CME 213, ME 339—Spring 2021

Eric Darve, ICME, Stanford

"Make everything as simple as possible, but not simpler." — Albert Einstein

MPI process mapping

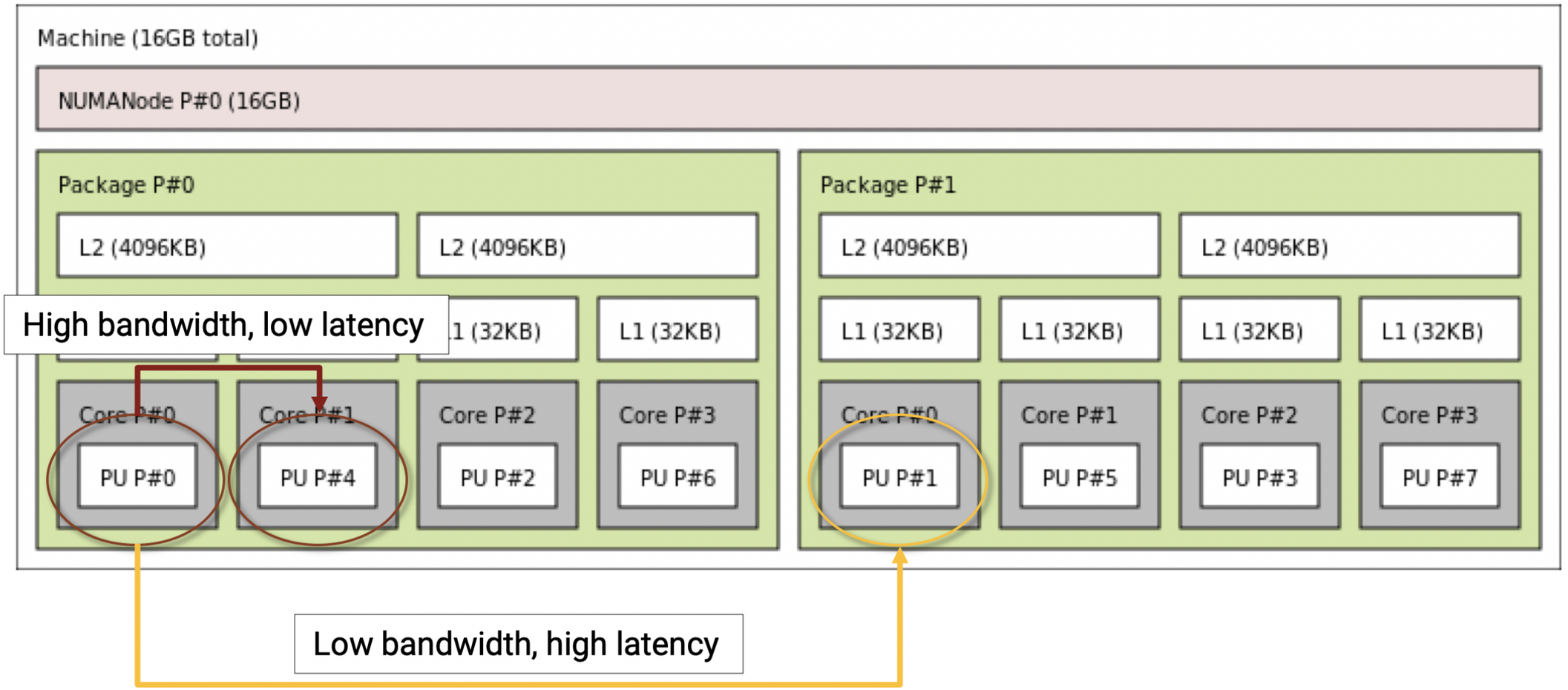
Mapping is important for performance

Lots of interprocess data exchange
→ processes should be close

Lots of memory accesses
→ processes should be far

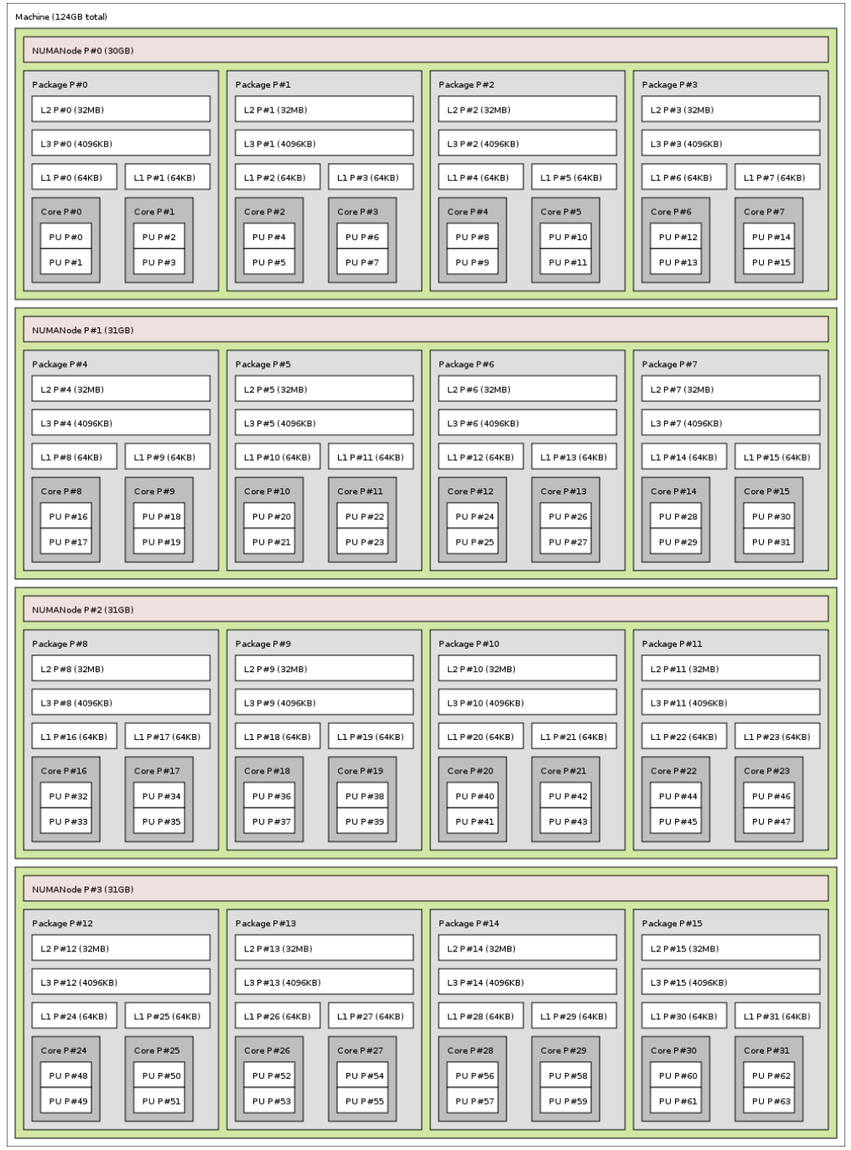
Example 1: 2-package quad-core Xeon

Pre-Nehalem, with 2 dual-core dies into each package


Machine (16GB total)
NUMANode P#0 (16GB)
Package P#0
L2 (4096KB)
L2 (4096KB)
L1 (32KB)
L1 (32KB)
Core P#0
Core P#1
Core P#2
Core P#3
PU P#0
PU P#4
PU P#2
PU P#6
Package P#1
L2 (4096KB)
L2 (4096KB)
L1 (32KB)
L1 (32KB)
L1 (32KB)
L1 (32KB)
Core P#0
Core P#1
Core P#2
Core P#3
PU P#1
PU P#5
PU P#3
PU P#7
High bandwidth, low latency
Low bandwidth, high latency

Example 2: PPC64-based system with 32 cores (each with 2 hardware threads)

The architecture can get complicated real fast!


Machine (124GB total)
NUMANode P#0 (30GB)
Package P#0
L2 P#0 (32MB)
L3 P#0 (4096KB)
L1 P#0 (64KB)
L1 P#1 (64KB)
Core P#0
PU P#0
PU P#1
Core P#1
PU P#2
PU P#3
Package P#1
L2 P#1 (32MB)
L3 P#1 (4096KB)
L1 P#2 (64KB)
L1 P#3 (64KB)
Core P#2
PU P#4
PU P#5
Core P#3
PU P#6
PU P#7
Package P#2
L2 P#2 (32MB)
L3 P#2 (4096KB)
L1 P#4 (64KB)
L1 P#5 (64KB)
Core P#4
PU P#8
PU P#9
Core P#5
PU P#10
PU P#11
Package P#3
L2 P#3 (32MB)
L3 P#3 (4096KB)
L1 P#6 (64KB)
L1 P#7 (64KB)
Core P#6
PU P#12
PU P#13
Core P#7
PU P#14
PU P#15
NUMANode P#1 (31GB)
Package P#4
L2 P#4 (32MB)
