# Creating a correct threaded version

A program has a *race condition* if the correct behavior of that program depends on the timing of its execution. With 2 or more threads, the program trap-omp.C has a race condition concerning the shared variable integral, which is the accumulator for the summation performed by that program's for loop.

- When threadct == 1, the single thread of execution updates the shared variable integral on every iteration, by reading the prior value of the memory location integral, computing and adding the value f(a+i*h), then storing the result into that memory location integral. (Recall that a variable is a named location in main memory.)
- But when threadct > 1, there are at least two independent threads, executed on separate physical cores, that are reading then writing the memory location integral. The incorrect answer results when the reads and writes of that memory location get out of order. Here is one example of how unfortunate ordering can happen with two threads:

```
Thread 1                             |  Thread 2
                                     |
code:   integral += f(a+i*h);        |  code:   integral += f(a+i*h);
                                     |
exec:   1. read value of  integral   |  exec:
        2. add  f(a+i*h)             |          1. read value of  integral
                                     |          2. add  f(a+i*h)
        3. write sum to  integral    |
                                     |          3. write sum to  integral
```

  In this example, during one poorly timed iteration for each thread, Thread 2 reads the value of the memory location integral before Thread 1 can write its sum back to integral. The consequence is that Thread 2 replaces (overwrites) Thread 1's value of integral, so the amount added by Thread 1 is omitted from the final value of the accumulator integral.

  Can you think of other situations where unfortunate ordering of thread operations leads to an incorrect value of integral? Write down at least one other bad timing scenario. *Note:* Thousands of occurrences of bad timing lead to the computed answer for integral being off by often 25% or more.

- One approach to avoiding this program's race condition is to use a separate local variable integral for each thread instead of a global variable that is shared by all the threads. But declaring integral to be private instead of shared in the pragma will only generate threadct partial sums in those local variables named integral -- the partial sums in those temporary local variables will *not* be added to the program's variable integral. In fact, the value in those temporary local variables will be discarded when each thread finishes its work for the parallel for if we simply make integral private instead of shared.

- Can you re-explain this situation in your own words?
- Fortunately, OpenMP provides a convenient and effective solution to this problem.
- The OpenMP clause reduction(+: integral) will
    a. cause the variable integral to be private (local) during the execution of each thread, *and*
    b. add the results of all those private variables, *then finally*
    c. store that sum of private variables in the *global* variable named integral.
- Add this clause to your OpenMP pragma, and remove the variable integral from the shared clause, then recompile and test your program. You should now see the correct answer 2.0 when computing with multiple threads -- a correct multi-core program!

A code segment is said to be *thread-safe* if it remains correct when executed by multiple independent threads. The body of this loop is *not* thread-safe.

Some libraries are identified as thread-safe, meaning that each function in that library is thread-safe. Of course, calling a thread-safe function doesn't insure that the code with that function call is thread-safe. For example, the function f() in our example, is thread-safe, but the body of that loop is *not* thread-safe.