Durham University

Performance Analysis Workshop Series Revision: MPI Refresher

T. Weinzierl [1]

[1] Durham University

23 April 2024

Outline

1. Blocking point-to-point routines
2. MPI progression and message exchange protocols
3. Non-blocking communication
4. Collectives

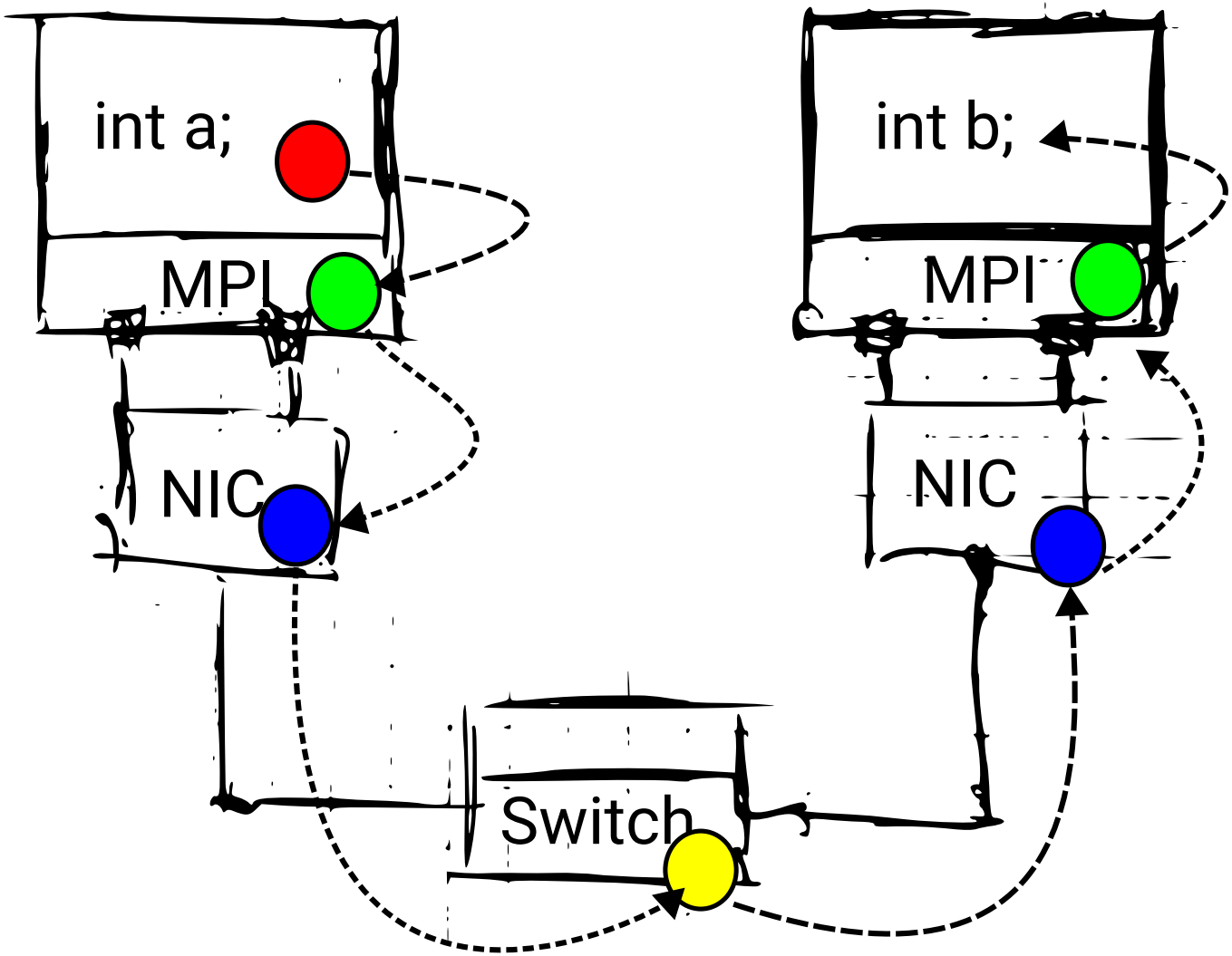
Send

```
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 number 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)

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 the MPI semantics.

Buffers in data exchange



Buffers

The term *buffer* in MPI is an abstraction:

- variables
- user-level buffers
- hardware/system buffers

- MPI implementations avoid copying if ranks are placed on the same node and matching receives are posted (no buffer attempt).
- Otherwise, message runs through all buffers, then through the network, and then bottom-up through all buffers again.
- MPI has to search through buffer to preserve FIFO semantics.

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 number 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)

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.

Two-rank example

