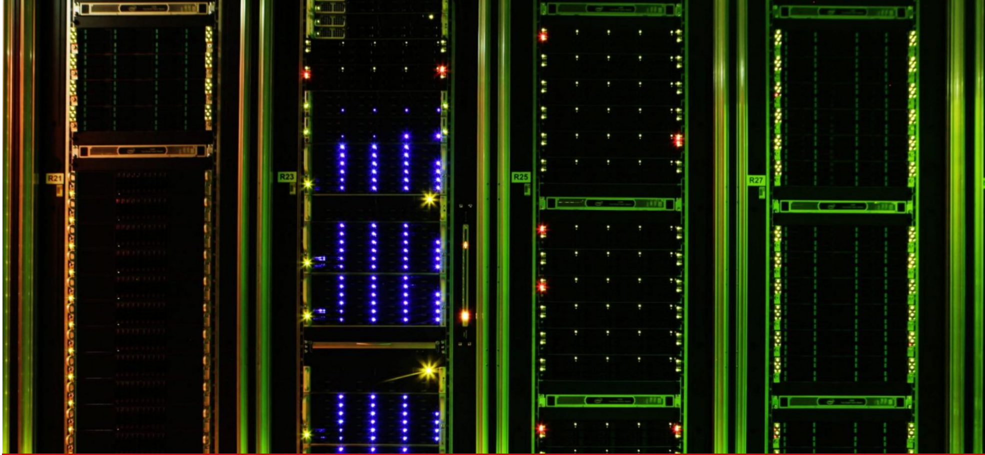


MATH-454 Parallel and High Performance Computing
Lecture 4: MPI basics

Pablo Antolin

Slides of N. Richart, E. Lanti, V. Keller's lecture notes

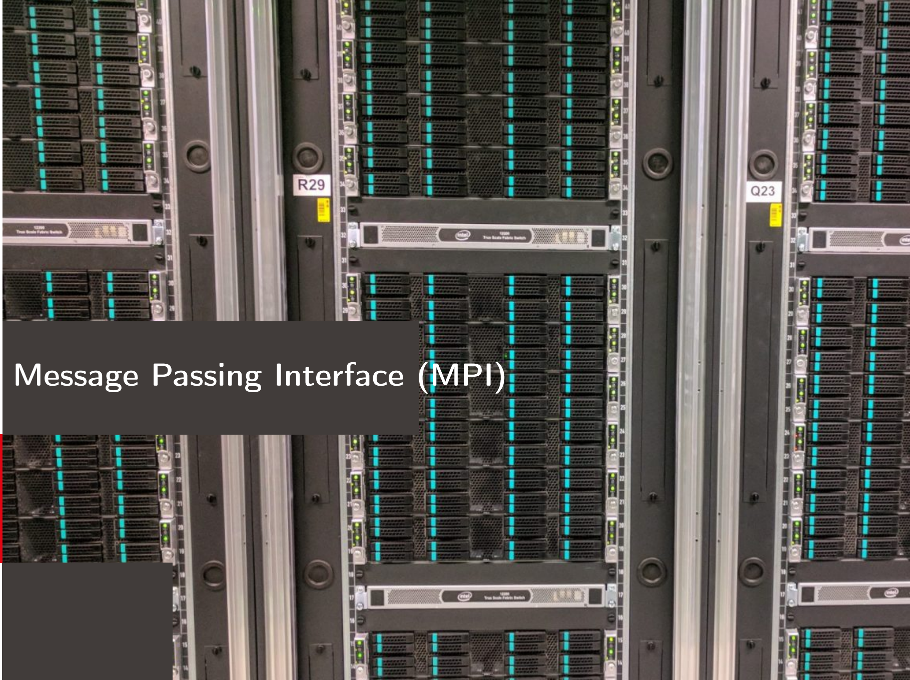
March 20 2025

EPFL

SCITAS

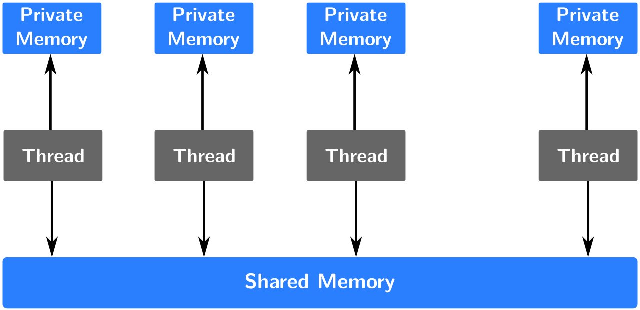
EPFL

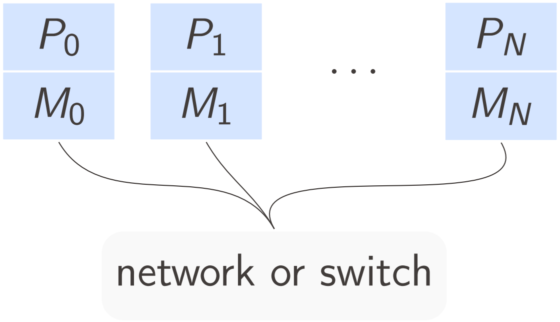
Message Passing Interface (MPI)



