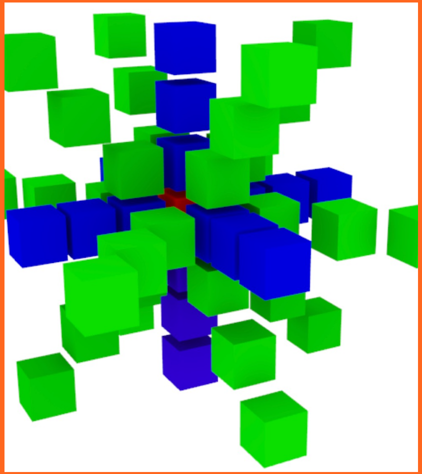
CS-E4690 – Programming Parallel Supercomputers

Basics of message passing interface (MPI)

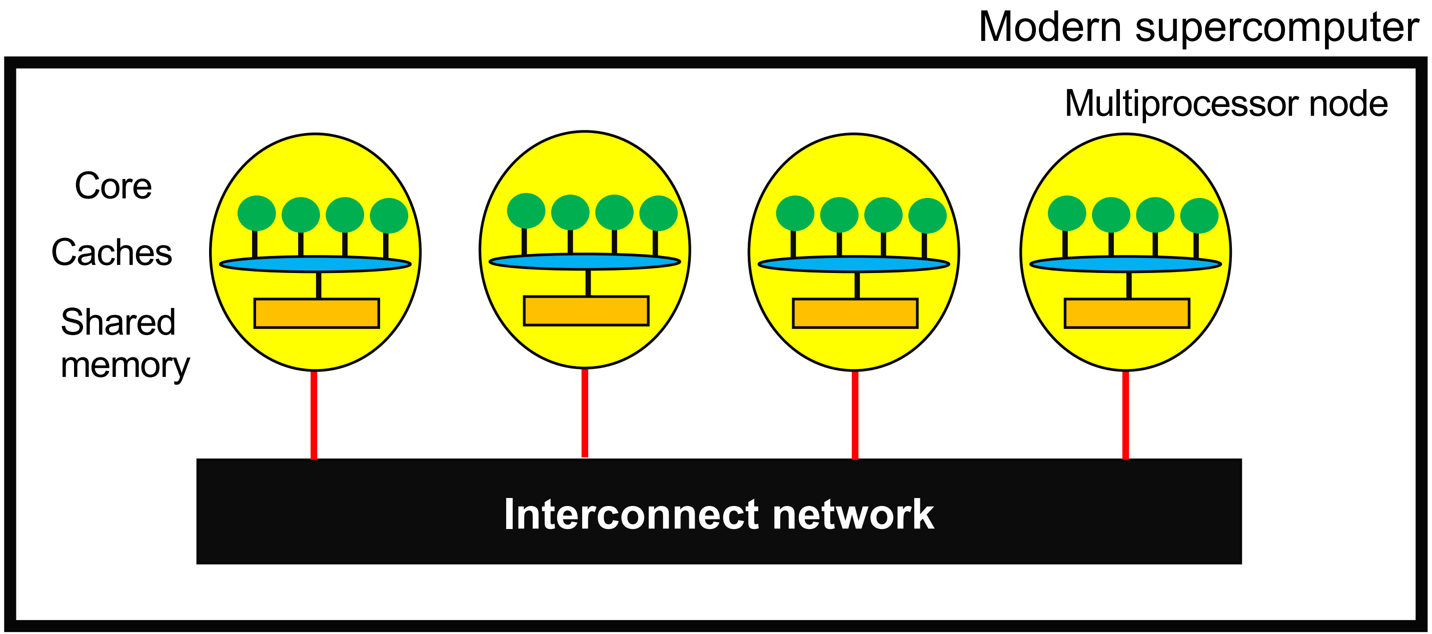
Maarit Korpi-Lagg

maarit.korpi-lagg@aalto.fi

Aalto University
School of Science



Recap of the situation



Current “software” landscape

- **MPI (developed since 1991, standardized in 1994, now at MPI-3, MPI-4 soon coming): several implementations - OpenMPI, MPICH, MPAVICH…**
    - Libraries that provide message passing functions
    - API to provide bindings to higher-level programming languages (Co-array Fortran, …, Python, R, Matlab, Java/Scala, Julia, Chapel, …)
- **Big data programming models: MapReduce; Hadoop, Spark, …**
    - Instead of (only) passing messages, a distributed file system providing data locality is used

Low or high-level programming?

