Final Report: Performance Benchmarking of Cannon's Distributed Matrix Multiplication Algorithm

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

***Abstract* — Cannon's algorithm (AKA "the 2D algorithm", or "roll-roll-compute") is a well-known fixed-storage-cost algorithm for computing matrix-matrix multiplication, a fundamental operation for many data-intensive computational processes. It is designed for distributed computing applications on 2D torus topologies, and improves on the performance of naive implementations through reduction in communication overhead.**

Here, I provide a two-fold evaluation of the performance of Cannon's algorithm, considering single-node performance and scalability, and multi-node scalability, benchmarked in controlled conditions.

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

I. Introduction

As scientific and business data continue to increase in scale, the computational performance and scalability of core linear algebra operations become increasingly important. The algorithms by Cannon [1] and Fox [2][3] are classics of distributed-memory computing, each providing a unique approach to reducing communications overhead in distributed dense matrix-matrix multiplication. They offer fixed memory costs and reasonable performance improvements over naive implementations while remaining relatively approachable for developers.

This work initially proposed to implement, analyze, and compare the performance of both algorithms, in both controlled and uncontrolled environments. Despite putting over 100 hours of work into it, it was not until the morning of 4/29 that I was able to complete a working implementation of Cannon's algorithm. These challenges will be addressed in detail later, and resulted in my dropping Fox's algorithm, and scaling back the analysis to controlled testing only.

I present a performance and scalability analysis of Cannon's algorithm, implemented with OpenMPI, and tested against a naive serial implementation on the new AMD nodes of NAU's Monsoon HPC cluster.

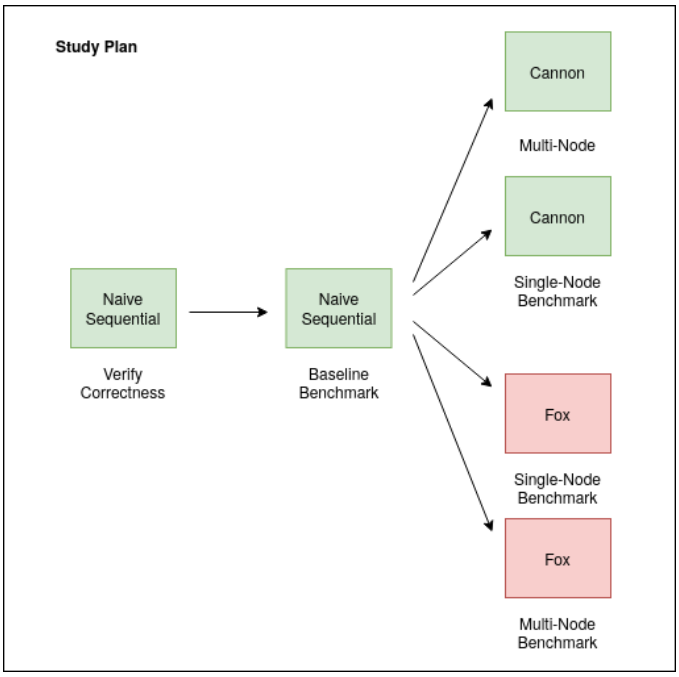


Figure 1: Programming deliverables over time from left to right. Green indicates completion, red indicates not attempted.

II. Background

Cannon's algorithm was initially published in the PhD dissertation of Lynn Eliot Cannon, in 1969. While it is well known, it has been surpassed by more contemporary algorithms in both theoretical and practical contexts. A 2021

publication [4] based on the Coppersmith-Winograd algorithm [5] sets a new best bound for theoretical computational complexity, but in-practice implementations tend not to use these approaches, because constant runtime factors make them impractical in most contexts.

Python, R, and other popular contemporary programming languages rely heavily on C and Fortran libraries (usually BLAS, LAPACK, and their derivatives) for efficient linear algebra computation. During the 1990s, attempts to provide library-quality distributed matrix multiplication were often focused on derivatives of Fox's algorithm [6]. Fox, like Cannon, performs poorly with very non-square data and, assumes a square 2D array of processors are serious limitations. Contemporary efforts have produced approaches that mitigate these issues [7], but there remains value to the developer in exploring foundational approaches.

