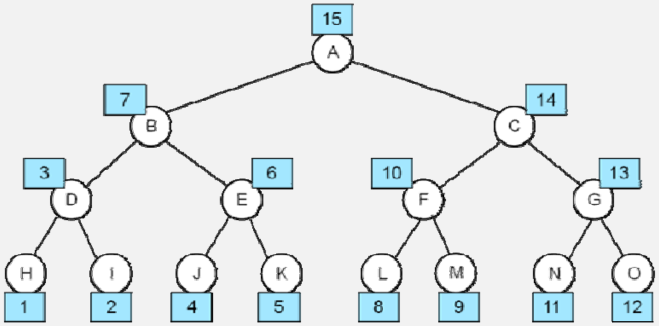
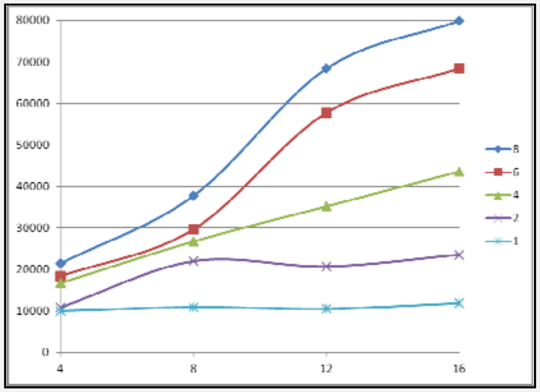
Tree Traversal using OpenMP Tasks

Mike Bailey

mjb@cs.oregonstate.edu

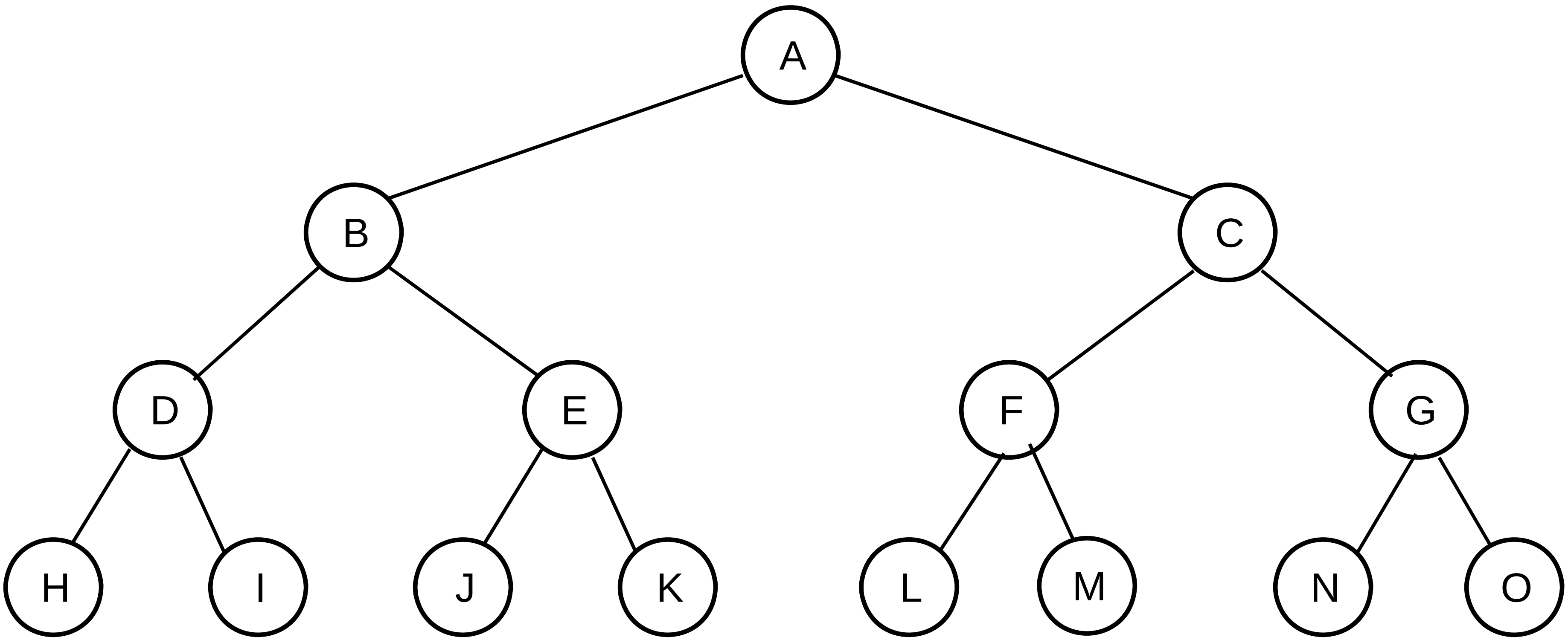
Oregon State University





Tree Traversal Algorithms

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.

