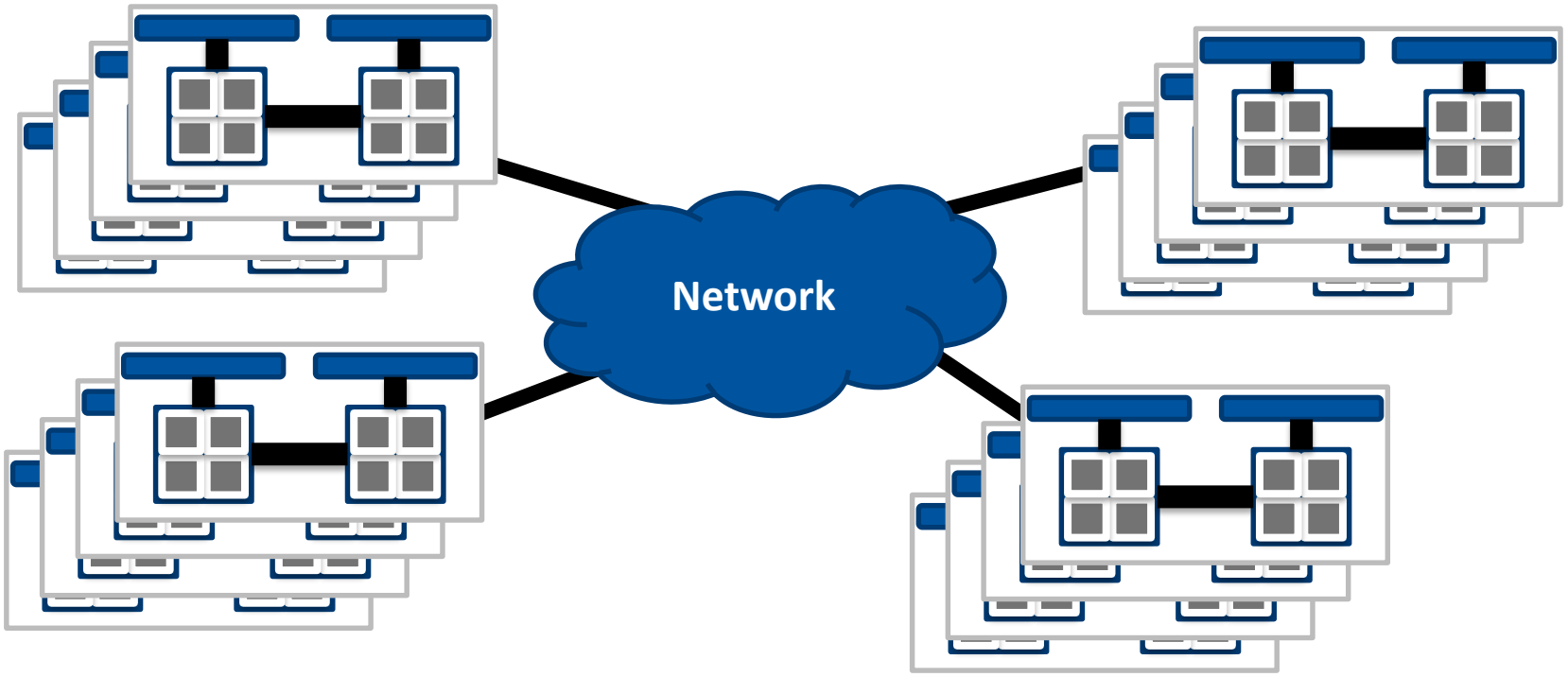
Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. **Distributed-memory programming with MPI**
   - The SPMD model revisited
   - MPI basics
     - Point-to-point communication
     - Non-blocking operations
     - Building and running MPI programs
   - Advanced MPI topics
     - Collective communication
     - Virtual topologies
     - Derived datatypes
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Motivation

- **Clusters**
  - → HPC market is dominated by distributed memory multicomputers (clusters)
  - → Many nodes with no direct access to other nodes' memory



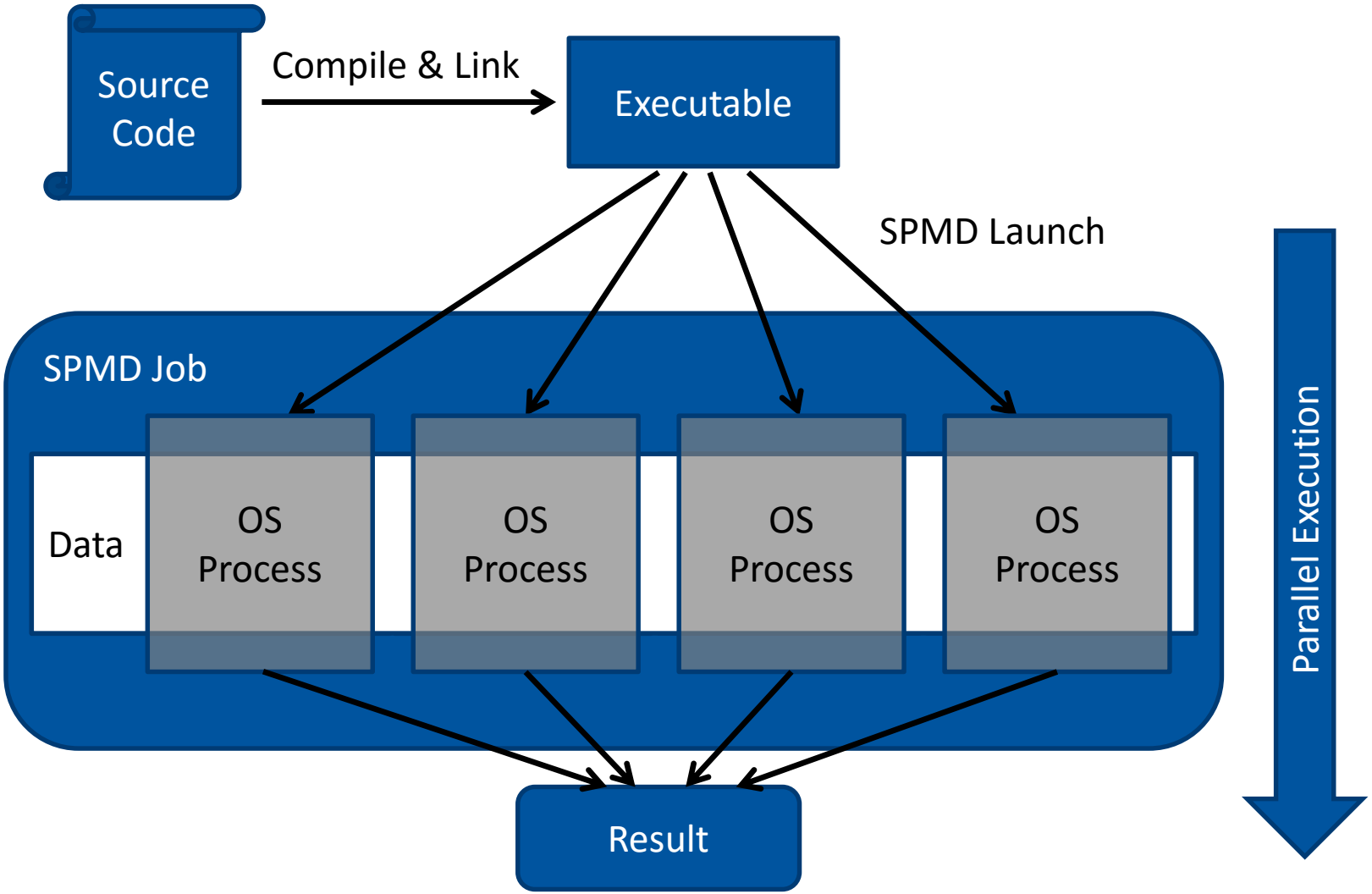
Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. **Distributed-memory programming with MPI**
   - **The SPMD model revisited**
   - MPI basics
     - Point-to-point communication
     - Non-blocking operations
     - Building and running MPI programs
   - Advanced MPI topics
     - Collective communication
     - Virtual topologies
     - Derived datatypes
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

SPMD

- **Single Program Multiple Data**
  - → Single source code (program) and in most cases single executable file
  - → Multiple processes working on different data
  - → Processes exchange data in the form of messages
- **The dominant parallel programming concept**
  - → Highly abstract ⇒ portable
  - → Highly scalable
  - → Widely supported
- **Drawbacks**
  - → Steep learning curve
  - → Harder to debug and profile than serial programs

SPMD – Program Lifecycle



SPMD – Identity

- **How to do useful work in parallel if source code is the same?**
  - → Each process receives a unique identifier
  - → Multiple code paths based on the ID

```
int my_id = get_my_id();

if (my_id == id_1) {
   // Code for process id_1
}
else if (my_id == id_2) {
   // Code for process id_2
}
else {
   // Code for other processes
}
```

SPMD – Identity

- **Identifiers:**
  - → global across the SPMD job
  - → unique
  - → do not change during the execution of the program
  - → well defined predictable set of values (e.g. [0 .. #processes-1])
  - → often used as communication addresses
- **Examples of bad identifiers:**
  - → OS PIDs – change with each execution, hard to predict, not unique with multiple OS instances (e.g., on clusters)
  - → TCP/IP port numbers – ditto
  - → GUIDs – hard to work with

SPMD – Data Exchange

- **Serial program**

```
a[0..9] = a[100..109];
```

- **SPMD program**

→ process 0: **aa** holds **a[0..9]**

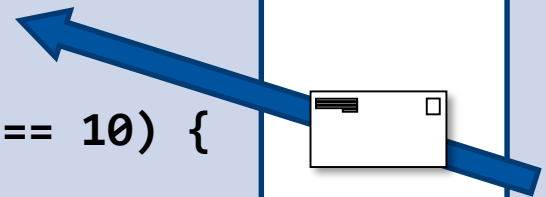
```
array aa[10];

if (my_id == 0) {
  recv(aa, 10);
}
else if (my_id == 10) {
  send(aa, 0);
}
```

process 10: **aa** holds **a[100..109]**

```
array aa[10];

if (my_id == 0) {
  recv(aa, 10);
}
else if (my_id == 10) {
  send(aa, 0);
}
```



SPMD – Data Exchange

- **Common data exchange operations**
  - → **`send(data, dst)`** – sends *data* to another process with ID of *dst*
  - → **`recv(data, src)`** – receives *data* from another process with ID of *src*
    - → wildcard sources usually possible, e.g., receive from any process

  - → **`bcast(data, root)`** – broadcasts *data* from *root* to all other processes
  - → **`scatter(data, subdata, root)`** – distributes *data* from *root* into *subdata* in all processes
  - → **`gather(subdata, data, root)`** – gathers *subdata* from all processes into *data* in process *root*
  - → **`reduce(data, res, op, root)`** – computes *op* over *data* from all processes and place the result in *res* in process *root*

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. **Distributed-memory programming with MPI**
   - The SPMD model revisited
   - **MPI basics**
     - Point-to-point communication
     - Non-blocking operations
     - Building and running MPI programs
   - Advanced MPI topics
     - Collective communication
     - Virtual topologies
     - Derived datatypes
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

History of MPI

- **Started in 1992 as a unification effort**
- **The MPI Forum – http://www.mpi-forum.org/**
- **MPI Standard versions**
  - → MPI-1 – p2p and collective communication, user-defined data types
    - →MPI-1.0 (Jun 1994) – first formal MPI definition
    - →MPI-1.3 (Jul 2008) – last in the MPI-1 series
  - → MPI-2 – one-sided comm, dynamic process management, parallel I/O
    - →MPI-2.0 (1997) – first formal extension to MPI-1.1 (includes MPI-1.2)
    - →MPI-2.1 (Sep 2008) – merger of MPI-1 and MPI-2 in one single standard
    - →MPI-2.2 (Sep 2009) – minor corrections; C++ bindings deprecated
  - → MPI-3.0 (Sep 2012)
    - →non-blocking and neighbourhood collective communication
    - →modern Fortran bindings added; C++ bindings deleted
  - → MPI-3.1 (June 2015) – latest published version

MPI – What it is and what it is not?

- **MPI is:**
  - → Message-passing standard defined in language-independent terms (LIS)
  - → Implemented in the form of a library of language-specific functions (bindings) and a language-independent runtime environment
  - → Bindings for C and Fortran specified in the standard, other languages supported by non-standard implementations (e.g. **`Boost.MPI`** for C++)
  - → Source-level compatibility across all implementations
- **MPI is not:**
  - → Not a language extension – doesn't require changes to existing compilers
  - → Not suitable for general distributed systems (e.g., GRID)
    - →no true support for dynamic processing and no built-in fault tolerance

Conventions in MPI and its bindings

- **Identifiers naming**
  - → Language-Independent Specification – **`MPI_OPERATION_NAME`**
  - → Fortran – same as specification (F90+ – case insensitive but usually all caps)
  - → C – **`MPI_First_capital_then_all_small`**
  - → Note: manual pages on Unix follow the C naming convention
  - → Constants in all languages – **`MPI_ALL_CAPS`**

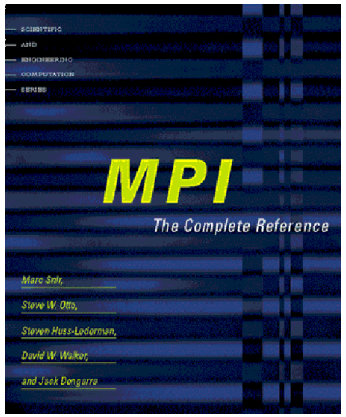
- **MPI library calling conventions**
  - → C – **`int MPI_Do_something(…)`** (MPI error code as return value)
  - → Fortran – **`SUBROUTINE MPI_DO_SOMETHING(…, ierr)`**
    - → Don't forget the last output argument where the error code is returned!

Accessing MPI

- **C**
  - → MPI interface provided in **`mpi.h`** – **`#include`** where necessary
- **FORTRAN 77**
  - → MPI constants provided in **`mpif.h`** – **`include`** in any function that uses MPI
  - → No arguments checking for MPI calls
    - → Omit **`ierr`** argument ⇒ code compiles ⇒ program crashes when run
  - → ***Use only in legacy FORTRAN projects!***
- **Fortran 90+**
  - → MPI interface provided by module **`mpi`** – **`use`** where necessary
  - → Arguments checking for most calls, better type safety
    - → Omit **`ierr`** argument ⇒ compile-time error in most cases

MPI: The Complete Reference

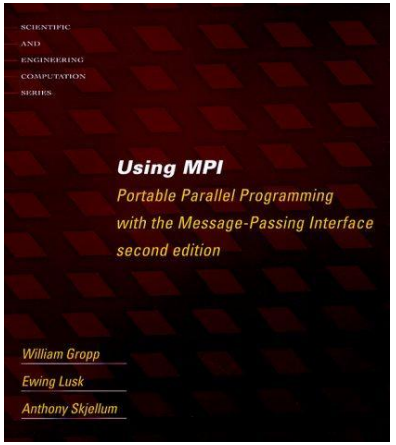
- **MPI: The Complete Reference Vol. 1**
  The MPI Core
  Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra.
  The MIT Press; 2nd edition; 1998



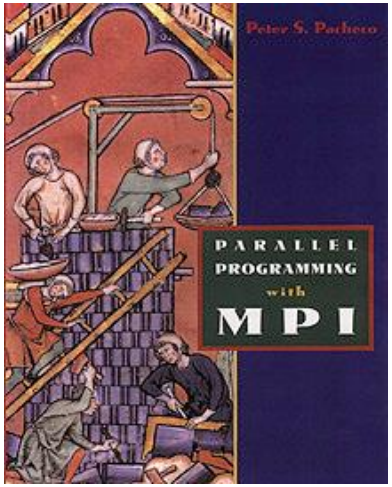
- **MPI: The Complete Reference Vol. 2**
  The MPI Extensions
  William Gropp, Steven Huss-Lederman, Andrew Lumsdain, Ewing Lusk, Bill Nitzberg, William Saphir, Marc Snir
  The MIT Press; 2nd edition; 1998

Using MPI

- **Using MPI**
  William Gropp, Ewing Lusk, Anthony Skjellum
  The MIT Press, Cambridge/London; 1999
- **Using MPI-2**
  William Gropp, Ewing Lusk, Rajeev Thakur
  The MIT Press, Cambridge/London; 2000



- **Parallel Programming with MPI**
  Peter Pacheco
  Morgan Kaufmann Publishers; 1996



MPI reference sources

- **The MPI Forum**
  - → http://www.mpi-forum.org/
  - → All published MPI standards available for download
- **RWTH Compute Cluster environment**
  - → Open MPI man pages
  - → Intel MPI man pages (after loading the appropriate module)
- **Internet searches**
  - → Documentation from any implementation should be fine in general
  - → Watch out for implementation-specific behaviour (where noted)

Hello, MPI!

