HPC Programming

Message Passing Interface (MPI), Part II

Peter-Bernd Otte, 19.11.2019

Overview: Next 4 lectures on MPI

- 19.11.2019 (today): Communication (standard, synchronous and asynchronous)
  - test for latency and bandwidth
  - message passing ring (blocking and non-blocking)
- 26.11.2019: collective communication (reduce, scatter, gather, reading user data, spreading input)
  - eg for matrix multiplication
- 3.12.2019: MPI with Python
- 10.12.2019: MPI file I/O, Common Pitfalls

Introduction MPI

Recap

1. Overview / Getting Started
2. Messages & Point-to-point Communication
3. Nonblocking Communication
4. Collective Communication
5. Dealing with I/O
6. Groups & Communicators
7. MPI Derived Datatypes
8. Common pitfalls and good practice (“need for speed”)

MPI: Communicators

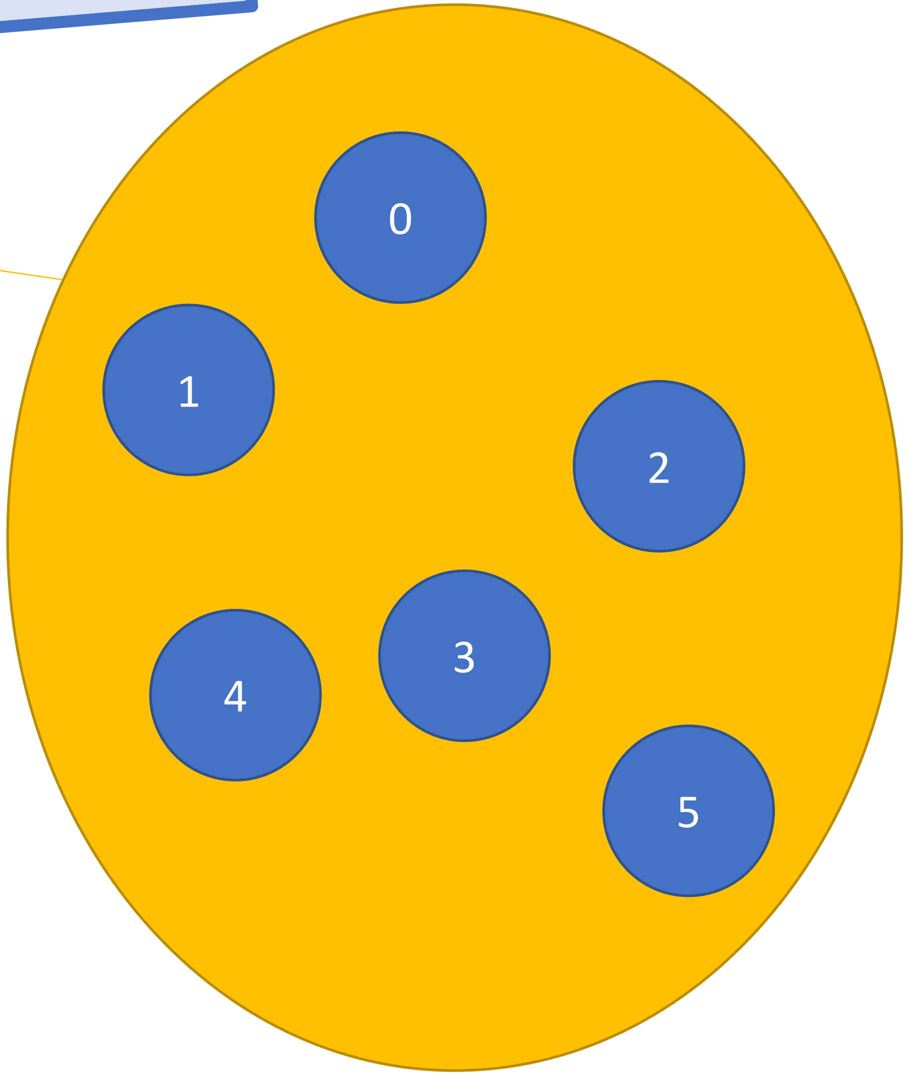
Recap

- MPI Communicator
  = group of processes that can send messages to each other.
- All processes are in `MPI_COMM_WORLD` communicator
  - Defining sub groups → see future lecture
- Number of members in communicator with

```
int MPI_Comm_size (
      MPI_Comm comm     /*in*/,
      int *comm_size_p /*out*/)
```

