Parallel Models

Different ways to exploit parallelism

Outline

- Shared-Variables Parallelism
  - threads
  - shared-memory architectures
- Message-Passing Parallelism
  - processes
  - distributed-memory architectures
- Practicalities
  - compilers
  - libraries
  - usage on real HPC architectures

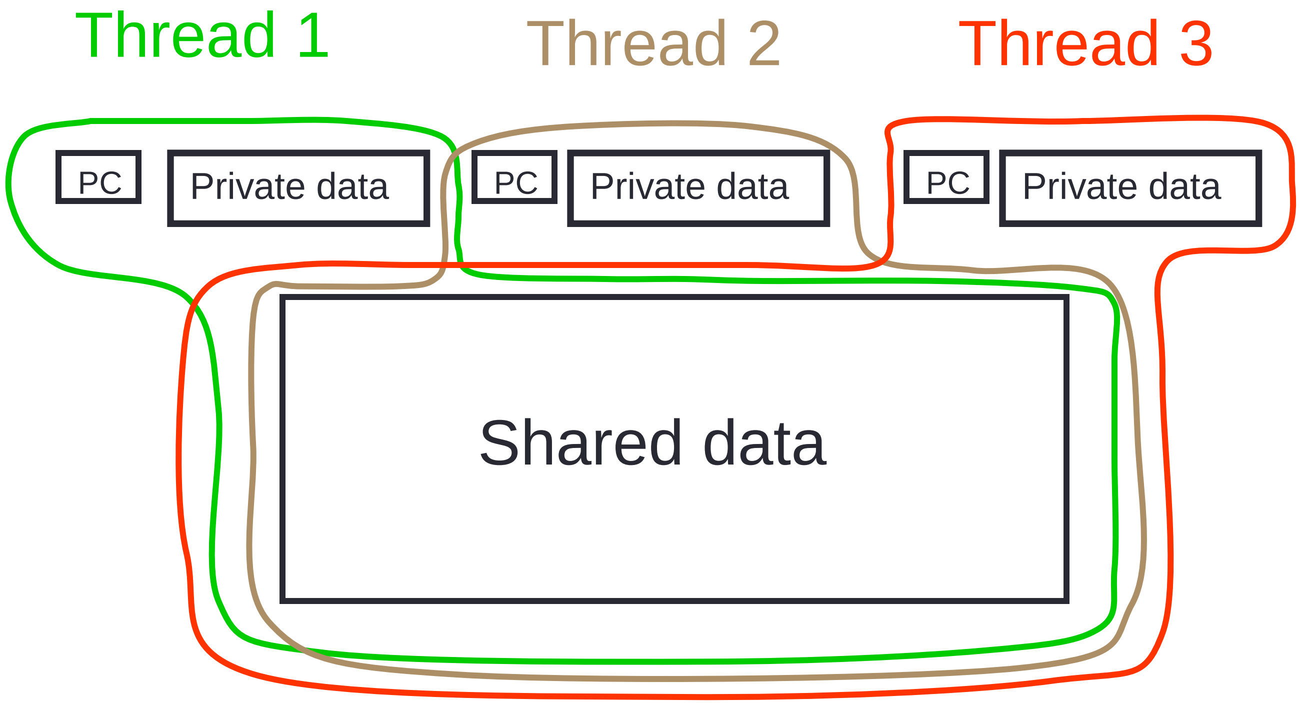
Shared Variables

Threads-based parallelism

Threaded Programming Model

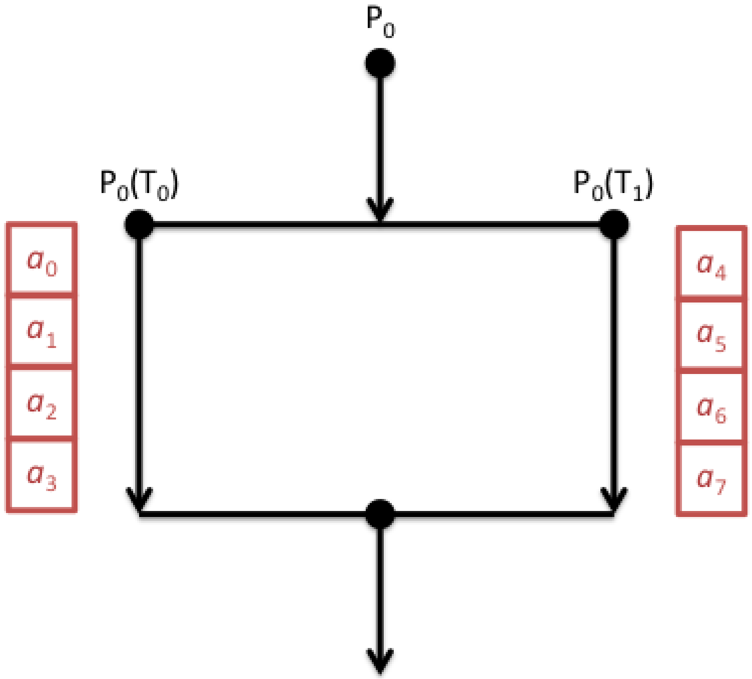
- Programming model for shared memory based on threads
  - threads are like processes, except that threads can share memory with each other (as well as having private memory)
- Shared data can be accessed by all threads
  - Private data can only be accessed by the owning thread
- Different threads can follow different flows of control through the same program
  - each thread has its own program counter
- Usually run one thread per CPU/core
  - but could be more
  - can have hardware support for multiple threads per core

Threads (cont.)



