From: Nathan Wong
Date: March 17, 2021
To: Professor Fukuda
Subject: CSS490 TermReport

Abstract

This report provides an overview of the work that has been done this quarter on the topic of benchmarking and comparing different platforms to help non-computing scientists make decisions on which library is best used when their application is being parallelized. These benchmarks will be compared towards social, behavioral and economical/environmental engineering applications. The libraries that are being judged are Flame, Mass C++ and RepastHPC. Methods to analyze these libraries are the runtime of the benchmarks, and the overall use case of each library. Overall use cases include strengths and weaknesses, as well as ease of implementation. Currently, there is minimal data to be found on the benchmarks that I have created this quarter. I will begin running and analyzing the benchmarks that were implemented this quarter at the start of the following quarter alongside implementing and analyzing the other benchmarks. This report highlights the implementation of the Social Network benchmark program and the MATSIM benchmark program using the RepastHPC library and the current status of the research.

Introduction

Overview of the research

This research is focused on testing three different libraries across seven different benchmarks. The three libraries that are being tested are Flame, Mass C++ and RepastHPC. These libraries all use multi agents to simulate models, however they implement them in different ways. Agents are robots that perform specific tasks automatically dependent on the program. We will be testing this library across seven different benchmarks. These benchmarks are Game of Life, Social Networks, Tuberculosis, Brain Grid, Bail-in Bail-out, MATSIM and VDT. These benchmarks represent social, behavioral and economical/ environmental applications. This quarter the tasks were to implement the Social Network and MATSIM benchmark for the RepastHPC library.

Justification of the research

In today's world, there are an enormous number of multi agent libraries available for high performance computing. While this is beneficial, as it allows more flexibility in computing programs, it would be hard for a non computing scientist to choose the optimal library for their project. This research will shed light on which conditions and parallelized application the libraries mentioned above would be beneficial for both computing and non computing scientists.

Structure of the report

This next section of the report will go into the specification and implementation of the benchmarks implemented. This section will include a description of classes and purposes, any helpful diagrams and methods for accomplishing the benchmark. The following section will then go into the current status of the research, which will state

where we are at in our research. Finally the last section will be the conclusion, summarizing the report and laying out the plan for future work.

Specification and Implementation

Social Network Specification

The Social Networking benchmark is meant to simulate the growth of a person's social circle. The Social Network Specification states that a group of Person agents are created. A user inputs the amount of agents to be created and either a variable *k* or *m*. Respectively, these variables represent the number of first degree friends or the divisor of the Person agents in order to find the degree of friends. When the program starts each agent will begin finding their first degree friends until a k-regular graph is formed. A k-regular graph is when each vertex has the same amount of edges. Figure 1.0 shows an example of a k-regular graph with k being equal to 3.h

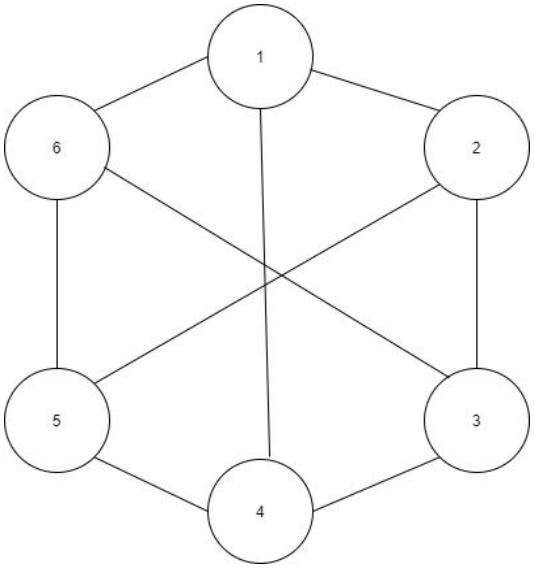

Figure 1.0 - K-regular graph with 3 edges

The output of the benchmark is each Agent's friends up to a certain amount of degrees.

Social Network Implementation

The Social Network implementation has two major classes. The Agent class and the Model class. The Agent class's main responsibility is to add another agentid to its friendlist, while the Model class's main responsibility is to tell the Agent class which agentid to add.

Agent Class

The Agent class holds data on its id and a vector of ints that consists of the id of its friends. The notable method is the play method.

The method play takes in an int variable and adds it into the vector.

Model Class

The Model class holds data for the amount of agents, amount of first degree friends, the amount of degrees to print and finally a set of all the Agents in the program. The notable methods are doSomething,init and print.

