Parallel Programming

Introduction to High Performance Computing
IN4049 TUDelft
2021/2022

Parallel programming

- Parallel programming models

    Distributed Memory: MPI,
    Shared Memory: OpenMP

  - GPU: CUDA (later)

- Freely available MPI libraries:
  - MPICH (http://www.mpich.org)
  - OpenMPI (http://www.open-mpi.org)

- Text/notes:
  - Notes MPI (on brightspace)
  - Website //www.mpi-forum.org/docs (lots info and tutorials)

Agenda

- Concepts
- Point-to-point communication
- Non-blocking operations
- Collective operations

Concepts: MPI Program

An MPI program is executed by a set of processes where each process has its own local data. Usually, one process is executed on one processor or core, but more processes can be executed on one processor/core.
Each process can access its local data and can exchange information and data with other processes by sending and receiving messages.

- In principle, each process could execute a different program - **MPMD: multiple program multiple data**.
- In practice, the **SPMD** (**single program multiple data**) style programming is implemented, where each process execute the same program (a process execute different parts of the program, selected by e.g., its process rank).

SPMD Model

- **Abstractions make programming and understanding easier**
- **<u>S</u>ingle <u>P</u>rogram <u>M</u>ultiple <u>D</u>ata**

→ Multiple instruction flows (instances) from a Single Program working on Multiple (different parts of) Data

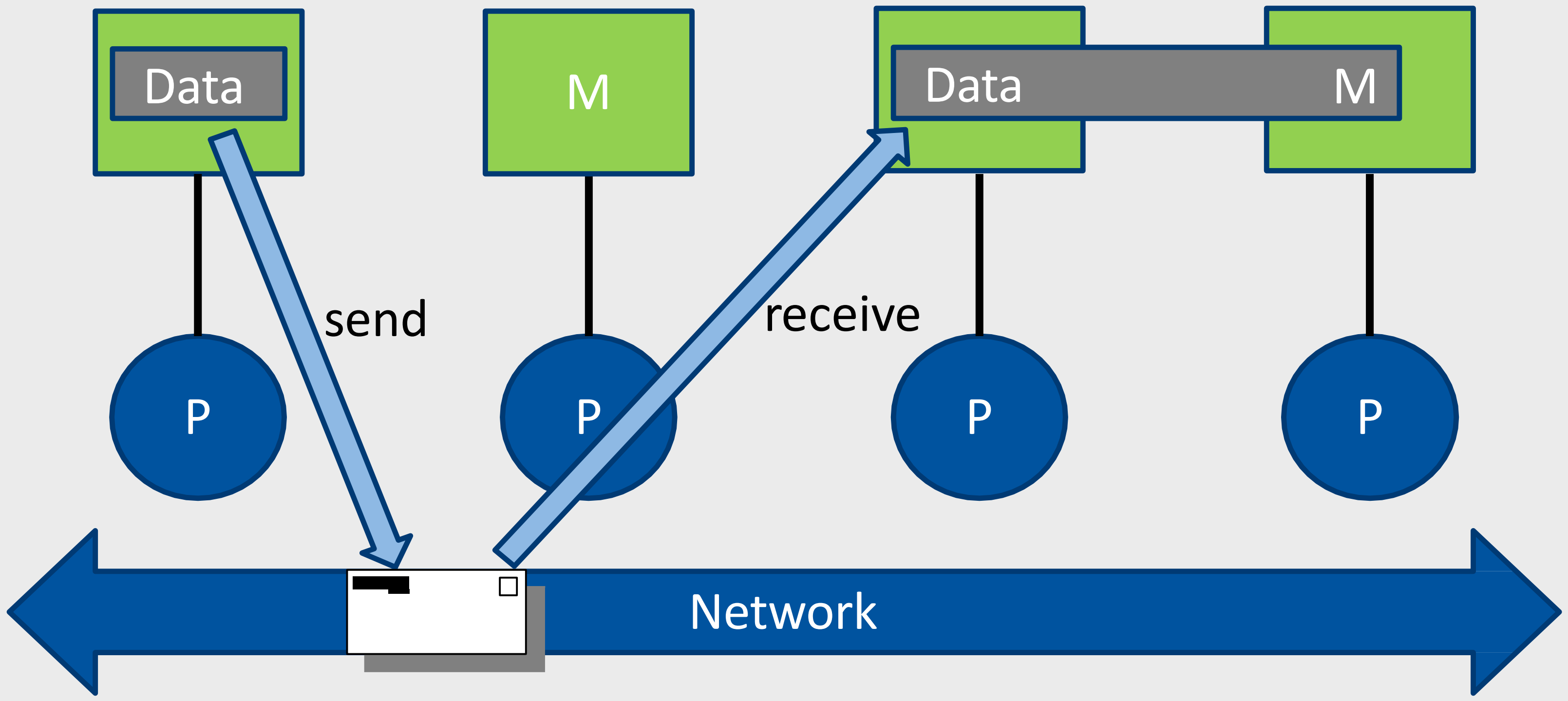
→ Instances could be threads (OpenMP) and/or processes (MPI)

→ Each instance receives a unique ID – can be used for flow control

```
if (myID == specificID)
{
     do_something();
}
else
{
     do_something_different();
}
```

Distributed Memory

- Each processing element (P) has its separate main memory block (M)



→ Data exchange is achieved through message passing over the network

■ Distributed Memory

→ Each processing element (P) has its separate main memory block (M)

→ Data exchange is achieved through message passing over the network

→ Message passing could be either explicit (MPI) or implicit (PGAS)

→ Programs typically implemented as a set of OS entities with own (virtual) address spaces – *processes*

→ No shared variables

- →No data races
- →Explicit synchronisation mostly unneeded
  - →Results as side effect of the send-receive semantics

Processes

- **A process is a running in-memory instance of an executable file**
  - → Executable code, e.g., binary machine instructions
  - → One or more threads of execution sharing memory address space
  - → Memory: data, heap, stack, processor state (CPU registers and flags)
  - → Operating system context (e.g. signals, I/O handles, etc.)
  - → PID
- **Isolation and protection**
  - → A process cannot interoperate with other processes or access their context (even on the same node) without the help of the operating system
  - → No direct inter-process data exchange (isolated/virtual address spaces)
  - → No direct inter-process synchronisation

MPI Basics

- **MPI Basics**
  - → Start-up, initialisation, finalisation, and shutdown
- **Point-to-Point Communication**
  - → Send and receive
  - → Basic MPI data types
  - → Message envelope
  - → Combined send and receive
  - → Send modes
  - → Non-blocking operations
  - → Common pitfalls

General Structure of an MPI Program

- **Start-up, initialisation, finalisation, and shutdown – C**

C

```
① #include <mpi.h>

② int main(int argc, char **argv)
  {
    … some code …
③   MPI_Init(&argc, &argv);

④
    … computation & communication …


⑤   MPI_Finalize();
⑥   … wrap-up …
    return 0;
  }
```

① Inclusion of the MPI header file

② Pre-initialisation mode: uncoordinated
- **No MPI function calls allowed with few exceptions**
- **All program instances run exactly the same code**

③ Initialisation of the MPI environment
Implicit synchronisation

④ Parallel MPI code Typically computation and communication

⑤ Finalisation of the MPI environment
Internal buffers are flushed

⑥ Post-finalisation mode: uncoordinated
- **No MPI function calls allowed with few exceptions**

General Structure of an MPI Program

- **Start-up, initialisation, finalisation, and shutdown – Fortran**

Fortran

```
PROGRAM example
  USE mpi_f08 ! USE mpi

   … some code …

  CALL MPI_Init(ierr)



  … computation   & communication …



  CALL MPI_Finalize(ierr)

  … wrap-up …
END
```

1. Inclusion of the MPI module
2. Pre-initialisation mode: uncoordinated
   - **No MPI function calls allowed with few exceptions**
   - **All program instances run exactly the same code**
3. Initialisation of the MPI environment Implicit synchronisation
4. Parallel MPI code Typically computation and communication
5. Finalisation of the MPI environment Internal buffers are flushed
6. Post-finalisation mode: uncoordinated
   - **No MPI function calls allowed with few exceptions**

}

General Structure of an MPI Program

- **How many processes are there in total?**
- **Who am I?**

C

```
#include <mpi.h>

int main(int argc, char **argv)
{
  … some code …
  int ierr = MPI_Init(&argc, &argv);
  … other code …
1 ierr = MPI_Comm_size(MPI_COMM_WORLD,
       &numberOfProcs);
2 ierr = MPI_Comm_rank(MPI_COMM_WORLD,
       &rank);
  … computation  & communication …
  ierr = MPI_Finalize();
  … wrap-up …
  return 0;
}
```

