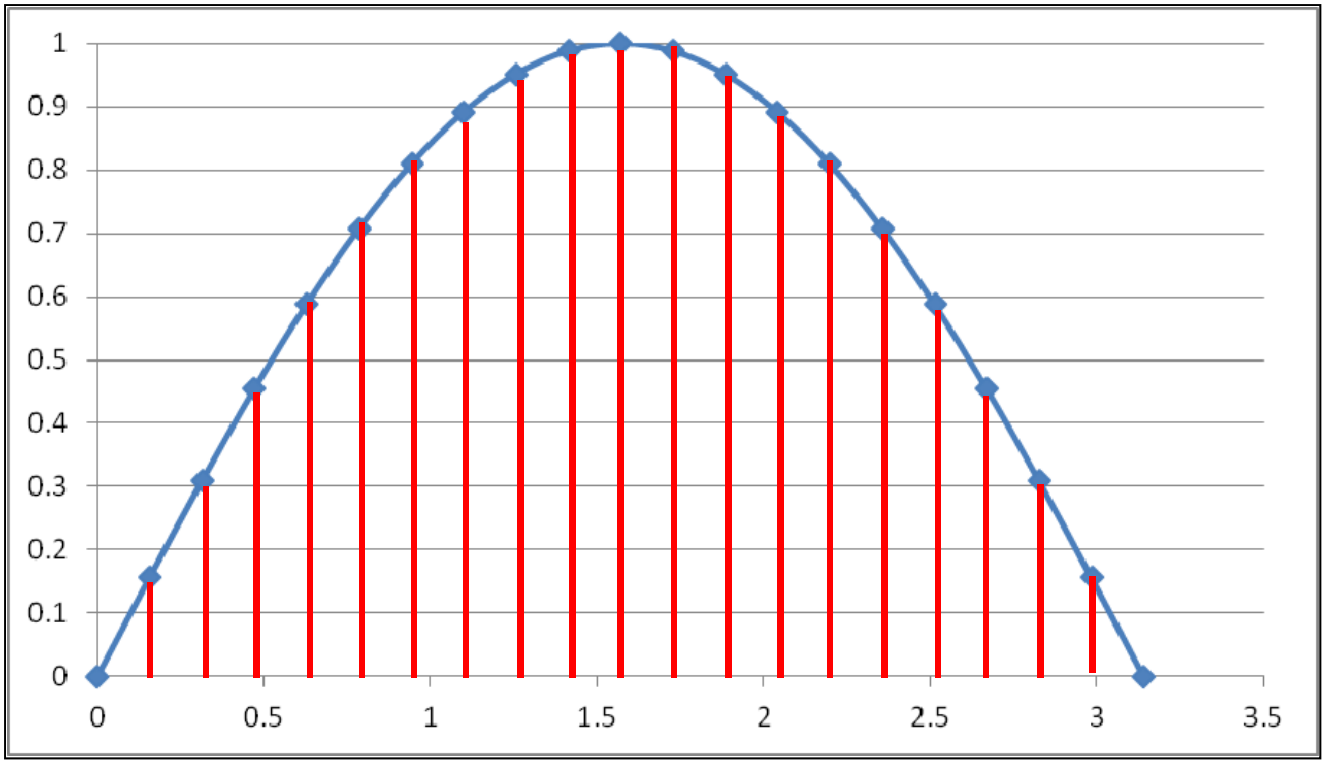
OpenMP Case Study:
Trapezoid Integration Example

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University

Find the area under the curve y = sin(x) for 0 ≤ x ≤ π using the Trapezoid Rule



Exact answer: $\int_0^{\pi} (\sin x)dx = -\cos x \Big|_0^{\pi} = 2.0$

Don't do it this way !

```
const  double A  =  0.;
const  double B  =  M_PI;

double dx = ( B - A ) / (float) ( numSubdivisions – 1 );
double sum = (  Function( A ) + Function( B )   )  /  2.;

omp_set_num_threads(  numThreads );

#pragma  omp  parallel  for  default(none),shared(dx,sum)
for( int i = 1; i < numSubdivisions - 1; i++ )
{
        double x = A + dx * (float) i;
        double f = Function( x );
        sum  +=  f;
}

sum *= dx;
```

Assembly code:

