The reduction clause

What if we wanted to write our trapezoidal rule openMP program a little differently?

If we developed a serial implementation of the trapezoidal rule, we'd probably use a slightly different function prototype. Rather than

```
void Trap(
    double a,
    double b,
    int n,
    double* global_result_p);
```

In this version of the trapizoidal program we don't utilize a pointer for the global result

we would probably define

```
double Trap(double a, double b, int n);
```

and our function call would be

```
global_result = Trap(a, b, n);
```

This is somewhat easier to understand and probably more attractive to all but the most fanatical believers in pointers.

We resorted to the pointer version, because we needed to add each thread's local calculation to get `global_result`. However, we might prefer the following function prototype:

```
double Local_trap(double a, double b, int n);
```

- What if instead of having a shared pointer variable like we do with global_result_p, we just simplify the body of the Trap function and the program by writing out the program slightly differently.
- Instead of Trap() we use local_trap() which is the same as Trap() but it doesn't have the global_result_p and more notably the function would not have a critical section inside of it

How could we modify our code to work with openMP and parallizing with this new version of writing our trapezoidal program?

- We would remove the crit section from the Trap() function and Rather, each thread would return its part of the calculation, the final value of its my_result variable. If we made this change, we might try modifying our parallel block so that it looks like this:

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
#      pragma omp critical
       global_result += Local_trap(double a, double b, int n);
    }
```

  - Now this would be bad because we are locking our paralllel function which means that each thread has to wait to do its calculation which would in turn make our program serial which defeats the purpose of the code
- We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call:

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
       double my_result = 0.0;  /* private */
       my_result += Local_trap(double a, double b, int n);
#      pragma omp critical
       global_result += my_result;
    }
```

  - Now the call to Local_trap is outside the critical section, and the threads can execute their calls simultaneously. Furthermore, since my_result is declared in the parallel block, it's private, and before the critical section each thread will store its part of the calculation in its my_result variable.

- OpenMP provides a cleaner alternative that also avoids serializing execution of Local_trap: we can specify that global_result is a reduction variable.
  - A reduction operator is an associative binary operation (such as addition or multiplication), and a reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands to get a single result.
    - Example of reduction:

      For example, if `A` is an array of `n` **int**s, the computation

      ```
      int sum = 0;
      for (i = 0; i < n; i++)
         sum += A[i];
      ```

      is a reduction in which the reduction operator is addition.
      - Reduction operator is addition
  - In OpenMP it may be possible to specify that the result of a reduction is a reduction variable. To do this, a reduction clause can be added to a parallel directive. In our example, we can modify the code as follows:

    ```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count) \
       reduction(+: global_result)
    global_result += Local_trap(double a, double b, int n);
    ```

  - First note that the `parallel` directive is two lines long. Recall that C preprocessor directives are, by default, only one line long, so we need to "escape" the newline character by putting a backslash (\) immediately before it.

    The code specifies that `global_result` is a reduction variable, and the plus sign ("+") indicates that the reduction operator is addition. Effectively, OpenMP creates a private variable for each thread, and the run-time system stores each thread's result in this private variable. OpenMP also creates a critical section, and the values stored in the private variables are added in this critical section. Thus the calls to `Local_trap` can take place in parallel.

    The syntax of the `reduction` clause is

    ```
    reduction(<operator>: <variable list>)
    ```

    - Note that subtraction still works as intended but floats and doubles may differ when different numbers of threads are used.
  - How it works:

    When a variable is included in a `reduction` clause, the variable itself is shared. However, a private variable is created for each thread in the team. In the `parallel` block each time a thread executes a statement involving the variable, it uses the private variable. When the `parallel` block ends, the values in the private variables are combined into the shared variable. Thus our latest version of the code

    ```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count) \
       reduction(+: global_result)
    global_result += Local_trap(double a, double b, int n);
    ```

    effectively executes code that is identical to our previous version:

    ```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
       double my_result = 0.0;  /* private */
       my_result += Local_trap(double a, double b, int n);
#      pragma omp critical
       global_result += my_result;
    }
    ```

- One final point to note is that the threads' private variables are initialized to 0. This is analogous to our initializing my_result to zero. In general, the private variables created for a reduction clause are initialized to the identity value for the operator. For example, if the operator is multiplication, the private variables would be initialized to 1.

Table 5.1 Identity values for the various reduction operators in OpenMP.

Operator	Identity Value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0