Shared Variables

Parallel Programming using Threads

Outline

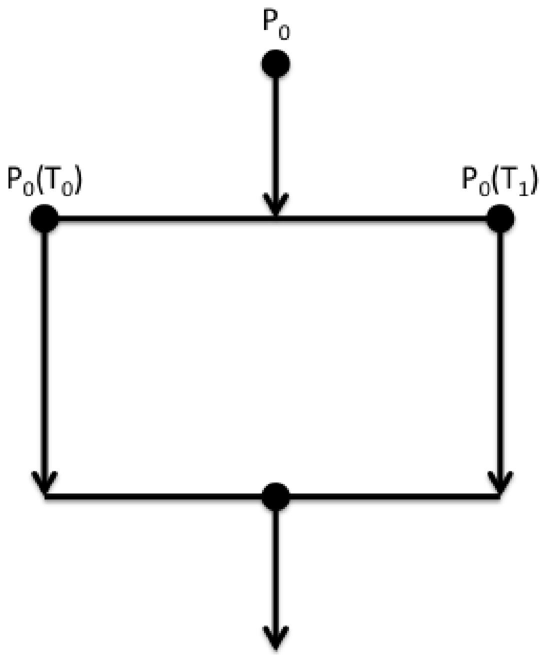
- Shared-Variables Parallelism
  - threads
  - shared-memory architectures
- Practicalities
  - operating systems
  - usage on real HPC architectures

Shared Variables

Threads-based parallelism

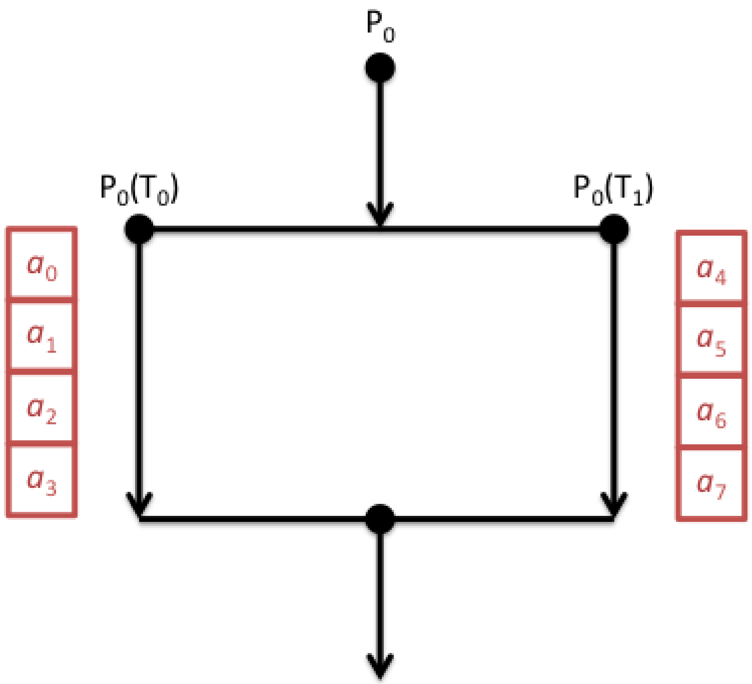
Threads

- For many applications each process has a single *thread*…
  - … but a single process can contain multiple threads
  - each thread is like a child process contained *within* parent process