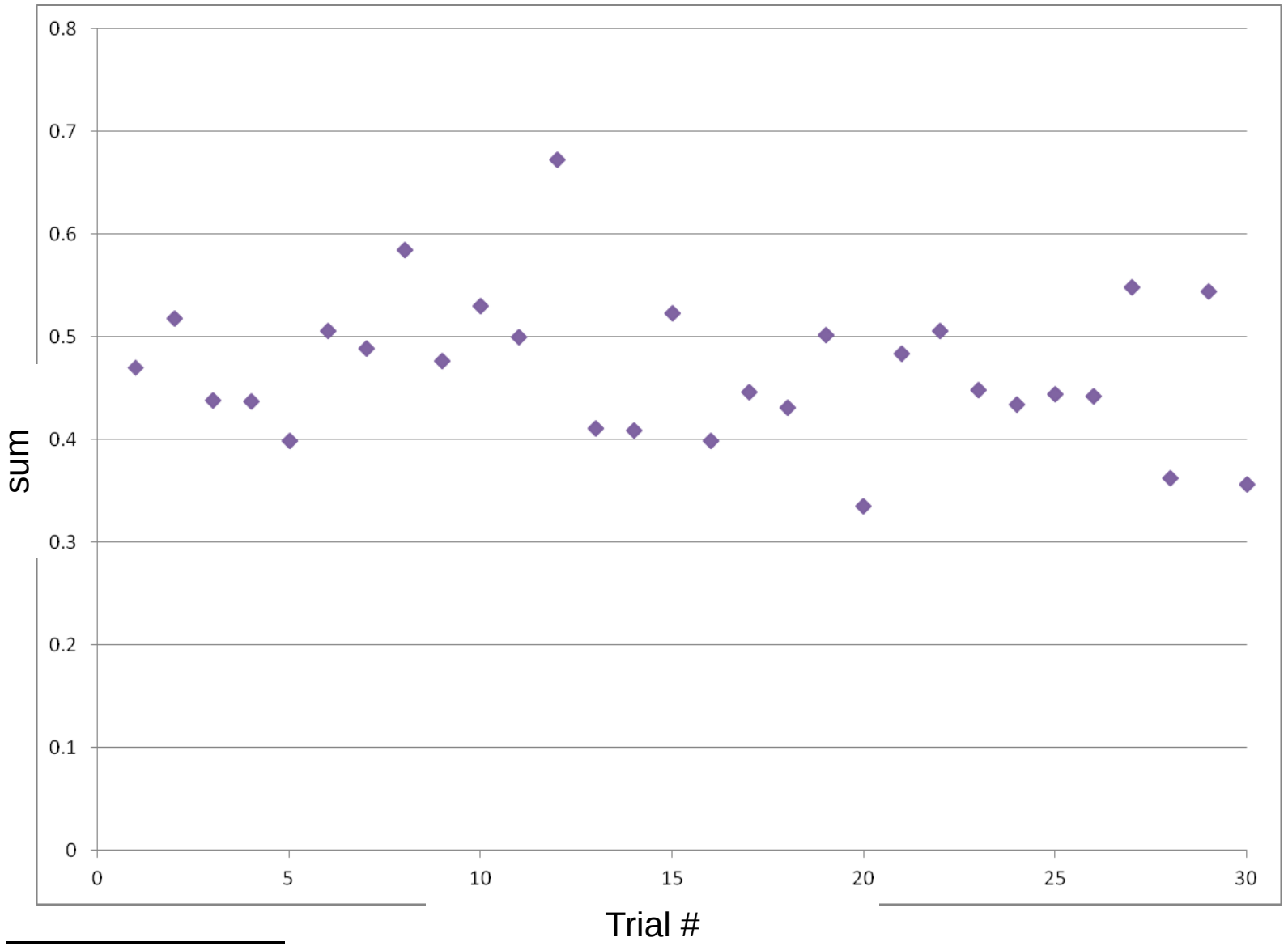
```
Load sum
Add   f
Store sum
```

What if the scheduler decides to switch threads right here?

The answer should be 2.0 *exactly*, but in 30 trials, it's not even close. And, the answers aren't even consistent. Why?

