

Message-Passing Programming with MPI

Message-Passing Concepts



Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

<http://creativecommons.org/licenses/by-nc-sa/4.0/>

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Acknowledge EPCC as follows: “© EPCC, The University of Edinburgh, www.epcc.ed.ac.uk”

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Overview

- This lecture will cover
 - message passing model
 - SPMD
 - communication modes
 - collective communications

Programming Models

Serial Programming

Concepts

Arrays	Subroutines
Control flow	Variables
Human-readable	OO

Languages

Python	C/C++
Java	Fortran
struct	if/then/else

Implementations

gcc -O3	pgcc -fast
icc	
crayftn	javac
craycc	

Message-Passing Parallel Programming

Concepts

Processes	Send/Receive
SPMD	Collectives
Groups	

Libraries

MPI

MPI_Init()

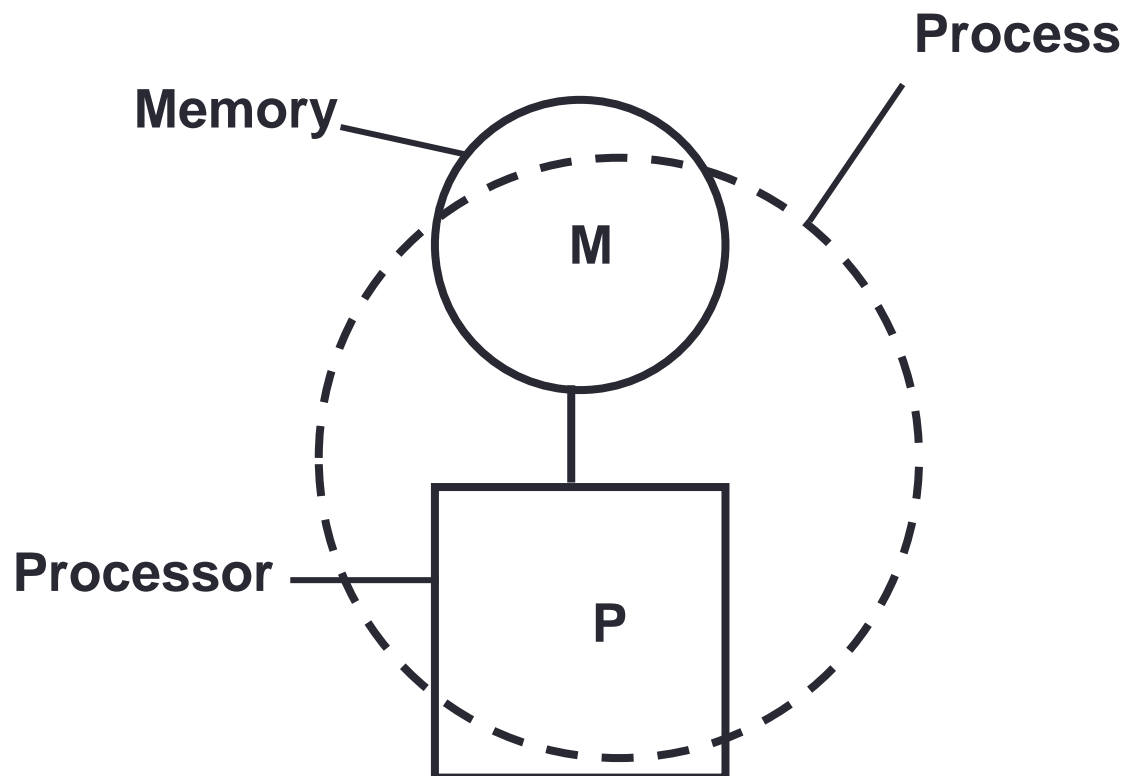
Implementations

Intel MPI	MPICH2
	Cray MPI
OpenMPI	IBM MPI

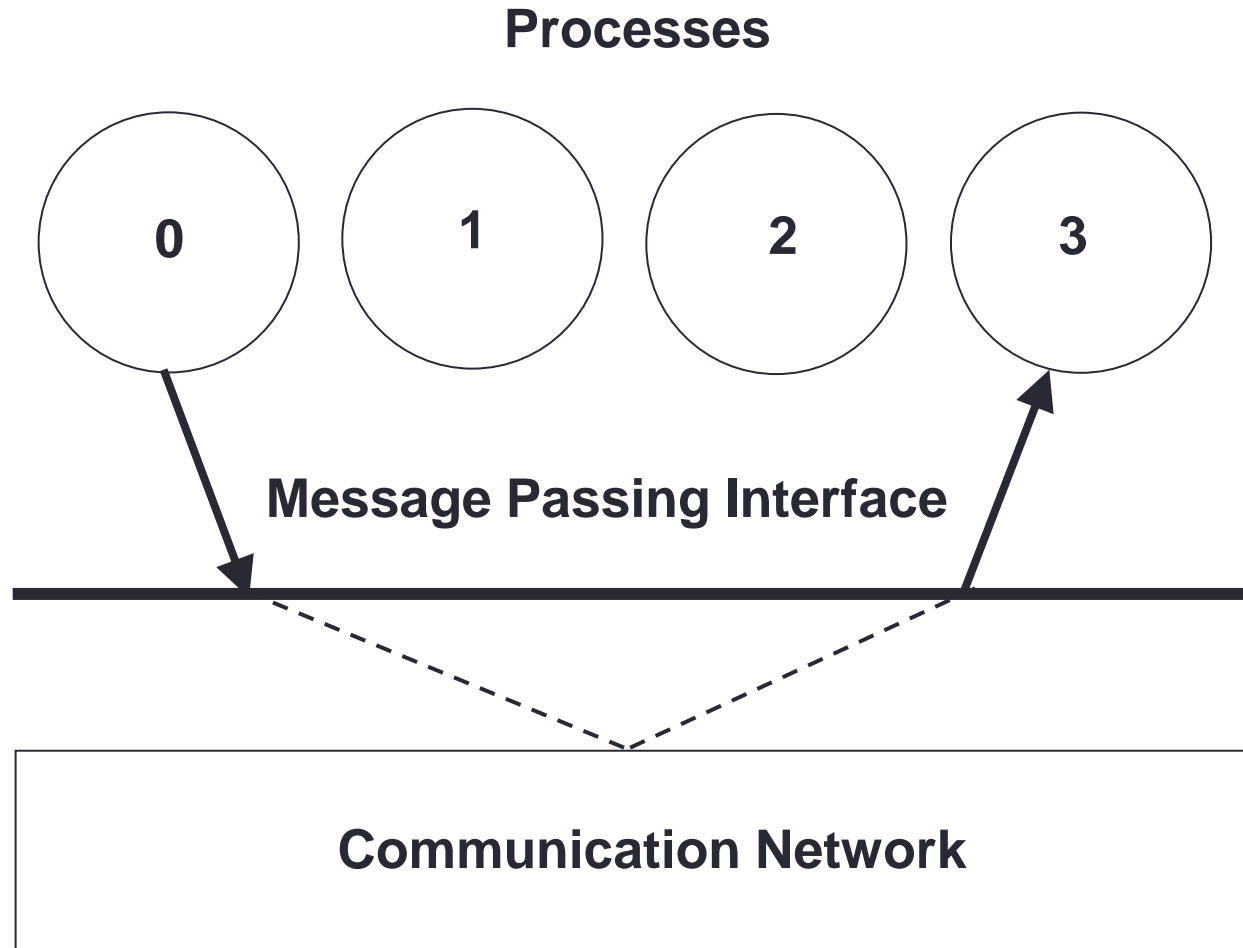
Message Passing Model

- The message passing model is based on the notion of processes
 - can think of a process as an instance of a running program, together with the program's data
- In the message passing model, parallelism is achieved by having many processes co-operate on the same task
- Each process has access only to its own data
 - ie all variables are private
- Processes communicate with each other by sending and receiving messages
 - typically library calls from a conventional sequential language

