# 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.

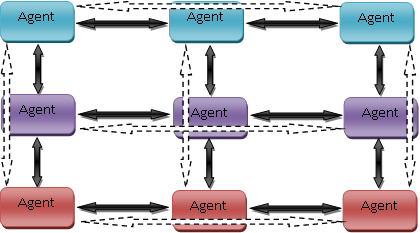


Figure 1 : 3 x 3 PHOLD simulations on three compute nodes

## 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:

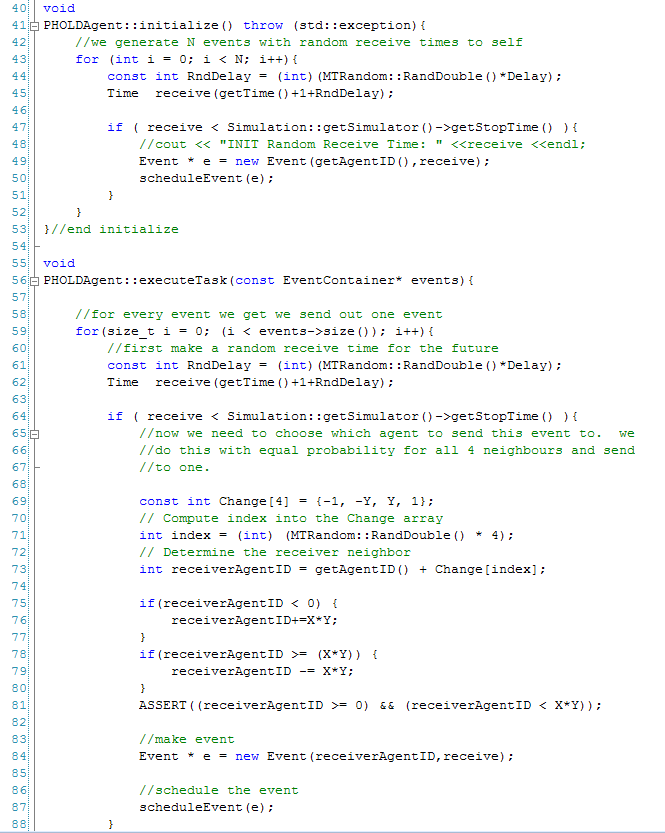
* *X*, this is the number of columns to have in the PHOLD grid
* *Y*, this is the number of rows to have in the PHOLD grid
* *N*, the number of events each agent sends every time.
* *Delay*, the maximum receive time that an agent can schedule an event for.
* *Nodes*, the number of compute nodes to use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

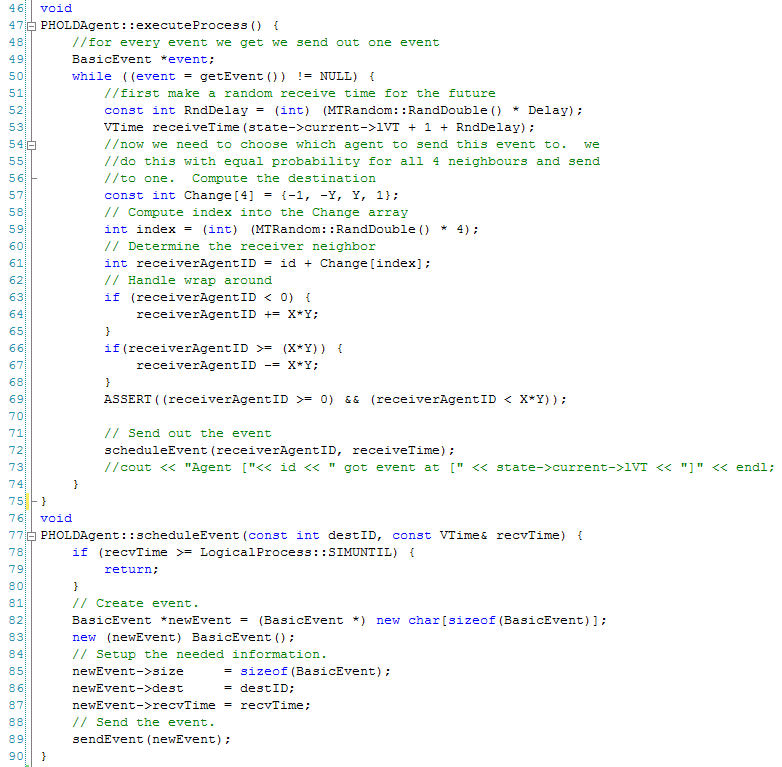
|  |  |
| --- | --- |
| Component | Details |
| CPU Model | Intel Xeon (x2) |
| CPU/Core Speed | 3.0 GHz (x2) |
| Main Memory (RAM) size | 4 GB |
| Operating system used | Linux 2.6.9-22.ELsmp |
| Interconnect type & speed (if applicable) | Infiniband @ 20Gbps |

