HPC Programming

Message Passing Interface (MPI), Part II

Peter-Bernd Otte, 27.11.2018

Overview: Next 3 lectures on MPI

- 27.11.2018: Communication (standard, synchronous and asynchronous)
  - test for latency and bandwidth
  - message passing ring (blocking and non-blocking)
- 4.12.2018: matrix multiplication (collective communication: Reduce, Scatter, Gather, reading user data, spreading input)
- 11.12.2018: MPI file I/O, Common Pitfalls

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”)

MPI: Communicators

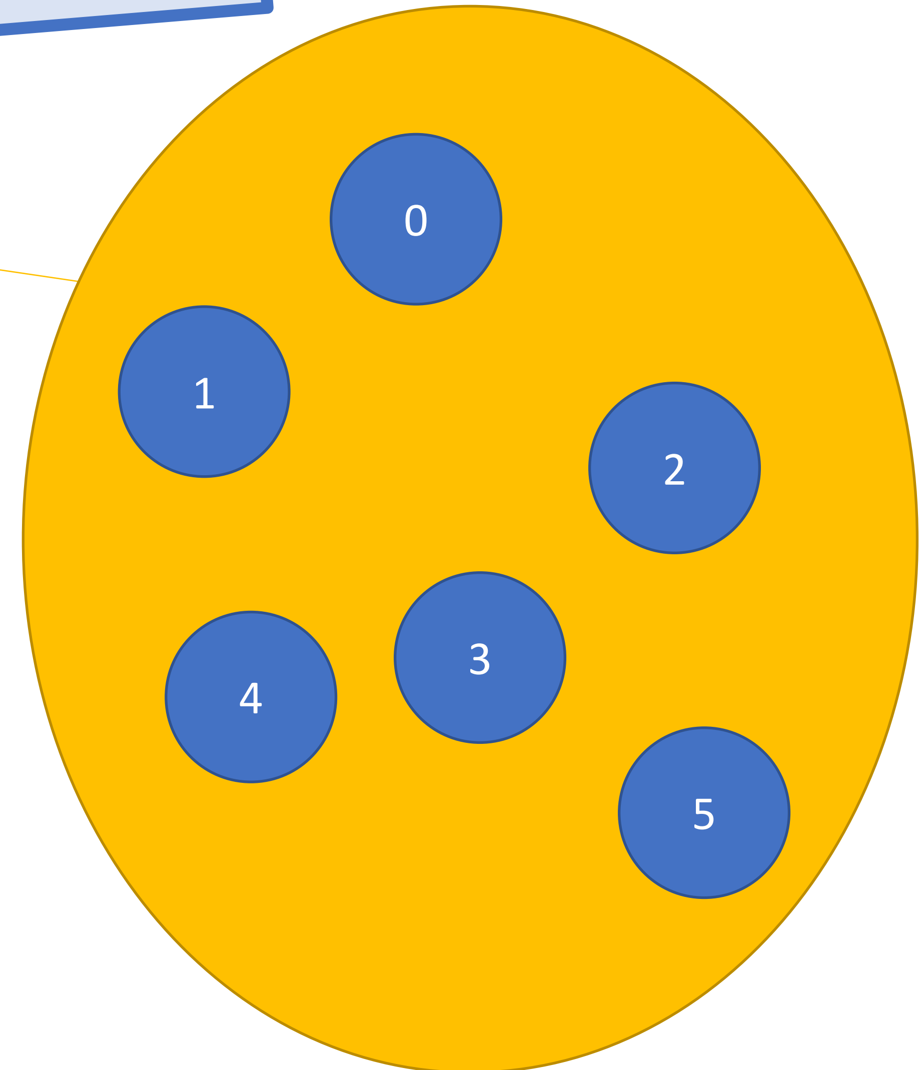
Recap

- 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

defines destination of message

- 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

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 {
  ...
}
```

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

- 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);
}
```

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.

