An Evaluation of Regent and Legion in the Parallel Programming Landscape

A CS242 Project Report By

Sierra Kaplan-Nelson
sierrakn@cs.stanford.edu

I. SUMMARY

I implemented two programs in Regent on Legion and parts of both in MPI and Charm++ to evaluate the performance and programmer efficiency on different parallel programming models for distributed architectures. The majority of my work was analyzing simulation code on all three systems and providing recommendations for the ideal use cases of each.

II. BACKGROUND

Legion is a fairly new runtime system and Regent is an even newer programming model that attempt to address the difficulty of writing high performance applications on distributed architectures with minimal complexity handling data partitioning and dependencies [1]. Regent is a dynamic language based on Lua optimized for high productivity that runs on the Legion runtime. [2] Very little research has been done so far analyzing the optimal use case for Regent and Legion and comparing the same program written in Regent with one written in traditional MPI or Charm++. I was interested in how writing implicit parallel programs differed in Regent vs. competing systems and whether it was able to achieve comparable or better performance with less complexity. Legion was created in 2012 and Regent was first developed in 2015, making them far newer than the more established systems I compared them to.

MPI (Message Passing Interface) has long been the canonical standard for writing parallel programs on distributed architectures. However, MPI focuses on describing parallelism alone and does not handle imbalanced loads or provide abstractions for describing data. It was created in the early 1990’s.

Charm++ is an object oriented parallel computing language built on C++ and developed in the Parallel Programming Laboratory at the University of Illinois. It has been around since the late 1980’s, although has existed in something close to its current state only for the last fifteen years. Charm++ allows programmers to describe their data in terms of objects and provides automatic load rebalancing by mapping objects to processors. [3]

I avoided comparing Legion with implicitly parallel languages that execute on shared memory such as CUDA or OpenMP. Writing parallel programs for distributed architectures is increasingly important as larger super computer systems are built, and the movement of data largely impacts performance on such systems. Parallel languages on shared memory do not optimize for avoiding data movement and therefore were not comparable. I also excluded Spark, which executes on clusters with distributed shared memory. Spark was left out because it is not nearly as performant as C++ based systems. [4]

III. APPROACH

Legion is currently being developed in collaboration with the Stanford Soleil-X project, a multi physics solver designed to conduct high fidelity simulations involving particles, fluids, and radiation. 1 I wrote one of the radiation solvers in Regent using the Discrete Ordinates Method 2 and profiled its performance vs. an implementation in MPI 3. I then wrote parts of the implementation in Charm++ to compare the method of data description and evaluate programmer efficiency.

The Discrete Ordinates Method involves doing parallel sweeps across a grid of values from many angles, updating the face and cell center values. Sweeps that do not affect the same face values can be parallelized with the final cell center values computed at the end. In addition, sweep values along a diagonal can be computed in parallel.

Fig. 1. A visual of parallel radiation sweeps in two dimensions. Calculations on nodes of the same color can also be performed in parallel.

1http://meetings.aps.org/Meeting/DFD17/Session/M30.5
2https://github.com/stanfordhpccenter/soleil-x
3https://github.com/stanfordhpccenter/dnsorp/
In addition, I wrote a stencil program in Regent that detects the edges in an image, a problem which is embarrassingly parallel and far simpler than DOM. I adapted a similar stencil program from a tutorial on Charm++

IV. RESULTS

The Regent implementation of DOM showed strong scaling for the small number of nodes it was tested on, but a significant performance loss from the Legion runtime system. The MPI implementation had no such runtime slowdown and also showed strong scaling when there were an equal number of angles from each quadrant. I was unable to test the performance of the equivalent Charm++ implementation due to implementing a simpler version.

Fig. 2. A graph showing the strong scaling on a small number of nodes of DOM in Regent. The initially sluggish line between 1 and 2 nodes can be explained by the Legion runtime runtime overhead.

From a programmer efficiency perspective, Regent was clearly the simplest in the DOM case. In MPI, I had to manage message sends and receives for each angle at the beginning of every iteration. Managing the private and shared data partitions in Regent was very straightforward. It took me a factor of ten times longer to manage splitting the data and sending the correct shared section in MPI, despite basing it off of an existing implementation. However, I acknowledge if I had programmed in MPI for years the results may have been different.

