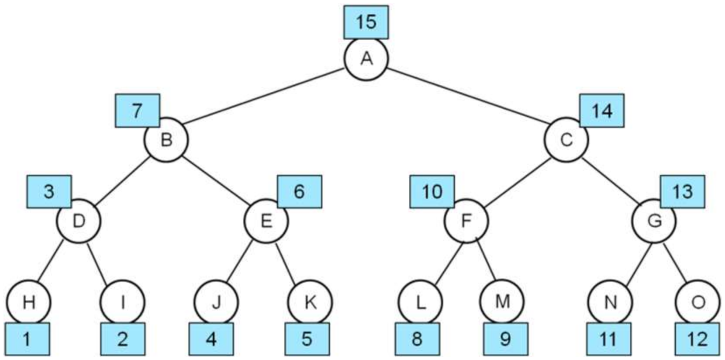
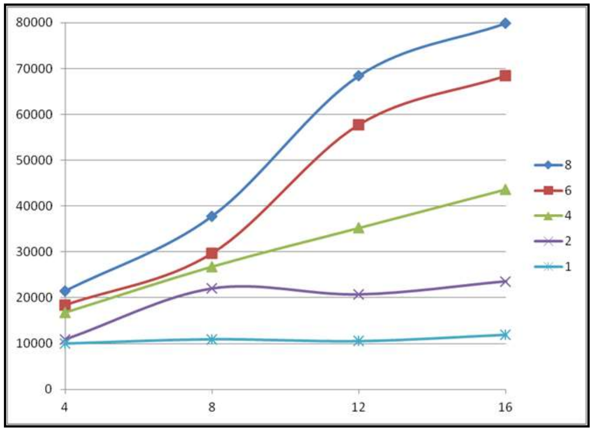
OpenMP Tasks



Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University
Computer Graphics



Remember OpenMP Sections?

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

```
#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:

#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.

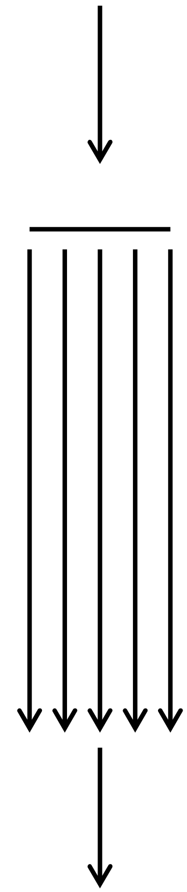
OpenMP Task Example: Something (Supposedly) Simple

Without this #pragma, thread #0 will have to do everything

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

		#pragma omp task
				fprintf( stderr, "B\n" );
}
```

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



#pragma omp task

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?)

Run #	1	2	3	4	5
	B	B	B	B	B
	A	B	A	A	A
	B	A	A	A	B
	A	A	B	B	A

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?

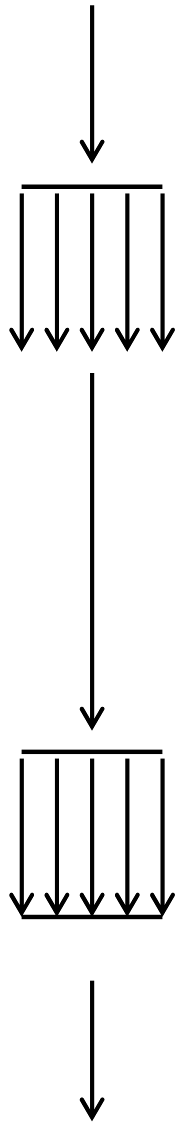
Not so simple, huh?

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 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?

The "single" Pragma

```
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 want:

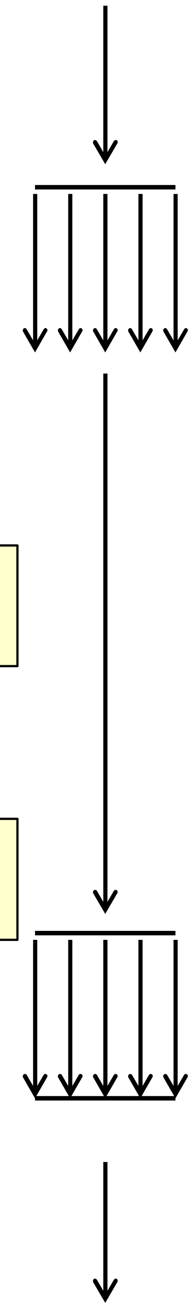
1. ***One*** thread to write the things to do down on the sticky notes
2. ***All*** threads to execute the sticky notes

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

```
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 (#pragma omp taskwait)

Causes all tasks to wait until they are completed (#pragma omp taskwait)



