CME 213

SPRING 2019

Eric Darve

Process mapping

It is important to control the binding and mapping of the processes.

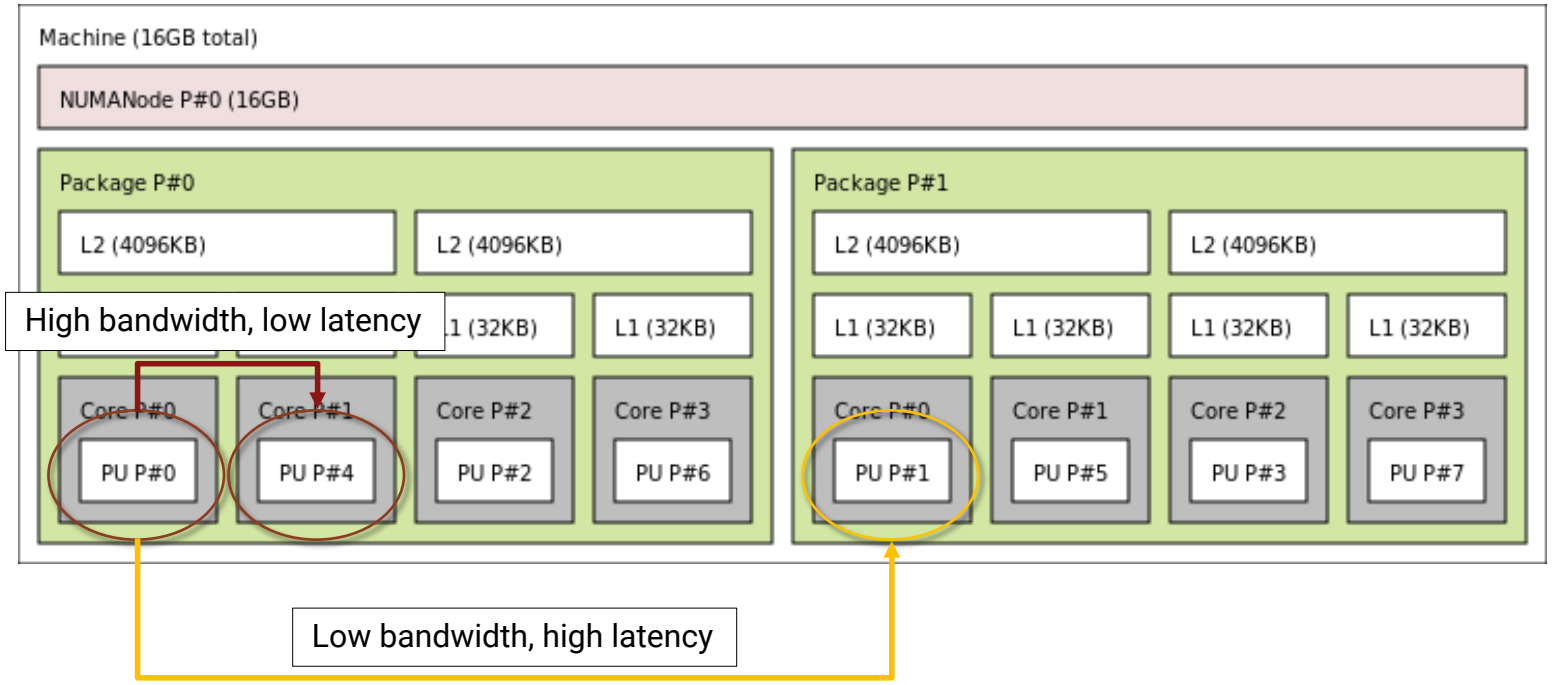
- Mapping: assign a starting location for a thread, e.g., which core or socket?
- Binding: the OS can move threads between cores or sockets. How should we restrict the movement of a thread?
  - a specific hardware thread
  - any hardware thread on a core
  - any hardware thread on a socket
  - any hardware thread on a NUMA domain

Why mapping? Optimization!

- Mapping of processes is important.
- You may want to use many nodes to use more memory.
- You may want to have all threads on the same socket for faster cache sharing access.
- You may want to have threads on different sockets to better utilize the memory buses (NUMA).
- You may want to have as many processes per node as the number of GPU processors.
- You may want to use OpenMP with your MPI code.

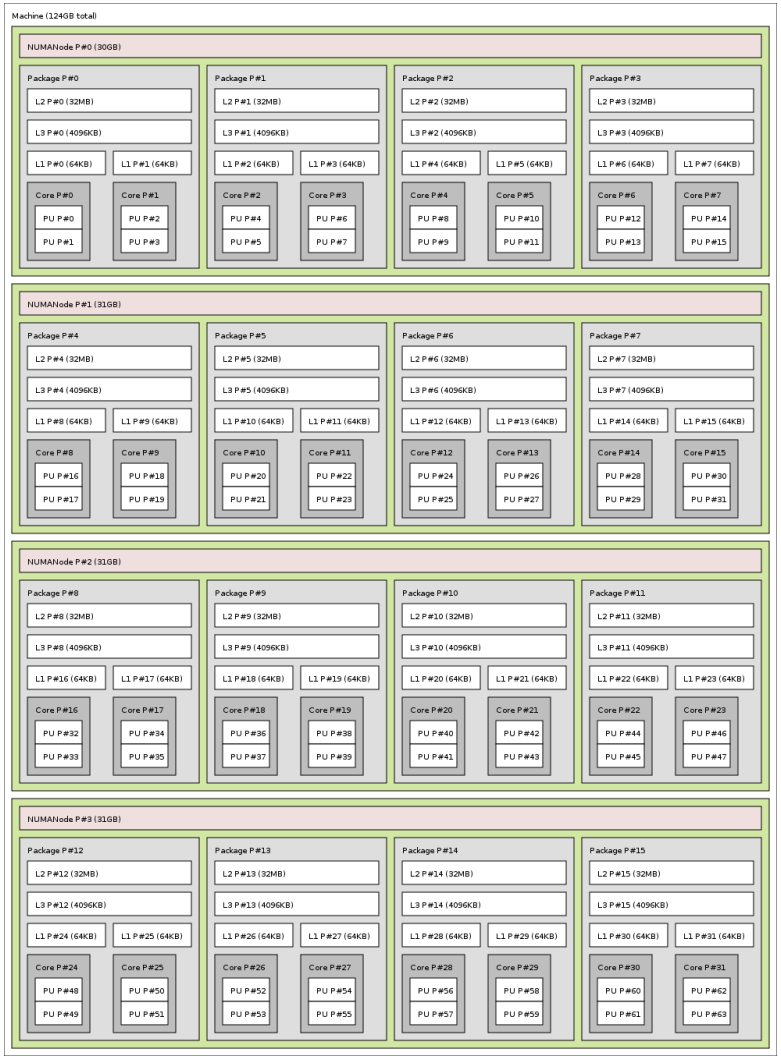
- General guidelines: if two threads exchange a lot of data they should be nearby in the system (high bandwidth).

