Synchronization Constructs

Critical, Atomic, and Locks

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Synchronization – Motivation

- Different threads need to coordinate with each other in a parallel program
- They communicate by writing and reading to the same shared variables
  - But they may step on each other's toes
  - E.g., simultaneously trying to write to the same variable
- Also, sometimes we need to signal from one thread to the other
  - Signaling, for example, that some value is ready for the other thread to use

Types of Synchronization

- Concurrent access to shared data may result in data inconsistency – mechanism required to maintain data consistency: mutual exclusion
- Sometimes code sections executed by different threads need to be sequenced in some particular order: event synchronization
  - From one thread to the others
  - Wait for all threads to complete some section of code before continuing on
    - This is called a barrier

Mutual Exclusion

- Mechanisms for ensuring the consistency of data that is accessed concurrently by several threads
  - Critical directive
  - Atomic directive
  - Library lock routines

Critical Section: Syntax

```
#pragma omp critical [name]
        structured_block
```

Critical section

- No other thread is allowed to execute any code inside any critical section in the program if one thread is inside it
- Other threads, if they encounter this directive, are made to wait, and only one of them will proceed into the critical section once the thread inside leaves
- The critical sections may be in many places in the code
- If you add a name, then the restriction applies only to those sections that share the name

Global vs. Named Critical Sections

- Access to unnamed critical sections is synchronized with accesses to *all* critical sections in the program: *global lock*
- To change the global lock behavior use the optional *name* parameter – access to a named critical section is synchronized only with other accesses to critical sections with the same name

```
…
                …
#pragma omp critical
{
  S1
}
…

#pragma omp critical
{ S2}

                …

#pragma omp critical
{ S3}
```

Only one thread can be inside any of S1, S2, or S3

```
…
                …
#pragma omp critical sec1
{
  S1
}
…

#pragma omp critical sec2
{ S2}

                …

#pragma omp critical sec2
{ S3}
```

At most one thread can be inside any of S2 or S3, and separately, another thread can be inside S1

Critical Section Example

```
cur_max = MINUS_INFINITY;
#pragma omp parallel for
for(i=0; i<n; i++){
      …

  if (a[i] > cur_max)
       cur_max = a[i];
        …
}
```

Critical Section Example

```
cur_max = MINUS_INFINITY;
#pragma omp parallel for
for(i=0; i<n; i++){
              …
#pragma omp critical
  {
    if (a[i] > cur_max)
      cur_max = a[i];
  }
        …
}
```

Critical Section Example

```
cur_max = MINUS_INFINITY; maxIndex = -1;
#pragma omp parallel for
for(i=0; i<n; i++){
             …
#pragma omp critical
  {
    if (a[i] > cur_max)
      { cur_max = a[i]; maxIndex = i}
  }
          …
}
```

Synchronization Constructs

Critical, Atomic, and Locks

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Atomic Directive

- Critical sections are expensive (we will see later why)
  - But processors support fast single-variable updates as if they are in critical
  - Atomic directive provides access to those hardware capabilities when available
- The body of an atomic directive is a single assignment statement
- There are restrictions on x, expr, operator, and binop – these restrictions ensure that the assignment statement can be translated into an atomic sequence of machine instructions to read, modify, and write the memory location for x
- Syntax:

```
#pragma omp atomic
<Restricted Statement form>
```

Restricted statement form can be one of :

```
x++; ++x; x--; --x;
x binop = expr
```

binop: binary operator, such as +, -

Variants of Atomic Directive

- Many other variants of atomic are supported
- They typically correspond to what the hardware can support as atomic operations
  - I.e., an atomic operation completes before another atomic operation can start on the same variable (no interleaving)
- We will cover only one variant defined by the `capture` clause

Atomic Directive with the `capture` Clause

- We want to atomically change a variable and "capture" its old (or new) value into another variable
- Syntax: two forms are allowed

```
#pragma omp atomic capture
<Restricted expression statement>
```

<Restricted expression statement > can be one of :

```
v= x++;
v = ++x;
v = x--;
v = --x;
v = x binop = expr ; etc…
```

```
#pragma omp atomic capture
<Restricted Structured Block>
```

Restricted Structured block can be:

```
{ v = x; x binop = expr}
{x binop = expr; v =x}
etc…
```

For additional forms see OpenMP 4.5 Standard
For a particular compiler see:
https://software.intel.com/en-us/node/524509

Atomic and Critical

```
#pragma omp parallel private(B)
{
  y =  compute(…);
#pragma omp atomic
  x = x + y;
}
```

```
#pragma omp parallel private(B)
{
  y =  compute(…);
#pragma omp critical
  {
     x = x + y;
  }
}
```

These two formulations are equivalent
But the one with atomic is more efficient

Library Lock routines

- Routines to:
  - In the following, `plock` is a pointer to a variable of type `omp_lock_t`
  - Create a lock
    - `omp_init_lock(plock)`
  - Acquire a lock, waiting until it becomes available if necessary
    - `omp_set_lock(plock)`
  - Release a lock, resuming a waiting thread, if one exists
    - `omp_unset_lock(plock)`
  - Try and acquire a lock but return instead of waiting if not available
    - `omp_test_lock(plock)` (returns true if lock acquired)
  - Destroy a lock
    - `omp_destroy_lock(plock)`

Library Lock Routines

- Locks are the most flexible of the mutual exclusion primitives because there are no restrictions on where they can be placed
- The previous routines don't support nested acquires – deadlock if tried!! – a separate set of routines exist to allow nesting
  - I.e. With nesting, the same thread may acquire the lock multiple times,
    - Each time a count is incremented, and unlock decrements the count
    - Only when the count reaches back to 0, can other threads lock the same variable

Mutual Exclusion Features

- Apply to critical, atomic as well as library routines:
  - NO fairness guarantee
  - Guarantee of progress
  - Careful when using multiple locks – lots of chances for deadlock
- Deadlocks: when a bunch of threads are waiting for each other in a circular fashion
  - Waits arise from locks held by others
  - Thread 1 locks A, requests B, while Thread 2 locks B, requests A
  - More generally: a circular wait: thread 1 waiting for thread 2, which waits for thread 3, which waits for thread 1