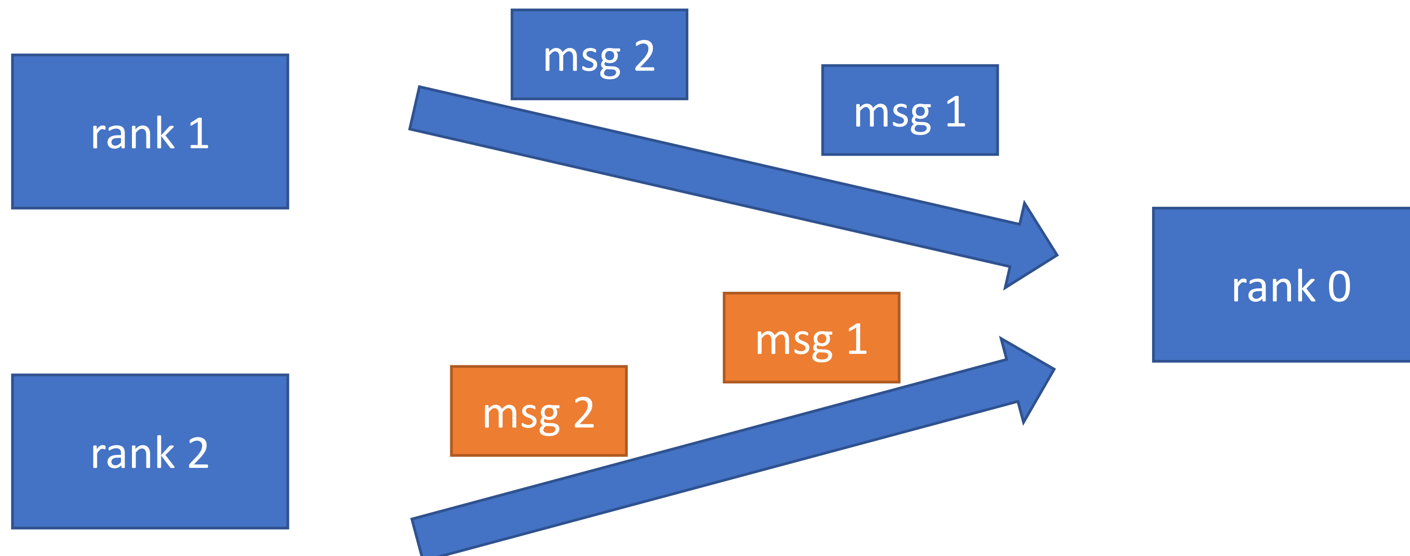
	Blocking		note
standard receive	MPI_Recv		works for all sending routines.

MPI: P2P communications, Pros and Cons

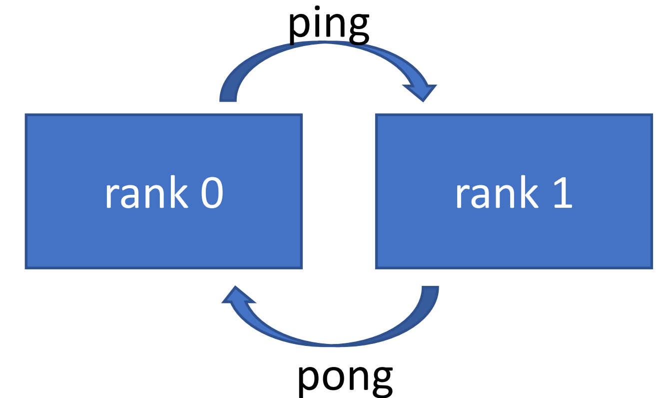
- synchronous send
  - risk of serialisation, waiting and/or deadlock
  - high latency but best bandwidth
- asynchronous send
  - no risks (except: take care of your buffers)
  - low latency but bad bandwidth
- standard send
  - risk of implementation and message dependence behaviour
  - plus risks of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