1 Obtains the number of processes (ranks) in the MPI program

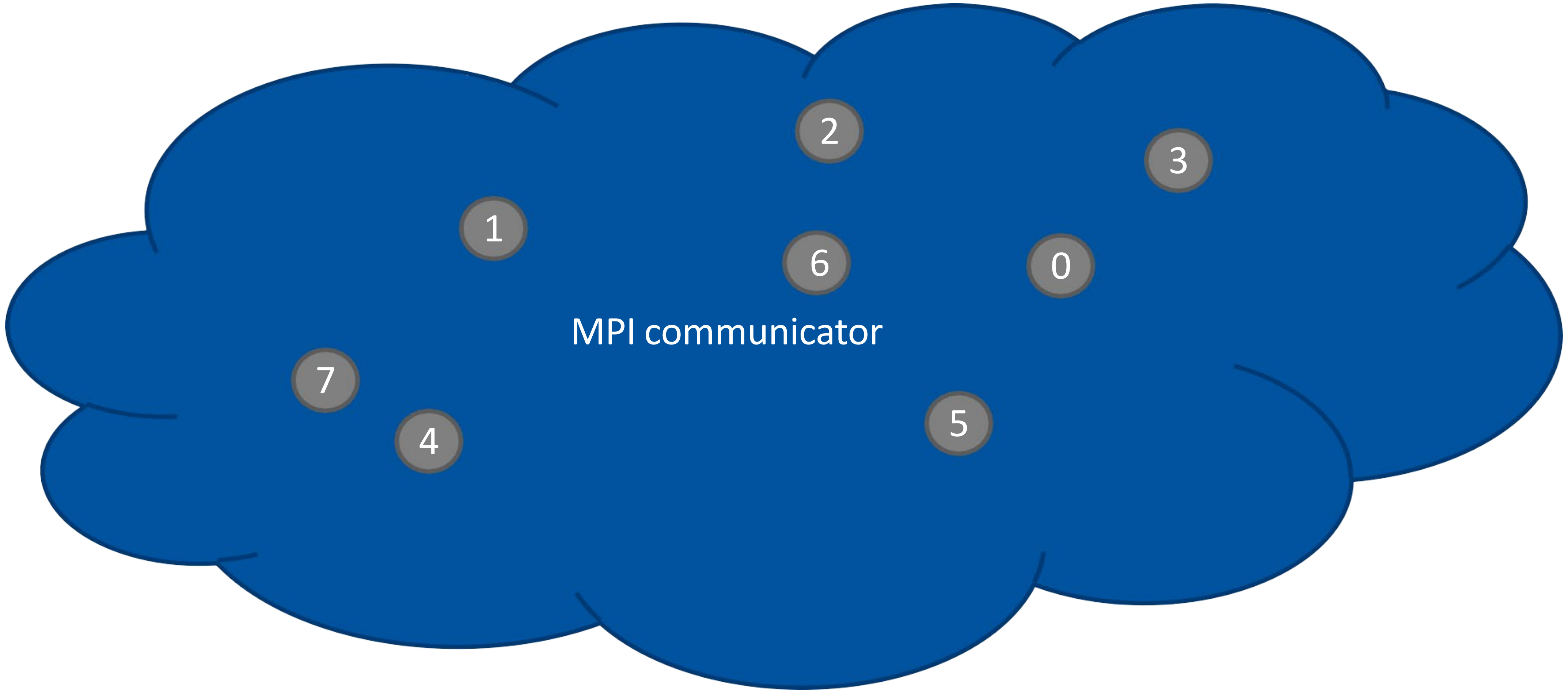
Example: if the job was started with 4 processes, then **numberOfProcs** will be set to 4 by the call

2 Obtains the identity of the calling process within the MPI program
NB: MPI processes are numbered starting from 0

Example: if there are 4 processes in the job, then **rank** receive value of 0 in the first process, 1 in the second process, and so on

Ranks

- **The processes in any MPI program are initially indistinguishable**
- **MPI_Init assigns each process a unique identity – rank**



Ranks

- **The processes in any MPI program are initially indistinguishable**
- **MPI_Init assigns each process a unique identity – rank**
  - → Without personality, the started MPI processes cannot do coordinated parallel work in the pre-initialisation mode
  - → Ranks range from 0 up to the total number of processes minus 1
- **Ranks are associated with the so-called communicators**
  - → Logical contexts where communication takes place
  - → Represent groups of MPI processes with some additional information
  - → The most important one is the world communicator **MPI_COMM_WORLD**
    - →Contains all processes launched *initially* as part of the MPI program
  - → Ranks are always provided in MPI calls in combination with the corresponding communicator

Basic MPI Use

- **Initialisation:**

```
C:        ierr = MPI_Init(&argc, &argv);
Fortran: CALL MPI_Init(ierr)
```

→ Initialises the MPI library and makes the process member of the world communicator

→ [C] Modern MPI implementations allow both arguments to be NULL, otherwise they *must* point to the arguments of **main()**

→ May not be called more than once for the duration of the program execution

- **Finalisation:**

```
C:        ierr = MPI_Finalize();
Fortran: CALL MPI_Finalize(ierr)
```

→ Cleans up the MPI library and prepares the process for termination

→ Must be called once before the process terminates

→ Having other code after the finalisation call is not recommended

Basic MPI Use

- **Number of processes in the MPI program:**

```
C:       ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
Fortran: CALL MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
```

→ Obtains the number of processes initially started in the MPI program (the size of the world communicator)

→ **size** is an integer variable

→ **MPI_COMM_WORLD** is a predefined constant *MPI handle* that represents the world communicator

- **Process identification:**

```
C:       ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
Fortran: CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
```

→ Determines the rank (unique ID) of the process within the world communicator

→ **rank** is an integer variable; receives value between 0 and #processes - 1

Agenda

→ Concepts

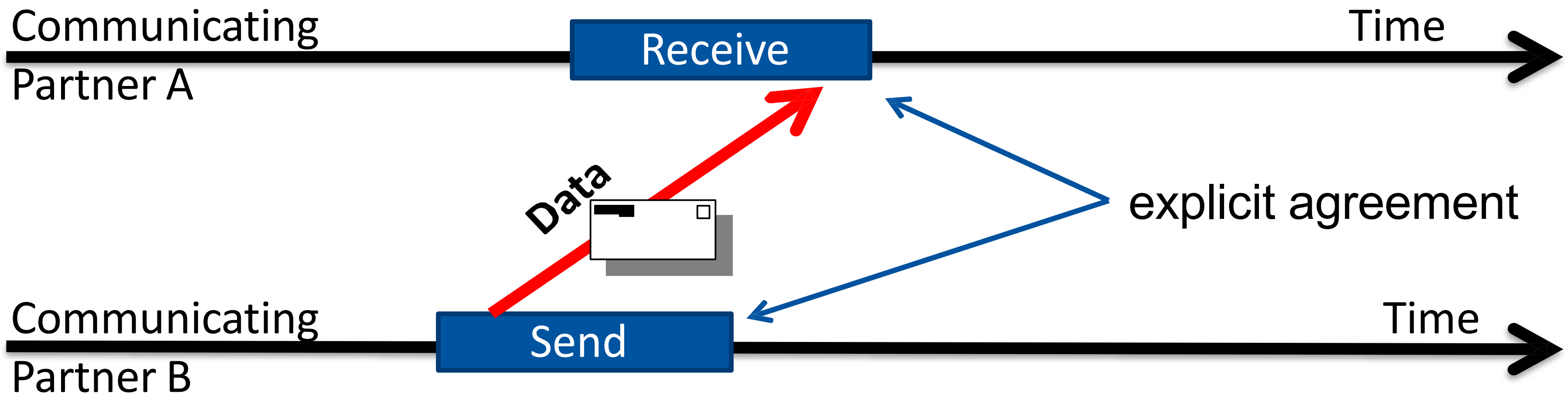
→ Point-to-point communication

→ Non-blocking operations

→ Collective operations

Message Passing

- **The goal is to enable communication between processes that share no memory space**



- **Explicit message passing requires:**
  - → Send and receive primitives (operations)
  - → Known addresses of both the sender and the receiver
  - → Specification of what has to be sent/received

Sending Data

- **Sending a message:**

What?

```
MPI_Send ( void *data, int count, MPI_Datatype type ,
           int dest, int tag, MPI_Comm comm )
```

To whom?

C

→ **data:**	location in memory of the data to be sent
→ **count:**	number of data elements to be sent (MPI is array-oriented)
→ **type:**	Handle of the *MPI datatype* of the buffer content
→ **dest:**	rank of the receiver
→ **tag:**	additional identification of the message
	ranges from 0 to UB (impl. dependant but not less than 32767)
→ **comm:**	communication context (communicator handle)

```
MPI_Send (data, count, type, dest, tag, comm, ierr)
```

Fortran

Receiving Data

- **Receiving a message:**

What?

```
MPI_Recv ( void *data, int count, MPI_Datatype type ,
           int source, int tag, MPI_Comm comm , MPI_Status *status)
```

C

From whom?

