Introduction to parallel programming and MPI

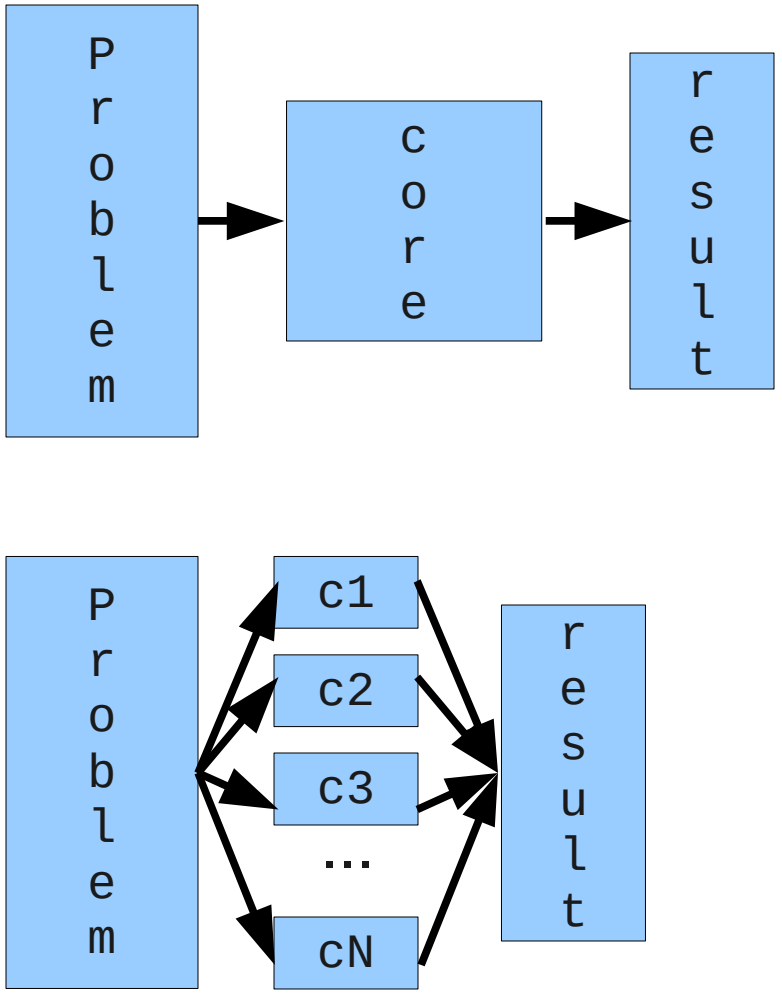
Jussi Enkovaara

jussi.enkovaara@csc.fi

CSC – Tieteen tietotekniikan keskus Oy
CSC – IT Center for Science Ltd.

What is parallel computing?

- Serial computing
  - single processing unit (core) is used for solving a problem
  - single task performed at once
- Parallel computing
  - multiple cores are used for solving a problem
  - problem is split into smaller subtasks
  - multiple subtasks are performed simultaneously



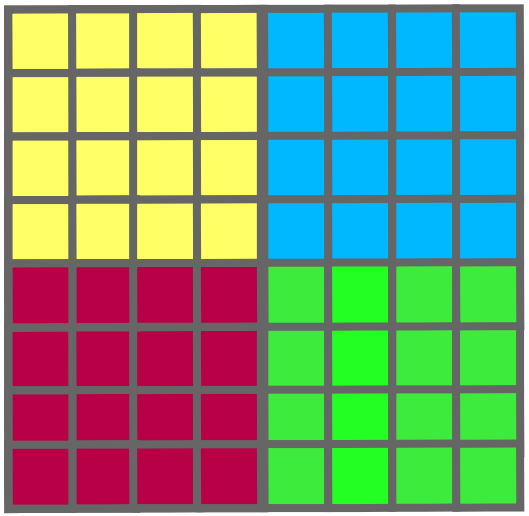
Why parallel computing

- Solve problems faster
  - CPU clock frequencies are no longer increasing
  - speed-up is obtained by using multiple cores
  - parallel programming is required for utilizing multiple cores
- Solve bigger problems
  - parallel computing may allow application to use more memory
  - apply old models to new length and time scales
  - grand challenges
  - new science
- Solve problems better
  - more precise models
  - new science