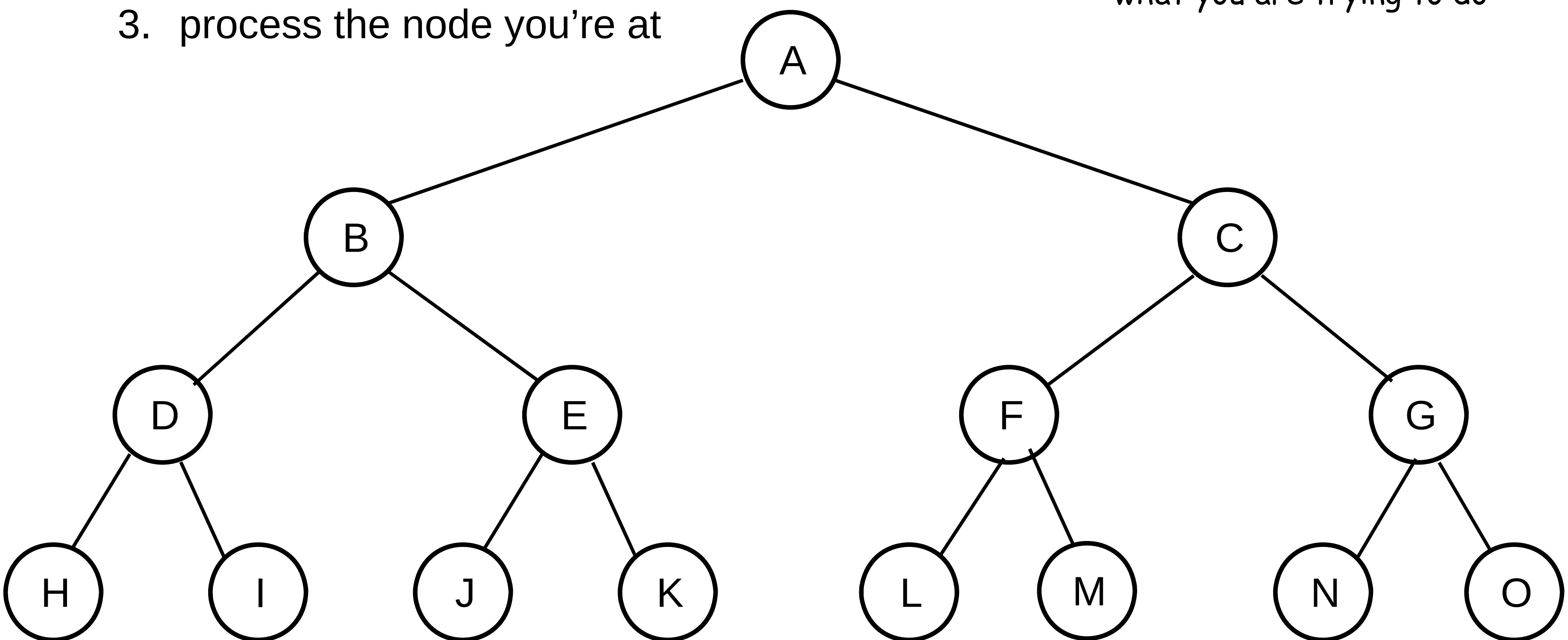
Tree Traversal Algorithms

• Common in graph algorithms, such as searching.

