Final Report

Benjamin Phan, CSS497 (August 31, 2016)

For my capstone project, I compared Mass against other agent based simulation (ABS) libraries, including RepastHPC and FLAME, on the basis of programmability and performance using an application called ‘RandomWalk’. Implementing the same program with the same application level logic over the three platforms would give a fair comparison along the grounds of an agent-based model with the agents moving in a space.

Table of Contents

Agent-based simulation system comparison

In addition to the work described in last quarter's term report, I ended up comparing MASS versus two other C/C++ agent-based simulations (ABS) frameworks, RepastHPC and FLAME, in order to reveal the efficiency of MASS C++ version. Originally, I was to compare MASS to other generally used Agent-Based modeling libraries, including DMASON and NetLogo, but ended up narrowing the selection to C/C++ based libraries to keep in line with ergonomics and performance at the language level. As with most ABS libraries, all three can run models in a distributed environment, and synchronize the model with the concept of iterations. Comparing not only on the basis of performance, I also took into consideration the programmability of each framework—or the effort required to develop a model with the corresponding framework.

Comparison

To compare the frameworks, I implemented an agent-based simulation called ‘Randomwalk’, in which agents move randomly around in a bounded space, attempting to move without collision each turn. I compared the three ABM frameworks along the lines of performance and programmability, to gauge which library a user might utilize given performance against difficulty in using the library. This assumes the user would use C/C++ for performance, and would run their models on distributed systems.

- Programmability: How hard is it to create a program using the library? This includes the difficulty of installations, dependencies, building, run, etc.
- Performance: What is the scale and speed of which the framework can support the model?

Randomwalk

Randomwalk is an agent-based model in which agents, called Nomads, are spawned at the center of a map, made of Land, after which they attempt to move randomly to an unoccupied, adjacent position on the map each turn. By adjacency, Randomwalk specifies the immediate north, east, south, west neighbor; thus excluding diagonal moves. The caveat is that the map is limited in size, and that the Nomads must not collide with each other. One of the purposes of this application is to eventually be able to evolve into an evacuation model, where the Nomads represent people in a building or city trying to evacuate, from say a fire, or a tsunami respectively. Currently in Randomwalk, at the start of the simulation, the Nomads all initialize in an inner square at the center of the map.

At first, based on the clause of no collision, it seemed that the collision avoidance would need to be implemented by Nomad agents to detect the movement of other nearby Nomad agents to detect and avoid collision. When Nomad agents move to an adjacent square, not only do they need to check that it is unoccupied to avoid collision, but also that no other Nomad will move into that square from two squares away too—collision detection and avoidance (figure-1). This lead to a strategy of space reservation each turn, where a Nomad would reserve a space and be confirmed that it is allowed to move

to the coordinate each turn, to ensure collision free movement. Unfortunately, the message interface of RepastHPC and MASS C++ had undocumented issues, and so this was not a viable option for comparison.

Rather than collision avoidance per se, I implemented a collision prevention algorithm that was approximately equally implementable in all three libraries for comparison. The orthogonal partitioning of movement space consists of for each turn, splitting the space into non-overlapping movement spaces, such that for iterations in the turn, all agents of a particular position in the movement space can move (figure-1). This movement-space consists of all possible moves any particular agent can move, and by making sure the movement spaces do not overlap, the agents can move without collision. For which iteration in a turn the Nomad moves, is calculated by mod x and mod y * 3 at the start of each turn, consuming a movement when it moves, as mod x and mod y * 3 change after movement. Since the movement range of a Nomad in Randomwalk is one, the movement space consists of a 3-by-3 square, thus x % 3 + y % 3 * 3 determines a unique position in the movement space of an agent, for all movement spaces on the map.

Movement Space: Each agent in RandomWalk may move to one adjacent unoccupied square per turn. Thus, for each agent the movement space consists of all possible coordinates it can move to at that turn.

Partitioned Space Movement: By ensuring for each turn calculation that the movement spaces for selected agents do not overlap, they are guaranteed not to collide. This is a form of orthogonal channel collision prevention.

