# Background and Related Work

This section will present popular agent based frameworks and some parallel simulation frameworks. As a part of our initial investigations we have already tried to use these past frameworks. We are also using the experiences and observations to drive the design and implementation of the proposed simulation environment. In addition, background on various ideas and tools we used to make MUSE is explained.

## What are Agent-based models

………….talk about what agent-based modeling is…………..

## Message Passing Interface (MPI)

The message passing programming paradigm is the most well known and widely used approaches for programming parallel computers (Grama, Gupta and Karypis). One of the main reasons it spread fast is because it imposes minimal requirements on the underlying hardware (Grama, Gupta and Karypis). In the early stages, many hardware developers have implemented custom MPI-compliant libraries that performed efficiently for their own hardware. This required developers to know many different libraries of programming with the message passing paradigm. The Message Passing Interface (MPI) was developed to solve this issue of too many different implementations.

MUSE will be using MPI version 2.0 for the message passing requirements. We decided to adopt MPI because it is well documented and widely used. Moreover, MPI handles hardware-specific details on passing messages between interconnected compute nodes. Lastly, since MUSE will primarily operate on Linux based distributed machines; it would be a benefit to use MPI, because it is supported by most supercomputers.

## Choosing the Programming Language (C++)

There are many variables to consider when developing a simulation environment. One design decision to be made is the language in which we choose to implement. All simulation frameworks or libraries that we will examine section 2.4 and 2.5 use some of the better known languages for implementation. The three languages include C++, Java, and Objective-C. We eliminated Objective-C as a candidate due to the following two reasons. The main reason is the development tools are scarce. The only development tools that are easily accessible are provided via Apple’s Xcode, which requires an Apple machine. This differs dramatically from C++ and Java, which both have many freely available tools to choose from. Although, Objective-C is more natural to code with, it also lacks for the ability to catch errors easily (Railsback and Lytinen). Ultimately, semantic gap does not make a difference if you do not have users to realize the improvements; this is why Objective-C is not a reliable solution.

In order to identify between Java and C++, we empirically explored the semantic gap between C++ and Java, both in terms of computation and communication. Note that these two aspects are crucial for realizing effective performance improvements in distributed memory super computer architectures. A discussion on the semantic gap between the languages is presented in the following subsection.

### 2.2.1 Semantic Gap

In distributed computing, the logical distance from the hardware that your code executes onto the high-level semantics you use to code in the given language is called the semantic gap. In other words, the smaller the semantic gap, the more the developer must worry about hardware details, which could slow down development time. On the bright side, it could allow developers to realize great increase in speed by taking advantage of hardware design. Thus, semantic gap is needed because it increases development time, but a good balance will allow significant performance increase. C++ has an excellent balance because it has been designed with the hardware in mind. Fortunately, it is able convert high-level code to assembly efficiently. Also more importantly, C++ allows the use of registers; this allows all microprocessors to optimize execution speed using registers. Java on the other hand uses a stack based Java Virtual Machine (JVM). This means no registers can be used. The cost of this is portability. When Java compiles code, it is first converted to Java byte code, and then runs on the JVM. There are many systems that can effectively run without the need for optimization, but a parallel simulation environment is not one of them. There are two types of semantic gaps, the first being computational gap, and the second being communicational gap.

Figure 1 : C vs. Java Computation Speed

Computation gap was already discussed above, and figure 1 shows the difference in speed computation wise. The computation test used was matrix multiplication of an *NxN* matrix. Started with a 50x50 matrix and we ran both C and Java five times each and got the average with a 95% confidence interval. As the size of the matrix increased you can clearly see the speed difference in computation. One odd detail to notice about the graph is the time it takes 100x100 matrix to finish computing is greater than the time it takes for a 500x500 matrix. This is due to cache affects. However, we can still see that C still has a better time, which is consistent.

Communication gap refers to the steps that must be taken to convert the high-level communication to the hardware level. Java relies heavily on stream I/O. These streams are mapped to the hardware. The high-level abstraction again allows developers to code with greater speed, but the overhead for managing the streams can be very expensive in the long run. C++ allows developers to send different size of data, this increases speed because the underlying hardware may transmit data as packets, via C++ you can send data packet at a time. For Java it is fixed as bytes, you can easily see the overhead for handling the conversion of bytes to packets. Figure 2 below exposes the difference in communication gap differences between C and Java. Keep in mind that the results are for C, but we can conclude with confidence the result would be similar with C++.