Data parallelism

- Data is distributed to processor cores
- Each core performs (nearly) identical tasks with different data
- Example: summing the elements of an 2 D array

core 1: ∑=



core 2: ∑=

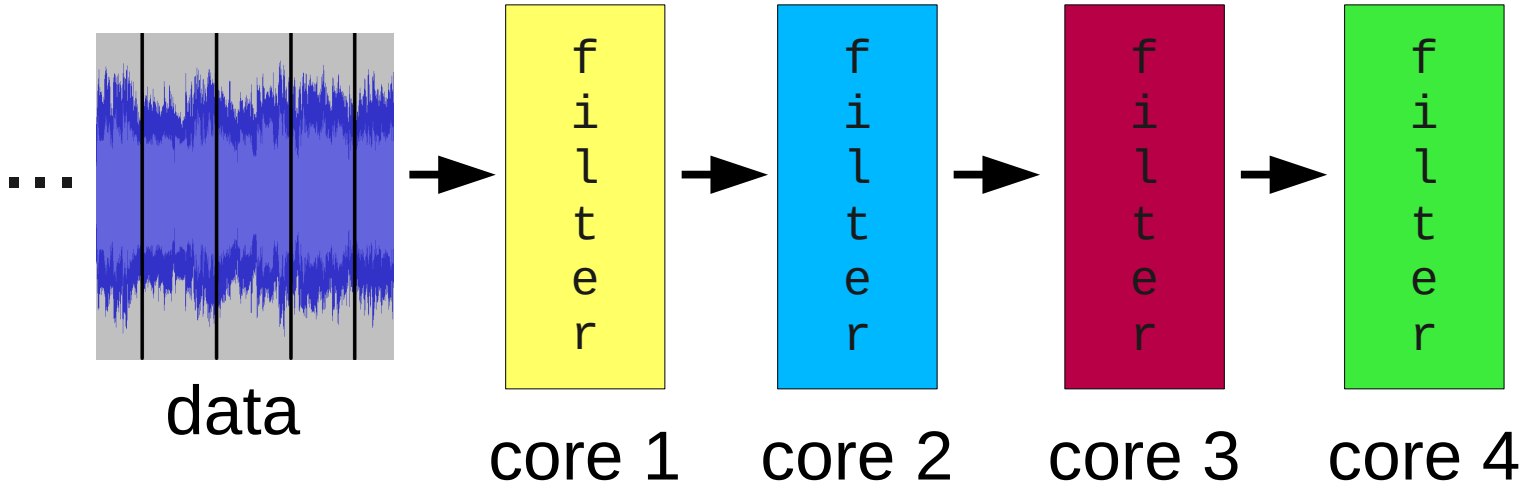
core 3: ∑=

core 4: ∑=

- Each core sums it's part of the array
- The individual sums have to be combined in the end

Task parallelism

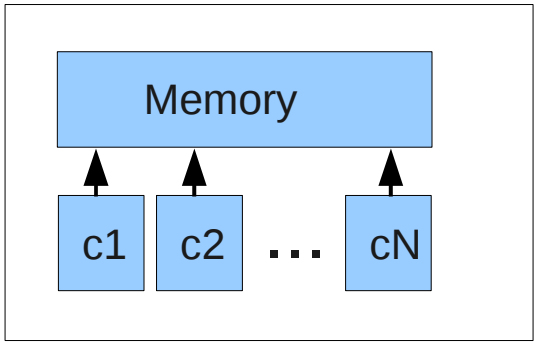
- Different cores perform different tasks with the same or different data
- Example: signal processing, four filters as separate tasks