## 1.2 Implementation and Code Snippets

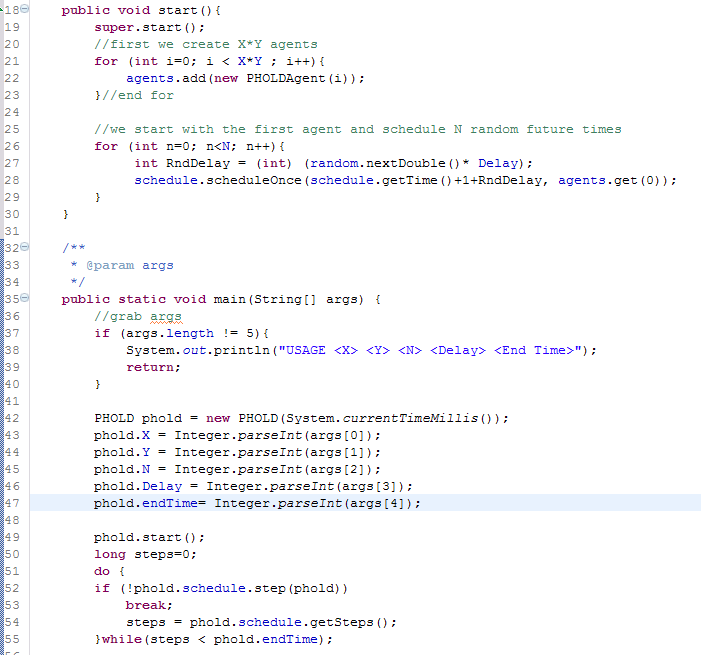
We have discussed in detail how PHOLD works. We have implemented PHOLD in three different frameworks. MUSE, WARPED, and MASON. In this section we will present the implementation code for each framework. The first implementation will be for MUSE.



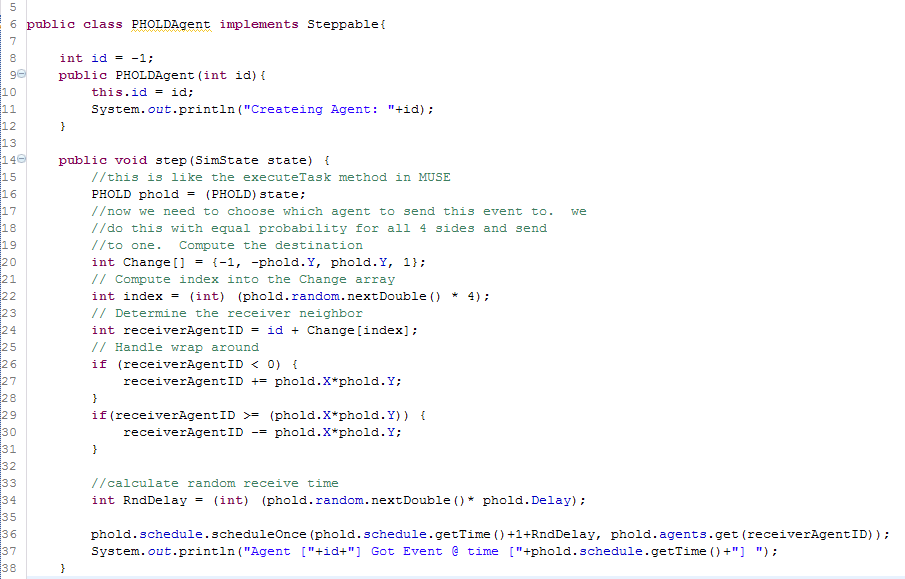
The next implementation is for the WARPED framework. Note that the implementation is almost identical to MUSE. However, the terminology is different. For example, when dealing with TimeWarp we could say that a Logical Process is similar to an agent. However, WARPED associated a Logical Process with a 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. The following is the PHOLD model and the *Start* method is what starts the entire simulation.



An agent in MASON is any class that implements the *Steppable* interface. We created an agent called *PHOLDAgent* and the following is the implementation in MASON.



When an agent process an event in MASON, the method the kernel calls is the *step* method. The implementation of PHOLD is the same in MASON, the only difference being the terminology used.

## 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

In this section we want to know just how scalable and efficient MUSE really is. In order to get an answer we run PHOLD under different variables and observe. For this experiment we will start with a 512 x 512 grid of agents for PHOLD. Starting from one node we will incrementally move up to 32 in powers of two. We will run each test five times and get the average run time. We will keep the number of agents constant at 512 x 512. Each agent will have an *N* value of three and a *Delay* of ten. The simulation will run for 500 time steps. The only variable changing will be the number of *Nodes*. 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