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

Without this #pragma, thread #0 will have to do everything

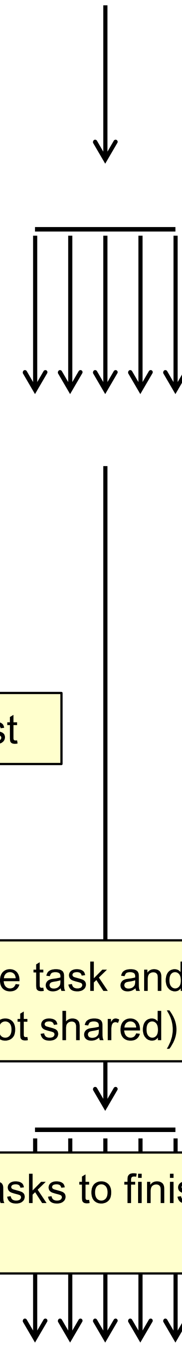
Without this #pragma, each thread will have to do a full traversal of the linked list – bad idea!

Writes “Process(p)” on a sticky note and adds it to the list

Copies the current value of p into the task and immediately makes it private (i.e., not shared)

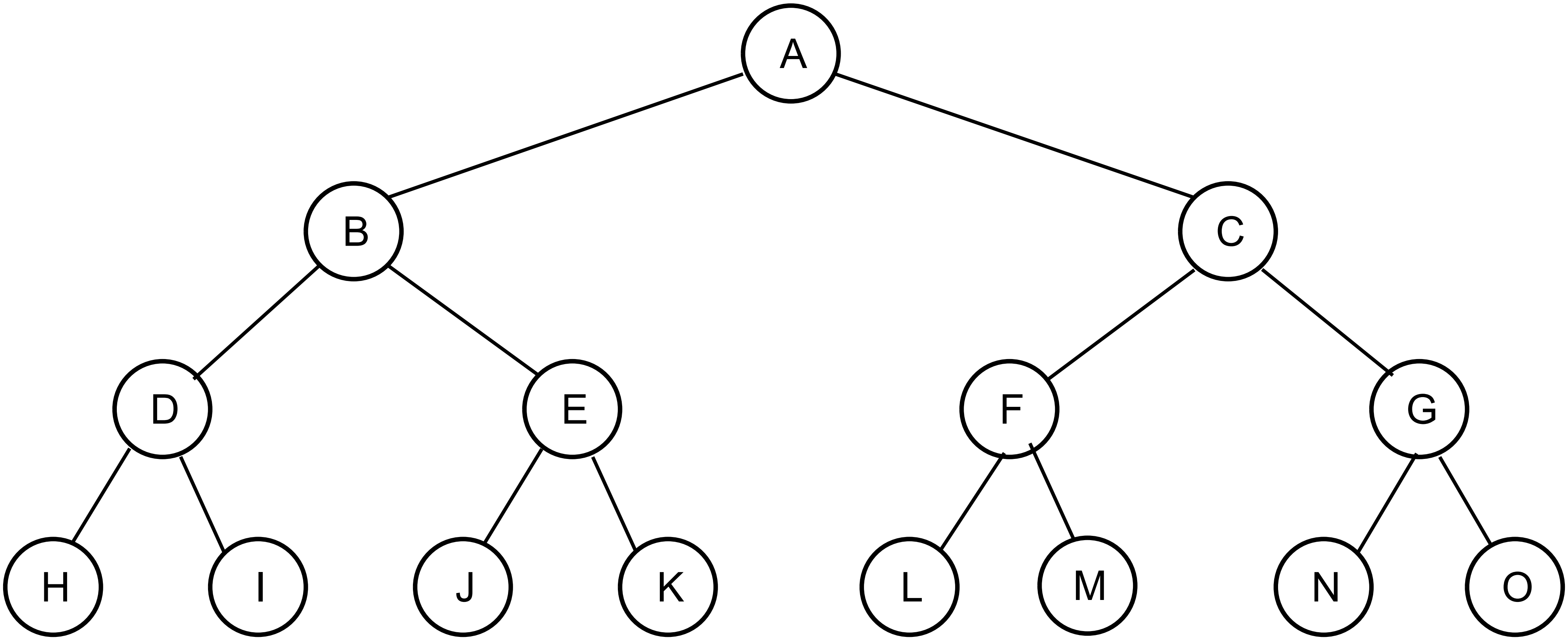
Put this here if you want to wait for all tasks to finish being executed before proceeding

```
#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
}
```



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 any order.
- We just need to visit all nodes.

