MASS C++ Benchmark Programs

Ian Dudder
Advisor: Dr. Munehiro Fukuda
CSS 497: Undergraduate Research Project

Project Purpose

MASS (Multi-Agent Spatial Simulation) is a parallel-computing library being developed by the Distributed Systems Lab (DSLab) at the University of Washington Bothell. This library specializes in the implementation of agent-based models for spatial simulations.

The DSLab team aims to compare MASS C++ to RepastHPC and Flame, two other parallel-computing libraries, using the criteria of execution speed and programmability. To ensure a fair comparison of these libraries, the same seven benchmark programs will be implemented in each of the three libraries and ran using the same input arguments and the same runtime environment.

The purpose of this capstone project is to implement four of these benchmark programs in MASS C++ to prepare for the upcoming evaluations of these libraries. The results of these evaluations will be summarized in an academic paper in the ACM's TOMACS (Transactions on Modeling and Computer Simulation) journal.

The four benchmark programs I will be implementing in MASS C++ are

1. Bail-in/Bail-Out
2. Virtual Development Team (VDT)
3. Social Network
4. Multi-Agent Transport Simulation (MATSim)

Benchmark 1: Bail-In/Bail-Out

Bail-In/Bail-Out is a financial model that simulates interactions between banks, firms, and households to evaluate the economic stability of the market and monitor for a bankruptcy of one of the participating banks.

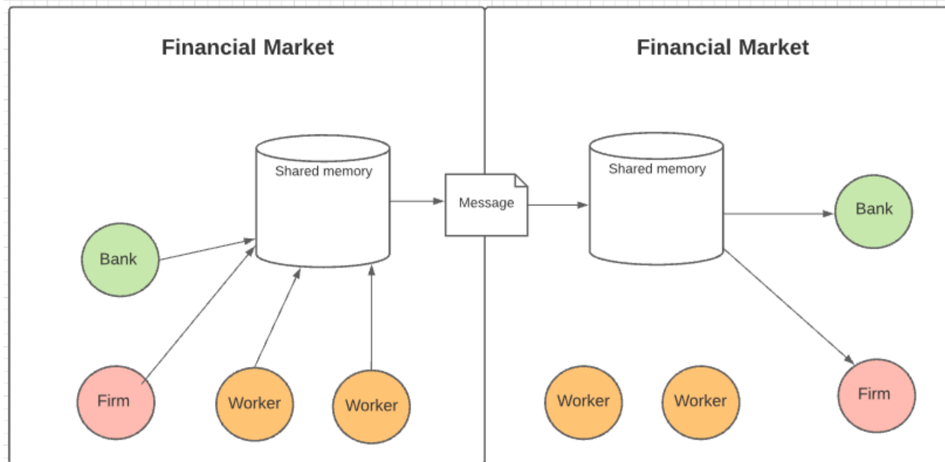
The household entities include workers and owners. Each owner is associated with a single firm and seeks to prevent the firm from going bankrupt. Workers are employed by a firm, earn a fixed wage, and deposit their wages after spending a fixed percentage.

Firms pay workers and try to avoid bankruptcy by taking out loans from banks when they cannot cover their own production costs. Meanwhile, banks seek to approve loans whenever they can and will take out loans from other banks when they cannot.

MASS Implementation

For the MASS C++ implementation of this benchmark, Firms, Banks, and Workers are all Agents, and Owners are objects stored by each individual Firm. The Financial Market class is the place on which these agents reside and interact with each other.

Figure 1: Bail-In/Bail-Out Design



The Financial Market is initialized as a one-dimensional array equal in size to the number of Firms in the simulation. Workers are evenly distributed across this array, and because there are fewer Banks than Firms, Banks populate the lower-indexed entries in the array.

The agents in the simulation use the Financial Market they reside on as a shared memory space, relying on it to communicate with other agents and send messages to other Financial Market places.

The simulation runs in rounds, with each round giving each entity a turn to act. The following sequence of events plays out during each round of the simulation:

1. Firms calculate their costs and profits, pay their workers, and pay back any outstanding loans. If neither the firm nor the firm's owner can cover the costs incurred during this round, they will approach three random banks to take out a loan at the lowest interest rate possible. However, if the firm is already in debt, they bankrupt and re-enter the simulation with default values.
2. Workers receive wages from their employing firms, spend a percentage of their wages consuming products, and deposit the leftover amount at their bank.
3. Banks receive payments, pay back loans, and seek loans from other banks as needed to cover their costs. If a bank cannot secure a loan and ends with a negative liquidity, it goes bankrupt and the simulation ends.

In most cases, Firms reside on a different place element than the Bank they need to communicate with, so it was not possible for them to know what the Bank's interest rate and liquidity were when trying to take out a loan. My solution to this was to have every

Financial Market place keep a local copy of the Bank's information in the "bank registry" allowing Firms to look up each Bank's interest rates and liquidities when they needed to take out a loan. Then Firms add a negative value to their Financial Market's outgoing transactions total to be sent to the Bank in a subsequent call to exchangeAll.

This works relatively well, however, the Financial Market is only able to update its bank registry at the end of each round, meaning that the local copies are not synchronized between places during the round. This runs the risk of two Firms taking out a loan from the same bank believing that the Bank can afford it, when in reality, granting the two loans causes the Bank to have a negative liquidity. However, this handling of the issue appears to be similar to the RepastHPC benchmark, so at the very least the flaw is consistent across library implementations.

Furthermore, the RepastHPC benchmark implementation deviates slightly from the original specification in that it reduces the workers' wages by 70% each round rather than keeping the wages fixed. Furthermore, the Bank's interest rates are the same for each Bank and are not randomized each round. These deviations are reflected in the MASS implementation to keep it consistent with the RepastHPC benchmark. However, it appears that the reduction in Worker wages each round causes a bankruptcy to happen very early, so to accommodate this, the program can be given a fixed number of rounds that it will always run regardless of when a bankruptcy occurs.

