From: Nathan Wong
Date: June 10, 2021
To: Professor Fukuda
Subject: CSS490 Term Report

<u>Abstract</u>

This report provides an overview of the work done in the spring quarter of 2021 on the topics of benchmarking and comparing different platforms in order to help researchers make decisions on the best available library for their application domains. There are seven benchmarks in total, and these benchmarks were chosen to simulate real world occurrences. These benchmarks are: The Game of Life, Social Networks, Tuberculosis, Brain Grid, Bail-in Bail-out MATSIM and VDT (Virtual Design Team). The libraries that will be running these seven benchmarks are Flame, Mass C++ and Repast HPC. 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. This report highlights the implementation of the VDT benchmark and the Bail-in Bail-out benchmark. Furthermore, this report also details the parallelization and data gathered from the previous quarter's benchmarks (Social Network and MATSIM) and the benchmarks mentioned above.

<u>Introduction</u>

Overview of the research

This research is focused on testing three different libraries across seven different benchmarks. The 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. Flame uses a messaging board where Agents can read and write messages and perform actions based upon the message received. Mass C++ makes use of a distributed array where each element is called a place. Each agent can then act within the places. Finally, Repast HPC utilizes a sharing and requesting system for agents. This allows agents to be shared within each process when needed. Agents are robots that perform specific tasks that are automatically dependent on the program. We will be testing these libraries 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 Bail-in Bail-out and VDT benchmark for the RepastHPC library.

Research Purposes

Just like there are many different libraries to do similar functions, the same is to be said with agent based modeling. While having many options is always a positive thing, it also shows a problem in choosing the optimal library for projects. Not a lot of research is done on the performance of agent based libraries, and with this research we will help show under which conditions and applications that the above libraries would be beneficial.

Structure of the Report
The next sections of the report will go into the specification and implementation of the VDT and Bail-in Bail-out benchmark. This section will include a description of classes and purposes, data flow diagrams and overall how the program works. The next section will evaluate the data and go in depth to explaining each occurrence. The following section will contain reflections about the work that was done these past two quarters. The final section will be the conclusion, summarizing the report and laying out next steps.

Specification and Implementation

VDT Specification
This simulation is meant to simulate different software engineering teams completing different projects. The simulation takes in an input file that consists of projects that need to be completed. The simulation also takes in a user input for the amount of engineers per team and of what type. Projects are defined as tasks, and have an array stating how many hours each engineer needs to work on it. There are five different engineers in total, and each engineer has a hierarchical status. In order of hierarchy these engineers are: Project Lead, Senior Software Engineer, Junior Software Engineer, Test Engineer and UX designer. This hierarchy can be seen as a tree where Tasks start from the root nodes and stop at the leaf nodes. Figure 1.0 depicts an example of an engineering team of 15. This team consists of 1 Project Lead, 2 Senior Software Engineers, 3 Junior Software Engineers, 4 Test Engineers and 5 UX designers.

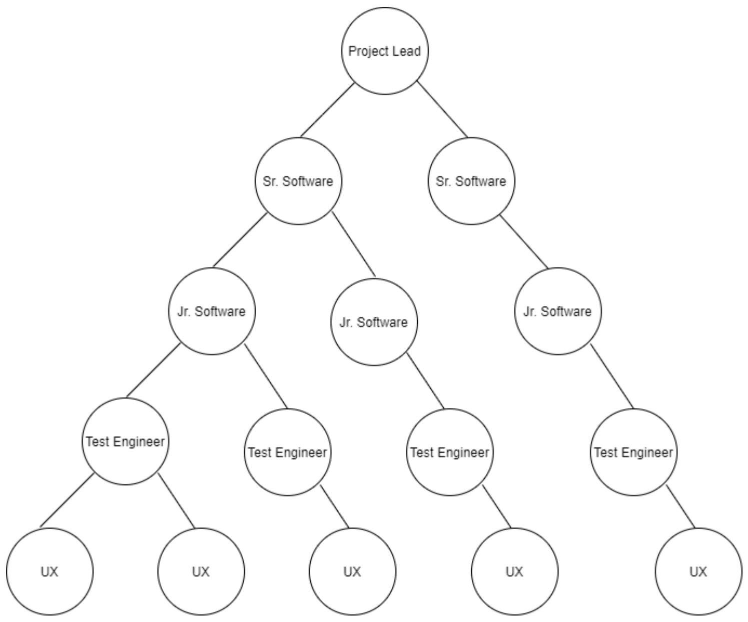


Figure 1.0 - Engineering team of 15

If an engineering team had less engineers down the hierarchy the node would simply point to one of the other leafs. The output of each benchmark is how long it took to complete all the tasks.

VDT Implementation

The VDT implementation has three major classes. These classes are the Task class, the Agent class and the Observer class. The Task class responsibility is to show how long each agent should work on it. The Agent class's responsibility is to work on the task then pass it on to the appropriate agent. Finally, the Observer's responsibility is to initialize all the agents and tasks, instruct all agents to play, and stop the simulation when the appropriate amount of tasks has been completed.