- Get rank of sub_process with

```
int MPI_Comm_rank (
      MPI_Comm comm    /*in */,
      int * my_rank_p /*out*/)
```



MPI: MPI_Send

Recap

- Sending a message to another receiving rank
- Syntax:

```
int MPI_Send(
      void            *msg_buf_p     /*in*/,
      int             msg_size       /*in*/,
      MPI_Datatype    msg_type       /*in*/,
      int             dest           /*in*/,
      int             tag            /*in*/,
      MPI_Comm        communicator   /*in*/);
```

defines contents of message (msg_buf_p, msg_size, msg_type)

defines destination of message (dest, tag, communicator)

- dest = receiving rank (defined in communicator)
- tag to distinguish messages
- defines the "communication universe",
  all processes are in: MPI_COMM_WORLD

MPI: MPI_Recv

Recap

- Receiving a message from another rank
- Syntax:

```
int MPI_Recv(
      void            *msg_buf_p     /*out*/,
      int             msg_size       /*in*/,
      MPI_Datatype    msg_type       /*in*/,
      int             source         /*in*/,
      int             tag            /*in*/,
      MPI_Comm        communicator   /*in*/,
      MPI_Status      *status_p      /*out*/);
```

defines contents of message

defines destination of message

- source = sender rank (defined in communicator). To accept all: MPI_ANY_SOURCE
- tag to distinguish messages. To accept from all: MPI_ANY_TAG
- defines the “communication universe”, no wildcard available, all processes are in: `MPI_COMM_WORLD`
- `stauts_p` to retrieve error information, or: `MPI_STATUS_IGNORE`

MPI: Make a match

Recap

- rank s calls: `MPI_Send(send_buf, send_buf_size, send_type, dest, send_tag, send_comm);`
- rank q calls: `MPI_Recv(recv_buf, recv_buf_size, recv_type, src, recv_tag, recv_comm, &status);`
- All 5 “green” parameters need to match to get message successfully through.
    - all mandatory to be equal, except :
    - recv_buf_size >= send_buf_size
    - dest = rank of receiving process, src = rank of sending process

Single Program, Multiple Data (SPMD)

Recap

- Standard MPI programming:
  - Write single executable
  - behaviour depends on its rank
    - eg rank=0: message collecting master, ranks>0: computing
  - Number of ranks from 1 to $O(10^4)$ on Himster2
    - $O(10^6)$ on extreme machines
  - called “Single Program, multiple Data”
- ⬄ Multiple-Program Multiple-Data (MPMD)
  - even mixture of different software possible with MPI: Fortran and C executable communicating fine

```
MPI_Comm_rank(MPI COMM WORLD, &my_rank);

if (my_rank == 0) {
  ...
} else {
  ...
}
```

2 helpful functions

- deal with Hyperthreading on Mogon2
- determine MPI message size
  eg receiving unknown number of intergers: `int numbers[MAX_NUMBERS];`

→ See next slides

Hyperthreading (HT) & other ressources in SLURM

Various levels of resource requirements:

- computationally intensive, little inter-process communication
  → use all cores in a multi-core system
  `srun --hint=compute_bound`

- memory bound, saturating the memory bandwidth
  → use a single core on each socket
  `srun --hint=memory_bound`

- highly communication intensive
  → use of in core multithreading (HT)
  `srun --hint=multithread`

