Parallel Programming

Introduction to MPI and OpenMP

July 05. – 06. 2016 | Jochen Kreutz

Timetable

	Tuesday	Wednesday
08:30		Introduction to OpenMP
09:00	Ideas and Basics of Parallel Computing MPI: introduction	
09:30		
10:00		
10:30	Coffee break	Coffee break
11:00	MPI: (Non-) Blocking P2P communication	OpenMP: parallel loops and regions, worksharing conctructs
11:30		
12:00		
12:30	Lunch break	Lunch break
13:00		
13:30	MPI: Blocking collective communication	MPI, OpenMP HandsOn
14:00		
14:30		
15:00		
15:30		
16:00	Coffee break	
16:30 – 19:00	MPI: Outlook (derived datatypes, communicators and groups, MPI I/O etc.)	

Slides and literature

- Slides: based on “Parallel Programming with MPI and OpenMP for JSC Guest Students (2015)” by F. Janetzko, A. Schnurrpfeil
  - *Available for download from JSC homepage (JSC online → documentation → presentations)*

- Parallel Programming:
  - *D.A. Patterson, J.L. Hennessy “Computer Organization and Design”, 4th Ed., Elsevier, Amsterdam (2009).*
  - *B. Barney “Introduction to Parallel Computing”, LLNL: https://computing.llnl.gov/tutorials/parallel_comp/*
  - *Wikipedia: http://en.wikipedia.org*

Slides and literature

- MPI:
  - W. Gropp, E. Lusk, A. Skjellum “Using MPI: Portable Parallel Programming with the Message-Passing Interface”, 2nd ed., MIT Press, Cambridge (1999).
  - W. Gropp, E. Lusk, R. Thakur “Using MPI-2: Advanced Features of the Message-Passing Interface”, MIT Press, Cambridge (1999).
  - W. Gropp, T. Hoefler, R. Thakur E. Lusk “Using Advanced MPI - Modern Features of the Message-Passing Interface”, MIT Press, Cambridge (2014).
  - The MPI Forum “MPI: A Message-Passing Interface Standard”, Version 3.0 (2012).
  - http://www.mpi-forum.org/

Parallel Programming

Ideas and Basics of Parallel Computing

Outline

Motivation – Why going parallel?

Hardware: Basic concepts and terminology

- *Basics*
- *Multiprocessor architectures*
- *Processes and threads*

Software: Programming concepts

- *Programming paradigms: SPSD, SPMD, MPMD*
- *Parallel programming models*

Quality and limits of parallel programming

- *Efficiency, speed-up and scalability*
- *Amdahl's law*

Motivation – Why going parallel?

Growing demands of simulations

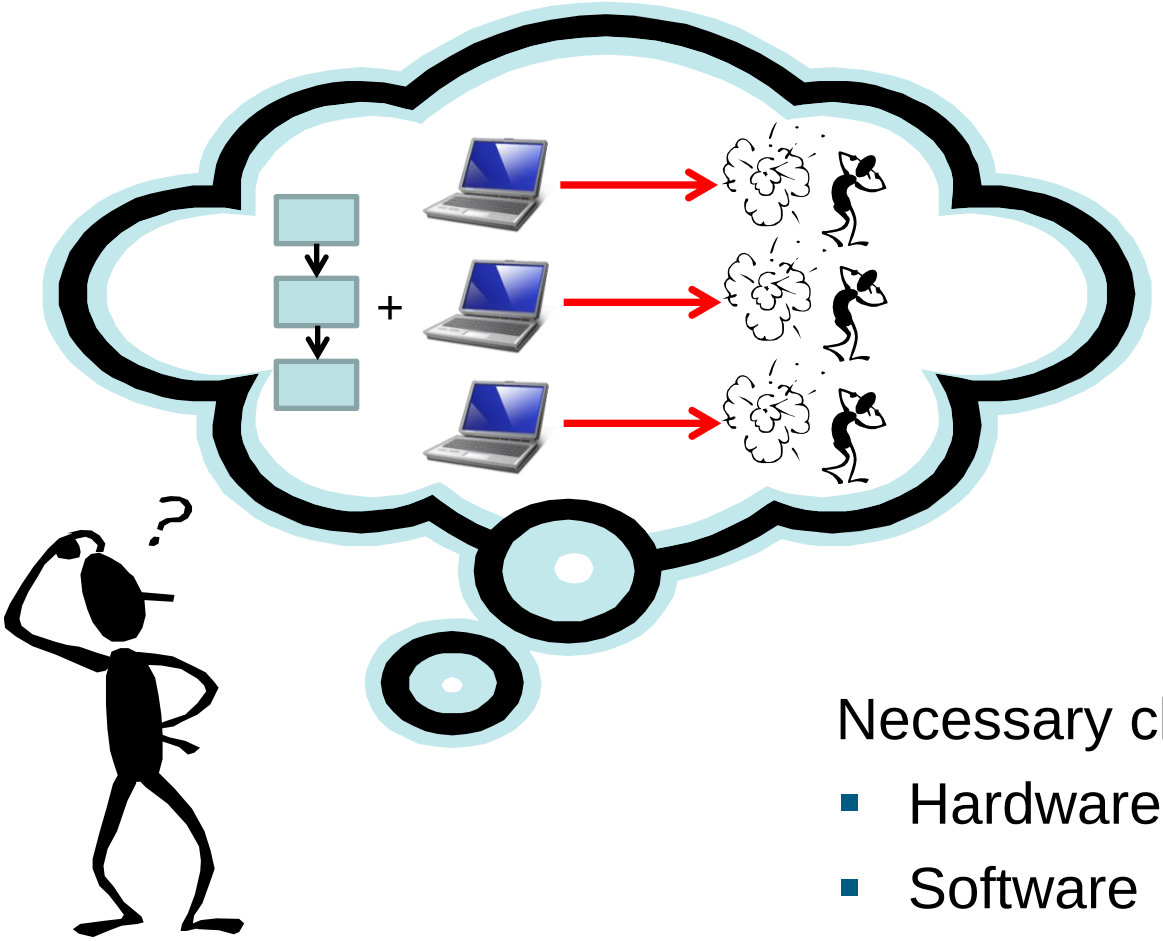
- Scientific problem sizes become larger
- Better accuracy/resolution required
- New kinds of scientific problems arise

Hardware limitations

- CPU frequency
- Cooling
- Power consumption

➢ What to do?

Parallel computing



Necessary changes

- Hardware
- Software

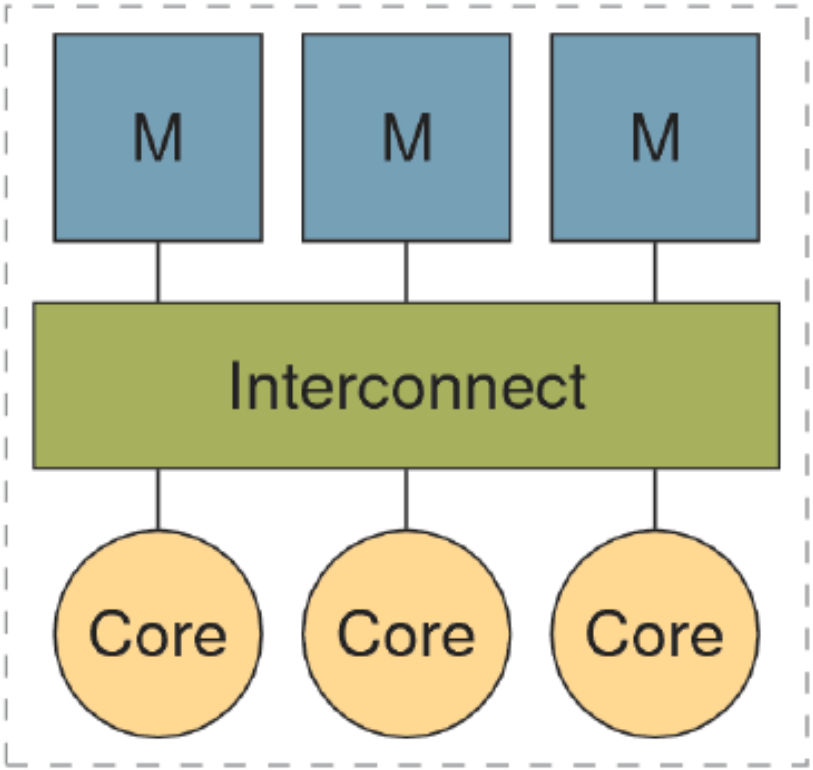
Hardware basics: Flynn's taxonomy

- Classification of computer architectures

	Single Instruction	Multiple Instructions
Single Data	SISD	MISD
Multiple data	SIMD	MIMD

- SISD: Uniprocessor, Pentium
- SIMD: SSE instruction of x86, vector processors, BGQ QPX
- MISD: not common, used for fault tolerance
- MIMD: Multiprocessor architecture

Multiprocessor architectures: shared memory

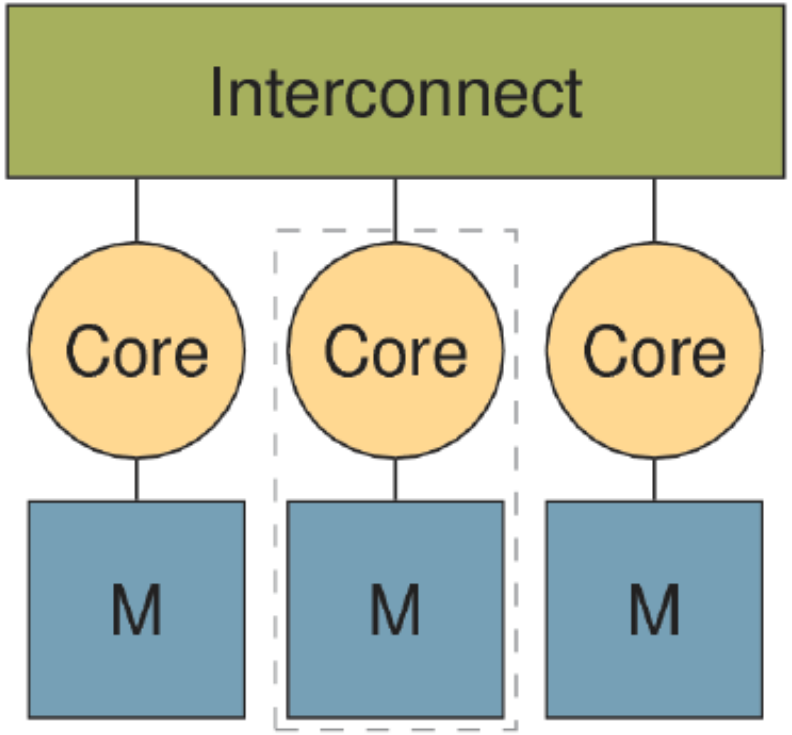


Characteristics

- All CPUs share the same memory
- Single address space
- Uniform memory access (UMA)

Nodes or systems of this type are called **Symmetric Multiprocessor** (SMP) nodes/systems

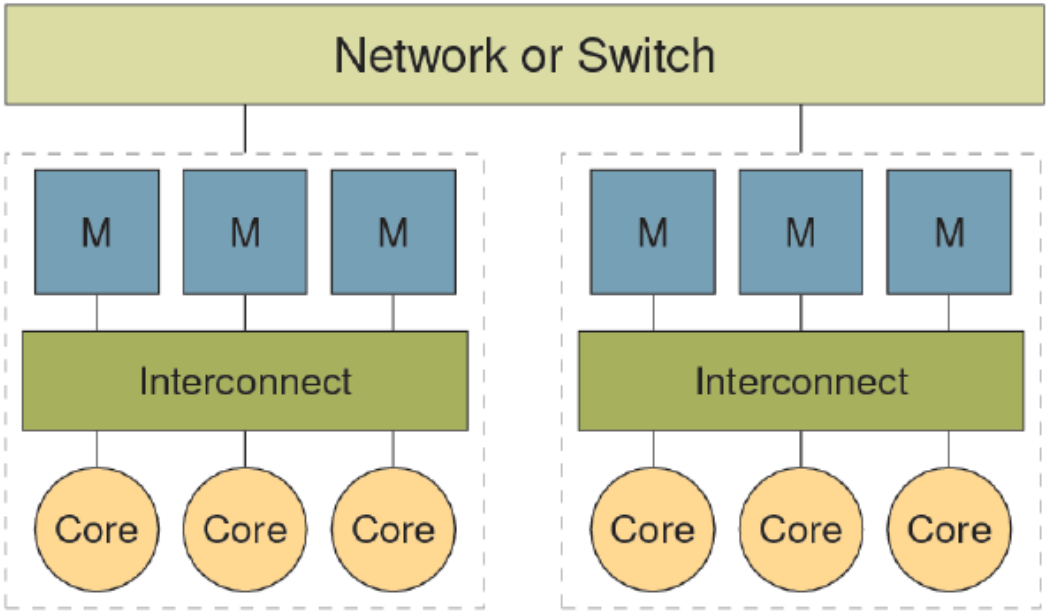
Multiprocessor architectures: distributed memory



Characteristics

- Each CPU has its own memory and address space
- Data exchange between memory of different CPUs
  - Via interconnect
  - Explicit data transfer necessary (message passing)

Nowadays multiprocessor architectures: hybrid distributed-shared architectures



Characteristics

- SMP with multiple cores (UMA within each SMP)
- Several SMP are combined in one compute node (CN) (NUMA between the SMP)
- CN are connected via a network
- Special nodes: CN, I/O nodes, login nodes

Processes and threads

Process

- Instance of the OS to execute a program
- Executes one or multiple → threads of execution

Thread

- Smallest unit of processing
- Sequence of instructions

A thread is started within a process and has

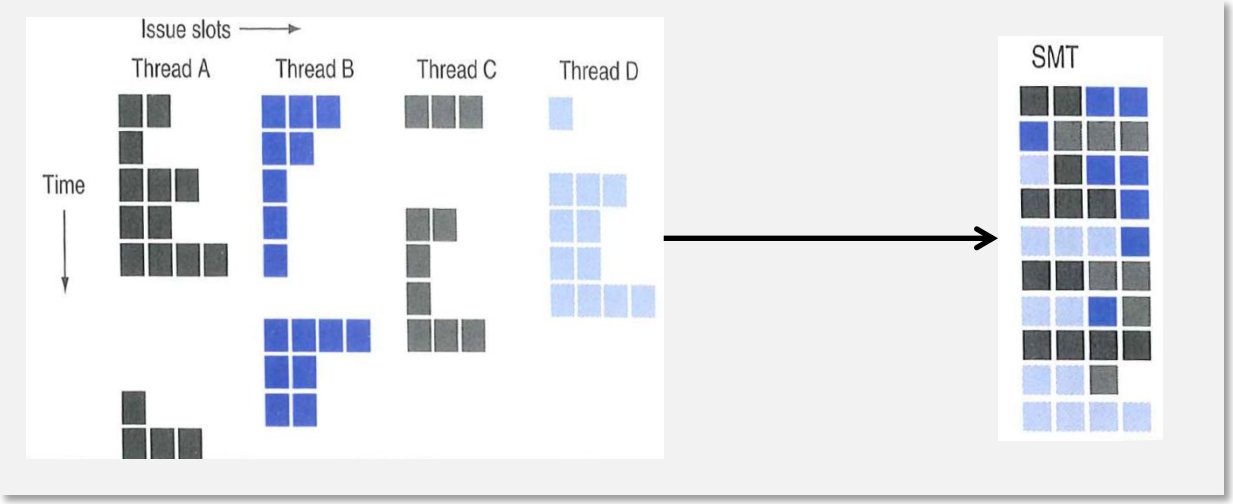
- A heap (for storing dynamic data)
- A stack (for storing local data)
- Access to global static data

Multithreading

Multithreading:

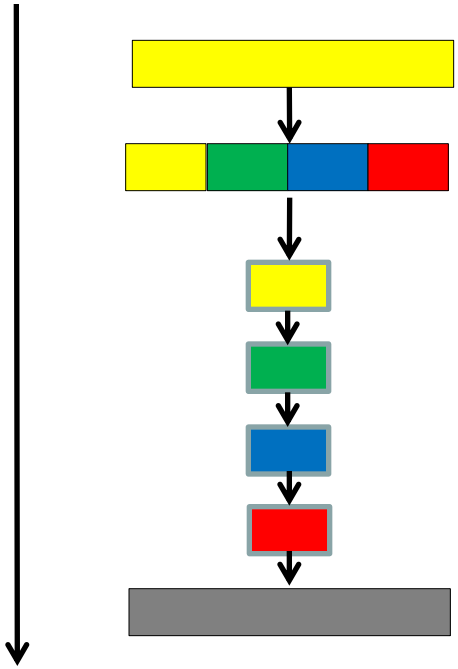
- A process can execute several threads
- Threads can be created and destroyed at run-time
- Threads share heap and static global data but have their own stack and registers

Example: Simultaneous multithreading (SMT) (Hardware MT)

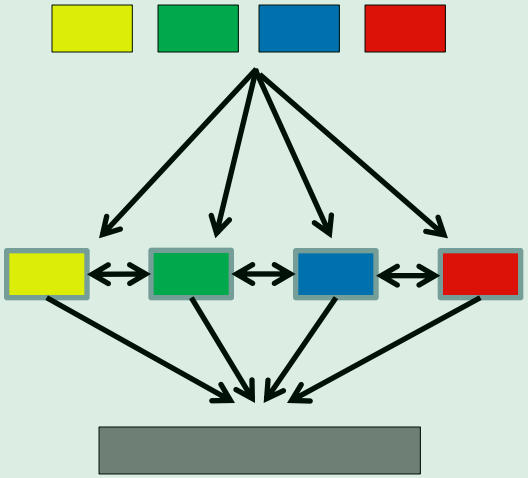


Software: programming paradigms

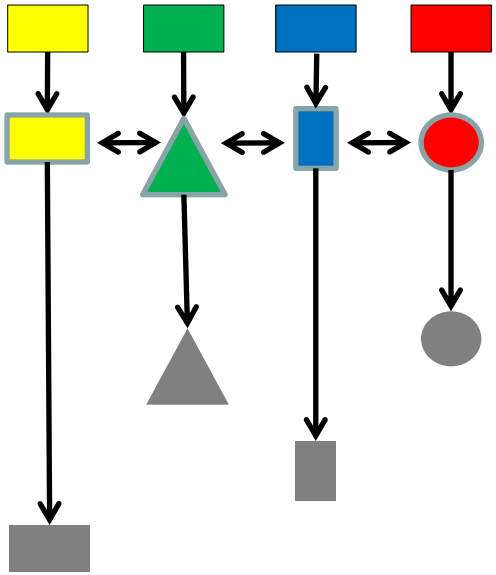
SPSD
(single program single data)



SPMD
(single program multiple data)



MPMD
(multiple programs multiple data)



Parallel programming models

Classification according to process interaction

1. Message passing
   - Parallel processes exchange data by passing messages
   - Examples: PVM, MPI
2. Shared memory
   - Parallel threads share a global address space
   - Examples: POSIX threads, OpenMP
3. Implicit
   - Process interaction is not visible to the programmer
   - Examples: PGAS (CAF, UPC), GA

Parallel scalability

(Parallel) Scalability

A measurement that indicates how efficient an application is parallelized when using increasing numbers of parallel processing elements (e.g. cores, processes, etc.)

Strong Scaling

The time to solve a problem is measured on increasing numbers of parallel processing elements while keeping the problem size is fixed.

