

Introduction to MPI

Introduction to the HPC System of the University Oldenburg

October 6 – 8, 2014

Stefan Albensoeder and Stefan Harfst

Contents

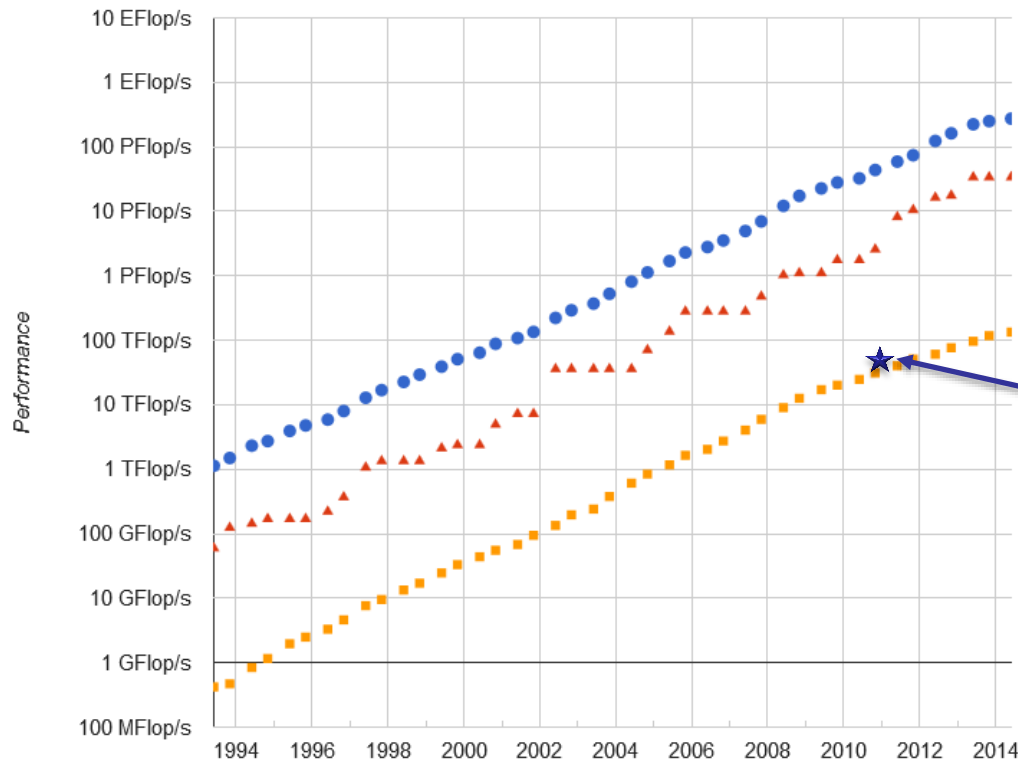
- Motivation for Parallel Computing
- Overview of Parallel Hardware Architectures and Programming Models
- Introduction to MPI
- Examples and Exercises using MPI

Why Parallel Computing?

- laws of physics set limit of how fast computers can become
 - speed of light, quantum scale, ... *(see Lloyd, 2000, Nature, 406, 1047)*
- of course before that there technical limitations
 - e.g. heat from CPU power dissipation → limits the cycle frequency
- what is the solution to the problem?
 1. develop more efficient algorithms
 2. parallel computing

The Fastest Computers on Earth

Performance Development



Theoretical Peak Performance
FLOW/HERO/UV100 \approx 45 Tflop/s
(plot shows measured performance)

