1. OPEN MPI

The Message Passing Interface (MPI) is a standardized and portable communication protocol designed to enable processes to exchange data in parallel computing environments [1]. MPI provides a set of functions for point-to-point messaging, collective communication, synchronization, and process management, allowing multiple tasks running on different processors or machines to coordinate their work efficiently. MPI has become the de facto standard for parallel programming on distributed-memory systems, ranging from small clusters to the world's largest supercomputers. OpenMPI is a specific, open-source implementation of the MPI standard and is one of several available implementations. The following scripts are for installing Open MPI.

```
# Linux installation
sudo apt openmpi-bin
sudo apt libopenmpi-dev

# MacOS installation
brew install open-mpi
```

A typical MPI program begins with MPI_Init, which sets up the MPI environment, followed by calls to obtain the process rank (its unique ID) and the total number of processes involved, i.e. the world_rank contains id corresponding to each process, and world_size holds the total number of processes that the program is running across. After initialization, the program enters the main parallel region, where all processes execute the code concurrently, and finally calls MPI_Finalize to clean up and terminate the MPI execution environment. The following code is the "hello-world" of an MPI program.

```
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);

// Parallel region

MPI_Finalize();
```

```
#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);

    printf("Hello from rank %d out of %d processors\n", world_rank, world_size);

    MPI_Finalize();
    return 0;
}
```

To run an MPI program using Open MPI, we must compile it with the *mpicc* command, then execute with the *mpirun* command. The *-np* option is used to define how many processes the application should launch, and the *–hostfile* option instructs MPI

[1] Decent MPI Document: https://hpc-tutorials.llnl.gov/mpi/

to run the program across multiple nodes in a distributed manner by reading the list of available hosts (and also CPUs) from the specified file.

```
# Compile Open MPI program
mpicxx <PROGRAM_NAME>.cpp -o <PROGRAM_NAME>

# Execute Open MPI program with <NUM_PROCESS> processes
mpirun -np <NUM_PROCESS> ./<PROGRAM_NAME>
```

1.1 MPI Point-to-Point

MPI point-to-point communication refers to the direct exchange of data between two specific processes, one acting as a sender and the other as a receiver. This form of communication is fundamental in distributed memory systems, enabling processes to coordinate and share data efficiently. Point-to-point communication supports both synchronous and asynchronous modes, offering flexibility in how data is transferred based on performance or synchronization needs.

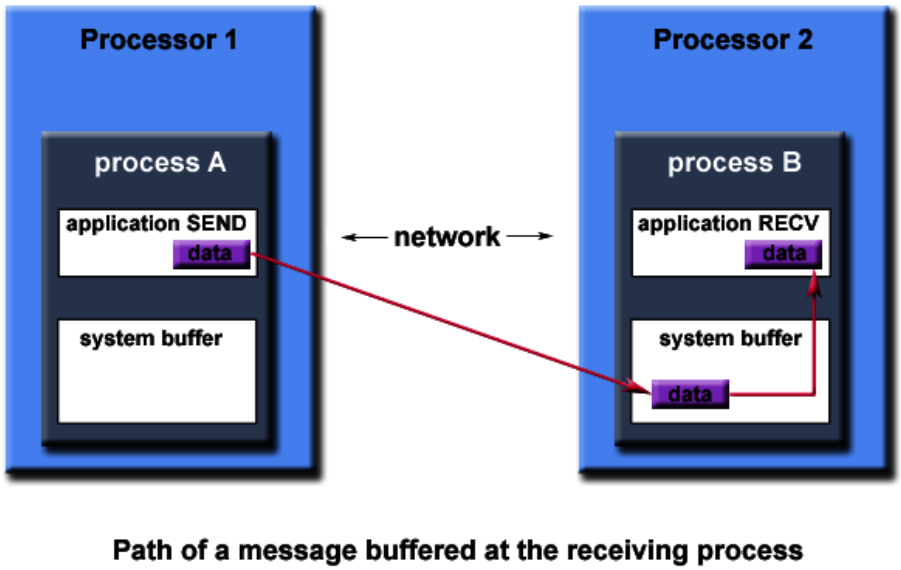

Figure 1: MPI Point-to-Point communication model.

MPI_Send and MPI_Recv are used for synchronous communication in MPI Point-to-Point. MPI_Send and MPI_Recv are blocking operations and ensure synchronization between processes during message transfer, with communication finishing only when both the sender and the receiver reach the communication point. Both functions return an integer error code MPI_SUCCESS if successful.