Figure 2 : C vs. Java Communication speed test

## Synchronization Methods

For all parallel simulation environments the parallel processes must be coordinated in order to ensure that events are processed in their correct causal order. These techniques are called synchronization strategies. Synchronization strategies can be broadly classified into two distinct categories, namely: synchronous and asynchronous strategies.

### Synchronous Method

Synchronous strategies were the first method that were developed and were inherently developed for single node. The main idea is that all processes must synchronize at each time step (Bailey and Snyder). However, such approaches are not effective for realizing horizontal scalability. When having to synchronize at each time step when working parallel simulation, the overhead of the synchronization time increase as the number of nodes increase. Realizing this being a serious issue, asynchronous methods were introduced. Another reason for introducing asynchronous methods was to eliminate the need for global queue storage of events (Bailey and Snyder).

### Asynchronous Method

Asynchronous methods can further be classified into two types, conservative and optimistic.

The most known and accepted conservative method is the CMB algorithm [ (Chandy and Misra), (Bryant)]. It was developed by Bryant (Bryant), Chandy (Chandy and Misra), and Misra (Chandy and Misra) independently. In this method each process keeps its own simulation clock. The clocks advance separately. Each process can advance its clock only if it is guaranteed that no event will arrive with a timestamp less than its clock value (Bailey and Snyder). If a parallel process needs to process an event with a timestamp greater than the global clock, then that process will perform a block operation. This operation not only accumulates the overhead of waiting to unblock as the number of processors increase, but it can also lead to a deadlock situation during simulation (Bailey and Snyder).

In optimistic methods, processes have their own clocks and each process’s clock is advanced whether or not they are guaranteed to be correct. If a future event arrives with a timestamp less than the current clock, some recovery mechanism is used to restore the simulation to a consistent state (Bailey and Snyder). Time Warp is a famous optimistic method that was invented by Jefferson (Jefferson). The overhead of the waiting time in the conservative methods is traded for the extra work done due to processing erroneous events and the rollbacks in time warp (Bailey and Snyder). Fortunately, Time Warp is not susceptible to deadlocks. This turns out to be a very good incentive for choosing Time Warp over a conservative method like CMB. Although it is known that conservative and optimistic methods sometime outperform one another (Bailey and Snyder). For large parallel environments, deadlocks are situations that can quickly get out of hand. Lastly, Time Warp has been heavily studied and every aspect has been dissected and ways to improve Time Warp is readily available [ (Jefferson), (LIN and LAZOWSKA), (Steinrnan), (Das and Fujimoto), and (Chen and Szymanski)]. For MUSE, we have decided to use Time Warp. We’ll further look at the Time Warp protocol in detail next.

#### 2.3.2.1 Time Warp

Time Warp is optimistic; hence events are processed as they are available. In Time Warp, a simulation is organized as a collection of communicating Logical Processes. Communication between logical processes is performed by exchanging virtual time stamped messages or events. Figure 3 below presents a conceptual view of a Time Warp Logical Processes (LP). As shown in the figure, each LP has an input queue, output queue, and a state queue. A LP advances its Local Virtual Time (LVT) by processing events from its input queue, updating its state, and generating new events. The input queue stores the messages that the LP should process. When a message is processed, the LP’s state gets modified. The state queue is used to collect the state of the LP at each time step. The output queue is used to store outgoing message from the LP. The three queues are used to recover from causal violations that are detected when a LP receives a straggler event. Straggler events have timestamps that are lower than the LVT of a given LP. Events in the queues are never fully committed, until it is safe, Time Warp uses GVT calculations for fossil collection. GVT and fossil collection will be described after we clarify how a LP recovers from a casual violation. If the case arises where a straggler event arrives, then a casual violation occurs and a rollback mechanism is used to restore to a consistent state. To perform a rollback the following three steps must take place [ (Jefferson), (LIN and LAZOWSKA)].

* + 1. Using the state queue, restore the state of the LP to a state earlier than the time stamp of the straggler event. Then set the LP’s LVT to that of the restored state.
    2. For every message that the LP has dispatched to other LPs are cancelled by sending an anti-message, which are typically stored in the output queue. These anti-messages undo all events that have been sent from the LP that is rolling back.
    3. Finally, the straggler message is reprocessed in the correct timestamp order.