Sequential Paradigm



Parallel Paradigm

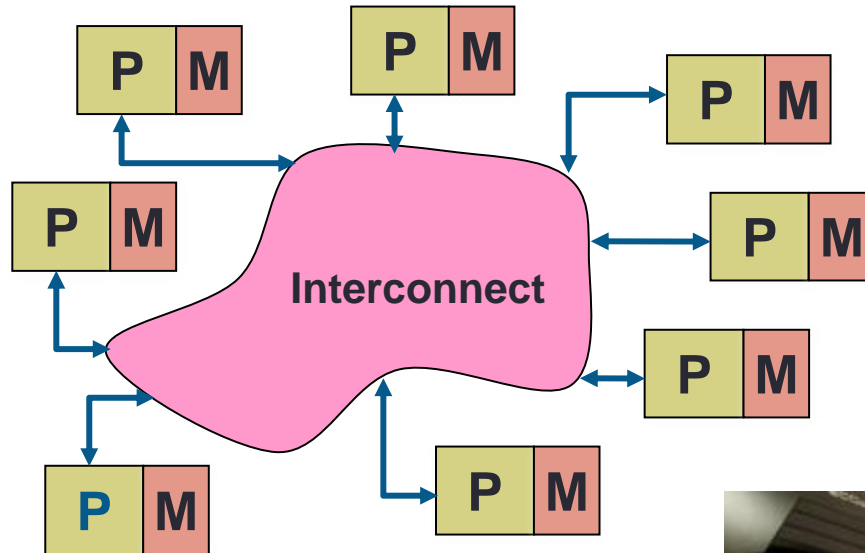


Cluster of Workstations

- Aim to run a *single* computation *across all* workstations

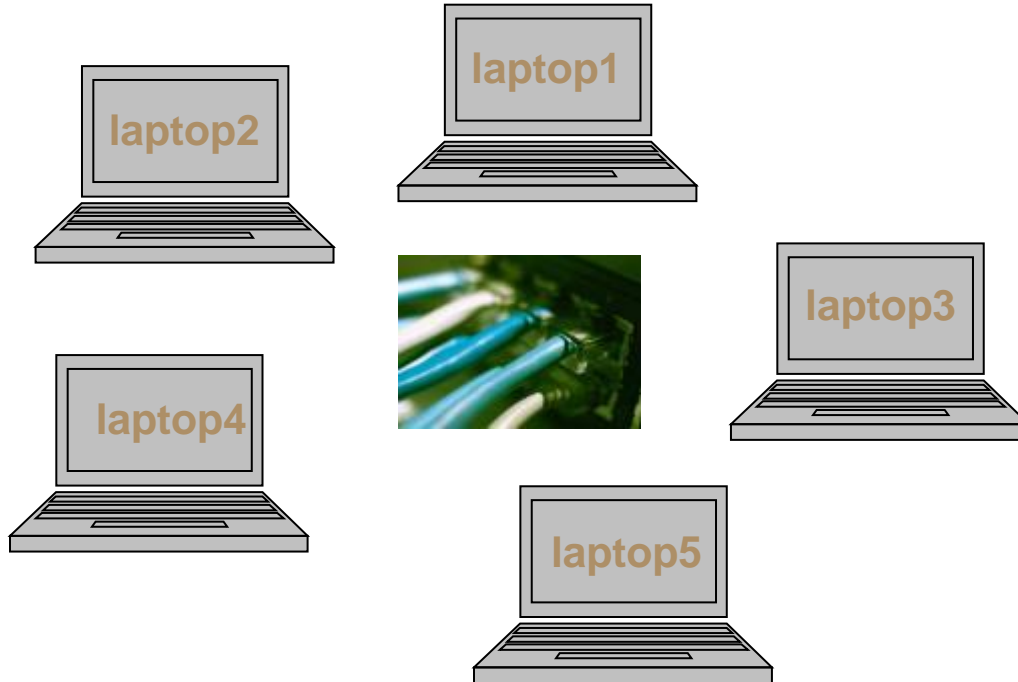


Distributed-Memory Architectures



Generic Parallel Machine

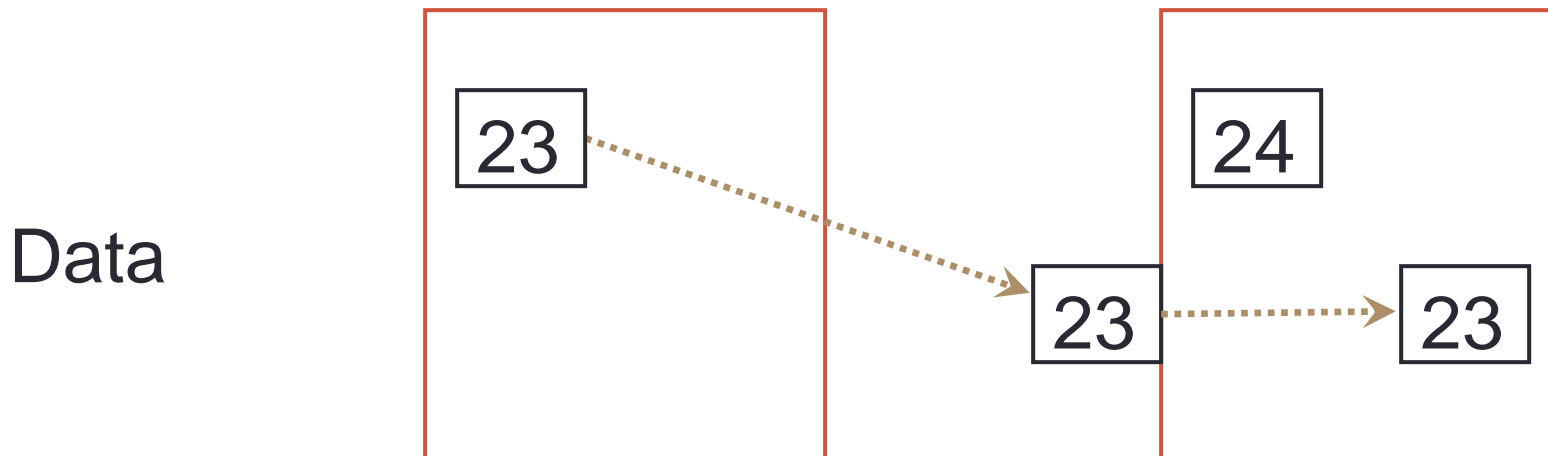
- Good conceptual model is collection of multicore laptops
 - connected together by a network



- Each laptop is called a compute node
 - each has its own operating system and network connection
- Suppose each node is a quadcore laptop
 - total system has 20 CPU-cores

Process Communication

	Process 1	Process 2
Program	$a=23$ $\text{Send}(2, a)$	$\text{Recv}(1, b)$ $a=b+1$



SPMD

- Most message passing programs use the Single-Program-Multiple-Data (SPMD) model
- All processes run (their own copy of) the same program
- Each process has a separate copy of the data
- To make this useful, each process has a unique identifier