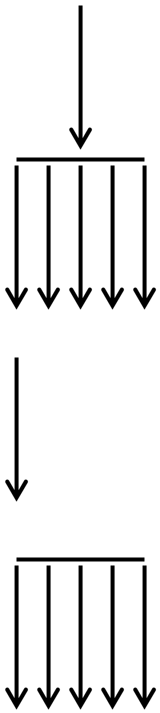
• If the tree is binary and is balanced, then the maximum depth of the tree is $\log_2$(# of Nodes)

• 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



Tree Traversal Algorithms



Without this, thread #0 has to do everything

```
#pragma omp parallel

        #pragma omp single

        Traverse( root );



#pragma omp taskwait
```

Without this, each thread does a full traversal

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

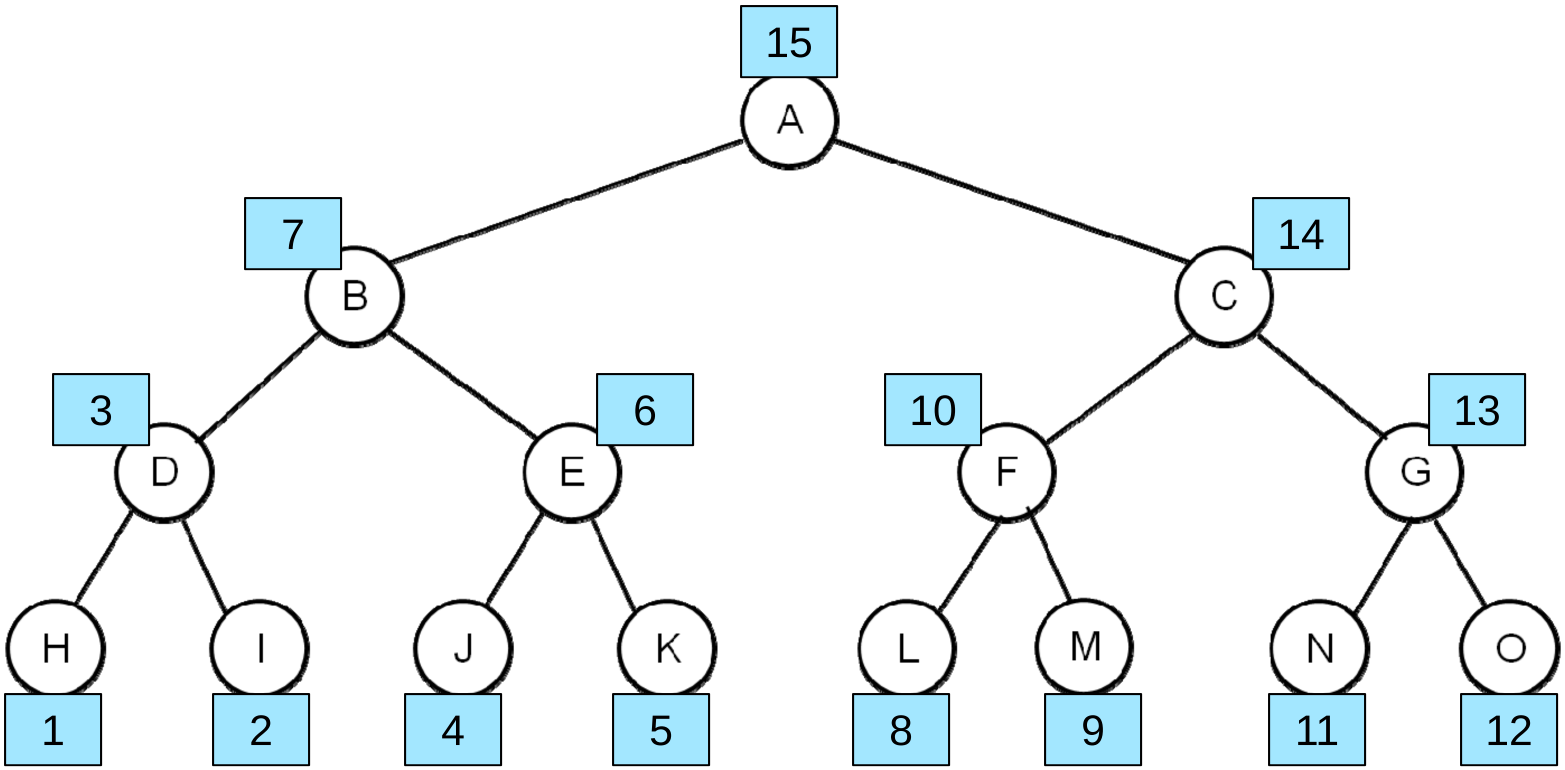
Parallelizing a Binary Tree Traversal with Tasks

```
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

Parallelizing a Binary Tree Traversal with Tasks