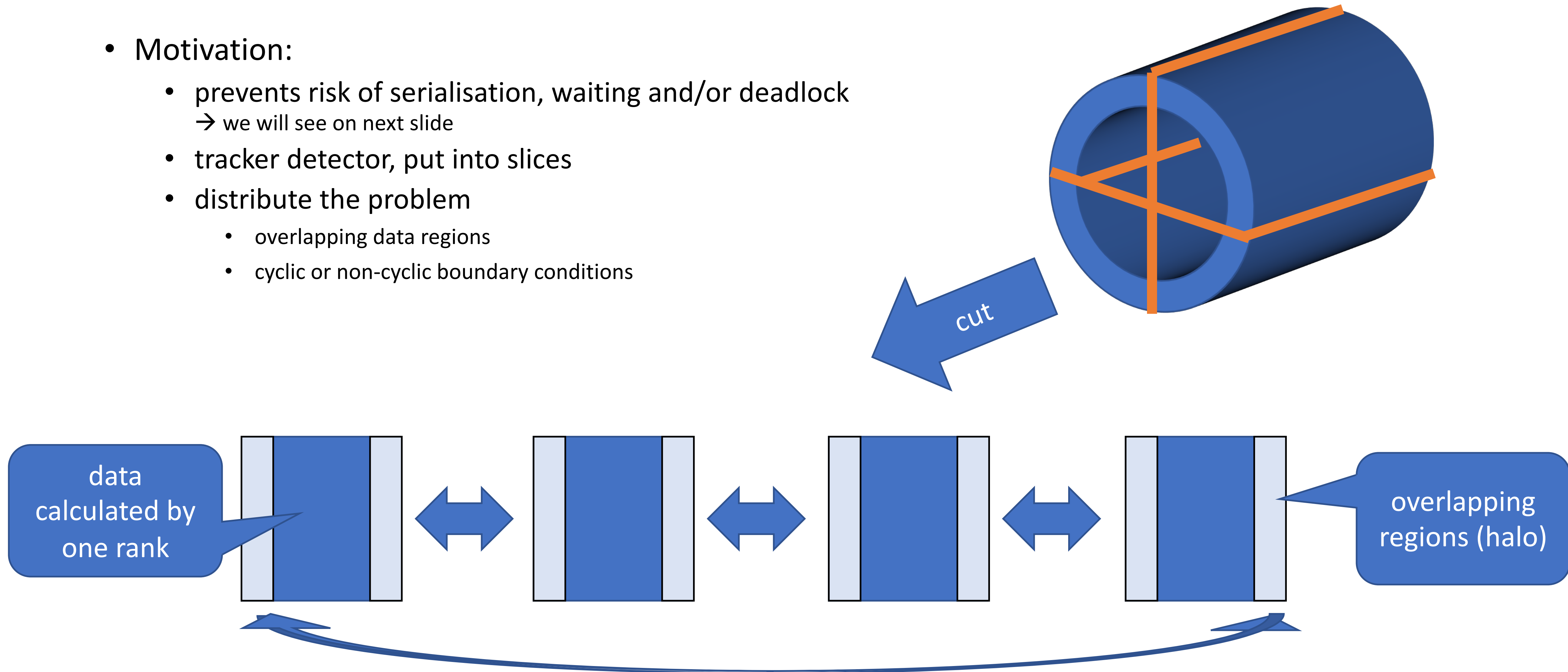
Introduction MPI

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

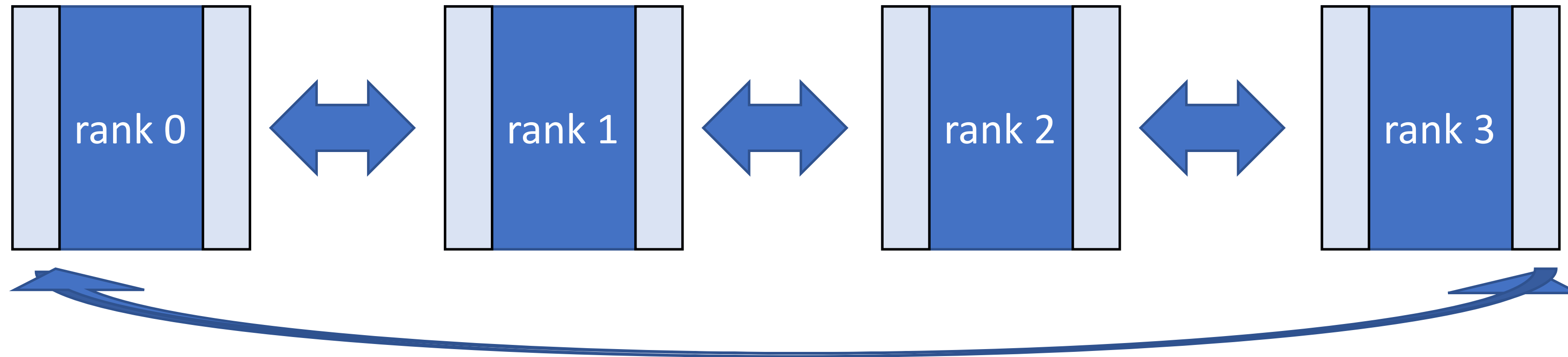
MPI: Non-Blocking Send & Receive

- 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

