 1;
                break;
            }
        }
        if (flag == 0) {
            local_count++;
        }
    }

    int total_count;
    MPI_Reduce(&local_count, &total_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    end_time = MPI_Wtime();
    execution_time = end_time - start_time;

    if (rank == 0) {
```

```
        printf("\n Number of prime numbers = %d \n Execution time is = %lf seconds\n", total_count, execution_time);
    }

    MPI_Finalize();
    return 0;
}
```

9.1. Compilation and Execution

- Compile the program:

```
bash compile.sh mpi_parallel_prime.c
```

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

- Run the program:

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

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


 Number of prime numbers = 664580
 Execution time is = 4.211497 seconds

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

9.2. Explanation

- **MPI Initialization**:
  - `MPI_Init`: Initializes the MPI execution environment.
  - `MPI_Comm_rank`: Gets the rank of the process.
  - `MPI_Comm_size`: Gets the number of processes.
- **Range Division**:
  - The range `[2, N]` is divided among the processes.
  - Each process computes the number of primes in its assigned range.
- **Partial Count Computation**:
  - Each process counts the primes in its range.
- **Reduction**:
  - `MPI_Reduce`: Reduces all partial counts to compute the total number of primes.
- **Timing**:
  - `MPI_Wtime`: Measures the execution time.

**Benefits

- **Parallelism**: The workload is distributed among multiple processes.
- **Efficiency**: The parallel version is faster for large values of `N` due to concurrent execution.
- **Scalability**: The program can scale with the number of processes.

By using MPI to parallelize the prime number counting, you can significantly reduce the execution time and handle larger computations more efficiently.

10. Serial Matrix Addition

```c
#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;
        }
    }
    /*
    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
```

Author: Abhishek Raj
Created: 2024-07-13 Sat 07:52