Barriers in OpenMP

Paolo Burgio
paolo.burgio@unimore.it

Outline

- Expressing parallelism
  - Understanding parallel threads
- ~~Memory~~ Data management
  - Data clauses
- Synchronization
  - Barriers, locks, critical sections
- Work partitioning
  - Loops, sections, single work, tasks…
- Execution devices
  - Target

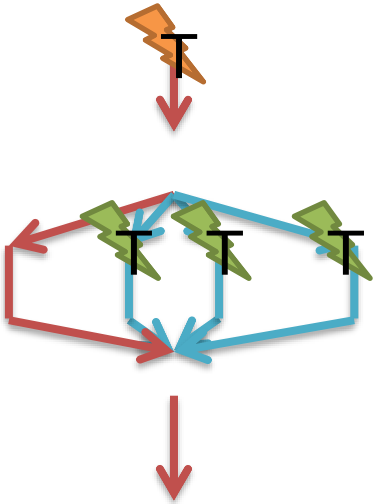
OpenMP synchronization

- OpenMP provides the following synchronization constructs:
  - `barrier`
  - `flush`
  - `master`
  - `critical`
  - `atomic`
  - `taskwait`
  - `taskgroup`
  - `ordered`
  - ..and OpenMP locks

Creating a parreg

- Master-slave, fork-join execution model
  - Master thread spawns a team of Slave threads
  - They all perform computation in parallel
  - At the end of the parallel region, implicit barrier

```
int main()
{
  /* Sequential code */

  #pragma omp parallel num_threads(4)
  {

    /* Parallel code */

  } // Parreg end: (implicit) barrier

  /* (More) sequential code */
}
```



OpenMP explicit barriers

```
#pragma omp barrier new-line
```

(a standalone directive)

- All threads in a team must wait for all the other threads before going on
  - "Each barrier region must be encountered by all threads in a team or by none at all"
  - "The sequence of barrier regions encountered must be the same for every thread in a team"
  - Why?
- Binding set is the team of threads from the innermost enclosing parreg
  - "It applies to"
- Also, it enforces a consistent view of the shared memory
  - We'll see this..

Exercise

Let's code!

- Spawn a team of (many) parallel Threads
  - Printing "Hello World"
  - Put a `#pragma omp barrier`
  - Reprint "Hello World" after
- What do you see?
  - Now, remove the `barrier` construct
- Now, put the barrier inside an `if`
  - E.g., `if(omp_get_thread_num() == 0) { ... }`
  - What do you see?
  - Error!!!!

