

Parallel Models

Different ways to exploit parallelism



EPSRC



NERC SCIENCE OF THE ENVIRONMENT



archer



CRAY
THE SUPERCOMPUTER COMPANY



epcc



Outline

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



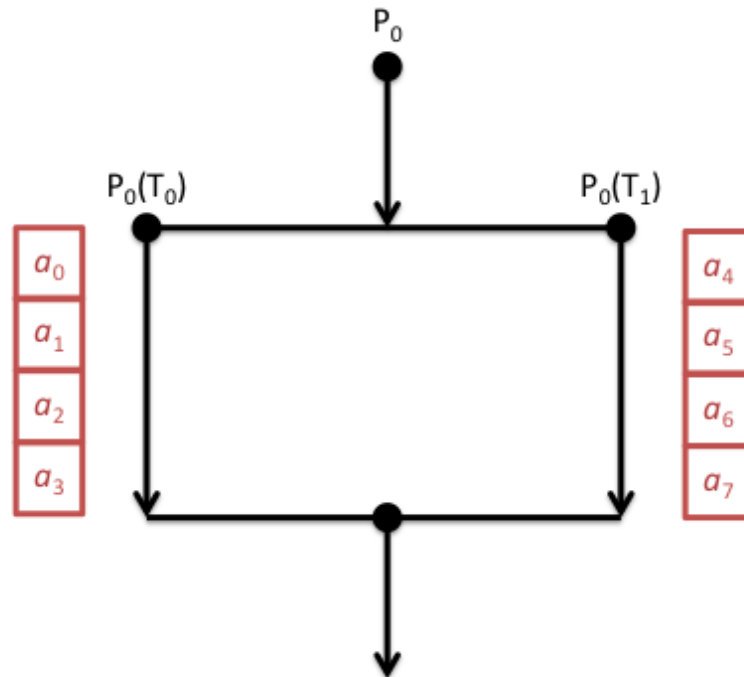
Shared Variables

Threads-based parallelism



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

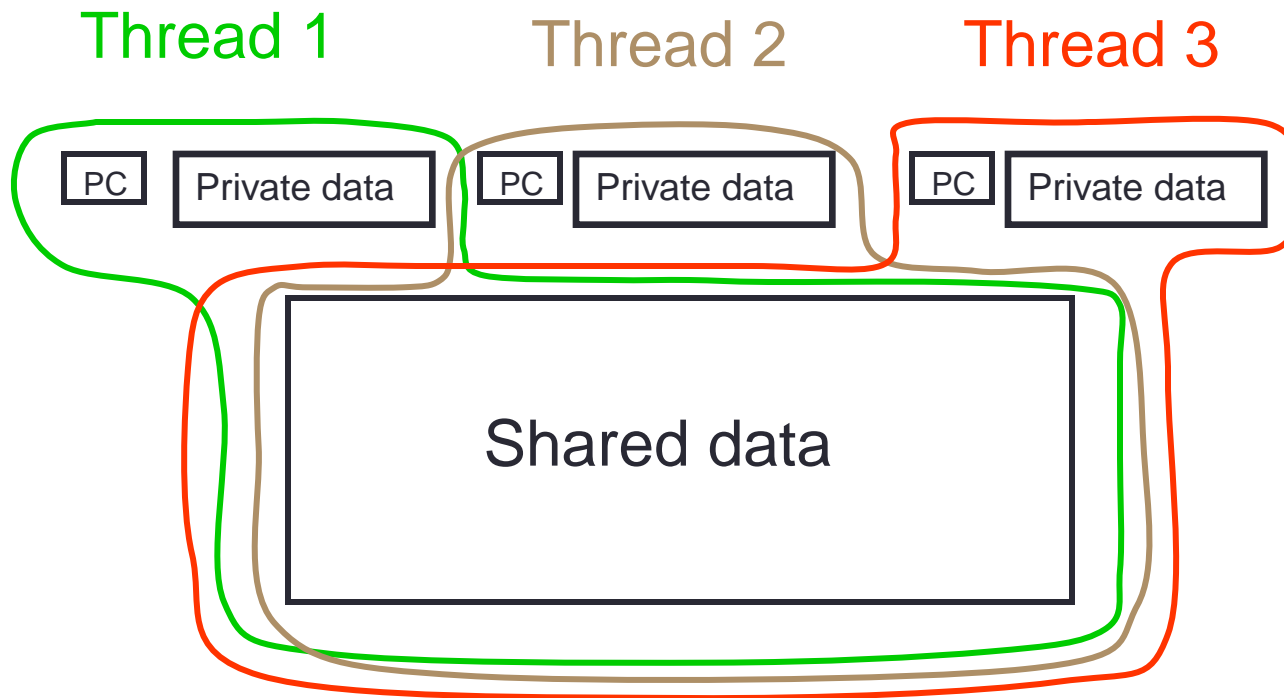


Analogy

- 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



Threads



Thread Communication

Thread 1

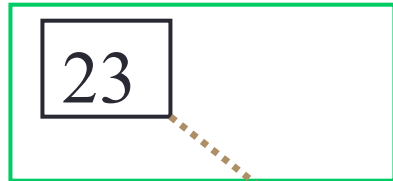
Thread 2

Program