Traverse(A);

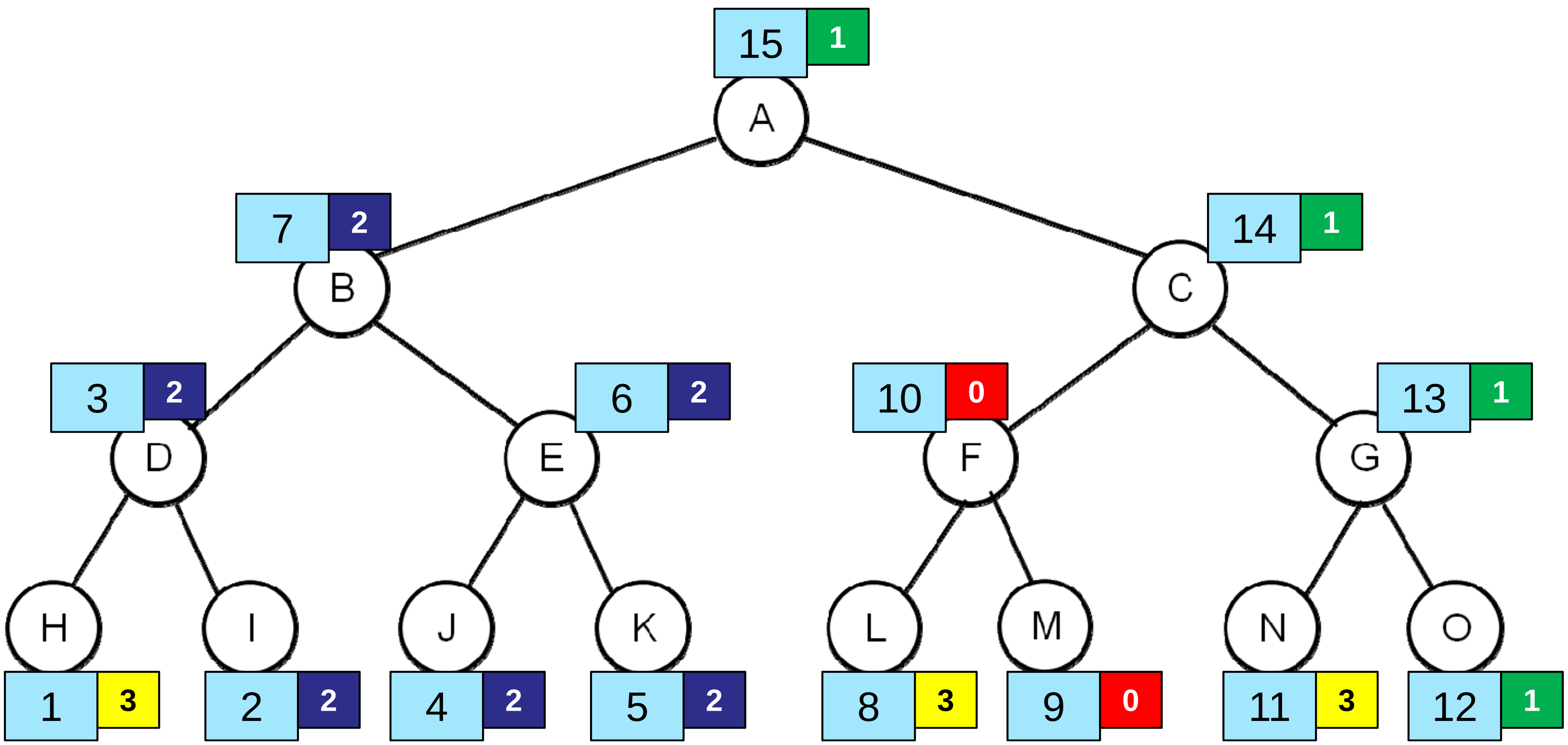


Parallelizing a Binary Tree Traversal with Tasks: *Tied*

Threads:

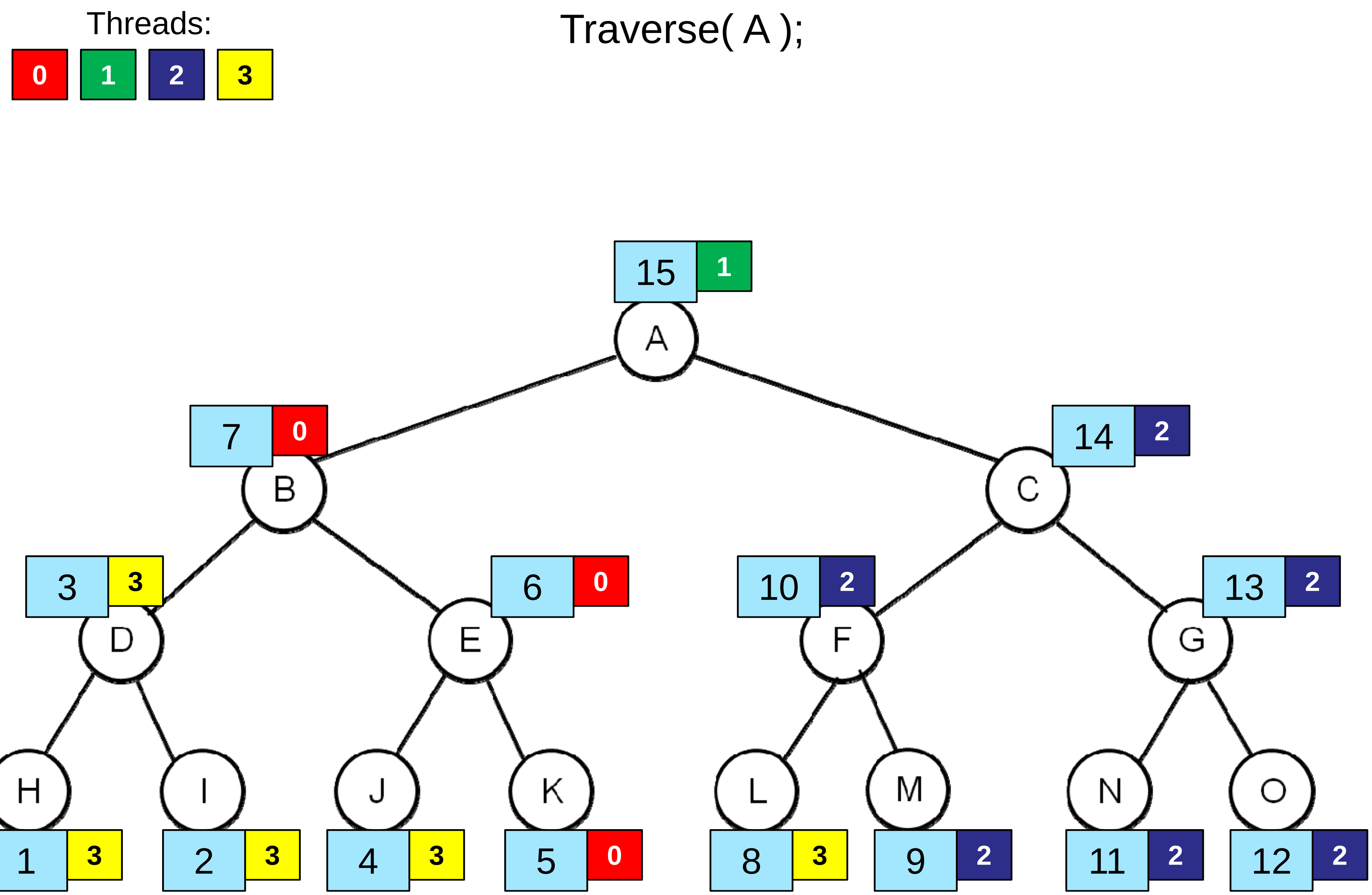
0 1 2 3

Traverse(A);



Parallelizing a Binary Tree Traversal with Tasks: *Untied*

Traverse(A);



How Evenly Tasks Get Assigned to Threads

6 Levels – g++ 4.9:

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

6 Levels – icpc 15.0.0:

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

12 Levels – g++ 4.9:

Thread #	Number of Tasks
0	2561
1	2
2	2813
3	2815

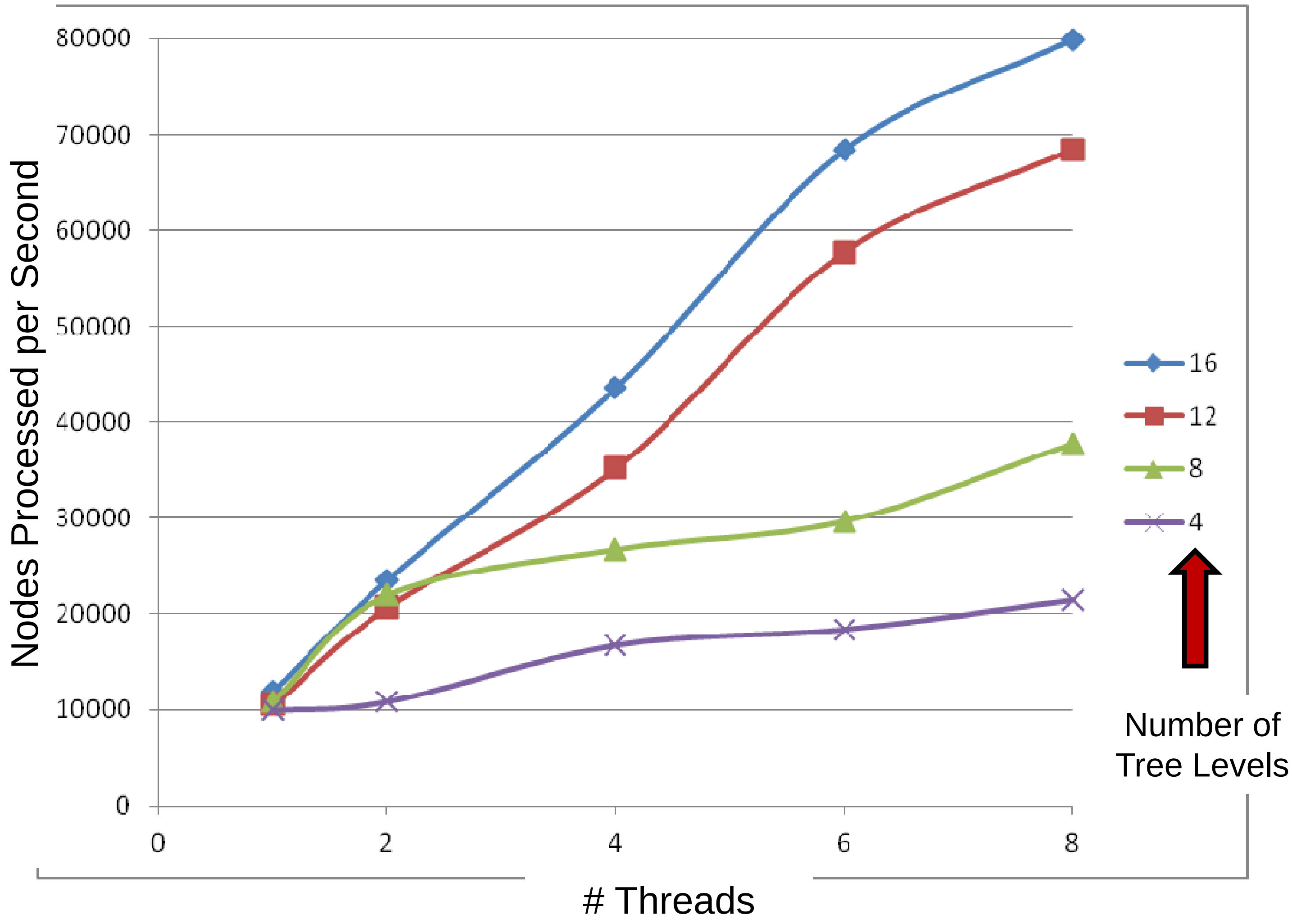
12 Levels – icpc 15.0.0:

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

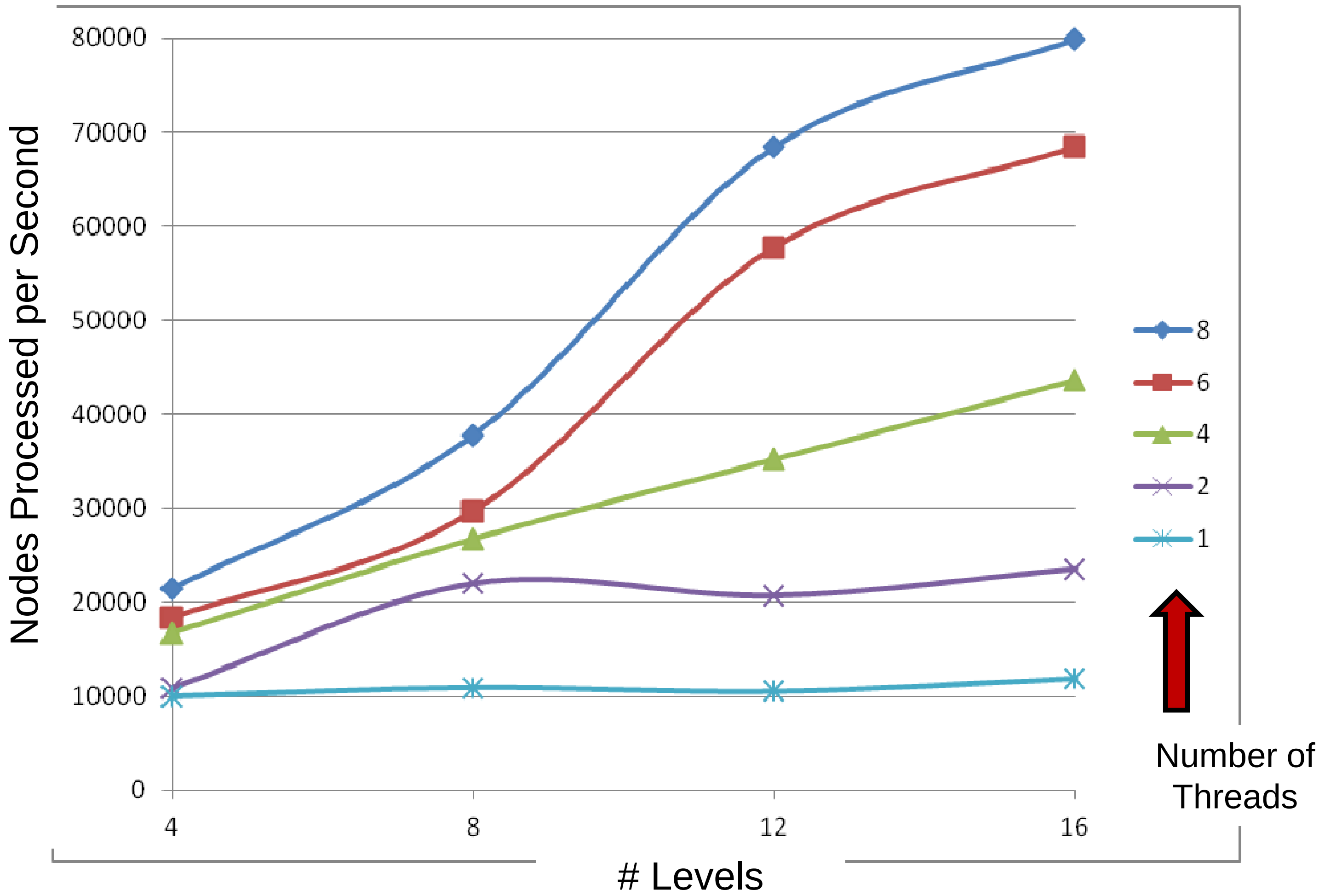