Shared-memory concepts

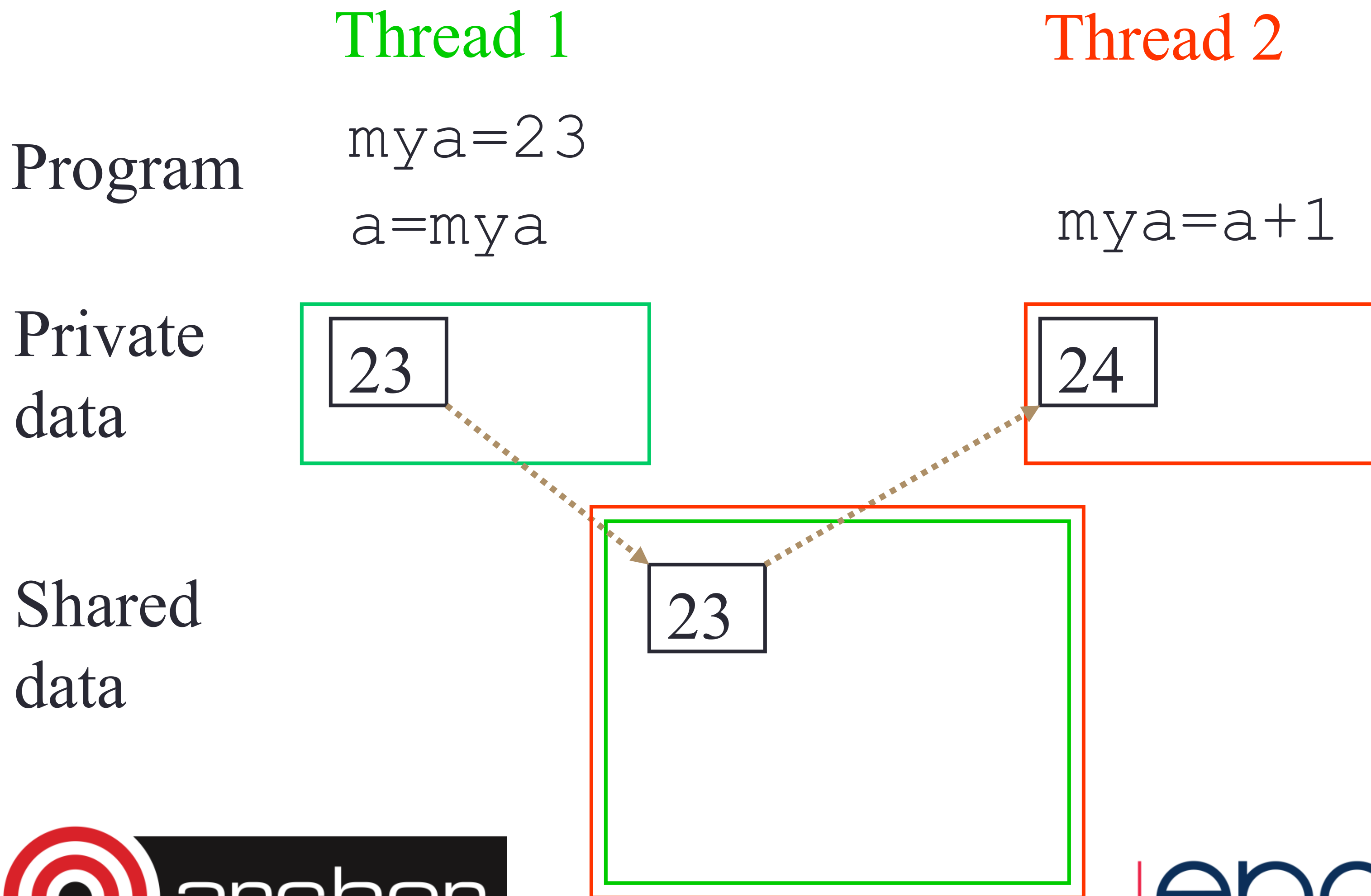
- Have already covered basic concepts
  - threads can all see data of parent process
  - can run on different cores
  - potential for parallel speedup



Analogy

- One very large whiteboard in a two-person office
  - the shared memory
- Two people working on the same problem
  - the threads running on different cores attached to the memory
- How do they collaborate?
  - working together
  - but not interfering
- Also need *private* data

Thread Communication



Synchronisation

- By default, threads execute asynchronously
- Each thread proceeds through program instructions independently of other threads
- This means we need to ensure that actions on shared variables occur in the correct order: e.g.

  thread 1 must write variable A before thread 2 reads it,

  or

  thread 1 must read variable A before thread 2 writes it.

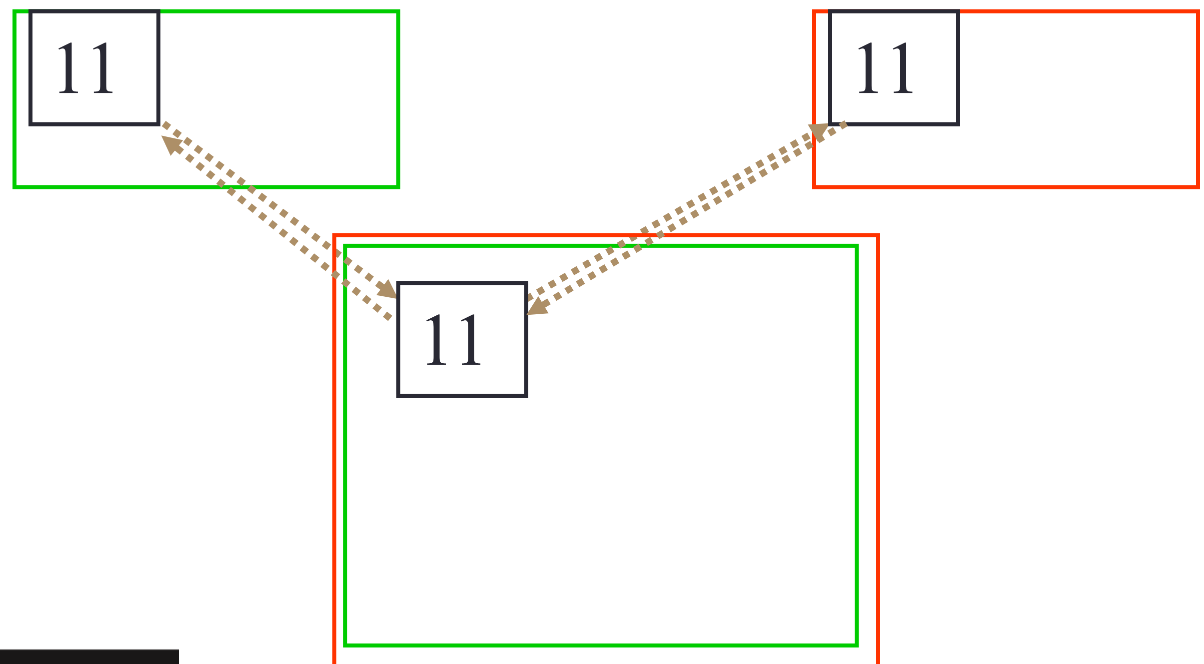
- Note that updates to shared variables (e.g. `a = a + 1`) are *not* atomic!
- If two threads try to do this at the same time, one of the updates may get overwritten.

Synchronisation example

	Thread 1	Thread 2
Program	load a add a 1 store a	load a add a 1 store a

CPU Registers

Memory



Synchronisation

- Synchronisation crucial for shared variables approach
  - thread 2's code must execute *after* thread 1
- Most commonly use global barrier synchronisation
  - other mechanisms such as locks also available
- Writing parallel codes relatively straightforward
  - access shared data as and when its needed
- Getting correct code can be difficult!