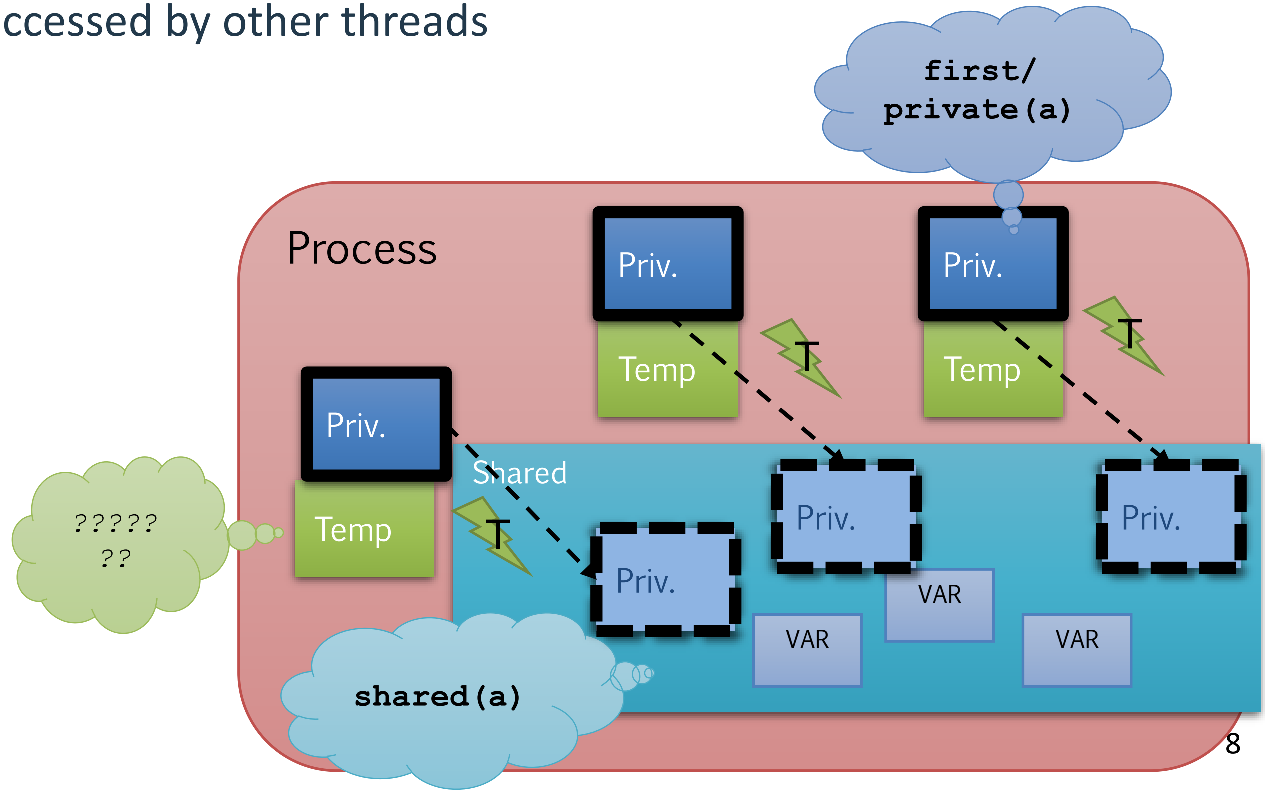
Effects on memory

- Besides synchronization, a barrier has the effect of making threads' temporary view of the shared memory consistent
  - You cannot trust any (potentially modified) `shared` vars before a barrier
  - Of course, there are no problems with `private` vars

- ..what???

The OpenMP memory model

- Shared memory with relaxed consistency
  - Threads have access to "a place to store and to retrieve variables, called the memory"
  - Threads can have a temporary view of the memory
    - Caches, registers, scratchpads…
    - Can still be accessed by other threads



