Parallel Programming course. MPI (detailed API overview)

Obolenskiy Arseniy, Nesterov Alexander

Nizhny Novgorod State University

November 9, 2024

Contents

1. Boost.MPI
2. Advanced Send/Receive API
3. Synchronization
4. Collective operations

Boost.MPI

Boost.MPI is a part of the Boost C++ libraries that provides C++ bindings for the Message Passing Interface (MPI).
Boost.MPI makes it easier to write distributed applications in C++ by wrapping the complex MPI API with C++-friendly abstractions, improving safety and reducing the amount of boilerplate code.
Key Features of Boost.MPI:

- Simplified use of MPI with C++ bindings.
- Supports complex data types through Boost.Serialization.
- Easier management of distributed tasks and communication.
- Compatible with common MPI implementations like MPICH, OpenMPI, MS MPI, etc.

Note: C API mappting ot Boost.MPI: link

For more details see Boost.MPI docs: link

Boost.MPI example

Listing 1: Hello World example with Boost MPI

```
#include <boost/mpi.hpp>
#include <iostream>

// Namespace alias for convenience
namespace mpi = boost::mpi;

int main(int argc, char* argv[]) {
  // Initialize the MPI environment
  mpi::environment env(argc, argv);
  mpi::communicator world;

  // Get the rank (ID) of the current process and the total number of processes
  int rank = world.rank();
  int size = world.size();

  if (rank == 0) {
    // If this is the root process (rank 0), send a message to another process
    std::string message = "Hello from process 0";
    world.send(1, 0, message);  // Send to process 1
    std::cout << "Process 0 sent: " << message << std::endl;
  } else if (rank == 1) {
    // If this is process 1, receive the message
    std::string received_message;
    world.recv(0, 0, received_message);  // Receive from process 0
    std::cout << "Process 1 received: " << received_message << std::endl;
  }

  return 0;
}

```

Why Using MPI_Send and MPI_Recv Is Not Enough?

Blocking Operations MPI_Send and MPI_Recv are blocking, causing processes to wait until communication completes. So they are the reason of:

- **Performance Bottlenecks:** Blocking calls can lead to idle CPU time, reducing parallel efficiency.
- **Lack of Overlap:** Cannot overlap computation with communication, limiting optimization opportunities.
- **Scalability Issues:** As the number of processes increases, blocking operations can significantly degrade performance.

MPI_Isend

Non-Blocking Send function. Initiates a send operation that returns immediately.

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

```
boost::mpi::request boost::mpi::communicator::isend(int dest, int tag,
const T* values, int n);
```

Parameters:

- buf: Initial address of send buffer
- count: Number of elements to send
- datatype: Data type of each send buffer element
- dest: Rank of destination process
- tag: Message tag
- comm: Communicator
- request: Communication request handle

Usage: Allows the sender to proceed with computation while the message is being sent.

MPI_Irecv

Non-Blocking Receive function. Initiates a receive operation that returns immediately.

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

```
boost::mpi::request boost::mpi::communicator::irecv(int source, int tag,
T& value);
```

Parameters:

- buf: Initial address of receive buffer
- count: Maximum number of elements to receive
- datatype: Data type of each receive buffer element
- source: Rank of source process or `MPI_ANY_SOURCE`
- tag: Message tag or `MPI_ANY_TAG`
- comm: Communicator
- request: Communication request handle

Usage: Allows the receiver to proceed with computation while waiting for the message.

What is synchronization in MPI?

Synchronization mechanisms are essential to coordinating processes. Sometimes we need to ensure that particular action has been already completed.
Synchronization facts:

- Process Coordination: Mechanism to ensure processes reach a certain point before proceeding
- Data Consistency: Ensures all processes have consistent data before computations
- Types of Synchronization:
  - Point-to-point synchronization: It involves explicit sending and receiving of messages between two processes using functions like `MPI_Send` and `MPI_Recv`
  - Collective synchronization: Collective operations (see next slides) are used, where all processes must participate
- Importance: Prevents race conditions and ensures program correctness

MPI_Barrier

Global Synchronization function. It blocks processes until all of them have reached the barrier.

```
int MPI_Barrier(MPI_Comm comm);
```

```
void boost::mpi::communicator::barrier();
```

Usage:

- Ensures all processes have completed preceding computations
- Commonly used before timing code segments for performance measurement
- Typical use case: Synchronize before starting a collective operation

Collective operations

Operations involving all processes within a communicator.

Characteristics:

- Implicit synchronization among processes.
- Cannot be initiated between subsets unless a new communicator is created.

Examples:

- Data movement operations (e.g., `MPI_Bcast`, `MPI_Gather`).
- Reduction operations (e.g., `MPI_Reduce`, `MPI_Allreduce`).

Benefits (why use them instead of send/recv?):

- Optimized for underlying hardware and common user scenarios.
- Simplifies code and improves readability.