The aforementioned three steps will insure that the LP is synchronized with other LPs. The Global Virtual Time (GVT) algorithm is used to garbage collect unneeded information from the three queues.



Figure 3 : A logical process in a time warp simulation (Radhakrishnan)

GVT is considered a safe point, because it is the time of the LP with the smallest LVT. When GVT is calculated, there is a guarantee that no event with a smaller time stamp will ever arrive at any of the LPs. One issue with Time Warp is the memory that is required (Jefferson). Until we calculate GVT, we have to store all of the incoming, outgoing events and all the states. The act of removing old events and states is known as fossil collection. If you have an algorithm that will calculate GVT, then you can iterate through all the LP queues and remove all events and states with a time stamp smaller than the GVT, we also commit all I/O operations with a smaller time stamp. Fossil collection keeps a control on the memory requirement. How often you fossil collect will be based on how fast you calculate GVT. Typically to reduce communication overhead from GVT calculation, another technique employed to reduce memory requirement is to throttle the optimism [ (Jefferson), (LIN and LAZOWSKA)]. This is achieved by having creating a restriction on the LP. A simple method is to wait until the difference between the event being processed and the GVT is within a given range. The algorithm we chose for GVT calculation is the Mattern’s GVT algorithm (Mattern).

#### 2.3.2.2 Mattern GVT Algorithm

The Mattern’s GVT algorithm is a simply yet effect way to approximate GVT. There are two main concepts to understand before we can realize our end goal, GVT calculation. First is the notion of a consistent cut.



Figure 4 : A time diagram with a cut (Mattern)

When GVT calculation starts, it begins from the process called the initiator and a control message is passed around to the remaining processes in a round robin fashion, we will call this a control round. When the control message gets back to the initiator we have a “cut”. A cut is consistent if no event from the future (to the right of the cut) lands in the past (to the left of the cut). Figure 4 above shows a consistent cut. The second main concept is the color of the process. The process starts out as a white process, when a control message reaches the process, the color changes to red. Therefore, when the initiator gets the control message back the control rounds ends and all processes should be colored red. Also any event that the process sends out inherits the color of the process, so if the process is white (red) then the event leaving the process is white (red) (Mattern). The third concept is the use of a vector for each process. The vector contains the number of white events that the process receives from another process. Hence, for process P1, it would contain vector V1. Each index in the vector is a reference to how many white events were received, for example V1[2] represents how many white events process P1 received from process P2. For more information about the vector counter, please refer to (Mattern). With these two ideas in mind we can go forward with describing the algorithm.

Mattern’s algorithm uses two control rounds to approximate the GVT. The first round is used to figure out which of the processes has the white event with the smallest timestamp. When the first control round has ended and all the vector V for all the process report a zero count then the smallest timestamp recorded in the control message is the new GVT. A second round is necessary if there is a process that reports a white event count greater than zero. For the second round, the control message will not move to the next process unless that process has received all the white events from the other processes. Once the second round is over we are assured that all white events have been received by the appropriate process and the initiator can finally broadcast the new GVT. For more information regarding the algorithm, please refer to Mattern’s paper (Mattern).

## Non-parallel Agent based Simulation frameworks

Railsback *et. al* presents a detailed survey of several agent-based simulation frameworks that are similar to MUSE. The varying platforms were compared in three areas. Programming experience, execution speed, and general simulation issues (Railsback and Lytinen). A bug’s life simulation was developed as a measuring tool. Programming experience exposes some of the features and characteristics of each platform. The execution speed testing was not a complete and controlled test, but it was enough to get a picture (Railsback and Lytinen). Lastly, general simulation issues were discussed for each platform and how they handle areas like model structures and scheduling.