→ **data:**	location of the receive buffer
→ **count:**	size of the receive buffer in data elements
→ **type:**	Handle of the MPI datatype of the data elements
→ **source:**	rank of the sender or **MPI_ANY_SOURCE** (wildcard)
→ **tag:**	message tag or **MPI_ANY_TAG** (wildcard)
→ **comm:**	communication context
→ **status:**	status of the receive operation or **MPI_STATUS_IGNORE**

```
MPI_Recv (data, count, type, src, tag, comm, status, ierr)
```

Fortran

MPI Datatypes

- **MPI is a library – it cannot infer the type of elements in the supplied buffer at run time and that's why it has to be told what it is**

- **MPI datatypes tell MPI how to:**
  - → read binary values from the send buffer
  - → write binary values into the receive buffer
  - → correctly apply value alignments
  - → convert between machine representations in heterogeneous environments

- **MPI datatype must match the language type(s) in the data buffer**
- **MPI datatypes are handles and cannot be used to declare variables**

MPI Datatypes

- **MPI provides many predefined datatypes for each language binding:**
  - → C

MPI data type	C data type
MPI_CHAR	char
MPI_SHORT	short
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_UNSIGNED_INT	unsigned int
…	…
MPI_BYTE	-

8 binary bits
no conversion

A Complete MPI Example

C

```c
#include <mpi.h>

int main(int argc, char **argv)
{
  int nprocs, rank, data;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,
       &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,
       &rank);
  if (rank == 0)
    MPI_Recv(&data, 1, MPI_INT, 1, 0,
       MPI_COMM_WORLD, &status);
  else if (rank == 1)
    MPI_Send(&data, 1, MPI_INT, 0, 0,
       MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```

1. Initialise the MPI library
2. Identify current process
3. Behave differently based on the rank
4. Communicate
5. Clean up the MPI library

Compiling MPI Programs

- **MPI is a typical library with C header files, Fortran modules, etc.**
- **Some MPI vendors provide convenience compiler wrappers:**

$$\begin{tikzcd}
\boxed{\texttt{c}} \arrow[r] & \boxed{\texttt{mpicc}} \\
\boxed{\texttt{c++}} \arrow[r] & \boxed{\texttt{mpic++}} \\
\boxed{\texttt{f90}} \arrow[r] & \boxed{\texttt{mpif90}}
\end{tikzcd}$$

Executing MPI Programs

- **Most MPI implementations provide a special launcher program:**

```
mpiexec –n nprocs … program <arg1> <arg2> <arg3> …
```

→ launches **nprocs** instances of **program** with command-line arguments **arg1, arg2, …** and provides the MPI library with enough information in order to establish network connections between the processes

- **The standard specifies the *mpiexec* program but does not require it:**

→ IBM BG/Q: **runjob --np 1024 …**

→ On the DAS5 cluster for the lab: **prun**

→ SLURM resource manager: **srun …** (batch jobs >15 min)

Executing MPI Programs

- **Most MPI implementations provide a special launcher program:**

```
mpiexec -n nprocs … program <arg1> <arg2> <arg3> …
```

  → launches nprocs instances of **program** with command-line arguments **arg1, arg2, …** and provides the MPI library with enough information in order to establish network connections between the processes

  → Sometimes called **prun** (on the computers in our lab)

- **The launcher often performs more than simply launching processes:**

  → Helps MPI processes find each other and establish the world communicator

  → Redirects the standard output of all ranks to the terminal

  → Redirects the terminal input to the standard input of rank 0

  → Forwards received signals (Unix-specific)

Message Reception and Status

- **The receive buffer must be able to fit the entire message**

→ send count ≤ receive count	**OK** (but check status)
→ send count > receive count	**ERROR** (message truncation)

- **The MPI status object holds information about the received message**

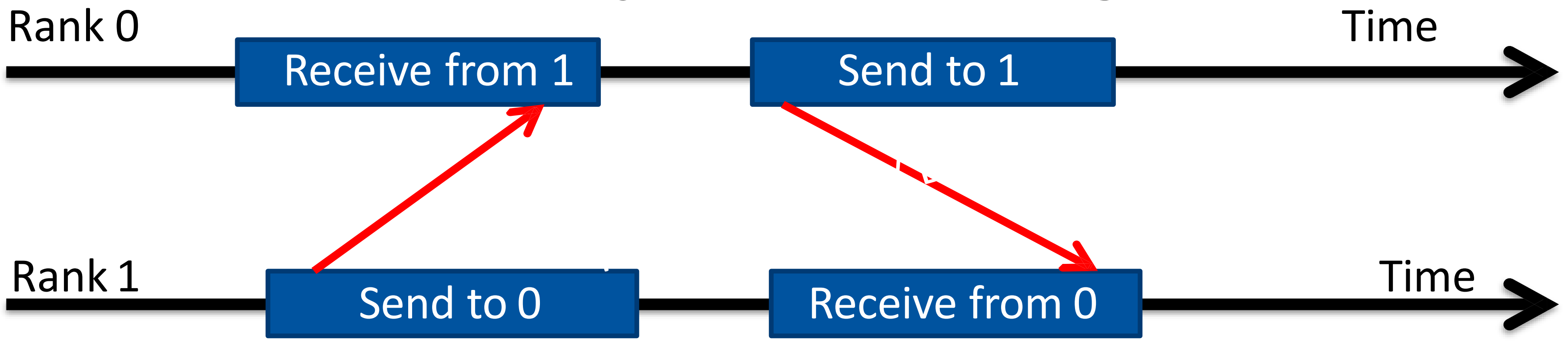
- **C: MPI_Status status;**

→ **status.MPI_SOURCE**	message source rank
→ **status.MPI_TAG**	message tag
→ **status.MPI_ERROR**	receive status code

Deadlocks

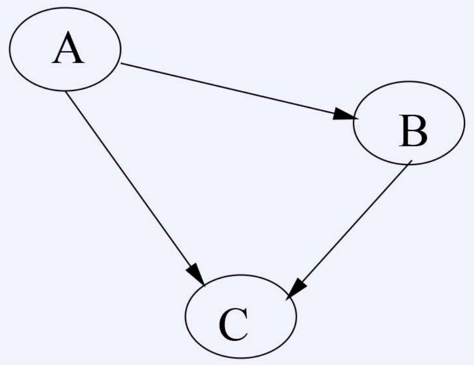
- **Both MPI_Send and MPI_Recv calls are blocking:**
  - → The receive operation only returns after a matching message has arrived
  - → The send operation ***might*** be buffered *(implementation-specific!!!)* and therefore return before the message is actually placed onto the network
  - → Larger messages are usually sent only when both the send and the receive operations are active (synchronously)
  - → **Never rely on any implementation-specific behaviour!!! (secure implementation)**

- **Deadlock prevention in a typical data exchange scenario:**



Communication scheduling (1)

Blocking communication - Deadlock



$$\begin{tikzcd} A \arrow[r] \arrow[dr] & B \arrow[d] \\ & C \end{tikzcd}$$

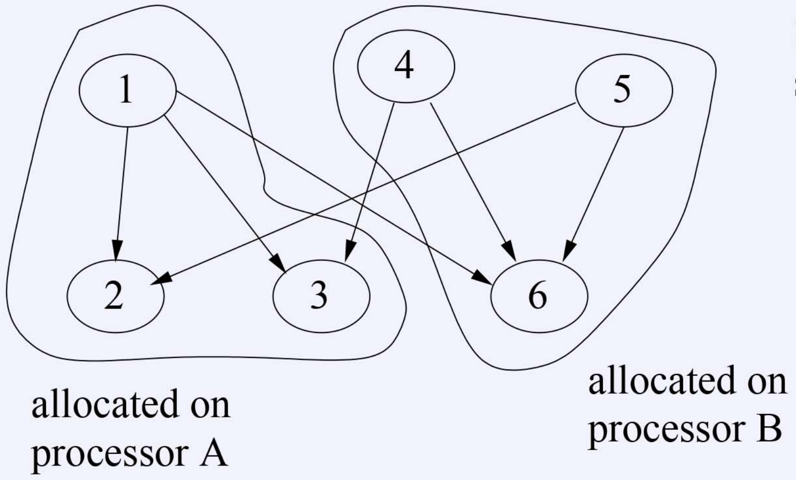
deeadlock:

Process A:	Process B:	Process C:
send(..., C);	receive(..., A);	
send(..., B);		
	send(..., C);	receive(..., B);
		receive(..., A);

The deadlock in this case can be often avoided by executing the receives in a sequence by receiving a message coming from a higher level process before that from a lower level process. (P.S. the levels can be obtained with a leveling algorithm). Extra complication occurs when there are messages coming from more than 1 processes at the same level.

Communication scheduling (2)

Blocking communication - Deadlock



More than one tasks are mapped onto the same processor: Some combination of task-scheduling and the sequence of send and receives may cause deadlock. For example:

Proessor A:	Processor B:
execute task (1);	execute task (4);
send(6, B);	send(3, A);
receive(5, B);	execute task (5);
execute task (2);	send(2, A);
receive(4, B);	receive(1, A);
execute task (3);	execute task (6);

Combined Send and Receive

```
MPI_Sendrecv (void *senddata, int sendcount, MPI_Datatype sendtype,
              int dest, int sendtag, void *recvdata, int recvcount,
              MPI_Datatype recvtype, int source, int recvtag,
              MPI_Comm comm, MPI_Status *status)
```

- **Combines message send and receive into a single call**

	Send	Receive
Data	**senddata**	**recvdata**
Count	**sendcount**	**recvcount**
Type	**sendtype**	**recvtype**
Destination	**dest**	**-**
Source	**-**	**source**
Tag	**sendtag**	**recvtag**
Communicator	**comm**	**comm**
Receive status	**-**	**status**

Combined Send and Receive