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);
```

- *buf:* Pointer to the data buffer to be sent.
- *count:* Number of elements to be sent/received.
- *datatype:* Data type of each element (e.g., MPI_INT, MPI_FLOAT).
- *dest:* Rank of the destination process.

- *tag:* Message tag to identify the message.
- *comm:* Communicator (usually MPI_COMM_WORLD).
- *source:* Rank of the sending process (or MPI_ANY_SOURCE).
- *tag:* Message tag (or MPI_ANY_TAG).
- *status:* An MPI_Status contains information about the received message.

MPI provides various options to suit different scenarios; however, this course only requires students to use the basic communication model. The following code demonstrates how to use MPI_Send and MPI_Recv in a basic setup.

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

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

    int world_rank = -1, world_size = 0;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_size < 2) {
        if (world_rank == 0) {
            printf("Need at least 2 processes.\n");
        }
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    if (world_rank == 0) {
        int temp_c = 27; // e.g., sensor reading
        // Synchronous: completes only when Rank 1 has posted its matching Recv.
        MPI_Ssend(&temp_c, 1, MPI_INT, 1, 100, MPI_COMM_WORLD);
        printf("Rank 0 Sent temperature %d C to Rank 1 (synchronous)\n", temp_c);
    } else if (world_rank == 1) {
        int received = -1;
        MPI_Recv(&received, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Rank 1 Received temperature %d C from Rank 0\n", received);
    }

    MPI_Finalize();
    return 0;
}
```

In contrast, MPI_Isend and MPI_Irecv are used for asynchronous communication. With non-blocking operations, the function returns immediately, and the actual data transfer happens in the background, allowing the program to perform other tasks while waiting for the data and avoiding deadlock occurs. MPI_Wait and MPI_Waitall are used to ensure communications have completed before proceeding. The prototype of non-blocking operations are similar to the blocking operations.

```c
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);

int MPI_Wait(MPI_Request *request, MPI_Status *status);
int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[]); // count is the number of requests/statuses
```

The dummy code below demonstrates how to use non-blocking operations. Each process calls the send/receive function and continues with its own work. When the transmitted data is required for a later task, MPI_Wait is used to ensure that the communication has completed successfully before continuing.

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

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

```
    int rank, size, send_data, recv_data;
    MPI_Request request;
    MPI_Status status;

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

    if (size < 2 || size % 2 == 1) {
        if (rank == 0) printf("Need an even number of processes (at least 2).\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    if (rank % 2 == 0) {
        send_data = rank;
        MPI_Isend(&send_data, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD, &request);
        printf("Process %d started non-blocking send.\n", rank);
        // Do local job
        MPI_Wait(&request, &status);  // Wait until send completes
        printf("Process %d completed send with data = %d.\n", rank, send_data);
    } else {
        MPI_Irecv(&recv_data, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD, &request);
        printf("Process %d started non-blocking receive.\n", rank);
        // Do local job
        MPI_Wait(&request, &status);  // Wait until receive completes
        printf("Process %d completed receive with data = %d\n", rank, recv_data);
    }

    MPI_Finalize();
    return 0;
}
```

1.2 MPI Collective

MPI collective communication involves data movement or synchronization across a group of processes within a communicator. Unlike point-to-point communication which occurs between pairs of processes, collective operations manage communication among all members of a group in a single function call. This makes them simpler to use and often more optimized on modern parallel systems.

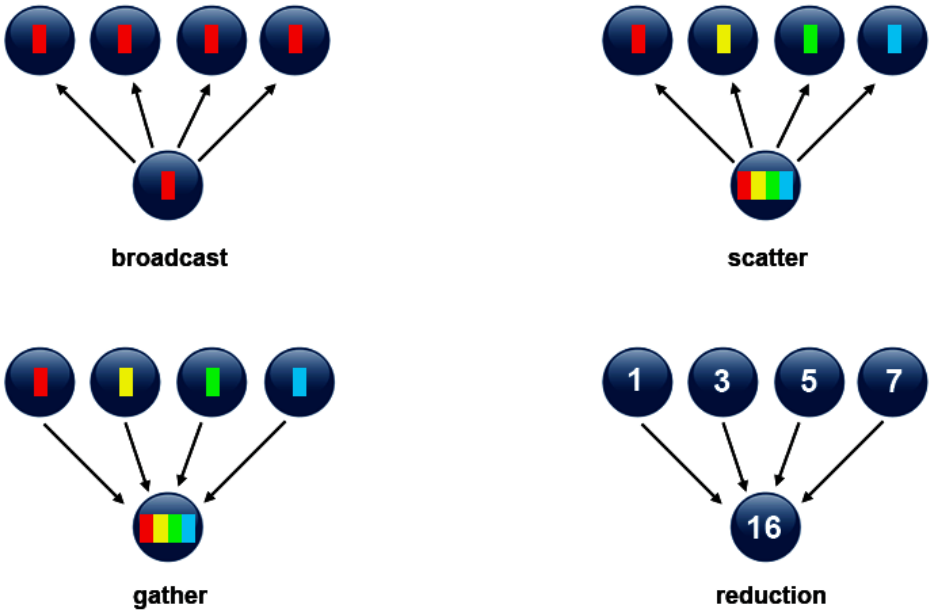

Figure 2: Examples of MPI collective communication model.