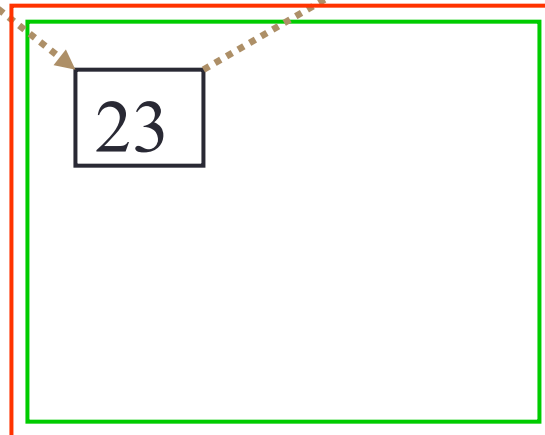
```
mya=23  
a=mya
```

```
mya=a+1
```

Private
data



Shared
data



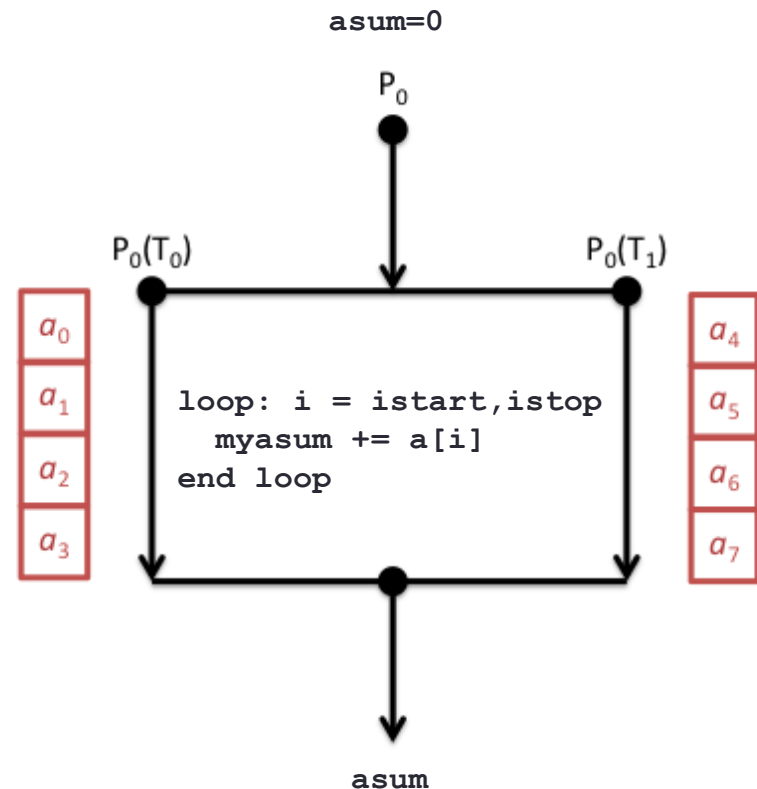
Synchronisation

- 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!



Specific example

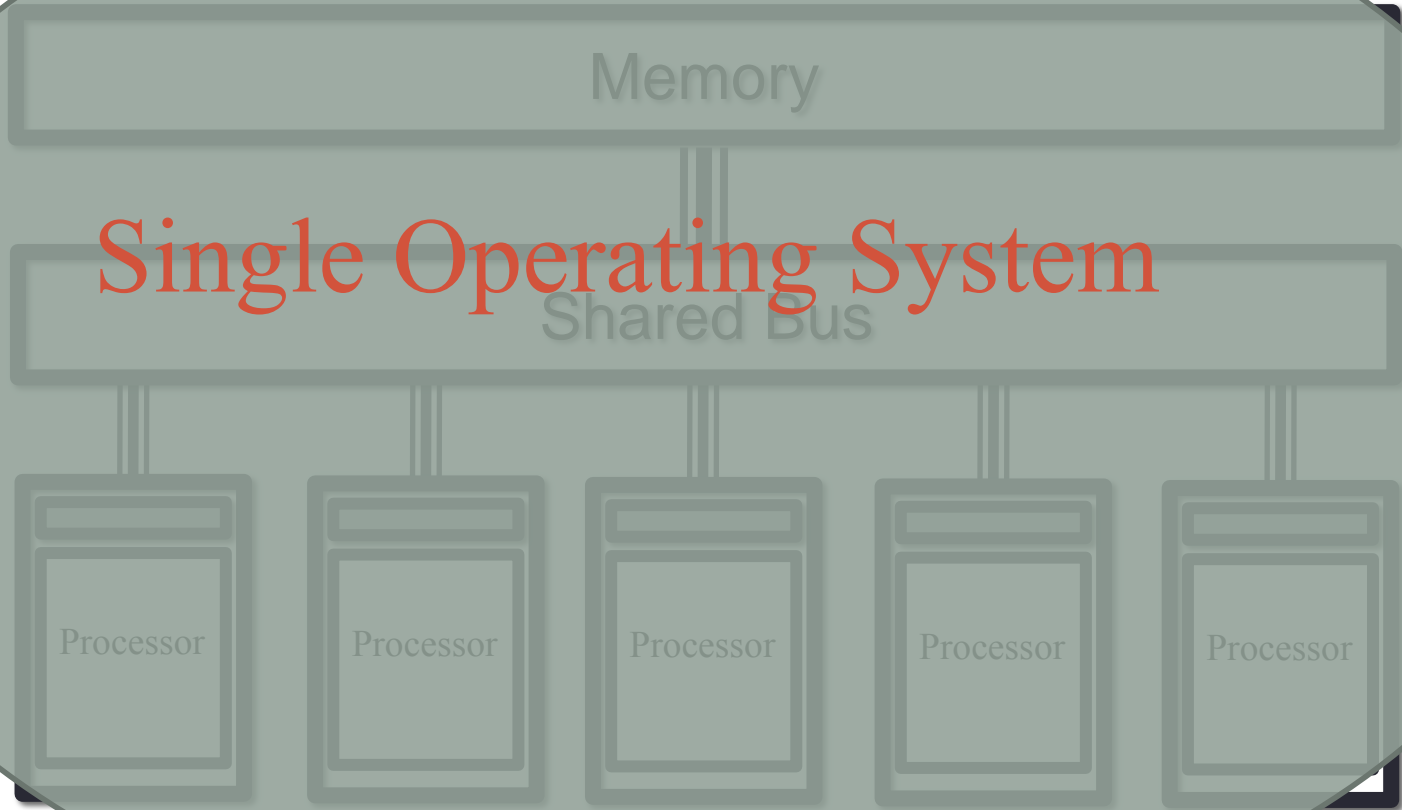
- Computing $asum = a_0 + a_1 + \dots + a_7$
 - shared:
 - main array: **a [8]**
 - result: **asum**
 - private:
 - loop counter: **i**
 - loop limits: **istart, istop**
 - local sum: **myasum**
 - synchronisation:
 - thread0: **asum += myasum**
 - barrier
 - thread1: **asum += myasum**



