Introduction to Parallel Programming with MPI

Dr. Alireza Ghasemi, Dr. Georg Hager

Erlangen National High Performance Computing Center

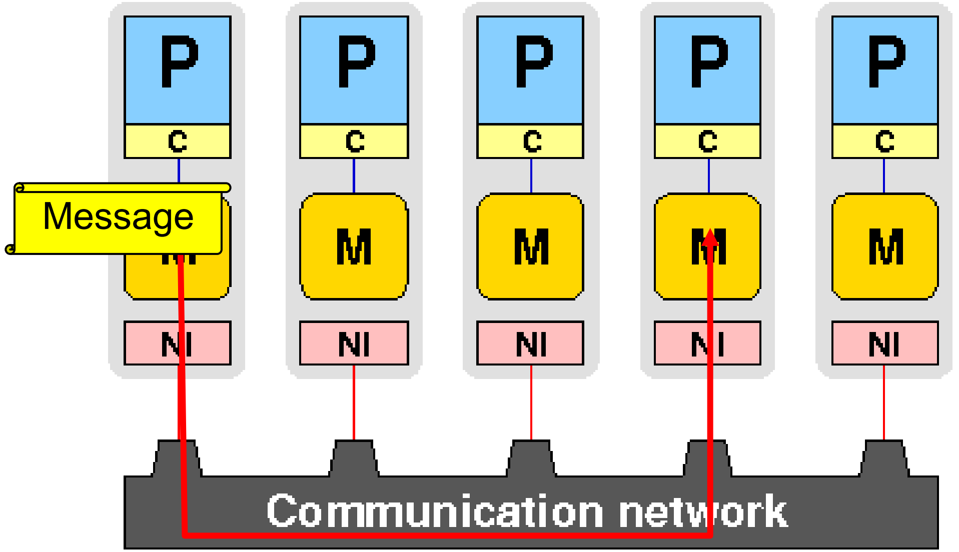
MPI: Essential Preliminaries

The message passing paradigm

Distributed-memory architecture:

Each process(or) can only access its dedicated address space.

No global shared address space



Data exchange and communication between processes is done by explicitly passing messages through a communication network

Message passing library:

- Should be flexible, efficient and portable
- Hide communication hardware and software layers from application developer

The message passing paradigm

- Widely accepted standard in HPC / numerical simulation: Message Passing Interface (MPI)

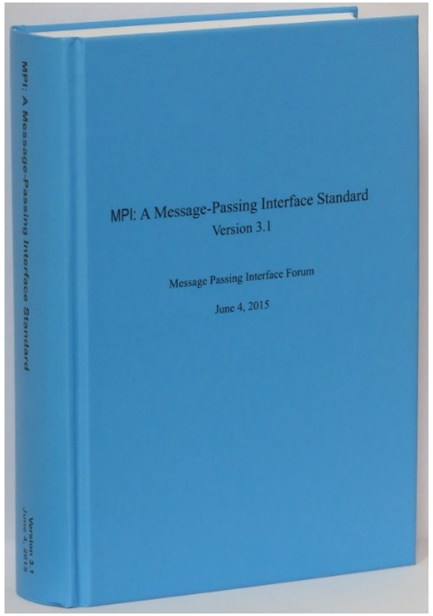
- Process-based approach: All variables are local!
- Same program on each processor/machine (SPMD)

- The program is written in a sequential language (Fortran/C[++]), but not restricted only to these two programming languages

- Data exchange between processes: Send/receive messages via MPI library calls
  - No automatic workload distribution