  HT enabled by default on Mogon 2, to disable:
  - `srun --hint=nomultithread --cpu-bind=verbose`
  - salloc --ntasks-per-core=1 -p parallel -N 2 --time=01:00:00 -A m2_himkurs

Test drive: HT with srun

```
srun -n 80 --hint=multithread --cpu-bind=verbose sleep 1
cpu-bind-threads=UNK  - z0822, task 41  1 [196475]: mask 0x2 set
cpu-bind-threads=UNK  - z0821, task 32 32 [188099]: mask 0x100000000 set
cpu-bind-threads=UNK  - z0821, task  6  6 [188073]: mask 0x40 set
cpu-bind-threads=UNK  - z0821, task  1  1 [188068]: mask 0x2 set
cpu-bind-threads=UNK  - z0822, task 42  2 [196476]: mask 0x4 set
cpu-bind-threads=UNK  - z0821, task  2  2 [188069]: mask 0x4 set
cpu-bind-threads=UNK  - z0821, task 37 37 [188104]: mask 0x2000000000 set
cpu-bind-threads=UNK  - z0821, task 39 39 [188106]: mask 0x8000000000 set
cpu-bind-threads=UNK  - z0821, task 10 10 [188077]: mask 0x400 set
…
```

MPI: Receive Message Count

- After calling MPI_Recv:
  - MPI_Status structure tells you actual sender and tag of the message.
  - Count of received elements? → MPI_Get_count
- MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count) returns:
  - datatype of the message
  - and count (total number of datatype elements received)
- Speed info: functions takes some time, information is not included in status

```
const int MAX_NUMBERS = 100;
int numbers[MAX_NUMBERS];
int number_amount;
if (my_rank == 0) {
  // Send a random amount of integers
  srand(time(NULL));
  number_amount = (rand() / (float)RAND_MAX) *
    MAX_NUMBERS;
  MPI_Send(numbers, number_amount, MPI_INT, 1, 0,
    MPI_COMM_WORLD);
  printf("0 sent %d numbers to 1\n", number_amount);
} else if (my_rank == 1) {
  MPI_Status status;
  MPI_Recv(numbers, MAX_NUMBERS, MPI_INT, 0, 0,
    MPI_COMM_WORLD, &status);
  MPI_Get_count(&status, MPI_INT, &number_amount);

  printf("1 received %d numbers from 0. „
    „Message source = %d, tag = %d\n",
    number_amount, status.MPI_SOURCE, status.MPI_TAG);
}
```

MPI: Find out message size

Dynamic solution:

- Using MPI_Probe to find out the message size
- MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)

```
int number_amount;
if (world_rank == 0) {
  const int MAX_NUMBERS = 100;
  int numbers[MAX_NUMBERS];
  srand(time(NULL));
  number_amount = (rand() / (float)RAND_MAX) *
    MAX_NUMBERS;
  MPI_Send(numbers, number_amount, MPI_INT, 1, 0,
    MPI_COMM_WORLD);
  printf("0 sent %d numbers to 1\n", number_amount);
} else if (world_rank == 1) {
  MPI_Status status;
  // Probe for an incoming message from process zero
  MPI_Probe(0, 0, MPI_COMM_WORLD, &status);

  // Get the message size
  MPI_Get_count(&status, MPI_INT, &number_amount);

  int* number_buf = (int*)malloc(sizeof(int) *
    number_amount);

  MPI_Recv(number_buf, number_amount, MPI_INT, 0, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  printf("1 dynamically received %d numbers from ”
    “0.\n", number_amount);
  free(number_buf);
}
```

Overview MPI send & recv functions

MPI: different communications modes (1)

	Blocking		note
standard send	MPI_Send		synchronous or asynchronous send (depending on message size and implementation) uses internal buffer.
synchronous send	MPI_SSend		Only completes when the receive has started
asynchronous (buffered) send	MPI_BSend		Completes after buffer copy (always).
ready send	MPI_RSend		problematic: mandatory to have matching receive already listening. Not discussed in this lecture. Might be fastest solution.