MPI:

- Low level, difficult to program
- Fault tolerance is left to the user to take care about
- Available and supported at every HPC center
- Standardized

During this course we use MPI

Higher-level languages:

- Easier to program
- Fault tolerance might be readily implemented
- Might not be provided everywhere
- You do not have to so much care,
  but also do not learn, about the internal
  workings of the distributed programming model

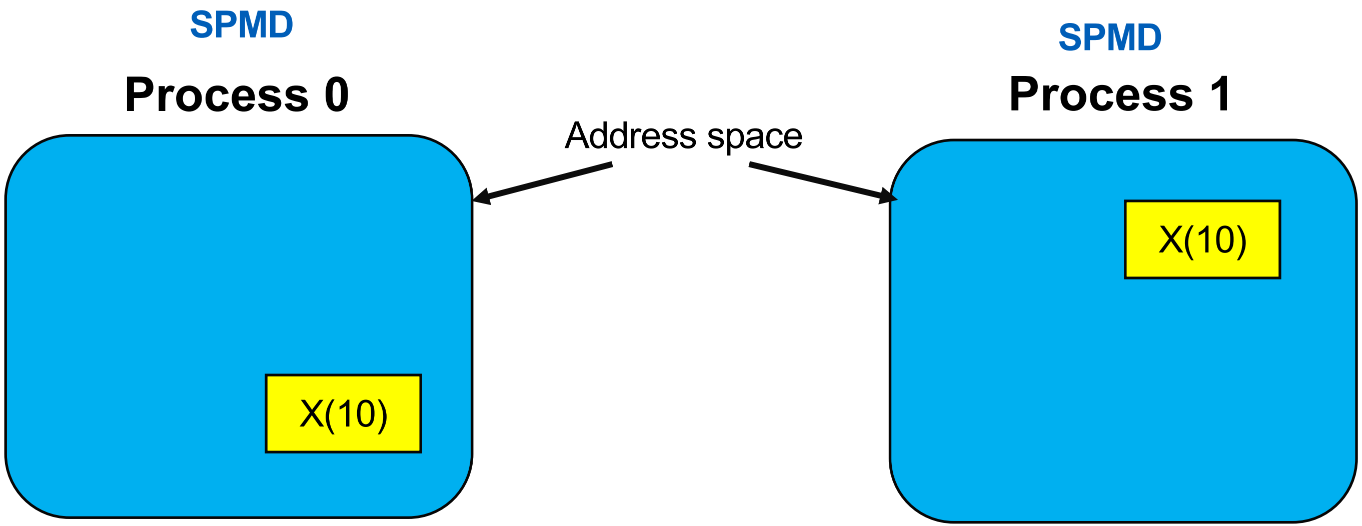
How to decide in practise?

1. I am lacking understanding of distributed memory programming, and will find the easiest way out with the high-level programming languages.

