Reproducibility in Parallel Computing using Floating-Point Arithmetic

Ganesh Gopalakrishnan, School of Computing, University of Utah, Salt Lake City

Main Idea: When one parallelizes routines such as `dot product`, the results obtained are a function of thread schedules. This problem vexes researchers, as they like to see bitwise reproducibility for easy regression testing. Given the scanty coverage of floating-point arithmetic in a practical end-user sense in today's curricula, this exercise prepares students with an eye-opening C experiment that shows that (a) parallel addition can give "truer answers" than serial addition, and (b) answers given by parallel addition can vary with thread scheduling (thus introducing result-reproducibility as a challenge).

The assignment itself is easy to describe fully (and I owe the basic idea to Miriam Leeser of Northeastern University; the detailed assignment is mine): Initialize an array `A` with the smallest floating-point number (say m), and then run the linear summation (left-hand side) and tree summation (right-hand side) for various array sizes (powers of two). Observe that after a while, the linear sum saturates (even if we add more m's), while with parallel sum, the value grows with the size of `A`.

```
for (int i=0; i<ASize-1; ++i)
  {  A[i+1] += A[i]; }
```

```
for (int str=1; str<=ASize/2; str*=2)
    { for (int i=0; i<ASize; i+=2*str)
       { A[i]+=A[i+str]; } }
```

Concepts Covered: (1) Floating-point round-off errors being a function of the values involved (in serial addition, once the accumulated value grows large, further m's do not budge it. (2) Floating-point arithmetic not being associative. (3) The tree pattern occurs widely (e.g., GPU reduction loops). (4) The notion of the smallest representable floating-point number (for floats, it is $1.2 \cdot 10^{-38}$; the students can check this). (5) Reproducibility is a central theme. (6) It also allows us to say that one must not "explain away" result variability in lower order fractional digits, as the root-cause may be a deeper bug – such as array indexing being off by 1, or a data race – in which case, it may still have a lurking uglier manifestation slated to occur at a a later inopportune moment (Murphi's law).

Targeted Students: Any student taking parallel computing or an earlier class.

In what contexts used before: This was assigned to an UG class I am teaching now (10 students). While no parallelism is involved in this example, this illustration helped them recognize result variability when the tree reduction was parallelized in OpenMP. In hindsight, I was glad that I introduced this example early in the course. When we ran TBB-reduction, its work-stealing reduction process, in its default mode, showed result-variability always.

What prerequisites assumed: Students only need a Wikipedia-level knowledge of floating-point, and basic C programming.

Strengths/Weaknesses: Strengths include the ability to appreciate result-divergence in a concrete setting. Weaknesses include this very tree pattern being not illuminating of reproducibility.

Variations of Interest: They can write a similar OpenMP reduction loop, and run it under dynamic scheduling, and see result-variability as a function of the number of threads and/or the specific execution. Likewise, while teaching TBB, such variability comes up naturally. They can then appreciate TBB's deterministic reduction option.

They can also switch-over the precision to double, and see the extent of variability (also coming up with m's value for doubles).