Task Class

The Task class holds a vector of hours, and ints that consist of the type, total Hours, priority, day and id. The methods of this class mainly consist of getters and setters.

Agent Class

The Agent class holds a queue of Agents that are meant to represent agents that work under this specific agents, a deque of Tasks, an observer pointer, a Task pointer that represents the current Task, and a set of ints that holds the id of Tasks that it has worked on. The methods of this class consist of getters, setters, as well as a few action methods. The most notable action method is called the Play method.

The method Play first checks if the agent is working on any tasks by seeing if currentTask is pointing to anything. If not, it checks if there are any tasks in the deque. If there are no tasks the method ends there. However, if there are tasks in the deque, it then randomly chooses whether to choose the first task, the last task or the one with the highest priority. Once a task is chosen, currentTask points to it, and the id is added into the workedOn set. At every tick of the simulation it decreases the hours needed to work on it by one. Once that hour reaches zero, the agent then uses the queue of agents that work under it in order to pass the task onto the hierarchy. If the agent is a UX agent, it notifies the observer that a task is completed, and then deletes the task. Figure 1.1 is a data flow diagram of the Play method.

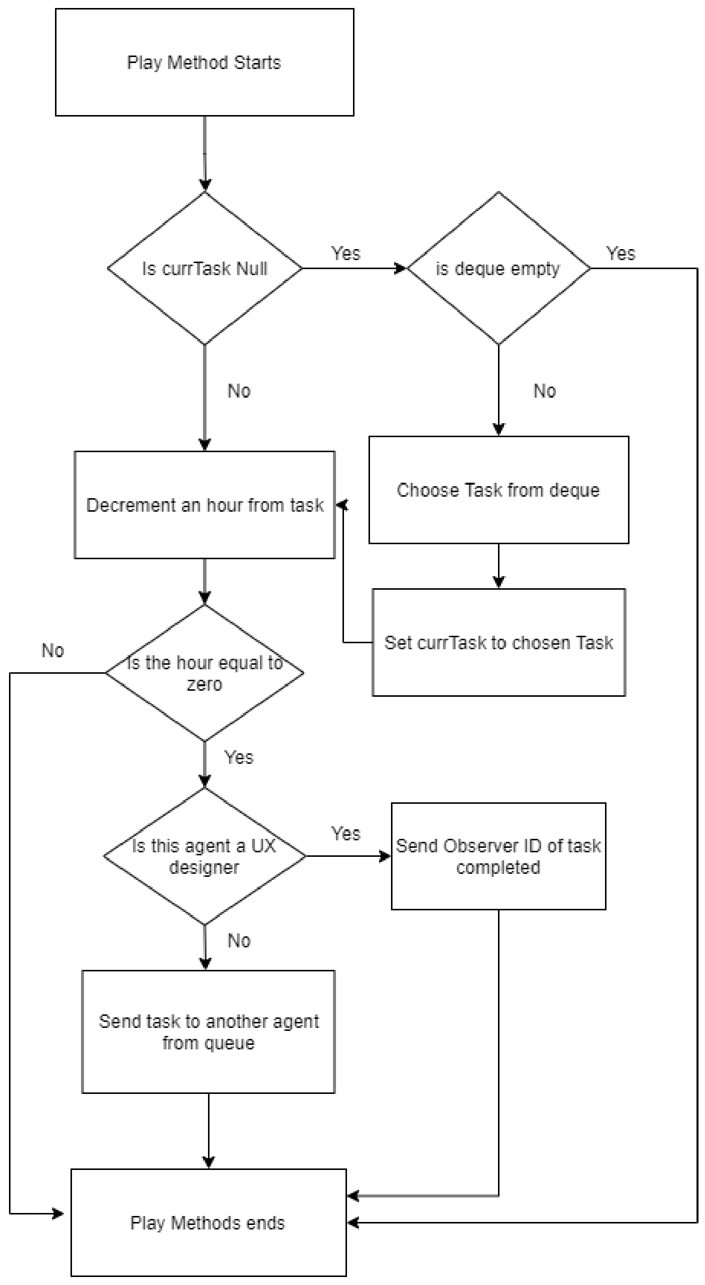

Play Method Starts
Is currTask Null
Yes
is deque empty
Yes
No
No
Decrement an hour from task
Choose Task from deque
Set currTask to chosen Task
No
Is the hour equal to zero
Yes
Is this agent a UX designer
Yes
Send Observer ID of task completed
No
Send task to another agent from queue
Play Methods ends

Figure 1.1 - Agent::play method

<u>Observer Class</u>
The Observer class has a Set of Agents, a deque of the five different engineers, a set of ints that represents completed tasks, as well as two ints representing hours and day. The key methods are distributingTasks, as well as the go method.

The distributingTasks methods distributes the tasks that were created to the correct agents. Using the deque it rotates the head agents and assigns tasks to them until there are no more tasks to give out.

