BEAR Challenge Day 2
Message Passing

Overview

- Motivation
- Message Passing Model
- Message Passing Interface (MPI)
- Point-to-Point Communication
- Thinking in Parallel

Motivation

Yesterday

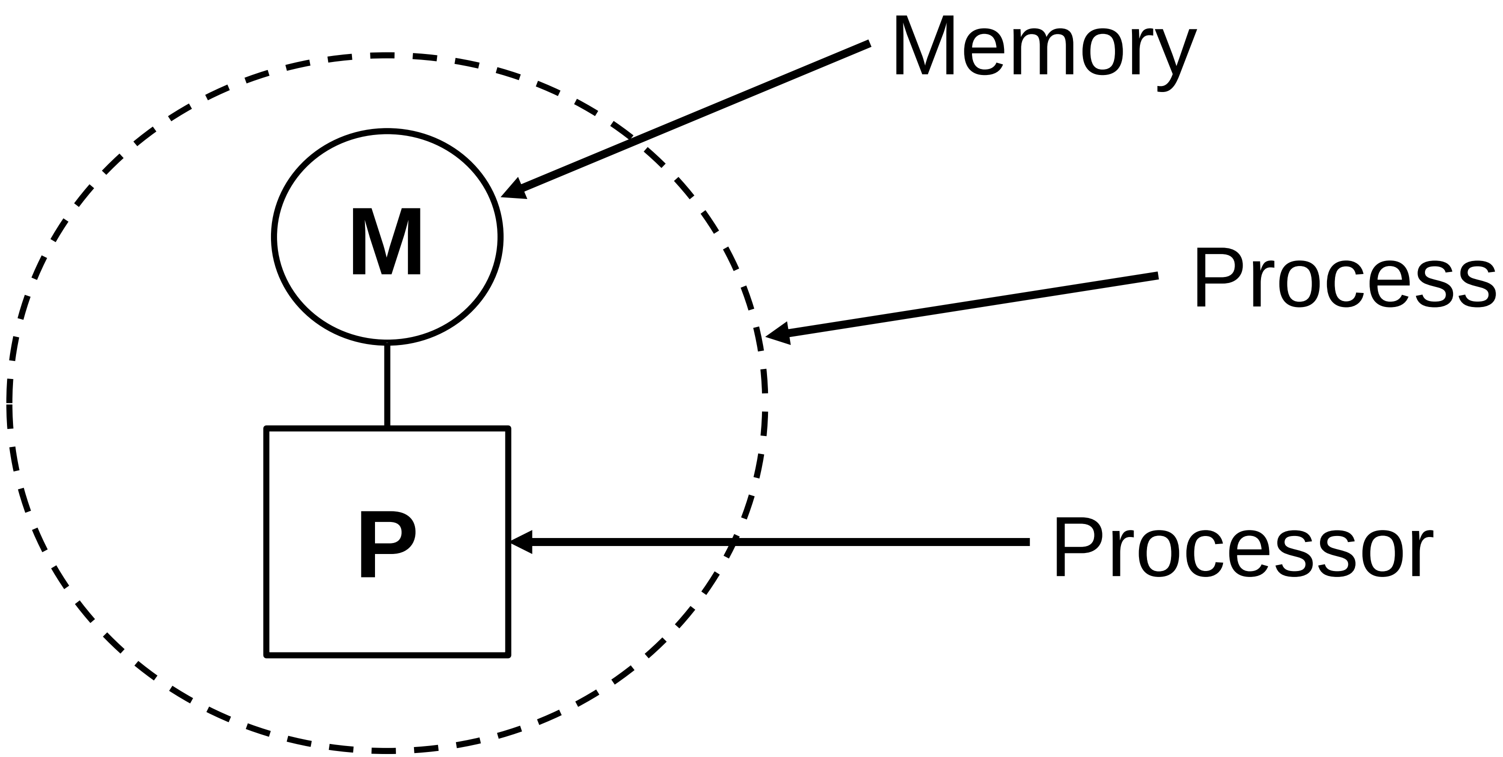
- Challenge 1 – high throughput
  - Lots of small jobs
- Bonus Challenge A – multithreaded
  - One job, using all of one node
- Challenge 2 – Cluster design
  - Interconnected nodes …