Programmability Analysis

In terms of programmability, this program was very difficult to model as an agent-based model. Banks and Firms very much needed frequent direct communication with each other, which was not possible when both classes were agents on different computing nodes. To work around this issue, I had to create the Financial Market Place to act as a shared memory and means of communication between the two entities. However, this slowed down the program will all the exchange all calls needed to let agents communicate.

In terms of quantitative data for the programmability, the lines of code (LoC), number of classes, and number of methods are also considered. The data is summarized in Table 1 with more details available in Appendix D.

Table 1: Bail-In/Bail-Out Quantitative Programmability Data

Quantitative Measure	Count
Number of Files	11
Number of Classes	5
Number of Methods	46
Total Lines of Code	1478
Lines of actual logic	578
Library-specific Boilerplate LoC	135
Non-Boilerplate Lines of Logic	443

Benchmark 2: Virtual Development Team (VDT)

Virtual Development Team simulates a team of 25 software engineers working on tasks to complete a project. Each engineer may be any of the five different types in the following engineering hierarchy:

1. Project Lead
2. User Experience Designer
3. Senior Software Developer
4. Junior Software Developer
5. Test Engineer

Each task must flow from top to bottom in this hierarchy, meaning that different numbers of each type of engineer on a team will result in a different completion time of the tasks. This program demonstrates which combination of software engineers on a team will result in the fastest completion time.

There are two types of tasks: production tasks and collaboration tasks. Production tasks require a certain number of hours from each type of engineer and flow from top to bottom in the engineer hierarchy. Collaboration tasks simulate meetings and require participating engineers to halt work on their production tasks to participate.

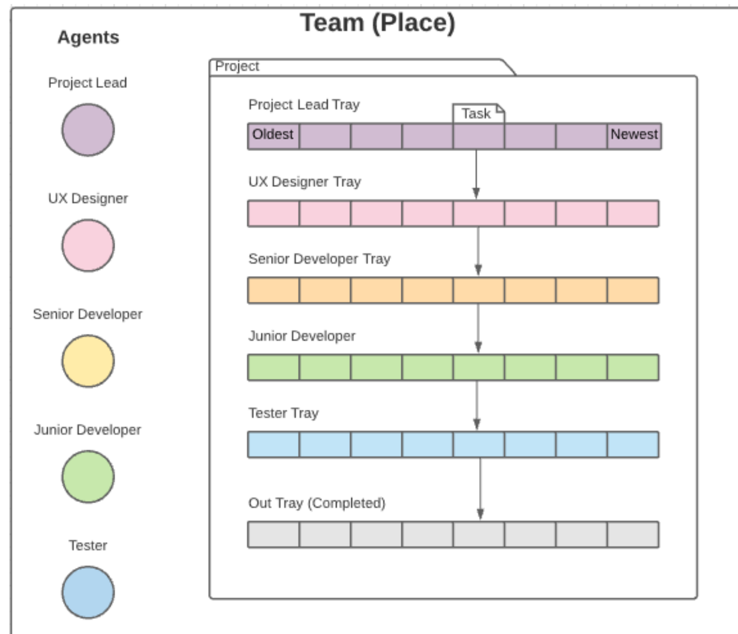
Engineers will choose tasks from their respective tray with a 20% chance to choose the newest task, a 20% chance to choose the oldest task, a 10% chance to choose a random task, and a 50% chance to choose the highest priority task. Additionally, to add a random real-life element to the simulation, there is a 30% chance that once a task has been selected, an exception occurs. This means that the engineer will have to return the task to the next tray above it with 1-8 extra hours added on for that type of engineer. Because there are no trays above project leads, if they have an exception, the number of hours required for project leads is doubled.

MASS Implementation

In MASS, Engineers are Agents and Teams are Places. Tasks are read in from a text file, as are a number of different configurations for the teams. The Teams are instantiated as a 2D matrix each with 25 Engineer agents. Engineers will be assigned a different type depending on what configuration their team has. If there are more Teams than configuration types, the configurations will repeat. Teams manage an array of trays, one for each type of engineer, where engineers take production tasks from.

The simulation runs one hour at a time, with each Team keeping track of the current day and hour. Each hour begins with the Team checking if it is time to start a collaboration task. If so, it marks it as in progress.

Figure 2: VDT Design



Next, engineers check to see if there is a collaboration task in progress. Collaboration tasks specify which types of engineers need to attend, and only require one engineer of each required type to attend, so the first engineer of each type that checks the collaboration task will be assigned to it.

Then engineers work on their production task if they have one or select a new one if they do not. Once the number of hours required by the engineer type reach zero, the engineer returns the task to the Team to be passed down a tray. Figure 2 shows the lifespan of a task as it moves down trays as engineers work on it.

Once all production and collaboration tasks are complete, the Team records the day and hour they completed them. Once all Teams are finished, the completion times for each team are printed to the program's output.

Programmability Analysis

This benchmark was conceptually very straightforward to model with MASS. Modeling Teams as places and Engineers as agents gave the simulation all the tools it needed to run properly. Furthermore, because Teams work completely independent of one another, this program did not require any exchangeAll calls, which means this program was not subject to the same communication issues as the other benchmark programs.

In terms of quantitative data, the data is summarized in Table 2 with more details in Appendix D.

