ICHEC
Irish Centre for High-end Computing

MPI: Day 4

Head to your cloned repository on Kay, then type `git pull`

ICHEC
Irish Centre for High-end Computing

Introduction to MPI

Instructor: Christopher Werner

Today's timetable

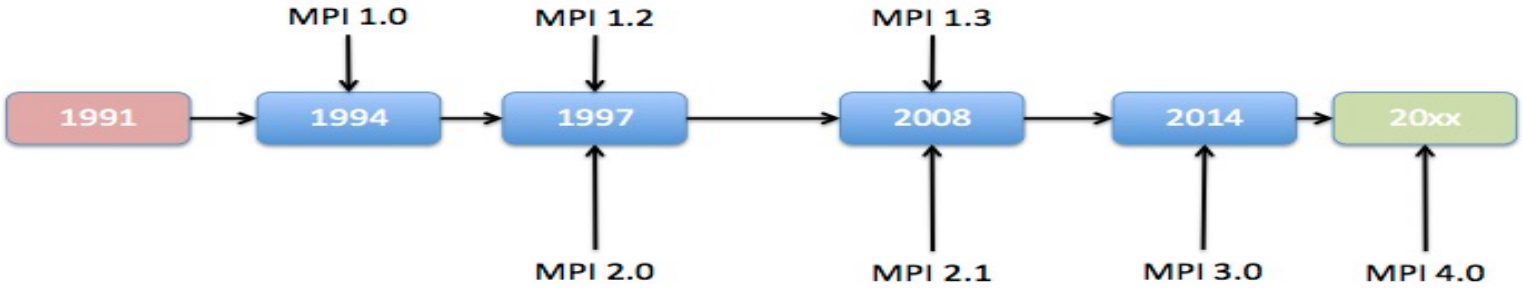
Time (CEST)	Topic	Speaker	Duration
09:00	Introduction to MPI and Process Model (+ hands-on)	Chris Werner	50 mins
09:50	Messages and Point-to-Point Communications (+ hands-on)	Adam Ralph	55 mins
10:45	BREAK		30 mins
11:15	Non-blocking communication (+ hands-on)	Ciarán O'Rourke	1 hour 15 mins
12:30	LUNCH		1 hour
13:30	Collective communication (+ hands on)	Ciarán O'Rourke	1 hour
14:30	Groups and Communicators Part 1	Adam Ralph	30 mins
15:00	BREAK		30 mins
15:30	Groups and Communicators (+ hands on)	Adam Ralph	40 mins
16:10	Advanced MPI techniques	Ciarán O'Rourke	10 mins
16:20	Summary + Q&A	Ciarán O'Rourke	10 mins
16:30	Wrap up of the SoHPC training week	Leon Kos	30 mins

Course content based on MPI course developed by Rolf Rabenseifner (HLRS), University of Stuttgart in collaboration with EPCC

MPI (Message Passing Interface)

- Standardized message passing library specification (IEEE)
  - for parallel computers, clusters and heterogeneous networks
  - not a specific product, compiler specification etc.
  - many implementations, MPICH, LAM, OpenMPI …
- Portable, with Fortran and C/C++ interfaces.
- Many functions
- Real parallel programming
- Notoriously difficult to debug

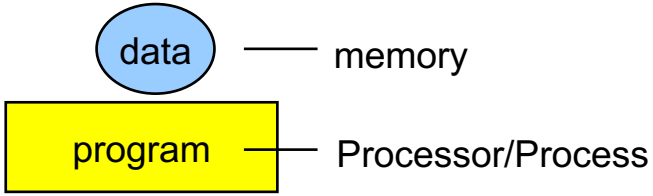
MPI Information



- **http://www.mpi-forum.org/docs/**
- **MPI: The Complete Reference**, Marc Snir and William Gropp et al, The MIT Press, 1998 (2-volume set)
- **Using MPI: Portable Parallel Programming With the Message-Passing Interface** and **Using MPI-2: Advanced Features of the Message-Passing Interface.** William Gropp, Ewing Lusk and Rajeev Thakur, MIT Press, 1999 – also available in a single volume *ISBN 026257134X.*
- **Parallel Programming with MPI**, Peter S. Pacheco, Morgen Kaufmann Publishers, 1997 - *very good introduction.*
- **https://computing.llnl.gov/tutorials/mpi/**