III. Approach

I began development by building an easily implemented, verified, and benchmarked serial naive algorithm for matrix multiplication. After checking its correctness on toy data against Python's `numpy.matmul`, I had a usable platform for developing the tools I would need to support distributed algorithm development. Random, distributed data set generation, sequential printing of distributed matrices, and programmatic approaches to limit the size of randomly-generated values to prevent data type overflow.

While doing this, I built basic tools to automate compile, run, validation, data aggregation, and summarization, using python, BASH, GNUMake, and Slurm jobscripts. These tools were tremendously helpful, dramatically reducing the friction of developing, testing, and benchmarking code.

I then spent roughly three weeks trying and failing to implement Cannon's algorithm correctly, due to a fundamental misunderstanding of the algorithm's generalization to distributed systems. In Cannon's original publication, the worked example operates on single matrices, rather than distributed submatrices. Row *i* of matrix A is shifting left by *i columns*, and column j of matrix B is shifted up by *j rows*. My initial work carried this paradigm through to the distributed submatrices, shifting each submatrix on a *per-value* scale, rather than sending entire submatrices from rank to rank intact. This logic was complex and fussy to build, but works properly.

Not surprisingly, this code fails to generate correct results. Interestingly, however, it correctly performs matrix-matrix multiplication on a number of different configurations of small-scale matrices, when naive matrix multiplication of submatrices is replaced with the Hadamard Product (scalar-scalar multiplication, mentioned in passing in relation to matrix multiplication by Nielsen [8]). I tested this successfully against 4, 9, and 16-core torus topologies, with matrices up to 6x6. As matrix size scales up, however, this approach fails, yielding global sums of the same order of magnitude that would be expected, but always slightly too small.

This led me to believe there was an error in my communications scheme, whereby I was losing or mis-placing updates very intermittently. After extensive debugging, refactoring, and hair-pulling, I decided to re-implement Cannon's algorithm entirely, using MPI's topology-aware communicators and their associated tools. The goal was to retain the semantics of the algorithm while reducing the amount of developer-specified behavior as much as was possible.

While considering a questionable-looking implementation of Cannon's algorithm that uses a similar approach [9], it struck me that most implementations I had seen were using 1D arrays to represent submatrices, but lacked any of the complex indexing I had been performing. I hand-worked some examples on paper, and confirmed that naive multiplication of submatrices works, so long as the scale at which submatrices are shifted between ranks matches the scale at which you multiply those submatrices. With this insight, and the tooling I had developed previously, I was able to build and validate Cannon's algorithm on 4096x4096 integer matrices in a few hours of concerted work.

Benchmarking benefited significantly from access to Monsoon's new 64-core AMD nodes. Because Cannon's algorithm expects a square number of cores, and because my implementation makes the additional assumption that the submatrices should be square (this only for lack of time), 64-core nodes allowed for a much more in-depth comparison of run times across 1, 2, 4, and 8 compute nodes.

