PRACE Course: Intermediate MPI

9-11 November 2022

Introduction to MPI

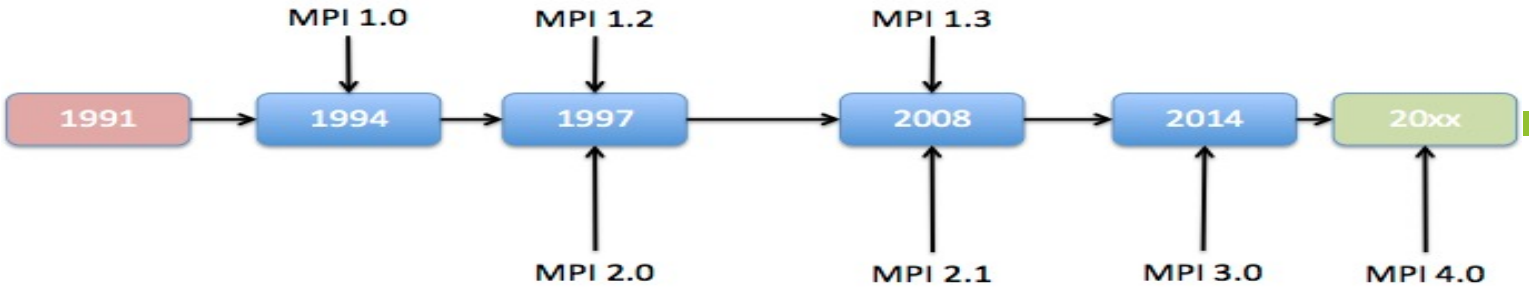
MPI (Message Passing Interface)?

- Standardized message passing library specification
  - for parallel computers, clusters and heterogeneous networks
  - not a specific product, not compiler specification etc.
  - many implementations, MPICH, LAM, OpenMPI …

- Portable, with Fortran and C/C++ interfaces.
- Many functions
- Real parallel programming
- Notoriously difficult to debug

Timeline

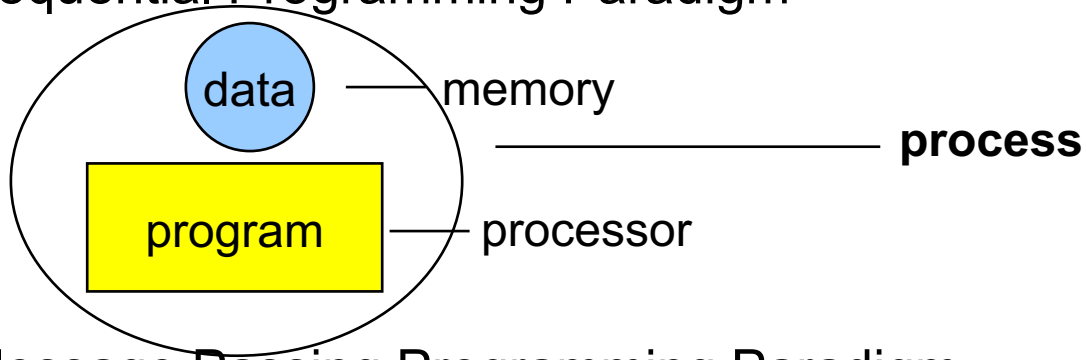
- MPI was a first attempt to define a standard set of communication calls.



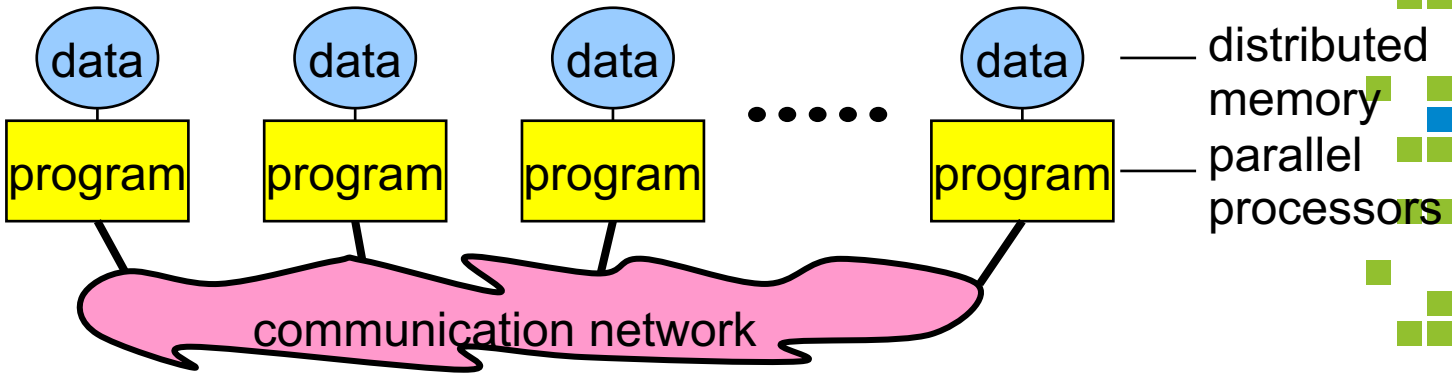
- MPI Forum: https://www.mpi-forum.org/, https://github.com/mpi-forum

