Lecture 12 - Advanced OpenMP

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

January 30, 2012

Last Lecture

- Main concepts
  - **parallel**
  - **for** (**ordered**)
  - **sections**
  - **single**
  - **master**
- Synchronization
  - **barrier**
  - **critical**
  - **atomic**
- Data sharing: **private**, **shared**, **threadprivate**

Memory Model

OpenMP uses a **relaxed-consistency, shared-memory** model

- All threads have a single place to store/load variables called *memory* (may not actually represent RAM)
- Each thread can have it's own *temporary* view of memory
- A thread's *temporary* view of memory is not required to be consistent with memory

Example Preventing Simultaneous Execution

```
                    a = b = 0
/* thread 1 */                      /* thread 2 */

atomic(b = 1) // [1]                atomic(a = 1) // [3]
atomic(tmp = a) // [2]              atomic(tmp = b) // [4]
if (tmp == 0) then                  if (tmp == 0) then
    // protected section                // protected section
end if                              end if
```

- Does this code actually prevent simultaneous execution?

Example Preventing Simultaneous Execution

```
                          a = b = 0
/* thread 1 */                                 /* thread 2 */

atomic(b = 1) // [1]                           atomic(a = 1) // [3]
atomic(tmp = a) // [2]                         atomic(tmp = b) // [4]
if (tmp == 0) then                             if (tmp == 0) then
    // protected section                           // protected section
end if                                         end if
```

Order				t1 tmp	t2 tmp
1	2	3	4	0	1
1	3	2	4	1	1
1	3	4	2	1	1
3	4	1	2	1	0
3	1	2	4	1	1
3	1	4	2	1	1

- Well, yes it does (at least intuitively)

Example Preventing Simultaneous Execution

```
                    a = b = 0
/* thread 1 */                          /* thread 2 */

atomic(b = 1) // [1]                    atomic(a = 1) // [3]
atomic(tmp = a) // [2]                  atomic(tmp = b) // [4]
if (tmp == 0) then                      if (tmp == 0) then
    // protected section                    // protected section
end if                                  end if
```

- With OpenMP's memory model however, this is incorrect
- The update from one thread may not be seen by the other

Flush

`#pragma omp` **flush** *[(list)]*

- Makes the thread's temporary view of memory consistent with main memory
- Enforces an order on the memory operations of the variables
- The variables in the list are called the *flush-set*, if none are given the compiler will determine them for you

Flush, In Other Words

Enforcing an order on the memory operations means:

- All read/write operations that are in the *flush-set* and happen before to the **flush** complete before the flush executes
- All read/write operations that are in the *flush-set* and happen after to the **flush** complete after the flush executes
- Flushes with overlapping *flush-sets* can not be reordered

Flush Correctness

In order to see a consistent value for a variable between two threads, this order must happen:

1. The value is written to the variable by the first thread
2. The variable is flushed by the first thread
3. The variable is flushed by the second thread
4. The value is read from the variable by the second thread

Same Example, with Flush

```
                              a = b = 0
/* thread 1 */                                 /* thread 2 */

atomic(b = 1)                                  atomic(a = 1)
flush(b)                                       flush(a)
flush(a)                                       flush(b)
atomic(tmp = a)                                atomic(tmp = b)
if (tmp == 0) then                             if (tmp == 0) then
    // protected section                           // protected section
end if                                         end if
```

- Are we guaranteed this will prevent simultaneous access now?

Same Example, with Flush Explanation

No

- The compiler can reorder the `flush(b)` from thread 1 or `flush(a)` from thread 2
- If `flush(b)` is reordered after the protected section, we will not get our intended operation

Same Example, Now Correct

```
                              a = b = 0
/* thread 1 */                                     /* thread 2 */

atomic(b = 1)                                      atomic(a = 1)
flush(a, b)                                        flush(a, b)
atomic(tmp = a)                                    atomic(tmp = b)
if (tmp == 0) then                                 if (tmp == 0) then
    // protected section                               // protected section
end if                                             end if
```

Where Flush Isn't Implied

- At entry to **for**
- At entry to or exit from **master**
- At entry to **sections**
- At entry to **single**
- At exit from **for**, **single** or **sections**, if the **nowait** clause is applied to the directive
    - **nowait** removes implicit flush along with the implicit barrier

This is not true for OpenMP versions before 2.5, so be careful

OpenMP Warning

The program should be able to compile as if the OpenMP directives are not there

```
if (a != 0)
    #pragma omp barrier
if (a != 0)
    #pragma omp taskyield
```

is incorrect, this would be correct:

```
if (a != 0) {
    #pragma omp barrier
}
if (a != 0) {
    #pragma omp taskyield
}
```

Task

`#pragma omp` **task** *[clause [[,] clause]*]*

- Generates a task for a thread in the team to run
- When a thread enters the region it may:
    - immediately execute the task
    - defer its execution (any other thread may be assigned the task)

Allowed Clauses: **if, final, untied, default, mergeable, private, firstprivate, shared**

