



Shared Memory Programming with OpenMP

Concepts



Overview



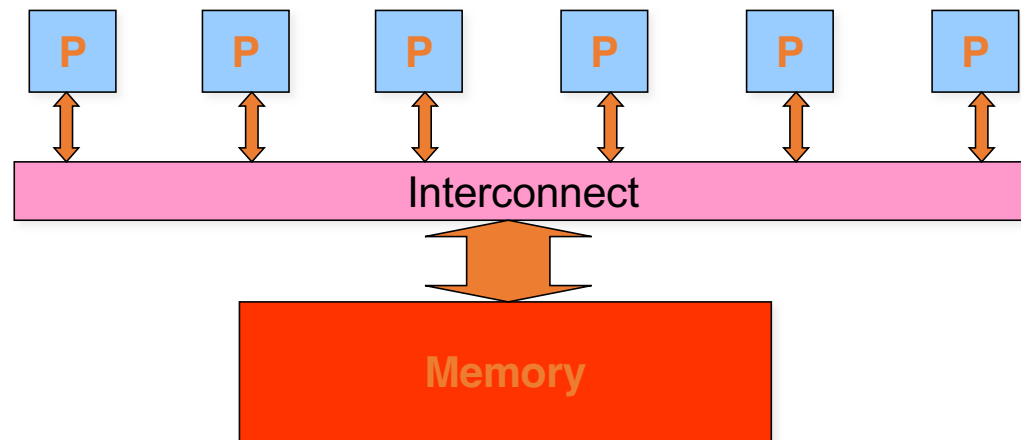
- Shared memory systems
- Basic Concepts in Threaded Programming

Shared memory systems



- Threaded programming is most often used on shared memory parallel computers.
- A shared memory computer consists of a number of processing units (CPUs/cores) together with some memory
- Key feature of shared memory systems is a *single address space* across the whole memory system.
 - every CPU/core can read and write all memory locations in the system
 - one logical memory space
 - all CPUs/cores refer to a memory location using the same address

Conceptual model

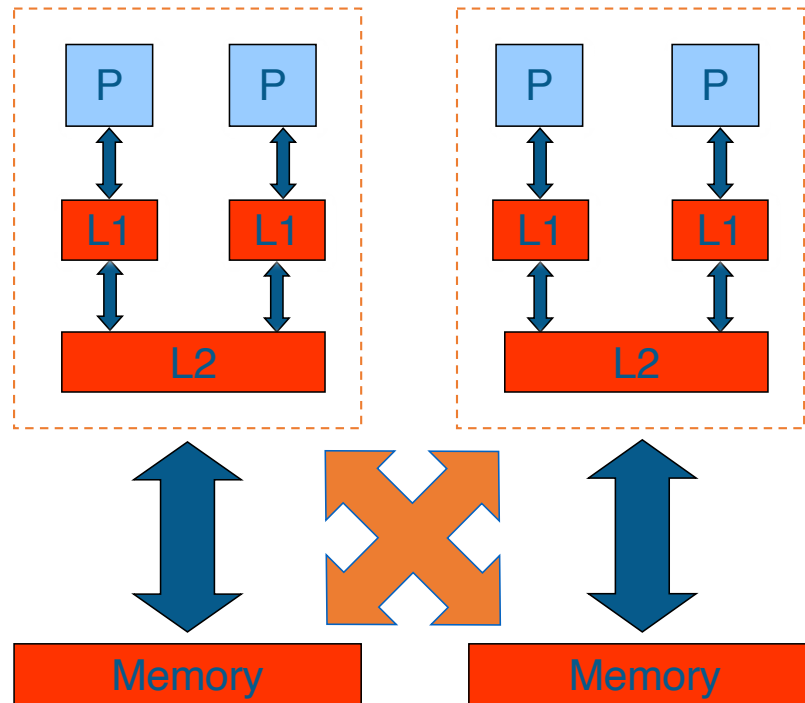


Real hardware



- Real shared memory hardware is more complicated than this.....
 - Memory may be split into multiple smaller units
 - There may be multiple levels of cache memory
 - some of these levels may be shared between subsets of processors
 - The interconnect may have a more complex topology
-but a single address space is still supported
 - Hardware complexity can affect performance of programs, but not their correctness

Real hardware example

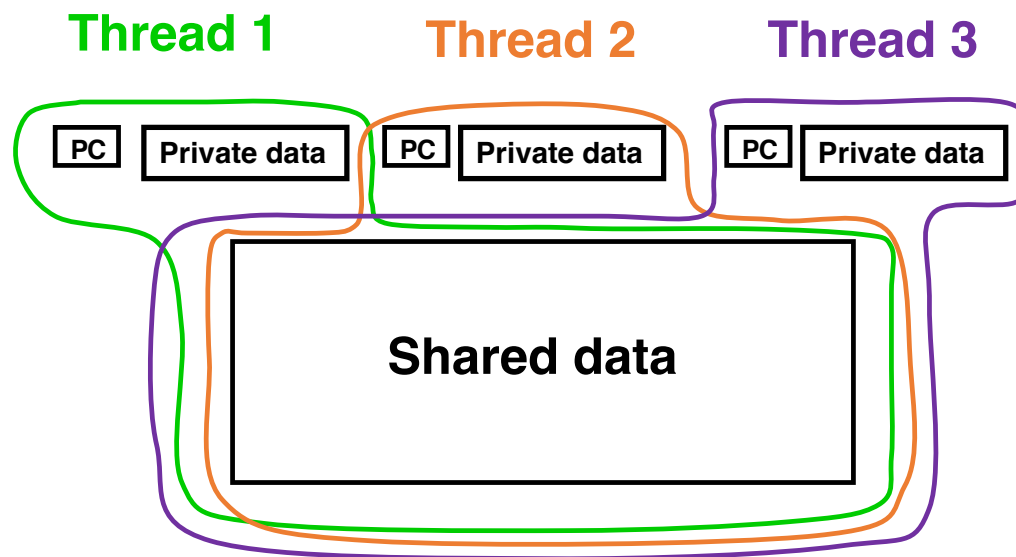


Threaded Programming Model



- The programming model for shared memory is based on the notion of 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.)