```
MPI_Sendrecv (void *senddata, int sendcount, MPI_Datatype sendtype,
              int dest, int sendtag, void *recvdata, int recvcount,
              MPI_Datatype recvtype, int source, int recvtag,
              MPI_Comm comm, MPI_Status *status)
```

- **Sends one message and receives one message (in any order) without deadlocking (unless unmatched)**
- **Send and receive buffers must not overlap!**

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

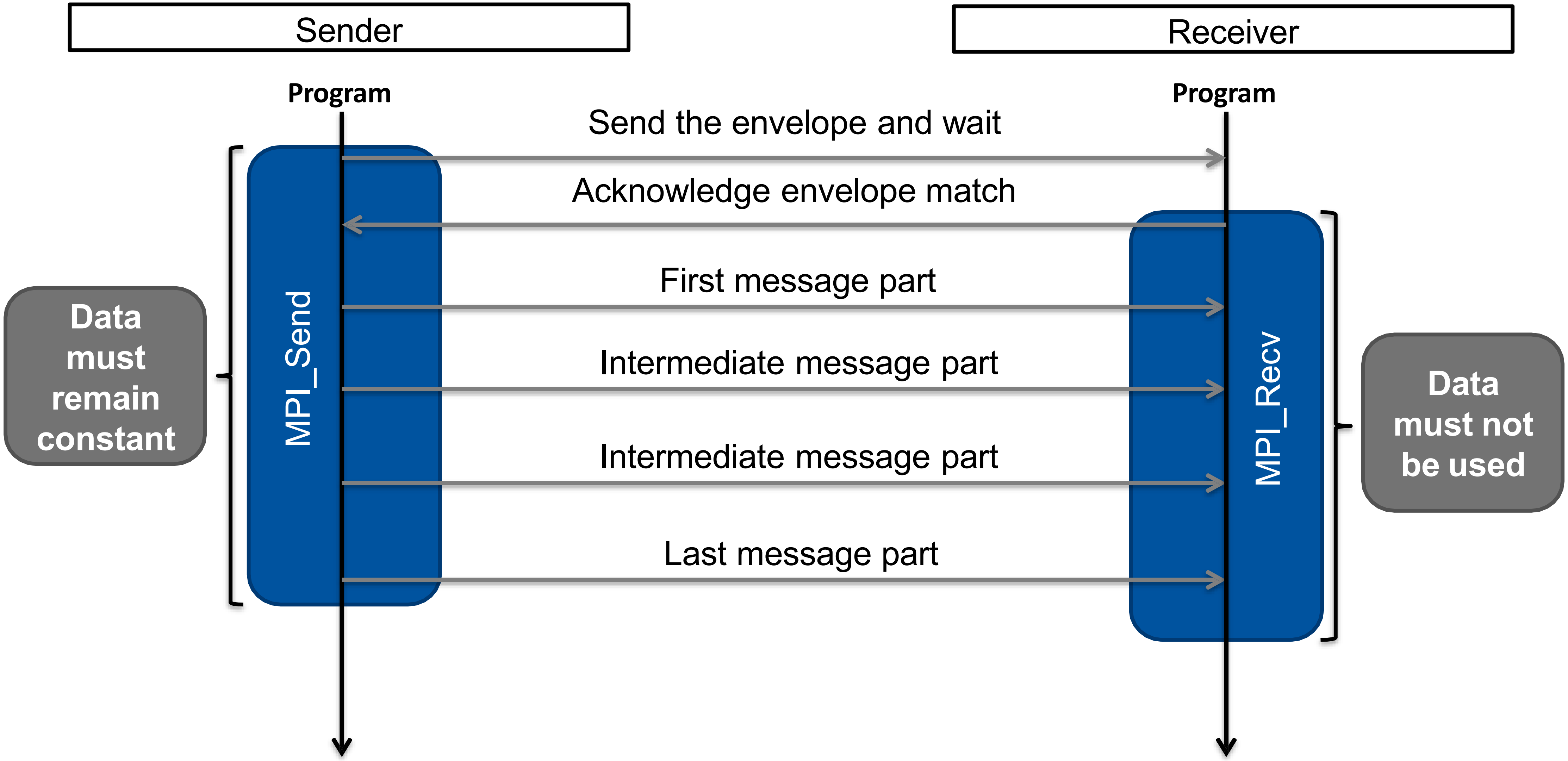
- **First sends a message to *dest*, then receives a message from *source*, using the same memory location, elements count and datatype for both operations**
- **Usually slower than MPI_Sendrecv and might use more memory**

Agenda

- → Concepts
- → Point-to-point communication
- → Non-blocking operations
- → Collective operations

Blocking Calls

- **Blocking send (w/o buffering) and receive calls:**



Sender
Receiver
Program
Program
Send the envelope and wait
Acknowledge envelope match
First message part
Intermediate message part
Intermediate message part
Last message part
MPI_Send
MPI_Recv
Data must remain constant
Data must not be used

Non-Blocking Calls

- **Non-blocking MPI calls return immediately while the communication operation continues asynchronously in the background**

- **Each non-blocking operation is represented by a request handle:**

→ C:	**MPI_Request**
→ Fortran:	**INTEGER**
→ Fortran 2008:	**TYPE(MPI_Request)**

- **Non-blocking operations are monitored by certain MPI calls, most notably by the *test* and *wait* MPI calls**
- **Blocking MPI calls are equivalent to making a non-blocking call and waiting immediately afterwards for the operation to complete**
- **Used to overlay communication and computation and to prevent possible deadlocks**

Non-Blocking Send and Receive

- **Initiation of non-blocking send and receive operations:**

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

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

→ **request:** on success set to the handle of the non-blocking operation

- **Blocking wait for completion:**

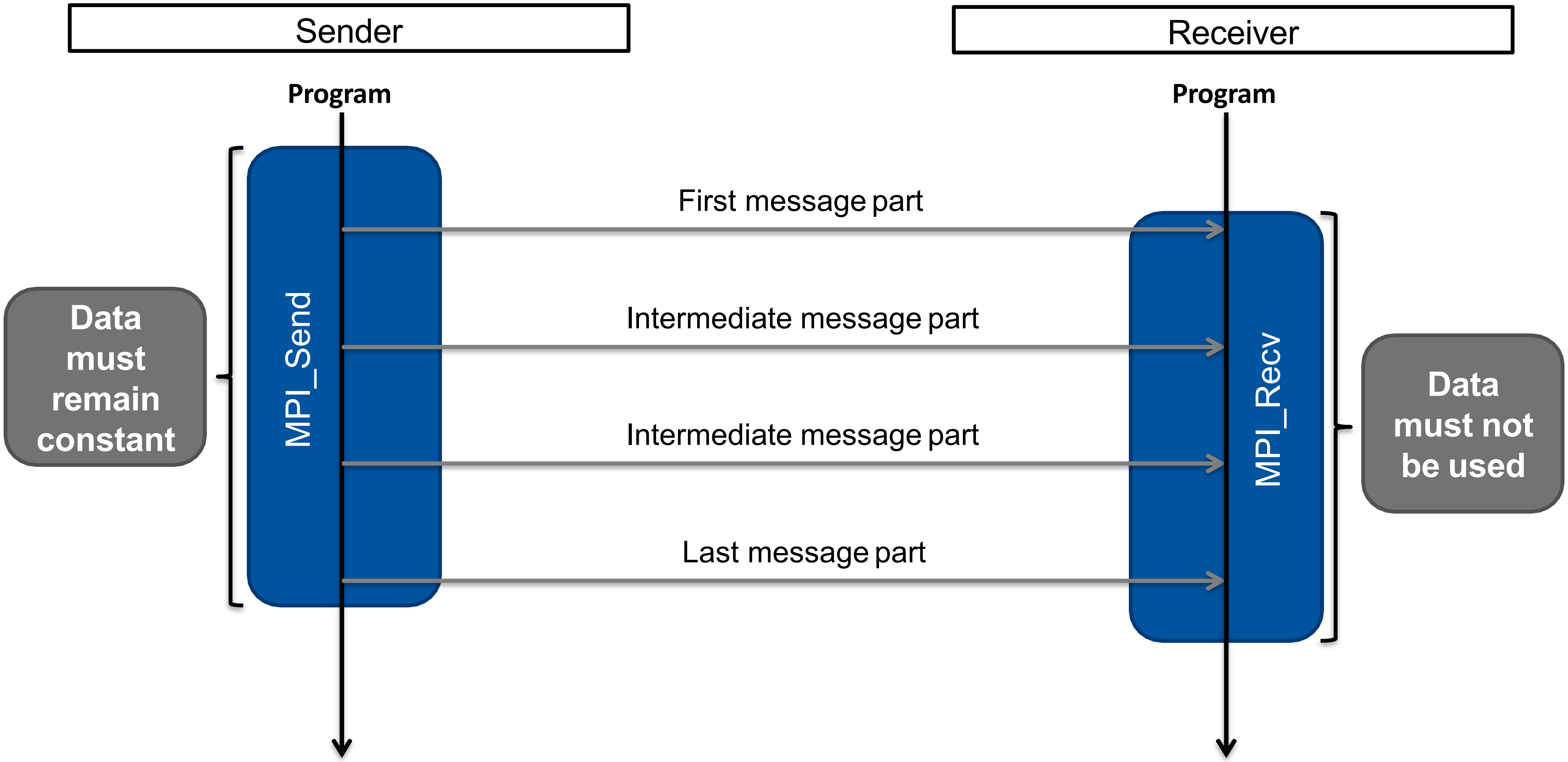
```
  MPI_Wait (MPI_Request *request, MPI_Status *status)
```

→ **request:** handle for an active non-blocking operation
freed and set to **MPI_REQUEST_NULL** upon successful return

→ **status:** status of the completed operation

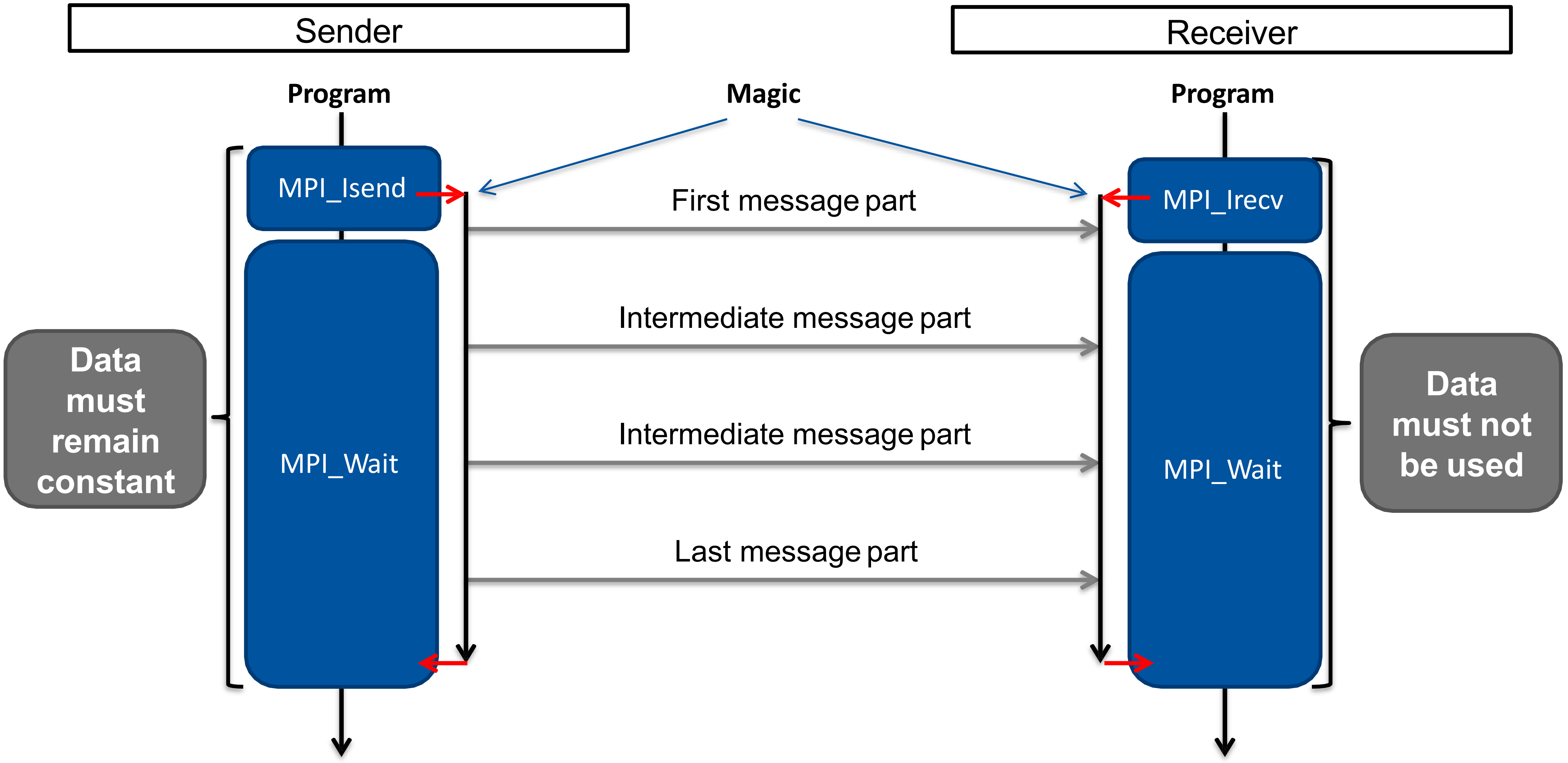
Communication: Blocking

- **Blocking send (w/o buffering) and receive calls:**



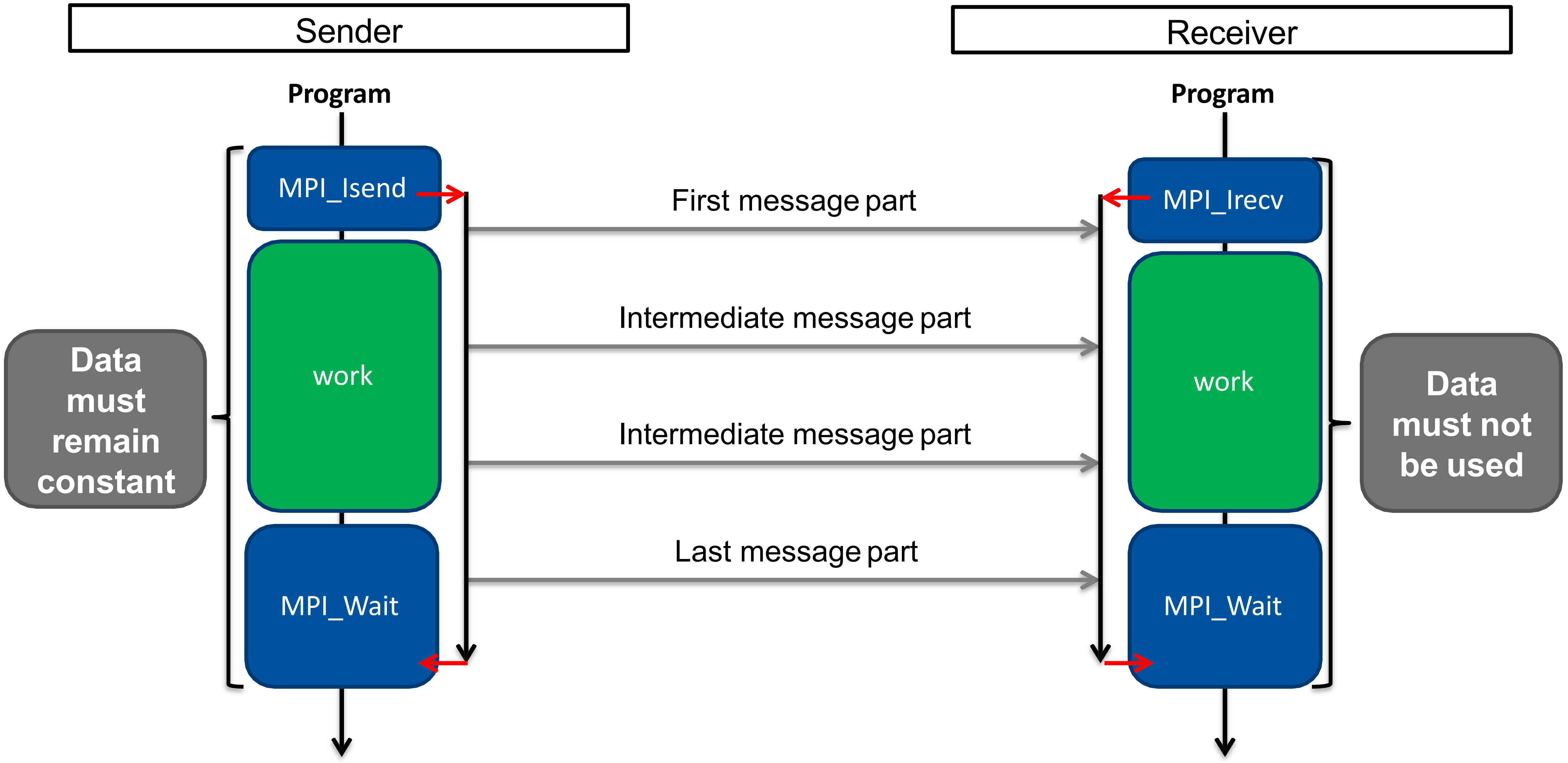
Communication: Non-blocking

- **Equivalent with non-blocking calls:**



Communication-Computation Overlay

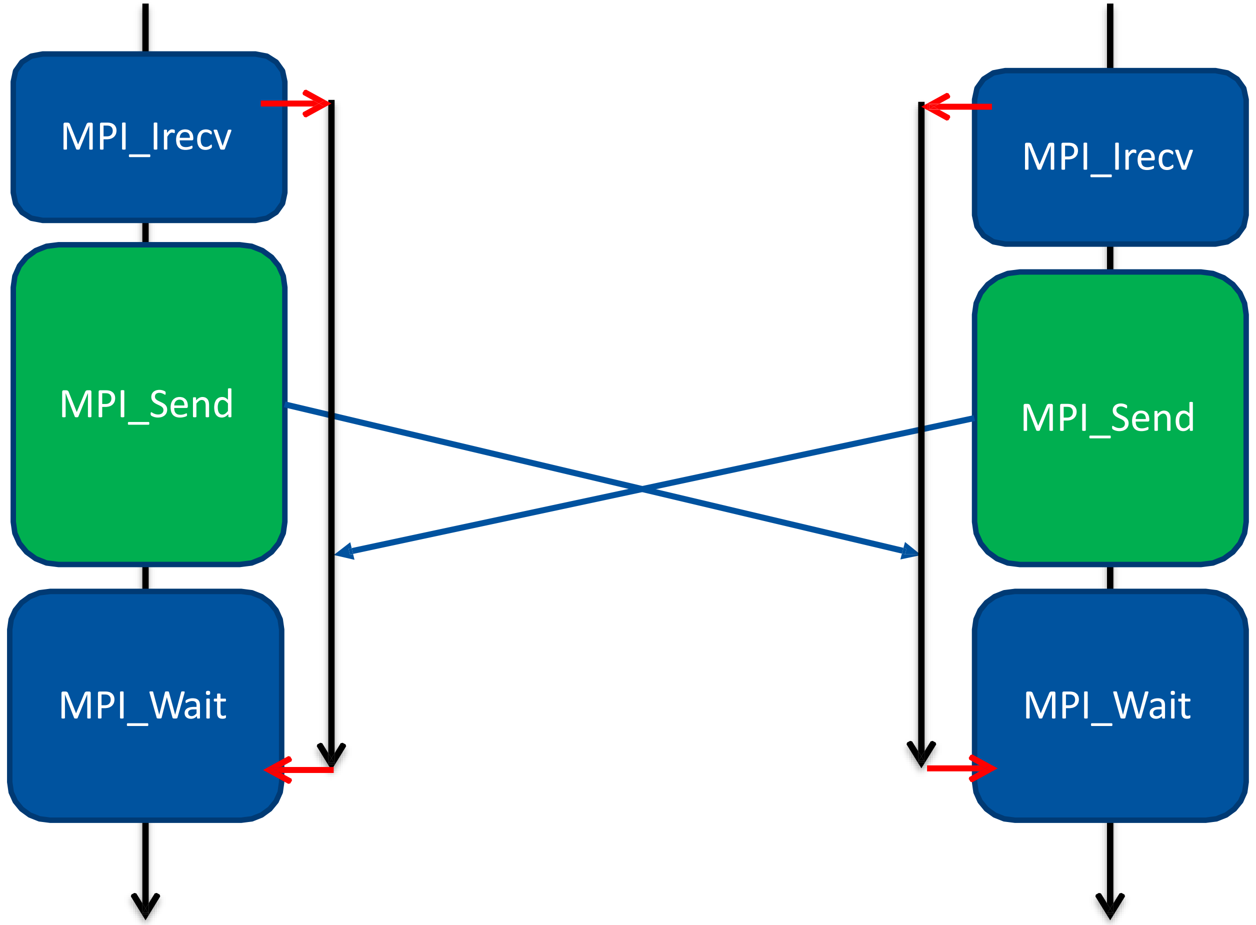
- **Other work can be done in between*:**



→ Higher efficiency/utilization

Deadlock Prevention

- **Non-blocking operations can be used to prevent deadlocks in symmetric code:**