The Message-Passing Programming Paradigm

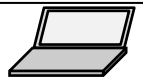
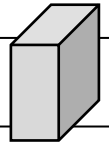
- Sequential Programming Paradigm

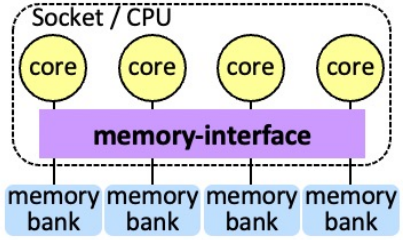


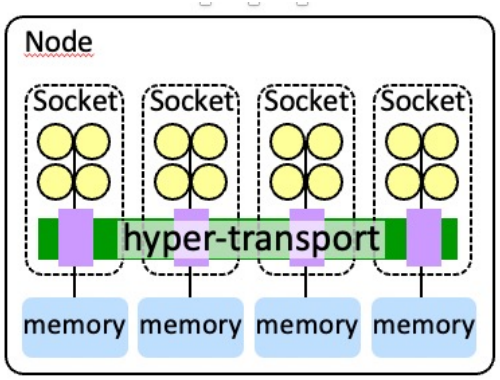
- Message Passing Programming Paradigm

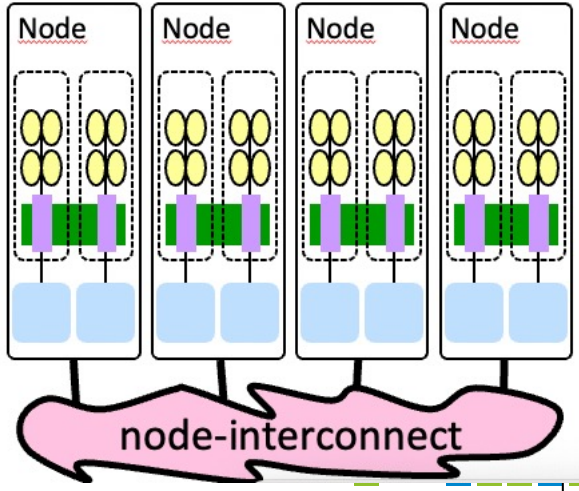


Parallel Hardware Architectures

Shared Memory Systems		Distrubited-Shared Memory Systems
Socket/CPU 	Node 	Cluster 
UMA, SMP	ccNUMA	NUMA



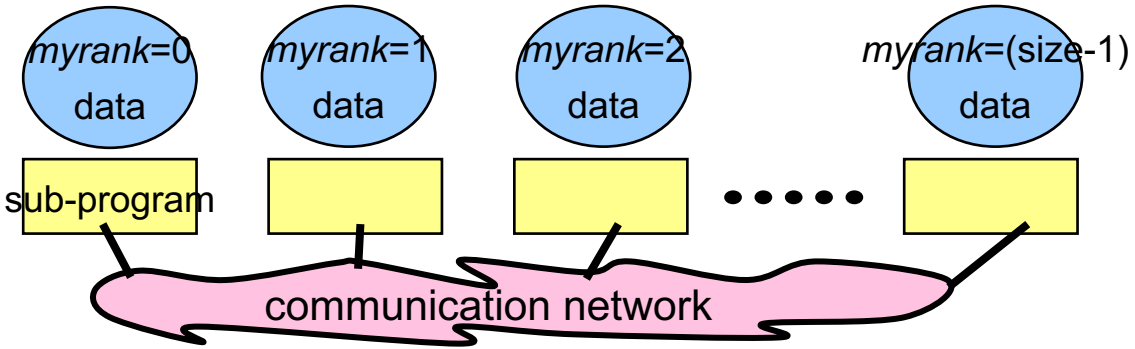




Data and Work Distribution

- **Each process** in a message passing program runs a sub-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

- To communicate together mpi-processes need identifiers: **rank = identifying number (**i.e., which process works on which data)



sum of elements of a vector

```
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

- The same program
- The same variables, but different values

SPMD

- Single Program, Multiple Data
- Same (sub-)program runs on each processor

- MPI allows also MPMD, i.e., Multiple Program Multiple Data
- but some vendors may be restricted to SPMD
- MPMD can be emulated with SPMD

```
if (myrank < .... /* process should run the ocean model */)
   ocean( /* arguments */ );
}
else {
   weather( /* arguments */ );

