# Benchmarking and Empirical testing

In this section we will go into empirically testing and benchmarking MUSE. In the next five subsections of this chapter, we will first describe the case study we will use for our experiments. Second, we will show implementation snippets of PHOLD for each framework in out testing. These frameworks, beside MUSE, will be MASON and WARPED. In the third subsection, a description about the metrics used is given. Fourth subsection is a talk on the data gathered from empirically testing MUSE. Finally, we discuss the benchmark results.

## Synthetic Simulation PHOLD

Like any framework, we would like to observe and test performance. Being a distributed framework, we are also interested in the scalability and efficiency of MUSE. For our experiments, we will implement and test with PHOLD (Fujimoto). PHOLD is supposed to synthetically test the typical workload of each agent in a simulation. It also allows you to scale and fine tune many variables to observe the impact in the simulation framework. For our tests, PHOLD will be an *X x Y* agent grid, where each agent in the grid will have four neighbors. When the simulation starts, each agent initializes by sending *N* events to itself. The *N* events have a random receive time for the future, with a max receive time defined by the variable *Delay*. During PHOLD simulation, when an agent receives an event, the following takes place.

1. Randomly select which neighbor to send the next event to.
2. Randomly choose a receive time from 1 to *Delay*.
3. Create and send the event.

The three steps above repeat until we send *N* events. This process happens for each agent until the simulation is over. The following figure 1 visually shows the PHOLD process. Since MUSE is distributed the different color agents represent the node they reside in. Hence, the figure shows a 3 x 3 grid of agents each color represents a compute node or a different process.

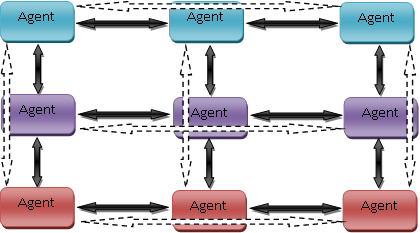


Figure 1 : 3 x 3 PHOLD simulations on three compute nodes

## There are different variables we can adjust. Depending on the variable we adjust we can observe different behaviors and see how well MUSE performs. The following are the different variables we can adjust:

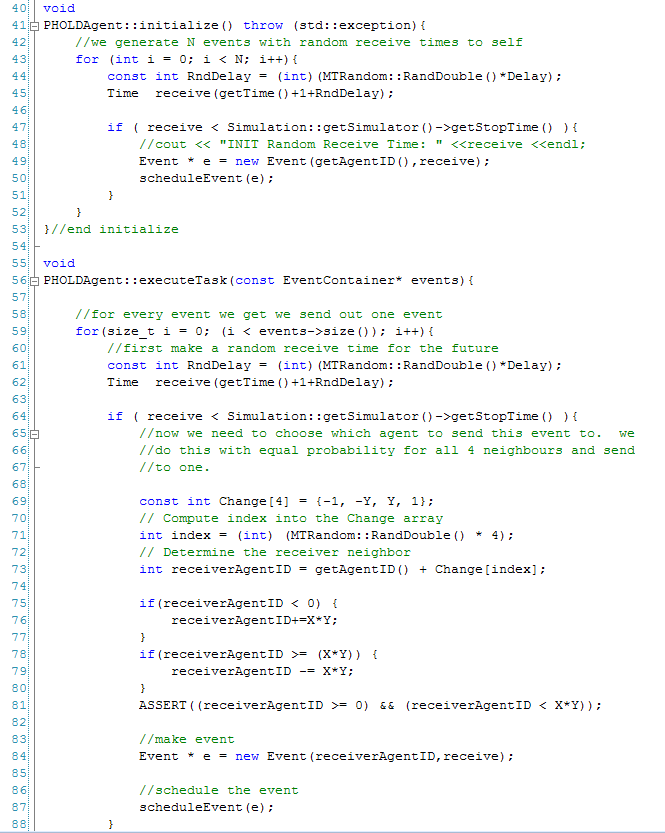
* *X*, this is the number of columns to have in the PHOLD grid
* *Y*, this is the number of rows to have in the PHOLD grid
* *N*, the number of events each agent sends every time.
* *Delay*, the maximum receive time that an agent can schedule an event for.
* *Nodes*, the number of compute nodes to use for the PHOLD simulation.