- **That is how MPI_Sendrecv is usually implemented**

Non-Blocking Request Testing

- **Test if given operation has completed:**

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

→ **flag:** **true** if the operation has completed, otherwise **false**

→ **status:** status of the completed operation, only set if **flag** is **true**

→ Can be (and usually is) called repeatedly inside a loop

→ Upon completion of the operation (i.e. when **flag** is **true**), the operation is freed and the request handle is set to **MPI_REQUEST_NULL**

- **If called with a null request (MPI_REQUEST_NULL):**

→ **MPI_Wait** returns immediately with an empty **status**

→ **MPI_Test** sets **flag** to **true** and returns an empty **status**

Communication Modes

- **There are four send modes in MPI:**
  - → Standard
  - → Synchronous
  - → Buffered
  - → Ready
- **Send modes differ in the relation between the completion of the operation and the actual message transfer**
- **Single receive mode:**
  - → Synchronous

Send Modes

- **Standard mode**
  - → The call blocks until the message has either been transferred or copied to an internal buffer for later delivery
- **Synchronous mode**
  - → The call blocks until a matching receive has been posted and the message reception has started
- **Buffered mode**
  - → The call blocks until the message has been copied to a user-supplied buffer. Actual transmission may happen at a later point
- **Ready mode (don't use!)**
  - → The operation succeeds only if a matching receive has already been posted. Behaves as standard send in every other aspect

Send Modes

- **Call names:**

→ **MPI_Send**	blocking standard send
→ **MPI_Isend**	non-blocking standard send
→ **MPI_Ssend**	blocking synchronous send
→ **MPI_Issend**	non-blocking synchronous send
→ **MPI_Bsend**	blocking buffered send
→ **MPI_Ibsend**	non-blocking buffered send
→ **MPI_Rsend**	blocking ready-mode send
→ **MPI_Irsend**	non-blocking ready-mode send

- **Buffered operations require an explicitly provided user buffer**
  - → **MPI_Buffer_attach (void *buf, int size)**
  - → **MPI_Buffer_detach (void *buf, int *size)**
  - → Buffer size must account for the envelope size (**MPI_BSEND_OVERHEAD**)

Send Modes

- **One rarely needs anything else except the standard send**

- **The synchronous send can be used to synchronise two ranks**

- **Simple correctness check**
  - → Replacing all blocking standard sends with blocking synchronous sends should not result in deadlock
  - → If program deadlocks, you are relying on the buffering behaviour of the standard send → change your algorithm

- **Buffered sends guarantee that messages are always buffered, but it is possible to run out of buffer space**
  - → No way to test if the buffer is still in use by MPI

Utility Calls

- **Attempt to abort all MPI processes in a given communicator:**