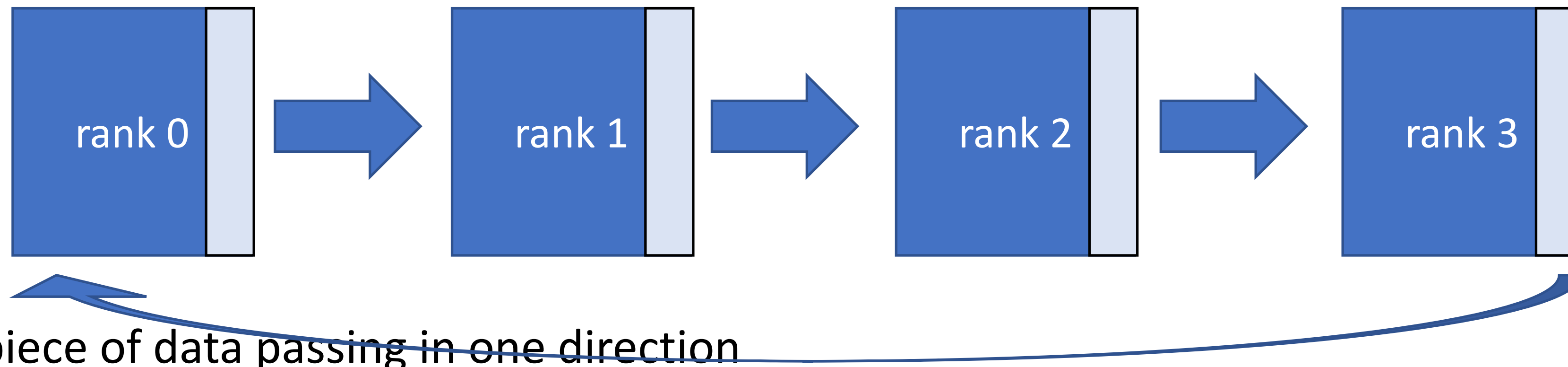


MPI: Non-Blocking Send & Receive

- for simplicity in this lecture, we reduce this



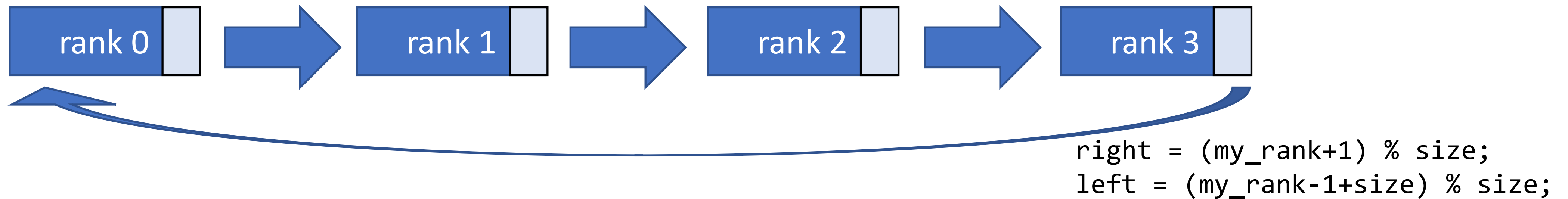
- to a 1D ring



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:

- non-blocking send
- 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

- 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

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)
```

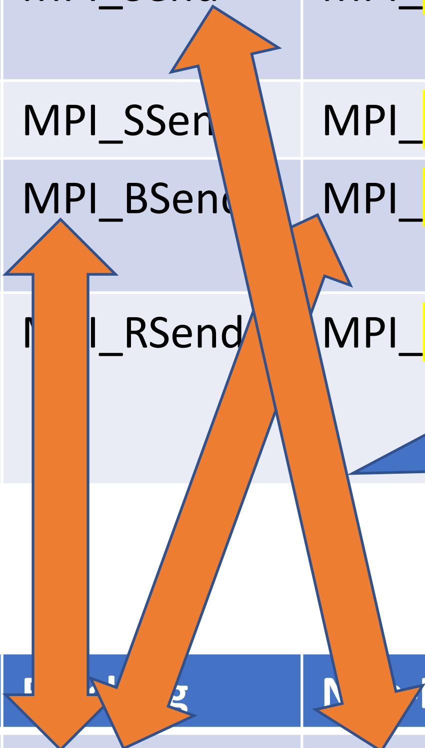
- See: https://www.mpich.org/static/docs/v3.2/www3/MPI_Sendrecv_replace.html

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

Set up your workbench

- Connect 2 times via SSH to Mogon2 / HIMster2
    1. Use the first SSH connection for editing (gedit, vi, vim, nano, geany) and
       module load mpi/OpenMPI/3.1.1-GCC-7.3.0
       compiling: mpicc -o ExecutableName SourceFileName.c
    2. Use the second connection for the interactive execution on the nodes (no execution on the head node!):
       salloc -p parallel -N 1 --time=01:30:00 -A m2_himkurs --reservation=himkurs -C skylake
       module load mpi/OpenMPI/3.1.1-GCC-7.3.0
       mpirun -n 2 ./ExecutableName
- Download the files via: wget https://www.hi-mainz.de/fileadmin/user_upload/IT/lectures/WiSe2018/HPC/files/MPI-03.zip && unzip MPI-03.zip

Hints:

- If the reservation with salloc –p parallel fails, try:
    - salloc -p devel -n 4 -A m2_him_exp
- The reserved resources with salloc can’t be overwritten with mpirun
    - Resources(salloc) => Resources(mpirun)
- Possible to check reservation with: squeue -u USERNAME

Exercise 3: Playing ping pong

Learning objectives:

- test latencies and bandwidth depending on packet size

Steps:

1. Download the skeleton from lecture webpage:
    - wget https://www.hi-mainz.de/fileadmin/user_upload/IT/lectures/WiSe2018/HPC/files/MPI-03.zip && unzip MPI-03.zip
2. Implement a ping pong behaviour as described on slide "MPI: Measuring latency and bandwidth" with 50 passages and 1 float in the message
3. Measure latency in ns for different number of messages
4. Print out the bandwidth
5. increase the message length to 32kByte, 1MB, 10MB and check latency and bandwidth
6. (optional) exclude the first ping-pong from the measurement to overcome start up delays. Is now the latency independent on the message count?
7. (optional) try MPI_Ssend instead of MPI_Send for different message sizes.

Exercise 4: Msg passing in a ring

Learning objectives:

- Testing blocking and non-blocking send & receive

Steps:

1. Download the skeleton from lecture webpage:
   - wget https://www.hi-mainz.de/fileadmin/user_upload/IT/lectures/WiSe2018/HPC/files/MPI-04.zip && unzip MPI-04.zip
2. ranks pass on information in a ring. Overall task is to compute the sum of all individual ranks (result for 3 ranks: 3, for 4 ranks: 6, etc.).
   - Each rank does “my_size” times:
   - receive data from previous rank and stores this in a local buffer,
   - increment the local buffer by the local “my_rank”
   - sends the new buffer to the next in the ring
3. Try this with blocking (will not work, deadlock!) and non-blocking communication
4. Bonus:
   - Use MPI_Irecv + MPI_Ssend + MPI_Wait methods
   - Use MPI_Sendrecv method

Solutions

Solution 3 (1/2)

```c
#include <stdio.h>
#include <mpi.h>

#define proc_A 0
#define proc_B 1
#define ping 17
#define pong 23
#define number_of_messages 50
#define length_of_message 1

int main(int argc, char *argv[]) {
int my_rank;
float buffer[length_of_message];
int i;
double start, finish, time;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Solution 3 (2/2)

```
start = MPI_Wtime();
for (i = 1; i <= number_of_messages; i++) {
  if (my_rank == proc_A) {
    MPI_Send(buffer, length_of_message,
MPI_FLOAT, proc_B, ping, MPI_COMM_WORLD);
    MPI_Recv(buffer, length_of_message,
MPI_FLOAT, proc_B, pong, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
  } else if (my_rank == proc_B) {
    MPI_Recv(buffer, length_of_message,
MPI_FLOAT, proc_A, ping, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    MPI_Send(buffer, length_of_message,
MPI_FLOAT, proc_A, pong, MPI_COMM_WORLD);
  }
}
finish = MPI_Wtime();
```

```
if (my_rank == proc_A) {
time = finish - start;
printf("Time for one messsage: %f nano
seconds.\n",
(float)(time / (2 * number_of_messages) *
1e9));
}

MPI_Finalize();
}
```

Solution 4

```c
int my_rank, size;
int snd_buf, rcv_buf;
int right, left;
int sum, i;

MPI_Status status;
MPI_Request request;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

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

sum = 0;
snd_buf = my_rank;
```

```c
for( i = 0; i < size; i++) {
  MPI_Issend(&snd_buf, 1, MPI_INT, right,
    TAG_to_right, MPI_COMM_WORLD, &request);

  MPI_Recv(&rcv_buf, 1, MPI_INT, left,
    TAG_to_right, MPI_COMM_WORLD, &status);

  MPI_Wait(&request, &status);

  snd_buf = rcv_buf;
  sum += rcv_buf;
}

printf ("PE%i:\tSum = %i\n", my_rank, sum);

MPI_Finalize();
```