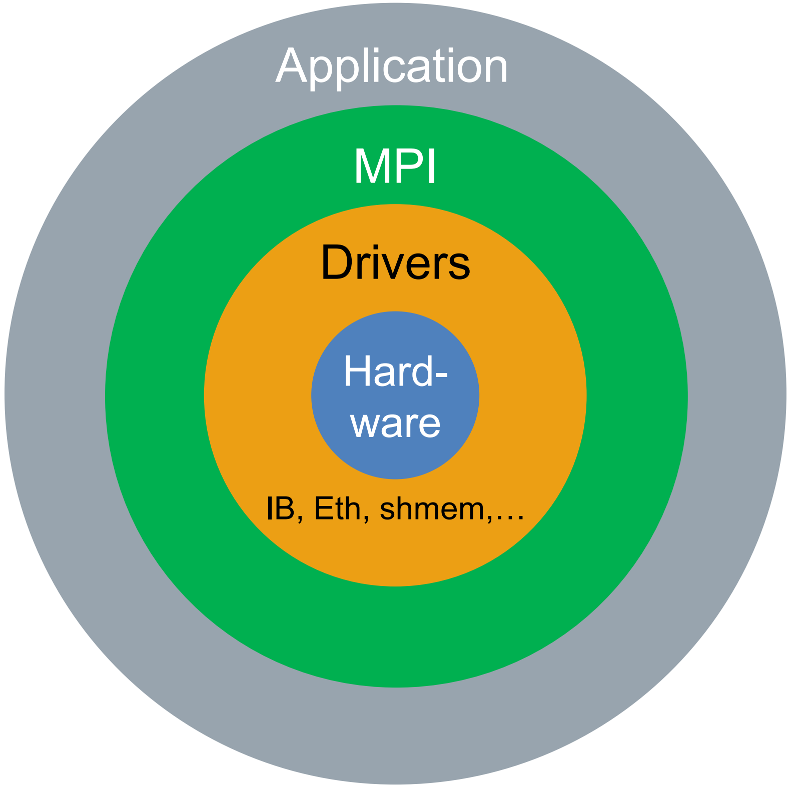
The MPI standard

- MPI forum – defines MPI standard / library subroutine interfaces
- Latest standard in use: MPI 3.1 (2015), 868 pages
  - MPI-4.1 was approved by the MPI Forum on 02.11.2023
- Members (approx. 60) of MPI standard forum
  - Application developers
  - Research institutes & computing centers
  - Manufacturers of supercomputers & software designers
- Successful free implementations (MPICH, mvapich, OpenMPI) and vendor libraries (Intel, Cray, HP,…)
- Documents: http://www.mpi-forum.org/

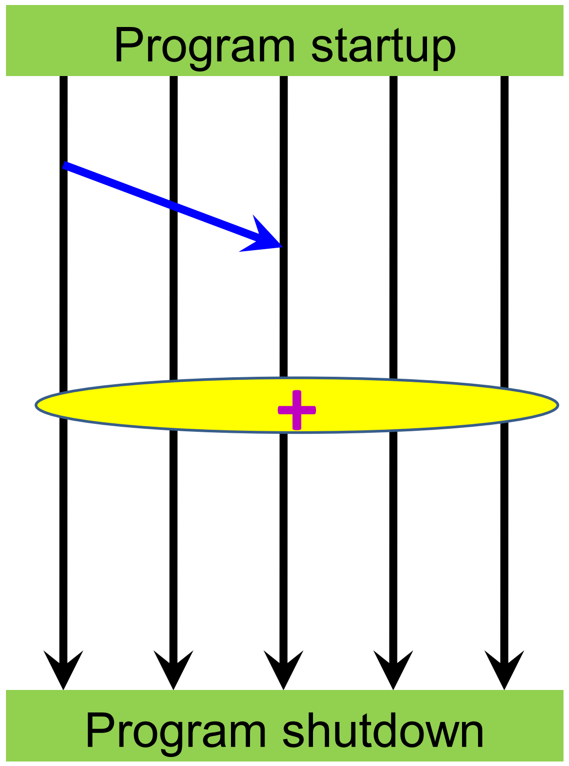

MPI: A Message-Passing Interface Standard
Version 3.1
Message Passing Interface Forum
June 4, 2015
MPI: A Message-Passing Interface Standard
Version 3.1
June 4, 2015

MPI goals and scope

- Portability is main goal: architecture- and hardware-independent code
- Fortran and C interfaces (C++ deprecated)
- Features for supporting parallel libraries
- Support for heterogeneous environments (e.g., clusters with compute nodes of different architectures)