Weak Scaling

The time to solve a problem is measured on increasing numbers of parallel processing elements while the problem size grows proportional to the number of parallel processing elements.

Strong scaling: speed-up and efficiency

Speedup

$$S(n) = \frac{t(1)}{t(n)}$$

S(n): Speedup on *n* cores or CPUS
t(1) : time on 1 core or CPU
t(n) : time on *n* cores or CPUs

Upper limit: $S(n) = n$ — Ideal Speedup on *n* cores

Lower limit: $S(n) = 1$ — No parallelism

Efficiency

$$E(n) = \frac{t(1)}{n \cdot t(n)} \cdot 100\%$$

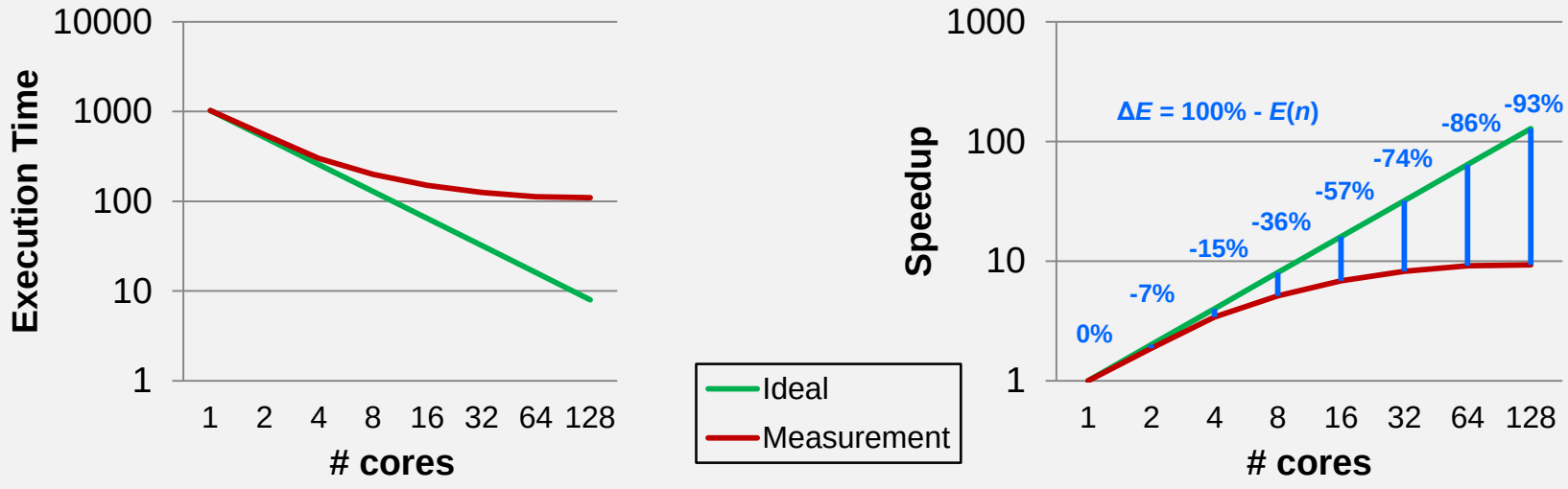
E(n): Efficiency on *n* cores or CPUS

$E(n) = 100\%$ — Ideal efficiency (perfect scaling)

$E(n) < 50\%$ — Considered generally as not efficient

Strong scaling

- CPU-bound simulations: solving a problem faster



! **Parallel overhead increases with number of processes** **!**

Weak scaling: efficiency

Efficiency

$$E(n) = \frac{t(1)}{t(n)} \cdot 100\%$$

E(n): Efficiency on *n* cores or CPUS
t(1) : time on 1 core or CPU
t(n) : time on *n* cores or CPUs

$$E(n) = 100\%$$

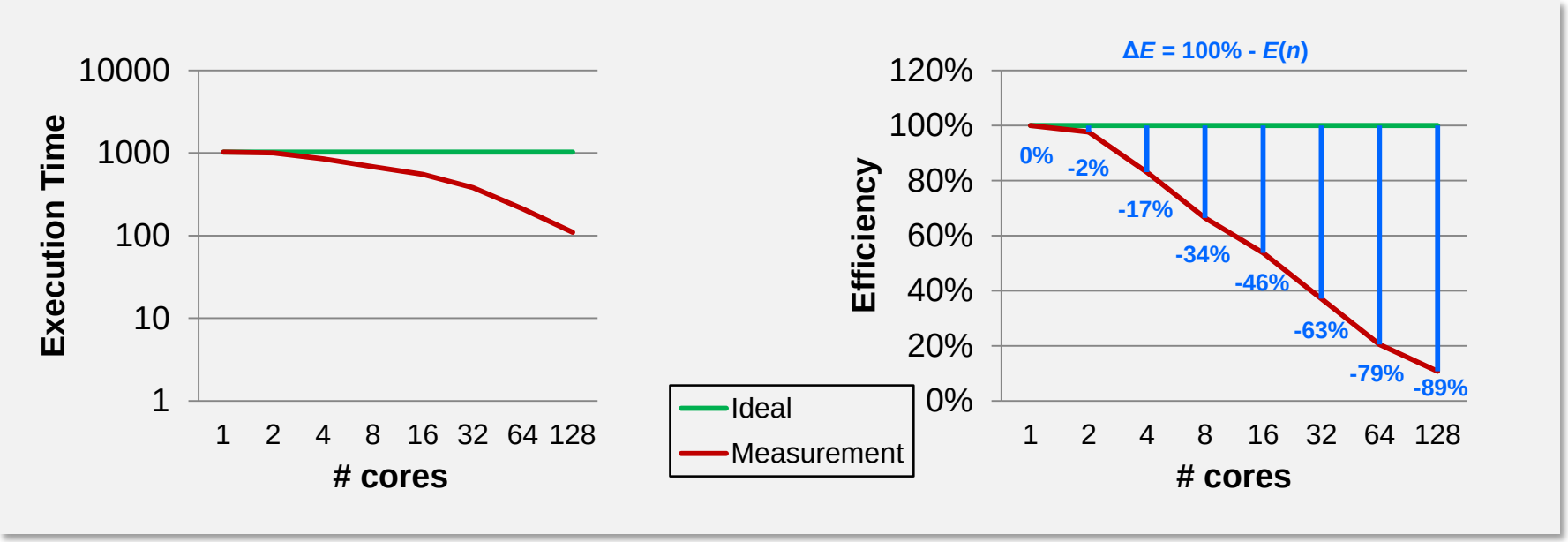
Ideal efficiency (perfect scaling)

$$E(n) < 50\%$$

Considered generally as not efficient

Parallelism: weak scaling

- Memory-bound simulations: solving larger problems



! Nearest-neighbor communication patterns needed for a good weak scaling behavior !

Amdahl's Law – Real speedup

Amdahl's Law

$$S_r = \frac{1}{\alpha + \frac{1-\alpha}{n}}$$

S_r: Real speedup

α : serial part (cannot be parallelized)

n : number of cores

Example

$\alpha = 0.1$
$n = 8$

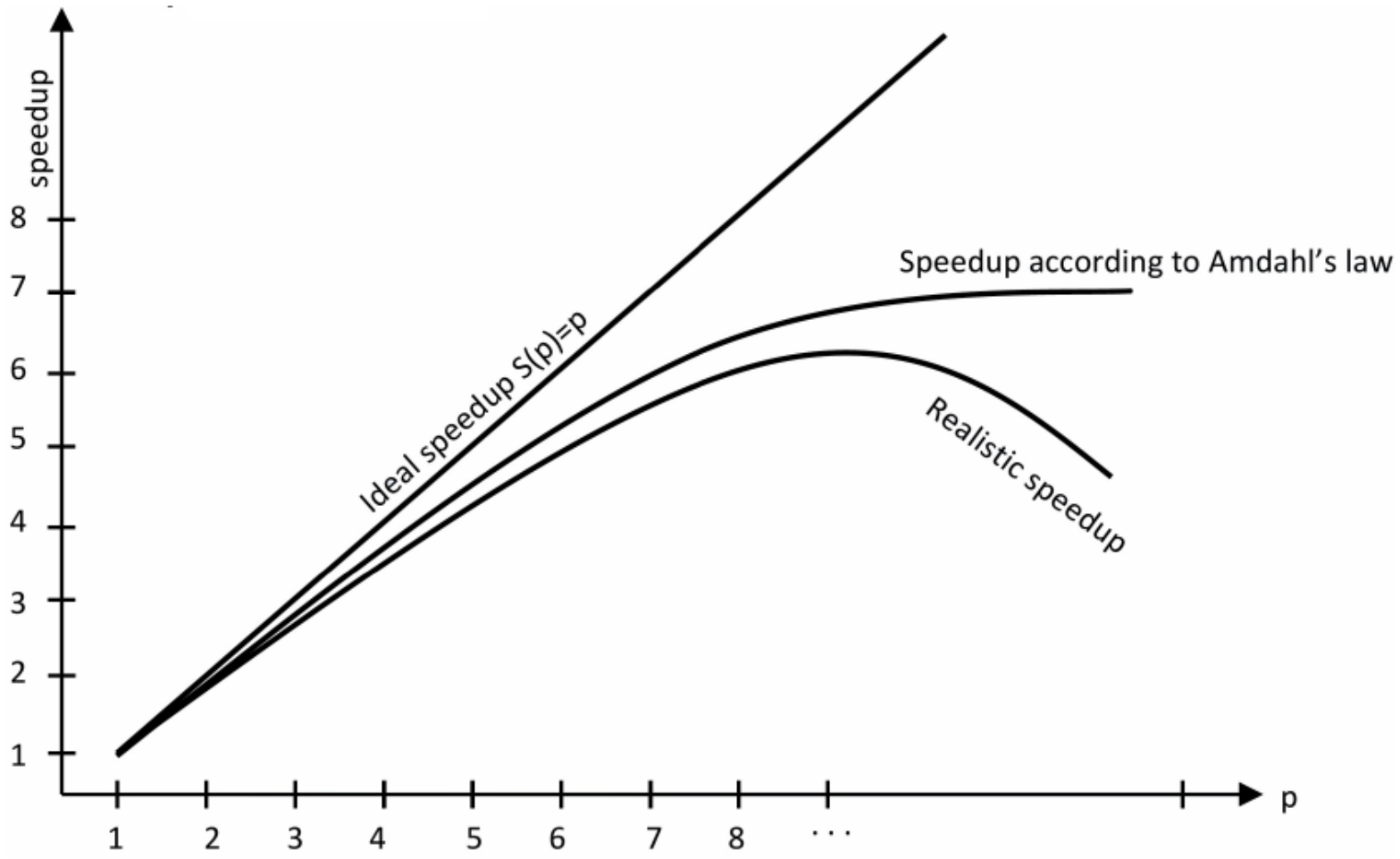
→

$S_r = 4.7$

$E(8) = 59\%$

! $$\lim_{n\to\infty}\left(\frac{1}{\alpha + \frac{1-\alpha}{n}}\right) = \frac{1}{\alpha}$$!

Amdahl's Law – Real Speedup



Parallelism – load balancing

Goal:

Divide work and/or communication between processors equally

- Work load on all processors is the same
- Communication load on all processors is the same

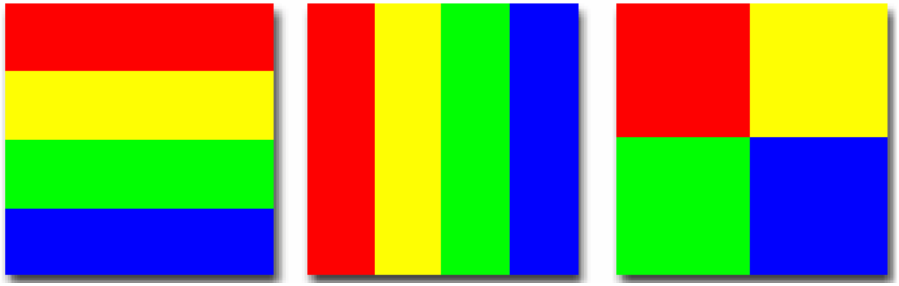
→ load balancing

- Many different types of load balancing problems:
  - Static (fixed, do it once) or dynamic (changing,adapt to load)
  - Parameterized or data dependent
  - Homegeneous or inhomogeneous
  - Low or high dimensional
  - Graph oriented, geometric, lexicographic, …
- Because of this diversity, many different approaches and tools are needed

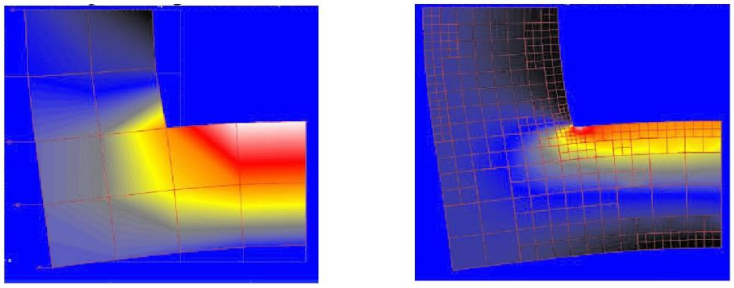
Parallelism – load balancing

- Domain decomposition
  - Static or dynamic decompostion of data set into data ranges (domains) creating the same (similar) processor load
  - Same task(s) being performed for all data anges
  - Targetting homogeneous compute resources

Static:



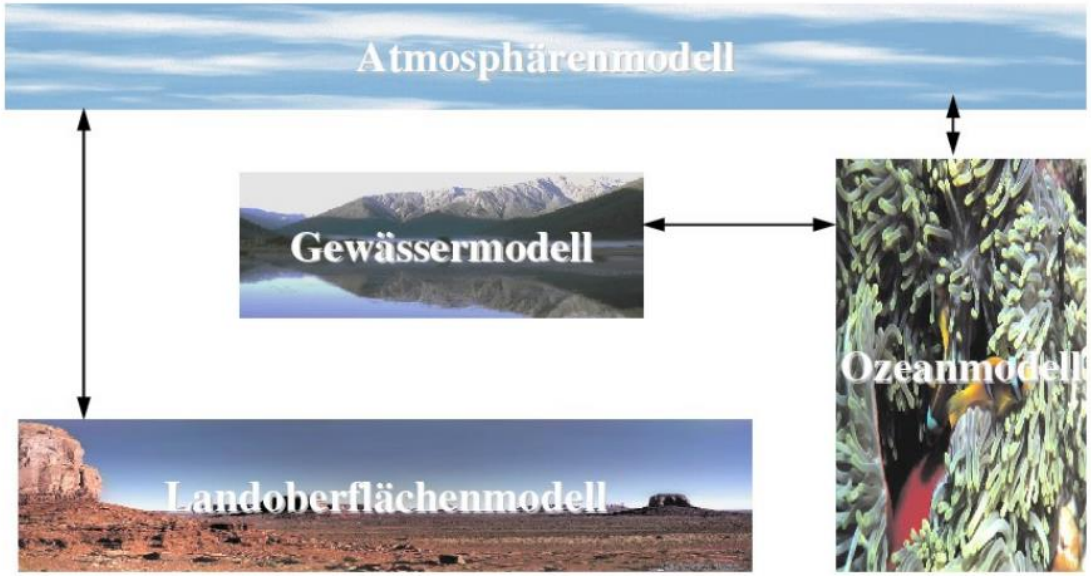
Dynamic



Parallelism – load balancing

- Functional decomposition
  - Different (independent) program parts being distributed and processed in parallel
  - Different hardware can be used (heterogenous compute resources)

- Example: climate simulaiton

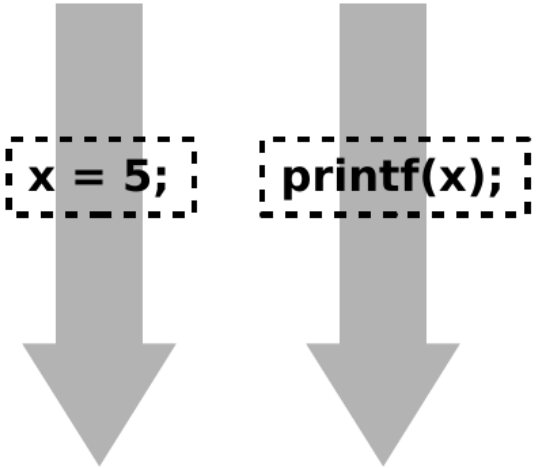


Data races – Race condition

- **Data races:**
  - Several threads access the same data (memory location) simultanously

- **Race condition:**
  - Result depends on the order of thread execution

- Example:
  - Init x = 1
  - Depending on which thread executed first, the result will be 5 or 1



Parallel Programming with MPI

Introduction

Introduction – What is MPI?