The go method is also quite simple, as the agents play method does most of the work. First the observer checks to see if it has the correct amount of tasks completed in its set, if it does then the simulation ends. If it does not, it instructs each agent that it controls to play. It will then increment the hour. Once the hour reaches over 24 it increases the day, and resets the hour to zero. It will then repeat this until the size of completed is equal to the expected amount. Once it has completed it will print out a statement saying which team finished and at which day and hour it finished at.

Bail-in Bail-Out Specification

The Bail-in Bail-out is an economic simulation. It contains the following agents: workers, firms, owners and banks. The worker agent's job is to bring home money, consume a portion of it and deposit the rest into the bank. The firm agent's job is to either turn a profit or a loss. If the firm has negative liquidity, it must go to its owner to check if they have any capital. If the owner has enough capital they may deposit their own money in order to have the firm break even. Otherwise the firm must go to a bank for a loan. It must then pay off the loan in the future or risk going bankrupt. The bank gives a loan and expects a return. It may loan money to other banks or firms. However if a firm is to go bankrupt they would not receive a return on their loan. Finally, if a bank was to go bankrupt the simulation would then halt.

Bail-in Bail-out Implementation

The Bail-in Bail-out Implementation has 6 major classes. These classes are the Worker Class, the Firm Class, the Owner Class, the Bank Class, the Observer class and the Messenger class. The Worker, Firm, Owner and Bank class are all considered agents and therefore perform their duties as instructed. The Observer class holds the process's agents and is tasked with creating Messenger agents to send to other processes when needed. It is also responsible for fulfilling any Messenger agents that it may receive. Finally the Messenger class's sole purpose is to go to other processes and let the observer know which agent to either take money from or give money to.

Worker Class

The Worker class holds the following data: an AgentID, a double representing wages, a double representing consumption, a double representing deposit, a Bank pointer, and an Observer pointer. The bank pointer is used to get the id of the bank that it plans on depositing money into. The observer pointer is needed in order to create a messenger agent to send to the appropriate process so the bank value actually gets updated. The main method for this class is the play method.

The play method takes in its wages. It then sets the deposit to the amount received multiplied by consumption. Once finished, it tells the observer to create a messenger agent to send to its bank with the deposit amount.

Firm Class
The Firm class holds the following data: an AgentId, a vector of Workers, a double representing productionCost, another double representing dividends, a shared context of banks, a map that represents debt, and finally a pointer to the observer. Like the reasoning above, a pointer to the observer is needed in order to create the messenger class. The shared context of banks is also needed in order to have a correct view of the interest rates. The main method for this class is the play method and the receiveLoan method.

The play method first calculates the productionCost by randomizing the current productionCost by a double ranging between 0.5 and 1.5. It then moves on to calculating the profit by multiplying the productionCost by another double in the range of 0.5 and 1.5. It then calculates the liquidity by going through each worker and subtracting the wages from each worker. Once that is done, if there is a profit it pays dividends. However, if there is not a profit and the firm goes into debt it then calls the receiveLoan method. It then goes through the map and pays off either half of the debt or all of the debt if it can afford to. If there is still debt it increases the debt amount by the interest rate and then returns.

The receiveLoan method takes in an int representing the amount of banks that the firm went to. We then use the shared context in order to get a vector of the banks, and then iterate through finding the one with the lowest interest. Once found, we instruct the observer to create a messenger agent with the amount that we need in order to be at zero.

Owner Class
The Owner class holds the following data: a double representing its capital, a Firm pointer representing the firm it owns, as well as an AgentId and an Observer pointer. The need for an observer pointer is the same reasoning that was mentioned in the previous two classes. The methods of this class mainly consist of setters and getters.

Bank Class
The Bank class holds the following data: an AgentId, a double representing liquidity, a double representing the current interest rate it offers, a shared context to other banks, a map representing debt and an Observer pointer. Like the classes previously mentioned, the observer pointer is needed in order to create messenger agents. The main method for the bank is the play method.

The play method simply checks to see if it’s liquidity has hit negative. If it has it requests a loan from other banks. It also pays off its current debt by either half or all of it. If it cannot pay off half of its debt, it is considered bankrupt.

Messenger Class
The Messenger class holds the following data: a double representing the amount to deposit or receive, an int representing the type (whether it is a bank, firm, owner or worker) and finally an int representing the id. This class mainly consists of getters and setters.

Observer Class

The Observer class holds the following data: a shared context of all the previously mentioned classes, as well as any providers and receivers needed to send agents across processes. It's main method is the play method, the sendMessenger method and the fulfillMessenger method.

The play method simply checks if any banks are bankrupt. If not, it instructs all the agents to play. Once they have finished playing, it then calls the sendMessenger agents, which sends the messenger to the appropriate process. Afterwards, it calls the fulfillMessenger process which goes through each Messenger and either deposits or takes away the money from its agents. It repeats until a bank agent has declared bankruptcy. Figure 1.2 depicts the play method in the Observer class.

