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>What if the scheduler decides to switch threads right here?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add f</td>
<td></td>
</tr>
<tr>
<td>Store sum</td>
<td></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?

<table>
<thead>
<tr>
<th>0.469635</th>
<th>0.398893</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.517984</td>
<td>0.446419</td>
</tr>
<tr>
<td>0.438868</td>
<td>0.431204</td>
</tr>
<tr>
<td>0.437553</td>
<td>0.501783</td>
</tr>
<tr>
<td>0.398761</td>
<td>0.334996</td>
</tr>
<tr>
<td>0.506564</td>
<td>0.484124</td>
</tr>
<tr>
<td>0.489211</td>
<td>0.506362</td>
</tr>
<tr>
<td>0.584810</td>
<td>0.448226</td>
</tr>
<tr>
<td>0.476670</td>
<td>0.434737</td>
</tr>
<tr>
<td>0.530668</td>
<td>0.444919</td>
</tr>
<tr>
<td>0.500062</td>
<td>0.442432</td>
</tr>
<tr>
<td>0.672593</td>
<td>0.548837</td>
</tr>
<tr>
<td>0.411158</td>
<td>0.363092</td>
</tr>
<tr>
<td>0.408718</td>
<td>0.544778</td>
</tr>
<tr>
<td>0.523448</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 = 0.;
#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;
```
Ways to Make the Summing Work: Reduction vs. Atomic vs. Critical

```cpp
#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;
}
```

```cpp
#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;
}
```

```cpp
#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

[Graph showing the speed comparison between Reduction, Atomic, and Critical]

1. Reduction
2. Atomic
3. 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