Table 2: VDT Quantitative Programmability Data

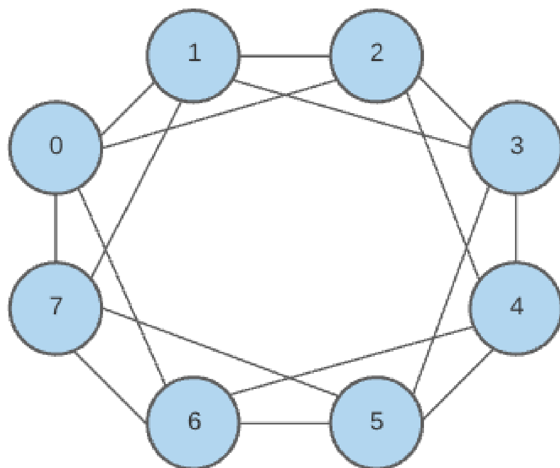
Quantitative Measure	Count
Number of Files	7
Number of Classes	3
Number of Methods	39
Total Lines of Code	1208
Lines of actual logic	571

Library-specific Boilerplate LoC	84
Non-Boilerplate Lines of Logic	487

Benchmark 3: Social Network

Social Network simulates a network of people each with a finite set of first-degree friends. Then it computes the degrees of friendship between people in the group. For example, a first-degree friend is someone that a person knows directly, and a second-degree friend is a friend of a friend.

Figure 3: Four-Regular Graph



The social network is modeled as a k-regular graph, a special kind of graph where each vertex (person) has the same number of edges (friends). Figure 2 displays a four-regular graph, where each vertex has exactly four edges.

A requirement for a k-regular graph is that either the number of edges or the number of vertices (or both) must be an even number.

MASS Implementation

In MASS, this benchmark was effectively modelled using only People as Places and did not require any agents. The algorithm to find any degree of friendship is simple. To get the ith degree of friendship, each Person asks their first-degree friends who their i-1 degree friends are. This algorithm leverages the exchangeAll method to pass this information between Persons in the simulation during each round.

Once execution is complete, each person prints out their list of friends for each degree of friendship.

Programmability

This benchmark program was very easy to model in MASS, with the Places class offering all functionality necessary to implement it. Because MASS uses an array of Places and allows them to add each other as neighbors, MASS is the perfect library to model a graph structure. Therefore, implementing the k-regular graph in MASS was very simple.

The quantitative data for SocialNet's programmability can be found in Table 3 with more details in Appendix D.

Table 3: Social Network Quantitative Programmability Data

Quantitative Measure	Count
Number of Files	3
Number of Classes	1

Number of Methods	10
Total Lines of Code	396
Lines of actual logic	180
Library-specific Boilerplate LoC	44
Non-Boilerplate Lines of Logic	136

Benchmark 4: Multi-Agent Transport Simulation

Multi-Agent Transport Simulation (MATSim) simulates a fleet of vehicles commuting during peak hours of traffic with similar start and end points in the roadway. With a limited number of cars that can occupy an intersection or road at one time, congestion on the roadway is bound to occur causing cars to be idle during their route.

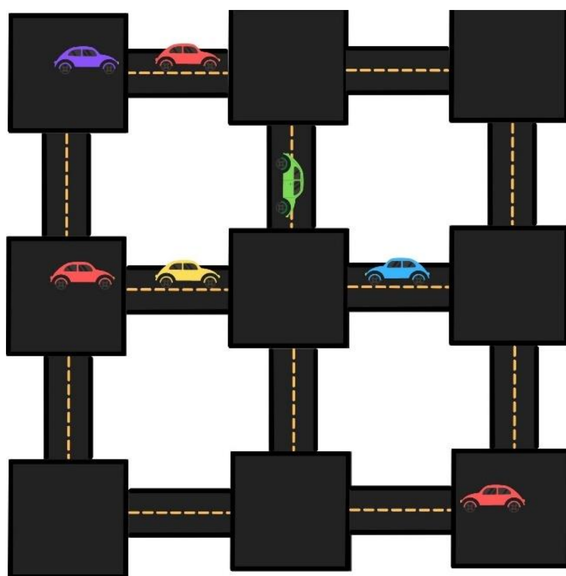
This simulation has cars do a morning route where they travel to a destination near the center of the city grid, and an evening route that has them travel back to their start point on the outskirts.

The city roadway is modeled as a graph where intersections are vertices and the roads between them are edges. The graph is created and stored in a text file prior to the start of the program. Similarly, agents' routes are calculated ahead of time and stored in a text file as well. The shortest path is always chosen for each agent's route.

Implementation

In MASS, this benchmark is modelled with Cars as Agents, Intersections as Places, and Roads as edges. The graph and routes are read in from a text file and placed into a shared memory segment on each computing node for the Intersections and Cars to read from.

Figure 4: MatSim Roadway



Cars drive along Intersections onto Roads, and then use MASS's migrate function to travel to the next Intersection in their route. This implementation uses direction-based collision handling to prevent intersections from exceeding their capacity. This means the simulation will only allow one direction of movement at a time, then pausing to update the number of open spots each intersection has. For example, cars will move north, and intersections will update their capacity before allowing the next wave of cars to move east.

Programmability

This benchmark has been conceptually easy to design and model in MASS. Again, MASS is very

good at implementing graphs, so modeling the roadway in MASS was mostly automatic. Additionally, the agent class offered all the needed functionality for cars to move through the roadway.

The quantitative data for MATSim's programmability is in Table 4 with more details available in Appendix D.

Table 4: MATSim Quantitative Programmability Data

Quantitative Measure	Count
Number of Files	9
Number of Classes	4
Number of Methods	38
Total Lines of Code	1195
Lines of actual logic	552
Library-specific Boilerplate LoC	104
Non-Boilerplate Lines of Logic	448

