OpenMP 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

Don't do it this way!

```cpp
cost double A = 0.;
cost 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;
```

Assemble 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?

<table>
<thead>
<tr>
<th>Trial #</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.469635</td>
</tr>
<tr>
<td>2</td>
<td>0.398893</td>
</tr>
<tr>
<td>3</td>
<td>0.517984</td>
</tr>
<tr>
<td>4</td>
<td>0.438868</td>
</tr>
<tr>
<td>5</td>
<td>0.437553</td>
</tr>
<tr>
<td>6</td>
<td>0.398761</td>
</tr>
<tr>
<td>7</td>
<td>0.506564</td>
</tr>
<tr>
<td>8</td>
<td>0.489211</td>
</tr>
<tr>
<td>9</td>
<td>0.506564</td>
</tr>
<tr>
<td>10</td>
<td>0.506564</td>
</tr>
<tr>
<td>11</td>
<td>0.506564</td>
</tr>
<tr>
<td>12</td>
<td>0.506564</td>
</tr>
<tr>
<td>13</td>
<td>0.506564</td>
</tr>
<tr>
<td>14</td>
<td>0.506564</td>
</tr>
<tr>
<td>15</td>
<td>0.506564</td>
</tr>
<tr>
<td>16</td>
<td>0.506564</td>
</tr>
<tr>
<td>17</td>
<td>0.506564</td>
</tr>
<tr>
<td>18</td>
<td>0.506564</td>
</tr>
<tr>
<td>19</td>
<td>0.506564</td>
</tr>
<tr>
<td>20</td>
<td>0.506564</td>
</tr>
<tr>
<td>21</td>
<td>0.506564</td>
</tr>
<tr>
<td>22</td>
<td>0.506564</td>
</tr>
<tr>
<td>23</td>
<td>0.506564</td>
</tr>
<tr>
<td>24</td>
<td>0.506564</td>
</tr>
<tr>
<td>25</td>
<td>0.506564</td>
</tr>
<tr>
<td>26</td>
<td>0.506564</td>
</tr>
<tr>
<td>27</td>
<td>0.506564</td>
</tr>
<tr>
<td>28</td>
<td>0.506564</td>
</tr>
<tr>
<td>29</td>
<td>0.506564</td>
</tr>
<tr>
<td>30</td>
<td>0.506564</td>
</tr>
</tbody>
</table>

Do it this way!

```cpp
cost double A = 0.;
cost 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,sum)
for( int i = 1; i < numSubdivisions - 1; i++ )
{
    double x = A + dx * (float) i;
    double f = Function( x );
    sum += f;
}
sum *= dx;
```

The answer should be 2.0 exactly, but in 30 trials, it's not even close.

And, the answers aren't even consistent. Why?
# 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;
   }

# 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 \( N \) time.