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

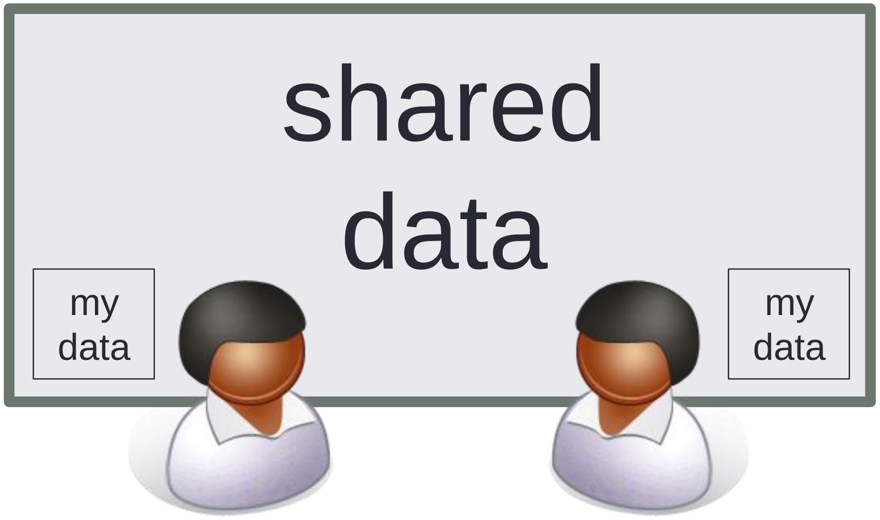


Analogy

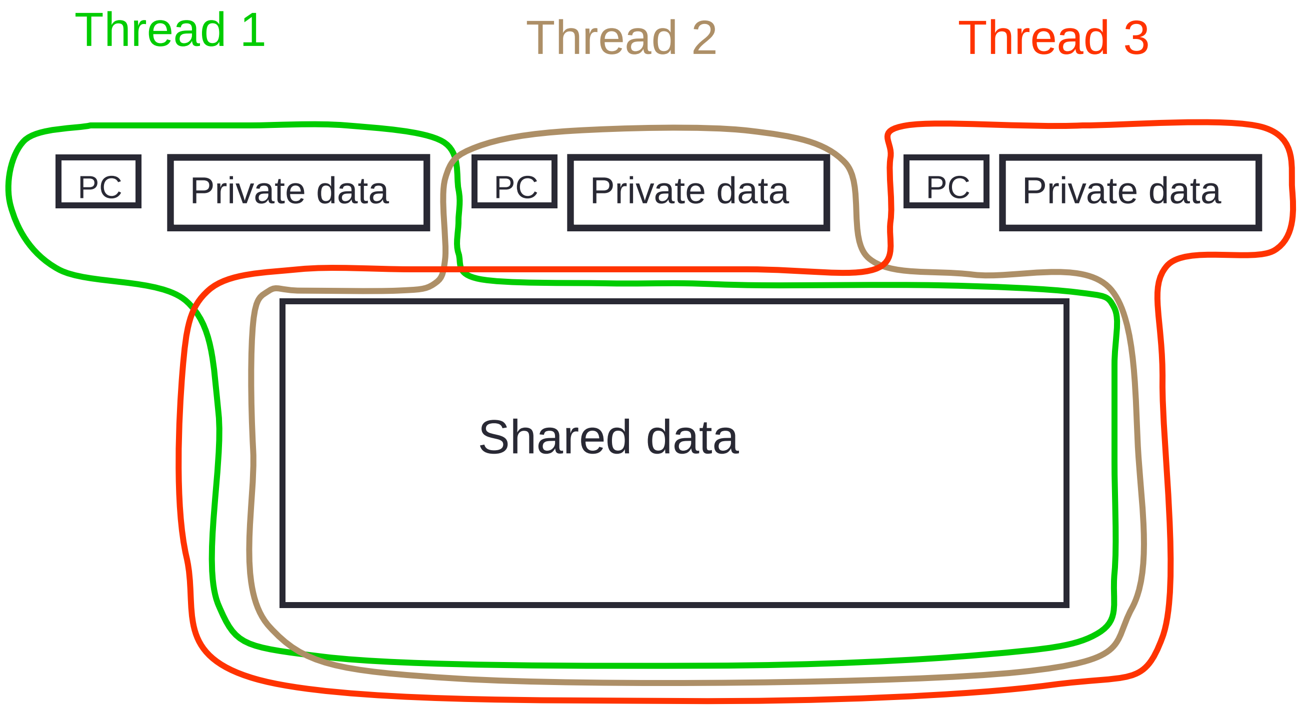
- One very large whiteboard in a two-person office
  - the shared memory
- Two people working on the same problem
  - the threads running on different cores attached to the memory

- How do they collaborate?
  - working together
  - but not interfering

- Also need *private* data



Threads


Thread 1
Thread 2
Thread 3
PC
Private data
PC
Private data
PC
Private data
Shared data