0.469635	0.398893
0.517984	0.446419
0.438868	0.431204
0.437553	0.501783
0.398761	0.334996
0.506564	0.484124
0.489211	0.506362
0.584810	0.448226
0.476670	0.434737
0.530668	0.444919
0.500062	0.442432
0.672593	0.548837
0.411158	0.363092
0.408718	0.544778
0.523448	0.356299

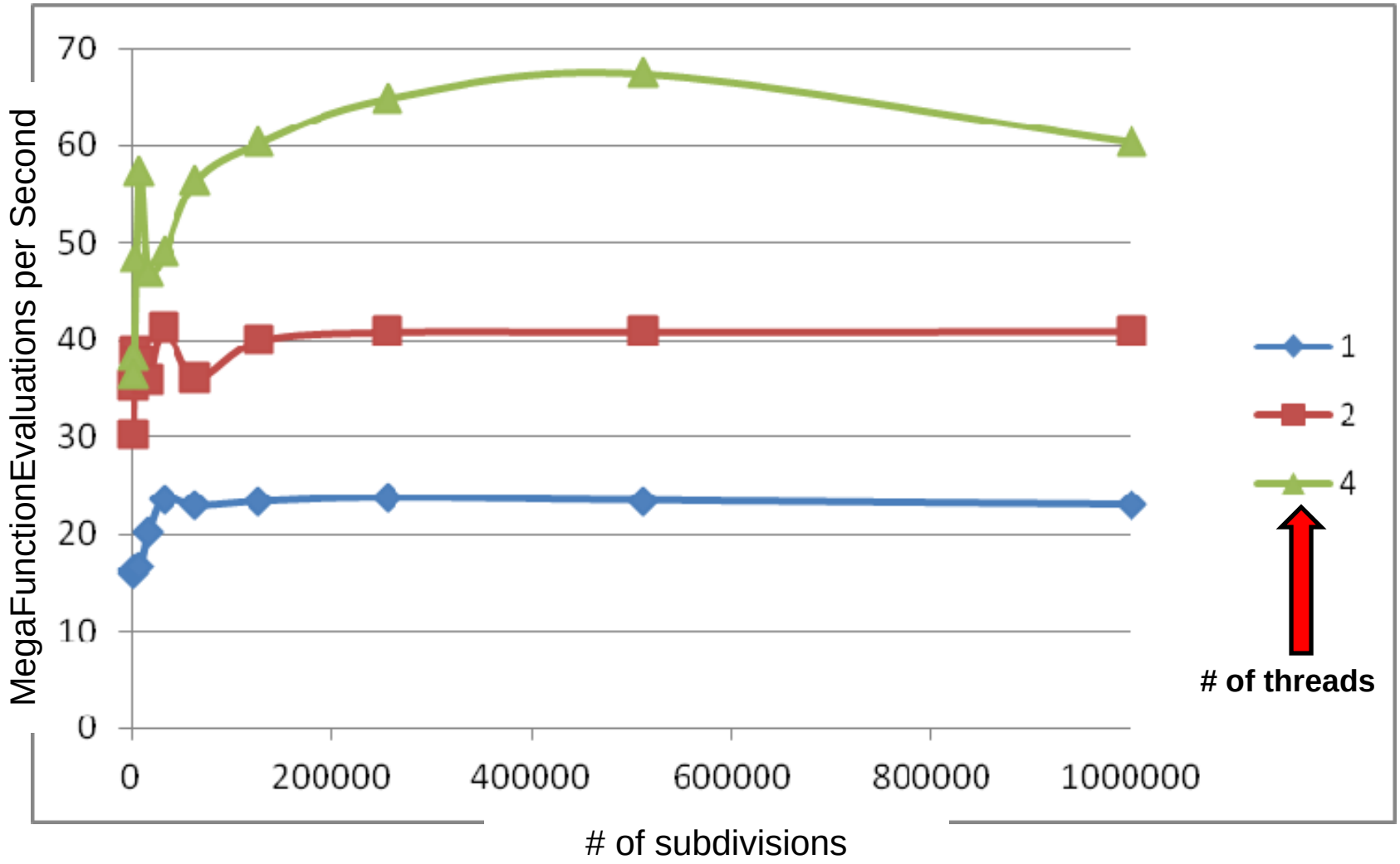
The answer should be 2.0 *exactly*, but in 30 trials, it's not even close. And, the answers aren't even consistent. Why?



