Multi-Threaded Matrix Multiply

Jesse Bannon
University of Washington, Tacoma

July 2, 2017

In this paper, I outline the performance characteristics between three implementations of dense matrix-matrix multiply: single-threaded, multi-threaded, and multi-threaded Strassen algorithm. Each implementation stores matrices row-major in memory and uses double-precision floating points. These metrics were gathered using an Intel i7 5820K CPU, which includes six cores and twelve threads using Intel's Hyper-Threading Technology.

Single-Threaded One core performs a matrix-matrix multiply $C = AB$ using a trivial triple-nested for-loop.

Multi-Threaded Each thread t_i performs a subset of the matrix-matrix multiply $C_i = AB_i$, where B is partitioned by consecutive columns $B_i = \{b_j, b_{j+1}, \ldots, b_{j+n}\}$ evenly amongst all t_i. In the case that B's columns are greater than the number of threads, we assign a single column to a subset of threads and leave the remaining idle.

Strassen Algorithm Each operation (dense matrix-matrix add, subtract, multiply) of the Strassen algorithm is multi-threaded amongst all threads. Every matrix-matrix multiply recursively calls Strassen algorithm, inconsequently performing a depth-first traversal in the recursion. After the split input matrix size reaches a certain threashold, the multi-threaded implementation is used.

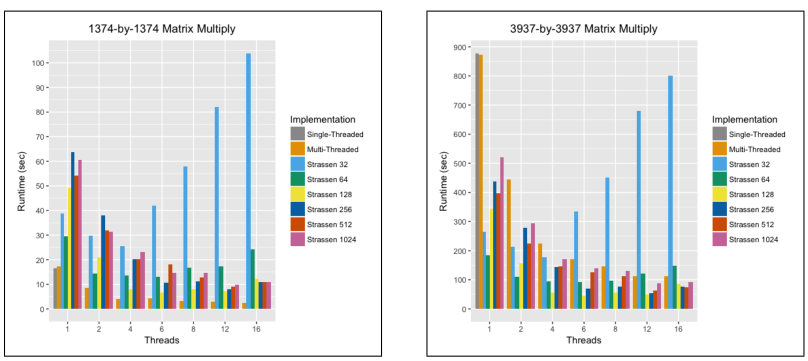

Figure 1: Elapsed time to perform dense matrix-matrix multiply

When multiplying a 1374-by-1374 matrix with itself, the 16-thread multi-threaded multiply performed the best with a 2.46 second runtime. Each thread was responsible to multiply the matrix with its (approximately) 1375-by-85 subset. The performance comes from the memory lookups on B when multiply its columns. The Strassen algorithm, in this case, over-parallelized the problem size.

The 3937-by-3937 matrix performs best using the 6-thread Strassen algorithm, with a 128-size stopping condition for dense matrix-matrix multiplies, in 45.11 seconds. The second best metric has the same configuration except it uses 12 threads. In this case, hyper-threading decreases performance. The problem size in this case is large enough to where the Strassen algorithm does not over-parallelize, which takes advantage of its lower complexity of $\mathcal{O}(n^{log_2 7})$ compared to the traditional matrix-multiply complexity of $\mathcal{O}(n^3)$.

Challenges The main challenge was implementing the Strassen algorithm. After a few hours of debugging, the implementation fell into place. Using the depth-first traversal allowed me to reuse my multithread matrix multiply which helped.

Further Improvements Parallelizing the Strassen algorithm in a way where multiple threads can work on different operations (multiply, add, subtract) simultaniously could perhaps yield better performance. This could potentially decrease memory-lookups by having multiple threads simultaneously fetch different parts of the matrices from main memory into the L3, where other threads would subsequently read that same data later.

Locking Dense matrix-matrix multiply does not require locking because the problem can be paritioned into completely independent sub-problems. As described above, each thread t_i is responsible for computing $C_i = AB_i$. Once all threads complete, C is formed. The closest thing to a lock in this problem is a barrier, to wait for all t_i to complete so we know C is formed.