Thread Communication

Thread 1

Thread 2

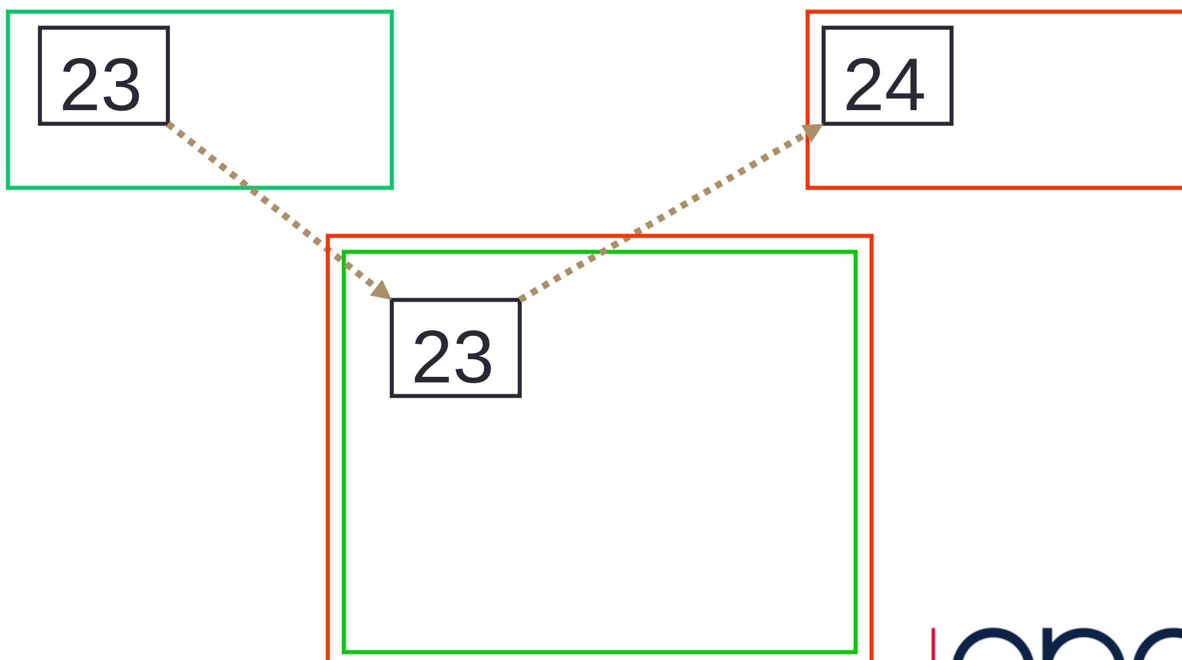
Program

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

```
mya=a+1
```

Private data

Shared data

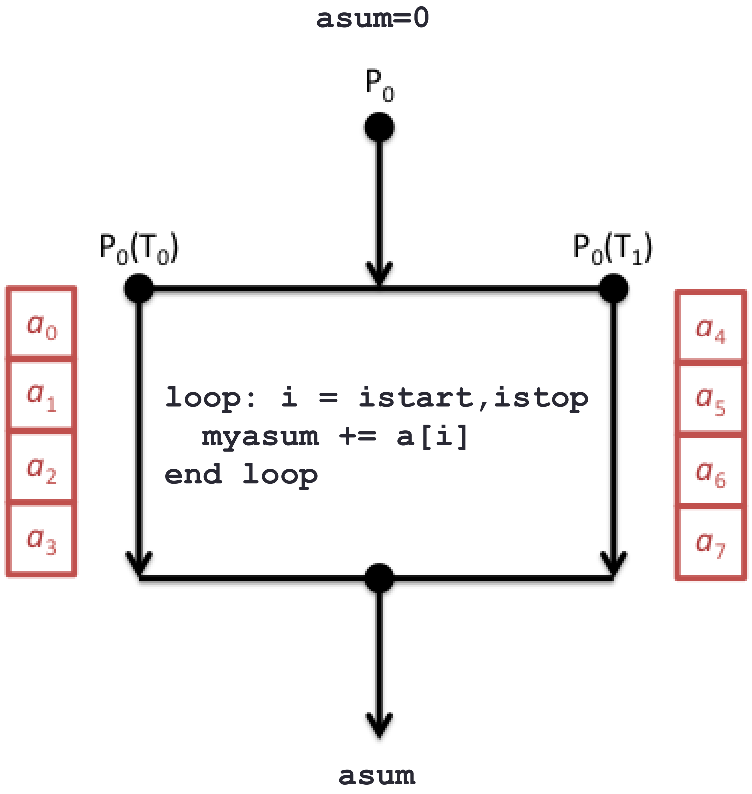


Synchronisation

- 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!