Parallel loops

- Loops are the main source of parallelism in many applications.
- If the iterations of a loop are *independent* (can be done in any order) then we can share out the iterations between different threads.
- e.g. if we have two threads and the loop

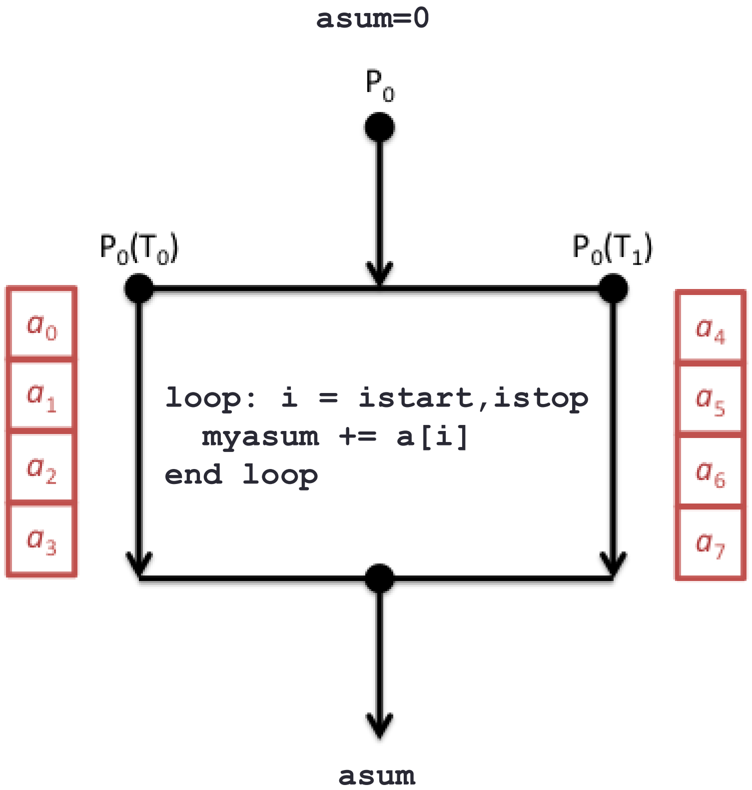
```
for (i=0; i<100; i++){
    a[i] += b[i];
}
```

we could do iteration 0-49 on one thread and iterations 50-99 on the other.

- Can think of an iteration, or a set of iterations, as a task.

Specific example

- Computing **asum** $= a_0 + a_1 + \ldots a_7$
  - shared:
    - main array: **a[8]**
    - result: **asum**
  - private:
    - loop counter: **i**
    - loop limits: **istart, istop**
    - local sum: **myasum**
  - synchronisation:
    - thread0: **asum += myasum**
    - barrier
    - thread1: **asum += myasum**



Reductions

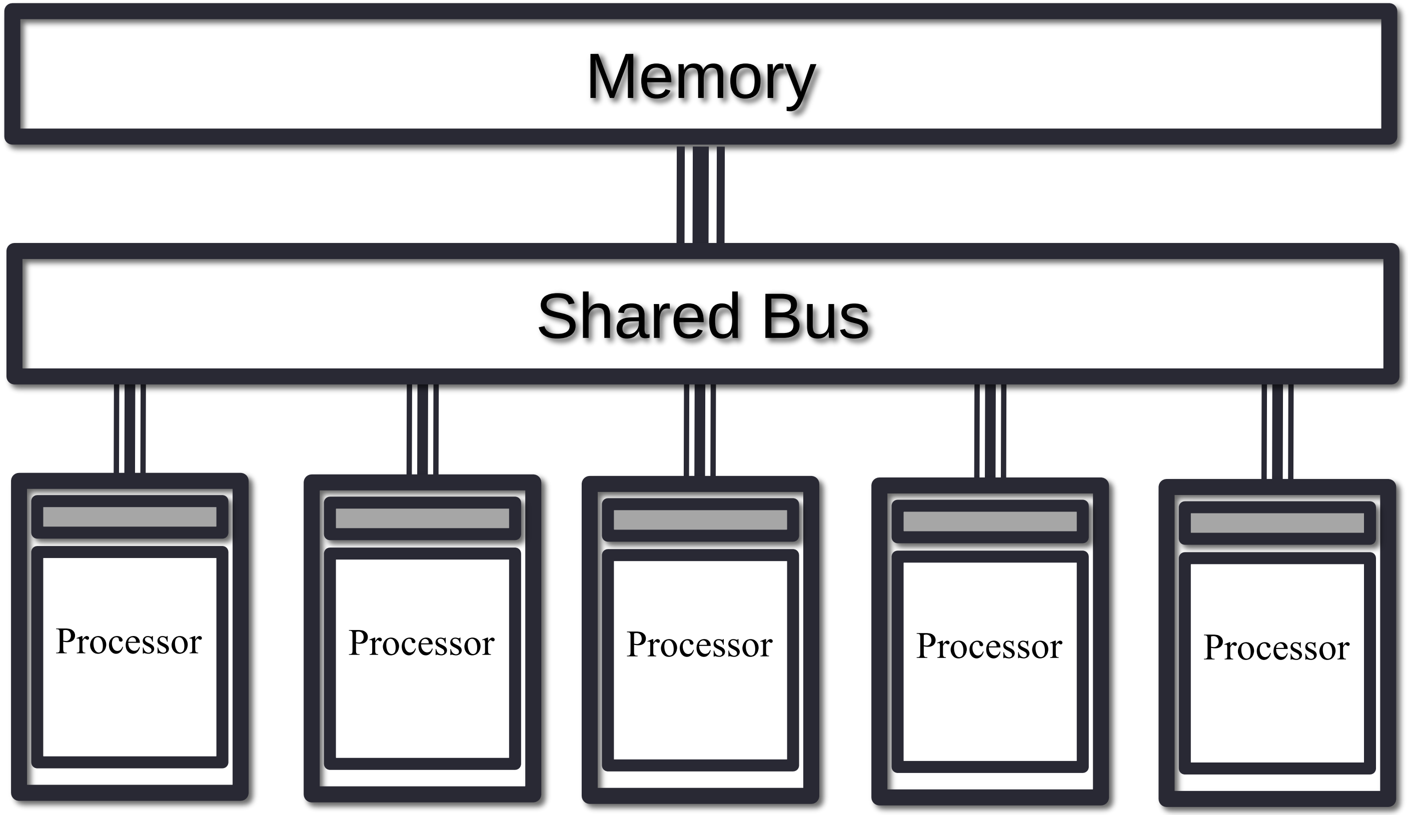
- A *reduction* produces a single value from associative operations such as addition, multiplication, max, min, and, or.

```
asum = 0;
for (i=0; i<n; i++)
    asum += a[i];
```