}
```

Example Output

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

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

  int ierror;
  int myRank,uniSize;
  int iMyName;
  char myName[MPI_MAX_PROCESSOR_NAME];

  ierror=MPI_Init(&argc,&argv);
  ierror=MPI_Comm_size(MPI_COMM_WORLD,&uniSize);
  ierror=MPI_Comm_rank(MPI_COMM_WORLD,&myRank);
  ierror=MPI_Get_processor_name(myName,&iMyName);

  printf("I am process %d out of %d running on %s\n", myRank,
uniSize, myName);

  ierror=MPI_Finalize();

  return ierror;
}
```

```
[bgursoy@login1 ~]$ mpirun -np 4 ./a.out
I am process 2 out of 4 running on login1.kay.ichec.ie
I am process 3 out of 4 running on login1.kay.ichec.ie
I am process 1 out of 4 running on login1.kay.ichec.ie
I am process 0 out of 4 running on login1.kay.ichec.ie
```

- Normally, you don't see the perfect output. Why?
- How can you print in order?

MPI Implementations

- There are many different implementations of the MPI specification.
  - **MPICH (latest: v4.0.2) (https://www.mpich.org/)**
  - **openMPI (latest: v4.1.4) (https://www.open-mpi.org/)**
  - deinoMPI
  - MPI-LAM
  - MPI-Pro
  - CHIMP-MPI
  - Sun-MPI
  - Intel-MPI

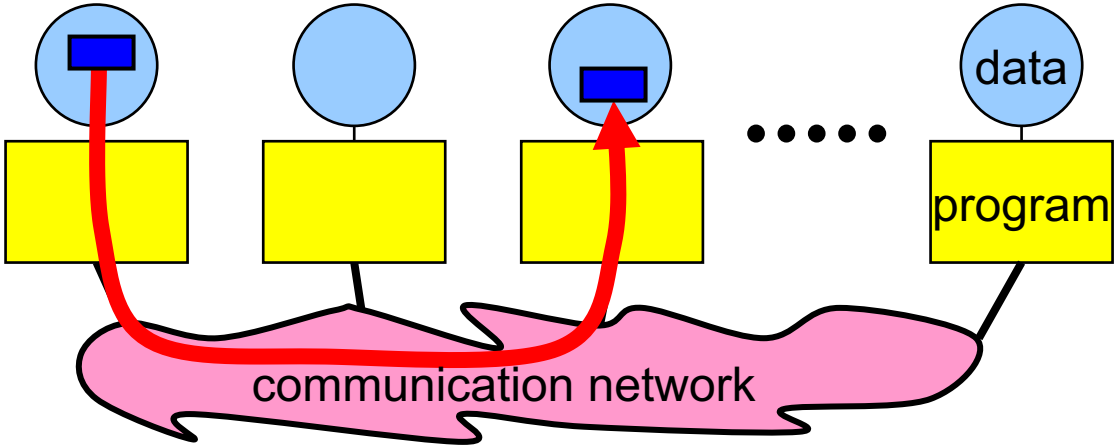
(https://www.mcs.anl.gov/research/projects/mpi/implementations.html)

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 – receiving process i.e., the ranks
  - source location – destination location
  - source data type – destination data type
  - source data size – destination buffer size



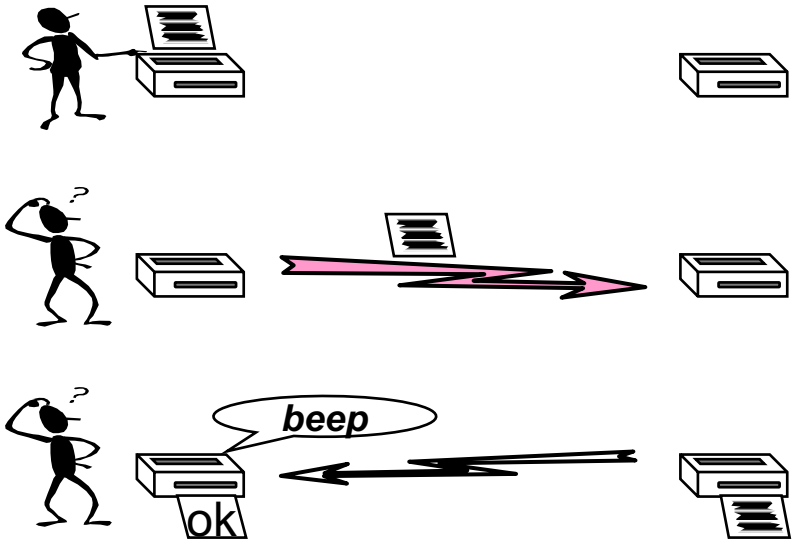
Point-to-Point Communication

- Point to point communication is the core of most MPI implementations.
- Simplest form of message passing.
- One process sends a message to another.