Evaluations

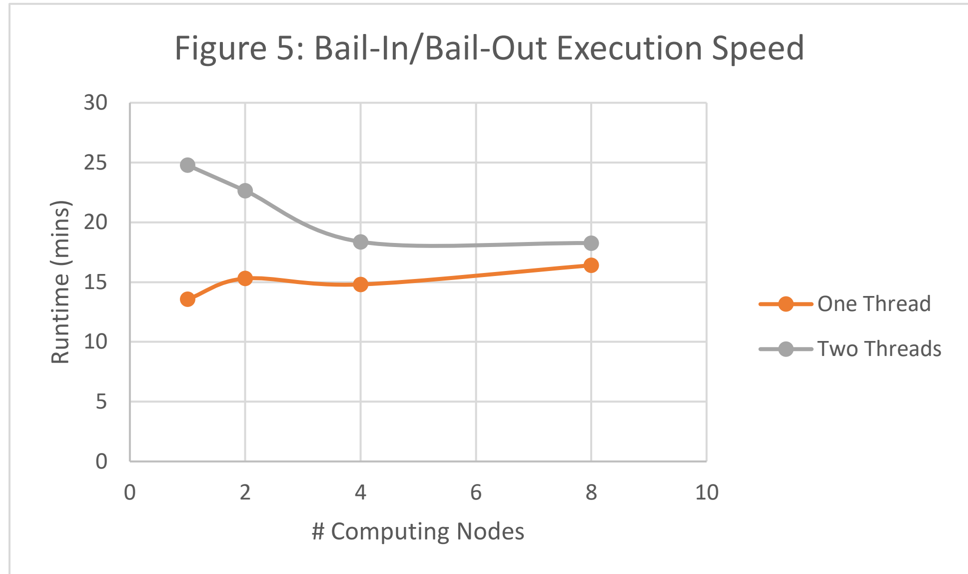
To collect evaluative data on each benchmark program, input arguments are chosen with the intention of having each program run on one computing node for around 30 minutes. Then the program is ran using a different number of threads and processes each time to observe how parallelization affects the runtime.

Note that the following evaluations are not the same evaluations that will be used to compare the libraries. Instead, all the benchmark programs for all three libraries will be re-ran using the same input arguments. The following results are merely to demonstrate how the benchmark programs can be parallelized and how they might be evaluated.

Bail-In/Bail-Out

Bail-In/Bail-Out was executed using 20,000 worker agents, 2,000 firms, and 5 banks with an initial production cost of $35,000, an initial interest rate of 0.8, and a liquidity of 20,000. Because a bankruptcy occurred very quickly, I set it to run 15,000 rounds so it could run for longer and the parallelizability could be observed better. Figure 5 summarizes the results.

This evaluation had a very strange outcome. On one thread, it got progressively slower when increasing the number of computing nodes. Another strange observation is that two threads delivered worse execution speed than one thread, but experienced faster execution speed when using more computing nodes.

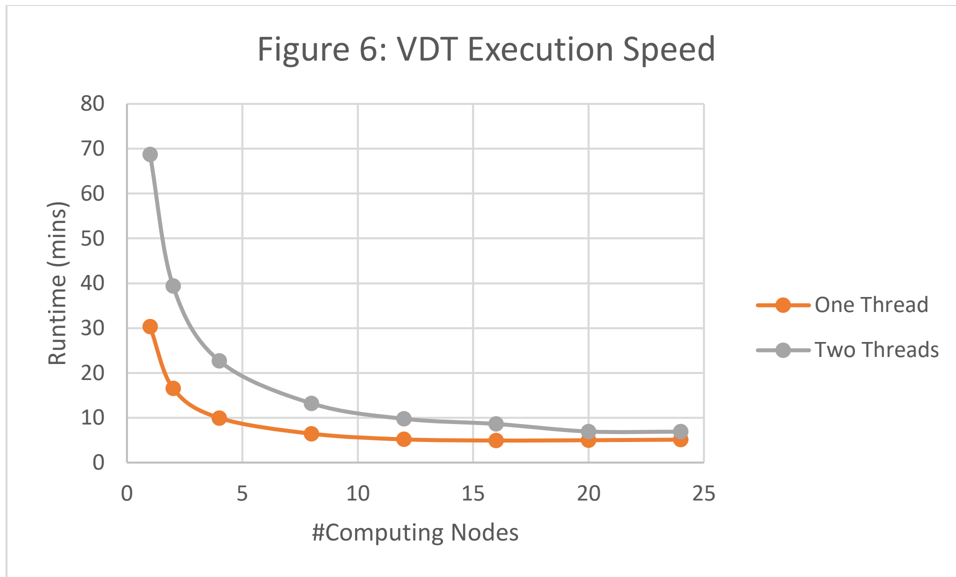


Virtual Development Team

Virtual Development Team was executed with an input task file containing 10,000 production tasks and 7 collaboration tasks, and it used 24x24 matrix of teams. Teams were given 4 different configurations from a text file:

1. Configuration 1: Leads (3), UX Designers (4), Senior Software Developer (5), Junior Software Developer (6), and Tester (7).
2. Configuration 2: Leads (2), UX Designers (4), Senior Software Developer (6), Junior Software Developer (8), and Tester (5).
3. Configuration 3: Leads (5), UX Designers (5), Senior Software Developer (5), Junior Software Developer (5), and Tester (5).
4. Configuration 4: Leads (6), UX Designers (5), Senior Software Developer (4), Junior Software Developer (5), and Tester (5).

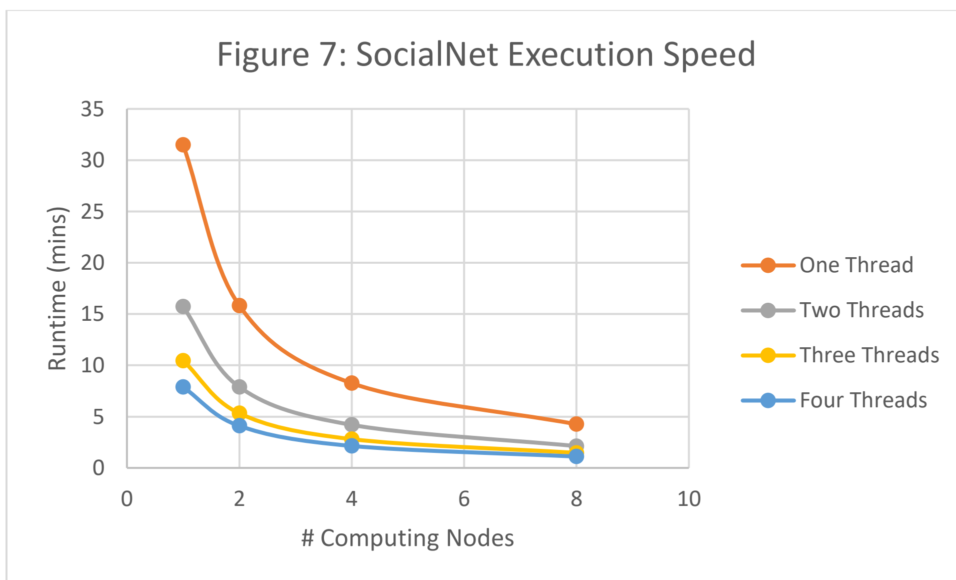
The results of these runs are displayed visually in the graph in Figure 6, demonstrating that the runtime went from 30 minutes with 1 computing node to just 5 minutes with 24 computing nodes. The complete tables used to construct this graph are available in Appendix A.