- Only one thread at a time updating `asum` removes all parallelism
  - each thread accumulates own private copy; copies reduced to give final result.
  - if the number of operations is much larger than the number of threads, most of the operations can proceed in parallel
- Want common patterns like this to be automated
  - **not** programmed by hand as in previous slide

Hardware

- Needs support of a shared-memory architecture



Threads: Summary

- Shared blackboard a good analogy for thread parallelism
- Requires a shared-memory architecture
  - in HPC terms, cannot scale beyond a single node

- Threads operate independently on the shared data
  - need to ensure they don't interfere; synchronisation is crucial

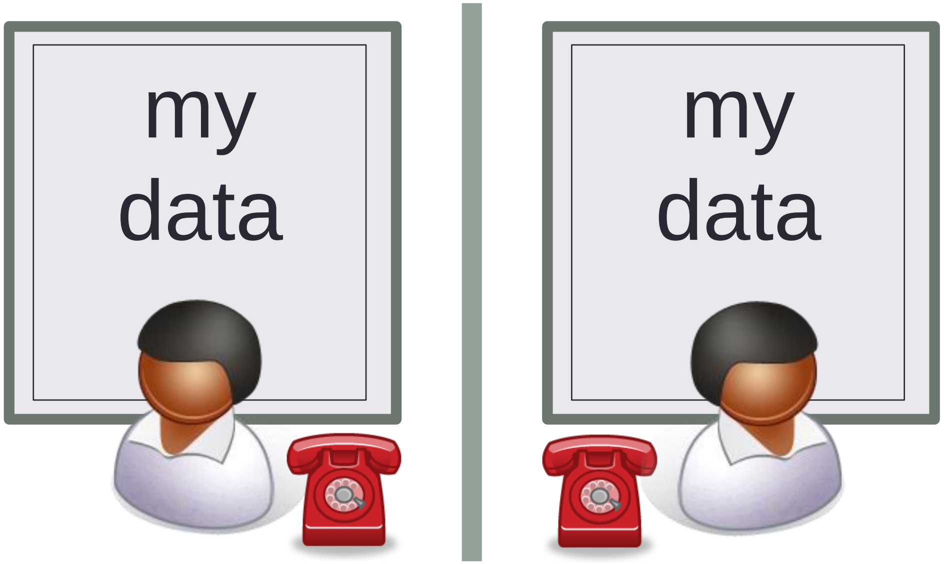
- Threading in HPC usually uses OpenMP directives
  - supports common parallel patterns such as reductions
  - e.g. loop limits computed by the compiler
  - e.g. summing values across threads done automatically

Message Passing

Process-based parallelism

Analogy

- Two whiteboards in different single-person offices
  - the distributed memory
- Two people working on the same problem
  - the processes on different nodes attached to the interconnect

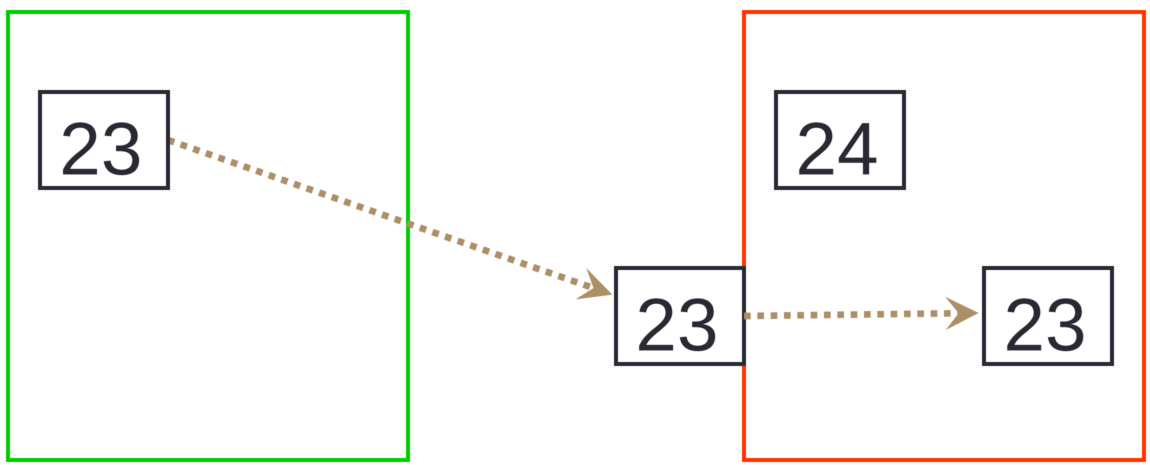


- How do they collaborate?
  - to work on single problem
- Explicit communication
  - e.g. by telephone
  - no shared data

Process communication

	Process 1	Process 2
Program	`a=23` `Send(2,a)`	`Recv(1,b)` `a=b+1`

Data



Synchronisation

- Synchronisation is automatic in message-passing
  - the messages do it for you

- Make a phone call …
  - … wait until the receiver picks up
- Receive a phone call
  - … wait until the phone rings

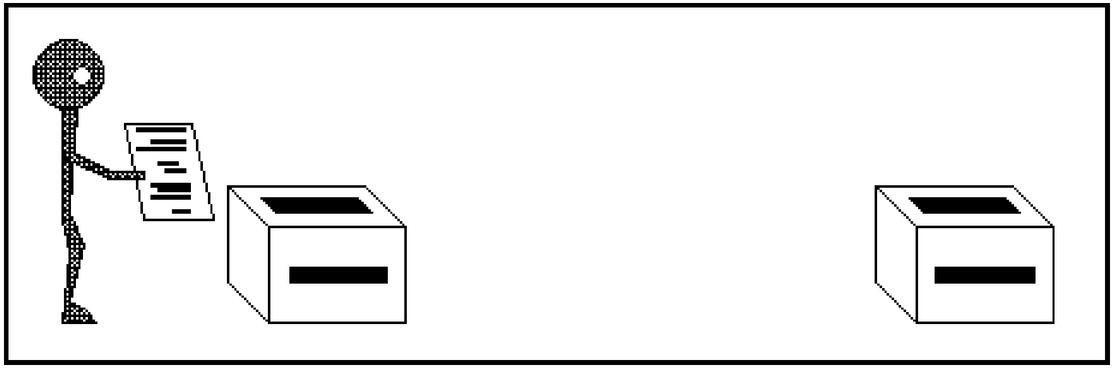
- No danger of corrupting someone else's data
  - no shared blackboard