MPI (Message-Passing Interface)

Purpose: provision of a means for communication between processes

- Industry standard for a message-passing programming model
- Provides *specifications* (no implementations)
- Implemented as a library with language bindings for Fortran and C
- Portable across different computer architectures

Current version of the standard: 3.1 (June 2015)

Version of the standard used throughout this course: **3.0** (September 2012)

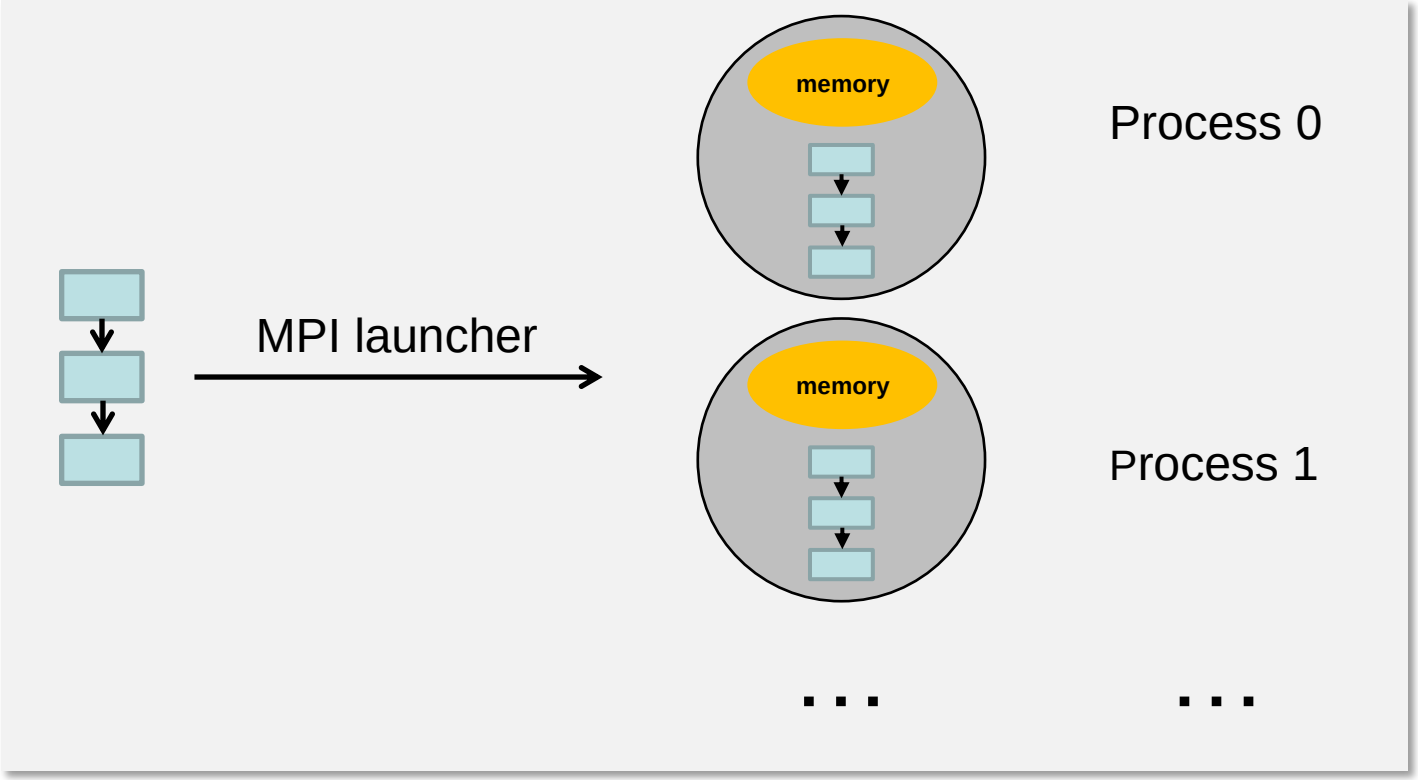
Brief history

History of MPI

<1992	several message-passing libraries were developed • PVM, P4, LAM …
1992	SC92: Several developers for message-passing libraries agreed to develop a standard for message passing
1994	MPI-1.0 Standard published
1997	Development of MPI-2 standard started
2008	MPI-2.1
2009	MPI-2.2
2012	MPI-3.0
2015	MPI-3.1, current version of the MPI standard

MPI – Programming model

SPMD – each process runs the same program



MPI terminology

Rank

A unique number assigned to each process of an MPI program within a group **(start at 0)**

Group

An ordered set of process identifiers (henceforth: processes)

Context

A property that allows the partitioning of the communication space

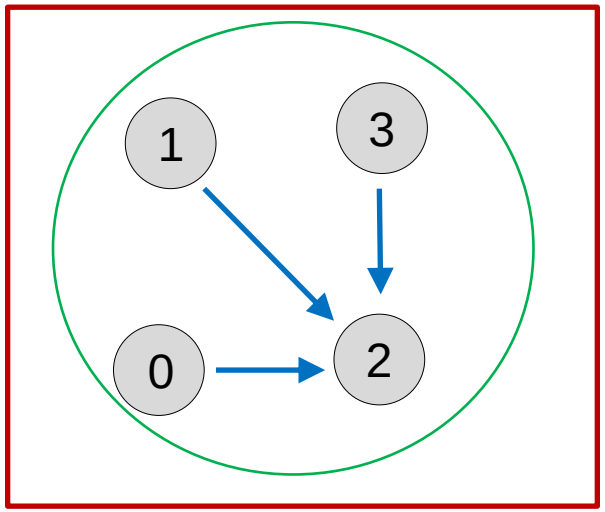
Communicator

Scope for communication operations within or between groups (intra-communicator or inter-communicator). Combines the concepts of group and context.

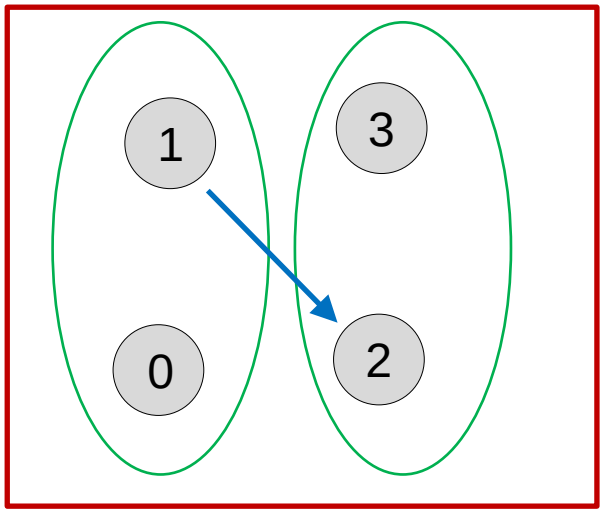
MPI terminology

(MPI 3.0, 6)

Intra-communicator



Inter-communicator



Process

Communicator

0 Rank

Communication

Group

The MPI infrastructure – Linking

Program must be linked against an MPI library

- Usually done using compiler wrappers

Fortran	`mpif08 myprog.f90 -o myprog`
C	`mpicc myprog.c -o myprog` `mpiCC myprog.cc -o myprog`

! **Names of these wrappers are not standardized! The prefix mpi is very common, however, other prefixes and names are possible, e.g. `mpcc` for the IBM XL C compiler on AIX.** **!**

Programs must be started with the MPI start-up mechanism

Examples	`mpiexec [opts] my_app.exe` `mpirun [opts] my_app.exe`

! **Names of these start-up mechanisms are not standardized! The shown commands are very common, however, other are possible, e.g. `runjob` on Blue Gene/Q (e.g. JUQUEEN).** **!**

Language bindings

MPI 3.0 Standard			
C	ISO C	`#include <mpi.h>`	
Fortran	Fortran77	`include 'mpif.h'`	
	Fortran90	`use mpi`	
	Fortran08	`use mpi_f08`	

Throughout this course function calls and examples for **C** and **Fortran08** (with a few exceptions in Fortran77) are presented

Hints for Fortran programmers

The Fortran08 interface is not available in all MPI implementations, yet. To convert Fortran08 function calls into Fortran77/90 calls, consider the following:

- Replace all `type` declarations by `integer` declarations
- Omit all `intent` statements
- Check necessary `KIND` parameters and dimensions of variables (→ MPI standard)
- The `ierror` argument is **mandatory in 77/90!**

MPI 3.0

Chapter 17.1

MPI function format

Generic format of MPI functions

C	`error = MPI_Function(parameter,...);`
Fortran	`call MPI_Function(parameter,...,ierror)`

! **MPI namespace:**
`MPI_` **and** `PMPI_` **prefixes must not be used for user-defined functions or variables since they are used by MPI.** **!**

! **The** `ierror` **parameter is *optional* in Fortran08 but it is *mandatory* in Fortran77/90.** **!**

MPI standard

http://www.mpi-forum.org/docs

110 CHAPTER 4. DATATYPES

MPI_TYPE_COMMIT(datatype)

INOUT	datatype	datatype that is committed (handle)

General name and parameter description

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

C syntax

```
MPI_Type_commit(datatype, ierror)
    TYPE(MPI_Datatype), INTENT(INOUT) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

Fortran08 syntax

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

Fortran77/90 syntax

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been commit-
...

Descript. of funtion

MPI Terminology – Data types

Basic Data Types

Data types which are defined within the MPI standard

- Basic data types for Fortran and C are **different**
- Examples:

C	C type	MPI basic type
	`signed int`	`MPI_INT`
	`float`	`MPI_FLOAT`
	`char`	`MPI_CHAR`

Fortran	Fortran type	MPI basic type
	`INTEGER`	`MPI_INTEGER`
	`REAL`	`MPI_REAL`
	`CHARACTER`	`MPI_CHARACTER`

Derived Data Types

Data types which are constructed from basic (or derived) data types

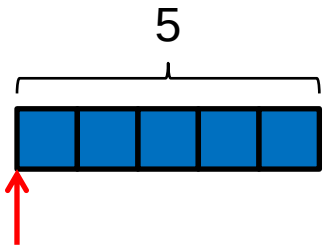
MPI Terminology – Messages

Message

A packet of data which needs to be exchanged between processes

Packet of data:

- An array of elements of an MPI data type (basic or derived data type)
- Described by
  - *Position in memory (address)*
  - *Number of elements*
  - *MPI data type*



Information for sending and receiving messages

- Source and destination process (ranks)
- Source and destination location
- Source and destination data type
- Source and destination data size

MPI terminology – Properties of procedures

Blocking

A procedure is blocking if return from the procedure indicates that the user is allowed to reuse resources specified in the call to the procedure.

Nonblocking

If a procedure is nonblocking it will return as soon as possible from to the calling process. However, the user is not allowed to reuse resources specified in the call to the procedure before the communication has been completed by an appropriate call at the calling process.

Examples

- Blocking 
- Nonblocking 

MPI terminology – Properties of procedures

Collective

A procedure is collective if all processes in a group need to invoke the procedure

Synchronous

A synchronized operation will complete successfully only if the (required) matching operation has started (send – receive).

Buffered (Asynchronous)

A buffered operation may complete successfully before a (required) matching operation has started (send – receive).

(Non-)Blocking – (A)Synchronous

(Non)-Blocking

- Statement about *reusability of message buffer*

(A)Synchronous

- Statement about *matching communication call*

Example

- Blocking, synchronous sending:
  - *Will return from call when buffer can be reused*
  - *After return receiving has started*
- Blocking, asynchronous sending:
  - *Will return from call when buffer can be reused*
  - *After return, receiving has not started necessarily, message may be buffered internally*

Basic program structure – Initialization

First call to an MPI function

C
```
int MPI_Init(int *argc, char ***argv)
```

Fortran
```
MPI_Init(ierror)
  integer, optional, intent(out) :: ierror
```

Exception (can appear before the above call)

C
```
int MPI_Initialized(int *flag)
```

Fortran
```
MPI_Initialized(flag, ierror)
  LOGICAL, INTENT(OUT) :: flag
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Basic program structure – Initialization

Last call to an MPI function

C
```
int MPI_Finalize(void)
```

Fortran
```
MPI_Finalize(ierror)
  integer, optional, intent(out) :: ierror
```

Exception (can appear after the above call)

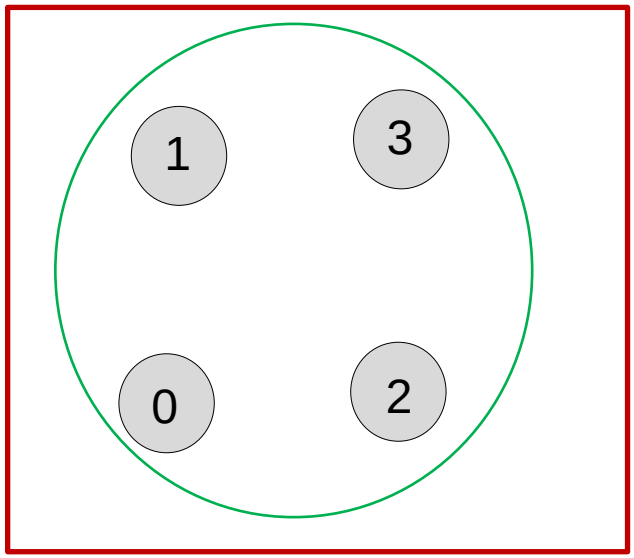
C
```
int MPI_Finalized(int *flag)
```

Fortran
```
MPI_Finalized(flag, ierror)
  LOGICAL, INTENT(OUT) :: flag
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Basic program structure – Communicator

```
mpiexec -np 4 my_application.exe
```

Default communicator available after initialization:



Handle:

```
MPI_COMM_WORLD
```

C	`MPI_Comm MPI_COMM_WORLD`
Fortran	`INTEGER :: MPI_COMM_WORLD`

Basic program structure – Communicator

Getting the total number of tasks

C

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

Fortran

```fortran
MPI_Comm_size(comm, size, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: size
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Example

C

```c
ierror = MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Fortran

```fortran
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
```

Basic program structure – Communicator

Getting the rank for each task

C

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

Fortran

```fortran
MPI_Comm_rank(comm, rank, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: rank
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Example

C

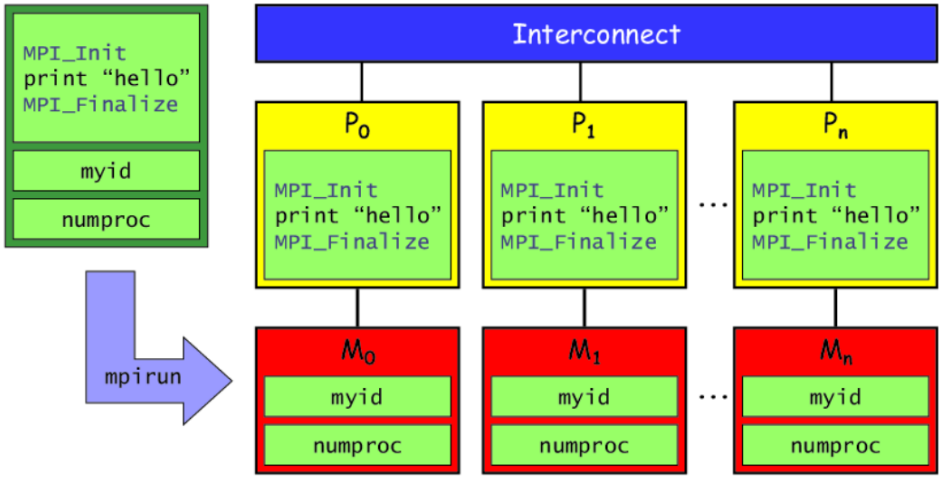
```c
ierror = MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

Fortran

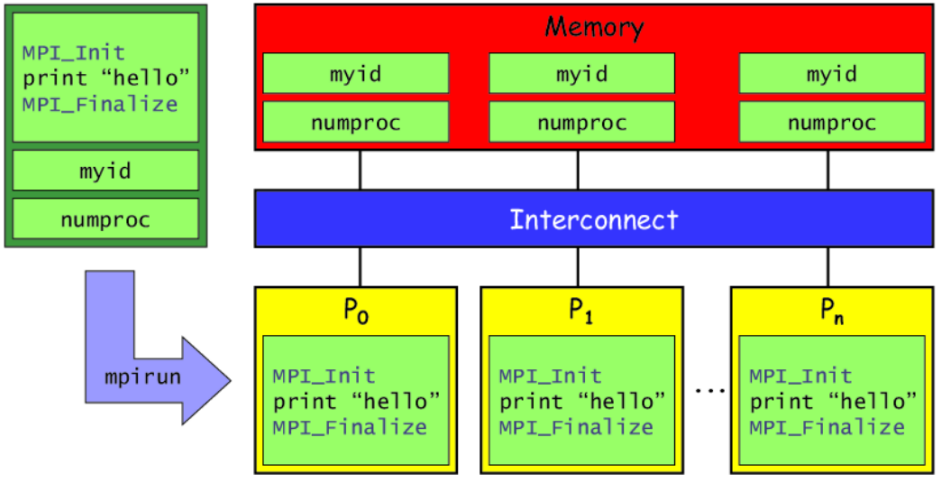
```fortran
call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierror)
```

MPI on shared and distributed memory systems

Distributed memory



Shared memory



JURECA – Login and compilation

Login

1. open a terminal on your PC
2. `ssh-add`
3. `ssh -X jureca`

Compilation

C++	`mpic++`
C	`mpicc`
Fortran	`mpif77` `mpif90`

Edit source code

Use editors on JURECA (login) node or remotely via **kate**:
`sftp://trainXXX@jureca.zam.kfa-juelich.de/homea/hpclab/trainXXX`

JURECA – running parallel jobs

Create an interactive session

- open a terminal and allocate a compute node:

```
ssh-add
ssh -X jureca
salloc --reservation=prace-kurs-batch-tue --nodes=1 --time=6:0:0
```

- wait for the prompt and open shell on compute node:

```
srun --forward-x --cpu_bind=none --pty /bin/bash –i
```

- Load environment (compiler, tools):

```
module load intel-para/2016a-mt Inspector/2016_update3
```

- start MPI applications with n tasks using:

```
mpiexec.hydra -np n <application>
```

Useful SLURM/Linux Commands on JURECA

SLURM	Command	Description
	`squeue-u <user id>`	Shows the status of jobs
	`scancel <job id>`	Aborts the job with the ID <job id>
	`scontrol show job <jobid>`	Show detailed information about a pending, running or recently completed job.

Linux	Command	Description
	`watch <command>`	Executes the command every 2 seconds, useful with → `showq` (cancel: ctrl+c)
	`top`	Shows all processes running on the current node with an update every 3 seconds (cancel: q)
	`<command> &`	Sends `<command>` to the background

Exercise

Exercise 1 – MPI introduction

1.1 Output of ranks

Write a program in C or Fortran running with 4 processes, where each process writes out its rank

```
I am rank 0
I am rank 1
I am rank 2
I am rank 3
```

1.2 Output of ranks and total number of processes

Extend the program in such a way, that rank 0 writes out the total number of processes

```
I am rank 0 and master of 4 tasks!
I am rank 1
I am rank 2
I am rank 3
```

1.3 Conditional output

Modify the program in such a way that only processes with odd ranks give output

```
I am rank 1
I am rank 3
```

Parallel Programming with MPI

Blocking Point-to-Point Communication

MPI terminology – Properties of procedures

Blocking

A procedure is blocking if return from the procedure indicates that the user is allowed to reuse resources specified in the call to the procedure.

Nonblocking

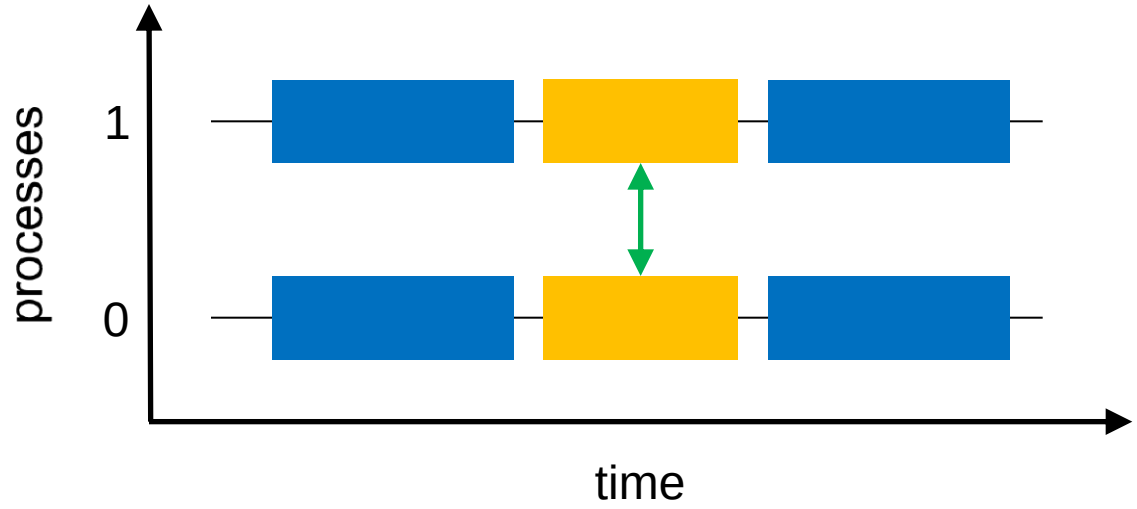
If a procedure is nonblocking it will return as soon as possible from to the calling process. However, the user is not allowed to reuse resources specified in the call to the procedure before the communication has been completed by an appropriate call at the calling process.

Examples

- Blocking
- Nonblocking

Blocking Communication

Computation interrupted by communication



Computation

Communication

Point-to-point communication

Properties of P2P communication

- Communication between two processes within the same communicator

! A process can send messages to itself !

- A *source* process sends a message to a destination process by a call to an *MPI send* routine
- A *destination* process needs to post a receive by a call to an *MPI receive* routine
- The destination process is specified by its rank in the communicator
- Every message sent with a point-to-point call, needs to be matched by a receive

Parts of messages

Message envelop

Contains information to distinguish messages

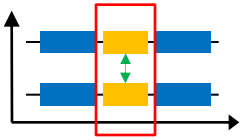
1. Source process: `SOURCE`
2. Destination process: `DEST`
3. A marker: `TAG`
4. The context of processes: `COMM`

Data part

Contains actual data to be sent/received and Needs three specifications

1. Initial address (send/receive buffer): `BUF`
2. Number of elements to be sent/received: `COUNT`
3. Datatype of the elements: `DATATYPE`

Sending messages



C

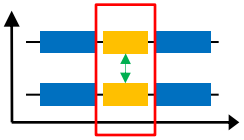
```c
int MPI_Send(const void* buf, int count,
             MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

