Capstone Term Paper
University of Washington Bothell

Evaluating Repast Simphony for Agent Based Geometric and Combinatoric Simulations

Maxwell Wenger
mdwenger@uw.edu

Supervised By
Dr. Munehiro Fukuda

December 17, 2020

Abstract

The purpose of my research project is to compare the performance and programmability of different tools that may be used for geometric and combinatoric simulations. Repast Simphony[1] is one of many products and paradigms we will be evaluating. The others products we are evaluating are as follows:

- MASS[2], an agent-based spacial simulation suite developed by our team at the Distributed Computing Laboratory at the University of Washington Bothell.
- JClik[3], a multithreading tool kit for Java. The evaluation for JCilk has been completed by Jonathan Acoltzi.
- IBM Aglets[4], which is a mobile agents platform for Java.

The performance of all of these tools will be evaluated on multiple axises. Statistics will be gathered about how each tool performs based on number of computing nodes, and the size of the simulation. The programmability will be evaluated on the amount of code, ratio of code to "boilerplate" code, and how fitting the paradigm was for the algorithm.

Although not all of platforms have been benchmarked and evaluated, an analysis can be done of Repast Simphony on performance and programmability. This paper will analyze Repast Simphony and discuss my progress on my capstone over the last quarter.

[1] https://repast.github.io/

[2] https://depts.washington.edu/dslab/MASS/

[3] http://supertech.csail.mit.edu/jCilkImp.html

[4] The original site is no longer online, but a mirror is being hosted at http://alumni.media.mit.edu/~stefanm/ibm/AgletsHomePage/index_new4.html

Contents

1 Quarter Progress Summary

This quarter I completed the benchmarking programs for Repast Simphony, and have

completed an analysis of the performance and programmability of those programs. I also broke away from the capstone project a bit this quarter and made some contributions to the Repast Simphony project and documentation myself.

1.1 Challenges Faced

I am very grateful for the experience of working on a research team. I have never worked on a team where I wasn't a part of or at least had a full understanding of what everyone was doing. This was an amazing experience in team work and dealing with information asymmetry in a team-environment. With that, there were some lessons I was able to take away from this experience.

I spent most of the quarter settling into the research team and understanding the project we were working on, and dealing with the Repast Simphony tool set. My biggest barrier to success this quarter was one of organization and communication. Being completely remote for this project limited easy communication to only a few sessions a week, where my exposure to the overall goals of the research project were limited, which lead me down a few dead-ends in my project.

The following is an itemized list of tasks I worked on that did not contribute to the research project:

- Reusable divide-and-concur classes to be used in closest pair of points. This was not used as I was under the impression we were using the same algorithm for each product, but instead we were to develop an algorithm that fit with the paradigm of the project.

- Point generator for closest pair of points. This was not used because we have a standard set of points that we are to use to benchmark our data on.

- Graph generator that generates graphs based on number of vertices and a percentage of how connected the graph will be. This was not used as we already have a class that generate the graph for us.

- Batch functionality to run batch Repast Simphony simulations. I thought we were to run Repast Simphony as a distributed simulation across many compute nodes, and the only way to do that in repast was with batch simulations. After going down the rabbit hole hadoop, repast simphony, and batch simulations, I was corrected in my understanding and realized that we are evaluating Repast Simphony on a single compute node and its multithreaded performance.

- Data collected on my personal machine. This was not nearly as large of a time sync as the others, but I collected all of my data on my personal machine rather than the same machine that we have been using for the other performance evaluations.

Even though there were many completed tasks that ultimately did not contribute to

the overall project, I still am personally better off because of them, as some of the greatest learning moments of this capstone project happened while working on those tasks. Moving forward, I need to take the following steps to ensure that I am making the best use of my time and contributing as much as I can be to the group:

- Understand the data we want to collect. I will do this by building the axis of the graphs and data we want to produce before starting the project, so I can verify that the way we are evaluating each project is how I think we are evaluating each project. This would have remedied the issue where I thought we were evaluating Repast Simphony on its multi-node performance.
- Design the test plan before designing the benchmark. I need to know what data we are feeding into each benchmark so I can write to that specification. This would have remedied the issue where I wrote my own test-case generation classes.
- Verify my general approach to the implementation early. By communicating and validating my approach early on, it would give me the opportunity to adjust the adjust my approach before making too much progress. This would have remedied the issue where I was trying to write a divide and concur algorithm for repast simphony.

I believe by taking those steps, I can better set myself up for success and contribute more meaningfully to the research team as a whole.

2 Algorithmic Approach

2.1 Closest Pair of Points

The closest pair of points problem is when you have a 2D plane of points, and you must determine the two points that have the smallest distance of all possible pair of points.

This problem is usually solved with a divide and concur algorithm, where the plane is recursively split in half until only two points remain per partition, then the board searches on either side of the partition in a range that is equal to or smaller than the smallest point found. This approach is possible in conventional programming, but breaks many rules in agent based modeling. Notably, the entire plane and all of the points must be known by a single entity.