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

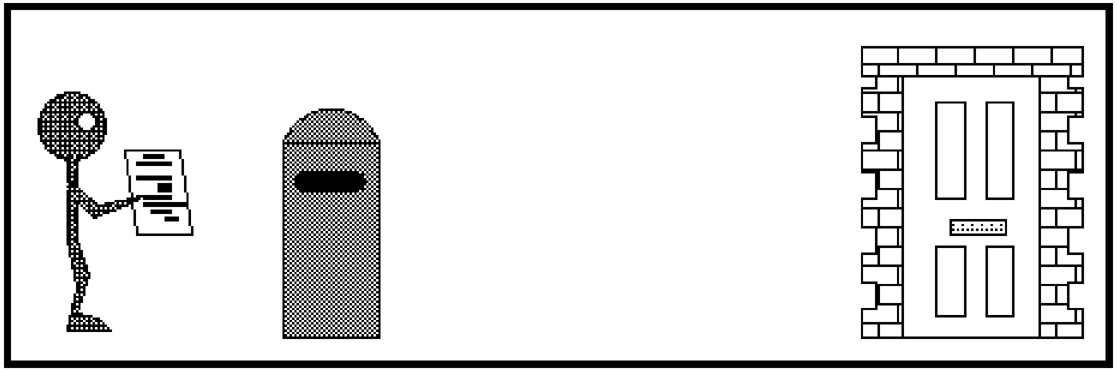
Synchronous send

- 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.



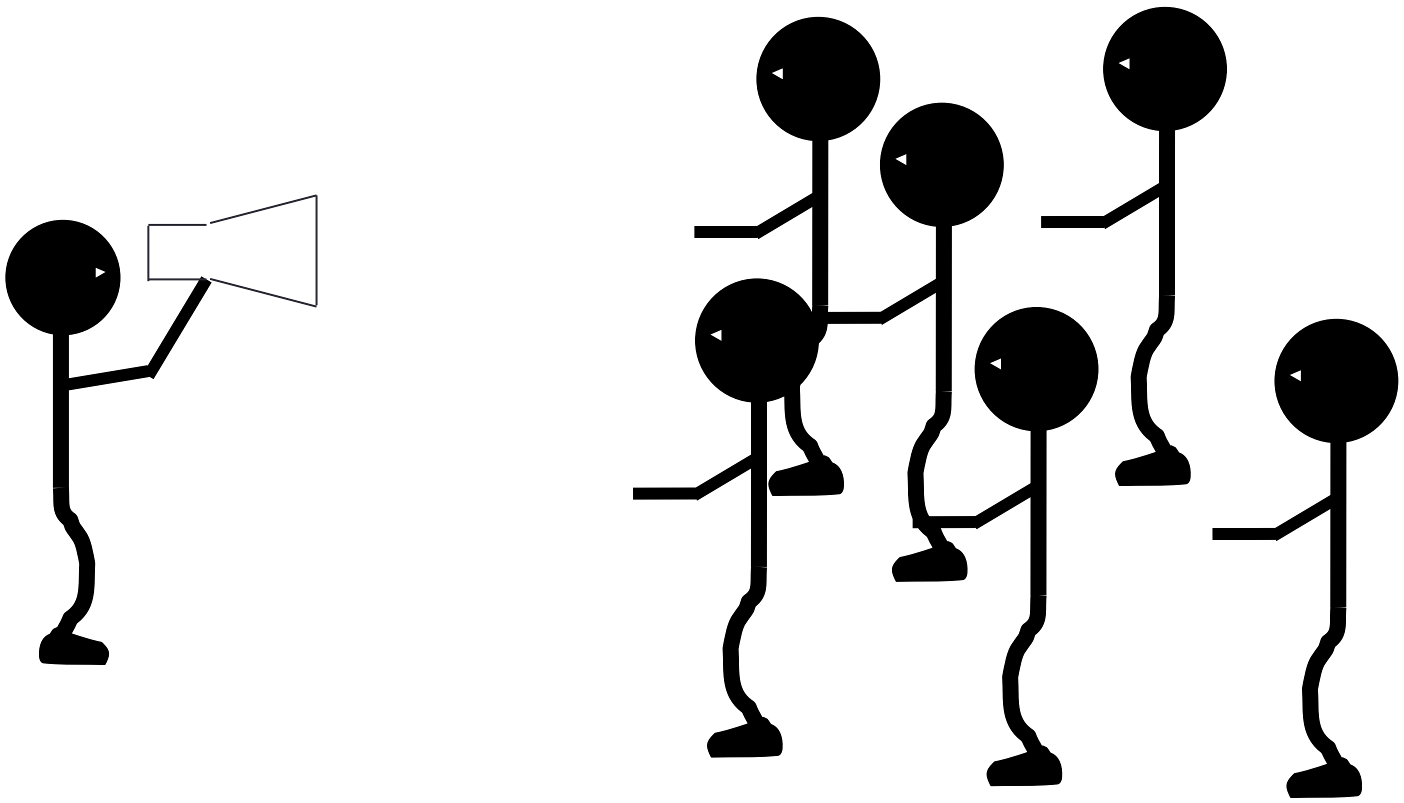
Point-to-Point Communications

- We have considered two processes
  - one sender
  - one receiver
- This is called point-to-point communication
  - simplest form of message passing
  - relies on matching send and receive
- Close analogy to sending personal emails