The frameworks under review were NetLogo, SWARM Objective-C, SWARM Java, Repast, and MASON. Each framework had advantages and disadvantages. NetLogo’s strong points include its detailed documentation and ease of use. However, it uses proprietary code, and users have to learn a custom language for modeling (Railsback and Lytinen). The original SWARM uses the Objective-C language. This is the most mature and stable framework, which makes it well organized (Railsback and Lytinen). While Objective-C is more natural to model with (Railsback and Lytinen), it has weak error-handling. Another downside is the availability of tools for developing with Objective-C. Java SWARM is simply a wrapper that allows Java developers to call Objective-C SWARM libraries. While Java has strong error-handling capabilities, the framework does not effectively take advantage of the two languages (Railsback and Lytinen). Moreover, both versions of SWARM proved to be the slowest for very complex models (Railsback and Lytinen).

Repast was meant to mimic SWARM using Java, but the design and organization of the framework has several drawbacks (Railsback and Lytinen). Furthermore, the learning curve for using the API is very steep, because it has numerous features, often making it overwhelming for most casual developers (Railsback and Lytinen). MASON is a light weight framework that aims to achieve high execution speeds (Railsback and Lytinen). It is also the most recent of all the frameworks and in terms of execution speed; it was indeed the fastest amongst those surveyed by Railsback *et. al*. One of MASON’s main issues was adding multiple agent actions, for example in the bug’s life simulation; the bugs had a move and grow action. Due to the way the scheduler was designed in MASON it was not trivial to add multiple actions (Railsback and Lytinen). MASON used the template method design pattern. Meaning if you want an agent to act you had to implement a method called “step” and perform the action in that method. “An advantage of this design is the time MASON saves in the scheduler, because it always knows to execute a method named ‘step’.” (Railsback and Lytinen) The real disadvantage to this design pattern MASON used is when you want to have all the bugs move, and then in the next time step all to grow. The agent had a reference to the scheduler and if he wanted to be scheduled for the next time step, the agent adds itself to the schedule for the next time step. Since you only have one method available this becomes a less trivial task to complete. MUSE will use the same template design pattern, but in our case we will get all the benefits and none of the drawbacks. At its heart MUSE is an agent base framework, the method that the agent must provide is “executeTask”. However since it is designed with parallelism in mind the only way to communicate with agents is with events. You can see that by providing different event types you can easily perform anything you want in the “executeTask” method. In the bugs life example, we simply would have a move event and a grow event. We simply schedule the needed event (action) at the right time. More on the design of the scheduler will be discussed in section 3.

## Parallel Non-Agent based simulation frameworks

In conjunction with our initial investigations, we also reviewed three parallel simulation frameworks namely WRAPED (Radhakrishnan), GTW (D. Das, R. Fujimoto and K. Panesar), and Parsec (R. Bagrodia). It must be noted that these are general purpose discrete event, parallel simulation frameworks and not necessarily agent-based simulation environments. The strong point of WARPED is the similarities is has to MUSE. This proved to be a valuable resource during the design stage of MUSE. One similarity to MUSE is the use of the Time Warp synchronization method. It also uses MPI as its message passing protocol and C++ as the language. However, several issues posed serious hurdles for effective use of the framework. The most important one is the lack of documentation. Furthermore, the simulator has not been actively maintained and therefore several issues prevented even compiling the core framework using recent compilers. Since WARPED development started in 1998, it clearly went through several upgrades in features, but the changes were not documented clearly. GTW also uses Time Warp, and similar to WARPED, it lacks documentation and has not been actively maintained. Furthermore, GTW was primarily developed for shared memory architectures while today’s supercomputing clusters primarily used distributed memory architectures. However, GTW includes several beneficial design solutions. One of the important design solutions that will be used in MUSE is controlling optimism during simulation. Controlling optimism is necessary because, Time Warp has a tendency to be too optimistic, this could lead to cascading rollbacks. GTW avoids cascading rollbacks by using time windows that throttle optimism (D. Das, R. Fujimoto and K. Panesar). Another attractive feature is the local message sends, meaning if a message is meant for the local LP it is simply enqueued directly to its input queue.

Parsec is most the complicated parallel framework from the group. Strong points of Parsec include its visual environment. Developers modeled via a GUI (R. Bagrodia). Parsec implements many conservative synchronization methods and many communication libraries (R. Bagrodia). However, conservative synchronization requires the modeler to be cognizant about look ahead in simulation-time during model development. Look ahead is necessary to avoid deadlocks that potentially occur during simulation. However, look ahead can be complex to extract when developing models and small look ahead negatively impacts simulation performance.