Fortran

```fortran
MPI_Send(buf, count, datatype, dest, tag, comm,
         ierror)
  TYPE(*), DIMENSION(..), INTENT(IN) :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- `buf` is the address of the message to be sent, with `count` elements of type `datatype`
- `dest` is the rank of the destination process within the communicator `comm`
- `tag` is a marker used to distinguish different messages

Receiving messages



C

```c
int MPI_Recv(void* buf, int count,
             MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm,
             MPI_Status *status)
```

Fortran

```fortran
MPI_Recv(buf, count, datatype, source, tag, comm,
         status, ierror)
  TYPE(*), DIMENSION(..) :: buf
  INTEGER, INTENT(IN) :: count, source, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- `buf`, `count` and `datatype` refer to the receive buffer
- `source` is the rank of the sending source process within the communicator `comm` (can be `MPI_ANY_SOURCE`)
- `tag` is the marker of the message to be received (can be `MPI_ANY_TAG`)
- `status` (output) contains information about the received message

Getting envelop information

Envelop information about a received message are obtained from the `status` variable (except for `count`)

C
`status.MPI_SOURCE`
`status.MPI_TAG`
`status.MPI_ERROR`

Fortran			
Fortran08	`status%MPI_SOURCE`	Fortran77/90	`status(MPI_SOURCE)`
	`status%MPI_TAG`		`status(MPI_TAG)`
	`status%MPI_ERROR`		`status(MPI_ERROR)`

C

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

Fortran

```
MPI_Get_count(status, datatype, count, ierror)
  TYPE(MPI_Status), INTENT(IN) :: status
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, INTENT(OUT) :: count
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Probing messages

C

```c
int MPI_Probe(int source, int tag, MPI_Comm comm,
              MPI_Status *status)
```

Fortran

```fortran
MPI_Probe(source, tag, comm, status, ierror)
  INTEGER, INTENT(IN) :: source, tag
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Returns after matching message is ready to be received

- Query of communication envelope
- `status` (output) contains information about the message to be received
- `source` is the rank of the sending source process within the communicator `comm` (can be `MPI_ANY_SOURCE`)
- `tag` is a marker used to prescribe that only a message with the specified `tag` should be received (can be `MPI_ANY_TAG`)

Communication modes – Send modes

Synchronous send: MPI_Ssend

- Only completes when the receive has started

Buffered send: MPI_Bsend

- Always completes (unless an error occurs) irrespective of whether a receive has been posted or not
- Needs a user-defined buffer (→ `MPI_BUFFER_ATTACH`, **MPIS3.0, 3.6**)

Standard send: MPI_Send

- Either synchronous or buffered
- Uses an internal buffer

Ready send: MPI_Rsend

- Always completes (unless an error occurs) irrespective of whether a receive has been posted or not
- May be started only if the matching receive is already posted

Recommendations and hints – Send modes

Synchronous send `MPI_Ssend`
▪ High latency, good bandwidth ▪ Risk of idle times, serialization, deadlocks
Buffered send `MPI_Bsend`
▪ Low latency, low bandwidth
Standard send `MPI_Send`
▪ Minimal transfer time ▪ Can be implemented as synchronous or buffered send – do not assume either case
Ready send `MPI_Rsend`
▪ Only use if logic of your program permits it, i.e. guarantee that the receive is posted before the send

Communication modes – Receive modes

Only one receive call for all send modes

Receive: MPI_Recv

- Completes when a message has arrived
- Same routine for all communication modes

Wildcards can be used for receiving messages

- Receiving from any source: `MPI_ANY_SOURCE`
- Receiving message with any tag: `MPI_ANY_TAG`

The actual source and tag are returned in the `STATUS` variable

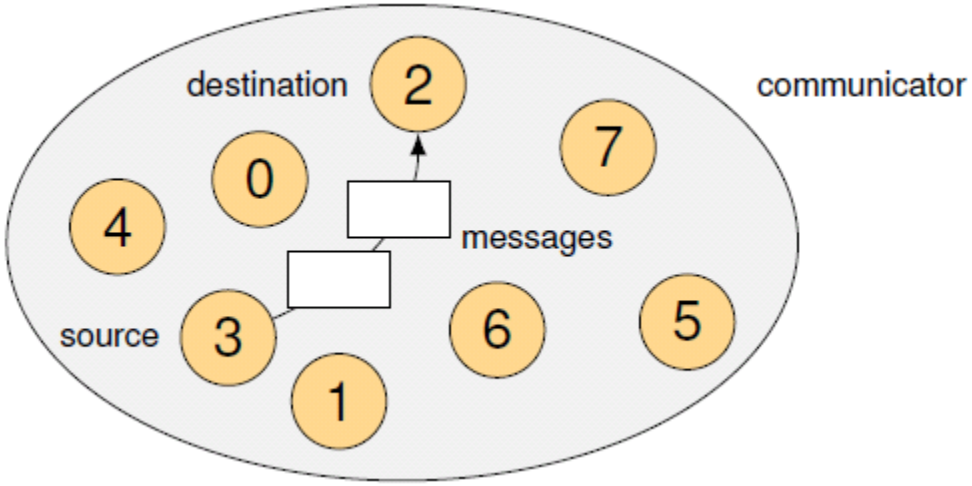
Point-to-point communication requirements

- Communicator must be the same
- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
  - `MPI_ANY_SOURCE` is also valid
- Tags must match
  - `MPI_ANY_TAG` is also valid
- Message datatypes must match
- Receive buffer must be large enough to hold the message
  - If it is not, behavior is undefined
  - Can be larger than the data received

Properties of MPI point-to-point communication

Message order preservation

- Messages sent from the same sender and which match the same `receive` call are received in the order they were sent
  - *Messages do not overtake each other if processes are single-threaded*



Properties of MPI point-to-point communication

Fairness

- Not guaranteed!
- It is the programmers responsibility to prevent starvation of send or receive operations

Resource limitations

- Any pending communication operation consumes system resources (e.g. buffer space) that are limited
- Errors may occur when a lack of resources prevents the execution of an MPI call

Pitfalls

Deadlock (standard send)

- Processes are waiting for sends or receives which can never be posted

```
…
Call MPI_Ssend(…,dest=my_right_neighbor,…)
Call MPI_Recv(…,source=my_left_neighbor,…)
…
```

- Do not have all processes sending or receiving at the same time with blocking calls
  - *Use special communication patterns*
  - *checked, odd-even, . . .*

Pitfalls

Performance penalties

- Late Receiver
  - *in synchronous mode, the sender waits for the receiver to post the receive call*
- Late Sender
  - *receiving process blocks in call until the sender starts to send the message*

Exercise

Exercise 2 – Blocking P2P communication

2.1 Ping-Pong

Write a ping-pong program

- Should run with 2 MPI processes with the following loop
  - Rank 0 sends a message (its rank) to rank1 (tag: 0) (`MPI_Ssend`)
  - After rank 1 receives the message it sends a message (its rank) to rank 0 (tag: 1) (`MPI_Recv`)
- The loop should be repeated 100 times
- The program should abort if it is run with the wrong number of MPI ranks and should issue a corresponding error message

2.2 Ping-Pong - MPI_Sendrecv

Use `MPI_Sendrecv` calls for the ping-pong program **(MPIS3.0, 3.10)**

2.3 Ping-Pong - Benchmarking

Extend the program

- Measure the time needed for the loop with 1 mio iterations
- See function `MPI_WTIME` **(MPIS3.0, 8.6)**
- Rank 0 should print the time needed for the loop

Parallel Programming with MPI

Nonblocking Point-to-Point Communication

MPI terminology – Properties of procedures

Blocking

A procedure is blocking if return from the procedure indicates that the user is allowed to reuse resources specified in the call to the procedure.

Nonblocking

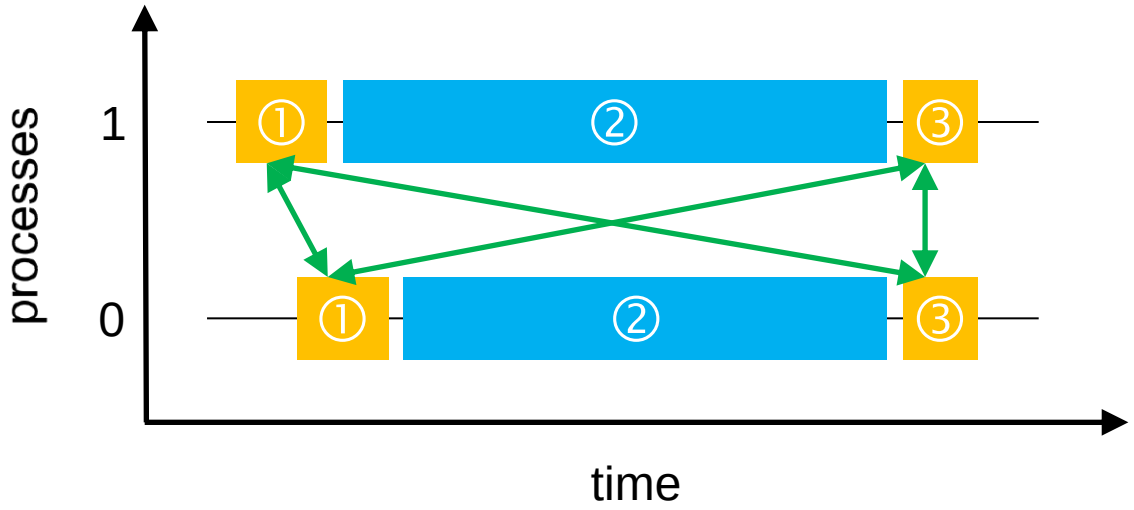
If a procedure is nonblocking it will return as soon as possible from to the calling process. However, the user is not allowed to reuse resources specified in the call to the procedure before the communication has been completed by an appropriate call at the calling process.

Examples

- Blocking
- Nonblocking

Nonblocking Communication

Solution for many pitfalls in blocking communication



Other work

Communication

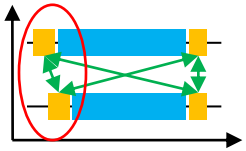
① Initialization of communication
② Attending other work/test for completion
③ Completion of communication

Nonblocking Communication

Goal: overlapping communication with other work

1. Initiate communication
   - Routines: `MPI_I…` (‘I’ for ‘immediate)
   - Nonblocking routines return before the communication has completed
   - Nonblocking routines have the same arguments as their blocking counterparts except for an extra `request` argument
2. User-application can attend other work
   - *Communication*, computation, . . .
3. Complete communication
   - Waiting for the communication request to finish

Phase ① – Communication modes



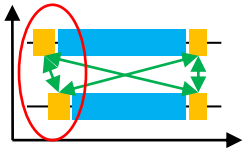
Send modes

- Synchronous send: `MPI_Issend`
- Buffered send: `MPI_Ibsend`
- Standard send: `MPI_Isend`
- Ready send: `MPI_Irsend`

Receive all modes

- Receive: `MPI_Irecv`
- Probe: `MPI_Iprobe`

Nonblocking synchronous send



C

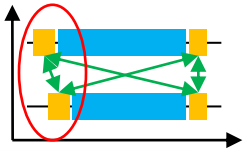
```c
int MPI_Issend(const void* buf, int count,
               MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm,
               MPI_Request *request)
```

Fortran

```fortran
MPI_Issend(buf, count, datatype, dest, tag, comm,
           request, ierror)
  TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
  INTEGER, INTENT(IN) :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Nonblocking send routines have the *same* arguments as their blocking counterparts except for the extra `request` argument
- Send buffer `buf` must not be accessed before the send has been successfully tested for completion with `MPI_WAIT` or `MPI_TEST`

Nonblocking receive



C

```c
int MPI_Irecv(void* buf, int count,
              MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm,
              MPI_Request *request)
```

Fortran

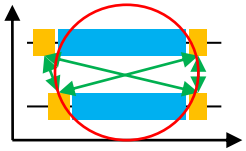
```fortran
MPI_Irecv(buf, count, datatype, source, tag, comm,
          request, ierror)
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
  INTEGER, INTENT(IN) :: count, source, tag
  TYPE(MPI_Datatype), INTENT(IN) :: datatype  TYPE(MPI_Comm),
  INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Nonblocking receive routines have the *same* arguments as their blocking counterparts except for the extra `request` argument
- Send buffer `buf` must not be accessed before the send has been successfully tested for completion with `MPI_WAIT` or `MPI_TEST`

The `request` handle

- Used for nonblocking communication
- Request handles must be stored in a variable of sufficient scope
  - C/C++ : `MPI_Request`
  - Fortran08: `TYPE(MPI_Request)`
  - Fortran77/90: `INTEGER`
- A nonblocking communication routine returns a value for the request handle
- This value is used by `MPI_WAIT` or `MPI_TEST` to test when the communication has completed
- If the communication has completed the request handle is set to `MPI_REQUEST_NULL`

Phase ② – Test



C

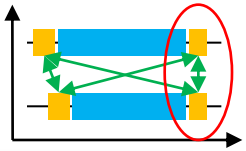
```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
```

Fortran

```
MPI_Test(request, flag, status, ierror)
  TYPE(MPI_Request), INTENT(INOUT) :: request
  LOGICAL, INTENT(OUT) :: flag
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- If communication associated with `request` is complete call returns `flag=true`, otherwise `flag=false` (nonblocking call)
- If several communications are pending **(MPIS3.0, 3.7.5)**
  - `MPI_Testall`
  - `MPI_Testsome`
  - `MPI_Testany`

Phase ③ – Wait



C

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

Fortran

```fortran
MPI_Wait(request, status, ierror)
  TYPE(MPI_Request), INTENT(INOUT) :: request
  TYPE(MPI_Status) :: status
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- Waits until communication associated with `request` is completed (call is blocking)
- If several communications are pending **(MPIS3.0, 3.7.5)**
  - `MPI_Waitall`
  - `MPI_Waitsome`
  - `MPI_Waitany`

Blocking vs. nonblocking operations

- A blocking send can be used with a nonblocking receive, and vice versa
- Nonblocking sends can use any mode, just like the blocking counterparts
  - Synchronous mode refers to the completion of the send (tested with `MPI_Wait/MPI_Test`), not to the initiation (`MPI_Issend` returns immediately!)
- A nonblocking operation immediately followed by a matching `MPI_Wait` is equivalent to the blocking operation
- Hints for Fortran: **MPIS3.0, 17.1, 17.2, A.3, A.4**

Overlapping communication

- Main goal is to overlap *communication* with *<u>communication</u>*
- Overlap with computation
  - Progress may only be done inside of MPI calls
  - Not all platforms perform significantly better than well placed blocking communication
  - If hardware support is present, application performance may significantly improve due to overlap

- General recommendation
  - Initiation of communication should be placed as early as possible
  - Synchronization/completion should be placed as late as possible

Exercise

Exercise 3 – Nonlocking P2P communication

3.1 Communication on a ring – blocking send, nonblocking receive

Write a program where all ranks calculate the sum of all rank numbers

- All ranks are arranged on a ring
- Each rank receives a partial sum from its left neighbor, adds its own rank number to the sum and passes the results to its right neighbor until all rank numbers have been summed up
- Use `MPI_Issend` and `MPI_Recv`

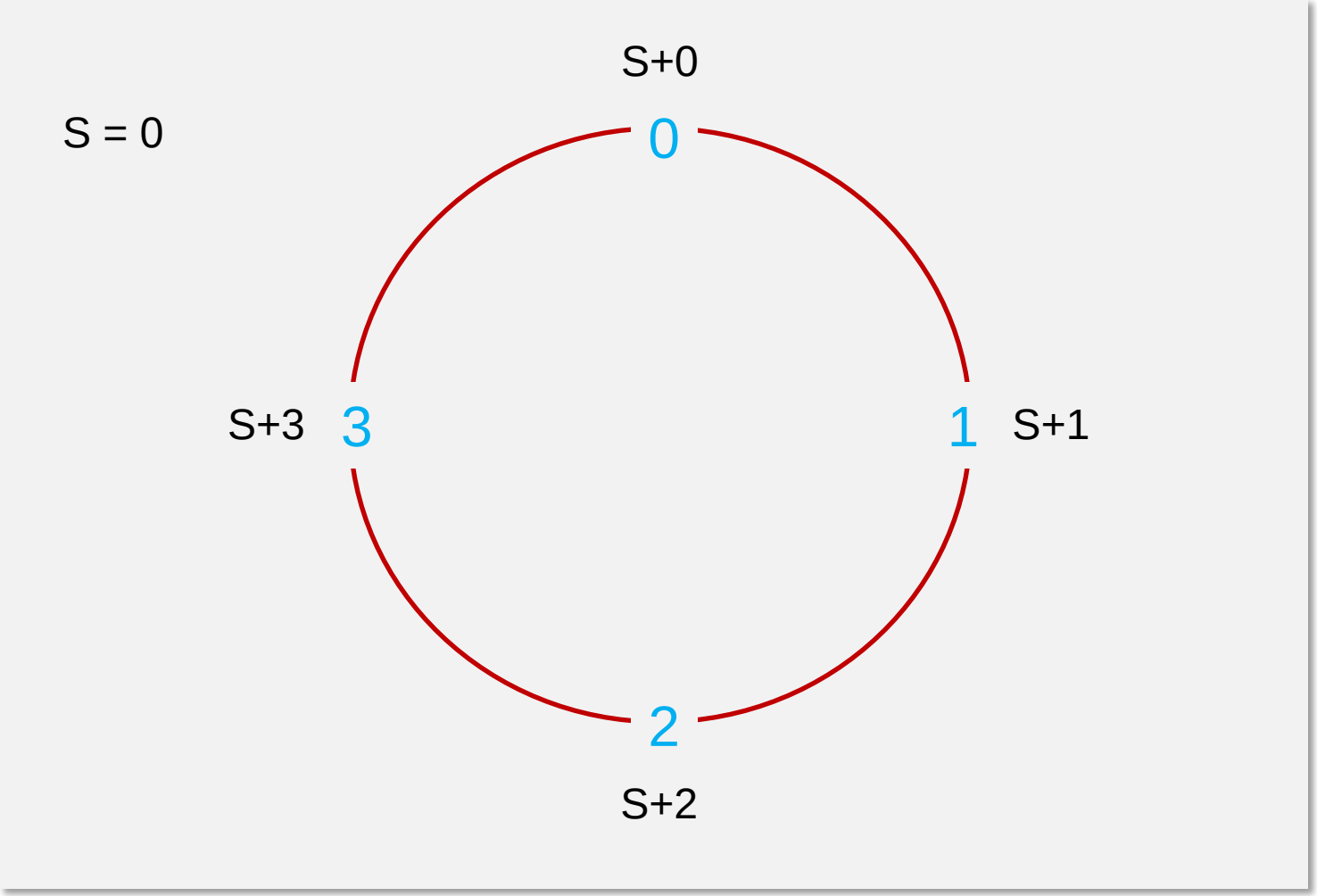
3.2 Communication on a ring – nonblocking send, blocking receive