Goals of this section

- Introduce distributed memory programming paradigm.
- Point-to-point communications.
- Collective communications.

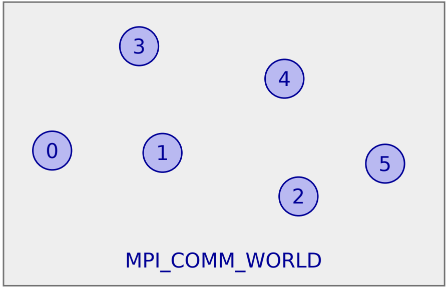

Private Memory
Private Memory
Private Memory
Private Memory
Thread
Thread
Thread
Thread
Shared Memory


P_0
M_0
P_1
M_1
$\cdots$
P_N
M_N
network or switch

- MPI is a *Message-Passing Interface* specification.
- There are many implementations (Intel MPI, OpenMPI, MPICH, MVAPICH, etc.)
- Library interface, not a programming language.
- It is standarized:
  - Defined by the MPI forum
  - Current version is MPI 4.1
- As such, it is portable, flexible, and efficient.
- Interface to C and Fortran in standard. C interface usable with C++. Also MPI bindings for other languages (Python, Julia, MATLAB, etc.).

mpi/hello_mpi.cc

```cpp
#include <iostream>
#include <mpi.h>

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

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

  std::cout << "I am process " << rank << " out of " << size << std::endl;

  MPI_Finalize();
  return 0;
}
```



- MPI code is bounded by a `MPI_Init` and a `MPI_Finalize`.
- MPI starts N processes numbered $0, 1, \ldots, N-1$.
  The number of every process is usually denoted as *rank*.
- They are grouped in a *communicator* of *size* N.
- After init, MPI provides a default communicator called **MPI_COMM_WORLD**.

Types of communications in MPI

- Point-to-Point (One-to-One).
- Collectives (One-to-All, All-to-One, All-to-All).
- One-sided/Shared memory (One-to . . .).
- Blocking and Non-Blocking of all types.

Send/Receive

Syntax

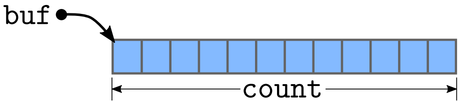
```
int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm);

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

- **buf**: pointer to the data to send/receive.
- **count**: number of elements to send/receive.
- **datatype**: datatype of the data to send/receive.
- **dest**, **source**: the rank of the destination/source of the communication.
- **tag**: a message tag to differentiate the communications.
- **comm**: communicator in which the communication happens.
- **status**: object containing information on the communication.

EPFL

Send/Receive

Details on the buffer



- Buffer is a pointer to the first data (`buf`), a size (`count`), and a `datatype`.
- Datatypes (extract):
  - `MPI_INT`
  - `MPI_UNSIGNED`
  - `MPI_FLOAT`
  - `MPI_DOUBLE`
- For `std::vector<double> vect`:
  - `buf = vect.data()`
  - `count = vect.size()`
  - `datatype = MPI_DOUBLE`

Constants:

- MPI_STATUS_IGNORE: to state that the status is ignored.
- MPI_PROC_NULL: placeholder for the source or destination.
- MPI_ANY_SOURCE: is a wildcard for the source of a receive.
- MPI_ANY_TAG: is a wildcard for the tag of a receive.

MPI_Status:

- Structure containing tag and source

```
MPI_Status status;
std::cout << "Tag: " << status.tag << " - Source: " <<
    status.source << std::endl;
```

- Size of the received buffer can be asked using the status:

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

Send/Receive

Example

mpi/send_recv.cc