- Processes can follow different control paths through the program, depending on their process ID
- Usually run one process per processor / core

Emulating General Message Passing (C)

```
main (int argc, char **argv)
{
    if (controller_process)
    {
        Controller( /* Arguments */ );
    }
    else
    {
        Worker      ( /* Arguments */ );
    }
}
```

Emulating General Message Passing (F)

```
PROGRAM SPMD
  IF (controller_process) THEN
    CALL CONTROLLER ( ! Arguments ! )
  ELSE
    CALL WORKER      ( ! Arguments ! )
  ENDIF
END PROGRAM SPMD
```

Messages

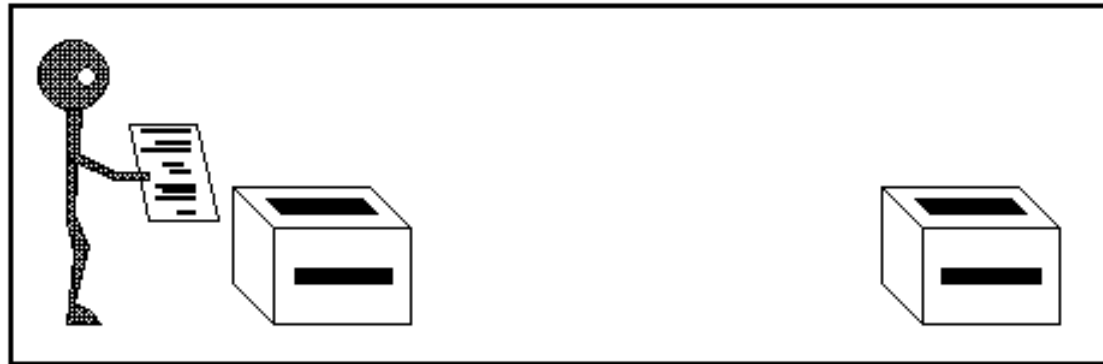
- A message transfers a number of data items of a certain type from the memory of one process to the memory of another process
- A message typically contains
 - the ID of the sending processor
 - the ID of the receiving processor
 - the type of the data items
 - the number of data items
 - the data itself
 - a message type identifier