Broadcast (MPI_Bcast)

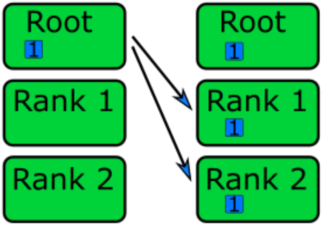
Send data from one process to all other processes.

`int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);`

`void broadcast(const communicator& comm, T& value, int root);` (needs `#include <boost/mpi/collectives.hpp>`)

Parameters:

- buffer: Starting address of buffer.
- count: Number of entries in buffer.
- datatype: Data type of buffer elements.
- root: Rank of broadcast root.
- comm: Communicator.



Source: https://pdc-support.github.io/introduction-to-mpi/07-collective/index.html

Reduction (MPI_Reduce)

Perform a global reduction operation (e.g., sum, max) across all processes.
Calculate the total sum of values distributed across processes.
Can be seen as the opposite operation to broadcast.

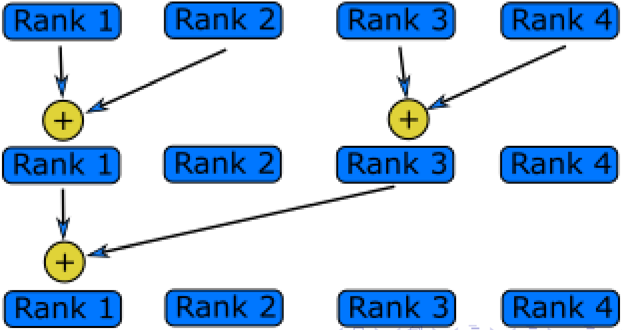
```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

```
void reduce(const communicator& comm, const T& in_value, T& out_value,
Op op, int root);
```

(needs #include <boost/mpi/collectives.hpp>)

Supported operations:

- MPI_SUM
- MPI_PROD
- MPI_MAX
- MPI_MIN



Source: https://pdc-support.github.io/introduction-to-mpi/07-collective/index.html

MPI_Gather

Collect data from all processes to a single root process.

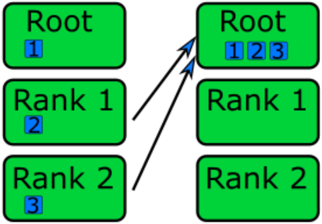
```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm);
```

```
void gather(const communicator& comm, const T& in_value, std::vector<T>&
out_values, int root);
```

(needs `#include <boost/mpi/collectives.hpp>`)

Parameters:

- sendbuf: Starting address of send buffer.
- recvbuf: Starting address of receive buffer (significant only at root).



Source: https://pdc-support.github.io/introduction-to-mpi/07-collective/index.html

MPI_Scatter

Distribute distinct chunks of data from root to all processes.

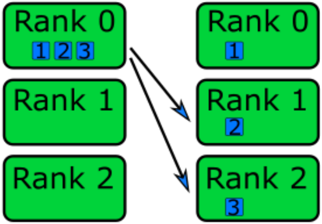
```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm);
```

```
void scatter(const communicator& comm, const std::vector<T>& in_values,
T& out_value, int root);
```

(needs `#include <boost/mpi/collectives.hpp>`)

Parameters:

- `sendbuf`: Starting address of send buffer (significant only at root).
- `recvbuf`: Starting address of receive buffer.



Source: https://pdc-support.github.io/introduction-to-mpi/07-collective/index.html

MPI_AllGather

Gather data from all processes and distributes the combined data to all processes.

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

```
void all_gather(const communicator& comm, const T& in_value,
std::vector<T>& out_values);
```

(needs #include <boost/mpi/collectives.hpp>)

Usage of this function reduces the need for separate gather and broadcast operations.

All-to-All (`MPI_Alltoall`)

Description: Each process sends data to and receives data from all other processes. It can be seen as transposing a matrix distributed across processes.

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

`void all_to_all(const communicator& comm, const std::vector<T>& in_values, std::vector<T>& out_values);` (needs `#include <boost/mpi/collectives.hpp>`)

Note: This operation is communication-intensive.

All API have not blocking versions

Non-Blocking collectives operations allow overlapping communication with computation.

Examples:

- `MPI_Ibcast`: Non-blocking broadcast.
- `MPI_Ireduce`: Non-blocking reduction.
- `MPI_Iallgather`: Non-blocking all-gather.

```
int MPI_Ibcast(void *buffer, int count, MPI_Datatype
datatype, int root, MPI_Comm comm, MPI_Request *request);
int MPI_Ireduce(const void *sendbuf, void *recvbuf, int
count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
comm, MPI_Request *request);
```

Usage flow is the same as for `MPI_Isend`/`MPI_Irecv`: Initiate the operation and later wait for its completion using `MPI_Wait` or `MPI_Test`.

Thank You!

References

1. MPI Standard https://www.mpi-forum.org/docs/
2. Boost.MPI Chapter in Boost documentation https://www.boost.org/doc/libs/1_86_0/doc/html/mpi.html
3. Open MPI v4.0.7 documentation: https://www.open-mpi.org/doc/v4.0/