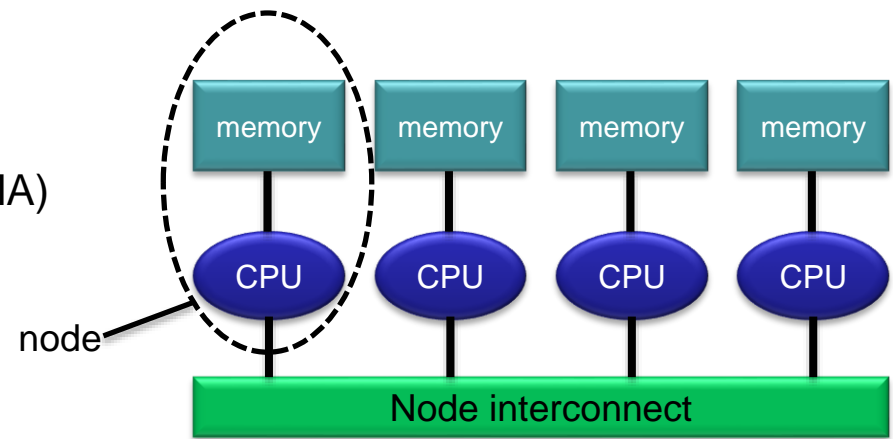
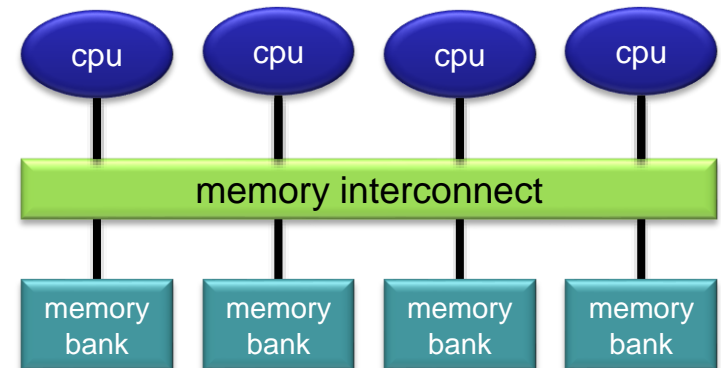
Lists

(taken from top500.org)

■ Sum ■ #1 ■ #500

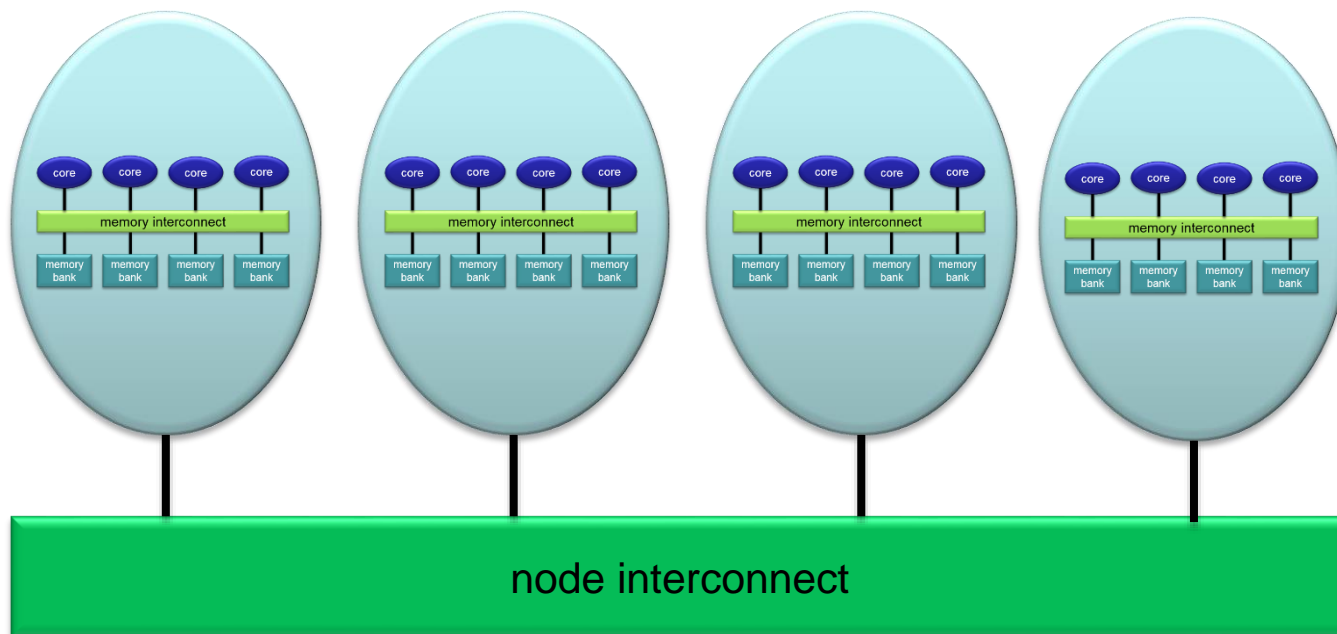
Parallel Hardware Architectures

- multiprocessor
 - shared memory
 - cores are connected to memory with the same speed
 - **Uniform Memory Access (UMA)**
 - **Symmetric Multi-Processing (SMP)**
- multicomputer
 - distributed memory
 - **Non-Uniform Memory Access (NUMA)**
 - nodes are connected by node-interconnect
 - different network topologies