```
if (rank==0) {
  int a=0;
  MPI_Send(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  MPI_Recv(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
}
if (rank==1) {
  int a=0;
  MPI_Send(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
}
```

Questionnaire:

- Does code deadlock?
- Where are messages buffered?
- What is implication for performance if rank 1 is delayed?

Common misconception

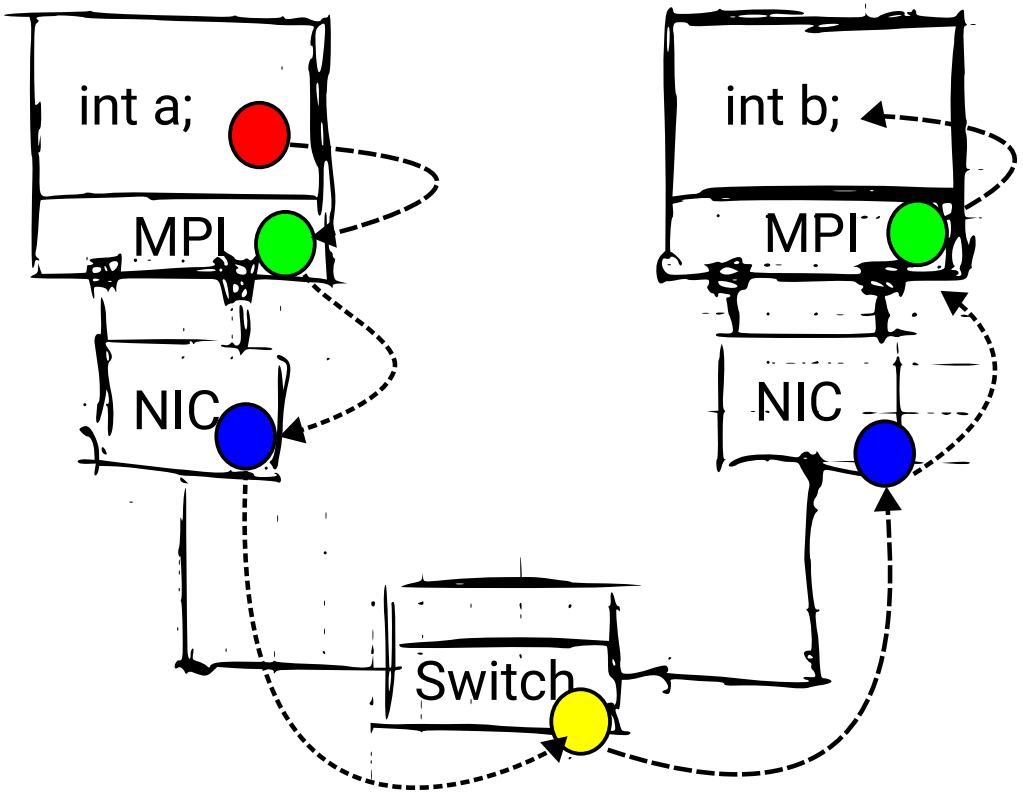
```
if (rank==0) {
  int a=0;
  MPI_Send(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  MPI_Recv(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
}
if (rank==1) {
  int a=0;
  MPI_Send(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
}
```

- Code does not deadlock unless lots of messages or huge messages.
  (one buffer along communication pathway is full)
- Code is as fast as OpenMP et al if ranks are balanced and reside on same node.
- Colleagues that start immediately with non-blocking "tuning" likely have no deep understanding of MPI.

Outline

1. Blocking point-to-point routines
2. MPI progression and message exchange protocols
3. Non-blocking communication
4. Collectives

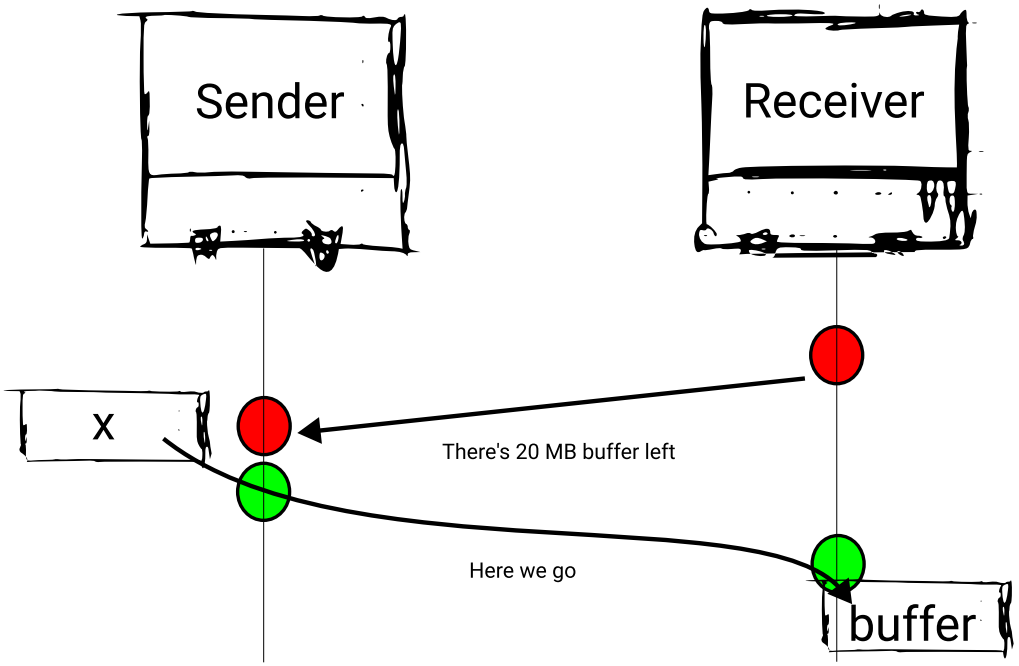
MPI Progression



MPI makes progress (*MPI progression*) if data moves from one buffer into another buffer.