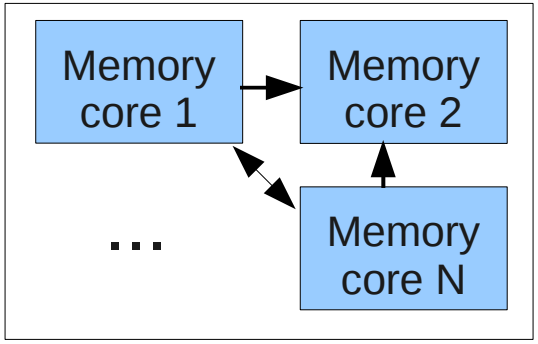


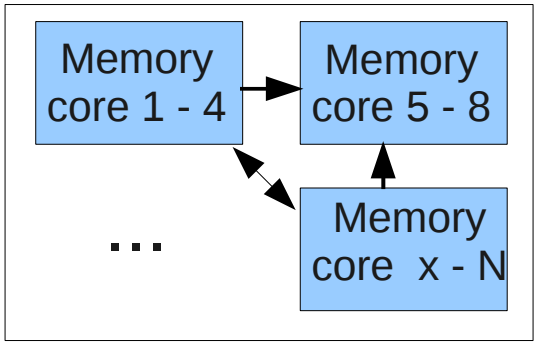
- Data is processed as segments
- Core 2 obtains a segment after core 1 has processed it; core 1 starts to process a new segment
- When the first segment gets to core 4, all cores are busy

Types of parallel computers

- Shared memory computers
  - all the cores can access the whole memory
  - hippu.csc.fi
- Distributed memory computers
  - all the cores have their own memory
  - communication is needed in order to access the memory of other cores
- Current supercomputers combine the distributed memory and shared memory approaches
  - shared memory nodes containing several cores
  - communication is needed between the nodes
  - louhi.csc.fi, murska.csc.fi

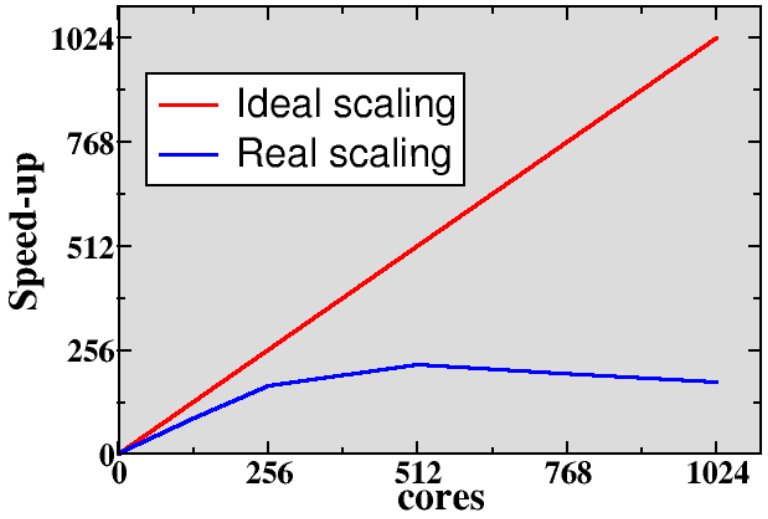




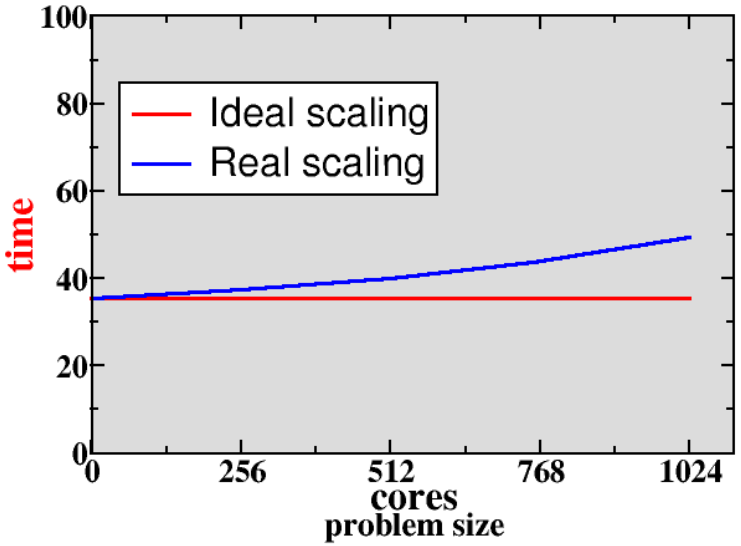


Parallel computing concepts

- Strong parallel scaling
  - constant problem size
  - execution time decreases in proportion to the increase in the number of cores
  - shortens the time to solve a problem

