

# Lecture 13 — OpenMP Memory Model

Patrick Lam

p.lam@ece.uwaterloo.ca

Department of Electrical and Computer Engineering  
University of Waterloo

December 7, 2017

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

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

We'll talk more about memory models later.

# 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?

# Possible States for Example

---

```
                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
end if                // protected section
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

Looks like it (at least intuitively).

---

```
                a = b = 0

/* thread 1 */
atomic(b = 1) // [1]
atomic(tmp = a) // [2]
if (tmp == 0) then
    // protected section
end if

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

---

Sorry! With OpenMP's memory model, no guarantees:  
the update from one thread may not be seen by the other.

# Flush: Ensuring Consistent Views of Memory

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

Makes the thread's temporary view of memory consistent with main memory; hence:

- enforces an order on the memory operations of the variables.

The variables in the list are called the *flush-set*. If no variables given, the compiler will determine them for you.

Enforcing an order on the memory operations means:

- All read/write operations on the *flush-set* which happen before the **flush** complete before the flush executes.
- All read/write operations on the *flush-set* which happen after the **flush** complete after the flush executes.
- Flushes with overlapping *flush-sets* can not be reordered.

To show a consistent value for a variable between two threads, OpenMP must run statements in this order:

- 1  $t_1$  writes the value to  $v$ ;
- 2  $t_1$  flushes  $v$ ;
- 3  $t_2$  flushes  $v$  also;
- 4  $t_2$  reads the consistent value from  $v$ .



## Take 2: Same Example, now improved with Flush

---

```
                a = b = 0

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

/* thread 2 */
atomic(a = 1)
flush(a)
flush(b)
atomic(tmp = b)
if (tmp == 0) then
    // protected section
end if
```

---

- OK. Will this now prevent simultaneous access?

No.

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

Probably not, but now you know what it does.

# Same Example, Finally Correct

---

	a = b = 0	
<i>/* thread 1 */</i>		<i>/* thread 2 */</i>
atomic(b = 1)		atomic(a = 1)
flush(a, b)		flush(a, b)
atomic(tmp = a)		atomic(tmp = b)
<b>if</b> (tmp == 0) then		<b>if</b> (tmp == 0) then
<i>// protected section</i>		<i>// protected section</i>
end <b>if</b>		end <b>if</b>

---

# OpenMP Directives 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** with a **nowait**
  - **nowait** removes implicit flush along with the implicit barrier

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

```
#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; or
- defer its execution.  
(any other thread may be assigned the task)

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

**`if`** (*scalar-logical-expression*)

When expression is `false`, generates an undeferred task—the generating task region is suspended until execution of the undeferred task finishes.

**`final`** (*scalar-logical-expression*)

When expression is `true`, generates a final task.

All tasks within a final task are *included*.

Included tasks are undeferred and also execute immediately in the same thread.

# if and final Clauses: Examples

```
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

- A suspended task can be resumed by any thread.
- “untied” is ignored if used with **final**.
- Interacts poorly with thread-private variables and `gettid()`.

## mergeable

- For an undeferred or included task, allows the implementation to generate a merged task instead.
- In a merged task, the implementation may re-use the environment from its generating task (as if there was no task directive).

For more: [docs.oracle.com/cd/E24457\\_01/html/E21996/gljyr.html](https://docs.oracle.com/cd/E24457_01/html/E21996/gljyr.html)

---

```
#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**: the output depends on whether or not the task got merged.

Merging tasks (when safe) produces more efficient code.

## #pragma omp **taskyield**

Specifies that the current task can be suspended in favour of another task.

Here's a good use of **taskyield**.

---

```
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);  
        }  
}
```

---

```
#pragma omp taskwait
```

Waits for the completion of the current task's child tasks.

# OpenMP Example: Tree Traversal

---

```
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);  
}
```

---

# OpenMP Example 2: Post-order Tree Traversal

---

```
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);  
}
```

---

Note: Used an explicit **taskwait** before processing.

# OpenMP Example: Parallel Linked List Processing

---

```
// 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;  
            }  
        }  
    }  
}
```

---

# OpenMP Example: 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, which are all assigned to the executing thread as it becomes available.

When too many tasks generated: suspends main thread, runs some tasks, then resumes the loop in main thread.



# OpenMP Example: Improved version of 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;
            #pragma omp task untied
            {
                for (i=0; i<LARGE_NUMBER; i++) {
                    #pragma omp task
                    process(item[i]);
                }
            }
        }
    }
}
```

---

- **untied** lets another thread take on tasks.

# About 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);
            }
        }
    }
}
```

---

Want it to run faster? Avoid these pitfalls:

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

# Example: Too Many Entries into 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];
        }
    }
}
```

---

Finished exploring OpenMP. Key points:

- How to use **flush** correctly.
- How to use OpenMP **tasks** to parallelize unstructured problems.