A bit of architecture…

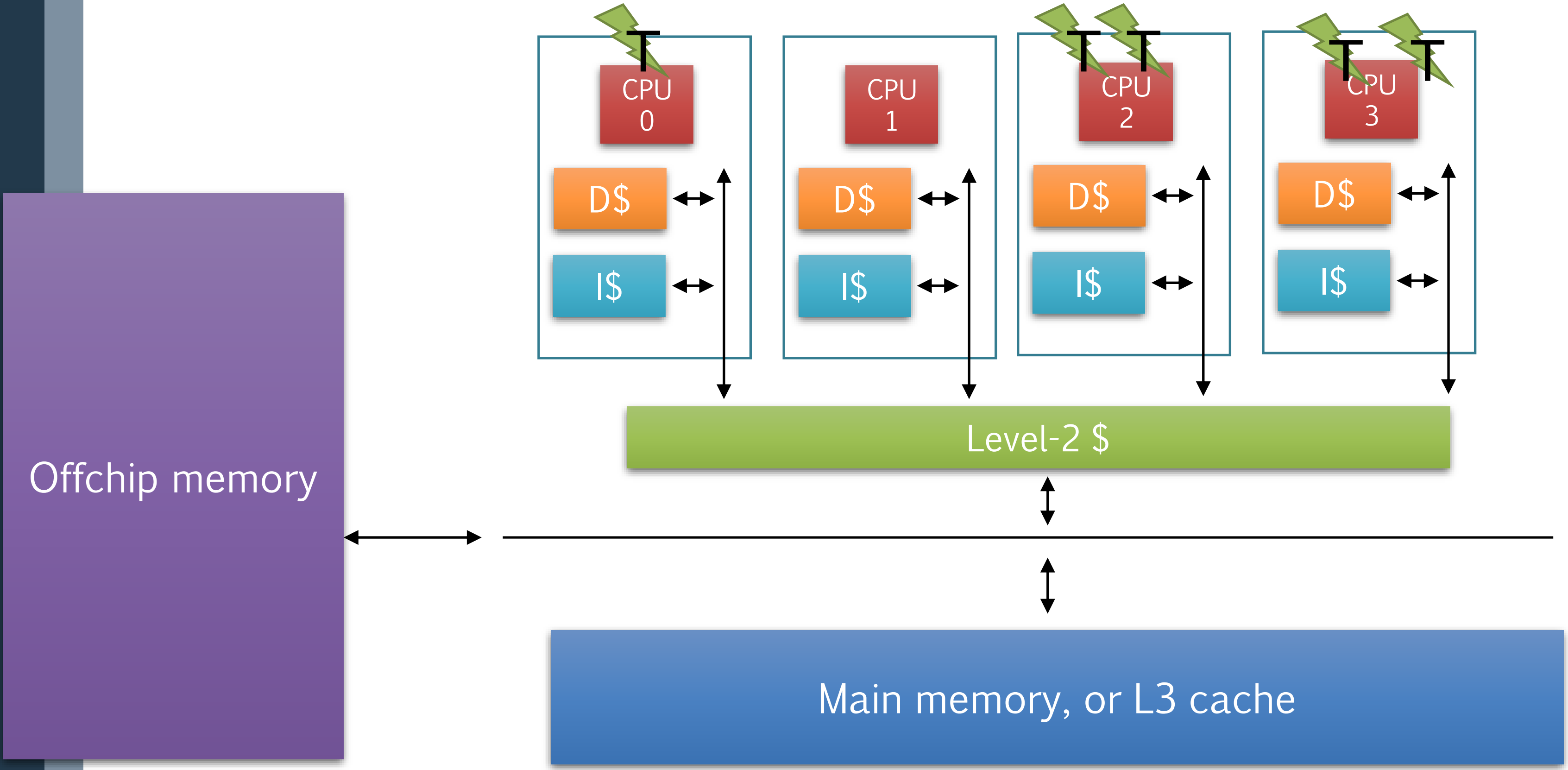
Caches in a nutshell

- A quick memory connected to the core processor
  - ..and to the main memory
  - Few KB of data
- (If any,) caches are a pure hardware mechanism
  - Used to store a copy mostly accessed data
  - To speedup execution even by 10-20 times
  - Istruction caches/Data caches
- They perform their work automatically
  - And transparently
  - Poor or no control at all at application level
  - Extremely dangerous in multi- and many-cores

Caches

eng.wikipedia.org

A cache is a hardware or software component that stores data so future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation, or the *duplicate of data stored elsewhere.*