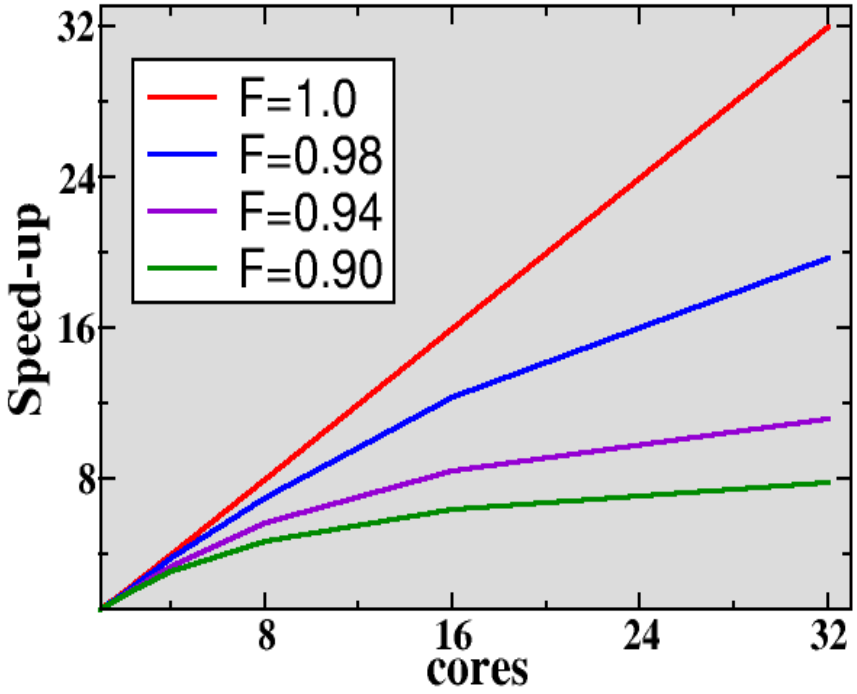


- Weak parallel scaling
  - increasing problem size
  - execution time remains constant when number of cores increases in proportion to the problem size
  - enables to solve larger problems



Amdahl's law

- Parallel programs contain often sequential parts
- Amdahl's law gives the maximum speed-up in the presence of non-parallelizable parts



Maximum speed-up:

$$\frac{1}{(1-F)+F/N}$$

F: parallel fraction

N: number of cores

More parallel concepts

- Synchronization
  - coordination of processes for maintaining correct runtime order and for keeping data coherent
- Granularity
  - amount of synchronization needed between subtasks
  - fine grained: lots of synchronization
  - coarse grained: synchronization less frequent
  - embarrassingly parallel: synchronization is needed rarely or never
- Load balance
  - distribution of workload to different cores
- Parallel overhead
  - additional operations which are not present in serial calculation
  - synchronization, redundant computations, communications

Programming models

- Message passing (MPI)
  - can be used both in distributed and shared memory computers
  - programming model allows good parallel scalability
  - programming is more complex
- Threads (pthreads, OpenMP)
  - can be used only in shared memory computers
  - limited parallel scalability
  - “simple” programming
- Hybrid programming
  - threads inside a node, message passing between nodes
  - can improve scaling with extreme core counts (> 10000)
- Other
  - PGAS (partitioned global address space) languages, e.g. unified parallel C or co-array Fortran