Use `MPI_Ssend` and `MPI_Irecv`

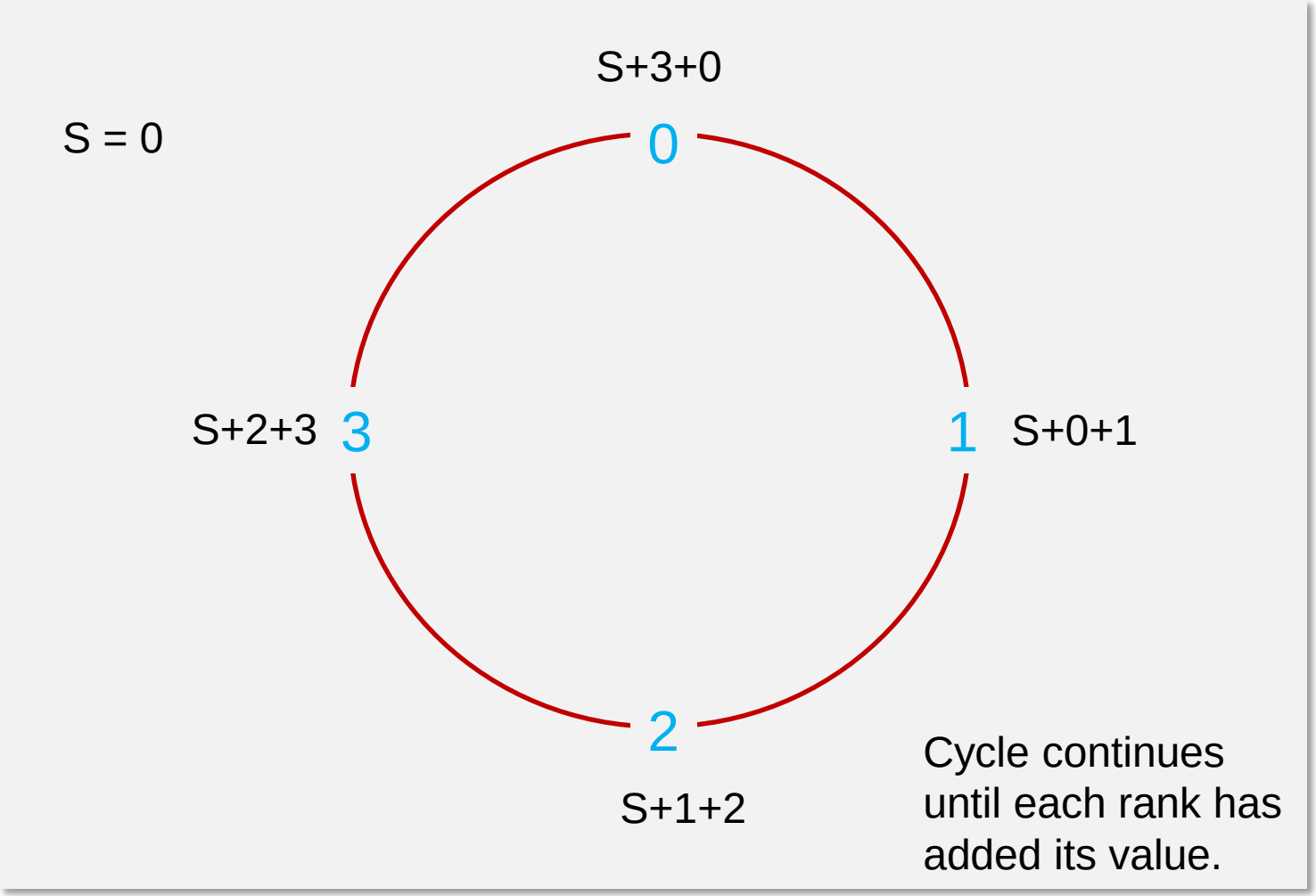
3.3 Question

What would happen if you would use `MPI_Ssend` and `MPI_Recv`?

Exercise – Communication on a ring



Exercise – Communication on a ring



Parallel Programming with MPI

Blocking Collective Communication

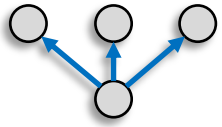
Characteristics of collective communication

- Collective action over a communicator.
  - All processes of the communicator must communicate, i.e. all processes must call the collective routine.
- Synchronization may or may not occur
- All collective operations are blocking
- No tags are used
- Receive buffers must have exactly the same size as send buffers

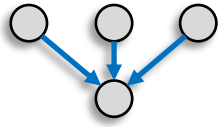
Collective communication – overview

Blocking collectives

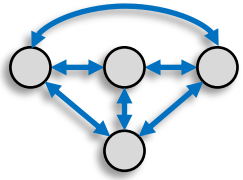
- One-to-all
  - `MPI_Bcast, MPI_Scatter[v]`



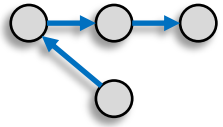
- All-to-one
  - `MPI_Gather[v], MPI_Reduce`



- All-to-all
  - `MPI_Allgather[v],MPI_All_to_all[v,w]`
    `MPI_Allreduce,`
    `MPI_Reduce_scatter`



- Other
  - `MPI_Scan, MPI_Exscan`



Barrier synchronization

C
```c
int MPI_Barrier(MPI_Comm comm)
```

Fortran
```fortran
MPI_Barrier(comm, ierror)
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Explicit synchronization between processes

- A process cannot leave the function call before all participating processes have entered the function
- Global synchronization always includes inter-process communication
- In general not needed (exceptions: profiling, debugging)

Broadcast

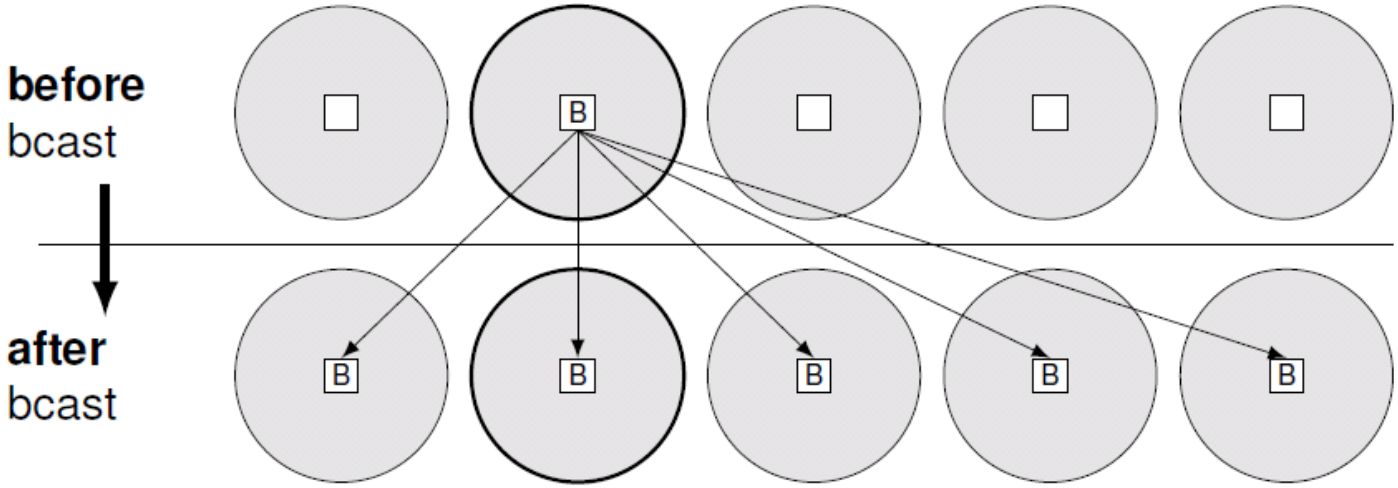
C

```
int MPI_Bcast(void* buffer, int count,
              MPI_Datatype datatype, int root,
              MPI_Comm comm)
```

Fortran

```
MPI_Bcast(buffer, count, datatype, root, comm,ierror)
```

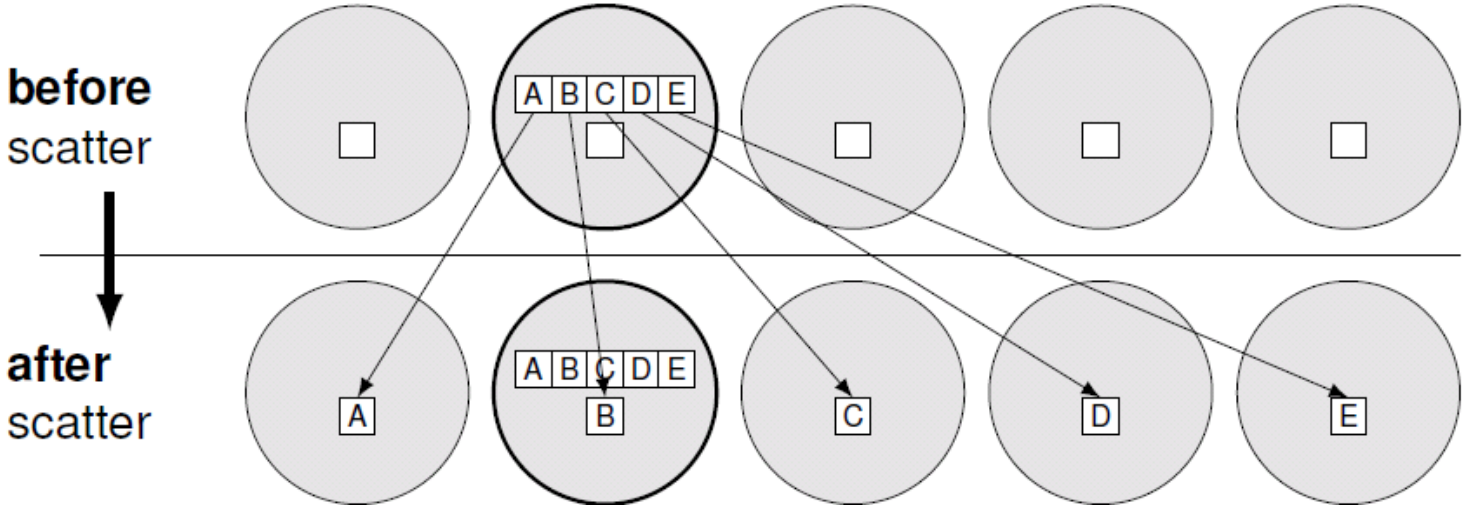
```
TYPE(*), DIMENSION(..) :: buffer
INTEGER, INTENT(IN) :: count, root
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```



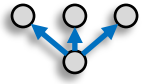
Scatter

C

```
int MPI_Scatter(const void* sendbuf, int sendcount,
                MPI_Datatype sendtype,void* recvbuf,
                int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```



Scatter



Fortran

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, root, comm, ierror)
```

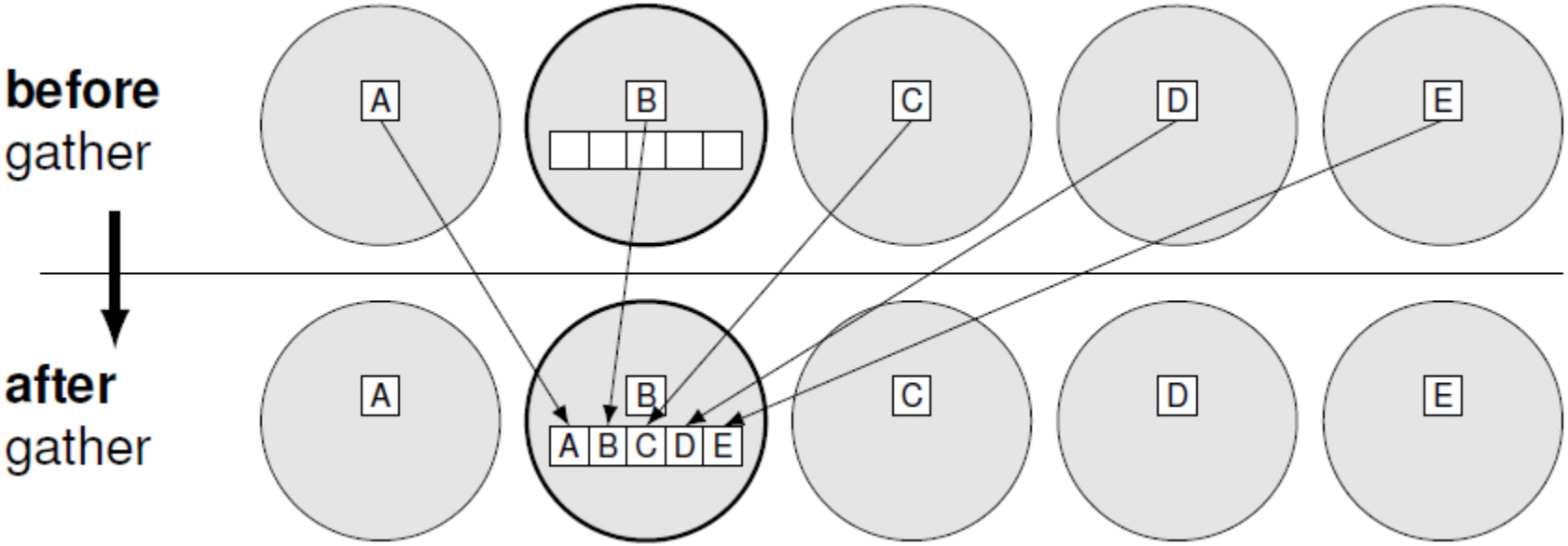
```
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcount, root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- If `MPI_IN_PLACE` is used for `recvbuf` on the root process `recvcount` and `recvtype` are ignored on the root process and the root process will not send data to itself
- All tasks have different data after completion of the call

Gather

C

```
int MPI_Gather(const void* sendbuf, int sendcount,
               MPI_Datatype sendtype, void* recvbuf,
               int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```



Gather

Fortran

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,
           recvcount, recvtype, root, comm, ierror)
```

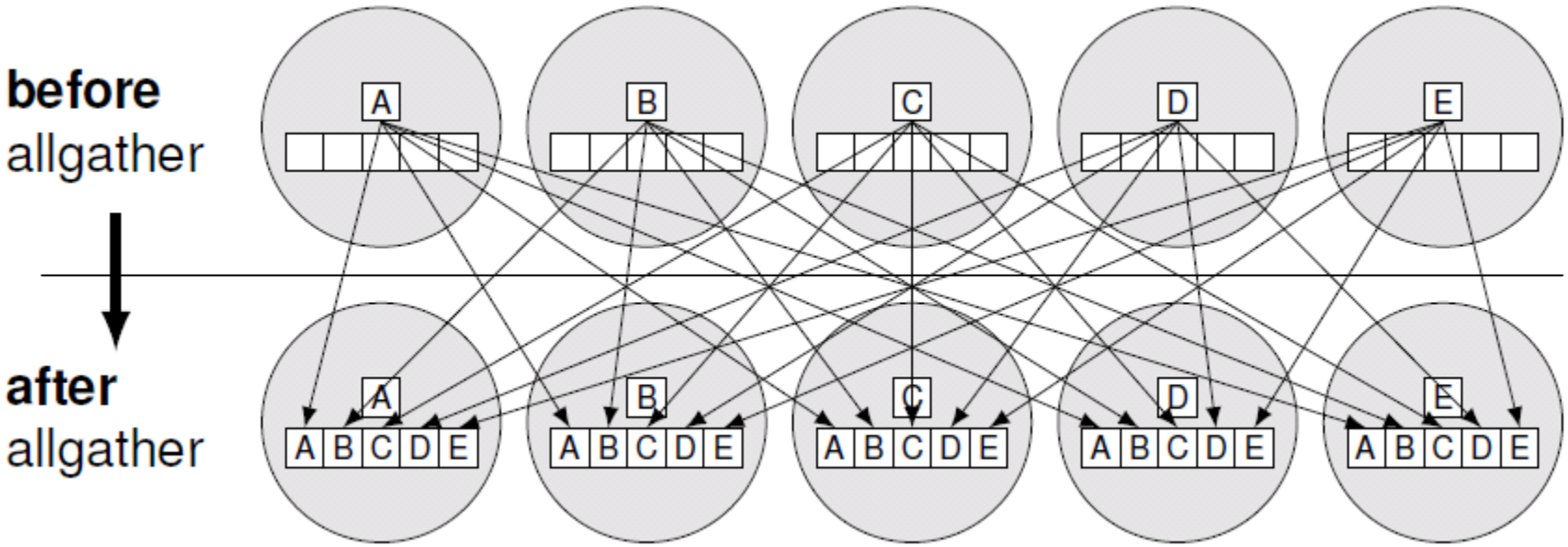
```
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcount, root
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- If `MPI_IN_PLACE` is used for `sendbuf` on the root process `sendcount` and `sendtype` are ignored on the root process and the root process will not send data to itself

Gather-to-all

C

```
int MPI_Allgather(const void* sendbuf, int sendcount,
                  MPI_Datatype sendtype, void* recvbuf,
                  int recvcount,MPI_Datatype recvtype,
                  MPI_Comm comm)
```



Gather-to-all

Fortran

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf,
              recvcount, recvtype, comm, ierror)
```

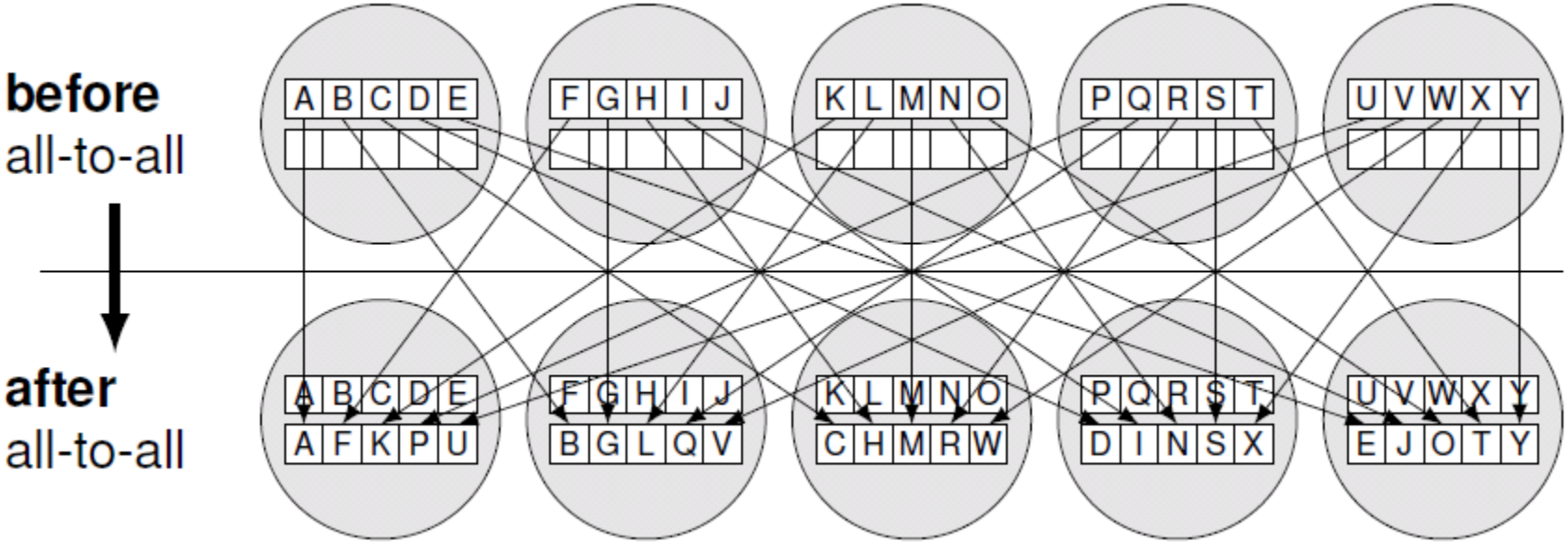
```
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- If `MPI_IN_PLACE` is used for `sendbuf` on all processes `sendcount` and `sendtype` are ingored and no process is sending data to itself
- All tasks have the same data after completion of the call