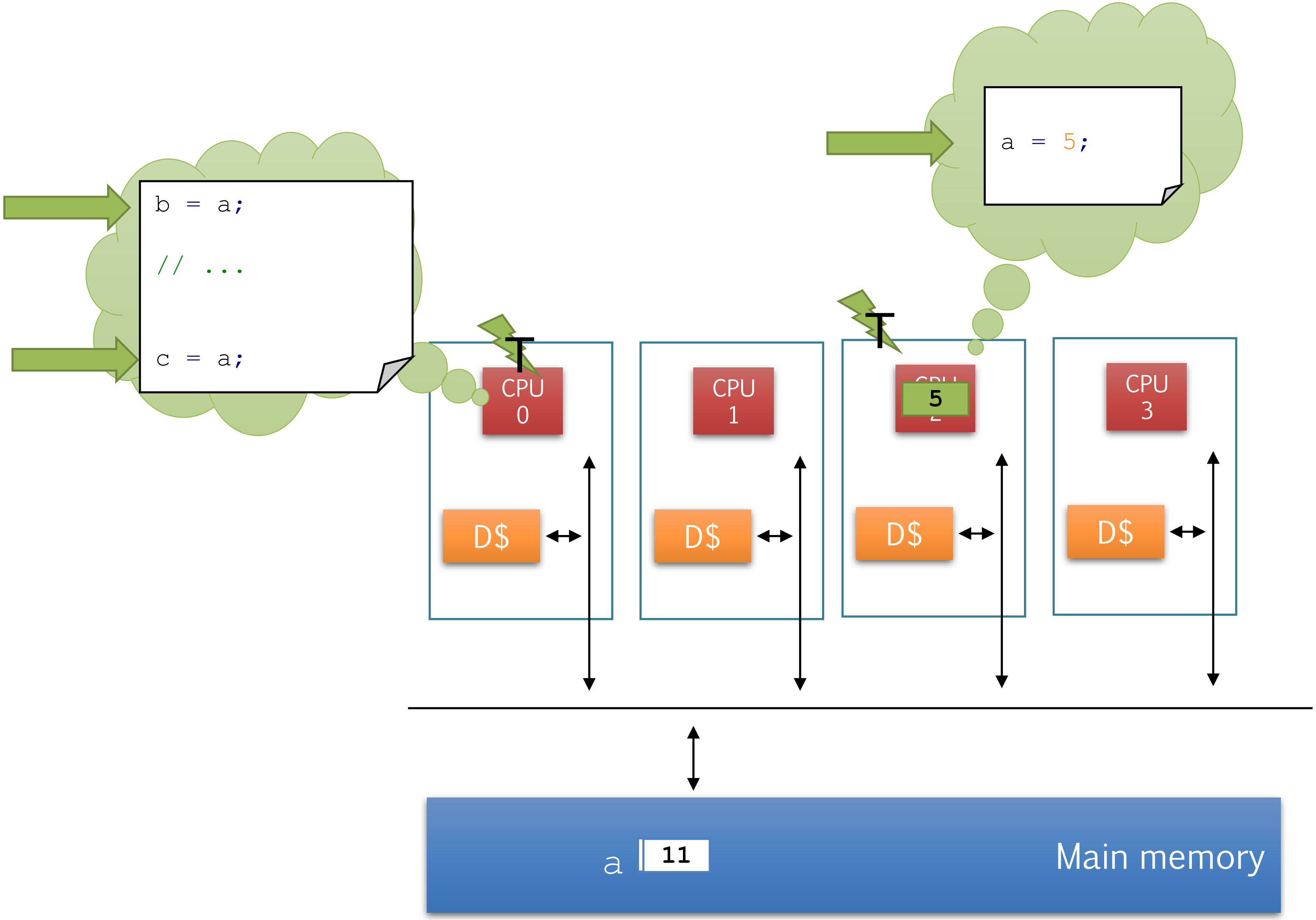


The catch(es)

- Caches are power hungry
  - Some embedded architectures do not have D$
- They are not suitable for critical systems
  - E.g., BOSCH removed D$s
- Hardware mechanism, poor control on them
  - Flush command (typically, all cache)
  - Color cache (assign to threads)
  - Prefetch (move data before it's actually needed)

Coherency problem in multi/many-cores!!

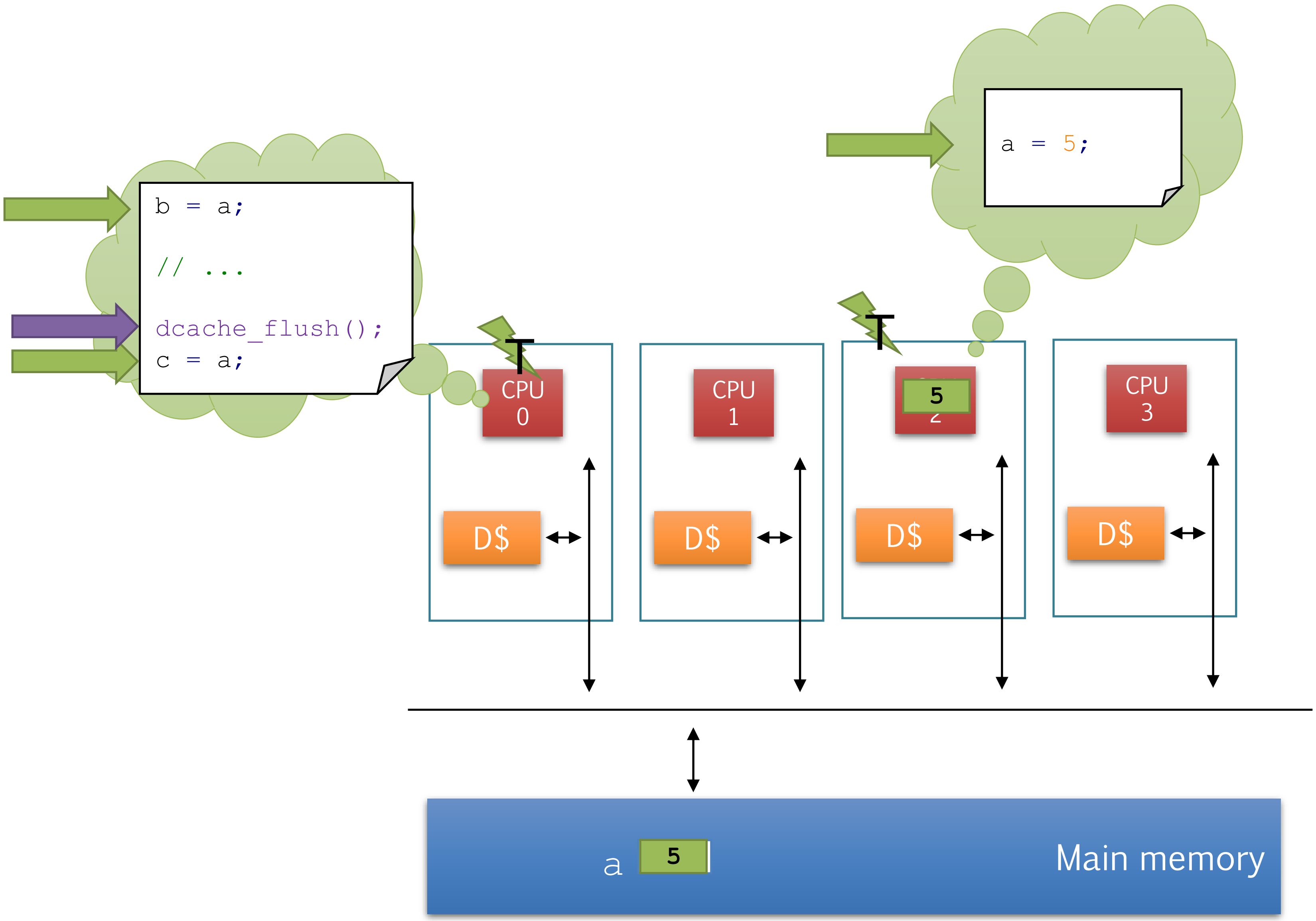
An example: read stale data



```
b = a;

// ...

c = a;
```

```
a = 5;
```