Collective Communications

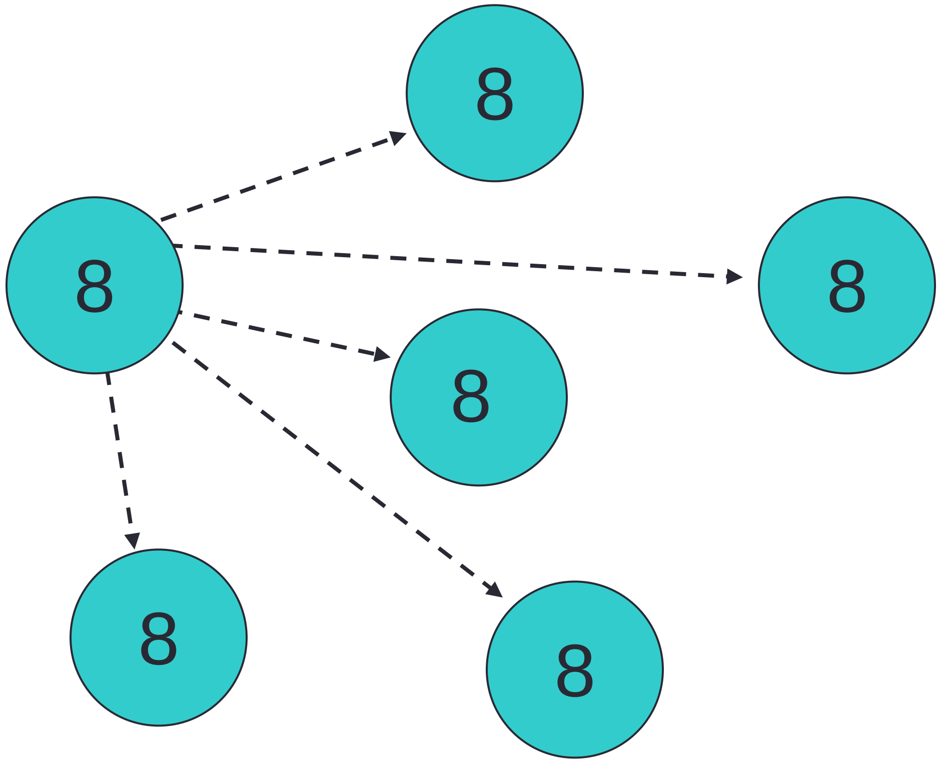
- A simple message communicates between two processes
- There are many instances where communication between groups of processes is required
- Can be built from simple messages, but often implemented separately, for efficiency