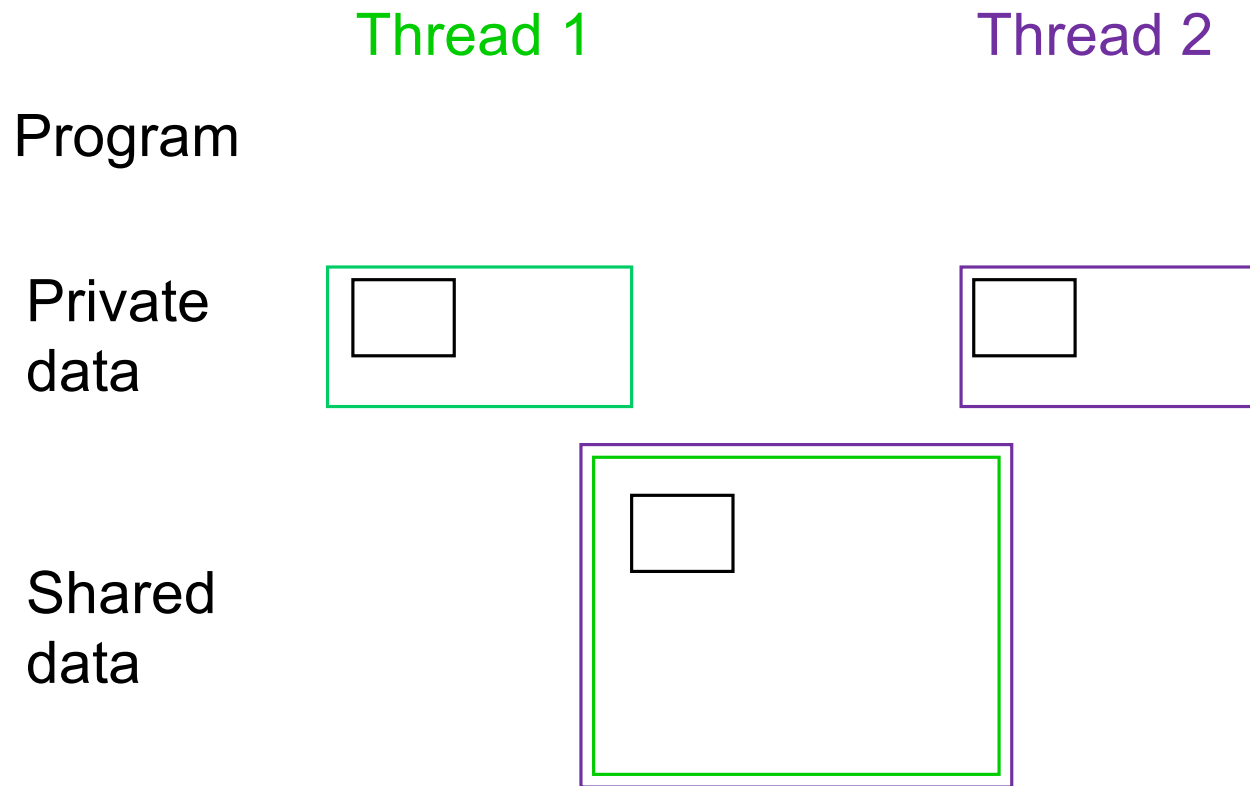


Thread communication

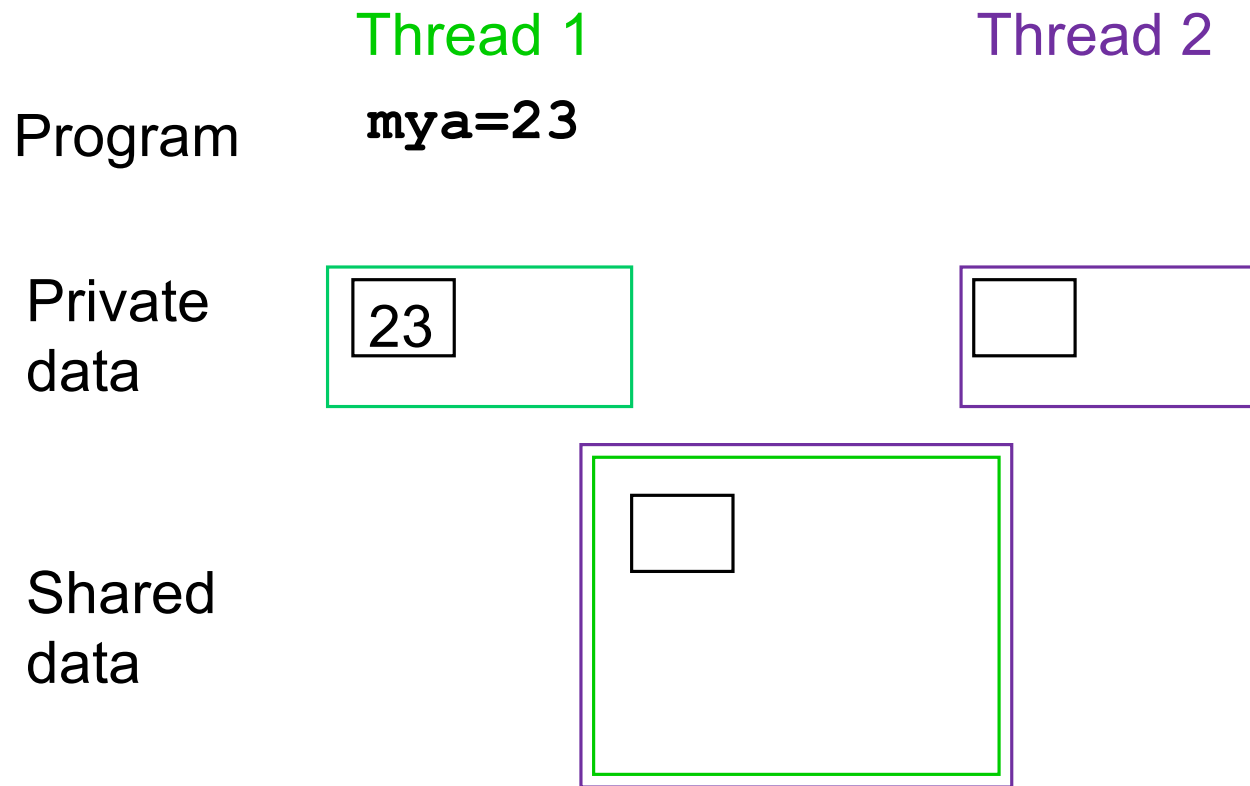


- In order to have useful parallel programs, threads must be able to exchange data with each other
- Threads communicate with each other via reading and writing shared data
 - thread 1 writes a value to a shared variable A
 - thread 2 can then read the value from A
- Note: there is no notion of messages in this model