Parallel Hardware Architectures

- most modern HPC systems (e.g. FLOW and HERO) are clusters of SMP/ccNUMA nodes

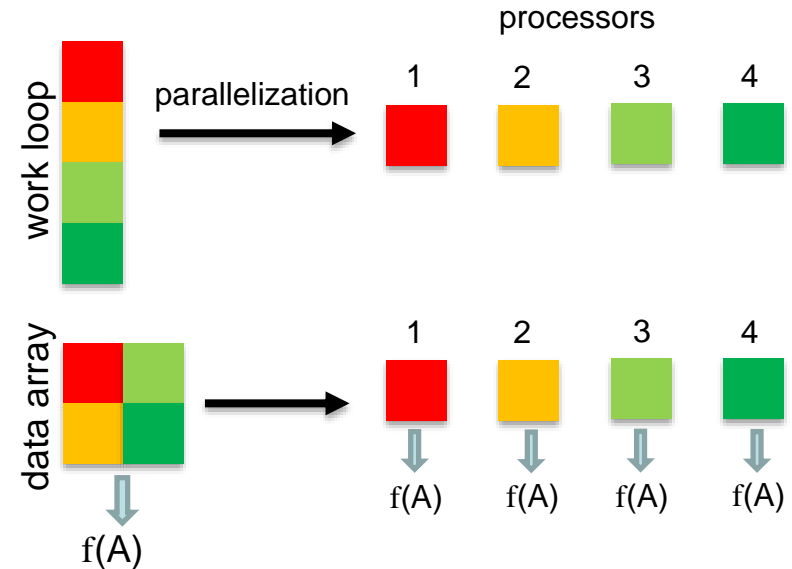


Parallelization Strategies

- two major resources for computations
 - processor
 - memory
- parallelization means
 - distributing the work
 - distributing the data (on distributed memory machines)
 - synchronization of work
 - communication of data (on distributed memory machines)
- parallel programming models provide the methods to achieve the above goals

Distributing Work and Data

- Work decomposition
 - based on loop decomposition
- Data decomposition
 - all the work for a local chunk of the data is done by the local processor
- Domain decomposition
 - work and data are distributed according to a higher model, e.g. reality



Parallel Programming Models

- two dominating programming models:
 - OpenMP: uses directives to define work decomposition
 - MPI: standardized message-passing interface
- other programming models
 - HPF (high-performance Fortran)
 - PGAS (Partitioned Global Address Space), e.g. Co-Array Fortran
UPC (Unified Parallel C)
- programming models for compute devices
 - CUDA
 - OpenCL
 - OpenACC

Overview MPI

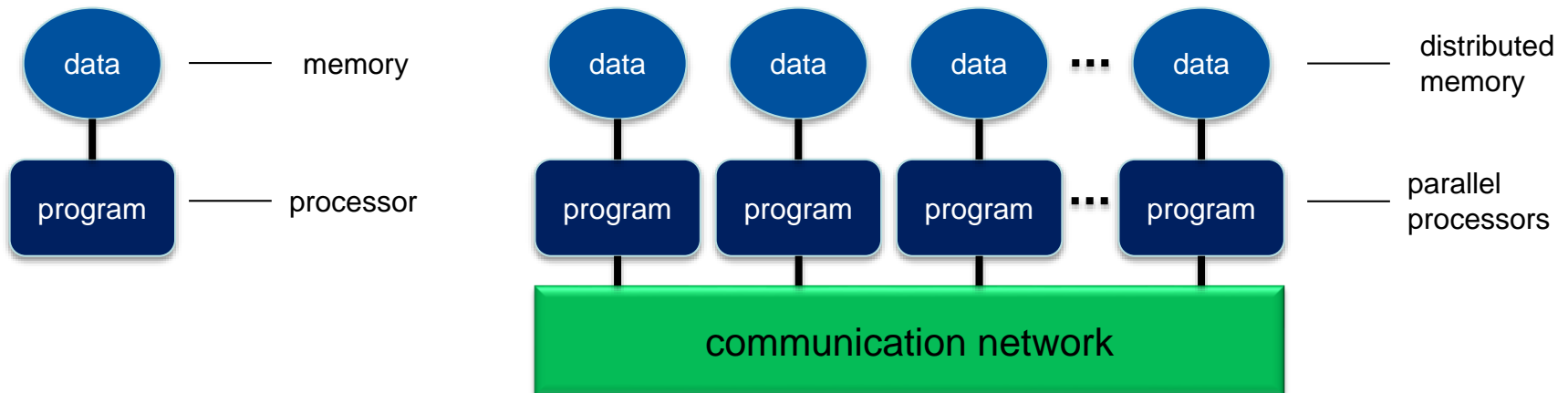
- Introduction to the Message Passing Interface
- Point-to-Point Communication
- Collective Communication
- Other and New Features of MPI
 - Derived Datatypes
 - Virtual Topologies
 - Process Creation and Management
 - One-sided Communication and Shared Memory
 - MPI and Threads
 - Parallel File I/O