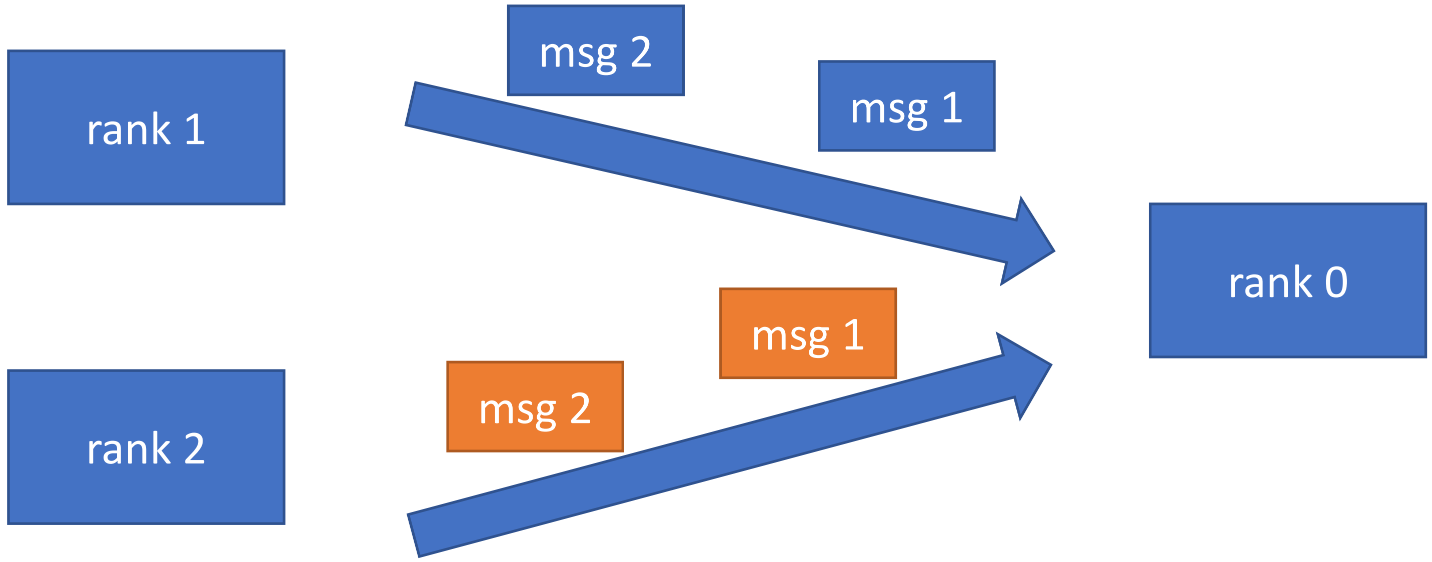
	Blocking		note
standard receive	MPI_Recv		works for all sending routines.

MPI: P2P communications, Pros and Cons

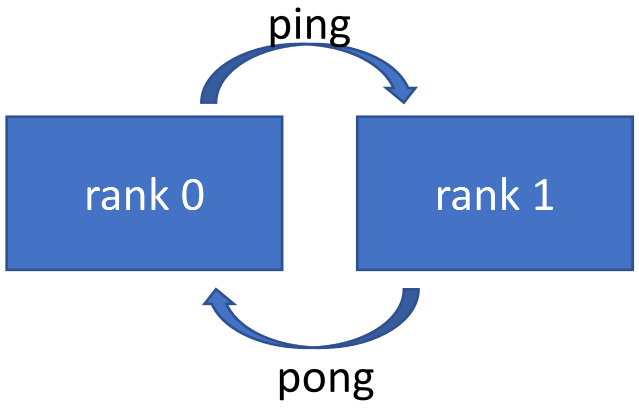
- synchronous send (MPI_SSend)
  - risk of serialisation, waiting and/or deadlock
  - high latency but best bandwidth
- asynchronous send (MPI_BSend)
  - no risks (except: take care of your buffers)
  - low latency but bad bandwidth
- standard send (MPI_Send)
  - risk of implementation and message dependence behaviour
  - plus risks of synchronous send

