

## Lecture 14 — OpenMP Tasks

Patrick Lam

2019-10-31

**Tasks: OpenMP's thread-like mechanism.**

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, because parallel sections constrain exactly how many threads are supposed to run, and there is also always a join at the end of the parallel section. On the other hand, the OpenMP runtime can assign any task to any thread that's running. Tasks therefore 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** and **final** Clauses.

**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.

Let's look at some examples of these 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 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.

### Taskyield.

```
#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);
        }
}

```

**Taskwait.**

```
#pragma omp taskwait
```

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

Two examples which show off tasks, from [ACD<sup>+</sup>09], include a web server (with unstructured requests) and a user interface which allows users to start tasks that are to run in parallel.

Here's pseudocode for the Boa webserver main loop from [ACD<sup>+</sup>09].

```

#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();
}

```

The untied qualifier lifts restrictions on the task-to-thread mapping. All it means is that a task that's been started by one thread could be picked up by another and carried forward. This can have bad side effects if there's thread-private data involved. The single directive indicates that the runtime is only to use one thread to execute the next statement; otherwise, it could execute  $N$  copies of the statement, which does belong to a OpenMP parallel construct.

```

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);
}

```

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

**Parallel Linked List Processing.** We can spawn tasks to process linked list entries. It's hard to use two threads to traverse the list, though.

```

// 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;
            }
        }
    }
}

```

**Using Lots of Tasks.** Let's see what happens if we spawn lots of tasks in a single directive.

```

#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]);
            }
        }
    }
}

```

In this case, the main loop (which executes in one thread only, due to `single`) generates tasks and queues them for execution. When too many tasks get generated and are waiting, OpenMP suspends the main thread, runs some tasks, then resumes the loop in the main thread.

Any thread may pick up a task and execute it. Without `untied`, a thread that starts a task has to finish running that task.

**Improved code.** If we untied the spawned tasks, that would enable the tasks to migrate between threads when suspended. Just make sure that there is no threadprivate data that is going to be wrong after a thread migration.

## More Scoping Clauses

Besides the `shared`, `private` and `threadprivate`, OpenMP also supports `firstprivate` and `lastprivate`, which work like this.

Pthreads pseudocode for `firstprivate` clause:

```
int x;

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

Pthread pseudocode for the `lastprivate` clause:

```
int x;

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

In other words, `lastprivate` makes sure that the variable `x` has the same value as if the loop executed sequentially.

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

Pthreads 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`.

**Defaults.** **default(shared)** makes all variables shared; **default(none)** prevents sharing by default (creating compiler errors if you treat a variable as shared.)

**Catch Me Outside...** A related problem with private variables is that sometimes you need access to them outside their parallel region. Here's some contrived code.

```
#include <omp.h>
#include <stdio.h>

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;
}

```

This yields something like the following 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

```

## References

- [ACD<sup>+</sup>09] Eduard Ayguadé, Nawal Copt, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20:404–418, March 2009. URL: <http://portal.acm.org/citation.cfm?id=1512157.1512430>.