L3 P#4 (4096KB)
L1 P#8 (64KB)
L1 P#9 (64KB)
Core P#8
PU P#16
PU P#17
Core P#9
PU P#18
PU P#19
Package P#5
L2 P#5 (32MB)
L3 P#5 (4096KB)
L1 P#10 (64KB)
L1 P#11 (64KB)
Core P#10
PU P#20
PU P#21
Core P#11
PU P#22
PU P#23
Package P#6
L2 P#6 (32MB)
L3 P#6 (4096KB)
L1 P#12 (64KB)
L1 P#13 (64KB)
Core P#12
PU P#24
PU P#25
Core P#13
PU P#26
PU P#27
Package P#7
L2 P#7 (32MB)
L3 P#7 (4096KB)
L1 P#14 (64KB)
L1 P#15 (64KB)
Core P#14
PU P#28
PU P#29
Core P#15
PU P#30
PU P#31
NUMANode P#2 (31GB)
Package P#8
L2 P#8 (32MB)
L3 P#8 (4096KB)
L1 P#16 (64KB)
L1 P#17 (64KB)
Core P#16
PU P#32
PU P#33
Core P#17
PU P#34
PU P#35
Package P#9
L2 P#9 (32MB)
L3 P#9 (4096KB)
L1 P#18 (64KB)
L1 P#19 (64KB)
Core P#18
PU P#36
PU P#37
Core P#19
PU P#38
PU P#39
Package P#10
L2 P#10 (32MB)
L3 P#10 (4096KB)
L1 P#20 (64KB)
L1 P#21 (64KB)
Core P#20
PU P#40
PU P#41
Core P#21
PU P#42
PU P#43
Package P#11
L2 P#11 (32MB)
L3 P#11 (4096KB)
L1 P#22 (64KB)
L1 P#23 (64KB)
Core P#22
PU P#44
PU P#45
Core P#23
PU P#46
PU P#47
NUMANode P#3 (31GB)
Package P#12
L2 P#12 (32MB)
L3 P#12 (4096KB)
L1 P#24 (64KB)
L1 P#25 (64KB)
Core P#24
PU P#48
PU P#49
Core P#25
PU P#50
PU P#51
Package P#13
L2 P#13 (32MB)
L3 P#13 (4096KB)
L1 P#26 (64KB)
L1 P#27 (64KB)
Core P#26
PU P#52
PU P#53
Core P#27
PU P#54
PU P#55
Package P#14
L2 P#14 (32MB)
L3 P#14 (4096KB)
L1 P#28 (64KB)
L1 P#29 (64KB)
Core P#28
PU P#56
PU P#57
Core P#29
PU P#58
PU P#59
Package P#15
L2 P#15 (32MB)
L3 P#15 (4096KB)
L1 P#30 (64KB)
L1 P#31 (64KB)
Core P#30
PU P#60
PU P#61
Core P#31
PU P#62
PU P#63