As is visible from the results of the run, the program actually runs slower on two threads than one thread. This is most likely due to an implementation issue of the benchmark program when dealing with critical sections and race conditions.

Social Network

Social Network was ran using a social network of 6,000 people each with 30 first-degree friends and looking for the 60th degree of friendship. These input arguments were used to execute the program on 1-8 computing nodes and 1-4 threads. The program was ran 3 times on each setting and the average of the runtimes was used to generate the graph in Figure 7. The table of each individual setting's runtime can be found in Appendix A.



Based on the graph, this benchmark program experiences a considerable reduction in runtime when using multiple processes and multithreading.

Conclusion

With these last four benchmark programs, MASS C++ now has all seven benchmark programs implemented and ready to evaluate. Appendix C acknowledges some flaws with these programs and sheds some light on some specific cases that may cause issues with these programs.

The next steps are for this project will be to evaluate all seven benchmark programs in terms of execution speed and programmability and compare the results to RepastHPC and Flame's benchmark programs. I am graduating this quarter, therefore will not be involved in this part of the process, but another student will be starting these evaluations next quarter. Appendix B gives detailed instructions on how to run these four benchmark programs, which will hopefully help the evaluations get started faster.

Appendix A: Execution Runtime Tables

The following tables display the different runtimes collected from running Bail-In/Bail-Out (table 5), VDT (table 6), and Social Network (table 7). Social Network was experiencing issues running on 9 or more computing nodes due to the size of the social network, so results were only collected for 1-8 computing nodes. However, Social Network's results are more reliable because each setting was repeated 3 times to produce an average runtime across the 3 runs. Similarly, Bail-In/Bail-Out had the same issue running on 9 or more computing nodes when the number of rounds requested was very large.

Table 5: Bail-In/Bail-Out Execution Results

One Thread		
Computing Nodes	Runtime (Microseconds)	Runtime (Minutes)
1	813548192	13.56
2	917869711	15.30
4	888352303	14.81
8	984405126	16.41
Two Threads		
Computing Nodes	Runtime (Microseconds)	Runtime (Minutes)
1	1486716503	24.78
2	1358628932	22.64
4	1102354839	18.37
8	1095992634	18.27

Table 6: VDT Execution Results

One Thread		
Computing Nodes	Runtime (Microseconds)	Runtime (Minutes)
1	1819802728	30.33
2	993540729	16.56
4	597151325	9.95
8	386432994	6.44
12	310718073	5.18
16	295463275	4.92
20	299134471	4.99
24	307124345	5.12
Two Threads		
Computing Nodes	Runtime (Microseconds)	Runtime (Minutes)
1	4545308701	68.76
2	2363594180	39.40

4	1361565164	22.70
8	791901644	13.20
12	585945742	9.77
16	516674615	8.61
20	416074910	6.93
24	413584386	6.89

Table 7: SocialNet Execution Results

One Thread					
Computing Nodes	Run 1 (μs)	Run 2 (μs)	Run 3 (μs)	Average (μs)	Average (mins)
1	1902898557	1876038847	1891407297	1890114900	31.50
2	948942114	952113309	947541237	949532220	15.83
4	495676658	496453668	494412358	495514228	8.26
8	254928253	254848755	256831371	255536126.3	4.26
Two Threads					
Computing Nodes	Run 1 (μs)	Run 2 (μs)	Run 3 (μs)	Average (μs)	Average (mins)
1	943001492	943250705	943072219	943108138.7	15.72
2	473075356	474833397	472895019	473601257.3	7.90
4	251617461	251647501	252060005	251774989	4.20
8	127304005	127921676	127444620	127556767	2.13
Three Threads					
Computing Nodes	Run 1 (μs)	Run 2 (μs)	Run 3 (μs)	Average (μs)	Average (mins)
1	629352124	627016792	625870869	627413261.7	10.46
2	319017893	319363791	319768563	319383415.7	5.32
4	168117694	168154116	168439246	168237018.7	2.80
8	86005741	85227401	92598584	87943908.67	1.47
Four Threads					
Computing Nodes	Run 1 (μs)	Run 2 (μs)	Run 3 (μs)	Average (μs)	Average (mins)
1	472751827	505578886	502813839	493714850.7	8.23
2	245946187	255379109	255010544	252111946.7	4.20
4	128516320	134937031	185807283	149753544.7	2.50
8	66418928	66421894	66431744	66424188.67	1.11

Appendix B: How to Run the Benchmark Programs

This appendix provides all the details required to run the benchmark programs. All four benchmark programs contain all the necessary source files and shell files in their respective Bitbucket repositories. However, some of these files may need to be edited based on your directory layout so that they can link to the MASS core library properly.