`if` and `final` Clauses

if *(scalar-logical-expression)*

- When the expression is equal to `false`, an undeferred task is generated
- The generating task tegion is suspended until execution of the undeferred task is completed

final *(scalar-logical-expression)*

- When the expression is equal to `true`, a final task is generated
- All tasks within a final task are *included*
- Included tasks are undeferred and execute sequentially

Example of `if` and `final` Clauses

```
void foo () {
    int i;
    #pragma omp task if(0) // This task is undeferred
    {
        #pragma omp task
        // This task is a regular task
        for (i = 0; i < 3; i++) {
            #pragma omp task
            // This task is a regular task
            bar();
        }
    }
    #pragma omp task final(1) // This task is a regular task
    {
        #pragma omp task // This task is included
        for (i = 0; i < 3; i++) {
            #pragma omp task
            // This task is also included
            bar();
        }
    }
}
```

`untied` and `mergeable` Clauses

untied

- A task suspended task can be resumed by any thread
- Will be ignored if used with **final**

mergeable

- The generated task is an undeferred task or an included task
- The implementation might generate a merged task instead
- Allows the implementation to re-use the environment from another task

mergeable Example

```
#include <stdio.h>
void foo () {
    int x = 2;
    #pragma omp task mergeable
    {
        x++; // x is by default firstprivate
    }
    #pragma omp taskwait
    printf("%d\n",x); // prints 2 or 3
}
```

- This is an incorrect usage of **mergeable** since the output depends on whether or not the task is merged
- Being able to merge tasks when safe, produces more efficient code

Taskyield

`#pragma omp` **taskyield**

- Specifies that the current task can be suspended for another task

Here's an example when it would be good to use:

```
void foo (omp_lock_t * lock, int n) {
    int i;
    for ( i = 0; i < n; i++ )
    #pragma omp task
    {
        something_useful();
        while (!omp_test_lock(lock)) {
            #pragma omp taskyield
        }
        something_critical();
        omp_unset_lock(lock);
    }
}
```

Taskwait

#pragma omp **taskwait**

- Waits for the completeion of the child tasks for the current task

Traversing a Tree

```
struct node {
    struct node *left;
    struct node *right;
};
extern void process(struct node *);

void traverse( struct node *p ) {
    if (p->left)
        #pragma omp task
        // p is firstprivate by default
        traverse(p->left);
    if (p->right)
        #pragma omp task
        // p is firstprivate by default
        traverse(p->right);
    process(p);
}
```

Traversing a Tree: Post-Order

```
struct node {
    struct node *left;
    struct node *right;
};
extern void process(struct node *);

void traverse( struct node *p ) {
    if (p->left)
        #pragma omp task
        // p is firstprivate by default
        traverse(p->left);
    if (p->right)
        #pragma omp task
        // p is firstprivate by default
        traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

- Explicit **taskwait** before processing

Parallel Execution for a Linked List

```
// node struct with data and pointer to next
extern void process(node* p);

void increment_list_items(node* head) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            node * p = head;
            while (p) {
                #pragma omp task
                {
                    process(p);
                }
                p = p->next;
            }
        }
    }
}
```

Lots of Tasks

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            int i;
            for (i=0; i<LARGE_NUMBER; i++)
                #pragma omp task
                // i is firstprivate, item is shared
                process(item[i]);
        }
    }
}
```

- The main loop generates tasks until we're at the limit
- Suspends main thread, finishes some tasks, then restarts the loop in original thread

Lots of Tasks Improved

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            int i;
            #pragma omp task untied
            {
                for (i=0; i<LARGE_NUMBER; i++)
                    #pragma omp task
                    process(item[i]);
            }
        }
    }
}
```

- **untied** let's another thread create tasks

Nesting Restrictions

- You cannot nest **for** regions
- You cannot nest **single** inside a **for**
- You cannot nest **barrier** inside a **critical/single/master/for**

```
void good_nesting(int n)
{
    int i, j;
    #pragma omp parallel default(shared)
    {
        #pragma omp for
        for (i=0; i<n; i++) {
            #pragma omp parallel shared(i, n)
            {
                #pragma omp for
                for (j=0; j < n; j++)
                    work(i, j);
            }
        }
    }
}
```

Performance Considerations

These are things to avoid:

1. Unnecessary flush directive
2. Using critical sections or locks instead of the atomic directive
3. Unnecessary concurrent memory writing protection
   - No need to protect local thread variables
   - No need to protect if only access is in **single** or **master**
4. Too much work in a critical section
5. Too many entries to critical sections

Too Many Entries to Critical Sections

```
#pragma omp parallel for
for (i = 0; i < N; ++i) {
    #pragma omp critical
    {
        if (arr[i] > max) max = arr[i];
    }
}
```

would be better as:

```
#pragma omp parallel for
for ( i = 0 ; i < N; ++i ) {
    #pragma omp flush(max)
    if (arr[i] > max) {
        #pragma omp critical
        {
            if (arr[i] > max) max = arr[i];
        }
    }
}
```

Summary

- Completed our exploration of OpenMP
- How to use **flush** correctly
- How to use OpenMP **tasks** to parallelize other problems