OpenMP Tasks

Remember OpenMP Sections?

Sections are independent blocks of code, able to be assigned to separate threads if they are available.

```c
#pragma omp parallel sections
{
#pragma omp section
  Task 1
#pragma omp section
  Task 2
}
```

There is an implied barrier at the end

OpenMP sections are static, that is, they are good if you know, when you are writing the program, how many of them you will need.

It would be nice to have something more Dynamic

Imagine a capability where you can write something to do down on a Post-It® note, accumulate the Post-It notes, then have all of the threads together execute that set of tasks.

You would also like to not have to know, ahead of time, how many of these Post-It notes you will write. That is, you want the total number to be dynamic.

Well, congratulations, you have just invented OpenMP Tasks!

OpenMP Task Example: Something (Supposedly) Simple

```c
omp_set_num_threads( 2 );
#pragma omp parallel default(none)
{
#pragma omp task
  fprintf( stderr, "A\n" );
#pragma omp task
  fprintf( stderr, "B\n" );
}
```

Without this, thread #0 has to do everything

- Writes `fprintf( stderr, "A\n" );` on a sticky note and adds it to the list of tasks
- Writes `fprintf( stderr, "B\n" );` on a sticky note and adds it to the list of tasks
- Adds the next line of code (or block of code) to the list of tasks

If You Run This a Number of Times, You Get This:

(Uh-oh, what Happened?)

1. Why do we not get the same output every time?
2. Why do we get 4 things printed when we only have print statements in 2 tasks?

The first answer is easy. Unless you make some special arrangements, the order of execution of the different tasks is undefined.

The second answer is that we actually asked the two threads to each put two tasks on the sticky notes, for a total of four. How can we get only one thread to do this?

If the `if` clause is used and the argument evaluates to 0, then the task is executed immediately, superseding whatever else that thread is doing.

One of the best uses of this is to process elements of a linked list or a tree.

You can create a task barrier with:

```c
#pragma omp taskwait
```

Tasks are very much like OpenMP Sections, but Sections are static, that is, the number of sections is set when you write the code, whereas Tasks can be created anytime, and in any number, under control of your program’s logic.
The "single" Pragma

```c
omp_set_num_threads(2);
#pragma omp parallel default(none)
{
    #pragma omp single
    {    
        #pragma omp task
        fprintf(stderr, "A\n");
        #pragma omp task
        fprintf(stderr, "B\n");
    }
}
```

When using Tasks, you only want one thread to write the things to do down on the sticky note, but you want all of the threads to be able to execute the sticky notes.

But, if you run this, the order of printing will still be non-deterministic. If you care about order, do this:

```c
omp_set_num_threads(2);
#pragma omp parallel
{
    #pragma omp single default(none)
    {
        #pragma omp task
        fprintf(stderr, "A\n");
        #pragma omp taskwait
        #pragma omp task
        fprintf(stderr, "B\n");
        #pragma omp taskwait
    }
}
```

Causes all tasks to wait until they are completed

A Better OpenMP Task Example: Processing each Element of a Linked List

```c
#pragma omp parallel default(none)
{
    #pragma omp single default(none)
    {
        element *p = listHead;
        while( p != NULL )
        {
            #pragma omp task firstprivate(p)
            Process( p );
            p = p->next;
        }
    }
    #pragma omp taskwait
}
```

Without this, each thread does a full traversal – bad idea!

Without this, thread #0 has to do everything.

Put this here if you want to wait for all tasks to finish being executed before proceeding.

A tree traversal is a way of visiting the nodes of a tree, where each node is visited once and only once. There are several common tree traversal algorithms, including:

- **Preorder traversal**
- **Inorder traversal**
- **Postorder traversal**

Given a tree:

![Tree Diagram]

- We would like to traverse it as quickly as possible.
- We are assuming that we do not need to traverse it in order.
- We just need to visit all nodes.

One more thing – Task Dependencies

Remember from before: unless you make some special arrangements, the order of execution of the different tasks is undefined. Here come some special arrangements.

```c
omp_set_num_threads(3);
#pragma omp parallel
{
    #pragma omp single default(none)
    {
        float a, b, c;
        #pragma omp task depend(OUT: a)
        a = 10.;
        #pragma omp task depend(IN: a, OUT: b)
        b = a + 16.;
        #pragma omp task depend(IN: b)
        c = b + 12.;
    }
    #pragma omp taskwait
}
```

This maintains the proper dependencies, but, because it involves all of the tasks, it essentially serializes the parallelism out of them. Be careful not to go overboard with dependencies!