- Communication modes:
  - Sending a message can either be synchronous or asynchronous
  - A synchronous send is not completed until the message has started to be received
  - An asynchronous send completes as soon as the message has gone
  - All messages must be received

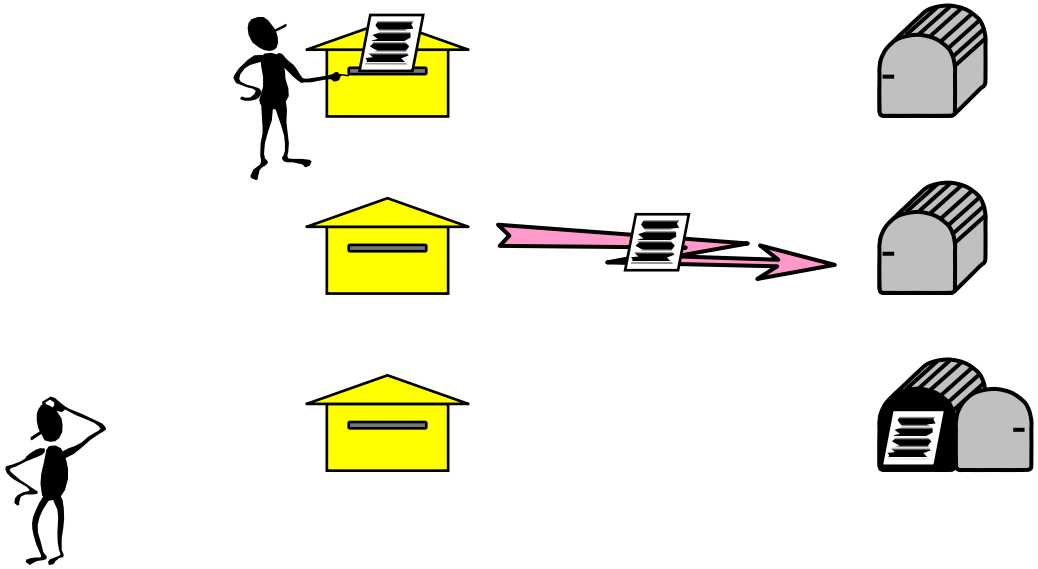
Synchronous Send

- The sender sends data and waits until it gets an information that the message is received.
- Analogy with faxing a letter. Know when letter has started to be received.



Asynchronous Send

- Only know when the message has left.
- Analogy with posting a letter. Only know when letter has been posted, not when it has been received.

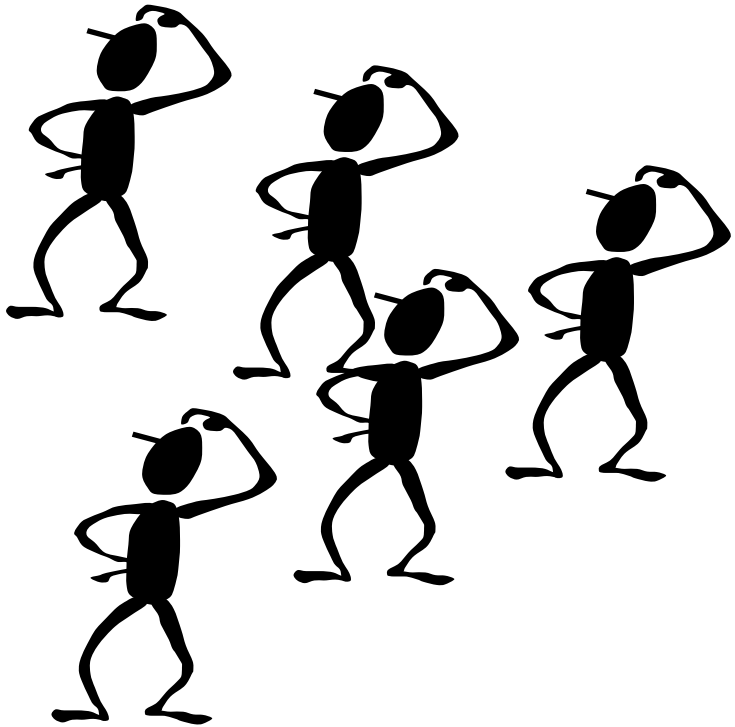


Collective Communications

- A simple message communicates between two processes.
- There are many instances where communication between groups of processes is required.
- Several processes are involved at a time.
- May allow optimized internal implementations, e.g., tree based algorithms
- Can be built out of point-to-point communications, but often implemented separately, for efficiency