History of MPI

- MPI is a standard with the prime goals
 - to provide a message-passing interface
 - to provide source-code portability
 - to allow efficient implementations
- MPI exists for more than 20 years
 - MPI-1.0 was released in June, 1994
 - MPI-2.0 was released in July, 1997 and provided additional functionality
 - MPI-3.0, the current standard, was released in October, 2012 and was developed for better platform and application support (in particular clusters of SMP nodes)

A Message-Passing Interface

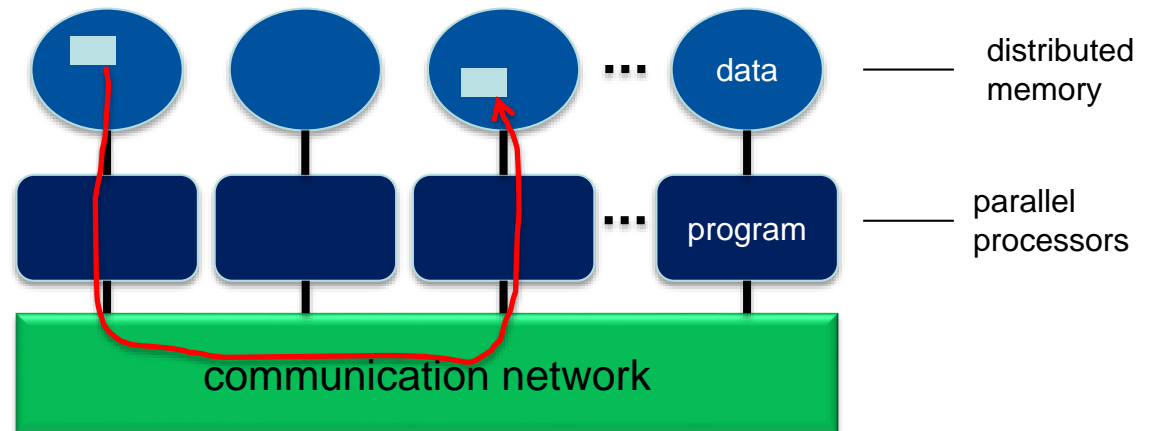
- sequential program vs. message-passing program



- message-passing programming paradigm:
 - each processor runs a (sub)program, typically the same (SPMD)
 - variables of subprograms have the same name but different (distributed) data
 - communication by special library routines → message passing