Mapping relies on two concepts:

1. Mapping: assign process to hardware component
2. Binding: restrict the motion of processes between hardware components

Binding

OS is responsible for assigning a hardware thread to each MPI process.

How do you control the placement of process threads?

`-bind-to object`

This determines how the OS can migrate a process.

Does the process stay with the same hardware thread or is it allowed to migrate to another thread (say on the same socket)?

`bind-to` options

Get all options using `$ mpirun -help`

Option	Hardware element
`hwthread`	bind to hardware thread
`core`	bind to core
`l1cache`	bind to process on L1 cache domain
`l2cache`	bind to process on L2 cache domain
`l3cache`	bind to process on L3 cache domain

Continued

Option	Hardware element
`socket`	bind to socket
`numa`	bind to NUMA domain
`board`	bind to motherboard

map-by

`map-by object`

Skip over object between bindings.

Options:
slot, hwthread, core, L1cache, L2cache, L3cache, socket, numa, board, node

Example usage:

```
mpirun -bind-to core -map-by core -np 4 ./a.out
```

Example output on `icme-gpu`

```
$ salloc --partition=CME -n 4 mpirun --report-bindings --oversubscribe \
         --bind-to hwthread --map-by hwthread ./mpi_hello
rank 0 bound to socket 0[core 0[hwt 0]]: [B./../../../../../../..]
rank 1 bound to socket 0[core 0[hwt 1]]: [.B/../../../../../../..]
rank 2 bound to socket 0[core 1[hwt 0]]: [../B./../../../../../..]
rank 3 bound to socket 0[core 1[hwt 1]]: [../.B/../../../../../..]
```

```
$ salloc --partition=CME -n 12 mpirun --report-bindings --oversubscribe \
         --bind-to hwthread --map-by core ./mpi_hello
rank 0 bound to socket 0[core 0[hwt 0]]: [B./../../../../../../..]
rank 1 bound to socket 0[core 1[hwt 0]]: [../B./../../../../../..]
rank 2 bound to socket 0[core 2[hwt 0]]: [../../B./../../../../..]
rank 3 bound to socket 0[core 3[hwt 0]]: [../../../B./../../../..]
rank 4 bound to socket 0[core 4[hwt 0]]: [../../../../B./../../..]
rank 5 bound to socket 0[core 5[hwt 0]]: [../../../../../B./../..]
rank 6 bound to socket 0[core 6[hwt 0]]: [../../../../../../B./..]
rank 7 bound to socket 0[core 7[hwt 0]]: [../../../../../../../B.]
rank 8 bound to socket 0[core 0[hwt 1]]: [.B/../../../../../../..]
rank 9 bound to socket 0[core 1[hwt 1]]: [../.B/../../../../../..]
rank 10 bound to socket 0[core 2[hwt 1]]: [../../.B/../../../../..]
rank 11 bound to socket 0[core 3[hwt 1]]: [../../../.B/../../../..]
```