Configuring Your File Directory

Before starting this section, make sure you have the latest version of the benchmark program from Bitbucket as well as the latest copy of the mass_cpp_core library. Once you have both the program and the library on your machine, you may begin the following steps:

1. Create symbolic links to the following three files in the mass_cpp_core directory:
    a. /mass_cpp_core/ubuntu/libmass.so
    b. /mass_cpp_core/ubuntu/mprocess
    c. /mass_cpp_core/ubuntu/killMProcess.sh
2. Find compile.sh in the benchmark program's directory. Edit line 2 of this file to point to the mass_cpp_core directory in relation to compile.sh's location. For example, if the mass_cpp_core folder is located one directory back from compile.sh, the line will read "export MASS_DIR=../mass_cpp_core".
3. Find run.sh in the benchmark program's directory. Edit line 1 of this file to point to the mass_cpp_core/ubuntu folder in relation to run.sh's location. For example, if the mass_cpp_core folder is located one directory back from run.sh, the line will read "LD_LIBRARY_PATH=../mass_cpp_core/ubuntu".
4. Make sure machinefile.txt lists all the machines you want to establish a connection with during execution. The file provided on Bitbucket lists hermes02-hermes12 and cssmpi1h-cssmpi12h and assumes that hermes01 is the master node. The master node should always be excluded from this file.
5. Compile the benchmark's source files with "./compile.sh" to ensure all executable files are up to date.

Once the above steps are complete, you are ready to execute the benchmark program. For benchmark-specific information, see the following sections.

Bail-In/Bail-Out Input Setup and Execution

This benchmark program can be found at:
https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/Ian_MASS_Benchmarks/Benchmarks/MASS/MASS_FinancialModeling/

This benchmark program requires only one additional text file to run and contains information on the number of agents to instantiate as well as initial values. This text file is named “simArgs.txt” by default. An example is given below to demonstrate how this file should be formatted:

```
Workers 20000
Firms 2000
Banks 5
Production 35000
Interest 0.8
Liquidity 20000
```

When you have the simulation arguments you want, use the following instructions to execute the program:

1. Execute run.sh with “./run.sh”.
2. Enter the following arguments when prompted:
    a. Number of Nodes – the number of computing nodes (machines) you want to use to run your program.
    b. Number of Threads – the number of threads to use.
    c. Number of Turns – provides the number of rounds the program will execute the simulation loop for. It is guaranteed to run this number of rounds regardless of when a bankruptcy happens.
    d. Port Number - the port number you want to use for communicating between remote computing nodes.
    e. Password - your UW password for the hermes machines. It is recommended that you establish an RSA key so that you can authenticate remote computing nodes without having to pass your password between remote computing nodes.
3. If the program terminates for any reason before MASS::finish is called, execute ./killMProcess.sh to clean up the processes on the other computing nodes before attempting to execute again.

Virtual Development Team Input Setup and Execution

This benchmark can be found at:
https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/Ian_MASS_Benchmarks/Benchmarks/MASS/MASS_VDT/

This benchmark file requires two text files, one to list the tasks for the engineers to complete and another to list the different combinations of each type of engineer for a team.

The task file should be formatted in two sections: the top section for collaboration tasks, and the bottom section for production tasks. Here is an example:

```
Collaboration

toMeet:h1,h2,h3,h4,h5, Type:1, Priority:9999, Length:L, ID:I, Day:d


Production

Hours:h1,h2,h3,h4,h5, Type:0, Priority:18, totalHours:T, ID:I
```

Under the “toMeet” and “Hours” section, h1 refers to the number of hours for project leads, h2 refers to the number for of user experience designers, and so on. Length L should equal the sum of all hours in the “toMeet” section, and totalHours T should equal the sum of all hours in the “Hours” section. Another thing to note is that the IDs of the tasks should be unique, and not repeated for collaboration tasks and production tasks.

The next text file needed to run is the team configuration file, which should be formatted as follows:

```
#Configurations

#Leads,#UX,#Srs,#Jrs,#Testers
```

Here is an example of the configuration file that was used to collect the evaluative data in this paper:

```
4
3,4,5,6,7
2,4,6,8,5
5,5,5,5,5
6,5,4,5,5
```

If you would like to generate a new task file, it is recommended that you use the program located in the Bitbucket repository at: https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/Nathan_Repast_HPC/Repast_VDT_Final/InputFileVDT/.

Once the two files are in the directory, follow the steps to execute the program:

1. Execute run.sh with “./run.sh”.
2. Enter the following arguments when prompted:
    a. Number of Nodes – the number of computing nodes (machines) you want to use to run your program.
    b. Number of Threads – the number of threads to use.

c. XSIZE – the number of teams to instantiate along the x-axis (i.e. columns in the places matrix). Note that the places matrix is split across this axis.

d. YSIZE – the number of teams to instantiate along the y-axis (i.e. rows in the places matrix).

e. Port Number - the port number you want to use for communicating between remote computing nodes.

f. Password - your UW password for the hermes machines. It is recommended that you establish an RSA key so that you can authenticate remote computing nodes without having to pass your password between remote computing nodes.

3. If the program terminates for any reason before MASS::finish is called, execute ./killMProcess.sh to clean up the processes on the other computing nodes before attempting to execute again.

Social Network Input Setup and Execution

This benchmark program can be found at:
https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/Ian_MASS_Benchmarks/Benchmarks/MASS/MASS_SocialNetwork/

Social Network does not require any text files as input. Therefore, you can immediately run the following instructions after compiling your program:

1. Execute run.sh with "./run.sh".
2. Enter the following arguments when prompted:

a. Number of Nodes – the number of computing nodes (machines) you want to use to run your program.

b. Number of Threads – the number of threads to use.

c. Number of People – the number of people you want in the social network (number of vertices in the k-regular graph). Note that either the number of people or the number of first-degree friends (or both) must be an even number for a k-regular graph to be generated.

d. Number of first-degree friends – the number of first-degree friends you want each person to have (the number of edges in the k-regular graph). Note that either the number of people or the number of first-degree friends (or both) must be an even number for a k-regular graph to be generated.

e. Fraction of the group – this value will be used to calculate the number of first-degree friends in the social network by dividing the number of people in

the group by this number. If you already entered a value for the number of first-degree friends, you can enter -1 here to have this argument ignored.

f. Port Number - the port number you want to use for communicating between remote computing nodes.

g. Password - your UW password for the hermes machines. It is recommended that you establish an RSA key so that you can authenticate remote computing nodes without having to pass your password between remote computing nodes.