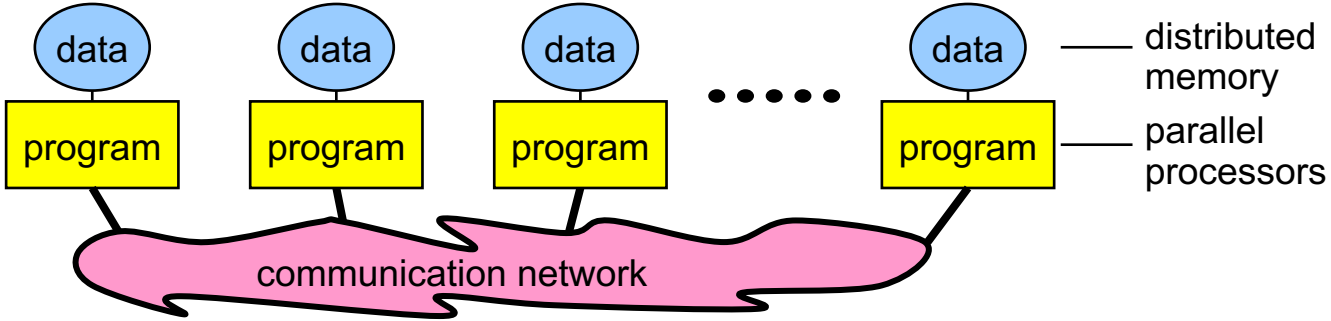
The Message Passing Programming Paradigm