Example 1: 2 quad-core



2-package quad-core Xeon (pre-Nehalem, with 2 dual-core dies into each package)

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



Process 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
```

- Specify the size of the hardware element to restrict each process thread to.
- 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 are `[mpirun -help]`

`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
`socket`	bind to socket
`numa`	bind to NUMA domain
`board`	bind to motherboard

map-by

map-by object

Skip over **object** between bindings.

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

Usage example:

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

mpiexec --report-bindings --oversubscribe …

if you want information on bindings, and to assign more than one process per hardware thread.

https://www.open-mpi.org/doc/current/man1/mpirun.1.php

--bind-to hwthread --map-by socket

```
mpirun --report-bindings --oversubscribe -mca btl ^openib -n 6 --bind-to hwthread --map-by socket ./mpi_hello

[mpi96:04035] MCW rank 0 bound to socket 0[core 0[hwt 0]]:
[B./../../../../../../../../../../../../../../../../../../../../../..][../../../../../../../../../../../../../../../../../../../../../../..]

[mpi96:04035] MCW rank 1 bound to socket 1[core 24[hwt 0]]:
[../../../../../../../../../../../../../../../../../../../../../../..][B./../../../../../../../../../../../../../../../../../../../../../..]

[mpi96:04035] MCW rank 2 bound to socket 0[core 0[hwt 1]]:
[.B/../../../../../../../../../../../../../../../../../../../../../..][../../../../../../../../../../../../../../../../../../../../../../..]

[mpi96:04035] MCW rank 3 bound to socket 1[core 24[hwt 1]]:
[../../../../../../../../../../../../../../../../../../../../../../..][.B/../../../../../../../../../../../../../../../../../../../../../..]

[mpi96:04035] MCW rank 4 bound to socket 0[core 1[hwt 0]]:
[../B./../../../../../../../../../../../../../../../../../../../../..][../../../../../../../../../../../../../../../../../../../../../../..]

[mpi96:04035] MCW rank 5 bound to socket 1[core 25[hwt 0]]:
[../../../../../../../../../../../../../../../../../../../../../../..][../B./../../../../../../../../../../../../../../../../../../../../..]
```

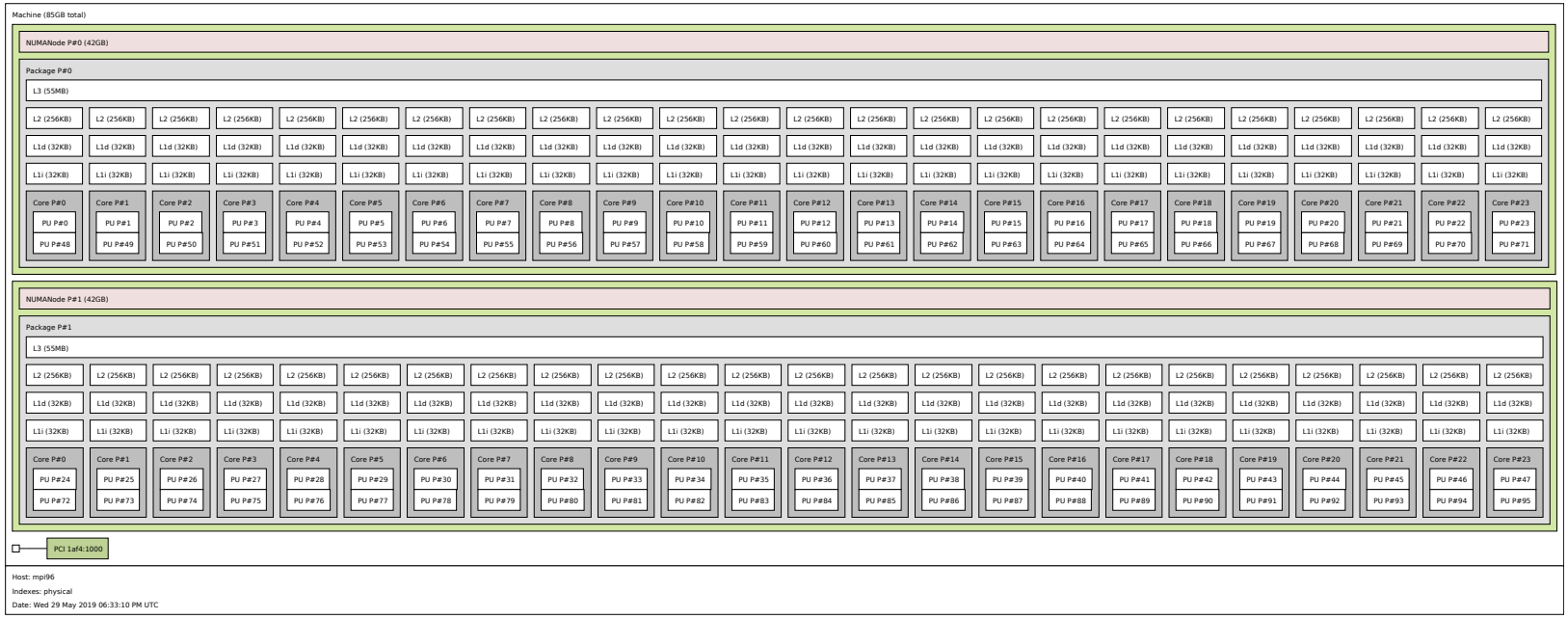
--bind-to hwthread --map-by core