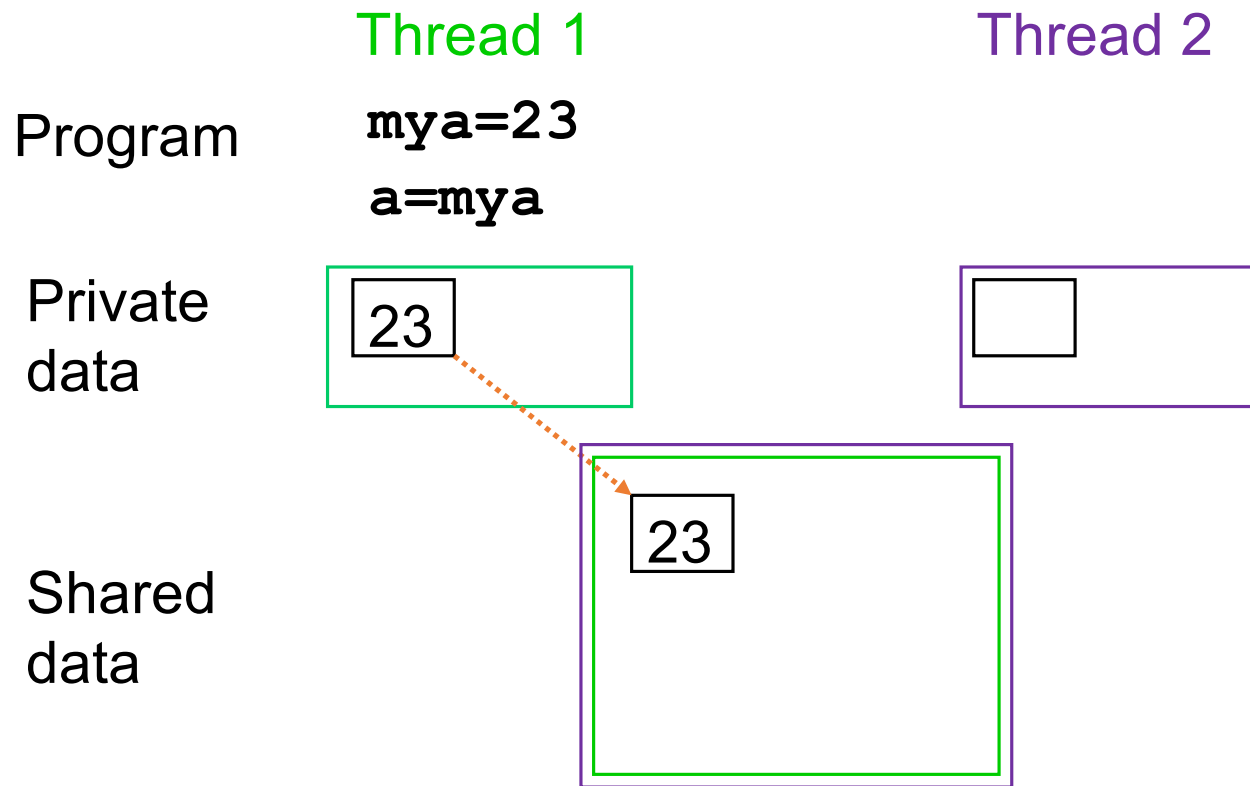
Thread Communication



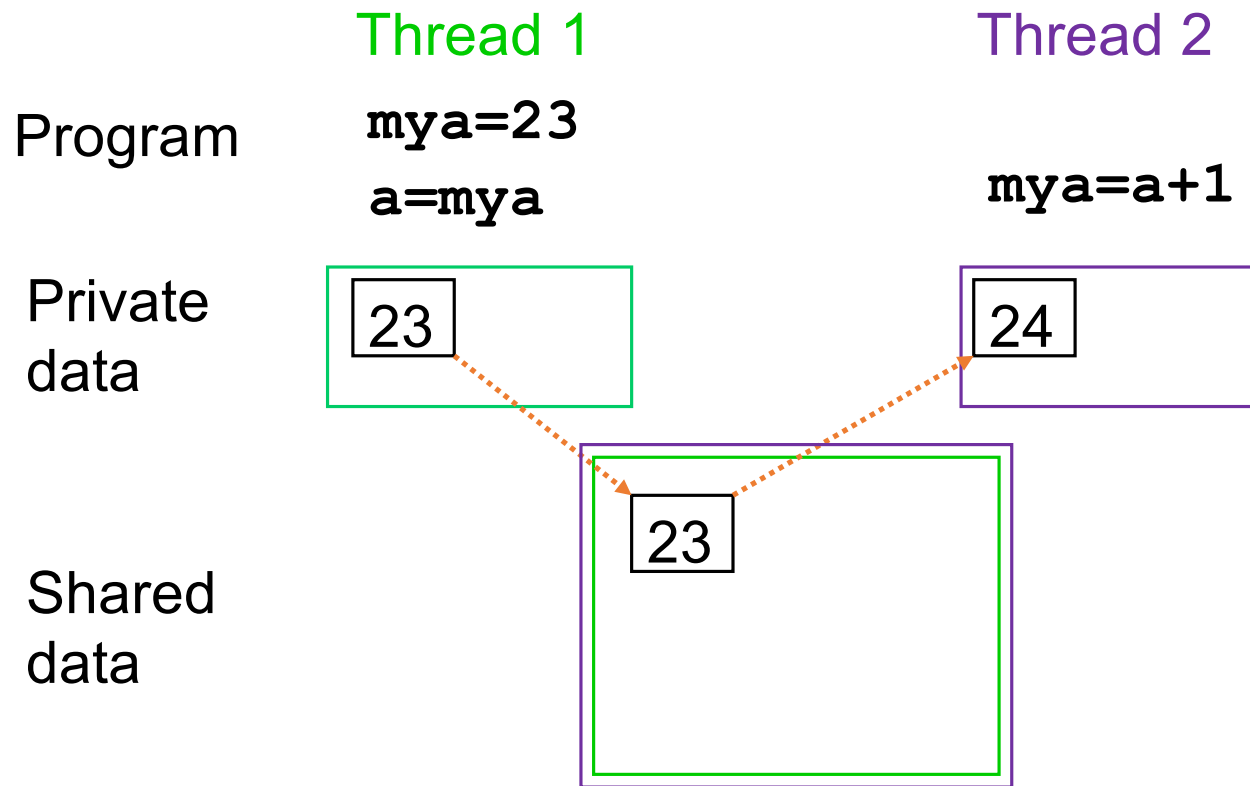
Thread Communication



Thread Communication