Today

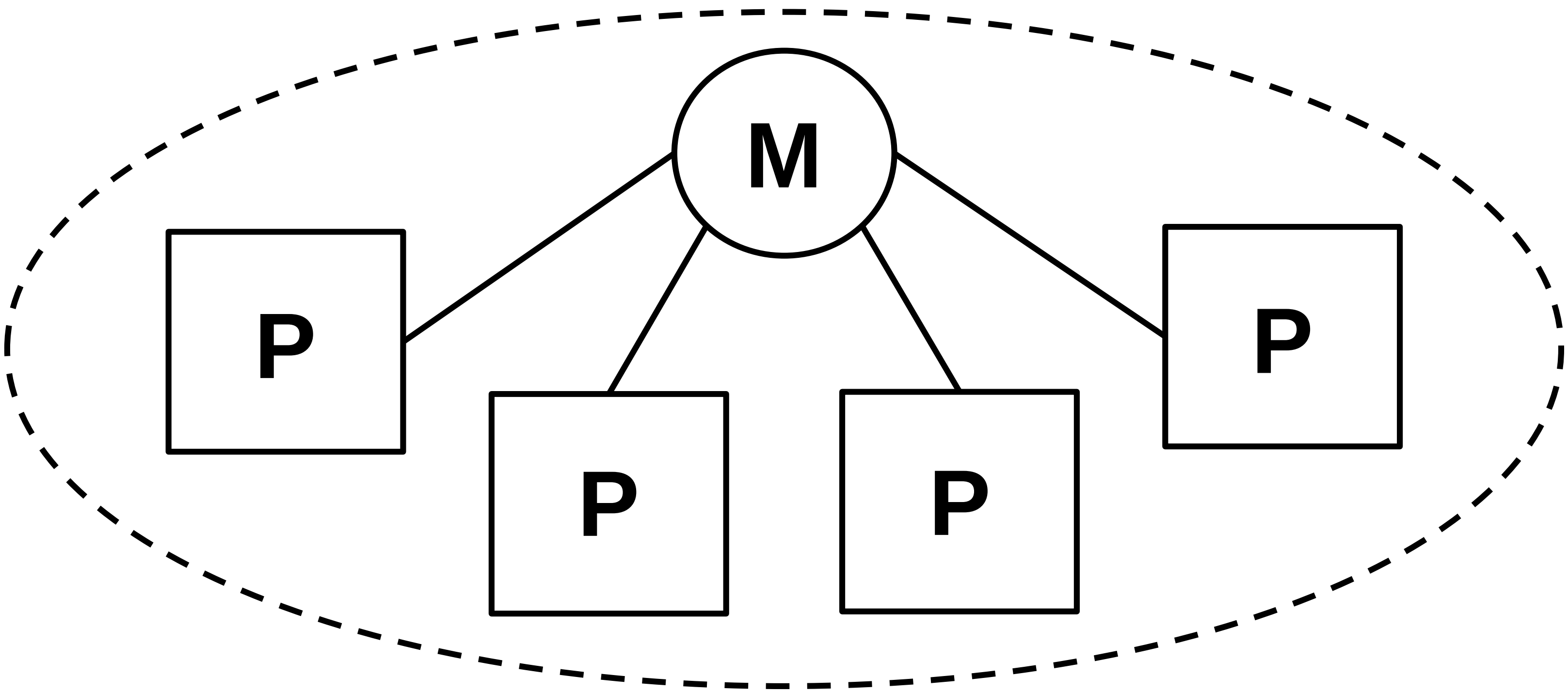
- One job
  - That uses all of your resources
- Need to coordinate the work across nodes
  - Communication

Message Passing Model (MPI)