- C

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, nprocs;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  printf("Hello, MPI! I am %d of %d\n",
         rank, nprocs);

  MPI_Finalize();
  return 0;
}
```

1. Header file inclusion – makes available prototypes of all MPI functions
2. MPI library initialisation – must be called before other MPI operations are called
3. MPI operations – more on that later
4. Text output – MPI programs also can print to the standard output
5. MPI library clean-up – no other MPI calls after this one allowed

Hello, MPI!

Fortran

```
program hello
  use mpi    !include ‘mpif.h’
  integer :: rank, nprocs, ierr

  call MPI_INIT(ierr)

  call MPI_COMM_RANK(MPI_COMM_WORLD, &
                     rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, &
                     nprocs, ierr)

  print ‘(“Process “,I0,” of “,I0)’, &
        rank, nprocs

  call MPI_FINALIZE(ierr)
end program hello
```

1. Module reference – makes available interfaces of all MPI functions
2. MPI library initialisation – must be called before other MPI operations are called
3. MPI operations – more on that later
4. Text output – MPI programs also can print to the standard output
5. MPI library clean-up – no other MPI calls after this one allowed

Initialisation and Clean-up

- **MPI library must be initialised first**

```
C:       int MPI_Init (int *argc, char ***argv)
Fortran: SUBROUTINE MPI_INIT (ierr)
```

→ No other MPI operations allowed before initialisation with few exceptions

→ (C) MPI-1: **`argc`** and **`argv`** must point to the arguments of **`main()`**

→ (C) MPI-2: both arguments can be **`NULL`** (to facilitate writing libraries)

→ MPI can only be initialised once for the lifetime of the program

- **Test if MPI was already initialised**

```
C:       int MPI_Initialized (int *flag)
Fortran: SUBROUTINE MPI_INITIALIZED (flag, ierr)
```

Initialisation and Clean-up

- **MPI library must be finalised before program completion**

```
C:       int MPI_Finalize (void)
Fortran: SUBROUTINE MPI_FINALIZE (ierr)
```

  - → No other MPI operations allowed after finalisation with few exceptions
  - → MPI can only be finalised once for the lifetime of the program
  - → Exiting without calling **`MPI_FINALIZE`** results in undefined behaviour

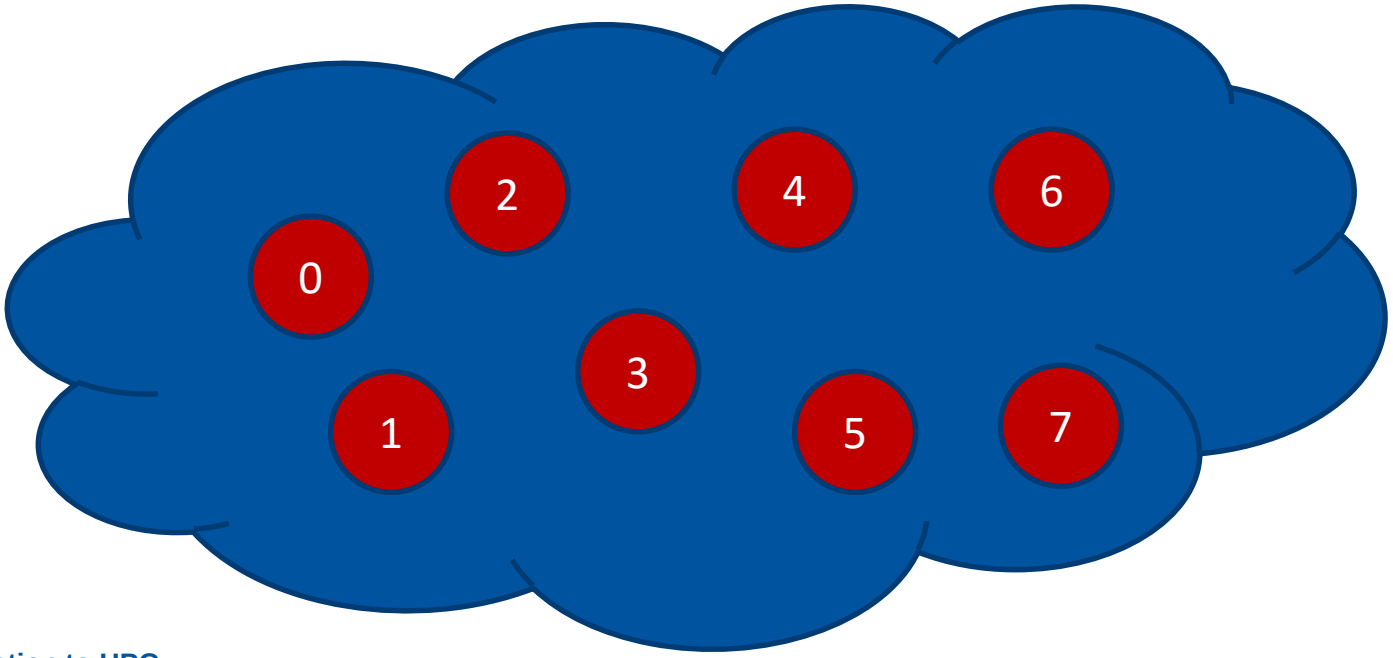
- **Test if MPI was already finalised**

```
C:       int MPI_Finalized (int *flag)
Fortran: SUBROUTINE MPI_FINALIZED (flag, ierr)
```

  - → Can be called at any time, even after **`MPI_FINALIZE`** (e.g. by **`atexit(3)`**)

Communicators

- **Each MPI communication happens within a context called communicator**
  - → Logical communication domains
  - → A group of participating processes + context
  - → Each MPI process has a unique numeric ID in the group – rank



Communicators

- **MPI communicators are referred by opaque handles**
  - → (C) communicator handles are of type **`MPI_Comm`**
  - → (Fortran) all handles are of type **`INTEGER`**
  - → Local values – don't pass around
- **Two predefined MPI communicators**
  - → **`MPI_COMM_WORLD`**
    - → Created by **`MPI_INIT`**
    - → Contains all processes launched initially as part of the MPI program
  - → **`MPI_COMM_SELF`**
    - → Contains only the current process
    - → Useful for talking to oneself

Query operations on communicators

- **How many processes are there in a given communicator?**

```
MPI_Comm_size (MPI_Comm comm, int *size)
```

  - → Returns the total number of MPI processes when called on **`MPI_COMM_WORLD`**
  - → Returns 1 when called on **`MPI_COMM_SELF`**

- **What is the rank of the calling process in a given communicator?**

```
MPI_Comm_rank (MPI_Comm comm, int *rank)
```

  - → Returned rank will differ in each calling process given the same communicator
  - → Ranks values are in **`[0, size-1]`** (always **`0`** for **`MPI_COMM_SELF`**)

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. **Distributed-memory programming with MPI**
   - The SPMD model revisited
   - MPI basics
     - **Point-to-point communication**
     - Non-blocking operations
     - Building and running MPI programs
   - Advanced MPI topics
     - Collective communication
     - Virtual topologies
     - Derived datatypes
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Messages

- **MPI passes data around in the form of messages**
- **Two components**
  - → Message content (user data)
  - → Envelope

Field	Meaning
Sender rank	Who sent the message
Receiver rank	To whom the message is addressed to
Tag	Additional message identifier
Communicator	Communication context

- **MPI retains the logical order in which messages between any two ranks are sent (FIFO)**
  - → But the receiver have the option to peek further down the queue

Sending messages

- **Messages are sent using the MPI_SEND family of operations**

```
MPI_Send (buf, count, datatype, dest, tag, comm)
```

message content | envelope

Parameter	Meaning
buf	Location of data in memory
count	Number of consecutive data elements to send
datatype	MPI data type handle
dest	Rank of the receiver
tag	Message tag
comm	Communicator handle

- **The MPI API is built on the idea that data structures are array-like**

  → No fancy C++ objects supported

MPI data types

- MPI has a powerful type system which tells it how to access memory content while constructing and deconstructing messages
- Complex data types can be created by combining simpler types

- Predefined MPI data type handles – C

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

MPI data types

- MPI has a powerful type system which tells it how to access memory content while constructing and deconstructing messages
- Complex data types can be created by combining simpler types

- Predefined MPI data type handles – Fortran

MPI data type handle	Fortran data type
MPI_INTEGER	INTEGER
MPI_REAL MPI_REAL8	REAL REAL(KIND=8)
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
…	…
MPI_BYTE	-

MPI data types

- **MPI is a library – it cannot infer the supplied buffer elements' type in runtime and hence has to be told what the type is**

- **MPI supports heterogeneous environments and implementations can convert between internal type representation on different architectures**

  → `MPI_BYTE` is used to send/receive data as-is without any conversion

- **MPI data type must match the language type of the data in the array**

- **Underlying data types must match in both communicating processes**

Receiving messages

- **Messages are received using the MPI_RECV operation**

```
MPI_Recv (buf, count, datatype, src, tag, comm, status)
```

message content | envelope

Parameter	Meaning
buf	Location in memory where to place data
count	Number of consecutive data elements that **buf** can hold
datatype	MPI data type handle
src	Rank of the sender
tag	Message tag
comm	Communicator handle
status	Status of the receive operation

Receiving messages

- **The next message with matching envelope is received**
  - → Wildcard specifiers possible
    - → **`MPI_ANY_SOURCE`** – matches messages from any rank
    - → **`MPI_ANY_TAG`** – matches messages with any tag
  - → Examine **`status`** to find out the actual values matched by the wildcard(s)

- **Status argument – C**
  - → Structure of type **`MPI_Status`** with the following fields
    - → **`status.MPI_SOURCE`** – source rank
    - → **`status.MPI_TAG`** – message tag
    - → **`status.MPI_ERROR`** – error code of the receive operation

Receiving messages

- **The next message with matching envelope is received**
  - → Wildcard specifiers possible
    - → **`MPI_ANY_SOURCE`** – matches messages from any rank
    - → **`MPI_ANY_TAG`** – matches messages with any tag
  - → Examine **`status`** to find out the actual values matched by the wildcard(s)

- **Status argument – Fortran**
  - → **`INTEGER`** array of size **`MPI_STATUS_SIZE`** with the following elements
    - → **`status(MPI_SOURCE)`** – source rank
    - → **`status(MPI_TAG)`** – message tag
    - → **`status(MPI_ERROR)`** – error code of the receive operation

Receiving messages

- **Inquiry about the number of elements received**

```
MPI_Get_count (status, datatype, count)
```

  - → **count** is set to number of elements of type **datatype** that can be formed by the content of the message or to **MPI_UNDEFINED** the number of elements is not integral
  - → **datatype** should match the **datatype** argument of **MPI_RECV**

- **If the receive status is of no interest – MPI_STATUS_IGNORE**

Receiving messages

- **buf/count must be large enough to hold the received message**
  - → OK

  - → Not OK – truncation error in **`MPI_RECV`**

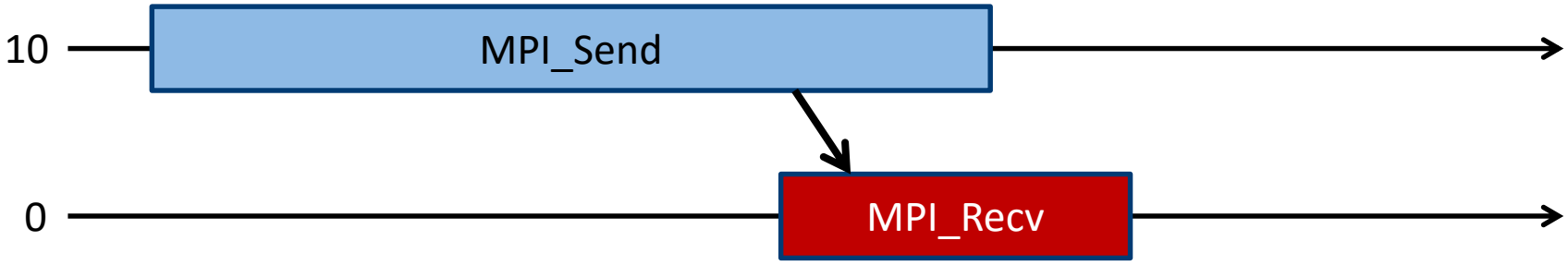
- **Probe for matching message before receiving it**
  - → `MPI_Probe(src, tag, comm, status)`

Example

- Back to our earlier SPMD example

```
int aa[10];
MPI_Status status;

if (rank == 0) {
  MPI_Recv(aa, 10, MPI_INT, 10, 0, MPI_COMM_WORLD, &status);
}
else if (rank == 10) {
  MPI_Send(aa, 10, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```



Operation completion

- **Two kinds of MPI calls**
  - → Blocking – do not return until the operation has completed
  - → Non-blocking – start the operation in background and return immediately
- **Operation completion**
  - → Sends complete once the message has been constructed and the user buffer is free for reuse
  - → Receives complete once a matching message has arrived and its content was placed in the user buffer
- **Non-blocking calls allow for computation/communication overlap**

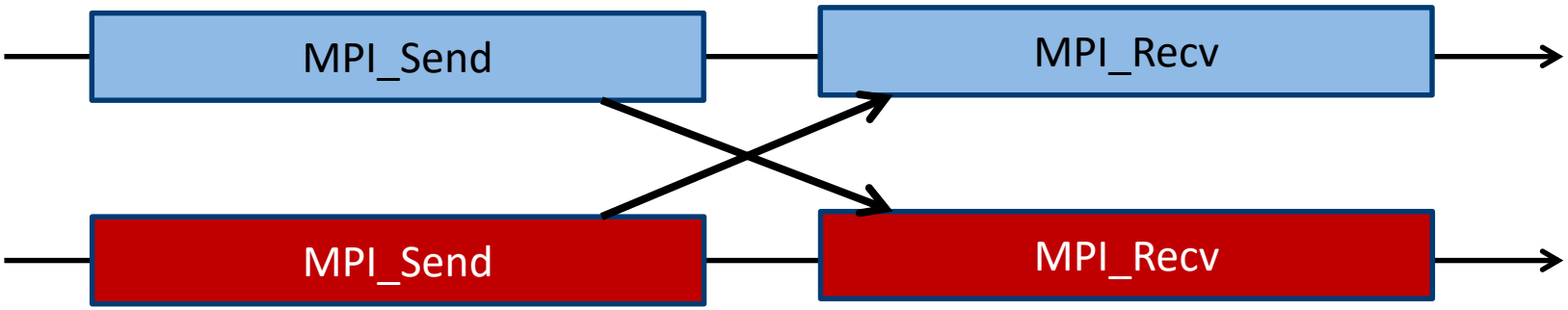
Different kinds of sends

- **Four different kinds of sends:**
  - → **`MPI_SSEND`** – synchronous send; completes once the corresponding receive was posted → guaranteed synchronizing effect
  - → **`MPI_BSEND`** – buffered send; completes once the message was stored away in a user-supplied buffer for later delivery
  - → **`MPI_SEND`** – standard send; completes once the message has been safely stored away (possibly transferred to its destination)
  - → **`MPI_RSEND`** – ready-mode send; completes only if the corresponding receive has already been posted when the send operation is initiated
- **The standard MPI_Send could be implemented either as synchronous or as buffered depending on the context – very implementation specific (!)**

Common problems – deadlock

- May occur when blocking operations are used

```
MPI_Send(to 1)          MPI_Send(to 0)
MPI_Recv(from 1)        MPI_Recv(from 0)
```



- Neither send operation will complete as both processes are waiting for the respective receive operation
- <u>May</u> succeed if standard send is implemented as buffered

Cure for deadlocks

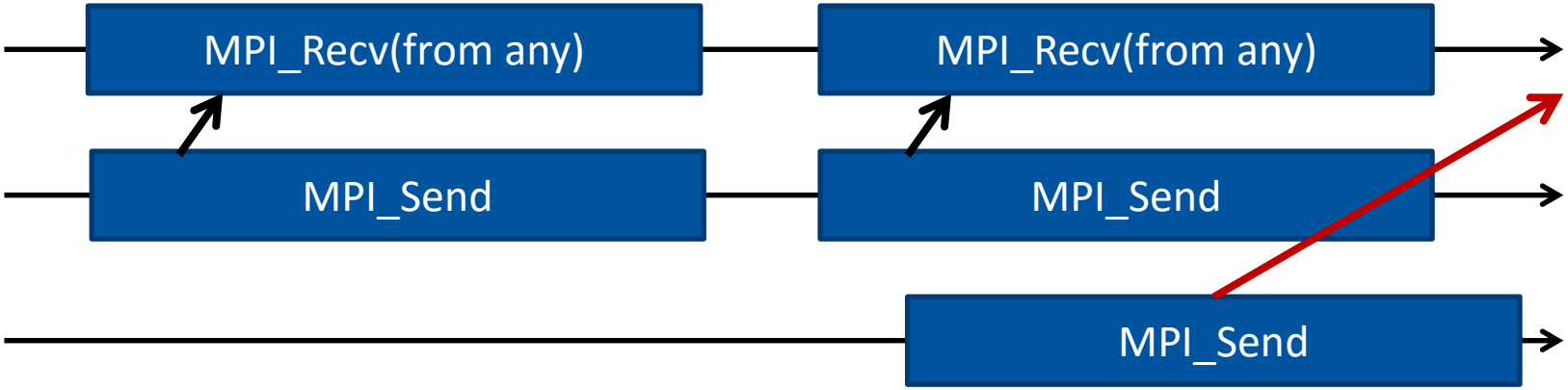
- **Simultaneous non-deadlocking send and receive operation**

```
MPI_Sendrecv (sdata, scount, sdtype, dest, stag,
              rdata, rcount, rdtype, src, rtag,
              comm, status)
```