On the other hand, Time Warp does not rely on look ahead making it easier for the model developer. However, like previously mentioned, Time Warp uses state saving and rollback to recover from causal violations; thereby requiring additional memory and CPU time for rollback processing. In other words, in conservatively synchronization simulations time is spent waiting for other parallel processes to coordinate while in Time Warp time is spent recovering from rollbacks. However, several Time Warp optimizations are available to minimize rollbacks and these optimizations can be implemented without impacting the API or placing overhead on the modeler. Consequently, we chose to use Time Warp as the synchronization protocol for MUSE.

## Choosing data structure for scheduling

Having a scalable and efficient simulation environment is very dependent on the data structure we use to maintain the events and the agents for scheduling. MUSE has a two tier scheduling system. The very top tier is the scheduler and it maintains the agents and knows which agent to process at any given time. The second tier is in the agent. All incoming events to a given agent must be stored and correctly scheduled in increasing fashion according to the time of the delivery. The heap data structure seemed a great fit for both tiers. The heap data structures under consideration are the Fibonacci heap (Fredman and Tarjan) and the Binary Heap. Binary heaps are heaps that are implemented with binary trees (Wikipedia). Fibonacci heaps have very impressive runtime results, however these results are amortized. The following table shows the runtimes of both binary and Fibonacci heaps.

|  |  |  |
| --- | --- | --- |
| Standard Operations | Fibonacci Heap | Binary Heap |
| Insert | O(1) | O(log\*n) |
| Get Min | O(1) | O(1) |
| Delete Min | O(log\*n) <amortized> | O(log\*n) |
| Decrease Key | O(1) <amortized> | O(log\*n) |
| Delete | O(log\*n) <amortized> | O(n) |
| Merge | O(1) | O(m log(n+m)) |

Fibonacci heap showed impressive runtimes, but we wanted to know just how much we have to amortize before we realize the gains. Binary heap on the other hand has good runtimes and no amortized costs. The two tiers make more use of different operations. Hence, there is a good chance that we would end up using a combination of the two heaps in MUSE. The first task we have done is identifying which operations were frequent in each tier. The first tier, once we add the agents we should never remove until the end of simulation. Therefore the only operation we want to compare is the *decrease key* operation. Decreasing the key in short is just an operation to reorder an element in the heap. We can draw an early conclusion here and say that Fibonacci heap should be used, but it is better to let the numbers speak. In the second tier, we frequently made use of the *insert, get min, and delete min*, whenever there was a rollback we also used the *delete* operation.

For binary heap implementation we will be using the *priority\_queue* from the C++ STL containers. Fibonacci heap we have found a nice C++ implementation. To get the source for the fibonacci heap implementation follow this reference (Kühl).

### 1.1.1 Fibonacci vs. Binary testing procedure

We have to find a good heap for both tiers, and we already discussed the heavily used operations for both tiers. With the first tier we want to test the key decreasing. To get a good idea we have a couple of controlled variables. We have fixed the time steps to 400. This allowed us to see a nice difference in performance between the two heaps and the time to run the tests was reasonable. The basic idea is to keep increasing the number of agents, starting from 100 and ending at 100,000 agents. At each time step we will iterate over the number of agents and randomly (P = .5) increase or decrease the value of the agent’s key, and call the *decrease* operation on the key. We will keep track of the time it takes to execute and take the average of five runs for each increase in the number of agents. Fibonacci heap implementation has a *change(element, key)* method which we can use. However, the priority queue does not implement a way to change the key, so the solution is to pop the top element and then update the value and push it back into the heap. This makes the runtime from *O(log\*n)* to *O(log\*n+log\*n)*. We simply added the runtimes for *delete min* and *insert* to get the updated runtime.

The second tier deals with events. To actually see something meaningful we fix our time steps to 5000 iterations. We will slowly increase the number of events in the heap starting from 100 events per time step all the way to 100,000 events per time steps. There are two cases to test in the second tier. First case is just going to be a test to see how long it takes to insert *X* number of events and then *delete min* until the heap is empty again. The second case is testing how long it takes to delete arbitrary elements from the heap. We will use the Iterators and just keep calling the *delete* operation and see which has the best time. The STL container *priority\_queue* does not support Iterators. In order to get elements in the back, we would have to remove all the elements and store the valid ones into a temporary storage. Once we remove the invalid elements, we would then push all the elements from the temporary storage back into the priority queue. Like the first tier we will run each five times and get the average time. The big deciding factor for the second tier will be the first case, as this is the most frequent operations. However, since *priority\_queue* does not support Iterators, we must add the second case into the comparison test.

