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;
```

- 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?
There are Three Ways to Make the Summing Work Correctly:

#1: Atomic

```c
#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;
}
```

- More lightweight than critical (#2)
- Uses a hardware instruction CMPXCHG (compare-and-exchange)
- Can only handle these operations:
  - `x++`, `++x`, `x--`, `--x`
  - `x op= expr`, `x = x op expr`, `x = expr op x`
  - where op is one of: `+`, `-`, `*`, `/`, `&`, `|`, `^`, `<<`, `>>`

#2: Critical

```c
#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;
}
```

- More heavyweight than atomic (#1)
- Allows only one thread at a time to enter this block of code (similar to a mutex)
- Can have any operations you want in this block of code

#3: Reduction

```c
#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;
}
```

- OpenMP creates code to make this as fast as possible
- Reduction operators can be: `+`, `-`, `*`, `&`, `|`, `^`, `&&`, `||`, `max`, `min`

---

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

(up = faster)

![Graph showing performance comparison between Reduction, Atomic, and Critical]
So, 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;
```

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(log₂N)

- **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

---

If You Understand NCAA Basketball Brackets, You Understand Power-of-Two Reduction

Source: ESPN
Why Not Do Reduction by Creating Your Own \textit{sums} Array, one for each Thread, Like This?

float *sums = new float [omp_get_num_threads()];
for( int i = 0; i < omp_get_num_threads(); i++ )
  sums[i] = 0.0;

#pragma omp parallel for private(myPartialSum),shared(sums)
for( int i = 0; i < N; i++ )
{
  myPartialSum = ...,
  sums[omp_get_thread_num()] += myPartialSum;
}

float sum = 0.0;
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.

• But the reason we don’t do this is that this method provokes a problem called \textbf{False Sharing}. We will get to that when we discuss caching.