  - → Combines the arguments of **MPI_SEND** and **MPI_RECV**
  - → Same communicator used for both operations
  - → Guaranteed to not deadlock
  - → A process can even talk to itself
- **Check the correctness of your algorithm**
  - → Simple test: Replacing all calls to **MPI_SEND** with **MPI_SSEND** should not lead to deadlocks in correct MPI applications

Common problems – race condition

- **MPI does not guarantee the order of reception of messages from different sources**

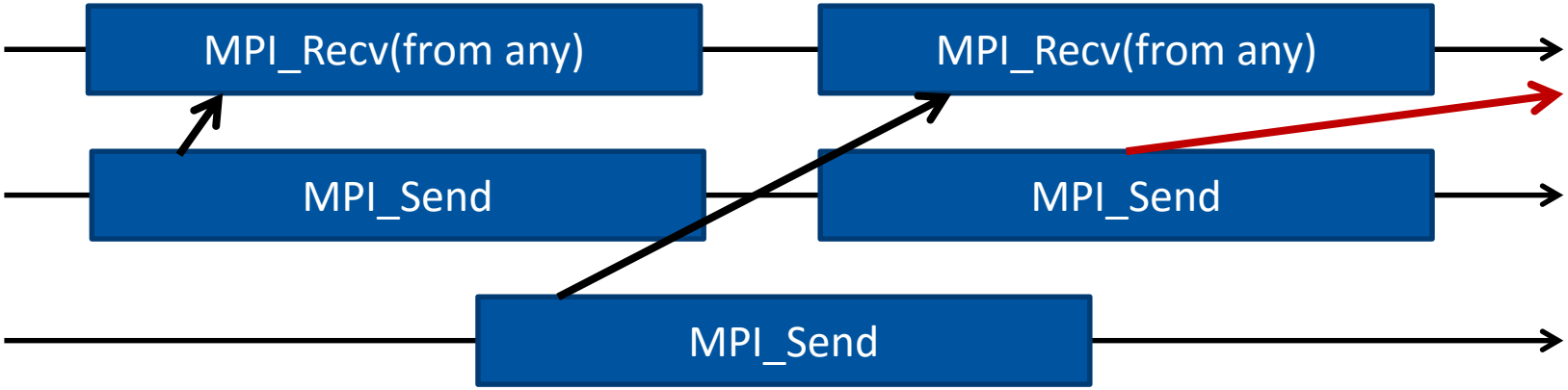


→ Occurs when wildcard source ranks are used in receive operations

→ In this particular example, we would like to receive two consecutive messages from the same rank. Having a second wildcard receive results in a race condition.

Common problems – race condition

- **MPI does not guarantee the order of reception of messages from different sources**



→ Occurs when wildcard source ranks are used in receive operations

→ Once you receive a message with **`MPI_ANY_SOURCE`**, use the source indicator in **`status`** in order to receive further messages from the same rank

Outline

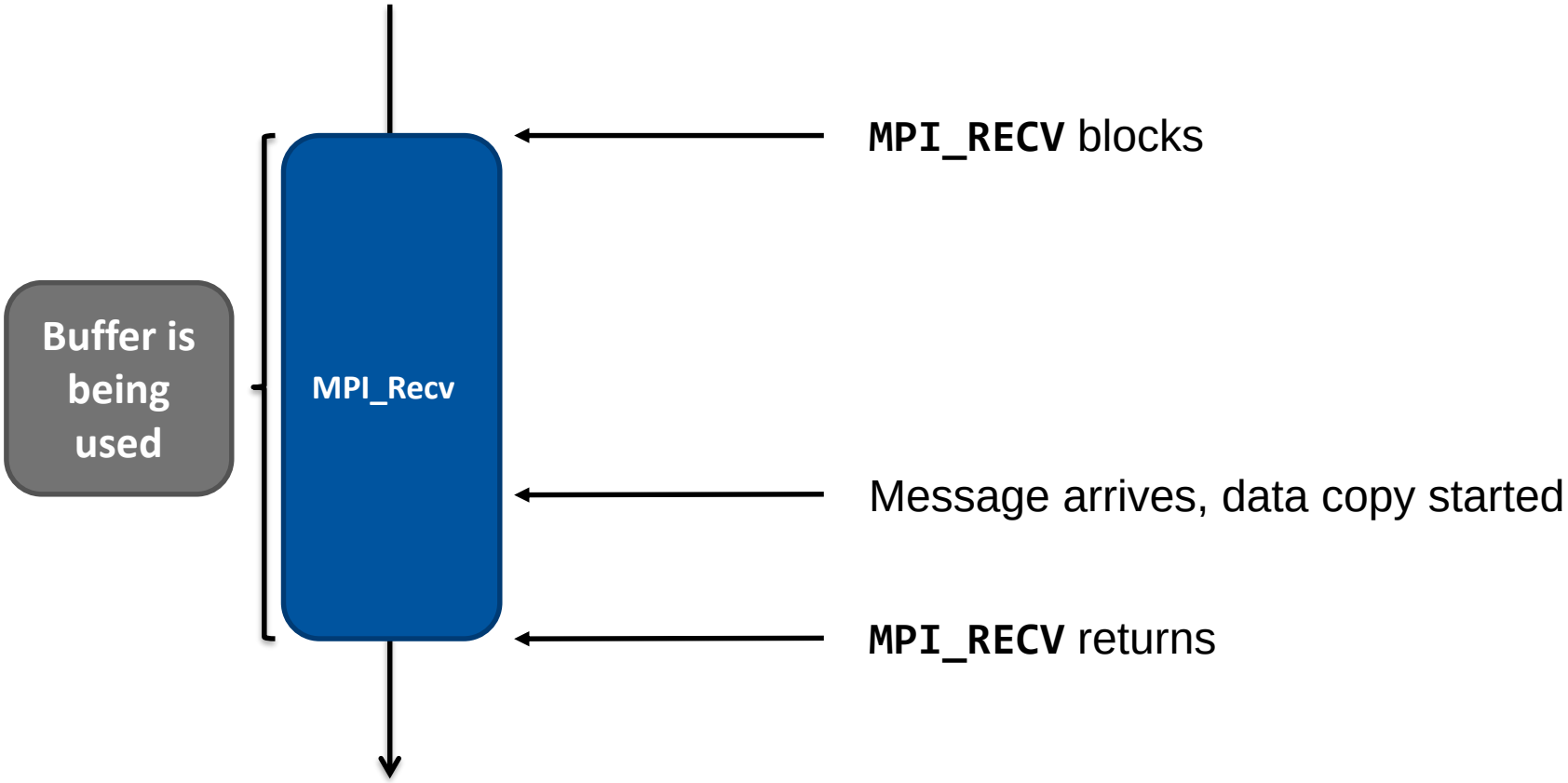
1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. **Distributed-memory programming with MPI**
   - **The SPMD model revisited**
   - MPI basics
     - Point-to-point communication
     - **Non-blocking operations**
     - Building and running MPI programs
   - Advanced MPI topics
     - Collective communication
     - Virtual topologies
     - Derived datatypes
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Non-blocking operations

- **Some blocking MPI operations may take very long time to complete**
  - → Serial overhead
  - → Bad parallel scaling (Amdahl's law)
- **MPI allows operations to run in the background**
  - → Initiate the operation
  - → Do something else
  - → Test the operation periodically for completion, OR
  - → Perform blocking wait for the operation to complete
- **Overlapping communication and computation can greatly improve parallel scaling and also prevents deadlocks**

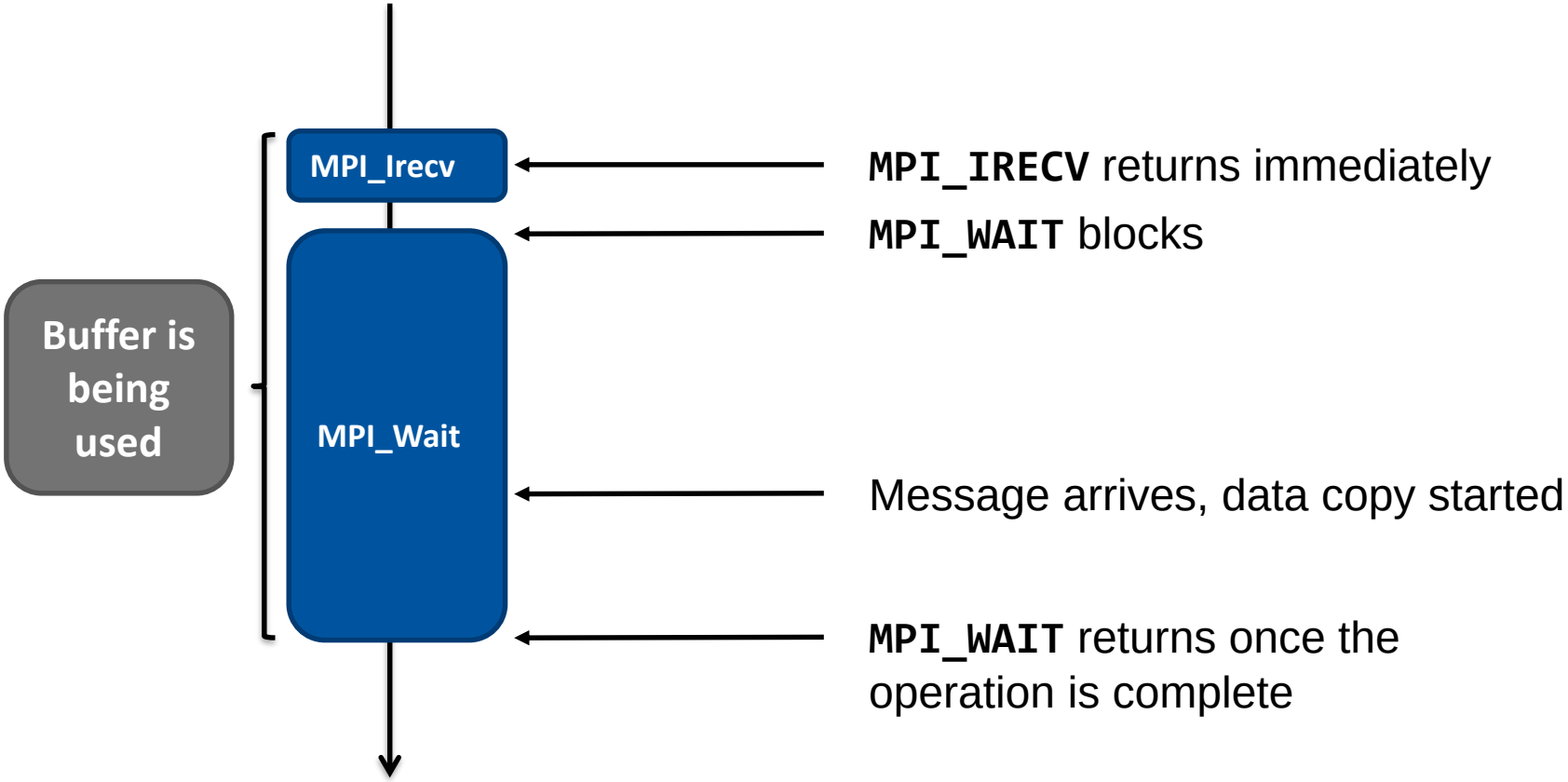
Non-blocking vs. blocking operations

- **Blocking operations do not return control until the operation has completed**



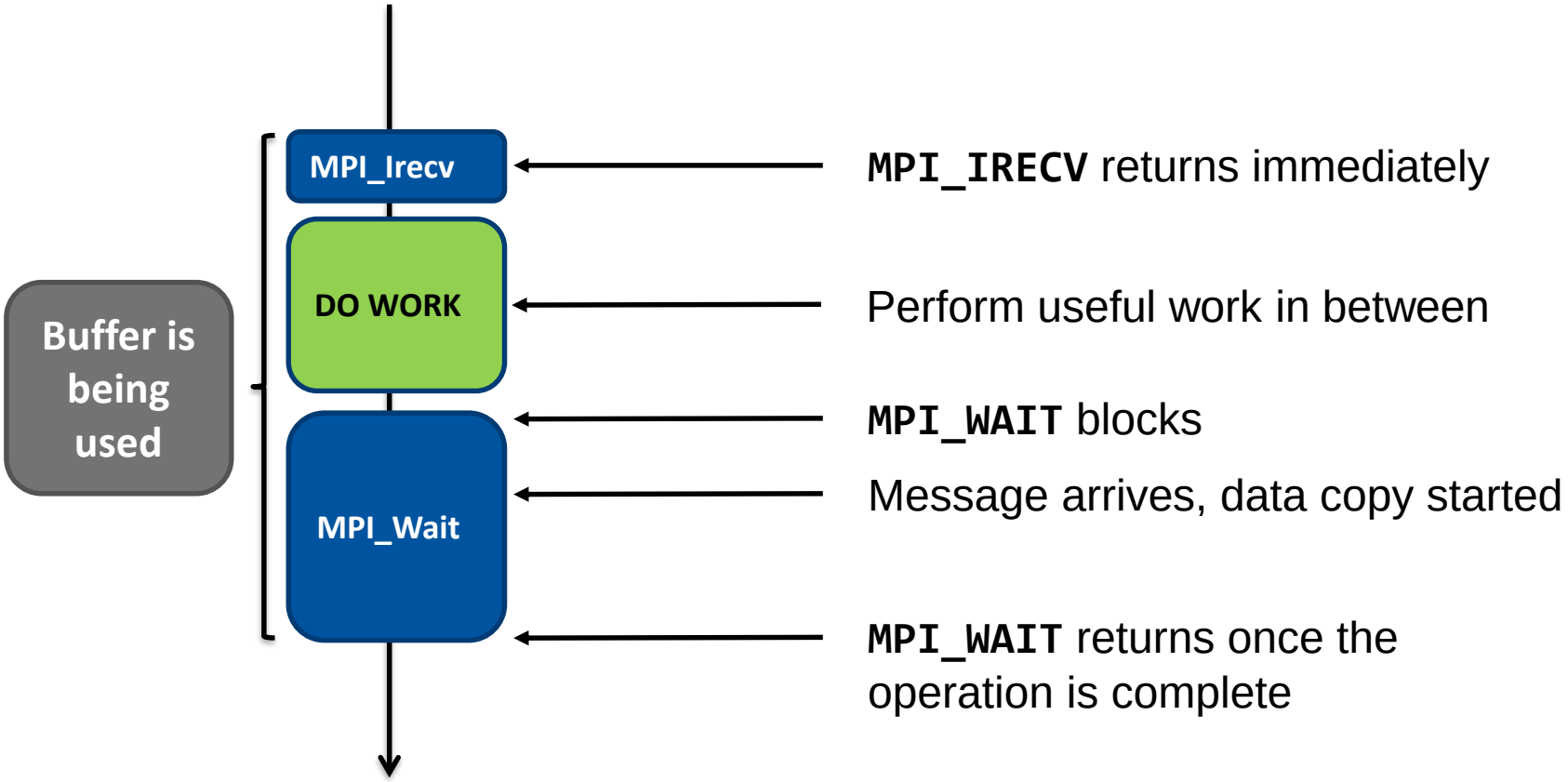
Non-blocking vs. blocking operations

- **Non-blocking operations return immediately and operation continues in the background; it must be waited or tested later on**



Non-blocking vs. blocking operations

- **Other useful work (computation) can be performed in between**



Initiating non-blocking operations

- **Non-blocking sends**

```
MPI_Isend (buf, count, datatype, dest, tag, comm, req)
```

→ **`req`** is set to a request handle that is then used to manipulate the request

→ All four MPI sends have non-blocking versions, e.g., **`MPI_IBSEND`**

- **Non-blocking receive**

```
MPI_Irecv (buf, count, datatype, src, tag, comm, req)
```

→ **`req`** replaces **`status`** – the receive status becomes known only after the operation is completed later on