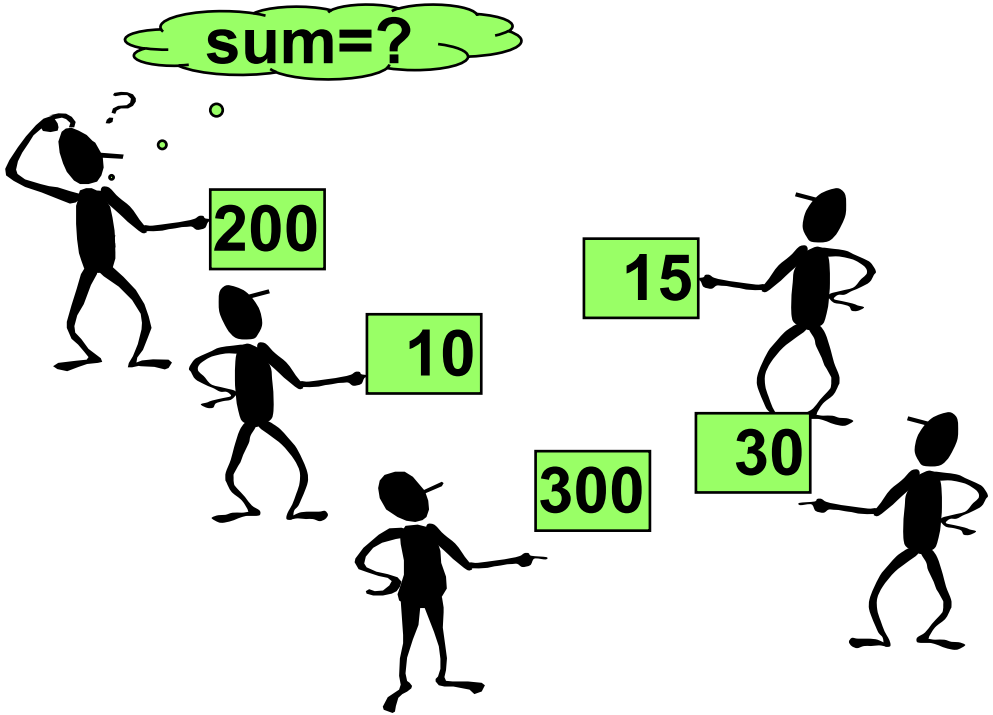
Broadcast

- A one-to-many communication.