Broadcast, scatter, gather, and reduce are the commonly used of MPI collective operations for data sharing and aggregation among processes. MPI_Bcast sends the same block of data from a root process to all other processes; MPI_Scatter distributes distinct

portions of an array from the root process to the others so that each process can work on a separate chunk; MPI_Gather collects equal-sized data portions from each process into a single array on the root process; MPI_Reduce combines data from all processes using a specified operation and delivers the aggregated result to the root process; finally, MPI_Barrier serves as a synchronization point where all processes pause until each one of them has reached the barrier, ensuring coordinated progression.

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,  int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,  int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
int MPI_Barrier(MPI_Comm comm);
```

- *root:* Rank that originates/collects the data.
- *buffer:* Pointer to the data buffer to be sent; on others, receives it in-place.
- *sendbuf:* Pointer to the data buffer to be sent.
- *sendcount:* Number of elements to be sent.
- *sendtype:* Data type of each element in the send buffer.
- *recvbuf:* Pointer to the data buffer where each process stores its portion.
- *recvcount:* Number of elements to be received.
- *recvtype:* Data type of each element in the receive buffer.
- *op:* Predefined reduction operation or user-defined operation.

MPI Reduction Operation		Data Type
MPI_MAX	maximum	integer, float
MPI_MIN	minimum	integer, float
MPI_SUM	sum	integer, float
MPI_PROD	product	integer, float
MPI_LAND	logical AND	integer
MPI_BAND	bit-wise AND	integer, MPI_BYTE
MPI_LOR	logical OR	integer
MPI_BOR	bit-wise OR	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double, long double
MPI_MINLOC	min value and location	float, double, long double

Table 1: The predefined MPI reduction operations. Users can also define their own reduction functions by using the MPI_Op_create routine.

MPI collective communication also provides asynchronous variants such as MPI_Ibcast, allowing communication and computation to overlap to reduce idle waiting time. However, this course only requires students to be familiar with basic blocking operations. The dummy code below shows how to use the aforementioned collective functions.

```cpp
#include <mpi.h>
#include <iostream>
#include <vector>
#include <numeric>
#include <cstdlib>
#include <algorithm>

int main(int argc, char** argv) {
    int world_rank = 0, world_size = 1;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // ------------------ Broadcast: share problem size N ------------------
    int N = 16; // default
    if (world_rank == 0 && argc > 1) {
        N = std::max(1, std::atoi(argv[1]));
    }
    MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Partition with padding so Scatter can send equal chunks
    int chunk  = (N + world_size - 1) / world_size; // ceil(N / world_size)
    int padded = chunk * world_size;

    // ------------------ Prepare data only on root ------------------
    std::vector<int> sendbuf; // only valid on root
    if (world_rank == 0) {
        sendbuf.resize(padded, 0);
        std::iota(sendbuf.begin(), sendbuf.begin() + N, 1); // 1..N, padding stays 0
    }

    // ------------------ Barrier: synchronize before timing --------------
    MPI_Barrier(MPI_COMM_WORLD);
    double t0 = MPI_Wtime();

    // ------------------ Scatter: distribute equal-sized chunks -----------
    std::vector<int> local(chunk, 0);
    MPI_Scatter(
        (world_rank == 0 ? sendbuf.data() : nullptr), chunk, MPI_INT, local.data(), chunk, MPI_INT, 0, MPI_COMM_WORLD
    );

    // ------------------ Local work: compute partial sum ------------------
    int local_sum = std::accumulate(local.begin(), local.end(), 0);

    // ------------------ Allreduce: global sum on every rank --------------
    int global_sum_all = 0;
    MPI_Allreduce(&local_sum, &global_sum_all, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    // ------------------ Reduce: global sum only to root ------------------
    int global_sum_root = 0; // meaningful only on rank 0
    MPI_Reduce(&local_sum, &global_sum_root, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // ------------------ Gather: collect each rank's partial sum ----------
    std::vector<int> partial_sums; // only meaningful on root
    if (world_rank == 0) partial_sums.resize(world_size, 0);
    MPI_Gather(
        &local_sum, 1, MPI_INT, (world_rank == 0 ? partial_sums.data() : nullptr), 1, MPI_INT, 0, MPI_COMM_WORLD
    );