An example: read stale data

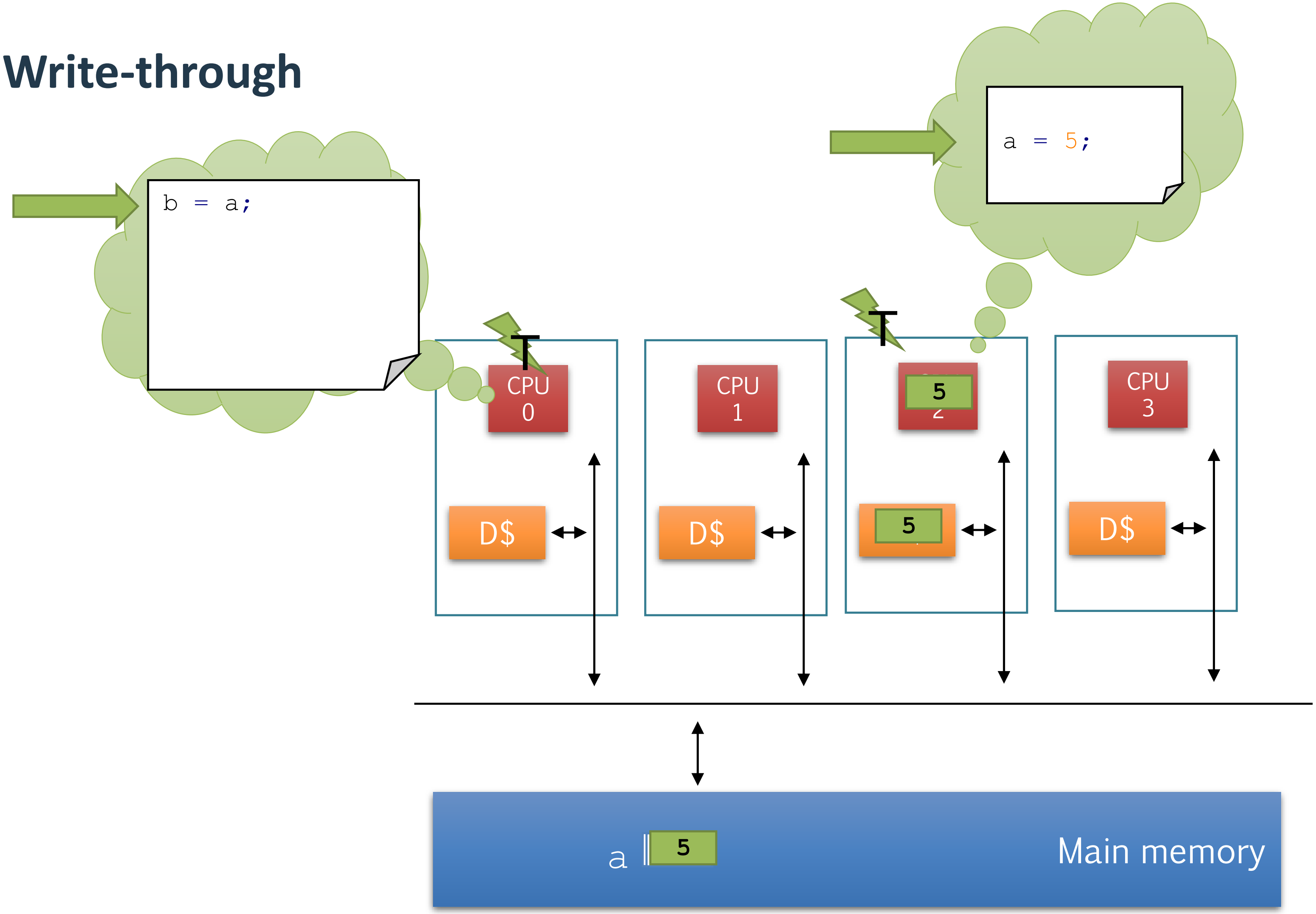


```
b = a;

// ...

dcache_flush();
c = a;
```

```
a = 5;
```