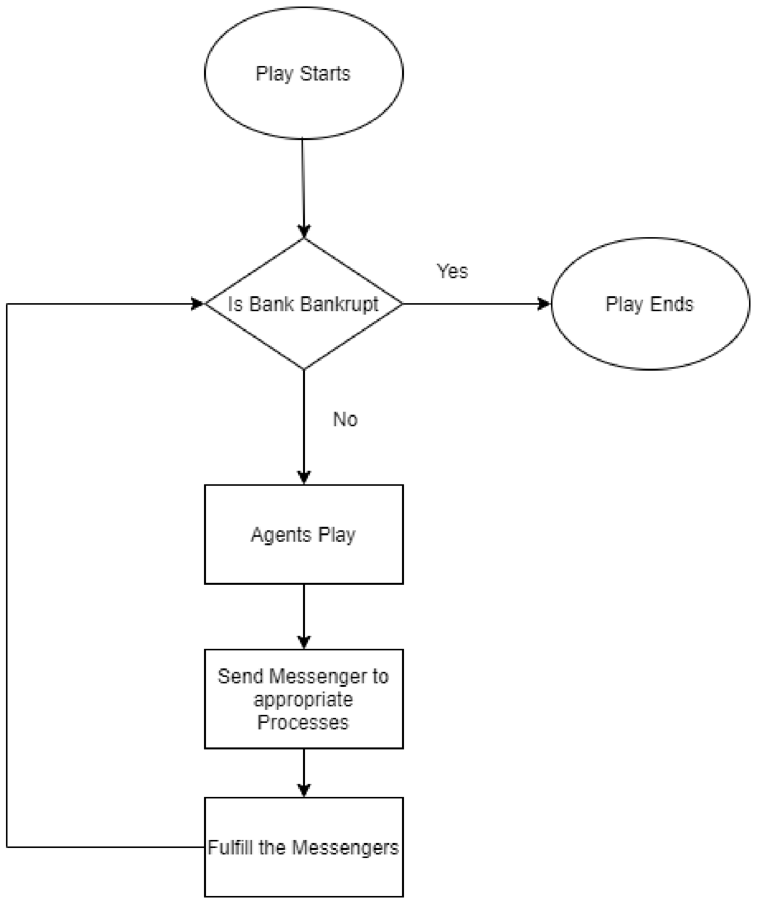


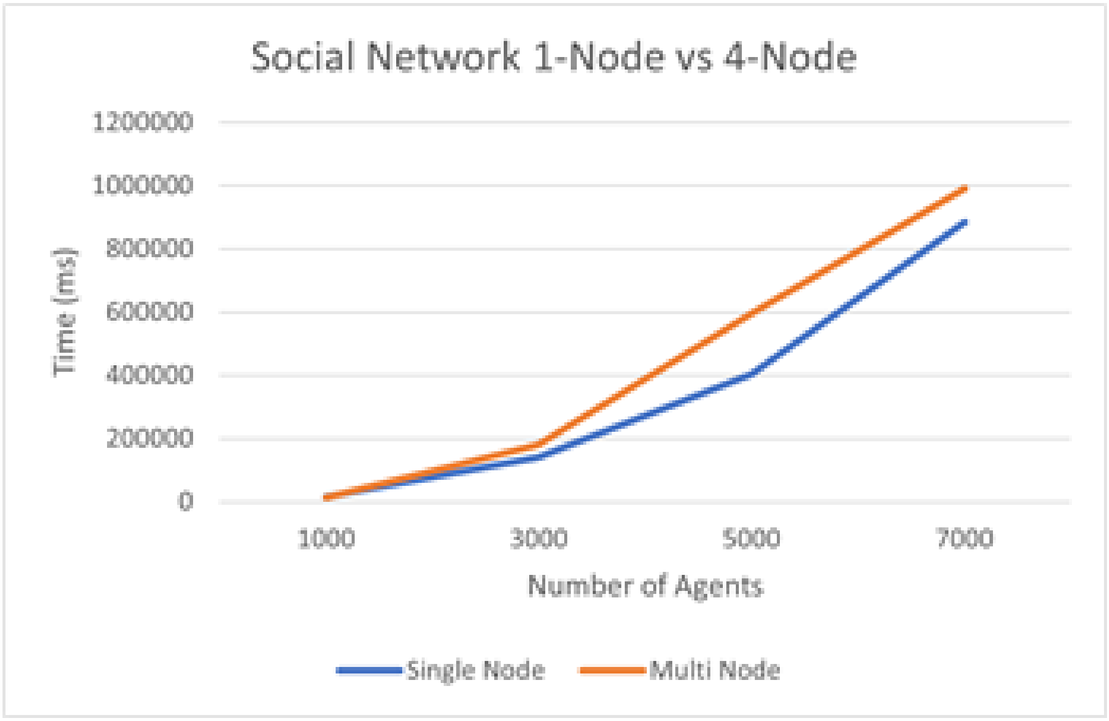
Figure 1.2 Observer::play method

The sendMessenger method works by going through its sharedContext of messenger and sending it to the appropriate process. The appropriate process is found by the function id % worldSize, where worldSize represents how many processes are in the world, and id representing the id of the agent it is trying to send to.