All-to-all

C

```
int MPI_Alltoall(const void* sendbuf, int sendcount,
                 MPI_Datatype sendtype, void* recvbuf,
                 int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)
```



All-to-all

Fortran

```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, comm, ierror)
```

```
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: sendcount, recvcount
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- If `MPI_IN_PLACE` is used for `sendbuf` on all processes `sendcount` and `sendtype` are ignored and no process is sending data to itself
- All tasks have different data after completion of the call

Global reduction operators

Associative operation over distributed data

- $d_0 \circ d_1 \circ d_2 \circ \ldots \circ d_{n-1}$
- d_i Data of process with rank i
- $\circ$ Associative operation

Examples

- Global sum or product
- Global maximum or minimum
- Global user-defined operation

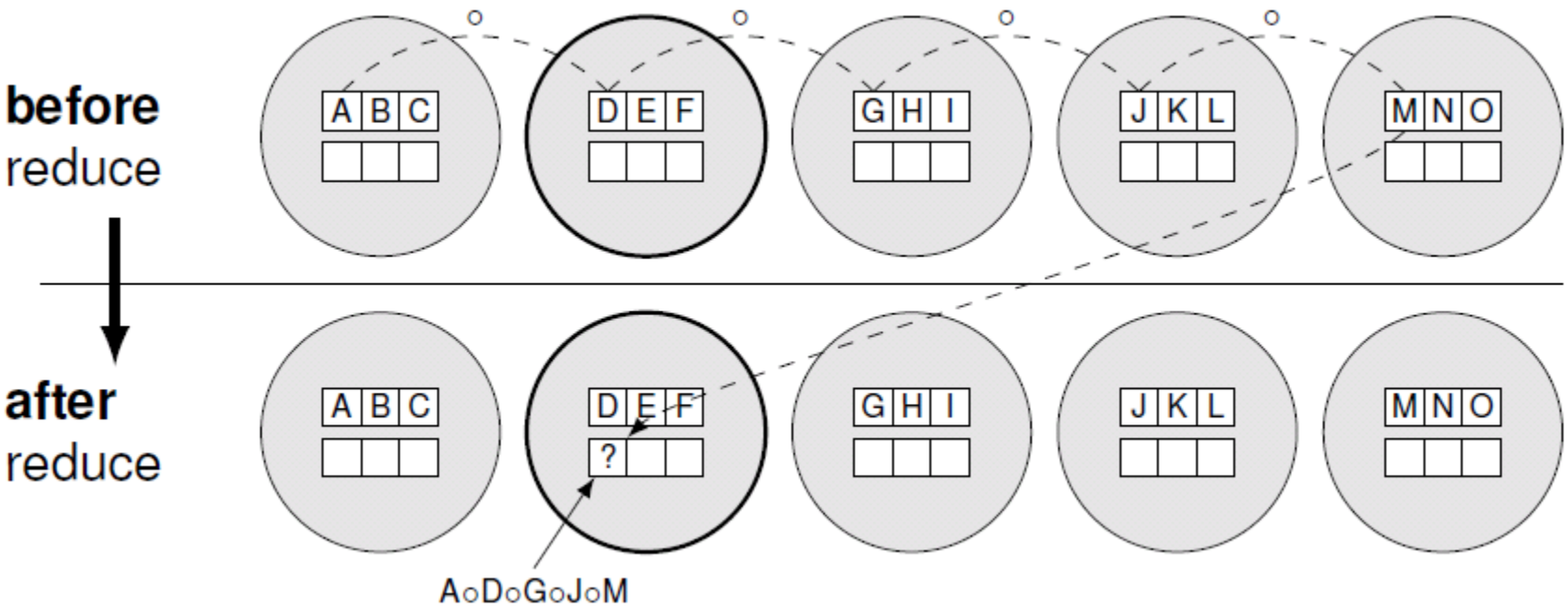
Order in which sub-reductions are performed is not defined

- Floating point rounding may depend on associativity
- Behavior may be non-deterministic

Reduce

C

```
int MPI_Reduce(const void* sendbuf, void* recvbuf,
               int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm)
```



Reduce

Fortran

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root,
           comm, ierror)
```

```
TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
TYPE(*), DIMENSION(..) :: recvbuf
INTEGER, INTENT(IN) :: count, root
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Op), INTENT(IN) :: op
TYPE(MPI_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

- If `MPI_IN_PLACE` is used for `sendbuf` on the root process; in this case input data at the root process is taken from `recvbuf`, which will be replaced by the result
- `recvbuf` only meaningful at the root process

Predefined reduction operation hand

Operation handle	Function/Result
MPI_MAX	Max. of all values
MPI_MIN	Min. of all values
MPI_SUM	Sum of all values
MPI_PROD	Product of all values
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Max. and its location **MPIS3.0, 5.9.4**
MPI_MINLOC	Min. and its location **MPIS3.0, 5.9.4**

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

Variants of collective communications routines

Routines with more flexibility (**MPIS3.0, 5.5 – 5.8**)

- `MPI_Scatterv`, `MPI_Allgatherv`, `MPI_Alltoallv`, `MPI_Gatherv`, `MPI_Alltoallw`

Extended/combined functionality (**MPIS3.0, 5.9-5.11**)

- `MPI_Allreduce`, `MPI_Reduce_scatter`
- `MPI_Scan, MPI_Exscan`

Exercise

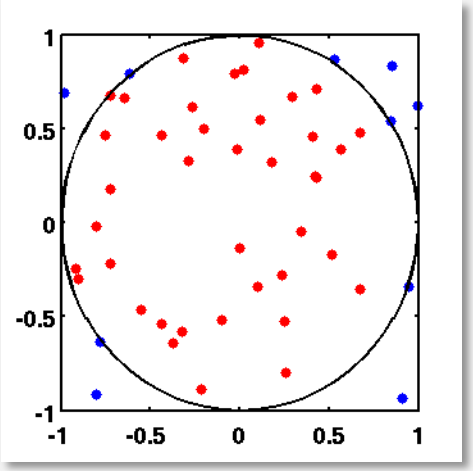
Exercise 4 – Collective communication

4.1 Calculation of π

Write a program that calculates π using the Monte Carlo Method

- The program should print the number of tasks it used, the difference to the exact value of π and the time needed for the calculation in sec.

Hints



$$\left.\begin{aligned} A_{\text{circle}} &= \pi r^2 \\ A_{\text{square}} &= 4r^2 \end{aligned}\right\} \quad \pi = 4\frac{A_{\text{circle}}}{A_{\text{square}}}$$

Monte Carlo:

- "Throwing randomly darts"
- Count hits in circle (n_{circle}) and total hits (n_{square})

$$A_{\text{circle}} \sim n_{\text{circle}}$$

$$A_{\text{square}} \sim n_{\text{square}}$$

Exercise

Random number generator

C

```c
/* Include the math header file */
  #include <math.h>
  #include <stdlib.h>
/* Initialize random number generator */
  srand(int seed);
/* Getting a random number */
  x =  int rand();
/* Quering the largest possible number */
  int x_max = RAND_MAX;
```

Fortran

```fortran
! Initialize random number generator
  real    :: x
  integer :: seed(1)
  call random_seed(PUT=seed)
! Get a random number 0<x<=1
  call random_number(x)
```

Reference value: π = 3.1415926535897932 38

Exercise

Exercise 4 – Collective communication

4.2 Calculation of sum of squares of all ranks

Write a program that calculates the sum of squares of all ranks.
Assume each process knows only its own rank

- Each rank should print this sum
- Only rank 0 should calculate the sum

4.3 Modification

Modify the above program so that

- Each rank should print this sum
- All processes calculate the sum

Again assume that each process knows only its own rank.

Parallel Programming with MPI

Nonblocking Collective Communication

MPI terminology – Properties of procedures

Blocking

A procedure is blocking if return from the procedure indicates that the user is allowed to reuse resources specified in the call to the procedure.

Nonblocking

If a procedure is nonblocking it will return as soon as possible from to the calling process. However, the user is not allowed to reuse resources specified in the call to the procedure before the communication has been completed by an appropriate call at the calling process.

Examples

- Blocking
- Nonblocking

Properties

Properties similar to nonblocking *point-to-point* communication

1. Initiate communication
   - Routines: `MPI_I…` ('I' for 'immediate)
   - Nonblocking routines return before the communication has completed
   - Nonblocking routines have the same arguments as their blocking counterparts except for an extra request argument
2. User-application can attend other work
   - *Communication*, computation . . .
3. Complete communication
   - Waiting for the communication request to finish

Same testing and completion routines (`MPI_Test`, `MPI_Wait`, …)

! ***Nonblocking*** **collective operations cannot be matched with** ***blocking*** **collective operations** **!**

Example: Bcast

Blocking routine

C

```
int MPI_Bcast(void *buffer, int count,
              MPI_Datatype datatype, int root,
              MPI_Comm comm)
```

Nonblocking routine

C

```
int MPI_Ibcast(void *buffer, int count,
               MPI_Datatype datatype, int root,
               MPI_Comm comm, MPI_Request *request)
```

Example: Bcast

Blocking routine

Fortran

```
MPI_Bcast(buffer,count,datatype,root,comm,ierror)

  TYPE(*), DIMENSION(..) :: buffer
  INTEGER, INTENT(IN) :: count, root
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Nonblocking routine

Fortran

```
MPI_Ibcast(buffer,count,datatype,root,comm,request,ierror)

  TYPE(*), DIMENSION(..) :: buffer
  INTEGER, INTENT(IN) :: count, root
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  TYPE(MPI_Request), INTENT(OUT) :: request
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Parallel Programming with MPI

Derived Datatypes

Motivation

With MPI communication calls only multiple consecutive elements of the same type can be sent.

Buffers may be non-contiguous in memory

- Sending only the real/imaginary part of a buffer of complex doubles
- Sending sub-blocks of matrices

Buffers may be of mixed type

- User defined data structures

```
struct buff_layout {
  int i[4];
  double d[5];
} buffer;
```

Solutions without MPI derived datatypes

Non-contiguous data of a single type

- Consecutive MPI calls to send and receive each element in turn
  - *Additional latency costs due to multiple calls*
- Copy data to a single buffer before sending it
  - *Additional latency costs due to memory copy*

Contiguous data of mixed types

- Consecutive MPI calls to send and receive each element in turn
  - *Additional latency costs due to multiple calls*

Derived datatypes

- General MPI datatypes describe a buffer layout in memory by specifying
  - *A sequence of basic datatypes*
  - *A sequence of integer (byte) displacements*
- Derived datatypes are derived from basic datatypes using constructors
- MPI datatypes are referenced by an opaque handle

! **MPI datatypes are opaque objects! Using the `sizeof()` operator on an MPI datatype handle will return the size *of the handle*, *not* the size *of an MPI datatype*** **!**

Example: Struct data

C

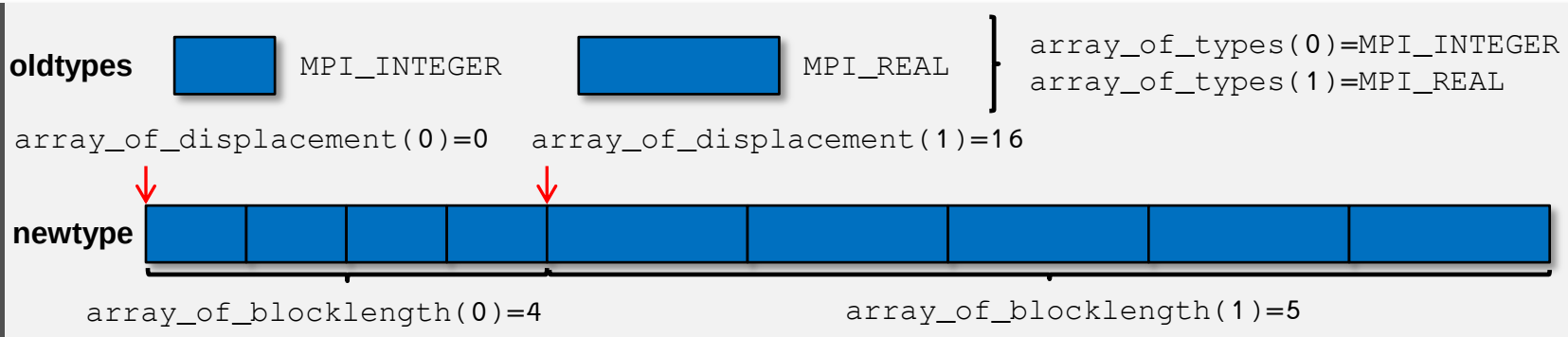
```c
int MPI_Type_create_struct(int count, const int array_of_blocklengths[],
                           const MPI_Aint array_of_displacements[],
                           const MPI_Datatype array_of_types[],
                           MPI_Datatype *newtype)
```

Fortran

```fortran
MPI_Type_create_struct(count, array_of_blocklengths, array_of_displacements,
                       array_of_types, newtype, ierror)
```

```fortran
  INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
  INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
  array_of_displacements(count)
  TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
  TYPE(MPI_Datatype), INTENT(OUT) :: newtype
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Example



Committing and freeing derived datatypes

Before using a derived datatype it needs to be committed

C

```
int MPI_Type_commit(MPI_Datatype *newtype)
```

Fortran

```
MPI_Type_commit(datatype, ierror)

  TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Marking derived datatypes for deallocation

C

```
int MPI_Type_free(MPI_Datatype *datatype)
```

Fortran

```
MPI_Type_free(datatype, ierror)

  TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Usage of MPI basic/derived datatypes

- Used to define a memory layout in MPI routines
- Never used like variable declarations

1. Generate a suitable derived datatype

2. Issue the corresponding MPI routine, for example, for sending the data

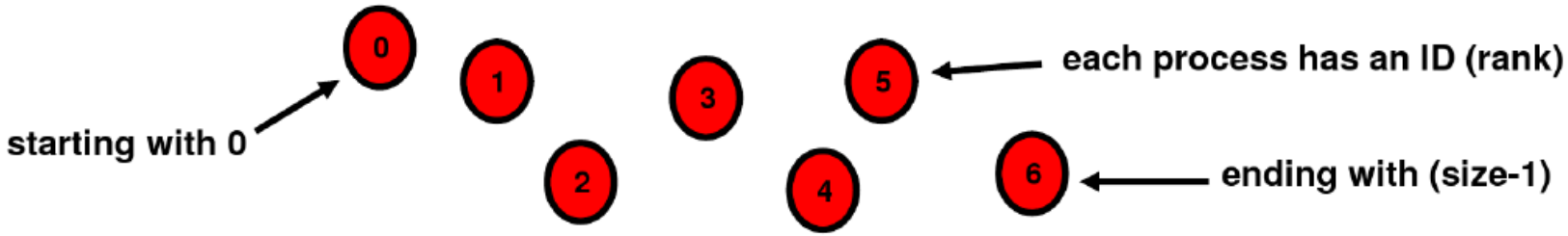
```
MPI_Send(&buf, 1, newtype, 1, 1, MPI_COMM_WORLD)
```

Parallel Programming with MPI

Communicators

MPI Groups

- A group is an ordered set of processes
- A Process can belong to one or more groups
- Each process has a distinct rank (starting with 0) within a group
- A group is a dynamic object and can be created and deleted within the program
- Predefined groups:
  - `MPI_GROUP_EMPTY` (empty group)
  - `MPI_GROUP_NULL` (used to declare groups invalid)



MPI Groups

- Available functions to handle MPI groups:
  - `MPI_Group_size`

    get size of group
  - `MPI_Group_translate_ranks`

    map ranks from one group to another
  - `MPI_Group_compare`

    possible results: `MPI_IDENT, MPI_SIMILAR, MPI_UNEQUAL`
  - `MPI_Group_union`

    combine two groups into a new one
  - `MPI_Group_incl, MPI_Group_excl`

    create a new group by explicitly including or excluding ranks from an existing group
  - `MPI_Group_comm`

    provides the group belonging to a communicator

MPI Communicators

- Constists of a group and a context
  - Context defines the communicators properties
- All kinds of MPI communication are based on communicators
- Motivation to use different communicators:
  - parallel libraries

- Predefined communicators:
  - `MPI_COMM_WORLD` (includes all MPI processes)
  - `MPI_COMM_SELF` (local process only)
  - `MPI_COMM_NULL` (used to declare communicators invalid)

MPI Communicators

- Available functions to handle MPI communicators:
  - `MPI_Comm_size`

    get size of communicator
  - `MPI_Comm_rank`

    get process rank within a certain communicator
  - `MPI_Comm_dup`

    duplicate communicator
  - `MPI_Comm_create`

    create communcator based on a group
  - `MPI_Comm_split`

    create disjunct subgroups
  - `MPI_Comm_free`

    deallocate communicator (set handle to `MPI_COMM_NULL`)

MPI Communicators

- Example: `MPI_Comm_split`

Rang	0	1	2	3	4	5	6	7	8	9
Prozess	A	B	C	D	E	F	G	H	I	J
color	0	ND	3	0	3	0	0	5	3	ND
key	3	1	2	5	1	1	1	2	1	0

ND = `MPI_UNDEFINED`

- Will result in new groups:
  - **F, G, A, D**
  - **E, I, C**
  - **H**
- Result for B and J will be `MPI_COMM_NULL`

Parallel Programming with MPI

Virtual Topologies

Virtual topologies

- <u>Goals</u>
  - Efficient programming, simplify source code
  - Make use of the underlaying hardware (e.g. network topology)
- MPI supports process topologies based on graphs or cartesian topologies
- Topology can be virtual or correspond to the physical hardware topology
- Process enumeration should be aligned with communication patterns
- Communication possible outside of the topology

Virtual topologies

- Create cartesian topology

C

```
int MPI_Cart_create ( MPI_Comm comm_old, int ndims, const
                      int* dims, const int* periods, int
                      reorder, MPI_Comm* comm_cart)
```