Sequential Paradigm



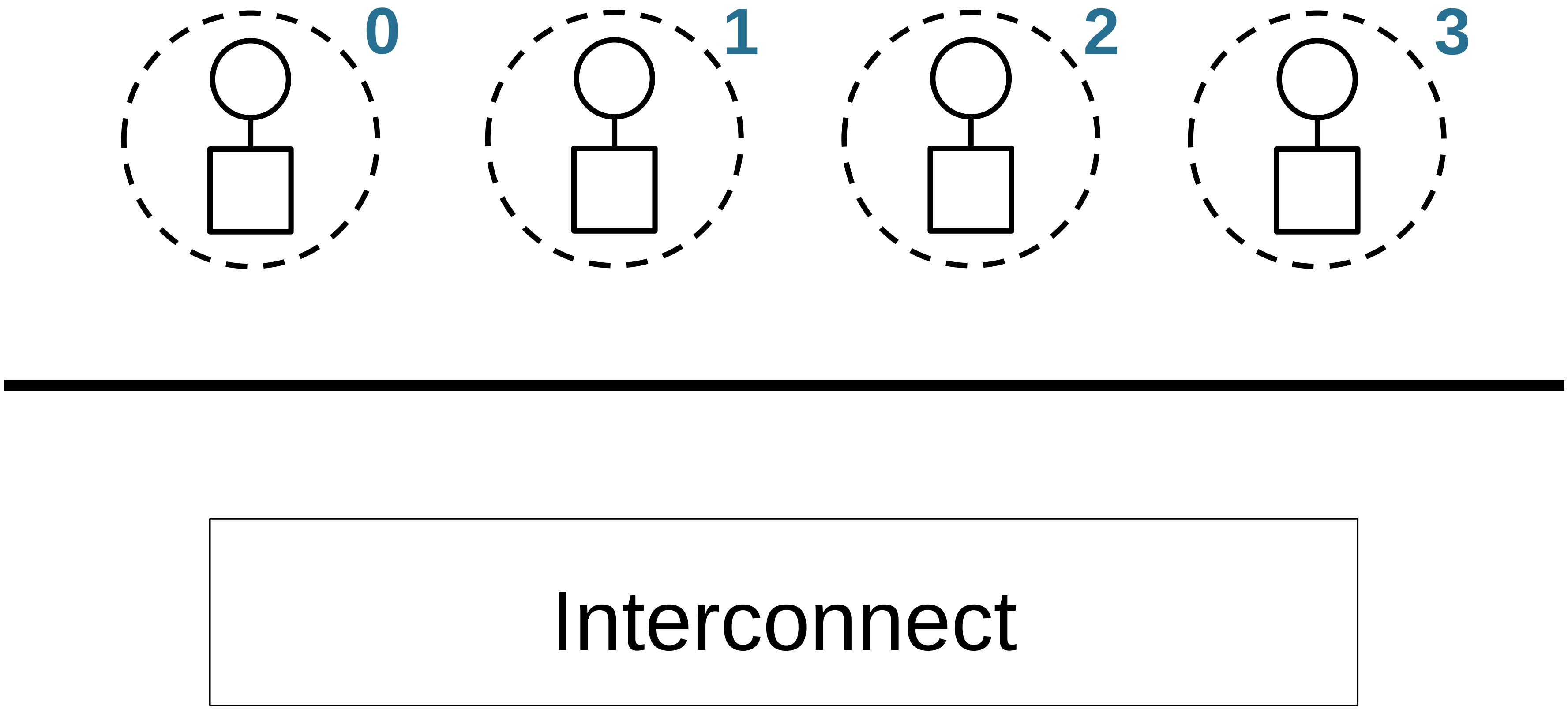
Threaded Paradigm



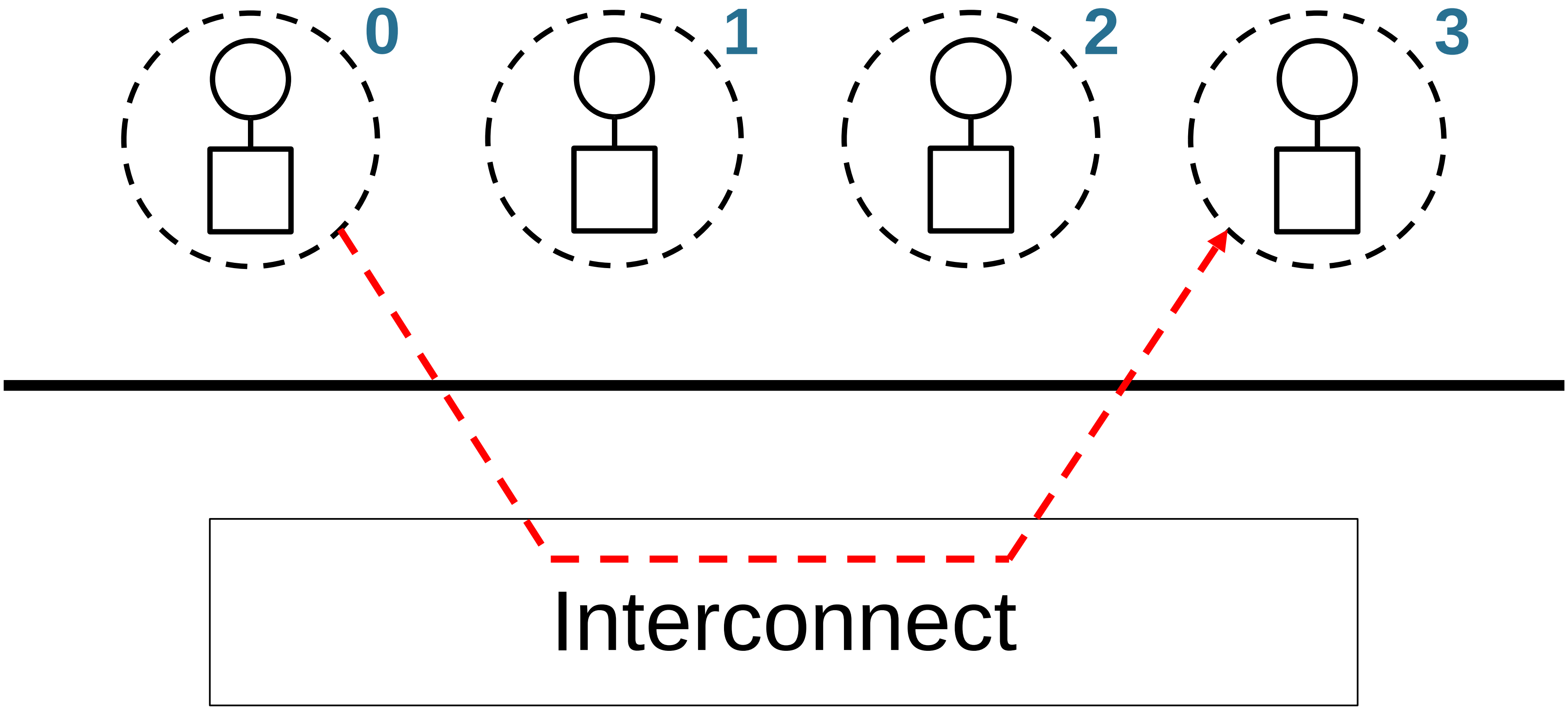
Message Passing Model

- Based on processes
  - Single instances of a running program with data
- Parallelism is achieved by coordinating work between these processes
- Each process has exclusive access to its data
- Processes communicate with messages

Message Passing Paradigm