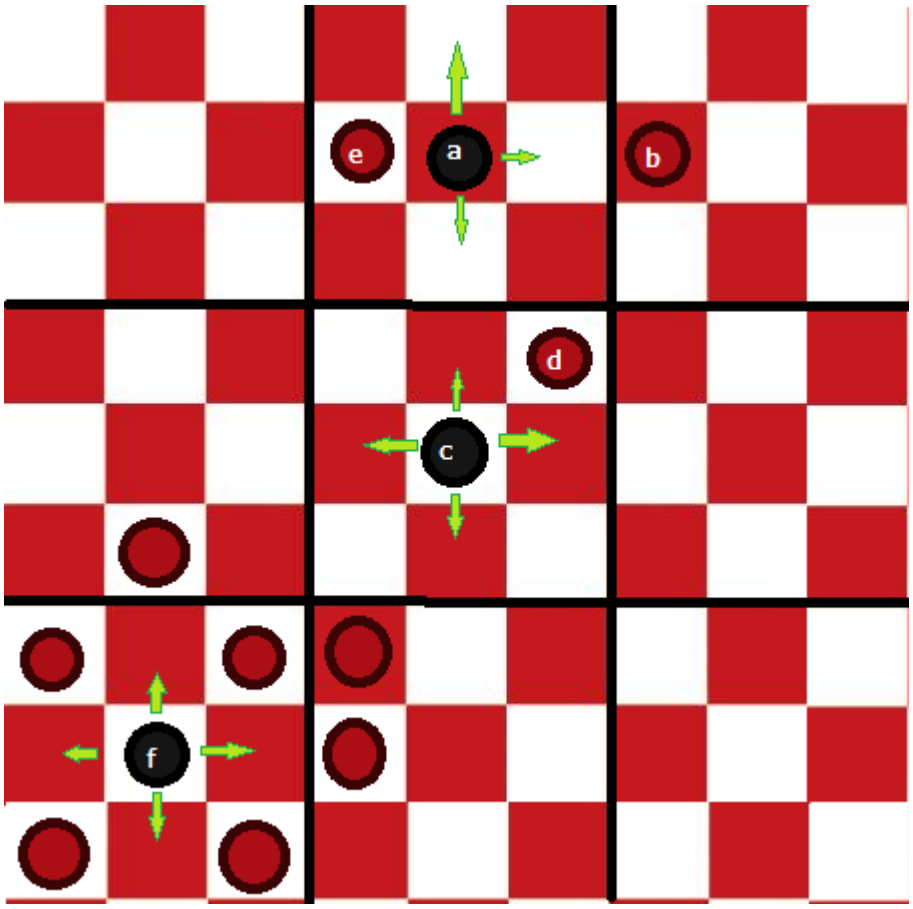


Figure 1: In this subiteration of the turn, all agents in the center square of the partitioning algorithm can move, marked as black checker pieces. The black checker pieces will move randomly in one of the green arrow directions without worrying about collision. Since they are at least two apart from each other, they have no chance of collision. Note how A can move right, and since the checkers in the same position of partition as B cannot move, A and B cannot collide.

To ensure the correctness of the implementation, simulations were also logged and unit-tested on all three frameworks compared. MASS' logging is the easiest to use; simply pass a string to Mass_base::log() and the string would be written out to a log file for that node. Meanwhile, RepastHPC uses an inflexible key-value based logging tool, which I figured out a workaround to record strings for the respective nodes. For FLAME, logs the entire simulation state every specifiable n-intervals, inconveniently creating a new .xml file each n-interval for each node—in other words, creating a new .xml file for the state of each node every n-intervals ran. Fortunately, I was able to create a specialized agent that logs the location of each Nomad in FLAME, letting me conveniently format and store only relevant information in my own logging file to bring in line with MASS and RepastHPC. Logging the locations of agents each step allows an agent's location to be traced throughout the simulation, and can be enabled/disabled—though the size of the file bloats for larger simulations, and the log-enable is still in the code files.

While logging allows visual inspection of an agent's migration throughout the simulation, ultimately, unit-testing is required to verify the correctness of the implementation. The unit-test I implemented involves hashing the locations of each agent at the end of each turn for collision, after which results are printed onto the screen and also logged. Fortunately, the unit-test, which was implemented after logging, passed for Randomwalk implemented in all three libraries, as suggested in log-file inspection.

Functionalities of the Three Frameworks

MASS C++

Not only does MASS support distributed agent based modeling, it also supports spatial modeling simulations (no moving agents, just space), and distributed computation. At the core of MASS is M++ threads, residing in each Place, which is kept in a distributed array called Places. Thus, each coordinate of the space in map contains a thread-scheduled Place on which agents can reside. These Place can also interact with each other, especially with message passing, to form agent-less spatial simulations. The Places array is distributed by column over the nodes running the simulation, in which

bordering Place coordinates can communicate information with each other about shadow-spaces—the neighbors at the border can directly exchange message with each other (see figure-3).

Agents and places in MASS C++ are extended to create custom agents and places, similar to the other two libraries. Here, they become Nomad and Land respectively. For initialization, I had to fix a bug for the Agents class that actually maps out the initial locations of the agents in the model, to allow Nomad to override it and customize the initialization. Fortunately, MASS C++ also has a GUI debugger and plenty of detailed models covering most of the library for reference.

RepastHPC

Like MASS, Repast comes in both a Java and C++ version, the C++ base version being RepastHPC. This would make a direct comparison against MASS C++ version. RepastHPC has several dependencies that must be installed, including Boost, MPI, and Curl, before it can be actually installed. While RepastHPC does contain tutorials about installation and model creation, they are inadequate, out of date, and at times even misleading. This is on top of the fact that RepastHPC pushes agent serialization and logging class extensions as user responsibilities, making the test model taking twice or more lines of code to develop. Repast also includes a Relogo extension to their RepastHPC library, though the extension was not necessary to complete Randomwalk.

Unfortunately, RepastHPC includes very few sample programs that touch only a small portion of its classes. Along with the lack of documentation, its class structure is convoluting. For instance, I spent a decent amount of time for logging, because there are several different classes involved in logging. The official logger class for Repast contains no documentation, and the tutorial for logging results refers to SVDataSet that records templated data sources, which unfortunately, takes only integer and double types for template. This was misleading as the comments in the class said it could take plain text. Fortunately, I was able to repurpose RepastHPC's Properties class, which originally is meant for populating a comma-separated-values file to actually log formatted text.

Space in RepastHPC is distributed as a ‘Shared Space’ between process nodes. Between each process is a concept called border-space, which is displayed in neighboring processes, much like MASS C++’s shadow space. Unlike in MASS C++, the border size can be specified on initialization of a Shared Space object (see figure-2 against figure-3).