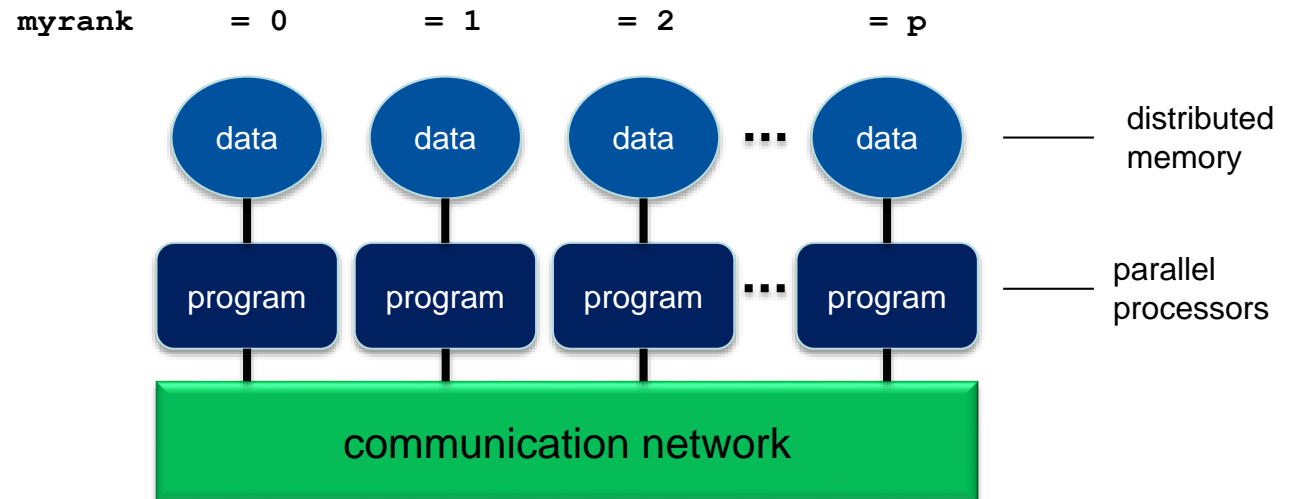
Message Passing

- messages are passed through the communication network
- messages require the following information:
 - sending and receiving process
 - data location
 - data type
 - data size



- in order to use the message-passing interface the program must be
 - connected to the MPI library (at compile time)
 - started with the MPI startup tool (mpirun or mpiexec)
 - at runtime MPI is initialized with special library calls (MPI_Init())

Process Identification



- processes in MPI are identified by their rank
 - determined by calling a library function
 - rank is used for addressing when sending messages
 - rank is used for making decisions, e.g. when distributing the data and work

Example: MPI_HelloWorld

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

int main(int argc, char *argv[])
{
    int my_rank, size;

    MPI_Init(&argc, &argv);

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

    if (my_rank == 0)
    {
        printf ("Hello world!\n");
    }

    printf("I am process %i out of %i.\n", my_rank, size);

    MPI_Finalize();
}
```

MPI Header and Module Files

- C/C++: `#include <mpi.h>`
- Fortran: `include "mpif.h"`
 or `use mpi`
 or `use mpi_f08`
 - the use of the old style include-statement is strongly discouraged as no compile-time argument checking can be done
 - recommended is the use of `mpi_f08`

MPI Library Calls

- in general an MPI library call has the form

C/C++: `error = MPI_Xxxxx(parameter, ...);`
 `MPI_Xxxxx(parameter, ...);`

Fortran: `CALL MPI_Xxxxx(parameter, ..., ierror)`

- in Fortran the use of `ierror` has change with MPI-3.0:
 if (and only if!) you are using the module file `mpi_f08`, `ierror` is an optional argument. In any other case **error cannot be omitted** otherwise terrible unforeseen things may happen.
- refer to the MPI-3.0 standard document to look up the definitions and argument list of available MPI functions

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

MPI_Init() and MPI_Finalize()

- MPI is initialized with
 - C/C++: `MPI_Init(&argc, &argv);`
 - Fortran: `CALL MPI_Init(ierr)`
 - must be the first MPI-routine that is called (few exceptions)
 - call as early as possible in your program
 - in C/C++ argv and argc are passed by reference (possibly cleans argv from unwanted MPI arguments)

- MPI is finalized with
 - C/C++: `MPI_Finalize();`
 - Fortran: `CALL MPI_Finalize(ierr)`
 - must be the last MPI-routine that is called (few exceptions)

MPI Communicators

- all MPI processes (subprograms) are combined in the communicator `MPI_COMM_WORLD`
 - `MPI_COMM_WORLD` is a handle predefined in the header files
 - each process in a communicator has its own rank starting from 0 until $(\text{size}-1)$
 - the size of a communicator and the rank of a process within the communicator can be determined with special library calls
 - it is possible to define your own communicators (e.g. for a subset of processes) and handles

MPI_Comm_size() and MPI_Comm_rank()

- to determine the size of a communicator use

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

- to determine the rank of a process within a communicator use

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

- note that **size** is the same on every process whereas **myrank** is different

Example: MPI_HelloWorld

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

int main(int argc, char *argv[])
{
    int my_rank, size;

    MPI_Init(&argc, &argv);

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

    if (my_rank == 0)
    {
        printf ("Hello world!\n");
    }

    printf("I am process %i out of %i.\n", my_rank, size);

    MPI_Finalize();
}
```