Broadcast: one to all communication



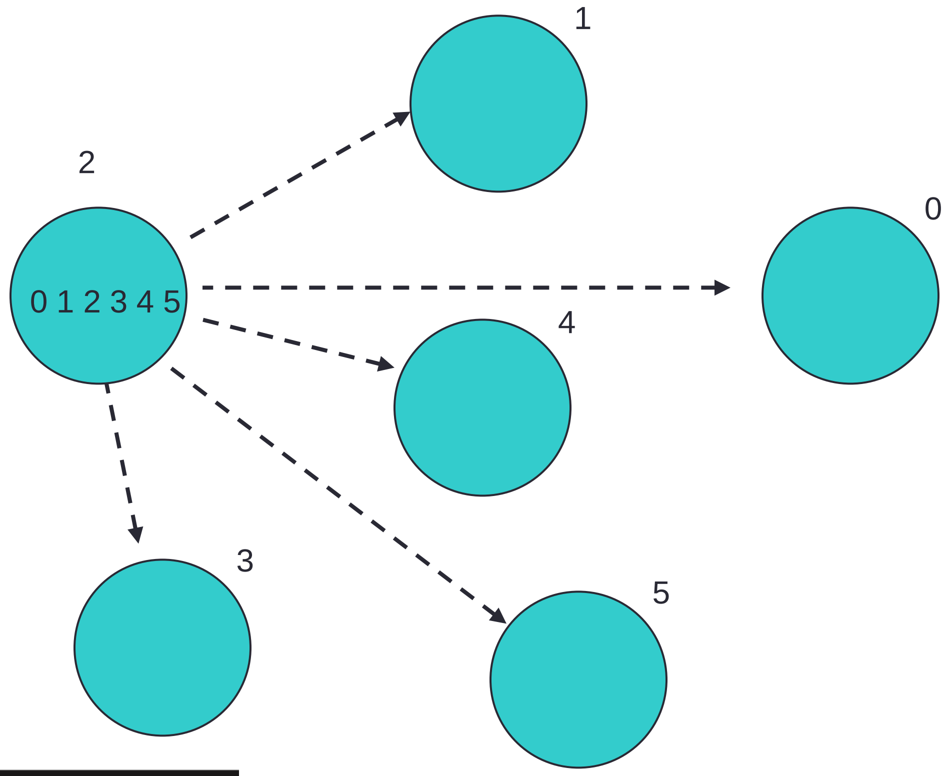
Broadcast

- From one process to all others



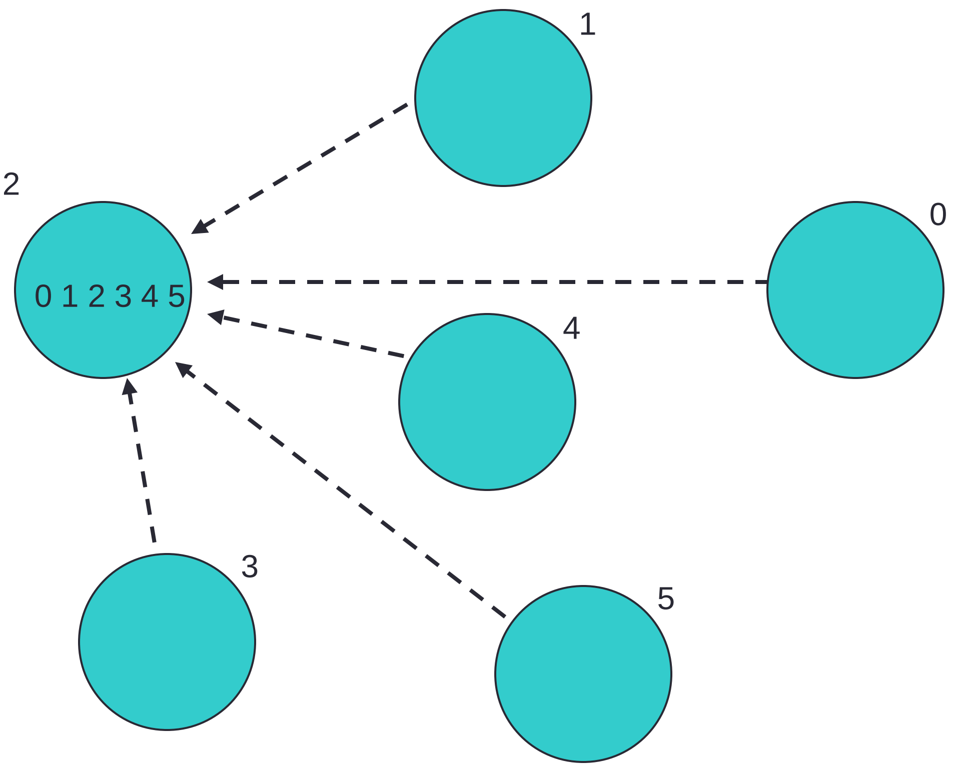
Scatter

- Information scattered to many processes



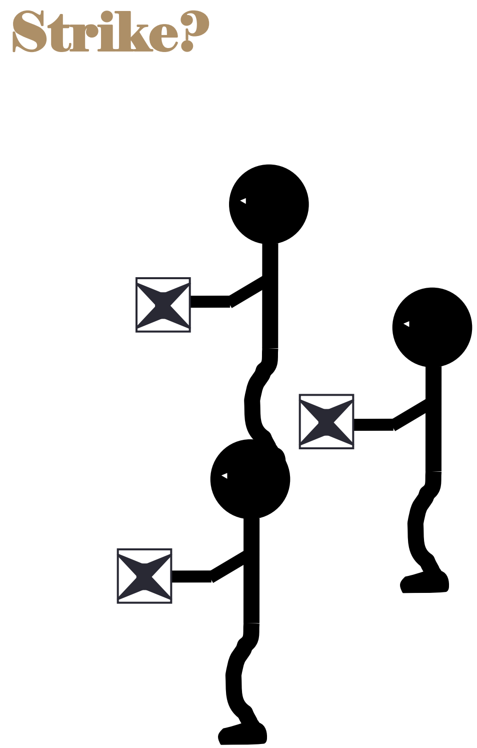
Gather

- Information gathered onto one process



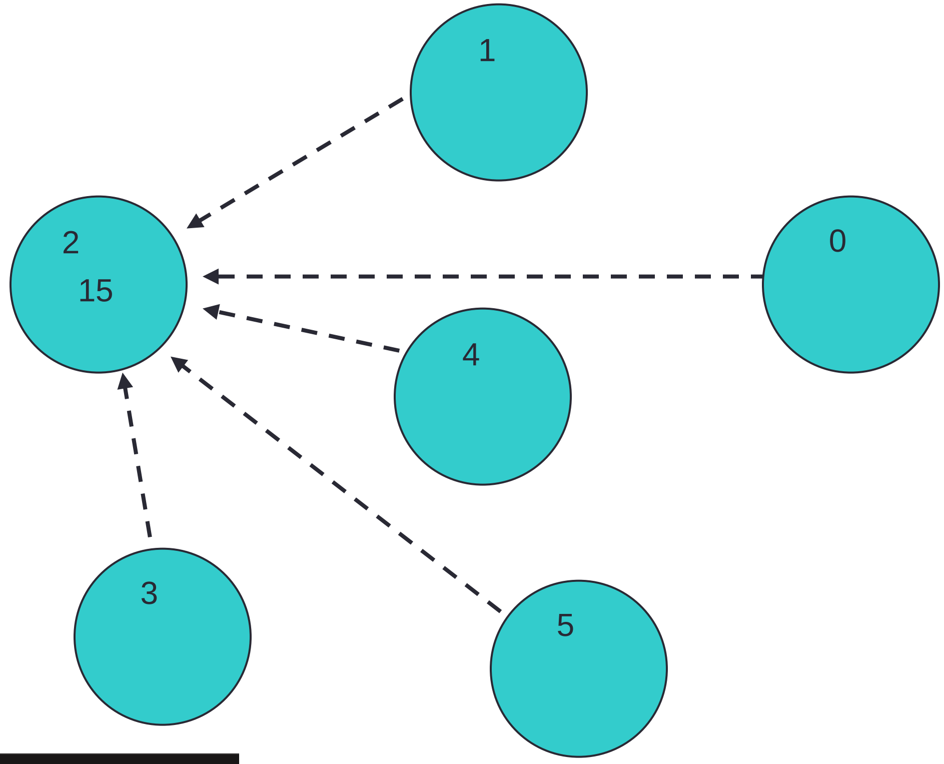
Reduction Operations

- Combine data from several processes to form a single result

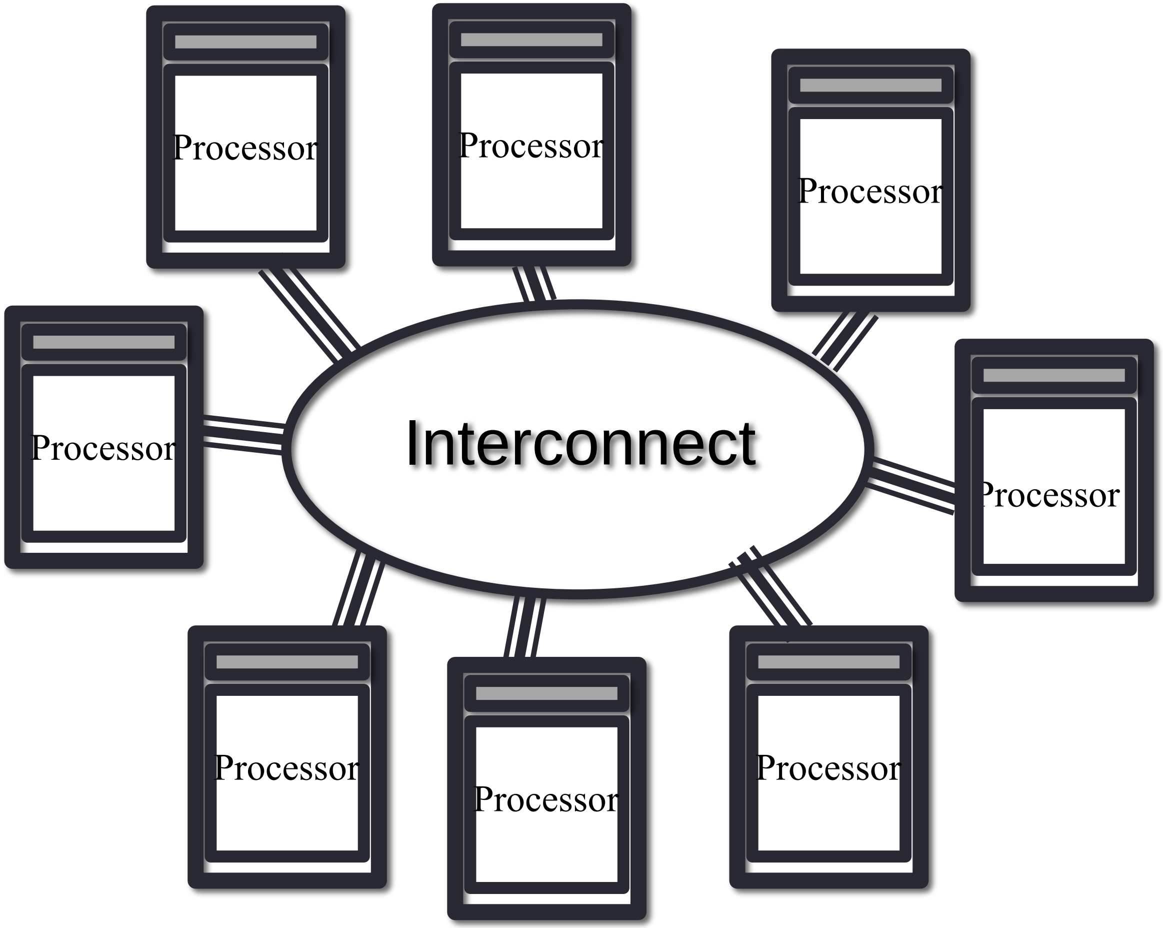


Reduction

- Form a global sum, product, max, min, etc.