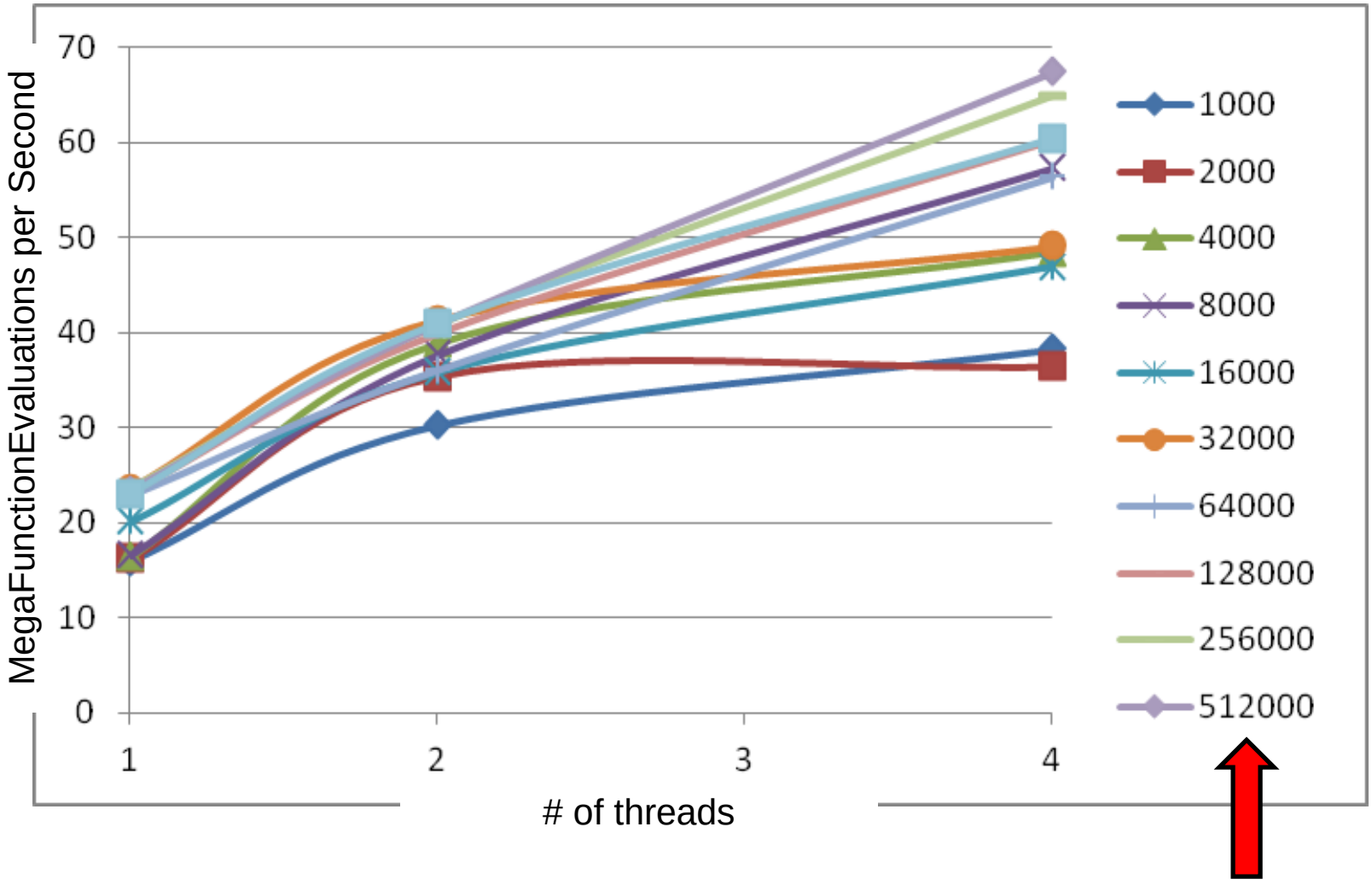
Do it this way !


```
const  double  A  =  0.;
const  double  B  =  M_PI;

double dx = ( B - A ) / (float) ( numSubdivisions – 1 );

omp_set_num_threads(  numThreads );

double sum = (  Function( A ) + Function( B )   )  /  2.;

#pragma  omp  parallel  for  default(none),shared(dx),reduction(+:sum)
for( int i = 1; i < numSubdivisions - 1; i++ )
{
        double x = A + dx * (float) i;
        double f = Function( x );
        sum  +=  f;
}

sum *= dx;
```

MegaFunctionEvaluations Per Second vs. Number of Subdivisions



MegaFunctionEvaluations Per Second vs. Number of Threads



# of subdivisions

Ways to Make the Summing Work: Reduction vs. Atomic vs. Critical

```
#pragma omp parallel for shared(dx),reduction(+:sum)
for( int i = 0; i < numSubdivisions; i++ )
{
        double x = A + dx * (float) i;
        double f = Function( x );
        sum += f;
}
```