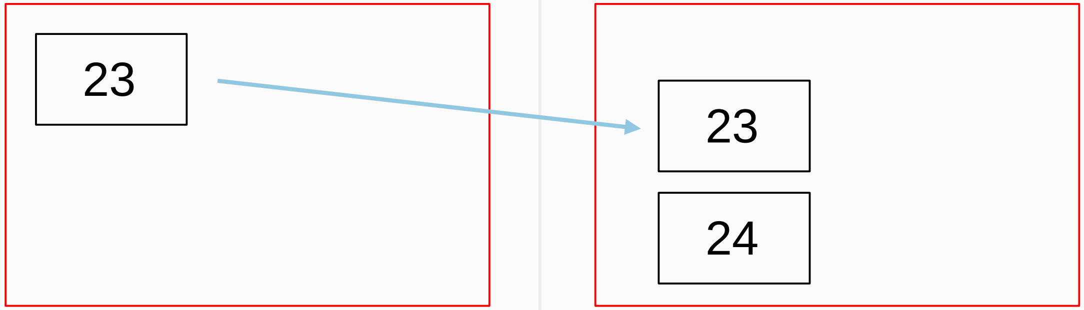


Message Passing Paradigm



Process Communication

	Process 1	**Process 2**
Program	a = 23 send(2, a)	recv(1, b) a = b+1
Data		



SPMD

- Single Program Multiple Data
- All processes run the same program
- Each has their own data
- Each process has an id
- Processes can follow different paths through the program
- Usually run one process per core