```
mpirun --report-bindings --oversubscribe -mca btl ^openib -n 6 --bind-to hwthread --map-by core ./mpi_hello

[mpi96:04049] MCW rank 0 bound to socket 0[core 0[hwt 0]]:
[B./../../../../../../../../../../../../../../../../../../../..][../../../../../../../../../../../../../../../../../../../..]

[mpi96:04049] MCW rank 1 bound to socket 0[core 1[hwt 0]]:
[../B./../../../../../../../../../../../../../../../../../../..][../../../../../../../../../../../../../../../../../../../..]

[mpi96:04049] MCW rank 2 bound to socket 0[core 2[hwt 0]]:
[../../B./../../../../../../../../../../../../../../../../../..][../../../../../../../../../../../../../../../../../../../..]

[mpi96:04049] MCW rank 3 bound to socket 0[core 3[hwt 0]]:
[../../../B./../../../../../../../../../../../../../../../../..][../../../../../../../../../../../../../../../../../../../..]

[mpi96:04049] MCW rank 4 bound to socket 0[core 4[hwt 0]]:
[../../../../B./../../../../../../../../../../../../../../../..][../../../../../../../../../../../../../../../../../../../..]

[mpi96:04049] MCW rank 5 bound to socket 0[core 5[hwt 0]]:
[../../../../../B./../../../../../../../../../../../../../../..][../../../../../../../../../../../../../../../../../../../..]
```

--bind-to socket --map-by socket

```
mpirun --report-bindings --oversubscribe -mca btl ^openib -n 6 --bind-to socket --map-by core ./mpi_hello

[mpi96:04091] MCW rank 0 bound to socket 0[core 0[hwt 0-1]], socket 0[core 1[hwt 0-1]], socket 0[core 2[hwt 0-1]], socket 0[core 3[hwt 0-1]], socket 0[core 4[
[BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../../../../../../../../../../../../../../../../../..]

[mpi96:04091] MCW rank 1 bound to socket 0[core 0[hwt 0-1]], socket 0[core 1[hwt 0-1]], socket 0[core 2[hwt 0-1]], socket 0[core 3[hwt 0-1]], socket 0[core 4[
[BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB/BB][../../../../../../../../../../../../../../../../../../../../../../../..]

[mpi96:04091] MCW rank 2 bound to socket 0[core 0[hwt 0-1]], socket 0[core 1[hwt 0-1]], socket 0[core 2[hwt 0-1]], socket 0[core 3[hwt 0-1]], socket 0[core 4[

[mpi96:04091] MCW rank 3 bound to socket 0[core 0[hwt 0-1]], socket 0[core 1[hwt 0-1]], socket 0[core 2[hwt 0-1]], socket 0[core 3[hwt 0-1]], socket 0[core 4[

[mpi96:04091] MCW rank 4 bound to socket 0[core 0[hwt 0-1]], socket 0[core 1[hwt 0-1]], socket 0[core 2[hwt 0-1]], socket 0[core 3[hwt 0-1]], socket 0[core 4[

[mpi96:04091] MCW rank 5 bound to socket 0[core 0[hwt 0-1]], socket 0[core 1[hwt 0-1]], socket 0[core 2[hwt 0-1]], socket 0[core 3[hwt 0-1]], socket 0[core 4[
```

lstopo --output-format pdf > lstopo.pdf



hwloc-info

