Computação em Larga Escala

Message Passage Interface (MPI) - Part II

Eurico Pedrosa

António Rui Borges

Universidade de Aveiro - DETI

2025-04-05

Communications

Non-Blocking Communication

- Although MPI supports non-blocking communication in various contexts, we will focus here on **non-blocking point-to-point communication**, where a process (the **source**) sends a message to another process (the **destination**).

Non-Blocking Communication

- In **non-blocking operations**, both `MPI_Isend` and `MPI_Irecv` return immediately, without waiting for the communication to complete:
  - `MPI_Isend` initiates the send operation and returns before the message is actually delivered.
  - `MPI_Irecv` initiates the receive operation and returns before any data has necessarily been received.
- Since these operations are not synchronized, MPI provides additional routines to monitor and ensure completion:
  - `MPI_Wait` — blocks the caller until the communication is complete.
  - `MPI_Test` — checks whether the communication has completed, but does not block if it hasn't.

Non-Blocking Communication


```c
int MPI_Isend(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);
```

- `buf` — pointer to the memory buffer where the message is stored (for `MPI_Isend`) or will be received into (for `MPI_Irecv`)
- `count` — number of elements in the buffer
- `datatype` — MPI data type of each buffer element (e.g., `MPI_INT`, `MPI_FLOAT`)
- `dest` — rank of the destination process (only for `MPI_Isend`)
- `source` — rank of the source process (only for `MPI_Irecv`); can also be `MPI_ANY_SOURCE` to receive from any sender
- `tag` — message tag for identifying message types; can be `MPI_ANY_TAG` for wildcard matching
- `comm` — communicator object identifying the communication context
- `request` — pointer to an `MPI_Request` object that can later be used to query or wait for completion (e.g., via `MPI_Wait` or `MPI_Test`)

Non-Blocking Communication

```c
int MPI_Wait(MPI_Request *request, MPI_Status *status);

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

- `request` — pointer to the `MPI_Request` object associated with the non-blocking operation
- `flag` — (only in `MPI_Test`) pointer to an integer set to true (non-zero) if the operation is complete, false (zero) otherwise
- `status` — pointer to an `MPI_Status` structure that holds information about the completed operation
  - Contains fields such as `MPI_TAG` and `MPI_ERROR`
  - If the status is not needed, use the predefined constant `MPI_STATUS_IGNORE`

Non-Blocking Communication - Example


```cpp
int main(int argc, char *argv[]) {
  int rank, size;
  MPI_Request request, request;
  int data = 12345, dest = 1, source = 0;

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

  if (size < 2) {
    std::cerr << "This program requires at least two processes." << std::endl;
    MPI_Abort(MPI_COMM_WORLD, 1);
  }
  // root process send the data
  if (rank == 0) {
    std::cout << "Process " << rank << ": Sending data " << data << " to process " << dest << "..." << std::endl;
    MPI_Isend(&data, 1, MPI_INT, dest, 0, MPI_COMM_WORLD, &request);
    // Do some other work while the send is in progress
    std::cout << "Process " << rank << ": Doing some other work while sending..." << std::endl;
    for (int i = 0; i < 1000; i++) {
      // Simulate some computation
      data++; // Just increment the data (which isn't being used in the send anymore)
      std::this_thread::sleep_for(std::chrono::milliseconds(1)); // add a slight delay
    }
    // Wait for the send to complete
    std::cout << "Process " << rank << ": Waiting for send to complete..." << std::endl;
    MPI_Wait(&request, &status); // Blocking wait. Alternative: MPI_Test
    std::cout << "Process " << rank << ": Send completed." << std::endl;
  }
```

Non-Blocking Communication - Example