Meanwhile, RepastHPC does contain support for network based simulations in which agents are networked to each other, and includes a rumor model as an example. Another interesting aspect of RepastHPC is for agents to exist simultaneously in multiple spaces, which may or may not be coterminous (overlay on top of each other directly), along with continuous space values, so that a space may either keep agent coordinate in integer or float-point format. They also have a large class tree for different types of spaces, grids, and even grid coordinate objects; unfortunately, this class tree is essentially undocumented.

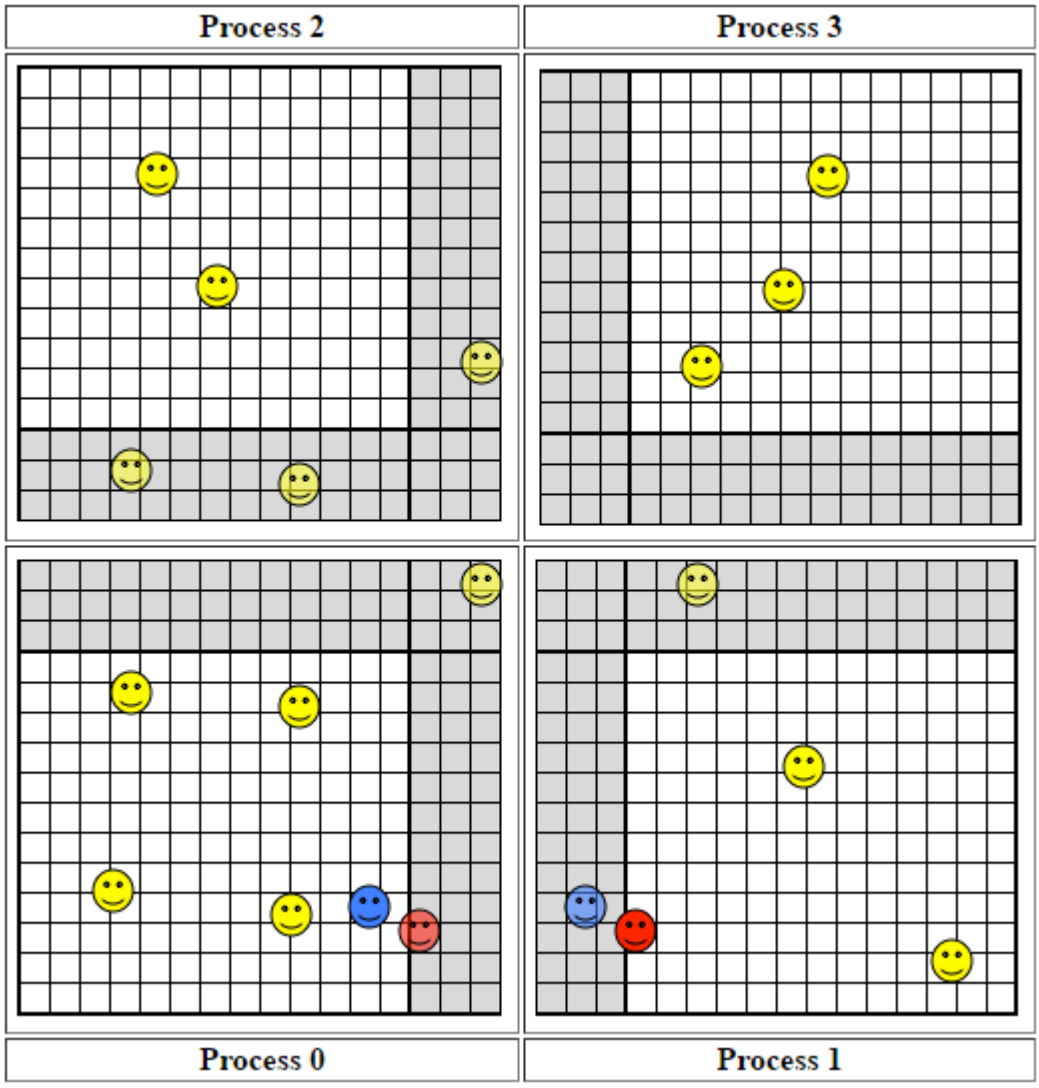


Figure 2: RepastHPC's buffer zones for space. Processes can see buffer-zone width of squares from adjacent ones in the shared space. This is why the red agent on Process 1 is seen by process 0, vice versa with the blue agent.