```
darve@mpi96:~/MPI$ hwloc-info
depth 0:          1 Machine (type #1)
 depth 1:         2 NUMANode (type #2)
  depth 2:        2 Package (type #3)
   depth 3:       2 L3Cache (type #4)
    depth 4:      48 L2Cache (type #4)
     depth 5:     48 L1dCache (type #4)
      depth 6:    48 L1iCache (type #4)
       depth 7:   48 Core (type #5)
        depth 8:          96 PU (type #6)
Special depth -3:         1 Bridge (type #9)
Special depth -4:         1 PCI Device (type #10)
```

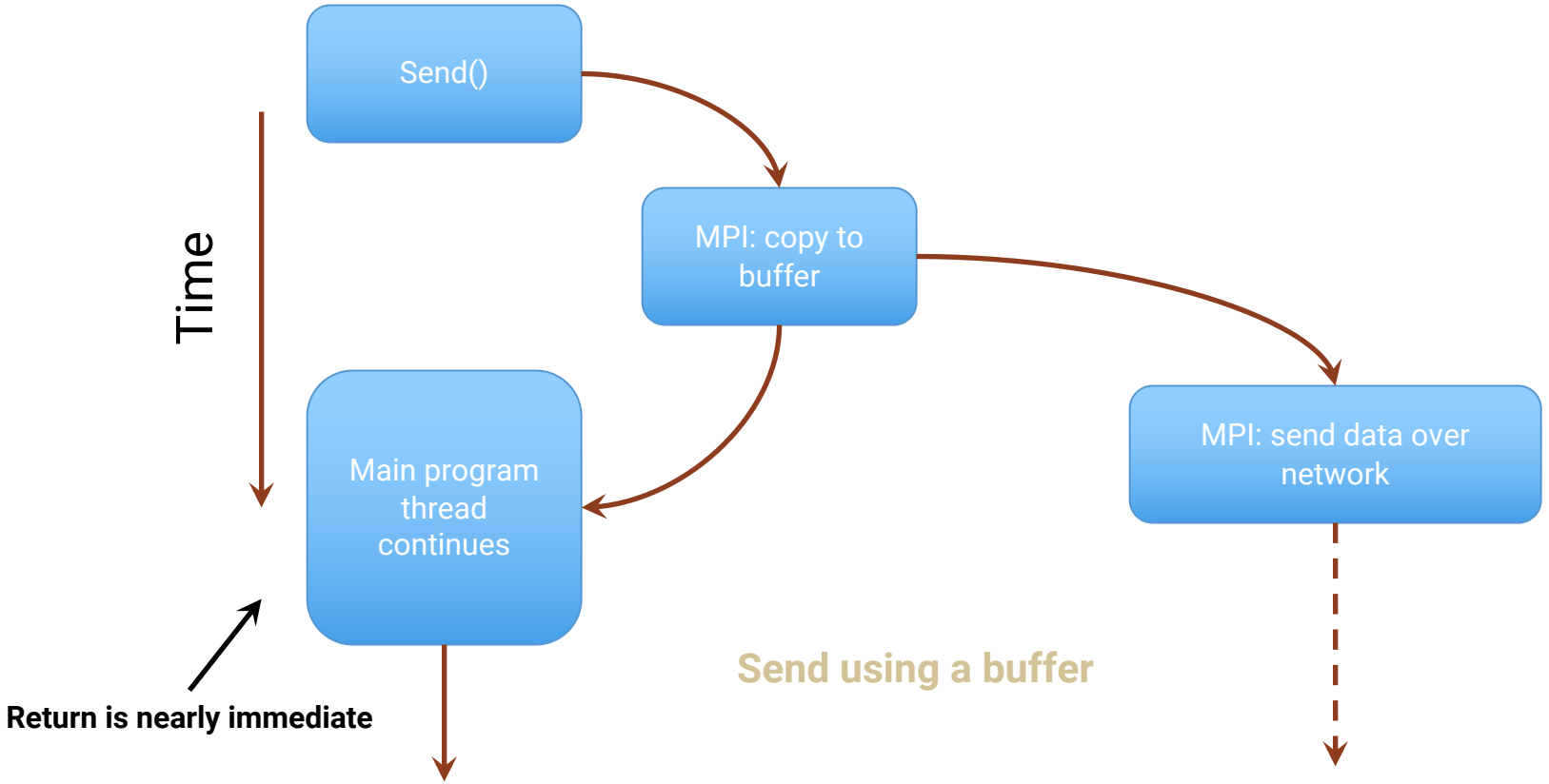
MPI
Point-to-point communication

Point-to-point communication

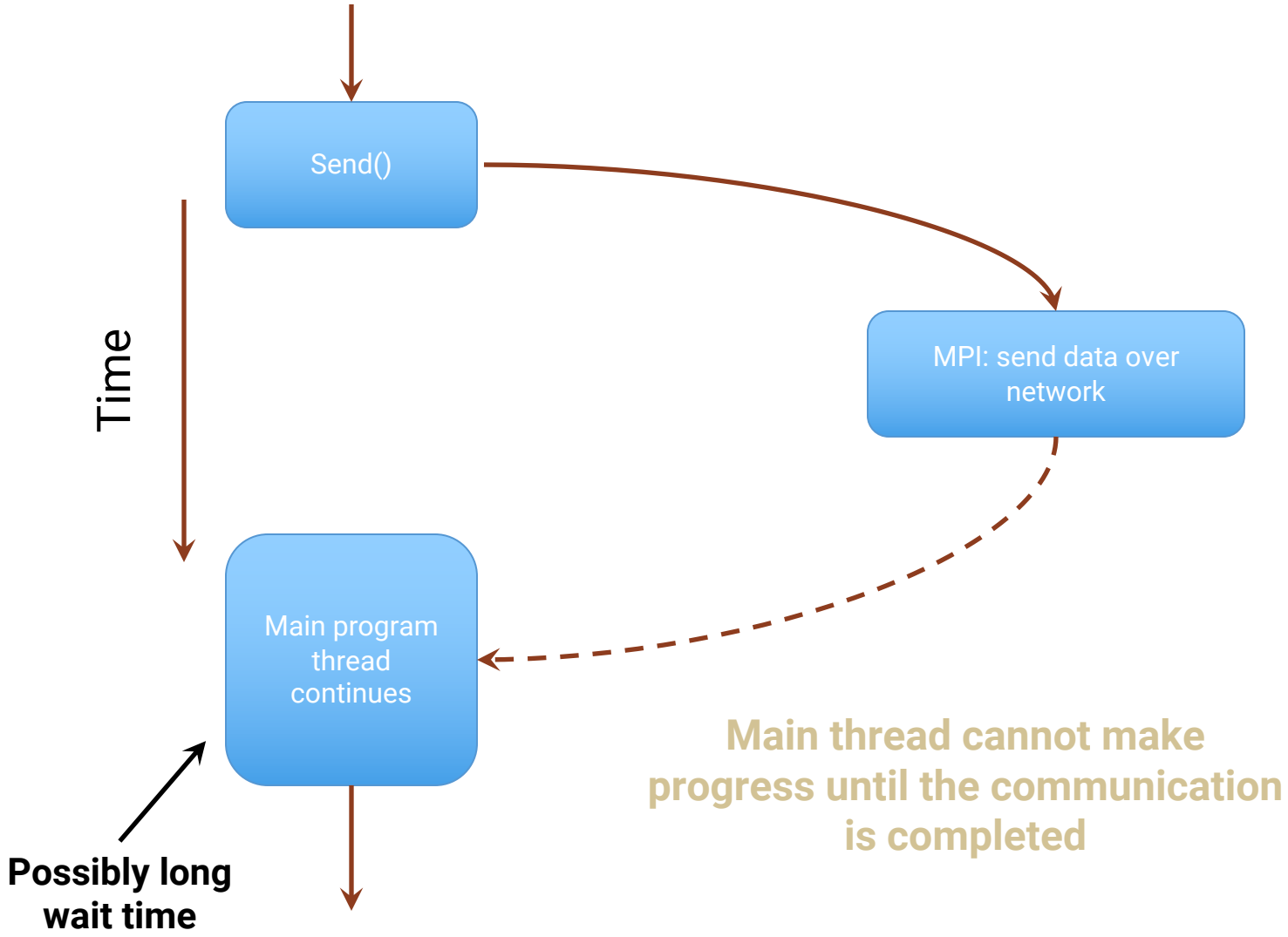
- There are a few technical details to understand regarding communication.
- This is important for performance and to understand whether a deadlock may occur in your program or not.
- Two main concepts:
  - Blocking/non-blocking
  - Synchronous/asynchronous (less common)

Two ways to communicate