```
  // rank 1 will receive the data
  if (rank == 1) {
    data = 0; // Initialize data before receiving
    std::cout << "Process " << rank << ": Receiving data from process " << source << "..." << std::endl;
    MPI_Irecv(&data, 1, MPI_INT, source, 0, MPI_COMM_WORLD, &request);
    // Do some other work while receiving
    std::cout << "Process " << rank << ": Doing some other work while receiving..." << std::endl;
    for (int i = 0; i < 500; i++) { // Simulate some computation
      data--; // Just decrement the data (which isn't being used in the recv yet)
      std::this_thread::sleep_for(std::chrono::milliseconds(1)); // add a slight delay
    }
  }
  // Check if the receive is complete using MPI_Test
  int flag = 0;
  while (!flag) {
      MPI_Test(&request, &flag, &status);
      if (!flag) {
          std::cout << "Process " << rank << ": Receive not yet complete, doing more work..." << std::endl;
          for (int i = 0; i < 100; i++) { // Simulate more work
             data--;
             std::this_thread::sleep_for(std::chrono::milliseconds(1)); // add a slight delay
          }
      }
  }

  std::cout << "Process " << rank << ": Receive completed. Received data: " << data << std::endl;
  MPI_Finalize();
  return 0;
}
```

Non-Blocking Communication - Example


```
mpirun -n 2 ./non-blocking
Process 0: Sending data 12345 to process 1...
Process 1: Receiving data from process 0...
Process 0: Doing some other work while sending...
Process 1: Doing some other work while receiving...
Process 1: Receive completed. Received data: 12345
Process 0: Waiting for send to complete...
Process 0: Send completed.
```

Collective synchronization

Collective synchronization ensures that all processes in a communicator wait for each other to reach the same point in program execution before any can proceed.

There is one primary collective synchronization mechanism in MPI:

- `MPI_Barrier` — acts like a physical barrier:
  - All participating processes block until **every** process in the communicator has reached the barrier.
  - The last process to arrive at the barrier releases all others, allowing them to continue.

This is useful for timing measurements, coordination of phases, or ensuring consistent program state across processes.

Collective synchronization


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

- `comm` — communicator object identifying the group of processes that must synchronize

All processes in the communicator must call `MPI_Barrier`. Each will block until all others have also called it. Only then do they all proceed.

Collective synchronization - Example


```
int main(int argc, char *argv[]) {
  int rank, size, i, work;
  double start_time, end_time;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  // Use C++'s random number generation
  std::random_device rd;  // Used to obtain a seed for the random number engine
  std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
  std::uniform_int_distribution<> distrib(1000, 10000); // Define the range
  // Simulate variable amounts of work performed by each process
  if (rank == 0) {
    work = 1000;
  } else if (rank == 1) {
    work = 5000;
  } else {
      work = distrib(gen); // Other processes do random work
  }

  start_time = MPI_Wtime();

  // Simulate doing work
  std::cout << "Process " << rank << ": Starting work, iterations = " << work << std::endl;
  for (i = 0; i < work; i++) {
    // Some dummy calculation to simulate work
    double temp = (double)i * i * i + 1.0;
    temp = std::sqrt(temp);
  }
```

Collective synchronization - Example


```cpp
  end_time = MPI_Wtime();

  std::cout << "Process " << rank << ": Work done. Local Time: " << end_time - start_time << std::endl;

  // Barrier Synchronization
  std::cout << "Process " << rank << ": Reaching barrier..." << std::endl;
  MPI_Barrier(MPI_COMM_WORLD); // All processes MUST reach here before continuing.
  std::cout << "Process " << rank << ": Past barrier." << std::endl;

  // After the barrier, all processes have completed their work.
  // We can now safely calculate the total execution time.
  if (rank == 0) {
    double total_time = MPI_Wtime() - start_time;
    std::cout << "Process " << rank << ": All processes have completed their work." << std::endl;
    std::cout << "Process " << rank << ": Approximate total execution time: " << total_time << std::endl;
  }