The doSomething method first checks if the amount of first degree friends is odd. If it is, it instructs it's local agents to add the agent opposite of it to its friends list. To find the opposite id, the Model class uses this equation agentID + (amountOfAgents/2)% amountOfAgents. If not it heads to the main loop of the program. The loop instructs the agent to use the equation (agentID + *i*) % numAgents, where i is a counter in order to find the id of the agent to add to it's friend list. If the id is not in the friends list, it then adds it into the friends list and increments *i*. The loop stops once *i* is bigger than the required amount of first degree friends.

The init method kept two counters. One that kept track of the id to give to the agent, and the other to keep track of which process to create the agent in. Both counters would continue to increment until the amount of agents were sufficient to the user input. If the process counter becomes bigger than the amount of processes it resets to zero.

The print method essentially displays iterations of a breadth first search. The print method would stop when the iteration is equal to the amount of degrees we want to print. In order to prevent duplicates we would have a set of visits and add each agentID to the set once we have printed them out, and load the unvisited ones into the queue.

The Model class serves as somewhat of a supervisor for the Agent class, instructing when each agent should play and communicating with the other processes to synchronize the agents. It is also responsible for displaying the output. Figure 1.1 depicts a flowchart of the doSomething method.

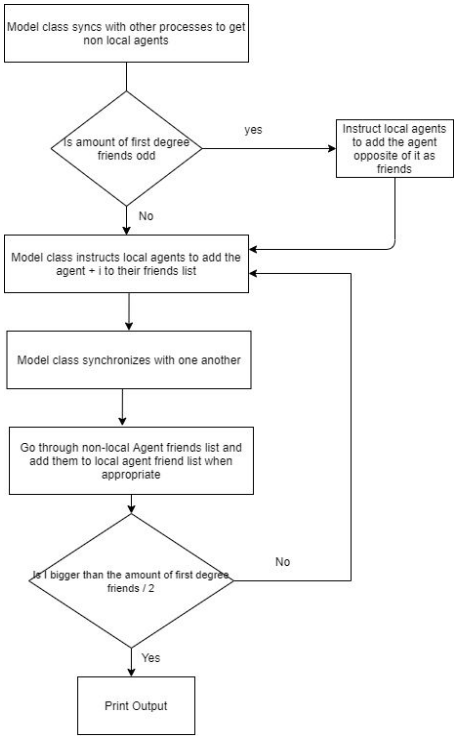

Figure 1.1 - Model::doSomething method

MATSIM specification

The MATSIM benchmark is meant to simulate the flow of traffic and congestion formed during people's morning and evening commute. The MATSIM specification stores a mesh graph of points and links. Each point and link has a capacity. The capacity of the points is the amount of links that are directed towards it, while the links capacity are predetermined. Car agents are placed around the edges of the graph as a starting point, and have a predetermined destination point and route to the destination point. Agents should then begin making their way to the destination point, however the agents cannot move to the point or link if it's capacity is full. Once all the Agents are at the destination point, the program will then have the agents travel back to the starting point.

MATSIM implementation

The MATSIM implementation has four classes. Points, Roads, Agents and Observers. The Points and Roads classes are used by the Observer to create the mesh graph during initialization, while the Agent class is used to move each Agent alongside the points and road. The data and methods of each class is highlighted below.

<u>Point Class</u>

The Point class consists of its x and y coordinates, id, maxcapacity, current capacity and a map of roads that represent edges. The methods of this class mainly consists of getters and setters.

Road Class

The Road class consists of its id, capacity, current capacity and the point id of the origin and destination of the road. The methods of this class mainly consist of getters and setters.

Agent Class

The Agent class's data consists of its id, tick, speed, distanceTraveled, currentIndex, the id of its starting and ending point and two vectors that contain the id of roads. The first vector contains a roadId that leads from the starting point to the ending point and the second vector is vice versa of the first. The notable method is the play method.

The method play takes in a map of points and roads that are provided by the Observer class. It returns either 0, 1 or 2. If the method returns a 0 it means that the agent has not moved to a new location or the road/point ahead is full. If the method returns a 1 it means that the agent has reached a new point. Finally, returning a 2 means that the agent is at its final destination. The method first checks if the agent is at the destination, if it is not it increases the tick. It then checks if it can move, and if it can it increases distanceTraveled by the speed. Once distanceTraveled is bigger than the road length a check is in place to see if the destination is not at capacity. Only once the destination is not at capacity does it move on.

