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 $0 \leq x \leq \pi$
using the Trapezoid Rule

Exact answer: $\int_0^\pi (\sin x)dx = -\cos x \bigg|_0^\pi = 2.0$
Don’t do it this way!

```c
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:**

<table>
<thead>
<tr>
<th>Load sum</th>
<th>Add f</th>
<th>Store sum</th>
</tr>
</thead>
</table>

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!

```c
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

The graph shows the relationship between MegaFunctionEvaluations per Second and the number of subdivisions, with different numbers of threads indicated by the lines: 1 thread (blue), 2 threads (red), and 4 threads (green). The graph indicates that increasing the number of subdivisions generally increases the number of function evaluations, but this relationship may saturate or plateau depending on the number of threads used.
MegaFunctionEvaluations Per Second vs. Number of Threads

Graphic showing the relationship between the number of MegaFunctionEvaluations per Second and the number of threads, with different lines representing various numbers of subdivisions.
Ways to Make the Summing Work: Reduction vs. Atomic vs. Critical

1. #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;
   }

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 atomic
       sum += f;
   }

3. #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;
   }
Speed of using Reduction vs. Atomic vs. Critical

![Bar graph comparing reduction, atomic, and critical speeds. Reduction is the fastest, followed by atomic, and then 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 $\log_2 N$ time instead of $N$ time.
Why Reduction is so Much Better in this Case

Serial addition:
8 numbers requires 7 steps

Parallel addition:
8 numbers requires 3 steps