In agent based modeling, you must choose a solution that can be carried about by independent and autonomous agents. The approach I took was to have the agents spawn in a near-radial pattern around each point, and the first moment another point's agent made contact with another point, we knew we have found the closest point. This algorithm was previously used to evaluate MASS[1].

2.2 Triangle Counter

The triangle counting problem is when you have a graph, and you wish to count all of the triangles that exist in the graph. A triangle

is defined as an instance where a vertex is exactly 3 hops from itself.

This problem was made quite well for an agent based approach, as each agent could be assigned a vertex, and the agent can performs a BFS starting at its assigned vertex, looking for a path back to its vertex at three hops. The challenge with this algorithm is a triangle must only be counted once. Taking the naïve approach, each triangle would be counted three times, as each agent on a triangle would BFS and find itself. The fix for this is to assign priorities to each vertex, and an agent is only allowed to travel to a vertex that has a lower priority than its home vertex. This ensures every triangle is only counted once.

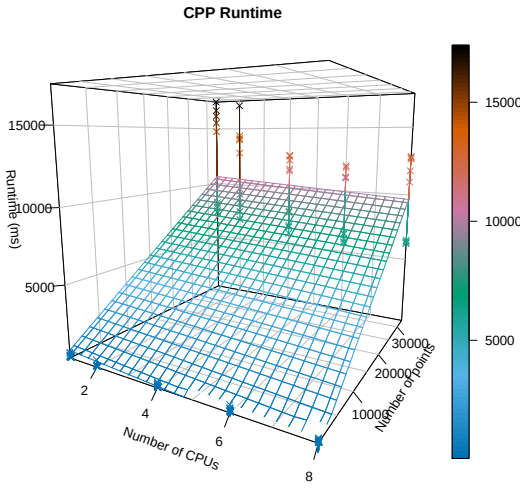


Figure 1: 3D Plot of the performance of CPP based on 1 to 8 available cores and 128, 256, and 3000 points in the simulation.

3 Results

The results section is broken up into two parts: performance and programmability. Each of those sections will discuss the different results from each criteria it was evaluated on for both the CPP and Triangle Counter programs.

3.1 Performance

When running these tests, I was extremely surprised at the result. I was convinced that adding CPUs would always increase performance. What I found that there are some types of programs benefit more from having more cores than other programs do.

3.1.1 Closest Pair of Points

The results from the CPP performance evaluation, as seen in Figure 1, were not surprising to me. Although I did expect a stronger correlation between number of cores and runtime, there was still a somewhat significant correlation. It seems that Repast Simphony does make use of the extra cores available for these geometric simulations. The unique feature of this simulation is that the number of agents in the simulation scale exponentially for each simulation tick, as the agents spawn in an increasingly larger circle each tick. So this simulation ended up with many agents in the simulation. It seems that the real multithreaded performance benefits happen when there are many agents to keep track of.

3.1.2 Triangle Counter

The results from the triangle counter performance evaluation as seen in Figure 2, on the other hand, were very surprising to me. I expected to see at least some correlation between performance and available cores. My suspension is that this is because this simulation had a lot of ticks, and very few agents to keep track of. This simulation only had n agents, and many of them didn't have any work to do. Yet, the simulation ran for hundreds, and sometimes thousands of ticks. It seems that multithreaded performance is more of a benefit when you have many agents, but it seems to have very little impact on simulations that have few agents, but a lot of ticks. This would make sense as each tick would be a fixed and sunk cost, that can't be done asynchronously, whereas agents can be parallelized.

3.1.3 Performance Comparison

As discussed, CPP has many agents with heavier workloads and few ticks, while triangle counter has many ticks, but few agents with light workloads. Because of this, we seemed to see a greater benefit of performance by increasing the cores in CPP than we did with triangle counter.

3.2 Programmability

Evaluating programmability is difficult to do because of how subjective it can be. In an attempt to add some quantitative data to the evaluation, the ratio between the number of lines of "boilerplate" code and program logic code will be calculated and compared. Boilerplate code is considered code that is only required for the provisioning or management of the tooling itself. Whereas program logic is the code that does the computation we are trying to do.

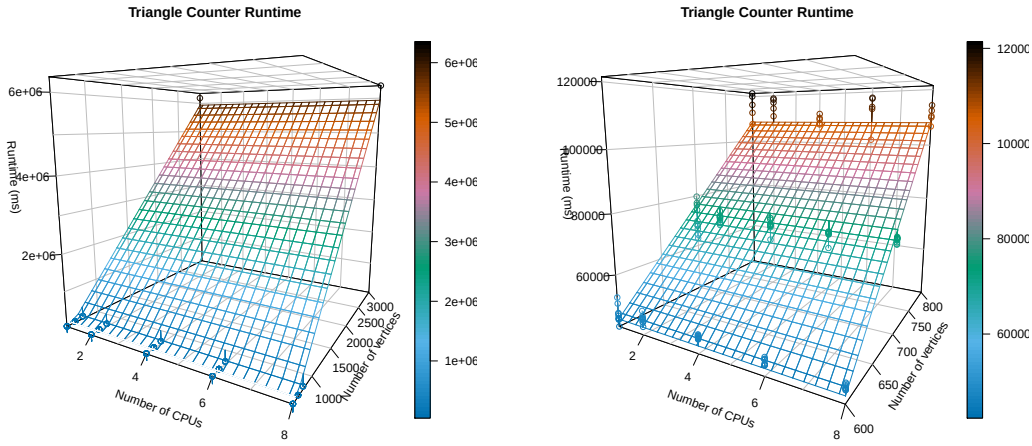
The Repast Simphony library will also be evaluated overall for how easy it is to work in their environment.