Message

- Specify
  - Who is sending
  - Who is receiving
  - Type of data
  - Amount of data
  - The data
  - Message type identifier

Point-to-point

- Simplest form
- One sender
- One receiver
- Must be matching send and receive commands

Group Communication

- More complicated
- One-to-many
  - Broadcast
- Many-to-one
  - Collect
- Some implementations and interconnects provide specialist versions of these functions

Message Passing Interface

What is MPI

- MPI is a library of functions
- We’ll be using the library in C
- Include
  - #include <mpi.h>
- Functions
  - MPI_Xxxxx(parameter, …)
- Datatypes
  - MPI_INT, MPI_FLOAT

Initialising and Finalising MPI

- Initialising
  - This must be the first MPI function call!!
  - `MPI_Init(&argc, &argv)`

- Finalising
  - This must be the last MPI function call!!
  - `MPI_Finalize()`

Who am I? How big is my world?

- A process can find its rank
  - `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`
  - Numbering is 0, 1, …, N-1

- How big is my world
  - `MPI_Comm_size(MPI_COMM_WORLD, &size)`

Hello World

- Example job, using 100 cores
- There is a hello world test job in
  - /rds/projects/2018/thompssj-bear-chal18/challenge_3/hello_world.c

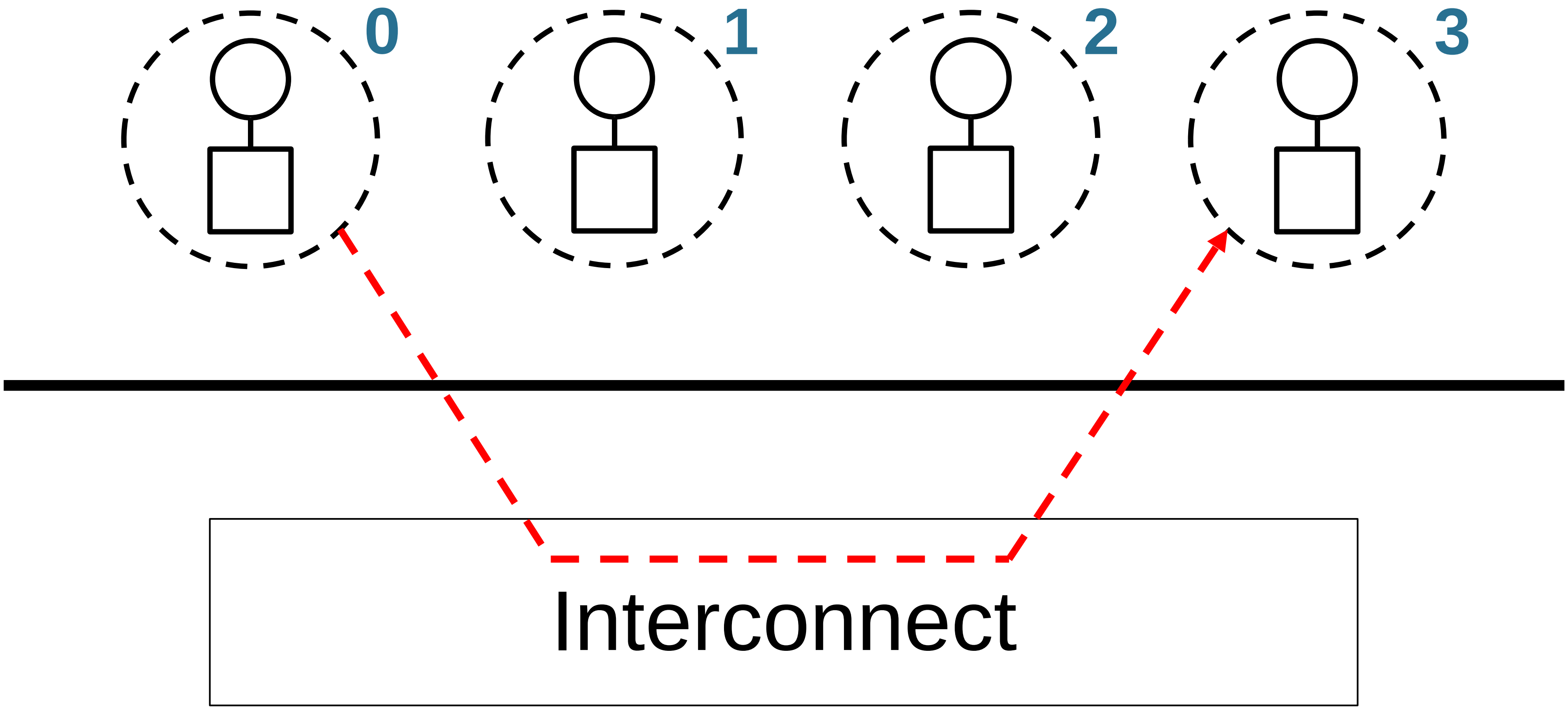
- The job script compiles and runs the code
  - /rds/projects/2018/thompssj-bear-chal18/challenge_3/run_hello_world.sh