## Our experiment will be held on a cluster which houses 128 compute nodes. The spec of each compute node is shown below.

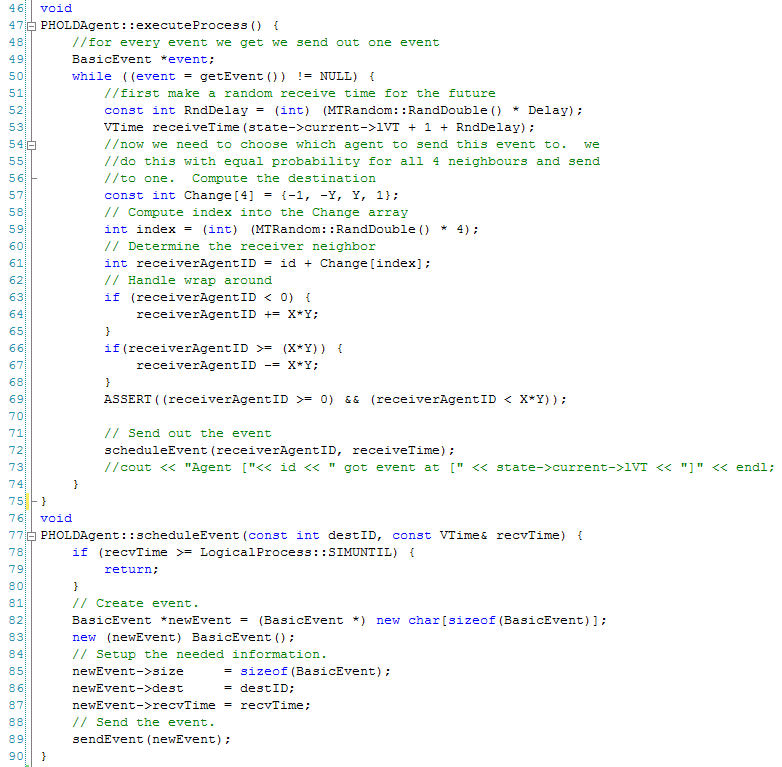
|  |  |
| --- | --- |
| Component | Details |
| CPU Model | Intel Xeon (x2) |
| CPU/Core Speed | 3.0 GHz (x2) |
| Main Memory (RAM) size | 4 GB |
| Operating system used | Linux 2.6.9-22.ELsmp |
| Interconnect type & speed (if applicable) | Infiniband @ 20Gbps |

## 1.2 Implementation and Code Snippets

We have discussed in detail how PHOLD works. We have implemented PHOLD in three different frameworks. MUSE, WARPED, and MASON. In this section we will present the implementation code for each framework. The first implementation will be for MUSE.



The next implementation is for the WARPED framework. Note that the implementation is almost identical to MUSE. However, the terminology is different. For example, when dealing with TimeWarp we could say that a Logical Process is similar to an agent. However, WARPED associated a Logical Process with a process or a compute node. This can get very confusing because the terminology that WARPED uses is not common. The following is an implementation of PHOLD in WARPED.



MASON uses Java as a language and is not a distributed framework. MASON works differently. To create a model you must create a class that extends the *SimState* class and to create an agent you implement from *Steppable* interface. Modeling PHOLD with MASON proved to rather time consuming and not straight forward.