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.

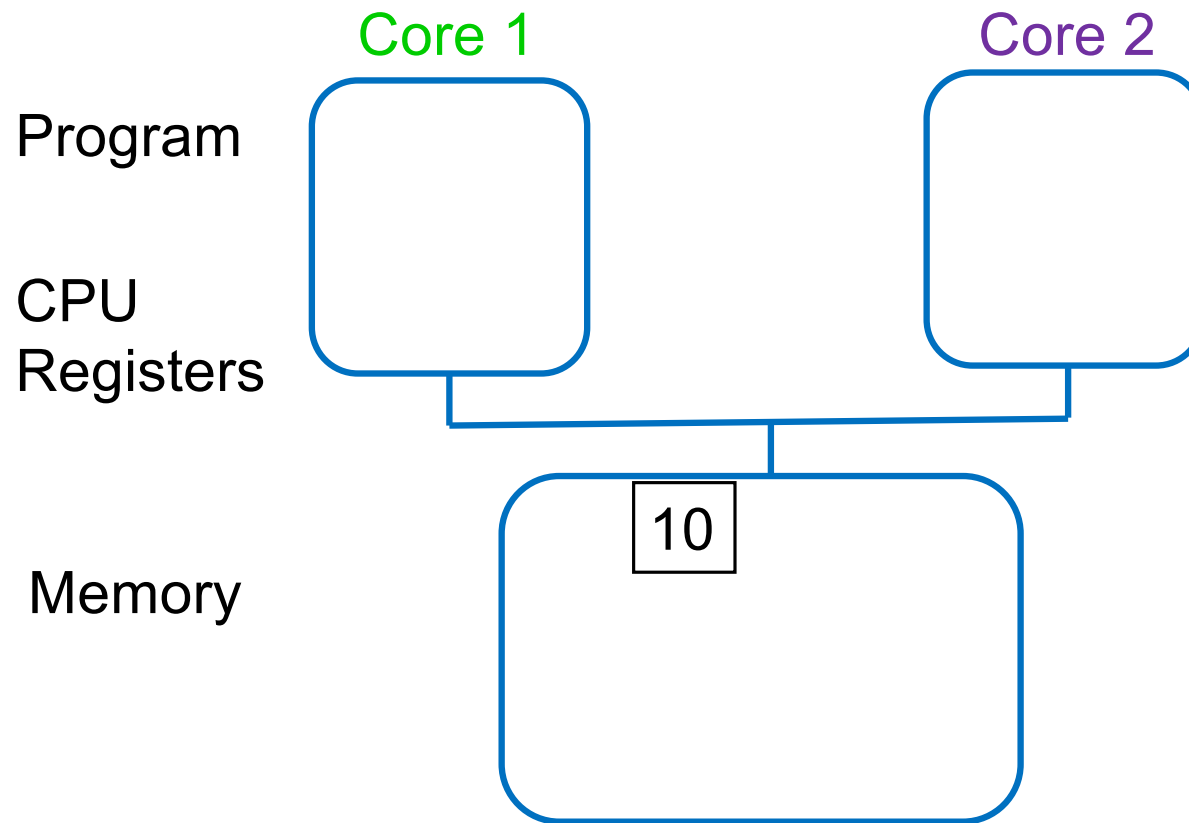
thread 1 must write variable A before thread 2 reads it,