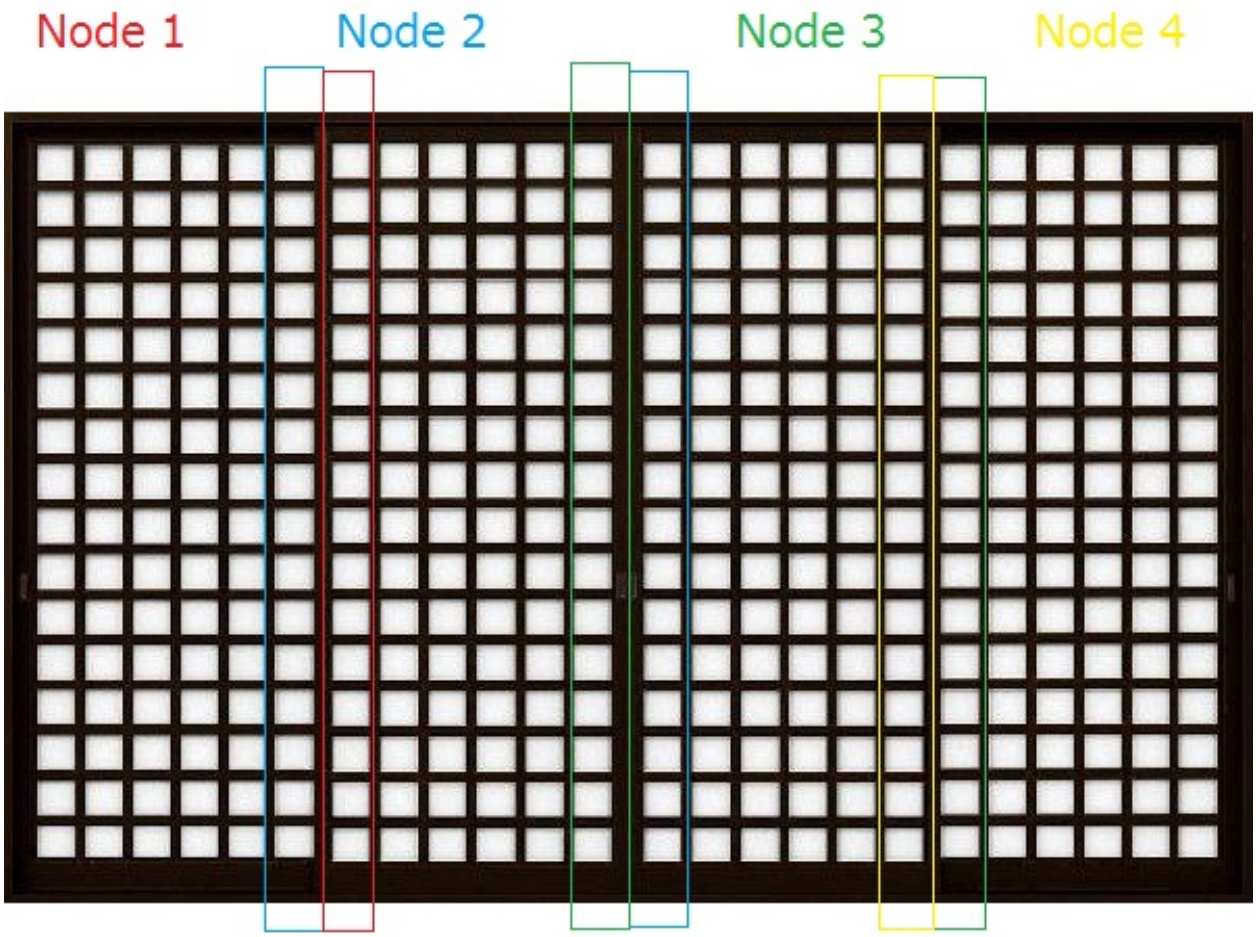


Figure 3: MASS' shadow space buffers are only one place wide, partitioning is always by column. Node 1 Place(s) in the blue rectangle column can exchange messages with those of node 2 in the red rectangle column. This is repeated for all columns as shown with by colored node names and corresponding shadow space columns.

FLAME

FLAME is C/XML based and is the last distributed ABM framework I compared with MASS C++. Like Repast, FLAME is also MPI based and thus needs MPI to be installed to run too.

FLAME's agents are declared in XML, which FLAME's xml parser can parse the agent's XML definition into a C-file. The implementation of the agents though, are written in separate C-files. This makes xmllint a desired tool for developing models in FLAME.

FLAME is unique from the rest of the models in that it is pure agent-based, that anything with mutable variables in the simulation must be declared as an agent of which you implement. Rather, space is tracked as a variable, which FLAME's parser automatically picks up as 'x', 'y', and optionally 'z'. These agents communicate with each other using messages shared over MPI.

FLAME also has a unique trait in that for each agent type, state transitions are required for each iteration, and are strictly enforced

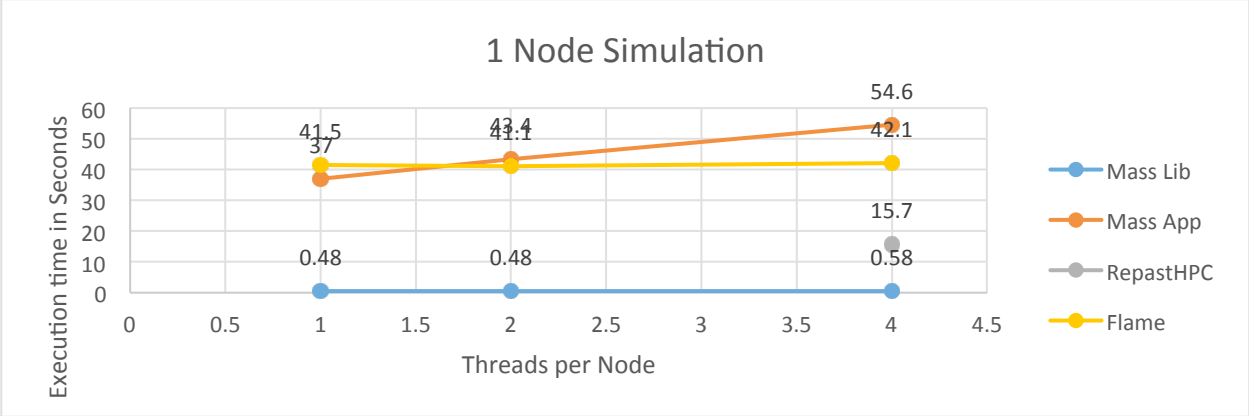
Results

Surprisingly, enabling or disabling the unit test and output had little effect on simulation runtime. Randomwalk was configured to spawn agents in the inner 20% by 20% of the map at initialization. Subsequently, the simulations were ran with unit test and output enabled. The simulations resulted as followed:

1 Node

I ran each for 2000 turns, with a 40x40 map.