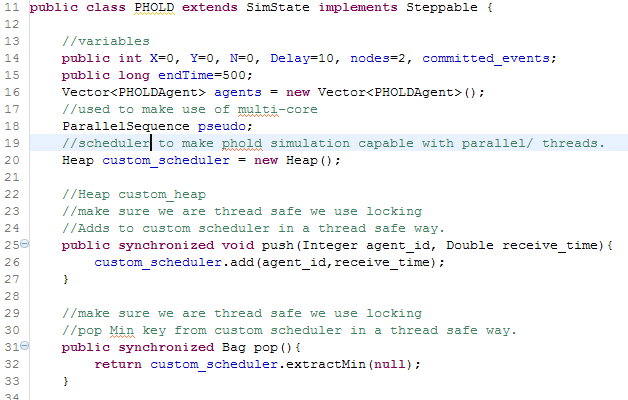


Figure 2 : MASON model part 1

We named the MASON model *PHOLD*; another important detail to note from line 11 in figure 2 is that the model also implements *Steppable*. To understand the reason behind making the model an agent, we need to understand what *ParallelSequence* is and how it works. *ParallelSequence* is MASON’s facility to enable multi-threading. The way to use *ParallelSequence* is to construct the object by passing in an array of agents. It is important that these agents can work independently, otherwise MASON throws thread exceptions. Once MASON’s main scheduler, located in the model is executing the agents in the *ParallelSequence* object, it locks the scheduler so there is no way to schedule from within the agents in the *ParallelSequence* object. The only way to get around this limitation is to create a custom scheduler on top of MASON’s main scheduler. We decided to use a binary heap, shown in line 20 of figure 2. Methods *push (line 25 figure 2)* and *pop (line 31 figure 2)* are both synchronized entry points because we are dealing with multiple threads and MASON’s *ParallelSequence* is not thread safe. We declared the *ParallelSequence* object in line 18 figure 2 and called it *pseudo,* as in pseudo parallel.

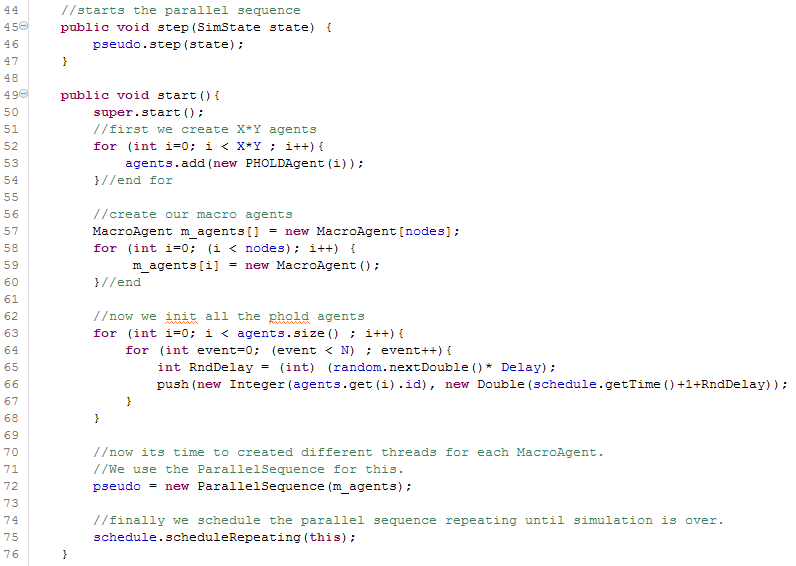


Figure 3 : MASON model part 2

Since agents in the *ParallelSequence* object have to be independent of each other, we created agents called *MacroAgent*. Depending on the number of threads we use, we create just as many *MacroAgent* objects. We do this in figure 3 from lines 57-60. The next step in the *start* method is to initialize all the agents by scheduling each agent *N* different future random times. This is done in figure 3 in lines 63-68. Notice that in line 66 figure 3, we are using the custom scheduler we created. In line 77 figure 3 we create the *ParallelSequence* object with the MacroAgent array we create in line 59 figure 3. Lastly, we repeatedly schedule the model itself, this is possible because the model is also an agent. Line 75 figure 3 is the only time we actually use MASON’s main scheduler. Also notice that when we step on the model, all we do is start the different threads and step on each *MacroAgent* concurrently.

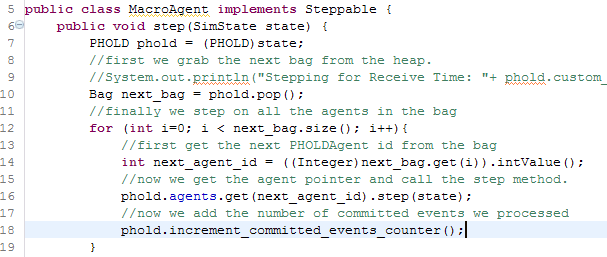


Figure 4 : MacroAgent step method

When the *MacroAgent* is “stepped” on, it first gets a set of *PHOLDAgent* IDs (line 10 figure 4). Then for each agent we do the following three steps.