Benchmarking a Binary Task-driven Tree Traversal

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

Performance vs. Number of Threads



Performance vs. Number of Levels



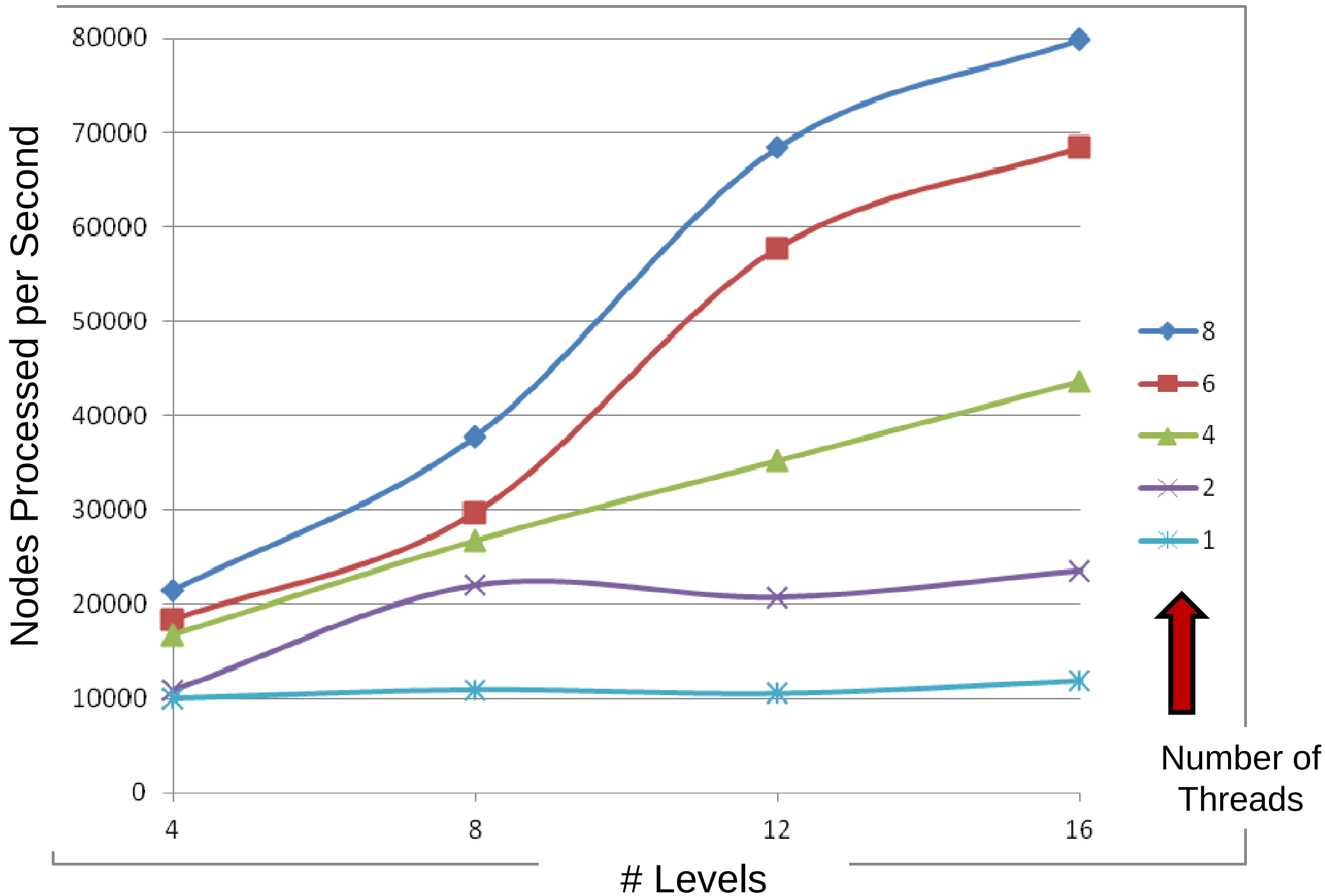
Parallelizing a Tree Traversal with Tasks

- 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

```
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

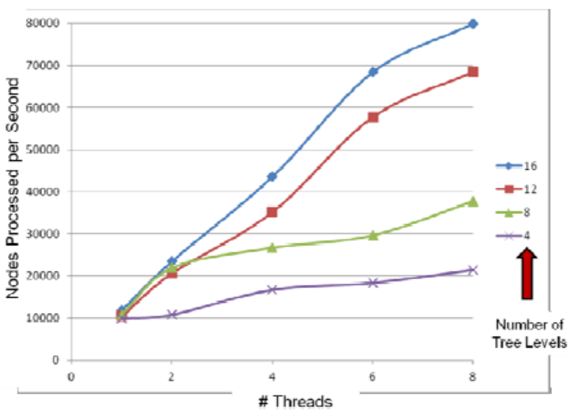


8-thread Speed-up ≈ 6.7

F_p ≈ ??%

Max Speed-up ≈ ??

Performance vs. Number of Threads



8-thread Speed-up ≈ 6.7

F_p ≈ ??%

Max Speed-up ≈ ??

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

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