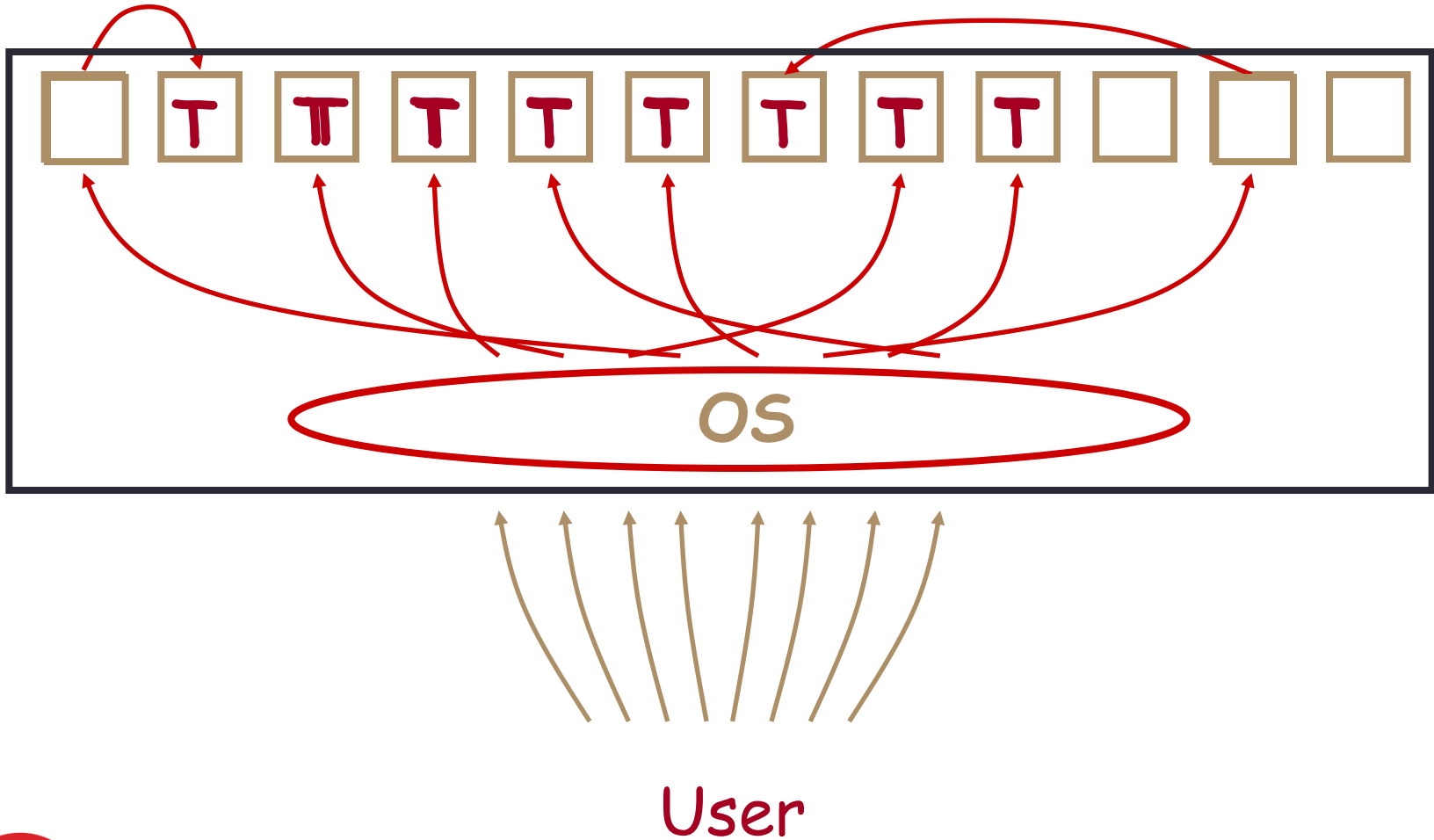
Hardware

- Needs support of a shared-memory architecture

Single Operating System



Thread Placement: Shared Memory



Threads in HPC

- Threads existed before parallel computers
 - Designed for *concurrency*
 - Many more threads running than physical cores
 - scheduled / descheduled as and when needed
- For parallel computing
 - Typically run a single thread per core
 - Want them all to run all the time
- OS optimisations
 - Place threads on selected cores
 - Stop them from migrating