The fulfillMessenger method works by going through each messenger and seeing which type of agent it is first. Based on the type, it then iterates through the appropriate shared context finding the agent that matches based off of id. Once it is found, it then either deposits or takes away the amount from the messenger. Once that messenger is completed, it is then removed from the shared context. This action is repeated until the shared context size is zero.

Performance

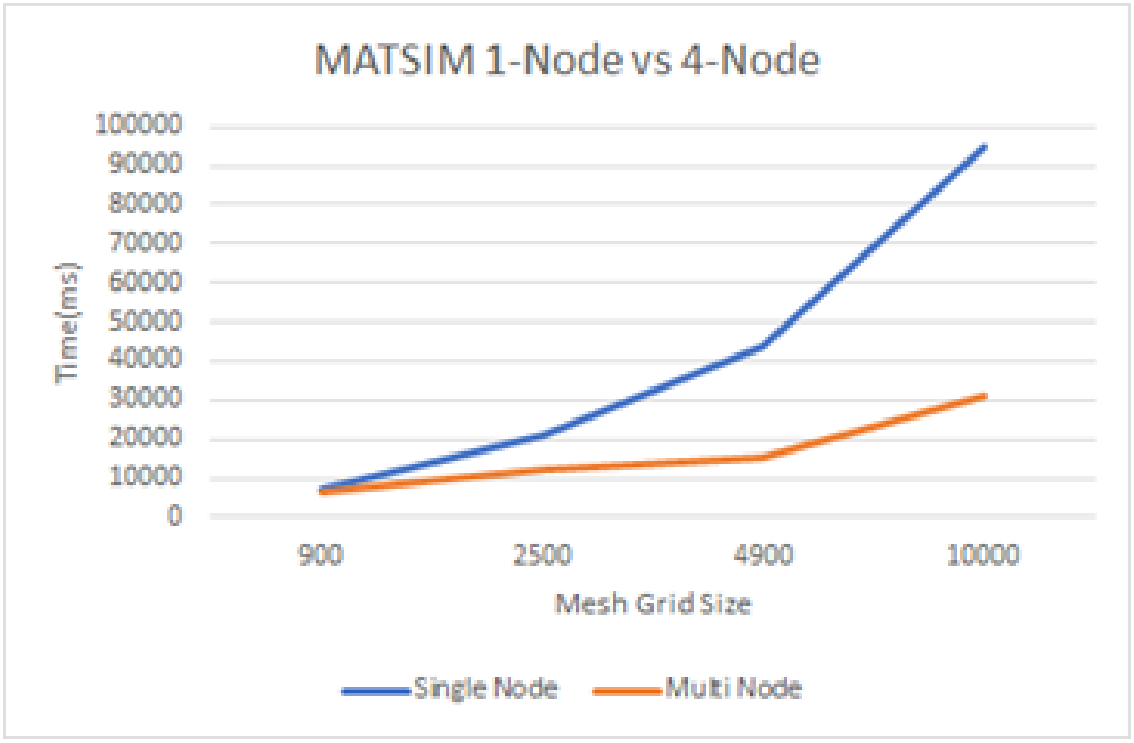
Social Network Performance



Graph 1.0 - Social Network Repast HPC 1-Node vs 4-Node performance

The graph above looks at the performance between 1-Node and 4-Node for the Social Network benchmark of Repast HPC. While the Time to run is very close, this is mainly due to the fact that most of the time is spent printing out the output. Printing out the output does not differ from 1-Node and 4-Node as they both use 1 process to print it out. This is because the output of the social network benchmark must remain consistent, and printing it out on more processes will mess with how it is being printed out. Despite this however, the 4-Node is still slightly faster than the single node.

