

Critical sections 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



Exercise

Let's
code!

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



Coherency problem

- › When multiple workers concurrently access the same shared data
 - Data races
 - Read stale data
 - ...

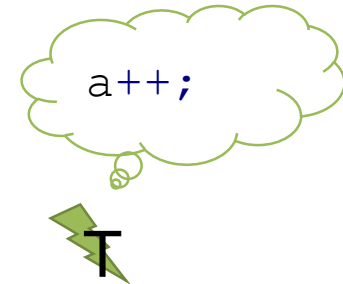
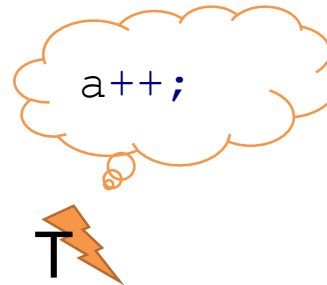
- › (Luckily, does not happen with read-only data)



Data races

- › The "traditional example"
- › a++ is **not** a single instruction!!!!

```
int a = 11;
```



```
LD R0, x TA    x = 0
INC R0 TA      x = 1
ST x, R0 TA    x = 1
LD R0, x TB    x = 1
INC R0 TB      x = 1
ST x, R0 TB    x = 2
```

```
LD R0, x TA    x = 0
LD R0, x TB    x = 0
INC R0 TB      x = 0
ST x, R0 TB    x = 1
INC R0 TA      x = 1
ST x, R0 TA    x = 1
```



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 blocking semantic



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)) /* ... */;`





The `omp_lock_t` type

`omp.h`

```
/* (1) Our implementation @UniBo (few years ago) */
typedef unsigned long omp_lock_t;

/* (2) ROSE compiler */
typedef void * omp_lock_t;

/* (3) GCC-OpenMP (aka Libgomp) */
typedef struct {
    unsigned char _x[@OMP_LOCK_SIZE@]
        __attribute__((__aligned__(@OMP_LOCK_ALIGN@)));
} omp_lock_t;
```

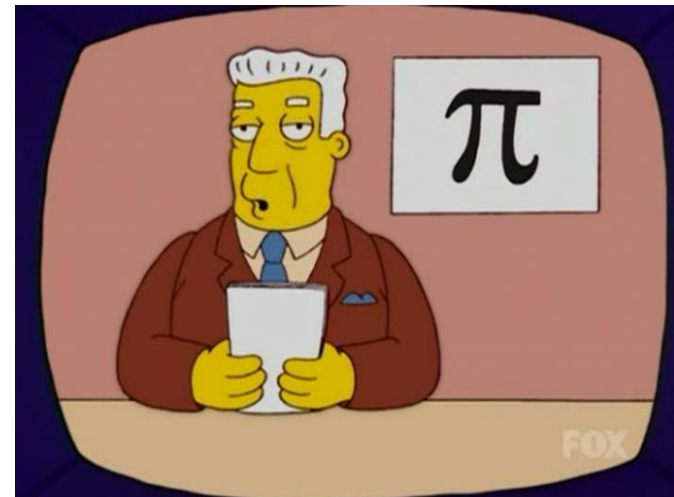
- › Implementation-defined, it represents a lock type
 - Different implementations, different optimizations
- › C routines for OMP lock accept a pointer to an `omp_lock_t` type
 - (at least)



Exercise

Let's
code!

- › Modify the "PI Montecarlo" exercise
 - Replace the variable in the `reduction` clause with a `shared` variable
 - Protect it using an OpenMP 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 expression 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...

```
/* 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);
```

› ...to this

```
/* 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 */

}

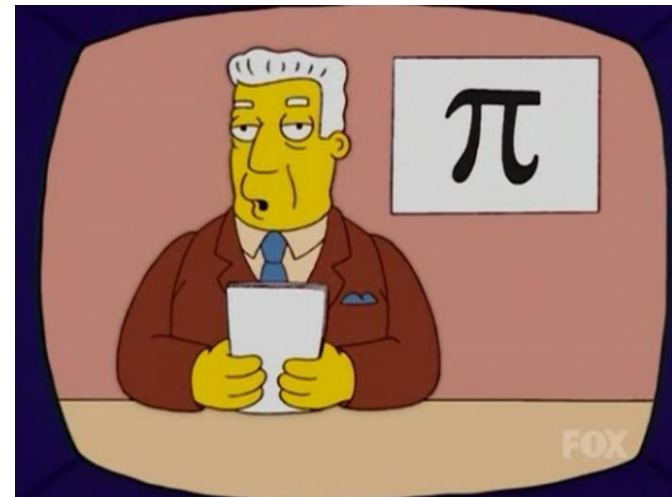
/* Now, other threads can go */
```




Exercise

Let's
code!