Tree Traversal Algorithms

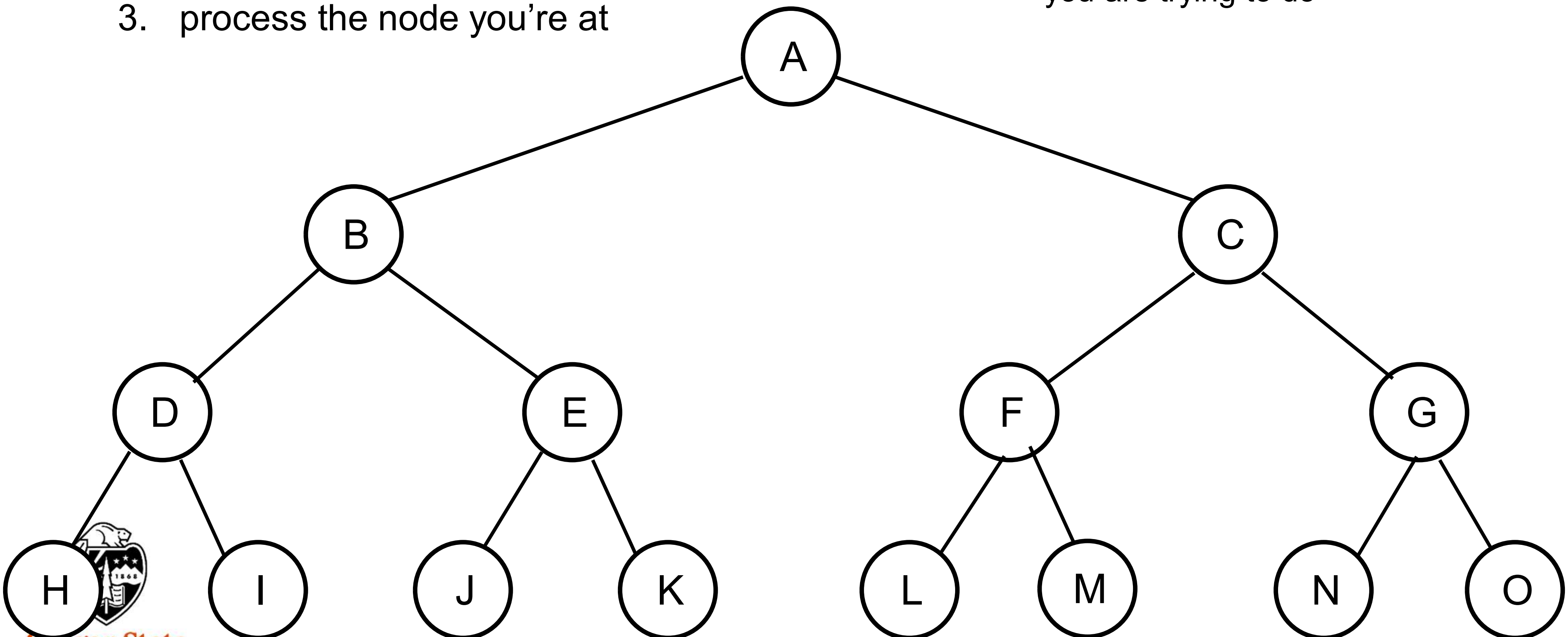
• This is common in graph algorithms, such as searching.

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

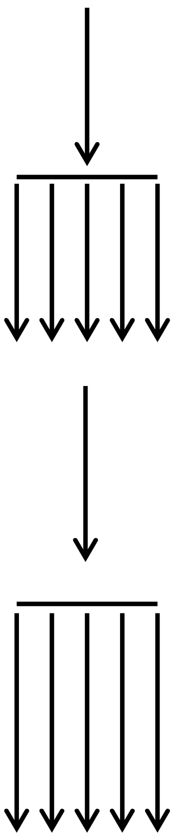
• Strategy at each node:

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



```
#pragma omp parallel

        #pragma omp single

        Traverse( root );

#pragma omp taskwait
```

Without this #pragma, thread #0 will have to do everything

Without this #pragma, each thread will have to do a full traversal of the binary tree – 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 firstprivate(n) untied
                Traverse( n->left );
        }