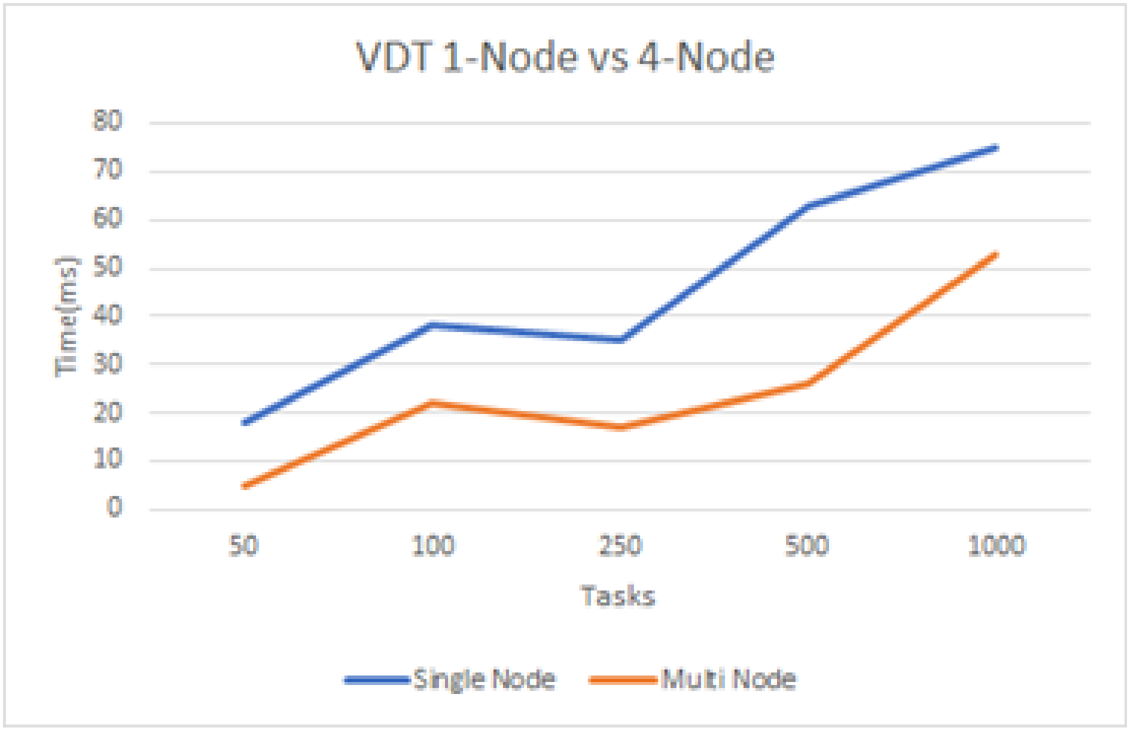
MATSIM performance



Graph 1.1 - MATSIM Repast HPC 1-Node vs 4-Node performance

By looking at the graph above it is evident that the 4-node heavily outperformed 1-node. This would be mainly due to the parallelization, as the processes have to process less node than the single node. As the mesh grid grows bigger, I expect the distance in time to increase between the Single Node and the Multi Node.

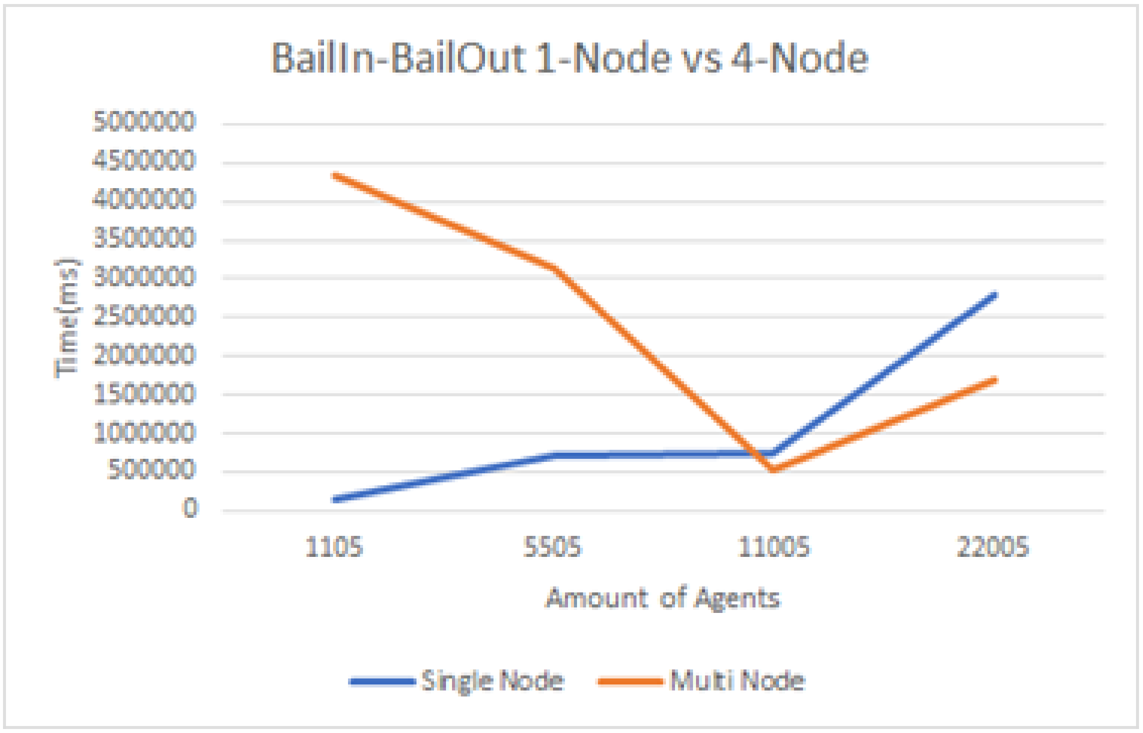
VDT performance



Graph 1.2 - VDT Repast HPC 1-Node vs 4-Node performance

The VDT performance between 1-Node and 4-Node follows a very similar trend. This is because single and multi nodes essentially work the same way. However, the difference is that the 4-Node only has to deal with one team each, while the 1-Node has to deal with multiple teams. This means that it has to spend more time processing as it is not parallelized. Therefore, as the amount of tasks increases I expect the trend to remain very similar.