Observer Class

The final class is the Observer class. The Observer class holds the data of finishedCount,currCount, beginningCount, a map of points and roads and a set of agents. Unlike the Social Network implementation, this set of agents are only local agents. The currCount is the amount of agents that are currently finished. The beginningCount is the amount of agents that are expected to be finished after the first portion of the simulation is done. Similarly the finishedCount is the amount of agents that are expected to be finished after the second portion of the simulation is done. The notable methods for the Observer class is the doSomething methods.

The method doSomething loops until the currCount exceeds either the finishedCount or beginningCount depending on which part of the simulation we are on. The observer has all the agents play and then takes action based off of what is returned. If a 0 is returned it does nothing. If a 2 is returned it increments currCount by one. However, if a one is returned it will then move the agent to a new process based off of the new roadId. The process is determined by modding the roadId by the amount of processes total. Once it has iterated through all the agents in its set it synchronizes any agents that moved with the other processes. The processes will then individually check each agent that moved, if the road or point that it is being moved to is full, it moves the agent back to the previous process. If the loop does not exit, the observer resets currCount to one. Figure 1.2 displays a flowchart of the doSomething method for the first portion of the program. A similar flowchart can be displayed for the second portion by swapping beginningCount with finishedCount.

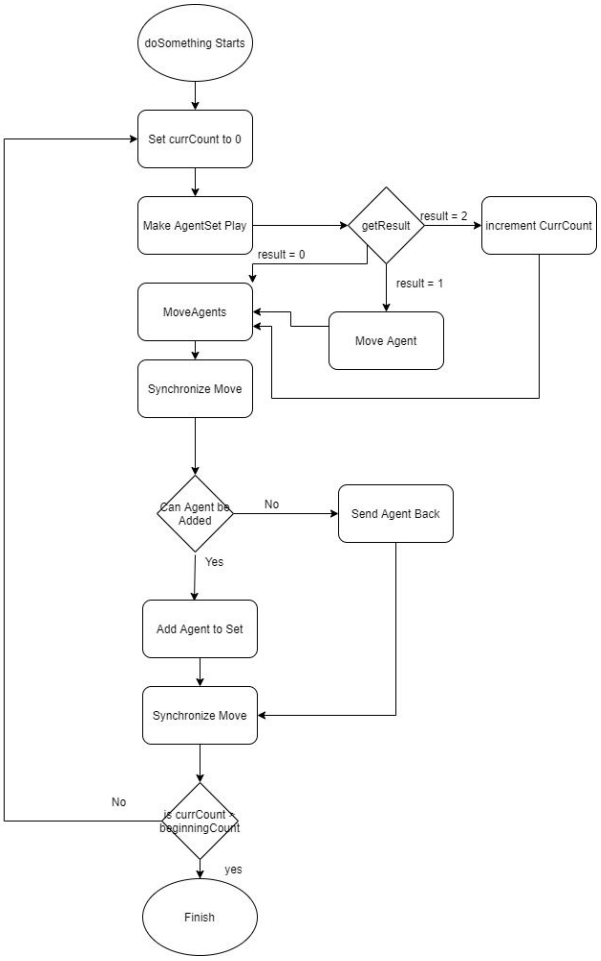

Figure 1.2 - Observer::doSomething method

Current Status

The table below shows the current status of the research as a whole.

	Game of Life	Social Network	TB	Brain Grid	Bail-in Bail-out	MATSIM	VDT
Flame	X	X	X	X		X	
MASS C++	X	X	X	X		X	
Repast HPC	X	X	X	X		X	

The X represents benchmarks that have finished implementing, however some benchmarks may still need data to be collected.

My personal status for this quarter is that I have finished implementing Social Network and MATSIM for RepastHPC, however I have yet to graph any data for them. I am hoping to begin to do that next quarter.

Conclusion

We are hoping to help bring clarity into which agent based modeling library would be the most beneficial under different circumstances. While we do not have all the data yet, we hope that once we do we are able to help computing and non computing scientists alike make informed decisions on the library that they should choose.

Future plans

For next quarter I will be continuing this research and implementing the Bail-in Bail-out and VDT benchmarks for RepastHPC. I will also begin graphing data for these benchmarks and the benchmarks mentioned above. This quarter has been quite a challenge, and has forced me to change the way I think when implementing programs. I heavily underestimated the difficulty of implementing programs in parallel as it required a vastly different mindset then implementations I am used to. Due to this, next quarter I plan on spending more time designing the benchmarks, and in doing so hope that it will make future benchmarks easier to implement and less time consuming.