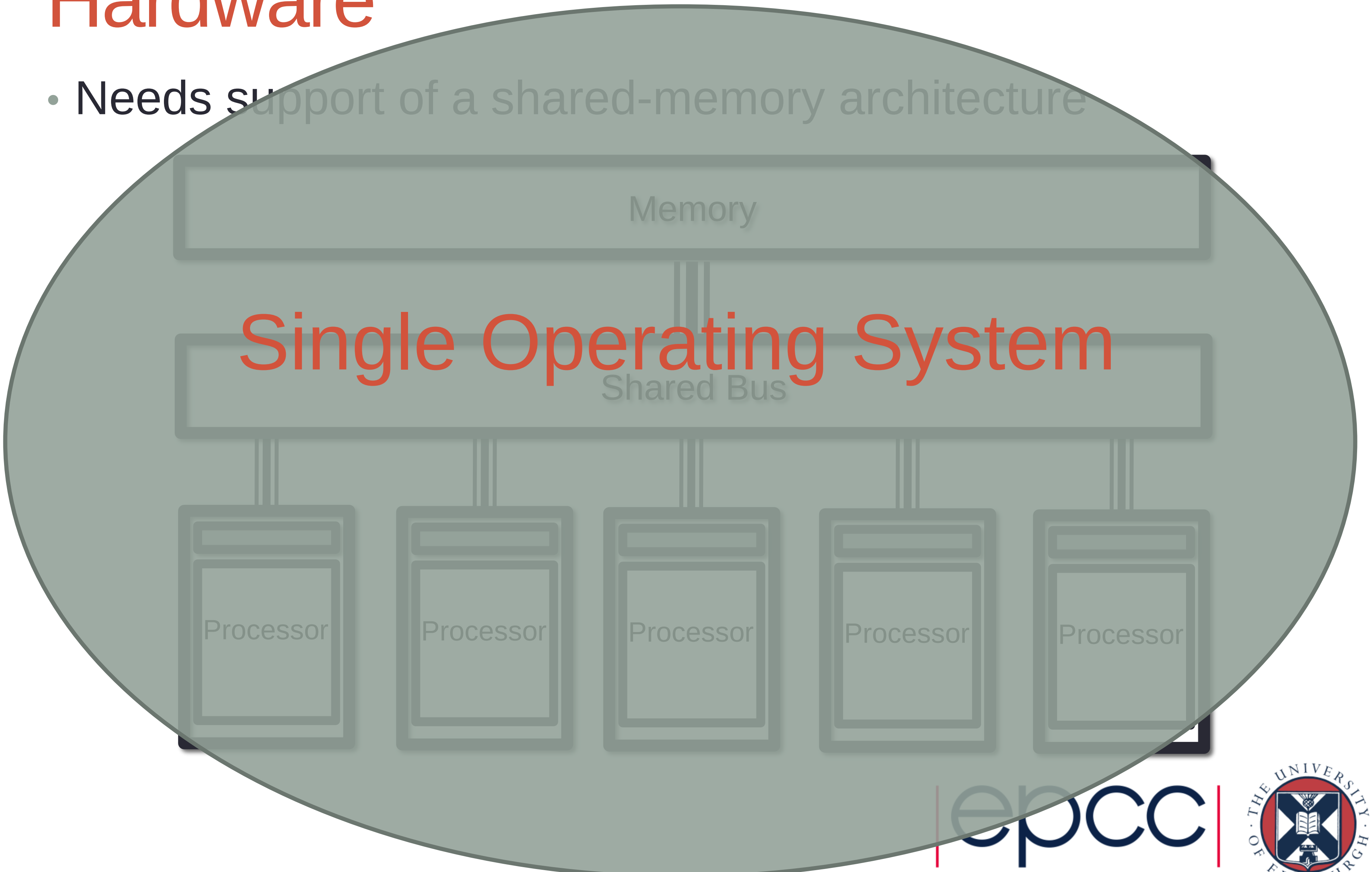
Specific example

- Computing **asum = a$_0$+ a$_1$ + … 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**