Hardware



- Natural map to distributed-memory
  - one process per processor-core
  - messages go over the interconnect, between nodes/OS's

Programming Models

Serial Programming

Concepts

Arrays
Control flow
Subroutines
Human-readable
Variables
OO

Languages

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

Implementations

gcc -O3
icc
pgcc -fast
crayftn
craycc
javac

Message-Passing Parallel Programming

Concepts

Processes
Groups
Send/Receive
SPMD
Collectives

Libraries

MPI
MPI_Init()

Implementations

MPICH2
Intel MPI
Cray MPI
OpenMPI
IBM MPI

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

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

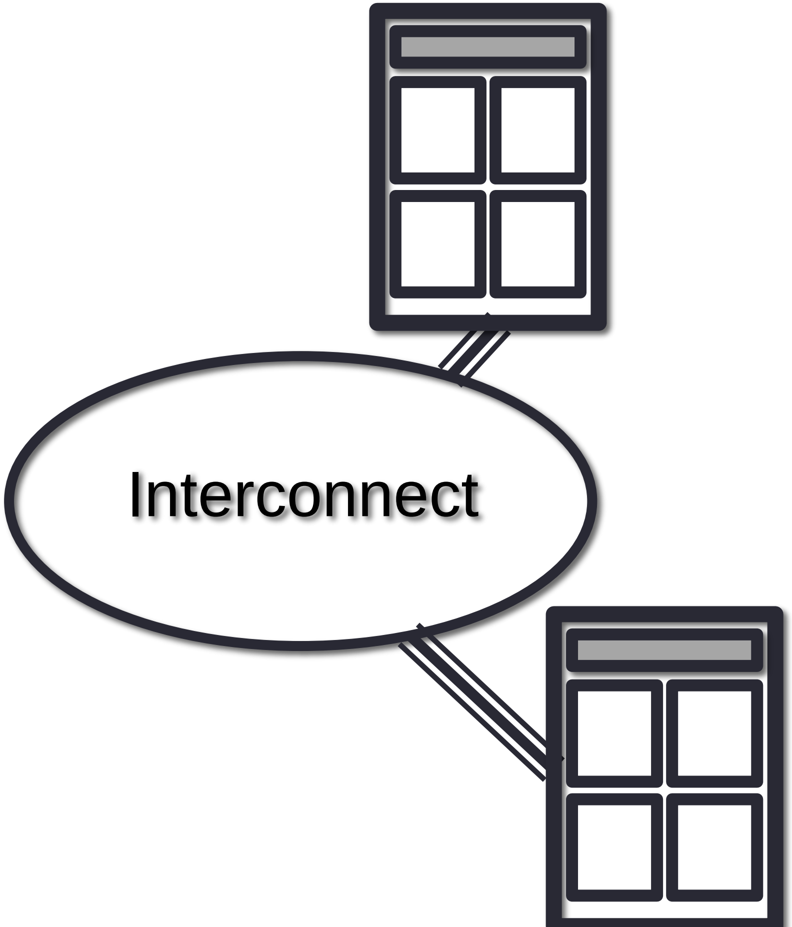
Summary (i)

- 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

Processes: Summary

- Processes cannot share memory
  - ring-fenced from each other
  - analogous to white boards in separate offices
- Communication requires explicit *messages*
  - analogous to making a phone call, sending an email, …
  - synchronisation is done by the messages
- Almost exclusively use Message-Passing Interface
  - MPI is a library of function calls / subroutines

Practicalities

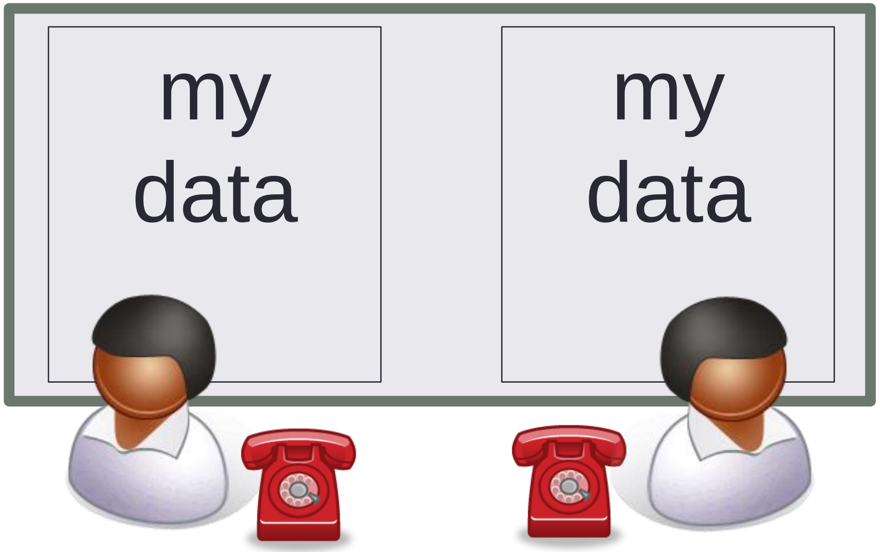


- 8-core machine might only have 2 nodes
  - how do we run MPI on a real HPC machine?
- Mostly ignore architecture
  - pretend we have single-core nodes
  - one MPI process per processor-core
  - e.g. run 8 processes on the 2 nodes
- Messages between processes on the same node are fast
  - but remember they also share access to the network

Message Passing on Shared Memory

- Run one process per core
  - don't directly exploit shared memory
  - analogy is phoning your office mate
  - actually works well in practice!

- Message-passing programs run by a special job launcher
  - user specifies #copies
  - some control over allocation to nodes



Summary

- Shared-variables parallelism
  - uses threads
  - requires shared-memory machine
  - easy to implement but limited scalability
  - in HPC, done using OpenMP compilers

- Distributed memory
  - uses processes
  - can run on any machine: messages can go over the interconnect
  - harder to implement but better scalability
  - on HPC, done using the MPI library