  MPI_Finalize();
  return 0;
}
```

Collective synchronization - Example

```
mpirun -n 3 ./barrier
Process 0: Starting work, iterations = 1000
Process 0: Work done. Local Time: 9e-06
Process 1: Starting work, iterations = 5000
Process 1: Work done. Local Time: 1.3e-05
Process 2: Starting work, iterations = 3882
Process 2: Work done. Local Time: 1.3e-05
Process 2: Reaching barrier...
Process 0: Reaching barrier...
Process 1: Reaching barrier...
Process 0: Past barrier.
Process 0: All processes have completed their work.
Process 0: Approximate total execution time: 5.4e-05
Process 1: Past barrier.
Process 2: Past barrier.
```

Sorting With MPI

The Challenge: Sorting Large Datasets

- **Sorting** is a fundamental task in computing and data processing.
- **Real-world datasets** often reach sizes of **gigabytes to terabytes (or more)**.
- **Sequential sorting** on a single processor becomes a bottleneck:
  - Limited by the **CPU speed** and **memory capacity** of one machine.
- **Goal**: Utilize **multiple processors or machines** to accelerate sorting through **parallel or distributed algorithms**.

Parallel Sorting with MPI

Sorting a large dataset in parallel using **MPI** involves three key stages:

1. **Distribute** the dataset across multiple processes.
2. **Sort locally** on each process.
3. **Merge or redistribute** the sorted chunks to obtain a **globally sorted result**.

There are several algorithmic approaches to this process, each with **trade-offs** depending on:

- **Dataset size and structure**
- **Initial data distribution**
- **Communication and synchronization costs**

Parallel Sorting with MPI

Examples of parallel sorting techniques:

- **Parallel Merge Sort**
- **Sample Sort**
- **Bitonic Sort**
- **Bucket Sort**

Choosing the right approach depends on the application's **performance goals** and **hardware characteristics**.

Parallel Merge Sort (MPI)

Inspired by the classic merge sort, this parallel approach leverages MPI to divide, sort, and merge large datasets efficiently.

1. Divide

- The dataset is **evenly distributed** among MPI processes.
- Each process receives a **chunk of the input data**.

2. Local Sort

- Each process independently sorts its local chunk using a **sequential sorting algorithm** (e.g., `quicksort`, `mergesort`).
- This step is **crucial for performance**, as it determines the quality of the global merge phase.

Parallel Merge Sort (MPI)

3. Merge

The core challenge is to **merge the sorted chunks** across processes.

Two common strategies are:

- **Pairwise Merge**:
  - Processes are paired.
  - Each pair exchanges data: one keeps the **lower half**, the other keeps the **upper half** after merging.
  - This continues in $\log_2(P)$ steps (where P is the number of processes) until data is globally sorted.
- **Merge Tree / Recursive Doubling**:
  - Processes are logically arranged in a **binary tree**.

Parallel Merge Sort (MPI)