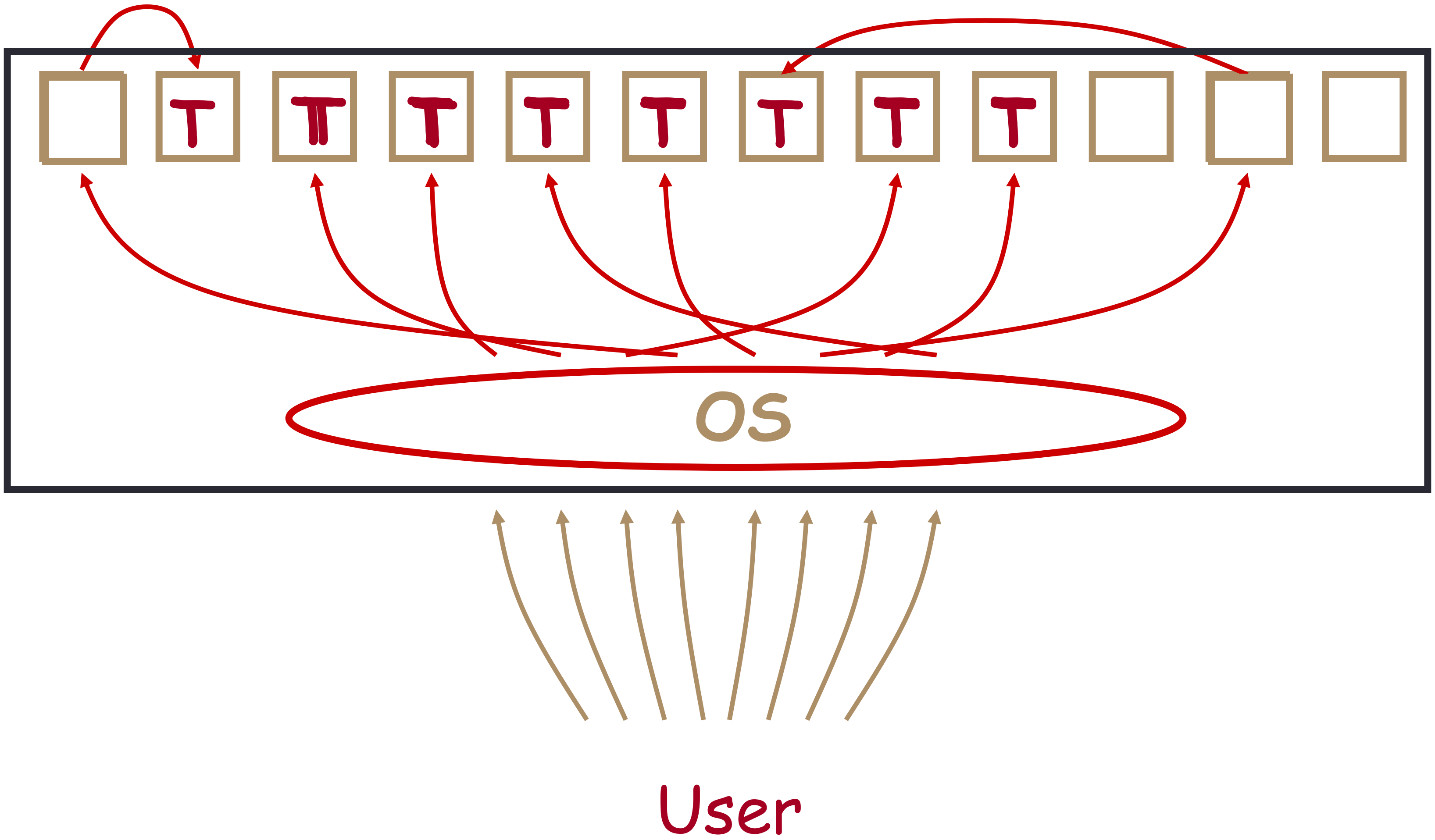


Hardware

- Needs support of a shared-memory architecture



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. 28 cores on Bridges)
  - 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 – see later)
  - on Bridges: could run one process per node, 28 threads per process
  - or 2 procs per node / 14 threads per process
  - or 4 / 7 ...

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
    - also have private data for local variables
    - 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