OpenMP Reduction Case Study: Trapezoid Integration Example

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

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!

\[
\begin{align*}
\text{const double A } & = 0.; \\
\text{const double B } & = M_\pi; \\
\text{double dx } & = (\text{B - A}) / (\text{float} (\text{numSubdivisions} - 1)); \\
\text{double sum } & = (\text{Function(A) + Function(B) \over 2}); \\
\text{omp_set_num_threads(numThreads);} \\
\text{#pragma omp parallel for default(none), shared(dx,sum)} \\
\text{for(int i = 1; i < numSubdivisions - 1; i++ )} \\
\{ \\
\text{double x = A + dx * (float) i; } \\
\text{double f = Function( x ); } \\
\text{sum += f; } \\
\} \\
\text{sum *= dx;}
\end{align*}
\]

- 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:

[Table showing assembly code results]

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

<table>
<thead>
<tr>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Trial 4</th>
<th>Trial 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.469635</td>
<td>0.398893</td>
<td>0.517984</td>
<td>0.446149</td>
<td>0.438868</td>
</tr>
<tr>
<td>0.437553</td>
<td>0.501763</td>
<td>0.398761</td>
<td>0.334996</td>
<td>0.489211</td>
</tr>
<tr>
<td>0.506564</td>
<td>0.484124</td>
<td>0.506362</td>
<td>0.488226</td>
<td>0.476670</td>
</tr>
<tr>
<td>0.523448</td>
<td>0.434737</td>
<td>0.363092</td>
<td>0.544778</td>
<td>0.356299</td>
</tr>
</tbody>
</table>
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;
```

There are Actually Three Ways to Make the Summing Work Correctly:

1. **Reduction**
   - Uses a built-in hardware instruction.
   - Disables the scheduler interrupts during the critical section.

2. **Atomic**
   - Enables the scheduler interrupts during the critical section.

3. **Critical**
   - Enables the scheduler interrupts during the critical section.

### Speed of using Reduction vs. Atomic vs. Critical

- **Reduction**
- **Atomic**
- **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 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 of $N$ time.

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

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

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

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

```c
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.

If You Understand NCAA Basketball Brackets, You Understand Reduction

Source: ESPN