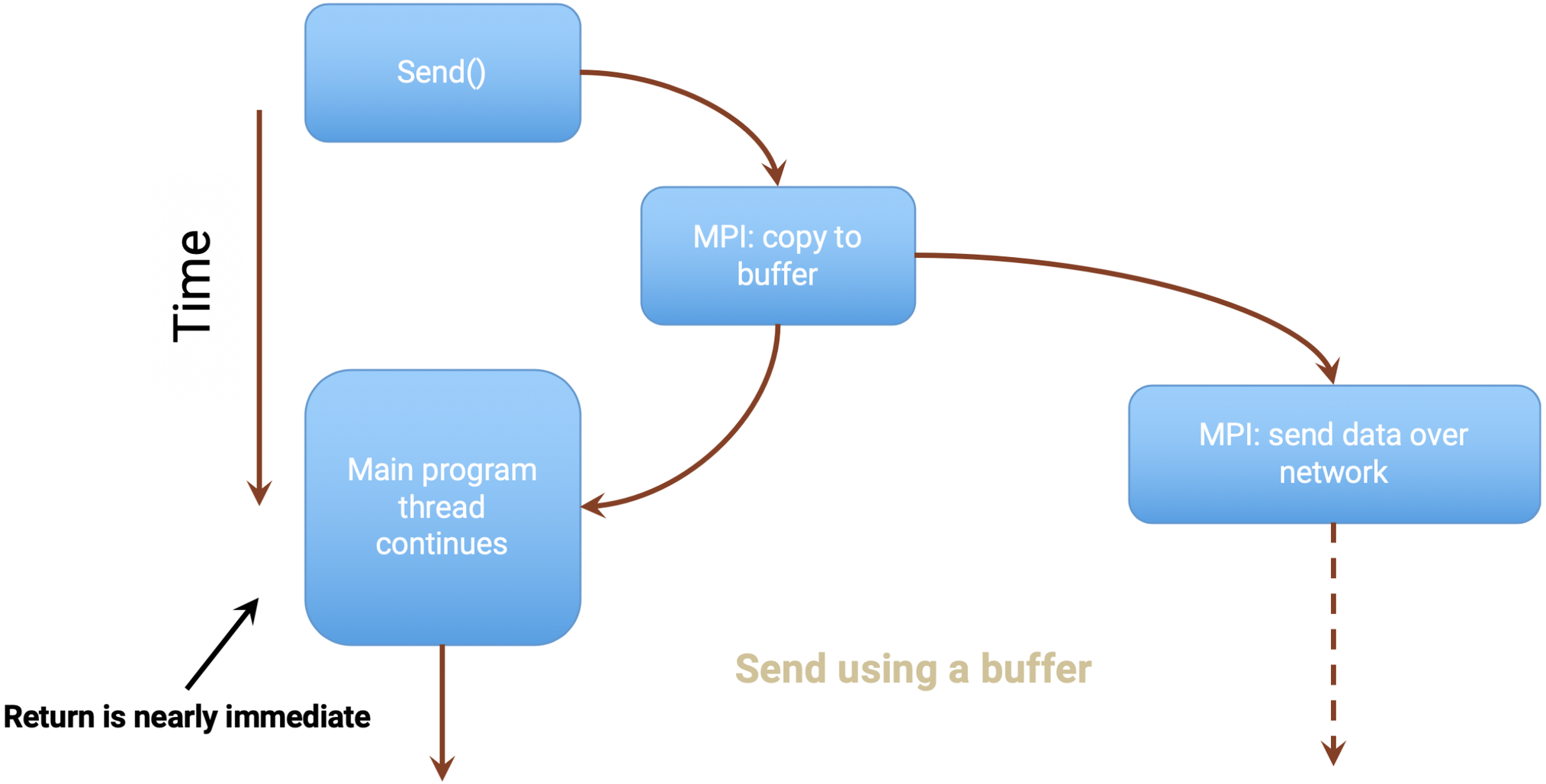
More information

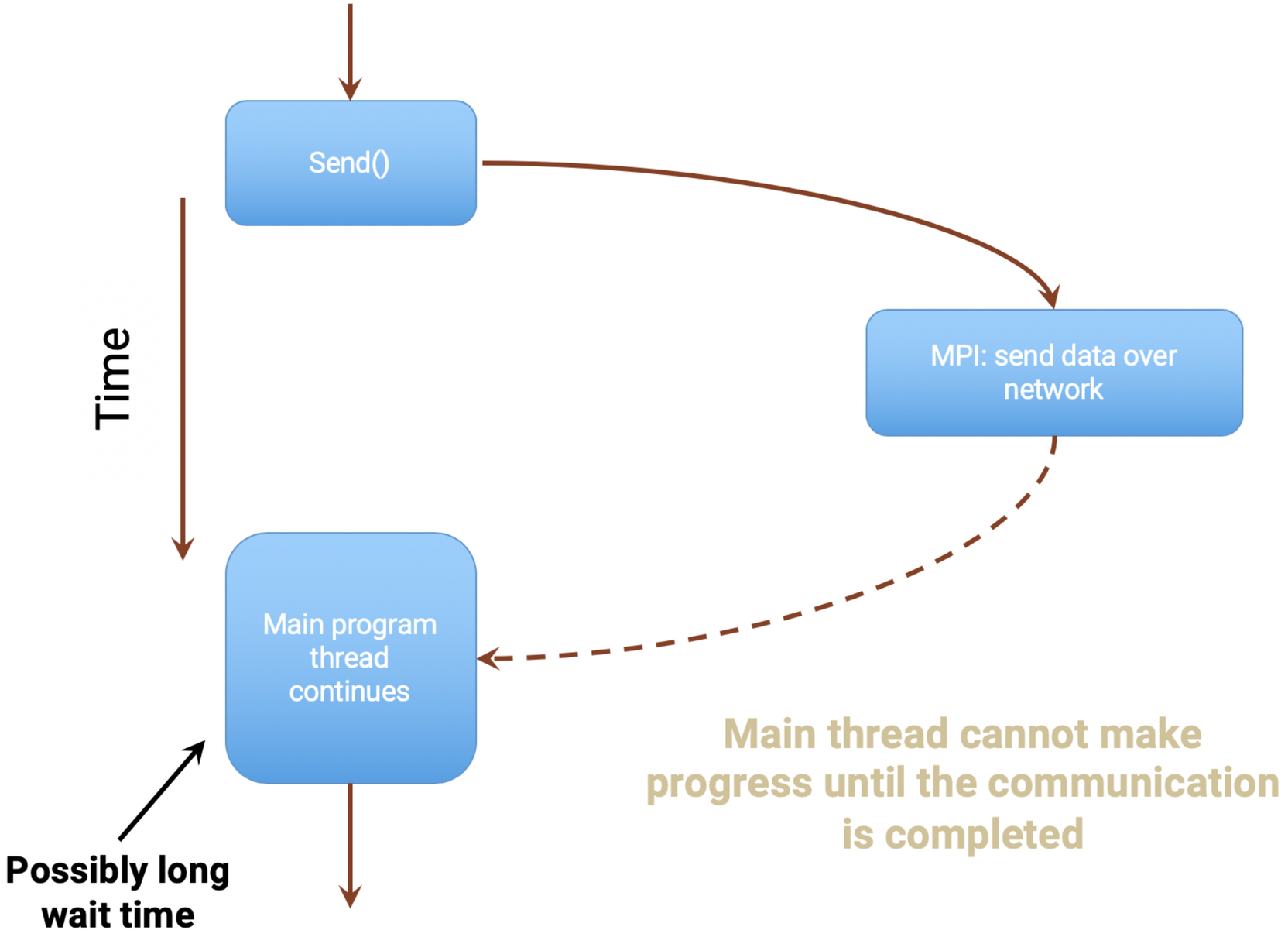
Mapping, Ranking, and Binding

MPI communications

Two strategies:

1. Buffered: send/receive appear to complete immediately
2. Non-buffered: saves memory but requires waiting


Send()
MPI: copy to buffer
MPI: send data over network
Main program thread continues
Time
Return is nearly immediate
