Durham University

PHYS52015 Core Ib: Introduction to High Performance Computing (HPC)

Session V: MPI—basics and blocking P2P communication

Christopher Marcotte

Michaelmas term 2023

Outline

MPI Basics
Point-to-point communication
Tags

pollev.com/christophermarcotte820

Starting point

- Shared memory programming:
  - Multiple threads can access each other's data
  - Linux/OS terminology: one process with multiple tasks
- Distributed memory programming:
  - Different machines connected through a network *or*
  - one (multicore) computer running several processes, as processes do not share memory *or*
  - combinations
- De-facto standard for distributed memory programming:
  message passing (MPI = message passing interface)
- Message passing:
  - works on all/most architectural variants, i.e. is most general
    (though perhaps slower than OpenMP, e.g.)
  - requires additional coding, as we have to insert send and receive commands
  - orthogonal to other approaches, in particular to OpenMP
    (merger called MPI+X or hybrid)

Historic remarks on MPI

- MPI = Message Passing Interface
- Prescribes a set of functions, and there are several implementations (IBM, Intel, mpich, . . .)
- Kicked-off 1992–94
- Open consortium (`www.mcs.anl.gov/mpi`)
- Mature and supported on many platforms (de-facto standard)
- Alive:
  - Extended by one-sided communication (which does not really fit to name)
  - C++ extension dropped due to lack of users
- Huge or small:
  - Around 125 functions specified
  - Most applications use only around six

A first MPI application

```c
#include <mpi.h>
#include <stdio.h>
int main(  int argc, char** argv ) {
    MPI_Init( &argc, &argv );
    printf("Hello world!\n");
    MPI_Finalize();
    return 0;
}
```

```
mpicc -O3 myfile.cpp
mpirun -np 4 ./a.out
```

- Use `mpicc` which is a wrapper around your compiler
- Use `mpirun` to start application on all computers (SPMD)
- Exact usage of mpirun differs from machine to machine

A first MPI application

```
#include <mpi.h>
#include <stdio.h>
int main(  int argc, char** argv ) {
    MPI_Init( &argc, &argv );
    printf("Hello world!\n");
    MPI_Finalize();
    return 0;
}
```

- MPI functions become available through one header `mpi.h`
- MPI applications are ran in parallel (mpi processes are called **ranks** to distinguish them from OS processes)
- MPI code requires explicit initialisation and shutdown
- MPI functions always start with a prefix `MPI_` and then one uppercase letter
- MPI realises all return values via pointers
- MPI's initialisation is the first thing to do and also initialises `argc` and `argv`

MPI terminology and environment

Rank: MPI abstracts from processes/threads and calls each SPMD instance a *rank*. The total number of ranks is given by *size*.

```
#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 );
  MPI_Finalize();
  return 0;
}
```

- See the name conventions and the call-by-pointer policy.
- Compare to OpenMP which offers exactly these two operations as well.
- Different to OpenMP, we will however need *ranks* all the time, as we have to specify senders and receivers.
- For the time being, rank is a continuous numbering starting from 0.

MPI Environment Management

```c
#include "mpi.h"        // required MPI include file
#include <stdio.h>

int main(int argc, char *argv[]) {
    int  numtasks, rank, len, errorcode;
    char hostname[MPI_MAX_PROCESSOR_NAME];
    // initialize MPI
    MPI_Init(&argc,&argv);
    // get number of tasks
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    // get my rank
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    // this one is obvious
    MPI_Get_processor_name(hostname, &len);
    printf ("Number of tasks= %d My rank= %d Running on %s\n",
                numtasks,rank,hostname);
    // MPI_Abort(MPI_COMM_WORLD, errorcode);
    // done with MPI
    MPI_Finalize();
    return 0;
}
```

Concept of building block

- Content
  - How does MPI initialise/shutdown
  - How to compile MPI codes
  - Find out local SPMD instance's rank and the global size
  - MPI conventions
- Expected Learning Outcomes
  - The student knows the framework of an mpi application, can compile it and can run it
  - The student can explain the terms rank and size
  - The student can identify MPI conventions at hands of given source codes

Peer-to-peer Message Passing

- The Message Passing of MPI is (largely) borne by two functions:
  - `MPI_Send`
  - `MPI_Recv`
- Which interact through coordination across distinct processes.

```
// on rank from
MPI_Send(&data, count, MPI_INT, to, tag, MPI_COMM);
// on rank to
MPI_Recv(&data, count, MPI_INT, from, tag, MPI_COMM);
```

- Most of the hard work of parallelising with MPI is managing the `Send`s and `Recv`s consistently.
- Today we'll focus on the *blocking* versions of these calls.