1. Convert the agent id into an integer value.
2. Since we have direct access to the model itself, we get a pointer to the agent with the corresponding id and execute its *step* method.
3. Finally, we increment the committed events counter via the thread safe method (line 18 figure 4).

The *PHOLDAgent* did not implement the Steppable interface, so technically was not an agent in MASON. However, since we implemented a custom scheduler it did not matter and can be regarded as an agent. The following figure is the implementation.

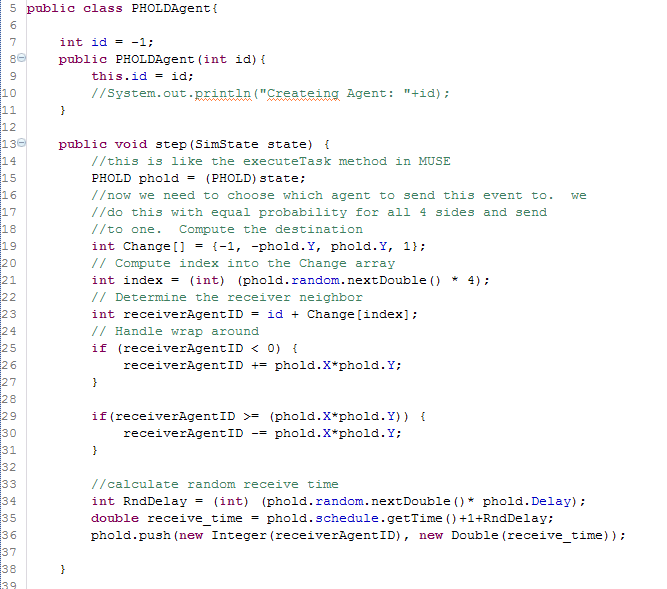


Figure 5 : PHOLDAgent implementation

## 1.3 Metrics used for MUSE analysis

The execution time of parallel algorithms depends on, the number of processing elements and the amount of communication between the processing elements. A metric focuses on a single aspect of a given algorithm. A single metric is typically insufficient for complete analysis and comparison of various algorithms. Several metrics are used for comparing and analyzing computational complexity of parallel programs.

Parallel programs typically do not scale linearly. This is mainly due to various overheads in parallel programs. Various possible factors that lead to overheads in parallel programs are:

* Inter process interactions: Processing elements generally interact and communicate data between one another. This form of interaction involves some amount of time being spent when the data communicated is waiting in the buffer to be sent or received (Grama, Gupta and Karypis).
* Idling of processing elements: Processing elements may go to idle state at certain instances due to synchronization. This is mainly due to the fact that it is difficult to predict the size of subtasks assigned to various processing elements (Grama, Gupta and Karypis).
* Excess computation: The difference in computation performed by a parallel program and the best serial program is the excess computation overhead incurred by the parallel program. Parallel program generally has to perform various tasks that in excess when compared to the serial program. This is mainly due to the fact that certain intermediate results cannot be re-used since they have been produced by various processing elements (Grama, Gupta and Karypis).

It is therefore important to study the performance of a parallel program and generate metrics that can be based on the comparison of a parallel program to its serial counterpart. Commonly used metrics for this purpose are as follows:

* Execution Time
* Speed up
* Efficiency
* Scalability

Execution time can be calculated for both the parallel algorithm and its serial counterpart. The serial runtime *(Ts)* is the wall clock time elapsed between the beginning and end of an execution of a sequential program, while the parallel runtime *(Tp)* is the time elapsed from the moment a parallel computation started to the time when the last processing element finished execution. Generally, parallel runtime has to be less than serial runtime for a reasonable size of input for the parallel program to be efficient.

A general interest while running a parallel program is to determine the performance gain that is achieved on parallelizing an application. Speedup is a metric that can be used for this purpose. Speedup is the ratio of time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with *p* identical processing elements (Grama, Gupta and Karypis). More formally, speedup is defined as the ratio of serial runtime to the parallel runtime.