```
MPI_Abort (MPI_Comm comm, int errorcode)
```

→ errorcode is returned to the OS if supported by the implementation.

→ Note: Open MPI does not return the error code to the OS.

- **Portable timer function:**

```
double MPI_Wtime ()
```

→ Returns the wall-clock time that has elapsed since an unspecified (but fixed for successive invocations) point in the past

- **Obtain a string ID of the processor:**

```
MPI_Get_processor_name (char *name, int *resultlen)
```

→ **name:** buffer of at least **MPI_MAX_PROCESSOR_NAME** characters

→ **resultlen:** length of the returned processor ID (w/o the **'\0'** terminator)

Message Passing: MVP Example

```
for(i=0; i<dim; i++) {
     y[i]=0;
     for(j=0; j<dim; j++) {
       y[i]=y[i] + A[i,j] * x[j]; } }
```

- Which processors gets which data?
  - A: chunk of rows, i= id**k*, …, (id +1)**k*-1
  - x: chunk of elements, #elements=*k=dim/procs*
- Communication?
- Synchronization?

Context: iterative solution of Ax=b, where the matrix A is constant and (already) distributed, but new vector x_part local at each processor id has to be sent to all other processors after each iteration.

Message Passing: MVP version 1

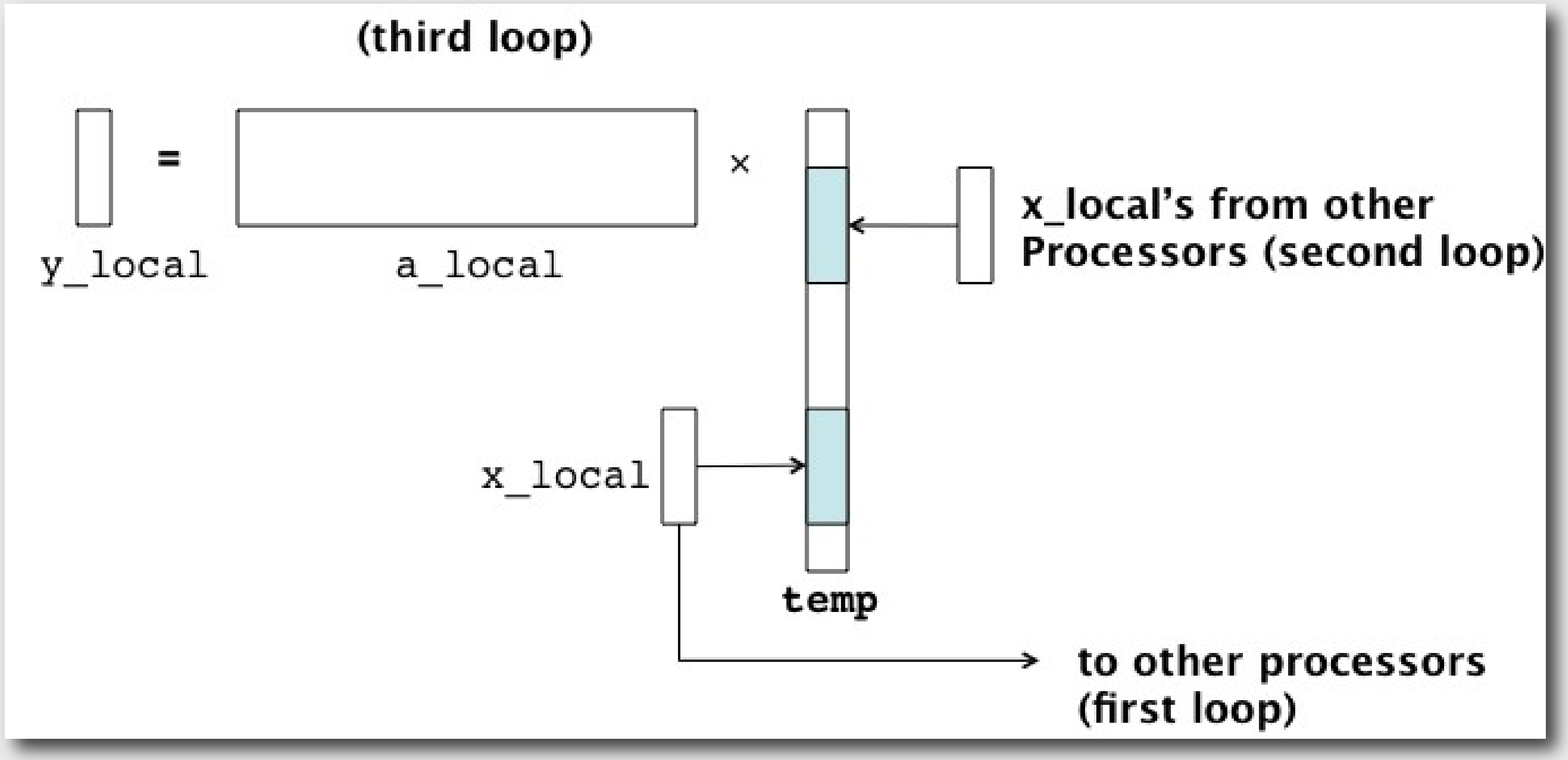
```
float a_local[m/procs,n], y_local[m/procs], x_local[n/procs];
float temp[n];
Int   mypid;