Example: MPI_HelloWorld

```
PROGRAM MPI_HelloWorld
  USE mpi_f08
  IMPLICIT NONE

  INTEGER rank, size

  CALL MPI_Init()

  CALL MPI_Comm_rank(MPI_COMM_WORLD, rank)
  CALL MPI_Comm_size(MPI_COMM_WORLD, size)

  IF (rank .EQ. 0) THEN
    WRITE(*,*) 'Hello world!'
  END IF

  WRITE(*,*) 'I am process', rank, ' out of', size

  CALL MPI_Finalize()

END PROGRAM MPI_HelloWorld
```


Compiling an MPI Program

- programs are compiled using a wrapper command:

```
C:          $ mpicc [options] <source.c> -o <executable>
C++:       $ mpic++ [options] <source.c> -o <executable>
Fortran:   $ mpifort [options] <source.c> -o <executable>
```

– uses the standard compiler (GCC, ICS) with some extra options

- example on HERO:

```
[abcd1234@hero02 ~]$ module load intel intel/ics/2013_sp1.3.174/64
[abcd1234@hero02 ~]$ module load openmpi/1.8.1/ics
[abcd1234@hero02 ~]$ mpicc MPI_HelloWorld.c -o MPI_HelloWorld
```

Running an MPI Program

- programs are executed using the MPI startup tool

```
$ mpirun -np <N> [options] <executable>
```

or

```
$ mpiexec -np <N> [options] <executable>
```

- example on HERO:

- note: do not normally run programs on the head nodes

```
[abcd1234@hero02 ~]$ mpirun -np 4 --mca btl ^openib MPI_HelloWorld
I am process 2 out of 4.
I am process 3 out of 4.
I am process 1 out of 4.
Hello world!
I am process 0 out of 4.
```

Running an OpenMPI Program with SGE

```
#!/bin/bash
##### SGE settings
#$ -cwd
#$ -N MPI_HelloWorld

##### parallel environment
#$ -l cluster=hero
#$ -pe openmpi 4
#$ -R y

##### requesting resources
#$ -l h_rt=0:10:0
#$ -l h_vmem=1000M
#$ -l h_fsize=100M

# loading modules
module load intel intel/ics/2013_sp1.3.174/64
module load openmpi/1.8.1/ics

# the --mca options tells MPI not to use Infiniband (prevents warning)
mpirun -np $NSLOTS -machinefile $TMPDIR/machines --mca btl ^openib MPI_HelloWorld
```

Sending and Receiving Messages

- the MPI library provides functions to send and receive messages:
 - sending: `MPI_Send(...)`
 - receiving: `MPI_Recv(...)`
 - any message sent must be received, otherwise → deadlock
 - function prototypes (here C/C++, Fortran is analogous)

```
MPI_Send(void* data, int count, MPI_Datatype datatype, int destination,
         int tag, MPI_Comm communicator)
```

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

MPI Data Types

- MPI needs to know the type of data that is send
- predefined handles are provided for standard data types, e.g.:
 - `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_CHAR`, ...
- you can also define handles for your own data types

MPI Send and Receive Example

```

...
int number;
if (my_rank == 0) {
    number = 42;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (my_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
           number);
}
...