*S=*

Theoretically, speedup should not exceed the number of processing elements. However, there are cases when speedup exceeds the number of processing elements, in which case it is called super linear speedup (Grama, Gupta and Karypis). This could mainly happen due to:

* Super linearity effects from caches: when program data is large and cannot be cached. In such cases, each individual process executes much faster compared to its serial counterpart (Grama, Gupta and Karypis).
* Super linearity effects due to exploratory decomposition: happens when the problem space is partitioned; once a parallel version identifies a solution, the parallel program terminates (Grama, Gupta and Karypis).

An ideal parallel system containing *p* processing elements can deliver a speedup equal to *p*. An ideal behavior is difficult to achieve since the processing elements are unable to devote 100% of their time to the computation of an algorithm. It is due to this reason a metric such as efficiency is used. Efficiency can be used to determine the percentage of time for which a processing element is useful (Grama, Gupta and Karypis). It is the ratio of speedup to the number of processing elements.

*E=*

In an ideal system, speedup is equal to *p* and efficiency is 1. In practical applications, speedup is generally less than *p* and efficiency is a value between 0 and 1.

An important metric used for evaluating the efficacy of a parallel algorithm is scalability. Scalability is defined as the measure of capacity of parallel program to increase its speedup in proportion to the number of processing elements and the size of the problem (Grama, Gupta and Karypis). A program is said to be scalable if it continues to remain efficient as the number of processing elements increases. Scalability and Efficiency are related metrics. An inefficient program is not a scalable program. In general, scalability focuses on the ability of a parallel program to maintain efficiency when both the problem size and the number of processing elements are simultaneously increased.

A common phenomenon seen in parallel programs is a decrease in efficiency as the number of processing elements is increased. In many cases, the efficiency of a parallel program increases if the problem size is increased while keeping the number processing elements a constant. This is a highly desirable concept that is expected of parallel programs. Using these metrics we will be able to obtain data to help us figure scalability and efficiency.

## 1.4 Empirical evaluation of MUSE

This section presents an empirical evaluation of MUSE. Specifically, experimental results are presented to illustrate the scalability and efficiency of MUSE. The empirical evaluation has been conducted using the PHOLD benchmark by varying its controllable variables. For benchmarking, experiments were initially conducted using a square grid consisting of 512 x 512 () agents participating in PHOLD. In subsequent experiments the number of compute nodes was increased in powers of two starting from one compute-node. The test were run five times to obtain average run times for each configuration. The number of agents was held at a constant value of 512 x 512 for these experiments. The *N* value for each agent was set to three and the *Delay* was set to ten units of simulation time. The simulation was run for 500 time steps. The control variable in the experiments was number of compute *Nodes* used for parallel simulation. The table below shows the execution times for each check point.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Agents | Events | Delay | Nodes | End Time | Execution Time (seconds) |
| 512 x 512 | 3 | 10 | 1 | 500 | 1663 |
| 512 x 512 | 3 | 10 | 2 | 500 | 645 |
| 512 x 512 | 3 | 10 | 4 | 500 | 291 |
| 512 x 512 | 3 | 10 | 8 | 500 | 142 |
| 512 x 512 | 3 | 10 | 16 | 500 | 65 |
| 512 x 512 | 3 | 10 | 32 | 500 | 33 |

Table 1: Execution times with increasing nodes

From the table and figure above we are able to calculate speedup and efficiency. For our serial program we use the execution time obtained from running PHOLD on MUSE with one node. Hence, we consider the serial runtime *Ts* = 1663 seconds. The observed speedup and efficiency of PHOLD simulation is shown in the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Agents | Nodes | Execution Time (seconds) | Speedup | Efficiency |
| 512 x 512 | 2 | 645 | 2.578295 | 1.289147 |
| 512 x 512 | 4 | 291 | 5.714777 | 1.428694 |
| 512 x 512 | 8 | 142 | 11.71127 | 1.463908 |
| 512 x 512 | 16 | 65 | 25.58462 | 1.599038 |
| 512 x 512 | 32 | 33 | 50.39394 | 1.574811 |

Earlier in the section we described the metrics we would use, there was a mentioned that ideal speedup is equal to the number of nodes we use to run the parallel algorithm. However, we also mentioned of special cases were the speedup is greater than the number of nodes used. The PHOLD simulation on MUSE observed super linear speedup. This caused our efficiency to be greater than one. These great results are thanks to the reduction in rollbacks by using the optimization trick WARPED uses. Another is the use of data structure for scheduling. From these results we can conclude that MUSE is very efficient for very large models. Another important detail to notice is that as the number of nodes increased, MUSE efficiency did not drop, but instead increased as well. These results are very desirable as previously mentioned. However, in order to calm MUSE as also being scalable, we must perform one more set of experiments.