mypid=proc(); k=n/procs;
for(i=0;i<procs;i++){
      if(mypid!=i) send(&x_local[0],k,i);}
for(i=0;i<procs;i++){
      if(mypid!=i)
         receive(&temp[i*k],k,i);
      else
         copy(temp[i*k],k,x_local)
}
for(i=0;i<k;i++)
  for(j=0;j<n;j++)
    y_local[i]=y_local[i]+a_local[i,j]*temp[j];
```

Send my chunk of x to all other processors

Receive all chunks of x from all other processors

Do all local MVPs

MPV: data structures – version 1


(third loop)
=
×
y_local
a_local
x_local's from other Processors (second loop)
x_local
temp
to other processors (first loop)

Message Passing: MVP – version 1

- Memory
  - *need local buffer (`temp[]`) of global size n anyway!*
- Performance issues
  - *communication with (proc-1) nodes, for each*
    - *sending n/proc items*
    - *receiving n/proc items*

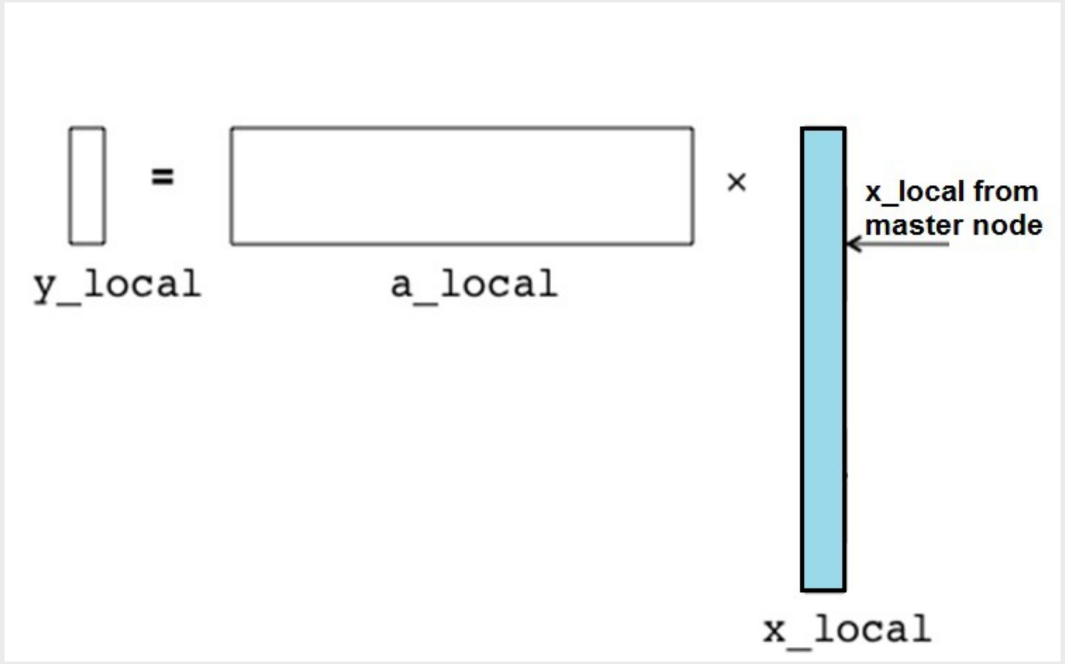
Message Passing: MPV – version 2

- Another version of local structures results in:

```
float a_local[m/procs,n], y_local[m/procs], x_local[n];
int mypid;

mypid=proc(); k=n/procs;
if (mypid==0)
      send_all(&x_local[0],n);
else
      receive(&x_local[0],n,0);

for(i=0;i<k;i++)
   for(j=0;j<n;j++)
      y_local[i]=y_local[i]+a_local[i,j]*x_local[j];

     …
Update part of x_local() on each processor;
if (mypid!=0) send(&x_local[0],k,0)
else receive x_local from processor 1,…, nr_procs-1
```

$\}$ Replicate x to all processors from processor 0

MPV: data structures – version 2



Message Passing: MVP – version 2

- Memory
  - *no need for x_local[n/procs]*
  - *replace temp with full x_local*

- Performance issues
  - *broadcast: sending n items*
  - *receiving n items*
  - *fewer but larger messages (same amount of data)*

Common Pitfalls – C/C++

- **Do not pass pointers to pointers in MPI calls**

```
int scalar;
MPI_Send(&scalar, MPI_INT, 1, …

int array[5];  MPI_Send(array,
MPI_INT, 5, …
… or …
MPI_Send(&array[0], MPI_INT, 5, …

int *pointer = new int[5];
Fill array pointer …
MPI_Send(pointer, MPI_INT, 5, …
… or …
MPI_Send(&pointer[0], MPI_INT, 5, …

// ERRONEOUS
MPI_Send(&pointer, MPI_INT, 5, …
```

&array will (often) work too, but is not recommended

Will result in the value of the pointer itself (i.e. the memory address) being sent, possibly accessing past allocated memory

Common Pitfalls – C/C++

- **Do not pass pointers to pointers in MPI calls**

```
void func (int scalar)
{
  MPI_Send(&scalar, MPI_INT, 1, …

void func (int& scalar)
{
  MPI_Send(&scalar, MPI_INT, 1, …

void func (int *scalar)
{
  MPI_Send(scalar, MPI_INT, 1, …

void func (int *array)
{
  MPI_Send(array, MPI_INT, 5, …
  … or …
  MPI_Send(&array[0], MPI_INT, 5, …
```

Common Pitfalls – C/C++

- **Use flat multidimensional arrays; arrays of pointers do not work**

```
// Static arrays are OK
int mat2d[10][10];
MPI_Send(&mat2d, MPI_INT, 10*10, …

// Flat dynamic arrays are OK
int *flat2d = new int[10*10];
Fill array flat2d …
MPI_Send(flat2d, MPI_INT, 10*10, …