Waiting for completion

- **Wait for a non-blocking operation to complete**

```
MPI_Wait (req, status)
```

→ Blocks until the operation identified by **`req`** completes

→ **`status`** receives the status of the operation

→only cancelation information is provided for send operations

→status can be **`MPI_STATUS_IGNORE`**

→ The request handle is deallocated at the end and set to **`MPI_REQUST_NULL`**

Testing for completion

- **Test without blocking if an operation has completed**

```
MPI_Test (req, flag, status)
```

  - → Tests the completion of the request **`req`** without blocking
  - → **`flag`** is set to the completion status (nonzero/**`.TRUE.`** or **`0`**/**`.FALSE.`**)
  - → **`status`** receives the status of the operation if it has completed
  - → The request handle is deallocated if the operation has completed

- **Note: The MPI standard allows for the implementations to postpone the actual operation until either `MPI_WAIT` or `MPI_TEST` was called**
  - → Often the case

Waiting/testing on multiple requests

- **MPI_WAITALL**
  - → waits for all specified requests to complete
- **MPI_WAITANY**
  - → waits for any specified request to complete
- **MPI_WAITSOME**
  - → waits for one or more of the specified requests to complete
- **MPI_TESTALL**
  - → tests for the completion of all specified requests
- **MPI_TESTANY**
  - → tests for the completion of any specified request
- **MPI_TESTSOME**
  - → tests for the completion of one or more of the specified requests

Timing functions

- **Measure local wall-clock time (with good precision)**

```
C:       double MPI_Wtime (void)
Fortran: DOUBLE PRECISION FUNCTION MPI_WTIME()
```

→ Returns the fractional number of seconds since some moment in the past

→ Usage hint: call it twice and subtract both results to obtain the wall-clock time that has elapsed between the two calls

- **Obtain the precision of the MPI timer**

```
C:       double MPI_Wtick (void)
Fortran: DOUBLE PRECISION FUNCTION MPI_WTICK()
```

→ Returns the shortest measurable time slice

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. **Distributed-memory programming with MPI**
   - The SPMD model revisited
   - MPI basics
     - Point-to-point communication
     - Non-blocking operations
     - **Building and running MPI programs**
   - Advanced MPI topics
     - Collective communication
     - Virtual topologies
     - Derived datatypes
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Develop with MPI – Compiler Wrappers

- **Provided by most MPI implementations**
- **Pass automatically to the real compiler all the arguments needed in order to find and link the MPI library**
- **Usually the compiler name prefixed with `mpi`**
  - → **`mpicc`** – C compiler + MPI options
  - → **`mpiCC`** | **`mpic++`** | **`mpicxx`** – C++ compiler + MPI options
  - → **`mpif77`** | **`mpif90`** | **`mpifort`** – Fortran compiler + MPI options
- **Accept all the options that the wrapped compiler accepts**
- **Should be used to link the final executable as they provide the correct linker options**

```
cluster$ mpicc -O2 -o program.exe program.c
```

Develop with MPI @ RWTH Compute Cluster

- **Two MPI implementations**
  - → Open MPI (default)
  - → Intel MPI

```
cluster$ module switch openmpi intelmpi/x.y
```

- **Universal environment variables**
  - → **`$MPICC`** – C compiler wrapper
  - → **`$MPICXX`** – C++ compiler wrapper
  - → **`$MPIF77`** – Fortran 77 compiler wrapper
  - → **`$MPIFC`** – Fortran 90+ compiler wrapper
  - → **`$MPIEXEC`** – MPI launcher
  - → **`$FLAGS_MPI_BATCH`** – Recommended launcher flags in batch mode

Hello, world!

- **1. Type the code on slide 755/756 into a text file named hello.c/.f90**
- **2. Compile:**
  - → C: `mpicc -o hello.exe hello.c`
  - → Fortran: `mpif90 -o hello.exe hello.f90`
- **3. Run:**
  - → `mpiexec -n 4 hello.exe`
- **You may also use the RWTH specific environment variables:**
  - → `$MPICC -o hello.exe hello.c`
  - → `$MPIEXEC -n 4 hello.exe`

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. **Distributed-memory programming with MPI**
   - The SPMD model revisited
   - MPI basics
     - Point-to-point communication
     - Non-blocking operations
     - Building and running MPI programs
   - Advanced MPI topics
     - **Collective communication**
     - Virtual topologies
     - Derived datatypes
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

SPMD – Data Exchange

- **Common data exchange operations**
  - → **`send(data, dst)`** – sends *data* to another process with ID of *dst*
  - → **`recv(data, src)`** – receives *data* from another process with ID of *src*
    - → wildcard sources usually possible, i.e. receive from any process

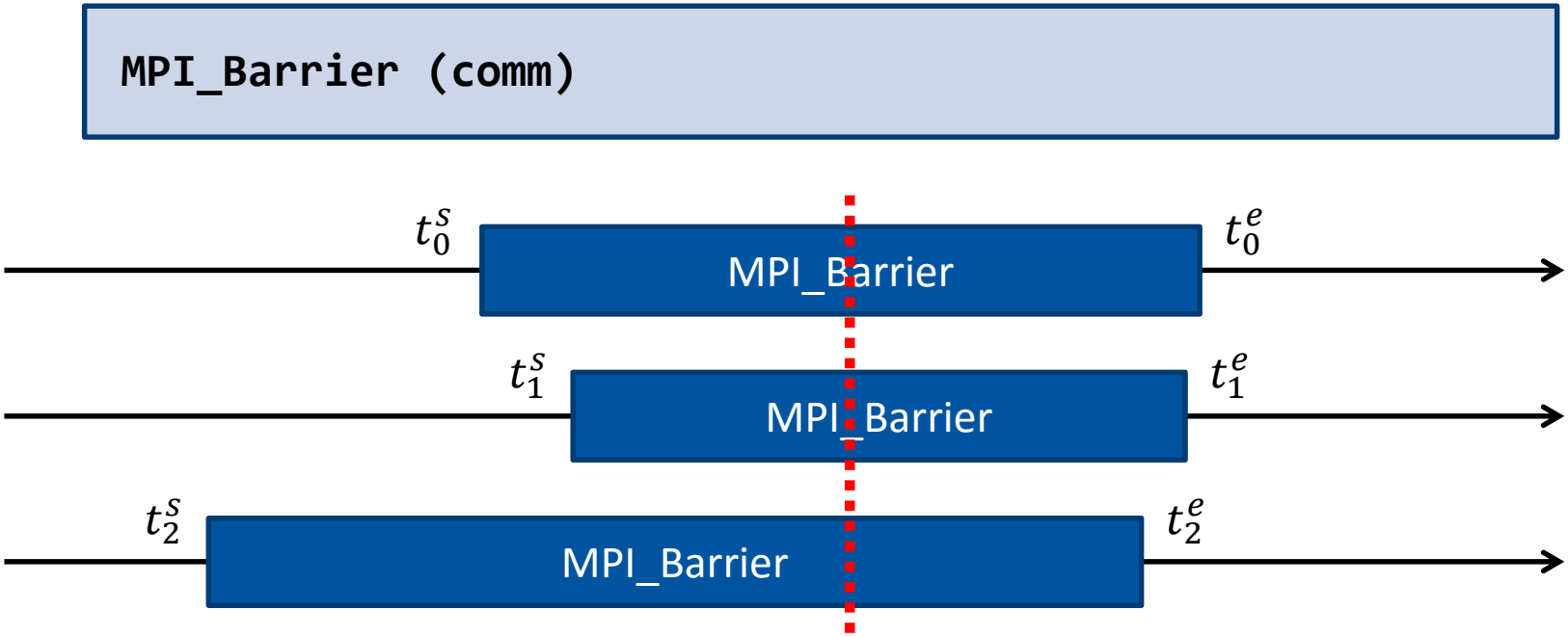
  - → **`bcast(data, root)`** – broadcasts *data* from *root* to all other processes
  - → **`scatter(data, subdata, root)`** – distributes *data* from *root* into *subdata* in all processes
  - → **`gather(subdata, data, root)`** – gathers *subdata* from all processes into *data* in process *root*
  - → **`reduce(data, res, op, root)`** – computes *op* over *data* from all processes and place the result in *res* in process *root*

Collective communication operations

- An MPI collective communication operation is one which involves all processes in a given communicator
- All processes must call the same MPI function and provide the same value for the “root” process rank (if required by the operation)
- Collective operations can be globally synchronous
- The scope of a collective operation is a single communicator
- Any collective operation can be replaced with a set of point-to-point communication operations (although generally not recommended)

Barrier synchronisation

- **Block until all processes in `comm` have entered the barrier**



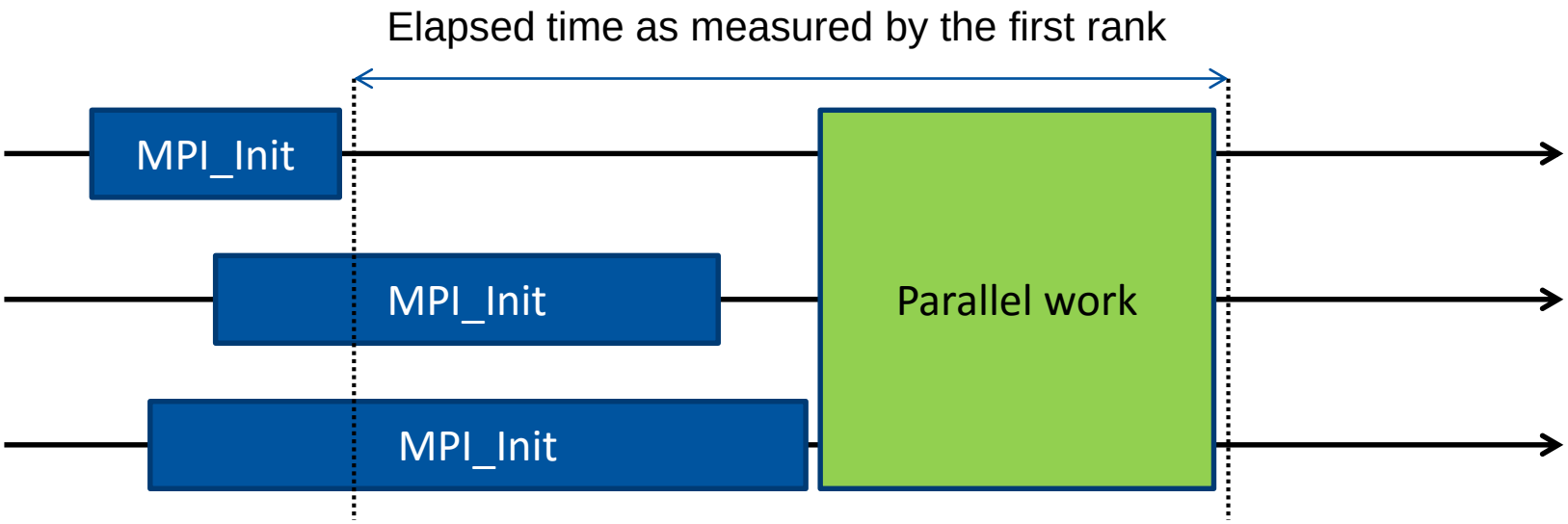
→ **`MPI_BARRIER`** guarantees that $\min_i t_i^e \geq \max_i t_i^s$

→ Processes are allowed to exit the barrier at different times, but not before all other processes have entered it

→ The only collective that is ***guaranteed synchronous***

Barrier synchronisation

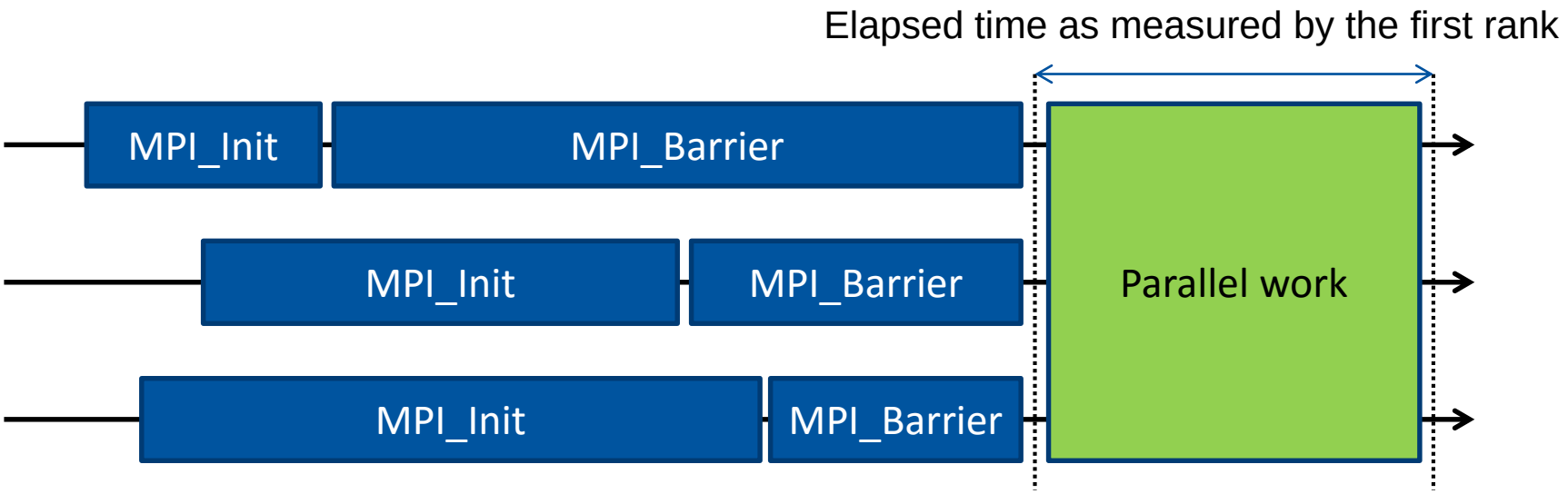
- **Useful when benchmarking**
  - → Always synchronise before taking time measurements



  - → Huge discrepancy between the actual work time and the measurement

Barrier synchronisation

- **Useful when benchmarking**
  - → Always synchronise before taking time measurements



  - → Dispersion of the barrier exit times is usually quite low

Data replication – broadcast

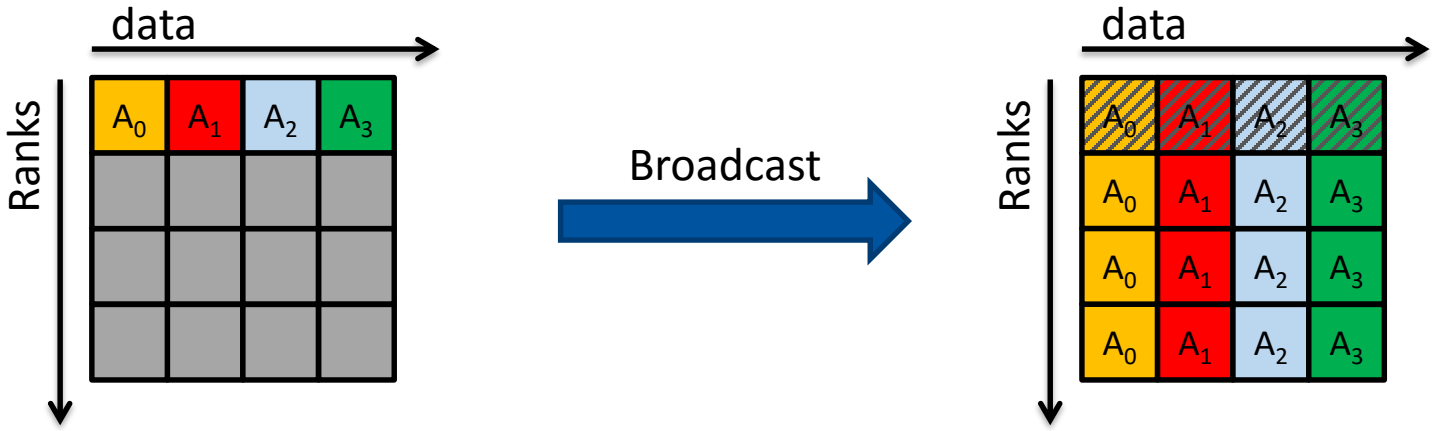
- **Replicate data from root process to all other processes in comm**

```
MPI_Bcast (data, count, datatype, root, comm)
```

→ **`data`** points to the data source in process with rank **`root`**

→ **`data`** points to the destination buffer in all other processes

→ the amount of data sent must be equal to the size of the receive buffer



Data replication – broadcast

- **Replicate data from process root to all other processes in comm**

```
MPI_Bcast (data, count, datatype, root, comm)
```