2. What is available in the system accessible for you now/near future?

3. I want to write portable code, and parallelize it only once, and keep on maintaining it with minimal effort

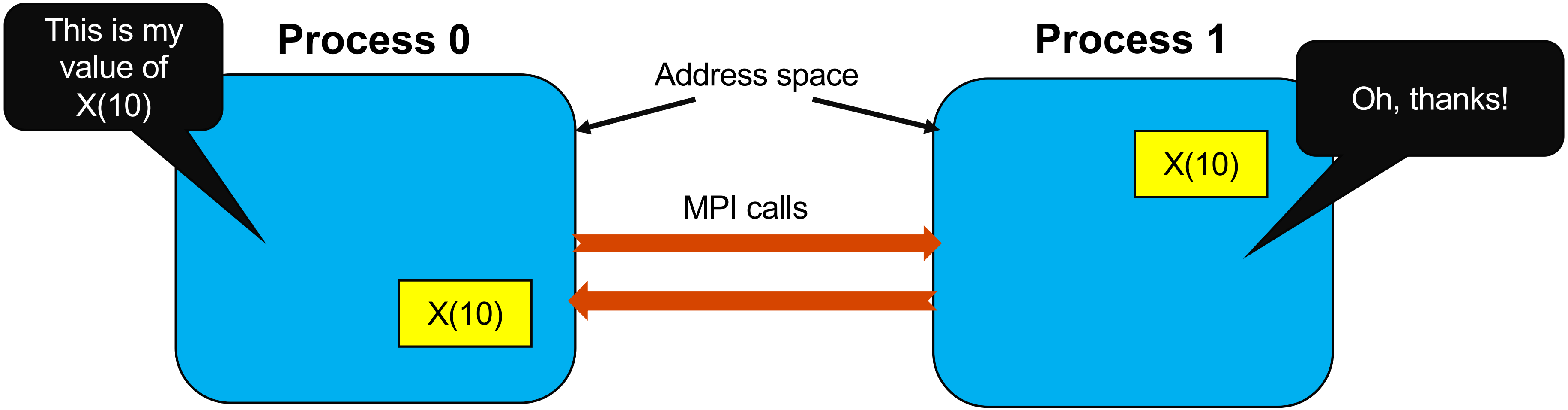
Distributed memory programming model



int X(100); ...
X(10)=my_rank*10;
printf(”My value at index 10 %d”,X(10));

int X(100); …
X(10)=my_rank*10;
printf(”My value at index 10 %d”,X(10);

Distributed memory programming model



int X(100); ...
X(10)=my_rank*10;
printf(”My value at index 10 %d”,X(10));

int X(100); …
X(10)=my_rank*10;
printf(”My value at index 10 %d”,X(10);

Fundamental idea

MPI libraries implement a message passing model, in which the sending and receiving of messages combines both data movement and synchronization. Processes have separate address spaces.

