Barriers and critical regions

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

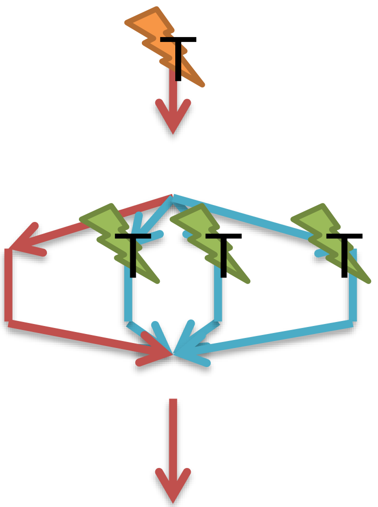
OpenMP synchronization

- 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..

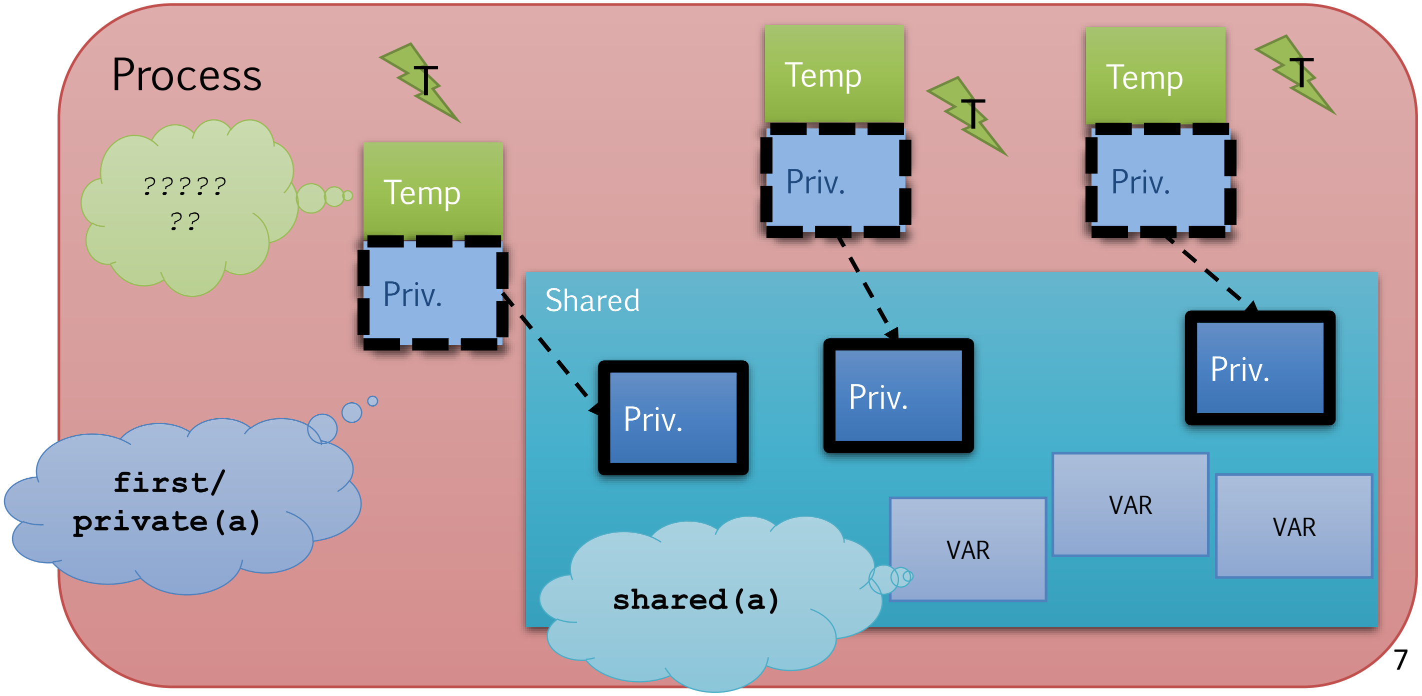
Effects on memory

- 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



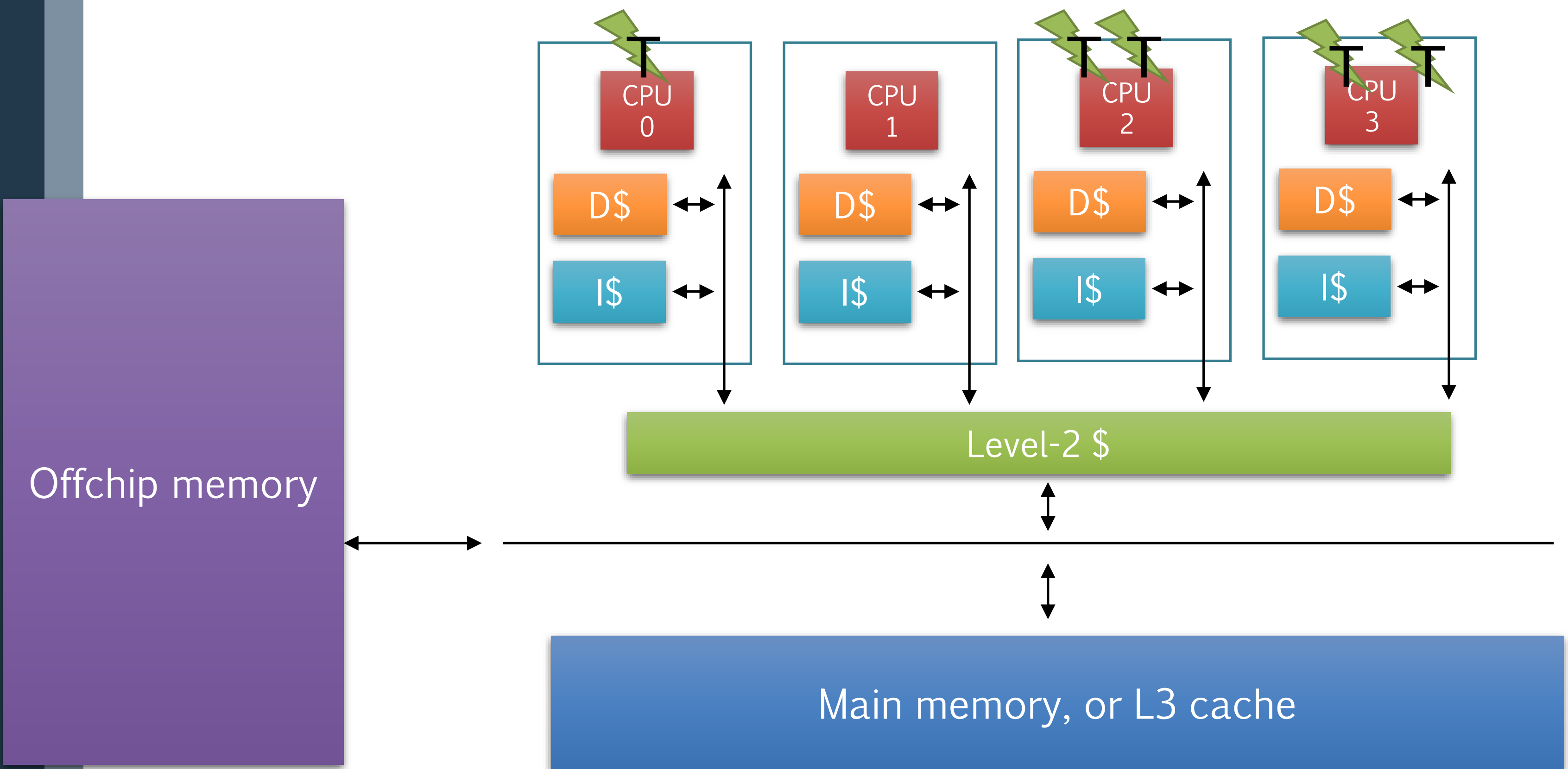
Temporary memory => Caches

- 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!!

