Parallel Programming course. Introduction

Obolenskiy Arseniy, Nesterov Alexander

Nizhny Novgorod State University

November 9, 2024

Contents

What is MPI?

MPI (Message Passing Interface) is a standardized and portable message-passing system, designed to function on a variety of parallel computing architectures.
Primarily used in high-performance computing (HPC) to allow different processes to communicate with each other in a distributed memory environment.

MPI: library vs standard

Aspect	MPI Standard	MPI Library
Definition	A formal set of specifications	A concrete software implementation
Purpose	Defines the behavior of message-passing systems	Provides a runnable implementation of the standard
Portability	Platform-agnostic guidelines	Implementations may be platform-specific
Performance	No direct impact on performance	Optimized for different platforms and hardware
Examples	MPI-1, MPI-2, MPI-3, MPI-4 (specifications)	MPICH, Open MPI, MS MPI (Microsoft MPI), Intel MPI

Table: Key Differences Between MPI Standard and MPI Library

"Hello, World" in MPI

Listing 1: Basic application written using MPI

```
1  #include <mpi.h>
2
3  #include <iostream>
4
5  int main(int argc, char** argv) {
6    MPI_Init(&argc, &argv);
7
8    int world_size;
9    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
10
11   int world_rank;
12   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
13
14   char processor_name[MPI_MAX_PROCESSOR_NAME];
15   int len_chars;
16   MPI_Get_processor_name(processor_name, &len_chars);
17
18   MPI_Barrier(MPI_COMM_WORLD);
19   std::cout << "Processor = " << processor_name << std::endl;
20   std::cout << "Rank = " << world_rank << std::endl;
21   std::cout << "Number of processors = " << world_size << std::endl;
22
23   MPI_Finalize();
24   return 0;
25 }
26
```

Compiling MPI application

Linux:

```
mpicc -o hello_mpi hello_mpi.c
```

Windows:

```
cl /I"C:\Program Files (x86)\Microsoft SDKs\MPI\Include"
hello_mpi.c /link /LIBPATH:"C:\Program Files (x86)\Microsoft
SDKs\MPI\Lib\x64" msmpi.lib
```

Running MPI application

Important! If you run application directly (`./hello_mpi`) you will not get expected result!

Linux:

```
mpirun -n 4 ./hello_mpi
```

Windows:

```
mpiexec -n 4 hello_mpi.exe
```

MPI initialization: MPI_Init()

`int MPI_Init(int *argc, char ***argv)`

It initializes the MPI environment and must be called before any other MPI function.

Listing 2: Basic application written using MPI

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char** argv) {
5      // Initialize the MPI environment
6      MPI_Init(&argc, &argv);
7
8      ...
9
10     return 0;
11 }
12
```

MPI data distribution example

Listing 3: MPI data distribution example

```
1   #include <mpi.h>
2   #include <stdio.h>
3   int main(int argc, char** argv) {
4       // Initialize the MPI environment
5       MPI_Init(&argc, &argv);
6
7       // Get the number of processes
8       int world_size;
9       MPI_Comm_size(MPI_COMM_WORLD, &world_size);
10
11      // Get the rank of the process
12      int world_rank;
13      MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
14
15      // Define message
16      int number = 0;
17      if (world_rank == 0) {
18          // If we are rank 0, set number to -1 and send it to process 1
19          number = -1;
20          MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
21          printf("Process 0 sent number %d to process 1\n", number);
22      } else if (world_rank == 1) {
23          // If we are rank 1, receive the number from process 0
24          MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
25          printf("Process 1 received number %d from process 0\n", number);
26      }
27
28      // Finalize the MPI environment
29      MPI_Finalize();
30      return 0;
31  }
32
```

Problems in multiprocessing environment

There are some typical errors that may occur in multiprocessing environment. Case when there are variables allocated on all processes in quite common.

Listing 4: Common variables usage example

