Approximation of PI using MPI

You task is to implement two MPI versions of the given sequential code that approximates the value of PI. In the MPI code, each process calculates a partial approximation sum and communicates it to process 0 (rank 0), which receives the values, sums up the partial sums, and prints the approximated value of Pi and the time it took for the code to run. The first version of your code should use only MPI point-to-point communication, and the second version should use MPI collective communication. Finally, execute one of the versions on ALMA using 2 nodes and 8 MPI Processes following the tutorial below.

Note: the initial code on the online platform is a sequential code, but it is still executed with 4 MPI processes (mpirun -np 4) by the platform, therefore it will produce an output 4 times since the whole program is executed on each process. We want to have an MPI program and split the work among these processes (see SPMD in the lecture slides).

MPI Walkthrough

Let us first start by transforming the sequential code so it uses MPI. First, we need to include the `mpi.h` header file, which we typically write on top of the code where other include statements are:

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  // ...
}
```

An MPI program starts with `MPI_Init()`, and ends with `MPI_Finalize()`. Moreover, when executed with "**`mpirun -np 4`**" we know that we will have 4 MPI processes, and each process will have its own ID (also known as rank in MPI) in the range from 0 to 3.

To get the number of available processes we call `MPI_Comm_size()`, which will save this value to the variable "`numprocs`". To get the process ID (rank) of each process we call `MPI_Comm_rank()` and the variable "`rank`" will have a unique value in the range from 0 to 3 on each process. So, the basic MPI code could look like this:

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  int numprocs, rank;

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

  // ...

  MPI_Finalize();
}
```

Implementing the Pi example

Let us now insert these calls into the sequential PI code (MPI calls in red):

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

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

    long num_steps = 10000000;

    auto ts_1 = std::chrono::high_resolution_clock::now();

    long i;
    double sum = 0.0, x = 0.0, pi = 0.0;
    double step = 1.0 / num_steps;

    for (i = 0; i < num_steps; i++) {
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }

    pi = sum * step;

    auto ts_2 = std::chrono::high_resolution_clock::now();

    std::cout << "pi: " << std::fixed << std::setprecision(4) << pi
              << ", execution time: " << (std::chrono::duration<double>(ts_2 - ts_1).count())
              << std::endl;

    MPI_Finalize();

    return 0;
}
```

Notice that we still use the C++ chrono library, but we really want to use `MPI_Wtime()`, so let us change the code to:

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

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

    long num_steps = 10000000;

    double t1 = MPI_Wtime();

    long i;
    double sum = 0.0, x = 0.0, pi = 0.0;
    double step = 1.0 / num_steps;

    for (i = 0; i < num_steps; i++) {
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }
```

```
    pi = sum * step;
    double t2 = MPI_Wtime();
    double elapsed_time = t2 -t1;


    std::cout << "pi: " << std::fixed << std::setprecision(4) << pi
              << ", execution time: " << elapsed_time
              << std::endl;

    MPI_Finalize();

    return 0;
}
```

Note that `elapsed_time` is different on each process, we need to decide which time really represents the execution time of the code.

Let us now utilize the `rank` variable to parametrize the for-loop across the available processes (`numprocs`):

```
    for ( i = rank; i < num_steps; i+=numprocs) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x); // partial sums
    }
```

After the loop is done, each process will have its own partial sum, which need to be summed up and printed to the std out on rank 0.

```
    // ...

    if ( rank != 0) {
        // send my partial sum to rank 0
    } else  {
        // receive partial sums from other ranks
        // calculate the total sum and
        pi = step * sum;
    }

    // end time measurements
    // print the results
    // ...
```

This functionality can be implemented using point-to-point communication using routines like `MPI_Send`/`MPI_Recv` (and variants like `MPI_Isend`/`MPI_Irecv`, …) or using collective communication, e.g., `MPI_Reduce`.

Task 1: Point-to-point communication

Implement parallel MPI code for approximating the value of Pi using point-to-point communication.

Useful routines:

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

See the following examples on point-to-point communication:

- https://moped.par.univie.ac.at/test.html?taskcode=mpi15 (point-to-point)
- https://moped.par.univie.ac.at/test.html?taskcode=mpi01 (one-to-many)

Check the MPI Basic Point-to-Point Example Walkthrough section below for more explanations.