MPI: Message Order Preservation

- Messages do not overtake, if same:
    - communicator (eg MPI_COMM_WORLD),
    - source rank and
    - destination rank
- true for: synchronous and asynchronous communications
- messages from different senders can overtake



MPI: Measuring latency and bandwidth



- test latency by replying to a short message
  - Step 1: “ping”
    - Rank 0 sends (in blocking mode) a single float to rank 1 with tag 17
    - Rank 1 is in blocking receive mode and awaits the message from rank 0
  - Step 2: “pong”
    - like step 1, but with interchanged roles of rank 0 and 1.
    - use tag 23 for messages for a better overview

  - Repeat this N times (2*N messages in total) and time it with
    - double MPI_Wtime() returns “time in seconds since an arbitrary time in the past.”
      - synchronized for all ranks!
    - latency [ns] = $\Delta t / (2*N) * 1E9$

- test bandwidth (=messages size in bytes / transfer time) by sending large chunks of data. Replace the single float by larger amounts of data eg 1kB, 1MB, 10MB

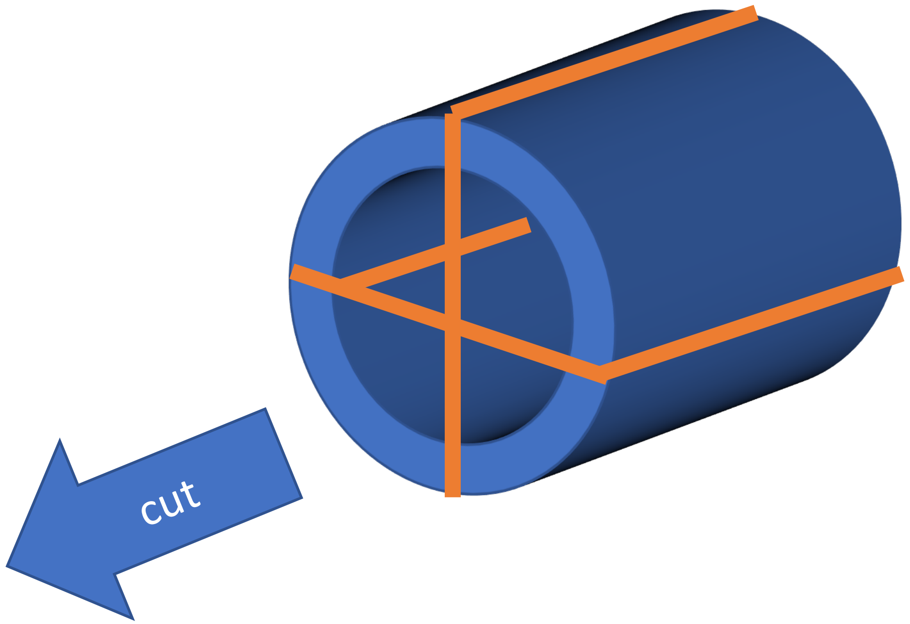
Introduction MPI

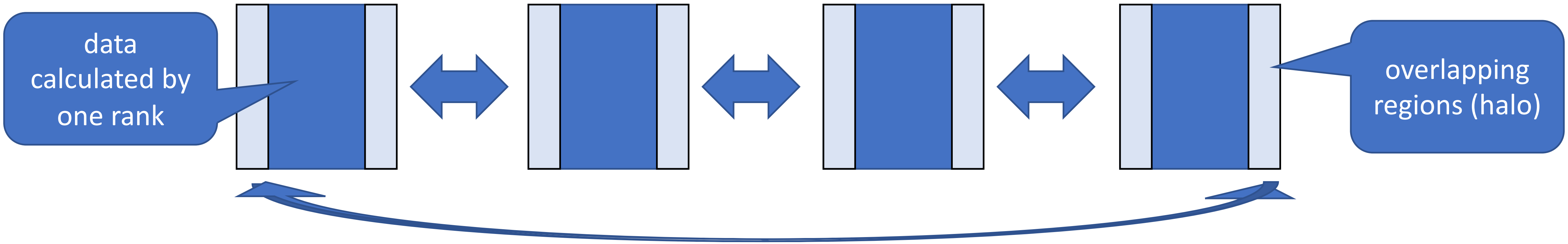
1. Overview / Getting Started
2. Messages & Point-to-point Communication
3. Nonblocking Communication
4. Collective Communication
5. Dealing with I/O
6. Groups & Communicators
7. MPI Derived Datatypes
8. Common pitfalls and good practice (“need for speed”)

prevents: risk of serialisation, waiting and/or deadlock

MPI: Non-Blocking Send & Receive

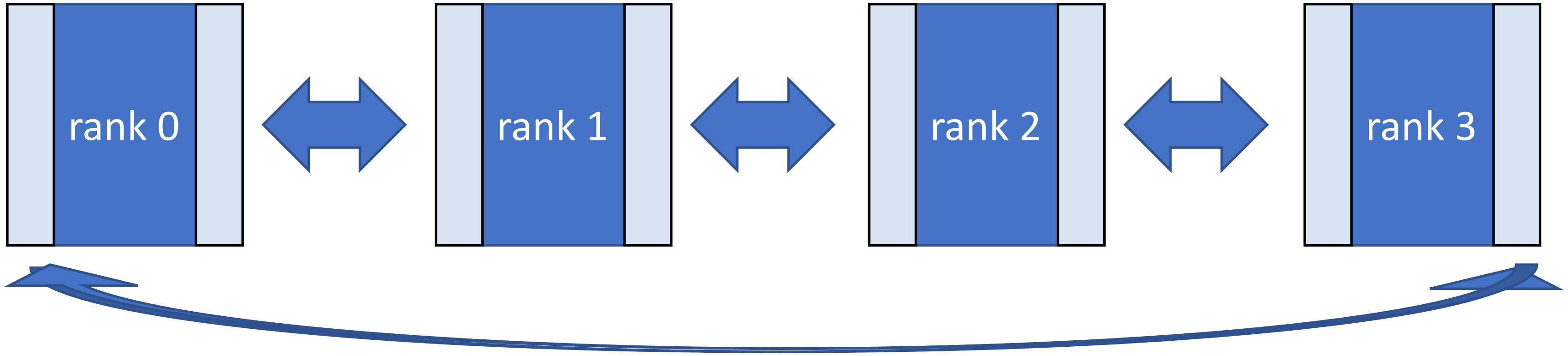
- Motivation:
  - prevents risk of serialisation, waiting and/or deadlock
    → we will see on next slide
  - tracker detector, put into slices
  - distribute the problem
    - overlapping data regions
    - cyclic or non-cyclic boundary conditions



