Lecture 19: OpenMP and General Synchronization

William Gropp
www.cs.illinois.edu/~wgropp
Not Everything is a “do loop”

- Not all loops are over a simple integer range
  - Lists, graphs, queues
- OpenMP provides several general techniques to handle the more general case
  - Tasks
  - Fine grain synchronization with locks
- Note there are other tools with specialized support for more general iterative operations
Simple List Insert

• Add elements into a list, maintaining sorted order
• Simple list structure
• typedef struct _listelm {
  int val;
  struct _listelm *next, *prev;
} listelm;
List Structure

• Head (of type listelm) points at first element of list.
• I.e., head->next always defined, but may be NULL
Serial Code Part 1

// First, find the insert location
ptr = head->next;
prev = head;
while (ptr && ptr->val < ival) {
    prev = ptr;
    ptr = ptr->next;
}

Serial Code Part 2

// Now insert
{
    listelm *newelm =
        (listelm*)malloc(sizeof(listelm));
    newelm->val = ival;
    newelm->next = ptr;
    newelm->prev = prev;
    prev->next = newelm;
    if (ptr)
        ptr->prev = newelm;
}
Inserting n Elements

- It is very hard to parallelize an individual insert element
- But we could parallelize inserting n elements:
  ```
  for (i=0; i<n; i++) {
      // get element value to insert
      ival = ...;
      ptr = head->next;
      prev = head;
      ... insert code
  }
  ```
Parallelizing the Loop

- Why can’t we simply do
  - #pragma omp parallel for
- Think about that and jot down an answer, then continue to the next slide
Race Condition

• Like the MAXLOC example, there is a race condition: if two threads try to insert at the same point, one insert will get lost (best case) or the list pointers will become inconsistent.
  ♦ Make sure that you can draw an example of how this can happen with two threads (do that now).
Two Threads Racing to Insert

• Both threads find the same prev and ptr; they race to insert before:

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>N0 = new element</td>
<td>N1 = new element</td>
</tr>
<tr>
<td>prev-&gt;next = N0</td>
<td>prev-&gt;next = N1</td>
</tr>
<tr>
<td></td>
<td>ptr-&gt;prev = N1</td>
</tr>
<tr>
<td>ptr-&gt;prev = N0</td>
<td></td>
</tr>
</tbody>
</table>
What Can Go Wrong

- Which new element you see depends on which way you go through the list
The Easy (but Wrong) Fix

• We can attempt to fix this by using 
  `#pragma omp critical` for the insert 
  operation (part 2 from the serial 
  code):

  ```c
  #pragma omp critical
  {
    listelm *newelm = malloc(...);
    ...
  }
  ```

• Why is this wrong?
Race to Insert Still Present

- One element is lost
What are the Fixes?

• Critical section the entire list
  ♦ No parallelism, but multiple threads can safely insert
  ♦ May be ok if inserts rare, accesses in a separate phase (and hence don’t require critical)

• Guard only the elements that are being updated
  ♦ So threads accessing/updating other (disjoint) parts of the list can do so concurrently and safely
  ♦ For this, we need locks
OpenMP Locks

• A thread lock is a form of mutual exclusion. A lock in OpenMP is an object (omp_lock_t) that can be held by at most one thread at a time. The four operations are:
OpenMP Locks

- `omp_init_lock(omp_lock_t *)` – initialize a lock
- `omp_set_lock(omp_lock_t*)` – wait until the lock is available, then set it. No other thread can set the lock until it is released
- `omp_unset_lock(omp_lock_t*)` – unset (release) the lock
- `omp_destroy_lock(omp_lock_t*)` – The reverse of `omp_init_lock`
Concurrent List Updates

• Note: Locks are not cheap!
  ♦ This example is only for illustrating the use of locks
  ♦ There are clever (and some even correct) algorithms that minimize or even eliminate the use of locks
  ♦ Use performance estimates to decide whether you must use more sophisticated techniques
    • You’ll need an estimate of lock cost
    • Costs can vary significantly by platform
Concurrent List Updates