Bail-in Bail-out Performance



Graph 1.3 - BailIn BailOut Repast HPC 1-Node vs 4-Node performance

This performance has the multi Node take longer in the beginning, but eventually it becomes faster at around the 10000 Agent mark. Interestingly enough the increase of agents causes the program to run faster. This may be because since there are more firms, there is a higher chance for a loan to be needed from a bank, and a higher chance for bankruptcy to occur. The trends are pretty random, this can be attributed to the amount of randomness that occurs in the program. Because of this, running the simulation multiple times may be needed to get an accurate result.

<u>Reflections</u>

Implementing these benchmarks this quarter was a lot easier than last quarter. I believe that spending extra time on design helped me be more efficient, and allowed me to complete benchmarks quicker. However, I still spent a lot of time debugging my methods. I believe that doing more unit testing would help solve this issue. While it may cause me to spend more time upfront, once I put the program together, less time would be needed to ensure that it works. I will be taking these reflections and lessons that I have learned in my future line of work. I hope to be able to continue to reflect upon the work that I have done and find ways to improve efficiency and accuracy.

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

While I will no longer be actively working on creating benchmarks, the future plans for this research is to verify the data that is shown above, finish the remaining benchmarks needed for other libraries and finally write a paper about our findings.

Special thanks

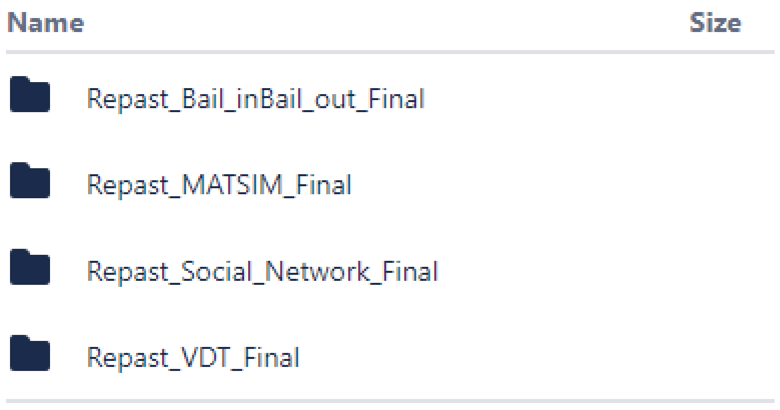
I would like to give special thanks to Dr. Fukuda and my lab partner Sarah Panther for their constant availability the past two quarters as they have helped me learn about parallelization and the Repast HPC library a lot easier.

APPENDIX

Running the Benchmarks

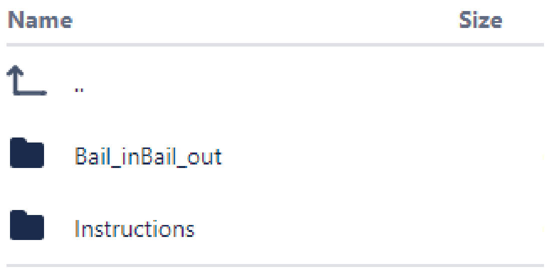
1. Make sure that you have setup mpi correctly on your local and remote machines (If you are unsure on how to set it up check out the file in the hermes machine. I have also attached it in each benchmark's instruction folder).
2. Clone the Benchmarks onto your local machine using the branch (https://bitbucket.org/mass_application_developers/mass_cpp_appl/branch/Nathan_Repast_HPC)
   2a.) This can be done by using the command git pull <remote repo>

   2b.) You can also call git clone using this link as well (https://NathanWong37@bitbucket.org/mass_application_developers/mass_cpp_appl.git)
3. The folder that you have cloned or pulled is set up to contain all four benchmarks in a single directory



Name	Size
Repast_Bail_inBail_out_Final	
Repast_MATSIM_Final	
Repast_Social_Network_Final	
Repast_VDT_Final	

4. Using the cd command change your directory to the appropriate benchmark
5. Upon opening the benchmarks you will see a folder with instructions and a folder with the program
   5a.) There are some directories with an input file program as well. If there is an input file program make sure to read the instructions on how to run it. Once you have ran the input file program, make sure to copy the file(s) produced into the program file.