An(other) example: $ writing policies

Write-through

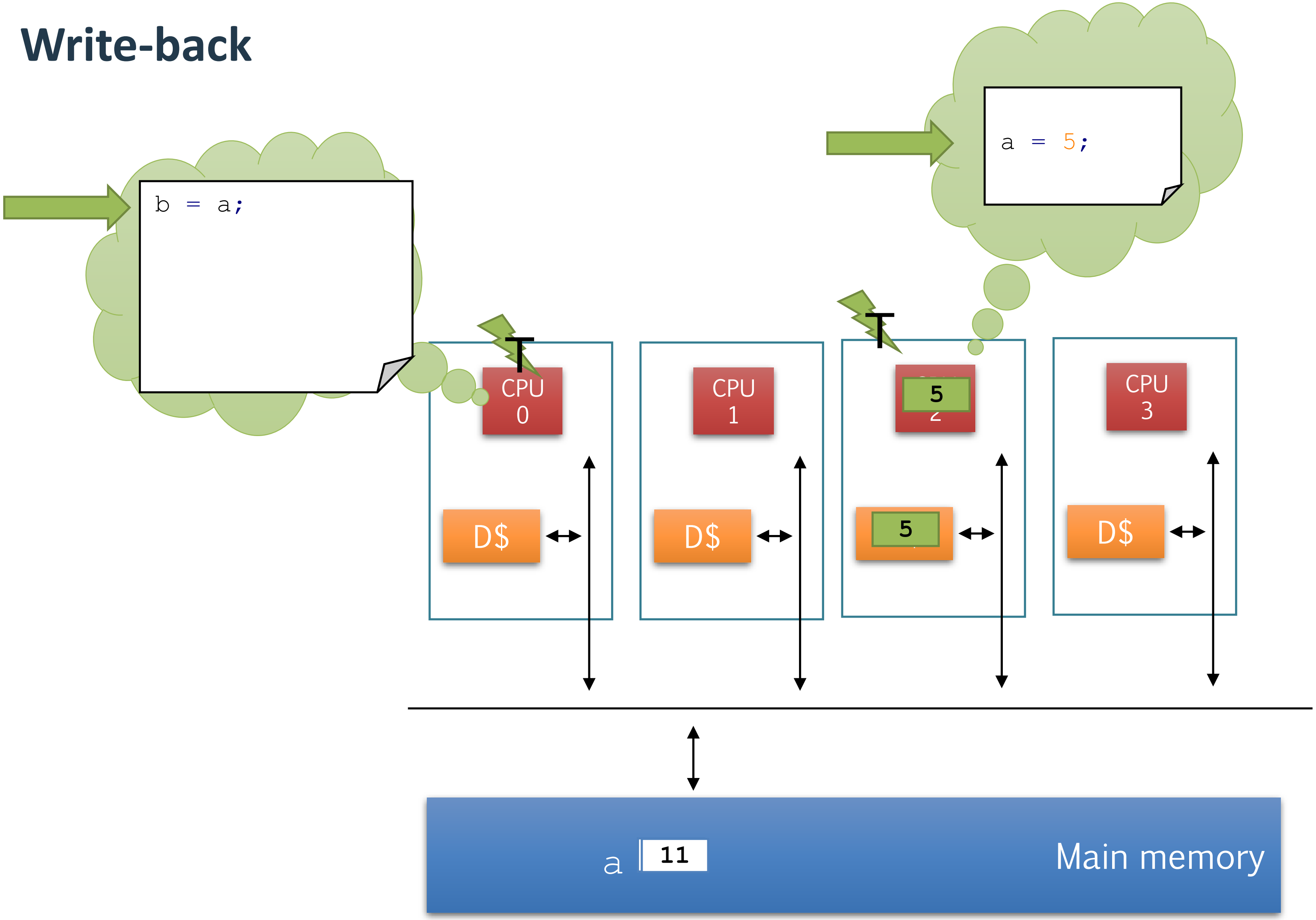


```
b = a;
```

```
a = 5;
```

An(other) example: $ writing policies

Write-back

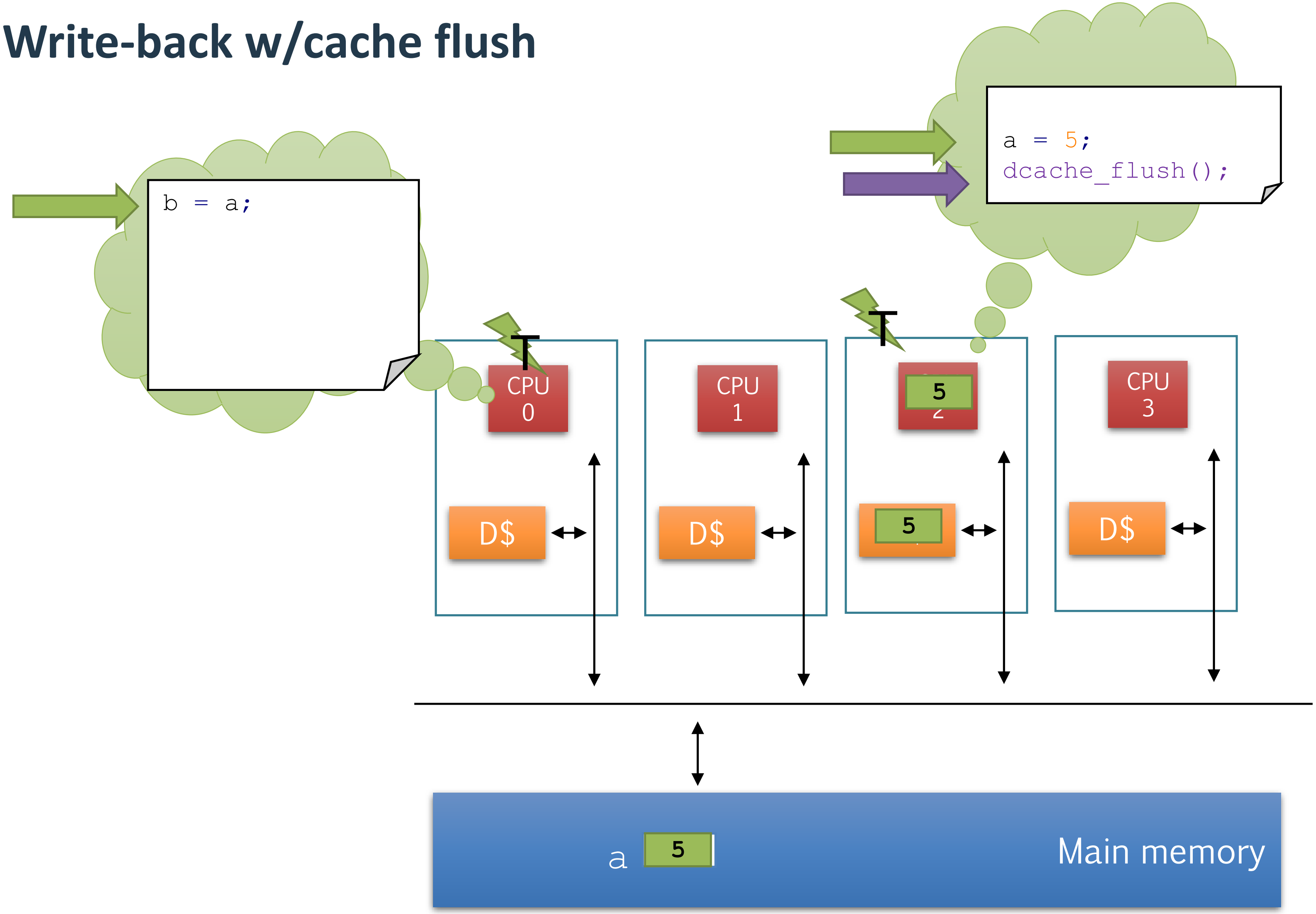


```
b = a;
```

```
a = 5;
```

An(other) example: $ writing policies

Write-back w/cache flush



```
b = a;
```

```
a = 5;
dcache_flush();
```

The `flush` directive

```
#pragma omp flush [(list)] new-line
```

- Binding thread set is the encountering thread
  - More "relaxed"
- "It executes the OpenMP flush operation"
  - Makes its temporary view of the shared memory consistent with other threads
  - "Calls to `dcache_flush()`"
- Enforces an order on the memory operations on the variables specified in `list`