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

        #pragma omp taskwait

        Process( n );
}
```

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

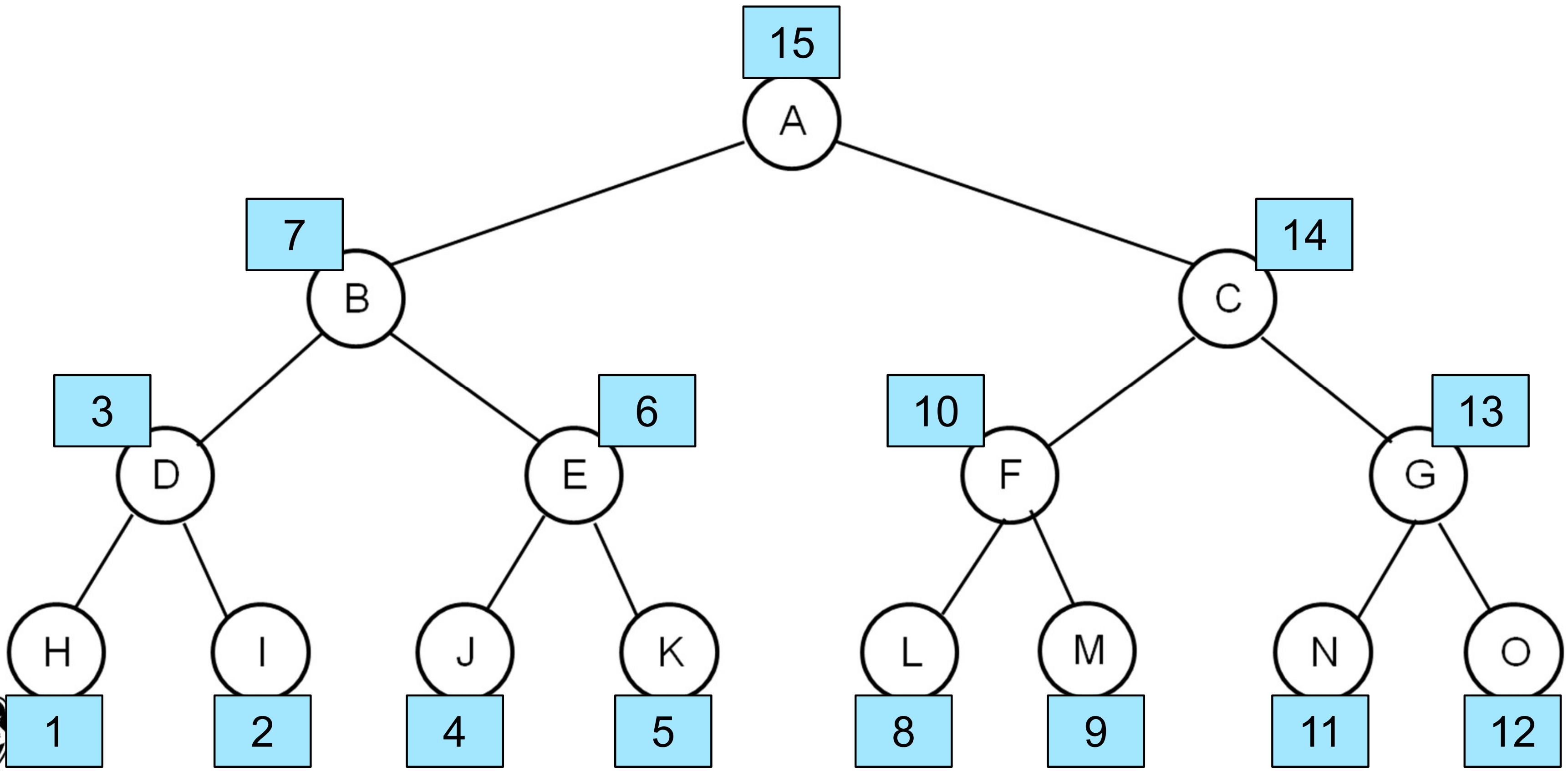
Oregon State University
Computer Graphics

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

Traverse(A);



Oregon State University
Computer Graphics