Task 2: Collective communication

Implement parallel MPI code for approximating the value of Pi using collective communication.

Useful routines:

- ```
  int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
                 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  ```
- ```
  int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
  ```

See the following examples on `MPI_Reduce/MPI_Allreduce` communication:

- https://moped.par.univie.ac.at/test.html?taskcode=mpi07
- https://moped.par.univie.ac.at/test.html?taskcode=mpi09

Task 3

Your task is to connect to ALMA and create a job script for Slurm for executing your code using 8 MPI processes over 2 ALMA nodes.

First, copy your code into pi-mpi.cpp on ALMA.
  ```

Then, compile your code with:

```
mpic++ -O2 -o pi-mpi pi-mpi.cpp
```

Afterwards, create a file called "jobscript.sh" that looks like this:

```
#!/bin/bash
#SBATCH -N 2
#SBATCH --ntasks 8
#SBATCH -t 30
mpirun ./pi-mpi
```

Finally, submit your code for execution with:

```
sbatch jobscript.sh
```

This command tells Slurm to execute your code using two nodes. You can check if you code is running with squeue or cancel it with scancel <job_id>.

MPI Basic Point-to-Point Example Walkthrough

Let us now discuss a simple point-to-point communication example in MPI. In this example, process 0 sends an array of integers to process 1. You can try the full example on the following link:

https://moped.par.univie.ac.at/test.html?taskcode=mpi15

Let's now discuss this basic example, here is the source code:

```
#include <mpi.h>
#include <iostream>
#include <vector>

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int n = 10;
  std::vector<int> array(n, 1);
  int send_count = 10;

  if (rank == 0) {
    array = { 0,1,2,3,4,5,6,7,8,9 }; // change some values on rank 0
    MPI_Send(array.data(), send_count, MPI_INT, 1, 0, MPI_COMM_WORLD);
    // sends "send_count" MPI_INT elements of array to dest rank 1 in MPI_COMM_WORLD with tag 0
  }

  if (rank == 1) {
    MPI_Recv(array.data(), send_count, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }

  // print first 3 elements on each rank
  std::cout << rank << ": " << array[0] << ", " << array[1] << ", " << array[2] << ", ..." << std::endl;

  MPI_Finalize();
}
```

In this code, rank 0, calls the `MPI_Send()` routine to send 10 elements of MPI_INT type of the array "array" to rank 1. On the other hand, rank 1 calls the `MPI_Recv()` routine to receive this data from rank 0, and save it into its own array "array", using the same data representation (MPI_INT). Note that fourth argument to `MPI_Send()` specifies the destination rank, and the fourth argument of the `MPI_Recv()` specifies the source rank. These arguments must "match" for the communication to succeed, e.g., if rank 0 sends to rank 1, then rank 1 needs to receive from rank 0.

The send routine has the following parameters:

```
int MPI_Send(void *buf,                 // send buffer
             int count,                 // number of elements
             MPI_Datatype datatype,     // data type
             int dest,                  // destination rank
             int tag,                   // tag
             MPI_Comm comm              // communicator
            );
```

Here, the first argument specified the initial memory address of the data we want to send. The second argument represents the number of elements, and the thirst argument is the type of data we want to send. Note that in MPI we need to specify MPI data types instead of native ones. In this case we represent C++ int with MPI_INT. The tag is used to send additional information with a message, which we do not use for this example, and for the communicator we use MPI_COMM_WORLD, which represents all available processes to MPI.

The receive routine has the following parameters:

```
int MPI_Recv(void *buf,                 // receive buffer
             int count,                 // number of elements
             MPI_Datatype datatype,     // data type
             int source,                // source rank
             int tag,                   // tag
             MPI_Comm comm,             // communicator
             MPI_Status status,         // status
            );
```

As with the send routine, for the receive routine the first argument is the initial address of the receive buffer which we use to save the data. For example, this could be an array which is big enough to receive the number of elements of the data types specified in the second and third argument. To receive we must specify the source rank that calls the send routine, and we need to specify the same tag we used for the send routine (or use a wildcard like MPI_ANY_TAG), and we need to use the same communicator (MPI_COMM_WORLD). Finally, the last argument (status) represents the status of the reception information returned by receive operations and can be used to read the source rank, tag, and error code of the received message.