Collective operations

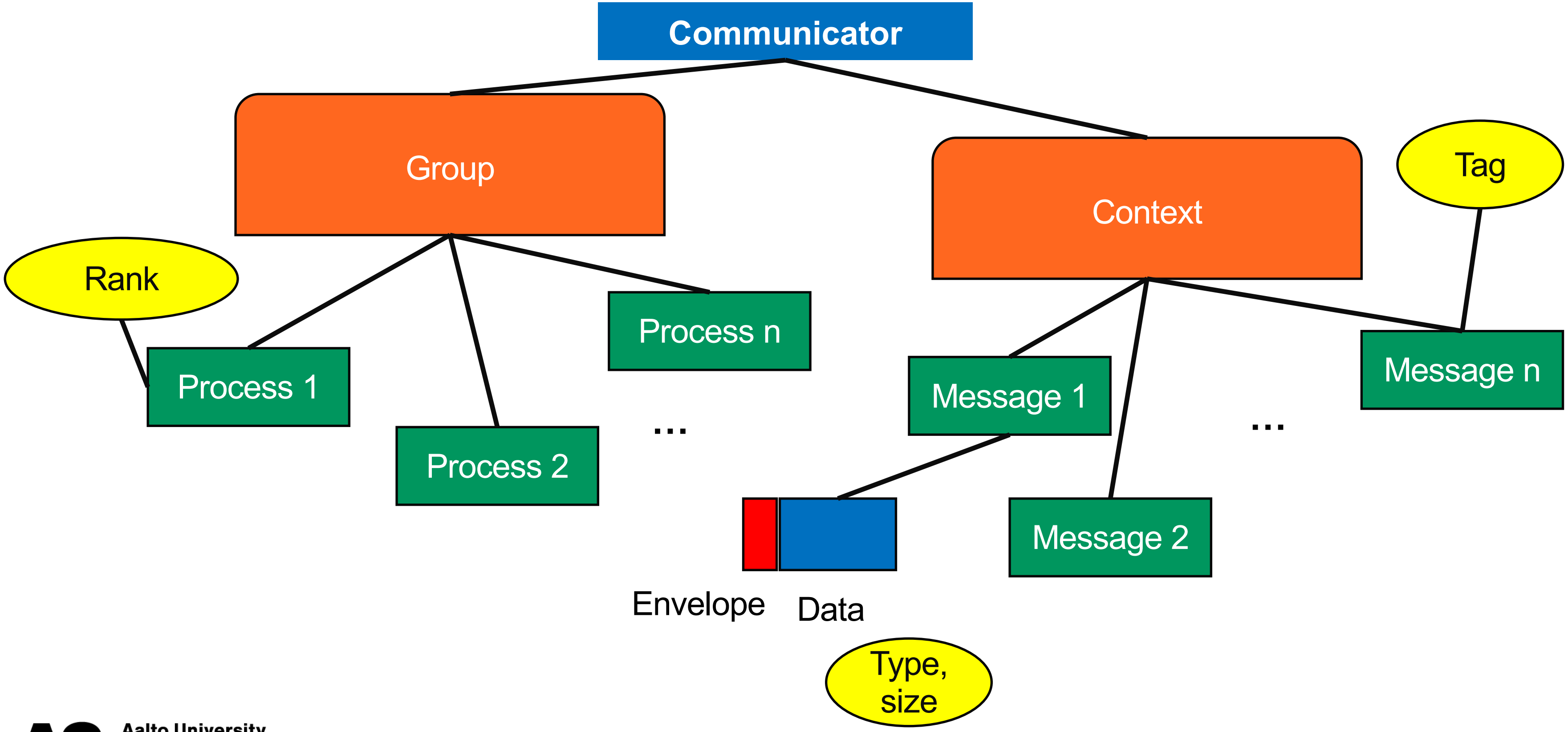
Point-to-point communication

Two high-level modes of operation; during this lecture, we start with point-to-point

But, how to arrange

- How many others are there, and where amongst them am I?
- Identification of sender and receiver
- Communication about what is going to be sent and received (prescription of data)
- Identification of the message (which data belongs where), if many are constantly sent?
- What is supposed to happen when the transmission is complete?

Communicator (def. MPI_COMM_WORLD)