// DOES NOT WORK
int **p2d[10] = new int*[10];
for (int i = 0; i < 10; i++)
   p2d[i] = new int[10];
MPI_Send(p2d, MPI_INT, 10*10, …
… or …
MPI_Send(&p2d[0][0], MPI_INT, 10*10, …
```

MPI has no way to know that there is a hierarchy of pointers

Common Pitfalls – C/C++

- **Passing pointer values around makes little to no sense**
  - → Pointer values are process-specific
  - → No guarantee that memory allocations are made at the same addresses in different processes
    - → Especially on heterogeneous architectures, e.g., host + co-processor
  - → No guarantee that processes are laid out in memory the same way, even when they run on the same host
    - → Address space layout randomisation
    - → Stack and heap protection

- **Relative pointers (=index offset) could be passed around**

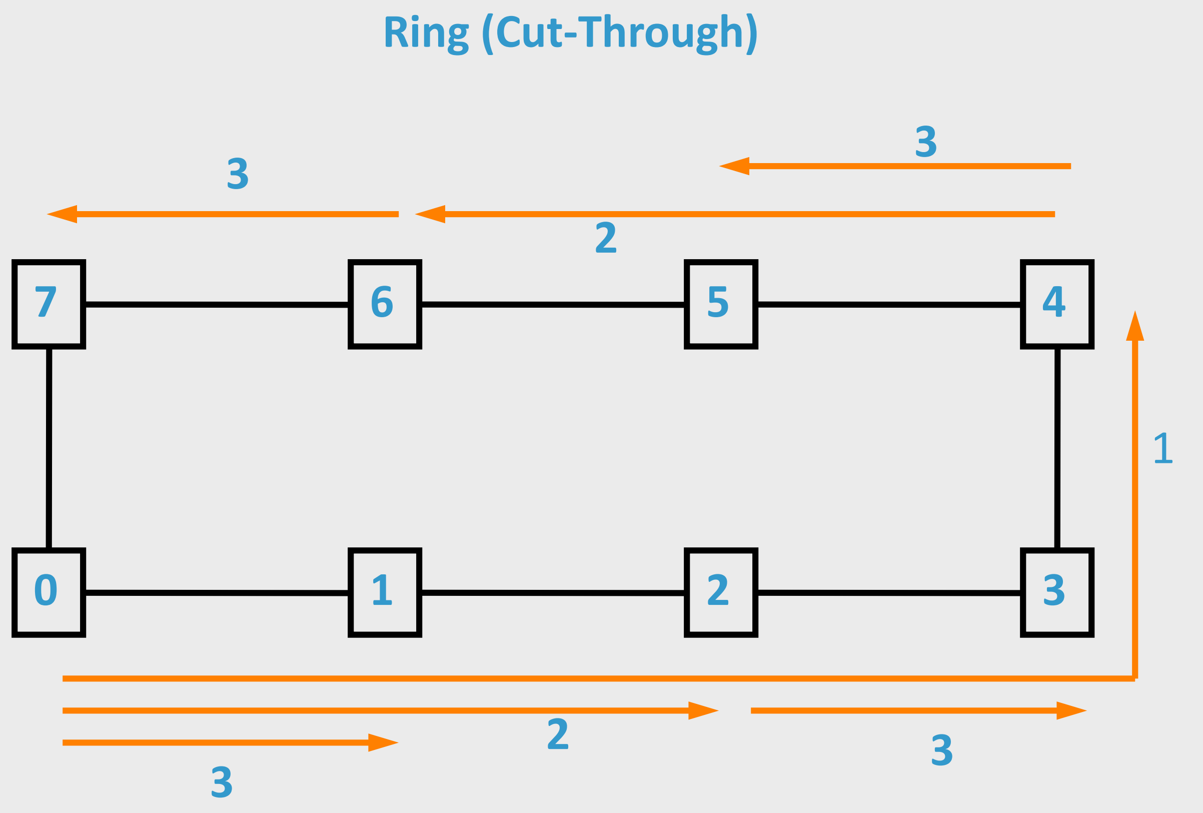
Message Passing: Summary

- No notion of global data
- Data communication is done by explicit message passing
  - expensive performance-wise
- Trade-off between:
  - one-copy data
    - *more communication is needed, less consistency issues*
  - local data replication
    - *less communication, consistency is problematic*
- Techniques to improve performance:
  - replicate read-only data
  - computation and communication overlapping
  - message aggregation

Collective operations

- Barrier — Synchronizes all processes (waits for all processes have arrived at the point of the call).
- Broadcast — Sends data from one to all processes.
- Gather — Gather data from all processes to one process.
- Scatter — Scatters data from one process to all processes.
- Reduction — Reduce to one single result, such as sums.

Example: Ring – one2all Broadcast



Example: MPI_Broadcast

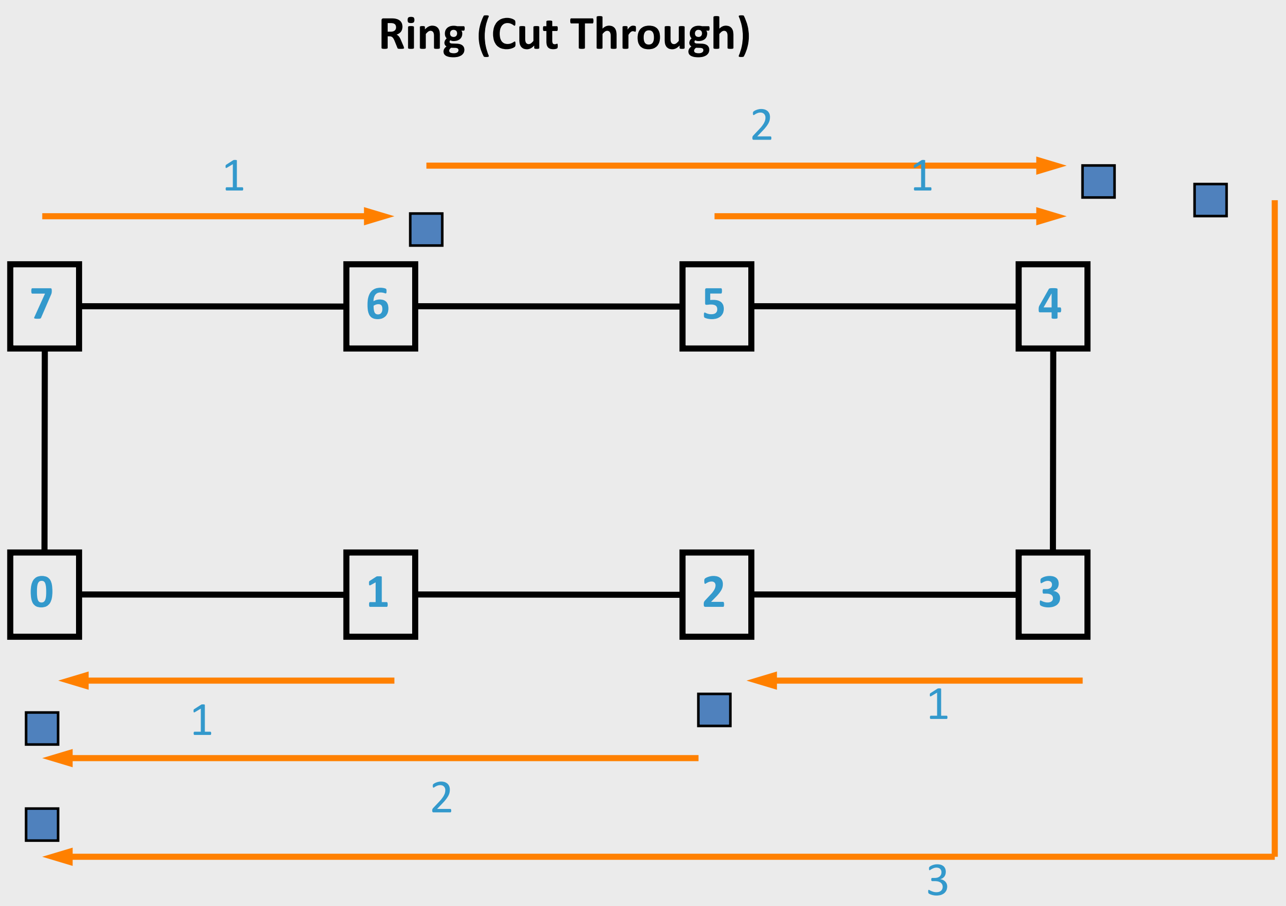
```
MPI_Bcast( void* data, int count, MPI_Datatype
datatype, int root, MPI_Comm communicator)
```

Built-in MPI collective functions are often faster (optimized) than user own implementations.

```
>>> cd tutorials
>>> ./run.py compare_bcast /home/kendall/bin/mpirun -n 16 machinefile
hosts ./compare_bcast 100000 10

Data size = 400000, Trials = 10
Avg my_bcast time = 0.510873
Avg MPI_Bcast time = 0.126835
```

Here my_bcast is a function implementing broadcast using a for-loop with (P-1) pairs of MPI_Send and MPI_Recv

Example: Ring – all2one Reduction

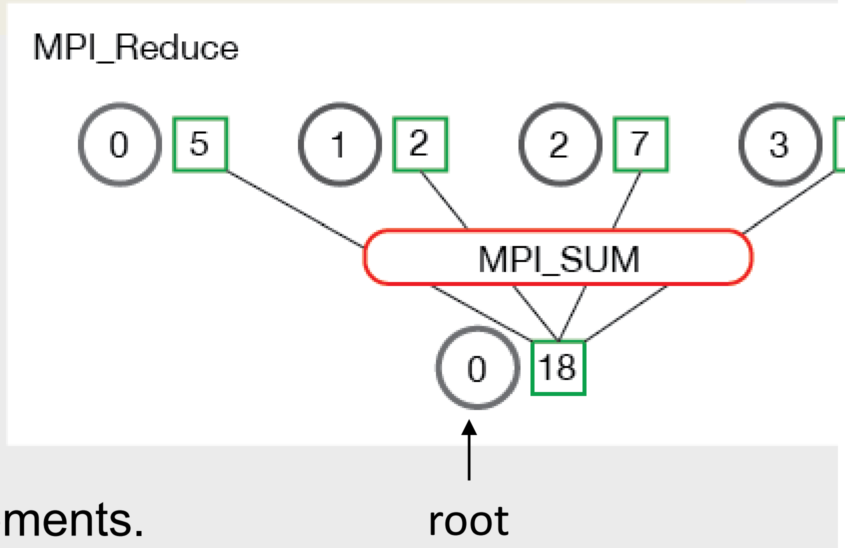


MPI_Reduce

MPI_Reduce(void* send_data, void* recv_data, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm communicator)

The reduction operations defined by MPI include:

- MPI_MAX - Returns the maximum element.
- MPI_MIN - Returns the minimum element.
- MPI_SUM - Sums the elements.
- MPI_PROD - Multiplies all elements.
- MPI_LAND - Performs a logical *and* across the elements.
- MPI_LOR - Performs a logical *or* across the elements.
- MPI_BAND - Performs a bitwise *and* across the bits of the elements.
- MPI_BOR - Performs a bitwise *or* across the bits of the elements.
- MPI_MAXLOC - Returns the maximum value and the rank of the process that owns it
- MPI_MINLOC - Returns the minimum value and the rank of the process that owns it.



Example: Compute average number using MPI_Reduce

```
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);
 // Sum the numbers locally
float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++) {
     local_sum += rand_nums[i];        }
 // Print the local sum and the local average on each process
printf("Local sum for process %d - %f, avg = %f\n", my_rank, local_sum,
          local_sum / num_elements_per_proc);
// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,
                    MPI_COMM_WORLD);
// Print the result
if (my_rank == 0) {
   printf("Total sum = %f, avg = %f\n", global_sum,
             global_sum / (nr_procs * num_elements_per_proc)); }
```