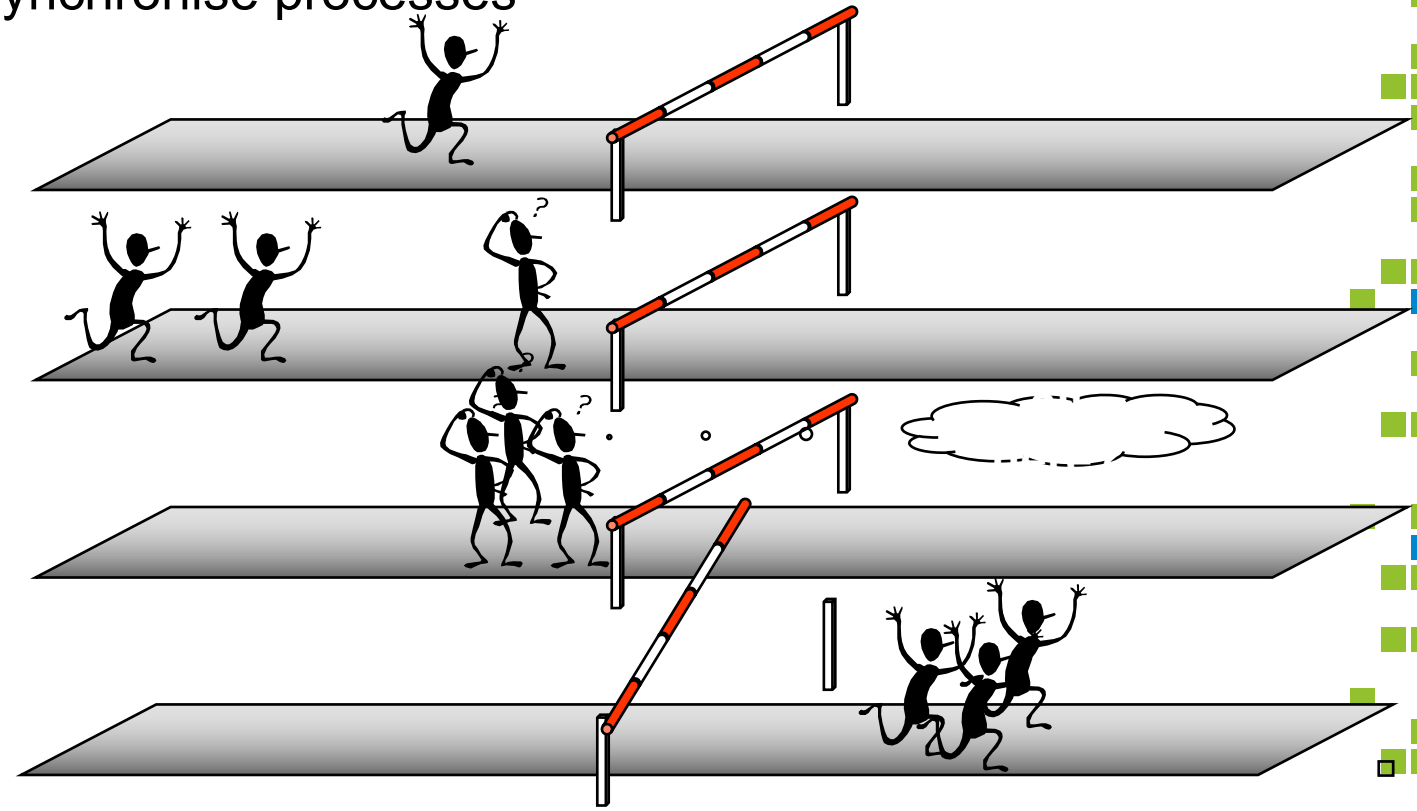
Reduction Operations

- Combine data from several processes to produce a single result.



Barriers

- Synchronise processes

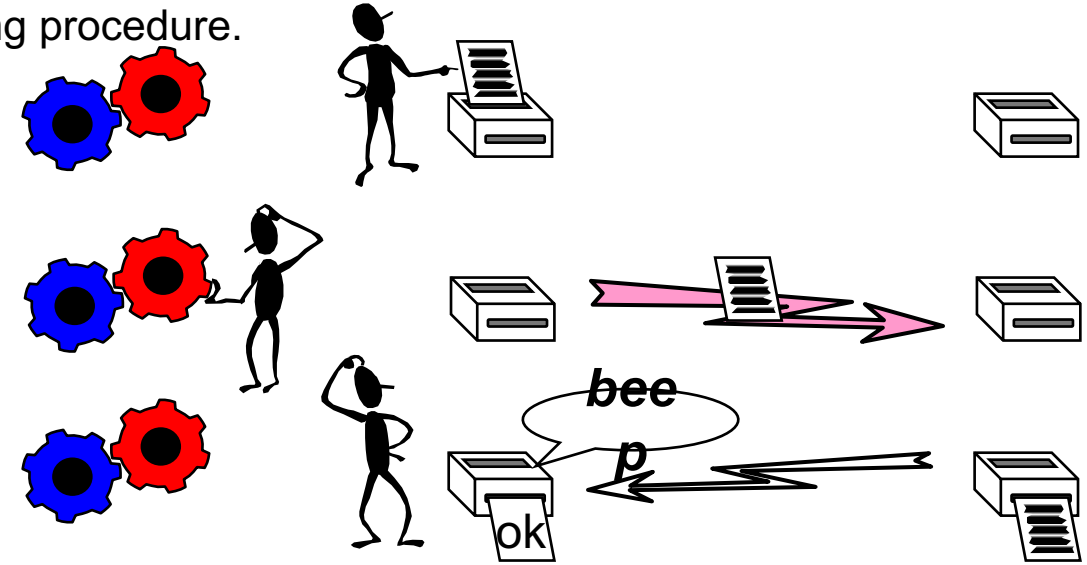


Blocking Operations

- MPI defines both blocking and non-blocking calls.
- Relates to the completion of an operation.
- 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.

Nonblocking Operations

- Non-blocking operations return immediately and allow the sub-program to perform other work.
- At some later time the sub-program must test or wait for the completion of the nonblocking operation
- A nonblocking procedure immediately followed by a matching wait is equivalent to a blocking procedure.



Scenario

- Case 1:

- Case 2:

- Case1: blocking; Case2: nonblocking
- P2 requires data from P1 for computation.

https://www.codingame.com/playgrounds/349/introduction-to-mpi/non-blocking-communications

ICHEC
Irish Centre for High-End Computing

Issues and Remarks

- Sends and receives in p2p must match
  - danger of deadlock
  - program will stall (forever!)