The next experiment is to check if the trend for execution time stays consistent. Here we will adjust the number of nodes while trying to maintain the number of agents to compute node roughly consistent. We start from one node and move up to 32 nodes in power of twos. We also increment our agents such that at any given checkpoint each compute node is working with around 8000 to 10,000 agents. The table below shows the results we obtained.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Agents | Events | Delay | Nodes | End Time | Execution Time (seconds) |
| 100 x 100 | 3 | 10 | 1 | 500 | 32 |
| 200 x 100 | 3 | 10 | 2 | 500 | 33 |
| 200 x 200 | 3 | 10 | 4 | 500 | 34 |
| 400 x 200 | 3 | 10 | 8 | 500 | 34 |
| 400 x 350 | 3 | 10 | 16 | 500 | 33 |
| 700 x 400 | 3 | 10 | 32 | 500 | 35 |

Table 2 : Execution time with increasing agents and nodes

We have observed super linear speedup and excellent efficiency with MUSE. From the second experiment we were able to see that as the number of agents increased and the number of nodes increased, execution time remained comparatively constant. This nice trend is that last piece of data we needed to conclude that indeed MUSE is a very scalable framework.

## 1.5 Benchmarking MUSE

We have seen that MUSE is very efficient and scalable. In this section we will compare MUSE against WARPED and MASON. MUSE empirical evaluation showed that as the model grew MUSE was still showing super linear speedup. This implies that the true strength of MUSE is exposed even more as the model gets larger. MASON and WARPED use different concepts, when dealing with the simulation as a whole. MASON for example, can only work with one process. All the agents have direct access to all other agents in the simulation. Also there is no concept of an “Event”. MASON uses conservative synchronization; these features make the framework very fast for small models. Lastly, MASON has multi-threaded capabilities. This means that if agents can execute independently, then we can use different threads and run them concurrently. However, MASON cannot maintain its impressive speed with very large models like MUSE can, this is because as the model grows, the overhead to synchronize becomes very costly. On the other side of the spectrum is WARPED. WARPED has some similarities to MUSE, for example, WARPED uses TimeWarp for synchronization. It is also a distributed framework like MUSE. However, it is not an agent-based framework. We will first benchmark MUSE vs. Warped and then MUSE vs. MASON.

Since WAPRED is distributed and uses TimeWarp we can do a direct comparison. PHOLD will be the simulation for the benchmark. The following will be the variables used in the benchmark.

* X x Y = 256 x 256
* N = 3
* Delay = 10
* Nodes = {1,2,4,8,16,32}
* End Time = 500

We start the PHOLD simulation with one node and start increasing nodes by powers of two. The following is the table with the results from the simulation.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Agents | Events | Delay | Nodes | End Time | MUSE Execution Time (seconds) | WARPED Execution Time (seconds) |
| 256 x 256 | 3 | 10 | 1 | 500 | 273 | 156469 |
| 256 x 256 | 3 | 10 | 2 | 500 | 137 | 35390 |
| 256 x 256 | 3 | 10 | 4 | 500 | 62 | 7006 |
| 256 x 256 | 3 | 10 | 8 | 500 | 30 | 1226 |
| 256 x 256 | 3 | 10 | 16 | 500 | 16 | 184 |
| 256 x 256 | 3 | 10 | 32 | 500 | 9 | 61 |

Table 3 : 256x256 PHOLD on MUSE and WARPED

MUSE ~1 min mark.

WARPED ~1 min mark.