- Progression engine needs CPU cycles (not free)
- In theory for each MPI call
- In practice (implementation) only for receives and probes

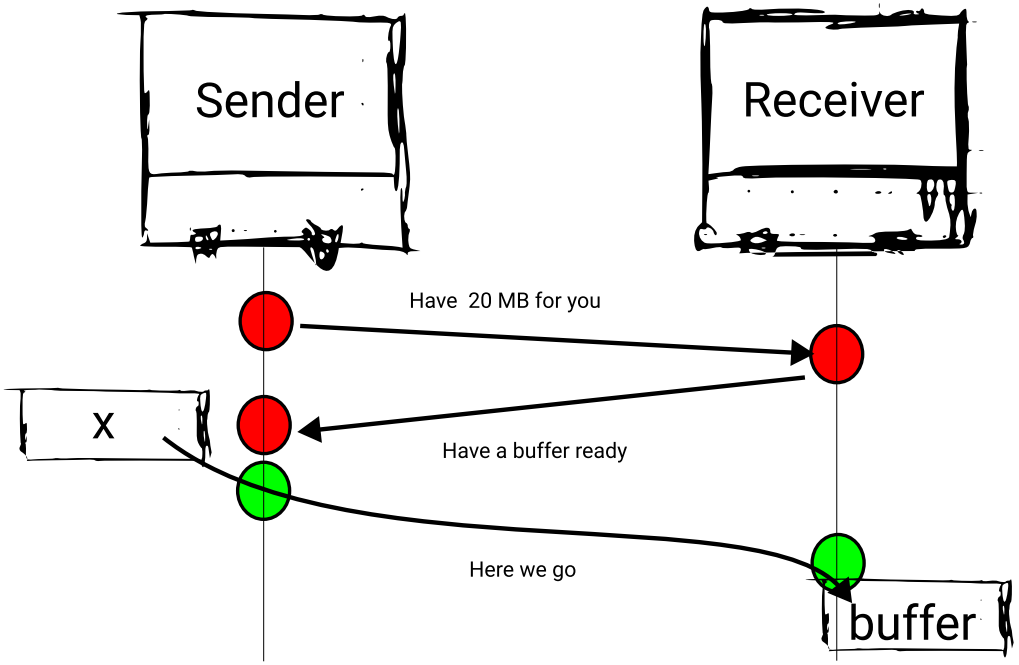
Eager message transfer



- Sender immediately hands data out
- Often realised by meta data agreement behind the scenes
- Works if
  - receive is already posted
  - message is small (*)
  - there are few messages in flight (*)

(*) use some MPI internal buffer for raw data

Rendezvous message transfer



- Sender notified receiver of intention to send
- Often realised by meta data agreement behind the scenes
- Sender waits until receiver gives green light
- Chosen if
  - many messages in flight
  - one buffer full
  - messages too large (*)

(*) there is a environment variable typically

Outline

1. Blocking point-to-point routines
2. MPI progression and message exchange protocols
3. Non-blocking communication
4. Collectives

Nonblocking P2P communication

- Non-blocking commands start with I (immediate return, e.g.)
- Non-blocking means that operation returns immediately though MPI might not have transferred data (might not even have started)
- Buffer thus is still in use and we may not overwrite it
- We explicitly have to validate whether message transfer has completed before we reuse or delete the buffer