1) using an MPI system buffer

To optimize the communication the MPI library uses two different strategies for communication:
buffered and non-buffered.

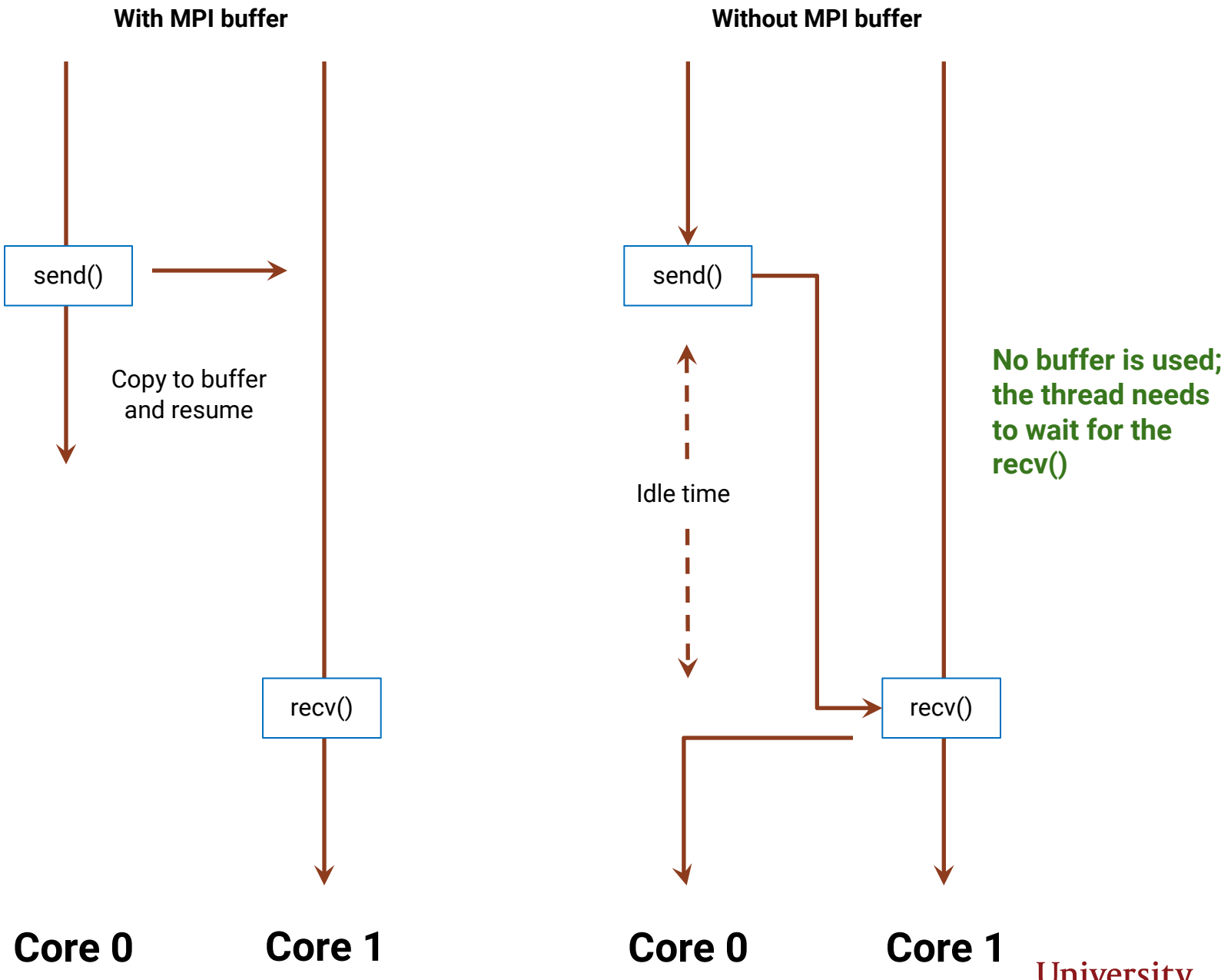


2) Without a buffer



What is the difference?

- Send and Recv are blocking operations:
  - The call does not return until the resources become available again
  - `Send`: data in buffer can be changed
  - `Recv`: data in buffer is available and can be used
- `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

Deadlocks

Deadlocks

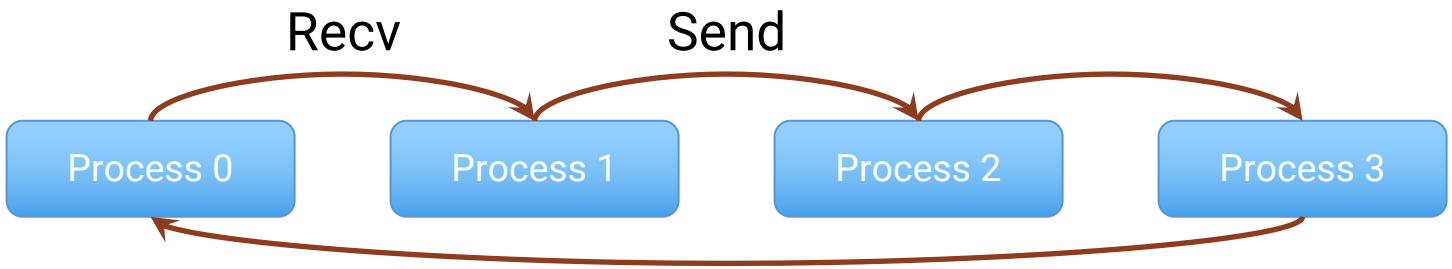
- Because we use blocking routines, deadlocks can occur:

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



- See MPI codes in **`mpi_deadlock/`** and diagram on next slide.
- Secure implementation: code is guaranteed to never deadlock; this should be true independent of whether buffers are used or not.

Ring communication



Code with deadlock