Semantics: `barrier` vs `flush`

`#pragma omp barrier`

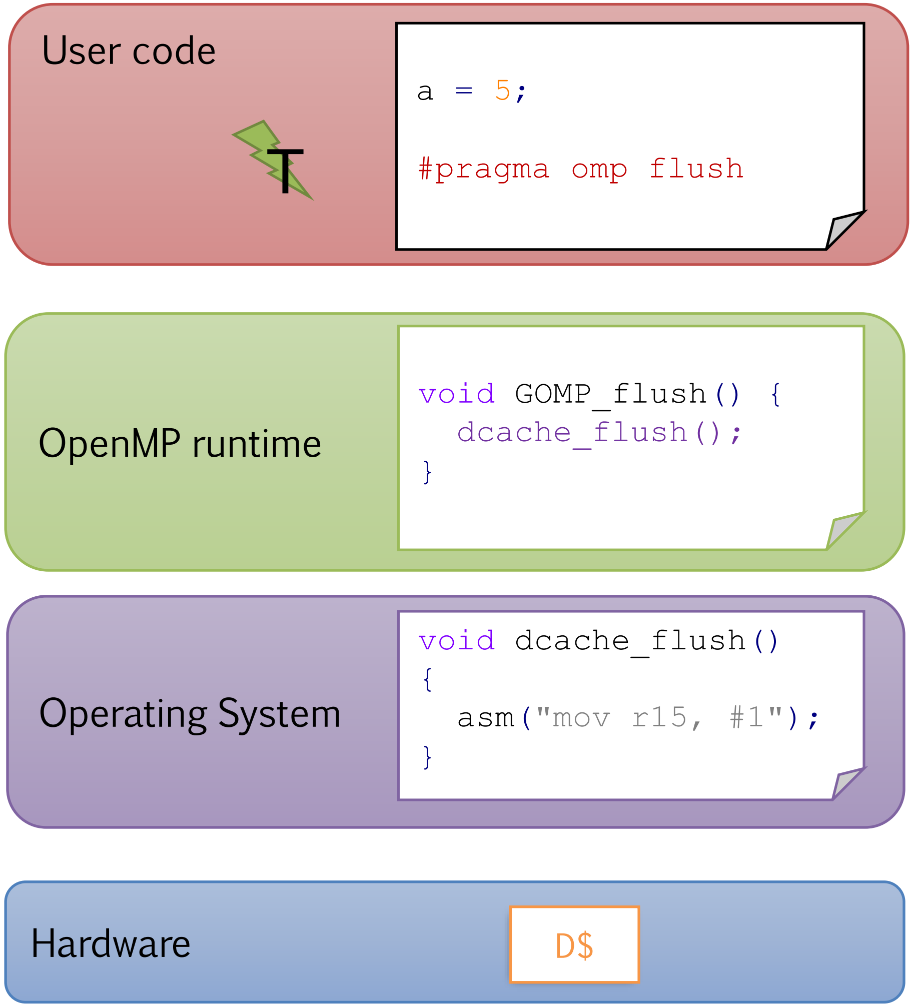
- Joins the threads of a team
- Applies to all threads of a team
- Forces consistency of threads' temporary view of the shared memory

`#pragma omp flush`

- Applies to one thread
- Forces consistency of its temporary view of the shared memory
- Much lighter!

OpenMP software stack

- Multi-layer stack
  - Engineered for portability



User code

```
a = 5;

#pragma omp flush
```

OpenMP runtime

```
void GOMP_flush() {
  dcache_flush();
}
```

Operating System

```
void dcache_flush()
{
  asm("mov r15, #1");
}
```

Hardware

How to run the examples

Let's code!

- Download the `Code/` folder from the course website

- Compile
- `$ gcc -fopenmp code.c -o code`

- Run (Unix/Linux)

```
$ ./code
```

- Run (Win/Cygwin)

```
$ ./code.exe
```

References

- "Calcolo parallelo" website
  - http://hipert.unimore.it/people/paolob/pub/Calcolo_Parallelo/
- My contacts
  - paolo.burgio@unimore.it
  - http://hipert.mat.unimore.it/people/paolob/
- Useful links
  - http://www.google.com
  - http://www.openmp.org
  - https://gcc.gnu.org/
- A "small blog"
  - http://www.google.com