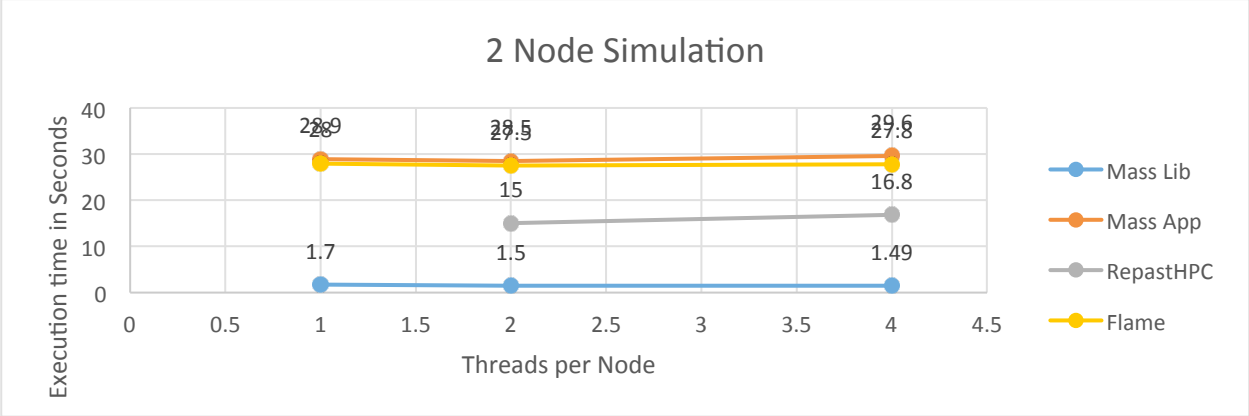
1 Node	1 Thread	2 Threads	4 Threads
Mass Lib	0.48	0.48	0.58
Mass App	37	43.4	54.6
RepastHPC			15.7
FLAME	41.5	41.1	42.1



2 Node

I ran each for 1000 turns, with a 60x60 map.

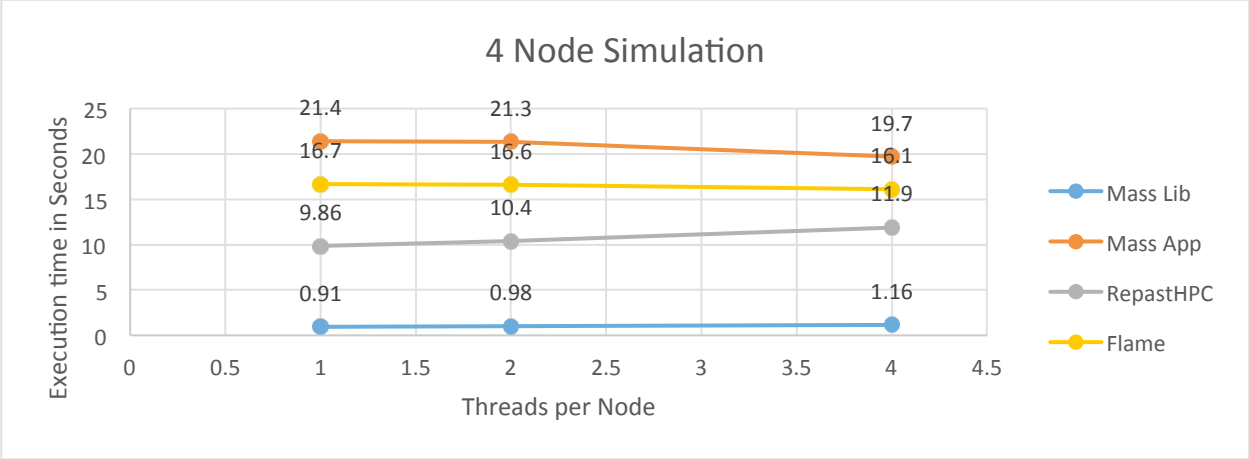
2 Node	1 Thread	2 Threads	4 Threads
Mass Lib	1.7	1.5	1.49
Mass App	28.9	28.5	29.6
RepastHPC		15	16.8
FLAME	28	27.5	27.8



4 Node

I ran each for 500 turns, with a 100x100 map.