Send using a buffer


Send()
MPI: send data over network
Time
Main program thread continues
Possibly long wait time
Main thread cannot make progress until the communication is completed

Summary send/recv with buffering

Send

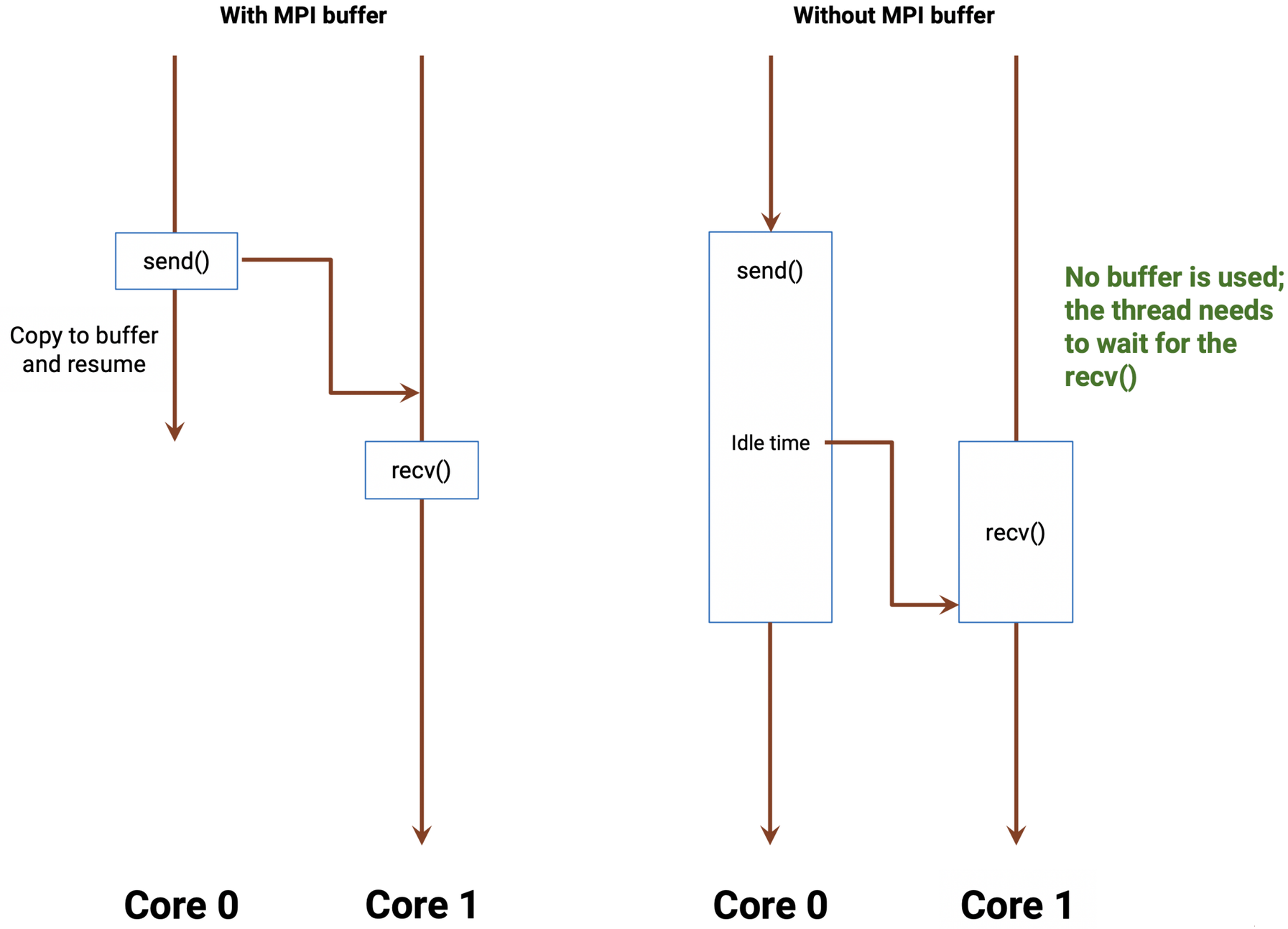
If MPI uses a separate system buffer, the data in `smessage` (user buffer space) is copied (fast); then the main thread resumes.