Send

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

- `buffer` is a pointer to the piece of data you want to send away.
- `count` is the numer of items
- `datatype` . . . self-explaining
- `dest` is the rank (integer) of the destination node
- `comm` is the so-called communicator (always `MPI_COMM_WORLD` for the time being)
- Result is an error code, i.e. 0 if successful (UNIX convention)

Blocking: MPI_Send is called `blocking` as it terminates as soon as you can reuse the buffer, i.e. assign a new value to it, without an impact on MPI.

Receive

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

- `buffer` is a pointer to the variable into which the received data shall be stored.
- `count` is the numer of items
- `datatype` ... self-explaining
- `dest` is the rank (integer) of the source node (may be `MPI_ANY`)
- `comm` is the so-called communicator (always `MPI_COMM_WORLD` for the time being)
- `status` is a pointer to an instance of `MPI_Status` and holds meta information
- Result is an error code, i.e. 0 if successful (UNIX convention)

> Blocking: MPI_Recv is called `blocking` as it terminates as soon as you can read the buffer, i.e. MPI has written the whole message into this variable.

Blocking communication

Blocking: MPI_Send is called `blocking` as it terminates as soon as you can reuse the buffer, i.e. assign a new value to it, without an impact on MPI.

Blocking: MPI_Recv is called `blocking` as it terminates as soon as you can read the buffer, i.e. MPI has written the whole message into this variable.

- If a blocking operation returns, it does **not** mean that the corresponding message has been received.
- Blocking and asynchronous or synchronous execution have nothing to do with each other though a blocking receive never returns before the sender has sent out its data.
- If a blocking send returns, the data must have been copied to the local network chip.
- The term blocking just refers to the safety of the local variable.
- With blocking sends, you never have a guarantee that the data has been received, i.e. blocking sends are not *synchronised*.

From Mathematics and Computer Science (MCS) at Argonne National Lab (ANL):

MPI has a number of different "send modes." These represent different choices of buffering (where is the data kept until it is received) and synchronization (when does a send complete). In the following, I use "send buffer" for the user-provided buffer to send.

[. . .]

Note that "nonblocking" refers ONLY to whether the data buffer is available for reuse after the call. No part of the MPI specification, for example, mandates concurrent operation of data transfers and computation.

MPI Datatypes

Datatypes: Note that the basic *C* datatypes are not allowed in MPI calls — MPI wraps (some of) these and you must use the MPI wrapper types in MPI calls.

MPI Datatypes

Datatypes: Note that the basic *C* datatypes are not allowed in MPI calls — MPI wraps (some of) these and you must use the MPI wrapper types in MPI calls.

```
MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
```

- There are more data types predefined.
- However, I've never used others than these.
- Note that there is no bool (C++) before MPI-2.
- In theory, heterogeneous hardware supported.
- Support for user-defined data types and padded arrays.

First code

What are the values of `a` and `b` on rank 2?

```
if (rank==0) {
  int a=0;
  MPI_Send(&a, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
  a=1;
  MPI_Send(&a, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
}
if (rank==2) {
  MPI_Status status;
  int a;
  MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  int b;
  MPI_Recv(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
}
```

First code

What are the values of `a` and `b` on rank 2?

```
if (rank==0) {
  int a=0;
  MPI_Send(&a, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
  a=1;
  MPI_Send(&a, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
}
if (rank==2) {
  MPI_Status status;
  int a;
  MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  int b;
  MPI_Recv(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
}
```

- MPI messages *from one rank* never overtake.
- Why is this called SPMD?

Other blocking Sends

- `MPI_Send` is effectively the simplest, but not the only, option for Sending blocking peer-to-peer messages.
- Others include:
  - `MPI_Ssend` — **S**ynchronous, blocking, send
  - `MPI_Bsend` — asynchronous, blocking, **B**uffered send
  - `MPI_Rsend` — **R**eady blocking send
- `MPI_Send` is buffered if the message fits in the MPI-managed buffer (like `MPI_Bsend`), otherwise synchronous (like `MPI_Ssend`)
- `MPI_Rsend` requires the recipient to have already called `MPI_Recv`

> Usage: `MPI_Recv` is always synchronous — it waits until the buffer is filled up with the complete received message.

Excursus/addendum: Buffered sends

```c
int  bufsize;
char *buf = malloc(bufsize);
MPI_Buffer_attach( buf, bufsize );
...
MPI_Bsend( /*... same as MPI_Send ...*/ );
...
MPI_Buffer_detach( &buf, &bufsize );
```

- If you use many tags, many blocking commands, and so forth, you stress your system buffers (variables, MPI layer, hardware)
- **This might lead to deadlocks though your code semantically is correct**
- MPI provides a send routine that buffers explicitly: `MPI_Bsend`
- `MPI_Bsend` makes use of a user-provided buffer to save any messages that can not be immediately sent.
- Buffers explicitly have to be added to MPI and removed at program termination.
- The `MPI_Buffer_detach` call does not complete until all messages are sent.