Practicalities

- Threading can only operate within a single node
 - Each node is a shared-memory computer (e.g. 24 cores on ARCHER)
 - Controlled by a single operating system
- Simple parallelisation
 - Speed up a serial program using threads
 - Run an independent program per node (e.g. a simple task farm)
- More complicated
 - Use multiple processes (e.g. message-passing – next)
 - On ARCHER: could run one process per node, 24 threads per process
 - or 2 procs per node / 12 threads per process or 4 / 6 ...



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
 - e.g. loop limits computed by the compiler
 - e.g. summing values across threads done automatically



Message Passing

Process-based parallelism



Analogy

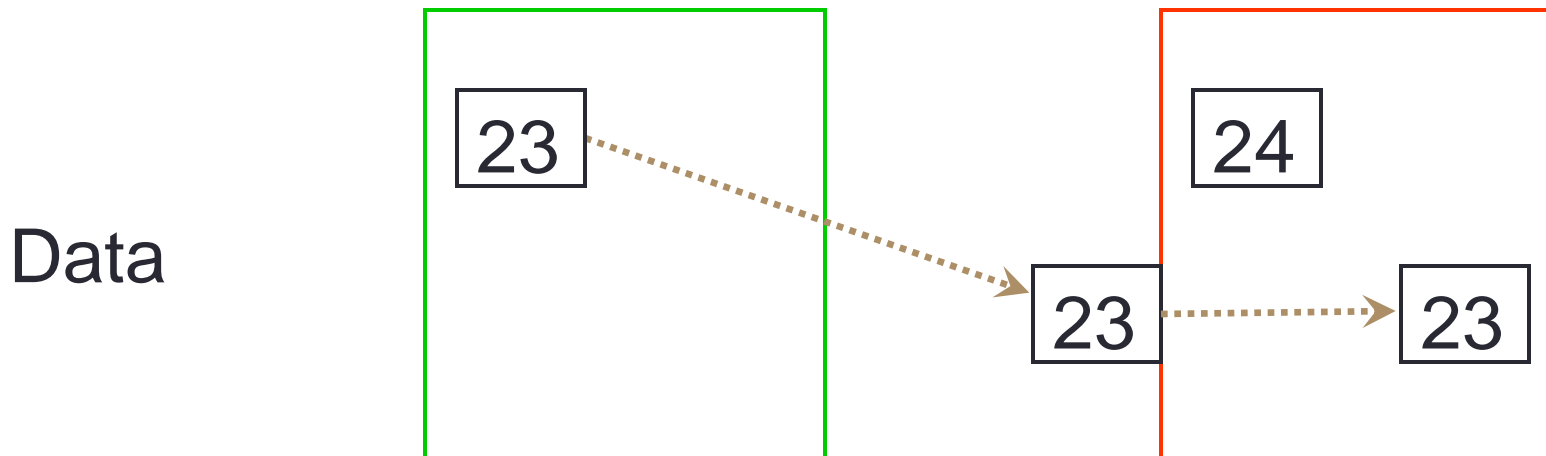
- Two whiteboards in different single-person offices
 - the distributed memory
- Two people working on the same problem
 - the processes on different nodes attached to the interconnect
- How do they collaborate?
 - to work on single problem
- Explicit communication
 - e.g. by telephone
 - no shared data



Process communication

Program

Process 1	Process 2
<code>a=23</code>	<code>Recv (1, b)</code>
<code>Send (2, a)</code>	<code>a=b+1</code>