→ **data** points to the data source in process with rank **root**

→ **data** points to the destination buffer in all other processes

→ example use:

```
int ival;

if (rank == 0)
   ival = read_from_somewhere();

MPI_Bcast(&ival, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Same value provided by all processes!

Data replication – broadcast

- Straightforward implementation of MPI_BCAST

```
void Bcast(void *data, int count, MPI_Type datatype,
           int root, MPI_Comm comm)
{
  int i, rank, nprocs;
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &nprocs);
  if (rank == root) {
    for (i = 0; i < nprocs; i++)
      if (i != root)
        MPI_Send(data, count, datatype, i, 999, comm);
  }
  else
    MPI_Recv(data, count, datatype, root, 999, comm,
             MPI_STATUS_IGNORE);
}
```

Data distribution – scatter

- Scatter chunks of data from `root` to all processes in `comm`

```
MPI_Scatter (sbuf, scount, sdtype, rbuf, rcount, rdtype,
             root, comm)
```

Argument	Meaning
sbuf	Data source
scount	Number of elements in each chunk
sdtype	Source data type handle
rbuf	Receive buffer
rcount	Capacity of the receive buffer
rdtype	Receive data type handle
root	Rank of the data source
comm	Communicator handle

Significant only at **root**

Significant at all processes

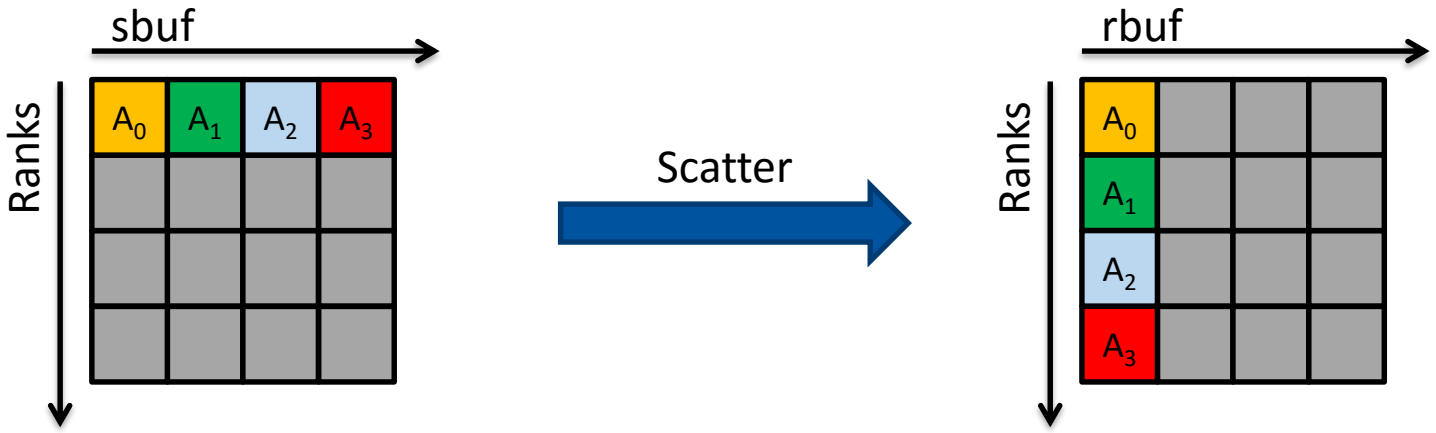
Data distribution – scatter

- **Scatter chunks of data from `root` to all processes in `comm`**

```
MPI_Scatter (sbuf, scount, sdtype, rbuf, rcount, rdtype,
             root, comm)
```

→ **`sbuf`** should be large enough to provide **`scount*nprocs`** elements

→ **`rbuf`** should be large enough to accommodate **`scount`** elements



Data collection – gather

- Gather chunks of data from all processes in `comm` to `root`

```
MPI_Gather (sbuf, scount, sdtype, rbuf, rcount, rdtype,
            root, comm)
```

Argument	Meaning
sbuf	Data source
scount	Number of elements to send
sdtype	Source data type handle
rbuf	Receive buffer
rcount	Number of elements in each chunk
rdtype	Receive data type handle
root	Rank of the data source
comm	Communicator handle

Significant at all processes

Significant only at **root**

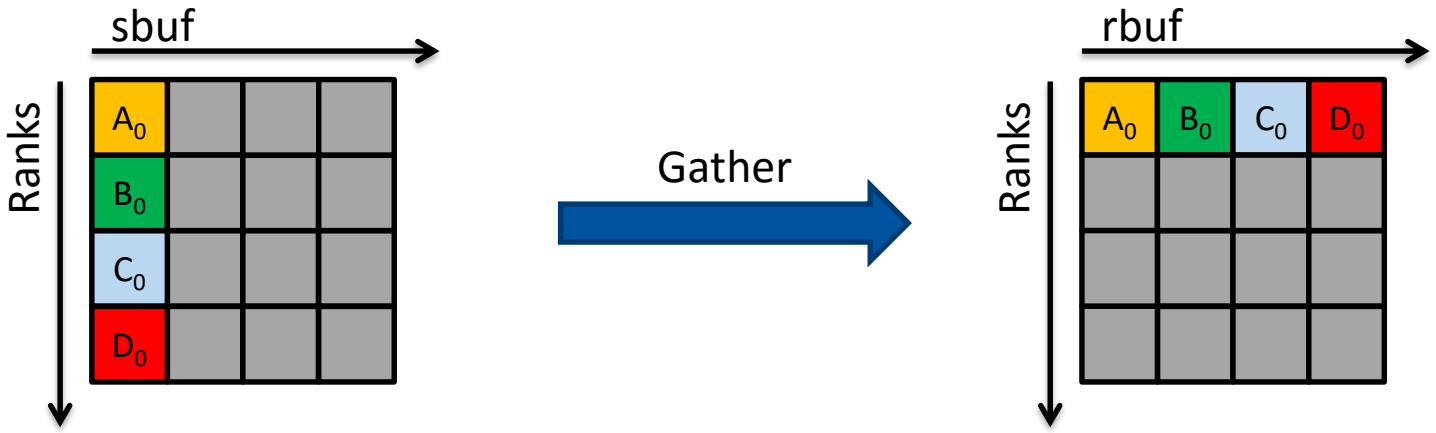
Data collection – gather

- **Gather chunks of data from all processes in `comm` to `root`**

```
MPI_Gather (sbuf, scount, sdtype, rbuf, rcount, rdtype,
            root, comm)
```

→ **`sbuf`** should be large enough to provide **`scount`** elements

→ **`rbuf`** should be large enough to accommodate **`rcount*nprocs`** elements



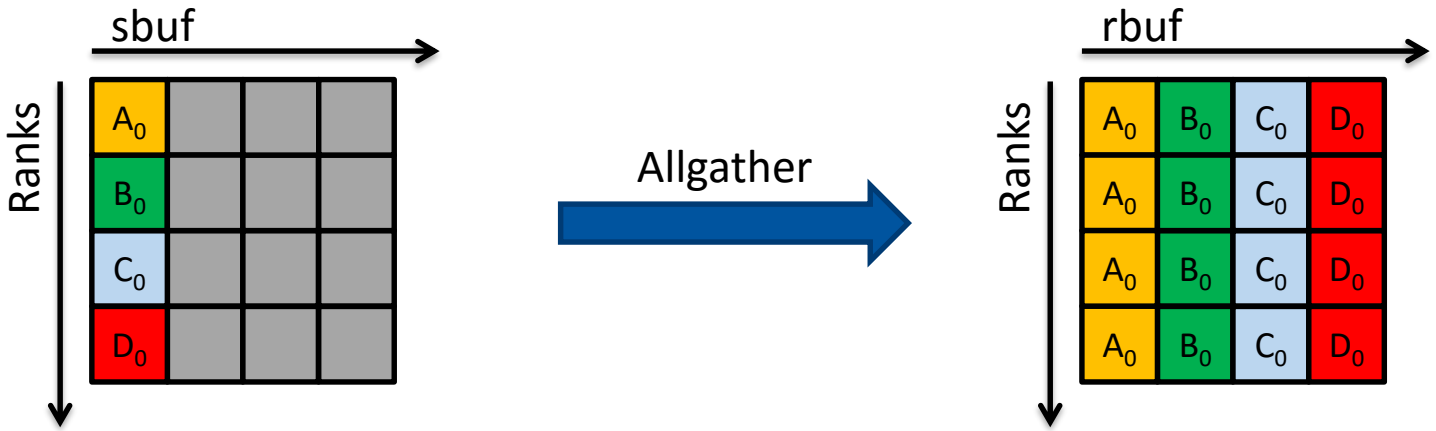
Data collection: "rootless" gather-to-all

- **Gather chunks of data from all processes in `comm` to all processes**

```
MPI_Allgather (sbuf, scount, sdtype,
               rbuf, rcount, rdtype, comm)
```

→ **`sbuf`** should be large enough to provide **`scount`** elements

→ **`rbuf`** should be large enough to accommodate **`rcount*nprocs`** elements



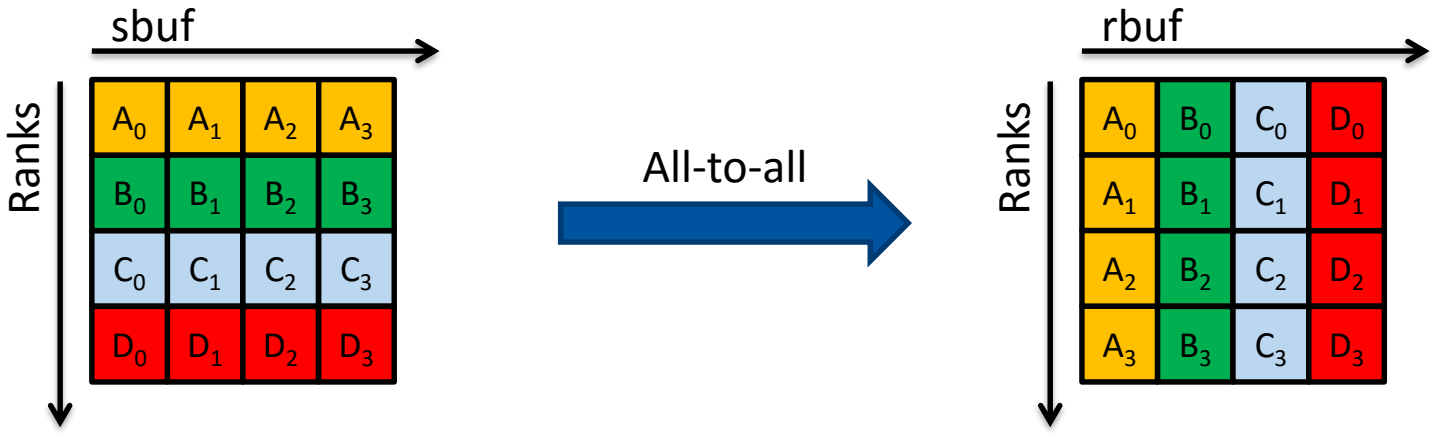
Data collection – all-to-all

- **Perform an all-to-all scatter/gather with all processes in `comm`**

```
MPI_Alltoall (sbuf, scount, sdtype,
              rbuf, rcount, rdtype, comm)
```

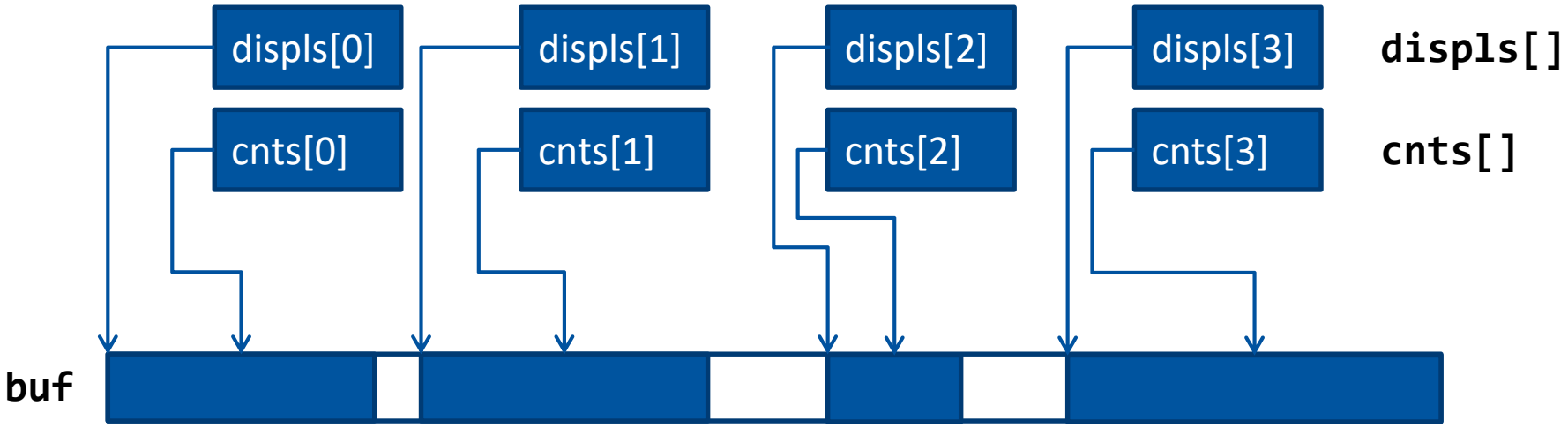
→ **`sbuf`** should be large enough to provide **`scount*nprocs`** elements

→ **`rbuf`** should be large enough to accommodate **`rcount*nprocs`** elements



Varying count versions

- **Position and length of each chunk can be explicitly specified with the so-called varying count (-V) versions**
  - → Displacement and count in datatype elements for each chunk specified



- **Useful when the problem size is not divisible by the number of MPI processes or when dealing with irregular domain decomposition**

Varying count scatter and gather

- **Scatter with varying count of elements**

```
MPI_Scatterv (sbuf, scnts, sdispls, sdtype,
              rbuf, rcount, rdtype, root, comm)
```

- **Gather with varying count of elements**