- **Sequential Programming Paradigm**



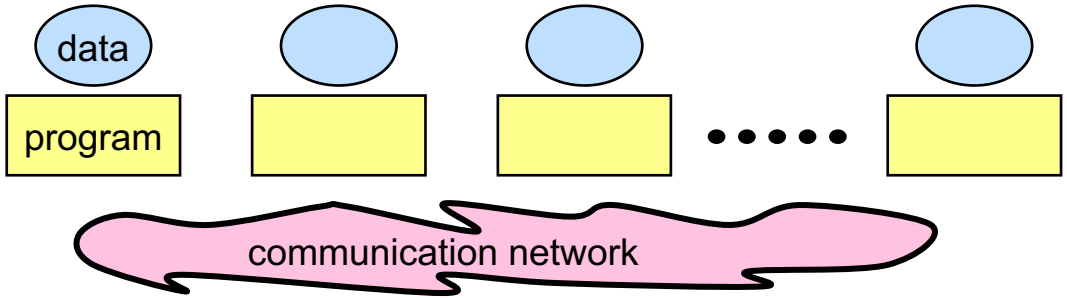
A processor may run many processes

- **Message Passing Programming Paradigm**



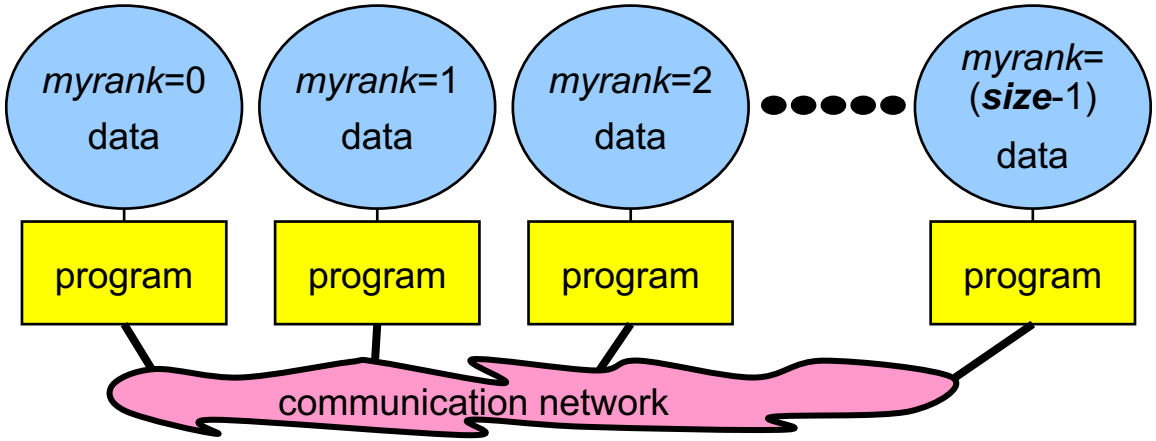
MPI Processes

- A **process** is a program performing a task on a **processor/core**
- Each processor/process in a message passing program runs a instance/copy of a ***program:***
  - written in a conventional sequential language: C/C++, Fortran, python
  - typically a single program operating on multiple dataset
  - the variables of each sub-program have
    - the same name
    - but different locations (distributed memory) and different data!
    - i.e., all variables are local to a process
  - communicate via special send & receive routines (***message passing***)



Data and Work Distribution

- To communicate together mpi-processes need identifiers: **rank = identifying number**
- all distribution decisions are based on the ***rank***
  - i.e., which process works on which data



Example: Sum of elements in a vector

- **Sequential code**

```
sum = 0
for (int i = 0; i < 1000 ;++i)
   sum = sum + array[i] ;
```

- **Parallel code**

```
sum = 0
chunkSize = 1000 / numProc
for (int i= rank*chunkSize; i< (rank + 1)*chunkSize; ++i)
         sum = sum + array[i] ;
      if( rank != root)
         send (partialsum) to root
       else
         receive(partialsum) from workers
```

On each processor

- The same program
- The same variables with different values

```
sum = 0
for (int i= 0; i < 500; ++i)
      sum = sum + array[i] ;
```

P_0

```
sum = 0
for (int i = 500; i < 1000; ++i)
      sum = sum + array[i] ;
```

P_1

What is SPMD?

- **S**ingle **P**rogram, **M**ultiple **D**ata
- Same (sub-)program runs on each processor
- MPI allows also MPMD, i.e., **Multiple** Program, ...
  - but some vendors may be restricted to SPMD
  - MPMD can be emulated with SPMD

Emulation of MPMD

- C/C++

```
main(int argc, char **argv) {
if (myrank < .... /* process should run the ocean model */){
        ocean( /* arguments */ );
     } else {
        weather( /* arguments */ );
     }
  }
```

- Fortran

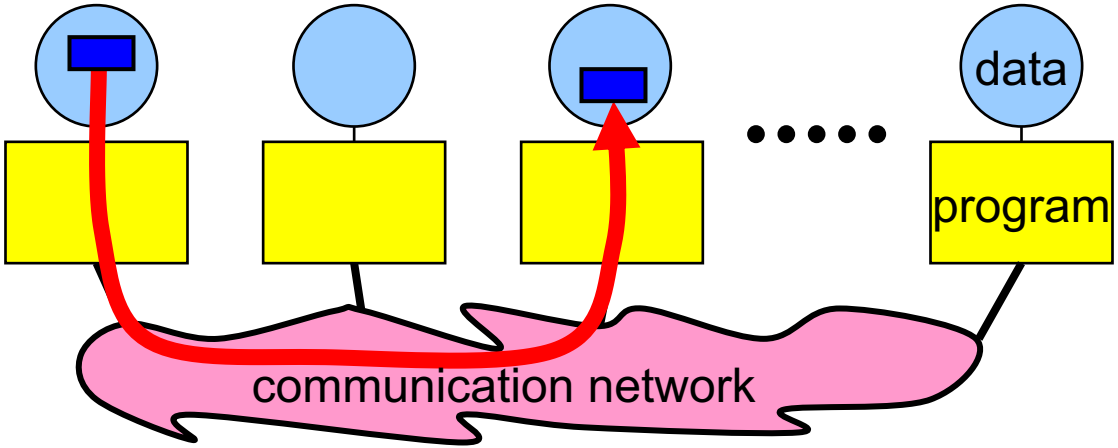
```
program forecast
if (myrank < ... ) then   !! process should run the ocean model
       call  ocean ( some arguments )
ELSE
       call  weather ( some arguments )
endif
end program forecast
```