```cpp
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

assert(size == 2 && "Works only with 2 procs");

int tag = 0;

if (rank == 0) {
  fill_buffer(buf);
  MPI_Ssend(buf.data(), buf.size(), MPI_INT, 1, tag, MPI_COMM_WORLD);
} else if (rank == 1) {
  MPI_Recv(buf.data(), buf.size(), MPI_INT, 0, tag, MPI_COMM_WORLD,
   ↪  MPI_STATUS_IGNORE);
}
```

Rank 0		Rank 1
send(1,0)	$\longrightarrow$	recv(0,0)
finalize()		finalize()

- MPI_Ssend: (S for Synchronous) function returns when other end posted matching recv and the buffer can be safely reused.
- MPI_Bsend: (B for Buffer) function returns immediately, send buffer can be reused immediately.
- MPI_Rsend: (R for Ready) can be used only when a receive is already listening (waiting).
- MPI_Send: acts like MPI_Bsend on small arrays, and like MPI_Ssend on bigger ones.

- `MPI_Ssend`: (S for Synchronous) function returns when other end posted matching recv and the buffer can be safely reused.
- `MPI_Bsend`: (B for Buffer) function returns immediately, send buffer can be reused immediately.
- `MPI_Rsend`: (R for Ready) can be used only when a receive is already listening (waiting).
- `MPI_Send`: acts like `MPI_Bsend` on small arrays, and like `MPI_Ssend` on bigger ones.

Be careful with deadlocks and race conditions!!

Send/Receive

Deadlock example

mpi/send_recv_deadlock.cc

```cpp
assert(size == 2 && "Works only with 2 procs");
int tag = 0;

if (rank == 0)
{
  fill_buffer(buf);
  MPI_Ssend(buf.data(), buf.size(), MPI_INT, 1, tag, MPI_COMM_WORLD);
  MPI_Recv(buf.data(), buf.size(), MPI_INT, 1, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else if (rank == 1)
{
  MPI_Ssend(buf.data(), buf.size(), MPI_INT, 0, tag, MPI_COMM_WORLD);
  MPI_Recv(buf.data(), buf.size(), MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Rank 0	Rank 1
send(1,0)	send(0,0)
barrier!	barrier!

mpi/send_recv_deadlock_2.cc

```
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// Let's now assume that we may have 2 or more processes.
int tag = 0;

if (rank == 0) {
  fill_buffer(buf);
  MPI_Ssend(buf.data(), buf.size(), MPI_INT, 1, tag, MPI_COMM_WORLD);
} else {
  MPI_Recv(buf.data(), buf.size(), MPI_INT, 0, tag, MPI_COMM_WORLD,
  ↪   MPI_STATUS_IGNORE);
}
MPI_Finalize();
```

With 2 processes:

Rank 0		Rank 1
send(1,0)	$\longrightarrow$	recv(0,0)
finalize()		finalize()

With 3 processes:

Rank 0		Rank 1	Rank 2
send(1,0)	$\longrightarrow$	recv(0,0)	recv(0,0)
finalize()		finalize()	barrier!

Syntax

```c
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);
```

- Combines a send and a receive, to help mitigate deadlocks.
- Has a in-place variant `MPI_Sendrecv_replace`.

Non-blocking send/receive

Syntax

```
int MPI_Isend(const 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);
```

- I for *immediate*.
- `request` in addition to parameters from blocking version.
  It is an object attached to the communication.
- Receive does not have a status.
- The communication starts but is not completed.
- **S**, **B**, and **R** variants are also defined.

Syntax

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

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

- For non-blocking communications, completion should be “manually” checked.
- `MPI_Test` or `MPI_Wait`.
- Send completed means the buffer can be reused.
- Receive completed means the buffer can be read.
- `status` is set at completion.
- `flag` is **true** if completed, **false** otherwise.

mpi/isend_recv.cc

```
MPI_Request request;
if (rank == 0) {
  fill_buffer(buf);
  MPI_Isend(buf.data(), buf.size(), MPI_INT, 1, 0, MPI_COMM_WORLD,
  ↪  &request);
} else if (rank == 1) {
  MPI_Recv(buf.data(), buf.size(), MPI_INT, 0, 0, MPI_COMM_WORLD,
  ↪  MPI_STATUS_IGNORE);
}

// While MPI communications are performed here I can do computations as
↪  long as buf is not modified in rank 0, or read/modified in rank 1.

MPI_Wait(&request, MPI_STATUS_IGNORE);
MPI_Finalize();
```

Non-blocking send/receive

Multiple completions

- MPI_Waitall, MPI_Testall waits or tests completion of all the pending requests.
- MPI_Waitany, MPI_Testany waits or tests completion of one out of many.
- MPI_Waitsome, MPI_Testsome waits or tests completion of all the specified requests.
- For arrays of statuses can use MPI_STATUSES_IGNORE.
- MPI_Request_get_status equivalent to MPI_Test but does not free completed requests.

