# Lecture 14 — OpenMP Tasks, Memory Model

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 20, 2018

The main new feature in OpenMP 3.0 is the notion of tasks.

When the program executes a `#pragma omp task` statement, the code inside the task is split off as a task and scheduled to run sometime in the future.

Tasks are more flexible than parallel sections.

They also have lower overhead.

`#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 the undeferred task finishes.

# `final` Clause

**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.

# Examples of if and final

```
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 we stan... wait...

**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).

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

Merging tasks (when safe) produces more efficient code.

# Yield

`#pragma omp` **taskyield**

This directive 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.

# Task Example 1: Web Server

```
#pragma omp parallel
  /* a single thread manages the connections */
  #pragma omp single nowait
  while (!end) {
    process any signals
    foreach request from the blocked queue {
      if (request dependencies are met) {
        extract from the blocked queue
        /* create a task for the request */
        #pragma omp task untied
          serve_request(request);
      }
    }
    if (new connection) {
      accept_connection();
      /* create a task for the request */
      #pragma omp task untied
        serve_request(new connection);
    }
    select();
  }
```

# Task Example 2: 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);
}
```

To guarantee a post-order traversal, insert an explicit `#pragma omp taskwait` after the two calls to `traverse` and before the call to `process`.

# Task Example 3: 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;
            }
        }
    }
}
```

# Make ALL the Tasks!

Let's see what happens if we spawn lots of tasks in a `single` directive.

```c
#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]);
            }
        }
    }
}
```

# Make ALL the Tasks!

In this case, the main loop generates tasks, which are all assigned to the executing thread as it becomes available (because of `single`).

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

It would be better to `untied` the spawned tasks, enabling them to run on multiple threads.

Surround the for loop with `#pragma omp task untied`.

# More Scoping

Besides the `shared`, `private` and `threadprivate`, OpenMP also supports `firstprivate` and `lastprivate`,
Firstprivate:

```
int x;

void* run(void* arg) {
    int thread_x = x;
    // use thread_x
}
```

Lastprivate:

```
int x;

void* run(void* arg) {
    int thread_x;
    // use thread_x
    if (last_iteration) {
        x = thread_x;
    }
}
```

`copyin` is like firstprivate, but for threadprivate variables.

Pseudocode for `copyin`:

```
int x;
int x[NUM_THREADS];

void* run(void* arg) {
  x[thread_num] = x;
  // use x[thread_num]
}
```

The `copyprivate` clause is only used with `single`.

It copies the specified private variables from the thread to all other threads. It cannot be used with `nowait`.

# Catch Me Outside...

```
int tid, a, b;

#pragma omp threadprivate(a)

int main(int argc, char *argv[])
{
    printf("Parallel #1 Start\n");
    #pragma omp parallel private(b, tid)
    {
        tid = omp_get_thread_num();
        a = tid;
        b = tid;
        printf("T%d: a=%d, b=%d\n", tid, a, b);
    }

    printf("Sequential code\n");
    printf("Parallel #2 Start\n");
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("T%d: a=%d, b=%d\n", tid, a, b);
    }

    return 0;
}
```

Produces as output:

```
% ./a.out
 Parallel #1 Start
T6: a=6, b=6
T1: a=1, b=1
T0: a=0, b=0
T4: a=4, b=4
T2: a=2, b=2
T3: a=3, b=3
T5: a=5, b=5
T7: a=7, b=7
 Sequential code
 Parallel #2 Start
T0: a=0, b=0
T6: a=6, b=0
T1: a=1, b=0
T2: a=2, b=0
T5: a=5, b=0
T7: a=7, b=0
T3: a=3, b=0
T4: a=4, b=0
```

- All threads share a single store called memory.
  (may not actually represent RAM)
- Each thread has 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?

# Not seeing the problem...?

```
                        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 |

Looks like it (at least intuitively).

# The Memory Model Contains Gotchas

```
                    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
```

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

# Flush Ensures Consistent Views of Memory

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

Makes the thread's temporary view of memory consistent with main memory.

It enforces an order on memory operations of variables.

The variables in the list are called the flush-set.

If no variables given, compiler determines them for you.

# Flush: Before is Before, After is After

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.

# Flush Correctness

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 */                      /* 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
```

Will this now prevent simultaneous access?

# No Luck Yet: Flush Fails

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.

# Should you use flush?

Probably not, but now you know what it does.

# Proper Use of Flush

```
                  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
```

# OpenMP Directives Where Flush Is Implied

- `omp barrier`
- at entry to, and exit from, **omp critical**;
- at exit from **omp parallel**;
- at exit from **omp for**;
- at exit from **omp sections**;
- at exit from **omp single**.

# 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.

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: Reducing 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];
        }
    }
}
```

Key points:

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