Introduction to MPI

Computational Science II (CAAM 520)

Christopher Thiele

Rice University, Spring 2020

Overview

- Distributed vs. shared memory parallelism
- Idea of message passing
- The Message Passing Interface (MPI)
- A basic example
- Running MPI applications
- Communicators
- Sending and receiving data

Distributed vs. shared memory parallelism

So far we used shared memory parallelism.

→ Workers were OpenMP ***threads***.

Next, we will learn about distributed memory parallelism.

→ Workers will be MPI ***processes***.

Key difference: Multi-threading is limited to a single computer, while processes on ***different*** computers can communicate via network.

Idea of message passing

By default, processes do not share memory.

If they do, such sharing is again limited to a single computer.

Instead, processes can ***pass messages*** between each other, i.e., they send and receive packages of data.

→ This approach works between processes on the same computer and between processes on different computers in a network.

The Message Passing Interface (MPI)

MPI is the standard interface for message passing and can be considered the standard for distributed memory parallelism.

- 1994: MPI 1.0
- 1997: MPI 2.0
- 2012: MPI 3.0
- MPI 4.0 is work in progress.

MPI - a first example

```
#include <mpi.h>

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("hello world from rank %d out of %d\n",
         rank, size);
  MPI_Finalize();
  return 0;
}
```

Building MPI applications

How can we build an MPI application?

Unlike OpenMP, which essentially extends C, C++, and Fortran, the MPI standard defines a function library.

Hence, there exist several competing implementations of MPI, many of which are independent of a specific language or compiler:

- MPICH
- MVAPICH2
- Open MPI (not to be confused with OpenMP)
- Intel MPI
- Cray MPI
- Microsoft MPI

Building MPI applications

To compile MPI code, we must tell the compiler where `mpi.h` is, where the MPI library is, etc.

For convenience, MPI implementations provide ***compiler wrappers***:

- `mpicc` - C compiler
- `mpicxx` - C++ compiler
- `mpif90` - Fortran 90 compiler

$\longrightarrow$ These are not compilers, just wrappers around GCC, the Intel compiler, etc.

Running MPI applications

How can we run an MPI application?

We must start ***multiple*** processes, possibly on different computers.

MPI implementations provide `mpiexec` and `mpirun` to do so. For example,

```
mpirun -n 4 ./my_mpi_app
```

starts four MPI processes on the local computer and "connects" them.

Running MPI applications

Always make sure that you compile and run your application using the same MPI implementation!

For example, you cannot compile your application with Open MPI and then run it with MVAPICH2's `mpirun`.

Initializing MPI

MPI must be initialized before it can be used, and it must be finalized at the end of your application.

```
#include <mpi.h>

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

  // ...

  MPI_Finalize();
  return 0;
}
```

Communicators

A ***communicator*** defines a subset of the MPI processes that were started with `mpirun`.

Individual processes within a communicator are identified with a unique ID, the ***rank***.

The communicator that contains ***all*** processes is `MPI_COMM_WORLD`.

→ We will use `MPI_COMM_WORLD` most of the time.

How can ranks share data?

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

const int my_result = do_work(rank);

// Add results from all ranks.
int sum = 0;
for (int r = 0; r < size; r++) {
  // r += ???;
}
```

$\longrightarrow$ How can ranks access each other's data?

Sending data with `MPI_Send`

To send data to other ranks, we can use `MPI_Send()`:

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

→ Send `count` contiguous items of type `datatype`, starting at `buf`, to rank `dest` in the communicator `comm`.

Sending data with `MPI_Send`

Example:

```c
double *array = malloc(128*sizeof(double));

// Send array to rank 1.
MPI_Send(array,
         128,
         MPI_DOUBLE,
         1,
         999,
         MPI_COMM_WORLD);
```

Receiving data with `MPI_Recv`

The target of a message must accept the message by calling `MPI_Recv()`:

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

→ Receive a message of (at most) `count` items of type `datatype` with tag `tag` from rank `source` in the communicator `comm` and store it in the buffer `buf`.

Receiving data with `MPI_Recv`

Example:

```
double *array = malloc(128*sizeof(double));

// Receive array from rank 0.
MPI_Status status;
MPI_Recv(array,
         128,
         MPI_DOUBLE,
         0,
         999,
         MPI_COMM_WORLD,
         &status);
```

The `MPI_Status` type

The `MPI_Status` type contains information about the message, such as

- `status.MPI_SOURCE`,
- `status.MPI_TAG`,

and the number of items that were received, which can be accessed using

```
int MPI_Get_count(const MPI_Status *status,
                  MPI_Datatype datatype,
                  int *count)
```

The MPI_Status type

If we are not interested in the status, we can pass MPI_STATUS_IGNORE to MPI_Recv instead of a status variable:

```
MPI_Recv(array,
         128,
         MPI_DOUBLE,
         0,
         999,
         MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
```

Blocking message passing

`MPI_Recv()` will block until the message has been received.

→ Watch out for deadlocks!

`MPI_Send()` will block until it is safe to use the send buffer again.

This does ***not*** ensure that the message has been received or even that it has been sent. `MPI_Ssend()` works just like `MPI_Send()`, but it blocks until the message has been received (synchronous send).

Blocking message passing

Blocking communication can cause problems:

```
int me, remote, my_data, remote_data;

MPI_Comm_rank(MPI_COMM_WORLD, &me);
// Assume that there are exactly two ranks.
remote = 1 - me;

// Exchange data with other rank.
MPI_Send(&my_data, 1, MPI_INT, remote,
         999, MPI_COMM_WORLD);
MPI_Recv(&remote_data, 1, MPI_INT, remote,
         999, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

→ This may or may not result in a deadlock! Why?

Blocking message passing

To avoid deadlocks when exchanging data, we can use `MPI_Sendrecv()`:

```
int MPI_Sendrecv(const void *sendbuf, int sendcount,
                 MPI_Datatype sendtype, int dest,
                 int sendtag,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvtype, int source,
                 int recvtag,
                 MPI_Comm comm, MPI_Status *status)
```

→ `MPI_Sendrecv()` is literally a combination of `MPI_Send()` and `MPI_Recv()`, but it avoids deadlocks.

Barriers

Just like OpenMP, MPI provides barriers to synchronize ranks:

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

For example, to wait for all MPI processes, use

```
MPI_Barrier(MPI_COMM_WORLD);
```

→ As with OpenMP, MPI barriers can cause poor performance and deadlocks.