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
- Receives are usually synchronous - the receiving process must wait until the message arrives

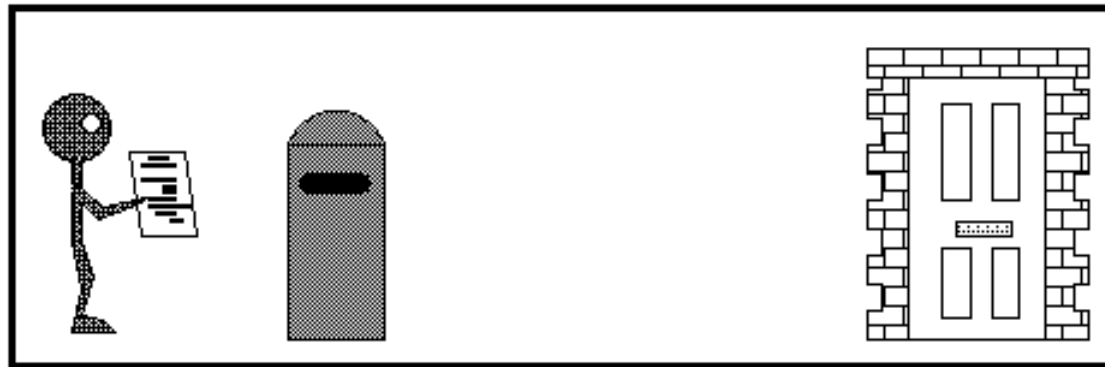
Synchronous send

- Analogy with faxing a letter.
- Know when letter has started to be received.



Asynchronous send

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



Launching a Message-Passing Program

- Write a *single piece* of source code
 - with calls to message-passing functions such as send / receive
- Compile with a *standard compiler* and link to a *message-passing library* provided for you
 - both open-source and vendor-supplied libraries exist
- 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
- Running is usually done via a launcher program
 - “please run N copies of my executable called *program.exe*”

Issues

- Sends and receives 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

Summary (i)

- 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

Summary (ii)

- Message-Passing is a programming model
 - that is implemented by MPI
 - the Message-Passing Interface is a library of function/subroutine calls
- Essential to understand the basic concepts
 - private variables
 - explicit communications
 - SPMD
- Major difficulty is understanding the Message-Passing model
 - a very different model to sequential programming

```
if (x < 0)
    print("Error");
exit;
```

A few things to note

- MPI can use both sending modes
 - MPI_Ssend : synchronous
 - MPI_Bsend : asynchronous (“Buffered” send)
- However, most people use MPI_Send
 - MPI can choose if this is synchronous or asynchronous
- Advantage
 - MPI can use the most efficient method
- Disadvantages
 - you don’t know what it’s doing – incorrect programs *may* run correctly
 - use MPI_Ssend for teaching and development: write correct programs
 - switch to MPI_Send for production: better performance

C++ Interface

- MPI is not an OO interface
 - however, can be called from C++
- Originally had different function calls, e.g.
 - `MPI::Intracomm comm;`
 - `...`
 - `MPI::Init();`
 - `comm = MPI::COMM_WORLD;`
 - `rank = comm.Get_rank();`
 - `size = comm.Get_size();`

**C++ interface is now
removed**

**Must therefore cross-call
to C**

Python

- I have (partially) added Python to this course
 - I am not a Python programmer!
- mpi4py package provides wrappers for standard library
 - mirrors the old C++ interface
 - <https://mpi4py.readthedocs.io/>
- Two versions of comms routines: upper and lower case
 - `comm.Send()` - for numpy arrays
 - `comm.send()` - for arbitray datatypes
- Using numpy arrays is critical to performance

Documentation

- ▶ MPI Standard available online
 - See: <http://www.mpi-forum.org/docs/>
 - currently version 4.0
- ▶ Available in printed form
 - <http://www.hlrs.de/mpi/mpi31/>
- ▶ Man pages available on Bridges2
 - must use the C style of naming: `man MPI_Routine_name`, e.g.:
 - `user@bridges2$ man MPI_Init`
- Great site with explanations and examples (and OpenMP)
 - www.rookiehpc.com



MPI Books