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



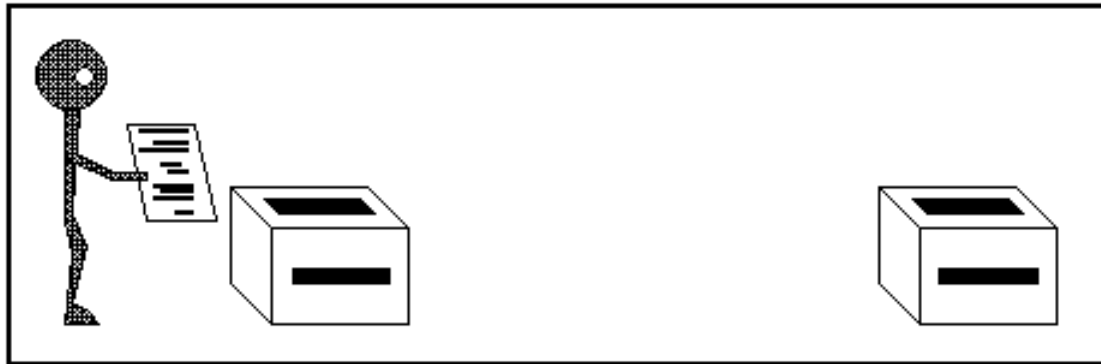
Communication modes

- 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



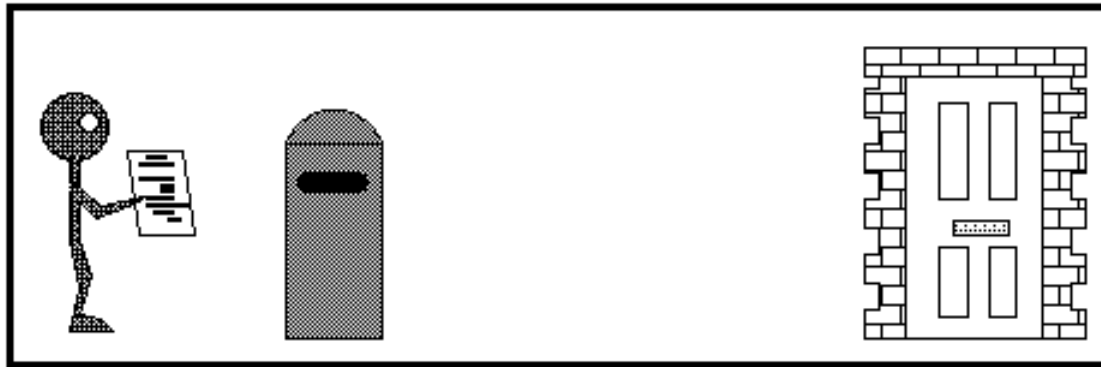
Synchronous send

- 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