or

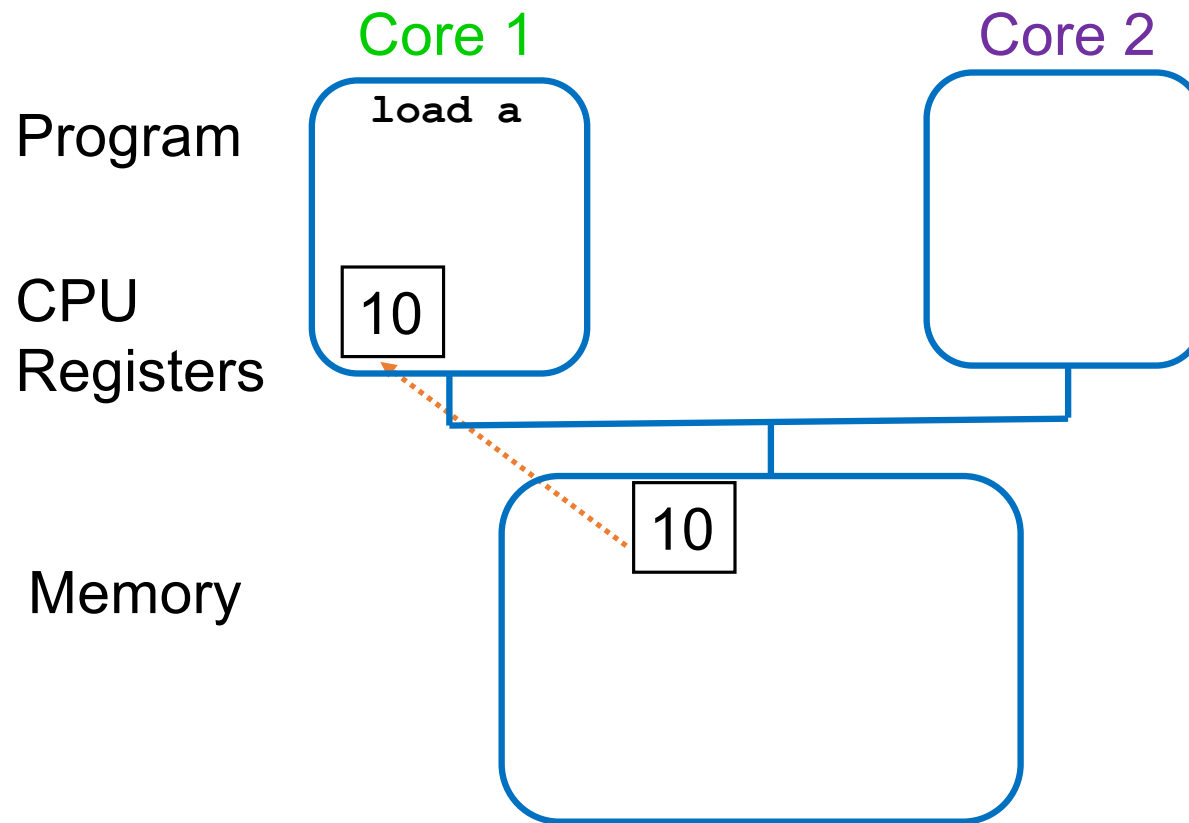
thread 1 must read variable A before thread 2 writes it.

- Note that updates to shared variables (e.g. $a = a + 1$) are *not* atomic!
- Requires (at least) 3 instructions: load, add, store
- If two threads try to do this “at the same time”, the instructions can be interleaved in time such that one of the updates may get overwritten.