```
// Receive from the lower process and send to the higher process.
int rank_sender = rank==0 ? world_size-1 : rank-1;
MPI_Recv(&number_recv, 1, MPI_INT, rank_sender,
        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("Process %d received \t %2d from process %d\n", rank,
    number_recv, rank_sender);

int rank_receiver = rank==world_size-1 ? 0 : rank + 1;
MPI_Send(&number_send, 1, MPI_INT, rank_receiver,
        0, MPI_COMM_WORLD);
printf("Process %d sent \t\t %2d to   process %d\n", rank,
    number_send, rank_receiver);
```

“Non-secure” code; can potentially deadlock

```
// Receive from the lower process and send to the higher process.
int rank_receiver = rank==world_size-1 ? 0 : rank + 1;
MPI_Send(&number_send, 1, MPI_INT, rank_receiver,
        0, MPI_COMM_WORLD);
printf("Process %d sent \t\t %2d to   process %d\n", rank,
    number_send, rank_receiver);

int rank_sender = rank==0 ? world_size-1 : rank-1;
MPI_Recv(&number_recv, 1, MPI_INT, rank_sender,
        0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("Process %d received \t %2d from process %d\n", rank,
    number_recv, rank_sender);
```

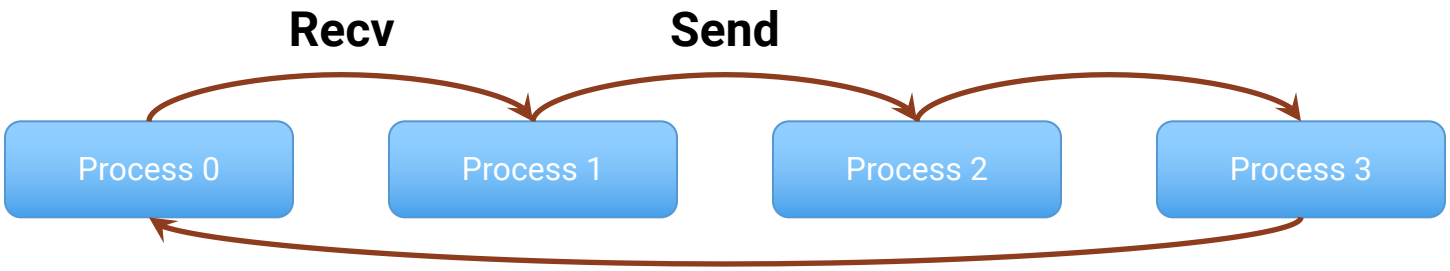
Correct code

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

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

Variant using `MPI_Sendrecv`

```
int rank_receiver = rank==world_size-1 ? 0 : rank + 1;
int rank_sender   = rank==0 ? world_size-1 : rank-1;
MPI_Sendrecv(&number_send, 1, MPI_INT, rank_receiver, 0,
             &number_recv, 1, MPI_INT, rank_sender, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

MPI_Sendrecv




```
MPI_Sendrecv(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)
```

NON-BLOCKING COMMUNICATIONS

Blocking vs non-blocking

Blocking = you process one communication/task at a time.

Non blocking: communicate while doing something else. Regularly check whether comm has completed.