3. If the program terminates for any reason before MASS::finish is called, execute ./killMProcess.sh to clean up the processes on the other computing nodes before attempting to execute again.

Multi-Agent Transport Simulation Input Setup and Execution

This benchmark program can be found at: https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/Ian_MASS_Benchmarks/Benchmarks/MASS/MASS_MatSim/

This benchmark requires two input files to run the program: Car_Agents.txt and Node_Links.txt.

Car_Agents.txt lists the cars' start and endpoints in the roadway graph and provides the route the car will take to get from the start to endpoint by listing the road IDs.

Node_Links.txt is the input file to generate the roadway graph, assigning IDs to intersections and roadways and indicating which intersections are connected.

To ensure these files are accurate, please use Nathan's program to automatically generate these text files located on Bitbucket at: https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/Nathan_Repast_HPC/Repast_MATSIM_Final/InputFileMATSIM/.

Once the text files are generated, use the steps below to execute the program:

1. Execute run.sh with "./run.sh".
2. Enter the following arguments when prompted:

   a. Number of Nodes – the number of computing nodes (machines) you want to use to run your program.

   b. Number of Threads – the number of threads to use.

   c. XSIZE – the number of intersections along the x-axis in the roadway graph. This value must be consistent with the Node_Links.txt file.

d. YSIZE – the number of intersections along the y-axis in the roadway graph. This value must be consistent with the Node_Links.txt file.

e. NumAgents – The number of car agents to instantiate on the places. This value must be consistent with the Car_Agents.txt file.

f. Port Number - the port number you want to use for communicating between remote computing nodes.

g. Password - your UW password for the hermes machines. It is recommended that you establish an RSA key so that you can authenticate remote computing nodes without having to pass your password between remote computing nodes.

3. If the program terminates for any reason before MASS::finish is called, execute ./killMProcess.sh to clean up the processes on the other computing nodes before attempting to execute again.
4. If the program terminates before File2ShmPlace::free has been called, the ipc shared memory segments will remain. Therefore, you must execute deleteShm.sh to remove all these segments from all the computing nodes.

Appendix C: Known Execution Issues

Currently, there are some issues that may cause the benchmark programs to crash or run more slowly during execution. These issues are described in the following sections.

Bail-in/Bail-Out:

- Crashes when running longer than 10-15 minutes (~20,000 rounds).
- Crashes when trying to run on 9 or more computing nodes when requesting to run a large number of rounds.

VDT:

- Currently takes longer to run on multiple threads than it does for one thread. The suspected cause of this is implementation issues with mutex locks causing a decrease in performance.

Social Network:

- Crashes when using 9 or more computing nodes when the social network or the number of first-degree friends is very large. To prevent this issue, try reducing the size of the social network.

MATSim:

- Crashes when sending the vacancy information between intersections on different computing nodes.

Appendix D: Lines of Code Counting

This section provides the output of the "LoC.java" program that counted the number of lines of code for each benchmark programs' source files. Note that Timer.h and Timer.cpp were excluded because they were only necessary for the evaluation of the program, not necessary for the benchmark itself to run.

Additionally, this section also gives the breakdown for the number of methods and library-specific boilerplate lines of code. The library-specific boilerplate lines of code is any code that is required for the library to run that is not related to the implementation of the benchmark program itself. This includes things such as function Id's, each class's callMethod section, and extern C functions. This count excludes whitespace and lines containing only the closing curly brace.

The number of methods excluded constructors and the callMethod function since those are considered boilerplate methods.

Bail-In/Bail-Out

Lines of Code program output:

```
Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Bank/Bank.cpp
Total:                 146
Definitions (imports): 8
Comments:              39
Blanks:                21
LoC of actual logic:   78

Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Bank/Bank.h
Total:                 136
Definitions (imports): 26
Comments:              55
Blanks:                31
LoC of actual logic:   24

Results of:          /home/NETID/imdudder/mass-
Ian/Loc_count/Bank/FinancialMarket.h
Total:                 257
Definitions (imports): 53
Comments:              94
Blanks:                58
LoC of actual logic:   52

Results of:          /home/NETID/imdudder/mass-
Ian/Loc_count/Bank/FinancialMarket.cpp
Total:                 259
Definitions (imports): 20
Comments:              80
Blanks:                34
LoC of actual logic:   125
```

```
Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Bank/Firm.cpp
Total:                 174
Definitions (imports): 12
Comments:              42
Blanks:                19
LoC of actual logic:   101

Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Bank/Firm.h
Total:                 139
Definitions (imports): 24
Comments:              55
Blanks:                35
LoC of actual logic:   25

Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Bank/main.cpp
Total:                 179
Definitions (imports): 21
Comments:              30
Blanks:                24
LoC of actual logic:   104

Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Bank/Owner.cpp
Total:                 11
Definitions (imports): 2
Comments:              1
Blanks:                5
LoC of actual logic:   3

Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Bank/Owner.h
Total:                 26
Definitions (imports): 4
Comments:              9
Blanks:                4
LoC of actual logic:   9

Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Bank/Worker.cpp
Total:                 62
Definitions (imports): 6
Comments:              18
Blanks:                9
LoC of actual logic:   29

Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Bank/Worker.h
Total:                 89
Definitions (imports): 11
Comments:              30
Blanks:                20
LoC of actual logic:   28

All files combined
Number of files:       11
Total:                 1478
Definitions (imports): 187
```

```
Comments:               453
Blanks:                 260
LoC of actual logic:    578
```

Number of Methods/Functions (per class):

- Bank: 9
- Financial Market: 15
- Firm: 12
- Main: 3 additional functions
- Owner: 2
- Worker: 5
- Total: 46