Semantics: `barrier` vs `flush`

`#pragma omp barrier`

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

`#pragma omp flush`

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

OpenMP synchronization

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

OpenMP locks

- Defined at the OpenMP runtime level
  - Symbols available in code including `omp.h` header

- General-purpose locks
  1. Must be initialized
  2. Can be set
  3. Can be unset

- Each lock can be in one of the following states
  1. Uninitialized
  2. Unlocked
  3. Locked

Locking primitives

omp.h

```
/* Initialize an OpenMP lock */
void omp_init_lock(omp_lock_t *lock);

/* Ensure that an OpenMP lock is uninitialized */
void omp_destroy_lock(omp_lock_t *lock);

/* Set an OpenMP lock. The calling thread behaves
   as if it was suspended until the lock can be set */
void omp_set_lock(omp_lock_t *lock);

/* Unset the OpenMP lock */
void omp_unset_lock(omp_lock_t *lock);
```

- The `omp_set_lock` has <u>blocking semantic</u>

OMP locks: example

- Locks **must** be
  - Initialized
  - Destroyed
- Locks can be
  - set
  - unset
  - tested
- Very simple example

```
/*** Do this only once!! */
/* Declare lock var */
omp_lock_t lock;
/* Init the lock */
omp_init_lock(&lock);



/* If another thread set the lock,
   I will wait */
omp_set_lock(&lock);


/* I can do my work being sure that no-
   one else is here */


/* unset the lock so that other threads
   can go */
omp_unset_lock(&lock);



/*** Do this only once!! */
/* Destroy lock */
omp_destroy_lock(&lock);
```

Exercise

Let's code!

- Spawn a team of (many) parallel Threads
  - Each incrementing a shared variable
  - What do you see?

- Protect the variable using OpenMP locks
  - What do you see?

- Now, comment the call to `omp_unset_lock`
  - What do you see?

Non-blocking lock set

omp.h

```
/* Set an OpenMP lock but do not suspend the execution of the thread.
   Returns TRUE if the lock was set */

int omp_test_lock(omp_lock_t *lock);
```

- Extremely useful in some cases. Instead of blocking
  - we can do useful work
  - we can increment a counter (to profile lock usage)

- Reproduce blocking set semantic using a loop
  - `while (!omp_test_lock(lock)) /* ... */;`

Let's do more

- Locks are extremely powerful
  - And low-level
- We can use them to build complex semantics
  - Mutexes
  - Semaphores..
- But they are a bit "cumbersome" to use
  - Need to initialize before, and release after
  - We can definitely do more!

pragma-level synchronization constructs

The `critical` construct

```
#pragma omp critical [(name) [hint(hint-expression)] ] new-line
  structured-block
```

Where *hint-expression* is an integer constant expressioon that evaluates to a valid lock hint

- "Restricts the execution of the associated structured block to a single thread at a time"
  - The so-called Critical Section
- Binding set: all threads everywhere (also in other teams/parregs)
- Can associate it with a "hint"
  - `omp_lock_hint_t`
  - Also locks can
  - We won't see this