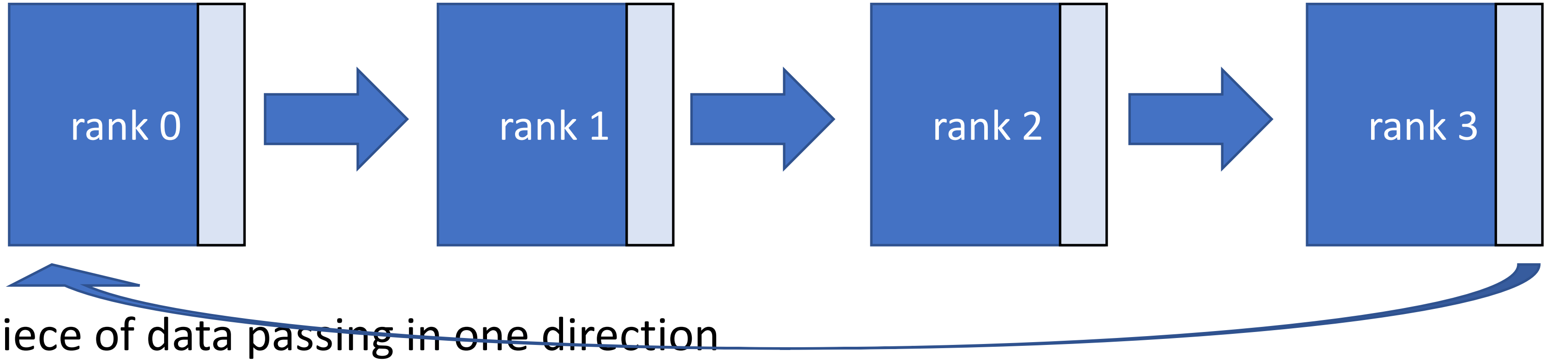


MPI: Non-Blocking Send & Receive

- for simplicity in this lecture, we reduce this



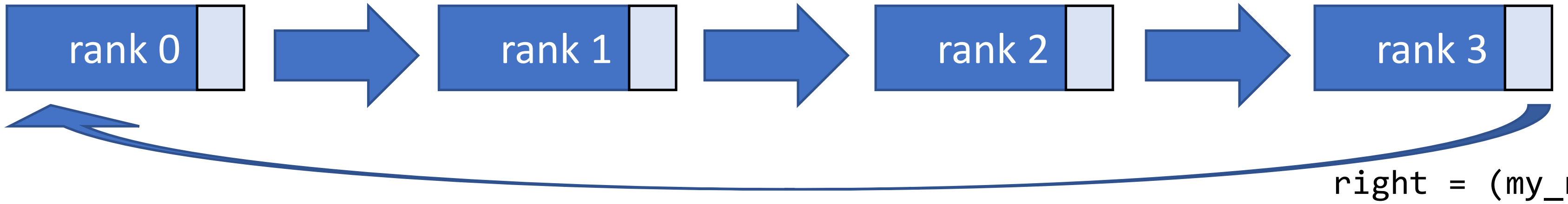
- to a 1D ring



with 1 piece of data passing in one direction

MPI: Non-Blocking Send & Receive

- to a 1D ring with 1 piece of data passing in one direction




```
right = (my_rank+1) % size;
left = (my_rank-1+size) % size;
```

- cyclic: MPI_Send(…to right…)
  MPI_Recv(…from left…)

deadlock!
All are waiting
for a receiver

- non-cyclic: for rank<size-2: MPI_Send(…to right…)
  for rank>0: MPI_Recv(…from left…)

serialisation!
highest rank starts,
rank 0 last

(hint: all this only true if MPI calls are synchronous sends)

MPI: different communications modes (2)

	Blocking	Non-Blocking	note
standard send	MPI_Send	MPI_ISend	synchronous or asynchronous send (depending on message size and implementation) uses internal buffer.
synchronous send	MPI_SSend	MPI_ISSend	Only completes when the receive has started
asynchronous (buffered) send	MPI_BSend	MPI_IBSend	Completes after buffer copy (always).
ready send	MPI_RSend	MPI_IRSend	problematic: mandatory to have matching receive already listening. Not discussed in this lecture. Might be fastest solution.

„i" stands for immediate return

	Blocking	Non-Blocking	note
standard receive	MPI_Recv	MPI_IRecv	works for all sending routines.

MPI: Non-Blocking communication

Solution: Non-Blocking communication

1. Start non-blocking communication
   - and return immediately
2. Process different work
3. Wait for non-blocking communication to complete.