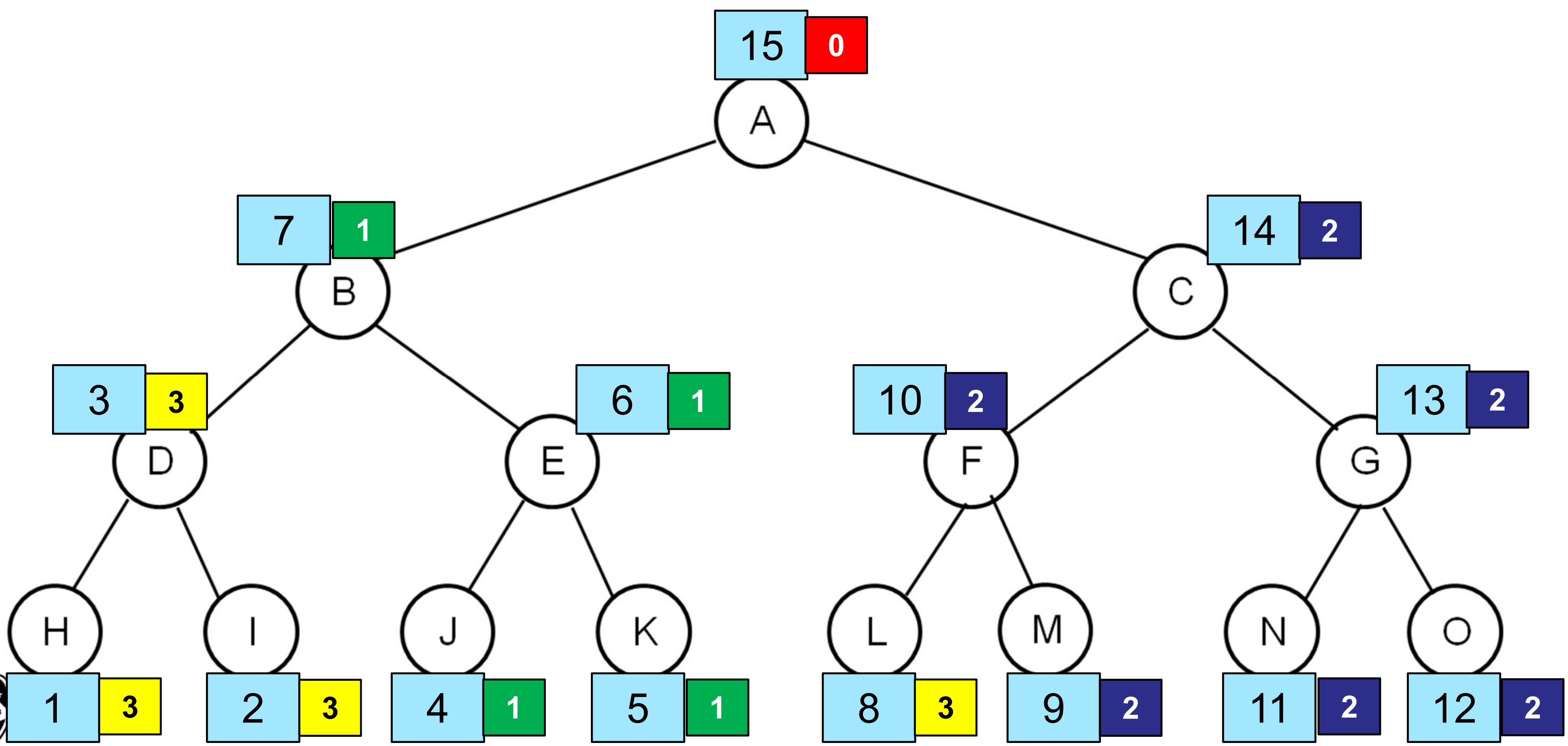
Parallelizing a Binary Tree Traversal with Tasks: *Tied*

(g++ 11.4)

Threads:

0 1 2 3

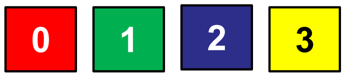
Traverse(A);



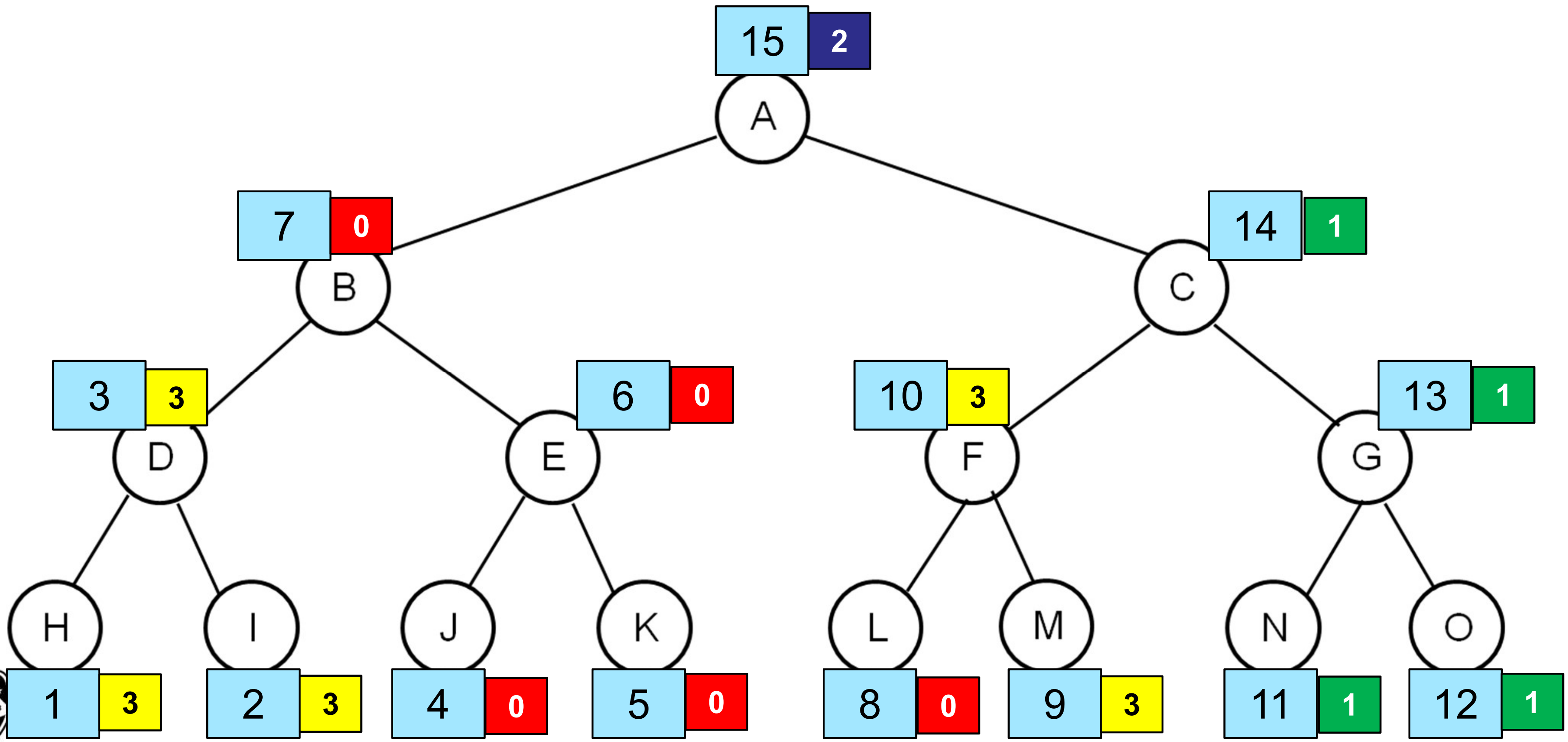
Parallelizing a Binary Tree Traversal with Tasks: *Untied*

(g++ 11.4)

Threads:



Traverse(A);



How Evenly Tasks Get Assigned to Threads