Aalto University
School of Science

C code in practise

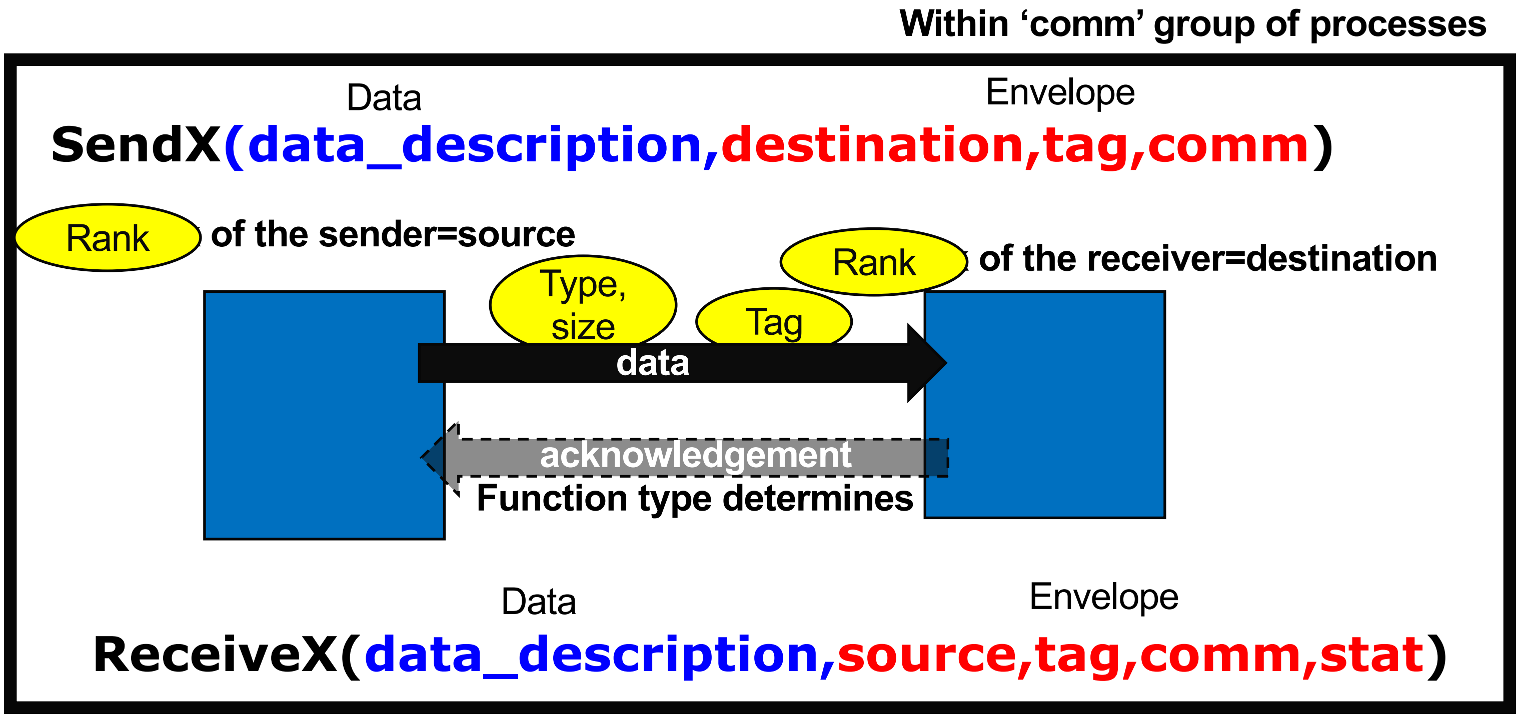
scripts/job_CPU_example.sh

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=2
```

```
#include “mpi.h”
int main(int argc, char *argv[]) {
int rank, size;
MPI_Init (&argc, &argv); /* Communicator set up */
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf(“My rank %d of %d\n”, rank, size);
MPI_Finalize(); /* Communicator deallocated */
}
```

Aalto University
School of Science

More detailed functionality



Two operation modes

Point-to-point (P2P) communications

Co-operative communication

Lecture 3

Blocking

MPI_Send
MPI_Recv
MPI_SendRecv
MPI_Bsend
…

MPI_Isend
MPI_Irecv…

Non-blocking

One-sided communication (RMA ops)

MPI_Get
MPI_Put …

Lecture 4

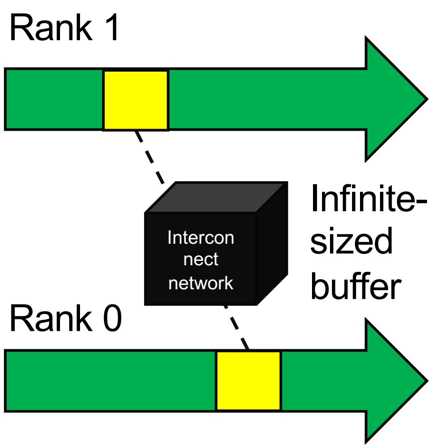
Collective communications

MPI_BCast

Lecture 4

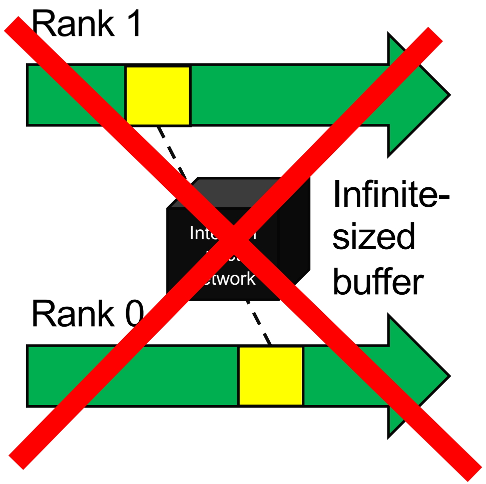
MPI_Scatter …

Blocking communication



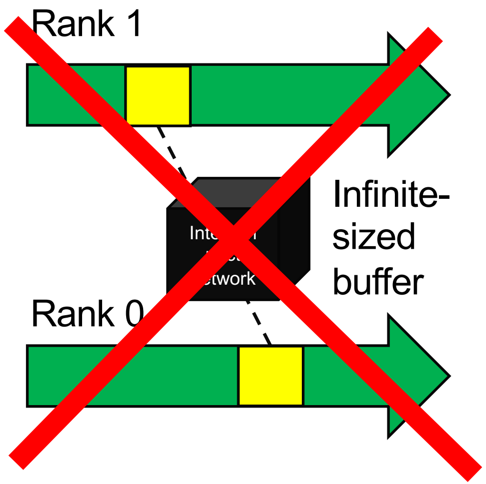
Yellow: communication
Green: computation
Grey: Idling

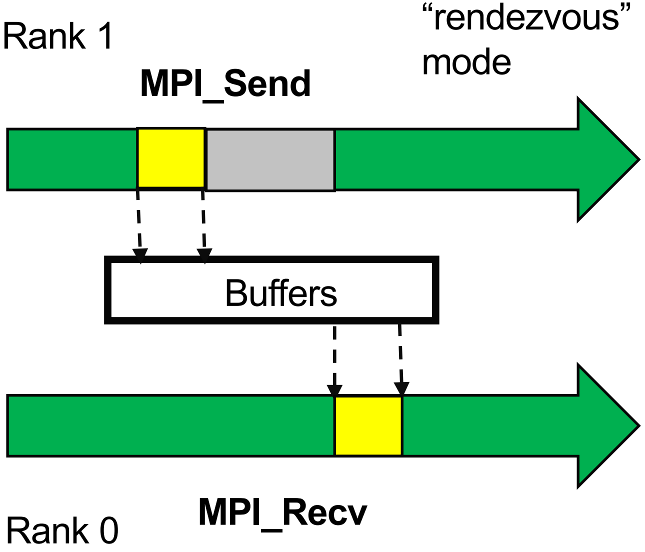
Blocking communication



Yellow: communication
Green: computation
Grey: Idling

Blocking communication





Yellow: communication
Green: computation
Grey: Idling

Sending call blocks until the receiving process has started.
Problem: If the receive cannot start for some reason, the system goes into a halt, called deadlock.

Blocking communication

- Exception: many MPI implementations optimize the non-blocking send with an eager protocol for short messages.