```
...
int main(int argc, char** argv) {
    ...
    // Pay attention to "number" variable
    int number; // NOTE: in this case data will be created on each process
    // If you leave this data uninitialized it may lead to an undefined behavior
    // on specific processes
    if (world_rank == 0) {
        // If we are rank 0, "number" variable will be initialized with -1.
        // OK here
        number = -1;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent number %d to process 1\n", number);
    } else if (world_rank == 1) {
        // If we are rank 1, receive the number from process 0
        // OK here
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received number %d from process 0\n", number);
    } else {
      // WARNING: If we are rank 2 and more, there is uninitialized "number" variable!
        // Potential mistake
    }
    ...
}
```

Problems in multiprocessing environment solution?

Example on previous slide may lead to an undefined behavior which is not obvious.

- In case of number of processes <= 2 there is no issue.
- In case of number of processes > 2 there is potential gap if we use `number` variable afterwards.

How to avoid memory issues?

- Do not initialize `number` variable outside `if` (create `number` inside `if` scope)
- Initialize `number` variable (e.g. `int number = 0;`)

Important! This is the only one simple case of potential error in multiprocessing environment. Pay attention to the safety because debugging in multiprocessing environment is more challenging that in single process environment.

MPI_Send()

```
int MPI_Send(const void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

Parameters:

- buf: The starting address of the data buffer to be sent.
- count: The number of elements in the buffer.
- datatype: The type of data being sent (e.g., `MPI_INT`, `MPI_FLOAT`).
- dest: The rank (ID) of the destination process.
- tag: A user-defined message identifier to differentiate messages.
- comm: The communicator that defines the group of processes within which the message is being sent (e.g., `MPI_COMM_WORLD`).

const void *buf, int count, MPI_Datatype datatype - data array description

MPI_Recv()

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

Parameters:

- buf: The starting address of the buffer where the received data will be stored.
- count: The maximum number of elements that the buffer can hold.
- datatype: The type of data being received (e.g., `MPI_INT`, `MPI_FLOAT`).
- source: The rank of the sending process. Use `MPI_ANY_SOURCE` to receive from any process.
- tag: The message identifier (tag). Use `MPI_ANY_TAG` to receive any message regardless of the tag.
- comm: The communicator for the group of processes within which the message is being received (e.g., `MPI_COMM_WORLD`).
- status: A structure that contains information about the received message, such as the actual source and tag.

MPI Communicators

MPI Communicator: A communicator in MPI defines a communication context, a group of processes that can send and receive messages from one another. The processes within a communicator are assigned unique ranks, which are integers that identify each process.
Ranks: Every process within a communicator has a rank, starting from 0. The rank helps to uniquely identify a process within the communicator.
Groups: A communicator has an associated group of processes. A group is an ordered set of processes, which MPI uses to determine communication partners.

MPI_COMM_WORLD

MPI_COMM_WORLD: The MPI_COMM_WORLD is the default communicator created when an MPI program starts. It includes all the processes that are initiated by the MPI runtime.

- Every process in the MPI application automatically becomes a part of MPI_COMM_WORLD, and this communicator is used for most communication operations in simple MPI programs.
- All the processes in the program are assigned a rank in MPI_COMM_WORLD, starting from 0 to the number of processes minus one.
- MPI_COMM_WORLD allows processes to exchange messages, perform collective operations (e.g. broadcasting, reducing, scattering, etc.), and more.

Performance measurement in MPI: `MPI_Wtime()`

`double MPI_Wtime(void)`

- `MPI_Wtime()` is a function provided by the MPI standard to measure the wall-clock time (in seconds) since some arbitrary point in the past.
- This function is often used for performance analysis in parallel programs to measure the execution time of sections of code.
- It returns a double precision floating-point number representing the current time. The returned time is in seconds.
- `MPI_Wtime()` is local to the process and does not guarantee synchronization between processes, meaning each process may have a different starting time reference.

Usage example:

```
1 double start = MPI_Wtime();
2 // Code to time
3 double end = MPI_Wtime();
4 double elapsed = end - start;
5
```

Documentation reference:

`https://www.mpich.org/static/docs/v3.2/www3/MPI_Wtime.html`

Thank You!

References

1. MPI Standard https://www.mpi-forum.org/docs/
2. MPICH guides: https://www.mpich.org/documentation/guides/
3. Microsoft MPI: https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi
4. OpenMPI docs: https://www.open-mpi.org/doc/