```
MPI_Gatherv (sbuf, scount, sdtype,
             rbuf, rcnts, rdispls, rdtype,
             root, comm)
```

- **The amount of data sent should be exactly equal to the size of the receive buffer – applies to all collective operations!**

  → Collective operations do not provide status information

Global reduction

- **Combine elements from all processes using a reduction operation**

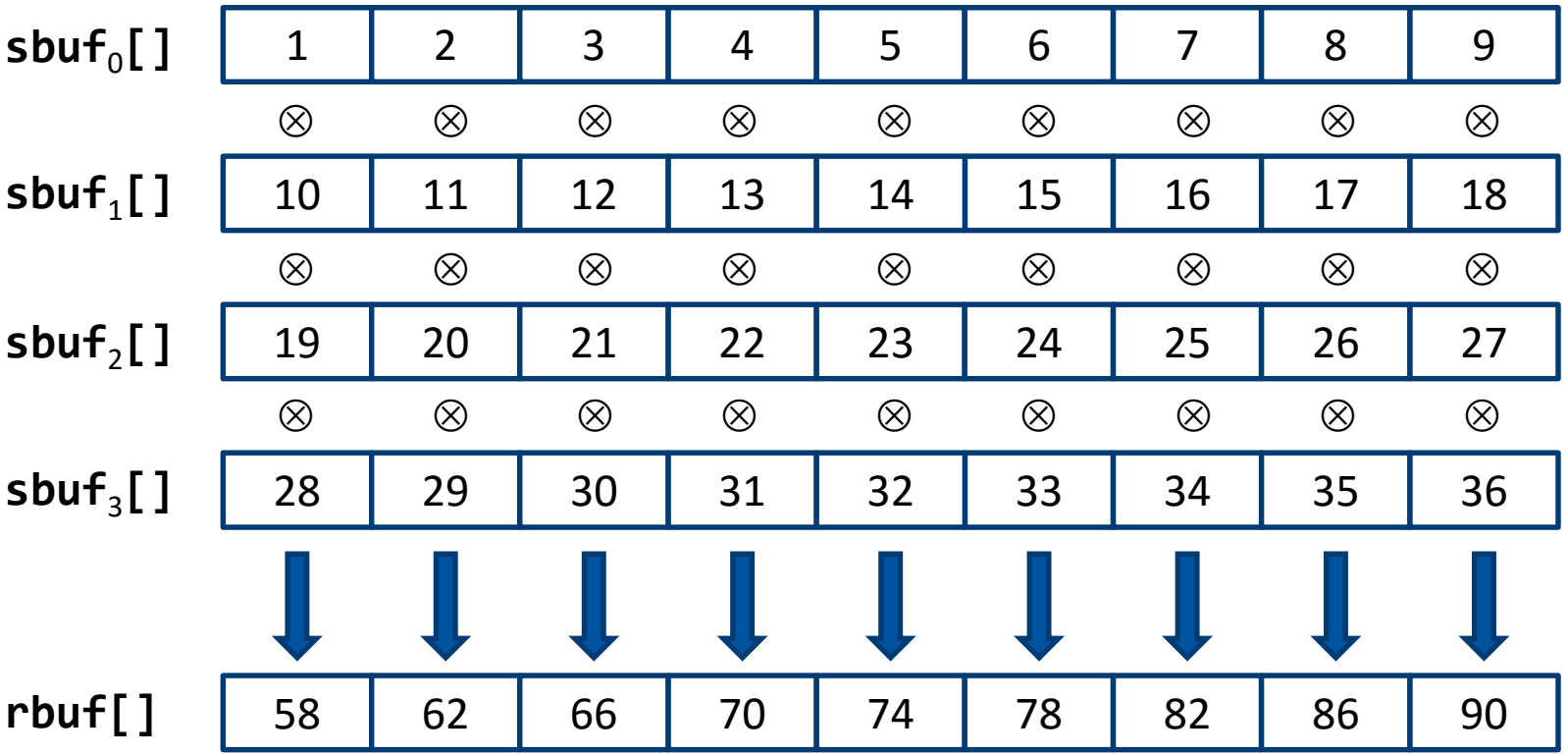
```
MPI_Reduce (sbuf, rbuf, count, dtype, op, root, comm)
```

Argument	Meaning
sbuf	Data source
rbuf	Receive buffer (significant only at **root**), different from **sbuf**
count	Number of elements in both buffers (same everywhere!)
dtype	Data type (same everywhere!)
op	Reduction operation
root	Rank of the data receiver
comm	Communicator handle

Global reduction

Global element-wise operation

→ **rbuf[i] = sbuf$_0$[i] op sbuf$_1$[i] op sbuf$_2$[i] op … sbuf$_{nranks-1}$[i]**



⊗ = MPI_SUM

Global reduction – predefined operations

- **MPI provides the following predefined reduce operations**

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND / MPI_BAND	logical / bit-wise AND
MPI_LOR / MPI_BOR	logical / bit-wise OR
MPI_LXOR / MPI_BXOR	logical / bit-wise XOR
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

→ All predefined operations are **associative** and **commutative**

→ Watch out for non-associativity of floating-point arithmetic!

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. **Distributed-memory programming with MPI**
   - The SPMD model revisited
   - MPI basics
     - Point-to-point communication
     - Non-blocking operations
     - Building and running MPI programs
   - Advanced MPI topics
     - Collective communication
     - **Virtual topologies**
     - Derived datatypes
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Communicators

- **Logical communication contexts**
  - → Process group
  - → Topology
- **Communicators are specified explicitly in each communication operation**
  - → Part of the message envelope, subject to matching
  - → Direct cross-communicator communication is not possible
    - → No wildcard communicator handles (i.e. nothing like **`MPI_COMM_ANY`**)
- **Communications provide logically distinct “namespaces”**
  - → Parallel libraries can create their own communicators in order not to interfere with user messaging

Intra- and intercommunicators

- **Intracommunicators**
  - → A single process group
  - → Every process can talk to any other process
  - → Example: **`MPI_COMM_WORLD`**
- **Intercommunicators**
  - → Two process groups – local and remote
  - → Every process can only talk to processes in the remote group
  - → Usually created by combining two communicators
  - → Example: one MPI job spawns another MPI job; both have their own **`MPI_COMM_WORLD`** and an intercommunicator is created so that both can communicate with each other

Virtual topologies

- **Each communicator can have an associated topology**
  - → Mapping between ranks and abstract addresses
  - → Connectivity (neighbouring links) information
- **Three different topology kinds:**
  - → no topology – e.g. **`MPI_COMM_WORLD`**
  - → graph topology – general connectivity graph
  - → Cartesian topology – regular *n*-dimensional grid of processes
- **Topologies allow for more intuitive mapping of process IDs to the domain decomposition scheme**

Problem Statement

- **Key to Scalability: Data Locality**
  - → Optimizations must take the trend to hierarchical architectures into account
    - →Network (see examples on next slides)
    - →Memory Hierarchy
    - →Processing Units
    - →Storage Hierarchy
- **Improving Data Locality: Virtual Topology**
  - → Express dependencies between software processing units
    - →MPI: processes exchanging messages
    - →OpenMP: multiple threads accessing the same memory location
  - → Virtual Topology = application communication pattern

Formalization (1 of 2)

- **Minimization problem**
  - → Optimization goal for the user: minimization of runtime
  - → Technical optimization goal: minimization of number of hops
    - → Different metrics: bandwidth (message size), latency (message volume), distance in hops (network properties), …
- **Model of the network: $H = (V_H, w_H)$**
  - → $V_H \in N$: vertices = execution units, i.e. nodes
  - → $w_H (u, v) \in N \times N$: edge weights, usually just 1 / 0 for 1 if present
- **Model of the (static) application: $A = (V_A, w_A)$**
  - → V_A: vertices = set of communicating processes
  - → w_A: edge weights = some metric for communication between processes

Formalization (2 of 2)

- **Topology Mapping: σ: $V_A \rightarrow V_H$**
  - → σ assigns each process instance a target node in the network
  - → Each mapping has an associated cost metric which constructs the optimization problem
  - → Two fundamental metric classes: dilation and congestion
- **Dilation: sum of the pairwise distances of neighbors in A mapped to H**
  - → $\sum_{u,v \in V_A} \boldsymbol{d_H}\big(\boldsymbol{\sigma}(\boldsymbol{u}), \boldsymbol{\sigma}(\boldsymbol{v})\big) \times \boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v})$
    - → d: shortest distance between vertices
- **Congestion: number of communication pairs per link**
  - → $\boldsymbol{C_e} = \sum_{u,v,\in V_A} \boldsymbol{p_e}(\boldsymbol{u}, \boldsymbol{v})$, $\boldsymbol{C_{max}} = \boldsymbol{max_e C_e}$
    - → p: probability that any routes from u to v cross the edge e

Algorithmic Solutions

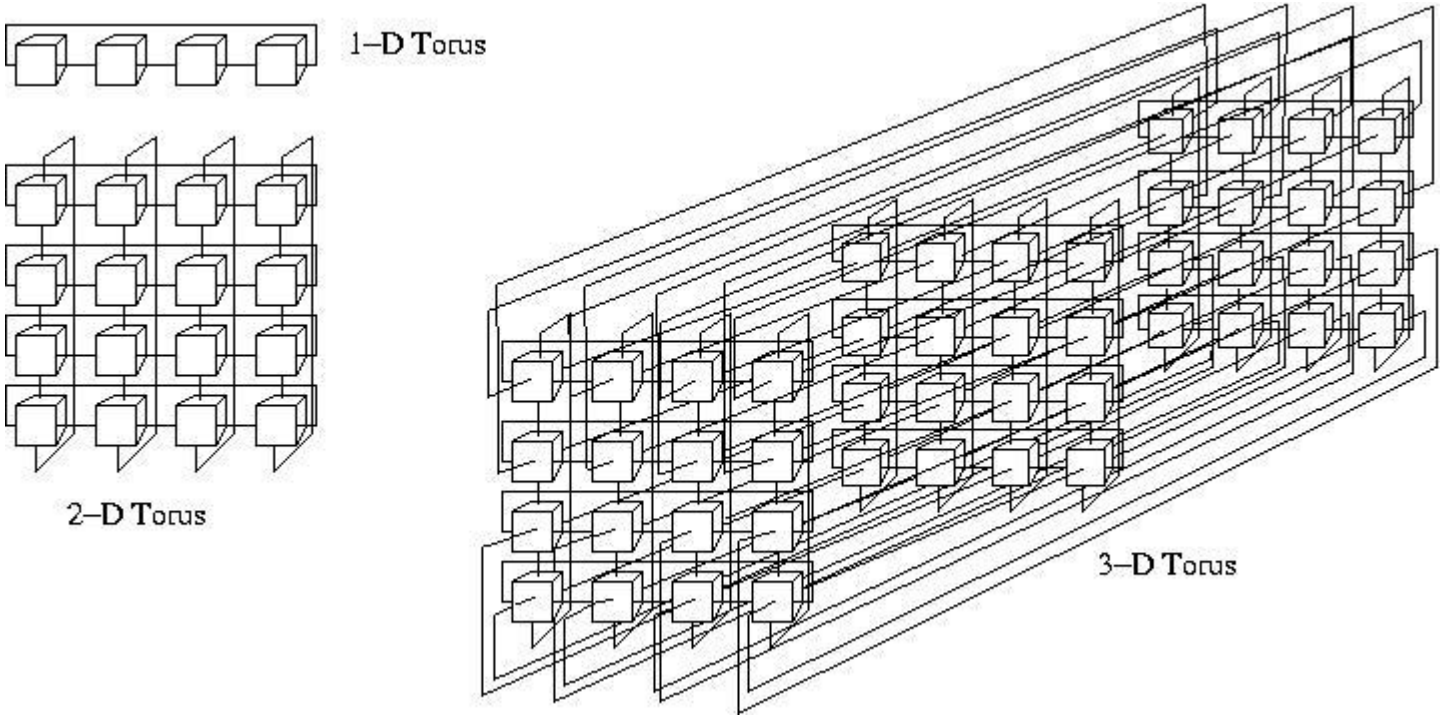
- **It can be shown ☺ that mapping arbitrary A to arbitrary H with optimizing a given metric is NP-hard**
- **Greedy Algorithm**
  - → Select two starting vertices
  - → Add other vertices to the mappings walking along the neighborhood
- **Graph Partitioning (bipartitioning)**
  - → Recursively cut both graphs into smaller pieces
  - → Perform mapping in the unfolding of the recursion
- **Mapping Enforcement – usable with MPI on current HPC systems**
  - → Application topology is described by the programmer, system topology is known
  - → Pre-defined app. top. can be mapped to sys top. by the scheduler

Application Study

- **Application study: multi-physics code from LLNL on BG/P**
  - → laser-plasma interactions in a 3D grid
    - →Z-direction is aligned with laser beam
  - → 3 phases / 3 communication patterns
    - →wave propagation: 2D FFTs in XY-planes (orthogonal to laser)
      - →MPI_Alltoall
    - →advection: consecutive planes in Z-directions (MPI_Send and _Recv)
    - →hydrodynamics: near-neighbour data exchange in all directions
      - →less frequent
- **Natural domain decomposition: n_z planes with n_x cols and n_y rows**
  - → Experiment setup: $n_x = 16$, $n_y = 8$, n_z according to no. of processors

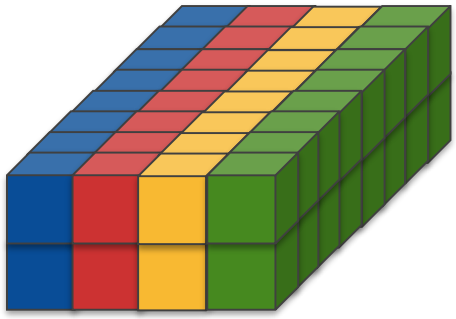
Source. Mapping Applications with Collectives over Sub-communicators on Torus Networks, A. Bhatele et al., SC 2012

Review: 3D Torus

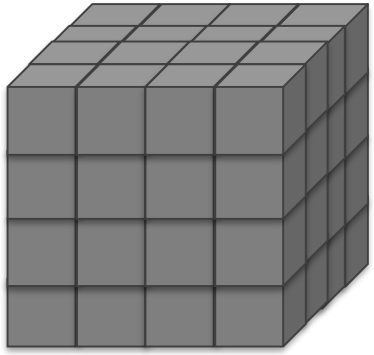


http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_506_Spring_2010/ch_12_PP

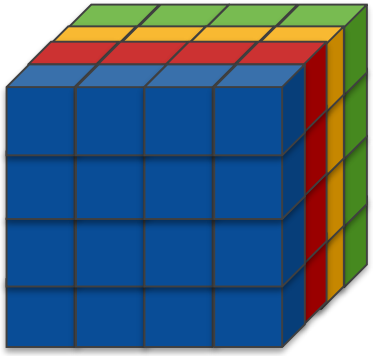
Mapping: Example



Application domain



Network domain

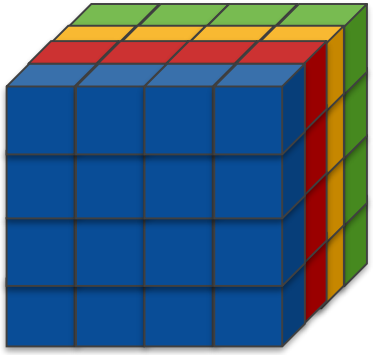


Mapped application ranks on network

Mapping: Divide

- **TXYZ mapping (default on IBM Blue Gene/P with a 3D torus network): hardware threads within a node → nodes along X direction of the torus → Y direction → Z direction**

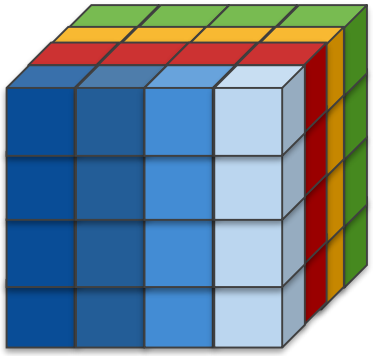
  → Good default for near-neighbor communication



- **Reordering of TXYZ could improve data locality for a specific data structure/ application, e.g., XYZT**

Mapping: Tile

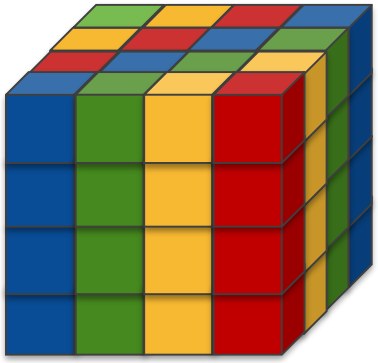
- **Distribution of the application domain into tiles of a specific size to match data access patterns of the application**



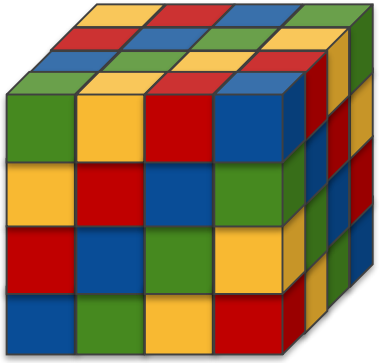
Tile of size 1x1x4

Mapping: Tilt

- **Mostly all-to-all communication is used for this application**
  - → A tilted mapping can increase the number of links available



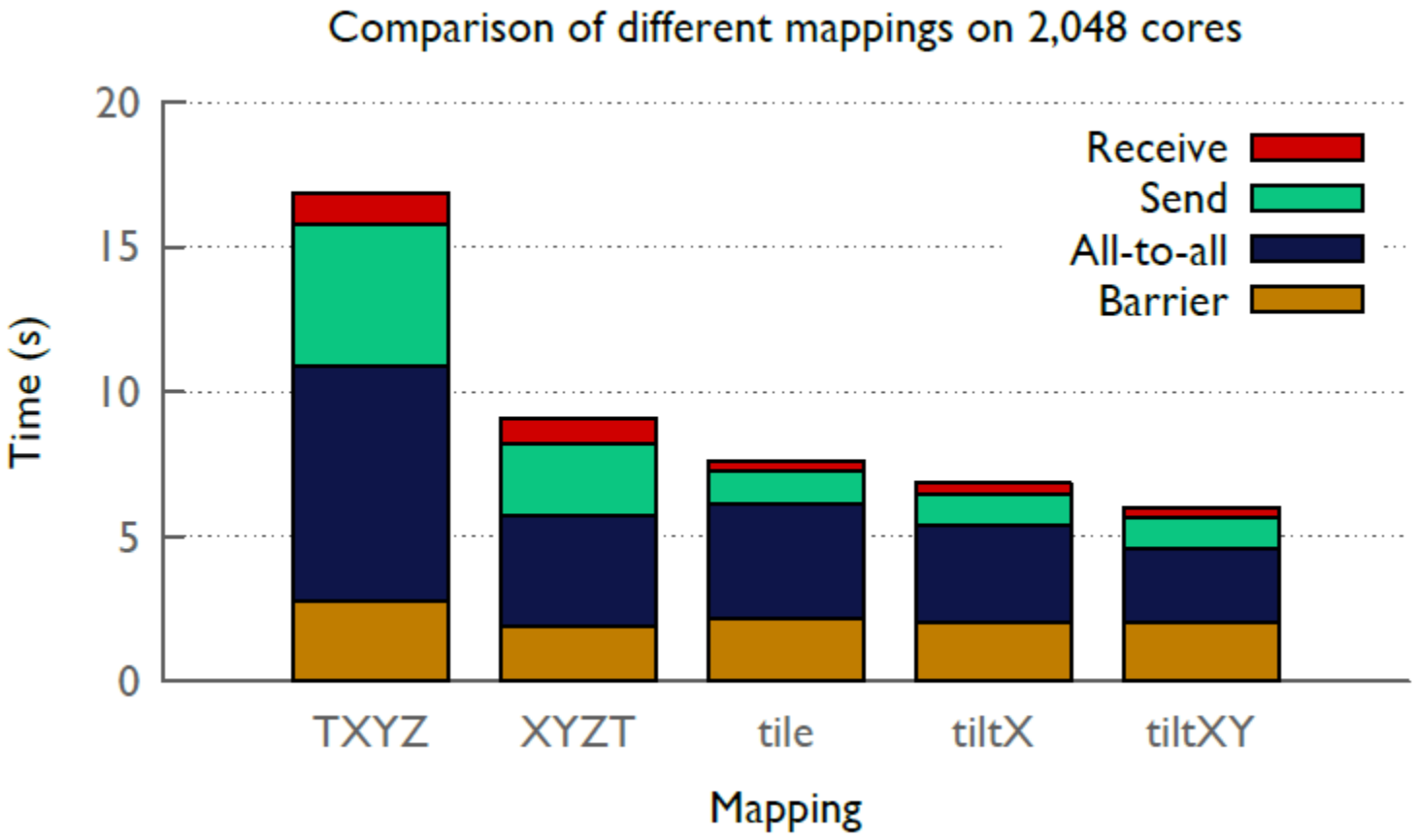
Tilt in x direction



Tilt in x and y directions

Evaluation of Mappings: Runtime

- **2048 cores on IBM Blue Gene/P: 3D Torus network**



Creating communicators

- **Duplicate an existing communicator**