    // ------------------ Barrier: synchronize end of the phase -----------
    MPI_Barrier(MPI_COMM_WORLD);
    double t1 = MPI_Wtime();

    // ------------------ Report on root ----------------------------------
    if (world_rank == 0) {
        long long expected = 1LL * N * (N + 1) / 2;
        std::cout << "N = " << N << ", world_size = " << world_size << "\n";
        std::cout << "Per-rank partial sums: ";
        for (int s : partial_sums) std::cout << s << " ";
        std::cout << "\nAllreduce global sum = " << global_sum_all
                  << " | Reduce(to root) = " << global_sum_root
                  << " | Expected = " << expected << "\n";
        std::cout << ((global_sum_all == expected && global_sum_root == expected)
                      ? "OK: sums match.\n" : "WARNING: mismatch.\n");
        std::cout << "Elapsed (synchronized) time = " << (t1 - t0) << " s\n";
    }

    MPI_Finalize();
    return 0;
}
```

2. RADIOACTIVE CONTAMINATION CONCENTRATION

In radioactive dispersion, the Advection-Diffusion-Decay (PDE) balance is commonly used because it provides a clear way to describe how radioactive material behaves in the air over time. The main processes of PDE includes: the wind carrying the cloud from one place to another, turbulence making it spread out, the natural decay of the radioactive particles reducing their strength, and the settling of dust onto the ground or into rain removing it from the air. By bringing all these effects together, the PDE acts like a simple rulebook that helps us understand and predict where the radioactive cloud will move, how far it will spread, and how quickly it will fade away over time. In particular, the PDE formula is described as below:

$$\boxed{\frac{\partial C}{\partial t} + u\nabla C = D\nabla^2 C - (\lambda + k)C}$$

- $C(x, y, t)$ is the concentration at position (x, y) of time step t.
- $u = (u_x, u_y)$ is the wind velocity from West to East and from North to South.
- D is the diffusion coefficient often inferred from experiments.
- λ is a physical constant representing the decay of radioactive.
- k is the deposition rate also inferred through experiments.

For the sake of simplicity, we use an upwind scheme to model how radioactive material is carried by the wind. The calculation can be done by following:

$$\boxed{u\nabla C \approx u_x \frac{C_{i,j} - C_{i-1,j}}{dx} + u_y \frac{C_{i,j} - C_{i,j-1}}{dy}}$$

For the diffusion factor, we can calculate the spreading effect by applying a discrete Laplacian to the concentration grid. In practical terms, it checks whether a cell has more or less radioactive material compared to its neighbors as illustrated below:

$$\boxed{\nabla^2 C \approx \frac{C_{i+1,j} - 2C_{i,j} + C_{i-1,j}}{dx^2} + \frac{C_{i,j+1} - 2C_{i,j} + C_{i,j-1}}{dy^2}}$$

In summary, the updated radioactive concentration at each location is determined by its previous value and the deviation given by the PDE over a small time step. This iterative behavior is widely used in computational science to simulate how systems evolve over time. In particular, the new concentration is calculated by:

$$\boxed{C_{new} = C_{old} + dt \times \frac{\partial C}{\partial t}}$$

3. EXERCISES

Through the data collected by “little boy” and “fat man”, GPT expert estimates that the physical coefficients of DF-61 ICBM are $D = 1000$, $k = 10^{-4}$, and $\lambda = 3 * 10^{-5}$. Suppose DF-61 is dropped in an urban area with wind velocity equal to $u = (3.3, 1.4)$.

In this exercise, a matrix 4000×4000 where each cell corresponds to a 10 km² area and contains a contamination value, is given. The task is to simulate how the radioactive contamination diffuses across the map, using zero padding at the boundaries, for the first 100 seconds. In particular, students need to implement a **sequential version** together with a **parallel version** and then conduct a time comparison. For the parallel version, students are required to become familiar with the basic MPI functions:

- The original map is divided into multiple independent blocks, which are then distributed to the other processes using MPI Scatter.
- In each iteration, every process calculates the updated radioactive values for its assigned block (each process has to exchange the boundary values with its neighboring processes using MPI Point-to-Point communication) and synchronizes with the other processes before moving to the next step using MPI Barrier.
- Additionally, in each iteration, the total number of uncontaminated blocks should also be reported using MPI Reduce.
- After 100 iterations, a stop signal is distributed to all processes using MPI Broadcast. And the partial blocks from each process are collected with MPI Gather to assemble the complete final map.

**** Submission:** Works must be done individually.

- Compress all necessary files as **"Lab_2_<Student ID>.zip"**.
- Students must submit and demonstrate their results in a detailed report.