Lab #3: OpenMP loop parallelization

PCSE 2015

Justification

An OpenMP parallel region creates a team of threads that initially all execute the same code. To obtain useful parallelism the work in this region needs to be split among the threads. In this exercise you will do so by explicitly dividing loop iterations.

Later you will learn idiomatic constructs for this; spelling it out yourself allows you better to explore the options and see their effect.

Lab assignment

The file `pi1.c` computes π by Riemann integration:

$$\pi = 4\int_0^1 y\,dx \quad \text{where } x^2+y^2=1$$

This is implemented as a loop that gradually accumulates the value $\pi/4$.

```
quarterpi = 0.; h = 1./nsteps;
for (i=0; i<nsteps; i++) {
  double
    x = i*h,
    y = sqrt(1-x*x);
  quarterpi += h*y;
}
pi = 4*quarterpi;
```

1. Put a parallel region around the loop. Inside the parallel region, before the loop, compute loop bounds for the current thread. That is, each thread computes an independent number of integration samples:

```
thread_num = ...
first_step = ...
last_step = ...
for (i=first_step; i<=last_step; i++) {
    .....
```

Make sure these variables are private:

- In C, declare them inside the parallel region, or
- In Fortran (or C), add a clause
  `$!OMP parallel private(thread_num)`

Observe that execution speeds up, but the numerical result is wrong. What is the reason for the incorrect result?

2. You can fix the problem with incorrect result by giving each thread its own location for accumulating a partial result; these partial results are then summed together after the loop.
   It is easiest to declare a static array of partial results: `double quarterpi[16]`, assuming that you will never have more than 16 threads[1].
   Test your solution. The value of π should now be correct.
   Important: use compiler optimization level `-O0` **'oh-zero'**
   Are you getting a speedup from multiple threads? As much as you expected?
3. The previous solution has a problem with 'false sharing' because up to 8 threads write to the same cacheline. Solve this problem by making sure that the locations that the threads write to are spaced 8 doubles apart. You should now get better speedup from using multiple threads.
4. Recompile and rerun the previous two codes, but now use compiler optimization level `-O2`. Is the previous behaviour still there?

1. You can also use dynamic allocation, in which case use the `posix_memalign` (see below), rather than `new` or `malloc`.