```
MPI_Comm_dup (comm, newcomm)
```

→ Collective call – all processes in comm must call it in order to complete

→ All topology information and cached values are copied

- **Split an existing communicator**

```
MPI_Comm_split (comm, color, key, newcomm)
```

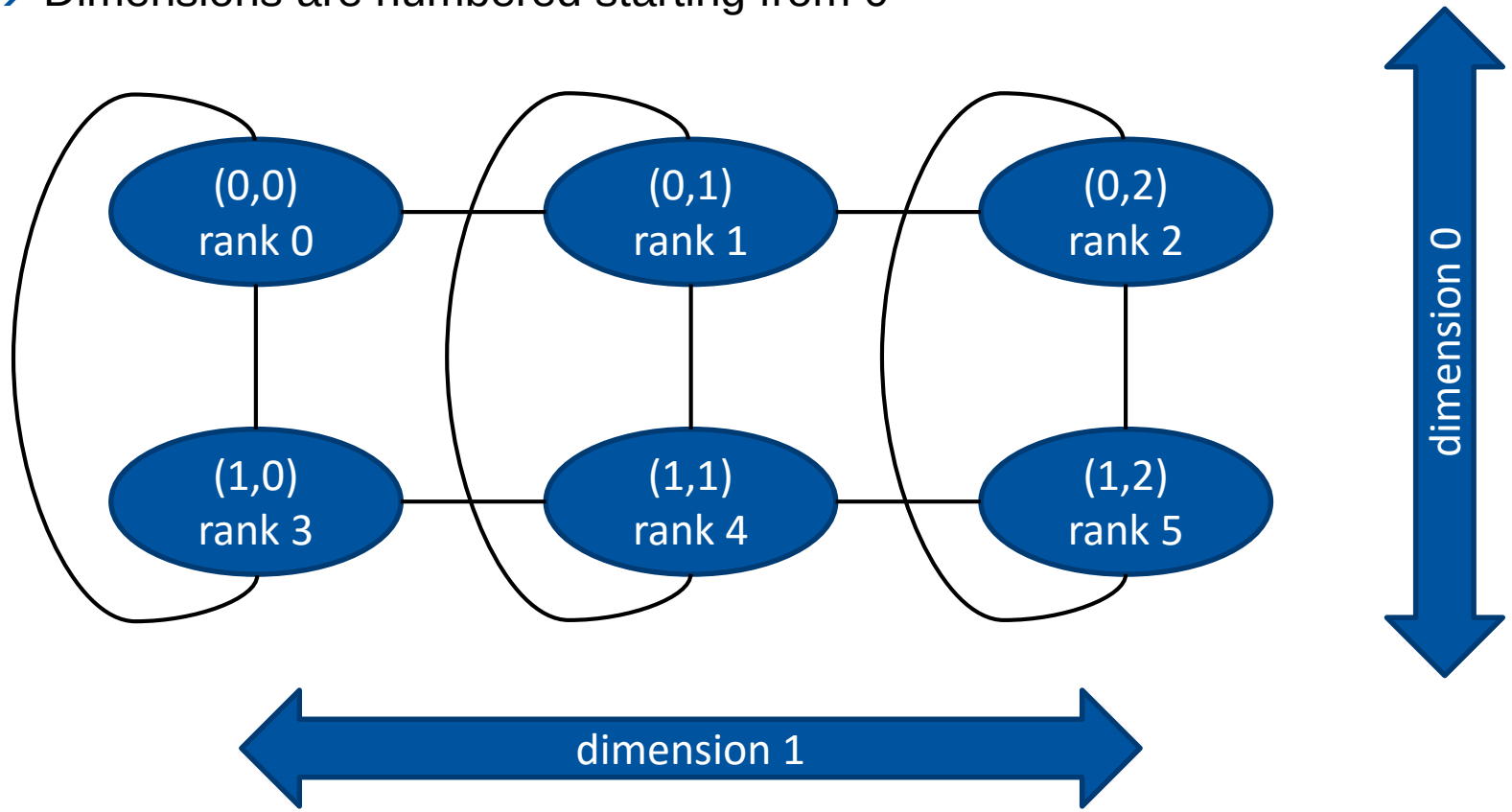
→ A new subcommunicator is created for each value of **`color`**

→ **`MPI_UNDEFINED`** if a process should not participate in the split

→ **`key`** is used to assign ranks within the new group (old ranks used if tie)

Cartesian topology

- **Regular n-dimensional grid**
  - → Dimensions are numbered starting from 0



Balanced Cartesian distribution

- **Create a balanced distribution of a number of processes**

```
MPI_Dims_create (nnodes, ndims, dims)
```

→ Computes the most balanced way to arrange **nnodes** processes into an **ndims**-dimensional grid

→ Non-zero elements in **dims** fix the number of processes in the corresponding dimension

→ Zero elements are filled with the computed optimal number of processes along the corresponding dimension

→ Error if the product of non-zero elements of **dims** does not divide **nnodes**

Balanced Cartesian distribution

- **Create a balance distribution of a number of processes**

```
MPI_Dims_create (nnodes, ndims, dims)
```

→ The computed sizes are set in non-increasing order

→ the lowest-numbered dimension receives the biggest size

→ Example (taken from the MPI standard):

dims before call	function call	dims on return
(0,0)	`MPI_Dims_create(6, 2, dims)`	(3,2)
(0,0)	`MPI_Dims_create(7, 2, dims)`	(7,1)
(0,3,0)	`MPI_Dims_create(6, 3, dims)`	(2,3,1)
(0,3,0)	`MPI_Dims_create(7, 3, dims)`	erroneous call

Cartesian topology

Construct a Cartesian topology

```
MPI_Cart_create (old_comm, ndims, dims, periods,
                 reorder, comm_cart)
```

- Creates a new communicator **`comm_cart`** from the process group of **`old_comm`** with an **`ndims`**-dimensional Cartesian topology attached
- **dims[]** – specifies the number of processes along each dimension
- **periods[]** – specifies the periodicity in each dimension
- **reorder** – if set to .TRUE./non-zero, hints the MPI runtime to reorder the ranks in the new communicator so that their physical connectivity matches as closely as possible the virtual one; otherwise ranks are kept

Coordinate conversion

- **Translate a Cartesian coordinate tuple into a rank**

```
MPI_Cart_rank (comm, coords, rank)
```

  - → **comm** – Cartesian communicator
  - → **coords** – an array of at least **ndims** elements – Cartesian coordinates
  - → **rank** – corresponding process rank in **comm**

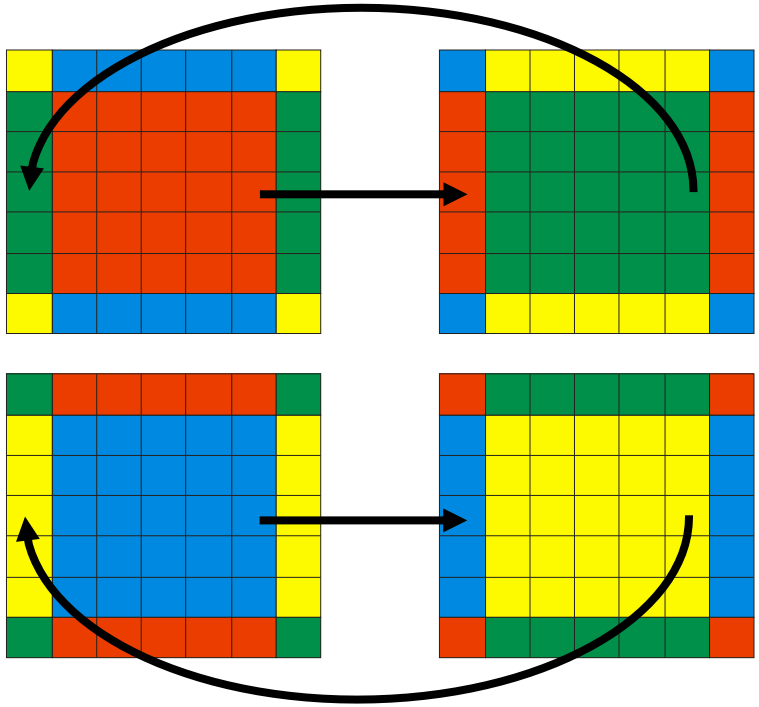
- **Translate a rank into a Cartesian coordinate tuple**

```
MPI_Cart_coords (comm, rank, maxdims, coords)
```

  - → **coords** – an array of **maxdims** elements to receive the coordinates
  - → **maxdims** should be equal to or larger than **ndims**

Cartesian shifts

- **Cartesian shifts are most often used for halo regions / ghost cells exchanges**
  - → Each process sends halo information to its right neighbour
  - → Each process receives halo information from its left neighbour



Cartesian shifts

- **Find ranks of neighbour processes**

```
MPI_Cart_shift (comm, dir, disp, source, dest)
```

→ Computes the ranks to communicate with in order to perform a data shift (using **`MPI_Sendrecv`**) at a distance of **`disp`** in direction **`dir`**

→ Equivalent to:

→ obtain the Cartesian coordinates of the calling process

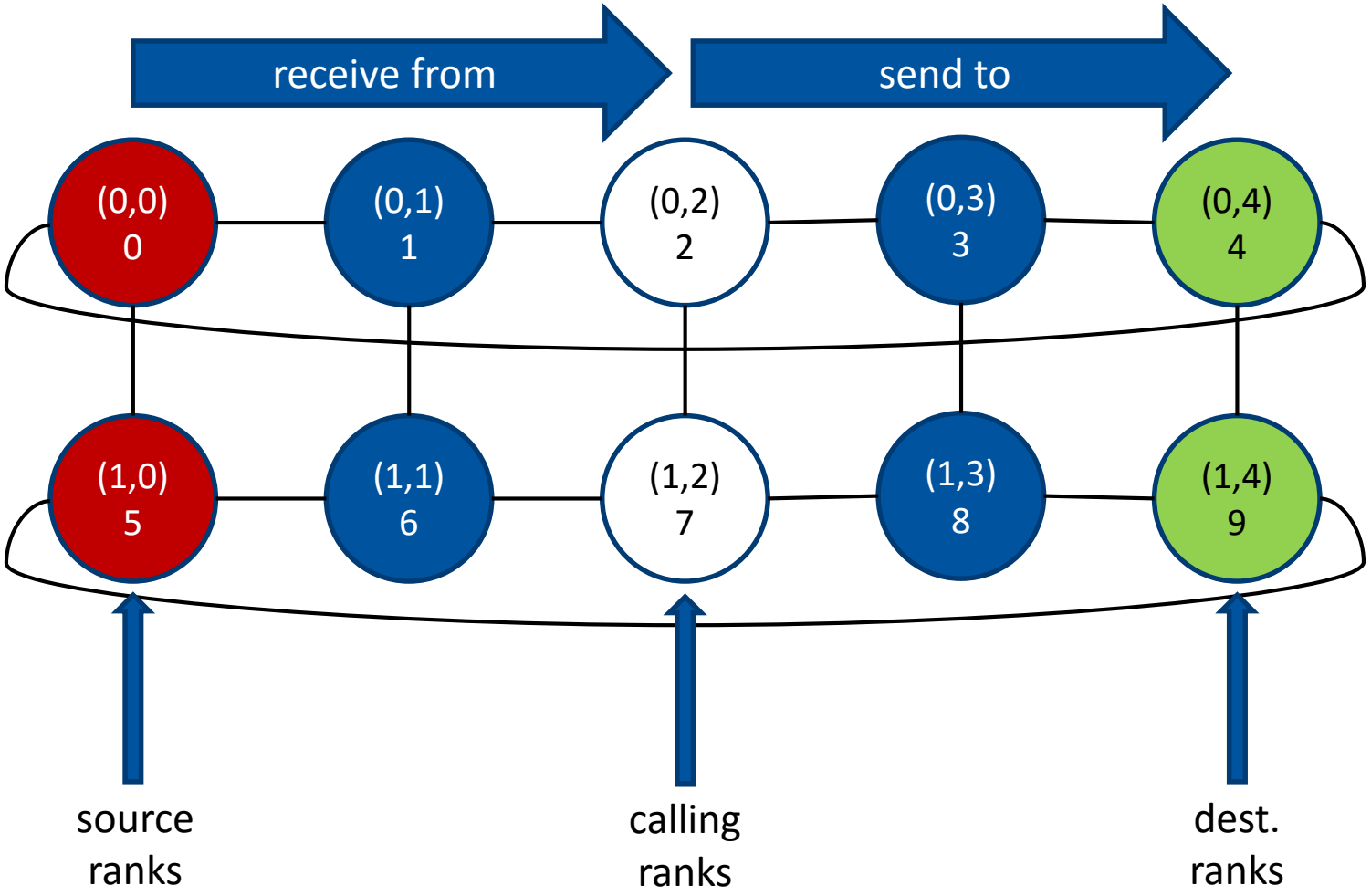
→ translate (…, $coord_{dir}$ + disp, …) into rank **`dest`**

→ translate (…, $coord_{dir}$ – disp, …) into rank **`source`**

→ If the source or the destination lies beyond a non-periodic boundary, the corresponding rank is set to **`MPI_PROC_NULL`**

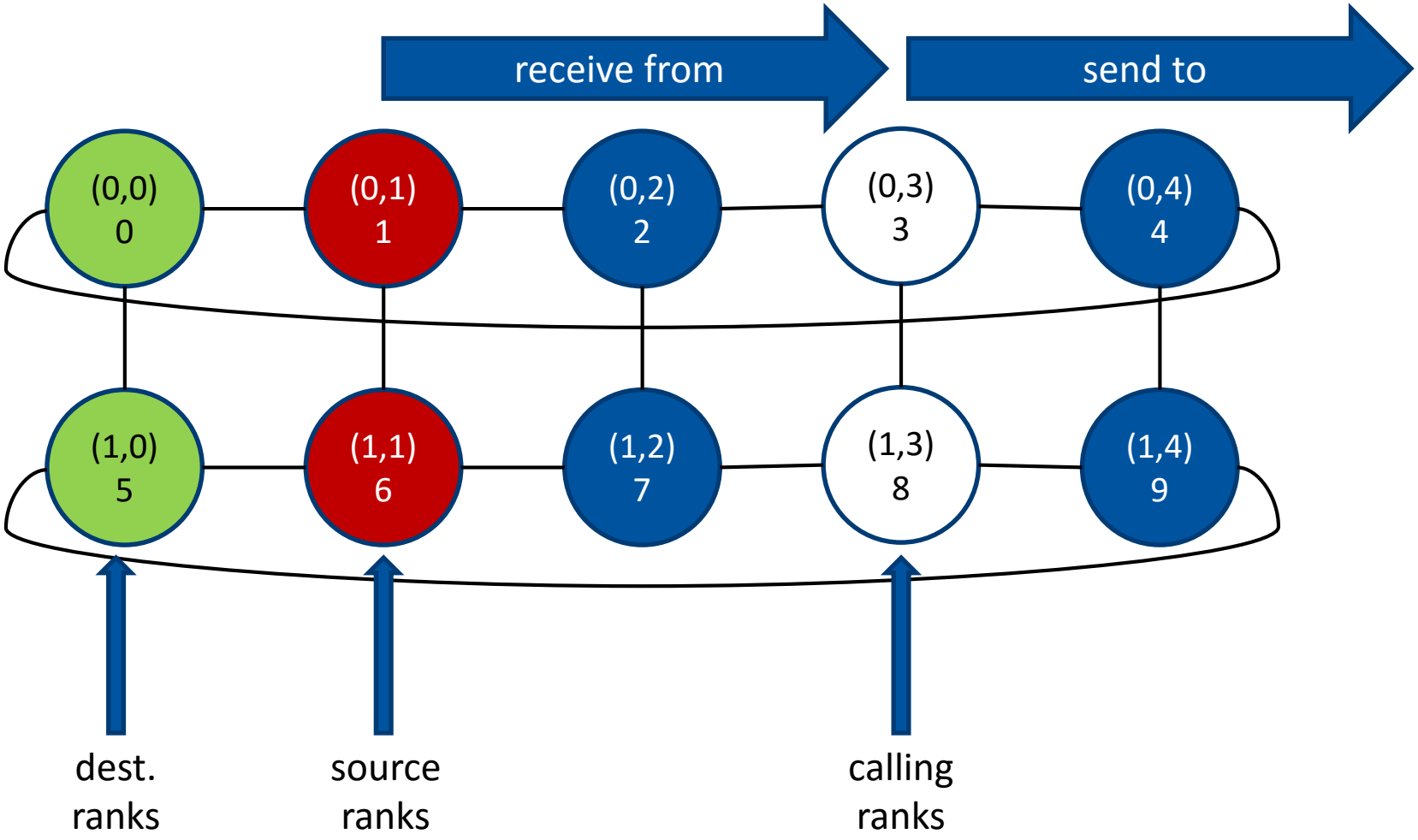
Cartesian shifts

- **Example: periodic boundary (dir = 1, disp = 2)**



Cartesian shifts

- **Example: periodic boundary (dir = 1, disp = 2)**



Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Shared-memory programming with OpenMP
8. **Distributed-memory programming with MPI**
   - The SPMD model revisited
   - MPI basics
     - Point-to-point communication
     - Non-blocking operations
     - Building and running MPI programs
   - Advanced MPI topics
     - Collective communication
     - Virtual topologies
     - **Derived datatypes**
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Derived datatypes

- **Basic MPI datatypes can be combined into more complex datatypes**
  - → Derived datatypes can be further combined into even more complex derived datatypes
- **MPI datatypes are essentially instructions for accessing the contents of the buffer memory**
  - → type sequence – *(basic (language) datatype, displacement)*
    - → displacements are relative to the beginning of the memory buffer and can be positive or negative
  - → type map – { ($type_0$, $disp_0$), …, ($type_{n-1}$, $disp_{n-1}$) }
  - → type signature – { $type_0$, …, $type_{n-1}$ }
- **The type signature at the sender must match that at the receiver**
  - → Also called type congruency

Characteristics of a datatype

- **Lower and upper bounds:**
  - → *lb*(datatype) = min $disp_j$
  - → *ub*(datatype) = max ($disp_j$ + sizeof($type_j$)) + padding
- **Extent**
  - → *extent*(datatype) = *ub*(datatype) – *lb*(datatype)
  - → The span in memory from the first to the last basic element
- **Size**
  - → *size*(datatype) = sum sizeof($type_j$)
  - → The total amount of bytes taken by the basic elements of the datatype, not counting the gaps between them

Characteristics of a datatype

- **Example: MPI_INT**
  - → type map = (*int*, 0)
  - → *lb* = 0
  - → *ub* = 4
  - → *extent* = 4 bytes
  - → *size* = 4 bytes

- **All predefined basic MPI datatypes have lower bound of 0**
  - → Means that data starts right at the location provided in the buffer pointer
- **Platform-specific alignment rules are taken into account**
  - → The upper bound is adjusted if necessary

Contiguous datatypes

- **Create a sequence of elements of some existing datatype**