If MPI does not use a separate system buffer, the main thread must wait until the communication over the network is complete.

Recv

If communication happens before the call, the data is stored in an MPI system buffer, and then simply copied into the user provided rmessage when `recv()` is called.

The user cannot decide whether a buffer is used or not; the MPI library makes that decision based on the resources available and other factors.


With MPI buffer
send()
Copy to buffer
and resume
recv()
Core 0
Core 1
Without MPI buffer
send()
Idle time
recv()
No buffer is used;
the thread needs
to wait for the
recv()
Core 0
Core 1

Send/Recv deadlocks

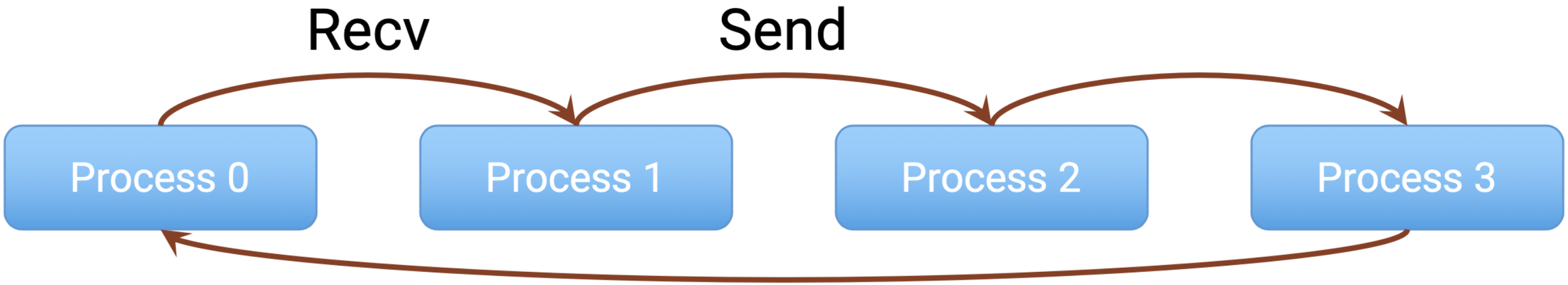
Very easy to achieve with Send/Recv

Send and Recv are both blocking

Process will wait until communication completes

Process 0	Process 1	Deadlock
Recv() Send()	Recv() Send()	Always
Send() Recv()	Send() Recv()	Depends on whether a buffer is used or not
Send() Recv()	Recv() Send()	Secure

Let's demonstrate these implementations on a ring communication example


Recv
Send
Process 0
Process 1
Process 2
Process 3

Deadlock

`ring_DL.cpp`

```
...
MPI_Recv(&number_recv, 1, MPI_INT, rank_sender,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Send(&number_send, 1, MPI_INT, rank_receiver,
            0, MPI_COMM_WORLD);
...
```

Uncertain case

`ring_NS.cpp`

```
...
MPI_Send(&number_send, 1, MPI_INT, rank_receiver,
            0, MPI_COMM_WORLD);
MPI_Recv(&number_recv, 1, MPI_INT, rank_sender,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
...
```

Correct implementation

`ring_SEC.cpp`

```
if (rank % 2 == 0) {
    MPI_Send(&number_send, 1, MPI_INT, rank_receiver,
                0, MPI_COMM_WORLD);
} else {
    MPI_Recv(&number_recv, 1, MPI_INT, rank_sender,
                0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

if (rank % 2 == 1){
    MPI_Send(&number_send, 1, MPI_INT, rank_receiver,
                0, MPI_COMM_WORLD);
} else {
    MPI_Recv(&number_recv, 1, MPI_INT, rank_sender,
                0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Other implementation using `Sendrecv`

`ring_SR.cpp`

```
  MPI_Sendrecv(&number_send, 1, MPI_INT, rank_receiver, 0, &number_recv, 1,
               MPI_INT, rank_sender, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Non-blocking MPI communications

Blocking

What we have seen so far

Process waits until MPI command completes

Advantages

- Simple to use.
- Issue command; once code returns, you know that the task is done (at least the resource is usable).
- Efficient.

However, this is too restrictive.

When communications are happening, you probably want to do something else, such as do some useful computation or issue other communications.

This is called overlapping communication and computation.

More generally, instead of blocking and wait for some data to perform the next task, you want to work on all the tasks for which data is available.

Then, check periodically for status of communication.

Non-blocking communications are also safer and help avoid deadlocks.