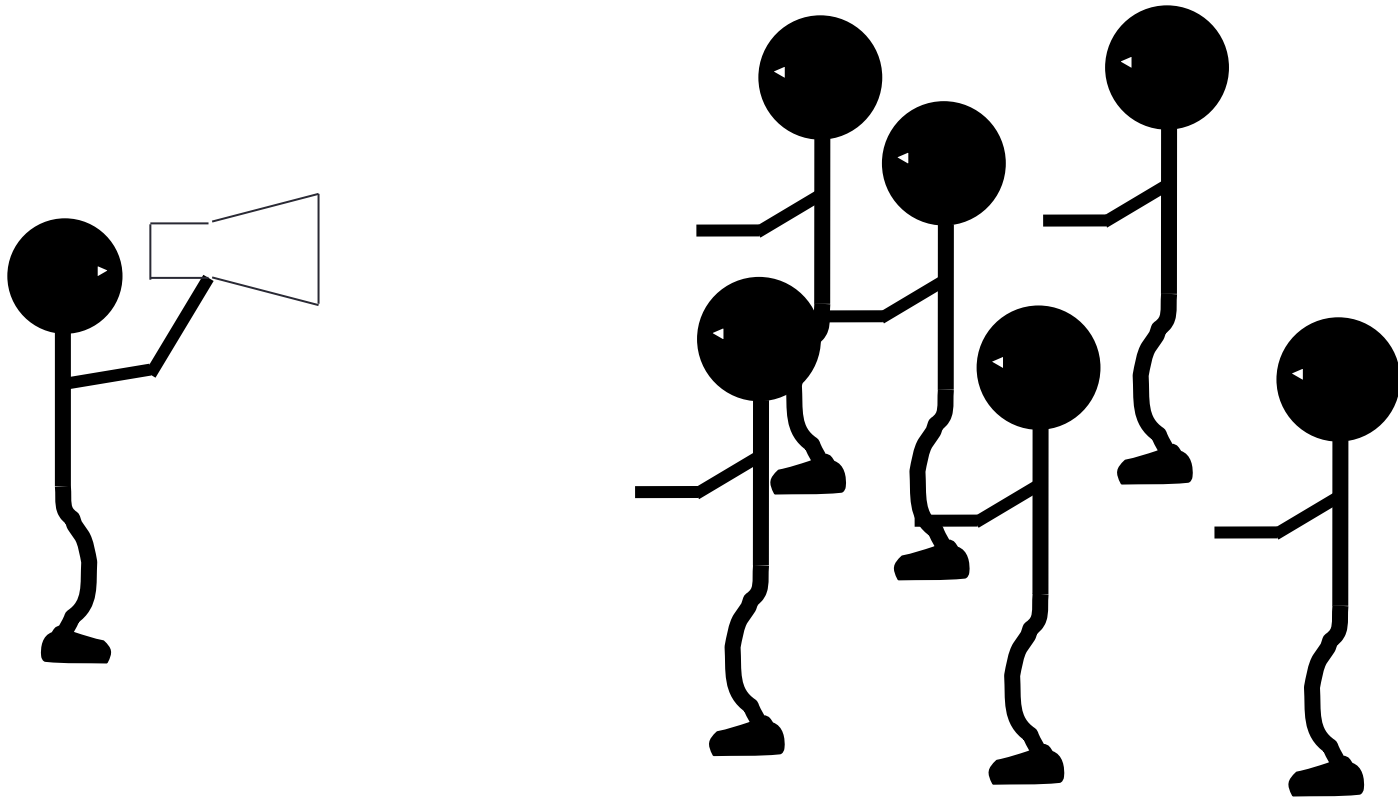


Collective Communications

- 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

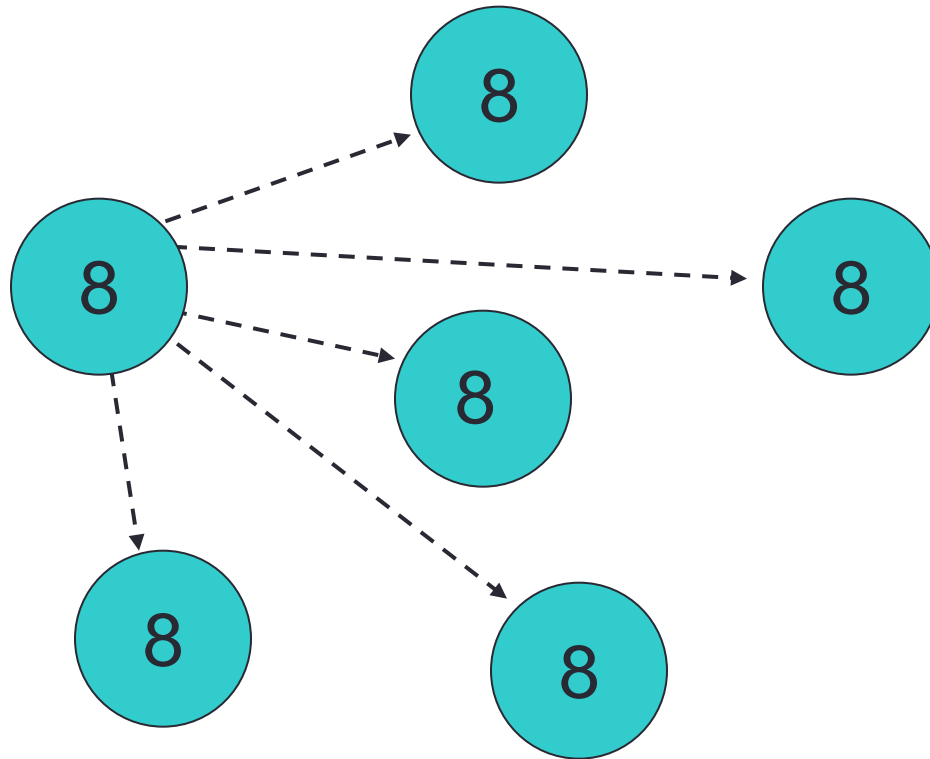


Broadcast: one to all communication



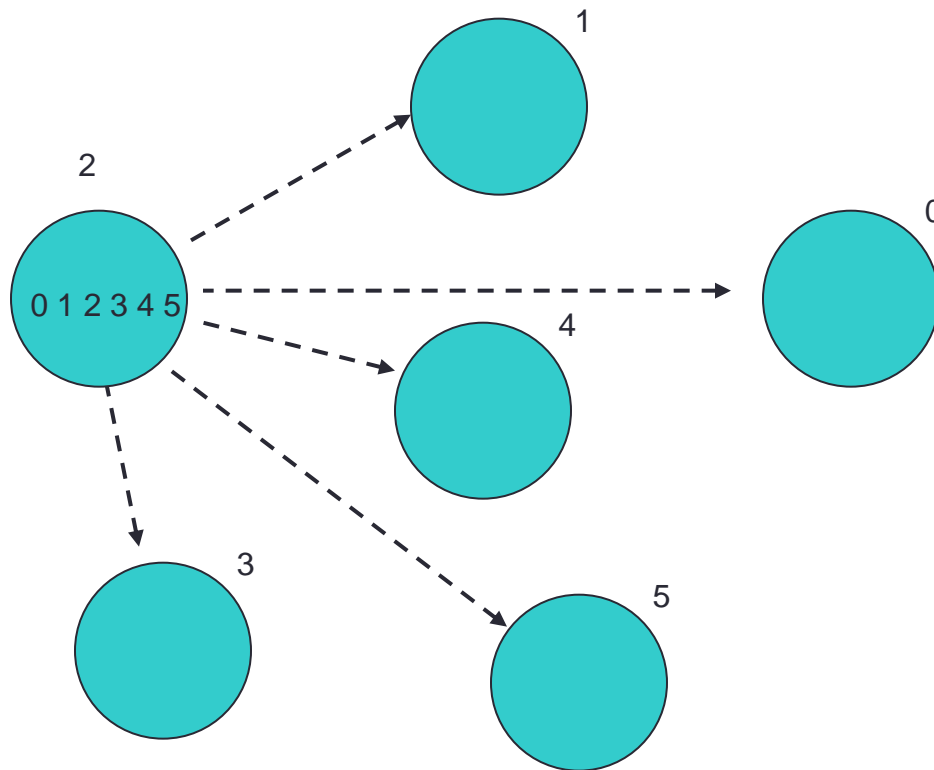
Broadcast

- From one process to all others