- Leaf nodes merge and pass results up to their parent.
- This continues until the **root holds the fully sorted dataset**.
- Efficient in communication, especially with **recursive doubling**.

4. Gather (Optional)

- If the fully sorted dataset is needed on a **single process** (e.g., rank 0), processes can **gather their portions** using `MPI_Gather` or `MPI_Gatherv`.
- Often, **data remains distributed**, which is ideal if **subsequent computation is also parallel**.

Parallel Merge Sort: Step-by-Step

Parallel merge sort with pairwise Merge

1. **Setup**
   - Call `MPI_Init`.
   - Use `MPI_Comm_rank` and `MPI_Comm_size` to get the **rank** and total number of **processes**.
2. **Data Distribution**
   - Use `MPI_Scatter` to divide the full dataset from **rank 0**.
   - Each process receives **N / size** elements (assuming `N` is divisible by `size`).
3. **Local Sort**
   - Each process sorts its chunk using a local algorithm (e.g., `std::sort()` or `qsort()`).

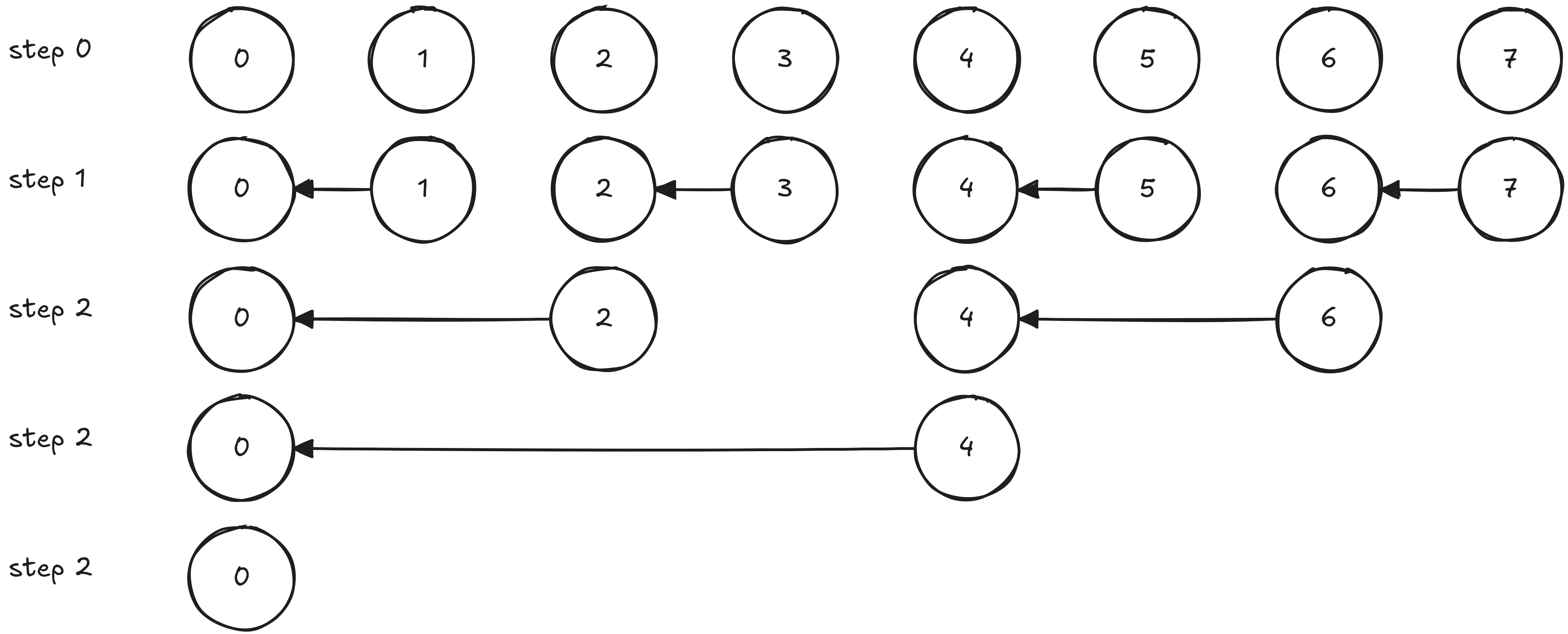
Parallel Merge Sort: Step-by-Step

4. **Merge Rounds**
   - Perform `log₂(size)` rounds of **pairwise merging**:
     - **Round 1**:
       - Processes with odd ranks send their sorted data to the rank just below (e.g., $1 \longrightarrow 0$, $3 \longrightarrow 2$).
       - Receiving processes merge their own data with received data and keep the **lower half**.
     - **Round 2**:
       - Every 4th process receives from its neighbor 2 steps away (e.g., $2 \longrightarrow 0$, $6 \longrightarrow 4$), merges, and keeps the lower half.
     - **Continue** this doubling pattern until the final round.

Parallel Merge Sort: Step-by-Step

5. **Final Merge**
   - In the last round, **rank `size/2`** sends to **rank 0**, which performs the final merge.
   - Now, **rank 0** holds the fully **sorted dataset**.
6. **Cleanup**
   - Finalize with `MPI_Finalize`.

Parallel Merge Sort: Step-by-Step



Parallel Merge Sort: Code Highlights

```
// 1. Scatter
MPI_Scatter(global_data.data(), local_n, MPI_INT,
            local_data.data(), local_n, MPI_INT, 0, MPI_COMM_WORLD);

// 2. Local Sort
std::sort(local_data.begin(), local_data.end());

// 3. Merge Loop (Conceptual)
for (int step = 1; step < size; step *= 2) {
    if (rank % (2 * step) == 0) { // I am a receiver
        if (rank + step < size) {
             // MPI_Probe to get size (optional)
             // MPI_Recv data from rank + step
             // local_data = merge(local_data, received_data);
        }
    } else { // I might be a sender
        int receiver = rank - step;
        if (receiver >= 0 && (rank - step) % (2 * step) == 0) {
             // MPI_Send local_data to receiver
             break; // My data is merged, I'm done with merging
        }
    }
}
```

Suggested Reading

- The Art of HPC by Victor Eijkhout of TACC
  - Volume 2: Parallel Programming for Science and Engineering
- Rookie HPC (MPI Documentation)
  - https://rookiehpc.org/mpi/docs/index.html