Message passing interface

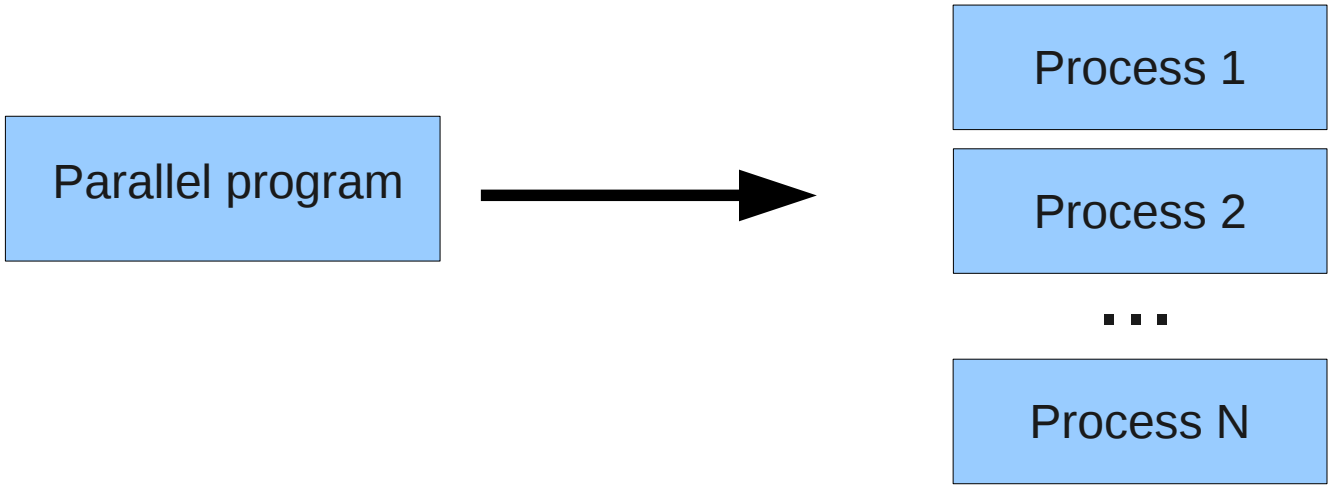
- MPI is an application programming interface (API) for communication between separate processes
- The most widely used approach for distributed parallel computing
- MPI programs are portable and scalable
  - the same program can run on different types of computers, from PC's to supercomputers
- MPI is flexible and comprehensive
  - large (over 120 procedures)
  - concise (often only 6 procedures are needed)
- MPI standardization by MPI Forum
  - open group
  - representatives from several organizations

Brief history of MPI

- 1992 standardization started
- 1994 MPI-1 1.0
- 1995 MPI-1 1.1, MPI-1 1.2
  - clarifications and corrections
- 1997 MPI-2
  - several new features
- 2009 MPI-2 2.2
  - September, Espoo!
  - few enhancements and clarifications
- 2011 MPI-3 ?
  - first draft published Nov 2010
  - major enhancements
  - not fully backward compatible?

