OpenMP Reduction Performance

Dr. Christopher Marcotte — *Durham University*

In a previous exercise, you learned the basics of data races in OpenMP, and how to avoid them using a manual approach, critical section, atomic update, or compiler-assisted reduction. Unfortunately, that problem wasn't compute-intensive enough to make the parallelism worthwhile. A calculation which is more likely to be worthwhile is a dot-product between two length-N vectors:

$$a \cdot b \equiv \sum_{n=1}^{N} a_n b_n.$$

Hint

You should complete the exercise about race conditions in OpenMP, first!

Exercise

You should first initialize a and b – in parallel! – where $a_i = i + 1$ and $b_i = -i \bmod(i, 2)$, cf.

reduction-template.c

```c
void init(double *a, double *b, size_t N){
  /* Intialise with some values */
  // Parallelize this loop!
  for (size_t i = 0; i < N; i++) {
    a[i] = i+1;
    b[i] = (-1)*(i%2) * i;
  }
}
```

Then parallelize the function `double dot(double *a, double *b, size_t N)` using the manual, critical, atomic, and reduction approaches.

Exercise

Investigate the weak scaling of the code. Estimate the serial fraction, and compare in a plot to Gustafson's law.

Hint

Since we supply the size of the vectors at the command line, it is pretty easy to perform a weak scaling experiment using bash scripting:

```bash
gcc -O1 -fopenmp reduction-template.c -o r
for (( n = 1 ; n <= 32 ; n+=1 )); do
  OMP_NUM_THREADS=${n} ./r $((1000000 * n))
done > weak_timing.dat
```

Core Localisation

The Hamilton compute nodes are dual-socket. That is, they have two chips in a single motherboard, each with its own memory attached. Although OpenMP treats this logically as a single piece of shared memory, the performance of the code depends on where the memory accessed is relative to where the threads are. OpenMP exposes some extra environment variables that control where threads are physically placed on the hardware. We'll look at how these variables affect the performance of our reduction. Use the implementation you determined to be the fastest above.

Hint

The relevant environment variables for controlling placement are `OMP_PROC_BIND` and `OMP_PLACES`. We need to set `OMP_PLACES=cores`. *Don't forget to do this in your submission script.*

There are two options for `OMP_PROC_BIND`: `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread`. The former places threads on cores that are physically close to one another, first filling up the first socket, and then the second. The latter spreads threads out between cores: thread 0 to the first socket, thread 1 to the second, and so on. We'll now look at the difference in scaling performance when using different values for `OMP_PROC_BIND`.

Exercise

Which value for `OMP_PROC_BIND` works better? Is there a difference at all?

Look up the other options for `OMP_PLACES`; why might `OMP_PLACES=threads` or `OMP_PLACES=sockets` be less immediately useful for measuring performance?

Aims

- Practice with practial implementation of level 1 BLAS routines in OpenMP
- Practice with parallel performance scaling and diagnostics
- Practice with the specification of OpenMP environment variables and their impact on performance