Message Passing System

- A sub-program needs to be connected to a message passing system
- A message passing system is similar to:
  - phone line
  - mail box
  - fax machine
  - etc.
- MPI:
  - program must be linked with an MPI library
  - program must be started with the MPI startup tool

Message Passing

- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:
  - sending process
  - source location
  - source data type
  - source data size
  - receiving process (i.e., the ranks)
  - destination location
  - destination data type
  - destination buffer size



Point to Point Communication

- Simplest form of message passing.
- One process sends a message to another.
- Different types of point-to-point communication:
  - synchronous send
    - Sender sends data and waits until it gets confirmation that the message has been received
  - buffered = asynchronous send
    - Only know when the message has left

Blocking/Non-blocking Operations

- Some sends/receives may **block** until another process acts:
  - synchronous send operation **blocks until** receive is issued;
  - receive operation **blocks until** message is sent.
- Blocking subroutine returns only when the operation has completed
- Non-blocking operations return immediately and allow the sub-program to perform other work

Collective Communications

- Collective communication routines are higher level routines.
- Several processes are involved at a time.
- May allow **optimized internal** implementations, e.g., tree based algorithms
- Examples; Broadcast, Scatter, Reduction etc.

Goals and Scope of MPI

- MPI‘s prime goals
  - To provide a message-passing interface.
  - To provide source-code portability.
  - To allow efficient implementations.
- It also offers:
  - A great deal of functionality.
  - Support for heterogeneous parallel architectures.
- With MPI-2/MPI-3:
  - Important additional functionality.
  - Backward compatibility with MPI-1.

ICHEC
Irish Centre for High-end Computing

PRACE

Process model and language bindings

Headers and Function Format

- Headers
  - C

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

  - Fortran

```
include 'mpif.h'
```

    or

```
use mpi
```

- Function Format
  - C

```
error = MPI_Xxxxxx(parameter, ...);
MPI_Xxxxxx( parameter, ... );
```

  - Fortran:

```
call MPI_Xxxxxx(parameter, ..., ierror)
```

Initialising MPI

- C:

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

- Fortran:

```
MPI_Init( ierror )
integer :: ierror
```

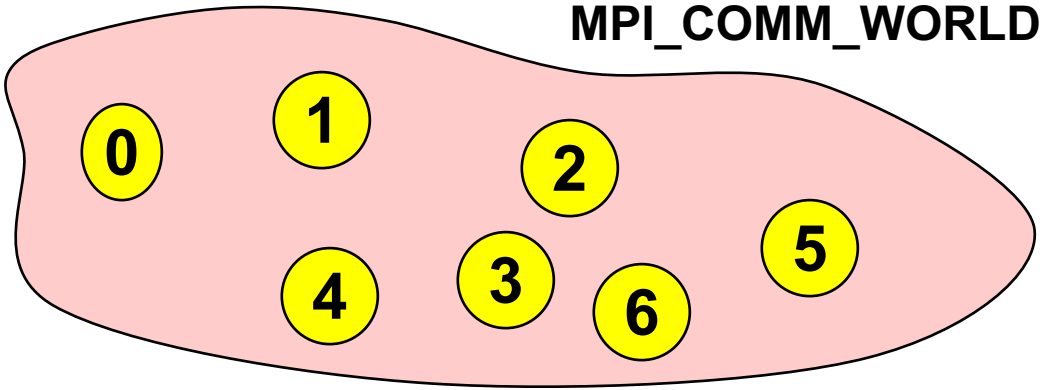
- Must be first MPI routine that is called
  - (except MPI_Initialized).

```
#include <mpi.h>
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    ....
```

```
program xxx
use mpi
implicit none
integer :: ierror
call MPI_Init(ierror)
....
```

Communicator MPI_COMM_WORLD

- All processes of an MPI program are members of the default **communicator *MPI_COMM_WORLD***.
- MPI_COMM_WORLD is a predefined **handle** in mpi.h and mpif.h.
- Each process has its own **rank** in a communicator:
  - starting with 0
  - ending with (size-1)