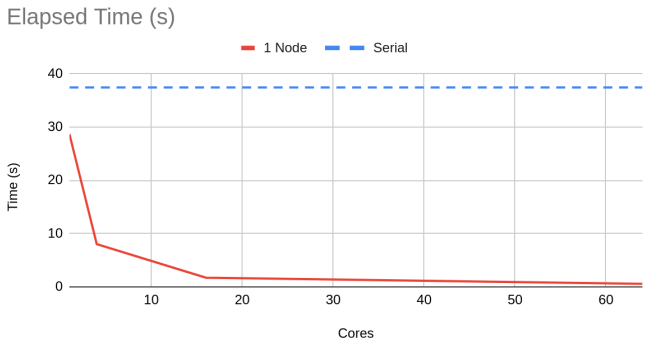
IV. Results

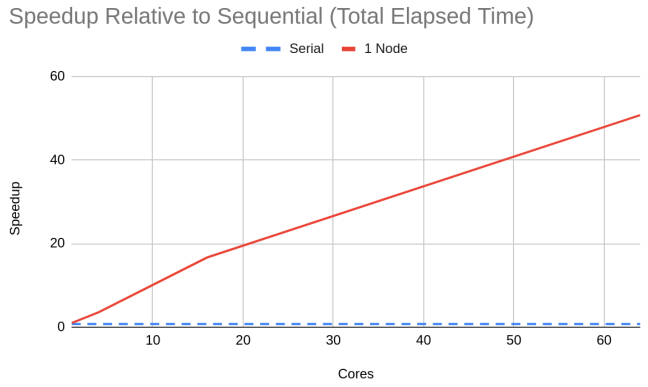
Below are summary benchmarks for the naive serial matrix multiplication algorithm. These (and all other) timings were generated by multiplying a pair of randomly-generated 4096 x 4096 point integer matrices on a single core, over ten iterations. The small standard deviation led me to believe that high levels of variability should not be expected from this timing data. Mean values calculated on ten exclusive runs will be used from this point on.

MEAN	37.431081
MEDIAN	37.441884
SD	0.092053

T1: Baseline timing in seconds of naive sequential algorithm run against a randomly-generated matrix of 4096 integer values, and timed with MPI_Wtime.

Cannon's algorithm shows good speedup on a single node, with parallel efficiencies between 0.79 and 1. as the algorithm scales up to 64 cores.



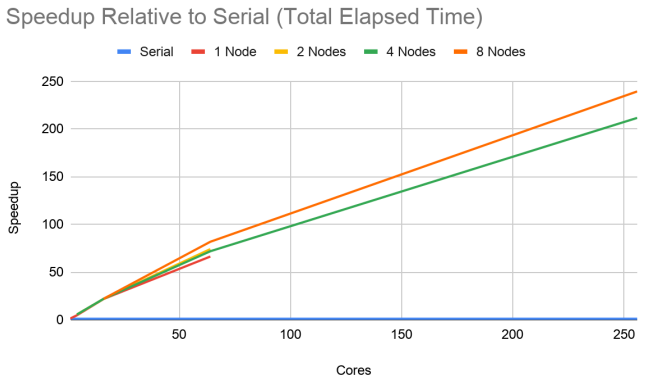


In an unfortunate oversight, I failed to re-run the serial benchmark code against the new AMD nodes before constructing the tables and graphs seen here. I expect that the naive serial algorithm would outperform Cannon's algorithm on 1 rank, through simpler logic, and by incurring no "communication" costs. Because varying numbers of nodes allow varying numbers of processor counts, we'll use this benchmark as a semantically unmeaningful point of reference.

	1 Node	2 Nodes	4 Nodes	8 Nodes
1	28.657	-	-	-
4	8.0114	6.7631	6.806	-
16	1.7183	1.7105	1.7046	1.7002
64	0.5649	0.5056	0.5213	0.4579
256	-	-	0.1769	0.1564

T2: Total Response Time of Cannon's Algorithm (s)

Because it is not possible for us to measure the speed of Cannon's algorithm on one core but two or more nodes, this external point of reference allows us to consider values we could not otherwise compute. Speedup data show a mild but relatively consistent decay in performance as the number of cores utilized per node increases. Simultaneously, we see a general improvement in performance as the number of nodes increases.



The tables below use Cannon's algorithm run on a single core on a single node as the reference value against which speedup is calculated, making it easier for us to draw meaningful overall conclusions about performance improvements and parallel efficiency.

Speedup	1 Node	2 Nodes	4 Nodes	8 Nodes
1	1	-	-	-
4	3.577	4.2373	4.2105	-
16	16.6775	16.7536	16.8116	16.8551
64	50.7293	56.6792	54.9722	62.5835
256	-	-	161.9955	183.2289

T3: Speedup relative to Cannon's algorithm run on one core on one node. Green indicates the reference value.

The algorithm scales remarkably well, maintaining strong speedup and high levels of parallel efficiency up to 256 cores. Values greater than one here are artifacts of our using a single reference value, but because that reference value allows for apples-to-apples comparison, I think we can draw useful conclusions.