How to use non-blocking communications

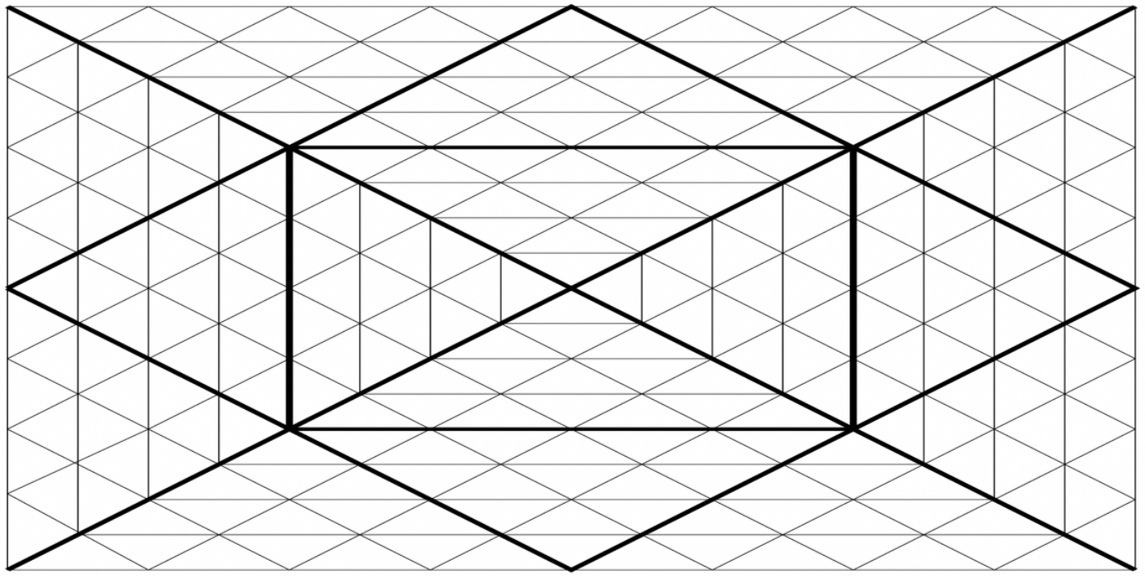
Online documentation

`MPI_Isend`

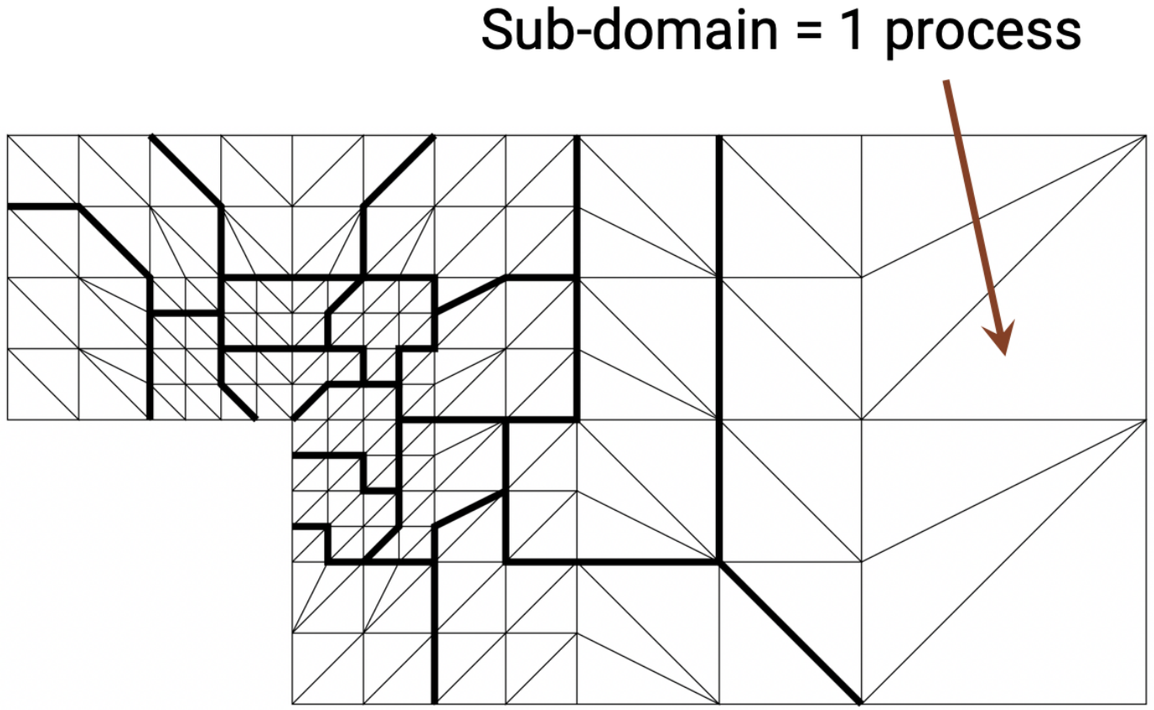
`MPI_Irecv`

`MPI_Test` and `MPI_Wait`

Motivating example



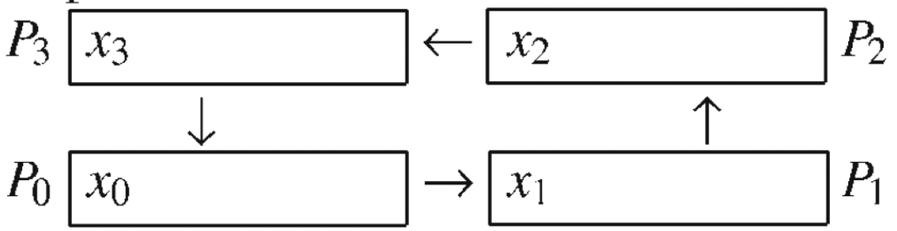
2D rectangular domain



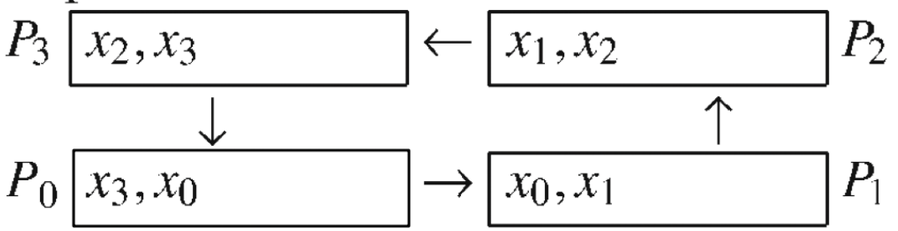
General domain

Gather ring example

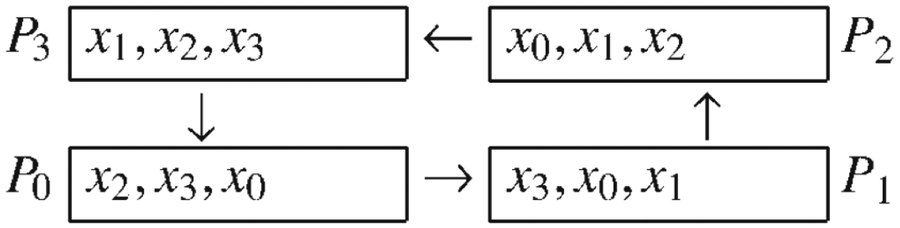
step 1



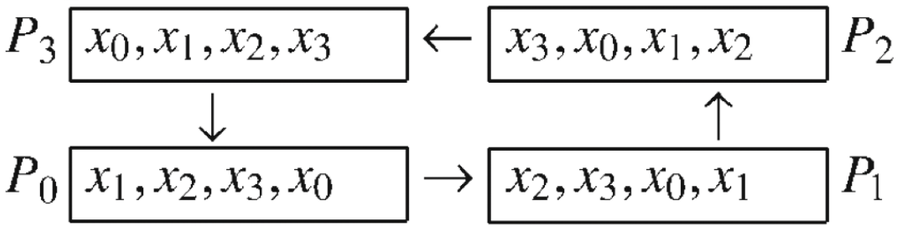
step 2



step 3



step 4



gather_ring.cpp

```
vector<MPI_Request> send_req(nproc - 1);
for (int i = 0; i < nproc - 1; ++i) {
    // Send to the right: Isend
    int *p_send = &numbers[(rank - i + nproc) % nproc];
    MPI_Isend(p_send, 1, MPI_INT, rank_receiver, 0, MPI_COMM_WORLD,
              &send_req[i]);
    // We can proceed; no need to wait now.
    // Receive from the left: Recv
    int *p_recv = &numbers[(rank - i - 1 + nproc) % nproc];
    MPI_Recv(p_recv, 1, MPI_INT, rank_sender, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    // We need to wait; we cannot move forward until we have that data.
}
```

The key MPI routines

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

`MPI_Request*` used to get information later on about the status of that operation.

```
int MPI_Irecv(void* buf, int count,
    MPI_Datatype datatype,
    int source, int tag,
    MPI_Comm comm, MPI_Request *request)
```

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

`flag` True if operation completed (logical)

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

Waits (blocks) for an MPI send or receive to complete

MPI send modes

Optimization!

Three main algorithmic variants:

1. `Buffered`—MPI uses a buffer to avoid blocking