```

Sending and Receiving Messages

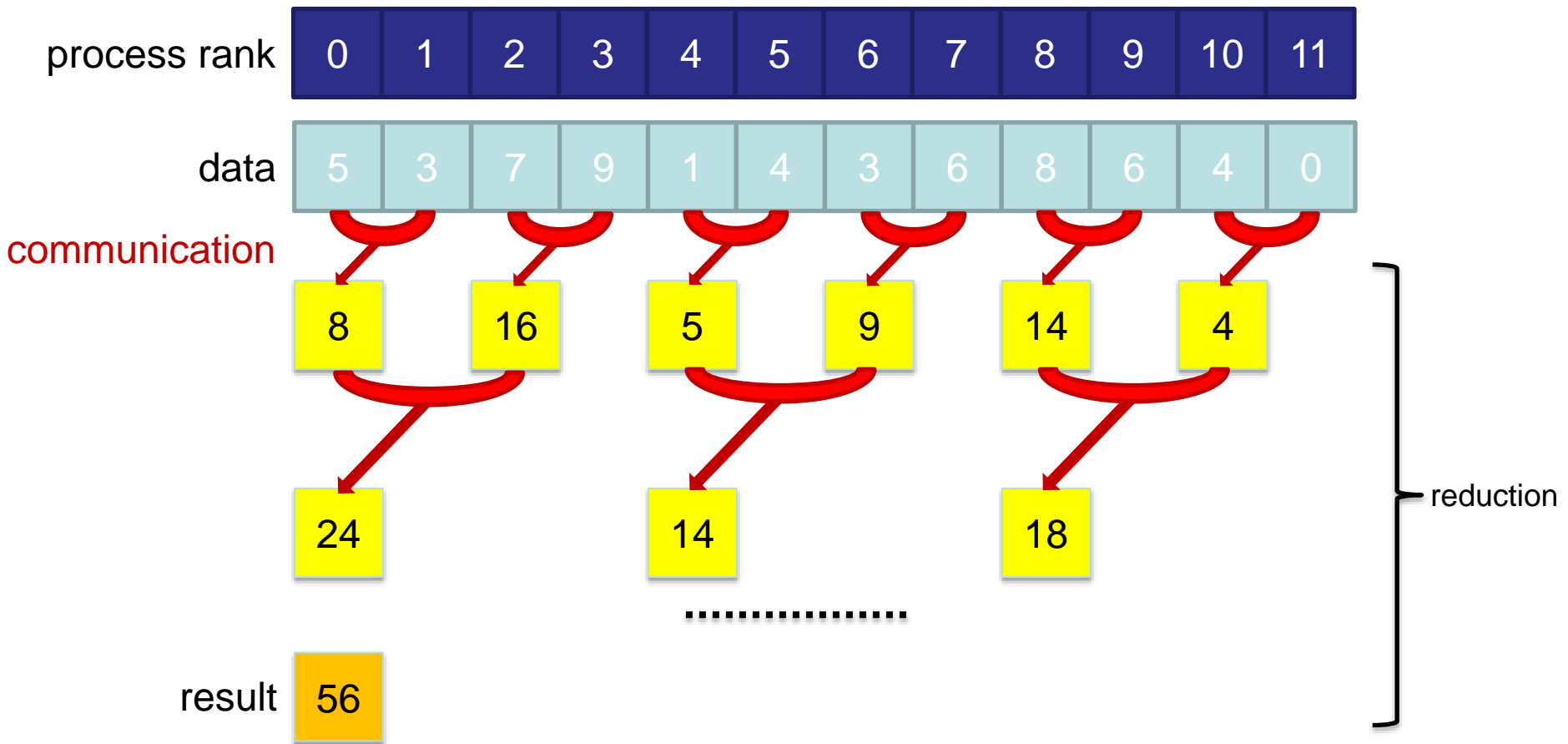
- message can be sent in different ways
 - synchronous vs. asynchronous:
sender receives a confirmation receiving of the message is initiated
 - unbuffered vs. buffered:
the message can be buffered so the sender can continue using the sent variable, requires additional memory
 - blocking vs. non-blocking:
send or receive functions return immediately allowing to overlap communication and computation

for details refer to the MPI-3.0 standard

Collective Communication

- so far we have looked at point-to-point communication
- MPI allows
 - one-to-all
 - all-to-one
 - all-to-all } communication
- example: calculate the sum of the elements of an array