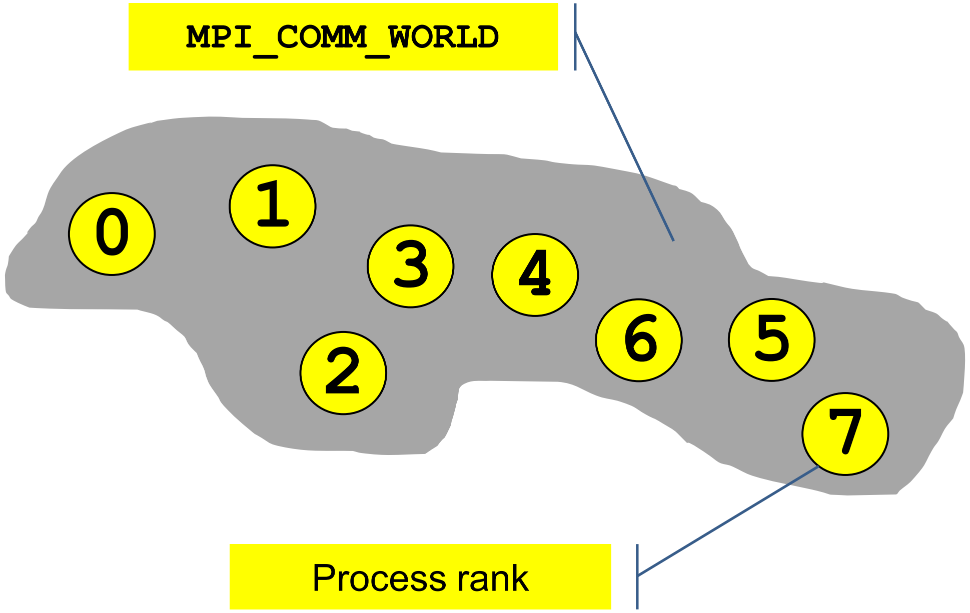
Parallel execution in MPI



- Processes run throughout program execution
- MPI startup mechanism:
  - launches tasks/processes
    - think of executing multiple copies of a program
  - establishes communication context ("communicator")
- MPI Point-to-point communication:
  - between pairs of tasks/processes
- MPI Collective communication:
  - between all processes or a subgroup
  - barrier, reductions, scatter/gather
- Clean shutdown by MPI

World communicator and rank

- Entities must be in a group/community to be able to communicate.
- Communicator is a handle
- `MPI_Init():`
  - **MPI_COMM_WORLD**
  - all processes
- **MPI_COMM_WORLD**
  - Fortran and C[++]



Initialization and finalization

- Startup of an MPI application is implementation dependent

- First call in MPI program: initialization of parallel machine

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

- Last call: clean shutdown of parallel machine

  ```
  int MPI_Finalize();
  ```

  Only “master” process is guaranteed to continue after finalize
- Stdout/stderr of each MPI process
  - usually redirected to console where program was started
  - many options possible, depending on implementation

Communicator and rank

- Communicator defines a set of processes (`MPI_COMM_WORLD`: all)

- rank: an integer identifying each process within a communicator
  - Obtain rank:
    ```
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ```
  - `rank =` 0,1,2,…, (number of processes in communicator – 1)
  - Not unique: one process may have distinct ranks in different communicators

- Obtain number of processes in communicator:
  ```
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  ```

MPI "Hello World!" in C

```
#include <mpi.h>

int main(char argc, char **argv) {
  int rank, size;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

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

  MPI_Finalize();
}
```

Never forget that these are pointers to the original variables!

Communicator required for (almost) all MPI calls

MPI “Hello World!” in Fortran

```
program hello
  use mpi
  implicit none
  integer:: rank, size, ierr
  !include "mpif.h"
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
  write(*,'(2(a,i))') &
    "Hello World! I am ",rank," of ",size

  call MPI_FINALIZE(ierr)
end program hello
```

By default, Fortran arguments are passed by reference!

Communicator required for (almost) all MPI calls

Compiling and running the code

- Compiling/linking
  - Headers and libs must be found by compiler
  - Most implementations provide wrapper scripts, e.g.,
    - `mpif77` / `mpif90`
    - `mpicc` / `mpiCC`
  - Behave like normal compilers/linkers
- Running
  - Details are implementation specific
  - Startup wrappers: `mpirun`, `mpiexec`, `aprun`, `poe`
    - `Job scheduler wrappers: srun`

```
$ mpiCC -o hello hello.cc
$ mpirun -np 3 ./hello
Hello World! I am 2 of 3
Hello World! I am 1 of 3
Hello World! I am 0 of 3
```