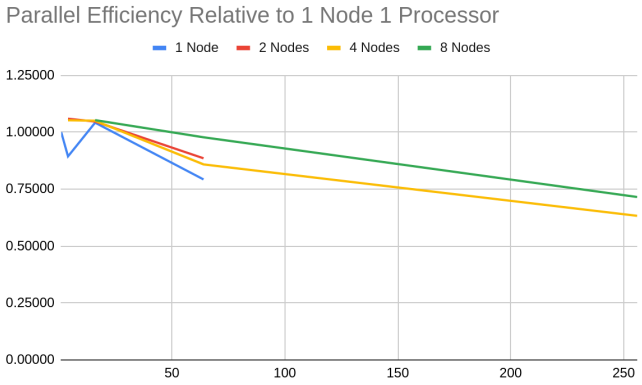
P. Effic	1 Node	2 Nodes	4 Nodes	8 Nodes
1	1.00000	-	-	-
4	0.89425	1.05933	1.05263	-
16	1.04234	1.04710	1.05073	1.05344
64	0.79265	0.88561	0.85894	0.97787
256	-	-	0.63279	0.71574

T4: Parallel Efficiency relative to Cannon's algorithm run on one core on one node.

V. Discussion and Conclusions

This brief analysis yielded two interesting results: the performance decay we see as cores per node increases, and the improvement in performance as the node count increases.

Many algorithms struggle to scale effectively across multiple nodes, penalized by high communication costs. Here, though, we have a strong candidate. With $O(n^{2.373 - 3})$ complexity, the computational costs are high enough that strong scalability is generally possible. It seems that Cannon's communication-avoiding approach is successful enough to expose a secondary bottleneck here, in memory access. This, in turn, can be alleviated by distributing our matrix over more nodes.



The final, benchmarked code for this algorithm (in the `topology` directory) is clear, effective, and far simpler than my initial approach, but leaves much room for improvement and extension. First and foremost, though it uses loop reordering to reduce the cost of cache misses, it fails to take full advantage of the speed of cache memory. The block size that each submatrix multiplies should be reduced to fit into L3 cache in order to do so. More practically, the algorithm does not accommodate non-square numbers of processors intelligently, opting to fail loudly rather than find the next smallest square. The benchmarking data is also quite poor, using sequential values rather than randomly-generated data. Had I had more time, it would have been trivial to port the random data generation tools I built over to the new implementation, but that, unfortunately, didn't work out.

The most useful takeaway from this work was the time I spent exploring MPI's direct support for abstractions that simplify common communication protocols. MPI_Cart_shift() makes finding neighboring rank numbers in a Cartesian topology trivial. MPI_Sendrecv_replace dramatically improves the readability of blocking communication. Some performance improvement could be squeezed out of using non-blocking communications and covering them with matrix multiplication compute costs, but this approach makes for lovely code, and that's a project for next time.

Thanks again for a great class, and for your patience with my work on this project. I can honestly say that, while this wasn't the hardest work I've done here at NAU, it was some of the most time-costly and emotionally taxing for all the missteps along the way. I learned a lot.

VI. 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.

[4] J. Alman and V. V. Williams, "A Refined Laser Method and Faster Matrix Multiplication," arXiv:2010.05846 [cs, math], Oct. 2020, Accessed: Apr. 29, 2021. [Online]. Available: http://arxiv.org/abs/2010.05846.

[5] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," Journal of Symbolic Computation, vol. 9, no. 3, pp. 251–280, Mar. 1990, doi: 10.1016/S0747-7171(08)80013-2.

[6] S. Huss-Lederman, E. M. Jacobson, and A. Tsao, "Comparison of scalable parallel matrix multiplication libraries," in Proceedings of Scalable Parallel Libraries Conference, Mississippi State, MS, USA, 1994, pp. 142–149, doi: 10.1109/SPLC.1993.365573.

[7] M. D. Schatz, R. A. van de Geijn, and J. Poulson, "Parallel Matrix Multiplication: A Systematic Journey," SIAM J. Sci. Comput., vol. 38, no. 6, pp. C748–C781, Jan. 2016, doi: 10.1137/140993478.

[8] F. Nielsen, "Parallel Linear Algebra," 2016, pp. 154-158

[9] "Cannon's algorithm for distributed matrix multiplication," OpenGenus IQ: Learn Computer Science, Oct. 13, 2018. https://iq.opengenus.org/cannon-algorithm-distributed-matrix-multiplication/ (accessed Apr. 29, 2021).