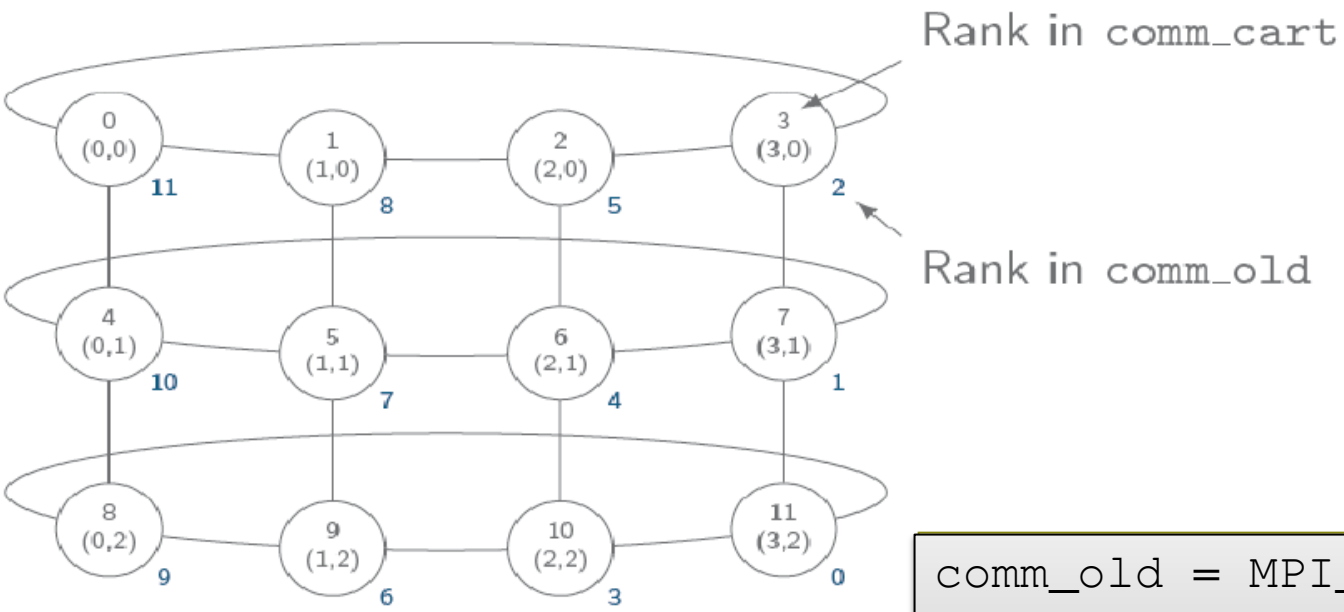
Fortran

```
MPI_CART_CREATE ( comm_old, ndims, dims, periods, reorder,
                  comm_cart, ierror ierror)
```

```
INTEGER :: comm_old, ndims, dims(*), comm_cart, ierror
LOGICAL :: periods(*), reorder
```

Virtual topologies

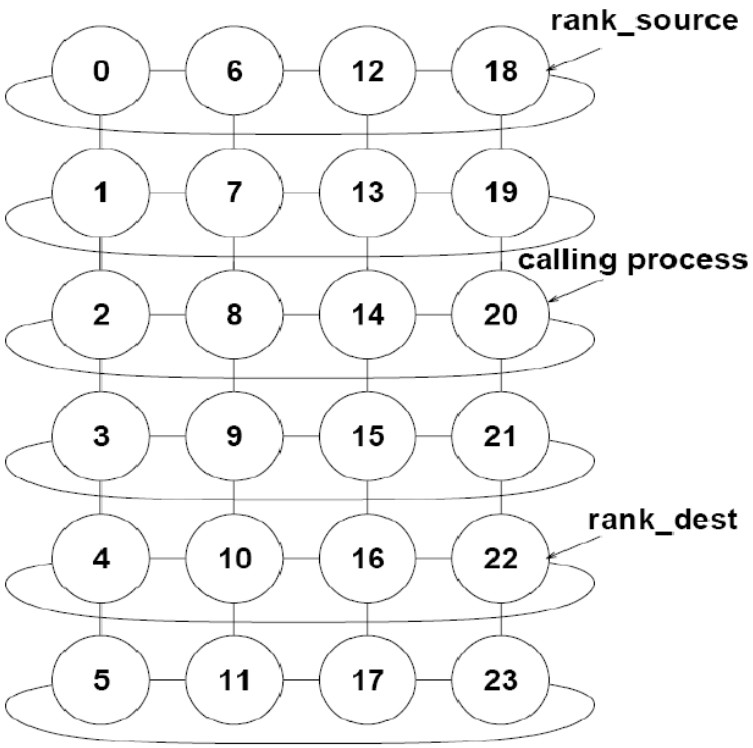
- Create cartesian topology




```
comm_old = MPI_COMM_WORLD
ndims = 2
dims = (4 , 3)
reriods = (true, false)
reorder = true
```

Virtual topologies

- Make use of cartesian topology




```
MPI_Cart_shift
```

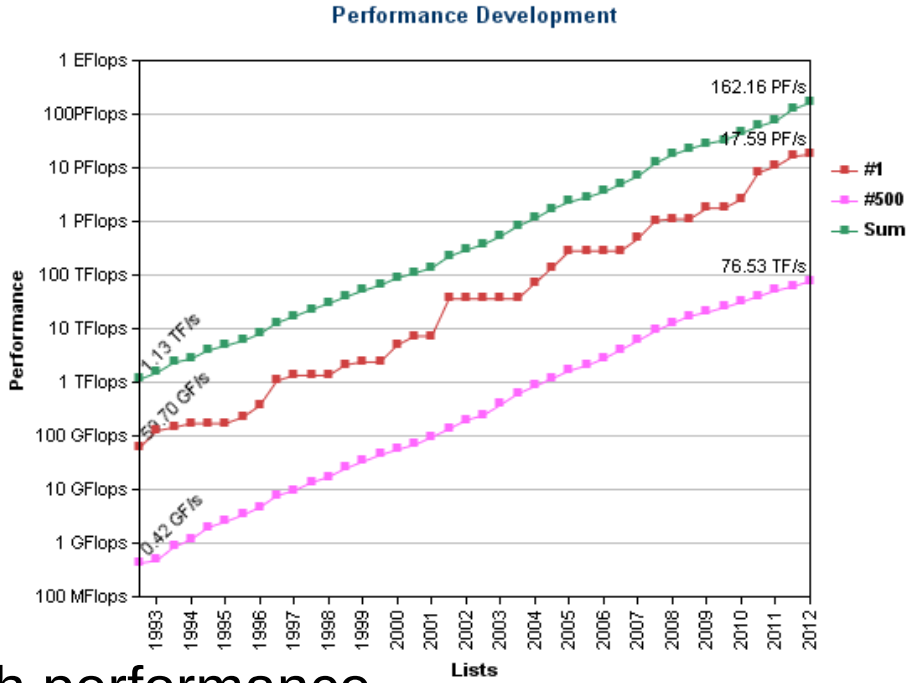
- direction = 0
- disp = 2
- rank_source = 18
- rank_dest = 22

Parallel Programming with MPI

Basics of I/O in HPC

Why to think about I/O in HPC?

- Fast growth of parallelism of HPC systems
  - Multi-core CPUs
  - GPUs
- Growth of problem sizes



Efficient usage of resources:

PERFORMANCE

- ► Distribution of data
- ► Parallel data access for high performance
- ► Data interface → managability of data

I/O Basics – Data storage

Area for meta data (number of blocks, modification date, etc.)

Meta data for file

Free file system block

File data

- Exclusive access to file system blocks
- Bandwidth → parallel file systems, multiple disks

Common I/O strategies

1. One process performs I/O
2. All or several processes write to one file
3. Each process writes to its own file (task-local files)

1. One process performs I/O

+ Simple to implement

– I/O bandwidth is limited to the rate of this single process
– Additional communication might be necessary
– Other processes may idle and waste computing resources during I/O time

2. All or several processes write to one file

+ Number of files is independent of number of processes
+ File is in canonical representation (no post-processing)

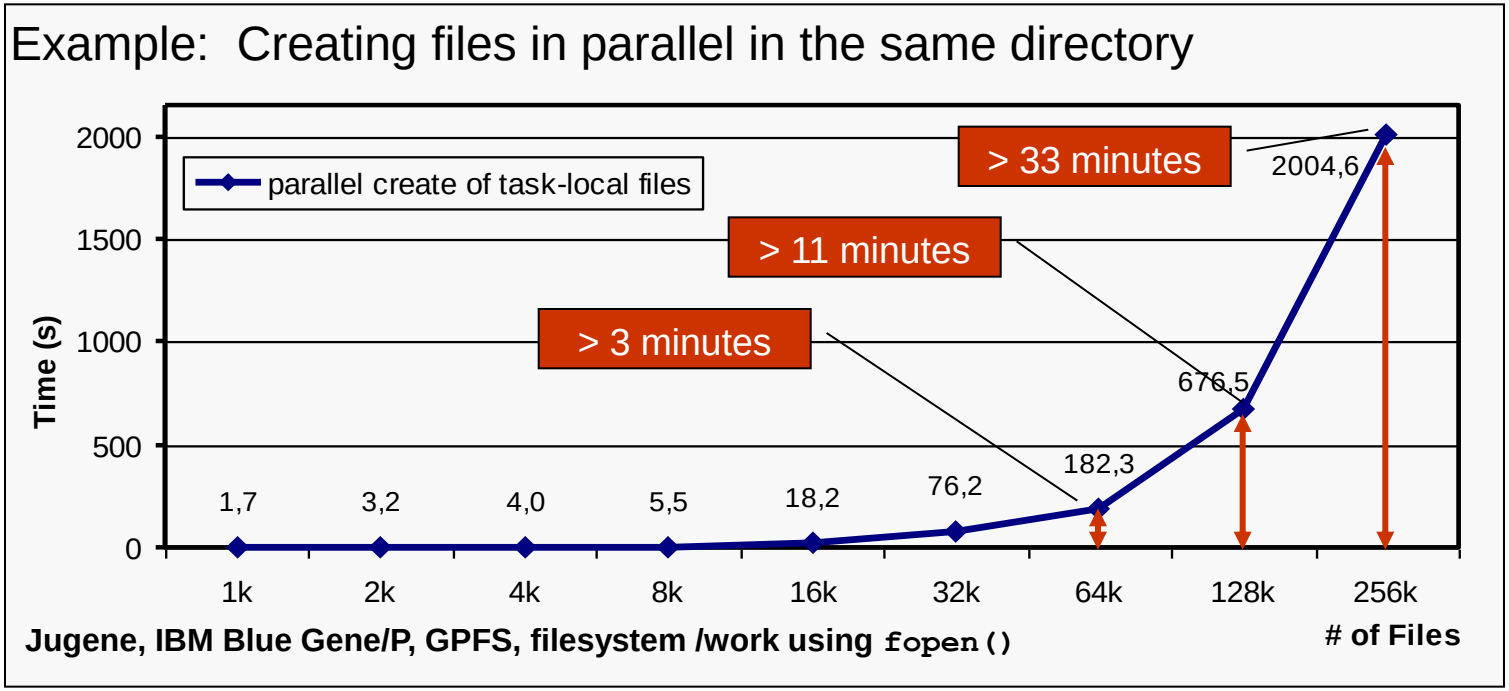
– Uncoordinated client requests might induce time penalties
– File layout may induce false sharing of file system blocks

3. Each process writes to its own file (task-local files)

+ Simple to implement
+ No coordination between processes needed
+ No false sharing of file system blocks

– Number of files quickly becomes unmanageable
– Files often need to be merged to create a canonical dataset
– File system might serialize meta data modification

Pitfall – Serialization of meta data modification



The *creation* of 256.000 files costs 142.500 core hours !

File handling

- Available functions for file handling:
  - *MPI_File_open*
  - *MPI_File_close*
  - *MPI_File_delete*
  - *…*

- Access modes (represented as bit vector)
  - *MPI_MODE_RDONLY*
  - *MPI_MODE_RDWR*
  - *MPI_MODE_WRONLY*
  - *MPI_MODE_CRATE*
  - *MPI_MODE_EXCL*
  - *…*

Parallel Programming with MPI

Code development with MPI

Code development stages

1. Programming
   - Tools: editors with syntax highlighting (e.g. vim, emacs,…), development tools (e.g. Parallel Tools Platform (PTP), syntax checker (e.g. forcheck)

2. Debugging
   - Tools: write/printf statements, classical debuggers (TotalView, DDT, GDB, …), MARMOT, MUST (for MPI codes), Intel® Inspector (for OpenMP codes)

3. Performance
   - Tools: performance analysis tools (Scalasca, Vampir, TAU, …)

Code development – Programming

Code development – FORCHECK

Selected Features

- Verification of conformance to all levels of Fortran standard
- Full static analysis of separate program units
- Reverse engineering tool
- Generates call trees, callby trees, use trees and module dependencies
- Provides an IDE

http://www.forcheck.nl

Code development – PTP

What is PTP:

- Integrated development environment (IDE) for parallel application development
- Based on Eclipse
- Open Source
- Developers:
  - IBM, U.Oregon, UTK, Heidelberg University, NCSA, UIUC, JSC, ...

http://www.eclipse.org/

Code development – Eclipse

File Edit Source Refactor Navigate Search Project Run Run Window Help

Quick Access | C/C++

Project Explor

- HelloWorld
  - Binaries
  - HelloWorld.cpp
  - HelloWorld - [x86_64/le]
  - Includes
  - Makefile

HelloWorld.cpp

```cpp
* HelloWorld.cpp
#include "mpi.h"
#include <iostream>

using namespace std;

int main(int argc, char **argv) {

    int size, rank;

    MPI_Init(&argc, &argv);

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

    cout<<"My rank is "<<rank<<endl;
    if(rank == 0){
        cout<<"We are "<<size<<" tasks"<<endl;
    }

    MPI_Finalize();
}
```

Makefile

```makefile
all: HelloWorld

HelloWorld: HelloWorld.cpp
    mpicxx -o HelloWorld HelloWorld.cpp
```

Problems | Tasks | Console | Properties | Call Graph | Remote Environments

CDT Build Console [HelloWorld]

```
16:13:07 **** Incremental Build of configuration Default_remote for project HelloWorld ****
make all
mpicxx -o HelloWorld HelloWorld.cpp

16:13:09 Build Finished (took 2s.408ms)
```

Writable | Smart Insert | 26 : 1

Code development – MUST

Tool for analyzing and checking MPI applications

- Checks usage of MPI calls during runtime
- Supports C and Fortran

MUST checks for the following classes of errors (among others)

- Communicator usage
- Datatype usage
- Leak checks (MPI resources not freed before calling MPI Finalize)
- Overlapping buffers passed to MPI
- Deadlocks resulting from MPI calls
- Basic checks for thread level usage (`MPI_Init_thread`)

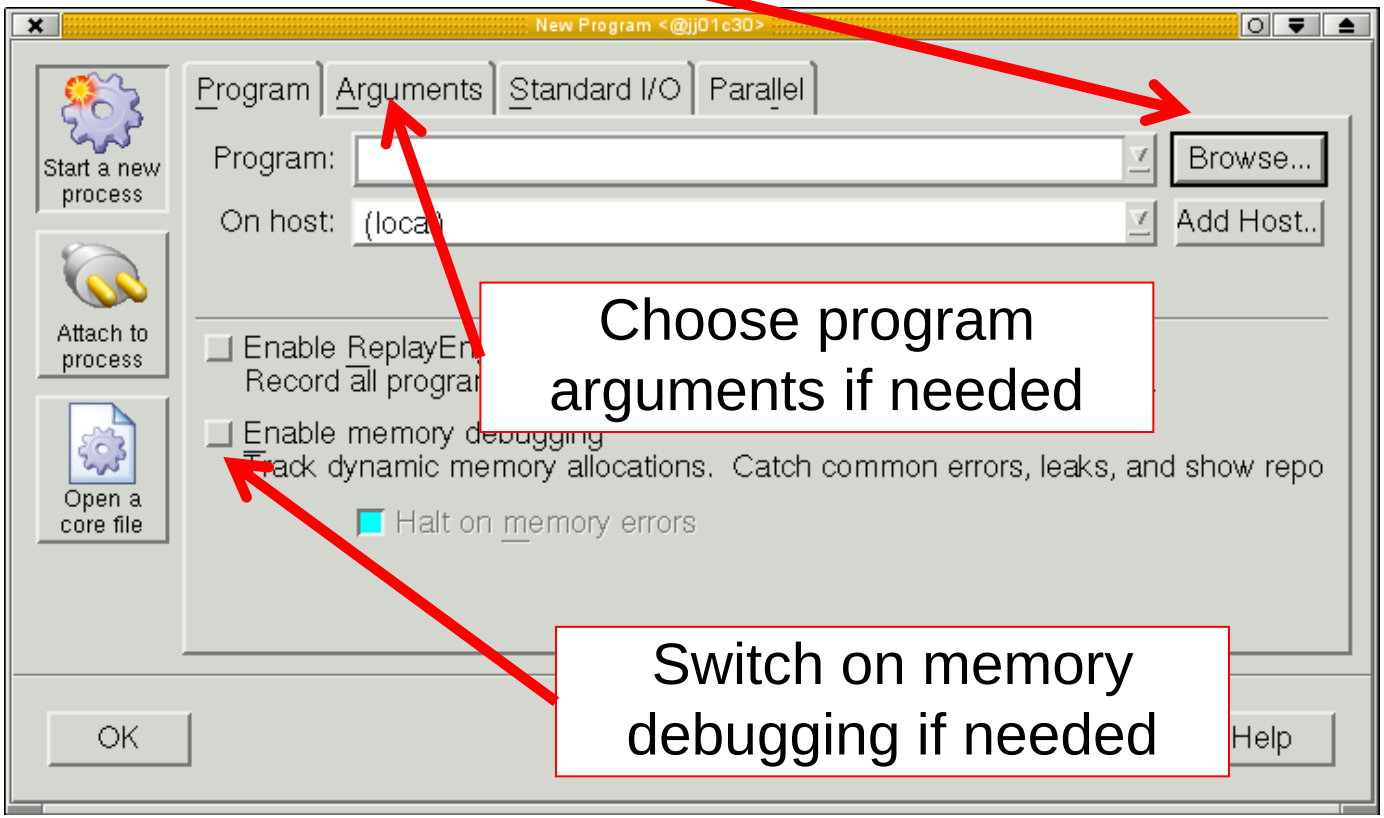
Code development – TotalView debugger

Very powerful tool for code debugging

- Supports C, C++, Fortran
- Available for many platforms
- serial, MPI, OpenMP, hybrid MPI/OpenMP supported
- Some features:
    - *Memory debugging*
    - *Breakpoints, evaluations points, barriers, batch debugging*
    - *Replay engine*
    - *2D Array view, call graphs, value manipulations*

