OpenMP Race Conditions

Dr. Christopher Marcotte — *Durham University*

Race conditions

One of the archetypal problems in simple parallel programming is the effective parallelization of sums, with independent summands. These occur in the dot (inner) product, and thus appear ubiquitously, e.g. in matrix-multiplication.

Today we will take a look at a simple example,

$$S = \sum_{n=1}^{N} n = \frac{N(N+1)}{2},$$

for which we have an analytical solution.

A simple serial program to implement this sum might look like this:

serial.c

```c
#include <stdio.h>

int main(){
  const int N = 40000;
  int sum = 0;

  for (int n=1; n<N+1; n++){
    sum+=n;
  }

  printf("Result is %d. It should be %d.\n", sum, N*(N+1)/2);
  return 0;
}
```

If we parallelise the `for` loop in the usual way using OpenMP, we will run into a data race. This arises because multiple threads are reading from and writing to the variable `int sum` simultaneously.

Challenge

Parallelize the above program **incorrectly** with a `#pragma omp parallel for` and run it across a range of thread values p and a range of N, and determine an approximate probability for the success of the incorrect program on your machine. Compare to the plot in Lecture 3.

There are several ways to approach a solution. We will show a manual way to do so, and then describe a few others for you to attempt (you may wish to come back to this exercise after a later lecture).

Hint

Funnily enough, the result of the race condition depends a bit on your processor architecture, clock speed, and how many OpenMP threads you have used relative to the number of physical cores. Obviously, producing indeterminate garbage is not what we want from computers[1].

If you get a correct answer for an incorrect implementation, try increasing the number of threads with `OMP_NUM_THREADS`. E.g. on my Apple Silicon based Mac, I could reliably get correct results from incorrect

[1] Despite all the time, money, and effort it took to generate software that can produce effectively indeterminate garbage.

implementations with small numbers of threads. I had to go up to 1024 threads to get reliably wrong answers with some of these wrong implementations.

Manual

The first approach is entirely manual. We allocate an array `int partialsums[4]` in which to accumulate the partial sums. We compute each partial sum *taking care with the OpenMP thread number (index)*. Then we accumulate into `int sum` with a serial `for` loop over the partial sum array.

manual.c

```c
#include <stdio.h>
#include <omp.h>
int main(){
    const int N = 40000;
    int partialsums[4] = {0,0,0,0};
    #pragma omp parallel for
    for (int n=1; n<N+1; n++){
        partialsums[omp_get_thread_num()]+=n;
    }
    int sum = 0;
    for (int n=0; n<4; n++){
        sum += partialsums[n];
    }
    printf("Result is %d. It should be %d.\n",
        sum, N*(N+1)/2);
    return 0;
}
```

Exercise

The parallel fraction of the serial program is $1 - f = 0$, identically. Estimate the parallel fraction of the manual implementation program as a function of N (assuming all additions cost the same and memory accesses (including allocations) are *free*).

Use `#pragma omp [...] default(none)` with the `private(...)` and `shared(...)` clauses to make the data availability of each variable fully explicit.

Can this code be run without the OpenMP header `<omp.h>`? How many threads should it be run with? Will this program run with fewer than four threads? Under what conditions will it give a correct answer?

We allocate a four element array in which to accumulate the partial sums – what happens if we use more than four OpenMP threads? Will it give a correct answer?

Adapt this code to work with *any* number of OpenMP threads. Estimate the new parallel fraction. Compare it to the parallel fraction you estimated earlier.

We can see that handling this pattern manually has some additional memory management which may be considered burdensome in a more interesting program.

A `#pragma omp critical` section

An arguably simpler method is to avoid the temporary shared array `int partialsums[M]` in favor of a private variable `int partialsum` on each thread. However, once you have these private `int partialsum`s, you still need to recover their values to accumulate into `int sum`. In the program snippet below, you have a hint — we have

`#pragma omp parallel` generating a whole parallel block (delineated by `{...}`) — to aid you in implementing a manual reduction using OpenMP.

critical.c

```c
#include <stdio.h>
#include <omp.h>
int main(){
  const int N = 40000;
  int sum = 0;
  #pragma omp parallel
  {
    // ??
    // ??
    for (int n=1; n<N+1; n++){
      // ??
    }
    // ??
  }
  printf("Result is %d. It should be %d.\n", sum, N*(N+1)/2);
  return 0;
}
```

Exercise

Replace the commented lines (e.g. `// ??`) to implement a parallel reduction using a private `int partialsum` variable and a `#pragma omp critical` region. Use `default(none)` with `private()` and `shared()` to make the data availability of each variable explicit. Estimate the parallel fraction of the implementation. Is it different from the manual approach above? Is it different from the version you implemented with arbitrary number of threads? What would your result be if you replaced `#pragma omp critical` with `#pragma omp single`?

An `#pragma omp atomic` update

Using a `#pragma omp critical` section is not the only way to reduce the intermediate calculations. While the `#pragma omp critical` blocks protects a section of code from multiple thread accesses, `#pragma omp atomic` protects a *variable* from multiple thread accesses.

Exercise

Adapt your implementation with the `#pragma omp critical` region to use an `#pragma omp atomic` update of `int sum` from the partial sums.

Where else could the `#pragma omp atomic` pragma be placed? Why is that placement better or worse than how you structured it?

Are the data availability tags `#pragma omp [...] private(...)` and `#pragma omp [...] shared(...)` changed? Why?

A `#pragma omp reduction` operator

This sort of pattern is called a reduction or fold, and OpenMP includes a specific pragma, `#pragma omp reduction(operator:variable)` to make this pattern easy to implement. In the pragma, `operator` should be a binary associative operator (like `+`) and `variable` should be the variable into which the reduction is stored. The reduction clause makes a private copy of the variable for per-thread calculations, and takes precedence over the data access clauses – therefore, one must be careful how the data access clauses change when using a reduction.

Exercise

Adapt the serial implementation to use the OpenMP `#pragma omp reduction` pragma to perform the reduction. Use `#pragma omp [...] default(none)` and data availability tags to make the memory sharing explicit. Are they different from what you expected based on your other implementations?

Investigate the *strong scaling* of your implementations. Can you make any conclusions about bottlenecks for this calculation? How might you change it to make the parallel performance scaling better?

Challenge

The logistic map is a simple model of population growth, and is given by the recurrence relation

$$x_{n+1} = rx_n(1 - x_n),$$

with $r \in [0, 4]$. Implement a code which produces N iterates of the logistic map starting from $x_0 = 0.94857394956$ for M different values of $0 \leq r \leq 4$, and then performs a reduction across the N values to calculate the average of the iterates. Plot the average against r. Test the performance of your code for different values of N and M, and estimate the serial fraction (using the same assumptions as earlier in this exercise, additionally assuming one multiplication costs k times an addition).

Aims

- Review of race conditions in OpenMP threaded code.
- Introduction of several ways to resolve this simple race condition.
- A gentle introduction to localisation environment variables for OpenMP threading.