Figure 6 : PHOLD Simulation on MUSE and WARPED

Benchmarking MUSE vs. WARPED reveals many interesting facts. First, remember that they are both distributed frameworks. This means that the more nodes we add to a steady size model the runtime will increamentally execute faster. Using this point of view, we see that WARPED needed 32 nodes to reach the approximate one minute mark. In contrast, MUSE only need four nodes. This fact indicates that MUSE is far more scalable and efficient, because MUSE utilizes more of the compute node and thus needs far less nodes. We also mentioned that MUSE true strenght is with very large models. The 256 x 256 PHOLD we simulated was a model that consisted of 65,536 agents. From WARPED point of view this turned out to be a very large model and in terms of runtime, MUSE execution time with one node was about 572 times faster. This is more evidence which indicates MUSE data structures are indeed much better with large models.

The next set of experiment will be used to benchmark MUSE vs. MASON. Since MASON uses different concepts as discussed earlier, we break the experiments into two steps. First, we will see how well MASON performs when PHOLD runs with {1, 2, 4, and 8} threads. We will then choose the best run time. Using the best runtime, in the second experiment, we will see how many nodes it takes to beat this runtime. We should also be able to prove that for very large models MUSE will again have the advantage. The following PHOLD variables were used for the first experiment in the benchmark of MUSE vs. MASON.

* X x Y = 512 x 512
* N = 3
* Delay = 10
* Threads = {1,2,4,8}
* End Time = 500

We used 512 x 512, which is a bigger model than the one used with WARPED, because MASON is a very fast framework and we needed a large enough model to see notable difference. The following is a table that shows the results.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Agents | Steps (N) | Delay | Threads | End Time | Execution Time (seconds) |
| 512 x 512 | 3 | 10 | 1 | 500 | 112 |
| 512 x 512 | 3 | 10 | 2 | 500 | 135 |
| 512 x 512 | 3 | 10 | 4 | 500 | 260 |
| 512 x 512 | 3 | 10 | 8 | 500 | 439 |

Table 4 : PHOLD on MASON

When we plotted the trend for PHOLD on MASON, we see that in fact as the numbers of threads increase, the runtime becomes slower. This can be explained by the overhead attained from synchronization. This fact is even more evident for models in which agents frequently communicate with each other. Hence, to minimize synchronization overhead we decided to use the runtime we obtained from the PHOLD simulation that uses one thread. In the second set of experiments, we run PHOLD on MUSE with the following variables.

* X x Y = 512 x 512
* N = 3
* Delay = 10
* Nodes = {8,9,10,16,32}
* End Time = 500

Since we have already simulated PHOLD with 512 x 512 earlier for MUSE empirical evaluation, we know that we need more than eight nodes to beat the best runtime from MASON (112 seconds). We run PHOLD simulation and we keep increasing the number of nodes until we beat MASON’s best time. The remaining runs are there just to show the trend. The following are the results in table format.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Agents | Events | Delay | Nodes | End Time | Execution Time (seconds) |
| 512 x 512 | 3 | 10 | 8 | 500 | 142 |
| 512 x 512 | 3 | 10 | 9 | 500 | 127 |
| 512 x 512 | 3 | 10 | 10 | 500 | 111 |
| 512 x 512 | 3 | 10 | 16 | 500 | 65 |
| 512 x 512 | 3 | 10 | 32 | 500 | 33 |

Threshold line, to show MASON best time of 112 seconds on 1 node.

After 10 nodes MUSE out performs MASON

This experiment reveals some important details concerning MUSE. First, we start to see some of MUSE limitations. One disadvantage that comes to light is when we benchmarked with one node. MASON can handle fairly large model with ease. MUSE’s runtime for a 512 x 512 PHOLD simulation was approximately 27 minutes. In contrast, MASON finished in just under two minutes. This is due to many reasons. First, MASON only works with one process; hence all agents can directly communicate. Second, there is no notation of “Events”, so the overhead of maintaining “Events” and making sure an agent can communicate in a distributed fashion is no longer there. However, there are important points to take from this benchmark. For any size model, MUSE will eventually outperform MASON for the following reasons.

1. The overhead incurred for maintaining “Events” and a medium for agent communication is fixed, even as the model grows.
2. MUSE is very efficient and scalable, so as the model gets larger MUSE gets even faster than MASON.