The critical section

› From this…

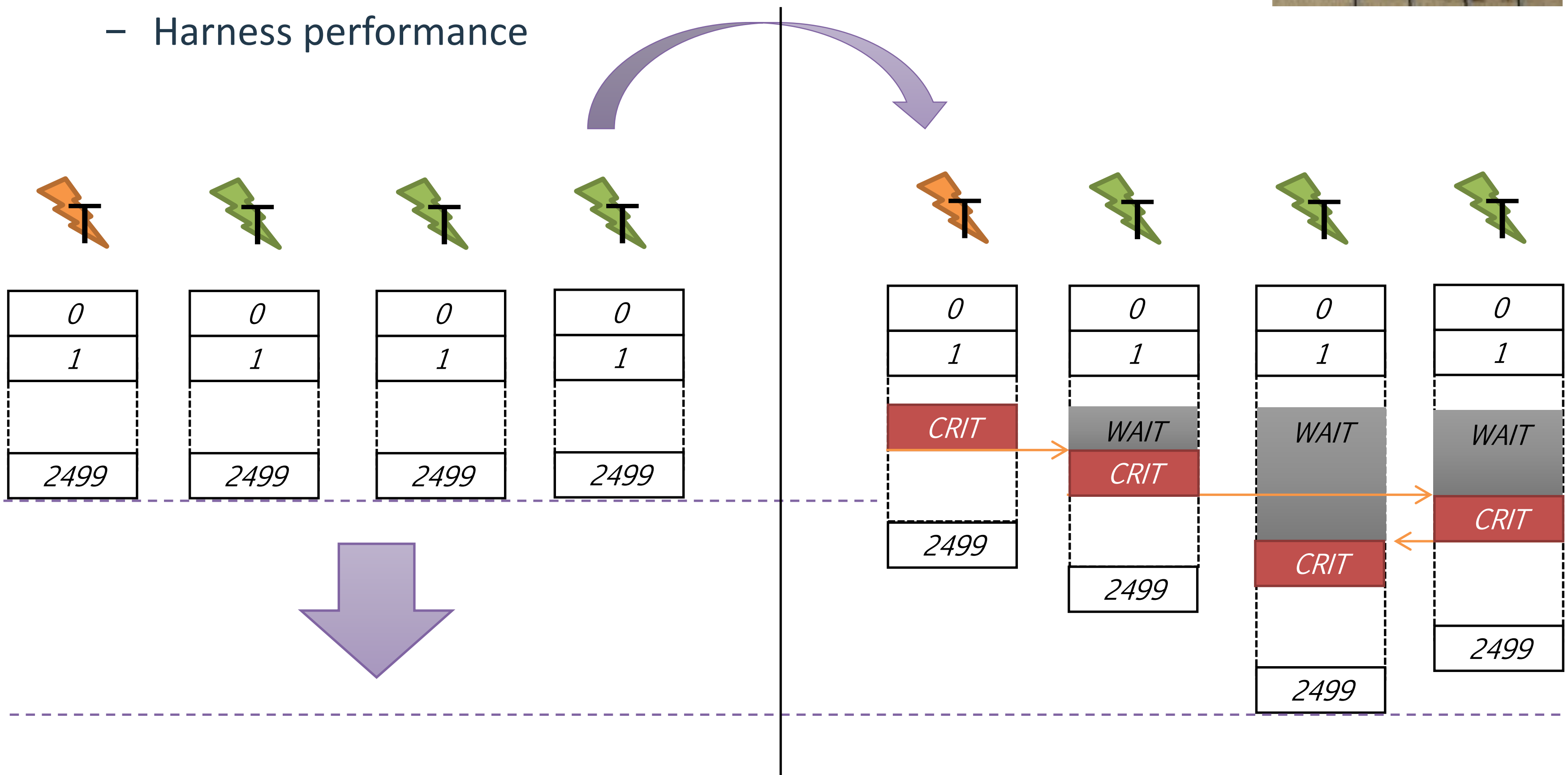
› …to this

```
/* Declare lock var */
omp_lock_t lock;
/* Init the lock */
omp_init_lock(&lock);


/* If another thread set the lock,
   I will wait */
omp_set_lock(&lock);

/* I can do my work being sure that no-
   one else is here */

/* unset the lock so that other threads
can go */
omp_unset_lock(&lock);


/* Destroy lock */
omp_destroy_lock(&lock);
```

```
/* If another thread is in, I must wait */

#pragma omp critical
{
  /* _Critical Section_
     I can do my work being sure
     that no- one else is here */

}

/* Now, other threads can go */
```

The risk of sequentialization

- Critical sections should be kept small as possible
  - They force code portions sequentialization
  - Harness performance



Even more flexible

- (Good) parallel programmers manage to keep critical sections small
  - Possibly, away from their code!
- Most of the operations in a critical section are always the same!
  - "Are you really sure you can't do this using `reduction` semantics?"
  - Modify a shared variable
  - Enqueue/dequeue in a list, stack..
- For single (C/C++) instruction we can definitely do better

The `atomic` construct

```
#pragma omp atomic [seq_cst] new-line
expression-stmt
```

- The atomic construct ensures that a specific storage location is accessed atomically
  - We will see only its simplest form
  - Applies to a single instruction, not to a structured block..
- Binding set: all threads <u>everywhere</u> (also in other teams/parregs)
- The `seq_cst` clause forces the atomically performed operation to include an implicit `flush` operation without a list
  - Enforces memory consistency
  - Does not avoid data races!!

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://algo.ing.unimo.it/people/andrea/Didattica/HPC/index.html
- 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