Compiling and running

- Load the module
  - `module load iomkl/2018a`
- Compile the code
  - `mpicc hello_world.c -o hello_world`
- Run the program
  - `mpirun hello_world`

Point-to-Point Communication

Send between processes 0 and 3



Point-to-point messaging in MPI

- Sender calls a send function
  - data to be sent
  - who to send to
- Receiver calls a receive function
  - where to store the incoming data
- Both have metadata describing the message

Sending and receiving

- Send (on process 0)
  - `int x;`
  - `MPI_Ssend(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);`

- Receive (on process 3)
  - `int y;`
  - `MPI_Status status;`
  - `MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);`

Sending and receiving

- Send (on process 0)
  - `int x;`
  - `MPI_Ssend(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);`

Data

- Receive (on process 3)
  - `int y;`
  - `MPI_Status status;`
  - `MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);`

Sending and receiving

- Send (on process 0)
  - `int x;`
  - `MPI_Ssend(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);`
- Receive (on process 3)
  - `int y;`
  - `MPI_Status status;`
  - `MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);`

Amount of data

Sending and receiving

- Send (on process 0)
  - `int x;`
  - `MPI_Ssend(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);`
- Receive (on process 3)
  - `int y;`
  - `MPI_Status status;`
  - `MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);`

Datatype

Sending and receiving

- Send (on process 0)
  - `int x;`
  - `MPI_Ssend(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);`

Send to

- Receive (on process 3)
  - `int y;`
  - `MPI_Status status;`
  - `MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);`

Receive From

Sending and receiving

- Send (on process 0)
  - `int x;`
  - `MPI_Ssend(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);`
- Receive (on process 3)
  - `int y;`
  - `MPI_Status status;`
  - `MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);`

Message tag

Sending and receiving

- Send (on process 0)
  - `int x;`
  - `MPI_Ssend(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);`

- Receive (on process 3)
  - `int y;`
  - `MPI_Status status;`
  - `MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);`

Communicator

Sending and receiving

- Send (on process 0)
  - `int x;`
  - `MPI_Ssend(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);`
- Receive (on process 3)
  - `int y;`
  - `MPI_Status status;`
  - `MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);`

Status

Synchronous Blocking Message-Passing

- Processes synchronise
- Sender process specifies the synchronous mode
- Blocking
  - both processes wait until the transaction has completed

For a communication to succeed

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Message types must match
- Receiver's buffer must be large enough

MPI Datatypes

MPI datatype	C datatype	Information
MPI::INT	int	2,147,483,647
MPI::LONG	long	9,223,372,036,854,770,000
MPI::UNSIGNED	unsigned int	4,294,967,295
MPI::UNSIGNED_LONG	unsigned long	18,446,744,073,709,500,000
MPI::FLOAT	float	32 bit
MPI::DOUBLE	double	64 bit
MPI::LONG_DOUBLE	long double	implementation specific

UNIVERSITY OF BIRMINGHAM

Thinking in Parallel

Thinking in Parallel

- A process to take control
  - Often, only one process will print output
  - Imagine thousands of process output at once?!
- Repeating tasks
  - In sequential, repeating the same process is wasting cycles
  - However, in parallel having each process do a pre-processing task may be