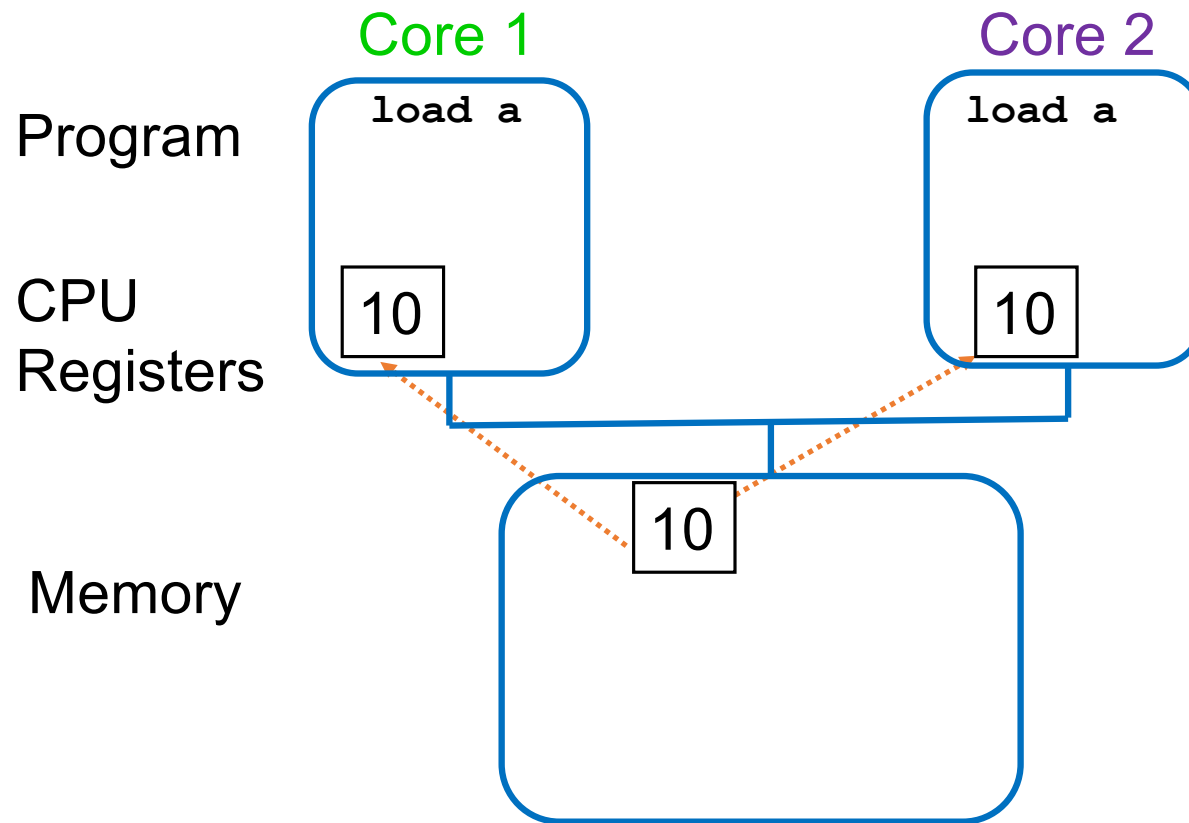
Synchronisation example



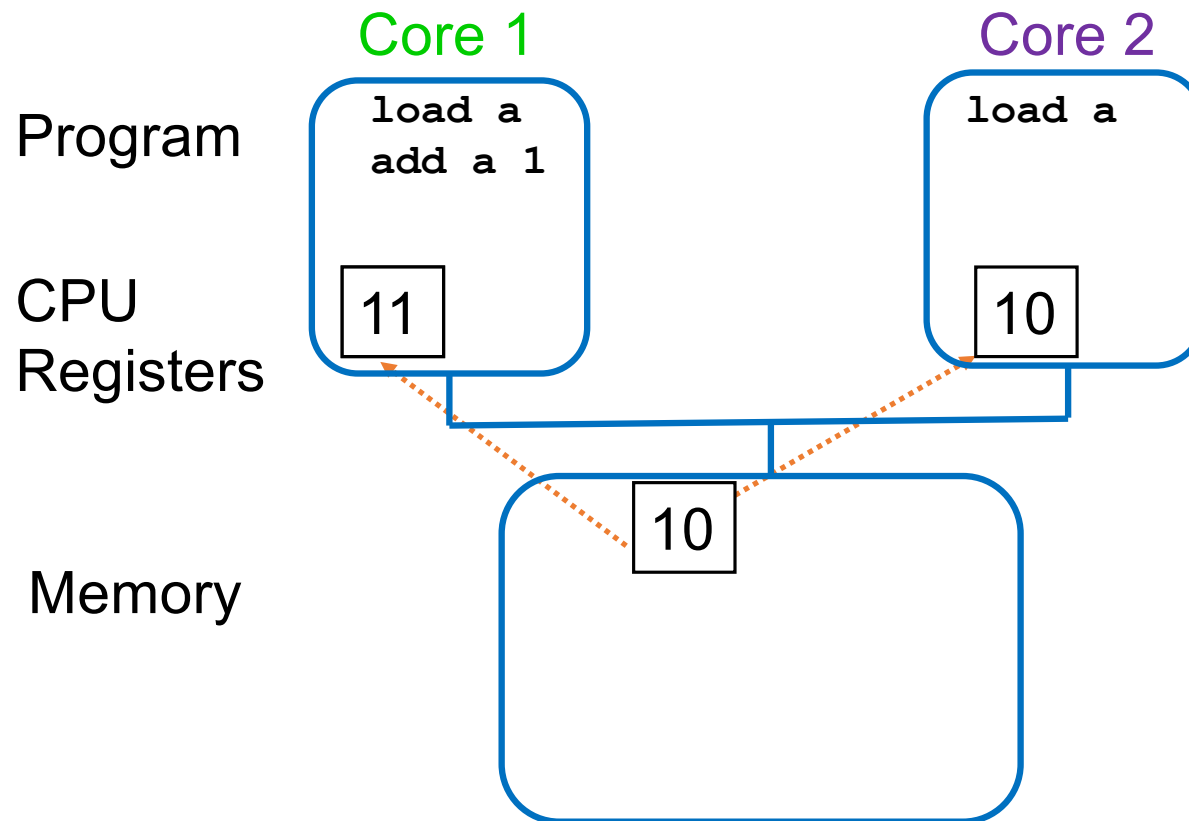
Synchronisation example



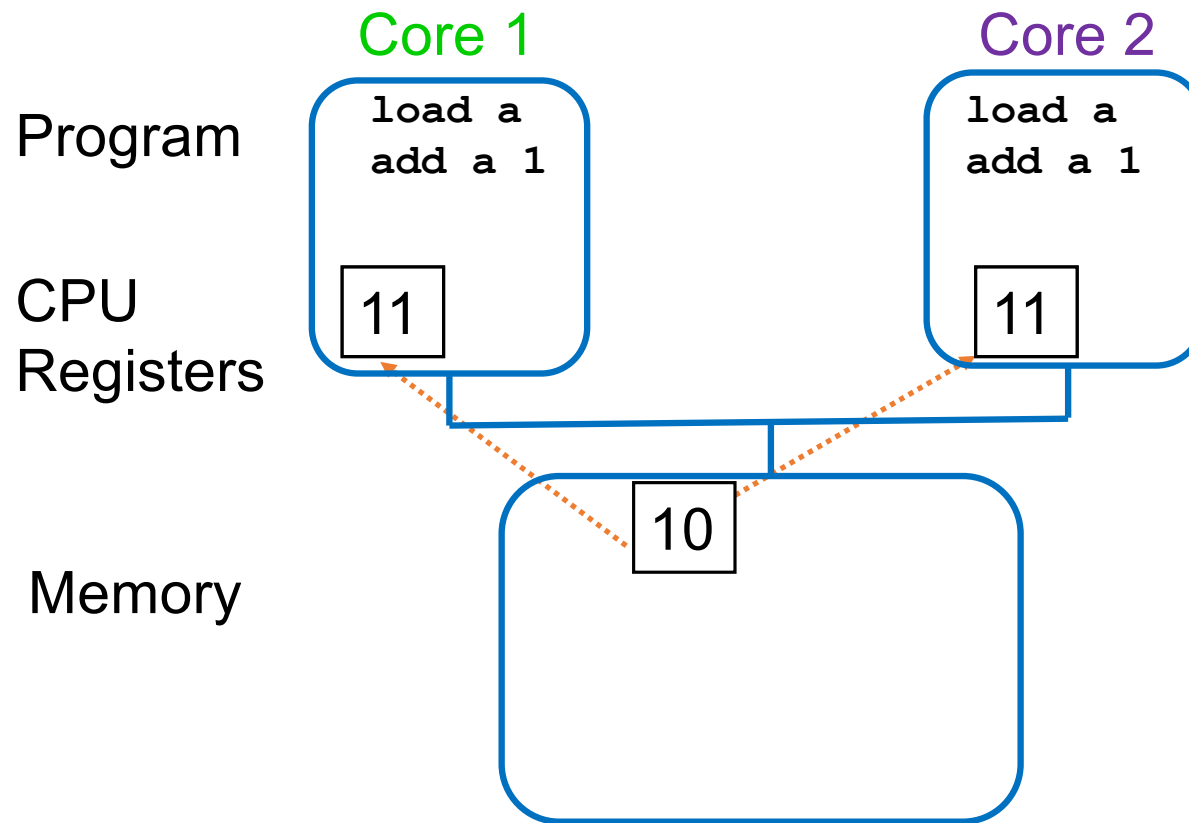
Synchronisation example



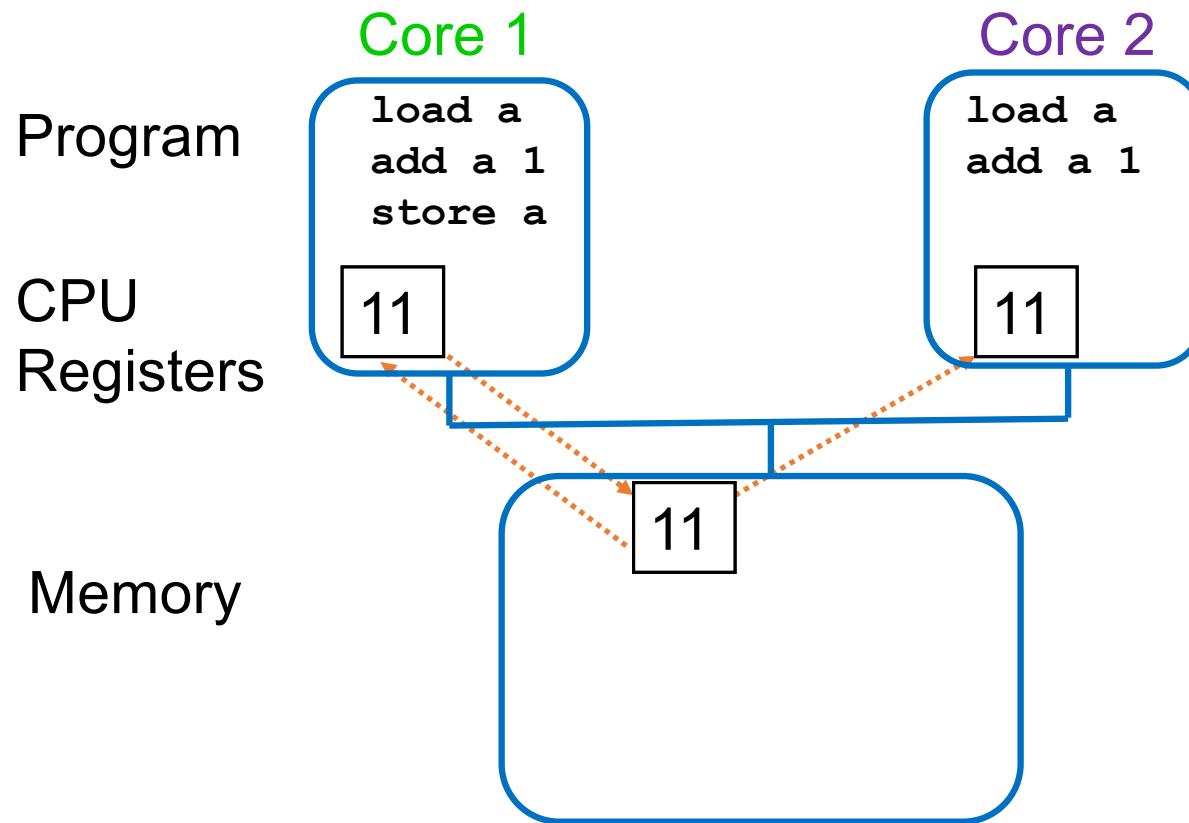
Synchronisation example



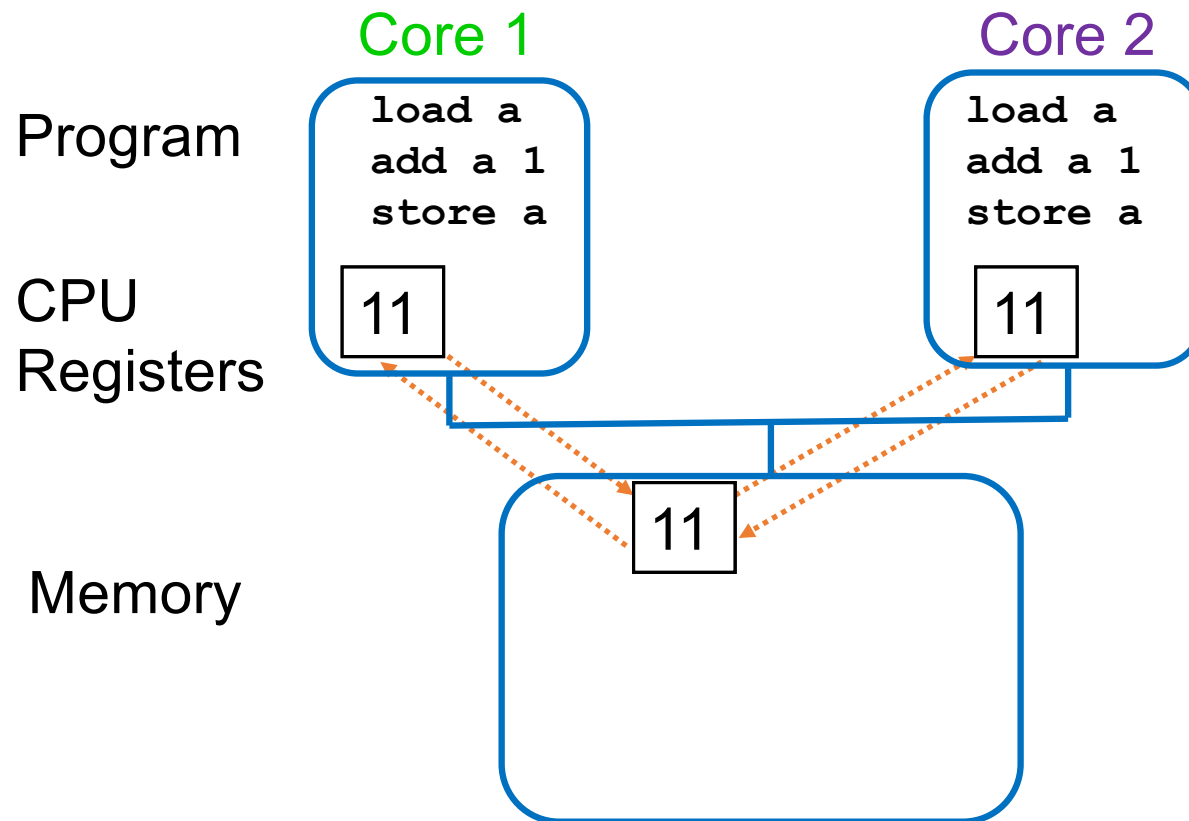
Synchronisation example



Synchronisation example



Synchronisation example



Race conditions

- This problem is an example of a **race condition**.
- A race condition occurs when multiple threads access the same memory location without synchronization, and at least one of the accesses is a write.
- Without synchronization, different time orderings of the accesses may occur, possibly resulting in non-deterministic behavior
 - Running the same program with the same inputs twice may give different results
- Very nasty type of bug
 - Not reliably detectable by conventional software testing procedures

Tasks



- A *task* is a piece of computation which can be executed independently of other tasks
- In principle we could create a new thread to execute every task
 - in practise this can be too expensive, especially if we have large numbers of small tasks
- Instead tasks can be executed by a pre-existing *pool* of threads
 - tasks are submitted to the pool
 - some thread in the pool executes the task
 - at some point in the future the task is guaranteed to have completed
- Tasks may or may not be ordered with respect to other tasks

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.

Reductions

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

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

- Allowing only one thread at a time to update **b** would remove all parallelism.
- Instead, each thread can accumulate its own private copy, then these copies are 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

Reusing this material



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

<https://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.