- The eager protocol keeps on sending the fully packed messages including the data and the envelope, assuming that the receiver can keep on receiving the full package.
- Problem: your code may work for with small system sizes, and deadlock with large system size.

Blocking communication

int MPI_Send(const void* buf, int count, MPI_Datatype datatype,

int dest,int tag, MPI_Comm comm)

UNIQUE dest and tag

Push communication mechanism

int MPI_Recv(void* buf, int count, MPI_Datatype datatype,

int source,int tag, MPI_Comm comm,

MPI_ANY_SOURCE

MPI_Status *status)

MPI_ANY_TAG

Structure containing source, tag, error, and length

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

Aalto University
School of Science

Note: MPI_Recv can receive messages sent in any mode.

Elementary data types

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

User defined data types can be useful, will be dealt with during the next lecture

Errors

- **Virtually all function calls return an error. In C, the returned MPI function value is the error, 0 indicating success.**
- **Implementation specific; refer to the documentation of your MPI library**
- **If a MPI function call causes an error, it, as a thumb rule, aborts by itself (relatively safe not to handle errors).**
- **Programmer can also inspect the error and abort the code using the default error handle** MPI_ERRORS_RETURN.

Questions: what would these lines of codes do?

MPI/MPI_SR_1.c – MPI/MPI_SR_3.c code examples are related to these questions

1)