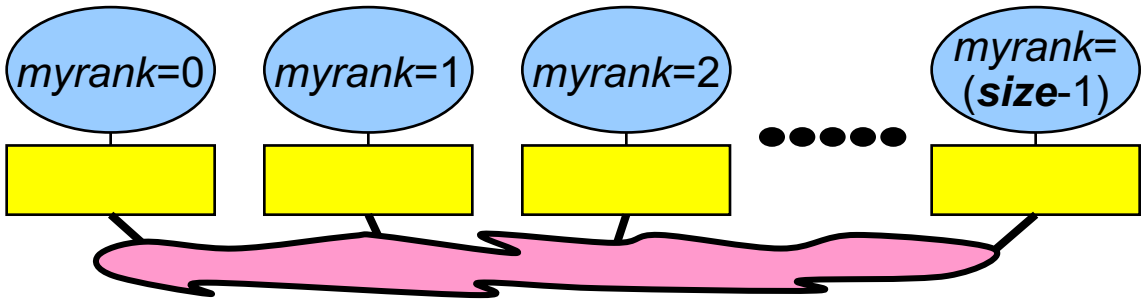


Handles

- Handles identify MPI objects.
- For the programmer, handles are
  - predefined constants in `mpi.h` or `mpif.h`
    - example: `MPI_COMM_WORLD`
    - predefined values exist only **after *`MPI_Init`*** was called
  - values returned by some MPI routines, to be stored in variables, that are defined as
    - **in Fortran: *`integer`***
    - **in C: special *`MPI typedefs`***
- Handles refer to internal MPI data structures

Rank and Size

- C: `int MPI_Comm_rank( MPI_Comm comm, int *rank)`
- Fortran:
```
MPI_Comm_rank( comm, rank, ierror)
integer :: comm, rank, ierror
```



- C: `int MPI_Comm_size(MPI_Comm comm, int *size)`
- Fortran:
```
MPI_Comm_size( comm, size, ierror)
integer :: comm, size, ierror
```

Exiting MPI

- C: `int MPI_Finalize()`
- Fortran:
```
MPI_Finalize( ierror )
        integer :: ierror
```

- **Must** be called last by all processes.
- After `MPI_Finalize`:
  - Further MPI-calls are forbidden, except `MPI_Finalized`.
  - Especially re-initialization with `MPI_Init` is forbidden

Simple C MPI implementation

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

int main(int argc, char **argv){
    int myRank, uniSize, ierror;

    ierror=MPI_Init(&argc,&argv);
    ierror=MPI_Comm_rank(MPI_COMM_WORLD,&myRank);
    ierror=MPI_Comm_Size(MPI_COMM_WORLD,&uniSize);
    printf("I am", myRank, "of", uniSize)
    ierror=MPI_Finalize();
return 0;
}
```

Simple Fortran MPI implementation

```
program testMPI
use mpi
implicit none
integer :: myRank,uniSize,ierror

call MPI_Init(ierror)
call MPI_Comm_rank(MPI_COMM_WORLD,myRank,ierror)
call MPI_Comm_Size(MPI_COMM_WORLD,uniSize,ierror)
print *, ‘I am ‘, myRank, 'of ‘, uniSize
call MPI_Finalize(ierror)
end program testMPI
```

Compilation of MPI code

- Compilation in C:
  - gcc compiler + Intel MPI: ***mpicc -o*** *prog prog.c*
  - Intel compiler + Intel MPI: ***mpiicc -o*** *prog prog.c*

- Compilation in Fortran:
  - gcc compiler + Intel MPI: ***mpif90 -o*** *prog prog.f90*
  - Intel compiler + Intel MPI: ***mpiifort -o*** *prog prog.f90*

- Executing program with *num* processes:

mpirun –np *num* ./prog

```
I am 1 of 4
I am 3 of 4
I am 0 of 4
I am 2 of 4
```

MPI Implementations

- The vendor of your computer/compiler
- MPICH
- MPI/LAM
- MPI/Pro
- openMPI
- deinoMPI