```
MPI_Type_contiguous (count, oldtype, newtype)
```

  - → The new datatype represents a contiguous sequence of **`count`** elements of the old datatype
  - → The elements are separated from each other by the extent of **`oldtype`**
    - →Padding (e.g., for alignment) is automatically taken care of
  - → A send/receive of one element of **`newtype`** is congruent with a receive/send of **`count`** elements of **`oldtype`**

- **Useful for sending whole matrix rows (C) or columns (Fortran)**

Vector datatypes

- **Create a sequence of equally spaced blocks of elements**

```
MPI_Type_vector (count, blen, stride, oldtype, newtype)
```

  - → The new datatype represents a sequence of **`count`** blocks, each containing **`blen`** elements of the old datatype
  - → Every two consecutive blocks are separated by **`stride`** elements each

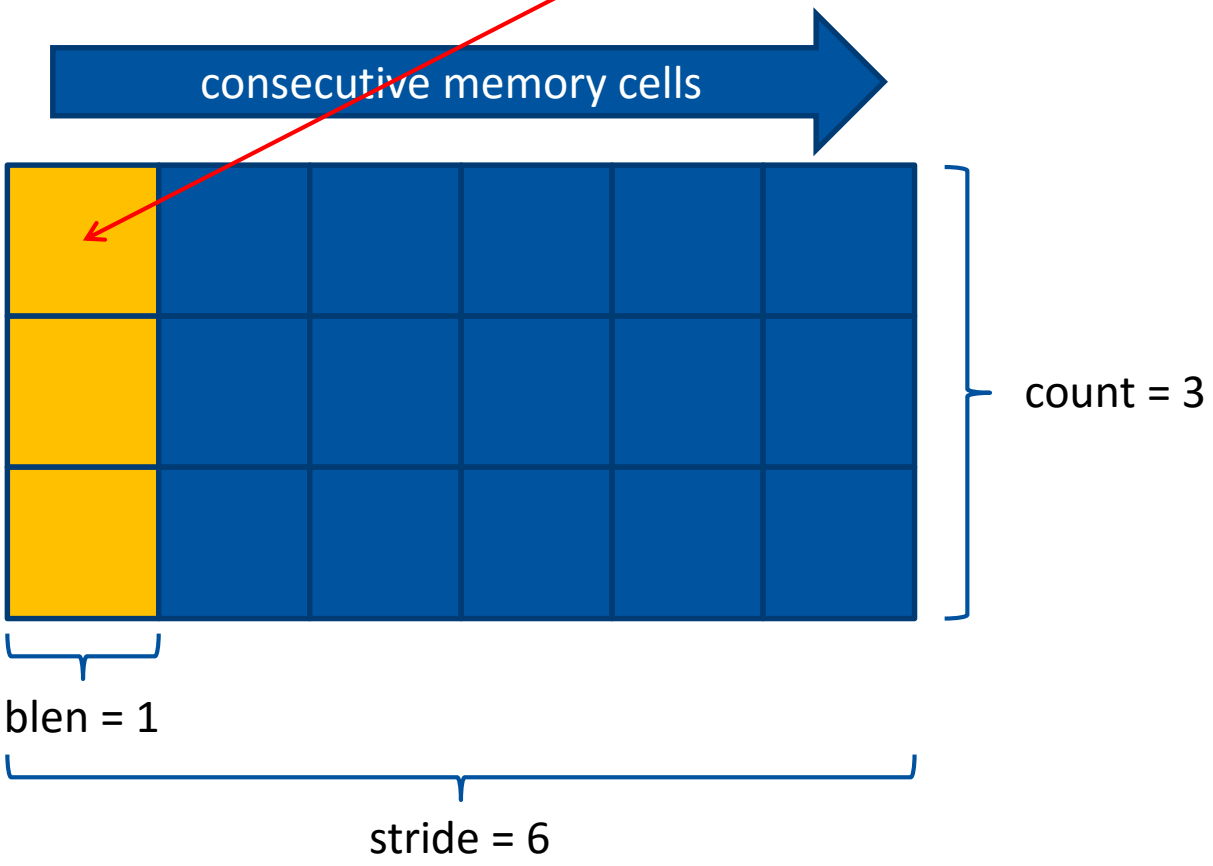
- **Useful for sending matrix columns (C) or rows (Fortran)**
  - → **`stride`** = row (C) / column (Fortran) length (in number of elements)
  - → **`blen`** = 1 (or the number of consecutive rows/columns)
  - → **`count`** = number of rows (C) / columns (Fortran)

Vector datatypes

- **Example: single column of a C matrix**

→ float mat[3][6]

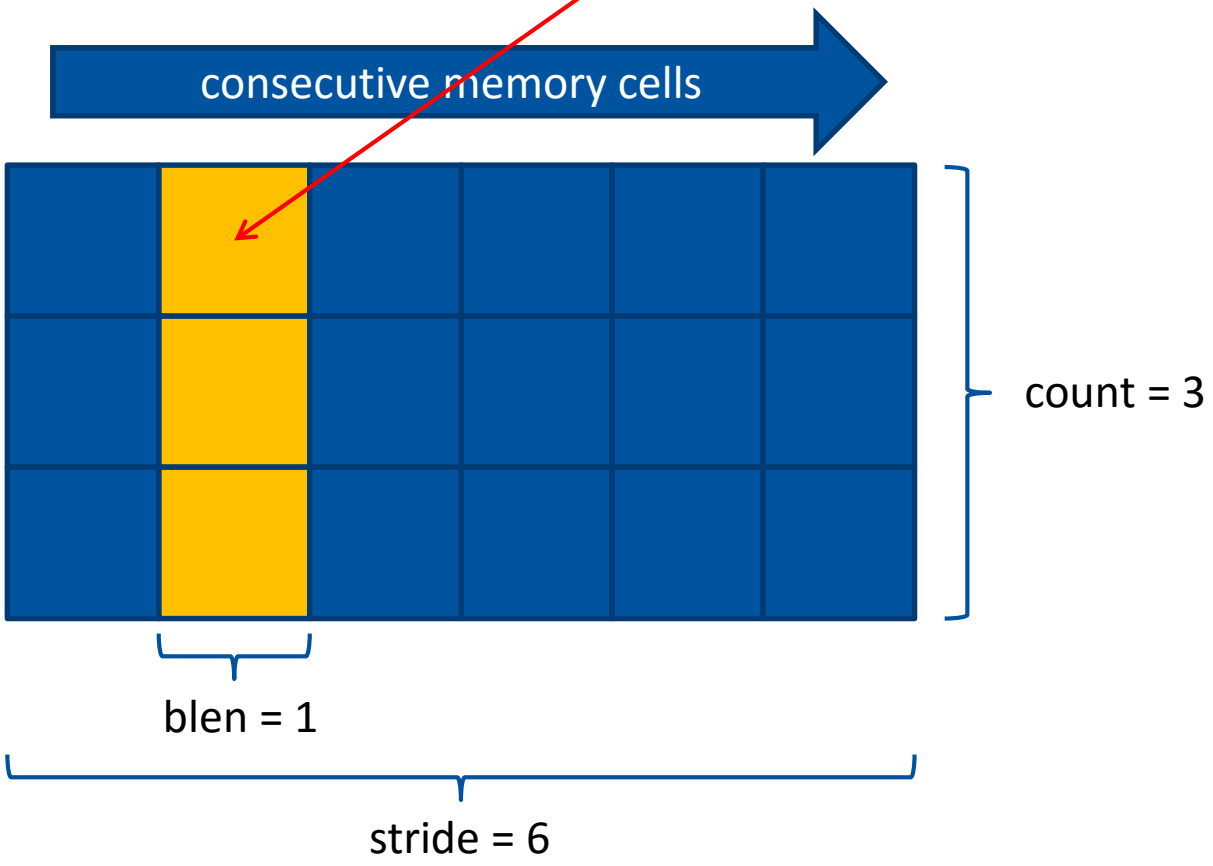
```
MPI_Send(&mat[0][0], 1, coltype, …)
```



Vector datatypes

- **Example: single column of a C matrix**
  - → float mat[3][6]

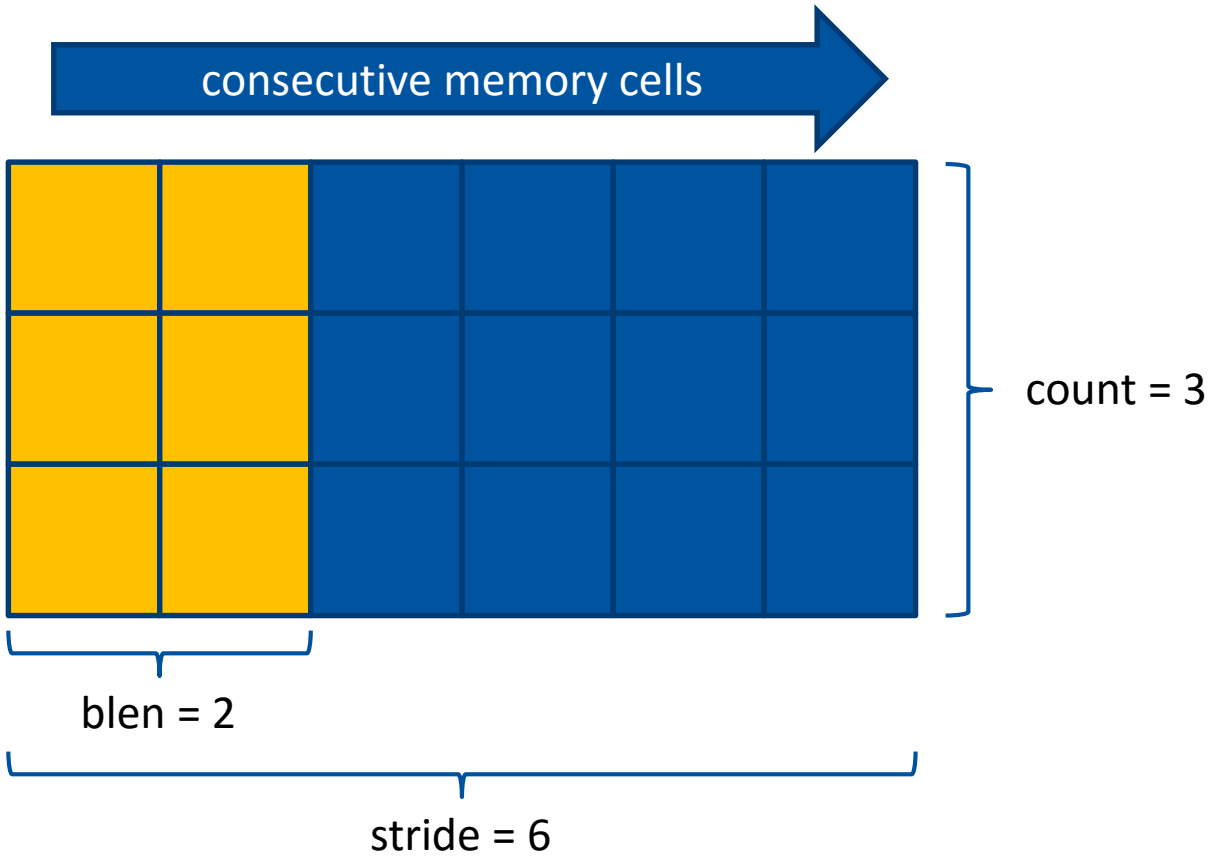
```
MPI_Send(&mat[0][1], 1, coltype, …)
```



Vector datatypes

- **Example: two consecutive columns of a C matrix**

  → float mat[3][6]



Using derived datatypes

- **Register a datatype for use with communication operations**

```
MPI_Type_commit (datatype)
```

  - → A datatype must be committed before it can be used in communications
  - → All predefined datatypes are already committed
  - → Intermediate datatypes, i.e. ones used in building more complex datatypes but not used in communication, can be left uncommitted

- **Deregister and free a datatype**

```
MPI_Type_free (datatype)
```

  - → Derived datatypes, build from the freed datatype, are not affected

Usage of derived datatypes

- **Datatypes can be mixed and matched on both sides of a communication operation as long as their type signatures match**
  - → Type congruency
  - → E.g. one can send 10 **`MPI_INT`** elements and receive them as a single element of a contiguous datatype with **`count`** = 10 and **`oldtype`** = **`MPI_INT`**
  - → Extra care should be taken when using derived datatypes in collective operations

- **If the amount of data in a received message is not enough to build an integer number of elements of a derived datatype, a count of `MPI_UNDEFINED` is returned by `MPI_Get_count`**

What you have learnt

- **Basic ideas of MPI**
  - → What is SPMD and how is it implemented by MPI?
  - → How MPI abstracts the communication?
- **Point-to-point operations**
  - → How to transfer data between processes in the form of messages?
  - → How to prevent deadlocks and overlap communication and computation?
- **Collective operations**
  - → How to scatter and gather data and perform operations on distributed data?
- **Virtual topologies**
  - → How to distribute processes over a regular grid and shift data between them?
- **Derived datatypes**
  - → How to combine MPI datatypes into more complex entities?