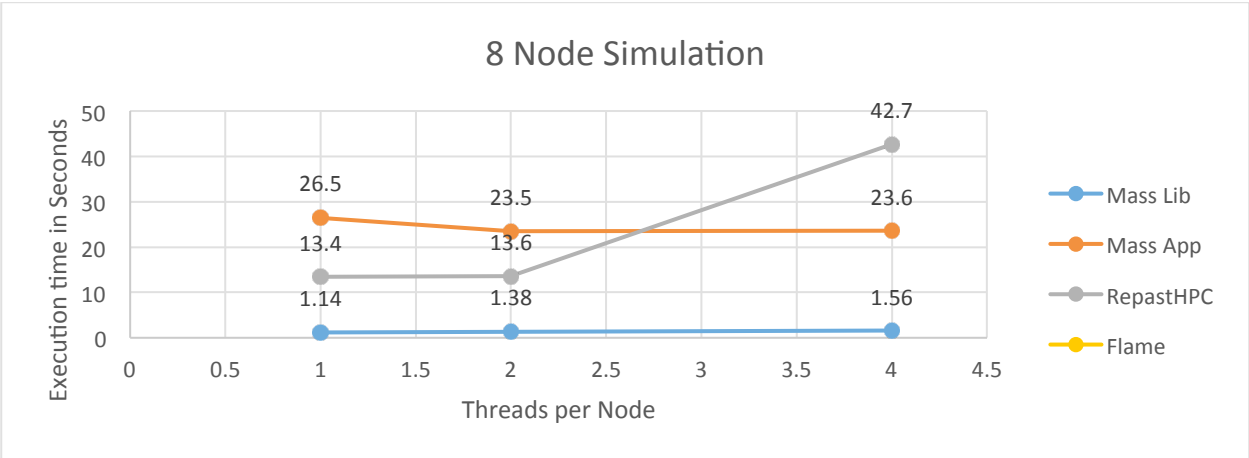
4 Node	1 Thread	2 Threads	4 Threads
Mass Lib	0.91	0.98	1.16
Mass App	21.4	21.3	19.7
RepastHPC	9.86	10.4	11.9
FLAME	16.7	16.6	16.1



8 Node

I ran each for 400 turns, with a 200x200 map.

8 Node	1 Thread	2 Threads	4 Threads
Mass Lib	1.14	1.38	1.56
Mass App	26.5	23.5	23.6
RepastHPC	13.4	13.6	42.7
FLAME			

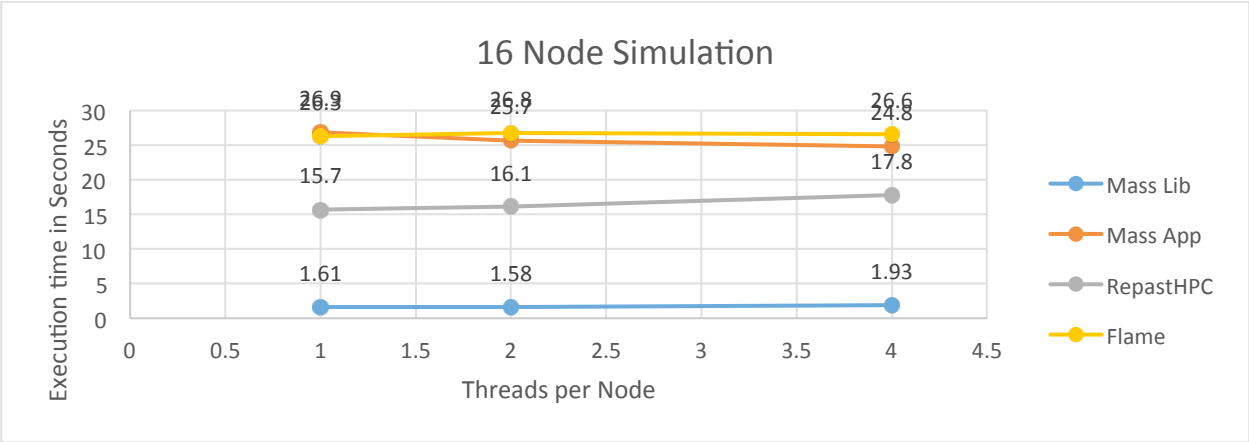


16 Node

I ran each for 300 turns, with a 300x300 map.

16 Node	1	2	4

	Thread	Threads	Threads
Mass Lib	1.61	1.58	1.93
Mass App	26.9	25.7	24.8
RepastHPC	15.7	16.1	17.8
FLAME	26.3	26.8	26.6



Note how RepastHPC fails if there is less than 4 processes, and FLAME breaks for 8 nodes.

As demonstrated in the results, MASS C++ outperforms all of the competitors by a factor of at least ten, using the internal collision prevention method. However, it is slightly slower when implementing a collision free migration scheme at the application level. MASS C++ also shows greater scalability in resilience to different number of nodes and processes per node.

Programmability

Out of all three frameworks, FLAME was the easiest to install. Meanwhile, MASS C++ and RepastHPC have pathing issues in their installation script, requiring the user to manually change path files in the scripts. RepastHPC also has several other dependencies, including Boost and MPI to run.

As for development, MASS C++ was the simplest to compile, link, and run a program. It just needs command arguments, which can be easily packaged into a compile.sh and run.sh file. Meanwhile, RepastHPC requires a long make file, with dependency paths listed, to compile and build a program, after which the executable can be run as an MPI

program. FLAME has the xparser that parses its agent declarations in XML, into C code, but there are syntax errors in the generated Rules.c file, which must be manually corrected if the XML declarations are modified. Only after that can FLAME actually build the model, with a functioning make file provided by xparser, creating an MPI executable model too. In the end, it was about even for FLAME and MASS C++ to develop a model such as Randomwalk. Since RepastHPC pushes more implementation dependencies on the user, such as serialization, it took twice as much code to develop Randomwalk in RepastHPC than MASS C++.

Thus, for simulations like randomwalk, involving massive numbers of agents migrating in space, MASS offers the strongest framework with decent programmability. With built in collision detection enabled with Kasey's implementation, the overall programmability of MASS exceeds FLAME.