- **Idea:** Lock both list elements – the one before and the one after the element to be inserted (for a singly linked list, need only lock the previous element)
- **First version:** Lock each element pair (prev and prev->next) while searching through the list.
Concurrent List Update

ptr = head->next;
prev = head;
/* Lock the elements that we are considering */
omp_set_lock(&prev->lock);
while (ptr) {
    omp_set_lock(&ptr->lock);
    if (ptr->val >= ival) break;
    omp_unset_lock(&prev->lock);
    prev = ptr;
    ptr = ptr->next;
}
/* We're guaranteed to hold the locks on the elements that we need */
Insert the Element

```c
listelm *newelm = (listelm*)malloc(sizeof(listelm));
newelm->val = ival;
newelm->next = ptr;
newelm->prev = prev;
newelm->prev->next = newelm;
omp_unset_lock(&prev->lock);
if (ptr) {
    ptr->prev = newelm;
    omp_unset_lock(&ptr->lock);
}
```
Speculation

- You can sometimes reduce the cost of an algorithm by *speculation*:
  - In this case, find a *candidate* location, then acquire locks and check that the location is still correct
    - If not, simply use the original algorithm to move to the correct location

- Performance model
  - Depends on the number of locks saved and cost of “failed” speculation
Speculation Step

```c
ptr = head->next;
prev = head;
/* Find a candidate location */
while (ptr && ptr->val < ival) {
    prev = ptr;
    ptr = ptr->next;
}
```
Lock and Check Location

/* Lock the elements that MAY be correct */
omp_set_lock(&prev->lock);
if (ptr) omp_set_lock(&ptr->lock);
/* Confirm that these are adjacent */
if (prev->next != ptr) { // Speculation failed
    if (ptr) omp_unset_lock(&ptr->lock);
    ptr = prev->next;
    while (ptr) {
        omp_set_lock(&ptr->lock);
        if (ptr->val >= ival) break;
        omp_unset_lock(&prev->lock);
        prev = ptr;
        ptr = ptr->next;
    }
} // same insert and unset lock code
Task Parallelism in OpenMP

- OpenMP provides ways to create run statements in separate, dynamically allocated tasks.
- `#pragma omp task` statement runs statement in a separate thread.
  - OpenMP manages the number of threads created, handles joining them back together.
Processing a Linked List

• “process” is a routine that computes on data connected with a linked list element

• `#pragma omp parallel`
  
  `{  
    #pragma omp single 
    `{  
      for(node* p = head; p; p = p->next) {  
        #pragma omp task
        process(p); // p is firstprivate by default
      }
    }  
  }  
`
Processing a Linked List

- "process" is a routine that computes on data connected with a linked list element
- `#pragma omp parallel`
  ```
  `{ 
    #pragma omp single
    `{ 
      for(node* p = head; p; p = p->next) {
        #pragma omp task
        process(p); // p is firstprivate by default
      }
    }
  }
  ```
Processing a Linked List

• “process” is a routine that computes on data connected with a linked list element

• `#pragma omp parallel`
  {
    `#pragma omp single`
    {
      for(node* p = head; p; p = p->next) {
        `#pragma omp task`
        process(p); // p is firstprivate by default
      }
    }
  }

```c
```
Some Last Comments

- Shared memory programming, even with good language support, is hard to both
  - Be correct
  - Perform well

- Two major questions are
  - In what order are statements executes
  - In what order do other threads see changes to memory performed by other threads?
Complications

• Consistency
  ♦ When does one thread see the results of an update to memory made by another thread?

• Sequential consistency
  ♦ Execution is as if the execution is some interleaving of the statements (not the hardware instructions)
  ♦ Code then executes “the way it looks”

• Sequential consistency is hard to make fast
  ♦ Other consistency models trade simplicity for performance
  ♦ Release consistency requires separate acquire and release actions on an object
More Complications

- Writes may be completed in an order that is different than the were issued. Consider this code:

  Thread 0
  A=1;
  B=2;
  A=0;

  Thread 1
  B=3;
  While (A);
  Printf( "%d\n", B );

What value is printed?
Does it matter if A and B are declared volatile?

If sequential consistency is provided, then the value printed is known.
For Discussion

• What problems do you have that might need fine grain synchronization?

• The best solution to synchronization performance problems is often to avoid the problem. How might the large number of locks be avoided in the list insert example?