2. `Eager`—MPI will try to send data immediately whether or not a `Recv` has been posted. Works well for small messages.
3. `Rendez-vous`—Send data only when `Recv` has been posted; buffering is not needed; requires a synchronization of the two processes

Online documentation

MPI standard Send

`MPI_Send`

Message size	Strategy
Small messages	eager
Large messages	rendez-vous

User has no control

Bsend

Send with user-specified buffering | `MPI_Bsend`

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

Allows the user to send messages without worrying about whether they are buffered.

The user must have provided buffer space using
`MPI_Buffer_attach(void *buf, int size)`

Ssend

Synchronous send; `rendez-vous` | `MPI_Ssend`

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

Blocks until buffer in sending task is free for reuse **and** destination process has started to receive message. Best performance for data transfer.

Can be used to detect potential deadlocks hidden by MPI buffering.

Rsend

Ready send; `eager` | `MPI_Rsend`

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

A ready send may only be called if the user can guarantee that a receive is already posted. It is an error if the receive is not posted before the ready send is called.

Uncommon

Send Modes	MPI function	Completion Condition
Standard send	`MPI_Send`	Message sent (receiver state unknown)
Buffered send	`MPI_Bsend`	Always completes, irrespective of the receiver
Synchronous send	`MPI_Ssend`	Only completes when the receive has completed
Ready send	`MPI_Rsend`	May be used only when the matching receive has already been posted

Useful resources

- LLNL tutorial
- LLNL MPI performance
- MPI standard version v3.1
- Open MPI documentation v4.1