Tree Traversal Algorithms

- This is common in graph algorithms, such as searching.
- If the tree is binary and is balanced, then the maximum depth of the tree is $\log_2$ (Number of Nodes)

Strategy at a node:
1. follow one descendant node
2. follow the other descendant node
3. process the node you’re at

The order could be rearranged, depending on what you are trying to do.
# Tree Traversal Algorithms

```
#pragma omp parallel
    #pragma omp single
    Traverse( root );

#pragma omp taskwait
```

Without this, thread #0 has to do everything – bad idea!

Without this, each thread does a full traversal – bad idea!

Put this here if you want to wait for all nodes to be traversed before proceeding.

---

## Parallelizing a Binary Tree Traversal with Tasks

```
void Traverse( Node *n )
{
    if( n->left != NULL )
    {
        #pragma omp task private(n) untied
        Traverse( n->left );
    }

    if( n->right != NULL )
    {
        #pragma omp task private(n) untied
        Traverse( n->right );
    }

    #pragma omp taskwait
    Process( n );
}
```

Put this here if you want to wait for both branches to be taken before processing the parent.

---

## Benchmarking a Binary Task-driven Tree Traversal

```
#define NUM 1024*1024

void Process( Node *n )
{
    for( int i = 0; i < NUM; i++ )
    {
        n->value = pow( n->value, 1.01 );
    }
}
```

---

## Parallelizing a Binary Tree Traversal with Tasks: Tied (g++ 10.2)

```
void Traverse( A );
```

```
15 1
14 1
13 2
12 2
11 2
10 2
9 2
8 2
7 2
6 1
5 1
4 1
3 1
2 1
1 1
```

## Parallelizing a Binary Tree Traversal with Tasks: Untied (g++ 10.2)

```
void Traverse( A );
```

```
15 1
14 1
13 2
12 2
11 2
10 2
9 2
8 2
7 2
6 1
5 1
4 1
3 1
2 1
1 1
```
### How Evenly Tasks Get Assigned to Threads

**g++ vs. icpc**

<table>
<thead>
<tr>
<th>Thread</th>
<th>Number of Tasks</th>
<th>g++ 10.2:</th>
<th>icpc 15.0.0:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3071</td>
<td>1</td>
<td>1902</td>
</tr>
<tr>
<td>1</td>
<td>2035</td>
<td>1</td>
<td>2068</td>
</tr>
<tr>
<td>2</td>
<td>2048</td>
<td>3</td>
<td>2089</td>
</tr>
<tr>
<td>3</td>
<td>2048</td>
<td>3</td>
<td>2089</td>
</tr>
</tbody>
</table>

**g++ 4.9 vs. g++ 10.2**

<table>
<thead>
<tr>
<th>Thread</th>
<th>Number of Tasks</th>
<th>g++ 4.9:</th>
<th>g++ 10.2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2561</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2813</td>
<td>2</td>
<td>3071</td>
</tr>
<tr>
<td>3</td>
<td>2815</td>
<td>3</td>
<td>2048</td>
</tr>
</tbody>
</table>

**g++ 10.2 -- Tied**

<table>
<thead>
<tr>
<th>Thread</th>
<th>Number of Tasks</th>
<th>g++ 10.2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3071</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2048</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3071</td>
<td>3</td>
</tr>
</tbody>
</table>

**g++ 10.2 -- Untied**

<table>
<thead>
<tr>
<th>Thread</th>
<th>Number of Tasks</th>
<th>g++ 10.2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>41</td>
<td>47</td>
</tr>
<tr>
<td>2</td>
<td>47</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>47</td>
<td>47</td>
</tr>
</tbody>
</table>

### Performance vs. Number of Levels

<table>
<thead>
<tr>
<th># Levels</th>
<th>Nodes Processed per Second</th>
<th># Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3071</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>2048</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>2035</td>
<td>12</td>
</tr>
</tbody>
</table>

8-thread Speed-up ≈ 6.7

F_p ≈ 97%

Max Speed-up ≈ 33x
Parallelizing a Tree Traversal with Tasks:
Summary

- Tasks get spread among the current "thread team".
- Tasks can execute immediately or can be deferred. They are executed at "some time".
- Tasks can be moved between threads, that is, if one thread has a backlog of tasks to do, an idle thread can come steal some workload.
- Tasks are more dynamic than sections. The task paradigm would still work if there was a variable number of children at each node.