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 Tasks

- An OpenMP task is a single line of code or a structured block which is immediately "written down" in a list of tasks.
- The new task can be executed immediately, or it can be deferred.
- 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.
- There has to be an existing parallel thread team for this to work. Otherwise one thread ends up doing all tasks and you don’t get any contribution to parallelism.
- 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.
OpenMP Task Example: Something (Supposedly) Simple

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

Without this, thread #0 has to do everything

If You Run This a Number of Times, You Get This: (What Happened?)

Run#

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

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?

Not so simple, huh?

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 each of the two threads to put two tasks on the sticky notes, for a total of four. How can we get only one thread to do this?

The "single" Pragma

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

Causes all tasks to wait until they are completed

But, if you run this, the order of printing will still be non-deterministic. To solve that problem, do this:

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

One more thing – Task Dependencies

```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!

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

Tree Traversal Algorithms

Given a tree:

- 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 \( \log_2(\# \text{ of Nodes}) \).

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

This order could be rearranged, depending on what you are trying to do.
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 nodes to be traversed before proceeding

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 );
}

Parallelizing a Binary Tree Traversal with Tasks:

Tied (gcc 8.2)

Try the untied variant instead of the tied variant to see the difference.
Parallelizing a Binary Tree Traversal with Tasks: Untied
(gcc 8.2)

Thread # Number of Tasks
0 1
1 32
2 47
3 47

Thread # Number of Tasks
0 2561
1 2813
2 2815

6 Levels – g++ 8.2:

Thread # Number of Tasks
0 1
1 31
2 46
3 18

Thread # Number of Tasks
0 29
1 31
2 41
3 26

6 Levels – icpc 15.0.0:

Thread # Number of Tasks
0 1999
1 2068
2 2035
3 2089

Benchmarking a Binary Task-driven Tree Traversal

void Process( Node *n )
{  
  for( int i = 0; i < 1024; i++ )
  {   
    n->value = pow( n->value, 1.1 );
  }
- 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.

```c
void Traverse( Node *n )
{
    for( int i = 0; i < n->numChildren; i++ )
    {
        if( n->child[i] != NULL )
        {
#pragma omp task
            Traverse( n->child[i] );
#pragma omp taskwait
        }
    }
    Process( n );
}
```
Performance vs. Number of Levels

- **8-thread Speed-up**: 6.7
- **F_p**: ??%
- **Max Speed-up**: ??

\[ F_p = \frac{n}{(n-1)} \left(1 - \frac{1}{\text{Speedup}}\right) = 97\% \]

\[ \text{max Speedup} = \frac{1}{1 - F_p} = 33x \]