```
Create helper variable (handle)
int a = 1;
trigger the send
do some work
check whether communication has completed
a = 2;
...
```

Isend

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

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

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

- Pass additional pointer to object of type `MPI_Request`.
- Non-blocking, i.e. operation returns immediately.
- Check for send completion with `MPI_Wait` or `MPI_Test`.
- `MPI_Irecv` analogous.
- The status object is not required for the receive process, as we have to hand it over to wait or test later.

Misconception

What non-blocking gives you:

- Mechanism to avoid deadlocks if MPI decides to switch to rendezvous
- Ability to avoid hard synchronisation points

What non-blocking does not mean:

- Still up to MPI to decide on eager vs. rendezvous
- Automatic MPI progression in the background

Outline

1. Blocking point-to-point routines
2. MPI progression and message exchange protocols
3. Non-blocking communication
4. Collectives

Definition: collective

Collective operation A collective (MPI) operation is an operation involving many/all nodes/ranks.

- In MPI, a collective operation involves all ranks of one communicator (introduced later)
- For MPI_COMM_WORLD, a collective operation involves all ranks
- Collectives are blocking (though the newest MPI standard introduces non-blocking collectives)
- Blocking collectives always synchronise all ranks, i.e. all ranks have to enter the same collective instruction before any rank proceeds

Classification of collectives

Type of collective	One-to-all	All-to-one	All-to-all
Synchronisation	Barrier		
Communication	Broadcast Scatter	Gather	Allgather
Computation		Reduce	Allreduce

- Synchronisation
- Communication: Synchronisation plus some movement of data
- Computation: Synchronisation plus data movements, where moved data is subject to manipulations

Non-blocking collectives

Type of collective	One-to-all	All-to-one	All-to-all
Synchronisation	Barrier		
Communication	Broadcast Scatter	Gather	Allgather
Computation		Reduce	Allreduce

As soon as we wait, we know:

- Synchronisation: All ranks have passed the barrier
  $\Rightarrow$ At least progress guarantee
- Communication: Global data exchange has finished
  $\Rightarrow$ I/O for example
- Computation: Result is available
  $\Rightarrow$ Residual calculation, e.g., if doesn't feed immediately into next step