Tree Traversal using OpenMP Tasks

Given a tree:

- 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.

Given a tree:

- 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.

• 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

A strategy at a node:
1. follow one descendent node
2. follow the other descendent node
3. process the node you're at

```
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 );
}
```

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

```
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 $\log_2(\# \text{ of Nodes})$

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 re-arranged, depending on what you are trying to do

```
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 );
}
```

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

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 );
}
```

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

Parallelizing a Binary Tree Traversal with Tasks
Parallelizing a Binary Tree Traversal with Tasks: Tied (gcc 8.2)

Traverse( A );

How Evenly Tasks Get Assigned to Threads

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 );
    }
}

Performance vs. Number of Threads
Parallelizing a Tree Traversal with Tasks

- 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.

Parallelizing an N-Tree Traversal with Tasks

```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
- \( F_p = 77\% \)
- Max Speed-up = ??

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

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