1

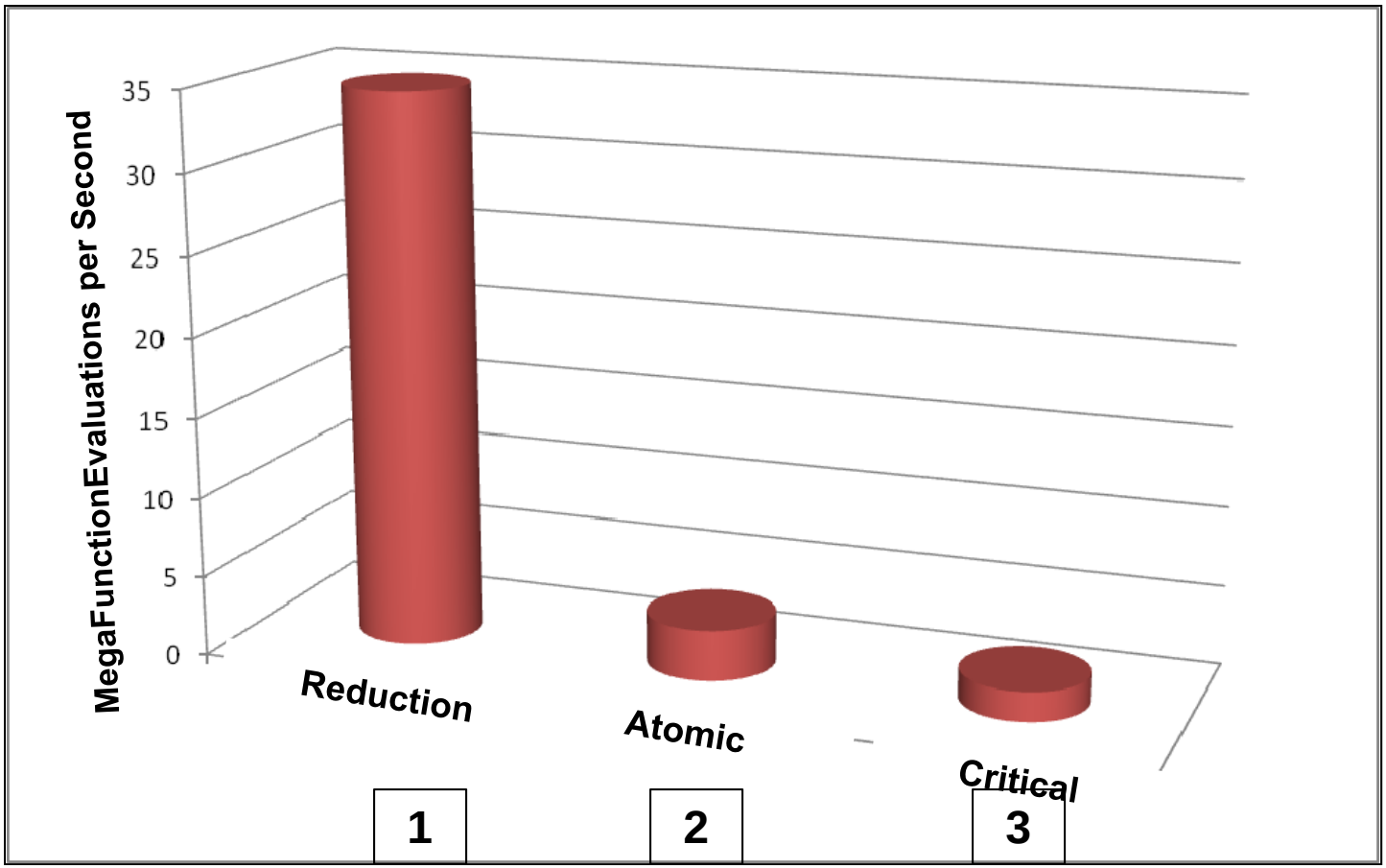
```
#pragma omp parallel for shared(dx)
for( int i = 0; i < numSubdivisions; i++ )
{
        double x = A + dx * (float) i;
        double f = Function( x );
        #pragma omp atomic
        sum += f;
}
```