g++ vs. icpc

6 Levels – g++ 11.4:

Thread #	Number of Tasks
0	1
1	41
2	42
3	43

6 Levels – icpc 15.0.0:

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

12 Levels – g++ 11.4:

Thread #	Number of Tasks
0	3071
1	1
2	3071
3	2048

12 Levels – icpc 15.0.0:

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

How Evenly Tasks Get Assigned to Threads
Tied vs. Untied

6 Levels – g++ 11.4 -- Tied:

Thread #	Number of Tasks
0	1
1	41
2	42
3	43

6 Levels – g++ 11.4 -- Untied:

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

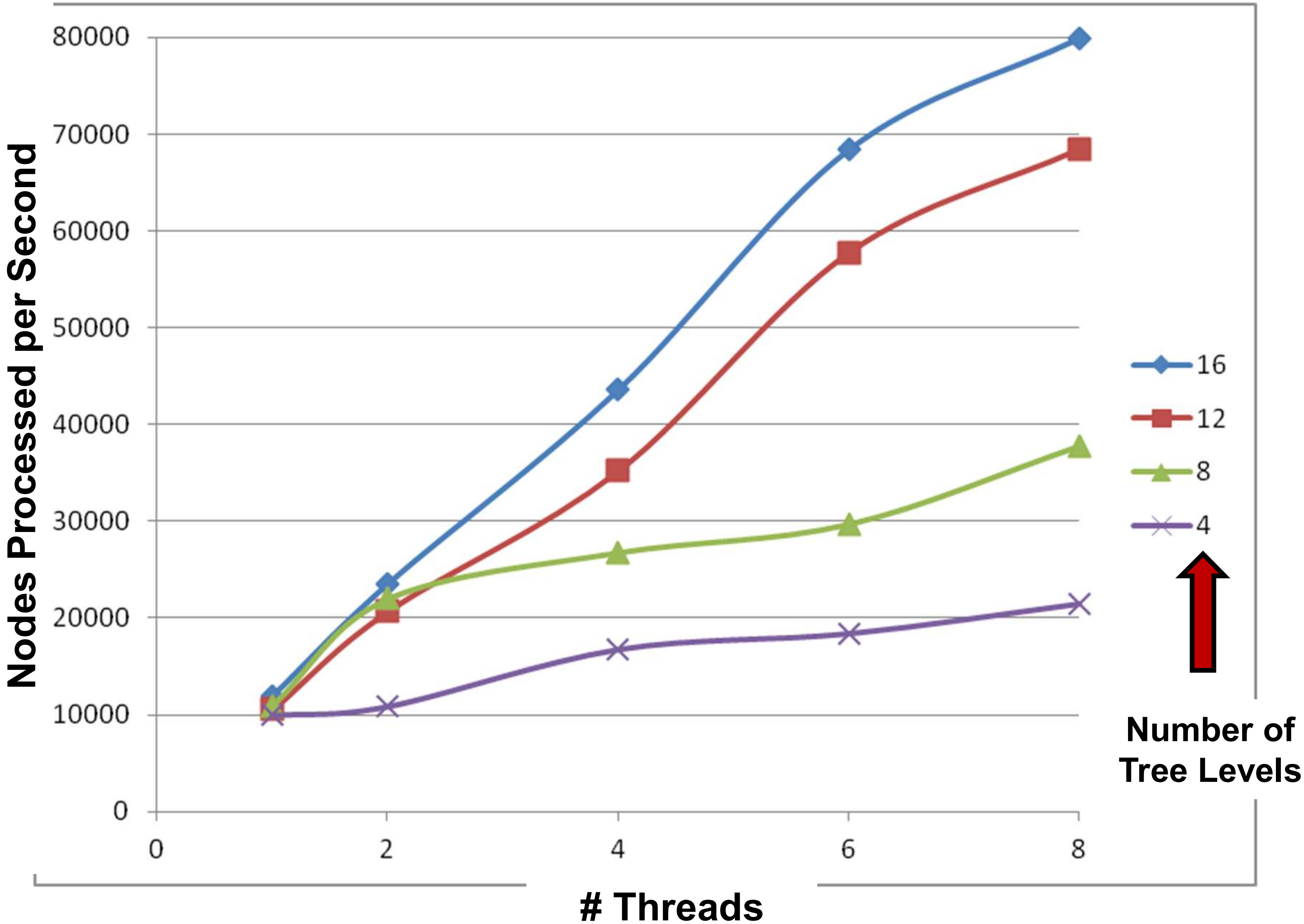
12 Levels – g++ 11.4 -- Tied:

Thread #	Number of Tasks
0	3071
1	1
2	3071
3	2048

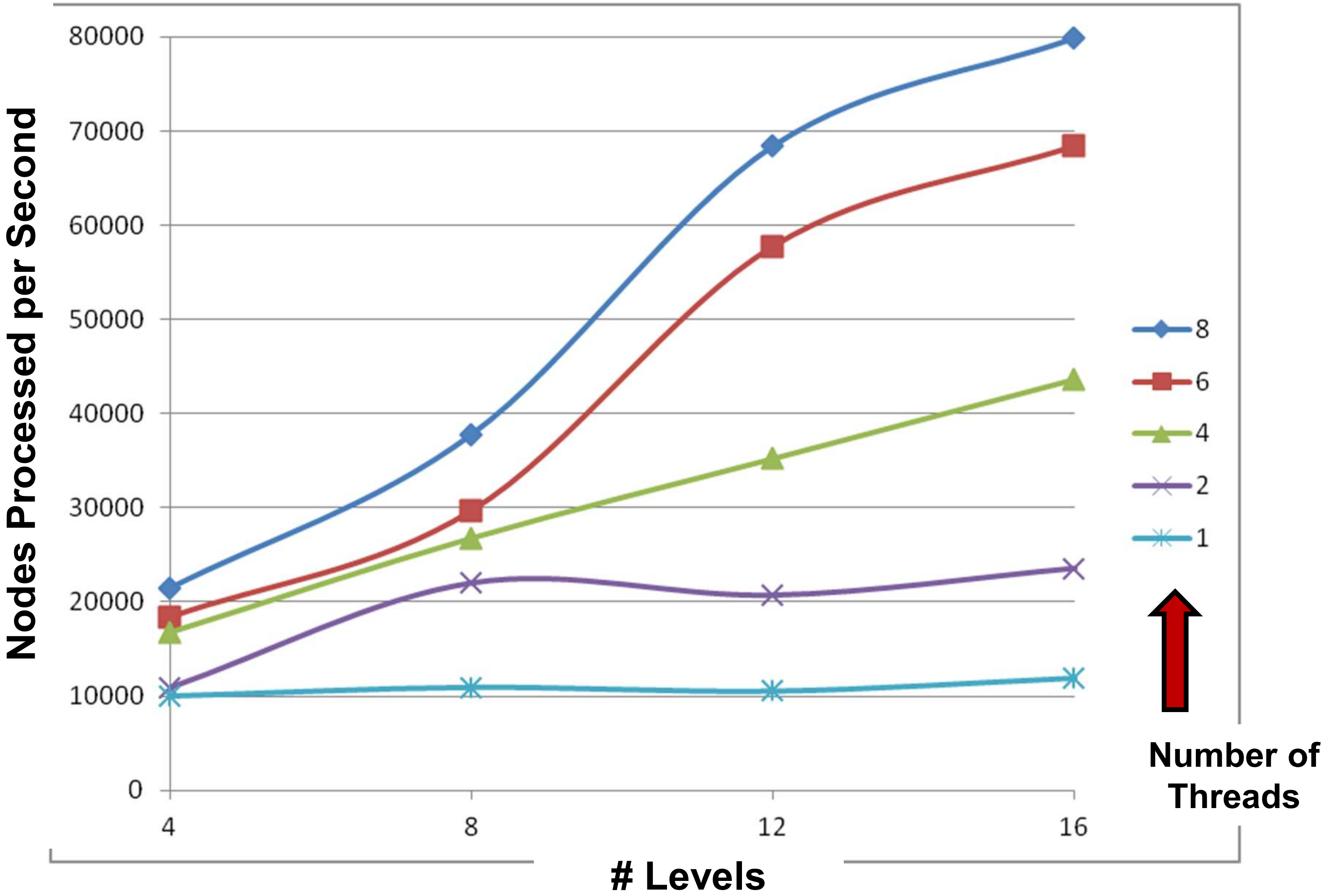
12 Levels – g++ 11.4 -- Untied:

Thread #	Number of Tasks
0	3071
1	1
2	2048
3	3071

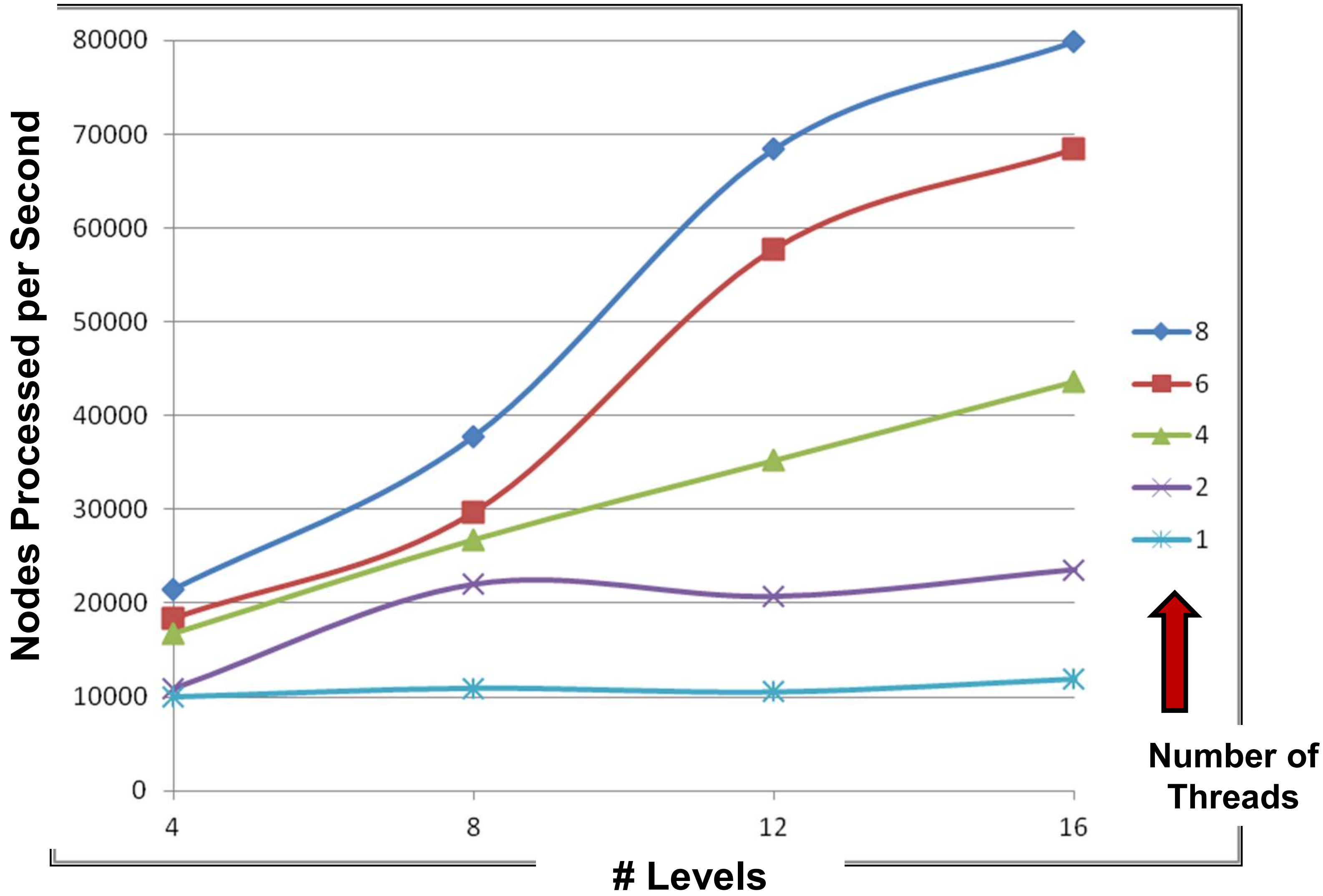
Performance vs. Number of Threads



Performance vs. Number of Levels



Performance vs. Number of Levels



8-thread Speed-up ≈ 6.7

F_p ≈ 97%

Max Speed-up ≈ 33x

Oregon State University
Computer Graphics

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.