Library-Specific Boilerplate LoC (per file)

- Bank.h: 22
- Bank.cpp: 4
- FinancialMarket.h: 49
- FinancialMarket.cpp: 4
- Firm.h: 13
- Firm.cpp: 4
- Main.cpp: 25
- Worker.h: 10
- Worker.cpp: 4
- Total: 135

Virtual Development Team

Lines of Code program output:

```
Results of:            /home/NETID/imdudder/mass-Ian/Loc_count/VDT/Engineer.cpp
Total:                  86
Definitions (imports): 5
Comments:               17
Blanks:                 13
LoC of actual logic:    51

Results of:            /home/NETID/imdudder/mass-Ian/Loc_count/VDT/main.cpp
Total:                  298
Definitions (imports): 19
Comments:               71
Blanks:                 25
LoC of actual logic:    183

Results of:            /home/NETID/imdudder/mass-Ian/Loc_count/VDT/Engineer.h
Total:                  86
Definitions (imports): 17
Comments:               30
```

```
Blanks:                 18
LoC of actual logic:    21

Results of:            /home/NETID/imdudder/mass-Ian/Loc_count/VDT/Task.cpp
Total:                  115
Definitions (imports): 2
Comments:               19
Blanks:                 26
LoC of actual logic:    68

Results of:            /home/NETID/imdudder/mass-Ian/Loc_count/VDT/Task.h
Total:                  97
Definitions (imports): 13
Comments:               39
Blanks:                 24
LoC of actual logic:    21

Results of:            /home/NETID/imdudder/mass-Ian/Loc_count/VDT/Team.cpp
Total:                  301
Definitions (imports): 14
Comments:               75
Blanks:                 33
LoC of actual logic:    179

Results of:            /home/NETID/imdudder/mass-Ian/Loc_count/VDT/Team.h
Total:                  225
Definitions (imports): 43
Comments:               85
Blanks:                 49
LoC of actual logic:    48

All files combined
Number of files:        7
Total:                  1208
Definitions (imports): 113
Comments:               336
Blanks:                 188
LoC of actual logic:    571
```

Number of Methods/Functions (per class):

- Engineer: 4
- Team: 14
- Task: 18
- Main: 3 additional functions
- Total: 39

Library-Specific Boilerplate LoC (per file)

- Engineer.h: 13

- Engineer.cpp: 4
- Team.h: 40
- Team.cpp: 4
- Main.cpp: 23
- Total: 84

Social Network

Lines of Code program output:

```
Results of:          /home/NETID/imdudder/mass-
Ian/Loc_count/SocialNet/main.cpp
Total:                 173
Definitions (imports): 18
Comments:              39
Blanks:                21
LoC of actual logic:   95

Results of:          /home/NETID/imdudder/mass-
Ian/Loc_count/SocialNet/Person.cpp
Total:                 112
Definitions (imports): 14
Comments:              22
Blanks:                16
LoC of actual logic:   60

Results of:          /home/NETID/imdudder/mass-
Ian/Loc_count/SocialNet/Person.h
Total:                 111
Definitions (imports): 23
Comments:              36
Blanks:                27
LoC of actual logic:   25

All files combined
Number of files:       3
Total:                 396
Definitions (imports): 55
Comments:              97
Blanks:                64
LoC of actual logic:   180
```

Number of Methods/Functions:

- Person: 7
- Main: 3 additional functions
- Total: 10

Library-specific Boiler-plate LoC:

- Person.h 22
- Person.cpp 4
- Main.cpp: 18
- Total: 44

MATSim

Lines of Code program output:

```
Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Matsim/Car.cpp
Total:                 247
Definitions (imports): 12
Comments:              47
Blanks:                24
LoC of actual logic:   164

Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Matsim/Car.h
Total:                 143
Definitions (imports): 26
Comments:              58
Blanks:                30
LoC of actual logic:   29

Results of:          /home/NETID/imdudder/mass-
Ian/Loc_count/Matsim/File2ShmPlace.cpp
Total:                 61
Definitions (imports): 8
Comments:              9
Blanks:                6
LoC of actual logic:   38

Results of:          /home/NETID/imdudder/mass-
Ian/Loc_count/Matsim/File2ShmPlace.h
Total:                 67
Definitions (imports): 17
Comments:              20
Blanks:                11
LoC of actual logic:   19

Results of:          /home/NETID/imdudder/mass-
Ian/Loc_count/Matsim/Intersection.cpp
Total:                 230
Definitions (imports): 18
Comments:              60
Blanks:                28
LoC of actual logic:   124

Results of:          /home/NETID/imdudder/mass-
Ian/Loc_count/Matsim/Intersection.h
Total:                 186
Definitions (imports): 45
Comments:              71
```

```
Blanks:                  40
LoC of actual logic:     30

Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Matsim/main.cpp
Total:                   176
Definitions (imports): 17
Comments:                40
Blanks:                  26
LoC of actual logic:     93

Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Matsim/Road.cpp
Total:                   48
Definitions (imports): 2
Comments:                2
Blanks:                  11
LoC of actual logic:     33

Results of:          /home/NETID/imdudder/mass-Ian/Loc_count/Matsim/Road.h
Total:                   37
Definitions (imports): 5
Comments:                4
Blanks:                  6
LoC of actual logic:     22

All files combined
Number of files:         9
Total:                   1195
Definitions (imports): 150
Comments:                311
Blanks:                  182
LoC of actual logic:     552
```

Number of Methods/Functions (per class):

- Car: 11
- File2ShmPlace: 2
- Intersection: 12
- Main: 2 additional functions
- Road: 11
- Total: 38

Library-Specific Boilerplace LoC (per file):

- Car.h: 19
- Car.cpp: 4
- File2ShmPlace.h: 10
- File2ShmPlace.cpp: 4
- Intersection.h: 37
- Intersection.cpp: 4

- Main.cpp: 26
- Total: 104