Execution model

- Parallel program is launched as set of **independent, identical processes**



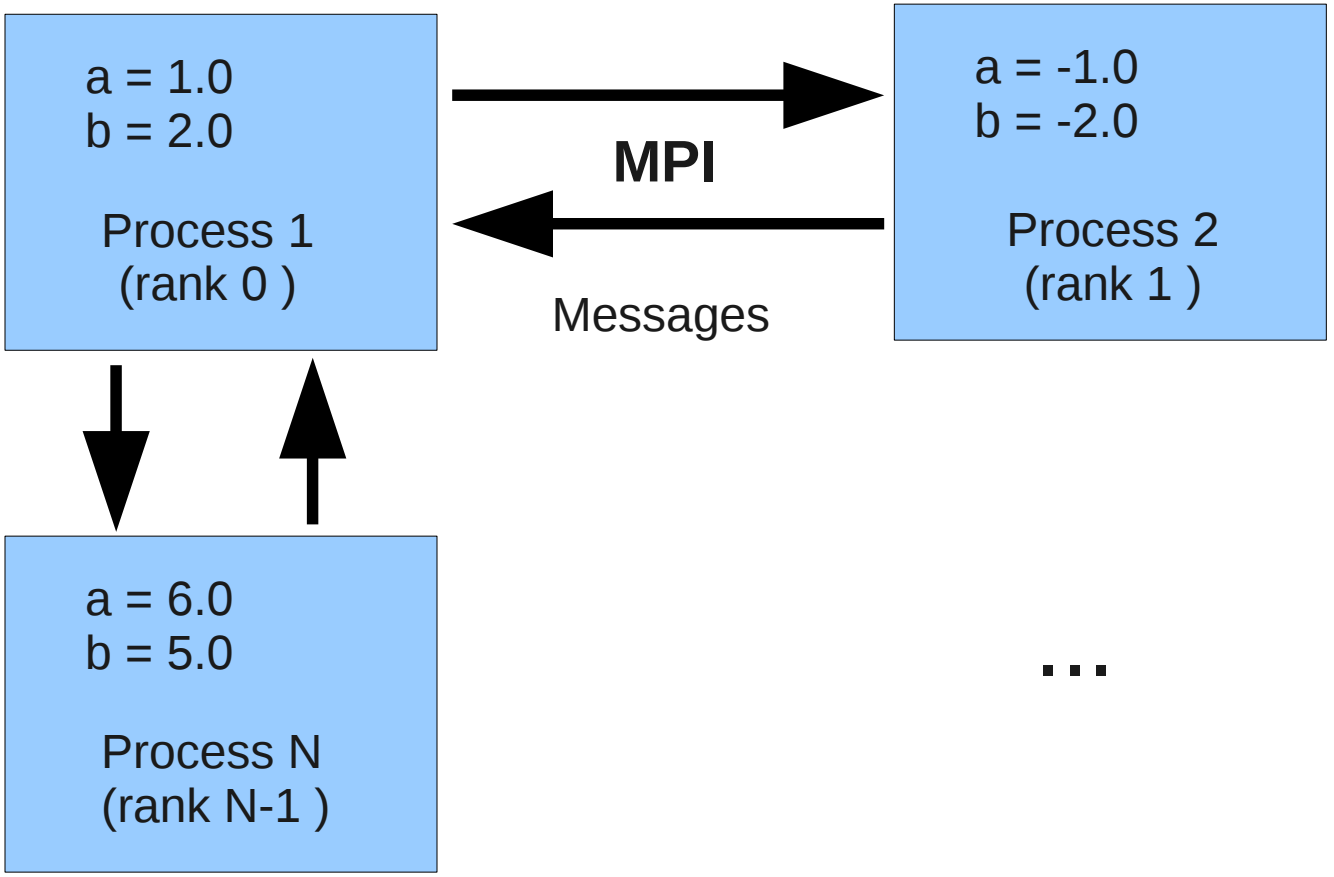
- All the processes contain the same program code and instructions
- Processes can reside in different nodes or even in different computers
- The way to launch parallel program is implementation dependent
  - mpirun, mpiexec, aprun, poe, ...

- MPI runtime assigns each process a **rank**, which can be used as an ID of the processes
  - ranks start from 0 and extent to N-1
- Processes can perform different tasks and handle different data based on their **rank**

```
...
if ( rank == 0 ) {
   …
   }
if ( rank == 1) {
   …
   }
...
```