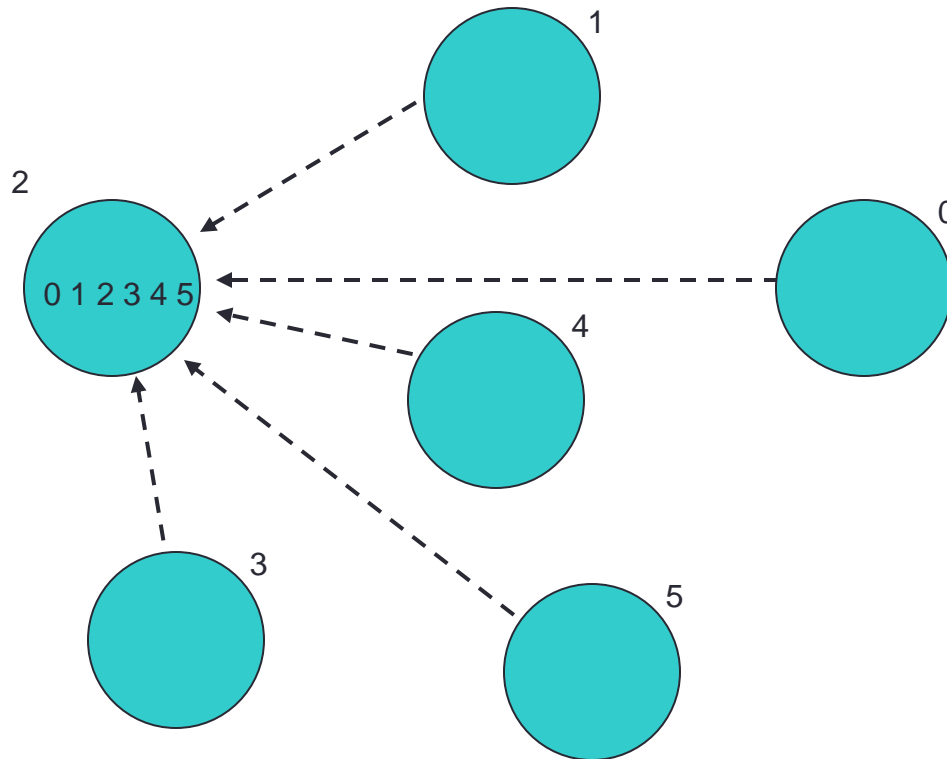
Scatter

- Information scattered to many processes



Gather

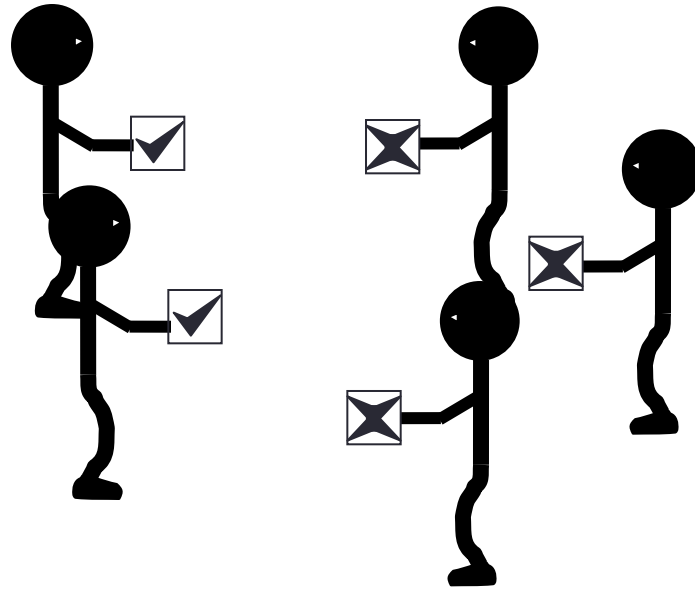
- Information gathered onto one process



Reduction Operations

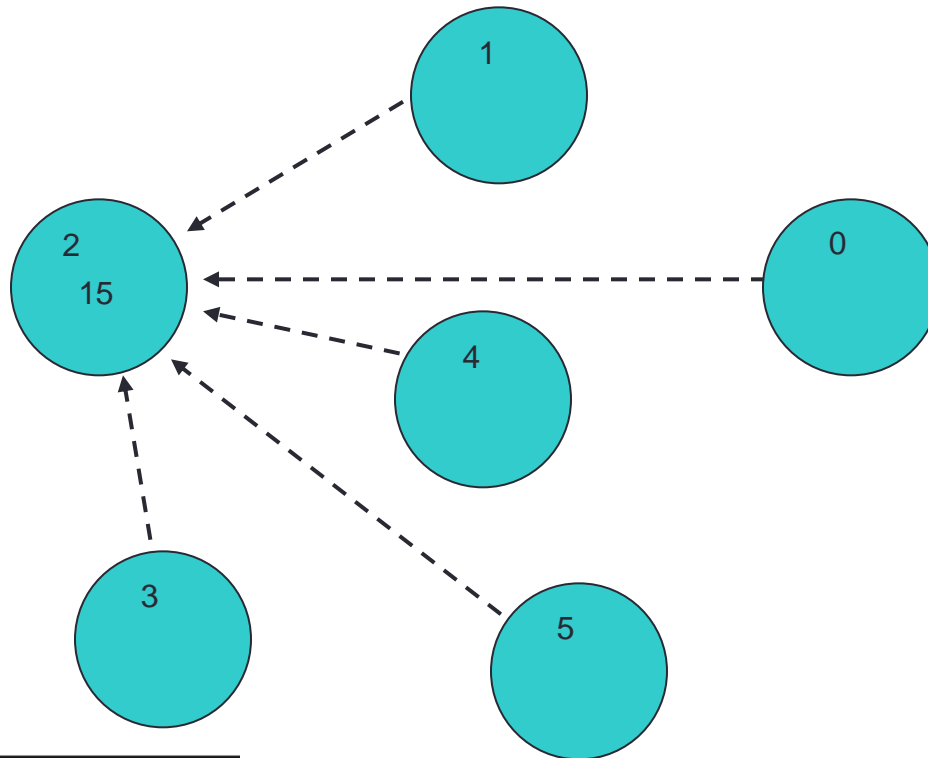
- Combine data from several processes to form a single result

Strike?

