Preliminary Report: Performance Benchmarking and "Real-world" performance of Distributed Matrix Multiplication Algorithms

Chris Keefe
School of Informatics, Computing & Cyber Systems
Northern Arizona University, Flagstaff, AZ, U.S.A.
ChrisKeefe@nau.edu

***Summary* — Though teaching and research commitments have prevented me from completing a full implementation of Cannon or Fox, I have made significant progress on understanding the algorithms and building robust, extensible components. Baseline benchmarks have been recorded with an improved implementation of my initial naive sequential implementation, and all major job tooling is in place to make running tests and aggregating data straightforward. Additionally, key conflicting deliverables have been handled, leaving more time to focus on this work.**

Keywords—distributed-memory computing, MPI, matrix multiplication, Cannon's Algorithm, Fox's Algorithm, performance benchmarks, applied computing

I. Progress Repoprt

Until today, this project has had to take a back seat to other research, teaching, and coursework requirements. Many of those have now been resolved, leaving me with more time to dedicate to my work here. I have made limited, but useful progress towards completing my goals.

Since our last meeting, I've developed a stronger comprehension of Cannon's algorithm, and believe I have cleared up my initial confusion around the scale at which it operates, through further reading, pen-and--paper worked examples, and developing my own pseudocode. Cannon's original publication [1] describes the algorithm in the somewhat limited terms of his era, sharing one value to each computational cell, and then calculating in a distributed fashion. As we discussed, contemporary implementations multiply cache-sized sub-matrices rather than individual values. Towards this end, I have refactored my initial naive sequential solution significantly for modularity, using function calls to improve the readability and flexibility of nested naive matrix multiplication subprocedures, matrix printing calls, etc.

As part of this refactor, I have significantly improved the robustness of my naive implementation, modifying the approach to random dataset generation to ensure that the widest possible range of values is used while guaranteeing that products never overflow their allocated space, even as the data reaches large scales. Handling this programmatically made data size selection straightforward, as discussed in the next section.

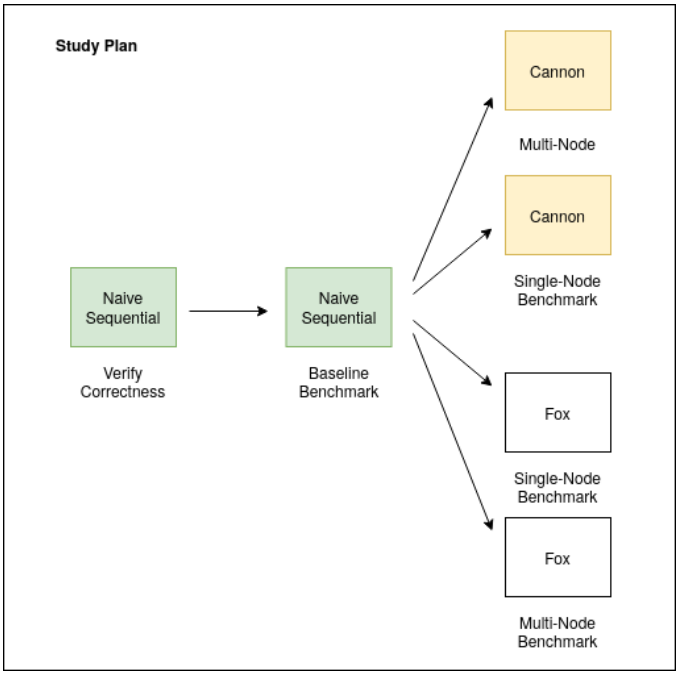

Figure 1: Programming deliverables over time from left to right. Green indicates completion, yellow some progress.

Finally, I have developed the build, run, and data summary tooling necessary to efficiently support the data collection and analysis components of this study. Makefiles, jobscripts, queuing scripts, a script for timing data aggregation, and a script for calculation and reporting of per-processor-count mean timings will make the downstream work

straightforward. Though I failed to take this work into account in the proposal, I feel strongly that it has been time well spent.

II. Challenges Addressed To Date

By profiling elapsed run times for the naive sequential algorithm across variously-sized randomly-generated square matrices, I was able to select what I hope will be a useful data size for testing my implementations across between one and eight nodes. I have selected square positive integer matrices with N=4096 values per side. A single skylake core is able to perform the computation in roughly 37 seconds. Even under perfect scaling, this would leave us with tenth-of-a-second run times when p=256. I have selected integer values primarily for ease of use - both large and small-scale validation are easier without floating-point error to contend with. If working with floating-point values is preferred, it should be straightforward to adjust.

III. Remaining Ideas to be Addressed

Untracked in the project proposal, it will also be necessary for me to select a cache-size appropriate scale for our small sub-matrix multiplications. This will require experimentation, which I plan to approach first with the naive sequential implementation, as it provides a simpler, more controlled system for testing. I will run matrices at various scales, using timings and cache-miss analysis with `perf` to select an appropriate size.

I will attempt to address questions about the impact of non-exclusive runs on job timing by using the `srun --begin` flag. The sbatch queuing script I have developed can be easily modified to spawn jobs on both available architectures at 30-minute intervals over two working days. This should yield 96 samples, spanning working and non-working hours, and will allow us to compare timings both in aggregate and as best- and worst-case scenarios.

At this time, I think it is unlikely that I will be able to squeeze in an investigation of heterogenous-architecture job execution. Though I have a pretty clear picture of the tooling and approach that would be required, the semester is drawing rapidly to a close.

IV. Preliminary Results

As discussed above, I've made far less progress than I had planned to date. Though the work I've done is good-quality and will reduce friction downstream, I still have a lot of code to write, and jobs to queue.

Below are the baseline sequential benchmarks discussed above. These timings were generated by multiplying a pair of randomly-generated 4096 x 4096 point integer matrices on a single core, over ten iterations. This benchmark is a good starting point for future comparisons, but obviously only a starting point.

MEAN	37.431081
MEDIAN	37.441884
SD	0.092053

Figure 2: Baseline timing in seconds of naive sequential algorithm run against a randomly-generated matrix of 4096 integer values, and timed with MPI_Wtime.

The codebase is in a private repository for academic integrity reasons, but I would be happy to share access if that will help you in considering my work so far. Thank you for your time and your patience.

V. Milestone Checklist

- ~~Naive sequential matrix multiplication in C (3/17/21)~~
- Cannon's algorithm implementaiton with OpenMPI (4/7/21)
- Single-node timings of Cannon's algorithm (4/9/21)
- Multi-node timings of Cannon (4/11/2021)
- ~~Preliminary report (4/14/21)~~
- Fox's algorithm with OpenMPI (4/17/2021)
- Single- and multi-node timings of Fox's algorithm (4/19/2021)
- Non-exclusive timings of both algorithms (4/25/21)
- Final report (4/30/21)

References

[1] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," phd, Montana State University, USA, 1969.

[2] G. C. Fox, S. W. Otto, and A. J. G. Hey, "Matrix algorithms on a hypercube I: Matrix multiplication", Parallel Computing, vol. 4, pp. 17-31. 1987. 4.

[3] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, Solving Problems on Concurrent Processors. vol. 1, Prentice Hall, 1988.