Program & Programmer Efficiency in Chapel and OpenMPI

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

***Abstract*—Software developers and data scientists must often balance development costs against performance costs, and highly optimized algorithms often require high levels of computational, mathematical, hardware-domain, and subject-domain expertise to implement. General-purpose systems programming languages like C and C++ allow expert developers to write highly performant distributed-memory code, but many developers do not have the requisite time or background to produce optimal results. Optimized MPI programs are often able to outperform programs developed on the Partitioned Global Address Space (PGAS) memory model, but the Chapel Language's first-class support for distributed memory computing concepts may offer tools that help offset the above challenges for developers.**

I propose an informal cost-benefit analysis of migration from C/openMPI-focused programming to programming workflows centered on the Chapel Language's builtin locality optimization tooling. Kayraklioglu et al [1]'s tooling for Chapel promises automated code optimization based on elastic net regressors, with near-MPI results for many common benchmarks. In this study, I will collect high-level "startup time" and development time data alongside performance data on naive and moderately locality-optimized matrix multiplication algorithms, and explore the possible value that Chapel's automatic optimizations present for the "working developer" or data scientist.

***Keywords*—distributed-memory computing, PGAS, automated optimization, Chapel, MPI, matrix multiplication, Cannon's Algorithm, performance benchmarks, applied computing**

I. Introduction

For this study, I will implement both naive and locality-optimized distributed-memory matrix multiplication algorithms (likely Cannon) [2] in C/MPI, and a naive distributed-memory matrix multiplication algorithm in the Chapel Language. I will compile the former with GCC's `O3`-level optimizations, and the latter with Chapel's `–fast` optimizations, alongside the LAPPS framework's auto-generated locality optimizations. I will collect high-level timing data on setup/development time for all implementations, tracking development environment setup costs, actual development time, and "learning" overhead. I expect significantly more learning overhead with Chapel, as the language and its memory model are both new to me (and most developers), but quantifying the time cost will be useful.

I will test these implementations against square matrices, collecting and analyzing runtime and scalability data. I will measure performance using speedup and simple parallel efficiency for `p = 1:max_cores` on a dedicated compute node. If time allows, I will expand the study to consider differences in how well the two approaches handle intra-node communications costs, by testing on a four-node configuration.

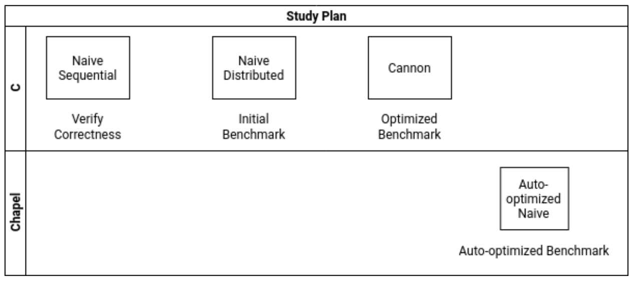

Figure 1: Programming deliverables by time from left to right.

II. Anticipated Challenges

The first and primary challenge I expect to encounter is my own lack of experience with the PGAS model and with Chapel's first-class support for distributed programming. Chapel's formal models of `Domain` and `Locale` are unfamiliar, and the language syntax as a whole looks deceptively simple. Though `forall` and `coforall` loops seem like "better" abstractions for parallel problems, I expect that translating these matrix multiplication algorithms from a paradigm I understand into a paradigm I do not will require significant learning upfront.

A related potential challenge lies in running distributed chapel code on the Monsoon cluster. Though our cluster uses a relatively common software stack, and Chapel's documentation seems to indicate that it plays well with Slurm, RHEL, etc, I expect I will encounter trouble along the way.

Generating or sourcing test data is unlikely to be a major problem, but it is one component of the study I have not had time to consider in depth yet. My preference would be to generate test data randomly, but further investigation will be required to ensure that comparable data can be generated using both languages. There may be value in working with an established data set for benchmarking purposes.

Algorithmically, I am unsure what kind of performance to expect from Cannon's algorithm, when running on one node versus multiple nodes. Based on a first reading, scaling Cannon's algorithm to distributed systems can be handled by scaling up the algorithm's communications optimizations from inter-core to inter-node communication. Given the higher costs of inter-node communication, this is probably beneficial overall. A more optimization-focused implementation would probably consider both intra- and inter-node communication. Given the time constraints, I do not expect to solve this problem in C/MPI.

III. Proposed Solution Ideas

The only way to solve the first "problem" above, that in which I will have to map algorithms from a known memory model to one I don't yet understand, is to learn the new model (at least roughly). Exposure to PGAS development is one of my primary goals here, and I plan to approach the problem incrementally. I will playtest solutions to smaller sub-problems and compose working results into a functioning implementation. In addition to my benchmarking data set, I will develop and run smaller data sets so that I can quickly verify the correctness of my solutions, and will use a reference implementation of matrix multiplication to confirm. (CBLAS, OPENBLAS, or one of the Python libraries built on these).

Should minor challenges come up with Monsoon/Chapel compatibility, the documentation seems to be pretty comprehensive. If that fails me I will raise a ticket with the monsoon team. The key here is going to be starting early. In the unfortunate event of a prohibitive incompatibility, I may have to reconsider the project's goals and scope. It may be possible to move to a temporary server over which I have administrative control, or I may have to focus on developing additional algorithms with MPI, or further optimizing the C implementations I already plan to produce.

I plan to address questions of data translatability with some basic statistical testing. I will generate random data using fixed seeds under both platforms and visually compare. Barring clearly identical results, I will visually compare histograms of the produced data, and summary statistics based on iterations of data production under varying seeds.

As for the final problem discussed above, in which the development costs of a locality-optimized MPI solution that can run on multiple nodes proves prohibitive, I am hopeful that Chapel's Locales will allow the auto-optimized solution to outperform my "stock" Cannon's algorithm for exactly this reason. If this turns out to be the case, I will read it not as a "problem" with my C implementation, but rather as data in support of the idea that an HPC-first language framework provides concrete value for low-developer-cost performance.

IV. Preliminary Progress

Frankly, this semester's work, study, and teaching load has kept me from making the progress I would like towards developing preliminary results. To date, I have successfully built Chapel on Monsoon three times, with gradually increasing levels of functionality, compiled and run basic executables locally, and spent enough time with the paper and the Chapel documentation to convince myself that this project is workable.

Total developer time to date has included roughly 30 minutes of C programming, and roughly five hours incrementally building and rebuilding Chapel. This approach is recommended by their documentation, presumably to give new users a high probability of success with the installation process, and provide an accessible "playground" for first experiencing the language locally. Because this isn't made clear at the top of the documentation, some unnecessary friction was introduced.

In addition to this preliminary setup work, 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.

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)~~
- Naive parallel matrix multiplication with OpenMPI (3/21/21)
- Initial multi-core and multi-node playtesting in Chapel (3/28/21)
- Data generation/loading in Chapel - multi-core and multi-node (4/6/2020)
- Preliminary report (4/7/21)
- Cannon's algorithm with OpenMPI (4/11/2021)
- Toy parallelism worked examples with Chapel for paper presentation (4/19/2021)
- Multi-core matrix multiply in Chapel (4/18/21)
- Benchmarking, troubleshooting, possible further optimizations (4/25/21)
- Final report (4/30/21)

References

[1] E. Kayraklioglu, E. Favry, and T. El-Ghazawi, "A Machine-Learning-Based Framework for Productive Locality Exploitation," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 6, pp. 1409–1424, Jun. 2021, doi: 10.1109/TPDS.2021.3051348.

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