- › Modify the "PI Montecarlo" exercise
 - Using critical section instead of locks

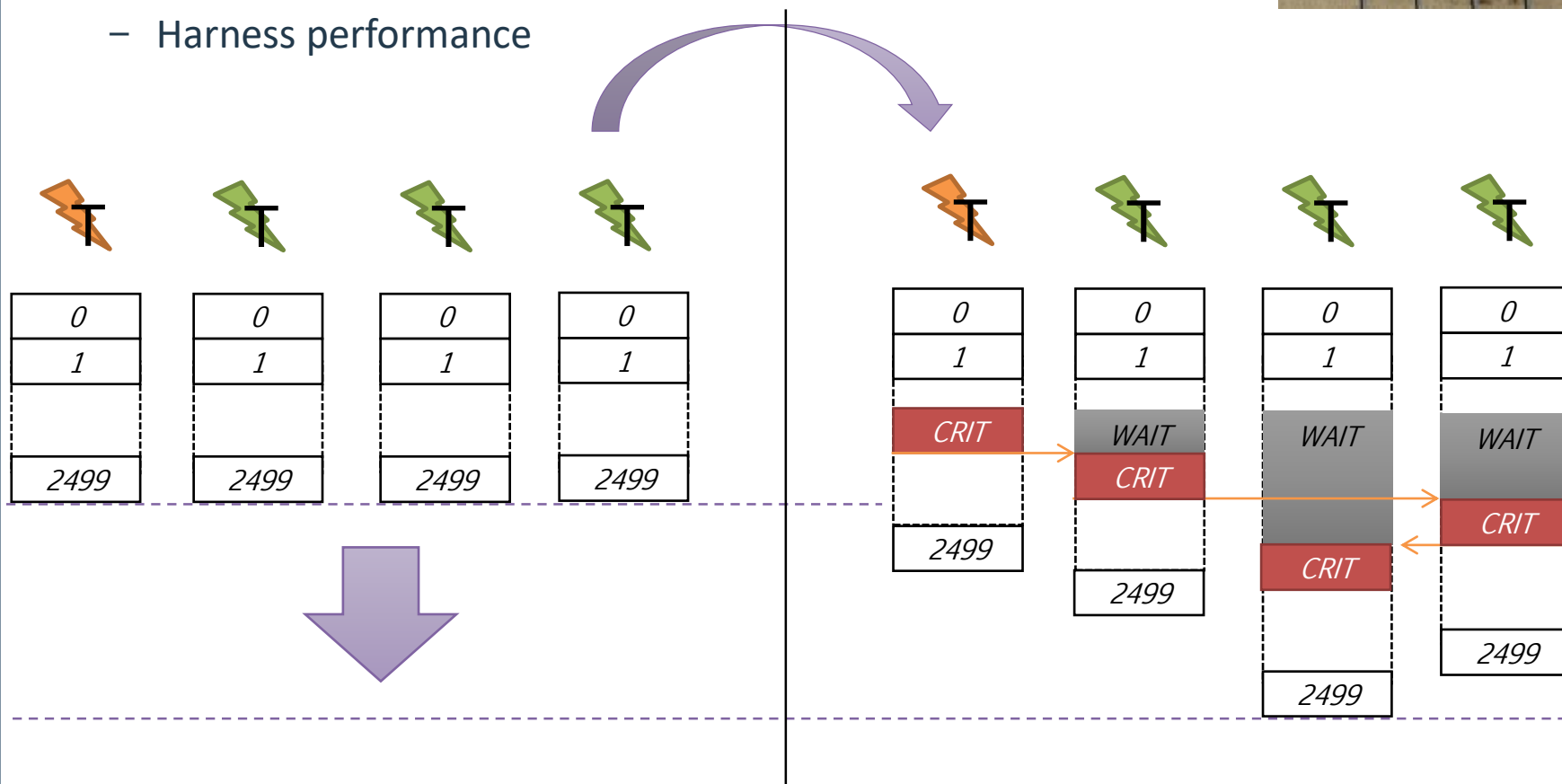




The risk of sequentialization



- › 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 everywhere (also in other teams/parregs)

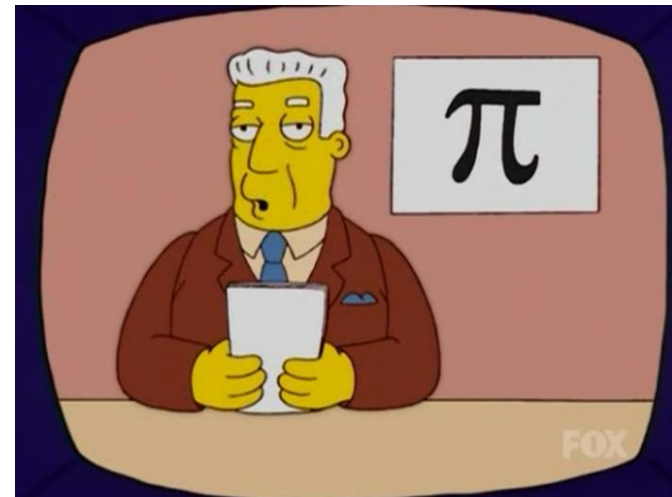
- › The `seq_cst` clause forces the atomically performed operation to include an implicit `flush` operation without a list
 - Enforces memory consistency
 - Does not avoid data races!!



Exercise

Let's
code!

- › Modify the "PI Montecarlo" exercise
 - Implementing the critical section with the `atomic` construct
 - (If possible)





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>