EPFL

Probing

Syntax

```c
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
    MPI_Status *status);

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status
    *status);
```

- Check incoming messages without receiving.
- Immediate variant returns **true** if matching message exists.
- Can be used in combination with a successive `MPI_Get_count` for deducing the size of an incoming message before actually recieving it. Thus, we can allocate a buffer for holding the message.

Syntax

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

- Collective communications **must** be called by all processes in the communicator.
- Barrier is hard synchronization.
- Avoid as much as possible.

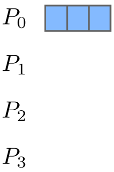
Collective communications

Broadcast

Syntax

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

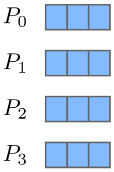
- The **sender** process sends data to every other process.



Syntax

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

- The **sender** process sends data to every other process.



Syntax

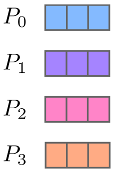
```c
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype
    sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int sender, MPI_Comm comm);
```

- The **sender** process sends a piece of the data to all processes.
- The `sendbuf`, `sendcount`, and `sendtype` are only relevant on the **sender**.

Syntax

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

- The **sender** process sends a piece of the data to all processes.
- The `sendbuf`, `sendcount`, and `sendtype` are only relevant on the **sender**.

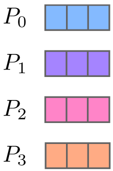


Collective communications
Gather

Syntax

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

- Every process sends its data to the **receiver** process.
- The `recvbuf`, `recvcount`, and `recvtype` are only relevant on the **receiver**.
- `recvcount` is the size per process, not the total size.



Syntax

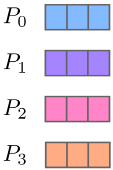
```c
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype
    sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int receiver, MPI_Comm comm);
```

- Every process sends its data to the **receiver** process.
- The `recvbuf`, `recvcount`, and `recvtype` are only relevant on the **receiver**.
- `recvcount` is the size per process, not the total size.

Syntax

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

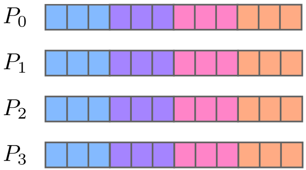
- Every process sends its data to all the other processes.



Syntax

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

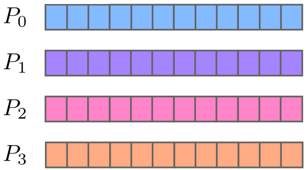
- Every process sends its data to all the other processes.



Syntax

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

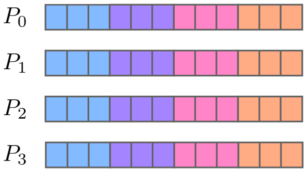
- Every process sends a piece of its data to all the other processes.



Syntax

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

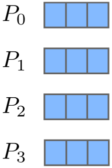
- Every process sends a piece of its data to all the other processes.



Syntax

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

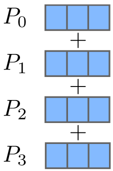
- Data from all processes are reduced on the **receiver** process.
- Common operations being `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD`.
- A `MPI_Allreduce` variant exists where all processes get the results.
- `MPI_IN_PLACE` can be passed to `sendbuf` of **receiver**, for *reduce*, and of all processes, for *allreduce*.



Syntax

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

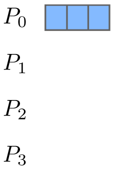
- Data from all processes are reduced on the **receiver** process.
- Common operations being `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD`.
- A `MPI_Allreduce` variant exists where all processes get the results.
- `MPI_IN_PLACE` can be passed to `sendbuf` of **receiver**, for *reduce*, and of all processes, for *allreduce*.



Syntax

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

- Data from all processes are reduced on the **receiver** process.
- Common operations being `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD`.
- A `MPI_Allreduce` variant exists where all processes get the results.
- `MPI_IN_PLACE` can be passed to `sendbuf` of **receiver**, for *reduce*, and of all processes, for *allreduce*.



Syntax

```c
int MPI_Scan(const void *sendbuf, void *recvbuf, int count,
    ↪ MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);

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

- Performs the prefix reduction on data.
- `MPI_Scan`: on process i contains the reduction of values from processes $[0, i]$
- `MPI_Exscan`: on process i contains the reduction of values from processes $[0, i[$.
- `MPI_IN_PLACE` can be passed to `sendbuf`.