Analysis

Mapping

Mass C++ distributes agents at the beginning of its application by a map function overridden by the agent to define where to place agents. Unfortunately, this is highly coupled with map dimensions input, and goes through a few layers of function calls starting from init in main.cpp. There was also a problem with the mapping function until I fixed it earlier the previous quarter, making it into a virtual override so that derived Agent like Nomad can define its own mapping algorithm. While it may seem cumbersome, this actually allows more control and distributes the initialization among the various processes in the simulation.

In RepastHPC, all processes, called ranks, inherit the same mapping function, because each holds a copy of the model instance. This complicates mapping, as one would have to define special cases for each process to map, the simplest being the 0^{th} rank initializing all of the agents. Again, since the shared space should only be created on one process, say the 0^{th}, it can create the context bag for agents and the corresponding space, then initialize and add agents to the space. To distribute to the rest of the ranks,

it merely needs to call synchronization of the space's context, which stores all the agents for the space.

FLAME does not provide documentation or tutorials for initializing large numbers of agents. Rather, I had to create a custom spawner agent that initializes all of the Nomad agents for the Randomwalk simulation, where FLAME includes instantiation for each agent natively. Unfortunately, all of the Nomads initialized by the spawner are placed onto a single process, and they do not migrate to other processes of the simulation during runtime. Rather, a single FLAME generated log of the starting state using the spawner must be used to copy data to a separate initial start state file '0.xml'. It is only on the initialization, from this initial state file, that agents are distributed among the various threads of the simulation.

Migration and Detection

Agents can move in MASS C++ simply by calling the Agent's move(), which reserves a place for the agent to move to, and updates it with the rest of the simulation on Agents' manageAll(). This is where Kasey's manage helpers can boost collision avoidance; otherwise, the agent would have to delegate the task of detecting occupancy of neighboring coordinates to the Place it is on, which exchanges occupancy information with all adjacent Places. This is just one strategy of many for detecting occupancy at the application level.

In RepastHPC, agents move by updating its location indices, in terms of x,y (z) coordinates, and then calling spatial synchronization functions to delegate them among the processes afterwards. They can detect each other by creating grid2d instances of either Moore or Von Neumann, to get number of agents around a radius of a certain point on the shared space. Unfortunately, this meant creating four for each adjacent square, to check the occupancy of each.

FLAME has all agents communicating with each other over a shared message space over MPI, from which they can filter messages by x,y (z) coordinates. Both agents and messages can have coordinates in FLAME, where messages can be filtered by x,y,z coordinates. Agents can then detect the occupancy by filtering location messages of

each agent, or in a special case, the logger agent can use the messages to log Nomad locations too. Unfortunately, this space does not scale to large functions, so Randomwalk in FLAME crashes when the map gets too large. Finally, agents do not migrate between processes during simulation runtime in FLAME.

Logging

Logging was the simplest in MASS C++, as merely calling MASS::log on a string would write it to the file corresponding to the process. Meanwhile, a custom logger agent had to be implemented for FLAME, and in Repast, it proved simplest to use its Properties writer, which normally logs simulation metadata, as a makeshift logger.

Mass C++

At the same time with superb agent-based spatial simulation performance, MASS C++ proved to be highly compact in terms of lines of code compared to the other languages. This enabled relatively easier development, though, parts of the library are declared and yet to be implemented, currently inhibiting its true potential.

Issues

- One of MASS C++'s issues has to do with Shadow-Space documentation. When Places exchange messages, they exchange with each of their neighboring coordinates, but at the edge of a partition, the messages cross nodes of the simulation. This is actually the only way in MASS to push messages across processes, but is poorly documented. Additionally, the range of this space is restricted to one, unlike in RepastHPC, which enables setting the size of the border space. On the other hand, getting out messages, that is, polling messages, will work across nodes.
- Places::PutInMessages() is not actually fully implemented yet.
- MASS installation has hard path dependencies; however, this should be resolved with Chris' work on installing and running MASS on AWS.

RepastHPC

RepastHPC is the largest of the three frameworks, with plenty of components for different flavors of simulations. Its lengthy tutorial and documentation only touched a fraction of the functionalities, meaning there was a large section of the library that I could not use. Repast also pushes a lot of responsibility to the implementation of the model, including extending appropriate Boost serialization for the agents.

- Separate serialization helper classes must be implemented for each agent, to move from node to node.
- While installing the dependent libraries worked as documented in the included ReadMe.txt, I had to modify the Make file extensively, updating paths, before I could install the library. Furthermore, the shell script or commands actually cannot build RepastHPC applications on the Linux Lab. Rather, I had to modify the make file for the example programs to build my tutorial programs and eventually RandomWalk.
- There is also an MPI bug in RepastHPC, when an agent bag (array, vector, etc) is reiterated for a second time in the same time-step. This duplicates agents and causes MPI to crash, because it is required that all of the agents have unique signatures.
- RepastHPC also has a scalability glitch that prevents the configuration file from declaring the node topography as shown in the Zombies example. While the additional processes are used as additional ranks for the simulation, only four of them contain agents. Strangely, RepastHPC also crashes when there are fewer than 4 processes.

Flame

FLAME is the smallest out of the three frameworks compared, requiring only MPI as an installed dependency. Since I installed MPICH with Repast, I used the same installation for FLAME too. I found FLAME's installation to be smooth, but inadequate documentation prevented it from being the most programmable library.