3.3 Repast Simphony

Repast Simphony is not just a tool, but an entire environment you must build your simulation into. To write and run repast simphony, you must do it through eclipse with the Repast Simphony installed. There are no other supported editors, so it would not be very comfortable to work in with larger projects unless eclipse is you editor of choice. Even using tools like Vim would be difficult to use as you need the repast plugin to run the project.

To configure simulation parameters, enter parameters, or even run the simulation, you must do it through the repast simphony control panel. This ties you very heavily to their environment in eclipse.

Outside of the tooling limitations, the developer experience is actually quite good. They have their own DSL called relogo that makes creating simple simulations quite easy, although all of the work I have done has been in repast Java (using Java rather than relogo). Creating agents is very simple as it is a plain Java class, with a few annotations. Most of my frustration when writing repast



(a) Including 3000 vertices datapoints.

(b) Exclusing 3000 vertices datapointgs.

Figure 2: 3D Plot of the performance of Triangle Counter based on 1 to 8 available cores and varying points in the simulation.

simphony was in creating the contexts, which is where most of the Repast Simphony tooling gets used. Type mismatches and incorrect configuration were the most common error I encountered. The challenge with this is that you never got these errors until runtime, and they never produced meaningful error messages. Rather you would get an error of something internal failing in Repast Simphony, and be required to look in the Repast Simphony source code to debug.

With Repast Java, it mostly stayed out of your way. With the Java annotations, there was very little boilerplate that you had to deal with in the code itself other than the context. Most of the interaction with repast revolves around the configuration of your simulations.

The documentation was alright. They would walk you through an example program, and they provided many example programs. What could use some improvement is the Repast Simphony reference[5]. It was a mix between Java docs and a guide, which was difficult because I couldn't use it for either. Because it was a guide, it was naturally incomplete, and limited its discussion to a narrow scope.

Overall it was very usable once you knew what you were doing.

3.4 Closest Pair of Points

The closest pair of points program ended up with 17 directories and 126 files total. Of

[5] https://repast.github.io/docs/RepastReference/RepastReference.html

which, only 7 files were source files. This divides out to only 5.6% of the project being source files. The amount of code is much more lean though. With 462 lines of code total, only 27 lines of that is boilerplate, with the remaining 94% of the lines of code being program logic. Although, there are also 394 lines of XML that configures the simulation. This XML is editable by the user via a text editor, but it is intended to be edited by the Repast Simphony Control Panel. Therefore, I do not count the XML as source code, but it is still present in the project.

3.5 Triangle Counter

The triangle project was very similar to the closest pair of points project in size. The triangle counter project has 17 directories and 123 files, of which only 11 of those files are program logic (8.9% program logic by number of files). This project had a total of 375 lines of Java code, with only 19 of those being boilerplate, leaving 94.9% of the Java code as program logic. This project had 370 lines of configuration code across 13 configuration files.

3.6 Comparison

Both closest pair of points and triangle counter have very similar project sizes, both in file size and number of files. They also had a very comparable amount of configuration.

The results of the quantitative analysis of the programmability of repast simphony echos my observations in the qualitative section. Although the projects are massive in size, both having only around 5% of the files being program logic, well over 90% of the code you write is program logic. Repast simphony is massive, but they hide most of it from you when you are writing your simulation. The weight of repast comes from the configuration and setup of the simulation, not writing the simulation logic itself.

4 Conclusion

Repast Simphony has its place in agent-based modeling systems. It takes advantage of multithreaded performance when you have agents that are computationally heavy, but starts to loose the performance benefits as the simulation tick count increases. It is really quick and easy to get a repast simulation running with the ReLogo DSL. If you need more functionality than ReLogo provides, Repast Java is really easy to pickup because of the low-overhead of boilerplate, even though it might take some time to debug and configure the simulation perfectly. I was very impressed by Repast and I see myself using it for light workloads when I want a graphical representation of the simulation and don't require large simulations.

5 Future Work

The first priority is to complete the benchmark programs for our mobile agents platform: IBM aglets. I plan to benchmark aglets with the same CPP and TC programs discussed in this paper.

I want to further explore my suspension of the delay in incrementing ticks, and the processing of the agent.

I would also like to explore and compare the performance of triangle counter between a high-performance single threaded CPU, and a low-performance single threaded CPU for high-tick-count simulations. I suspect that these types of simulations benefit more from faster single core performance than they do from multi-node computing.

References

[1] S. Gokulramkumar, "Agent Based Parallelization of Computational Geometry Algorithms," p. 14.