- Possible to write very complicated programs, but …
  - most scientific codes have a simple structure
  - often results in simple communications patterns

- Use collective communications where possible
  - may be implemented in efficient ways

Launching a Message-Passing Program

- Write a single piece of source code
  - with calls to message-passing functions such as p2p or collectives
- Compile with a standard compiler (on laptop/cluster) and link to a message-passing library provided for you
  - both open-source and vendor-supplied libraries exist
- Running is usually done via a launcher program
  - “please run N copies of my executable called program.exe”
- Run multiple copies of same executable on parallel machine
  - each copy is a separate process
  - each has its own private data completely distinct from others
  - each copy can be at a completely different line in the program

Header Files

- C

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

- Fortran

```
use mpi_f08
use mpi (or include 'mpif.h)
```

- Available since MPI-3.0
- Recommended for full consistency with Fortran standard

- MPI-2.0 interface

- Not recommended

MPI Function Format

- C:

  - case sensitive

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

- Fortran:

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

  - not case sensitive

  - ierror is optional with **only** mpi_f08 module since MPI-3.0

- MPI_..... namespace is reserved for MPI constants and routines,i.e. application routines and variable names must not begin with MPI_ .

Initialising MPI

- MPI_Init() must be called before any other MPI routine (except MPI_Initialized() and few others).

- C:

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

- Fortran:

  MPI_INIT(IERROR)
  INTEGER, OPTIONAL :: IERROR

- MPI_Init(NULL, NULL);

For MPI-2.0 and higher.

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

```
program xxx
use mpi_f08
implicit none
call MPI_Init()
....
```

Example - Initialized

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

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

  int ierror, myRank, unisize, iMyName;
  char myName[MPI_MAX_PROCESSOR_NAME];
  int flag = 0;

  ierror=MPI_Initialized(&flag);
  if (flag==0){
    printf("MPI_Initialized returned false before MPI_Init.\n");
  }

  ierror=MPI_Init(&argc,&argv);
  ierror=MPI_Comm_size(MPI_COMM_WORLD,&uniSize);
  ierror=MPI_Comm_rank(MPI_COMM_WORLD,&myRank);
  ierror=MPI_Get_processor_name(myName,&iMyName);
  printf("I am process %d out of %d running on %s.\n”,
myRank,uniSize,myName);
  ierror=MPI_Finalize();

  return ierror;
}
```

- When you run it using12 processes, how many time is the message printed?

Exiting MPI

- C:

  int MPI_Finalize()

- Fortran:

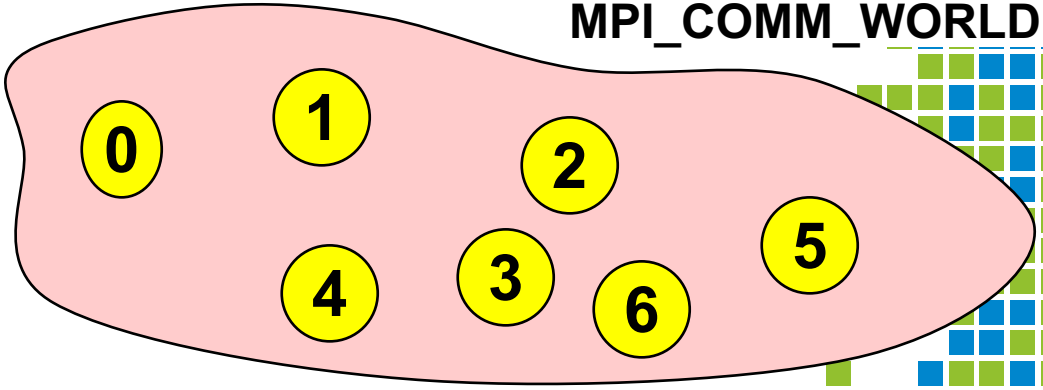
  MPI_Finalize(ierror)
  integer, optional :: ierror

  - Using mpi_f08

- Must be called last by all processes.
- User must ensure the completion of all pending communications (locally) before calling finalize
- After MPI_Finalize:
  - Further MPI-calls are forbidden.
  - Especially re-initialization with MPI_Init is forbidden

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, mpi_f08, mpi modules and mpif.h.
- Each process has its own rank in a communicator:
  - starting with 0
  - ending with (size-1)



Handles

- Handles refer to internal MPI data structures

- For the programmer, handles are
  - predefined constants in C include file mpi.h or Fortran mpi_f08 or mpi modules or mpif.h
  - example: MPI_COMM_WORLD
  - can be used in initialization expressions or assignments.
  - 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:

    mpi_f08 module TYPE(MPI_Comm) :: sub_comm

    mpi module and mpif.h. INTEGER sub_comm
  - in C:

    special MPI typedefs MPI_Comm sub_comm;

Rank and Size

- C:

  int MPI_Comm_rank(MPI_Comm comm, int *rank)

- Fortran:

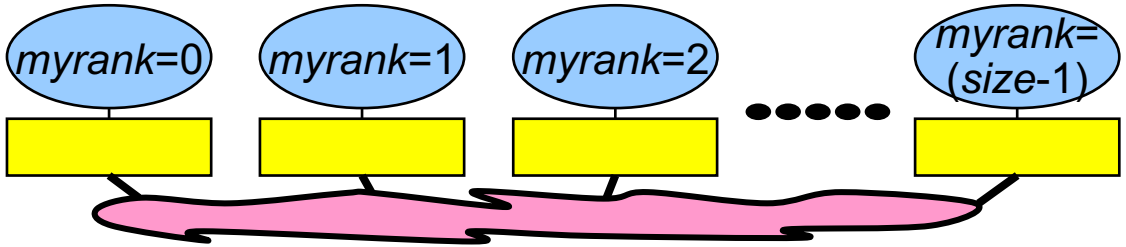
  MPI_Comm_rank(comm, rank, ierror)
  TYPE(MPI_Comm)::comm, integer :: rank; integer, optional::ierror

- C:

  int MPI_Comm_size(MPI_Comm comm, int *size)

- Fortran:

  MPI_Comm_size(comm, size, ierror)
  TYPE(MPI_Comm)::comm, integer :: size; integer, optional::ierror



Example - Hello MPI in C