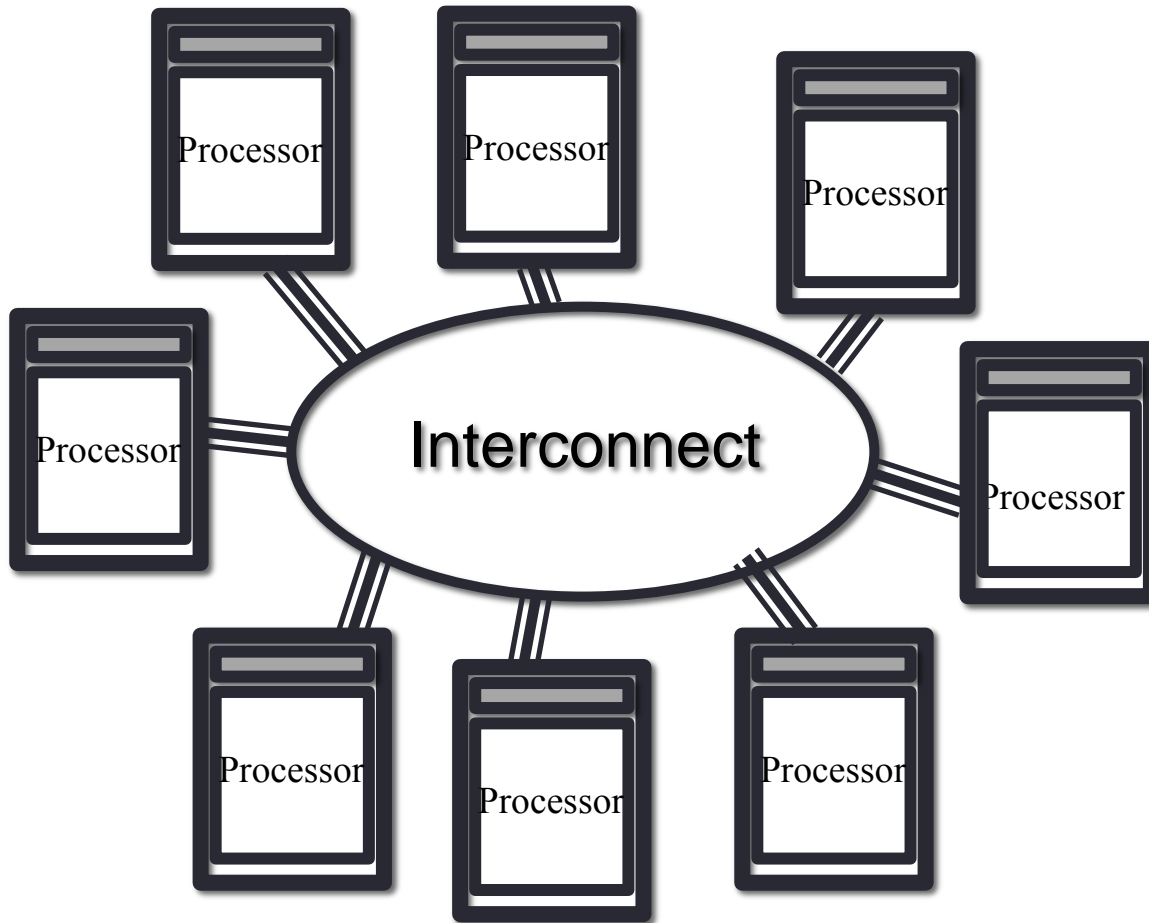


Reduction

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



Hardware



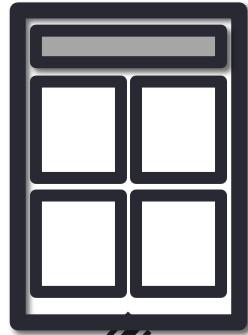
- Natural map to distributed-memory
 - one process per processor-core
 - messages go over the interconnect, between nodes/OS's

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



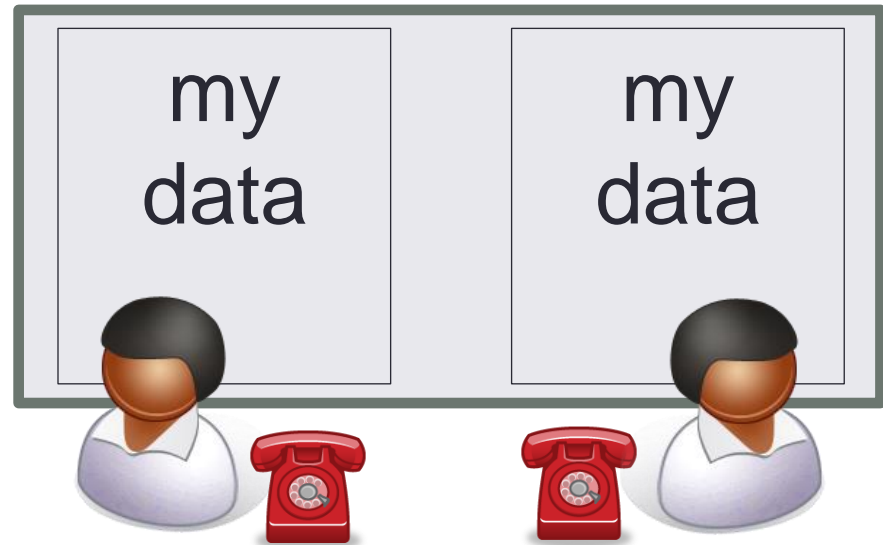
Interconnect



- 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 processor-cores 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