Blocking vs non-blocking

Blocking communications are convenient:

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

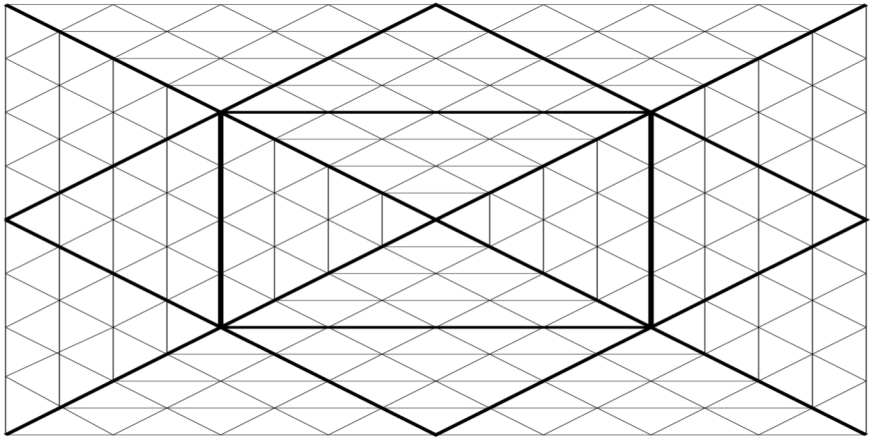
However, this is too restrictive.

- 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 communications and computations.
- Non-blocking communications are safer and avoid deadlocks.

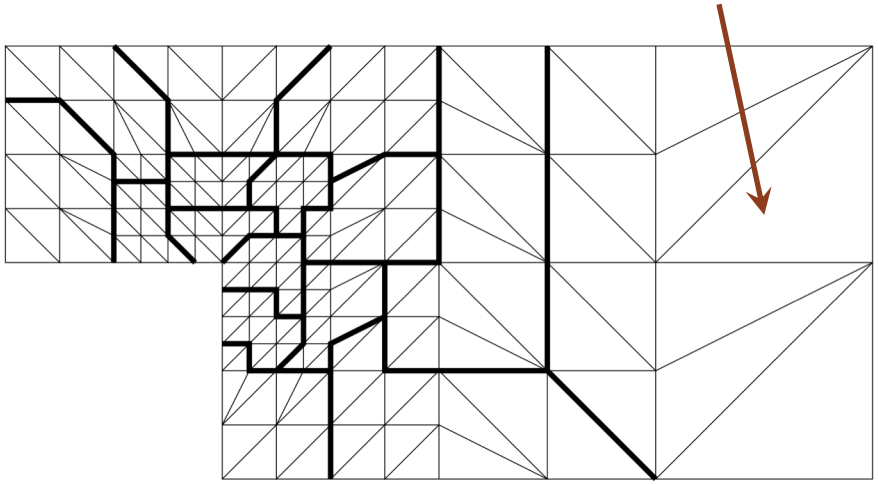
Non-blocking communications can be used for that purpose:

- Non-blocking routines return almost immediately.
- Test routines are used to check the status of the communication.

Finite-element analysis



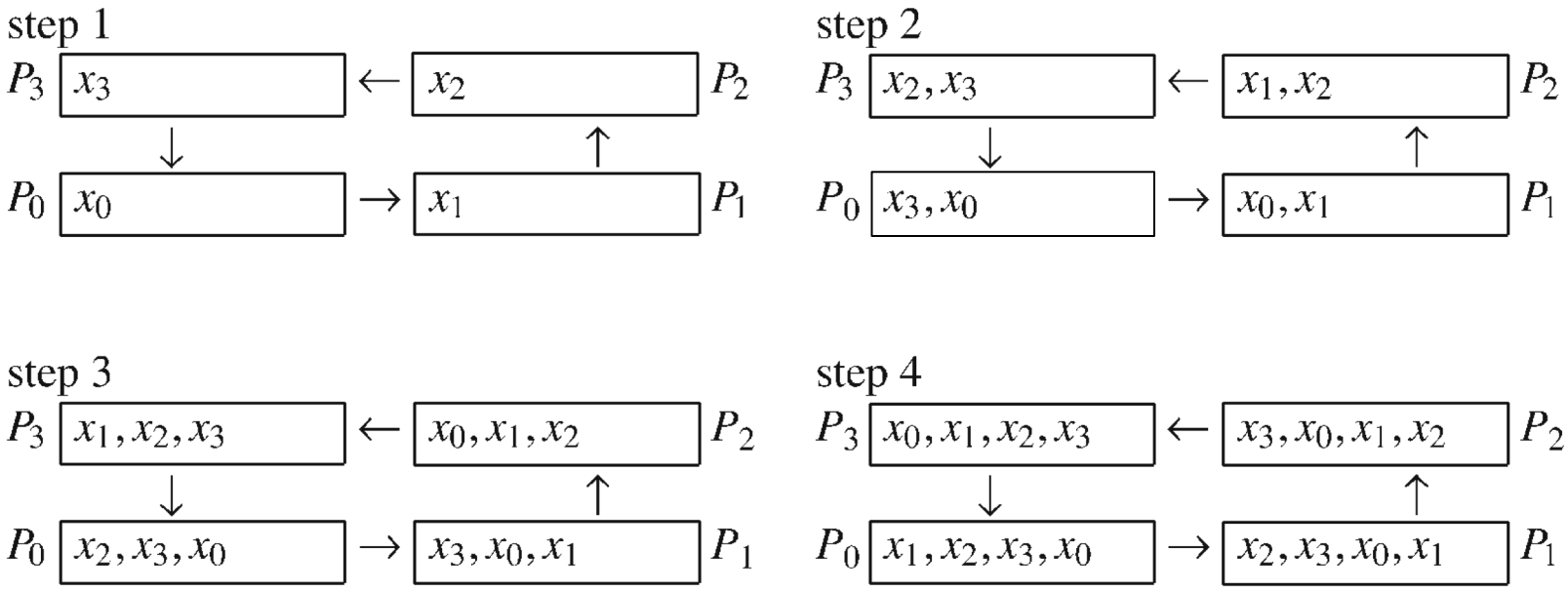
2D rectangular domain



General domain

- MPI communication using Send and Recv. If buffers are used the program will run correctly. Otherwise, deadlocks are possible.
- It's easier to launch all the communications in a non-blocking manner, and then test to check whether they have completed or not.