2

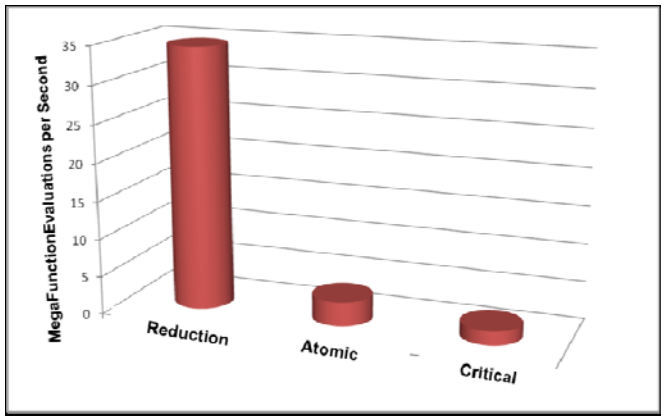
```
#pragma omp parallel for shared(dx)
for( int i = 0; i < numSubdivisions; i++ )
{
        double x = A + dx * (float) i;
        double f = Function( x );
        #pragma omp critical
        sum += f;
}
```

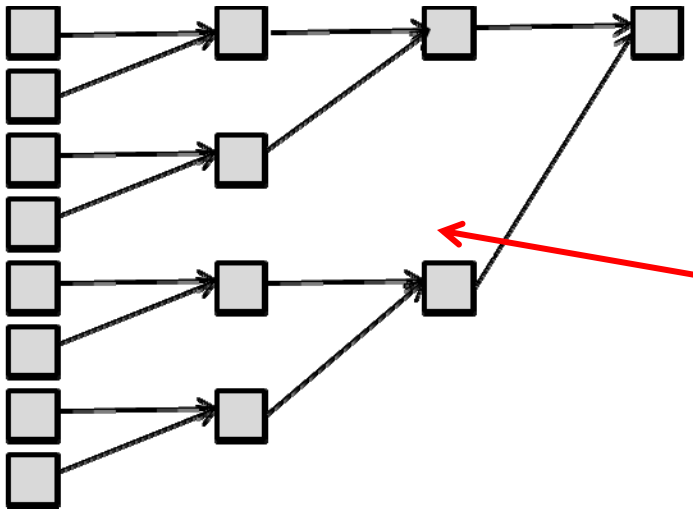
3

Speed of using Reduction vs. Atomic vs. Critical



Two Reasons Why Reduction is so Much Better in this Case

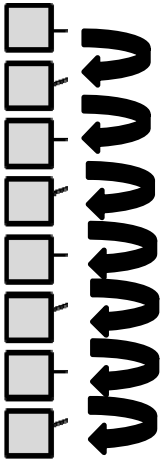




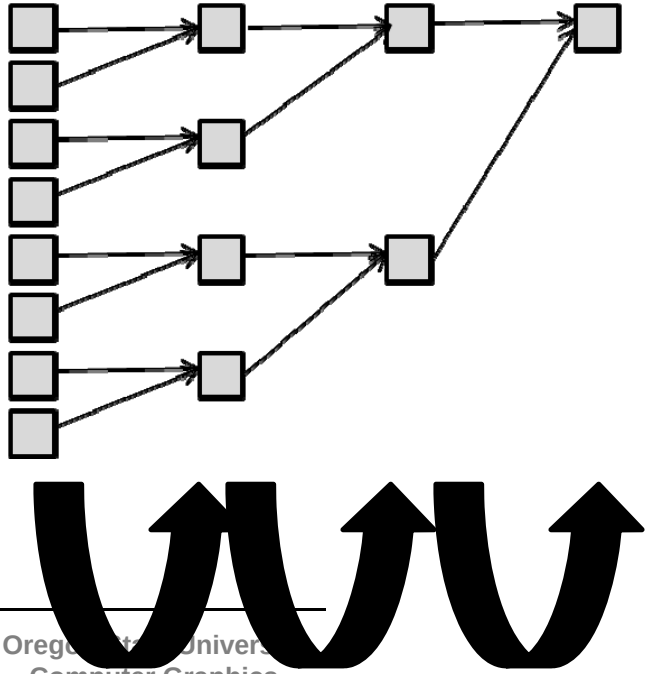
1. Reduction secretly creates a temporary private variable for each thread's running ***sum***. Each thread adding into its running ***sum*** doesn't interfere with any other thread adding into its running ***sum***, and so threads don't need to slow down to get out of the way of each other.

2. Reduction automatically creates a binary tree structure, like this, to add the N running sums in $\mathbf{log_2 N}$ time instead **N** time.

Why Reduction is so Much Better in this Case



Serial addition:
8 numbers requires 7 steps



Parallel addition:
8 numbers requires 3 steps