This can be accomplished by either:

- non-blocking send, or
- non-blocking receive

MPI: Non-Blocking communication

This can be accomplished by:

- non-blocking send
    1. MPI_Isend();
    2. Different_Work();
    3. MPI_Wait(); //Waits until MPI_ISend completed / send buffer is read out

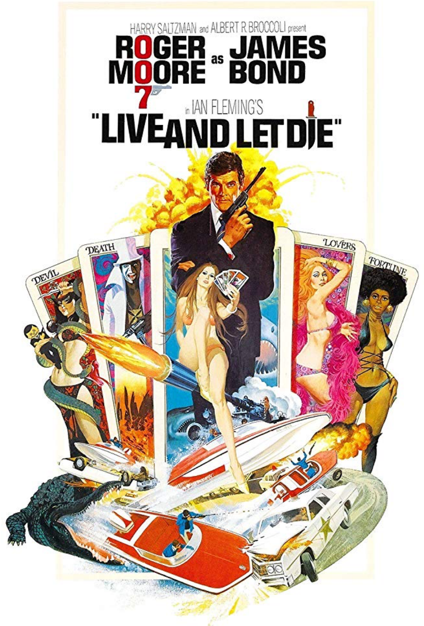
  OR

- non-blocking receive
    1. MPI_Irecv();
    2. Different_Work();
    3. MPI_Wait(); //Waits until MPI_IRecv completed / receive buffer is filled

MPI: Request Handles

- To get a “handle” (or reference) on the ongoing non-blocking communication
  - type: MPI_Request
- Programmer stores is locally

- Live and let die:
  - Retrieved from a nonblocking communication routine
  - used and freed in MPI_Wait



Source: imdb.com

MPI: Non-blocking synchronous send

Syntax:

- `int MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- `int MPI_Wait(MPI_Request *request, MPI_Status *status)`

- buf should not be accessed during Issend and Wait!
- Blocking Ssend == Issend + Wait
- Status is always empty

MPI: Non-blocking synchronous receive

Syntax:

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

- buf should not be accessed during Irecv and Wait!
- Status status is returned in Wait

- Instead of blocking MPI_Wait:
  - Tests for the completion of a request:
  - `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
  - Several request handles: MPI_Waitany, MPI_Testany, MPI_Waitall, MPI_Testall, MPI_Waitsome, MPI_Testsome
- Wait or successful Test is mandatory for each non-blocking communication!

MPI: Send-Receive all-in-one

- equivalent to (and therefore deadlock free):
  - MPI_Irecv
  - MPI_Send
  - MPI_Wait

- MPI_Sendrecv (different send and receive buffer)

```
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)
```

- MPI_Sendrecv_replace (same send and receive buffer)

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int
source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

- See: https://www.mpich.org/static/docs/v3.2/www3/MPI_Sendrecv_replace.html

MPI: different communications modes (2)

	Blocking	Non-Blocking	note
standard send	MPI_Send	MPI_ISend	end (depending tion)
synchronous send	MPI_SSen	MPI_ISSend	e has started
asynchronous (buffered) send	MPI_BSen	MPI_IBSend	ays).
ready send	I_RSend	MPI_IRSend	matching cussed in this .

All combinations valid!
(also mix of blocking and non-blocking calls)

What is the fastest? As long as non-blocking is used, no answer by MPI standard. Application, MPI library and machine dependent.

	g	Blocking	note
standard receive	MPI_Recv	MPI_IRecv	works for all sending routines.

Exercises 3 and 4

- Test latency and bandwidth with exercise 3 “Ping pong”
  https://gitlab.rlp.net/pbotte/learnhpc/tree/master/mpi/exercise3

- Message passing in a ring with exercise 4 (for large detector simulations or matrix computation)
  https://gitlab.rlp.net/pbotte/learnhpc/tree/master/mpi/exercise4