OpenMP Reduction Case Study: Trapezoid Integration Example

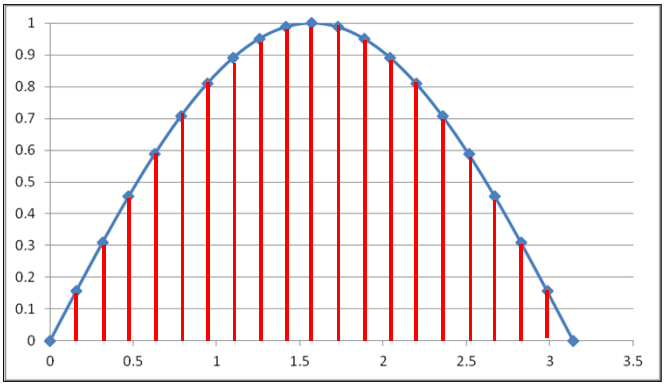
Oregon State University

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University
Computer Graphics

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



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

Oregon State University
Computer Graphics

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;
```

- There is no guarantee when each thread will execute this line
- There is not even a guarantee that each thread will finish this line before some other thread interrupts it.

Assembly code:

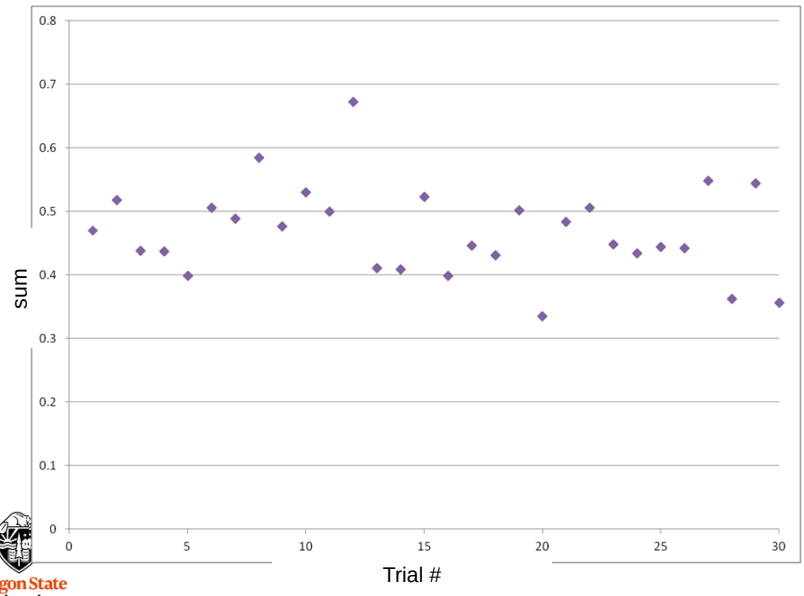
Load sum
Add f
Store sum

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;
```

There are Actually Three Ways to Make the Summing Work Correctly: Reduction vs. Atomic vs. Critical

7

```
#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

Reduction

```
#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

Uses a built-in hardware instruction.

```
#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;
}
```

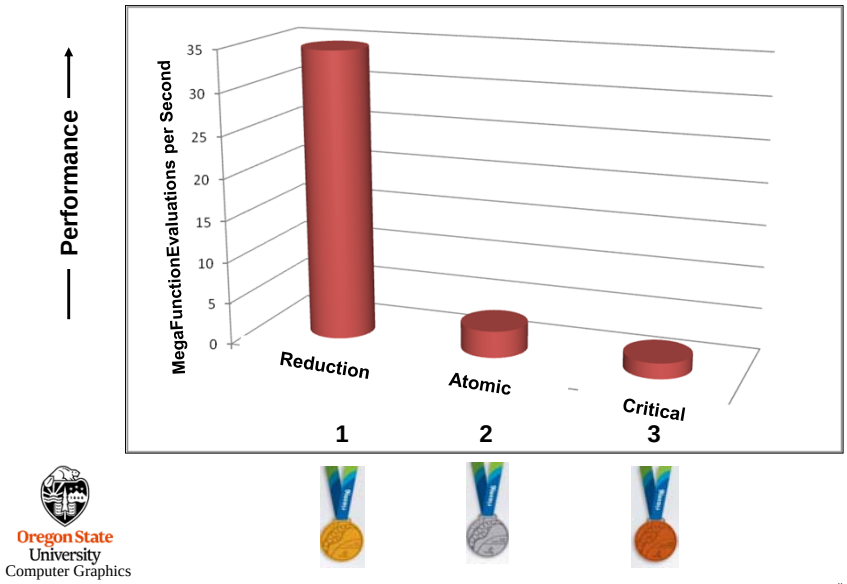
3

Disables the scheduler interrupts during the critical section.

mjb – March 15, 2020

Speed of using Reduction vs. Atomic vs. Critical

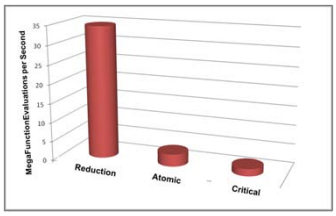
8



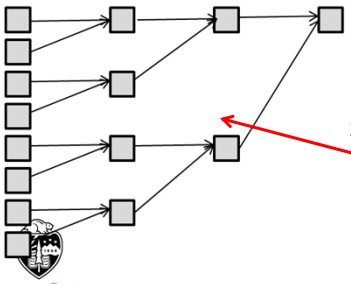
Oregon State University
Computer Graphics

mjb – March 15, 2020

Two Reasons Why Reduction is so Much Better in this Case




```
#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;
}
```



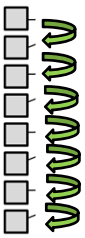
1. Reduction secretly creates a temporary private variable for each thread's running ***sum***. Each thread adding into its own running ***sum*** doesn't interfere with any other thread adding into *its* own running ***sum***, and so threads don't need to slow down to get out of the way of each other.

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

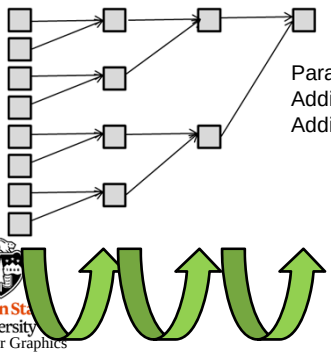
Oregon State University
Computer Graphics

O(N) vs. O($\log_2$N)

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



Serial addition:
Adding 8 numbers requires 7 steps
Adding 1,048,576 (1M) numbers requires 1,048,575 steps



Parallel addition:
Adding 8 numbers requires 3 steps
Adding 1,048,576 (1M) numbers requires 20 steps

Oregon State University
Computer Graphics

If You Understand NCAA Basketball Brackets, You Understand Reduction

11

Oregon State University
Computer Graphics

PN

mjb – March 15, 2020

Why Not Do Reduction by Creating Your Own *sums* Array, one for each Thread?

12

```
float *sums = new float [  omp_get_num_threads( )  ];
for( int i = 0; i < omp_get_num_threads( ); i++ )
        sums[i] = 0.;

#pragma omp parallel for private(myPartialSum),shared(sums)
 for( int i = 0; i < N; i++ )
{
      myPartialSum = …

      sums[  omp_get_thread_num( )  ]   +=   myPartialSum;
}

float sum = 0.;
for( int i= 0; i < omp_get_num_threads( ); i++ )
        sum += sums[i];

delete [ ] sums;
```

• This seems perfectly reasonable, it works, and it gets rid of the problem of multiple threads trying to write into the same reduction variable.

• The reason we don't do this is that this method provokes a problem called **False Sharing**. We will get to that when we discuss caching.

C

2020