Name	Size
..	
Bail_inBail_out	
Instructions	

6. Afterwords change the directory to the benchmark program
7. A make file is provided in each benchmark directory, along with the repast hpc library. Ensure that you have booted up your mpi by typing the command CSSmpdboot
8. Afterwords to compile the program simply time make./compile
9. Running the program is dependent on the benchmark you want to run, as some take in an input file. (Replace # from the commands below with the amount of nodes you want to run)
   9a.) To run the social networking benchmark type the command
   ```
   mpirun -n # ./main.exe config.props model.props
   ```
   9b.) To run the MATSIM benchmark type the command
   ```
   mpirun -n # ./main.exe config.props model.props Node_link.txt
   Car_Agents.txt
   ```
   9c.) To run the VDT benchmark type the command
   ```
   mpirun -n # ./main.exe config.props model.props Tasks.txt
   ```
   9d.) To run the Bail-in Bail-out benchmark type the command
   ```
   mpirun -n # ./main.exe config.props model.props
   ```
10. To alter the environment you can simply configure model.props for both social networking and Bail-in Bail-out. More details are in the instructions. To alter the environment for MATSIM and VDT, you will need a new input file that can be generated using the input program.

<u>Collecting the Data</u>

I collected the data by running the benchmarks using a variety amount of agents for each benchmark. I would then run it around three times and keep the lowest amount of time that was produced. Afterwards I would use excel in order to draw and display the graphs. All the time is in milliseconds.

Social Network Data

Table 1.0 - Social Network 1000 Agents

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	1000	**14406**	19545

2	1000	18341	**15470**
3	1000	21261	283832

Table 1.1 - Social Network 3000 Agents

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	3000	**17990**	**14105**
2	3000	24816	21047
3	3000	29102	25614

Table 1.2 - Social Network 5000 Agents

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	5000	627839	**403686**
2	5000	682465	419081
3	5000	**59533**	417836

Table 1.3 - Social Network 7000 Agents

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	7000	1074548	961341
2	7000	**990786**	1041686
3	7000	994347	**884775**

For the Social Network simulation I made sure to make it print out to the third degree, and also have three connections.

MATSIM Data
For the MATSIM data the agents are based off of the mesh grid size and will always be 1/10th of it. This one runs faster as it doesn’t have to print out as much, and can print whenever an agent has finished. The number that was chosen was the closest whole number I could get to having a size of 1000, 2500, 5000 and 10000.

Table 2.0 - MATSIM 900 size mesh grid (90 Agents)

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	90	**7220**	6957
2	90	7613	6610
3	90	7843	**6588**

Table 2.1 - MATSIM 2500 size mesh grid (250 Agents)

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	250	21236	**12200**
2	250	21281	14795
3	250	**21154**	15252

Table 2.2 - MATSIM 4900size mesh grid (490 Agents)

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	490	43832	16918
2	490	49845	15845
3	490	**43653**	**15586**

Table 2.2 - MATSIM 10000mesh grid (1000 Agents)

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	1000	**94899**	38472
2	1000	981242	41002
3	1000	1031037	**31167**

VDT Data
VDT data the agents are kept in 3 teams of 25, however the amount of tasks are increased.

Table 3.0 - VDT 50 Tasks

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	75	**18**	6
2	75	22	**5**
3	75	19	**5**

Table 3.1 - VDT 100 Tasks

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	75	**38**	25
2	75	40	26
3	75	47	**22**

Table 3.2 - VDT 250 Tasks

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	75	**35**	31
2	75	57	34
3	75	52	**17**

Table 3.3 - VDT 500 Tasks

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	75	69	37

2	75	**63**	**26**
3	75	65	29

Table 3.4 VDT 1000 Tasks

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	75	88	55
2	75	79	58
3	75	**75**	**53**

Bail-in Bail-out Data

The Bail-in Bail-out Data consists of 1000, 5000, 10000 and 20000 workers. 1/10th of the amount of workers makes up owners and firms. Then there is a consistent 5 banks throughout each simulation. The data presented below is rather hard to make sense of, as a series of unlucky rolls can bankrupt the program, or a series of lucky rolls could cause the program to run for a while.

Table 4.0 Bail-in Bail-out (1000 Workers, 50 firms, 50 owners, 5 banks)

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	1105	2046014	58024519
2	1105	**139949**	**4340559**
3	1105	13108364	2981063723

Table 4.1 Bail-in Bail-out (5000 Workers, 250 firms, 250 owners, 5 banks)

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	5505	9695437	9106742
2	5505	**708689**	159690719
3	5505	14101038	**3150911**

Table 4.2 Bail-in Bail-out (10000 Workers, 500 firms, 500 owners, 5 banks)

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	11005	**755821**	**521470**
2	11005	514127972	7863174
3	11005	757311	9647125

Table 4.3 Bail-in Bail-out (20000 Workers, 1000 firms, 1000 owners, 5 banks)

Trial	Agents	SingleNode time (ms)	MultiNode time (ms)
1	22005	108691056	**1678082**
2	22005	3109483	2483410
3	22005	**2810700**	4103289

If you have any questions please email me at : wongnat7@uw.edu
Thanks,
Nathan Wong