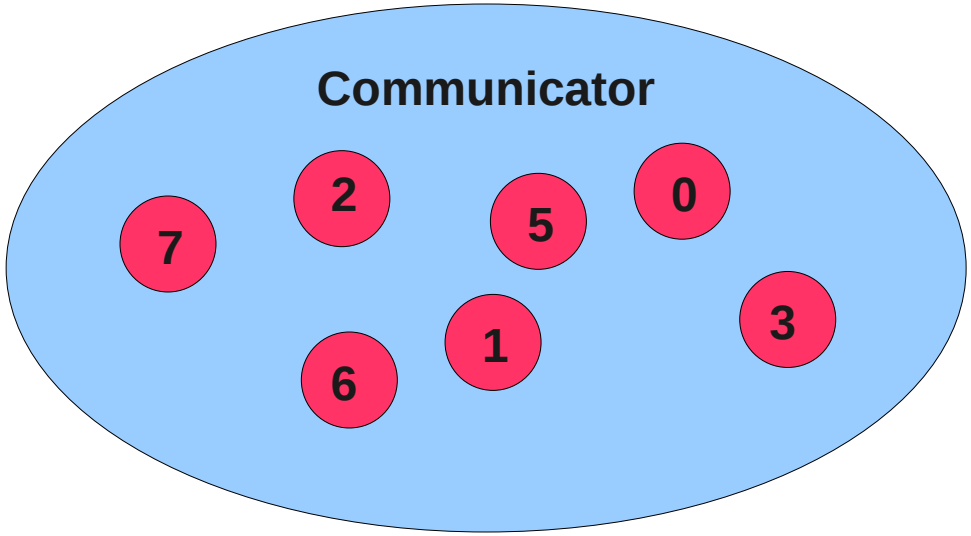
Data model

- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving messages



MPI communicator

- Communicator is an object connecting a group of processes
- Initially, there is always a communicator **MPI_COMM_WORLD** which contains all the processes
- Most MPI functions require communicator as an argument
- Users can define own communicators



MPI library

MPI library contains functions and procedures for:

- Obtaining information about the communicator
  - number of processes
  - rank of the process
- Communication between processes
  - sending and receiving messages between two processes
  - sending and receiving messages between several processes
- Synchronization between processes
- Advanced features

Programming MPI

- MPI standard defines interfaces to C and Fortran programming languages
- C call convention
  - rc = MPI_Xxxx(parameter, …)
  - some arguments have to passed as pointers
- Fortran call convention
  - CALL MPI_XXXX(parameter, ..., rc)
  - return code in the last argument
- C++ interface has been deprecated in the current standard, so **do not** use it!
- There are unofficial bindings to Python, Perl and Java

First five MPI commands

- Set-up the MPI environment

  MPI_Init()
- Information about the communicator

  MPI_Comm_size(comm, size)

  MPI_Comm_rank(comm, rank)

  Parameters
  - **comm** communicator (in this course always **MPI_COMM_WORLD**)
  - **size** number of processes in the communicator
  - **rank** rank of this process
- Synchronize processes

  MPI_Barrier(comm)
- Finalize MPI environment

  MPI_Finalize()

Presenting syntax

Creating a communicator

- MPI_Comm_split creates new communicators based on 'colors' and 'keys'

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

comm        communicator handle
color       control of subset assignment (non-negati
            Processes with the same color are in the
            communicator
key         control of rank assignment (integer)
newcomm     new communicator handle
```

If colo

not
ne

Creating a communicator

- C and Fortran bindings

```
int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm
     newcomm)
MPI_COMM_SPLIT (comm, color, key, newcomm, rc)
INTEGER comm, color, key, newcomm, rc
```

- Return code values

```
MPI_SUCCESS     No error; MPI routine completed successfully.
MPI_ERR_COMM    Invalid communicator. A common error is to use a
                null communicator in a call
MPI_ERR_INTERN  This error is returned when some part of the
                implementation is unable to acquire memory
```

First five MPI commands

- C/C++ bindings
  - int **MPI_Init**(int *argc, char **argv)
  - int **MPI_Comm_size**(MPI_Comm comm, int *size)
  - int **MPI_Comm_rank**(MPI_Comm comm, int *rank)
  - int **MPI_Barrier(**MPI_Comm comm**)**
  - **MPI_Finalize()**
- Fortran bindings
  - **MPI_INIT**(ierror)
  - **MPI_COMM_SIZE**(comm, size, ierror)
  - **MPI_COMM_RANK**(comm, rank, ierror)
  - **MPI_BARRIER(**comm, ierror**)**
  - **MPI_FINALIZE(**ierror**)**
  - integer comm, size, rank, ierror

Writing MPI-program

- Include MPI header files
- C
  - #include <mpi.h>
- Fortran
  - INCLUDE 'mpif.h'
- Call MPI_Init()
- Write the actual program
- Call MPI_Finalize() before exiting from the main program

Summary

- In MPI, a set of independent processes is launched
- Processes are identified by **rank**
- Data is always local to the process
- Processes can exchange data by sending and receiving messages
- MPI library contains functions for
  - obtaining information about the communicator (MPI_Comm_size, MPI_Comm_rank)
  - synchronizing (MPI_Barrier)
  - communication between processes