Collective Communication



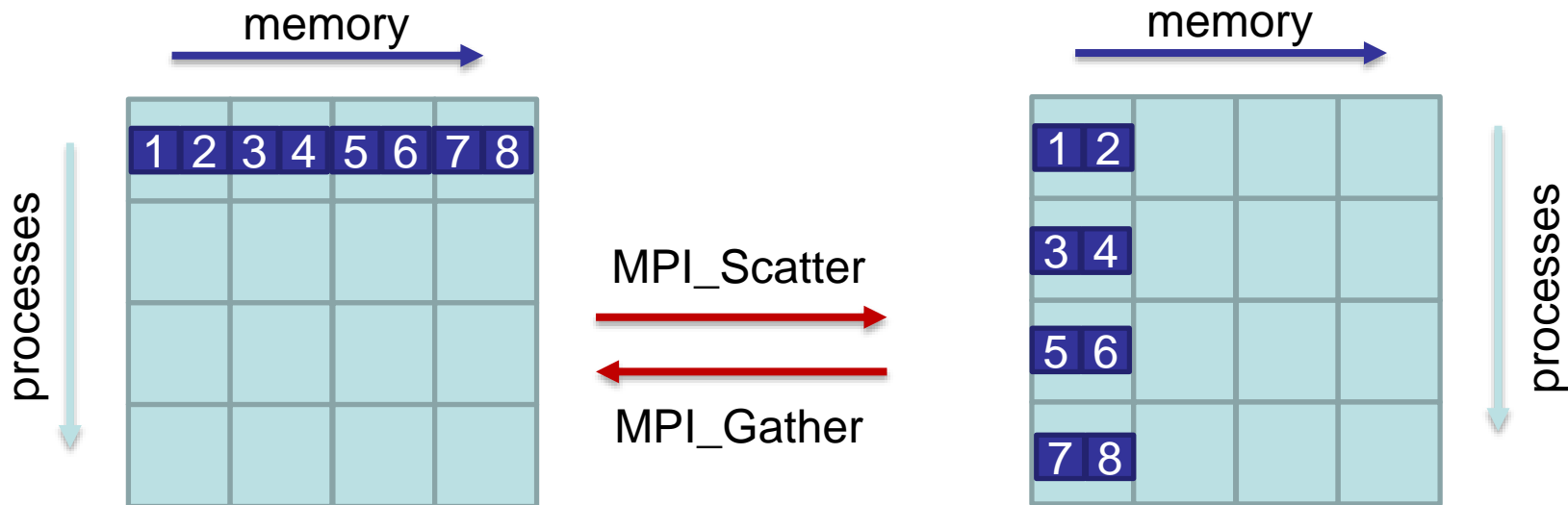
Collective Communication

- so far we have looked at point-to-point communication
- MPI allows knows
 - one-to-all
 - all-to-one
 - all-to-all
 } communication
- example: calculate the sum of the elements of an array
- MPI collective communication is very efficient due to tree-based communication
- collective communication can still be very expensive, in particular all-to-all

Collective Communication

- a selection of collective MPI communications:
 - MPI_Bcast(...) sending data from one process to all others
 - MPI_Scatter(...) distributing an array of data from one to all
 - MPI_Gather(...) collecting an array of data from all to one
 - MPI_Reduce(...) reduction operation defined by a handle, e.g. MPI_SUM
 - MPI_Barrier(...) used to synchronize all processes
 - ...
- some also have all-to-all variant, e.g. MPI_Allreduce
- since MPI-3.0 also non-blocking calls

Example: MPI_Scatter and MPI_Gather



Summary

- MPI allows the parallelization of programs by distribution data and work through the passing of messages
 - MPI is a platform-independent standard
 - very flexible but also more time-consuming to program
 - read the standard document for more details
 - <http://www.mpi-forum.org/docs/>
- MPI library calls can be used to send and receive messages
 - whenever needed use higher-level functions for collective communication for efficiency
- with MPI you can introduce new types of errors in your program, e.g. deadlocks
 - however, many errors in parallel programs stem from error in the serial code

MPI Exercises and Examples

- Programming Exercises include
 - MPI_HelloWorld
 - MPI_PingPong
 - MPI_RingSend
 - Timing MPI_Send()
- Do as many exercises as you can or like
 - Solutions are provided
but do not look at solutions until you completed the exercise!

Exercise 1: MPI_HelloWorld

- Write an MPI_HelloWorld program following the example given before
 - compile and run the program on the cluster
 - what do you notice when you run the program several times?
Can you explain the behavior?

Exercise 2: MPI_PingPong

- Write an MPI_PingPong program to run with two processes doing the following:
 - initialize a counter
 - one process increments the counter and sends it to the other
 - the other process receives the message and then increments the counter and sends it back
 - repeat until n messages have been sent

Exercise 3: MPI_RingSend

- Write an MPI_RingSend program to run with any number of processes computing on all processes the sum of all ranks:
 - each process receives the current sum from its left neighbor (rank-1)
 - each process adds its own rank to the current sum
 - each process sends the new current sum to its right neighbor
 - the ring is terminated after one round
 - note: the left/right neighbor for rank $0/(size-1)$ is $(size-1)/0$
 - is MPI_RingSend a truly parallel program when using MPI_Send() and MPI_Recv()?

Exercise 4: Timing MPI_Send()

- Write an MPI program that sends an array of doubles from one process to another
 - using MPI_Wtime() measure the time it takes to complete the MPI_Send() call
 - determine this time as function of N where N is the size of the array
 - plot the result and explain