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

***Abstract*—Cannon's algorithm, (AKA roll-roll-compute) and Fox's algorithm (variously described as broadcast-multiply-roll, or broadcast-roll-compute) are well-known fixed-storage-cost algorithms for computing matrix-matrix multiplication, a fundamental operation for many data-intensive computational processes. Both are designed for distributed-memory computing applications, and both can be applied to 2d torus topologies (though Fox's algorithm was initially designed for use on hypercube topologies). The relative similarity of these two algorithms makes them relatively controlled test subjects for research into comparative performance at scale.**

I propose a three-fold evaluation of the performance of these algorithms, considering single-node performance and scalability, multi-node scalability, and "in the wild" non-exclusive performance on shared machines, including targeted investigation of runtimes on heterogenous architectures if time permits.

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

I. Introduction

I will implement a naive sequential matrix multiplication algorithm, and the Cannon [1] and Fox [2][3] algorithms for distributed-memory matrix multiplication. All programs will be written in C, using MPI for parallelism, and will be compiled with GCC's 03-level optimizations. The naive sequential implementation will be used for data set development, validation of correctness, and for baseline performance benchmarking.

I will test the Cannon and Fox implementations against a randomly-generated square matrix of floating-point values, collecting and analyzing runtime and scalability data. I will measure performance using speedup and simple parallel efficiency for p = 1:max_cores on a single dedicated compute node, and again on two, four, six, and eight-node configurations, so long as parallel efficiency remains reasonably good. I am generally more interested in applied, "daily use" performance than in rarefied benchmarks, so will produce architecture-specific, non-exclusive timings over multiple time points, in order to provide some insight into performance loss in the field.

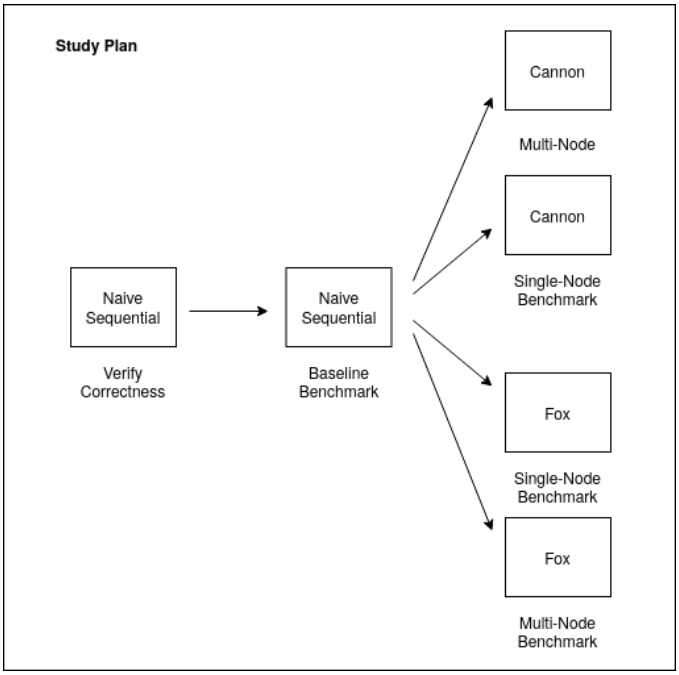

Figure 1: Programming deliverables over time from left to right.

Monsoon's SLURM job scheduler offers support for user-specified heterogeneous jobs. Unless handled proactively, architectural heterogeneity seems likely to contribute to load imbalance and negatively impact performance. Should time allow, I would like to take advantage of this tooling to investigate the impacts of multi-architecture runs on performance.

II. Anticipated Challenges

Selecting an appropriate data size at which to work will require some playtesting, and probably compromise. A data set that runs quickly on a single core will likely not be useful when scaled up to 128 or 256 cores, and test data selected for multi-node runs could take a very long time to run sequentially.

There is no reasonable way to control the amount and character of system traffic, so any non-exclusive timings will be subject to the vagaries of chance. Previous tinkering I've done with timing non-exclusive jobs has resulted in negligible impact on runtime. Jobs also launched immediately, leading me to believe that system traffic may have been unusually low at the time. Though this cannot be strictly controlled for, I will have to organize my data collection to ensure that I collect a reasonably representative sample of timings.

If I have time to test on heterogenous nodes, working around slurm's heterogeneity support will be a challenge. Though heterogeneous job scheduling is possible, jobs must be explicitly split into parts. Those parts may be run concurrently on a shared MPI_COMM_WORLD. The algorithms (or at least the data distribution) will require modification in order to make this work.

III. Proposed Solution Ideas

Data size selection should be a straightforward problem to solve. Matrix multiplication tests in the literature frequently use square matrices with between N=1000 and N=4000. Generating data randomly (with a fixed seed) will allow me the flexibility scale up or down to produce useful and reasonable run times.

As discussed above, the most important aspect of "in the field" data collection is to sample as representatively as possible. I will use the `srun --begin` flag to queue jobs for execution at regular intervals over the course of two working days. I will consider both mean and median timings when interpreting the results.

I am hopeful that accommodating controlled heterogenous-architecture job execution will not be too challenging, consisting only of splitting the data to be computed on across different node generations. If that proves not to be the case, I may be able to consider this question informally, by running non-exclusive jobs without any architecture specification, and capturing data on which nodes were used.

IV. Preliminary Progress

This semester's work, study, and teaching load has kept me from making the progress I would like towards developing preliminary results. This issue has only been compounded by the failure of my previous proposal, due to challenges with running Chapel code on Monsoon.

To date, I have written a naive sequential implementation of matrix generation, multiplication, and printing. This implementation uses loop re-ordering to produce best-possible cache-reuse for the naive algorithm, correctly performs the multiplication, and takes 47.715 seconds to multiply two randomly-generated N=2048 integer matrices, on a single core on my local machine.

In addition, I have located and reviewed primary and secondary source materials describing the algorithms and their implementation. This research has raised some preliminary implementation questions I will need to answer, but put me in a reasonable position to finish the work in time.

The first phase in the project proper will involve transforming this algorithm for use with larger-scale matrices on distributed-memory systems. The product of that work will undergo a more rigorous verification and benchmarking, and will serve as the "naive" control in this performance experiment. Manual optimization with Cannon's algorithm and automated optimization will constitute the two test cases.

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.