Code development – TotalView

Choose your executable

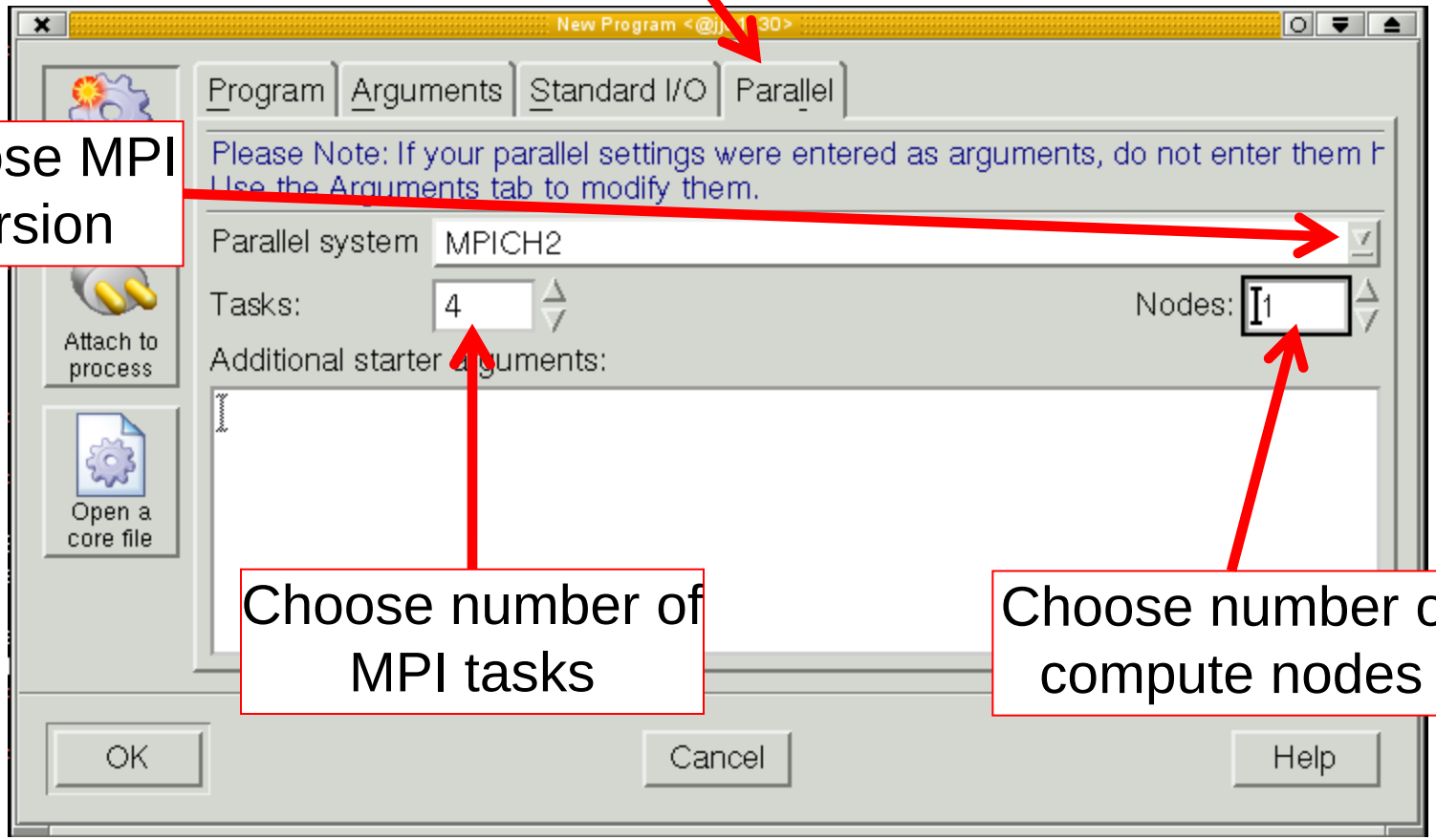


Choose program arguments if needed

Switch on memory debugging if needed

Code development – TotalView

Choose MPI settings

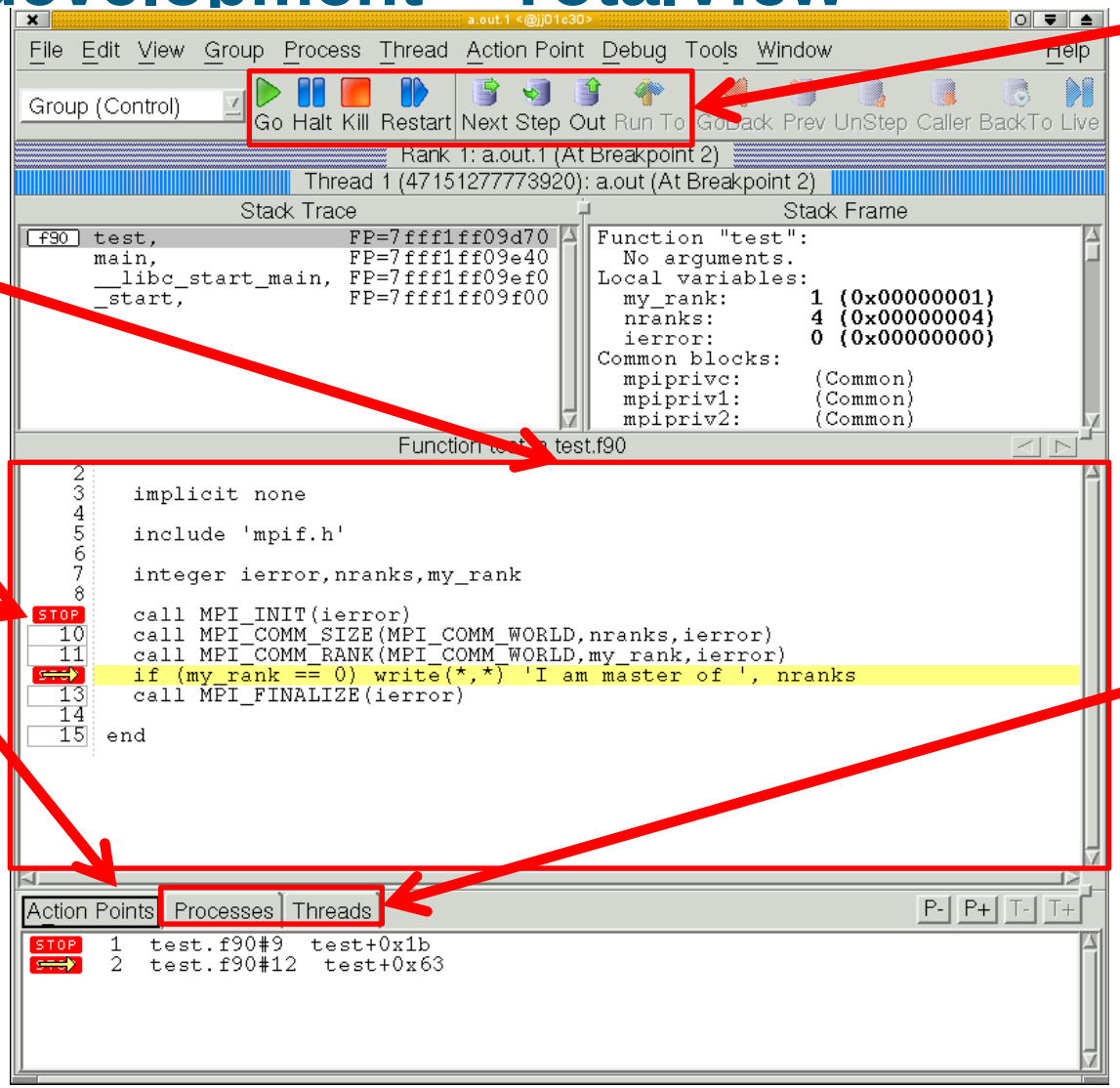


Choose MPI version

Choose number of MPI tasks

Choose number of compute nodes

Code development – TotalView



Navigation

Source code window

Action points

Process and thread view

Parallel Programming with MPI

Common mistakes and pitfalls

Common mistakes

Wrong API usage

- Missing `ierror` argument in Fortran77/90
- Collective routines not called on all ranks of `com`

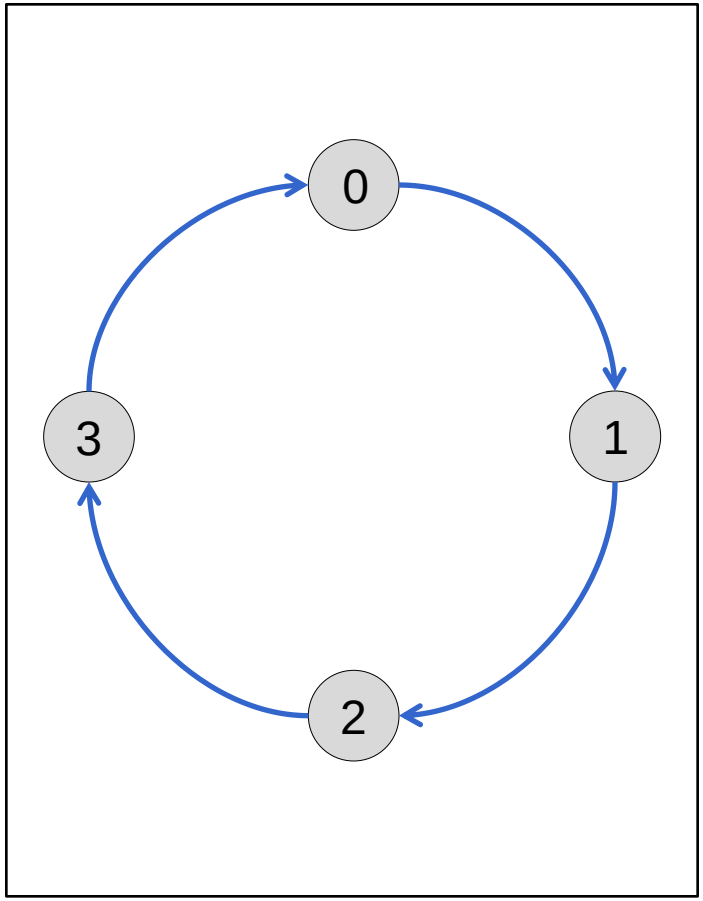
Wrong variable declarations

- Using `INTEGER` where `MPI_OFFSET_KIND` or `MPI_ADDRESS_KIND` is needed
- `status` variable not declared with dimension `MPI_STATUS_SIZE` (Fortran77/90)

Nonblocking communication

- Reusing buffers before it is save to do so
- Missing `MPI_Wait[…]`

Pitfall 1 – Blocking point-to-point communication

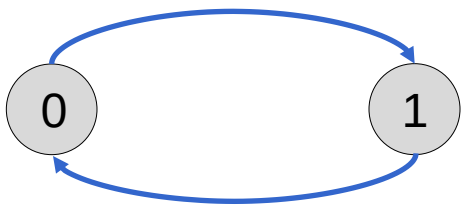



```
…
Call MPI_Ssend(…,dest=my_right_neighbor,…)
Call MPI_Recv(…,source=my_left_neighbor,…)
…
```

- Processes are waiting for sends or receives which can never be posted → **Deadlock**
- Do not have all processes sending or receiving at the same time with blocking calls
  - Use special communication patterns for example, even-odd
  - Use `MPI_Sendrecv`
  - Use nonblocking routines

Pitfall 2 – Blocking point-to-point communication

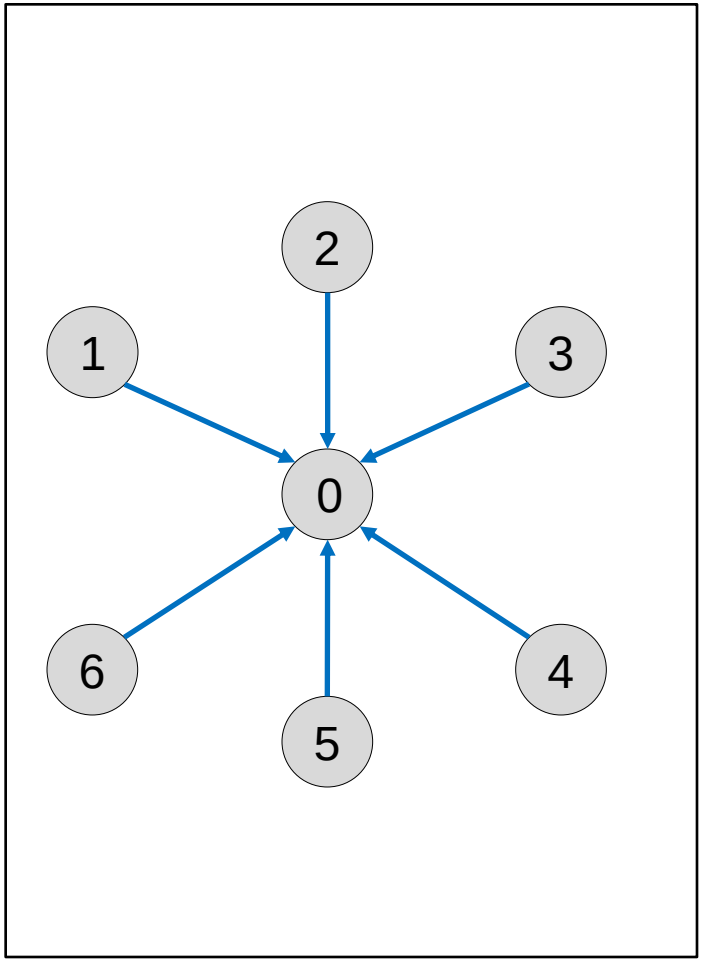
Assumption: `MPI_Send` is implemented as buffered send




```
if (myrank == 0) {
  ierr = MPI_Send(…,dest=1,…)
  ierr = MPI_Recv(…,source=1,…)
}
else {
  ierr = MPI_Send(…,dest=0,…)
  ierr = MPI_Recv(…,source=0,…)
}
```

- `MPI_Send` will return immediately if the message was buffered
- If buffer is filled, `MPI_Send` will be *synchronous*! → **Deadlock**
- Avoid posting many sends/large buffer without corresponding receives or better: DO NOT ASSUME BUFFERING!

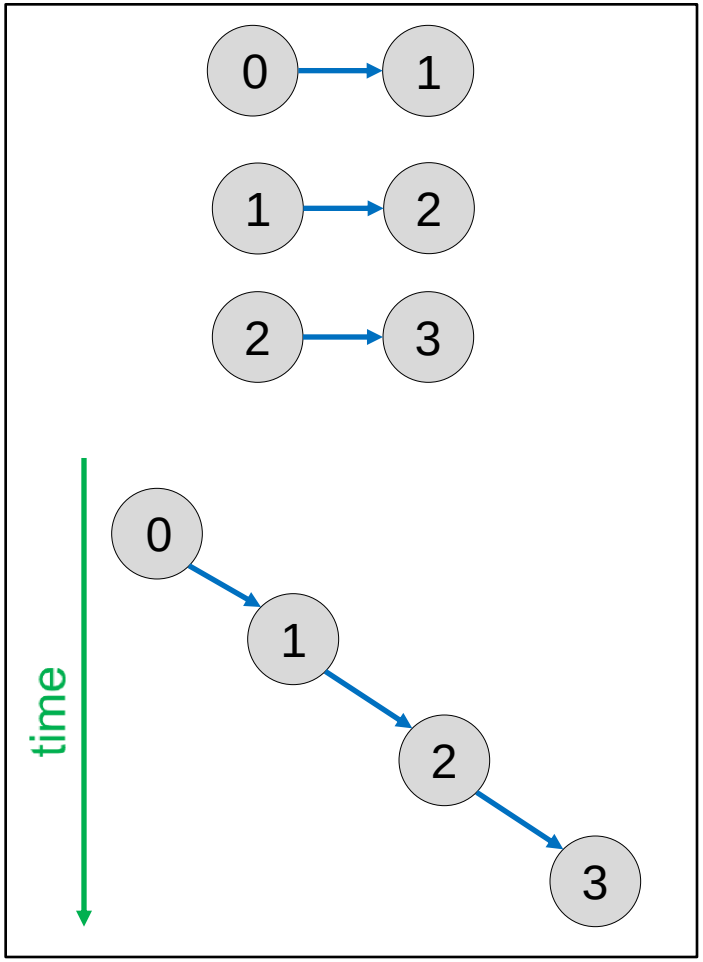
Pitfall 3 – Blocking point-to-point communication




```
if (my_rank != 0){
  for (i=1;i<=100000;i++){
    ierr = MPI_Send(…,dest=0,…)
  }
}
else {
  → receive messages
}
```

- Will fill-up message and/or envelop buffers
  → **Performance penalty**
  - Try to combine messages
  - Use → collective communication
  - Posting receives before sends reduces buffer space

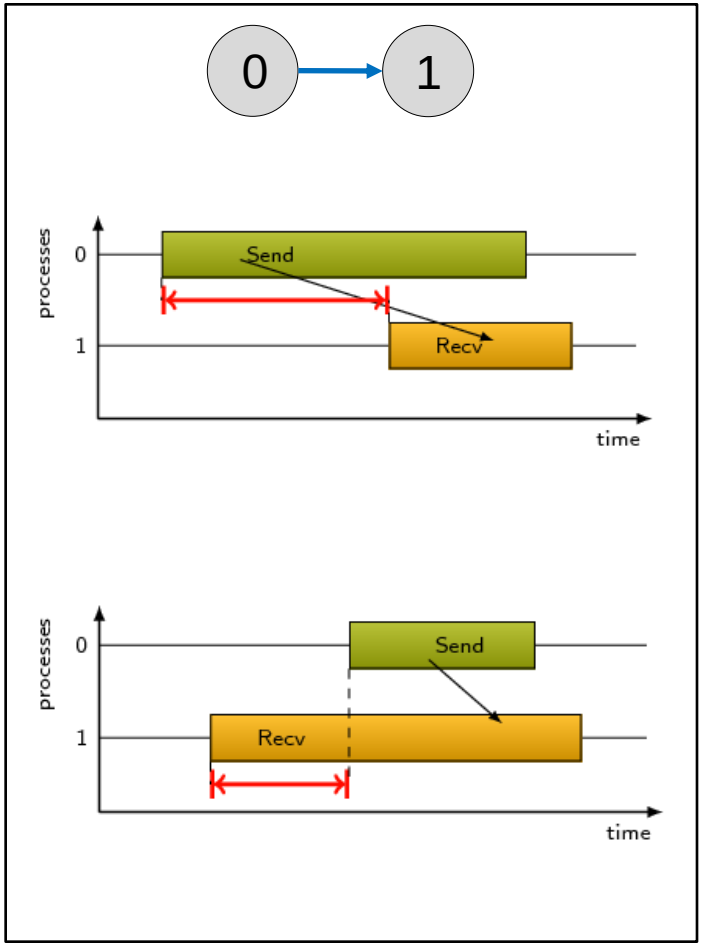
Pitfall 4 – Blocking point-to-point communication




```
if (my_rank == 0) then
  call MPI_Ssend(…,dest=1,…)
else if (myrank == 1) then
  call MPI_Recv(…,source=0,…)
  call MPI_Ssend(…,dest=2,…)
else if (myranks == 2) then
  call MPI_Recv(…,source=1,…)
  call MPI_Ssend(…,dest=3,…)
else if (myrank == 3) then
  call MPI_Recv(…,source=2,…)
endif
```

- Usage of synchronized sends might lead to **serialization**
- Use buffered send or → nonblocking send/receive

Pitfall 5 – Blocking point-to-point communication



Example:

Communication between task 0 and 1

- Sender waits for receiver to call corresponding receive operation
- Performance penalty
- Use nonblocking calls

- Receiver waits for sender to call corresponding send operation
- Performance penalty
- Use nonblocking calls

Pitfall 6 – Collective communication

Collective routines

- ALL ranks of a communicator have to execute them
- Do not mix P2P and collective routines!

```
…
if (my_rank == 0)
{
  MPI_Bcast(&result, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
else
{
  MPI_Recv(&result, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, stat);
}
…
```

Pitfall 7 – Collective communication

Do not mix blocking and nonblocking collectives!

```
…
if (my_rank == 0)
{
  MPI_Bcast(&result, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
else
{
  MPI_Ibcast(&result, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD, &request);
}
…
```

Pitfall 8 – Collective communication

Blocking collective

- Operations must be executed in the same order on all participating tasks
- Otherwise a deadlock will occur

```
…
if (my_rank == 0)
{
  MPI_Bcast(&result1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&result2, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
}
else
{
 MPI_Bcast(&result2, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
 MPI_Bcast(&result1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
…
```