Gather ring using non-blocking communications



See MPI code: **`gather_ring.C`**

```
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.
}
```

Non-blocking versions of send and recv

- Replace: `MPI_send` → `MPI_Isend`

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

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

- What does I stand for?

Immediate

Testing and waiting

- There is a similar non-blocking receive:

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

- Test the status of the request using:

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

- Flag is 1 if request has been completed, 0 otherwise.

- Wait until request completes:

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

MPI Send/Receive modes

- Multiple optimizations of send/receive operations.
- Two main MPI library protocols for implementation:
  - `eager`: send data immediately whether or not a Recv has been posted. Works well for small messages.
  - `rendez-vous`: send data only when Recv has been posted; buffering is not used; requires a synchronization of the two processes.
- `MPI_Send` eager for small messages
- `MPI_Send` rendez-vous for large messages
- `MPI_Ssend` rendez-vous (Blocking Synchronous); matching Recv required for completion; buffering not needed; may be more efficient in some cases
- `MPI_Rsend` eager (Blocking Ready); requires that Recv has been already posted

Application: bucket and sample sort

- Bucket sort is a simpler parallel algorithm.
- Assume we have a sequence of integers in the interval [a,b].
- Split [a,b] into p sub-intervals.
- Move each element to the appropriate bucket (prefix sum required again).
- Sort each bucket in parallel.
- Simple and efficient!
- A variant of this is the radix sort.
- Problem: how should we split the interval? This process may lead to intervals that are unevenly filled.
- Improved version: sample (or splitter) sort.

Videos of sorting algorithms

Radix sort

https://www.youtube.com/watch?v=LyRWppObda4

15 sorting algorithms

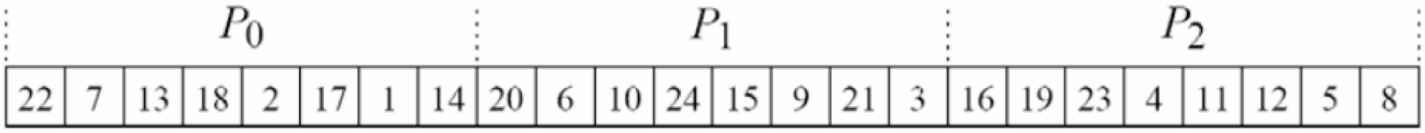
https://www.youtube.com/watch?v=kPRA0W1kECg

Disparity plot

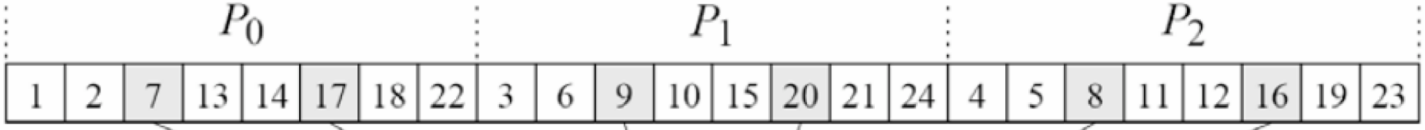
https://www.youtube.com/watch?v=bcwwM6EoveA

- Color = number; distance from center = how far from the correct position the number is

Distributed memory

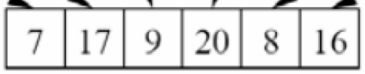


Initial element distribution

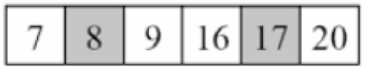


Local sort & sample selection

`MPI_Allgather()`

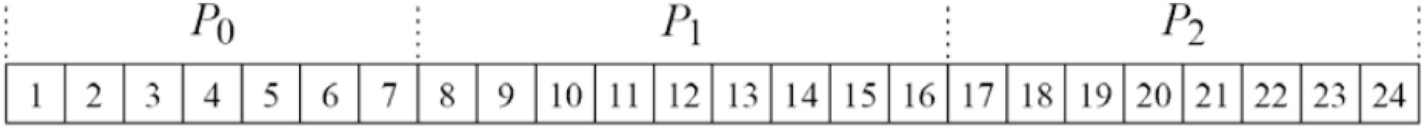


Sample combining



Global splitter selection

`MPI_Alltoall()` and `MPI_Alltoallv()`



Final element assignment

Main challenges

- Load-balancing: the sub-sequences are not always of equal sizes.
  - Depending on the distribution of data, we can get load-imbalances of up to 2x.
- The **`AllToAll`** communication is a bit complicated because the number of data per process is not the same. Hence, we need **`MPI_Alltoallv.`**
- This requires an **`MPI_Alltoall`** to first gather the size of the data to be sent.
- Similarly, the test section at the end requires **`MPI_Gatherv.`**

Efficiency = Tseq / (n Tpar)

Scalability

```
1 process
Efficiency: 0.594325;   runtime seq: 3.91603,   par: 6.58903

2 processes
Efficiency: 0.604003;   runtime seq: 3.79678,   par: 3.14301

6 processes
Efficiency: 0.563082;   runtime seq: 3.93684,   par: 1.16526

12 processes
Efficiency: 0.734754;   runtime seq: 3.85693,   par: 0.43744

24 processes
Efficiency: 0.64589;    runtime seq: 3.8786,    par: 0.25021

48 processes
Efficiency: 0.517663;   runtime seq: 3.80723,   par: 0.153222

64 processes
Efficiency: 0.413869;   runtime seq: 3.84825,   par: 0.145285

80 processes
Efficiency: 0.286249;   runtime seq: 3.83706,   par: 0.167558

96 processes
Efficiency: 0.240625;   runtime seq: 3.8167,    par: 0.165225
```