```
#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;
}
```

Example - Hello MPI in Fortran

```
program testMPI
  use mpi_f08
  implicit none
  integer :: myRank,uniSize
  integer:: 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
```

Compiling the MPI Program

- C:

  mpicc -o prog prog.c

- Fortran:

  mpifort -o prog prog.f90

- Open MPI's wrapper compilers to compile MPI applications

- Intel MPI is shipped with two sets of compiler wrappers for GCC (mpicc, mpif90) and for Intel compilers (mpiicc, mpiifort)

```
[bgursoy@login1 ~]$ module load openmpi
[bgursoy@login1 ~]$ which mpicc
/ichec/packages/openmpi/intel/3.1.2/bin/mpicc
[bgursoy@login1 ~]$ ls -l /ichec/packages/openmpi/intel/3.1.2/bin/mpicc
lrwxrwxrwx. 1 marco ichec 12 Nov  3  2021 /ichec/packages/openmpi/intel/3.1.2/bin/mpicc -> opal_wrap
per
[bgursoy@login1 ~]$ mpicc -show
icc -I/ichec/packages/openmpi/intel/3.1.2/include -fexceptions -pthread -L/ichec/packages/libfabric/
1.7.1/lib -L/usr/lib64 -Wl,-rpath -Wl,/ichec/packages/libfabric/1.7.1/lib -Wl,-rpath -Wl,/usr/lib64
-Wl,-rpath -Wl,/ichec/packages/openmpi/intel/3.1.2/lib -Wl,--enable-new-dtags -L/ichec/packages/open
mpi/intel/3.1.2/lib -lmpi
[bgursoy@login1 ~]$
```

Executing the MPI Program

- Start mechanism is implementation dependent
- mpirun –np *number_of_processes* ./executable (most implementations)
- mpiexec –n *number_of_processes* ./executable (with MPI-2 and later)

- Check man mpirun & man mpiexec

```
[bgursoy@login1 ~]$ module load openmpi
[bgursoy@login1 ~]$ which mpirun
/ichec/packages/openmpi/intel/3.1.2/bin/mpirun
[bgursoy@login1 ~]$ which mpiexec
/ichec/packages/openmpi/intel/3.1.2/bin/mpiexec
[bgursoy@login1 ~]$ ls -l /ichec/packages/openmpi/intel/3.1.2/bin/mpirun
lrwxrwxrwx. 1 marco ichec 7 Nov  3  2021 /ichec/packages/openmpi/intel/3.1.2/bin/mpirun -> orterun
[bgursoy@login1 ~]$ ls -l /ichec/packages/openmpi/intel/3.1.2/bin/mpiexec
lrwxrwxrwx. 1 marco ichec 7 Nov  3  2021 /ichec/packages/openmpi/intel/3.1.2/bin/mpiexec -> orterun
[bgursoy@login1 ~]$
```

Example – get_version

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

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

  int ierror;
  int version, subversion;

  ierror=MPI_Init(&argc,&argv);
  ierror=MPI_Get_version(&version,&subversion);
  printf ("Version: Library: %d.%d, mpi.h: %d.%d\n",
           version, subversion, MPI_VERSION, MPI_SUBVERSION);
  ierror=MPI_Finalize();

  return ierror;
}
```

Fortran:

```
call
MPI_Get_version(
version,subversion,
ierror)
```

```
[bgursoy@login1 ~]$ mpirun -np 1 ./a.out
Version: Library: 3.1, mpi.h: 3.1
```

Summary

- MPI‘s prime goals
  - To provide a message-passing interface.
  - To provide source-code portability.
  - To allow efficient implementations.

- Messages are the only form of communication
  - all communication is therefore explicit

- Most systems use the SPMD model
  - all processes run exactly the same code
  - each has a unique ID
  - processes can take different branches in the same codes

- Basic communications form is point-to-point
  - collective communications implement more complicated patterns that often occur in many codes