```
$ mpirun -np 1 ./hello :
-np 1 ./hello : -np 1
./hello
Hello World! I am 1 of 3
Hello World! I am 0 of 3
Hello World! I am 2 of 3
```

Point-to-Point Communication

It is a communication between two processes where a sender (source process) sends message to a receiver (destination process).

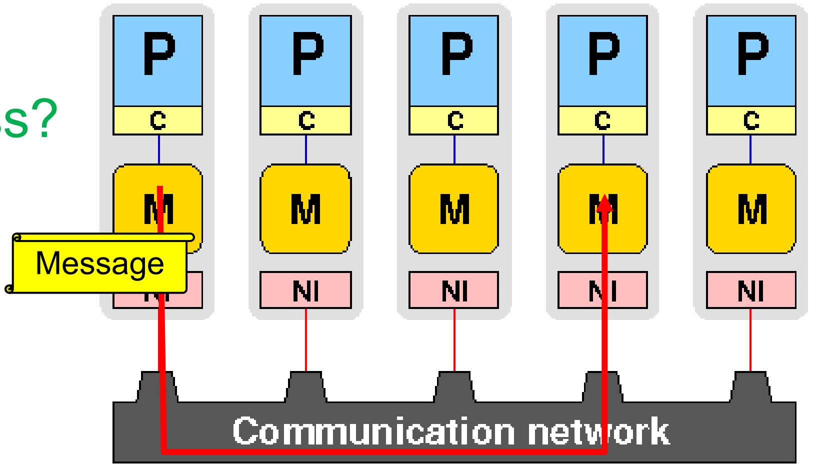
- Procedure (C/C++ binding, Fortran binding, Fortran 2008 binding)
- Message data
  - Buffer (address)
  - Datatype (basic or derived?)
  - Count (number of elements, not bytes)
- Message envelope
  - Source
  - Destination
  - Tag

Point-to-point communication: message envelope

- Which process is sending the message?
- Where is the data on the sending process?
- What kind of data is being sent?
- How much data is there?

- Which process is receiving the message?
- Where should the data be left on the receiving process?
- How much data is the receiving process prepared to accept?

- Sender and receiver must pass their information to MPI separately



MPI point-to-point communication

- Processes communicate by sending and receiving messages
- MPI message: array of elements of a particular type

rank i
sender
rank j
receiver

- Data types
  - Basic
  - MPI derived types

MPI_SEND

- C/C++ binding:

```
#include <mpi.h>
int MPI_Send(const void *buf, int count, MPI_Datatype
datatype, int dest,int tag, MPI_Comm comm)
```

- buf: address of the first entry of the buffer to be sent
- count: number of elements to be sent (note that it is not bytes!)
- datatype: type of the data
- dest: rank of the destination process within the communicator comm
- tag: nonnegative integer which is additional transferred with the message
  - Usage: the program can categorize the messages to identify one set to another.

MPI_SEND

- Fortran binding:

```
use MPI or the older form: include ’mpif.h’
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,IERROR)
   <type>    BUF(*)
   INTEGER    COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

- Fortran 2008 binding:

```
use MPI_F08
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
```

MPI_RECV

- C/C++ binding:

```
#include <mpi.h>
int MPI_Recv(void *buf, int count, MPI_Datatype
datatype,int source,
    int tag, MPI_Comm comm, MPI_Status *status)
```

- buf: address of the first entry of the buffer in which the data will be stored
  - Must be large enough
- source: rank of the source (sender) process within the communicator comm
- status: contains information about received messages, to be explained later

MPI_RECV

- Fortran binding:

```
use MPI or the older form: include ’mpif.h’
MPI_RECV(BUF,COUNT,DATATYPE, SOURCE,TAG,COMM,STATUS,IERROR)
       <type>    BUF(*)
       INTEGER    COUNT, DATATYPE, SOURCE, TAG, COMM
       INTEGER    STATUS(MPI_STATUS_SIZE), IERROR
```

- Fortran 2008 binding:

```
use MPI_F08
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
```

Quiz:

1. Which of the following is correct?
   a) There is a mechanism for automatic workload distribution in MPI
   b) MPI allows for data transfer through a communication network
   c) In MPI, workload can be split among processes according to their ranks
   d) To execute an application, MPI determines the startup procedure

2. Is the rank of a process within a communicator unique?
   a) Yes b) No

3. Does `count` in MPI_Send and MPI_Recv determine the number of bytes in the poiont-to-point communication?
   a) Yes b) No

Exercise 1: MPI “Hello World!” in C

```
#include <mpi.h>

int main(char argc, char **argv) {
   int irank, nrank;

  MPI_FIXME(FIXME,FIXME);
  MPI_Comm_FIXME(MPI_COMM_WORLD, &nrank);
  MPI_Comm_rank(FIXME, FIXME);

  printf(“Hello World! I am %d of %d\n”, irank, nrak);

  MPI_FIXME();
}
```

Never forget that these are pointers to the original variables!

Communicator required for (almost) all MPI calls

Exercise 2: calculating π using Monte Carlo method

In this exercise you practice:

1. Workload distribution
2. Eliminating repetition of work done by processes
3. Collecting results of all processes

Question: Can we improve the accuracy by increasing the number random points, i.e. $\boldsymbol{nn > 10^9}$?