```
…
your_id=1-my_id
MPI_Send(&sendbuf,1,MPI_INT,your_id,0,comm);
MPI_Recv(&recvbuf,1,MPI_INT,your_id,0,comm,&status);
…
```

2)

What would happen if you used MPI_Rsend function?

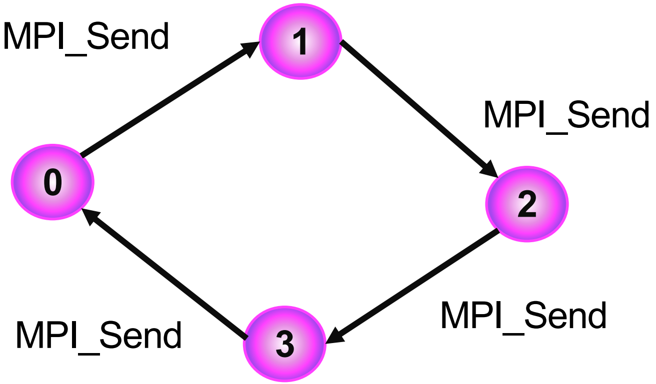
```
…
your_id=1-my_id
MPI_Recv(&recvbuf,1,MPI_INT,your_id,0,comm,&status);
MPI_Send(&sendbuf,1,MPI_INT,your_id,0,comm);
…
```

3)

Case 1) if you would send larger messages? What is happening here?

Deadlock

Processes wait for each other to do something, and the code hangs.



Cycles in waiting-for-graphs indicate deadlocks.

Question

Will the following pseudocode deadlock with MPI_Send and MPI_Recv?

MPI/MPI_SR_4.c code example is related to these questions

…

```
next_id = my_id+1; prev_id = my_id-1;
if ( /* I am not the last processor */ ) send(target=next_id);
if ( /* I am not the first processor */ ) receive(source=prev_id)
```

…

Would you call this efficient parallel execution? What actually happens? Why are the results very difficult to interpret?

Pair-wise co-operative MPI_Sendrecv

- **How the prevent deadlocks? 1. Avoid unsafe operations; one alternative is to use…**
- **Use MPI_Sendrecv(....from... ...to...);** with the right choice of source and destination.
- For example:

MPI_Comm_rank(comm,&nproc); ….

MPI_Sendrecv(.... /* from: */ nproc-1 ... ... /* to: */ nproc+1 ...);

- **Then you always need a "pair" to communicate with**
- **If not, then you need to use** "MPI_PROC_NULL"

Aalto University
School of Science

Question

Will the efficiency of this code be any better with MPI_Sendrecv?

```
…
next_id = my_id+1; prev_id = my_id-1;
if ( /* I am not the last processor */ ) send(target=next_id);
if ( /* I am not the first processor */ ) receive(source=prev_id)
…
```

MPI/MPI_SR_5.c code example is related to this question

Synchronous blocking send MPI_Ssend

- Another alternative is to use…
- MPI_Ssend();
- ”S” for “Synchronous”, meaning that the receiver is *always forced* to send an acknowledge.
- It will not avoid deadlocks.
- In this case, all unsafe operations should always deadlock, helping you out to debug and write “safer” code.

Buffered blocking communication

MPI_**B**send "Buffered"

3. Force buffering