Concept of building block

- Content
  - Study MPI send and receive
  - Discuss term blocking
  - Introduce predefined data types
  - Study a simple MPI program
- Expected Learning Outcomes
  - The student knows signature and semantics of MPI's send and receive
  - The student knows the predefined MPI data types
  - The student can explain the term blocking
  - The student can write down a simple correct MPI program (incl. the initialisation and the shutdown)
- Material & further reading
  - Gropp et al.: Tutorial on MPI: The Message-Passing Interface
    `https://www.mcs.anl.gov/research/projects/mpi/tutorial/gropp/talk.html`

MPI tags

Tag: A **tag** is a meta attribute of the message when you send it away. The receive command can filter w.r.t. tags.

Message arrival: Two MPI messages *with the same tag may not overtake*.

- With tags, we can make messages overtake each other.
- Tags are typically used to distinguish messages with different semantics.
- Tags are arbitrary positive integers. There is no need to explicitly register them.
- Extreme scale: Too many tags might mess up MPI implementation.
- For sends, real tag is mandatory. Receives may use `MPI_ANY_Tag` (wildcard).

Example 1/3

The following snippet shall run on a rank `p0`:

```
int a=1;
int b=2;
int c=3;
MPI_Send(&a,1,MPI_INT,p1,0,MPI_COMM_WORLD);
MPI_Send(&b,1,MPI_INT,p1,0,MPI_COMM_WORLD);
MPI_Send(&c,1,MPI_INT,p1,1,MPI_COMM_WORLD);
```

The following snippet shall run on a rank `p1`:

```
int u;
int v;
int w;
MPI_Recv(&u,1,MPI_INT,p0,0,MPI_COMM_WORLD,&status);
MPI_Recv(&v,1,MPI_INT,p0,0,MPI_COMM_WORLD,&status);
MPI_Recv(&w,1,MPI_INT,p0,0,MPI_COMM_WORLD,&status);
```

What is the value of u,v,w on rank `p1`?

Example 2/3

The following snippet shall run on a rank `p0`:

```
int a=1;
int b=2;
int c=3;
MPI_Send(&a,1,MPI_INT,p1,0,MPI_COMM_WORLD);
MPI_Send(&b,1,MPI_INT,p1,0,MPI_COMM_WORLD);
MPI_Send(&c,1,MPI_INT,p1,1,MPI_COMM_WORLD);
```

The following snippet shall run on a rank `p1`:

```
int u;
int v;
int w;
MPI_Recv(&u,1,MPI_INT,p0,0,MPI_COMM_WORLD,&status);
MPI_Recv(&v,1,MPI_INT,p0,1,MPI_COMM_WORLD,&status);
MPI_Recv(&w,1,MPI_INT,p0,0,MPI_COMM_WORLD,&status);
```

What is the value of u,v,w on rank `p1`?

Example 3/3

The following snippet shall run on a rank `p0`:

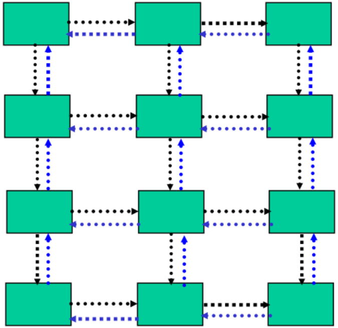
```
int a=1;
int b=2;
int c=3;
MPI_Send(&a,1,MPI_INT,p1,2,MPI_COMM_WORLD);
MPI_Send(&b,1,MPI_INT,p1,1,MPI_COMM_WORLD);
MPI_Send(&c,1,MPI_INT,p1,0,MPI_COMM_WORLD);
```

The following snippet shall run on a rank `p1`:

```
int u;
int v;
int w;
MPI_Recv(&u,1,MPI_INT,p0,0,MPI_COMM_WORLD,&status);
MPI_Recv(&v,1,MPI_INT,p0,1,MPI_COMM_WORLD,&status);
MPI_Recv(&w,1,MPI_INT,p0,2,MPI_COMM_WORLD,&status);
```

What is the value of u,v,w on rank `p1`?

A communication classic: Data exchanged in a grid



- Solution 1: Send everything out (blocking), receive everything blocking
  $\Rightarrow$ High pressure on MPI's buffers
- Solution 2: Split up communication into four phases (up, down, left, right)
  $\Rightarrow$ reduce pressure on communication subsystem
- Solution 3: Use four different tags and use `MPI_ANY` as source
  $\Rightarrow$ Use tags to identify semantics but receive in same order subsystem delivers messages

Concept of building block

- Content
  - Introduce definition of a tag
  - Make definition of not-overtaking explicit and study usage
  - Introduce buffered sends
- Expected Learning Outcomes
  - The student knows definition of tags
  - The student can explain their semantics w.r.t. message arrival
  - The student can use tags in MPI statements