### 1.1.2 Fibonacci vs. Binary data collection, results, and discussion

The table below is the collection data when we compared the two heaps for tier one. Keep in mind that that the execution times are the average of five runs and represent execution time in seconds.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Agents | Time steps | Fibonacci execution time | Binary execution time | Speedup |
| 100 | 400 | 6 | 16 | 2.66 |
| 1000 | 400 | 78 | 222 | 2.84 |
| 10000 | 400 | 949 | 2721 | 2.86 |
| 100000 | 400 | 14090 | 34179 | 2.42 |

The graph above shows clearly the trends we expected. The results above were expected because for binary the best and worst case is *O(log\*n+log\*n).* However, fibonacci heap has a best amortized run time of *O(1),* but the worst case is *O(log\*n).* Derived proof of worst case times for fibonacci heap can be found in the reference (Fredman and Tarjan). From these results the choice for tier one is fibonacci heap.

The table below is the collection data when we compared the two heaps for tier two. Here are the execution times of case one and case two combined as discussed earlier.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Events | Time steps | Fibonacci execution time | Binary execution time | Speedup |
| 100 | 5000 | 2 | 1 | 2 |
| 1000 | 5000 | 23 | 18 | 1.27 |
| 10000 | 5000 | 300 | 203 | 1.47 |
| 100000 | 5000 | 3236 | 1794 | 1.80 |

The results for tier one were expected, however the results from the test for tier two revealed surprising information. The amortized cost of the *delete min* operation proved to be too great. Since *priority\_queue* did not support Iterators, we had to be fair and add the time to actually remove arbitrary elements from the heap. We purposely took the naïve approach and just popped all elements into a temporary storage and pushed in the valid elements back into the heap. Fibonacci still proved to be slower than the binary heap. From the results we can clearly conclude that the amortized run times claimed by fibonacci heaps would require very large data in the heap and for our purposes was not needed. Hence, for tier two, we will use the binary heap, but can we do better?

For tier two, you can see from the speedup column in the table that while there was speedup, they were not great. This was due greatly to the performance hit we take from *priority\_queue* and its lack of iterators. Earlier we showed the runtime to delete arbitrary elements from the binary heap took worst case *O(log\*n+log\*n).* We believe the speedup would be even greater if we had the worst case to the original runtime of *O(n).* This need to improve our performance was the motivation behind the development of the *BinaryHeapWrapper*. This implementation used a vector and represented it as a binary heap. Most importantly it allow us to remove arbitrary elements at *O(n).* *BinaryHeapWrapper* was specialized for MUSE and the disadvantage being reusability, the binary heap implementation could only be used within the MUSE framework. We ran the tests for tier two and compared *BinaryHeapWrapper* against *priority\_queue* and fibonacci heap*.* The following table shows the results from the experiment. Since we have two binary heaps, we decided to change the names in the table to the actual class names.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Events | Time steps | Fibonacci execution time | Binary execution time | BinaryHeapWrapper execution time |
| 100 | 5000 | 2 | 1 | 1 |
| 1000 | 5000 | 23 | 18 | 13 |
| 10000 | 5000 | 300 | 203 | 125 |
| 100000 | 5000 | 3236 | 1794 | 1041 |

We did not show the speedup in the table above, but we can clearly see a dramatic increase in performance from the custom built binary heap.

From these experiments and results we were able to show that although fibonacci heap had great amortized times, there are cases where the benefits do not outweigh the cost. However, there was a case where fibonacci turnout to be very beneficial and hence us using it for tier one. Using fibonacci heap for tier one, means that fibonacci heap is really great when you have a case where you have frequent key changes and minimum popping from the heap. This was the exact scenario for tier one. Lastly, we showed that priority\_queue actually turned out to be faster for tier two over fibonacci heap, but its lack of iterator support motivated the need for a custom built binary heap. The BinaryHeapWrapper turned out to have the best time and speedup out of all three heaps and thus was chosen as the heap for tier two.