In Charm++, describing the data was entirely different. While Legion and Regent describe data in terms of regions, Charm++ uses indexed collections called chare arrays and makes it simple to send messages to entire arrays simultaneously. Unlike in MPI but similar to Regent, these arrays are mapped to processors by the runtime system automatically. I took a straightforward approach of implementing the chare arrays in Charm++ using the same math used to create region partitions in Regent. While I did not finish the entire DOM implementation in Charm++, the experience of partitioning data was remarkably similar to Regent. However, dependencies between data are not detected automatically at runtime in Charm++, so ensuring the sweeps synchronized at certain points proved more difficult. I also was unable to create disjoint regions and partition trees as in Regent, so I had to copy the data explicitly before partitioning it.

Regent also had the advantage of being based on the simple scripting language Lua rather than C++. This improved my experience as a programmer immensely, but had the drawback of requiring a long compilation in order to be converted into the more efficient Legion implementation. A lot of custom functionality is also obscured in Regent, so if I had wanted to write a custom mapper between tasks and nodes I would have had to use the more primitive Legion API. Currently, the majority of code can be written in Regent but in order to really boost performance programmers still have to deal with bare C++ and Legion.

For the simple stencil program Charm++ and Regent were both much simpler to use than MPI. I actually thought Charm++ was easier to use than Regent to separate the data between private and shared regions and describe parallel tasks on each chunk. My different experience using Charm++ for this program could be because I am more familiar with DOM in Regent, but I think it also points to differences in Regent and Charm’s designs. Region was written to describe complex region trees and is overly complex when the data can be described very simply. Charm’s implementation is more straightforward.

V. CONCLUSION

Regent is a modern, ambitious project that succeeds at combining the productivity of a dynamic language like Lua with the performance of a C++ distributed runtime system. It is ideal for writing scalable, complex simulations on data that require heavy dependency analysis. It is a little too heavyweight for parallel programs that are embarrassingly parallel or do not require interesting data partitioning.

Charm++ provides many of the same benefits, including dynamic load rebalancing and describing data in terms of objects. It does not allow the creation of region trees or perform runtime dependency analysis. This lighter weight implementation is optimal for programs that do not require complex data relationships. I know it is being used for several large simulations currently, including a Quantum Chemistry application called OpenAtom. I would like to spend more time analyzing these simulations and how Charm++ is used in practice. I was unable to find if anything resembled the sweeps in DOM in this simulation. There is more of an explicit representation of data partitions than in MPI, but programmers are still responsible for the majority of describing relationships between data.

Currently Legion is less well tested and proven than Charm++, but I think Regent offers many advantages for building large scale simulations that Charm++ lacks. However, Charm++ seems a much better option than bare MPI for the majority of distributed programs.

\(^4\)http://charmplusplus.org/tutorial/Basic2DJacobi.html

\(^5\)https://en.wikipedia.org/wiki/OpenAtom
VI. Future Work

The Discrete Ordinates Method for a Radiation solver is an interesting problem, but has been fully optimized for many systems previously. What is truly interesting about the current Soleil-X project using Legion is that it is an integrated turbulence, particle, and radiation solver. I would like to compare the performance and programmer efficiency writing the entire simulation in Regent on Legion and in MPI and Charm++. The integrated simulation involves describing data in different ways for each of the simulation steps and therefore poses more interesting, unexplored problems for parallelism.

There is a lot of repetition in the DOM code describing eight different regions and sweeps. It would be interesting to leverage the meta-programming capabilities of Regent to simplify this code, which is impossible to do in C++ implementations.

In the past, several projects were interesting in using Legion but decided against it due to its complexity, including a Stanford video analysis project called Scanner \(^6\). An interesting project would be rewriting parallel projects like these in Regent and evaluating both the code complexity and the default mapper’s performance.

All three of the systems I analyzed are written in C++, a language extremely prone to bugs. Fascinating future work could include translating the core of one of these implementations into Rust or another safer systems language and analyzing the performance. It would be an unenviable task however, and the barrier to entry for new users would increase.

References


\(^6\)https://github.com/scanner-research/scanner