Issues

- FLAME is an XML defined, C based library, so Agent types are defined in XML, and their functions are implemented in C. Since it runs as a C program, the XML is parsed into C by FLAME's xparser program. Unfortunately, the C code generated by this has errors in rules.c. This means I have to fix this error every time I change the XML agent type definition.
- FLAME also has scant documentation, particularly on transitioning, which is defined in XML. While FLAME can produce transition graphs, it poorly explains the rules on transitioning. The transition conditions are actually entry, not end, conditions; and the agent type must transition from 'start' state to 'end' state.
- The next issue I encountered in FLAME is that, despite allowing agents to be created during simulation, all of these new agent instances are placed into a single node. Since in FLAME, agents do not migrate between processes, FLAME fails to scale with more nodes, unless the agents are all initialized at the start state '0.xml' file. Thus, I have a version where I spawn the nomads, extract the agent data from the generated XML log, and place the nomads into a separate 0.xml.
- FLAME also has a scalability issue, particularly, it cannot run RandomWalk long over 8 nodes, with any number of processes per node. Instead, it crashes mid-simulation, from poor MPI handling.
- Finally, FLAME has poor handling of simulation data over shared memory in MPI. As a result, it will crash if too many agents are added, or the map is too big.

Conclusion

MASS C++ outperformed FLAME and RepastHPC because of its unique concentration on spatial simulations, and its efficient, transparent strategy in parallelizing the model among the different nodes and processes running a simulation. This is especially true on the basis of scalability in a distributed environment, because MASS includes illustrations of its partitioning strategy and behaves so, whereas RepastHPC does not behave as it describes partitioning, even with illustrations (FLAME's partitioning is entirely undocumented). On the other hand, RepastHPC supports network simulations

and multiple simultaneous spaces, while FLAME is a lighter weight framework to begin with.

Appendix

This section contains brief explanations for running RandomWalk on the three frameworks. Since the tutorials, for the most part, give adequate details on installation, building, and running a program, this will cover the process as a brief overview. On my local directory ‘benp2’ it is possible just to run the application on any of the frameworks with run.sh. For RepastHPC and FLAME, it is imperative to have a path to MPI first, which is included with the appended ‘samplecommands’ files.

Log Files

RandomWalk also uses or implements logging utilities to log the results of simulation runs. The generated .txt files are the same format for RandomWalk on all 3 frameworks, with all logging enabled. This includes the turn, whether unit tests hold, and the locations of the agents.

Note, that some logs are append, so it is recommended to delete the previous log before running the next simulation run.

Hardware/Software Dependencies

All of the three frameworks compared need to run on a Unix machine, and are all designed for distributed agent based modeling simulations. MASS needs only a few libraries, and only if you use them in your application, such as NetCDF. RepastHPC needs an installation of MPI, Boost, CURL, but these are included in the RepastHPC download and the install script actually successfully installs these dependencies. FLAME requires MPI, so I used the MPICH included with RepastHPC running on my local directory.

Mass

I did not install MASS, but the documentation to build and run MASS applications with compile.sh and run.sh are accurate.

RepastHPC

The install script for RepastHPC works well for installing Repast's dependencies. Unfortunately, the make file must be modified, specifically, paths must be specified to various locations as explained in its ReadMe to successfully install the framework and samples applications. Building a program was not as simple as compiling with libraries as one would for a Linux shell script; rather, I had to copy and modify the Make file for the sample 'Rumor Model'—comments added during the process in the one I created for RandomWalk. The compiled and linked binaries can then be run as an MPI program—be sure to include the config and properties files, but those paths are accounted for in its Run.sh script.

FLAME

The FLAME installation, build, and run instructions are quite accurate. One thing to note is that in Rules.c generated by their XML parser, 'xparser', there is an issue with the agent parameter name that does not match its use in the function. The xparser generates a make file to compile the generated C source code for the model, which then can be run as an MPI program.

Summary

This work revealed that MASS C++ not only performs well against other agent based modeling frameworks, but also is quite programmable on its own terms. Many lessons can be drawn from this research:

- Thorough and accurate documentation, sample programs, and clear functionality decide the usability of a software framework.
- Illustrations help the user understand the underlying mechanics of the system, providing better utilization.
- Agent-based modeling frameworks have different areas of focus. For instance, MASS C++ also can run spatial simulations and distributed computing; whereas RepastHPC allows for network simulations and multi spatial agent contexts (agents existing in multiple spaces).

- Usability is a key factor in whether the tool is used or not. NetLogo may be the weakest ABM framework, but its simple GUI development, frequent updates, and active message boards make it one of the most popular.

Further development

While my capstone project may be complete, there can still be further development on this topic. We have not yet install the three frameworks and run RandomWalk on AWS. Further topics could different types of simulations, or such as pure spatial or network messaging simulations include.

Citation

Special acknowledgement to Dr. Fukuda and the team for helping me through the research project.

"D-MASON: Distributed Multi-Agents Based Simulations Toolkit." D-MASON: Distributed Multi-Agents Based Simulations Toolkit. ISISLab, n.d. Web. Apr. 2016. <https://sites.google.com/site/distributedmason/>

Greenough, Chris, and Holcombe. "Download Latest." FLAME Website. Science and Technology Facilities Council, 2013. Web. June 2016. <http://www.flame.ac.uk/>

"The Repast Suite." Repast Suite. ARGONNE NATIONAL LABORATORY, 2015 Web. Apr. 2016. <http://repast.sourceforge.net/>

Wilensky, Uri. "NetLogo Home Page." NetLogo Home Page. Northwestern University, 2016. Web. Apr. 2016. <http://ccl.northwestern.edu/netlogo/index.shtml>