```
int bufsize; /* Size of data + MPI_BSEND_OVERHEAD
*/
char *buf = malloc( bufsize );
MPI_Buffer_attach( buf, bufsize );
...
MPI_Bsend( ... same as MPI_Send ... );
...
MPI_Buffer_detach( &buf, &bufsize );
…
```

User is responsible for allocating large enough buffers.

Question: is this more efficient? You can try it out.

MPI/MPI_SR_6.c code example is related to this question

Blocking communication

Pros

Programmer has **full control** about where the data is: if the send call returns, the data has been successfully received, and the send buffer can be used for other purposes or de-allocated.

Buffering possible, so programmer can collect small messages into larger ones.

Cons

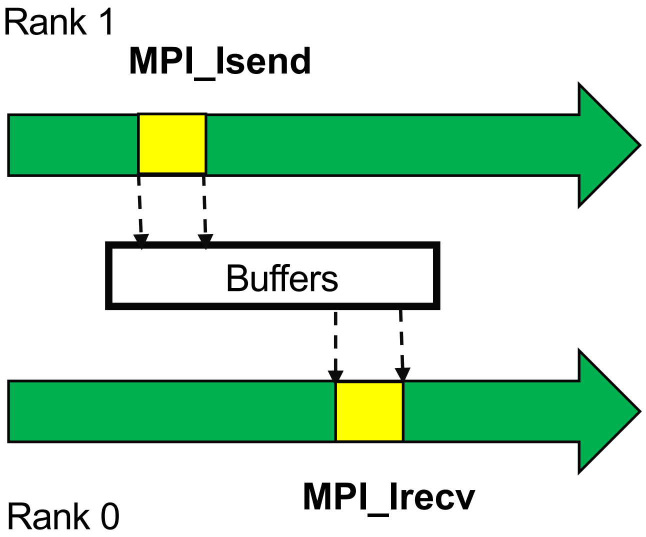
Unsafe operations cause deadlocks – one needs to be careful in ordering the calls.

Overlapping computation and communication is challenging.

Non-blocking communication

Immediate or **Incomplete** **MPI_Isend and MPI_Irecv: they tell the runtime system "Here is my data, please send it forward as I instruct" or "I am expecting certain type of data to come to this provided buffer space".**

"Posting"



Non-blocking communication

```
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)
```

Non-blocking routines yield an **MPI_Request** object. This request can then be used to query whether the operation has completed. **MPI_Irecv** routine does not yield an **MPI_Status** object. This is because the status object describes the actually received data, and at the completion of the **MPI_Irecv** call there is no received data yet.

Non-blocking communication

MPI_STATUS_IGNORE

Int MPI_Wait(MPI_Request *request, MPI_Status *status);

int MPI_Waitall(int count, MPI_Request array_of_requests[],
MPI_Status array_of_statuses[])

MPI_STATUSES_IGNORE

One needs to **wait** for the completion of the non-blocking routines. There are various functions for that. They pass the **MPI_Request object** as a reference and return an MPI_status. If you are not interested in the status, then you can specify MPI_STATUS(ES)_IGNORE instead. These calls **deallocate** the handle after and set it to MPI_REQUEST_NULL. Waitall waits for **multiple** messages, and hence works with **arrays of requests and statuses**.

Non-blocking communication

int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status *status) *MPI_STATUS_IGNORE*

int MPI_Waitsome(int incount, MPI_Request array_of_requests[], int *outcount, int array_of_indices[], MPI_Status array_of_statuses[]) *MPI_STATUSES_IGNORE*

If one wishes to wait for **one or some** messages separately, then Waitany and Waitsome functions can be used. NB! Only after the corresponding wait call it is safe to use the buffer that has been sent, or has received its contents. To send multiple messages with non-blocking calls you therefore have to allocate multiple buffers (unlike in the blocking case).

MPI/MPI_SR_7.c code gives a simple example of non-blocking send+recv.

MPI_Testx

- For every "Wait" there is a corresponding "Test".
- While "Waits" are blocking, "Tests" are non-blocking, and can be used for polling if communication is completed.

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

- Flag is set to true if the communication described by the specified handle has completed.

Useful reading:

MPI 4 standard: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

MPI 3 (version 3.1) standard: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

OpenMPI documentation: https://www.open-mpi.org/doc/